Зуйкевич Лидия, 7 группа

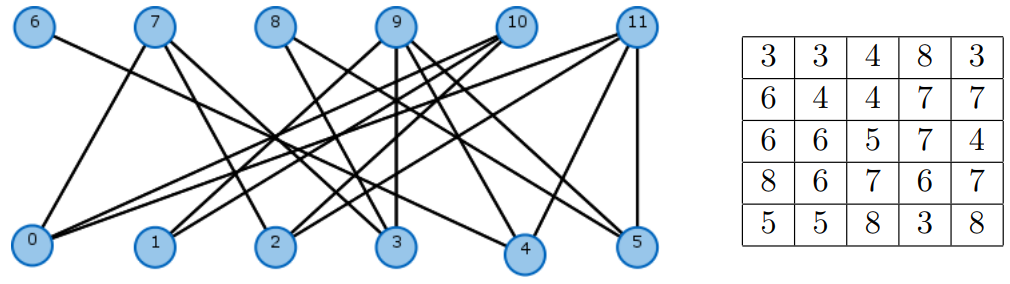
Лабораторная работа №6

Вариант 7

**Задание:**

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

**Условие:**



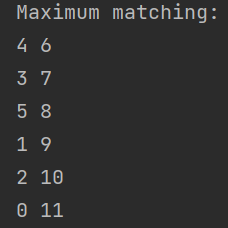
**Решение:**

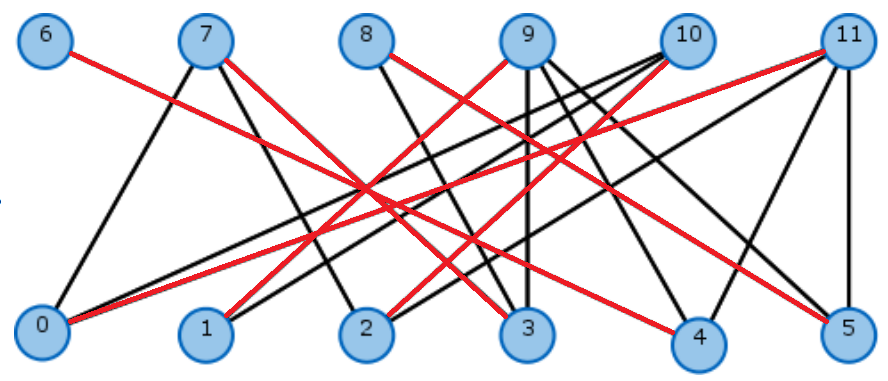
**1. Максимальное паросочетание**

Будем использовать алгоритм Куна, основанный на поиске пополняющего пути.

Изначальное паросочетание пустое. Т.к. граф двудольный, обход идет только по вершинам первой доли.

**Результат:**

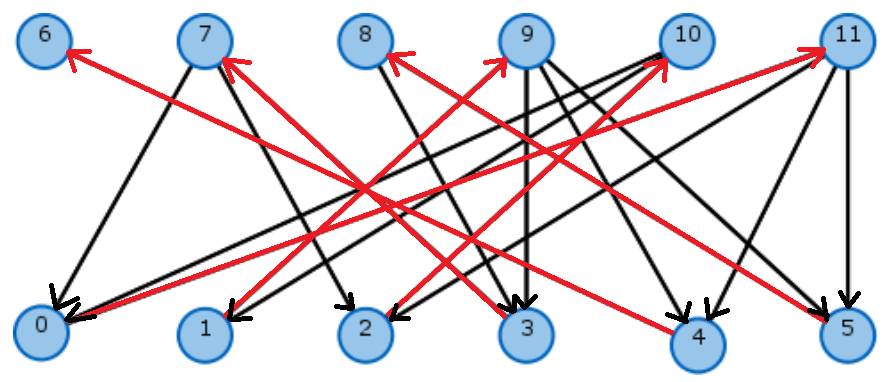
****

****

**2. Минимальное вершинное покрытие**

Используем построенное максимально паросочетание. Ориентируем ребра графа:

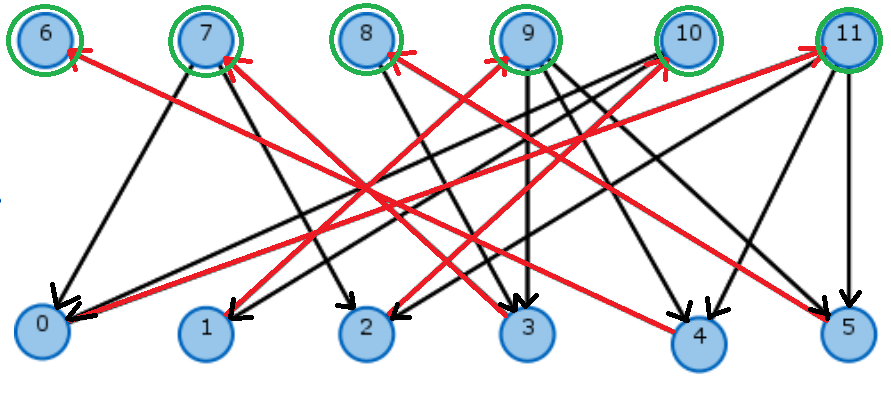
Ребра паросочетания – из правой доли в левую, не из паросочетания – из левой в правую.



Запускаем поиск в глубину из всех вершин левой доли, в которые не входят дуги, получаем множества посещенных ( и непосещенных( вершин правой и левой доли. Минимальное вершинное покрытие: .

**Результат:**

****

****

**Листинг программы:**

import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import java.util.StringTokenizer;  
  
  
public class MaxMatchingMinVCover {  
 private int n;  
 private int k;  
 List[] list;  
 private int[] is\_visited;  
 private int[] marks;  
 private int[] matching;  
  
  
 MaxMatchingMinVCover(String filename) throws IOException{  
 BufferedReader br = new BufferedReader(new FileReader(filename));  
 StringTokenizer st = new StringTokenizer(br.readLine(), " ");  
  
 *//!! Количество вершин 1-ой доли* this.n = Integer.*parseInt*(st.nextToken());  
 *//!! Количество вершин 2-ой доли* this.k = Integer.*parseInt*(st.nextToken());  
  
 this.is\_visited = new int[n + k];  
 this.marks = new int[n + k];  
 this.matching = new int[k + n];  
 this.list = new List [n + k];  
 for(int i = 0; i < n + k; i++){  
 list[i] = new ArrayList<Integer>();  
 }  
  
 *// Первая доля - которая с 0, во входном файле номер вершины 1-ой доли и номера смежных вершин 2-ой доли* String line = br.readLine();  
 while(line != null){  
 st = new StringTokenizer(line, " ");  
 int i = Integer.*parseInt*(st.nextToken());  
  
 while(st.hasMoreTokens()){  
 int v = Integer.*parseInt*(st.nextToken());  
 list[i].add(v);  
 }  
 line = br.readLine();  
 }  
 }  
  
  
 public void print\_lists(){  
 for(int i = 0; i < n + k; i++){  
 for(int j = 0; j < list[i].size(); j++)  
 System.*out*.print(list[i].get(j) + " ");  
 System.*out*.println();  
 }  
 }  
  
  
 public boolean kuhn\_algorithm(int v){  
 if (is\_visited[v] == 1)  
 return false;  
  
 is\_visited[v] = 1;  
  
 for (int i = 0; i < list[v].size(); i++) {  
 int t = Integer.*parseInt*(list[v].get(i).toString());  
  
 if (matching[t] == -1 || kuhn\_algorithm(matching[t])) {  
 matching[t] = v;  
 return true;  
 }  
 }  
 return false;  
 }  
  
  
 public void DepthFirstSearch (int start) {  
 is\_visited[start] = 1;  
  
 for (int i = 0; i < list[start].size(); i++) {  
 int v = Integer.*parseInt*(list[start].get(i).toString());  
 if (is\_visited[v] == 0) {  
 DepthFirstSearch(v);  
 }  
 }  
 }  
  
  
 public void min\_vertex\_cover(){  
 Arrays.*fill*(marks, 1);  
 for (int i = 0; i < n; i++){  
 for(int j = 0; j < list[i].size(); j++){  
 int v = Integer.*parseInt*(list[i].get(j).toString());  
  
 if(matching[v] != i) {  
 list[v].add(i);  
 }  
 else{  
 marks[v] = 0;  
 }  
 }  
 }  
  
 Arrays.*fill*(is\_visited, 0);  
 for(int i = n; i < n + k; i++){  
 if(marks[i] == 1 && is\_visited[i] == 0)  
 DepthFirstSearch(i);  
  
 }  
  
 System.*out*.println("Minimum Vertex Cover:");  
 for(int i = 0; i < n; i++){  
 if (is\_visited[i] == 1){  
 System.*out*.print(i + " ");  
 }  
 }  
  
 for(int i = n - 1; i < n + k; i++){  
 if(is\_visited[i] == 0){  
 System.*out*.print(i + " ");  
 }  
 }  
 }  
  
  
 public void solve(){  
 Arrays.*fill*(matching, -1);  
  
 for (int v = 0; v < n; v++) {  
 Arrays.*fill*(is\_visited, 0);  
 kuhn\_algorithm(v);  
 }  
  
 System.*out*.println("Maximum matching:");  
 for (int i = 0; i < k + n; i++)  
 if (matching[i] != -1){  
 System.*out*.println(matching[i] + " " + i);  
 }  
  
 min\_vertex\_cover();  
 }  
  
  
 public static void main(String[] args) throws IOException{  
 MaxMatchingMinVCover obj = new MaxMatchingMinVCover("input.txt");  
 *//obj.print\_lists();* obj.solve();  
 }  
}

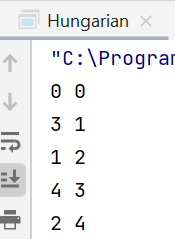
**3. Задача о назначениях**

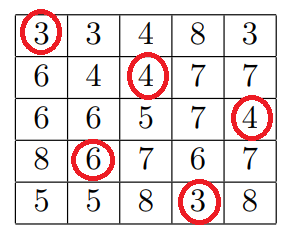
Используем венгерский алгоритм.

**Листинг программы:**

import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.StringTokenizer;  
  
public class Hungarian {  
  
 private int n;  
  
 private boolean[][] primes;  
 private boolean[][] stars;  
 private boolean[] rowsCovered;  
 private boolean[] colsCovered;  
 private int[][] costs;  
  
 public Hungarian(String filename) throws IOException {  
 BufferedReader br = new BufferedReader(new FileReader(filename));  
 StringTokenizer st = new StringTokenizer(br.readLine(), " ");  
  
 this.n = Integer.*parseInt*(st.nextToken());  
 this.costs = new int[n][n];  
  
 primes = new boolean[n][n];  
 stars = new boolean[n][n];  
  
 *// Инициализация массивов с покрытием строк/столбцов* rowsCovered = new boolean[n];  
 colsCovered = new boolean[n];  
 Arrays.*fill*(rowsCovered, false);  
 Arrays.*fill*(colsCovered, false);  
 *// Инициализация матриц* for (int i = 0; i < n; i++) {  
 Arrays.*fill*(primes[i], false);  
 Arrays.*fill*(stars[i], false);  
 }  
  
 for(int i = 0; i < n; i++){  
 st = new StringTokenizer(br.readLine(), " ");  
 for(int j = 0; j < n; j++){  
 costs[i][j] = Integer.*parseInt*(st.nextToken());  
 }  
 }  
 }  
  
  
 public void print\_matr(int[][] matr){  
 for(int[] row: matr){  
 for(int elem: row)  
 System.*out*.print(elem+ " ");  
 System.*out*.println();  
 }  
 }  
  
  
 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 ther 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[n][];  
 for (int j = 0; j < n; j++) {  
 toRet[j] = new int[] {  
 this.findStarRowInCol(j), j  
 }; *// O(n)* }  
 return toRet;  
 }  
  
 */\*  
 \* resets prime information  
 \*/* public void resetPrimes() {  
 for (int i = 0; i < n; i++) {  
 Arrays.*fill*(primes[i], false);  
 }  
 }  
  
  
 */\*  
 \* resets covered information, O(n)  
 \*/* public void resetCovered() {  
 Arrays.*fill*(rowsCovered, false);  
 Arrays.*fill*(colsCovered, 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[n];  
 boolean[] colStars = new boolean[n];  
  
 Arrays.*fill*(rowStars, false);  
 Arrays.*fill*(colStars, false);  
  
 for (int j = 0; j < n; j++) {  
 for (int i = 0; i < n; 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 < n; i++) {  
 if (!rowsCovered[i]) {  
 for (int j = 0; j < n; 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 < n; i++) {  
 if (rowsCovered[i]) {  
 for (int j = 0; j < n; j++) {  
 costs[i][j] = costs[i][j] + minUncovered;  
  
 }  
 }  
 }  
  
 *// subtract that value from all the UNcovered columns* for (int j = 0; j < n; j++) {  
 if (!colsCovered[j]) {  
 for (int i = 0; i < n; 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 < n; i++) {  
 if (!rowsCovered[i]) {  
 for (int j = 0; j < n; 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 < int[] > (2 \* n);  
 ArrayList < int[] > starLocations = new ArrayList < int[] > (2 \* n);  
 primeLocations.add(location);  
  
 int currentRow = location[0];  
 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 k = 0; k < locations.size(); k++) {  
 int[] location = locations.get(k);  
 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 k = 0; k < starLocations.size(); k++) {  
 int[] starLocation = starLocations.get(k);  
 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 < n; 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 < n; i++) {  
 if (stars[i][theCol]) {  
 return i;  
 }  
 }  
 return -1;  
 }  
  
  
 public int findStarColInRow(int theRow) {  
 for (int j = 0; j < n; 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 < n; 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 < n; j++) {  
 colsCovered[j] = false;  
 for (int i = 0; i < n; 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 < n; i++) { *//for each row* int rowMin = Integer.*MAX\_VALUE*;  
 for (int j = 0; j < n; j++) { *// grab the smallest element in that row* if (costs[i][j] < rowMin) {  
 rowMin = costs[i][j];  
 }  
 }  
 for (int j = 0; j < n; j++) { *// subtract that from each element* costs[i][j] = costs[i][j] - rowMin;  
 }  
 }  
  
 for (int j = 0; j < n; j++) { *// for each col* int colMin = Integer.*MAX\_VALUE*;  
 for (int i = 0; i < n; i++) { *// grab the smallest element in that column* if (costs[i][j] < colMin) {  
 colMin = costs[i][j];  
 }  
 }  
 for (int i = 0; i < n; i++) { *// subtract that from each element* costs[i][j] = costs[i][j] - colMin;  
 }  
 }  
 }  
  
  
 public static void main(String[] args) throws IOException{  
 Hungarian obj = new Hungarian("input1.txt");  
 obj.print\_matr(obj.execute());  
 }  
  
}

**Результат:**

****

****

Суммарная стоимость: 20