Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(ФГБОУ ВО «СибГУТИ»)

*Кафедра прикладной математики и кибернетики*

Курсовая работа по дисциплине

«Алгоритмы и вычислительные методы оптимизации»

Выполнил: студент гр. ИП-513

Санин И. В.

Проверил:

ассистент кафедры ПМиК

Бочкарев Б.В.

Новосибирск

2018

**Оглавление**

[Постановка задачи 3](#_Toc514704131)

[Теоретическая часть 3](#_Toc514704132)

[Метод Гаусса-Жордана 3](#_Toc514704133)

[Симплекс-метод 4](#_Toc514704134)

[Метод искусственного базиса 9](#_Toc514704135)

[Метод минимальной стоимости 10](#_Toc514704136)

[Метод потенциалов 10](#_Toc514704137)

[Листинг 12](#_Toc514704138)

[fracGauss.cpp 12](#_Toc514704139)

[fracmatrixsymplex.h 17](#_Toc514704140)

[fracmatrixsymplex.cpp 18](#_Toc514704141)

[planningmatrix.h (Транспортная задача) 26](#_Toc514704142)

[planningmatrix.cpp (Транспортная задача) 27](#_Toc514704143)

[Заключение 35](#_Toc514704144)

# Постановка задачи

Рассмотреть следующие алгоритмы и вычислительные методы оптимизации решения задач линейного программирования:

* Метод Гаусса-Жордана
* Симплекс-метод
* Метод искусственного базиса
* Алгоритм нахождения первоначального опорного плана в транспортной задаче (метод минимальной стоимости)
* Метод потенциалов

# Теоретическая часть

## Метод Гаусса-Жордана

Рассмотрим систему из m линейных уравнений с n неизвестными:

*а11х1 + а12х2 + ... + а1nхn =b1*

*а21х1 + а22х2 + ... + а2nхn =b2*

*...*

*аm1х1 + аm2х2 + ... + аmnхn =bm*

Обозначим через А = (аij) матрицу системы, через X = (xj) - матрицу-столбец, состоящую из неизвестных, через А0 = (bi) - матрицу-столбец, состоящую из свободных членов; тогда систему выше можно записать в виде матричного уравнения

*АХ = А0.* (1)

При m = n матрица А является квадратной; если ее определитель |А| != 0, то А-1 - обратная ей матрица. Умножая матричное уравнение (1) слева на обратную матрицу, получаем

*ЕХ = А-1А0*,

где Е — единичная матрица, а произведение матрицы А-1А0 - решение системы.

Нахождение обратной матрицы через алгебраические дополнения - трудоемкий вычислительный процесс, поэтому при решении системы линейных уравнений воспользуемся численным методом, который позволяет с помощью элементарных преобразований за конечное число шагов найти решение (если оно существует) и при необходимости получить обратную матрицу. Этот метод называется методом полного исключения неизвестных или методом Жордана — Гаусса. Суть метода состоит в том, что, рассмотрев первое уравнение, а в нем неизвестное с коэффициентом, отличным от нуля (он в дальнейшем называется разрешающим элементом), и разделив первое уравнение на этот коэффициент, с помощью первого уравнения исключают это неизвестное из всех уравнений, кроме первого. Выбрав во втором уравнении неизвестное с коэффициентом, отличным от нуля, и разделив на него второе уравнение, с помощью второго уравнения исключают другое неизвестное из всех уравнений, кроме второго, и т. д., т. е. с помощью одного уравнения производят полное исключение одного неизвестного.

Процесс продолжают до тех пор, пока не будут использованы все уравнения. При этом возможны следующие случаи.

1. В процессе исключений левая часть i-го уравнения системы обращается в нуль, а правая часть равна некоторому числу, отличному от нуля, т. е. имеет место равенство 0 = bi != 0. Это означает, что система не имеет решений, так как i-му уравнению не могут удовлетворить никакие значения неизвестных.

2. Левая и правая части 1-го уравнения обращаются в нуль. Это означает, что 1-е уравнение является линейной комбинацией остальных, ему удовлетворяет любое найденное решение системы, поэтому оно может быть отброшено.

3. После того, как все уравнения использованы для исключения неизвестных, либо будет получено решение, либо доказано, что система несовместна.

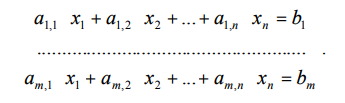
## Симплекс-метод

Метод предназначен для решения общей задачи линейного программирования.

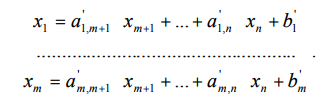
Пусть имеем следующую задачу:



с системой ограничений следующего вида:



Разрешим эту систему относительно переменных x1, …, xm:



Векторы условий, соответствующие x1, …, xm, образуют базис. Переменные x1, …, xm назовем базисными переменными. Остальные переменные задачи - небазисные.

Целевую функцию можно выразить через небазисные переменные:



Если приравнять небазисные переменные нулю, то соответствующие базисные переменные примут значения



Вектор x с такими компонентами представляет собой угловую точку многогранника решений (допустимую) при условии, что 

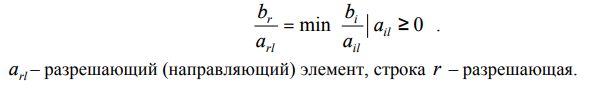
Теперь необходимо перейти к другой угловой точке с меньшим значением целевой функции. Для этого следует выбрать некоторую небазисную переменную и некоторую базисную так, чтобы после того, как мы “поменяем их местами”, значение целевой функции уменьшилось. Такой направленный перебор в конце концов приведет нас к решению задачи

Формализованный алгоритм симплекс метода состоит из двух основных этапов: 1) построение опорного плана; 2) построение оптимального плана.

Для того чтобы опорный план был оптимален, при минимизации целевой функции необходимо, чтобы коэффициенты в строке целевой функции были неположительными (в случае максимизации - неотрицательными). Т.е. при поиске минимума мы должны освободиться от положительных коэффициентов в строке Q(x).

Если при поиске минимума в строке целевой функции есть коэффициенты больше нуля, то выбираем столбец с положительным коэффициентом в строке целевой функции в качестве разрешающего. Пусть это столбец с номером l.

Для выбора разрешающей строки (разрешающего элемента) среди положительных коэффициентов разрешающего столбца выбираем тот (строку), для которого отношение коэффициента в столбце свободных членов к коэффициенту в разрешающем столбце минимально:

(2)

Для перехода к следующей симплексной таблице (следующему опорному плану с меньшим значением целевой функции) делается шаг модифицированного жорданова исключения с разрешающим элементом arl .

Если в разрешающем столбце нет положительных коэффициентов, то целевая функция неограничена снизу (при максимизации – неограничена сверху).

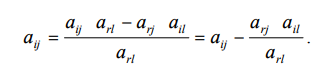
Шаг модифицированного жорданова исключения над симплексной таблицей.

1. На месте разрешающего элемента ставится 1 и делится на разрешающий элемент.

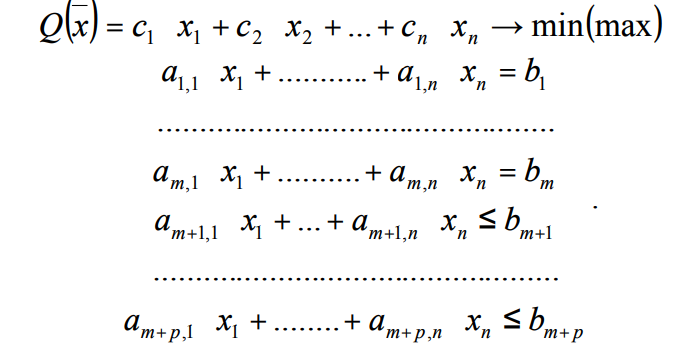
2. Остальные элементы разрешающего столбца меняют знак на противоположный и делятся на разрешающий элемент.

3. Остальные элементы разрешающей строки делятся на разрешающий элемент.

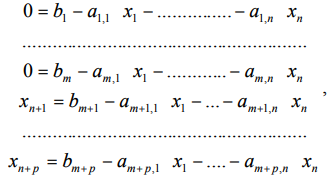
4. Все остальные элементы симплексной таблицы вычисляются по следующей формуле:



Пусть необходимо решить задачу:

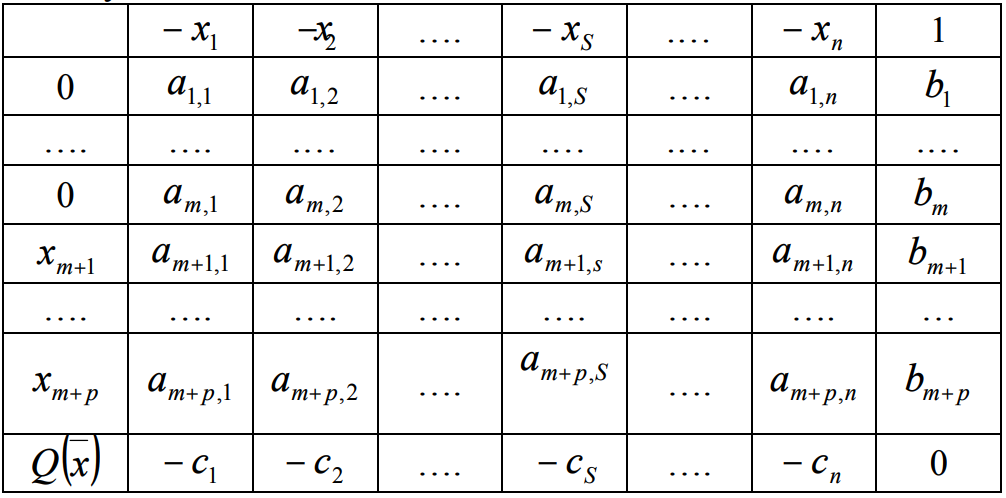


Введем дополнительные переменные, чтобы преобразовать ограничения-неравенства к равенствам. В ограничениях-равенствах дополнительные переменные должны быть нулевыми. Тогда система ограничений принимает вид:



где 

В качестве базисных переменных будем брать систему дополнительно введенных переменных. Тогда симплексная таблица для преобразованной задачи будет иметь следующий вид:



Правила выбора разрешающего элемента при поиске опорного плана.

1. При условии отсутствия “0-строк” (ограничений-равенств) и “свободных” переменных (т.е. переменных, на которые не наложено требование неотрицательности).

* Если в столбце свободных членов симплексной таблицы нет отрицательных элементов, то опорный план найден.
* Есть отрицательные элементы в столбце свободных членов, например bi < 0. В такой строке ищем отрицательный коэффициент ail , и этим самым определяем разрешающий столбец l. Если не найдем отрицательный ail, то система ограничений несовместна (противоречива).
* В качестве разрешающей выбираем строку, которой соответствует отношение (2).
* После того, как разрешающий элемент найден, делаем шаг модифицированного жорданова исключения с направляющим элементом arl и переходим к следующей симплексной таблице.

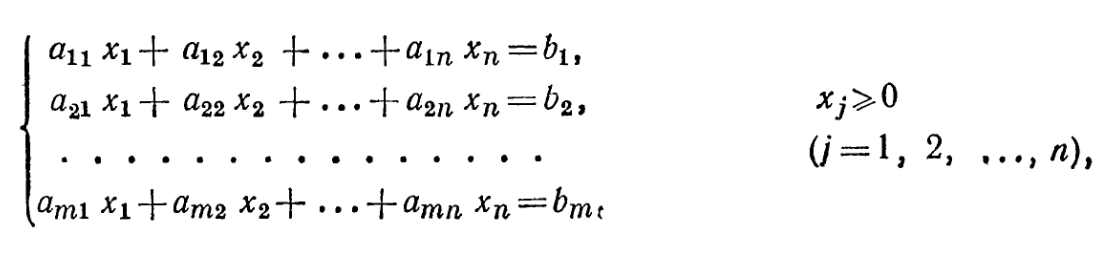
2. В случае присутствия ограничений-равенств и “свободных” переменных поступают следующим образом.

* Выбирают разрешающий элемент в “0-строке” и делают шаг модифицированного жорданова исключения, после чего вычеркивают этот разрешающий столбец. Данную последовательность действий продолжают до тех пор, пока в симплексной таблице остается хотя бы одна “0-строка” (при этом таблица сокращается).
* Если же присутствуют и свободные переменные, то необходимо данные переменные сделать базисными. И после того, как свободная переменная станет базисной, в процессе определения разрешающего элемента при поиске опорного и оптимального планов данная строка не учитывается (но преобразуется).

## Метод искусственного базиса

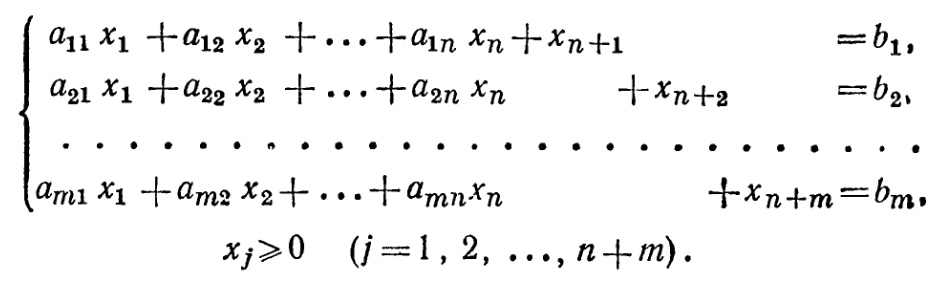
Рассмотрим общую задачу линейного программирования.

Найти минимальное значение линейной функции Z = С1х1 + С2х2 + ... + + Сnхn при ограничениях



где bi ≥ 0 и система ограничений не содержит единичной матрицы.

Для получения единичной матрицы к каждому равенству прибавим по одной переменной хn+l ≥ 0 (l = 1, 2, ..., m), которые назовем искусственными, и рассмотрим расширенную задачу, связанную с отысканием наименьшего значения линейной функции Z = С1х1 + С2х2 + ... + + Сnхn + Мхn+1 + ... + Мхn+m при ограничениях



Величина М предполагается достаточно большим положительным числом, если задача решается на отыскание минимального значения линейной функции, и достаточно малым отрицательным числом, если находится максимальное значение линейной функции. Единичные векторы Аn+1, Аn+2, ..., Аn+m> соответствующие искусственным переменным, образуют искусственный базис.

Для отыскания оптимального плана исходной задачи используют следующую теорему:

*Если в оптимальном плане X = (x1, х2, ..., хn, 0, ..., 0) расширенной задачи искусственные переменные хn+i = 0 (i = 1, 2, ..., m), то план X = (x1, х2, ..., хn) является оптимальным планом исходной задачи.*

Применение симплексного метода к расширенной задаче обеспечивает построение плана, в котором каждое из искусственных переменных хn+l=0. Если первоначальная задача не обладает планами (т.е. она не совместна), то оптимальное решение расширенной задачи содержит по крайней мере одно хn+l > 0.

Для отыскания оптимального плана расширенной задачи в случае, если заранее не задана величина М, применяется симплексный метод с составлением симплексных таблиц, которые имеют на одну строку больше, чем обычная симплексная таблица. По этой (m+2)-й строке определяют вектор, подлежащий включению в базис. Итерационный процесс по (m+2)-й строке проводят для исключения из базиса всех искусственных векторов, затем процесс отыскания оптимального плана продолжают по (m+1)-й строке.

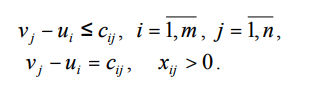
## Метод минимальной стоимости

Для построения опорного плана транспортной задачи обычно используется один из двух методов: метод северо-западного угла или метод минимального элемента. Рассмотрим второй.

В таблице отыскивается min{cij} и в первую очередь заполняется соответствующая клетка: xij = min{ai,bj}. Затем вычеркивается остаток соответствующей строки, если ai < bj, или столбца, если ai > bj, и корректируем остатки запасов и неудовлетворенного спроса. В оставшихся клетках таблицы снова отыскивается минимальная стоимость перевозки и заполняется соответствующая клетка и т.д.

## Метод потенциалов

Метод позволяет находить оптимальный план перевозок транспортной таблицы. В основе лежит следующая теорема:

*Для того, чтобы некоторый план X = [xij]m❌n транспортной задачи был оптимальным, необходимо и достаточно, чтобы ему соответствовала такая система m+n чисел u1, u2, ..., um; v1, v2, ..., vn, для которой выполняются условия:*

*(3)*

*(4)*

ui и vj называются потенциалами соответствующих пунктов отправления и пунктов назначения. Условия (3)-(4) называются условиями потенциальности.

План X будем называть потенциальным, если для него существует система ui и vj, удовлетворяющая (3)-(4).

Алгоритм метода потенциалов состоит из предварительного этапа и повторяющегося основного этапа.

* Предварительный этап.

1. Каким-либо способом ищется допустимый план X (методом северо-западного угла или минимального элемента).
2. Для полученного плана строится система m+n чисел u1, u2, ..., um; v1, v2, ..., vn, таких, что выполняется условие (4).
3. Построенная система ui и vj исследуется на потенциальность (то есть план X исследуется на оптимальность). Для этого проверяется условие (3).

Если система непотенциальна, то переходят к основному этапу (т.к. план не оптимален), иначе оптимальный план найден.

* Основной этап.

1. Улучшаем план, то есть от плана X переходим к плану X' такому, что Q(X) ≥ Q(X').
2. Для плана X' строим новую систему u'i и v'j, такую, то выполняется условие (4).
3. Исследуем систему u'i и v'j на потенциальность. Если система непотенциальна, то переходим на п.1. Иначе найден оптимальный план.

# Листинг

## fracGauss.cpp

|  |
| --- |
| #include <cmath>  #include <cstdio>  #include <iostream>  #include <string>  #include <cstdlib>  using namespace std;  class Frac  {  public:  Frac() : numer(0), denom(1) {};    Frac(int a, int b) {  if(b) {  numer = a;  denom = b;  toShorten();  setSign();  } else {  numer = 0;  denom = 1;  }  };    Frac(double num, int pres = 1000000) {  if(pres <= 0) {  numer = 0;  denom = 1;  } else {  numer = (int) (num \* pres);  denom = pres;  toShorten();  setSign();  }  };    //+-\*/    Frac operator+ (const Frac& sec) {  int resDenom = NOK(this->denom, sec.denom);  int resNumer = this->numer \* (resDenom / this->denom) + sec.numer \* (resDenom / sec.denom);  Frac res(resNumer, resDenom);  return res;  };    Frac operator+ (const double& sec) {  Frac tmpSec(sec);  Frac res = \*this + tmpSec;  return res;  };    Frac operator- (const Frac& sec) {  int resDenom = NOK(this->denom, sec.denom);  int resNumer = this->numer \* (resDenom / this->denom) - sec.numer \* (resDenom / sec.denom);  Frac res(resNumer, resDenom);  return res;  };    Frac operator- (const double& sec) {  Frac tmpSec(sec);  Frac res = \*this - tmpSec;  return res;  };    Frac operator\* (const Frac& sec) {  int resDenom = this->denom \* sec.denom;  int resNumer = this->numer \* sec.numer;  Frac res(resNumer, resDenom);  return res;  };    Frac operator\* (const double& sec) {  Frac tmpSec(sec);  Frac res = \*this \* tmpSec;  return res;  };    Frac operator/ (const Frac& sec) {  int resDenom = this->denom \* sec.numer;  int resNumer = this->numer \* sec.denom;  Frac res(resNumer, resDenom);  return res;  };    Frac operator/ (const double& sec) {  Frac tmpSec(sec);  Frac res = \*this / tmpSec;  return res;  };    //+-\*/=    Frac operator+= (const Frac& sec) {  \*this = \*this + sec;  return \*this;  };    Frac operator+= (const double& sec) {  Frac tmpSec(sec);  \*this = \*this + tmpSec;  return \*this;  };    Frac operator-= (const Frac& sec) {  \*this = \*this - sec;  return \*this;  };    Frac operator-= (const double& sec) {  Frac tmpSec(sec);  \*this = \*this - tmpSec;  return \*this;  };    Frac operator\*= (const Frac& sec) {  \*this = \*this \* sec;  return \*this;  };    Frac operator\*= (const double& sec) {  Frac tmpSec(sec);  \*this = \*this \* tmpSec;  return \*this;  };    Frac operator/= (const Frac& sec) {  \*this = \*this / sec;  return \*this;  };    Frac operator/= (const double& sec) {  Frac tmpSec(sec);  \*this = \*this / tmpSec;  return \*this;  };    //<>==    bool operator> (const Frac& sec) {  int resNumer = this->numer - sec.numer;  return resNumer > 0;  };    bool operator> (const double& sec) {  Frac tmpSec(sec);  return \*this > tmpSec;  };    bool operator< (const Frac& sec) {  int resNumer = this->numer - sec.numer;  return resNumer < 0;  };    bool operator< (const double& sec) {  Frac tmpSec(sec);  return \*this < tmpSec;  };    bool operator== (const Frac& sec) {  int resNumer = this->numer - sec.numer;  return resNumer == 0;  };    bool operator== (const double& sec) {  Frac tmpSec(sec);  return \*this == tmpSec;  };      Frac invert() {  Frac tmp (denom, numer);  return tmp;  };    Frac pow(int x) {    Frac tmp(numer, denom);    if(x < 0) {  tmp = tmp.invert();  x \*= -1;  }    Frac res = tmp;    for(int i = 0; i < x; ++i) {  res = res \* tmp;  }    return res;  }    void setNewNum(int newNumer, int newDenom) {  numer = newNumer;  denom = newDenom;  setSign();  toShorten();  }    void print() {  printf("%7i/%i", numer, denom);  }    private:  int numer;  int denom;  int GCD(int a, int b) {  return b ? GCD(b, a%b) : a;  };  int NOK(int a, int b) {  return a\*b / GCD(a, b);  };  void toShorten() {  int k = GCD(numer, denom);  numer /= k;  denom /= k;  };  void setSign() {  if(denom < 0) {  numer \*= -1;  denom \*= -1;  }  }  };  int size;  Frac \*\*arr;  Frac \*ans;  void swap(int fst, int sec)  {  int i;  Frac buff;  for(i = 0; i < (size + 1); ++i)  {  buff = arr[fst][i];  arr[fst][i] = arr[sec][i];  arr[sec][i] = buff;  }  }  int compare(int colNum)  {  Frac buff = arr[colNum][colNum];  int i;  int num = colNum;// number of row with max "a"  for(i = colNum + 1; i < size; ++i)  {  if(arr[i][colNum] > buff)  {  buff = arr[i][colNum];  num = i;  }  }  return num;  }  void print\_arr () {  for(int k = 0; k < size; ++k)  {  for(int l = 0; l < size+1; ++l) {  arr[k][l].print();  cout << " ";  }  cout << endl;  }  cout << endl;  }  int main() {    int i, j, k;  int numer;  Frac q;    FILE \*f = fopen("matrix.txt", "r+");    fscanf(f, "%i", &size);  cout << "size = " << size << endl << endl;  int bufNumer, bufDenom;  arr = new Frac \*[size];  ans = new Frac [size];  for(i = 0; i < size; ++i) {  arr[i] = new Frac [size + 1];  for(j = 0; j < size + 1; ++j) {  fscanf(f, "%i/%i", &bufNumer, &bufDenom);  arr[i][j].setNewNum(bufNumer, bufDenom);  arr[i][j].print();  cout << " ";  }    cout << endl;  }    cout << endl;    for(i = 0; i < size; ++i)  {  numer = compare(i);  if(numer != i)  swap(numer, i);  Frac buf = arr[i][i];  for(j = i; j < size+1; ++j) {  arr[i][j] /= buf;  }  for(j = 0; j < size; ++j)  {  if(j == i) continue;  q = arr[j][i];    for(k = i; k < size+1; ++k)  {  arr[j][k] -= arr[i][k]\*q;  }  }  print\_arr();  }    printf("Pryamoi khod\n");  for(i = 0; i < size; ++i)  {  for(j = 0; j < size+1; ++j) {  arr[i][j].print();  cout << " ";  }  cout << endl;  }    for(i = size-1; i >= 0; --i)  {  q = arr[i][size];  for(j = i+1; j < size; ++j)  {  q -= arr[i][j] \* ans[j];  }  ans[i] = q/arr[i][i];  ans[i] = arr[i][size];  }    printf("\nAnswers:\n");  for(j = 0; j < size; ++j) {  cout << "x" << j << " = ";  ans[j].print();  cout << endl;  }    system("pause");  return 0;  } |

## fracmatrixsymplex.h

|  |
| --- |
| #ifndef FRACMATRIXSYMPLEX\_H  #define FRACMATRIXSYMPLEX\_H  #include "../FracClass/frac.h"  #include <matrixlimitation.h>  #include <vector>  #include <iostream>  using namespace std;  class FracMatrixSymplex  {  public:  FracMatrixSymplex(Frac \*\*matrix, int x, int y, bool toMin = false, bool fake = false);  FracMatrixSymplex(vector<MatrixLimitation> matrix, MatrixLimitation target);  void rowAdd(int row, Frac \*add);  void rowMulti(int row, Frac mult);  void colMulti(int col, Frac mult);  // functions for vector<vector<Frac>> only  void rowInsert(int row, vector<Frac> add);  void colInsert(int col, vector<Frac> add);  void rowAdd(int row, vector<Frac> add);  void rowMulti2(int row, Frac mult);  void colMulti2(int col, Frac mult);  bool setNewSnapshot(); // return false if snapshot was set before  int columnIsBasis(int col);  void symplexMethod();  bool rebase(int inputColumn, int outputColumn);  void printMatrix();  void setBasis(int \*columns);  bool basisWasBefore(int \*columns);  void setZeroInArtif();  friend std::ostream &operator<<(std::ostream &str, FracMatrixSymplex& outMatrix);  private:  Frac \*\*m;//[xsize][ysize]  vector<pair<Frac \*\*, int \*>> baseSnapshot;  vector<vector<Frac>> m2;  int xsize;  int ysize;  int xmainvar;  bool toMin;  bool isFake;  void setBasisInColumn(int column, int row);  void findOptForBasisPlan(int &optimalRow, int &optimalCol);  void findLeadForBasisPlan(int &leadRow, int &leadCol);  void rectangleMethod(int optRow, int optColumn);  bool isOptimal();  };  #endif // FRACMATRIXSYMPLEX\_H |

## fracmatrixsymplex.cpp

|  |
| --- |
| #include "fracmatrixsymplex.h"  using namespace std;  FracMatrixSymplex::FracMatrixSymplex(Frac \*\*matrix, int x, int y, bool isMin, bool fake) : m(matrix), xsize(x), ysize(y)  {  toMin = isMin;  isFake = fake;  if(toMin && isFake) {  xmainvar = xsize - 2;  } else {  xmainvar = xsize - 1;  }  setNewSnapshot();  }  FracMatrixSymplex::FracMatrixSymplex(vector <MatrixLimitation> matrix, MatrixLimitation target)  {  //all limitations is equal length  // vector <Frac> buffer;  // for(int i = 0; i < matrix[0].getLine().size(); ++i) {  // m2.push\_back(buffer);  // for(int j = 0; j < matrix.size(); ++j) {  // m2[i].push\_back(matrix[j].getLine()[i]);  // }  // }  m2.resize(matrix[0].getLine().size());  for(unsigned int i = 0; i < matrix.size(); ++i) {  rowInsert(i, matrix[i].getLine());  }  rowInsert(matrix.size(), target.getLine());  int basei = 0;  for(unsigned int i = 0; i < matrix.size(); ++i) {  vector<Frac> buffer;  buffer.resize(m2[0].size());  switch(matrix[i].getLimit()) {  case MLIMIT\_EQSMALLER:  buffer[basei] = Frac(1, 1);  colInsert(m2.size() - 1, buffer);  basei++;  break;  case MLIMIT\_EQBIGGER:  buffer[basei] = Frac(1, 1);  colInsert(m2.size() - 1, buffer);  rowMulti2(basei, Frac(-1, 1));  basei++;  break;  // cases for all  }  buffer.clear();  }  // vector <Frac> matrBuff; // buffer for limitations/pointerf  // for(unsigned int i = 0; i < m2[0].size(); ++i) {  // matrBuff = matrix[i].getLine();  // for(unsigned int j = 0; j < m2.size(); ++j) {  // m2[j][i] = matrBuff[j];  // }  // matrBuff.clear();  // }  // for(unsigned int i = 0; i < m2.size(); ++i) {  // m2[i].resize(matrix.size() + 1);  // }  // matrBuff = target.getLine();  // for(unsigned int j = 0; j < m2.size(); ++j) {  // m2[j][m2[0].size() - 1] = matrBuff[j];  // }  // cout << endl << endl << "constructor:" << endl;  // for(unsigned int i = 0; i < m2[0].size(); ++i) {  // for(unsigned int j = 0; j < m2.size(); ++j) {  // cout << m2[j][i] << " ";  // }  // cout << endl;  // }  // cout << endl;  // xsize = matrix.size();  // ysize = 0;  // int numOfAddVar = 0; // after m[i][numOfAddVar] - artifical variables  // int numOfMainVar = 0;  // for(int i = 0; i < matrix.size(); ++i) { // def size  // if(ysize < matrix[i].getLine().size())  // ysize = matrix[i].getLine().size();  // }  // numOfMainVar = ysize;  // for(int i = 0; i < matrix.size(); ++i) { // def num of additional variables  // int bufLim = matrix[i].getLimit();  // if(bufLim != 0) {  // if(bufLim < 0)  // ysize++;  // if(bufLim > 0) {  // ysize += 2;  // }  // numOfAddVar++;  // }  // }  // int addVarCount = 0, addVarArtifCount = 0;  // m = new Frac \*[xsize + 1];  // if(target.getLimit() > 0) {  // for(int i = 0; i < xsize; ++i) {  // m[i] = new Frac[ysize];  // for(int j = 0; j < ysize; ++j) {  // m[i][j] = matrix[i].getLine()[j];  // }  // switch(matrix[i].getLimit()) {  // case 1:  // m[i][numOfMainVar + addVarCount] = -1.0;  // m[i][numOfMainVar + numOfaddVar + addVarArtifCount] = 1.0;  // addVarCount++;  // break;  // case -1:  // m[i][numOfMainVar + addVarCount] = 1.0;  // addVarCount++;  // break;  // }  // }  // m[xsize] = new Frac[ysize];  // for(int j = 0; j < ysize; ++j) {  // m[xsize][j] = target.getLine()[j];  // }  // isMin = false;  // }  // if(target.getLimit() < 0) {  // for(int i = 0; i < xsize; ++i) {  // m[i] = new Frac[ysize];  // for(int j = 0; j < ysize; ++j) {  // m[i][j] = matrix[i].getLine()[j];  // }  // }  // m[xsize] = new Frac[ysize];  // for(int j = 0; j < ysize; ++j) {  // m[xsize][j] = target.getLine()[j];  // }  // isMin = true;  // }  }  void FracMatrixSymplex::rowAdd(int row, Frac \*add)  {  for(int i = 0; i < ysize; ++i) {  m[row][i] += add[i];  }  }  void FracMatrixSymplex::rowMulti(int row, Frac mult)  {  for(int i = 0; i < ysize; ++i)  m[row][i] \*= mult;  }  void FracMatrixSymplex::colMulti(int col, Frac mult)  {  for(int i = 0; i < xsize; ++i)  m[i][col] \*= mult;  }  void FracMatrixSymplex::rowInsert(int row, vector<Frac> add)  {  for(unsigned int i = 0; i < m2.size(); ++i) {  vector<Frac>::iterator it = m2[i].begin() + row;  m2[i].insert(it, add[i]);  }  }  void FracMatrixSymplex::colInsert(int col, vector<Frac> add)  {  if(add.size() != m2[0].size())  add.resize(m2[0].size());  vector<vector<Frac>>::iterator it = m2.begin() + col;  m2.insert(it, add);  }  void FracMatrixSymplex::rowAdd(int row, vector<Frac> add)  {  if(add.size() != m2.size()) {  cout << "Error in rowAdd<" << row << ">" << endl;  return;  }  for(unsigned int i = 0; i < m2.size(); ++i) {  m2[i][row] += add[i];  }  }  void FracMatrixSymplex::rowMulti2(int row, Frac mult)  {  for(unsigned int i = 0; i < m2.size(); ++i) {  m2[i][row] \*= mult;  }  }  void FracMatrixSymplex::colMulti2(int col, Frac mult)  {  for(unsigned int i = 0; i < m2[col].size(); ++i)  m2[col][i] \*= mult;  }  bool FracMatrixSymplex::setNewSnapshot()  {  Frac \*\*snapshot;  snapshot = new Frac \*[xsize];  for(int i = 0; i < xsize; ++i) {  snapshot[i] = new Frac[ysize];  for(int j = 0; j < ysize; ++j) {  snapshot[i][j] = m[i][j];  }  }  bool res;  for(unsigned int i = 0; i < baseSnapshot.size(); ++i) {  res = true;  for(int j = 0; j < xsize; ++j) {  for(int k = 0; k < ysize; ++k) {  if(!(snapshot[j][k] == (baseSnapshot[i].first)[j][k])) {  res = false; // vector[i] is diff  break; // its useless for us now  }  }  }  if(res == true) // if snapshot is equal vector[i]  return false;  }  //snapshot is original  int \*baseSnap = new int[ysize];  for(int i = 0; i < ysize; ++i) {  int b = columnIsBasis(i);  if(b != -1) {  baseSnap[i] = b + 1;  } else {  baseSnap[i] = 0;  }  }  baseSnapshot.push\_back(make\_pair(snapshot, baseSnap));  // cout << "SNAPSHOT: <";  // for(int i = 0; i < ysize; ++i) {  // cout << baseSnapshot[baseSnapshot.size() - 1].second[i] << " ";  // }  // cout << "\b>" << endl;  // for(int i = 0; i <= xsize; ++i) {  // for(int j = 0; j < ysize; ++j) {  // baseSnapshot[baseSnapshot.size() - 1].first[i][j].print();  // cout << " ";  // }  // cout << endl;  // }  return true;  }  int FracMatrixSymplex::columnIsBasis(int col)  {  int basisRow = -1;  for(int i = 0; i < xmainvar; ++i) {  if(m[i][col] == 1) {  if(basisRow == -1) {  basisRow = i;  } else {  return -1;  }  } else {  if(m[i][col] != 0)  return -1;  }  }  return basisRow;  }  void FracMatrixSymplex::symplexMethod()  {  int lRow, lCol;  cout << endl << "START M:" << endl;  printMatrix();  cout << endl << endl;  if(toMin && isFake) {  setZeroInArtif();  }  while(!isOptimal()) {  findLeadForBasisPlan(lRow, lCol);  rectangleMethod(lRow, lCol);  printMatrix();  if(!setNewSnapshot())  break;  }  }  bool FracMatrixSymplex::rebase(int inputColumn, int outputColumn)  {  int basisRow = -1;  // check that output column is basis and that input column wasn't basis before  basisRow = columnIsBasis(outputColumn);  if(basisRow == -1 || columnIsBasis(inputColumn) != -1) // basisRow is the num of row that is not null in output basis  return false;  setBasisInColumn(inputColumn, basisRow);  setNewSnapshot();  return true;  }  void FracMatrixSymplex::printMatrix()  {  cout << "Matrix:" << endl;  for(int i = 0; i < xmainvar; ++i) {  for(int j = 0; j < ysize; ++j) {  m[i][j].print();  cout << " ";  }  cout << endl;  }  cout << "Target function:" << endl;  for(int i = xmainvar; i < xsize; ++i) {  for(int j = 0; j < ysize - 1; ++j) {  if(!toMin)  (m[i][j] \* -1.0).print();  else  m[i][j].print();  cout << "\*x[" << j + 1 << "] + ";  }  cout << "\b\b";  cout << "= ";  if(toMin && isFake)  (m[i][ysize - 1] \* -1.0).print();  else  m[i][ysize - 1].print();  cout << endl;  }  }  void FracMatrixSymplex::setBasis(int \*columns)  {  for(int i = 0; i < xmainvar; ++i) {  setBasisInColumn(columns[i], i);  }  }  bool FracMatrixSymplex::basisWasBefore(int \*columns)  {  bool res;  for(unsigned int i = 0; i < baseSnapshot.size(); ++i) { // for every snapshot...  res = true;  for(int j = 0; j < xmainvar; ++j) { // if input basis was exist, res will be true  if(columns[j] != (baseSnapshot[i].second)[j]) {  res = false;  break;  }  }  if(res)  return res;  }  return false;  }  void FracMatrixSymplex::setZeroInArtif()  {  for(int bi = 0; bi < xmainvar; ++bi) {  int i = ysize - 2 - bi;  int basRow = columnIsBasis(i);  if(basRow == -1) {  cout << "CRITICAL ERROR! CRITICAL ERROR!" << endl;  return;  }  Frac a = m[xsize - 1][i];  for(int j = 0; j < ysize; ++j) {  m[xsize - 1][j] -= m[basRow][j] \* a;  }  }  }  ostream &operator<<(ostream &str, FracMatrixSymplex &outMatrix)  {  for(unsigned int i = 0; i < outMatrix.xsize; ++i) {  for(unsigned int j = 0; j < outMatrix.ysize; ++j) {  str << outMatrix.m[i][j] << " ";  }  str << endl;  }  return str;  }  void FracMatrixSymplex::setBasisInColumn(int column, int row)  {  Frac \*bufRow = new Frac[ysize]; // buffer for making '0'  rowMulti(row, m[row][column].invert()); // making '1' in m[row][column]  for(int i = 0; i < xsize; ++i) { // for every row  if(i != row) {  for(int j = 0; j < ysize; ++j) { // "(i) - (row) \* m[i][column]"  bufRow[j] = m[row][j] \* m[i][column] \* -1.0;  }  rowAdd(i, bufRow);  }  }  delete bufRow;  }  void FracMatrixSymplex::findOptForBasisPlan(int &optimalRow, int &optimalCol)  {  Frac min = m[0][ysize - 1];  for(int i = 1; i < xsize; ++i) {  if(m[i][ysize - 1] < 0.0)  if(m[i][ysize - 1] < min) {  min = m[i][ysize - 1];  optimalRow = i;  }  }  optimalCol = 0;  min = m[optimalRow][0];  for(int j = 1; j < ysize; ++j) {  if(m[optimalRow][j] < min) {  min = m[optimalRow][j];  optimalCol = j;  }  }  if(min > 0.0 && min == 0.0) {  cout << "CRITICAL ERROR! WTF IS GOING ON!" << endl;  }  }  void FracMatrixSymplex::findLeadForBasisPlan(int &leadRow, int &leadCol)  {  leadCol = 0;  Frac min = (m[xsize - 1][0] < 0.0) ? m[xsize - 1][0] : 0.0;  if(toMin && !isFake) {  for(int i = 0; i < ysize - 1; ++i) {  if(m[xsize - 1][i] > 0.0) {  leadCol = i;  break;  }  }  } else {  for(int i = 0; i < ysize - 1; ++i) {  if(m[xsize - 1][i] < min) {  min = m[xsize - 1][i];  leadCol = i;  }  }  }  leadRow = 0;  Frac k = (m[0][ysize - 1] / m[0][leadCol]);  for(int i = 1; i < xmainvar; ++i) {  Frac kbuf = m[i][ysize - 1] / m[i][leadCol];  if(k > 0) {  if(kbuf < k && kbuf > 0.0) {  leadRow = i;  k = kbuf;  }  } else {  if(kbuf > 0.0) {  leadRow = i;  k = kbuf;  }  }  }  cout << "Lead for basis plan:" << leadRow << " " << leadCol << endl;  }  void FracMatrixSymplex::rectangleMethod(int optRow, int optColumn)  {  cout << "Multiplcation: (" << optRow << ") \* ";  m[optRow][optColumn].invert().print();  cout << endl;  rowMulti(optRow, m[optRow][optColumn].invert());  for(int i = 0; i < xsize; ++i) {  if(i != optRow)  for(int j = 0; j < ysize; ++j) {  if(j != optColumn) {  m[i][j] = m[i][j] - (m[i][optColumn] \* m[optRow][j] / m[optRow][optColumn]);  }  }  }  for(int i = 0; i < xsize; ++i)  if(i != optRow)  m[i][optColumn] = 0;  }  bool FracMatrixSymplex::isOptimal()  {  for(int i = 0; i < ysize; ++i) {  if(toMin && isFake) {  if(m[xsize - 1][i] > 0.0) {  return false;  }  } else {  if(m[xsize - 1][i] < 0.0) {  return false;  }  }  }  return true;  } |

## planningmatrix.h (Транспортная задача)

|  |
| --- |
| #ifndef PLANNINGMATRIX\_H  #define PLANNINGMATRIX\_H  #include "volumeandcost.h"  #include "container.h"  #include <vector>  #include <utility>  #include <string>  #include <iostream>  #include <list>  #include "potential.h"  using namespace std;  class PlanningMatrix  {  public:  PlanningMatrix(vector<vector<VolumeAndCost>> m, vector<Container> store, vector<Container> cons);  int planCost();  bool checkLimit();  bool isCloseSystem();  void northWestAngle();  void minCostMethod();  void potentialMethod();  friend std::ostream &operator<<(std::ostream &str, const PlanningMatrix& outM);  private:  void printError(string error); // deprecated  void setConsEmpty();  void setStorFull();  void clearMatrix();  void transfering(int from, int to);  void findMinCost(vector<int> delRows, vector<int> delCols, int &x, int &y);  bool haveElem(vector<int> vec, int a);  void checkMinsInRow(int row, vector<int> &checkRow);  void checkMinsInCol(int col, vector<int> &checkCol);  void setPotential();  bool findMinNegRating(int &x, int &y);  bool potIsDefined();  list <pair<int, int>> DFSH(pair<int, int> start,  pair<int, int> finish,  pair<int, int> previous,  list<pair<int, int> > cycle);  list <pair<int, int>> DFSV(pair<int, int> start,  pair<int, int> finish,  pair<int, int> previous,  list<pair<int, int> > cycle);  list <pair<int, int>> findCycle(int x, int y);  void remakeCycle(int stx, int sty);  list <pair<int, int>> mainCycle;  vector<vector<VolumeAndCost>> matrix;  vector<Container> storage;  vector<Container> consumption;  vector <Potential> potU;  vector <Potential> potV;  int xsize, ysize;  };  #endif // PLANNINGMATRIX\_H |

## planningmatrix.cpp (Транспортная задача)

|  |
| --- |
| #include "planningmatrix.h"  PlanningMatrix::PlanningMatrix(vector<vector<VolumeAndCost> > m, vector<Container> store, vector<Container> cons) :  matrix(m), storage(store), consumption(cons)  {  //resizing vectores  if(matrix.size() < storage.size())  matrix.resize(storage.size());  else  storage.resize(matrix.size());  for(unsigned int i = 0; i < matrix.size(); ++i) {  if(matrix[i].size() < consumption.size())  for(unsigned int i = 0; i < matrix.size(); ++i)  matrix[i].resize(consumption.size());  else  consumption.resize(matrix[i].size());  }  xsize = matrix.size();  ysize = matrix[0].size();  potU.resize(xsize);  potV.resize(ysize);  }  int PlanningMatrix::planCost()  {  int z = 0;  for(int i = 0; i < xsize; ++i) {  for(unsigned int j = 0; j < matrix[i].size(); ++j) {  z += matrix[i][j].fullCost();  }  }  return z;  }  bool PlanningMatrix::checkLimit()  {  int buffer;  for(int i = 0; i < xsize; ++i) {  buffer = 0;  for(int j = 0; j < ysize; ++j) {  buffer += matrix[i][j].getV();  }  if(buffer != storage[i].getSize()) {  return false;  }  }  for(int j = 0; j < ysize; ++j) {  buffer = 0;  for(int i = 0; i < xsize; ++i) {  buffer += matrix[i][j].getV();  }  if(buffer != consumption[j].getSize()) {  return false;  }  }  return true;  }  bool PlanningMatrix::isCloseSystem()  {  int c = 0, s = 0;  for(unsigned int i = 0; i < storage.size(); ++i)  s += storage[i];  for(unsigned int i = 0; i < consumption.size(); ++i)  s += consumption[i];  if(c == s) {  return true;  } else {  printError("system is not close");  return false;  }  }  ostream &operator<<(ostream &str, const PlanningMatrix &outM)  {  str << "[ u\\v ] ";  for(int i = 0; i < outM.ysize; ++i)  str << "[" << outM.potV[i] << "] ";  str << endl;  for(int i = 0; i < outM.xsize + 1; ++i) {  if(i != outM.xsize) {  str << "[" << outM.potU[i] << "] ";  for(unsigned int j = 0; j < outM.matrix[i].size(); ++j) {  str << outM.matrix[i][j] << " ";  }  str << outM.storage[i] << endl;  } else {  str << " ";  for(unsigned int j = 0; j < outM.consumption.size(); ++j) {  str << outM.consumption[j] << " ";  }  }  }  return str;  }  void PlanningMatrix::printError(string error)  {  cout << "Error: " << error << "." << endl;  }  void PlanningMatrix::setConsEmpty()  {  for(unsigned int iterc = 0; iterc < consumption.size(); ++iterc)  consumption[iterc].take(consumption[iterc].getStore());  }  void PlanningMatrix::setStorFull()  {  for(unsigned int iters = 0; iters < storage.size(); ++iters)  storage[iters].put(storage[iters].getSize());  }  void PlanningMatrix::clearMatrix()  {  for(int i = 0; i < xsize; ++i) {  for(unsigned int j = 0; j < matrix[i].size(); ++j) {  matrix[i][j].setWork(false);  }  }  }  void PlanningMatrix::transfering(int from, int to)  {  int buff, buff2;  buff = storage[from].take(storage[from].getSize());  cout << "take " << buff << " from A[" << from << "]..." << endl;  // some from storage put into consumption  buff2 = consumption[to].put(buff);  cout << "put into B[" << to << "],remained " << buff2 << "..." << endl;  matrix[from][to].setV(buff - buff2);  if(buff2 > 0) // we have remainder -> return into storage  storage[from].put(buff2);  }  void PlanningMatrix::findMinCost(vector<int> delRows, vector<int> delCols, int &x, int &y)  {  int min = 100000000;  for(int i = 0; i < xsize; ++i) {  if(haveElem(delRows, i)) // filled before  continue;  for(unsigned int j = 0; j < matrix[i].size(); ++j) {  if(haveElem(delCols, j))  continue;  if(matrix[i][j].getC() < min) {  min = matrix[i][j].getC();  x = i;  y = j;  }  }  }  }  bool PlanningMatrix::haveElem(vector<int> vec, int a)  {  bool ret = false;  for(unsigned int i = 0; i < vec.size(); ++i)  if(vec[i] == a) {  ret = true;  break;  }  return ret;  }  void PlanningMatrix::checkMinsInRow(int row, vector<int> &checkRow)  {  int min = matrix[row][0].getC();  for(unsigned int j = 1; j < matrix[row].size(); ++j) {  if(min > matrix[row][j].getC()) {  checkRow.clear();  checkRow.push\_back(j);  }  if(min == matrix[row][j].getC())  checkRow.push\_back(j);  }  }  void PlanningMatrix::checkMinsInCol(int col, vector<int> &checkCol)  {  int min = matrix[0][col].getC();  for(int i = 1; i < xsize; ++i) {  if(min > matrix[i][col].getC()) {  checkCol.clear();  checkCol.push\_back(i);  }  if(min == matrix[i][col].getC())  checkCol.push\_back(i);  }  }  void PlanningMatrix::setPotential()  {  for(int i = 0; i < xsize; ++i)  potU[i].setSet(false);  for(int i = 0; i < ysize; ++i)  potV[i].setSet(false);  potU[0].setValue(0);  while(!potIsDefined()) {  for(int i = 0; i < xsize; ++i) { // find C[i]  if(potU[i].defined()) {  // cout << "Found u[" << i << "]" << endl;  for(int j = 0; j < ysize; ++j) { // calc all what we can  if(matrix[i][j].isWorking() && !potV[j].defined())  potV[j].setValue(matrix[i][j].getC() - potU[i].getValue()); // C[i][j] - U[i] = V[j]  }  }  }  for(int j = 0; j < ysize; ++j) {  if(potV[j].defined()) { // find V[j]  // cout << "Found v[" << j << "]" << endl;  for(int i = 0; i < xsize; ++i) { // calc all what we can  if(matrix[i][j].isWorking() && !potU[i].defined())  potU[i].setValue(matrix[i][j].getC() - potV[j].getValue()); // C[i][j] - V[j] = U[i]  }  }  }  }  }  bool PlanningMatrix::findMinNegRating(int &x, int &y)  {  int min = 999999;  int buf;  for(int i = 0; i < xsize; ++i) {  for(int j = 0; j < ysize; ++j) {  if(!matrix[i][j].isWorking()) {  buf = matrix[i][j].getC() - potU[i].getValue() - potV[j].getValue();  if(buf < min) {  min = buf;  x = i;  y = j;  }  }  }  }  return min < 0 ? true : false;  }  bool PlanningMatrix::potIsDefined()  {  for(int i = 0; i < xsize; ++i) {  if(!potU[i].defined())  return false;  }  for(int i = 0; i < ysize; ++i) {  if(!potV[i].defined())  return false;  }  return true;  }  list<pair<int, int> > PlanningMatrix::DFSH(pair<int, int> start, pair<int, int> finish,  pair<int, int> previous, list<pair<int, int> > cycle)  {  list <pair<int, int> > path = cycle;  if(!(start.first == finish.first && start.second == finish.second) || previous.first < 0 || previous.second < 0) {  list <pair<int, int> > neighbours;  for (int j = 0; j < ysize; ++j) {  bool inCycle = false;  if (j != start.second && matrix[start.first][j].isWorking()) {  for(list<pair<int, int>>::iterator it = cycle.begin(); it != cycle.end(); ++it) {  if(it->first == start.first && it->second == j) {  inCycle = true;  break;  }  }  if(!inCycle)  neighbours.push\_back(make\_pair(start.first, j));  }  }  neighbours.remove(previous);  for(list<pair<int, int>>::iterator it = neighbours.begin(); it != neighbours.end(); ++it) {  list<pair<int, int>> newPath = cycle;  pair<int, int> neighbour = make\_pair(it->first, it->second);  newPath.push\_back(neighbour);  newPath = DFSV(neighbour, finish, start, newPath);  if(newPath.back().first == finish.first && newPath.back().second == finish.second) {  path = newPath;  break;  }  }  }  return path;  }  list<pair<int, int> > PlanningMatrix::DFSV(pair<int, int> start,  pair<int, int> finish,  pair<int, int> previous,  list<pair<int, int> > cycle)  {  list <pair<int, int> > path = cycle;  if(!(start.first == finish.first && start.second == finish.second) || previous.first < 0 || previous.second < 0) {  list <pair<int, int> > neighbours;  for (int i = 0; i < xsize; ++i) {  if (i != start.first && matrix[i][start.second].isWorking()) {  bool inCycle = false;  for(list<pair<int, int>>::iterator it = cycle.begin(); it != cycle.end(); ++it) {  if(it->second == start.second && it->first == i) {  inCycle = true;  break;  }  }  if(!inCycle)  neighbours.push\_back(make\_pair(i, start.second));  }  }  neighbours.remove(previous);  for(list<pair<int, int>>::iterator it = neighbours.begin(); it != neighbours.end(); ++it) {  list<pair<int, int>> newPath = cycle;  pair<int, int> neighbour = make\_pair(it->first, it->second);  newPath.push\_back(neighbour);  newPath = DFSH(neighbour, finish, start, newPath);  if(newPath.back().first == finish.first && newPath.back().second == finish.second) {  path = newPath;  break;  }  }  }  return path;  }  list<pair<int, int> > PlanningMatrix::findCycle(int x, int y)  {  matrix[x][y].setV(0);  mainCycle.clear();  pair<int, int> start = make\_pair(x, y);  cout << "Start/finish: (" << x << "," << y << ")" << endl;  cout << "Finding..." << endl;  mainCycle = DFSH(start, start, make\_pair(-1, -1), mainCycle);  mainCycle.pop\_back();  return mainCycle;  }  void PlanningMatrix::remakeCycle(int stx, int sty)  {  pair<int, int> min = make\_pair(-1, -1);  int minV = 9999999;  bool parity = false; // 0 is start element (enpty field), so substract from 1, 3, 5 & addition for 2, 4, 6...  for(list<pair<int, int>>::iterator k = mainCycle.begin(); k != mainCycle.end(); k++) {  if(!parity) {  int x = k->first;  int y = k->second;  if(minV > matrix[x][y].getV()) {  min = make\_pair(x, y);  minV = matrix[x][y].getV();  }  }  parity = !parity;  }  parity = false;  for(list<pair<int, int>>::iterator k = mainCycle.begin(); k != mainCycle.end(); k++) {  if(!parity) {  int x = k->first;  int y = k->second;  int sub = matrix[x][y].getV() - minV;  (sub > 0) ? matrix[x][y].setV(sub) : matrix[x][y].setWork(false);  }  parity = !parity;  }  parity = false;  for(list<pair<int, int>>::iterator k = mainCycle.begin(); k != mainCycle.end(); k++) {  if(parity) {  int x = k->first;  int y = k->second;  int add = matrix[x][y].getV() + minV;  matrix[x][y].setV(add);  }  parity = !parity;  }  matrix[stx][sty].setV(minV);  }  void PlanningMatrix::northWestAngle()  {  setConsEmpty();  setStorFull();  clearMatrix();  int iters = 0;  for(unsigned int iterc = 0; iterc < consumption.size(); ++iterc) { // filling consumption  cout << "Filling B[" << iterc<< "] consumption" << endl;  while(!consumption[iterc].isFull()) {  transfering(iters, iterc);  if(storage[iters].isEmpty()) {  iters++;  }  cout << \*this << endl << endl;  }  }  cout << endl << "Z = " << planCost() << endl;  }  void PlanningMatrix::minCostMethod()  {  vector<int> deletedRow;  vector<int> deletedCol;  setConsEmpty();  setStorFull();  clearMatrix();  cout << \*this << endl;  int x, y;  while(!checkLimit()) {  cout << "Limit checked..." << endl;  findMinCost(deletedRow, deletedCol, x, y);  cout << "Transfer from " << x  << " to " << y << endl;  transfering(x, y);  if(storage[x].isEmpty())  deletedRow.push\_back(x);  if(consumption[y].isFull() && !storage[x].isEmpty())  deletedCol.push\_back(y);  cout << \*this << endl << endl;  }  cout << endl << "Z = " << planCost() << endl;  }  void PlanningMatrix::potentialMethod()  {  int x, y;  setPotential();  while(findMinNegRating(x, y)) {  cout << \*this << endl << endl;  cout << "Setting potential..." << endl;  setPotential();  mainCycle = findCycle(x, y);  cout << "Cycle:" << endl << "(" << x << "," << y << ")" << endl;  for(list<pair<int, int>>::iterator k = mainCycle.begin(); k != mainCycle.end(); k++) {  cout << "(" << k->first << "," << k->second << ")" << endl;  }  cout << "(" << x << "," << y << ")" << endl;  remakeCycle(x, y);  }  cout << "RESULT:" << endl << \*this << endl << endl;  } |

# Заключение

В ходе выполнения курсовой работы были рассмотрены следующие алгоритмы и вычислительные методы оптимизации решения задач линейного программирования:

* Метод Гаусса-Жордана
* Симплекс-метод
* Метод искусственного базиса
* Метод минимальной стоимости
* Метод потенциалов