Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4

Predmet / Subject

– Umelá inteligencia / Artificial intelligence –

- Dokumentácia / Documentation -

Zadanie č.1

Ak. Rok / Academic term: 2022/2023, zimný semester

Cvičiaci / Instructors:
Ing. Martin Komák, PhD.

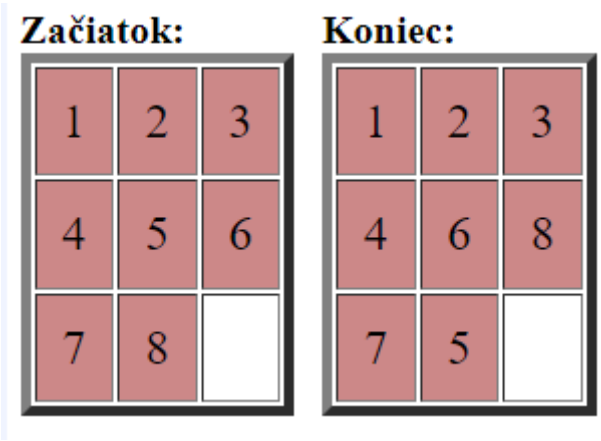
Študent / Student:
Adam Grík

Bratislava, 2022.

Obsah / Content:

1 Úvod do problematiky

Mojou úlohou v zadaní č. 1 bolo nájsť riešenie 8-hlavolamu. Hlavolam je zložený z ôsmich očíslovaných políčok 1-8, a z jedného prázdneho políčka (v mojom prípade som medzeru nahradil znakom *). Políčka môžeme posúvať, hore, dole, vľavo a vpravo, ale len ak je tým smerom medzera, inými slovami políčko s medzerou možeme zamieňať za čísla od neho vpravo, vľavo, hore alebo dole. Na začiatku je vždy daná nejaká východisková pozícia a nejaká cieľová pozícia. Cieľom zadania je nájsť postupnosť krokov, ktoré vedú zo začiatočnej pozície do koncovej.



Pozn.: Obrázok prevzatý z: http://www2.fiit.stuba.sk/~kapustik/z2d.html

Stav: Stav predstavuje aktuálne rozloženie políčok. Môže byť reprezentovaný jednorozmerným poľom alebo ako napíklad v mojom riešení je reprezentovaný dvojrozmerným poľom.

Operátor: Operátor má za úlohu vrátiť nový stav. Sú štyri: vpravo, vľavo, dole a hore.

Heuristická funkcia: Algoritmus, ktorý som implementoval v zadaní č.1 potrebuje dodatočnú informáciu o danom probléme, konkrétne odhad vzdialenosti od cieľového stavu, a to sa nazýva heuristická funkcia. Poznáme niekoľko typov heuristických funkcií v mojom riešení som použil prvé dva typy:

1. Počet políčok, ktoré nie sú na svojom mieste .
2. Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície.

Uzol: V riešenom algoritme potrebujeme vytvoriť stromovú štruktúru riešenia, a to vytvoríme tak, že jednotlivé stavy nahradíme uzlami. Uzol obsahuje rôzne informácie, v mojom riešení uzol obsahuje tieto informácie:

1. Stav, dvojrozmerné pole, ktoré uzol reprezentuje
2. Odkaz na predchodcu
3. Odkaz na potomkov (vpravo, vľavo, hore, dole)
4. Informáciu o pohybe vďaka ktorému uzol vznikol
5. Hodnotu heuristickej funkcie

2 Algoritmus lačného hľadania

Na riešenie danej problematiky som použil algoritmus lačného hľadania, kde som následne porovnával výsledky heuristík 1. a 2.

Hlavnou pointou lačného hľadania je stratégia najskôr najlepší. Pri tejto stratégií sa minimalizuje odhadovaná cena dosiahnutia cieľa. Vyberie sa uzol, ktorý je najbližšie k cieľu. Je to uzol, ktorý reprezentuje stav, ktorý je podľa odhadu najbližší k cieľovému stavu, ale stále hovoríme len o odhade, pretože cena dosiahnutia cieľa z daného stavu sa nedá vždy určiť presne.

Heuristickú funkciu využijeme práve na vypočítanie odhadu ceny cesty z daného uzla do cieľového uzla.

Lačné hľadanie je veľmi podobné hľadaniu do hĺbky. Uprednostňuje postup po jednej ceste až do cieľa. Lačné hľadanie nie je úplne a neoptimalizuje riešenie, pretože ak sa hľadanie rozbehne smerom, ktorým vedie nekonečná cesta, nikdy to nemusí rozpoznať a nevráti sa z nej skúsiť iné cesty, tým pádom sa nám algoritmus zacyklí. Neoptimalizuje preto, že môže nájsť najskôr riešenie, ktoré je horšie než najlepšie.

Časová zložitosť: $O(b^m)$, b je faktor vetvenia, m je maximálna hĺbka priestoru hľadania
Pamäťová zložitosť: $O(b^m)$

3 Stručný opis riešenia

Riešnie tohto zadania som implementoval v jazyku Java.

```
public class Tree {
    private Node root = null;
    private Node end = null;

    private static class Node{
        String[][] value;
        Node left;
        Node right;
        Node up;
        Node down;
        Node parent;
        String move;
        int heuristicValue;

        public Node(String[][] value, Node left, Node right, Node up, Node down, Node parent) {
            this.value = value;
            this.left = left;
            this.right = right;
            this.up = up;
            this.down = down;
            this.parent = parent;
        }
    }
```

Ako prvé som si zadefinoval hlavú triedu Tree (ktorá reprezentuje danú stromovú štruktúru), následne som si zadefinoval triedu Node (ktoré reprezentuje jednotlivé uzly).

```
// method for move up
public static String[][] moveUp(String[][] array) {
    String[][] copy = copy(array);
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if(array[i][j].equals("*")) {
                if(i != 0) {
                    String tmp = copy[i][j];
                    copy[i][j] = copy[i-1][j];
                    copy[i-1][j] = tmp;
                    return copy;
                }
            }
        }
    }
    return null;
}
```

Ako druhé som si vytvoril jednotlivé metódy na posúvanie prázdneho políčka (moveUp, moveDown, moveRight a moveLeft), kde som vďaka cyklu for vymieňal jednotlivé políčka, tak aby vznikol daný pohyb, návratom tejto funkcie bolo pole ktoré vzniklo po danom pohybe.

```
// method for create a first heuristic
public static int firstHeuristic(String[][] node, String[][] endPosition) {
    int value = 0;
    String[] first = new String[9];
    String[] second = new String[9];
    int pom = 0;
    // for loop across two new arrays, and copy values from actual node and end position
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            first[pom] = node[i][j];
            second[pom] = endPosition[i][j];
            pom++;
        }
    }
    // compare first and second array and compare if its value are not equals
    for(int i = 0; i < 9; i++) {
        if(!first[i].equals("*")) {
            if (!first[i].equals(second[i])) {
                value++;
            }
        }
    }
    return value;
}
```

Metóda na zistenie heuristickej hodnoty č. 1. Do tejto metódy som si poslal hodnoty aktuálneho uzla a hodnoty koncového uzla. Potom som si vytvoril dve jednorozmerné polia, do ktorých som kopíroval hodnoty aktuálneho uzla a koncového uzla. Následne som tieto

dve novovytvorené polia porovnával v cykle for, a ak sa hodnoty nerovnali, tak som zvýšil hodnotu heuristiky, ktorá bola nakonci návratovou hodnotou.

```java
// method for create a second heuristic
public static int secondHeuristic(String[][] node, String[][] endPosition) {
    int pom = 1;
    int value = 0;
    while(pom < 9) {
        for(int i = 0; i < 3; i++) {
            for(int j = 0; j < 3; j++) {
                if (!node[i][j].equals("*")) {
                    if (Integer.parseInt(node[i][j]) == pom) {
                        for (int k = 0; k < 3; k++) {
                            for (int l = 0; l < 3; l++) {
                                if (!endPosition[k][l].equals("*")) {
                                    if (Integer.parseInt(endPosition[k][l]) == Integer.parseInt(node[i][j])) {
                                        int x = Math.abs(k - i);
                                        int y = Math.abs(l - j);
                                        value = value + (x + y);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
        pom++;
    }
    return value;
}
```

Metóda na zistenie heuristickej hodnoty č. 2. V tejto funkcii som vo viacerých for cykloch prechádzal najprv pole aktuálneho uzla a potom pole koncového uzla. Našiel som si jednotlivé hodnoty čiže 1-8. Následne som si vypočítal absolútnu hodnotu, rozdielu ich jednotlivých pozícií. Potom som tieto hodnoty pripočítaval do premennej value, ktorá reprezentovala hodnotu heuristickej funkcie č. 2, ktorú som následne posielal ako návratovú hodnotu.

```java
// method for do work in first heuristic
public void doAllStuff(Node node, Node end, HashSet<List> set, int createdNodes, int typeOfHeuristic) {
    // initialize int arraylist for first heuristic
    ArrayList<Integer> heuristicArrayList = new ArrayList<Integer>();

    // left move
    String[][] left = moveLeft(node.value);
    if(left != null) {
        if(set.add(twoDArrayToList(left))) {
            node.left = new Node(left, left: null, right: null, up: null, down: null, node);
            node.left.move = "left";
//              System.out.println("left");
//              HelpClass.printArray2D(node.left.value);
            if(typeOfHeuristic == 1) {
                node.left.heuristicValue = firstHeuristic(node.left.value, end.value);
            }
            else {
                node.left.heuristicValue = secondHeuristic(node.left.value, end.value);
            }
//              System.out.println(node.left.firstHeuristic);
            heuristicArrayList.add(node.left.heuristicValue);
            createdNodes++;
        }
    }
    else {
        node.left = null;
    }
```

Jadro celého môjho algoritmu prebieha vo funkcii doAllStuff(), ktorá je rozdelená na 4 časti podľa pohybov. Ja vysvetlím funkciu len na jednom pohybe, pretože ostatné pohyby sú založené na presne tom istom postupe.
Ako prvé si vytvorím nové dvojrozmerné pole vďaka funkcií moveLeft(), kde posielam hodnotu aktuálneho uzla. Následne v prvej podmienke zistím či bolo vytvorené, pretože nemusí byť vždy vytvorené, ak sa prázdne políčko nachádza úplne vľavo tak sa nemôže posunúť doľava. V ďalšej podmienke kontrolujem či už daný uzol nebol vytvorený. Ak sú tieto dve podmienky v poriadku, tak vytvorím nového potomka aktuálneho uzla (node.left), následne zisťujem hodnoty heuristických funkcií, kde následne túto hodnotu posielam do ArrayListu heuristických hodnôt. Takýmto princípom vykonám všetky štyri pohyby.

```java
//      System.out.println("Heuristic arraylist " + heuristicArrayList);
    Node bestNodeFirstHeuristic = returnBestNode(heuristicArrayList, node.left, node.up, node.right, node.down);
//      System.out.println("Best node in first heuristic: ");
    assert bestNodeFirstHeuristic != null;
    System.out.println(bestNodeFirstHeuristic.move);
    HelpClass.printArray2D(bestNodeFirstHeuristic.value);
    System.out.println("------------------");

    // if actual best node not equals end node, do this function recursively
    if(!twoDArrayToList(bestNodeFirstHeuristic.value).equals(twoDArrayToList(end.value))) {
        doAllStuff(bestNodeFirstHeuristic, end, set, createdNodes, typeOfHeuristic);
    }
    else{
        System.out.println("Number of created nodes: " + createdNodes);
    }

}
```

Následne po všetkých daných pohyboch, pokračujem ešte vo funkcií doAllStuff(), kde ešte hľadám uzol s najmenšou heuristickou hodnotou vďaka funkcií returnBestNode().

A nakoniec porovnám či sa uzol s najlepšiou heuristickou hodnotou nerovná koncovému uzlu, ak sa nerovná opakujem funckiu rekurzívne, ak sa rovná skončím funkciu, čiže aj celý program a vypíšem počet vytvorených uzlov.

4 Testovanie

V mojom testovaní som porovnával hlavne výsledky heuristík č.1 a č.2. Porovnával som ich časovú náročnosť a počet vytvorených uzlov na dosiahnutie úspešného ukončenia programu.

	ZAČIATOK	KONIEC	ČAS (milisekudny)	POČET VYTVORENÝCH UZLOV
Heuristika č.1	283 164 7*5	123 8*4 765	22.0	20
Heuristika č.2	283 164 7*5	123 8*4 765	22.0	12

	ZAČIATOK	KONIEC	ČAS (milisekundy)	POČET VYTVORENÝCH UZLOV
Heuristika č.1	*64 573 281	526 438 7*1	29.0	268
Heuristika č.2	*64 573 281	526 438 7*1	24.0	146

	ZAČIATOK	KONIEC	ČAS (milisekundy)	POČET VYTVORENÝCH UZLOV
Heuristika č.1	35* 428 167	173 485 *26	30.0	284
Heuristika č.2	35* 428 167	173 485 *26	27.0	176

	ZAČIATOK	KONIEC	ČAS (milisekundy)	POČET VYTVORENÝCH UZLOV
Heuristika č.1	638 *71 245	283 571 *64	22.0	164
Heuristika č.2	638 *71 245	283 571 *64	33.0	276

V prvom type testovania som testoval stredne náročnú obtiažnosť vstupu. Môžeme vidieť že výsledky ukazujú, že heuristika č. 2 je vo väčšine z testovaných prípadov efektívnejšie po oboch smeroch oproti heuristike č.1.

	ZAČIATOK	KONIEC	ČAS (milisekudny)	POČET VYTVORENÝCH UZLOV
Heuristika č.1	123 456 78*	123 468 75*	9.0	10
Heuristika č.2	123 456 78*	123 468 75*	8.0	10

	ZAČIATOK	KONIEC	ČAS (milisekudny)	POČET VYTVORENÝCH UZLOV
Heuristika č.1	123 56* 784	123 586 *74	7.0	9
Heuristika č.2	123 56* 784	123 586 *74	7.0	9

Pri nízkej náročnosti obtiažnosti vstupu, môžeme vidieť že výsledky oboch heuristík sú viacmenej totožné.

	ZAČIATOK	KONIEC	ČAS (milisekudny)	POČET VYTVORENÝCH UZLOV
Heuristika č.1	863 124 5*7	3*5 847 216	45.0	701
Heuristika č.2	863 124 5*7	3*5 847 216	-	-

	ZAČIATOK	KONIEC	ČAS (milisekudny)	POČET VYTVORENÝCH UZLOV
Heuristika č.1	462 51* 783	36* 827 415	-	-
Heuristika č.2	462 51* 783	36* 827 415	35.0	336

Ako môžeme vidieť na obrázku vyššie, pri testovaní vznikli aj prípady také, že jedna heuristika riešenie našla, ale druhá nie pretože sa program dostal do nekonečnej vetvy.

5 Záver

Lačné hľadanie je z pohľadu výkonnosti pomerne efektívne, avšak iba v prípade že vie nájsť tú správnu cestu, pretože lačné hľadanie nie je úplné. To znamená, že nie vždy nájde toto hľadanie riešenie.

Prvý prípad neúspešnosti môže nastať ak sa hľadanie vyberie smerom ktorým vedie nekonečná cesta, nikdy to nemusí rozoznať, nevráti sa z nej skúsiť iné smery, čo znamená, že celý program sa zacyklí.

Druhý prípad neúspešnosti môže nastať vtedy, ak si ukladáme všetky už vytvorené uzly, a nevytvárame duplikáty týchto uzlov. Tým pádom sa program môže dostať do takého stavu, že príde uzol, pri ktorom po všetkých daných pohyboch, vzniknú také uzly ktoré už boli raz vytvorené, čo bude znamenať že program nevytvorí žiaden ďalší nasledujúci uzol, tým pádom skončí bez riešenia.

Ak však už lačné hľadanie nájde riešenie, tak je vo väčšine prípadoch, vďaka použitiu heuristickej funkcie oveľa efektívnejšie ako napríklad klasické hľadanie do hĺbky alebo šírky.