Oleksandr Oksanich, AIS ID: 122480

Umelá inteligencia

Zadanie č. 1 — Prehľadávanie stavového priestoru (problém 2)

Obsah

Riešený problem

Úlohou je nájsť riešenie 8-hlavolamu. Hlavolam je zložený z 8 očíslovaných políčok a jedného prázdneho miesta. Políčka je možné presúvať hore, dole, vľavo alebo vpravo, ale len ak je tým smerom medzera. Je vždy daná nejaká východisková a nejaká cieľová pozícia a je potrebné nájsť postupnosť krokov, ktoré vedú z jednej pozície do druhej.

Príkladom môže byť nasledovná začiatočná a koncová pozícia:

1	2	3
4	5	6
7	8	

6	4	7
8	5	
3	2	1

Na riešenie problému treba použiť algoritmus lačného hľadania a porovnať výsledky dvoch heuristík.

Opis riešenia

Pre túto implementáciu som použil programovací jazyk Java (verzie 17) a vývojové prostredie IntelliJ IDEA (2023.3 EAP).

Oleksandr Oksanich, AIS ID: 122480

Reprezentácia údajov

Problém je reprezentovaný ako mriežka 3x3 (resp. MxN, keďže moja implementácia umožňuje uviesť inú veľkosť hlavolamu ako 3x3), kde každé políčko obsahuje jedinečné číslo 1 až 8 (resp. 1 až M*N – 1) alebo prázdne miesto označené 0. Mriežka je reprezentovaná ako 2D pole.

Okrem toho kvôli väčšej miere abstrakcie som vytvoril triedu State, ktorá sa používa na kontrolu vstupných údajov, kopírovanie stavov a ich výpis.

Heuristické funkcie

Moja implementácia umožňuje vybrať jednú z dvoch nasledujúcich heuristických funkcií:

1. Počet políčok, ktoré nie sú na svojom mieste
2. Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície (*Manhattan distance*)

```
protected int cellsOutOfPlace(MNPuzzle.State currentState) {
    var wrongPositionCount = 0;

    for (int i = 0; i < puzzle.rows(); i++) {
        for (int j = 0; j < puzzle.columns(); j++) {
            if (currentState.toArray()[i][j] != 0
                    && currentState.toArray()[i][j] != targetState.toArray()[i][j])
{
                wrongPositionCount++;
            }
        }
    }

    return wrongPositionCount;
}


protected int totalDistance(MNPuzzle.State currentState) {
    var totalDistance = 0;

    var currentCoordsByValue = new int[puzzle.rows() * puzzle.columns()][2];
    var targetCoordsByValue = new int[puzzle.rows() * puzzle.columns()][2];

    for (int i = 0; i < puzzle.rows(); i++) {
        for (int j = 0; j < puzzle.columns(); j++) {
            currentCoordsByValue[currentState.toArray()[i][j]] = new int[] {i, j};
            targetCoordsByValue[targetState.toArray()[i][j]] = new int[] {i, j};
        }
    }

    // The loop starts with 1 in order to exclude the empty cell
    for (int i = 1; i < puzzle.rows() * puzzle.columns(); i++) {
        var currentCoords = currentCoordsByValue[i];
        var targetCoords = targetCoordsByValue[i];

        for (int j = 0; j < 2; j++) {
            totalDistance += Math.abs(targetCoords[j] - currentCoords[j]);
        }
```

```
    }

    return totalDistance;
}
```

Štruktúra uzla

Každý uzol v tejto implementácii obsahuje nasledujúce údaje:

- Aktuálny stav
- Hodnota heuristickej funkcie
- Odkaz na predchodcu
- Posledný použitý operator

Aktuálny stav a hodnota heuristickej funkcie sú použité na priame porovnanie stavov počas riešenia problému (čím bližšie hodnota je k 0, tým bližšie aktuálny stav je k cieľovému).

Odkaz na predchodcu a posledný operátor sú reprezentačné údaje, ktoré sa používajú na finálny výpis celej cesty riešenia.

```
public record Node(
        MNPuzzle.State currentState,
        int heuristicValue,
        Node parent,
        Operator lastOperator
) {}
```

```
public enum Operator {
    UP(-1, 0),
    DOWN(1, 0),
    LEFT(0, -1),
    RIGHT(0, 1);

    private final int x;
    private final int y;

    Operator(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // ...
}
```

Použitý algoritmus

Na riešenie problému sa používa algoritmus lačného hľadania (čiže *greedy algorithm*). V každom kroku algoritmus vyhodnocuje možné ťahy na základe heuristickej funkcie a vyberá ťah, ktorý sa javí ako približujúci hlavolam k cieľovému stavu.

Oleksandr Oksanich, AIS ID: 122480

Na tento účel moja implementácia používa dátovú štruktúru PriorityQueue zo štandardnej knižnice Java, ktorá umiestni uzol s najnižšou heuristickou hodnotou na začiatok. Použitie prioritného radu pomáha efektívne vybrať ďalší stav.

Riešenie sa hľadá iteratívne, kým sa nenájde uzol s heuristickou hodnotou 0 (problém je vyriešený) alebo rad nezostane prázdny (riešenie nie je). Počas každej iterácie prázdne políčko sa presunie všetkými možnými smermi (teda sa aplikujú všetky možné operátory) a všetky nové stavy, ktoré sa predtým nepoužili (porovnávajú sa reťazce reprezentujúce stavy kvôli optimalizácii), sa pridajú do radu. Už použité stavy sa do radu znova nepridávajú, aby sa zabránilo nekonečným cyklom a optimalizoval sa výkon algoritmu.

```
public Result solve(HeuristicFunction heuristics) {
    var iterationCount = 0;
    var nodeCount = 0;

    var now = System.nanoTime();

    var usedStates = new HashSet<String>();

    var currentNode = new Node(
            initialState,
            Objects.requireNonNull(heuristics) ==
HeuristicFunction.OUT_OF_PLACE_COUNT ? cellsOutOfPlace(initialState) :
totalDistance(initialState),
            null,
            null
    );
    var nodeQueue = new
PriorityQueue<>(Comparator.comparingInt(Node::heuristicValue));

    nodeQueue.add(currentNode);

    nodeCount++;

    while (!nodeQueue.isEmpty()) {
        iterationCount++;

        currentNode = nodeQueue.poll();

        if (currentNode.heuristicValue() == 0 || (iterationLimit > 0 &&
iterationCount >= iterationLimit)) {
            break;
        }

        usedStates.add(currentNode.currentState().toString());

        var emptyCoords = getEmptyCellIndex(currentNode.currentState());
        var r = emptyCoords[0];
        var c = emptyCoords[1];

        // The empty cell can be moved up
        if (r > 0) {
            var stateCopy = currentNode.currentState().copy();
```

```
            move(stateCopy, r, c, Operator.UP);

            var node = new Node(
                    stateCopy,
                    heuristics == HeuristicFunction.OUT_OF_PLACE_COUNT ?
cellsOutOfPlace(stateCopy) : totalDistance(stateCopy),
                    currentNode,
                    Operator.UP
            );

            if (!usedStates.contains(node.currentState().toString())) {
                nodeQueue.add(node);
                nodeCount++;
            }
        }

        // The empty cell can be moved down
        if (r < puzzle.rows() - 1) {
            var stateCopy = currentNode.currentState().copy();

            move(stateCopy, r, c, Operator.DOWN);

            var node = new Node(
                    stateCopy,
                    heuristics == HeuristicFunction.OUT_OF_PLACE_COUNT ?
cellsOutOfPlace(stateCopy) : totalDistance(stateCopy),
                    currentNode,
                    Operator.DOWN
            );

            if (!usedStates.contains(node.currentState().toString())) {
                nodeQueue.add(node);
                nodeCount++;
            }
        }

        // The empty cell can be moved left
        if (c > 0) {
            var stateCopy = currentNode.currentState().copy();

            move(stateCopy, r, c, Operator.LEFT);

            var node = new Node(
                    stateCopy,
                    heuristics == HeuristicFunction.OUT_OF_PLACE_COUNT ?
cellsOutOfPlace(stateCopy) : totalDistance(stateCopy),
                    currentNode,
                    Operator.LEFT
            );

            if (!usedStates.contains(node.currentState().toString())) {
```

```
                nodeQueue.add(node);
                nodeCount++;
            }
        }

        // The empty cell can be moved right
        if (c < puzzle.columns() - 1) {
            var stateCopy = currentNode.currentState().copy();

            move(stateCopy, r, c, Operator.RIGHT);

            var node = new Node(
                    stateCopy,
                    heuristics == HeuristicFunction.OUT_OF_PLACE_COUNT ?
cellsOutOfPlace(stateCopy) : totalDistance(stateCopy),
                    currentNode,
                    Operator.RIGHT
            );

            if (!usedStates.contains(node.currentState().toString())) {
                nodeQueue.add(node);
                nodeCount++;
            }
        }
    }

    now = System.nanoTime() - now;

    return new Result(
            currentNode.currentState().equals(targetState),
            iterationCount,
            nodeCount,
            now,
            currentNode
    );
}
```

Spôsob testovania a zhodnotenie riešenia

Testovacie príklady

Na testovanie implementácie daného algoritmu som vytvoril sadu príkladov rôznej veľkosti a zložitosti, ktorú som uložil ako textový súbor medzi projektových súborov:

```
// rows|columns : initial_state : target_state
/// 3x3
3|3 : 3|8|7|2|6|4|0|5|1 : 0|3|7|2|8|1|5|4|6 // 1
3|3 : 4|2|7|1|0|8|3|6|5 : 4|7|8|2|0|5|1|3|6 // 2
3|3 : 2|6|7|4|8|3|5|1|0 : 6|0|7|2|4|8|5|1|3 // 3
3|3 : 8|1|3|2|5|7|0|4|6 : 8|3|7|4|2|1|0|5|6 // 4
3|3 : 2|4|7|1|0|5|8|6|3 : 4|7|5|2|6|0|1|8|3 // 5
3|3 : 2|1|4|6|7|0|3|5|8 : 1|4|7|2|0|6|3|5|8 // 6
```

Oleksandr Oksanich, AIS ID: 122480

```
3|3 : 4|1|6|8|5|7|3|2|0 : 8|4|1|5|2|6|3|0|7 // 7
3|3 : 1|8|2|0|4|3|7|6|5 : 1|2|3|4|5|6|7|8|0 // 8
3|3 : 4|0|3|7|2|1|5|8|6 : 2|0|3|4|1|6|7|5|8 // 9
3|3 : 2|1|0|5|4|3|8|7|6 : 4|2|3|5|1|6|0|8|7 // 10

/// 3x4
3|4 : 7|10|1|8|3|11|4|0|6|5|9|2 : 10|1|11|8|0|5|9|4|7|3|6|2 // 11

/// 4x3
4|3 : 6|3|4|9|0|7|5|2|10|1|8|11 : 1|2|3|4|5|6|7|8|9|10|11|0 // 12

/// 4x4
4|4 : 11|8|9|1|10|0|7|6|15|13|4|5|14|3|2|12 : 0|8|9|1|11|13|7|6|10|4|5|12|15|14|3|2 // 13
4|4 : 15|12|10|1|5|2|4|6|8|7|9|0|14|13|11|3 : 12|10|4|1|15|2|7|6|14|5|0|9|13|11|8|3 // 14
4|4 : 2|0|8|3|1|5|7|4|9|6|10|11|13|14|15|12 : 1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|0 // 15
```

Reprezentácia výsledkov

Na reprezentáciu výsledkov som vytvoril triedu Result, ktorá obsahuje nasledujúce údaje:

- či bol hlavolam vyriešený
- počet všetkých iterácií
- počet všetkých uzlov
- čas riešenia v nanosekundách
- finálny uzol s odkazom na všetkých predchodcov

```java
public record Result(
        boolean isSolved,
        int iterationCount,
        int nodeCount,
        long timeNs,
        Node finalNode
) {
    // ...
}
```

Tieto údaje som použil na porovnanie efektivity dvoch heuristických funkcií.

Príklad výstupu program

Uvádzam príklad výstupu pre 1. testovací príklad a 2. heuristiku:

```
Result[isSolved=true, iterationCount=11, nodeCount=21, timeNs=1571000,
finalNode=Node[currentState=0|3|7|2|8|1|5|4|6, heuristicValue=0, parent=…]]

3|8|7

2|6|4

0|5|1

|

RIGHT

|
```

3|8|7

2|6|4

5|0|1

|

UP

|

3|8|7

2|0|4

5|6|1

|

RIGHT

|

3|8|7

2|4|0

5|6|1

|

DOWN

|

3|8|7

2|4|1

5|6|0

|

LEFT

|

3|8|7

2|4|1

5|0|6

|

UP

|

3|8|7

2|0|1

5|4|6

Oleksandr Oksanich, AIS ID: 122480

```
|
UP
|
3|0|7
2|8|1
5|4|6
|
LEFT
|
0|3|7
2|8|1
5|4|6
```

Výsledky testovania

Príklad a jeho veľkosť	1. heuristika			2. heuristika (Manhattan distance)		
	Počet iterácií	**Počet uzlov**	**Čas, ns**	**Počet iterácií**	**Počet uzlov**	**Čas, ns**
1, 3x3	15	29	9154000	11	21	2206100
2, 3x3	9	16	962200	9	16	938100
3, 3x3	6	11	526200	6	11	521600
4, 3x3	13	25	1102300	9	18	746900
5, 3x3	8	14	444300	8	14	543900
6, 3x3	10	18	605100	8	15	567800
7, 3x3	8	14	626000	8	14	405900
8, 3x3	11	23	646200	10	20	497000
9, 3x3	9	16	427500	9	16	351000
10, 3x3	43	77	2037800	11	19	413200
11, 3x4	181	349	5257300	15	30	429000
12, 4x3	3093	5678	59546700	268	526	4613300
13, 4x4	11	23	1091100	11	23	201900
14, 4x4	469	1053	7773800	16	35	3907500
15, 4x4	536	1109	5786100	21	46	299700
Priemer	294.80	563.67	6399106.67	28.00	54.93	1109526.67

Výsledky pri zámene začiatočného a koncového uzla

Príklad a jeho veľkosť	1. heuristika			2. heuristika (Manhattan distance)		
	Počet iterácií	**Počet uzlov**	**Čas, ns**	**Počet iterácií**	**Počet uzlov**	**Čas, ns**
1, 3x3	17	32	11454300	10	19	1480800
2, 3x3	427	724	23055700	9	16	474700
3, 3x3	6	12	175600	6	12	241700
4, 3x3	86	155	1646900	11	21	362500
5, 3x3	8	13	155000	8	13	178300

6, 3x3	57	100	898900	8	16	260400
7, 3x3	8	15	132400	8	15	217900
8, 3x3	293	484	4563900	12	22	197200
9, 3x3	10	19	186400	9	16	159600
10, 3x3	215	357	3231000	30	53	605200
11, 3x4	542	1015	8267700	15	31	286100
12, 4x3	1700	3210	56771000	509	966	12655000
13, 4x4	11	21	11089800	11	21	2744600
14, 4x4	20	43	236000	133	286	6011800
15, 4x4	113	236	1622500	19	43	406300
Priemer	234.20	429.07	8232473.33	53.20	103.33	1752140.00

Záver

Porovnali sme vlastnosti oboch heuristických funkcií pre rôzne dĺžky riešenia. Počet políčok mimo svojho miesta ako heuristická hodnota môže fungovať dobre pre jednoduché prípady, ale má problémy so zložitejšími hlavolamami. Heuristika *Manhattan distance* je vo všeobecnosti efektívnejšia, najmä pri riešení zložitých hlavolamamov, pretože berie do úvahy relatívnu polohu políčok. Zámena začiatočného a koncového uzla síce ovplyvňuje výsledky testovania (najmä pri 2. heuristike), ale vo všeobecnosti tento trend sa zachováva.

Veľkou výhodou algoritmu lačného hľadania je, že je relatívne jednoduchý na implementáciu a dokáže poskytnúť pomerne rýchle riešenia pre väčšinu prípadov 8-hlavolamov.

Avšak algoritmus nezaručuje optimálne riešenie, pretože sa zameriava na **lokálne** optimálne cesty. Okrem toho kvalita riešenia výrazne závisí od zvolenej heuristickej funkcie.