# Отчёт к лабораторной работе №1

Выполнил: Фроловский А. ИУ7-82

Содержательная постановка

В распоряжении работодателя имеется n работ (заданий) и n исполнителей. Затраты на выполнение i-м работником j-й работы составляют условных единиц. Каждый работник может выполнить только 1 задание, при этом все задания должны быть выполнены. Необходимо составить план распределения работ между исполнителями, чтобы общие затраты были минимальны.

Математическая постановка

где:

– матрица стоимостей;

- матрица назначений, в которой

Описание венгерского метода

Алгоритм состоит из следующих действий:

1. Подготовительный этап венгерского метода:
   1. В каждом столбце матрицы находим наименьший элемент и вычитаем его из этого столбца
   2. В каждой строке полученной матрицы находим наименьший элемент и вычитаем его из соответствующей строки; полученную матрицу обозначим
   3. Строим первоначальную систему независимых нулей:

Просматриваем матрицу по столбцам сверху вниз (столбцы выбираем слева направо) в поисках 0. Если в одной строке с найденным нулем нет 0\*, то отмечаем его \*. Обозначим k – число нулей в полученной СНН (система независимых нулей).

1. Основной этап венгерского метода:
   1. Если k = n, то переходим к пункту 2.11
   2. Выделяем ‘+’ столбцы, содержащие 0\*
   3. Если среди невыделенных элементов есть 0, переходим к пункту 2.5
   4. Среди невыделенных элементов находим наименьший элемент h>0, вычитаем его из невыделенных строк, добавляем его к выделенным столбцам; среди полученных невыделенных элементов выбираем 0
   5. Отмечаем его “ ‘ ”
   6. Если в одной строке с этим 0’ нет 0\*, переходим к 2.8
   7. Выделяем строку с этим 0’, снимаем выделение со столбца, в котором стоит 0\* (из этой же строки); переходим к пункту 2.3
   8. Строим непродолжаемую L-цепочку:
   9. В L-цепочке заменяем 0\* на 0, 0’ заменяем на 0\*
   10. Снимаем все выделения со строк и столбцов; переходим к пункту 2.1
   11. Записывает оптимальное решение:

В позициях матрицы X, соответствующих 0\* в , ставим 1, в остальные ставим 0

* 1. Печать ,

Текст программы

public class Matrix

{

public Matrix(int degree)

{

values = new double[degree][degree];

}

public Matrix(double[][] sourceMatrix)

{

int n = sourceMatrix.length;

values = new double[n][n];

for (int i = 0; i < n; i++)

System.arraycopy(sourceMatrix[i], 0, values[i], 0, n);

}

public Matrix(Matrix sourceMatrix)

{

int n = sourceMatrix.getDegree();

values = new double[n][n];

for (int i = 0; i < n; i++)

System.arraycopy(sourceMatrix.values[i], 0, values[i], 0, n);

}

public final void setVal(int i, int j, double val)

{

values[i][j] = val;

}

public final double getVal(int i , int j)

{

return values[i][j];

}

public final int getDegree()

{

return values.length;

}

public void print()

{

int n = values.length;

for (int i = 0; i < n; i++)

printRow(i);

System.out.printf("\n");

}

private void printRow(int indexRow)

{

int n = values.length;

System.out.printf("\n");

for (int j = 0; j < n; j++)

System.out.printf("%4.1f ", values[indexRow][j]);

}

private double[][] values;

}

public class ExtendMatrix extends Matrix

{

public ExtendMatrix(int degree)

{

super(degree);

markColumns = new byte[degree];

markRows = new byte[degree];

marks = new char[degree][degree];

}

public ExtendMatrix(double[][] sourceMatrix)

{

super(sourceMatrix);

int degree = sourceMatrix.length;

markColumns = new byte[degree];

markRows = new byte[degree];

marks = new char[degree][degree];

}

public ExtendMatrix(Matrix sourceMatrix)

{

super(sourceMatrix);

int degree = sourceMatrix.getDegree();

markColumns = new byte[degree];

markRows = new byte[degree];

marks = new char[degree][degree];

}

public void delMarkFromAllElements(char mark)

{

int n = this.getDegree();

for (int i = 0; i < n; i++)

delMarkFromAllElementsRow(i, mark);

}

private void delMarkFromAllElementsRow(int indexRow, char mark)

{

int n = this.getDegree();

for (int j = 0; j < n; j++)

if (hasElemMark(indexRow, j, mark))

setMarkToElem(indexRow, j, (char) 0);

}

public void delAllMarksFromRowsAndColumns()

{

int n = super.getDegree();

for (int i = 0; i < n; i++)

{

markColumns[i] = 0;

markRows[i] = 0;

}

}

public void delMarkFromElem(int indexRow, int indexColumn)

{

marks[indexRow][indexColumn] = (char) 0;

}

public void setMarkToElem(int indexRow, int indexColumn, char mark)

{

marks[indexRow][indexColumn] = mark;

}

public void setMarkToColumn(int indexColumn)

{

markColumns[indexColumn] = 1;

}

public void resetMarkFromColumn(int indexColumn)

{

markColumns[indexColumn] = 0;

}

public void setMarkToRow(int indexRow)

{

markRows[indexRow] = 1;

}

public void resetMarkFromRow(int indexRow)

{

markRows[indexRow] = 0;

}

public boolean hasElemMark(int indexRow, int indexColumn, char mark)

{

if (marks[indexRow][indexColumn] == mark)

return true;

return false;

}

public boolean hasRowMark(int indexRow)

{

if (markRows[indexRow] == 1)

return true;

return false;

}

public boolean hasColumnMark(int indexColumn)

{

if (markColumns[indexColumn] == 1)

return true;

return false;

}

public void print()

{

System.out.printf("\n");

printPlusOverMarkedColumns();

printMatrix();

System.out.printf("\n");

}

private void printPlusOverMarkedColumns()

{

int n = this.getDegree();

for (int j = 0; j < n; j++)

if (!hasColumnMark(j))

System.out.printf(" ");

else

System.out.printf(" + ");

}

private void printMatrix()

{

int n = this.getDegree();

for (int i = 0; i < n; i++)

printRow(i);

}

private void printRow(int indexRow)

{

System.out.printf("\n");

printAllElementsRowWithMarks(indexRow);

printPlusNearMarkedRow(indexRow);

}

private void printAllElementsRowWithMarks(int indexRow)

{

int n = this.getDegree();

for (int j = 0; j < n; j++)

{

double elem = this.getVal(indexRow, j);

printElement(elem);

printElementMark(indexRow, j);

}

}

private void printElement(double val)

{

System.out.printf("%4.1f", val);

}

private void printElementMark(int indexRow, int indexColumn)

{

if (hasElemMark(indexRow, indexColumn, '\*'))

System.out.printf("\*");

else if (hasElemMark(indexRow, indexColumn, '\''))

System.out.printf("\'");

else

System.out.printf(" ");

}

private void printPlusNearMarkedRow(int indexRow)

{

if (hasRowMark(indexRow))

System.out.printf(" + ");

}

private byte[] markColumns;

private byte[] markRows;

private char[][] marks;

}

public class HungarianMethod

{

public Matrix solveTaskMaximizationWithPrintOutput(Matrix initMatrix)

{

Matrix modifMatrix = transformMaximizationTaskToMinimizationTask(initMatrix);

return solveTaskMinimizationWithPrintOutput(modifMatrix);

}

public Matrix solveTaskMaximization(Matrix initMatrix)

{

Matrix modifMatrix = transformMaximizationTaskToMinimizationTask(initMatrix);

return solveTaskMinimization(modifMatrix);

}

public Matrix solveTaskMinimizationWithPrintOutput(Matrix initMatrix)

{

establishPrintOutput();

return solveStandartTask(initMatrix);

}

public Matrix solveTaskMinimization(Matrix initMatrix)

{

establishNotPrintOutput();

return solveStandartTask(initMatrix);

}

private Matrix solveStandartTask(Matrix sourceMatrix)

{

Matrix initMatrix = new ExtendMatrix(sourceMatrix);

printOutput(initMatrix);

ExtendMatrix prepMatrix = prepStage(initMatrix);

printOutput(prepMatrix);

return mainStage(prepMatrix);

}

private ExtendMatrix prepStage(Matrix matrix)

{

Matrix prepMatrix = createZeroElementsInMatrix(matrix);

return buildInitIndependentZerosSystem(prepMatrix);

}

private Matrix createZeroElementsInMatrix(Matrix matrix)

{

Matrix prepMatrix = createZeroElementsInMatrixColumns(matrix);

return createZeroElementsInMatrixRows(prepMatrix);

}

private Matrix createZeroElementsInMatrixColumns(Matrix matrix)

{

int n = matrix.getDegree();

Matrix resMatrix = new Matrix(matrix);

for (int j = 0; j < n; j++)

{

double minElem = minElemColumn(resMatrix, j);

resMatrix = subNumberFromColumnElements(resMatrix, j, minElem);

}

return resMatrix;

}

private Matrix subNumberFromColumnElements(Matrix matrix, int indexColumn, double num)

{

int n = matrix.getDegree();

Matrix resMatrix = new Matrix(matrix);

for (int i = 0; i < n; i++)

{

double elem = resMatrix.getVal(i, indexColumn) - num;

resMatrix.setVal(i, indexColumn, elem);

}

return resMatrix;

}

private Matrix createZeroElementsInMatrixRows(Matrix matrix)

{

int n = matrix.getDegree();

Matrix resMatrix = new Matrix(matrix);

for (int i = 0; i < n; i++)

{

double minElem = minElemRow(resMatrix, i);

resMatrix = subNumberFromRowElements(resMatrix, i, minElem);

}

return resMatrix;

}

private Matrix subNumberFromRowElements(Matrix matrix, int indexRow, double num)

{

int n = matrix.getDegree();

Matrix resMatrix = new Matrix(matrix);

for (int j = 0; j < n; j++)

{

double elem = resMatrix.getVal(indexRow, j) - num;

resMatrix.setVal(indexRow, j, elem);

}

return resMatrix;

}

private ExtendMatrix buildInitIndependentZerosSystem(Matrix matrix)

{

int n = matrix.getDegree();

ExtendMatrix resMatrix = new ExtendMatrix(matrix);

for (int j = 0; j < n; j++)

resMatrix = findAndMarkAcceptableColumnZeroElements(resMatrix, j);

return resMatrix;

}

private ExtendMatrix findAndMarkAcceptableColumnZeroElements(ExtendMatrix matrix, int indexColumn)

{

int n = matrix.getDegree();

ExtendMatrix resMatrix = matrix;

for (int i = 0; i < n; i++)

{

if (isFitElement(resMatrix, i, indexColumn))

{

resMatrix.setMarkToElem(i, indexColumn, '\*');

break;

}

}

return resMatrix;

}

private boolean isFitElement(ExtendMatrix matrix, int indexRow, int indexColumn)

{

double elem = matrix.getVal(indexRow, indexColumn);

return isElementEqualsZero(elem) && !isExistMarkedStarZeroInRow(matrix, indexRow);

}

private boolean isElementEqualsZero(double elem)

{

return Math.abs(elem) < EPS;

}

private boolean isExistMarkedStarZeroInRow(ExtendMatrix matrix, int indexRow)

{

int n = matrix.getDegree();

for (int j = 0; j < n; j++)

if (matrix.hasElemMark(indexRow, j, '\*'))

return true;

return false;

}

private Matrix mainStage(ExtendMatrix matrix)

{

ExtendMatrix workMatrix = matrix;

int n = matrix.getDegree();

int k = computeCountZeroInIZS(matrix);

while (k < n)

{

workMatrix = executeMainActionsAlgorithm(workMatrix);

k = computeCountZeroInIZS(workMatrix);

}

return writeOptimalSolution(workMatrix);

}

private int computeCountZeroInIZS(ExtendMatrix matrix)

{

int count = 0;

int n = matrix.getDegree();

for (int j = 0; j < n; j++)

count += computeCountZeroInColumnIZS(matrix, j);

return count;

}

private int computeCountZeroInColumnIZS(ExtendMatrix matrix, int indexColumn)

{

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

if (matrix.hasElemMark(i, indexColumn, '\*'))

return 1;

return 0;

}

private ExtendMatrix executeMainActionsAlgorithm(ExtendMatrix matrix)

{

ExtendMatrix workMatrix = matrix;

workMatrix = markColumnsWithMarkedStarZero(workMatrix);

printOutput(workMatrix);

return changeIZS(workMatrix);

}

private ExtendMatrix markColumnsWithMarkedStarZero(ExtendMatrix matrix)

{

int n = matrix.getDegree();

ExtendMatrix resMatrix = matrix;

for (int j = 0; j < n; j++)

resMatrix = checkAndMarkColumnWithMarkedStarElement(resMatrix, j);

return resMatrix;

}

private ExtendMatrix checkAndMarkColumnWithMarkedStarElement(ExtendMatrix matrix, int indexColumn)

{

ExtendMatrix resMatrix = matrix;

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

if (matrix.hasElemMark(i, indexColumn, '\*'))

{

resMatrix.setMarkToColumn(indexColumn);

break;

}

return resMatrix;

}

private ExtendMatrix changeIZS(ExtendMatrix matrix)

{

ExtendMatrix workMatrix = matrix;

boolean flgMarkedStarZeroInRow = true;

while (flgMarkedStarZeroInRow)

{

if (!checkExistUnmarkedZero(workMatrix))

workMatrix = improveMatrix(workMatrix);

int unmarkZeroJ = findColumnWithUnmarkElem(workMatrix);

int unmarkZeroI = findRowWithUnmarkElem(workMatrix, unmarkZeroJ);

workMatrix.setMarkToElem(unmarkZeroI, unmarkZeroJ, '\'');

printOutput(workMatrix);

if (checkExistMarkStarElementInRow(workMatrix, unmarkZeroI))

{

workMatrix.setMarkToRow(unmarkZeroI);

int indMarkedStarZero = findIndexMarkedStartZeroInRow(workMatrix, unmarkZeroI);

workMatrix.resetMarkFromColumn(indMarkedStarZero);

printOutput(workMatrix);

}

else

{

flgMarkedStarZeroInRow = false;

LChain lchain = new LChain();

ArrayList<ElementLChain> masElem = lchain.getLChain(workMatrix, unmarkZeroI, unmarkZeroJ);

workMatrix = replaceFromL(workMatrix, masElem);

workMatrix.delAllMarksFromRowsAndColumns();

workMatrix.delMarkFromAllElements('\'');

printOutput(workMatrix);

}

}

return workMatrix;

}

private ExtendMatrix improveMatrix(ExtendMatrix matrix)

{

ExtendMatrix resMatrix = matrix;

double h = findMinUnmarkedElem(resMatrix);

resMatrix = subHFromUnmarkedRows(resMatrix, h);

resMatrix = addHToMarkedColumns(resMatrix, h);

printOutput(resMatrix);

return resMatrix;

}

private double findMinUnmarkedElem(ExtendMatrix matrix)

{

double minElem = 0;

boolean flgFindUnmarkedElem = false;

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

if (!matrix.hasRowMark(i))

for (int j = 0; j < n; j++)

if (!matrix.hasColumnMark(j))

{

if (!flgFindUnmarkedElem)

{

flgFindUnmarkedElem = true;

minElem = matrix.getVal(i, j);

}

else

{

double elem = matrix.getVal(i, j);

if (elem < minElem)

minElem = elem;

}

}

return minElem;

}

private ExtendMatrix subHFromUnmarkedRows(ExtendMatrix matrix, double h)

{

int n = matrix.getDegree();

ExtendMatrix res = matrix;

for (int i = 0; i < n; i++)

if (!res.hasRowMark(i))

for (int j = 0; j < n; j++)

res.setVal(i, j, res.getVal(i, j) - h);

return res;

}

private ExtendMatrix addHToMarkedColumns(ExtendMatrix matrix, double h)

{

int n = matrix.getDegree();

ExtendMatrix res = matrix;

for (int j = 0; j < n; j++)

if (res.hasColumnMark(j))

for (int i = 0; i < n; i++)

res.setVal(i, j, res.getVal(i, j) + h);

return res;

}

public int findColumnWithUnmarkElem(ExtendMatrix matrix)

{

int n = matrix.getDegree();

for (int j = 0; j < n; j++)

if (!matrix.hasColumnMark(j) && isContentColumnUnmarkElem(matrix, j))

return j;

return -1;

}

private boolean isContentColumnUnmarkElem(ExtendMatrix matrix, int indexColumn)

{

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

if (!matrix.hasRowMark(i) && isElementUnmarked(matrix, i, indexColumn))

return true;

return false;

}

private boolean isElementUnmarked(ExtendMatrix matrix, int indexRow, int indexColumn)

{

double elem = matrix.getVal(indexRow, indexColumn);

return (isElementEqualsZero(elem) &&

!matrix.hasElemMark(indexRow, indexColumn, '\*') &&

!matrix.hasElemMark(indexRow, indexColumn, '\''));

}

public int findRowWithUnmarkElem(ExtendMatrix matrix, int indexColumn)

{

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

if (!matrix.hasRowMark(i) && isElementUnmarked(matrix, i, indexColumn))

return i;

return -1;

}

private boolean checkExistUnmarkedZero(ExtendMatrix matrix)

{

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

if (!matrix.hasRowMark(i) && checkExistUnmarkedZeroInRow(matrix, i))

return true;

return false;

}

private boolean checkExistUnmarkedZeroInRow(ExtendMatrix matrix, int indexRow)

{

int n = matrix.getDegree();

for (int j = 0; j < n; j++)

if (checkElementForEqualsZeroAndUnmarked(matrix, indexRow, j))

return true;

return false;

}

private boolean checkElementForEqualsZeroAndUnmarked(ExtendMatrix matrix, int indexRow, int indexColumn)

{

double elem = matrix.getVal(indexRow, indexColumn);

return !matrix.hasColumnMark(indexColumn) && isElementEqualsZero(elem);

}

private boolean checkExistMarkStarElementInRow(ExtendMatrix matrix, int i)

{

int n = matrix.getDegree();

for (int j = 0; j < n; j++)

if (matrix.hasElemMark(i, j, '\*'))

return true;

return false;

}

private int findIndexMarkedStartZeroInRow(ExtendMatrix matrix, int i)

{

int n = matrix.getDegree();

for (int j = 0; j < n; j++)

if (matrix.hasElemMark(i, j, '\*'))

return j;

return -1;

}

private ExtendMatrix replaceFromL(ExtendMatrix matrix, ArrayList<ElementLChain> lchain)

{

ExtendMatrix resMatrix = matrix;

for (ElementLChain elem : lchain)

if (elem.getMark() == '\*')

resMatrix.setMarkToElem(elem.getI(), elem.getJ(), (char) 0);

else if (elem.getMark() == '\'')

resMatrix.setMarkToElem(elem.getI(), elem.getJ(), '\*');

return resMatrix;

}

private double minElemColumn(Matrix matrix, int j)

{

double minElem = matrix.getVal(0, j);

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

if (minElem > matrix.getVal(i, j))

minElem = matrix.getVal(i, j);

return minElem;

}

private double minElemRow(Matrix matrix, int i)

{

double minElem = matrix.getVal(i, 0);

int n = matrix.getDegree();

for (int j = 0; j < n; j++)

if (minElem > matrix.getVal(i, j))

minElem = matrix.getVal(i, j);

return minElem;

}

private Matrix transformMaximizationTaskToMinimizationTask(Matrix initMatrix)

{

int n = initMatrix.getDegree();

Matrix modifMatrix = new Matrix(n);

double maxElem = findMaxElem(initMatrix);

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

modifMatrix.setVal(i, j, maxElem - initMatrix.getVal(i, j));

return modifMatrix;

}

private double findMaxElem(Matrix matrix)

{

double maxElem = matrix.getVal(0, 0);

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

if (maxElem < matrix.getVal(i, j))

maxElem = matrix.getVal(i, j);

return maxElem;

}

private Matrix writeOptimalSolution(ExtendMatrix costMatrix)

{

int n = costMatrix.getDegree();

Matrix res = new Matrix(n);

for (int j = 0; j < n; j++)

res = addColumnInAssignmentMatrix(res, costMatrix, j);

return res;

}

private Matrix addColumnInAssignmentMatrix(Matrix assignmentMatrix, ExtendMatrix costMatrix, int indexColumn)

{

Matrix resMatrix = assignmentMatrix;

int n = resMatrix.getDegree();

for (int i = 0; i < n; i++)

resMatrix = establishValueOfAssignmentMatrixElement(assignmentMatrix, costMatrix, i, indexColumn);

return resMatrix;

}

private Matrix establishValueOfAssignmentMatrixElement(Matrix assignmentMatrix, ExtendMatrix costMatrix,

int indexRow, int indexColumn)

{

if (costMatrix.hasElemMark(indexRow, indexColumn, '\*'))

assignmentMatrix.setVal(indexRow, indexColumn, 1);

else

assignmentMatrix.setVal(indexRow, indexColumn, 0);

return assignmentMatrix;

}

private void printOutput(Matrix matrix)

{

if (hasEstablishPrintOutput())

matrix.print();

}

private void establishPrintOutput()

{

this.flgPrintOutput = true;

}

private void establishNotPrintOutput()

{

this.flgPrintOutput = false;

}

private boolean hasEstablishPrintOutput()

{

return this.flgPrintOutput;

}

private boolean flgPrintOutput;

private final double EPS = 1e-6;

}

public class ElementLChain

{

public ElementLChain() {};

public ElementLChain(int i, int j, char mark, double val)

{

aI = i;

aJ = j;

aMark = mark;

aVal = val;

}

public void setI(int i)

{

aI = i;

}

public int getI()

{

return aI;

}

public void setJ(int j)

{

aJ = j;

}

public int getJ()

{

return aJ;

}

public void setMark(char mark)

{

aMark = mark;

}

public char getMark()

{

return aMark;

}

public void setVal(double val)

{

aVal = val;

}

public double getVal()

{

return aVal;

}

private int aI = 0;

private int aJ = 0;

private char aMark = 0;

private double aVal = 0;

}

public class LChain

{

public LChain()

{

chain = new ArrayList<ElementLChain>();

}

public ArrayList<ElementLChain> getLChain(ExtendMatrix matrix, int startIndexRow, int startIndexColumn)

{

boolean flg = true;

int n = matrix.getDegree();

chain = new ArrayList<ElementLChain>();

chain.add(new ElementLChain(startIndexRow, startIndexColumn, '\'',

matrix.getVal(startIndexRow, startIndexColumn)));

int findIndexRow;

int findIndexColumn;

int oldIndexColumn = startIndexColumn;

while (flg)

{

findIndexRow = findIndexRowElementWithStar(matrix, oldIndexColumn);

if (findIndexRow >= 0)

{

findIndexColumn = findIndexColumnElementWithUnary(matrix, findIndexRow);

if (findIndexColumn >= 0)

{

chain.add(new ElementLChain(findIndexRow, oldIndexColumn, '\*', 0));

chain.add(new ElementLChain(findIndexRow, findIndexColumn, '\'', 0));

oldIndexColumn = findIndexColumn;

}

else

flg = false;

}

else

flg = false;

}

return chain;

}

private int findIndexRowElementWithStar(ExtendMatrix matrix, int indexColumn)

{

int n = matrix.getDegree();

for (int i = 0; i < n; i++)

if (matrix.hasElemMark(i, indexColumn, '\*'))

return i;

return -1;

}

private int findIndexColumnElementWithUnary(ExtendMatrix matrix, int indexRow)

{

int n = matrix.getDegree();

for (int j = 0; j < n; j++)

if (matrix.hasElemMark(indexRow, j, '\''))

return j;

return -1;

}

private ArrayList<ElementLChain> chain;

}

Результаты расчётов для задач из индивидуальных вариантов

Вариант 6

Задача минимизации

10,0 8,0 6,0 4,0 9,0

11,0 9,0 10,0 5,0 6,0

5,0 10,0 8,0 6,0 4,0

3,0 11,0 9,0 6,0 6,0

8,0 10,0 11,0 8,0 7,0

7,0 0,0\* 0,0 0,0 5,0

7,0 0,0 3,0 0,0\* 1,0

2,0 2,0 2,0 2,0 0,0\*

0,0\* 3,0 3,0 2,0 2,0

3,0 0,0 3,0 2,0 1,0

+ + + +

7,0 0,0\* 0,0 0,0 5,0

7,0 0,0 3,0 0,0\* 1,0

2,0 2,0 2,0 2,0 0,0\*

0,0\* 3,0 3,0 2,0 2,0

3,0 0,0 3,0 2,0 1,0

+ + + +

7,0 0,0\* 0,0' 0,0 5,0

7,0 0,0 3,0 0,0\* 1,0

2,0 2,0 2,0 2,0 0,0\*

0,0\* 3,0 3,0 2,0 2,0

3,0 0,0 3,0 2,0 1,0

+ + +

7,0 0,0\* 0,0' 0,0 5,0 +

7,0 0,0 3,0 0,0\* 1,0

2,0 2,0 2,0 2,0 0,0\*

0,0\* 3,0 3,0 2,0 2,0

3,0 0,0 3,0 2,0 1,0

+ + +

7,0 0,0\* 0,0' 0,0 5,0 +

7,0 0,0' 3,0 0,0\* 1,0

2,0 2,0 2,0 2,0 0,0\*

0,0\* 3,0 3,0 2,0 2,0

3,0 0,0 3,0 2,0 1,0

+ +

7,0 0,0\* 0,0' 0,0 5,0 +

7,0 0,0' 3,0 0,0\* 1,0 +

2,0 2,0 2,0 2,0 0,0\*

0,0\* 3,0 3,0 2,0 2,0

3,0 0,0 3,0 2,0 1,0

+ +

7,0 0,0\* 0,0' 0,0 5,0 +

7,0 0,0' 3,0 0,0\* 1,0 +

2,0 2,0 2,0 2,0 0,0\*

0,0\* 3,0 3,0 2,0 2,0

3,0 0,0' 3,0 2,0 1,0

7,0 0,0 0,0\* 0,0 5,0

7,0 0,0 3,0 0,0\* 1,0

2,0 2,0 2,0 2,0 0,0\*

0,0\* 3,0 3,0 2,0 2,0

3,0 0,0\* 3,0 2,0 1,0

0,0 0,0 1,0 0,0 0,0

0,0 0,0 0,0 1,0 0,0

0,0 0,0 0,0 0,0 1,0

1,0 0,0 0,0 0,0 0,0

0,0 1,0 0,0 0,0 0,0

= 28

Задача максимизации

1,0 3,0 5,0 7,0 2,0

0,0 2,0 1,0 6,0 5,0

6,0 1,0 3,0 5,0 7,0

8,0 0,0 2,0 5,0 5,0

3,0 1,0 0,0 3,0 4,0

1,0 3,0 5,0 4,0 0,0\*

0,0\* 2,0 1,0 3,0 3,0

5,0 0,0\* 2,0 1,0 4,0

8,0 0,0 2,0 2,0 3,0

3,0 1,0 0,0\* 0,0 2,0

+ + + +

1,0 3,0 5,0 4,0 0,0\*

0,0\* 2,0 1,0 3,0 3,0

5,0 0,0\* 2,0 1,0 4,0

8,0 0,0 2,0 2,0 3,0

3,0 1,0 0,0\* 0,0 2,0

+ + + +

1,0 3,0 5,0 4,0 0,0\*

0,0\* 2,0 1,0 3,0 3,0

5,0 0,0\* 2,0 1,0 4,0

8,0 0,0 2,0 2,0 3,0

3,0 1,0 0,0\* 0,0' 2,0

+ + +

1,0 3,0 5,0 4,0 0,0\*

0,0\* 2,0 1,0 3,0 3,0

5,0 0,0\* 2,0 1,0 4,0

8,0 0,0 2,0 2,0 3,0

3,0 1,0 0,0\* 0,0' 2,0 +

+ + +

1,0 3,0 4,0 3,0 0,0\*

0,0\* 2,0 0,0 2,0 3,0

5,0 0,0\* 1,0 0,0 4,0

8,0 0,0 1,0 1,0 3,0

4,0 2,0 0,0\* 0,0' 3,0 +

+ + +

1,0 3,0 4,0 3,0 0,0\*

0,0\* 2,0 0,0' 2,0 3,0

5,0 0,0\* 1,0 0,0 4,0

8,0 0,0 1,0 1,0 3,0

4,0 2,0 0,0\* 0,0' 3,0 +

+ +

1,0 3,0 4,0 3,0 0,0\*

0,0\* 2,0 0,0' 2,0 3,0 +

5,0 0,0\* 1,0 0,0 4,0

8,0 0,0 1,0 1,0 3,0

4,0 2,0 0,0\* 0,0' 3,0 +

+ +

1,0 3,0 4,0 3,0 0,0\*

0,0\* 2,0 0,0' 2,0 3,0 +

5,0 0,0\* 1,0 0,0' 4,0

8,0 0,0 1,0 1,0 3,0

4,0 2,0 0,0\* 0,0' 3,0 +

+

1,0 3,0 4,0 3,0 0,0\*

0,0\* 2,0 0,0' 2,0 3,0 +

5,0 0,0\* 1,0 0,0' 4,0 +

8,0 0,0 1,0 1,0 3,0

4,0 2,0 0,0\* 0,0' 3,0 +

+

1,0 3,0 4,0 3,0 0,0\*

0,0\* 2,0 0,0' 2,0 3,0 +

5,0 0,0\* 1,0 0,0' 4,0 +

8,0 0,0' 1,0 1,0 3,0

4,0 2,0 0,0\* 0,0' 3,0 +

1,0 3,0 4,0 3,0 0,0\*

0,0\* 2,0 0,0 2,0 3,0

5,0 0,0 1,0 0,0\* 4,0

8,0 0,0\* 1,0 1,0 3,0

4,0 2,0 0,0\* 0,0 3,0

0,0 0,0 0,0 0,0 1,0

1,0 0,0 0,0 0,0 0,0

0,0 0,0 0,0 1,0 0,0

0,0 1,0 0,0 0,0 0,0

0,0 0,0 1,0 0,0 0,0