Цель

Решение задачи поиска в пространстве состояний. Реализовать стратегии информированного (эвристического) поиска. В работе рассматривается решение задачи «Головоломка 8-ка».

Постановка задачи

Рассматривается задача «Головоломка 8-ка».

Задана доска с 8 пронумерованными фишками и с одним пустым участком.

Фишка, смежная с пустым участком, может быть передвинута на этот участок. Требуется достичь указанного целевого состояния.

Начальное состояние:

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

Целевое состояние:

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

Разобрать по шагам (на некоторую глубину) реализацию алгоритма A* с использованием эвристических функций h_1 и h_2.

– h1 – число фишек, стоящих не на своем месте;

– h2 – суммарное по всем фишкам число шагов до целевого положения (манхэттенское расстояние).

Описание выбранных структур данных (классов), представление функции определения последователей

Формализованный обобщенный алгоритм поиска представлен в виде функции GeneralSearch:

```
GeneralSearch (Problem, CheckerCycle, Queuing) {
    nodes;
    startNode = CreateNode(null, problem.initState, 0);
    nodes.add(startNode);
    while (true) {
        if (nodes.isEmpty()) return failure;
        tmpNode = nodes.pop();
        if(problem.goalTest(tmpNode.state))return Solution(tmpNode, startNode);
        nodes.Queuing (nodes,
expend(CheckerCycle ,tmpNode,problem.operations(tmpNode.state))
        );
 }
```

Более подробно обобщенный поиск представлен в лабораторной работе №1. Далее будут описаны доработки, реализующие поиск A*.

Поиск A* разработан на основе обобщенного поиска:

function **ASearch** (problem, **Eval-Fn**)

GeneralSearch (problem, **Queuing-Fn**(Eval-Fn))

Eval-Fn – функция оценки.

function **Eval-Fn** (node1, node)

compare(node1, node2) // сравнивает первый и второй узел в соответствии с заданной функцией оценки.

Quening-Fn – построение очереди вершин, ожидающих раскрытия. Упорядочивает вершины в соответствии с Eval-Fn.

function **Quening-Fn** (nodes, Eval-Fn)

sort(nodes, Eval-Fn) //упорядочивает вершины в соответствии с Eval-Fn

Описание алгоритма

A*-поиск – разновидность **поиска сначала лучший. (Best-First Seach – BFS)**, в котором порядок обхода вершин определяется эвристической функцией f (n) = g(n) + h(n), где h(n) – оценочная стоимость самого дешевого пути из состояния вершины n в целевое состояние (в случае Головоломки «Восьмерки» - это число фишек, стоящих не на своем месте или манхэттенское расстояние), g(h) – минимальная стоимость пути до узла n (в случае Головоломки «Восьмерки» - это глубина от начального состояния до состояния n).

Исходный код на языке программирования Java

```
import java.util.*;

public class GeneralSearch {
    Problem problem;
    CheckerCycle checkerCycle;
    Distance distance;
    public Solution generalSearch(Problem problem, CheckerCycle
checkerCycle, Queuing queuing, Distance distance) {
        this.checkerCycle = checkerCycle;
        this.problem = problem;
        this.distance = distance;
        LinkedList<Node> nodes = new LinkedList<>();
        Node startNode = new Node(null, problem.initState, 0,
problem.goalState, distance);
        nodes.add(startNode);
        while (true) {
            if (nodes.isEmpty()) return null;
            //Uncomment for step-by-step output
            /*Scanner scan=new Scanner(System.in);
            scan.nextLine();
            System.out.println("Fringer: ");
            for (Node node : nodes)
              System.out.println(node.state + ", cost: " +
node.finalCost + "\n");*/

            Node tmpNode = nodes.pop();
            //Uncomment for step-by-step output
            /*System.out.println("Current node: ");
            System.out.println(tmpNode.state);*/
```

```
            if (problem.goalTest(tmpNode.state)) return new
Solution(tmpNode, startNode);
            queuing.setQueue(nodes, expend(tmpNode,
problem.operations(tmpNode.state)));
        }
    }

    public List<Node> expend(Node node, List<State> newStates) {
        List<Node> newNodes = new LinkedList<>();
        for (State state: newStates) {
            Node newNode = new Node(node, state, node.depth + 1,
this.problem.goalState, distance);
            if (checkerCycle.check(newNode, node.parent))
newNodes.add(newNode);
             //Uncomment for step-by-step output
            //else System.out.println("Cycle prevented: \n" +
newNode.state);
        }
         //Uncomment for step-by-step output
        /*System.out.println("Added nodes: ");
        for (Node addedNode : newNodes)
        System.out.println(addedNode.state + "\n");*/

        return newNodes;
    }
}

interface Queuing {
    void setQueue(LinkedList<Node> queue, List<Node> nodes);
}

class DFSQueuing implements Queuing{
    public void setQueue(LinkedList<Node> queue, List<Node>
nodes) {
        for (Node node : nodes) {
            queue.add(0, node);
        }
    }
}

class BFSQueuing implements Queuing{
    public void setQueue(LinkedList<Node> queue, List<Node>
nodes) {
        queue.addAll(nodes);
    }
}

class DFSLimQueuing implements Queuing{
    int depthMax;

    public DFSLimQueuing(int depthMax) {
        this.depthMax = depthMax;
```

```
    }

    public void setQueue(LinkedList<Node> queue, List<Node>
nodes) {
        for (Node node : nodes) {
            if (node.depth < depthMax)
                queue.add(0, node);
            else
                break;
        }
    }
}

class BestQueuing implements Queuing {
    Comparator<Node> evalFunc;

    public BestQueuing(Comparator<Node> evalFunc) {
        this.evalFunc = evalFunc;
    }

    public void setQueue(LinkedList<Node> queue, List<Node>
nodes) {
        queue.addAll(nodes);
        queue.sort(evalFunc);
    }
}

class GreedyComparator implements Comparator<Node> {
    @Override
    public int compare(Node node1, Node node2) {
        if (node1.costToGoal > node2.costToGoal) return 1;
        else if (node1.costToGoal < node2.costToGoal) return -1;
        return 0;
    }
}

class AStarComparator implements Comparator<Node> {
    @Override
    public int compare(Node node1, Node node2) {
        if (node1.finalCost > node2.finalCost) return 1;
        else if (node1.finalCost < node2.finalCost) return -1;
        return 0;
    }
}

class Node {
    Node parent;
    State state;
    int depth;
    int costToGoal;
    int finalCost;

    public Node(Node parent, State state, int depth, State
```

```
goalState, Distance distance) {
        this.parent = parent;
        this.state = state;
        this.depth = depth;
        this.costToGoal = state.distance(goalState, distance);
        this.finalCost = this.depth + this.costToGoal;
    }
}

interface State {
    int distance(State goal, Distance distance);
}

class StatePuzzleEight implements State {
    byte[] positionNumber = new byte[9];
    byte zeroPosition;

    StatePuzzleEight (byte ... positionNumber) {
        for (byte i = 0; i < 9; i++) this.positionNumber[i] =
positionNumber[i];
        for (byte i = 0; i < 9; i++) if (positionNumber[i] == 0)
zeroPosition = i;
    }

    @Override
    public int distance(State goal, Distance distance){
        return distance.distance(this, goal);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;

        StatePuzzleEight state = (StatePuzzleEight) o;

        if (zeroPosition != state.zeroPosition) return false;
        return Arrays.equals(positionNumber,
state.positionNumber);
    }

    @Override
    public int hashCode() {
        int result = Arrays.hashCode(positionNumber);
        result = 31 * result + (int) zeroPosition;
        return result;
    }

    @Override
    public String toString() {
        return positionNumber[0] + " " + positionNumber[1] + " "
+ positionNumber[2] + "\n" +
```

```
                positionNumber[3] + " " + positionNumber[4] + "
" + positionNumber[5] + "\n" +
                positionNumber[6] + " " + positionNumber[7] + "
" + positionNumber[8];
    }
}

interface Distance {
    int distance(State state, State goalState);
}

class WrongPositionDistancePuzzleEight implements Distance {
    public int distance(State state, State goalState) {
        StatePuzzleEight statePuzzleEight = (StatePuzzleEight)
state;
        StatePuzzleEight goalPuzzleState = (StatePuzzleEight)
goalState;
        int count = 0;
        for (byte i = 0; i < 9; i++) {
            if (goalPuzzleState.positionNumber[i] !=
statePuzzleEight.positionNumber[i]) count++;
        }
        return count;
    }
}

class ManhattanDistancePuzzleEight implements Distance{
    public int distance(State state, State goalState) {
        StatePuzzleEight statePuzzleEight = (StatePuzzleEight)
state;
        StatePuzzleEight goalPuzzleState = (StatePuzzleEight)
goalState;
        int count = 0;
        for (byte i = 0; i < 9; i++) {
            int columnDiff =
Math.abs(goalPuzzleState.positionNumber[i]%3 -
statePuzzleEight.positionNumber[i]%3);
            int lineDiff =
Math.abs(goalPuzzleState.positionNumber[i]/3 -
statePuzzleEight.positionNumber[i]/3);
            count += columnDiff + lineDiff;
        }
        return count;
    }
}

abstract class Problem {
    State initState;
    State goalState;

    public Problem(State initState) {
        this.initState = initState;
    }
```

```
    public Problem(State initState, State goalState) {
        this.initState = initState;
        this.goalState = goalState;
    }

    abstract boolean goalTest(State state);

    abstract LinkedList<State> operations(State state);
}

class ProblemPuzzleEight extends Problem {

    public ProblemPuzzleEight(State initState) {
        super(initState);
        this.goalState = new StatePuzzleEight(new byte[] {1, 2,
3, 4, 5, 6, 7, 8, 0});
    }

    public ProblemPuzzleEight(State initState, State goalState)
{
        super(initState, goalState);
    }

    public boolean goalTest(State state) {
        return state.equals(this.goalState);
    }

    @Override
    public LinkedList<State> operations(State state) {
        StatePuzzleEight statePuzzleEight = (StatePuzzleEight)
state;
        byte zeroPosition = statePuzzleEight.zeroPosition;
        byte[] positionNumber = statePuzzleEight.positionNumber;

        LinkedList<State> newStates = new LinkedList<>();

        //swap zero and left number
        if (zeroPosition % 3 != 0)
            newStates.add(new
StatePuzzleEight(swap(positionNumber.clone(), zeroPosition,
(byte) (zeroPosition - 1))));

        if (zeroPosition % 3 != 2)
            newStates.add(new
StatePuzzleEight(swap(positionNumber.clone(), zeroPosition,
(byte) (zeroPosition + 1))));

        //swap zero and top number
        if (zeroPosition / 3 != 0)
            newStates.add(new
StatePuzzleEight(swap(positionNumber.clone(), zeroPosition,
(byte) (zeroPosition - 3))));
```

```
        //swap zero and bottom number
        if (zeroPosition / 3 != 2)
            newStates.add(new
StatePuzzleEight(swap(positionNumber.clone(), zeroPosition,
(byte) (zeroPosition + 3))));

        return newStates;
    }

    public byte[] swap(byte[] positionNumber, byte
firstPosition, byte secondPosition) {
        byte tmp = positionNumber[firstPosition];
        positionNumber[firstPosition] =
positionNumber[secondPosition];
        positionNumber[secondPosition] = tmp;
        return positionNumber;
    }


}

interface CheckerCycle {
    boolean check(Node node, Node parent);
}

class CheckerCycleLastParent implements CheckerCycle {
    public boolean check(Node node, Node parent) {
        return parent == null ||
!node.state.equals(parent.state);
    }
}

class CheckerCycleParents implements CheckerCycle {
    public boolean check(Node node, Node parent) {
        Node tmpNode = parent;
        while (tmpNode != null) {
            if (node.state.equals(tmpNode.state)) return false;
            tmpNode = tmpNode.parent;
        }
        return true;
    }
}

class CheckerCycleMemory implements CheckerCycle {
    Set<State> previous = new HashSet<>();

    public boolean check(Node node, Node parent) {
        if (previous.contains(node.state)) return false;
        previous.add(node.state);
        return true;
    }
}
```

```
class Solution {
    LinkedList<Node> solutionNodes;

    public Solution(Node goalNode, Node startNode) {
        this.solutionNodes = new LinkedList<>();
        Node tmpNode = goalNode;
        solutionNodes.add(tmpNode);
        while (tmpNode != startNode) {
            tmpNode = tmpNode.parent;
            solutionNodes.addFirst(tmpNode);
        }
    }

    @Override
    public String toString() {
        StringBuilder solutionString = new StringBuilder();
        for (Node node: solutionNodes)
solutionString.append(node.state).append("\n\n");
        return "Depth: " +
solutionNodes.get(solutionNodes.size()-1).depth + "\n" +
                "Solution: \n" + solutionString;
    }

    public String toString(int firstN, int lastN) {
        StringBuilder solutionString = new StringBuilder();

        solutionString.append("First
").append(firstN).append("\n");
        int i = 0;
        for (Node node: solutionNodes) {
            solutionString.append(node.state).append("\n\n");
            i++;
            if (i >= firstN) break;
        }

        solutionString.append("Last
").append(firstN).append("\n");
        List<Node> solutionNodesReversed =
solutionNodes.subList(solutionNodes.size()-lastN,
solutionNodes.size());
        i = 0;
        for (Node node: solutionNodesReversed) {
            solutionString.append(node.state).append("\n\n");
            i++;
            if (i >= lastN) break;
        }

        return "Depth: " +
solutionNodes.get(solutionNodes.size()-1).depth + "\n" +
                "Solution: \n" + solutionString;
    }
}
```

```
class BFS {
    GeneralSearch generalSearch;
    CheckerCycle checkerCycle;
    Queuing queuing;

    BFS() {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = new CheckerCycleMemory();
        this.queuing = new BFSQueuing();
    }

    BFS(CheckerCycle checkerCycle) {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = checkerCycle;
        this.queuing = new BFSQueuing();
    }

    public Solution search(Problem problem) {
        return generalSearch.generalSearch(problem,
checkerCycle, queuing, null);
    }
}

class DFS {
    GeneralSearch generalSearch;
    CheckerCycle checkerCycle;
    Queuing queuing;

    DFS() {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = new CheckerCycleLastParent();
        this.queuing = new DFSQueuing();
    }

    DFS(CheckerCycle checkerCycle) {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = checkerCycle;
        this.queuing = new DFSQueuing();
    }

    public Solution search(Problem problem) {
        return generalSearch.generalSearch(problem,
checkerCycle, queuing, null);
    }
}

class DFSLim {
    GeneralSearch generalSearch;
    CheckerCycle checkerCycle;
    Queuing queuing;

    DFSLim(int depthMax) {
```

```
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = new CheckerCycleParents();
        this.queuing = new DFSLimQueuing(depthMax);
    }

    DFSLim(CheckerCycle checkerCycle, int depthMax) {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = checkerCycle;
        this.queuing = new DFSLimQueuing(depthMax);
    }

    public Solution search(Problem problem) {
        return generalSearch.generalSearch(problem,
checkerCycle, queuing, null);
    }
}

class DFSIterative{
    GeneralSearch generalSearch;
    CheckerCycle checkerCycle;

    DFSIterative() {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = new CheckerCycleParents();
    }

    DFSIterative(CheckerCycle checkerCycle) {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = checkerCycle;
    }

    public Solution search(Problem problem) {
        Solution solution;
        for (int i = 1; i < 100; i++) {
            solution = generalSearch.generalSearch(problem,
checkerCycle, new DFSLimQueuing(i), null);
            if (solution != null) return solution;
        }
        return null;
    }
}

class GreedyS {
    GeneralSearch generalSearch;
    CheckerCycle checkerCycle;
    Queuing queuing;
    Distance distance;

    GreedyS(Distance distance) {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = new CheckerCycleParents();
        this.queuing = new BestQueuing(new GreedyComparator());
        this.distance = distance;
```

```
    }

    GreedyS(CheckerCycle checkerCycle, Distance distance) {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = checkerCycle;
        this.queuing = new BestQueuing(new GreedyComparator());
        this.distance = distance;
    }

    public Solution search(Problem problem) {
        return generalSearch.generalSearch(problem,
checkerCycle, queuing, distance);
    }
}

class AStarS {
    GeneralSearch generalSearch;
    CheckerCycle checkerCycle;
    Queuing queuing;
    Distance distance;

    AStarS(Distance distance) {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = new CheckerCycleParents();
        this.queuing = new BestQueuing(new AStarComparator());
        this.distance = distance;
    }

    AStarS(CheckerCycle checkerCycle, Distance distance) {
        this.generalSearch = new GeneralSearch();
        this.checkerCycle = checkerCycle;
        this.queuing = new BestQueuing(new AStarComparator());
        this.distance = distance;
    }

    public Solution search(Problem problem) {
        return generalSearch.generalSearch(problem,
checkerCycle, queuing, distance);
    }
}
```

Результат работы, включая экспериментальные оценки временной (количества шагов) и емкостной (единиц памяти) сложности для своего варианта.

```
Problem problem1 = new ProblemPuzzleEight(
  new StatePuzzleEight(new byte[] {3, 6, 4, 2, 5, 8, 0, 1, 7}),
  new StatePuzzleEight(new byte[] {0, 1, 2, 3, 4, 5, 6, 7, 8}));

AStarS aStarS = new AStarS(new CheckerCycleLastParent(), new
ManhattanDistancePuzzleEight());
```

```
Solution solution = aStarS.search(problem1);
System.out.println(solution);
```

В соответствии с вариантом 2, найдем решение при помощи поиска A*, h – манхэттенское расстояние и с использованием проверки всех предков - CheckerCycleParents.

Решение найдено моментально. Напомним, что с такими же начальным и конечным состояниями и с использованием проверки всех предков, неинформированный поиск не справился – в течение 10-15 минут решение так и не было найдено.

Решение представлено на рисунке 1.

```
Depth: 24
Solution:
3 6 4
2 5 8
0 1 7

3 6 4
0 5 8
2 1 7

3 6 4
5 0 8
2 1 7

3 0 4
5 6 8
2 1 7

0 3 4
5 6 8
2 1 7

5 3 4
0 6 8
2 1 7

5 3 4
6 0 8
2 1 7

5 3 4
6 1 8
2 0 7

5 3 4
6 1 8
0 2 7

5 3 4
0 1 8
6 2 7

0 3 4
5 1 8
6 2 7

3 0 4
5 1 8
6 2 7

3 1 4
5 0 8
6 2 7

3 1 4
0 5 8
6 2 7

3 1 4
5 2 8
6 0 7

0 1 4
5 2 8
6 7 0

3 1 4
5 2 0
6 7 8

3 1 4
5 0 2
6 7 8

3 1 4
0 5 2
6 7 8

0 1 4
3 5 2
6 7 8

1 0 4
3 5 2
6 7 8

1 4 0
3 5 2
6 7 8

1 4 2
3 5 0
6 7 8

1 4 2
3 0 5
6 7 8

1 0 2
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8
```

Рисунок 1.

В худшем случае A*-поиск имеет следующую сложность алгоритма:

Временная – O(b^d)

Емкостная – O(b^2), где b – число вершин последователей, d – глубина.

Подсчитаем количество шагов – количество раскрытых вершин, при различных h и стратегиях проверки цикла:

```
AStarS aStarS = new AStarS(new CheckerCycleParents(), new
WrongPositionDistancePuzzleEight());
```

h1, проверка всех предков – 122513 шагов.

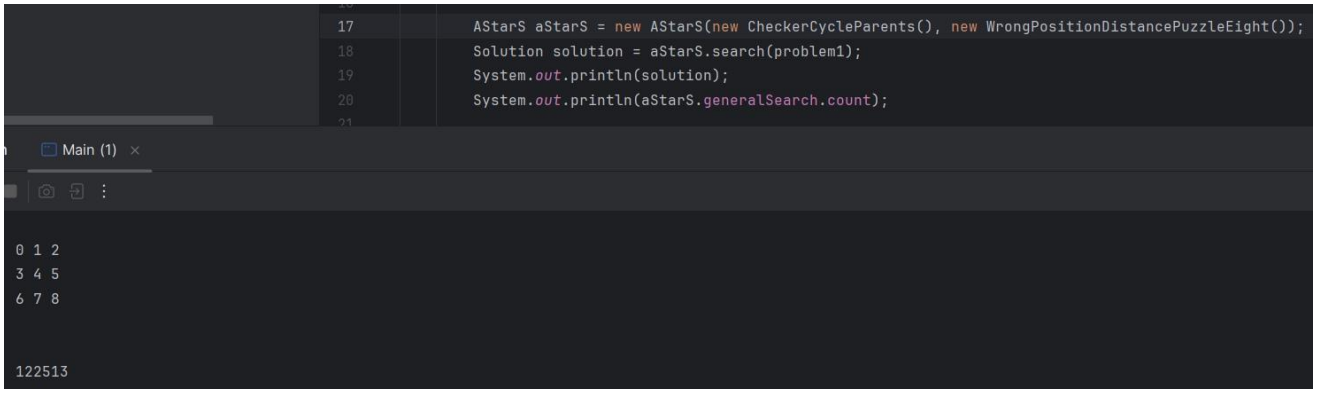

Рисунок 1.1.

```
AStarS aStarS = new AStarS(new CheckerCycleMemory(), new
WrongPositionDistancePuzzleEight());
```

h1, запоминание ранее достигнутых состояний в Хэш-таблице – 49767 шагов.

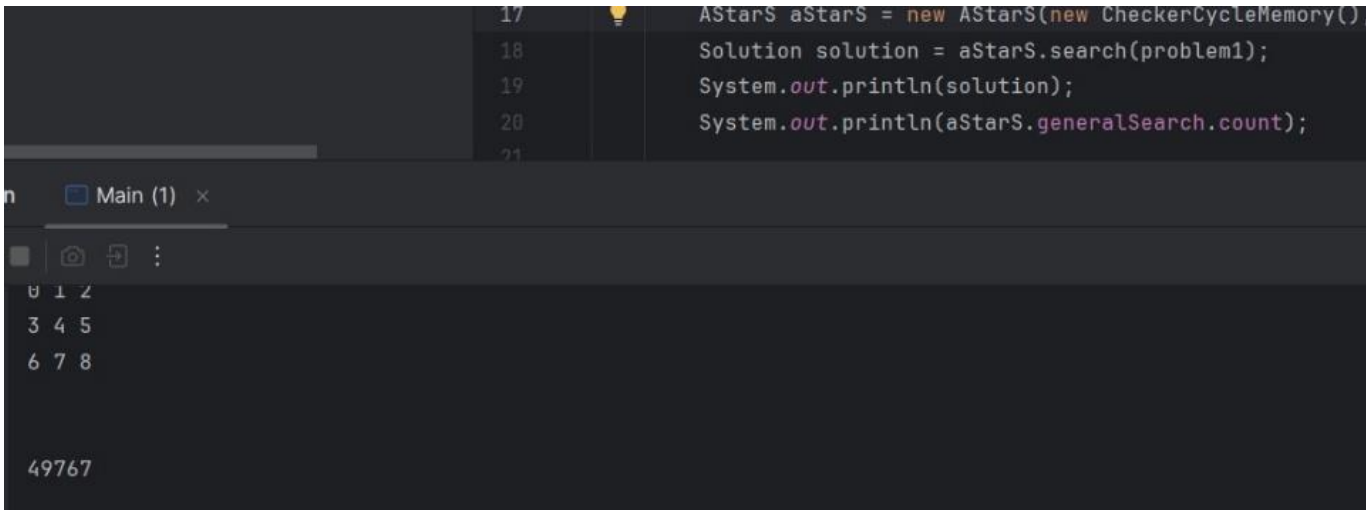

Рисунок 1.2.

```
AStarS aStarS = new AStarS(new CheckerCycleParents(), new
ManhattanDistancePuzzleEight());
Solution solution = aStarS.search(problem1);
```

h2, проверка всех предков – 7848 шагов..

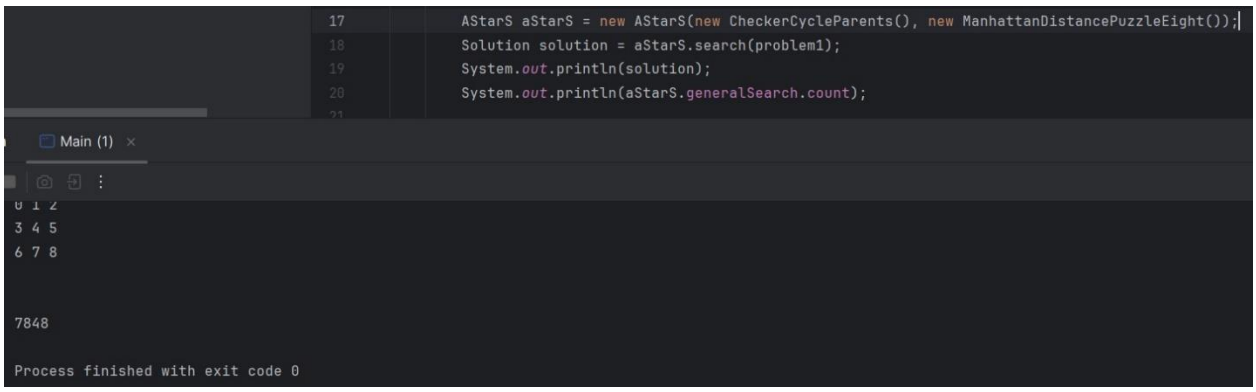

Рисунок 1.3.

```
AStarS aStarS = new AStarS(new CheckerCycleMemory(), new
ManhattanDistancePuzzleEight());
Solution solution = aStarS.search(problem1);
```

h2, запоминание ранее достигнутых состояний в Хэш-таблице – 4877 шагов.

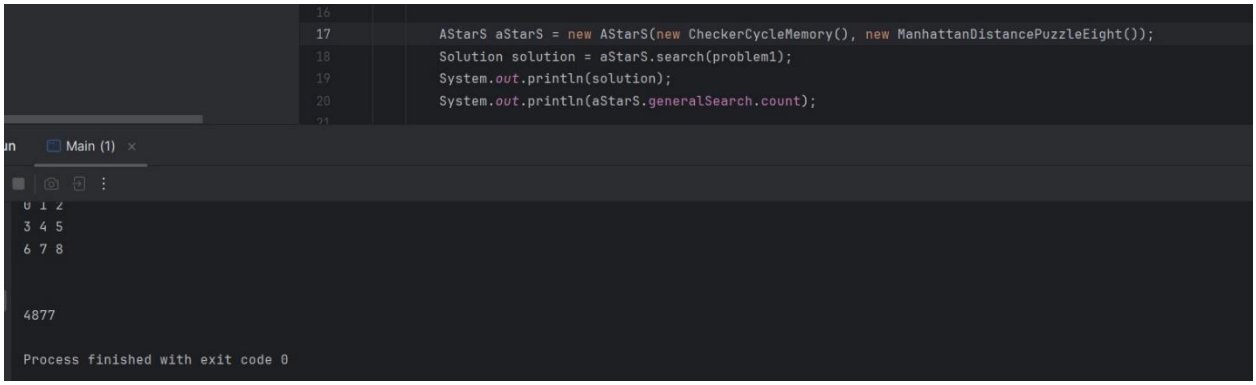

Рисунок 1.4.

Если учесть, что пустой элемент в Головоломке имеет одинаковую вероятность оказаться в любой позиции, то среднее число преемников b = (4*1 + 3*4 + 2*4) / 9 = 2,67.

Сравним теоретическую сложность и экспериментальную оценку времени (количество шагов). Решение найдено на глубине 24. Теоретическая сложность: O(b^n) = O(2,67^24) = 17229404720, в то время как количество шагов для h1: 49767, h2: 4877. Как видим, экспериментальная оценка времени далека от теоретической сложности алгоритма (меньше на несколько порядков) в этом варианте, поэтому решение находится очень быстро.

Экспериментальная емкостная оценка сложность близка к экспериментальной временной оценки (в пределах одного порядка).

Также заметим, что количество шагов при использовании манхэттенского расстояния - h2 заметно меньше по сравнению с h1. Это связано с эффективным коэффициентом ветвления (effective branching factor) – среднее число преемников узла в дереве поиска после применения эвристик. На рисунке 2 два приведено сравнение стоимости поиска и эффективного коэффициента ветвления для поиска с итеративным углублением и поиска A* с эвристиками h1 и h2. Данные усреднены по 100 примерам 8-ки, для решений различной глубины. Из сравнения видно, что манхэттенское расстояние имеет меньший эффективным коэффициентом ветвления и меньшую стоимость.

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Рисунок 2.

Можно сделать вывод, что хорошими эвристиками являются эвристики с низким эффективным коэффициентом ветвления.

Примечание: программа позволяет провести пошаговый анализ работы; для этого необходимо запустить код программы и раскомментировать фрагменты, отмеченные как «//Uncomment for step-by-step output». Пример фрагмента вывода при пошаговой работе приведен на рис.3.

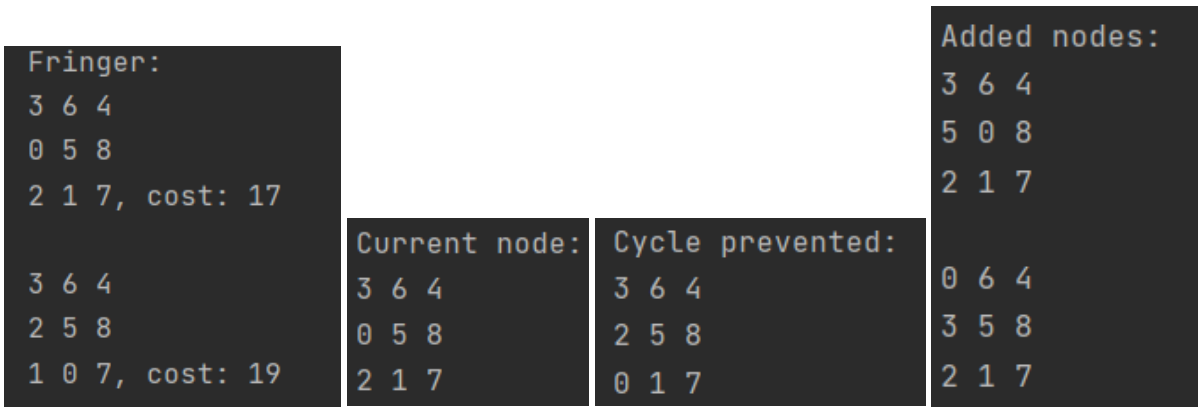

Рисунок 3.

Сравнение результатов информированного и неинформированного поиска

	Экспериментальная оценка временной сложности (к-во шагов)			
Неинформ.	DFS, Parents	DFS, Memory	DFSLim, Parents (L=25)	DFSLim, Memory (L=201)
	? (решение не было найдено в теч.20 мин.)	219763	1714054	17498
Информ.	H1, Parents	H1, Memory	H2, Parents	H2, Memory
	122513	49767	7848	4877

Видно, что решение информированного поиска находится быстрее, чем неинформированного, его эффективность выше.

Вывод

В результате работы было закреплено на практике понимание теоретических основ информированного (эвристического) поиска.