大规模线性规划问题的优化解法

项子越

14336214

2017 年 6 月

简介

本文探究在计算机上进行大规模线性规划问题求解的方案. 本方案使用更适用于计算机使用的改进单纯形法, 并且使用人工变量法避免求初始基矩阵的逆, 从而提高了所能计算的矩阵的维数. 同时鉴于计算机内存空间有限的特性, 本方案选择将矩阵放置在硬盘中运算, 同时使用稀疏矩阵技术减少内存使用.

其次本文还提供了本方案的 C++ 实现, 经验证后发现本方案确实可行.

目录

1 改进单纯形法

[1]使用计算机计算大规模线性规划问题的时, 传统的单纯形法存在一定的不足之处, 而改进单纯形法更适合计算机使用. 以下简单描述改进单纯形法.

符号列表	
符号	说明
$\boldsymbol{A}$	线性规划问题标准形式的系数矩阵 满足 $\boldsymbol{A} \in \mathbb{R}^{m\times n}(n>m)$, 且 $r(\boldsymbol{A})=m$.
$\boldsymbol{p_k}$	$\boldsymbol{A}$ 的第 k 列
$\boldsymbol{B}$	可行基矩阵
$\boldsymbol{N}$	非基变量矩阵
$\boldsymbol{x_B}$	基变量的解向量
$\boldsymbol{b}$	右端项向量
z	当前目标函数值
$\boldsymbol{c_B}$	基变量的价值系数向量
$\boldsymbol{c_N}$	非基变量的价值系数向量
$\boldsymbol{J}$	非基变量的下标集

令 $\boldsymbol{y_k} = \boldsymbol{B^{-1}p_k}$, 以下几个结论与传统单纯形法相同, 因而是显然的:

- $\boldsymbol{x_B} = \boldsymbol{B^{-1}b}$
- $z = \boldsymbol{c_B B^{-1} b}$
- 最优判别准则:

  $\boldsymbol{\sigma = c_N - c_B B^{-1} N \le 0}$

- 进基变量选择准则:

  $\sigma_k = \max\limits_{j\in \boldsymbol{J}}(c_j - \boldsymbol{c_B B^{-1} p_j}),$ $\quad x_k$进基

[1]本节参考 [1, p. 66]

- 离基变量选择准则:

$$\theta = \min\left\{ \frac{(\boldsymbol{x_B})_l}{\boldsymbol{y_{lk}}} \middle| \boldsymbol{y_{lk}} > 0 \right\}, \qquad (\boldsymbol{x_B})_l \text{离基}$$

考虑当前可行基

$$\boldsymbol{B} = (\boldsymbol{p_{B_1}}, \boldsymbol{p_{B_2}}, \boldsymbol{p_{B_3}}, \ldots, \boldsymbol{p_{B_l}}, \ldots, \boldsymbol{p_{B_m}}), \tag{1}$$

若基本可行解不是最优解, 且 x_k 为进基变量, $\boldsymbol{x_{B_l}}$ 为离基变量, 那么下一可行基为

$$\hat{\boldsymbol{B}} = (\boldsymbol{p_{B_1}}, \boldsymbol{p_{B_2}}, \ldots, \boldsymbol{p_k}, \ldots, \boldsymbol{p_{B_m}}). \tag{2}$$

用 $\boldsymbol{e_i}$ 表示第 i 个分量为 1, 其余分量为 0 的单位向量, 因此可得

$$\begin{aligned} \hat{\boldsymbol{B}} &= \boldsymbol{B}(\boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{y_k}, \ldots, \boldsymbol{e_m}) \\ &= \boldsymbol{BT} \end{aligned} \tag{3}$$

$$\text{其中} \qquad \boldsymbol{T} = (\boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{y_k}, \ldots, \boldsymbol{e_m}).$$

显然, 此时有

$$\hat{\boldsymbol{B}}^{-1} = \boldsymbol{T}^{-1}\boldsymbol{B}^{-1}, \tag{4}$$

记 $\boldsymbol{y_k} = (y_{1k}, \ldots, y_{lk}, \ldots, y_{mk})^T$, 其中$y_{lk}$为主元素, 则有

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & \cdots & y_{1k} & \cdots & 0 \\ 0 & 1 & \cdots & y_{2k} & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \cdots & y_{lk} & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \cdots & y_{mk} & \cdots & 1 \end{bmatrix},$$

而此时有

$$\boldsymbol{T}^{-1} = \begin{bmatrix} 1 & 0 & \cdots & -y_{1k}/y_{lk} & \cdots & 0 \\ 0 & 1 & \cdots & -y_{2k}/y_{lk} & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \cdots & 1/y_{lk} & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \cdots & -y_{mk}/y_{lk} & \cdots & 1 \end{bmatrix}. \tag{5}$$

不妨令 $\boldsymbol{\xi} = (-y_{1k}/y_{lk}, \ldots, 1/y_{lk}, \ldots, -y_{mk}/y_{lk})^T$, 则可记

$$\boldsymbol{E} = \boldsymbol{T^{-1}} = (\boldsymbol{e_1}, \ldots, \boldsymbol{\xi}, \ldots, \boldsymbol{e_m})^T, \tag{6}$$

代入 (4) 式可得

$$\hat{\boldsymbol{B}} = \boldsymbol{E}\boldsymbol{B^{-1}}. \tag{7}$$

以上结论表明由当前基逆 $\boldsymbol{B^{-1}}$ 到下一基逆 $\hat{\boldsymbol{B}}^{-1}$ 只需计算并存储 $\boldsymbol{E}$ 矩阵, 而 $\boldsymbol{E}$ 矩阵的存储只需要知道 $\boldsymbol{\xi}$ 向量及其位置, 其余列向量均同单位矩阵, 故存储量大大减少. 由于 $\boldsymbol{E}$ 是稀疏矩阵, 改进单纯形法在一定程度上会保持基矩阵的稀疏特性.

2 人工变量与大 M 法

[2]在原问题中加入人工变量后可以避免求初始可行基的逆, 因此可增大可计算问题的规模. 同时, 因为初始可行基将会是稀疏矩阵, 因此十分适合配合改进单纯形法使用. 在加入人工变量后, 将目标函数设置为

$$\max z = \sum_{i=1}^{n} c_n x_n - M x_{n+1} - M x_{n+2} - \cdots - M x_{n+m}, \tag{8}$$

其中 M 是一个很大的正数. 因为是对目标函数最大化, 因此这样做可以保证人工变量可以很快地从基变量中换出去. 实际应用中一般可以取

$$M = 200 \max_{i \in [1,n]} |c_i|. \tag{9}$$

在最后得到的最优解中, 若人工变量都在非基变量位置, 那么当前最优解去掉人工变量后即为原问题最优解.

若最优解中包含等于零的人工变量, 那么可以将某非基变量引入基变量中来替换该人工变量, 从而得到问题的最优解.

若最优解中包含不等于零的人工变量, 那么原问题无可行解.

[2]本节参考 [1, p. 53]

3 最优化流程

改进单纯形法配合大 M 法的基本流程图1所示.

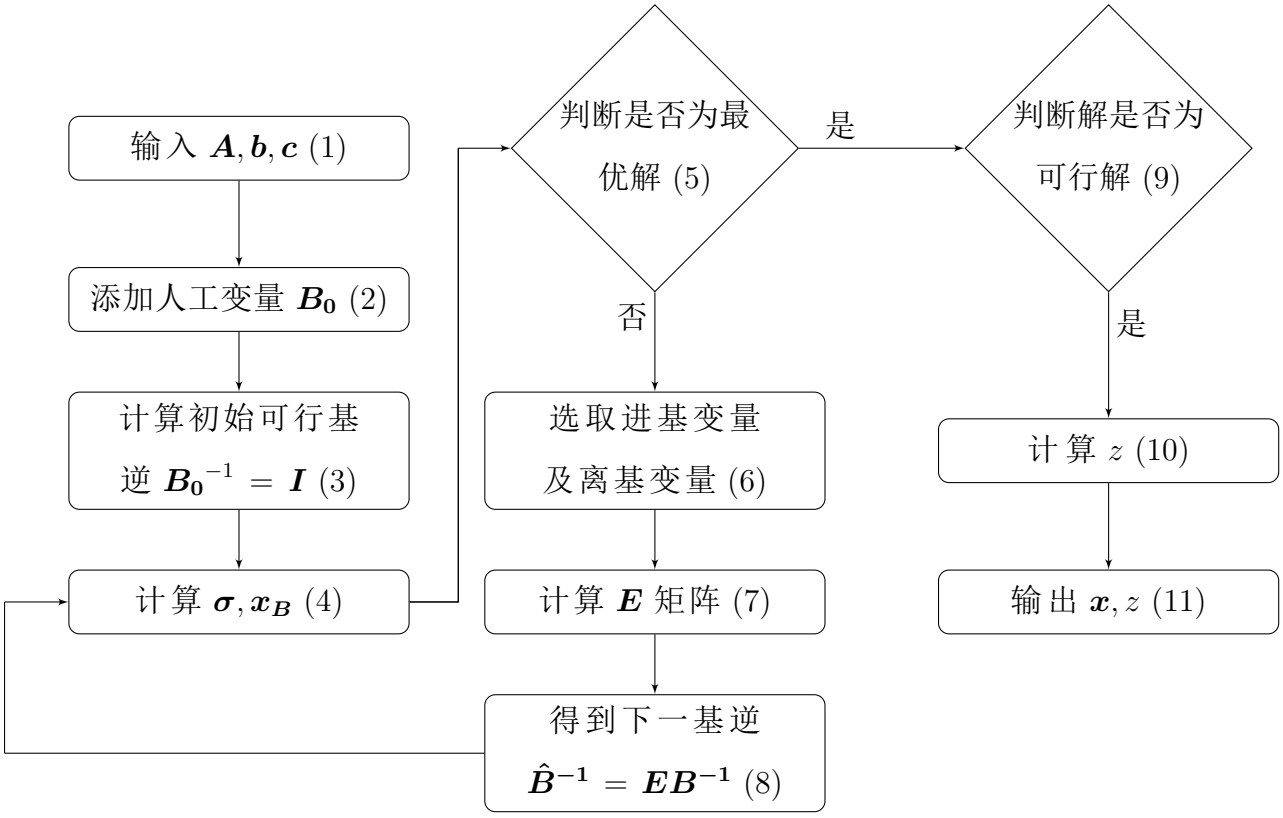

图 1: 改进单纯形法的基本流程图

该流程中的矩阵都可以使用稀疏矩阵来表示, 当内存超过用量时自动停止计算.

3.1 流程分步分析

以下讨论一些特定步骤的最优化处理, 以致于可以让计算机尽可能地处理更大的问题. 在讨论中假定计算机的内存空间有限, 而硬盘空间是近乎无限的, 并且矩阵是按行存储于设备中.

3.1.1 第 1 步

在大规模的线性问题中, $\boldsymbol{A}$ 是一个非常巨大的矩阵, 无法被完整地存储在计算机内存中, 因此完整的矩阵将会被保存到计算机的硬盘中. 计算机的硬盘的随机访问速度大约是内存的十万分之一, 顺序访问速度大约是内存的七分之一.[2, Fig. 3] 这就代表对 $\boldsymbol{A}$ 的访问将会非常的昂贵, 尤其是按列访问, 因为此时需要磁盘多次寻址, 等同于该列所有元素都

会触发随机访问请求. 一个补偿的方法是将该矩阵的转置也储存在内存中, 这样在以后读取列的时候速度将会有明显的提升.

3.1.2 第 4 步

$\boldsymbol{x_B}$ 向量可以用内存中的数据直接得出, 而计算 $\boldsymbol{\sigma}$ 向量需要使用 $\boldsymbol{N}$ 矩阵, 因此要在硬盘中按行读取 $\boldsymbol{A}$ 的转置矩阵. 在计算时可以先计算 $\boldsymbol{c_B B^{-1}}$, 然后再与 $\boldsymbol{N}$ 相乘.

3.1.3 第 7 步

$\boldsymbol{E}$ 矩阵是一个只有 $2m-1$ 个元素的稀疏矩阵, 存储它的空间非常的小. 同时由于初始可行基逆是稀疏矩阵, 因此最后相乘后所得的新基逆也会具有一定的稀疏特性.

4 C++ 实现与测试

本文中的算法的 C++ 实现代码可见附录中, 同是代码也托管于 GitHub 上[3]. 在测试中可以发现在硬盘中读取矩阵的速度与在内存中相当, 可能是由于磁盘缓存的缘故. 这就意味着计算速度并未受到太大的影响. 但是生成转置矩阵非常耗时, 往往需要比写入原矩阵多花一百倍左右的时间.

但是因为很难构造出一个包含稀疏矩阵的测试问题, 因此所写的测试案例使用的是以随机数填充的一个稠密矩阵. 即使如此, 对一个 2000×3000 稠密矩阵进行测试时, 可以发现开始计算时占用的内存约为原矩阵的五十分之一. 随着基逆矩阵变得越来越稠密, 程序内存占用也越来越大. 但是还是可以看出本方案可以用于应对大规模线性规划问题.

由于在一些部分未实现并行化处理, 在多核处理器上运行时可以发现处理器占用率并不高, 因此本实现的处理速度比较慢. 对以上 2000×3000 规模的问题处理可能需要 10 至 20 分钟左右的时间. 在进行改进之后可以达到更快的运行速度.

5 附录

5.1 C++ 实现代码

[3] https://github.com/xziyue/Large-Scale-Linear-Programming

```cpp
#ifndef DEF___INCLUDE_HPP
#define DEF___INCLUDE_HPP

// Dependent on the Eigen library
#include <Eigen/Dense>
#include <Eigen/Sparse>

#include <vector>
#include <iostream>
#include <stdexcept>
#include <utility>
#include <string>
#include <fstream>
#include <iomanip>
#include <chrono>
#include <memory>
#include <limits>
#include <map>

using namespace std;


inline void expr_check(bool expr, const char* info) {
    if (!expr) {
        cerr << info << "\n";
        throw runtime_error{ info };
    }
}

// error tolerance
const double mach_eps = 1.0e-8;


template<typename V>
inline bool fpeq(V left, V right) {
    if (abs(right - left) < (V)mach_eps)return true;
    else return false;
}

#define COMPILE_TEST


#endif // !DEF___INCLUDE_HPP


#ifndef DEF_ONDISKMATRIXTYPEINFO_HPP
#define DEF_ONDISKMATRIXTYPEINFO_HPP




struct TypeInfoDefined {
    static constexpr bool pass = true;
```

```
    static constexpr char* info = "";
};

struct TypeInfoUndefined {
    static constexpr bool pass = false;
    static constexpr char* info = "Type info undefined";
};

template<typename CheckClass>
struct TemplateChecker {
    using CheckClassType = CheckClass;
    TemplateChecker() {
        if (!CheckClass::pass) {
            cerr << CheckClass::info << "\n";
            throw runtime_error{ CheckClass::info };
        }
    }
};



template<typename V, typename InfoDefineChecker = TypeInfoUndefined>
struct TypeInfo :public TemplateChecker<InfoDefineChecker> {
    const int32_t type_size = sizeof(V);
    const char type_hint[3] = { '\0','\0','\0' };
};

template<typename InfoDefineChecker>
struct TypeInfo<double, InfoDefineChecker> :public TemplateChecker<
   TypeInfoDefined> {
    const int32_t type_size = sizeof(double);
    const char type_hint[3] = { 'f','6','4' };
};

template<typename InfoDefineChecker>
struct TypeInfo<float, InfoDefineChecker> :public TemplateChecker<
   TypeInfoDefined> {
    const int32_t type_size = sizeof(float);
    const char type_hint[3] = { 'f','3','2' };
};


#endif // !DEF_ONDISKMATRIXTYPEINFO_HPP

#ifndef DEF_ONDISKMATRIX_HPP
#define DEF_ONDISKMATRIX_HPP


constexpr int type_hint_length = 3;

struct OnDiskMatrixHeader {
    int32_t rows;
    int32_t cols;
```

```
    int32_t type_size;
    char type_hint[type_hint_length];
};

constexpr int OnDiskMatrixHeader_size = sizeof(OnDiskMatrixHeader);


template<typename V>
class OnDiskMatrixBase :protected TypeInfo<V> {
public:
    using value_type = V;
    using MatrixType = Eigen::Matrix<value_type, -1, -1,Eigen::RowMajor>;

    // opens a matrix from file
    OnDiskMatrixBase(const string &filename) {
        file.open(filename, ios::binary | ios::in | ios::out);

        // read the header
        file.seekg(ios::beg);
        file.read(reinterpret_cast<char*>(&header), OnDiskMatrixHeader_size
   );

        // check validity
        verify_type();
    }

    // create a new empty matrix
    OnDiskMatrixBase(const string &filename, int rows, int cols) {
        // initialize header
        header.rows = rows;
        header.cols = cols;
        header.type_size = type_size;
        for (auto i = 0; i < type_hint_length; ++i) {
            header.type_hint[i] = type_hint[i];
        }

        // open file
        file.open(filename, ios::binary|ios::in|ios::out|ios::trunc);
        //file.seekp(ios::beg);

        // write header
        file.write(reinterpret_cast<const char*>(&header),
   OnDiskMatrixHeader_size);

        // fill the matrix with initialization value
        fill(value_type{});
    }

    virtual ~OnDiskMatrixBase() {
        //file.close();
    };
```

```
    // force writing data into file
    void flush() {
        file << std::flush;
    }

    MatrixType read_row(int row_ptr) {
        MatrixType ret{ 1,header.cols };
        file.seekg(get_element_location(row_ptr, 0));
        file.read(reinterpret_cast<char*>(ret.data()), header.cols *
type_size);
        return ret;
    }

    // fill the entire matrix with a value
    void fill(const value_type &val) {
        MatrixType row{1,header.cols};
        row.fill(val);
        for (auto i = 0; i < header.rows; ++i) {
            write_row(row, i);
        }
    }

    void write_row(const MatrixType& matrix, int row_ptr) {
        assert(matrix.cols() == header.cols);

        file.seekp(get_element_location(row_ptr, 0));
        file.write(reinterpret_cast<const char*>(matrix.data()), header.
cols * type_size);
        flush();
    }

    // avoid using
    value_type get_element(int row_ptr, int col_ptr) {
        value_type ret;
        file.seekg(get_element_location(row_ptr, col_ptr));
        file.read(reinterpret_cast<char*>(&ret), type_size);
        return ret;
    }

    // avoid using
    value_type set_element(const value_type &val, int row_ptr, int col_ptr)
    {
        file.seekp(get_element_location(row_ptr, col_ptr));
        file.write(reinterpret_cast<const char*>(&val), type_size);
    }

    // generate another matrix, which is the transpose of this matrix
    void generate_transpose_matrix(const string& filename) {
        // generate a new matrix
        OnDiskMatrixBase<value_type> new_matrix{ filename,header.cols,
header.rows };
        for (auto i = 0; i < header.rows; ++i) {
            auto row = move(read_row(i));
```

```
            row.transposeInPlace();
            new_matrix.write_col(row, i);
        }
    }

    const OnDiskMatrixHeader &get_header() const { return header; }

    int rows() const { return header.rows; }
    int cols() const { return header.cols; }

protected:
    OnDiskMatrixHeader header;
    fstream file;

    streampos get_element_location(int row_ptr, int col_ptr) {
        assert(row_ptr > -1 && row_ptr < header.rows);
        assert(col_ptr > -1 && col_ptr < header.cols);

        return (streampos)OnDiskMatrixHeader_size +
            ((streampos)row_ptr * (streampos)header.cols +
            (streampos)col_ptr) * type_size;
    }

    void write_col(const MatrixType& matrix, int col_ptr) {
        assert(matrix.rows() == header.rows);

        for (auto i = 0; i < header.rows; ++i) {
            file.seekp(get_element_location(i, col_ptr));
            file.write(reinterpret_cast<const char*>(&(matrix(i,0))),
   type_size);
        }
        flush();
    }

    virtual void verify_type() {

        expr_check(header.type_size == type_size, "The size of value type
   does not match.");
        for (auto i = 0; i < type_hint_length; ++i) {
            expr_check(header.type_hint[i] == type_hint[i], "The type
   description information does not match.");
        }

    }
};


template<typename V>
class OnDiskMatrix :public OnDiskMatrixBase<V> {
public:
    using base_type = OnDiskMatrixBase<V>;

    OnDiskMatrix(const string &filename) :base_type{ filename } {};
```

```
    OnDiskMatrix(const string &filename, int rows, int cols) :base_type{
   filename,rows,cols } {}
    OnDiskMatrix(const OnDiskMatrix& other) = delete;
    OnDiskMatrix(OnDiskMatrix&& other) = delete;
    virtual ~OnDiskMatrix(){}
};



#endif // !DEF_ONDISKMATRIX_HPP

#ifndef DEF_SIMPLEXMETHOD_HPP
#define DEF_SIMPLEXMETHOD_HPP


struct InfiniteSolutionsError :public exception {
    using exception::exception;
};

struct NoSolutionError :public exception {
    using exception::exception;
};

template<typename V>
class SimplexMethod {
public:
    using value_type = V;
    using const_reference = const V&;
    using reference = V&;
    using DiskMatrixType = OnDiskMatrix<value_type>;
    using SparseMatrixType = Eigen::SparseMatrix<value_type>;
    using DenseMatrixType = Eigen::Matrix<value_type, -1, -1>;
    using SolutionType = map<int, value_type>;

    // run simplex method with original matrix
    SimplexMethod(const string &filename, const DenseMatrixType &_vec_b,
   const DenseMatrixType &_vec_c) {
        init_not_extended(filename,_vec_b,_vec_c);
    }

    /*
    // NOT IMPLEMENTED YET
    // run simplex method with matrix with artificial variables already
    SimplexMethod(const string &extended_mat_filename, const string &
   extended_mat_trans_filename,
        const DenseMatrixType &_vec_b, const DenseMatrixType &_vec_c) {


    }
    */

    // returns the map of solutions, set val to be the maximum value
    SolutionType solve(value_type &val) {
```

```
        auto run = 0;
        while (!run_once()) {
            cout << "running iteration :" << (run++) << "\n";
        }

        cout << "solving finished.\n";
        // generate solution map
        SolutionType sol;
        auto x_b = get_x_b_vec();
        for (auto i = 0; i < x_b.rows(); ++i) {
            sol.insert(SolutionType::value_type{base[i],x_b(i,0)});
        }

        // not the most efficient way to get z though...
        val = get_z();
        return sol;
    }

protected:
    using TripletType = Eigen::Triplet<value_type>;
    unique_ptr<DiskMatrixType> ondisk_mat;
    unique_ptr<DiskMatrixType> ondisk_trans;
    DenseMatrixType vec_b;
    DenseMatrixType vec_c;
    SparseMatrixType B_inv;
    vector<int> base;
    vector<int> non_base;


    template<typename K>
    typename vector<K>::size_type guaranteed_sequencial_find(const vector<K
   > &vec, const K &target) {
        using size_t = vector<K>::size_type;
        size_t pos = 0;
        for (; pos < vec.size(); ++pos) {
            if (vec[pos] == target)return pos;
        }
        throw runtime_error{ "target not found." };
    }

    template<typename K>
    typename vector<K>::size_type guaranteed_find_max(const vector<K> &vec)
    {
        using size_t = vector<K>::size_type;
        size_t pos = 0;
        K value = vec[pos];

        for (size_t i = 1; i < vec.size(); ++i) {
            if (vec[i] > value) {
                value = vec[i];
                pos = i;
            }
        }
```

```
        return pos;
    }

    value_type get_z() {
        return (get_c_b_vec() * B_inv * vec_b)(0, 0);
    }

    DenseMatrixType get_x_b_vec() {
        return B_inv * vec_b;
    }

    DenseMatrixType get_c_n_vec() {
        DenseMatrixType ret{ 1,non_base.size() };
        auto i = 0;
        for (auto iter = non_base.begin(); iter != non_base.end(); ++iter,
++i) {
            ret(0, i) = vec_c(0, *iter);
        }
        return ret;
    }

    DenseMatrixType get_c_b_vec() {
        DenseMatrixType ret{ 1,base.size() };
        auto i = 0;
        for (auto iter = base.begin(); iter != base.end(); ++iter, ++i) {
            ret(0, i) = vec_c(0, *iter);
        }
        return ret;
    }

    DenseMatrixType get_sigma_vec() {
        DenseMatrixType ret{ 1,non_base.size() };


        auto c_b{ move(get_c_b_vec()) };
        auto c_n{ move(get_c_n_vec()) };

        DenseMatrixType product_row = c_b * B_inv;

        auto i = 0;
        for (auto iter = non_base.begin(); iter != non_base.end(); ++iter
,++i) {
            auto col = ondisk_trans->read_row(*iter);;
            col.transposeInPlace();
            ret(0, i) = (product_row * col)(0, 0);
        }

        return c_n - ret;
    }

    void base_alteration(int out_pos, int in_pos, const DenseMatrixType &
y_k) {
```

```
        value_type major_element = y_k(out_pos,0);

        vector<TripletType> elements;

        // add elements that are in the major column
        for (auto i = 0; i < B_inv.rows(); ++i) {
            if (i == out_pos)elements.emplace_back(TripletType{ i, out_pos,
 (value_type)1 / major_element });
            else elements.emplace_back(TripletType{ i, out_pos, -y_k(i,0) /
 major_element });
        }

        // add elements in the rest of columns
        for (auto i = 0; i < B_inv.cols(); ++i) {
            if (i != out_pos)elements.emplace_back(TripletType{ i,i,(
value_type)1});
        }

        // form the E matrix
        SparseMatrixType mat_e{ B_inv.rows(),B_inv.cols() };
        mat_e.setFromTriplets(elements.begin(), elements.end());

        B_inv = (mat_e * B_inv).eval();

        // set the base vectors
        auto out = base[out_pos];
        base[out_pos] = non_base[in_pos];
        non_base[in_pos] = out;
    }

    bool run_once() {
        // optimal condition check
        bool optimal = true;
        auto sigma_vec = get_sigma_vec();

        for (auto i = 0; i < sigma_vec.cols(); ++i) {
            if (sigma_vec(0, i) > mach_eps) {
                optimal = false;
                break;
            }
        }

        if (optimal) {
            // make sure no artificial variable is in the base
            for (auto iter = base.begin(); iter != base.end(); ++iter) {
                if (*iter >= B_inv.rows())throw NoSolutionError{ "no
solution" };
            }
            return true;
        }


        // find the one that should go into base
```

```
        auto into_base = 0;
        value_type max_val = sigma_vec(0, 0);
        for (auto i = 1; i < sigma_vec.cols(); ++i) {
            if (sigma_vec(0, i) > max_val) {
                into_base = i;
                max_val = sigma_vec(0, i);
            }
        }

        // determine if there is infinite solution
        bool no_sol = true;
        auto p_k = ondisk_trans->read_row(non_base[into_base]);
        p_k.transposeInPlace();
        for (auto i = 0; i < p_k.rows(); ++i) {
            if (p_k(i,0) > mach_eps) {
                no_sol = false;
                break;
            }
        }

        if (no_sol)throw InfiniteSolutionsError{ "infinite solution" };

        // find the element that should go out of base
        DenseMatrixType vec_test{ vec_b.rows(),1 };
        auto y_k = (B_inv * p_k);
        auto x_b = get_x_b_vec();
        for (auto i = 0; i < y_k.rows(); ++i) {
            if (y_k(i,0) < mach_eps)vec_test(i,0) = numeric_limits<
   value_type>::max();
            else vec_test(i,0) = x_b(i,0) / y_k(i,0);
        }

        auto out_of_base = 0;
        value_type min_val = vec_test(0,0);
        for (auto i = 1; i < vec_test.rows(); ++i) {
            if (vec_test(i,0) < min_val) {
                min_val = vec_test(i,0);
                out_of_base = i;
            }
        }

        base_alteration(out_of_base,into_base,y_k);

        return false;
    }
private:
    void warn_no_solution(){
        cerr << "the current problem does not have a solution.";
        throw runtime_error{ "no solution." };
    }

    void fill_row(const DenseMatrixType &old_row, DenseMatrixType &new_row,
    int fill_pos) {
```

```
        // copy from old row
        for (auto i = 0; i < old_row.cols(); ++i) {
            new_row(0, i) = old_row(0, i);
        }

        // fill the rest with zero
        for (auto i = old_row.cols(); i < new_row.cols(); ++i) {
            new_row(0, i) = (value_type)0.0;
        }

        // set target position one
        new_row(0, old_row.cols() + fill_pos) = (value_type)1.0;
    }

    // after init, check whether the size of matrices are correct
    void vector_size_check(DiskMatrixType &original_matrix, const
DenseMatrixType &_vec_b, const DenseMatrixType &_vec_c) {
        const char* size_matching_info = "matrix size does not match";
        expr_check(original_matrix.cols() >= original_matrix.rows(), "the
size of matrix is invalid");
        expr_check(original_matrix.rows() == _vec_b.rows(),
size_matching_info);
        expr_check(original_matrix.cols() == _vec_c.cols(),
size_matching_info);

        for (auto i = 0; i < _vec_b.rows(); ++i) {
            expr_check(_vec_b(i, 0) > mach_eps, "invalid b vector");
        }
    }

    value_type find_big_M(const DenseMatrixType &vec_c) {
        auto max = numeric_limits<value_type>::min();
        for (auto i = 0; i < vec_c.cols(); ++i) {
            if (abs(vec_c(0, i)) > max)max = abs(vec_c(0, i));
        }

        // suppose 200 is a good amplification
        return (value_type)200 * max;
    }

    void init_not_extended(const string &filename, const DenseMatrixType &
_vec_b, const DenseMatrixType &_vec_c) {
        // open the matrix file
        OnDiskMatrix<value_type> original_mat{ filename };

        vector_size_check(original_mat, _vec_b, _vec_c);

        // create a new matrix
        cout << "generating extended matrix...\n";
        auto new_filename = filename + string{ "_extended" };
        ondisk_mat = move(unique_ptr<DiskMatrixType>{ new DiskMatrixType{
new_filename,original_mat.rows(),original_mat.cols() + original_mat.rows
() } });
```

```
        // add artificial variables
        DenseMatrixType new_row{ 1,ondisk_mat->cols() };
        for (auto i = 0; i < original_mat.rows(); ++i) {
            auto old_row = move(original_mat.read_row(i));
            fill_row(old_row, new_row, i);
            //cout << old_row << endl;
            //cout << new_row << endl;
            ondisk_mat->write_row(new_row, i);
        }


        // generate transpose matrix
        cout << "generating transpose matrix...\n";
        auto trans_filename = filename + string{ "_t" };
        ondisk_mat->generate_transpose_matrix(trans_filename);
        ondisk_trans = move(unique_ptr<DiskMatrixType>{new DiskMatrixType{
    trans_filename }});

        // copy b
        vec_b = _vec_b;

        // init c
        vec_c = DenseMatrixType{ 1,ondisk_mat->cols() };
        for (auto i = 0; i < _vec_c.cols(); ++i) {
            vec_c(0, i) = _vec_c(0, i);
        }
        // fill big M value
        auto big_M = find_big_M(_vec_c);
        for (auto i = _vec_c.cols(); i < vec_c.cols(); ++i) {
            vec_c(0, i) = -big_M;
        }

        // set base and non-base pointers
        for (int i = 0; i < _vec_c.cols(); ++i) {
            non_base.push_back(i);
        }
        for (int i = (int)_vec_c.cols(); i < (int)vec_c.cols(); ++i) {
            base.push_back(i);
        }

        // set B_inv matrix
        vector<TripletType> elements;
        B_inv = move(SparseMatrixType{ ondisk_mat->rows() ,ondisk_mat->rows
    () });
        for (auto i = 0; i < ondisk_mat->rows(); ++i) {
            elements.emplace_back(TripletType(i, i, (value_type)1));
        }
        B_inv.setFromTriplets(elements.begin(), elements.end());
    }
};
```

```
#endif // !DEF_SIMPLEXMETHOD_HPP
```

参考文献

[1] 何坚勇. 最优化方法. 第一版. 清华大学出版社, 2007.

[2] Adam Jacobs. *The Pathologies of Big Data*. 2009. URL: http://queue.acm.org/detail.cfm?id=1563874 (visited on 06/07/2017).