МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Факультет информационных технологий и программирования

Кафедра информационных систем

Лабораторная работа №3

Линейное программирование

Выполнил студент группы №3307:   
Егоров Николай Валерьевич

Проверила:

Сайферт Екатерина

САнкт-Петербург  
 2018

***Общей (стандартной) задачей линейного программирования*** называется задача нахождения минимума линейной целевой функции (линейной формы) вида:

f ( x ) = ∑ j = 1 n c j x j = c 1 x 1 + c 2 x 2 + … + c n x n . {\displaystyle f(x)=\sum \_{j=1}^{n}c\_{j}x\_{j}=c\_{1}x\_{1}+c\_{2}x\_{2}+\ldots +c\_{n}x\_{n}.}

Задача, в которой фигурируют ограничения в форме неравенств, называется ***основной задачей линейного программирования (ОЗЛП)***

∑ j = 1 n a i j x j ⩾ b i , ( i = 1 , 2 , … , m ) {\displaystyle \sum \_{j=1}^{n}a\_{ij}x\_{j}\geqslant b\_{i},\quad (i=1,\;2,\;\ldots ,\;m)} 

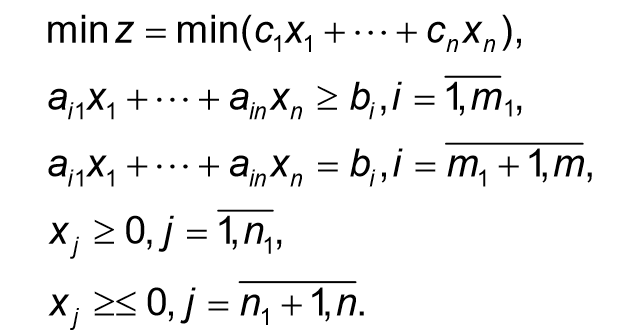
x j ⩾ 0. ( j = 1 , 2 , … , n ) {\displaystyle x\_{j}\geqslant 0.\quad (j=1,\;2,\;\ldots ,\;n)}

Задача линейного программирования будет иметь ***канонический вид***, если в основной задаче вместо первой системы неравенств имеет место система уравнений с ограничениями в форме равенства:

∑ j = 1 n a i j x j = b i . ( i = 1 , 2 , … , m ) {\displaystyle \sum \_{j=1}^{n}a\_{ij}x\_{j}=b\_{i}.\quad (i=1,\;2,\;\ldots ,\;m)}

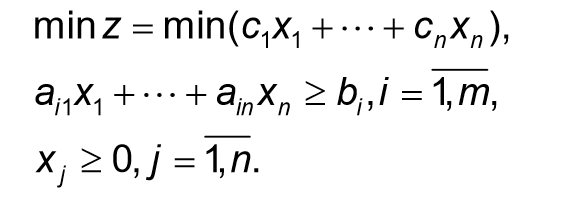


Аналогично можно написать общую постановку для задачи минимизации:



# Симплекс-метод

Стандартная форма задачи ЛП минимизации – это задача нахождения минимума целевой функции при ограничениях типа нестрогого неравенства (больше или равно) и при наличии ограничений на знак для всех переменных:

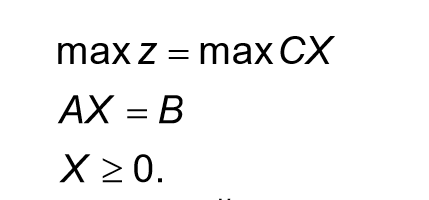


В матричном и векторном виде эта задача может быть записана

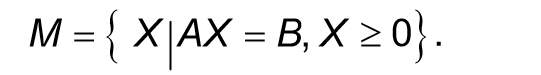
так:



Рассмотрим каноническую задачу ЛП максимизации в матричной форме:



Множество допустимых решений задачи является замкнутым выпуклым многогранником и может быть найдено как множество решений системы линейных уравнений:

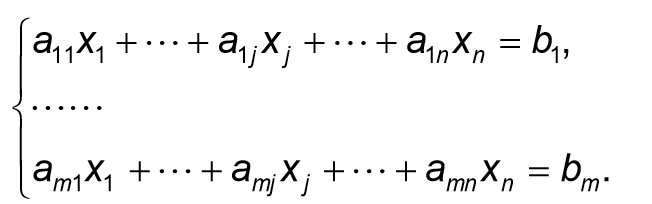


Напомним, что система линейных уравнений 

совместна, если выполнено условие:

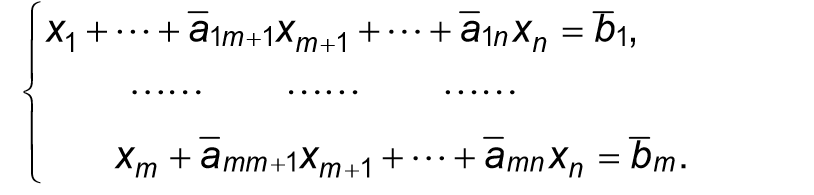


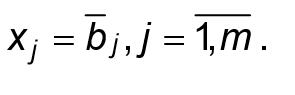
Это условие означает, что ранг матрицы системы равен числу уравнений, и все уравнения системы независимы (ни одно из уравнений не является следствием остальных). Предположим, что система линейных уравнений AX= B совместна, запишем ее в виде:



Базисным решением СЛУ, зависящим от множества индексов S={1,… ,m} будем называть решение СЛУ, которое находится по указанным ниже правилам.

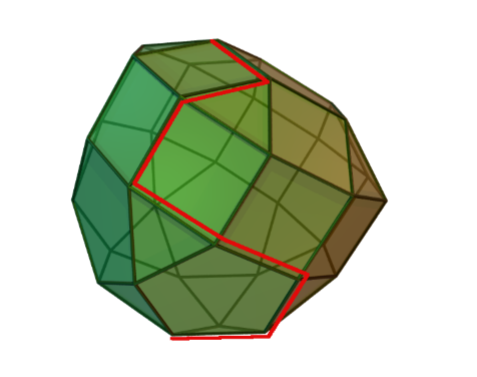
* Для нахождения базисного решения, зависящего от множества индексов S={1,… ,m} (количество индексов в множестве S равно m – числу уравнений), надо привести данную систему, используя метод Гаусса к диагональной форме по переменным x1..xm , которые называются базисными переменными данного решения. Получим



* Полагая переменные, не вошедшие в диагональную форму (небазисные переменные) равными нулю ( ), получаем значения для базисных переменных: 

Решение общей распределительной задачи выполняется в два этапа:

1. Находят любое решение (как правило, неоптимальное), удовлетворяющее ограничениям, или убеждаются, что решения не существует. Этот этап называется определением опорного плана (базиса).
2. Производится последовательное улучшение данного плана до получения оптимального. В некоторых задачах опорный план определяется легко, в противном случае используют специальные методы получения опорного плана (метод искусственного базиса).

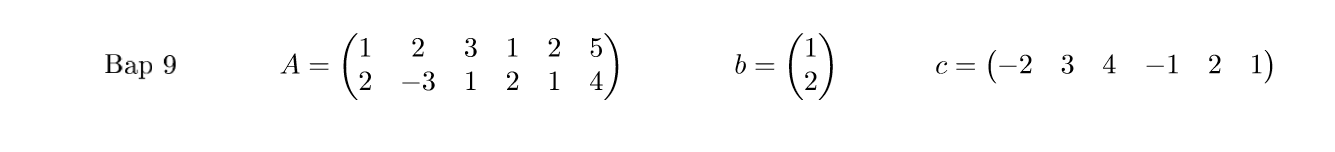


Выпуклая фигура, соответствующая области решений, называется симплексом. Поиск оптимального решения осуществляется с помощью переходов по её ребрам.

Эффективность:

Число итераций при решении задач ЛП в стандартной форме с M ограничениями и N переменными заключено между M и 3M. Среднее число итераций 2M. Верхняя граница числа итераций равна 2M +N.

Требуемое машинное время пропорционально . Число ограничений больше влияет на вычислительную эффективность, чем число переменных, поэтому при формулировке задач ЛП нужно стремиться к уменьшению числа ограничений пусть даже путём роста числа переменных.

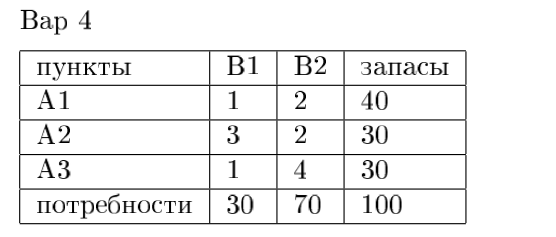


Результат:

-2.0

[ -1.0 ; 0.0 ; 0.0 ; 0.0 ; 0.0 ; 0.0 ; ]

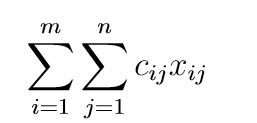
# Транспортная задача



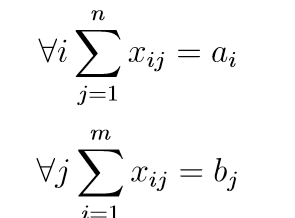
Задача о перевозке товара. m поставщиков товара поставляют A = (a1,...,am) единиц товара, и n потребителей потребляющих B = (b1,...,bn) единиц товара. Причем . Задана стоимость перевозки cij, нужно вычислить сколько товара каждый поставщик должен поставлять каждому потребителю для минимизации общей стоимости перевозок.

Или, в терминах линейного программирования:

Минимизировать



При



Сперва найдем опорный план каким-либо методом, например, симплекс-методом или методом северно-западного угла. Далее улучшаем методом падающего камня:

1. Находим пустые клетки и проводим для них расчет: находим кратчайший путь от пустой до рассматриваемой.
2. Обход выполняется квадратиком, те движение только вертикально и горизонтально, клетки на одной линии.
3. Считаем для каждой пустой клетки ее предположительную стоимость.
4. Если такой среди таких стоимостей есть тот, что отрицателен, значит, надо внести изменение в исходный план и убрать ненужные составляющее.

Результат:

Optimal solution

Supplier number 1 route:

20 - 5

Supplier number 2 route:

- 30 5

Total costs: 180.0

Simplex.java

import org.jscience.mathematics.number.Rational;  
import java.util.\*;  
  
import static java.lang.Double.*POSITIVE\_INFINITY*;  
  
public class Simplex {  
  
 // returns max/min c\*x such that A\*x <= b, x >= 0  
 public static Rational simplex(Rational[][] A, Rational[] b, Rational[] c, Rational[] x, boolean flag\_min) {  
 int m = A.length;  
 int n = A[0].length + 1;  
 int[] index = new int[n + m];  
 for (int i = 0; i < n + m; i++) {  
 index[i] = i;  
 }  
 Rational[][] a = new Rational[m + 2][n + 1];  
 for (Rational[] a1 : a) {  
 Arrays.*fill*(a1, Rational.*ZERO*);  
 }  
 int L = m;  
 for (int i = 0; i < m; i++) {  
 for (int j = 0; j < n - 1; j++) {  
 a[i][j] = A[i][j].opposite();  
 }  
 a[i][n - 1] = Rational.*ONE*;  
 a[i][n] = b[i] ;  
 if (a[L][n].compareTo(a[i][n]) > 0) {  
 L = i;  
 }  
 }  
 for (int j = 0; j < n - 1; j++) {  
 if(flag\_min) {  
 a[m][j] = c[j].opposite();  
 } else {  
 a[m][j] = c[j];  
 }  
 }  
 a[m + 1][n - 1] = Rational.*ONE*.opposite();  
 for (int E = n - 1;;) {  
 if (L < m) {  
 int t = index[E];  
 index[E] = index[L + n];  
 index[L + n] = t;  
 a[L][E] = a[L][E].inverse();  
 for (int j = 0; j <= n; j++) {  
 if (j != E) {  
 a[L][j] = a[L][j].times(a[L][E].opposite());  
 }  
 }  
 for (int i = 0; i <= m + 1; i++) {  
 if (i != L) {  
 for (int j = 0; j <= n; j++) {  
 if (j != E) {  
 a[i][j] = a[i][j].plus(a[L][j].times(a[i][E]));  
 }  
 }  
 a[i][E] = a[i][E].times(a[L][E]);  
 }  
 }  
 }  
 E = -1;  
 for (int j = 0; j < n; j++) {  
 if (E < 0 || index[E] > index[j]) {  
 if (a[m + 1][j].isPositive() || a[m + 1][j].isZero() && a[m][j].isPositive()) {  
 E = j;  
 }  
 }  
 }  
 if (E < 0) {  
 break;  
 }  
 L = -1;  
 for (int i = 0; i < m; i++) {  
 if (a[i][E].isNegative()) {  
 Rational d;  
 if (L < 0 || (d = a[L][n].divide(a[L][E]).minus(a[i][n].divide(a[i][E]))).isNegative()|| d.isZero()  
 && index[L + n] > index[i + n]) {  
 L = i;  
 }  
 }  
 }  
 // if (L > 0) {  
 // return Rational.valueOf(322,1L);  
 // }  
 }  
 if (a[m + 1][n].isNegative()) {  
 return null;  
 }  
 if (x != null) {  
 Arrays.*fill*(x, Rational.*ZERO*);  
 for (int i = 0; i < m; i++)  
 if (index[n + i] < n - 1) {  
 if (flag\_min) {  
 x[index[n + i]] = a[i][n].opposite();  
 }  
 else x[index[n + i]] = a[i][n];  
  
 }  
  
 }  
 return flag\_min ? a[m][n].opposite() : a[m][n] ;  
 }  
  
 // Usage example  
 public static void main(String[] args) {  
 long[][] a = { { 4, -1 }, { 2, 1 }, { -5, 2 } };  
 long[] b = { 8, 10, 2 };  
 long[] c = { 1, 1 };  
 Rational[] x = new Rational[c.length];  
  
 Rational res ;  
  
 a = new long[][] { { 3, 4, -3 }, { 5, -4, -3 }, { 7, 4, 11 } };  
 b = new long[] { 23, 10, 30 };  
 c = new long[] { -1, 1, 2 };  
 x = new Rational[c.length];  
  
 res = *simplex*(*cnv*(a), *cnv*(b), *cnv*(c), x,false);  
 System.*out*.println(Rational.*valueOf*(57,8) .equals(res));  
 System.*out*.println(res);  
 System.*out*.println(Arrays.*toString*(x));  
  
  
  
 a = new long[][] { {1,2,3,1,2,5}, {2,-3,1,2,1,4} };  
 b = new long[] { 1,2 };  
 c = new long[] { -2, 3, 4,-1, 2, 3 };  
 x = new Rational[c.length];  
  
  
 res = *simplex*(*cnv*(a), *cnv*(b), *cnv*(c), x,true);  
 System.*out*.println(res.doubleValue() );  
 System.*out*.println(*rationalToString*(x));  
  
  
 a = new long[][] { { 1, 3,1 }, { 2, 2, 4 }};  
 // a = new long[][] { { 1, 2 }, { 3,2 }, {1,4}};  
 b = new long[] { 30, 70 };  
 c = new long[] { 40, 30, 30 };  
 x = new Rational[c.length];  
  
 res = *simplex*(*cnv*(a), *cnv*(b), *cnv*(c), x,true);  
 System.*out*.println(res.doubleValue());  
  
  
  
 }  
  
 private static Rational[] cnv(long[] a) {  
 Rational[] res = new Rational[a.length];  
 for (int i = 0; i < a.length; i++) {  
 res[i] = Rational.*valueOf*(a[i],1L);  
 }  
 return res;  
 }  
  
  
 private static Rational[][] cnv(long[][] a) {  
 Rational[][] res = new Rational[a.length][];  
 for (int i = 0; i < a.length; i++) {  
 res[i] = *cnv*(a[i]);  
 }  
 return res;  
 }  
  
  
 private static String rationalToString(Rational[] a){  
 String res="[ ";  
 for (Rational i : a){  
 res+=i.doubleValue()+" ; ";  
  
 }  
 res+="]";  
 return res;  
 }  
}

Transpartation\_problem.java

import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.util.\*;  
import static java.util.Arrays.*stream*;  
import static java.util.stream.Collectors.*toCollection*;  
  
  
  
  
public class Transpartation\_problem {  
  
 private static int[] *demand*;  
 private static int[] *supply*;  
 private static double[][] *costs*;  
 private static Shipment[][] *matrix*;  
  
 public void init(String filename) throws Exception {  
  
 try (Scanner sc = new Scanner(new File(filename))) {  
 int numSources = sc.nextInt();  
 int numDestinations = sc.nextInt();  
  
 List<Integer> src = new ArrayList<>();  
 List<Integer> dst = new ArrayList<>();  
  
 for (int i = 0; i < numSources; i++)  
 src.add(sc.nextInt());  
  
 for (int i = 0; i < numDestinations; i++)  
 dst.add(sc.nextInt());  
  
 // fix imbalance  
 int totalSrc = src.stream().mapToInt(i -> i).sum();  
 int totalDst = dst.stream().mapToInt(i -> i).sum();  
 if (totalSrc > totalDst)  
 dst.add(totalSrc - totalDst);  
 else if (totalDst > totalSrc)  
 src.add(totalDst - totalSrc);  
  
 *supply* = src.stream().mapToInt(i -> i).toArray();  
 *demand* = dst.stream().mapToInt(i -> i).toArray();  
  
 *costs* = new double[*supply*.length][*demand*.length];  
 *matrix* = new Shipment[*supply*.length][*demand*.length];  
  
 for (int i = 0; i < numSources; i++)  
 for (int j = 0; j < numDestinations; j++)  
 *costs*[i][j] = sc.nextDouble();  
 }  
 }  
  
  
 public void northWestCornerRule() {  
  
 for (int r = 0, northwest = 0; r < *supply*.length; r++)  
 for (int c = northwest; c < *demand*.length; c++) {  
  
 int quantity = Math.*min*(*supply*[r], *demand*[c]);  
 if (quantity > 0) {  
 *matrix*[r][c] = new Shipment(quantity, *costs*[r][c], r, c);  
  
 *supply*[r] -= quantity;  
 *demand*[c] -= quantity;  
  
 if (*supply*[r] == 0) {  
 northwest = c;  
 break;  
 }  
 }  
 }  
 }  
  
  
 public void steppingStone() {  
 double maxReduction = 0;  
 Shipment[] move = null;  
 Shipment leaving = null;  
  
 fixDegenerateCase();  
  
 for (int r = 0; r < *supply*.length; r++) {  
 for (int c = 0; c < *demand*.length; c++) {  
  
 if (*matrix*[r][c] != null)  
 continue;  
  
 Shipment trial = new Shipment(0, *costs*[r][c], r, c);  
 Shipment[] path = getPath(trial);  
  
 double reduction = 0;  
 double lowestQuantity = Integer.*MAX\_VALUE*;  
 Shipment leavingCandidate = null;  
  
 boolean plus = true;  
 for (Shipment s : path) {  
 if (plus) {  
 reduction += s.costPerUnit;  
 } else {  
 reduction -= s.costPerUnit;  
 if (s.quantity < lowestQuantity) {  
 leavingCandidate = s;  
 lowestQuantity = s.quantity;  
 }  
 }  
 plus = !plus;  
 }  
 if (reduction < maxReduction) {  
 move = path;  
 leaving = leavingCandidate;  
 maxReduction = reduction;  
 }  
 }  
 }  
  
 if (move != null) {  
 double q = leaving.quantity;  
 boolean plus = true;  
 for (Shipment s : move) {  
 s.quantity += plus ? q : -q;  
 *matrix*[s.r][s.c] = s.quantity == 0 ? null : s;  
 plus = !plus;  
 }  
 steppingStone();  
 }  
 }  
  
 public LinkedList<Shipment> matrixToList() {  
 return *stream*(*matrix*)  
 .flatMap(row -> *stream*(row))  
 .filter(s -> s != null)  
 .collect(*toCollection*(LinkedList::new));  
 }  
  
  
 public Shipment[] getNeighbors(Shipment s, LinkedList<Shipment> lst) {  
 Shipment[] nbrs = new Shipment[2];  
 for (Shipment o : lst) {  
 if (o != s) {  
 if (o.r == s.r && nbrs[0] == null)  
 nbrs[0] = o;  
 else if (o.c == s.c && nbrs[1] == null)  
 nbrs[1] = o;  
 if (nbrs[0] != null && nbrs[1] != null)  
 break;  
 }  
 }  
 return nbrs;  
 }  
  
 public void fixDegenerateCase() {  
 final double eps = Double.*MIN\_VALUE*;  
  
 if (*supply*.length + *demand*.length - 1 != matrixToList().size()) {  
  
 for (int r = 0; r < *supply*.length; r++)  
 for (int c = 0; c < *demand*.length; c++) {  
 if (*matrix*[r][c] == null) {  
 Shipment dummy = new Shipment(eps, *costs*[r][c], r, c);  
 if (getPath(dummy).length == 0) {  
 *matrix*[r][c] = dummy;  
 return;  
 }  
 }  
 }  
 }  
 }  
  
  
  
  
  
  
  
  
  
  
  
  
 public Shipment[] getPath(Shipment s) {  
 LinkedList<Shipment> path = matrixToList();  
 path.addFirst(s);  
  
 // remove (and keep removing) elements that do not have a  
 // vertical AND horizontal neighbor  
 while (path.removeIf(e -> {  
 Shipment[] nbrs = getNeighbors(e, path);  
 return nbrs[0] == null || nbrs[1] == null;  
 }));  
  
 // place the remaining elements in the correct plus-minus order  
 Shipment[] stones = path.toArray(new Shipment[path.size()]);  
 Shipment prev = s;  
 for (int i = 0; i < stones.length; i++) {  
 stones[i] = prev;  
 prev = getNeighbors(prev, path)[i % 2];  
 }  
 return stones;  
 }  
  
  
  
 static void printPath(String filename) throws IOException {  
  
 double totalCosts = 0;  
 try( FileWriter writer = new FileWriter(filename, false)) {  
 writer.write("Optimal solution \n");  
 for (int r = 0; r < *supply*.length; r++) {  
 writer.write("Supplier number "+ (r+1) + " route:\n");  
 for (int c = 0; c < *demand*.length; c++) {  
  
 Shipment s = *matrix*[r][c];  
 if (s != null && s.r == r && s.c == c) {  
 writer.write((int) s.quantity+ " ");  
 totalCosts += (s.quantity \* s.costPerUnit);  
 } else  
 writer.write("- ");  
 }  
 writer.write("\n");  
 }  
  
 writer.write("\nTotal costs: "+ totalCosts) ;  
  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.printf("%nTotal costs: %s%n%n", totalCosts);  
 }  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
}