**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра ВТ**

**ОТЧЕТ**

**по лабораторной работе №1**

**по дисциплине «Искусственный интеллект»**

**Тема: Методы неинформированного (слепого) поиска**

Вариант 3

|  |  |  |
| --- | --- | --- |
|  |  | Баталев И.А. |
|  |  | Зайков Д.Г. |
|  |  | Панарин А.Е. |
| Студенты гр. 8305 |  | Платунова К.Д. |
| Преподаватель |  | Родионов С.В. |

Санкт-Петербург

2021

**Цель работы**

Практическое закрепление понимания общих идей поиска в пространстве состояний и стратегий слепого поиска.

**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:

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

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).

В программе должна быть реализована функция определения повторного состояния.

Функция достижения целевого состояния: она позволяет определить, соответствует ли данное состояние целевой конфигурации, которая задана.

Программа должна поддерживать пошаговый режим (очередной шаг выполняться по нажатию клавиши) и выводить на каждом шаге:

– вновь добавленные после раскрытия вершины,

– выявленные повторные вершины,

– текущее состояние «каймы» (вершин, ожидающих раскрытия);

– текущую вершину, выбираемую для раскрытия на данном шаге;

– факт достижения целевого состояния.

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

Для представления вершины был создан массив, в котором игровое поле представлено в виде элементов массива, хранящих номер размещённой в ячейке фишки.

Были созданы правила поиска новых состояний: создаются и добавляются в стек возможные переходы из текущего состояния, после рассматривается уже верхушка стека.

**Описание алгоритма**

*Стратегия поиска в глубину*

Поиск сначала в глубину всегда раскрывает одну из вершин на самом глубоком уровне дерева. Останавливается, когда поиск достигает цели или заходит в тупик (ни одна вершина не может быть раскрыта). В последнем случае выполняется возврат назад и раскрываются вершины на более верхних уровнях. Недостаток поиска в глубину – способность углубляться в неверном направлении.

*Стратегия поиска с итеративным углублением*

В ограниченном по глубине поиске не всегда удается хорошо задать ограничения. Итеративно углубляющийся поиск – стратегия, которая обходит выбор лучшего ограничения глубины, пробуя все возможные ограничения.

**Исходный код**

Main.java

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

int state[][] = {

{0, 4, 3},

{6, 2, 1},

{7, 5, 8}

};

SearchTree searchTree = new SearchTree(state);

// в глубину

/\*

long startTime = System.currentTimeMillis();

searchTree.DFS(Integer.MAX\_VALUE);

long endTime = System.currentTimeMillis();

System.out.println(endTime - startTime);

\*/

// с итеративным углублением

long startTime = System.currentTimeMillis();

searchTree.IDS();

long endTime = System.currentTimeMillis();

System.out.println(endTime - startTime);

}

}

SearchTree.java

import java.util.\*;

public class SearchTree {

private Node initialNode;

private final int[][] searchState = {

{1, 2, 3},

{4, 0, 5},

{6, 7, 8}

};

private Deque<Node> nodeQueue;

private Deque<Node> checkedNodes;

public SearchTree(int[][] state) {

nodeQueue = new ArrayDeque<>();

checkedNodes = new ArrayDeque<>();

int emptyI = 0;

int emptyJ = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (state[i][j] == 0) {

emptyI = i;

emptyJ = j;

}

}

}

this.initialNode = new Node(state, emptyI, emptyJ, null);

}

public void makeQueue(Node node) {

int emptyI = node.getEmptyI();

int emptyJ = node.getEmptyJ();

if (emptyI > 0) {

nodeQueue.push(makeNode(node, emptyI - 1, emptyJ));

}

if (emptyI < 2) {

nodeQueue.push(makeNode(node, emptyI + 1, emptyJ));

}

if (emptyJ > 0) {

nodeQueue.push(makeNode(node, emptyI, emptyJ - 1));

}

if (emptyJ < 2) {

nodeQueue.push(makeNode(node, emptyI, emptyJ + 1));

}

}

public boolean checkCycle(Node node) {

for (Node n:

checkedNodes) {

int[][] state = n.getState();

int[][] newState = node.getState();

if (compareState(n.getState(), node.getState())) {

return false;

}

}

return true;

}

public boolean compareState(int[][] s1, int[][] s2) {

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (s1[i][j] != s2[i][j]){

return false;

}

}

}

return true;

}

public Node makeNode(Node node, int emptyI, int emptyJ) {

int[][] state = new int[3][3];

for (int i = 0;i < 3; i++) {

for (int j = 0; j < 3; j++) {

state[i][j] = node.getState()[i][j];

}

}

int oldEmptyI = node.getEmptyI();

int oldEmptyJ = node.getEmptyJ();

state[oldEmptyI][oldEmptyJ] = state[emptyI][emptyJ];

state[emptyI][emptyJ] = 0;

return new Node(state, emptyI, emptyJ, node);

}

public boolean goalTest(int[][] state) {

boolean result = true;

for (int i = 0; i < 3; i++) {

for (int j = 0; j< 3; j++) {

if (state[i][j] != searchState[i][j]) {

result = false;

break;

}

}

}

return result;

}

public boolean DFS(int deep) {

makeQueue(initialNode);

while (true) {

if (nodeQueue.isEmpty()) {

return false;

}

Node node = nodeQueue.removeFirst();

if (goalTest(node.getState())) {

showSolution(node);

return true;

} else {

if (node.getDepth() < deep && checkCycle(node)) {

makeQueue(node);

checkedNodes.add(node);

}

}

}

}

public boolean IDS() {

for (int i = 0; i < Integer.MAX\_VALUE; i++) {

nodeQueue = new ArrayDeque<>();

checkedNodes = new ArrayDeque<>();

if (DFS(i)) {

return true;

}

}

return false;

}

public void showSolution(Node node) {

while (node.getParentNode() != null) {

System.out.println(showState(node.getState(), node.getDepth()));

node = node.getParentNode();

}

System.out.println(showState(node.getState() , node.getDepth()));

}

public String showState(int[][] state, int deep) {

StringBuilder stringBuilder = new StringBuilder();

for (int i = 0; i < 3; i++) {

stringBuilder.append("[");

for (int j = 0; j < 2; j++) {

stringBuilder.append(state[i][j]).append(" ");

}

stringBuilder.append(state[i][2]);

stringBuilder.append("]\n");

}

stringBuilder.append("Deep = ").append(deep);

return stringBuilder.toString();

}

public Node getInitialNode() {

return initialNode;

}

public void setInitialNode(Node initialNode) {

this.initialNode = initialNode;

}

public int[][] getSearchState() {

return searchState;

}

public Deque<Node> getNodeQueue() {

return nodeQueue;

}

public void setNodeQueue(Deque<Node> nodeQueue) {

this.nodeQueue = nodeQueue;

}

@Override

public String toString() {

return "SearchTree{" +

"\ninitialNode=\n" + initialNode +

"\nsearchState=\n" + Arrays.toString(searchState[0]) +

"\n" + Arrays.toString(searchState[1]) +

"\n" + Arrays.toString(searchState[2]) + "\n" +

"nodeQueue=\n" + nodeQueue +

'}';

}

}

Node.java

import java.util.Arrays;

public class Node {

private int[][] state;

private Node parentNode;

private int depth;

private int emptyI;

private int emptyJ;

public Node(int[][] state, int emptyI, int emptyJ, Node parentNode) {

this.state = state;

this.emptyI = emptyI;

this.emptyJ = emptyJ;

this.parentNode = parentNode;

this.depth = parentNode == null ? 0 : parentNode.getDepth() + 1;

}

public int[][] getState() {

return state;

}

public void setState(int[][] state) {

this.state = state;

}

public Node getParentNode() {

return parentNode;

}

public void setParentNode(Node parentNode) {

this.parentNode = parentNode;

}

public int getDepth() {

return depth;

}

public void setDepth(int depth) {

this.depth = depth;

}

public int getEmptyI() {

return emptyI;

}

public void setEmptyI(int emptyI) {

this.emptyI = emptyI;

}

public int getEmptyJ() {

return emptyJ;

}

public void setEmptyJ(int emptyJ) {

this.emptyJ = emptyJ;

}

@Override

public String toString() {

return "Node{" +

"\nstate=\n" + Arrays.toString(state[0]) + "\n" + Arrays.toString(state[1]) + "\n" + Arrays.toString(state[2]) + "\n" +

"parentNode=" + parentNode +

", depth=" + depth +

", emptyI=" + emptyI +

", emptyJ=" + emptyJ +

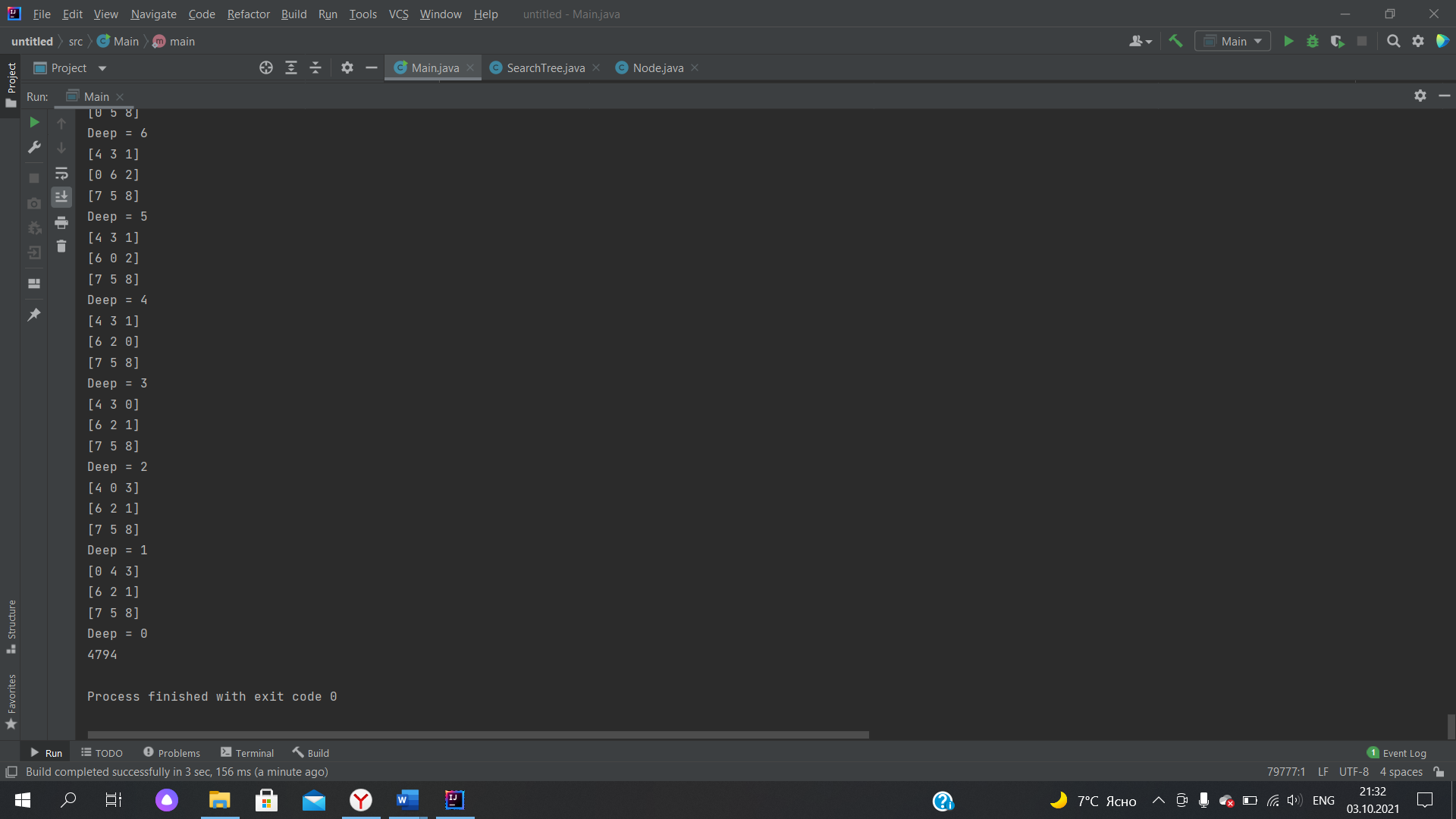
'}';

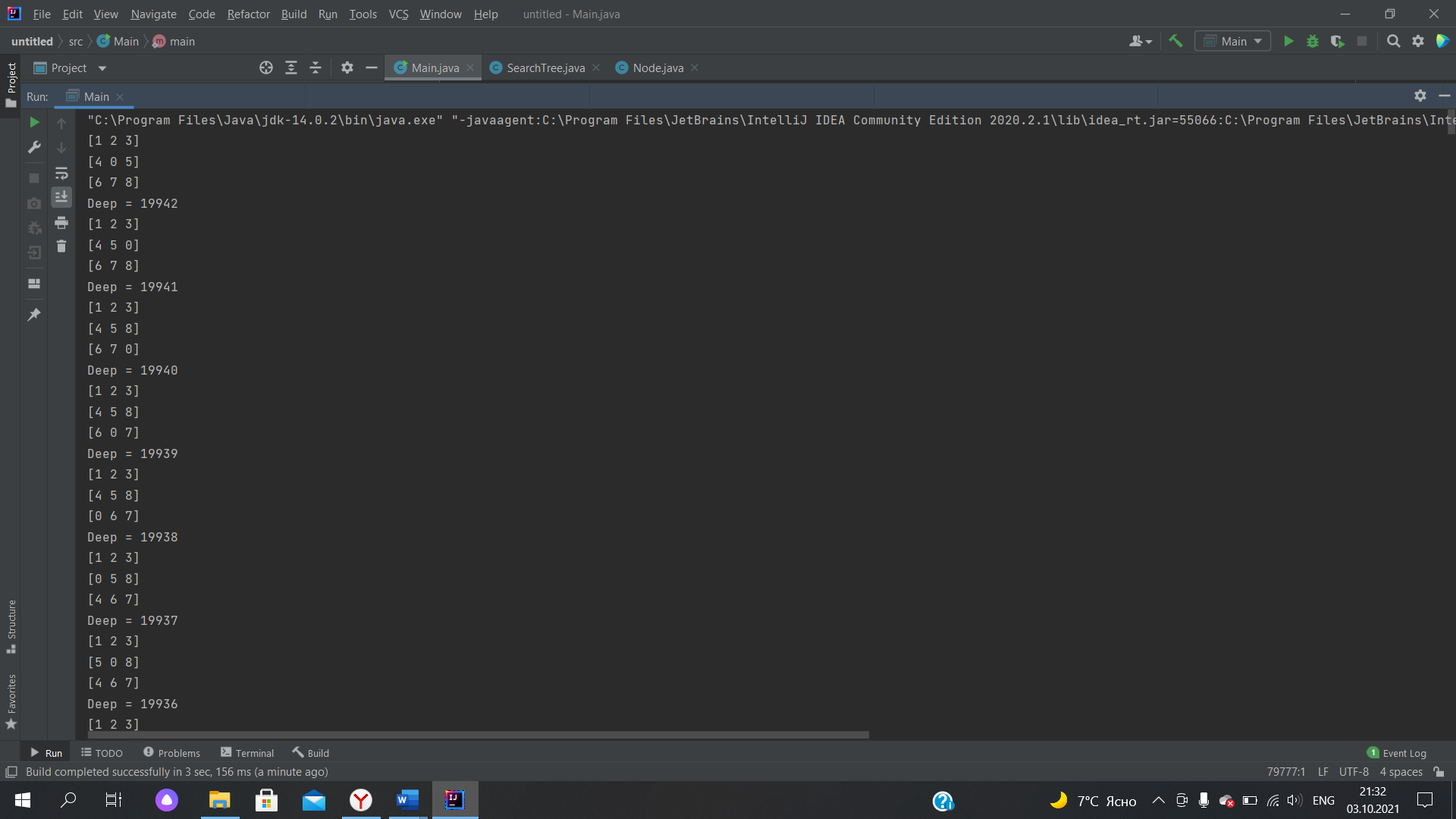
}

}

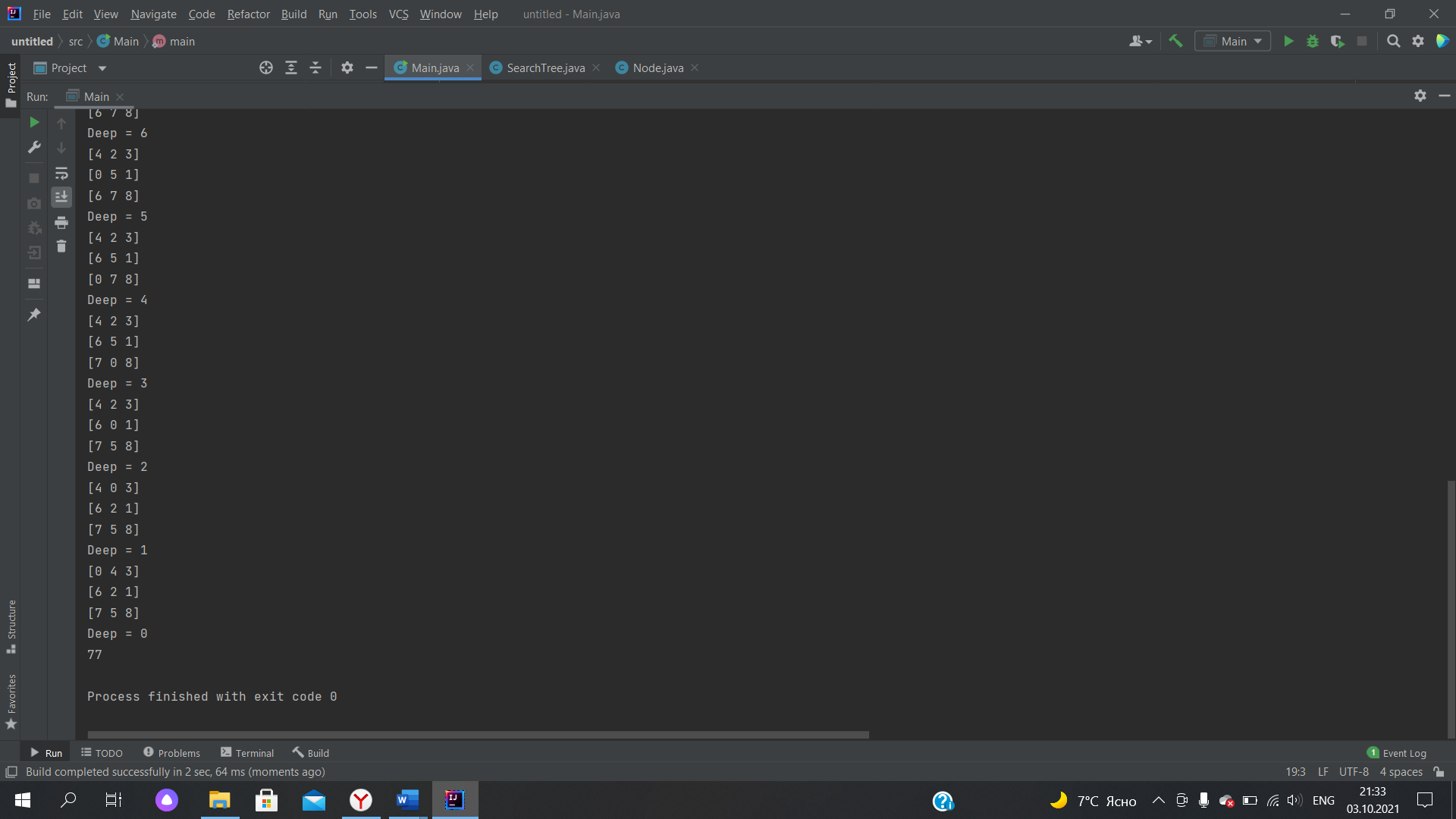
**Результат работы**

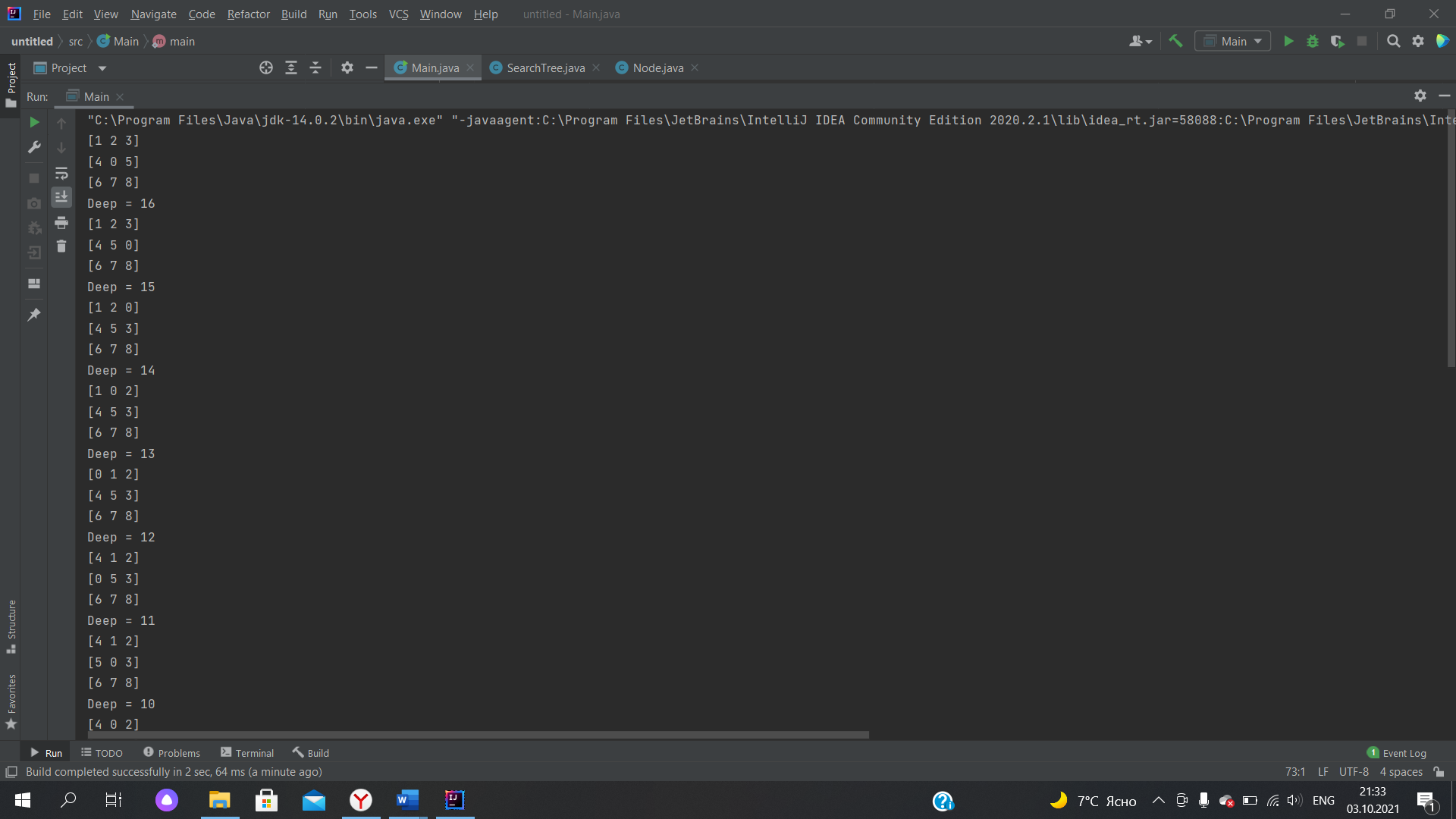
*Стратегия поиска в глубину*





*Стратегия поиска с итеративным углублением*





**Вывод**

Закреплено понимание общих идей поиска в пространстве состояний и стратегий слепого поиска.

В работе были реализованы две стратегии поиска: в глубину и с итеративным углублением. Вторая оказалась более эффективной, чем первая, так как эта стратегия обходит выбор лучшего ограничения глубины, пробуя все возможные ограничения.