Цель

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

Начальное состояние:

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

Целевое состояние:

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

Поиск в глубину, с ограничением глубины.

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**

Состояния. Описание состояния определяет местонахождение каждой из этих 8 фишек и пустого участка на одном из 9 квадратов.

Начальное состояние задано.

Функция последователей. Эта функция формирует допустимые состояния, которые являются результатом попыток осуществления указанных четырех действий теоретически возможных ходов (Left, Right, Up или Down).

Проверка цели. Она позволяет определить, соответствует ли данное состояние целевой конфигурации, которая задана (см. вариант задания).

Стоимость пути. Каждый этап имеет стоимость 1, поэтому стоимость пути равна количеству этапов пути.

Рассматриваются методы поиска, в которых используется явно заданное **дерево поиска**, создаваемое с помощью начального состояния и функции последователей, которые совместно задают пространство состояний.

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

Описание структур данных и функций выполнено в псевдокоде

Был использован формализованный обобщенный алгоритм поиска, на основе которого были построены поиск в ширину, в глубину, с ограничением глубины.

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))

);

}

Expend(CheckerCycle, node, operations) – функция раскрытия вершин, генерирует множество ее последователей с использованием операторов Operators;

CheckerCycle – проверка цикла, используется в Expend;

nodes – очередь вершин, ожидающих раскрытия (кайма), **add** – добавление в конец;

goalTest(State) – проверка состояния на соответствие целевому;

Queuing (Queue, Elements) – добавляет в очередь Queue множество элементов Elements;

CreateNode(Parent, State, depth) – функция, создает вершину с родителем Parent, состоянием State, глубиной depth**;**

Operations(State) **–** возвращает допустимые состояния из состояния State, которые являются результатом попыток осуществления указанных четырех действий теоретически возможных ходов (Left, Right, Up или Down).

Описание алгоритмов

Поиск в глубину – раскрывает одну из вершин на самом глубоком уровне дерева. Останавливается, когда достигнута цель или заходит в тупик – ни одна вершина нижнего уровня не может быть раскрыта.

Временная сложность – O(b^m), емкостная сложность - O(b*m), где b - число вершин-последователей, m – максимальная глубина пространства поиска.

На рисунке 1 представлена иллюстрация поиска в глубину:

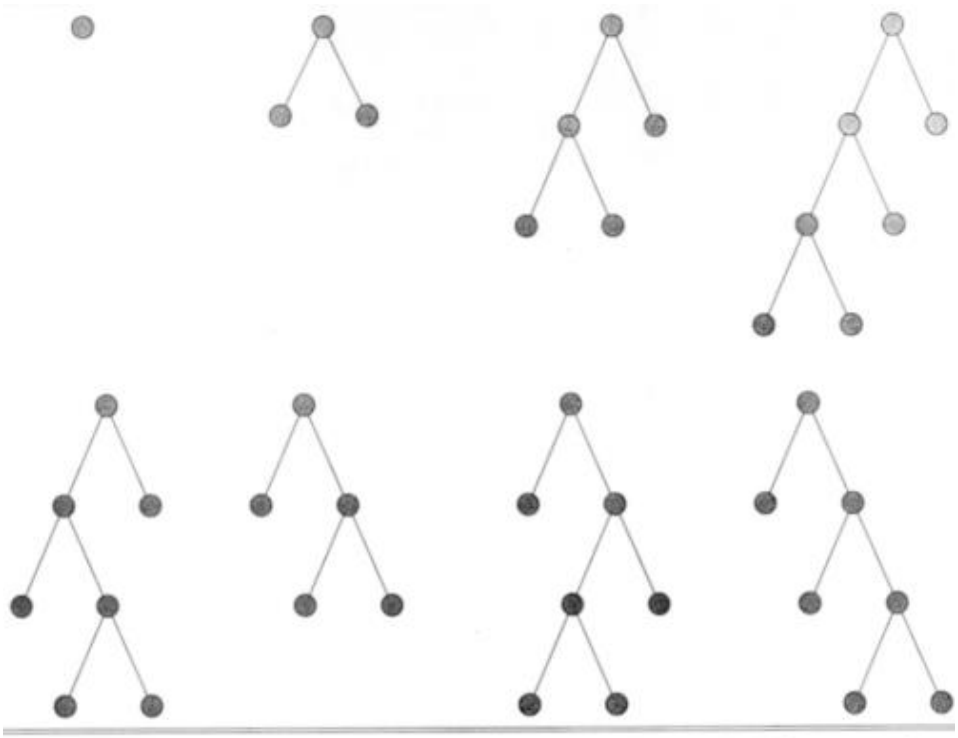


Figure 3.14 Depth-first search trees for a binary search tree. Nodes at depth 3 are assumed to have no successors.

Рисунок 1.

Ограниченный по глубине поиск – поиск в глубину, при котором накладываются ограничения на максимальную глубину пути (чтобы избежать недостатков поиска в глубину).

Временная сложность – O(b^L), емкостная сложность - O(b*L), где b - число вершин-последователей, L – ограничение глубины пространства поиска.

Описание алгоритмов для решения проблемы зацикливания

1. Исключить попадание в состояние, из которого только что вышли. функция раскрытия должна блокировать генерацию потомка в дереве поиска решений, если его состояние совпадает с состоянием родителя

2. Исключить циклы в дереве поиска: функция раскрытия вершин должна блокировать генерацию последователя, состояние которого совпадает с состоянием любого предка данной вершины

3. Блокировать генерацию состояний, ранее сгенерированных в дереве поиска. **Авторское решение**: ранее сгенерированные вершины сохранять в Хэш-таблицу, для общего случая временная сложность: O(1), емкостная: O(b^d).

Исходный код

Был выбран объектно-ориентированный язык Java. Важное понятие: **интерфейс** в языке программирования Java — это абстрактный тип, который используется для объявления поведения, которое должны реализовать классы. Это позволяет обобщенному алгоритма поиску generalSearch работать с разной реализацией проблем (интерфейс Problem), состояний (State) и т.д. В данной работе классы (ProblemPuzzleEight, StatePuzzleEight) реализованы только для решения «Игры в восьмерку», но ту же самую функцию generalSearch можно использовать для решения других задач (не меняю фунцию поиска generalSearch), если для этих задач реализовать классы проблемы и состояния.

```
import java.util.*;

public class GeneralSearch {
    CheckerCycle checkerCycle;
    public Solution generalSearch(Problem problem, CheckerCycle
checkerCycle, Queuing queuing) {
        this.checkerCycle = checkerCycle;
        LinkedList<Node> nodes = new LinkedList<>();
        Node startNode = new Node(null, problem.initState, 0);
        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 + "\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);
            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 Node {
    Node parent;
    State state;
    int depth;

    public Node(Node parent, State state, int depth) {
        this.parent = parent;
        this.state = state;
        this.depth = depth;
    }
}

interface State {

}

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 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];
    }
}

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(0).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);
    }
}

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);
    }
}

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);
    }
}
```

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

В соответствии с вариантом 2, найдем решение при помощи DFS и с использованием проверки всех предков - CheckerCycleParents. В конструктор ProblemPuzzleEight передается два состояния - начальное и целевое.

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

DFS dfs = new DFS(new CheckerCycleParents());
Solution solutionDFS = dfs.search(problem);
System.out.println(solutionDFS);
```

Нет решения. Также нет решения и по теореме существование решения (https://e-maxx.ru/algo/15_puzzle, доказательство см. там же) (рисунок 2):

Существование решения

Здесь мы рассмотрим такую задачу: по данной позиции на доске сказать, существует ли последовательность ходов, приводящая к решению, или нет.

Пусть дана некоторая позиция на доске:

$$\begin{matrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{matrix}$$

где один из элементов равен нулю и обозначает пустую клетку $a_z = 0$.

Рассмотрим перестановку:

$$a_1 a_2 \dots a_{z-1} a_{z+1} \dots a_{15} a_{16}$$

(т.е. перестановка чисел, соответствующая позиции на доске, без нулевого элемента)

Обозначим через N количество инверсий в этой перестановке (т.е. количество таких элементов a_i и a_j, что $i < j$, но $a_i > a_j$).

Далее, пусть K — номер строки, в которой находится пустой элемент (т.е. в наших обозначениях $K = (z-1) \text{ div } 4 + 1$).

Тогда, **решение существует тогда и только тогда, когда $N + K$ чётно**.

Рисунок 2

N + K = 12 + 3 = 15 – нечетно

Изменим начальное состояние (поменяем 7 и 0 местами) и найдем для него решение.

```
Problem problem = 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}));

DFS dfs = new DFS(new CheckerCycleParents());
Solution solutionDFS = dfs.search(problem);
System.out.println(solutionDFS.toString(3, 3));
```

Проверка зацикливания последнего предка и проверка всех предков оказалась неэффективной – в течение нескольких минут алгоритм не мог найти решение. При этом, если проверять зацикливание при помощи сохранения вершин в Хэш-таблицу, решение найдется меньше, чем за секунду.

Глубина, на которой нашлось решение – 63196. Вывод в консоль - первые три шага и последние три шага, представлены на рисунке 3:

```
"C:\Program F
Depth: 63196
Solution:
First 3
3 6 4
2 5 8
0 1 7

3 6 4
0 5 8
2 1 7

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

1 0 2
3 4 5
6 7 8

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

Рисунок 3

Временная сложность – O(b^m), емкостная сложность – O(b*m). O() – это верхняя граничная оценка сложности алгоритма. b = 2, 3, 4. Экспериментально емкостная сложность, действительно, имеет кол-во единиц

памяти b*63196, а вот временная сложность далека от своей верхней оценки, иначе решение со с количеством шагов b^63196 не было найдено. Скорее всего, количество шагов совпадает или близко с количеством единиц затраченной памяти, если учитывать то, что решение найдено быстро.

Решение не оптимальное по стоимости пути, если считать, что каждое перемещение ячейки имеет стоимость 1 ед.

Найдем оптимальное решение при помощи BFS (поиска в ширину), и затем приблизимся к оптимальному при помощи DFS, ограниченного по глубине.

```
BFS bfs = new BFS(new CheckerCycleMemory());
Solution solutionBFS = bfs.search(problem);
System.out.println(solutionBFS);
```

Оптимальное решение, найденное при помощи BFS, имеет количество шагов (глубину) – 24. Решение представлено на рисунке 4.

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

3 6 4
2 5 8
1 0 7

3 6 4
2 5 8
1 7 0

3 6 4
2 5 0
1 7 8

3 6 4
2 0 5
1 7 8

3 6 4
0 2 5
1 7 8

3 6 4
1 2 5
0 7 8

3 6 4
1 2 5
7 0 8

3 6 4
1 2 5
7 8 0

3 6 4
1 2 0
7 8 5

3 6 4
1 0 2
7 8 5

3 0 4
1 6 2
7 8 5

0 3 4
1 6 2
7 8 5

1 3 4
0 6 2
7 8 5

1 3 4
1 6 2
7 8 5

1 4 0
6 3 2
7 8 5

1 4 2
6 3 0
7 8 5

1 4 2
6 3 5
7 8 0

1 4 2
6 3 5
7 0 8

1 4 2
6 3 5
0 7 8

1 4 2
0 3 5
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 4

Теперь при помощи DFS с ограничением глубины 25 найдем такое же решение.

```
DFSLim dfsLim = new DFSLim(new CheckerCycleParents(), 25);
Solution solutionDFSLim = dfsLim.search(problem);
System.out.println(solutionDFSLim);
```

Решение нашлось и с использованием проверки только предков вершины. Решение совпадает с решением на рисунке 4.

Экспериментальная временная и емкостная сложности совпадает с теоретической. Емкостная сложность b*24 единиц памяти и временная b^24 количество шагов.

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.5.

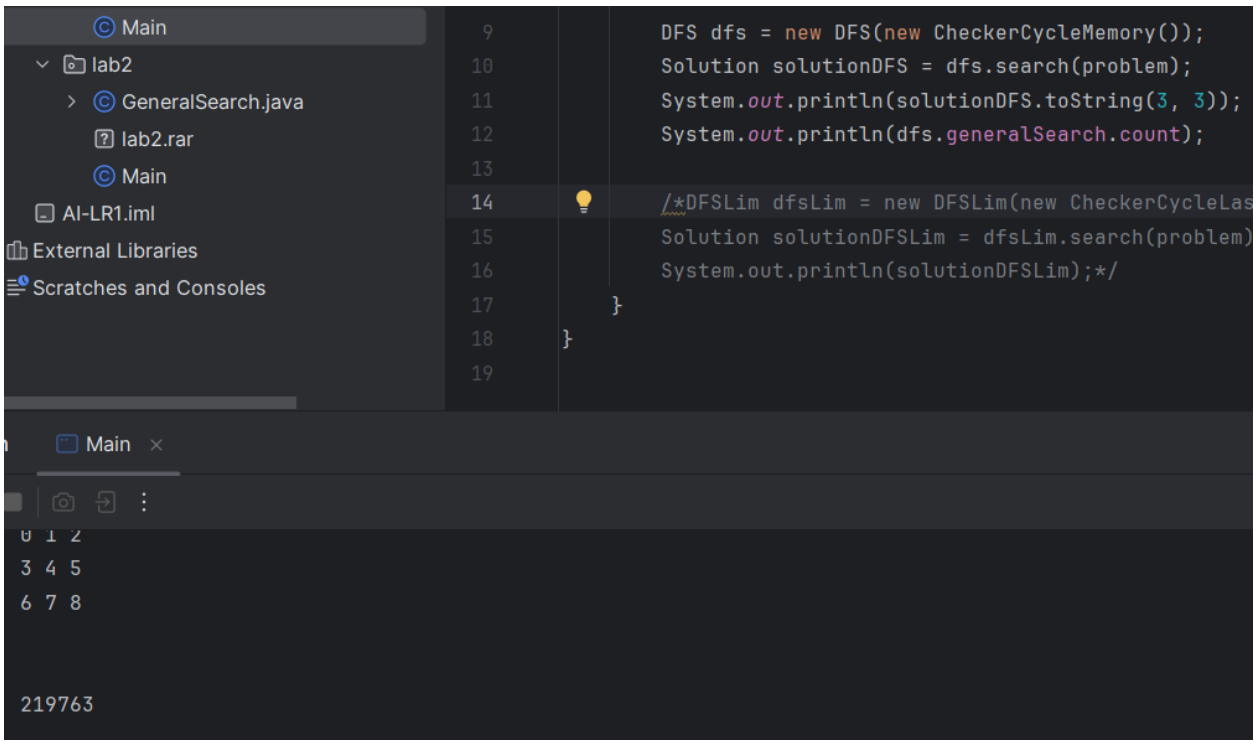
```
Fringer:
6 5 4
3 0 8
2 1 7

6 4 0
3 5 8
2 1 7

3 6 4
5 0 8
2 1 7

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

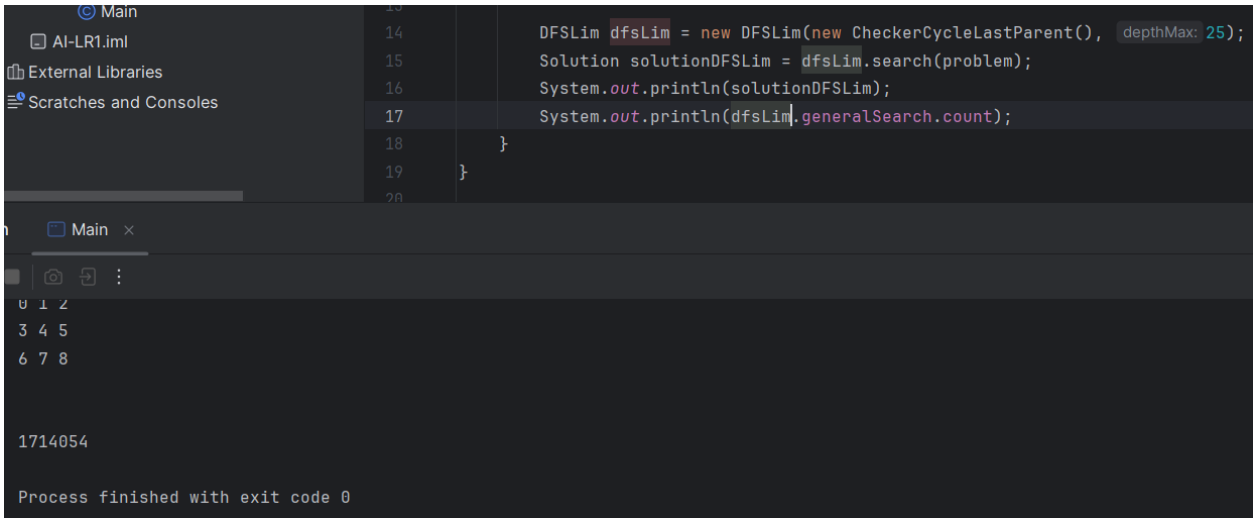
```
Current node:
6 5 4
3 0 8
2 1 7
```

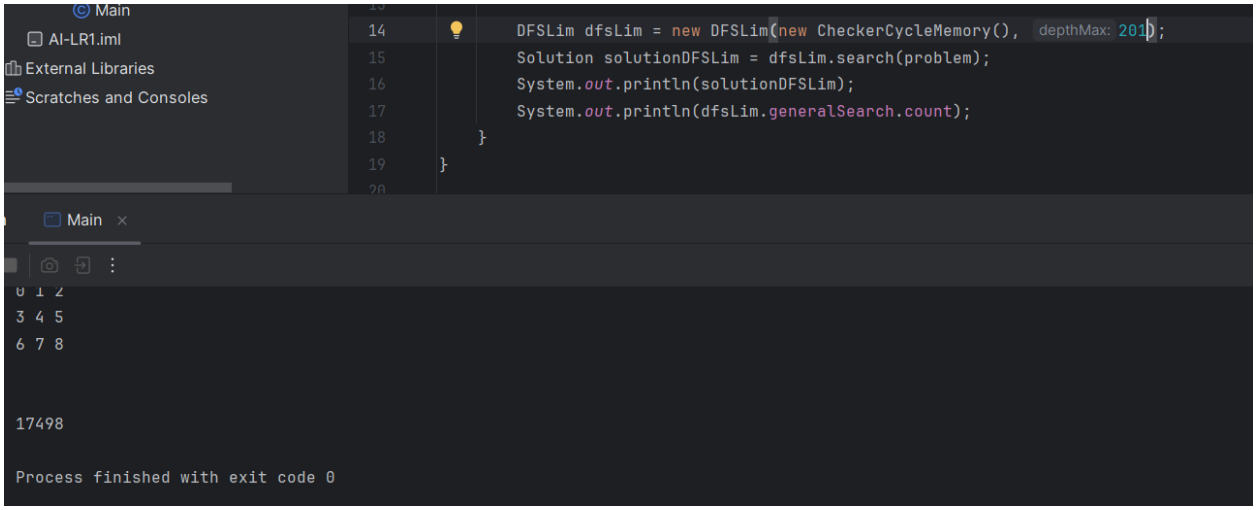
```
Cycle prevented:
6 0 4
3 5 8
2 1 7
```

```
Added nodes:
6 5 4
0 3 8
2 1 7

6 5 4
3 8 0
2 1 7

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

Рисунок 5.


Main
lab2
GeneralSearch.java
lab2.rar
Main
AI-LR1.iml
External Libraries
Scratches and Consoles
DFS dfs = new DFS(new CheckerCycleMemory());
Solution solutionDFS = dfs.search(problem);
System.out.println(solutionDFS.toString(3, 3));
System.out.println(dfs.generalSearch.count);
/*DFSLim dfsLim = new DFSLim(new CheckerCycleLas
Solution solutionDFSLim = dfsLim.search(problem)
System.out.println(solutionDFSLim);*/
}
}
Main
3 4 5
6 7 8
219763


Main
AI-LR1.iml
External Libraries
Scratches and Consoles
DFSLim dfsLim = new DFSLim(new CheckerCycleLastParent(), depthMax: 25);
Solution solutionDFSLim = dfsLim.search(problem);
System.out.println(solutionDFSLim);
System.out.println(dfsLim.generalSearch.count);
}
}
Main
3 4 5
6 7 8
1714054
Process finished with exit code 0


Main
AI-LR1.iml
External Libraries
Scratches and Consoles
DFSLim dfsLim = new DFSLim(new CheckerCycleMemory(), depthMax: 201);
Solution solutionDFSLim = dfsLim.search(problem);
System.out.println(solutionDFSLim);
System.out.println(dfsLim.generalSearch.count);
}
}
Main
3 4 5
6 7 8
17498
Process finished with exit code 0

Вывод

В результате работы были закреплены на практике понимания общих идей поиска в пространстве состояний и стратегий слепого поиска.