Лабораторная работа №3

Кендысь Алексей, 3 курс, 7а группа

Условие задачи

Задача 8. Станки

Конвейер состоит из N различных станков. Есть N рабочих. Известна матрица C размера $N \times N$, где элемент c_{ij} задаёт производительность i-го рабочего на j-м станке. Необходимо определить, каким должно быть распределение рабочих по станкам (каждый рабочий может быть назначен только на один станок, на каждом станке может работать только один рабочий), чтобы производительность всего конвейера была максимальной. Производительность конвейера при некотором распределении рабочих по станкам равна минимальной производительности рабочих на назначенных им на конвейере станках. Если решение не единственно, вывести решение, первое в лексикографическом порядке среди всех решений.

Формат входных данных

Первая строка содержит число рабочих (станков) N ($1 \le N \le 500$). Затем идут N строк файла, которые задают матрицу C производительностей ($1 \le c_{ij} \le 1000$).

Формат выходных данных

В первой строке выведите максимальную возможную производительность конвейера. Во второй строке — номера станков, на которые должны быть распределены рабочие $1, 2, \ldots, N$ соответственно.

Пример

входной файл	выходной файл
4 4 2 6 1 5 2 3 3 4 7 1 4 4 3 2 5	5 3 1 2 4

Алгоритм

Задача из условия – это задача о назначениях. Задана матрица производительностей, требуется максимизировать общую производительность. Т.к. в обычной постановке задачи о назначениях производится минимизация, то матрица из условия преобразовывается. В итоге получаем следующий алгоритм:

1. Находится наибольший элемент матрицы $C^* = \max\limits_{i,j} c_{ij}$.
2. Строится новая матрица $R = (r_{ij})$, где $r_{ij} = C^* - c_{ij},\ i,j = \overline{1,N}$.
3. Для матрицы R применяется Венгерский алгоритм.

Листинг программы

```
import java.io.*;
import java.util.*;

class Hungarian {

    private final int numRows;
    private final int numCols;

    private final boolean[][] primes;
    private final boolean[][] stars;
    private final boolean[] rowsCovered;
    private final boolean[] colsCovered;
    private final int[][] costs;

    public Hungarian(int[][] theCosts) {
```

```
        costs = theCosts;
        numRows = costs.length;
        numCols = costs[0].length;

        primes = new boolean[numRows][numCols];
        stars = new boolean[numRows][numCols];

        // Инициализация массивов с покрытием строк/столбцов
        rowsCovered = new boolean[numRows];
        colsCovered = new boolean[numCols];
        for (int i = 0; i < numRows; i++) {
            rowsCovered[i] = false;
        }
        for (int j = 0; j < numCols; j++) {
            colsCovered[j] = false;
        }
        // Инициализация матриц
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                primes[i][j] = false;
                stars[i][j] = false;
            }
        }
    }

    public int[][] execute() {
        subtractRowColMins();

        this.findStars(); // O(n^2)
        this.resetCovered(); // O(n);
        this.coverStarredZeroCols(); // O(n^2)

        while (!allColsCovered()) {
            int[] primedLocation = this.primeUncoveredZero(); // O(n^2)

            // It's possible that we couldn't find a zero to prime, so we have
to induce some zeros, so we can find one to prime
            if (primedLocation[0] == -1) {
                this.minUncoveredRowsCols(); // O(n^2)
                primedLocation = this.primeUncoveredZero(); // O(n^2)
            }

            // is there a starred 0 in the primed zeros row?
            int primedRow = primedLocation[0];
            int starCol = this.findStarColInRow(primedRow);
            if (starCol != -1) {
                // cover the row of the primedLocation and uncover the star
column
                rowsCovered[primedRow] = true;
                colsCovered[starCol] = false;
            } else { // otherwise, we need to find an augmenting path and start
over.
                this.augmentPathStartingAtPrime(primedLocation);
                this.resetCovered();
                this.resetPrimes();
                this.coverStarredZeroCols();
            }
        }

        return this.starsToAssignments(); // O(n^2)

    }
```

```
    /*
     * the starred 0's in each column are the assignments.
     * O(n^2)
     */
    public int[][] starsToAssignments() {
        int[][] toRet = new int[numCols][];
        for (int j = 0; j < numCols; j++) {
            toRet[j] = new int[] {
                    this.findStarRowInCol(j), j
            }; // O(n)
        }
        return toRet;
    }

    /*
     * resets prime information
     */
    public void resetPrimes() {
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                primes[i][j] = false;
            }
        }
    }


    /*
     * resets covered information, O(n)
     */
    public void resetCovered() {
        for (int i = 0; i < numRows; i++) {
            rowsCovered[i] = false;
        }
        for (int j = 0; j < numCols; j++) {
            colsCovered[j] = false;
        }
    }

    /*
     * get the first zero in each column, star it if there isn't already a star
in that row
     * cover the row and column of the star made, and continue to the next
column
     * O(n^2)
     */
    public void findStars() {
        boolean[] rowStars = new boolean[numRows];
        boolean[] colStars = new boolean[numCols];

        for (int i = 0; i < numRows; i++) {
            rowStars[i] = false;
        }
        for (int j = 0; j < numCols; j++) {
            colStars[j] = false;
        }

        for (int j = 0; j < numCols; j++) {
            for (int i = 0; i < numRows; i++) {
                if (costs[i][j] == 0 && !rowStars[i] && !colStars[j]) {
                    stars[i][j] = true;
                    rowStars[i] = true;
                    colStars[j] = true;
                    break;
```

```
                }
            }
        }
    }

    /*
     * Finds the minimum uncovered value, and adds it to all the covered rows
then
     * subtracts it from all the uncovered columns. This results in a cost
matrix with
     * at least one more zero.
     */
    private void minUncoveredRowsCols() {
        // find min uncovered value
        int minUncovered = Integer.MAX_VALUE;
        for (int i = 0; i < numRows; i++) {
            if (!rowsCovered[i]) {
                for (int j = 0; j < numCols; j++) {
                    if (!colsCovered[j]) {
                        if (costs[i][j] < minUncovered) {
                            minUncovered = costs[i][j];
                        }
                    }
                }
            }
        }

        // add that value to all the COVERED rows.
        for (int i = 0; i < numRows; i++) {
            if (rowsCovered[i]) {
                for (int j = 0; j < numCols; j++) {
                    costs[i][j] = costs[i][j] + minUncovered;

                }
            }
        }

        // subtract that value from all the uncovered columns
        for (int j = 0; j < numCols; j++) {
            if (!colsCovered[j]) {
                for (int i = 0; i < numRows; i++) {
                    costs[i][j] = costs[i][j] - minUncovered;
                }
            }
        }
    }

    /*
     * Finds an uncovered zero, primes it, and returns an array
     * describing the row and column of the newly primed zero.
     * If no uncovered zero could be found, returns -1 in the indices.
     * O(n^2)
     */
    private int[] primeUncoveredZero() {
        int[] location = new int[2];

        for (int i = 0; i < numRows; i++) {
            if (!rowsCovered[i]) {
                for (int j = 0; j < numCols; j++) {
                    if (!colsCovered[j]) {
                        if (costs[i][j] == 0) {
                            primes[i][j] = true;
                            location[0] = i;
```

```
                            location[1] = j;
                            return location;
                        }
                    }
                }
            }
        }

        location[0] = -1;
        location[1] = -1;
        return location;
    }

    /*
     * Starting at a given primed location[0=row,1=col], we find an augmenting
path
     * consisting of a primed , starred , primed , ..., primed. (note that it
begins and ends with a prime)
     * We do this by starting at the location, going to a starred zero in the
same column, then going to a primed zero in
     * the same row, etc., until we get to a prime with no star in the column.
     * O(n^2)
     */
    private void augmentPathStartingAtPrime(int[] location) {
        // Make the arraylists sufficiently large to begin with
        ArrayList<int[]> primeLocations = new ArrayList<>(numRows + numCols);
        ArrayList<int[]> starLocations = new ArrayList<>(numRows + numCols);
        primeLocations.add(location);

        int currentRow;
        int currentCol = location[1];
        while (true) { // add stars and primes in pairs
            int starRow = findStarRowInCol(currentCol);
            // at some point we won't be able to find a star. if this is the
case, break.
            if (starRow == -1) {
                break;
            }
            int[] starLocation = new int[] {
                    starRow, currentCol
            };
            starLocations.add(starLocation);
            currentRow = starRow;

            int primeCol = findPrimeColInRow(currentRow);
            int[] primeLocation = new int[] {
                    currentRow, primeCol
            };
            primeLocations.add(primeLocation);
            currentCol = primeCol;
        }

        unStarLocations(starLocations);
        starLocations(primeLocations);
    }


    /*
     * Given an arraylist of  locations, star them
     */
    private void starLocations(ArrayList < int[] > locations) {
        for (int[] location : locations) {
            int row = location[0];
```

```
            int col = location[1];
            stars[row][col] = true;
        }
    }

    /*
     * Given an arraylist of starred locations, unstar them
     */
    private void unStarLocations(ArrayList < int[] > starLocations) {
        for (int[] starLocation : starLocations) {
            int row = starLocation[0];
            int col = starLocation[1];
            stars[row][col] = false;
        }
    }


    /*
     * Given a row index, finds a column with a prime. returns -1 if this isn't
possible.
     */
    private int findPrimeColInRow(int theRow) {
        for (int j = 0; j < numCols; j++) {
            if (primes[theRow][j]) {
                return j;
            }
        }
        return -1;
    }




    /*
     * Given a column index, finds a row with a star. returns -1 if this isn't
possible.
     */
    public int findStarRowInCol(int theCol) {
        for (int i = 0; i < numRows; i++) {
            if (stars[i][theCol]) {
                return i;
            }
        }
        return -1;
    }


    public int findStarColInRow(int theRow) {
        for (int j = 0; j < numCols; j++) {
            if (stars[theRow][j]) {
                return j;
            }
        }
        return -1;
    }

    // looks at the colsCovered array, and returns true if all entries are true,
false otherwise
    private boolean allColsCovered() {
        for (int j = 0; j < numCols; j++) {
            if (!colsCovered[j]) {
                return false;
            }
```

```
        }
        return true;
    }

    /*
     * sets the columns covered if they contain starred zeros
     * O(n^2)
     */
    private void coverStarredZeroCols() {
        for (int j = 0; j < numCols; j++) {
            colsCovered[j] = false;
            for (int i = 0; i < numRows; i++) {
                if (stars[i][j]) {
                    colsCovered[j] = true;
                    break; // break inner loop to save a bit of time
                }
            }
        }
    }

    private void subtractRowColMins() {
        for (int i = 0; i < numRows; i++) { //for each row
            int rowMin = Integer.MAX_VALUE;
            for (int j = 0; j < numCols; j++) { // grab the smallest element in
that row
                if (costs[i][j] < rowMin) {
                    rowMin = costs[i][j];
                }
            }
            for (int j = 0; j < numCols; j++) { // subtract that from each
element
                costs[i][j] = costs[i][j] - rowMin;
            }
        }

        for (int j = 0; j < numCols; j++) { // for each col
            int colMin = Integer.MAX_VALUE;
            for (int i = 0; i < numRows; i++) { // grab the smallest element in
that column
                if (costs[i][j] < colMin) {
                    colMin = costs[i][j];
                }
            }
            for (int i = 0; i < numRows; i++) { // subtract that from each
element
                costs[i][j] = costs[i][j] - colMin;
            }
        }
    }

}

class AssignmentProblem {
    private final int n;
    private final int[][] c;
    private int productivity;
    private int[] lexOrder;

    public AssignmentProblem() throws IOException {
        StreamTokenizer st = new StreamTokenizer(new BufferedReader(new
FileReader("input.txt")));

        st.nextToken();
```

```
        n = (int) st.nval;

        c = new int[n][n];
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                st.nextToken();
                c[i][j] = (int) st.nval;
            }
        }
    }

    public void solve() {
        Hungarian h = new Hungarian(minToMax(c));
        int[][] res = h.execute();
        productivity = getProductivity(res);
        lexOrder = getOrder(res);
        lexicographicalOrder(lexOrder);
    }

    public void out() throws IOException {
        PrintWriter pw = new PrintWriter("output.txt");

        pw.print(productivity);
        pw.print('\n');

        pw.print(lexOrder[0]);
        for(int i = 1; i < n; i++) {
            pw.print(' ');
            pw.print(lexOrder[i]);
        }

        pw.close();
    }

    private int[] getOrder(int[][] hungaryRes) {
        int[] res = new int[n];

        Arrays.sort(hungaryRes, (o1, o2) -> {
            int i1 = o1[0];
            int i2 = o2[0];
            return i1 - i2;
        });

        for (int i = 0; i < n; i++) {
            res[i] = hungaryRes[i][1] + 1;
        }

        return res;
    }

    private void lexicographicalOrder(int[] order) {
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (c[i][order[i] - 1] == c[j][order[i] - 1] &&
                        c[i][order[j] - 1] == c[j][order[j] - 1] &&
                        order[i] > order[j]) {
                    int temp = order[i];
                    order[i] = order[j];
                    order[j] = temp;
                }
            }
        }
    }
```

```
    private int getProductivity(int[][] hungaryRes) {
        int res = Integer.MAX_VALUE;

        for (int[] pos : hungaryRes) {
            if (c[pos[0]][pos[1]] < res) {
                res = c[pos[0]][pos[1]];
            }
        }

        return res;
    }

    private int[][] minToMax(int[][] x) {
        int maxX = getMaxInMatrix(x);
        int[][] res = copyMatrix(x);

        for (int i = 0; i < x.length; i++) {
            for (int j = 0; j < x[i].length; j++) {
                res[i][j] = maxX - res[i][j];
            }
        }

        return res;
    }

    private int[][] copyMatrix(int[][] matrix) {
        int[][] res = new int[matrix.length][];

        for(int i = 0; i < matrix.length; i++) {
            int[] row = matrix[i];
            res[i] = new int[row.length];
            System.arraycopy(row, 0, res[i], 0, row.length);
        }

        return res;
    }

    private int getMaxInMatrix(int[][] matrix) {
        int maxElement = Integer.MIN_VALUE;

        for (int[] row : matrix) {
            for (int elem : row) {
                if (elem > maxElement) {
                    maxElement = elem;
                }
            }
        }

        return maxElement;
    }
}

public class Main {
    public static void main(String[] args) throws IOException {
        AssignmentProblem ap = new AssignmentProblem();
        ap.solve();
        ap.out();
    }
}
```

Результат

Входной файл input.txt:

```
4
4 2 6 1
5 2 3 3
4 7 1 4
4 3 2 5
```

Выходной файл output.txt:

```
5
3 1 2 4
```