Vilniaus Universitetas

Matematikos ir Informatikos Fakultetas
III kursas 1 grupė
Ričardas Kazakevičius

Užduotis 11:
Kelio labirinte paieška su grįžimu atgal

Duota: Labirintas, turintis m eilučių ir n stulpelių bei pradinis taškas (i,j).
Rasti: Kelią išeiti iš labirinto naudojant paiešką su grįžimu (backtracking).

Uždavinys įgyvendintas Java programavimo kalba. Programoje labirintas yra vaizduojamas dviem stačiakampėmis m x n matricomis: vienoje vienetu žymime, jei yra horizontali pertvara ir nuliu, jei jos nėra; kitoje analogiškai žymime vertikalias pertvaras.

Pavyzdžiui, pasirinkus m ir n reikšmes lygias penketui, o pertvarų skaičių lygu dvidešimčiai, tada matricos bus vaizduojamos taip:

Horizontalias pertvaras turinti matrica	Vertikalias pertvaras turinti matrica
0 0 0 0 0	0 1 1 1 0
0 0 0 1 1	0 1 1 0 0
0 1 1 1 1	0 0 0 1 0
1 0 1 1 1	0 1 1 0 0
0 0 0 0 0	0 1 0 1 0

Labirinte vaikštant į viršų arba į apačią yra žiūrima į horizontalias pertvaras turinčia matricą ir pajudama yra viena pozicija tik tada, jeigu reikšmė nebus lygi vienetui. Analogiškai vaikštant į kairę arba į dešinę yra žiūrima į vertikalias pertvaras turinčia matricą ir tikrinama langelio reikšmė.

Programoje kuriant labirintą, reikia nurodyti eilučių, stulpelių ir pertvarų skaičius. Programa puse pertvarų priskiria vienai matricai, likusią dalį kitai matricai. Pertvarų pozicijos sugeneruojamos visiškai atsitiktinai. Toliau metodui findPath reikia pateikti keturis skaičius – pirmi du skaičiai yra labirinto pradžios koordinates, likę du – labirinto pabaigos (išėjimo) koordinatės.

Žingsnių labirinte algoritmo su grįžimu atgal paaiškinimas bei realizacija:

Metodas step priima trys parametrus – koordinates x ir y su kuriomis norima daryti žingsnį bei matrica, kurią tikrinsime (horizontalias arba vertikalias pertvaras turinti matrica). Pradžioje yra patikrinama ar nėra pasiektas finišas. Jei nėra, tada patikrinama ar esama pozicija yra leistina, tai yra ar neperžengėme labirinto ribų ir ar nėra pertvaros esamoje celėje bei ar celė nėra jau pažymėta kaip aplankyta ar negalima. Jei ne, tada pasižymime esamą celę kaip kelio dalį ir bandome eiti į dešinę, jei nepavyksta tada einame žemyn, jei vėl nepavyksta - į kairę ir galiausiai jei nepavyksta tada einama į apačią. Jei ir pastarasis bandymas nesėkmingas atžymime esama celę - ji jau nebebus mūsų kelio link finišo dalis. Pabaigoje galiausiai metodas step gražins true jeigu finišas bus pasiektas ir false jeigu neišeis surasti finišo.

Algoritmo realizacija:

```
 private boolean step(int x, int y, Matrix matrix) {
       ++steps;

       if ((y == finishCol)  &&  (x == finishRow)) { // found exit
          pathmatrix.data[x][y] = "F"; // exit position
          return true;
       }

       if ( (x == row) || (y == col) || (x < 0) || (y < 0) || ("1".equals(matrix.data[x][y]))
             || ("*".equals(pathmatrix.data[x][y]))  || ("!".equals(pathmatrix.data[x][y])) ){
          return false;
       }

       // mark location as part of path
       pathmatrix.data[x][y] = "*";

       boolean result;

       // try to go right
       result = step(x, y+1, vmatrix);
       if (result)
          return true;

       // try to go down
       result = step(x+1, y, hmatrix);
       if (result)
          return true;

       // try to go left
       result = step(x, y-1, vmatrix);
       if (result)
          return true;

       // try to go up
       result = step(x-1, y, hmatrix);
       if (result)
          return true;

       // mark ! as this cell is a dead end
       pathmatrix.data[x][y] = "!";

       return false;
    }
```

Sudėtingumo analizė atlikta su keturiais skirtingais skaičiais (n,m) = { (5,5), (8,8), (11,11), (14,14) } bei pertvarų skaičius kiekvienam bandymui - (n+m)/2, n*m ir trys skaičiai vienodai didėjantis nuo pradžios ir pabaigos. Pavyzdžiui, pirmas bandymas atliktas su n ir m lygiais penkiems, o pertvarų skaičiai - 5, 10, 15, 20, 25. Visi bandymai atliekami su vienodomis pradžios ir pabaigos koordinatėmis – pradžia - (0,0), pabaiga – (m-1,n-1).

Siekiant gauti kuo tikslesnius rezultatus, kiekvienas bandymas buvo atliekamas šimtą kartų. Analizės rezultatai pavaizduoti lentelėmis. Pirma lentelės eilutė reiškia labirinto eilučių ir stulpelių skaičių. Antra – pertvarų skaičius. Trečia eilutė - kiek iš šimto bandymų pavyko rasti išėjimą iš labirinto. Ketvirta – žingsnių vidurkis.

(N,M) = (5,5)					
Partitions	5	10	15	20	25
Times exit found	99	78	25	2	0
Average steps	15	21	20	12	6

(N,M) = (8,8)					
Partitions	8	22	36	50	64
Times exit found	100	90	61	11	0
Average steps	25	32	53	39	15

(N,M) = (11,11)					
Partitions	11	38	65	92	121
Times exit found	100	96	76	24	0
Average steps	32	47	71	69	23

(N,M) = (14,14)					
Partitions	14	60	106	152	196
Times exit found	100	97	75	26	0
Average steps	41	57	103	122	32

Išvada: analizuojant bandymo rezultatus, galima padaryti išvada, kad esant mažam pertvarų skaičiui (N+M) - beveik visada pavyksta rasti išėjimą iš labirinto, o esant didelėms pertvarų reikšmėms (N*M) – niekada nepavyksta rasti išėjimo. Su likusiomis pertvarų reikšmėms galima pastebėti, kad kuo didesnės N ir M reikšmėmis – tuo didesnė tikimybė rasti išėjimą su proporcingai vienodomis pertvarų reikšmėmis. Analizuojant kiek programa atliko žingsnių, galima teigti, kad programa dirba trumpiausiai su dideliu pertvaru skaičiumi – (N*M). Šiek tiek ilgiau užtrunka kai pertvarų skaičius mažas (N+M), o ilgiausiai programa dirba su maždaug (N*M) / 2 pertvarų skaičiui.

Norint programa paliesti per exe failą, galima nurodyti 7 sveikuosius skaičius, kurios reikšmės reiškia – eilučių skaičius, stulpelių skaičius, pertvarų skaičius, pradinė pozicija eilutėje, pradinė pozicija stulpelyje, finišo pozicija eilutėje, finišo pozicija stulpelyje. Nenurodžius jokiu parametrų, numatytos reikšmės yra: 5, 5, 10, 0, 0, 4, 4.

Literatūra:

1. V. Dičiūnas, Algoritmų Analizės Pagrindai, 2005, skyrelis 3.3.
2. https://www.cs.bu.edu/teaching/alg/maze/

Programos kodas:

```
public class Main {

    public static void main(String[] args) {
        int m = 5;
        int n = 5;
        int partitions = 10;
        int startRow = 0;
        int startCol = 0;
        int finishRow = m-1;
        int finishCol = n-1;

        if (args.length == 7) {
            m = Integer.parseInt(args[0]);
            n = Integer.parseInt(args[1]);
            partitions = Integer.parseInt(args[2]);
            startRow = Integer.parseInt(args[3]);
            startCol = Integer.parseInt(args[4]);
            finishRow = Integer.parseInt(args[5]);
            finishCol = Integer.parseInt(args[6]);
        }

        Maze maze = new Maze(m, n, partitions);
        maze.findPath(startRow, startCol, finishRow, finishCol);
    }
}
```

```
/**
 * This class defines maze.
 * "0" value means clear position. "1" value means that there is an obstacle.
 * "*" means path. "!" means dead end.
 * "S" value means maze start location. "F" value means maze finish position.
*/
import java.util.Arrays;
import java.util.Random;

public class Maze {
   private final int row;
   private final int col;
   private int finishRow;
   private int finishCol;
   public int[] randomNumbersArray;
   /**
    * hmatrix defines where is horizontal partitions in maze
    */
   private final Matrix hmatrix;
   /**
    * vmatrix defines where is vertical partitions in maze
    */
   private final Matrix vmatrix;

   private final Matrix pathmatrix;
   public int steps = 0;

   public Maze(int m, int n, int partitions) {
      this.row = m;
      this.col = n;
      hmatrix = new Matrix(row, col, "Horizontal");
      hmatrix.init(randPosForHMatrix(partitions / 2));

      vmatrix = new Matrix(row, col, "Vertical");
      vmatrix.init(randPosForVMatrix(partitions / 2  + (partitions % 2)));

      pathmatrix = new Matrix(row, col, "Path");
      pathmatrix.init(new int[0]);
   }

   boolean trace = true; // print matrix
   boolean showSteps = false; // print x and y positions in matrix

   /**
    * @param x start position index in row
    * @param y start position index in column
    * @param finishRow finish position index in row
    * @param finishCol finish position index in column
    * @return true if path found, false if not
    */
   public boolean findPath(int x, int y, int finishRow, int finishCol) {
      this.finishRow = finishRow;
```

```
    this.finishCol = finishCol;
    if (step(x,y,hmatrix)) {
        pathmatrix.data[x][y] = "S"; // solved, start position
        if (trace) {
            print(hmatrix);
            print(vmatrix);
            System.out.println("Solved!");
            print(pathmatrix);
        }
        return true;
    }
    else {
        if (trace) {
            print(hmatrix);
            print(vmatrix);
            System.out.println("Path not found!");
            print(pathmatrix);
        }
        return false;
    }
}

private boolean step(int x, int y, Matrix matrix) {
    ++steps;

    if (showSteps)
        System.out.println("x=" + x + " y=" + y);

    if ((y == finishCol) && (x == finishRow)) { // found exit
        pathmatrix.data[x][y] = "F"; // finish position
        return true;
    }

    if ( (x == row) || (y == col) || (x < 0) || (y < 0) || ("1".equals(matrix.data[x][y]))
            || ("*".equals(pathmatrix.data[x][y])) || ("*".equals(pathmatrix.data[x][y])) ){
        return false;
    }

    // mark location as part of path
    pathmatrix.data[x][y] = "*";

    boolean result;

    // try to go right
    result = step(x, y+1, vmatrix);
    if (result)
        return true;

    // try to go down
    result = step(x+1, y, hmatrix);
    if (result)
        return true;
```

```
    // try to go left
    result = step(x, y-1, vmatrix);
    if (result)
        return true;

    // try to go up
    result = step(x-1, y, hmatrix);
    if (result)
        return true;

    // mark location as dead end
    pathmatrix.data[x][y] = "!";

    return false;
}

/**
 * returns array with different numbers in range m..(n-1)*m
 * Amount of numbers is given by parameter partitions
 */
private int[] randPosForHMatrix(int partitions) {

    randomNumbersArray = new int[partitions];
    Arrays.fill(randomNumbersArray, -1); // eliminate zeros from array

    int numbersGenerated = 0;

    while(numbersGenerated != partitions) {

        Random rand = new Random();

        int number = rand.nextInt((row)*(col));

        if ((number >= col) && (number < col*(row-1))
                && !numberIsInArray(number, randomNumbersArray)){

            randomNumbersArray[numbersGenerated] = number;
            ++numbersGenerated;
        }
    }
    return randomNumbersArray;
}

/**
* returns array with different numbers in range 0..n*m,
* excluding numbers which are in first and last columns positions
* Amount of numbers is given by parameter partitions
*/
private int[] randPosForVMatrix(int partitions) {

    randomNumbersArray = new int[partitions];
```

```
        int numbersGenerated = 0;

        while(numbersGenerated != partitions) {

            Random rand = new Random();

            int number = rand.nextInt(row*col);

            if (((number % row) != 0) && (((number - (col-1)) % col) != 0)
                    && !numberIsInArray(number, randomNumbersArray)){

                randomNumbersArray[numbersGenerated] = number;
                ++numbersGenerated;
            }
        }
        return randomNumbersArray;
    }

    /**
     * returns true if number is in array
    */
    private boolean numberIsInArray(int number, int[] array) {

        for (int i = 0; i < array.length; ++i) {
            if (array[i] == number) {
                return true;
            }
        }
        return false;
    }

    private void print(Matrix matrix) {
        System.out.println(matrix.name);
        for (int i = 0; i < row; ++i) {

            for (int j = 0; j < col; ++j) {

                System.out.print(matrix.data[i][j] + " ");
            }
            System.out.println("");
        }
        System.out.println("");
    }
}
```

```
public class Matrix {
    private final int rows;
    private final int columns;
    public final String data[][];
    public String name;

    public Matrix(int m, int n, String name) {
        this.rows = m;
        this.columns = n;
        data = new String[m][n];
        this.name = name;
    }
    /**
     * initializing matrix
    */
    public void init(int[] randomNumbersArray) {
        int position = 0;

        for (int i = 0; i < rows; ++i) {

            for (int j = 0; j < columns; ++j) {

                if (numberIsInArray(position, randomNumbersArray)) {
                    data[i][j] = "1";
                }
                else {
                    data[i][j] = "0";
                }
                ++position;
            }
        }
    }
    /**
     * @return is @param number is in @param array
     */
    private boolean numberIsInArray(int number, int[] array) {

        for (int i = 0; i < array.length; ++i) {
            if (array[i] == number) {
                return true;
            }
        }
        return false;
    }
}
```

```
public class Test {
   private int m = 11; // number of rows in maze
   private int n = 11; // number of columns in maze
   int partitions = 121;

   private int startRow = 0;
   private int startCol = 0;
   private int endRow = m-1;
   private int endCol = n-1;
   private Maze maze;

   public void test() {

      int testTimes = 100;
      int counter = 0;
      int steps = 0;

      for (int i = 0; i < testTimes; ++i) {

         maze = new Maze(m, n, partitions);

         if (maze.findPath(startRow, startRow, endRow, endCol))
            ++counter;
         steps += maze.steps;
      }
      System.out.println("Counter = " + counter);
      System.out.println("Average steps = " + steps/100);
   }
}
```