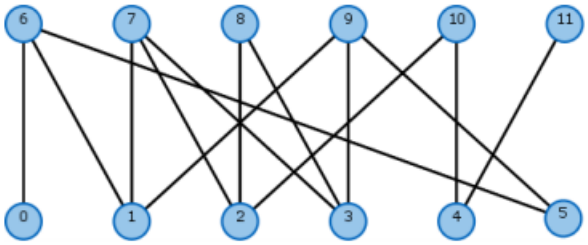
Лабораторная работа №6

Кендысь Алексей, 3 курс, 7а группа

Условие задачи

Решите две задачи. 1. Найдите максимальное паросочетание и минимальное вершинное покрытие в двудольном графе. 2. Решите задачу о назначениях:

Вариант 8.



5	9	5	1	1
5	3	3	6	7
3	9	7	5	2
4	1	6	9	7
4	2	1	8	8

Алгоритм

Для решения задачи на максимальное паросочетание задача сводится к задаче о максимальном потоке, для решения которой используется алгоритм Форда-Фалкерсона.

Минимальное вершинное покрытие находится из максимального паросочетания с помощью следующего алгоритма:

1. Построить максимальное паросочетание.
2. Ориентировать ребра:
   - Из паросочетания — из правой доли в левую.
   - Не из паросочетания — из левой доли в правую.
3. Запустить обход в глубину из всех свободных вершин левой доли, построить множества L^+, L^-, R^+, R^-.
4. В качестве результата взять $L^- \cup R^+$.

Для решения задачи о назначениях используется 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 cost;
    private int[] lexOrder;

    public AssignmentProblem() throws IOException {
        StreamTokenizer st = new StreamTokenizer(new BufferedReader(new
FileReader("input2.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(copyMatrix(c));
        int[][] res = h.execute();
        cost = getCost(res);
        lexOrder = getOrder(res);
        lexicographicalOrder(lexOrder);
    }

    public void out() throws IOException {
        PrintWriter pw = new PrintWriter("output3.txt");

        pw.print("Назначение:\n");
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                pw.print(c[i][j]);

                if (lexOrder[i] == j + 1) {
                    pw.print('*');
                }
                else {
                    pw.print(' ');
                }

                pw.print(' ');
            }
```

```
            pw.print('\n');
        }

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

        pw.print("\nСтоимость: ");
        pw.print(cost);

        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 getCost(int[][] hungaryRes) {
        int res = 0;

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

        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;
```

```
    }
}

class Edge {
    private final int source;
    private final int capacity;
    private int flow;
    private final int weight;
    private final int target;

    public Edge(int source, int capacity, int flow, int weight, int target) {
        this.source = source;
        this.capacity = capacity;
        this.flow = flow;
        this.weight = weight;
        this.target = target;
    }

    public int getSource() {
        return source;
    }

    public int getFlow() {
        return flow;
    }

    public void setFlow(int flow) {
        this.flow = flow;
    }

    public int getWeight() {
        return weight;
    }

    public int getTarget() {
        return target;
    }

    public int available() {
        return this.capacity - this.flow;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Edge edge = (Edge) o;
        return source == edge.source && capacity == edge.capacity && flow ==
edge.flow && weight == edge.weight && target == edge.target;
    }
}

class InitialGraph {
    protected final int n;
    protected final int m;
    protected final Edge[] flowEdges;
    protected List<Integer>[] adjLists;
    protected final int s;
    protected final int t;
    protected int nLeft;

    protected InitialGraph() throws IOException {
        StreamTokenizer st = new StreamTokenizer(new BufferedReader(new
FileReader("input1.txt")));

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

```
            st.nextToken();
            m = (int) st.nval;

            this.s = 1;
            this.t = n + 2;
            this.flowEdges = new Edge[(m + n) * 2];

            adjLists = new List[n];
            for (int i = 0; i < n; i++) {
                adjLists[i] = new LinkedList<>();
            }

            int u1, u2;
            int i = 0;
            for(int j = 0; j < m; j++) {
                st.nextToken();
                u1 = (int) st.nval + 2;

                if (adjLists[u1 - 2].isEmpty()) {
                    nLeft++;
                    i = addEdge(s, 1, u1, i);
                    i = addEdge(u1, 0, s, i);
                }

                st.nextToken();
                u2 = (int) st.nval + 2;

                i = addEdge(u1, 1, u2, i);
                i = addEdge(u2, 0, u1, i);

                if (adjLists[u2 - 2].isEmpty()) {
                    i = addEdge(u2, 1, t, i);
                    i = addEdge(t, 1, u2, i);
                }

                adjLists[u1 - 2].add(u2 - 2);
                adjLists[u2 - 2].add(u1 - 2);
            }
        }

        protected int getSourceOfEdge(int edgeInd) {
            return this.flowEdges[edgeInd].getSource();
        }

        private int addEdge(int source, int cap, int target, int i) {
            this.flowEdges[i] = new Edge(source, cap, 0, 0, target);
            return i + 1;
        }

        protected int available(int edgeInd) {
            return this.flowEdges[edgeInd].available();
        }
    }

    class Graph extends InitialGraph {
        private final int[] pred;
        private final double[] dist;

        public Graph() throws IOException {
            this.dist = new double[n + 2];
            this.pred = new int[n + 2];
        }

        public void findMaxFlowMinCost() {
            while (this.findShortestPathBellmanFord()) {
                this.processFlowFordFulkerson();
            }
```

```
}

private boolean findShortestPathBellmanFord() {
    this.dist[super.s - 1] = 0;
    Arrays.fill(this.dist, super.s, this.dist.length, Double.POSITIVE_INFINITY);
    Arrays.fill(this.pred, -1);

    for (int i = 0; i < n + 1; i++) {
        if (!this.performRelaxation()) {
            break;
        }
    }

    return this.dist[super.t - 1] != Double.POSITIVE_INFINITY;
}

private boolean performRelaxation() {
    boolean changeHappened = false;
    for (int j = 0; j < super.flowEdges.length; j++) {
        Edge edge = super.flowEdges[j];
        if (edge.available() > 0) {
            int v = edge.getSource();
            int u = edge.getTarget();
            int c = edge.getWeight();

            if (this.dist[u - 1] > this.dist[v - 1] + c) {
                this.dist[u - 1] = this.dist[v - 1] + c;
                this.pred[u - 1] = j;
                changeHappened = true;
            }
        }
    }

    return changeHappened;
}

private void processFlowFordFulkerson() {
    int cMin = this.findMinC();
    for (int i = super.t; i != this.s;) {
        int edgeInd = this.getEdgeIndex(i);
        this.pushEdge(edgeInd, cMin);
        i = super.getSourceOfEdge(edgeInd);
    }
}

private int findMinC() {
    int cMin = Integer.MAX_VALUE;
    for (int i = this.t; i != this.s;) {
        int edgeInd = this.getEdgeIndex(i);
        if (super.available(edgeInd) < cMin) {
            cMin = super.available(edgeInd);
        }

        i = super.getSourceOfEdge(edgeInd);
    }

    return cMin;
}

private int getEdgeIndex(int targetVertex) {
    return this.pred[targetVertex - 1];
}

private void pushEdge(int edgeInd, int flow) {
    Edge edge = super.flowEdges[edgeInd];
    edge.setFlow(edge.getFlow() + flow);
```

```
        Edge reverseEdge = super.flowEdges[edgeInd ^ 1];
        reverseEdge.setFlow(reverseEdge.getFlow() - flow);
    }
}

class MaxMatchingProblem {
    public Graph g;
    public final List<Edge> maxMatching;

    public MaxMatchingProblem() throws IOException {
        g = new Graph();
        maxMatching = new ArrayList<>();
    }

    public void solve() {
        g.findMaxFlowMinCost();

        for (int i = 0; i < g.flowEdges.length; i += 2) {
            Edge edge = g.flowEdges[i];

            if (edge.getFlow() == 1 && edge.getSource() != g.s && edge.getTarget()
!= g.t) {
                maxMatching.add(edge);
            }
        }
    }

    public void out() throws IOException {
        PrintWriter pw = new PrintWriter("output1.txt");

        pw.print("Максимальное паросочетание: {");

        Edge edge = maxMatching.get(0);
        pw.print("{");
        pw.print(edge.getSource() - 2);
        pw.print(',');
        pw.print(edge.getTarget() - 2);
        pw.print('}');

        for (int i = 1; i < maxMatching.size(); i++) {
            edge = maxMatching.get(i);

            pw.print(",{");
            pw.print(edge.getSource() - 2);
            pw.print(',');
            pw.print(edge.getTarget() - 2);
            pw.print('}');
        }
        pw.print('}');

        pw.close();
    }
}

class MinVertexCoverProblem {
    private final Graph g;
    private final List<Integer>[] adjLists;
    private final List<Edge> maxMatching;
    private final boolean[] visited;
    private final int[] result;

    public MinVertexCoverProblem(Graph g, List<Edge> maxMatching) {
        this.g = g;
        this.adjLists = g.adjLists;
        this.maxMatching = maxMatching;
        this.visited = new boolean[g.n];
        result = new int[maxMatching.size()];
```

```
    }

    public void solve() {
        transformAdjLists();

        for (int i = 0; i < g.nLeft; i++) {
            for (int j = g.nLeft; j < g.n; j++) {
                if (adjLists[j].contains(i)) {
                    break;
                }
                else if (j == g.n - 1) {
                    dfs(i);
                }
            }
        }

        int k = 0;
        for (int i = 0; i < g.n; i++) {
            if ((i < g.nLeft && !visited[i]) || (i >= g.nLeft && visited[i])) {
                result[k] = i;
                k++;
            }
        }
    }

    public void out() throws IOException {
        PrintWriter pw = new PrintWriter("output2.txt");

        pw.print("Минимальное вершинное покрытие: {");

        pw.print(result[0]);
        for (int i = 1; i < result.length; i++) {
            pw.print(',');
            pw.print(result[i]);
        }
        pw.print('}');

        pw.close();
    }

    private void transformAdjLists() {
        for (int u = 0; u < g.nLeft; u++) {
            ListIterator<Integer> it = adjLists[u].listIterator();
            while(it.hasNext()){
                Integer v = it.next();
                Edge edge = new Edge(u + 2, 1, 1, 0, v + 2);

                if (maxMatching.contains(edge)) {
                    it.remove();
                }
                else {
                    adjLists[v].remove((Integer) u);
                }
            }
        }
    }

    private void dfs(int vertex) {
        visited[vertex] = true;

        for (int adj : adjLists[vertex]) {
            if (!visited[adj])
                dfs(adj);
        }
    }
}
```

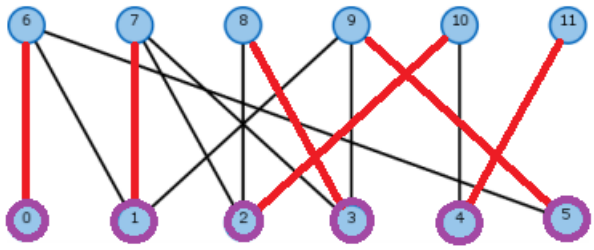
```
public class Main {
    public static void main(String[] args) throws IOException {
        MaxMatchingProblem mmp = new MaxMatchingProblem();
        mmp.solve();
        mmp.out();

        MinVertexCoverProblem mvcp = new MinVertexCoverProblem(mmp.g,
mmp.maxMatching);
        mvcp.solve();
        mvcp.out();

        AssignmentProblem ap = new AssignmentProblem();
        ap.solve();
        ap.out();
    }
}
```

Результат

Задание 1:

```
Максимальное паросочетание: {{0,6},{1,7},{2,10},{3,8},{4,11},{5,9}}
```

```
Минимальное вершинное покрытие: {0,1,2,3,4,5}
```



Задание 2:

```
Назначение:
5  9  5  1* 1
5* 3  3  6  7
3  9  7  5  2*
4  1* 6  9  7
4  2  1* 8  8

Ответ: 4 1 5 2 3
Стоимость: 10
```