**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 2 з дисципліни

«Проектування алгоритмів»

**«Неінформативний, інформативний та локальний пошук»**

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*Піонтківський Віталій ІП-12*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc81070688)

[2 ЗаВдання 4](#_Toc81070689)

[3 Виконання 8](#_Toc81070690)

[3.1 Псевдокод алгоритмів 8](#_Toc81070691)

[3.2 Програмна реалізація 8](#_Toc81070692)

[3.2.1 Вихідний код 8](#_Toc81070693)

[3.2.2 Приклади роботи 8](#_Toc81070694)

[3.3 Дослідження алгоритмів 8](#_Toc81070695)

[Висновок 11](#_Toc81070696)

[Критерії оцінювання 12](#_Toc81070697)

# Мета лабораторної роботи

Мета роботи – розглянути та дослідити алгоритми неінформативного, інформативного та локального пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

# ЗаВдання

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту (таблиця 2.1) за допомогою алгоритму неінформативного пошуку **АНП**, алгоритму інформативного пошуку **АІП,** що використовує задану евристичну функцію Func, або алгоритму локального пошуку **АЛП та бектрекінгу,** що використовує задану евристичну функцію Func.

Програму реалізувати на довільній мові програмування.

**Увага!** Алгоритм неінформативного пошуку **АНП,** реалізовується за принципом «AS IS», тобто так, як є, без додаткових модифікацій (таких як перевірка циклів, наприклад).

Провести серію експериментів для вивчення ефективності роботи алгоритмів. Кожний експеримент повинен відрізнятись початковим станом. Серія повинна містити не менше 20 експериментів для кожного алгоритму. Початковий стан зафіксувати у таблиці експериментів. За проведеними серіями необхідно визначити:

* середню кількість етапів (кроків), які знадобилось для досягнення розв’язку (ітерації);
* середню кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе;
* середню кількість згенерованих станів під час пошуку;
* середню кількість станів, що зберігаються в пам’яті під час роботи програми.

Передбачити можливість обмеження виконання програми за часом (30 хвилин) та використання пам’яті (1 Гб).

**Використані позначення:**

* **8-ферзів** – Задача про вісім ферзів полягає в такому розміщенні восьми ферзів на шахівниці, що жодна з них не ставить під удар один одного. Тобто, вони не повинні стояти в одній вертикалі, горизонталі чи діагоналі.
* **8-puzzle** – гра, що складається з 8 однакових квадратних пластинок з нанесеними числами від 1 до 8. Пластинки поміщаються в квадратну коробку, довжина сторони якої в три рази більша довжини сторони пластинок, відповідно в коробці залишається незаповненим одне квадратне поле. Мета гри – переміщаючи пластинки по коробці досягти впорядковування їх по номерах, бажано зробивши якомога менше переміщень.
* **Лабіринт** – задача пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху. Структура лабіринту зчитується з файлу, або генерується програмою.
* **LDFS** – Пошук вглиб з обмеженням глибини.
* **BFS** – Пошук вшир.
* **IDS** – Пошук вглиб з ітеративним заглибленням.
* **A\*** – Пошук А\*.
* **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
* **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь А може стояти на одній лінії з ферзем В, проте між ними стоїть ферзь С; тому А не б’є В).
* **F2** – кількість пар ферзів, які б’ють один одного без урахування видимості.
* **H1** – кількість фішок, які не стоять на своїх місцях.
* **H2** – Манхетенська відстань.
* **H3** – Евклідова відстань.
* **COLOR** – Задача розфарбування карти самостійно обраної країни, не менше 20 регіонів (областей). Необхідно розфарбувати карту не більше ніж у 4 різні кольори. Мається на увазі приписування кожному регіону власного кольору так, щоб кольори сусідніх регіонів відрізнялись. Використовувати евристичну функцію, яка повертає кількість пар суміжних вузлів, що мають однаковий колір (тобто кількість конфліктів). Реалізувати алгоритм пошуку із поверненнями (backtracking) для розв’язання поставленої задачі. Для підвищення швидкодії роботи алгоритму використати евристичну функцію, а початковим станом вважати випадкову вершину.
* **HILL –** Пошук зі сходженням на вершину з використанням із використанням руху вбік (на 100 кроків) та випадковим перезапуском (кількість необхідних разів запуску визначити самостійно).
* **ANNEAL** – Локальний пошук із симуляцією відпалу. Робоча характеристика – залежність температури Т від часу роботи алгоритму t. Можна розглядати лінійну залежність: T = 1000 - k∙t, де k – змінний коефіцієнт.
* **BEAM** – Локальний променевий пошук. Робоча характеристика – кількість променів k. Експерименти проводи із кількістю променів від 2 до 21.
* **MRV** – евристика мінімальної кількості значень;
* **DGR** – ступенева евристика.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Задача** | **АНП** | **АІП** | **АЛП** | **Func** |
| 1 | Лабіринт | LDFS | A\* |  | H2 |
| 2 | Лабіринт | LDFS | RBFS |  | H3 |
| 3 | Лабіринт | BFS | A\* |  | H2 |
| 4 | Лабіринт | BFS | RBFS |  | H3 |
| 5 | Лабіринт | IDS | A\* |  | H2 |
| 6 | Лабіринт | IDS | RBFS |  | H3 |
| 7 | 8-ферзів | LDFS | A\* |  | F1 |
| 8 | 8-ферзів | LDFS | A\* |  | F2 |
| 9 | 8-ферзів | LDFS | RBFS |  | F1 |
| 10 | 8-ферзів | LDFS | RBFS |  | F2 |
| 11 | 8-ферзів | BFS | A\* |  | F1 |
| 12 | 8-ферзів | BFS | A\* |  | F2 |
| 13 | 8-ферзів | BFS | RBFS |  | F1 |
| 14 | 8-ферзів | BFS | RBFS |  | F2 |
| 15 | 8-ферзів | IDS | A\* |  | F1 |
| 16 | 8-ферзів | IDS | A\* |  | F2 |
| 17 | 8-ферзів | IDS | RBFS |  | F1 |
| 18 | Лабіринт | LDFS | A\* |  | H3 |
| 19 | 8-puzzle | LDFS | A\* |  | H1 |
| 20 | 8-puzzle | LDFS | A\* |  | H2 |
| 21 | 8-puzzle | LDFS | RBFS |  | H1 |
| 22 | 8-puzzle | LDFS | RBFS |  | H2 |
| 23 | 8-puzzle | BFS | A\* |  | H1 |
| 24 | 8-puzzle | BFS | A\* |  | H2 |
| 25 | 8-puzzle | BFS | RBFS |  | H1 |
| 26 | 8-puzzle | BFS | RBFS |  | H2 |
| 27 | Лабіринт | BFS | A\* |  | H3 |
| 28 | 8-puzzle | IDS | A\* |  | H2 |
| 29 | 8-puzzle | IDS | RBFS |  | H1 |
| 30 | 8-puzzle | IDS | RBFS |  | H2 |
| 31 | COLOR |  |  | HILL | MRV |
| 32 | COLOR |  |  | ANNEAL | MRV |
| 33 | COLOR |  |  | BEAM | MRV |
| 34 | COLOR |  |  | HILL | DGR |
| 35 | COLOR |  |  | ANNEAL | DGR |
| 36 | COLOR |  |  | BEAM | DGR |

# Виконання

## Псевдокод алгоритмів

FUNCTION solveLDFS

IF (!areSame(matrix, solutionMatrix) && depth < limit)

IF (this.moveLeft(matrix))

IF (solveLDFS(matrix, depth + 1))

solution.add(old\_matrix);

RETURN true;

this.moveRight(matrix);

END IF

END IF

END IF

IF (this.moveUp(matrix))

IF (solveLDFS(matrix, depth + 1))

solution.add(old\_matrix);

RETURN true;

END IF

this.moveDown(matrix);

END IF

IF (this.moveRight(matrix))

IF (solveLDFS(matrix, depth + 1))

solution.add(old\_matrix);

RETURN true;

END IF

this.moveLeft(matrix);

END IF

IF (this.moveDown(matrix))

IF (solveLDFS(matrix, depth + 1))

solution.add(old\_matrix);

RETURN true;

END IF

this.moveUp(matrix);

END IF

ESLE IF(areSame(matrix, solutionMatrix))

solution.add(matrix);

RETURN true;

ESLE

System.out.print("Limit\n");

RETURN false;

END IF

RETURN false;

FUNCTION solve A\*

NodeSecond parent = new NodeSecond(-1);

NodeSecond currentNode = new NodeSecond(getStartMatrix(), parent, 0);

ArrayList<NodeSecond> closedList = new ArrayList<>();

ArrayList<NodeSecond> openList = new ArrayList<>();

int counter = 0;

WHILE(!currentNode.solved(getSolutionMatrix()))

IF(openList.size()> counter)

System.out.println("Elements in open list:"+openList.size());

System.out.println("Elements in closed list:"+closedList.size());

counter +=1000;

END IF

currentNode.doChildren();

FOR (int i = 0; i< closedList.size(); i++)

FOR (int j = 0; j< currentNode.getChildren().size(); j++)

IF(PuzzleObj.areSame(currentNode.getChildren().get(j).getPosition(), closedList.get(i).getPosition()))

currentNode.getChildren().remove(currentNode.getChildren().get(j));

END IF

END FOR

END FOR

closedList.add(currentNode);

FOR(int i =0; i<currentNode.getChildren().size(); i++)

openList.add(currentNode.getChildren().get(i));

END FOR

currentNode = getMinNode(openList, getSolutionMatrix());

openList.remove(currentNode);

END WHILE

## Програмна реалізація

### Вихідний код

PuzzleObj.java

package com.puzzle;  
  
import java.util.ArrayList;  
  
public class PuzzleObj {  
  
  
 //Matrix with which you start  
 int[][] startMatrix = new int[3][3];  
  
 //Solved matrix  
 protected int[][] solutionMatrix =new int[3][3];  
  
  
 //solution  
 protected ArrayList<int[][]> solution= new ArrayList<>();  
  
  
  
  
  
 //Getters and Setters  
 public int[][] getStartMatrix() {  
 return startMatrix;  
 }  
  
 public void setStartMatrix(int[][] startMatrix) {  
 this.startMatrix = startMatrix;  
 }  
  
 public int[][] getSolutionMatrix() {  
 return solutionMatrix;  
 }  
  
 public void setSolutionMatrix(int[][] solutionMatrix) {  
 this.solutionMatrix = solutionMatrix;  
 }  
  
  
  
 public ArrayList<int[][]> getSolution() {  
 return solution;  
 }  
  
 public void setSolution(ArrayList<int[][]> solution) {  
 this.solution = solution;  
 }  
  
 //move empty cell  
 //moving methods returns false,  
 //when 0 can't be replaced in their directions  
  
 //move left  
 public static boolean moveLeft(int[][] currentMatrix){  
 int zeroX =-1, zeroY = -1;  
  
 for(int i = 0; i <3; i++){  
 for(int j = 0; j <3; j++) {  
 if (currentMatrix[i][j] == 0) {  
 zeroX = j;  
 zeroY = i;  
  
 }  
 }  
 }  
 if(zeroX-1<0){  
  
 return false;  
 }  
 else{  
 int curr = currentMatrix[zeroY][zeroX];  
 currentMatrix[zeroY][zeroX] = currentMatrix[zeroY][zeroX-1];  
 currentMatrix[zeroY][zeroX-1]= curr;  
 return true;  
 }  
 }  
  
 //move right  
 public static boolean moveRight( int[][] currentMatrix){  
 int zeroX =-1, zeroY = -1;  
  
 for(int i = 0; i <3; i++){  
 for(int j = 0; j <3; j++) {  
 if (currentMatrix[i][j] == 0) {  
 zeroX = j;  
 zeroY = i;  
  
 }  
 }  
 }  
 if(zeroX+1>2){  
  
 return false;  
 }  
 else{  
 int curr = currentMatrix[zeroY][zeroX];  
 currentMatrix[zeroY][zeroX] = currentMatrix[zeroY][zeroX+1];  
 currentMatrix[zeroY][zeroX+1]= curr;  
 return true;  
 }  
 }  
  
 //move up  
 public static boolean moveUp( int[][] currentMatrix){  
 int zeroX =-1, zeroY = -1;  
  
 for(int i = 0; i <3; i++){  
 for(int j = 0; j <3; j++) {  
 if (currentMatrix[i][j] == 0) {  
 zeroX = j;  
 zeroY = i;  
  
 }  
 }  
 }  
 if(zeroY-1<0){  
  
 return false;  
 }  
 else{  
 int curr = currentMatrix[zeroY][zeroX];  
 currentMatrix[zeroY][zeroX] = currentMatrix[zeroY-1][zeroX];  
 currentMatrix[zeroY-1][zeroX]= curr;  
 return true;  
 }  
 }  
  
 //move down  
 public static boolean moveDown( int[][] currentMatrix){  
 int zeroX =-1, zeroY = -1;  
  
 for(int i = 0; i <3; i++){  
 for(int j = 0; j <3; j++) {  
 if (currentMatrix[i][j] == 0) {  
 zeroX = j;  
 zeroY = i;  
  
 }  
 }  
 }  
 if(zeroY+1>2){  
 return false;  
 }  
 else{  
 int curr = currentMatrix[zeroY][zeroX];  
 currentMatrix[zeroY][zeroX] = currentMatrix[zeroY+1][zeroX];  
 currentMatrix[zeroY+1][zeroX]= curr;  
 return true;  
 }  
 }  
  
 //compare two matrixs 3x3  
 public static boolean areSame(int[][] firstMatrix, int[][] secondMatrix){  
 boolean flag = true;  
 for (int i = 0; i<3; i++){  
 for (int j = 0; j<3; j++){  
 if(firstMatrix[i][j] != secondMatrix[i][j]){  
 flag = false;  
 }  
 }  
 }  
 return flag;  
 }  
  
  
  
}

LDFS.java

package com.puzzle;  
  
public class LDFS extends PuzzleObj{  
  
 //limit for LDFS  
 private int limit;  
  
 public int getLimit() {  
 return limit;  
 }  
  
 public void setLimit(int limit) {  
 this.limit = limit;  
 }  
  
 //solving  
  
 public boolean solveLDFS(int[][] old\_matrix, int depth) {  
 //print matrix  
 for (int i = 0; i<3; i++){  
 for (int j = 0; j<3; j++){  
 System.*out*.print(old\_matrix[i][j] + " ");  
 }  
 System.*out*.print("\n");  
 }  
 System.*out*.print("\n");  
 //  
 int[][] matrix = new int[3][3];  
 for (int i = 0; i < 3; i++) {  
 System.*arraycopy*(old\_matrix[i], 0, matrix[i], 0, 3);  
 }  
 if (!*areSame*(matrix, solutionMatrix) && depth < limit) {  
  
 if (this.*moveLeft*(matrix)) {  
 if (solveLDFS(matrix, depth + 1)) {  
 solution.add(old\_matrix);  
 return true;  
 }  
 this.*moveRight*(matrix);  
 }  
  
 if (this.*moveUp*(matrix)) {  
 if (solveLDFS(matrix, depth + 1)) {  
 solution.add(old\_matrix);  
 return true;  
 }  
 this.*moveDown*(matrix);  
 }  
 if (this.*moveRight*(matrix)) {  
 if (solveLDFS(matrix, depth + 1)) {  
 solution.add(old\_matrix);  
 return true;  
 }  
 this.*moveLeft*(matrix);  
 }  
 if (this.*moveDown*(matrix)) {  
 if (solveLDFS(matrix, depth + 1)) {  
 solution.add(old\_matrix);  
 return true;  
 }  
 this.*moveUp*(matrix);  
 }  
 }else if(*areSame*(matrix, solutionMatrix)){  
 solution.add(matrix);  
 return true;  
 }  
 else {  
 System.*out*.print("Limit\n");  
 return false;  
 }  
 //solution.add(matrix);  
 return false;  
 }  
  
 public void printSolution(){  
 System.*out*.println("Solution:");  
 for (int k =0; k<solution.size(); k++ ){  
 System.*out*.println(k);  
 for(int i = 0; i <3; i++){  
  
 for(int j = 0; j <3; j++){  
 System.*out*.print( solution.get(k)[i][j]+" ");  
 }  
 System.*out*.print("\n");  
 }  
 }  
 }  
  
  
 //methods to print matrix  
 public void printMatrix(String name){  
 if("solutionmatrix".equals(name.toLowerCase())){  
 printSolutionMatrix();  
 }  
 else if("startmatrix".equals(name.toLowerCase())){  
 printStartMatrix();  
 }  
  
 }  
  
 private void printSolutionMatrix(){  
 System.*out*.println("Solution matrix:");  
 for(int i = 0; i <3; i++){  
 for(int j = 0; j <3; j++){  
 System.*out*.print( solutionMatrix[i][j]+" ");  
 }  
 System.*out*.print("\n");  
 }  
 }  
  
 private void printStartMatrix(){  
 System.*out*.println("Start matrix:");  
 for(int i = 0; i <3; i++){  
 for(int j = 0; j <3; j++){  
 System.*out*.print( startMatrix[i][j]+" ");  
 }  
 System.*out*.print("\n");  
 }  
 }  
  
  
}

NodeSecond.java

package com.puzzle;  
  
import java.util.ArrayList;  
  
public class NodeSecond {  
 //fields  
 private NodeSecond parent;  
 private int[][] position;  
 private ArrayList<NodeSecond> children;  
 private int score;  
 private int g;  
  
 //Getters and Setters  
 //SOME ARE UNUSUAL  
 public NodeSecond getParent() {  
 return parent;  
 }  
  
 public void setParent(NodeSecond parent) {  
 this.parent = parent;  
 }  
  
 public int[][] getPosition() {  
 return position;  
 }  
  
 public void setPosition(int[][] position) {  
 this.position = position;  
 }  
  
 public ArrayList<NodeSecond> getChildren() {  
 return children;  
 }  
  
 public void setChildren(ArrayList<NodeSecond> children) {  
 this.children = children;  
 }  
  
 public int getScore(int[][]solution) {  
 setScore(solution);  
 return score+g;  
 }  
  
 public void setScore(int score) {  
 this.score = score;  
 }  
  
 public int getG() {  
 return g;  
 }  
  
 public void setG(int g) {  
 this.g = g;  
 }  
  
 //only for first parent  
 public NodeSecond(int g){  
 this.g = g;  
 }  
  
 //constructor  
 public NodeSecond(int[][]position, NodeSecond parent, int g){  
 this.position = position;  
 this.parent = parent;  
 this.g = g;  
 this.children = new ArrayList<>();  
 }  
  
 //creating child-nodes (only if it is possible)  
 //by moving zero to different sides  
 public void doChildren(){  
 //moving left  
 if(PuzzleObj.*moveLeft*(position)){  
 NodeSecond prev = this.parent;  
 boolean unique =true;  
 while(prev.getG()!=-1){  
 if(PuzzleObj.*areSame*(prev.getPosition(), position)){  
 unique = false;  
 }  
 prev = prev.getParent();  
 }  
 int[][]matrix = new int[3][3];  
 for(int i =0; i<3; i++){  
 System.*arraycopy*(position[i], 0, matrix[i], 0, 3);  
 }  
 if(unique){  
 children.add(new NodeSecond(matrix, this, g+1));  
 }  
 PuzzleObj.*moveRight*(position);  
 }  
  
 //moving right  
 if(PuzzleObj.*moveRight*(position)){  
 NodeSecond prev = this.parent;  
 boolean unique =true;  
 while(prev.getG()!=-1){  
 if(PuzzleObj.*areSame*(prev.getPosition(), position)){  
 unique = false;  
 }  
 prev = prev.getParent();  
 }  
 int[][]matrix = new int[3][3];  
 for(int i =0; i<3; i++){  
 System.*arraycopy*(position[i], 0, matrix[i], 0, 3);  
 }  
 if(unique){  
 children.add(new NodeSecond(matrix, this, g+1));  
 }  
 PuzzleObj.*moveLeft*(position);  
 }  
  
 //move up  
 if(PuzzleObj.*moveUp*(position)){  
 NodeSecond prev = this.parent;  
 boolean unique =true;  
 while(prev.getG()!=-1){  
 if(PuzzleObj.*areSame*(prev.getPosition(), position)){  
 unique = false;  
 }  
 prev = prev.getParent();  
 }  
 int[][]matrix = new int[3][3];  
 for(int i =0; i<3; i++){  
 System.*arraycopy*(position[i], 0, matrix[i], 0, 3);  
 }  
 if(unique){  
 children.add(new NodeSecond(matrix, this, g+1));  
 }  
 PuzzleObj.*moveDown*(position);  
 }  
  
 //move down  
 if(PuzzleObj.*moveDown*(position)){  
 NodeSecond prev = this.parent;  
 boolean unique =true;  
 while(prev.getG()!=-1){  
 if(PuzzleObj.*areSame*(prev.getPosition(), position)){  
 unique = false;  
 }  
 prev = prev.getParent();  
 }  
 int[][]matrix = new int[3][3];  
 for(int i =0; i<3; i++){  
 System.*arraycopy*(position[i], 0, matrix[i], 0, 3);  
 }  
 if(unique){  
 children.add(new NodeSecond(matrix, this, g+1));  
 }  
 PuzzleObj.*moveUp*(position);  
 }  
 }  
  
  
 //score calculator (manhattan distance)  
 private void setScore(int [][]solution){  
 this.score = 0;  
 for (int i = 0; i< 3; i++){  
 for (int j = 0; j <3; j++){  
 for (int k = 0; k < 3; k++){  
 for (int l = 0; l<3; l++){  
 if(solution[k][l] == this.position[i][j] && solution[k][l] !=0){  
 this.score+= Math.*abs*(k-i)+ Math.*abs*(l-j);  
 }  
 }  
 }  
 }  
 }  
 }  
  
 //just to string method  
 @Override  
 public String toString() {  
 String result ="\n";  
 for (int i = 0; i < 3; i++) {  
  
 for (int j = 0; j < 3; j++) {  
 result+=position[i][j] + " ";  
 }  
 result+="\n";  
 }  
 return result;  
 }  
  
 //show are solution and node's position similar  
 public boolean solved(int[][]solutionPosition){  
 boolean flag = true;  
 for (int i = 0; i<3; i++){  
 for (int j = 0; j<3; j++){  
 if(position[i][j] != solutionPosition[i][j]){  
 flag = false;  
 }  
 }  
 }  
 return flag;  
 }  
  
}

AStar.java

package com.puzzle;  
  
  
  
  
import java.util.ArrayList;  
  
public class AStar extends PuzzleObj{  
  
 //lists with nodes, needed to algorithm  
 private ArrayList<int[][]> closedList;  
  
 //Getters and Setters  
 public ArrayList<int[][]> getClosedList() {  
 return closedList;  
 }  
  
  
 //method to get node with min score from array  
 //used for "openList"  
 private NodeSecond getMinNode(ArrayList<NodeSecond> children, int[][]solution){  
 int min = Integer.*MAX\_VALUE*;  
 int index = 0;  
 for(int i = 0; i< children.size(); i++){  
 if(children.get(i).getScore(solution)<min){  
 min = children.get(i).getScore(solution);  
 index = i;  
 }  
 }  
 return children.get(index);  
 }  
 public void solve(){  
  
 //make first node and it's parent  
 NodeSecond parent = new NodeSecond(-1);  
 NodeSecond currentNode = new NodeSecond(getStartMatrix(), parent, 0);  
  
 //make main arrays for a A\* algorithm  
 ArrayList<NodeSecond> closedList = new ArrayList<>();  
 ArrayList<NodeSecond> openList = new ArrayList<>();  
  
 //counter (only for watching what was done)  
 int counter = 0;  
 while(!currentNode.solved(getSolutionMatrix())){  
 //show current situation  
 if(openList.size()> counter){  
 System.*out*.println("Elements in open list:"+openList.size());  
 System.*out*.println("Elements in closed list:"+closedList.size());  
 counter +=1000;  
 }  
  
 //form children  
 currentNode.doChildren();  
  
 //remove children from parent list if they are in the closed list  
 for (int i = 0; i< closedList.size(); i++){  
 for (int j = 0; j< currentNode.getChildren().size(); j++){  
 if(PuzzleObj.*areSame*(currentNode.getChildren().get(j).getPosition(), closedList.get(i).getPosition())){  
 currentNode.getChildren().remove(currentNode.getChildren().get(j));  
 }  
 }  
  
 }  
  
 closedList.add(currentNode);  
 for(int i =0; i<currentNode.getChildren().size(); i++) {  
 openList.add(currentNode.getChildren().get(i));  
 }  
 currentNode = getMinNode(openList, getSolutionMatrix());  
 openList.remove(currentNode);  
 }  
  
 System.*out*.print("Solution:\n");  
 int steps = 0;  
 while(currentNode.getG()!=-1){  
 System.*out*.print(currentNode +"\n");  
 currentNode = currentNode.getParent();  
 steps+=1;  
 }  
 System.*out*.print("\nNumber of steps:"+steps);  
  
 System.*out*.print("\nOpen:"+openList.size());  
 System.*out*.print("\nClosedList:"+closedList.size());  
 }  
}

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.

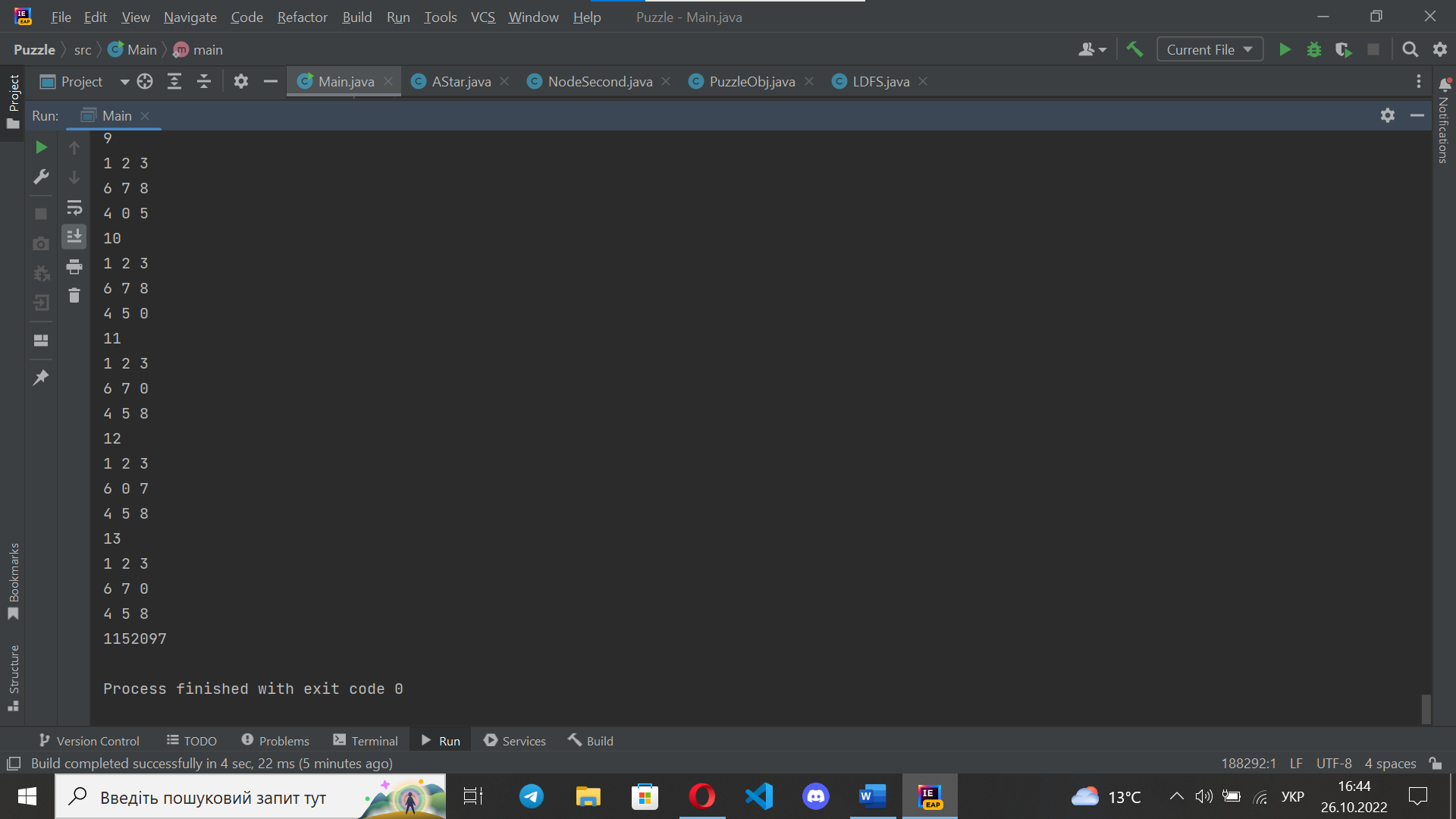


Рисунок 3.1 – Алгоритм…

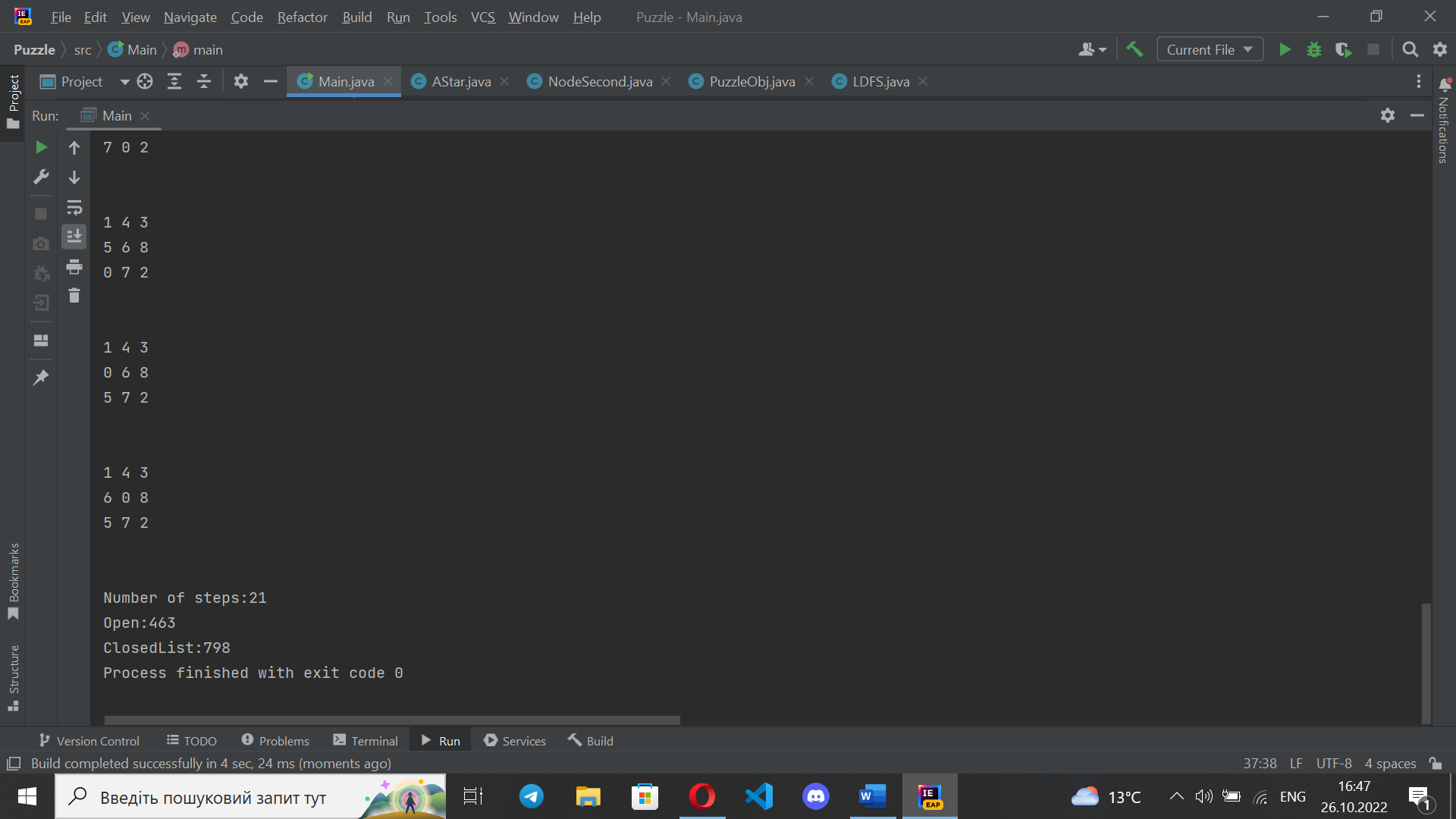


Рисунок 3.2 – Алгоритм …

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму LDFS, задачі 8-puzzle для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання 8-puzzle LDFS

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пам’яті |
| {1,2,3},  {6,7,0},  {4,5,8} | 1152097 | 362880 | 1152097 |
| {0,7,8},  {2,5,4},  {6,1,3} | 2396745 | 362880 | 2396745 |
| {1,2,3},  {4,5,8},  {6,0,7} | 89411 | 362880 | 89411 |
| {4,1,3},  {7,2,6},  {0,5,8} | 47 | 362880 | 47 |
| {5,4,2},  {1,8,3},  {7,0,6} | 380919 | 362880 | 380919 |
| {1,5,2},  {4,8,3},  {0,7,6} | 4525 | 362880 | 4525 |

В таблиці 3.2 наведені характеристики оцінювання алгоритму A\*, задачі 8-puzzle для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання 8-puzzle A\* Manhattan

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пам’яті |
| {5,3,4},  {8,2,0},  {1,7,6} | 230 | 362880 | 230 |
| {0,7,8},  {2,5,4},  {6,1,3} | 1065 | 362880 | 1065 |
| {5,1,3},  {7,8,6},  {4,2,0} | 606 | 362880 | 606 |
| {7,1,0},  {8,3,2},  {5,6,4} | 274 | 362880 | 274 |
| {5,4,1},  {7,0,8},  {3,2,6} | 1343 | 362880 | 1343 |
| {8,2,3},  {0,4,7},  {5,1,6} | 1528 | 362880 | 1528 |

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритми неінформативного та інформативного пошуку розв’язку 8-puzzle LDFS, A\* (Manhattan distance) відповідно. Було створено псевдокод цих алгоритмів та написано програмне забезпечення, що його відображає. Програма складається з 4 класів, один материнський з основними методами головоломки, два дочірні, що містять алгоритми пошуку розв’язку LDFS та A\* і допоміжний клас, для A\* - Node.

Під час тестування програми було виявлено, що евристична формула, що базується на Манхетенській відстані погано підходить для пошуку розв’язку даної задачі. Тому час пошуку часто перевищує 30 хвилин.

Неінформативний пошук LDFS з лімітом в 17 завжди завершить свою роботу при обмеженні часу 30 хвилин, проте якщо найкоротший шлях довший за 17 кроків – відповідь не буде знайдено.

Критерії оцінювання

За умови здачі лабораторної роботи до 23.10.2022 включно максимальний бал дорівнює – 5. Після 23.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 10%;
* програмна реалізація алгоритму – 60%;
* дослідження алгоритмів – 25%;
* висновок – 5%.