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) *ІП-14 Бабіч Денис* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе Ілля Елдарійович* _____
(прізвище, ім'я, по батькові)

Київ 2022

Зміст

1 Мета лабораторної роботи

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

2 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 A*.
- **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
- **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь A може стояти на одній лінії з ферзем B, проте між ними стоїть ферзь C; тому A не б’є B).
- **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 T 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

3 Виконання

1.1 Псевдокод алгоритмів

LDFS

```
function LDFS(labyrinth, limit) returns void:
    toVisit = Stack<Node>()
    next = List<Node>()

    if (Recursive_LDFS(labyrinth.StartNode, limit)) then
        PrintResult(labyrinth, true, toVisit.Count)
    else
        PrintResult(labyrinth, false, toVisit.Count)
    endif

function Recursive_LDFS(node, depth) returns bool:
        states += 1

        if (node == labyrinth.EndNode) then
            return true
        endif

        if (depth < 1) then
            return false
        endif

        iterations += 1

        if (node != labyrinth.StartNode) then
            node.Character = '*'
        endif

        labyrinth.Expand(node, next, out isBlindCorner)

        if (isBlindCorner) then
            blindCorners += 1
        endif

        if (next.Count < 1) then
            if (toVisit.Count < 1) then
                return false
            endif
        endif
        return Recursive_LDFS(toVisit.Pop(), --depth)
```

```
        if next.Count > 1 then
            for (i = 1, i < next.Count) do
                toVisit.Push(next[i])
            endloop
        endif

        return Recursive_LDFS(next[0], --depth)
```

RBFS

```
function RBFS(labyrinth) returns void:
    next = List<Node>()

    labyrinth.CalculateHeuristic()

    if (Recursive_RBFS(labyrinth.StartNode, float.MaxValue)) then
        PrintResult(labyrinth, true, statesInMemory)
    else
        PrintResult(labyrinth, false, statesInMemory)
    endif

function Recursive_RBFS(node, f_limit) returns bool:
        if (node == labyrinth.EndNode) then
            return true
        endif

        if (node != labyrinth.StartNode) then
            node.Character = '*'
        endif

        states += 1
        iterations += 1

        while (true) do
            labyrinth.Expand(node, next, out isBlindCorner)

            if (isBlindCorner) then
                blindCorners += 1
            endif
```

```
        if (next.Count < 1) then
            return false
        endif

        if (next.Count > 1) then
            next = next.OrderBy(node => node.Heuristic).ToList()
            statesInMemory += 1
        endif

        Node best = next[0]

        if (best.Heuristic > f_limit) then
            return false
        endif

        alternative = float.MaxValue

        if (next.Count > 1) then
            alternative = next[1].Heuristic
        endif

        result = Recursive_RBFS(best, Min(f_limit, alternative))

        if (result)
            return result
endloop
```

1.2 Програмна реалізація

1.2.1 Вихідний код

Program.cs

```
namespace SearchingAlgorithm
{
    internal sealed class Program
    {
        static void Main()
        {
            Console.InputEncoding = System.Text.Encoding.Unicode;
            Console.OutputEncoding = System.Text.Encoding.Unicode;

            string path = GetLabyrinthPath();

            Labyrinth labyrinth = new Labyrinth(path);
            labyrinth.CreateLabyrinth();
            Console.WriteLine(labyrinth);

            string algorithmName = ChooseAlgorithm().ToLower();

            switch (algorithmName)
            {
                case "rbfs":
                    SearchingAlgorithms.RBFS(labyrinth);
                    break;
                case "ldfs":
                    int limit = GetLimitValue();
                    SearchingAlgorithms.LDFS(labyrinth, limit);
                break;
            }
        }

        private static string GetLabyrinthPath()
        {
            string mazeName;
            string path;

            do
            {
                Console.Write("Введіть назву лабіринта, який бажаєте дослідити: ");

                mazeName = Console.ReadLine();
                path = $"Mazes\\{mazeName}.txt";

                if (File.Exists(path))
                    return path;

                Console.WriteLine("Обраного лабіринту не існує!");

            } while (true);
        }

        private static string ChooseAlgorithm()
        {
            string algorithmName = String.Empty;

            do
            {
             Console.Write("Введіть алгоритм пошуку, яким бажаєте скористатися (RBFS/LDFS):");
```

```
            algorithmName = Console.ReadLine();

                if (algorithmName.Equals("rbfs", StringComparison.OrdinalIgnoreCase)
                    || algorithmName.Equals("ldfs", StringComparison.OrdinalIgnoreCase))
                    return algorithmName;

                Console.WriteLine("Обраного алгоритму не існує!");

            } while (true);
        }

        private static int GetLimitValue()
        {
            int limit;

            do
            {
                Console.Write("Введіть глибину пошуку: ");

                if (Int32.TryParse(Console.ReadLine(), out limit))
                {
                    if (limit > 0)
                        return limit;

                    Console.WriteLine("Введене значення має бути більше 0!");
                }
                else
                    Console.WriteLine("Введене значення має бути числом!");

            } while (true);
        }
    }
}
```

Algorithms.cs

```
namespace SearchingAlgorithm
{
    internal static class SearchingAlgorithms
    {
        private static int iterations = 0;
        public static int Iterations { get => iterations; }

        private static int states = 0;
        public static int States { get => states; }

        private static int statesInMemory = 0;
        public static int StatesInMemory { get => statesInMemory; }

        private static int blindCorners = 0;
        public static int BlindCorners { get => blindCorners; }

        private static bool isBlindCorner;

        public static void LDFS(Labyrinth labyrinth, int limit)
        {
            Stack<Node> toVisit = new Stack<Node>();
            List<Node> next = new List<Node>();

            if (Recursive_LDFS(labyrinth.StartNode, limit))
                PrintResult(labyrinth, true, toVisit.Count);
            else
                PrintResult(labyrinth, false, toVisit.Count);
```

```
    bool Recursive_LDFS(Node node, int depth)
    {
        ++states;

        if (node == labyrinth.EndNode)
            return true;

        if (depth < 1)
            return false;

        ++iterations;

        if (node != labyrinth.StartNode)
            node.Character = '*';

        labyrinth.Expand(node, next, out isBlindCorner);

        if (isBlindCorner)
            ++blindCorners;

        if (next.Count < 1)
        {
            if (toVisit.Count < 1)
                return false;

            return Recursive_LDFS(toVisit.Pop(), --depth);
        }

        if (next.Count > 1)
        {
            for (int i = 1, length = next.Count; i < length; ++i)
                toVisit.Push(next[i]);
        }

        return Recursive_LDFS(next[0], --depth);
    }
}

public static void RBFS(Labyrinth labyrinth)
{
    List<Node> next = new List<Node>();
    labyrinth.CalculateHeuristic();

    if (Recursive_RBFS(labyrinth.StartNode, float.MaxValue))
        PrintResult(labyrinth, true, statesInMemory);
    else
        PrintResult(labyrinth, false, statesInMemory);

    bool Recursive_RBFS(Node node, float f_limit)
    {
        ++states;

        if (node == labyrinth.EndNode)
            return true;

        if (node != labyrinth.StartNode)
            node.Character = '*';

        ++iterations;

        while (true)
        {
            labyrinth.Expand(node, next, out isBlindCorner);
```

```
                    if (isBlindCorner)
                        ++blindCorners;

                    if (next.Count < 1)
                        return false;

                    if (next.Count > 1)
                    {
                        next = next.OrderBy(node => node.Heuristic).ToList();
                        statesInMemory += next.Count - 1;
                    }

                    Node best = next[0];

                    if (best.Heuristic > f_limit)
                        return false;

                    float alternative = float.MaxValue;

                    if (next.Count > 1)
                        alternative = next[1].Heuristic;

                    bool result = Recursive_RBFS(best, MathF.Min(f_limit, alternative));

                    if (result)
                        return result;
                }
            }
        }

        private static void PrintResult(Labyrinth labyrinth, bool found, int savedStates)
        {
            statesInMemory = savedStates;

            Console.Clear();
            Console.WriteLine(labyrinth);
            Console.WriteLine(found ? "\tЗнайдено!\n" : "\tНе знайдено!\n");
            Console.WriteLine($"\tКількість ітерації: {Iterations}\n
                              \tКількість глухих кутів: {BlindCorners}\n" +
    $"\tЗагальна кількість станів: {States}\n\tКількість станів у пам'яті: {StatesInMemory}");

            Reset();
        }

        private static void Reset()
        {
            states = 0;
            iterations = 0;
            blindCorners = 0;
            statesInMemory = 0;
        }
    }
}
```

Labyrinth.cs

```
    internal sealed class Node
    {
        public int X { get; set; }
        public int Y { get; set; }

        public char Character { get; set; }

        public float Heuristic { get; set; }
```

```
    public Node(int x, int y, char character)
    {
        this.X = x;
        this.Y = y;
        Character = character;
    }

    public override string ToString() => $"{this.Character}";
}


internal sealed class Labyrinth
{
    private Node[,] labyrinth;

    public string Path { get; set; }

    private int width;

    private int height;

    private string labyrinthAsText = String.Empty;

    public Labyrinth(string path) { this.Path = path; }

    public Node StartNode { get; private set; }

    public Node EndNode { get; private set; }

    public Node this[int rowIndex, int columnIndex]
    {
        get { return labyrinth[rowIndex, columnIndex]; }
        set { labyrinth[rowIndex, columnIndex] = value; }
    }

    public void CreateLabyrinth()
    {
        LabyrinthIntoString();

        string[] splittedText = labyrinthAsText.Split('\n');

        bool startSet = false, endSet = false;

        width = splittedText[0].Length;
        height = splittedText.Length;

        labyrinth = new Node[height, width];

        for (int i = 0; i < height; ++i)
        {
            for (int j = 0; j < width; ++j)
            {
                labyrinth[i, j] = new Node(i, j, splittedText[i][j]);

                if (!startSet)
                {
                    if (labyrinth[i, j].Character == 'S')
                    {
                        StartNode = labyrinth[i, j];
                        StartNode.Heuristic = 0;
                        startSet = true;
                    }
                }
```

```
            if (!endSet)
            {
                if (labyrinth[i, j].Character == 'F')
                {
                    EndNode = labyrinth[i, j];
                    endSet = true;
                }
            }
        }
    }
}

public void Expand(Node node, List<Node> nextNodes, out bool isBlindCorner)
{
    nextNodes.Clear();
    int wallsCounter = 0;

    if (node.X - 1 >= 0)
    {
        if (labyrinth[node.X - 1, node.Y].Character != '#')
        {
            if (labyrinth[node.X - 1, node.Y].Character != '*'
                  && labyrinth[node.X - 1, node.Y].Character != 'S')
                nextNodes.Add(labyrinth[node.X - 1, node.Y]);
        }
        else
            ++wallsCounter;
    }

    if (node.X + 1 < this.height)
    {
        if (labyrinth[node.X + 1, node.Y].Character != '#')
        {
            if (labyrinth[node.X + 1, node.Y].Character != '*'
                  && labyrinth[node.X + 1, node.Y].Character != 'S')
                nextNodes.Add(labyrinth[node.X + 1, node.Y]);
        }
        else
            ++wallsCounter;
    }

    if (node.Y + 1 < this.width)
    {
        if (labyrinth[node.X, node.Y + 1].Character != '#')
        {
            if (labyrinth[node.X, node.Y + 1].Character != '*'
                  && labyrinth[node.X, node.Y + 1].Character != 'S')
                nextNodes.Add(labyrinth[node.X, node.Y + 1]);
        }
        else
            ++wallsCounter;
    }

    if (node.Y - 1 >= 0)
    {
        if (labyrinth[node.X, node.Y - 1].Character != '#')
        {
            if (labyrinth[node.X, node.Y - 1].Character != '*'
                  && labyrinth[node.X, node.Y - 1].Character != 'S')
                nextNodes.Add(labyrinth[node.X, node.Y - 1]);
        }
        else
            ++wallsCounter;
    }
```

```
            isBlindCorner = wallsCounter > 2 ? true : false;
        }

        public void CalculateHeuristic()
        {
            for (int i = 0; i < height; ++i)
            {
                for (int j = 0; j < width; ++j)
                    labyrinth[i, j].Heuristic = CalculateEuclideanDistance(labyrinth[i, j],
                                                                           this.EndNode);
            }

            float CalculateEuclideanDistance(Node node, Node goal) =>
      MathF.Sqrt(MathF.Pow(node.X - goal.X, 2) + MathF.Pow(node.Y - goal.Y, 2));
        }

        private void LabyrinthIntoString()
        {
            string temp = String.Empty;

            using (StreamReader sr = new StreamReader(Path))
                temp = sr.ReadToEnd();



            for (int i = 0; i < temp.Length; ++i)
            {
                if (temp[i] != '\r')
                    labyrinthAsText += temp[i];
            }
        }

        public override string ToString()
        {
            string result = "\n";

            for (int i = 0; i < height; ++i)
            {
                result += '\t';
                for (int j = 0; j < width; ++j)
                    result += labyrinth[i, j];
                result += '\n';
            }

            return result;
        }
    }
```

1.2.2 Приклади роботи

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

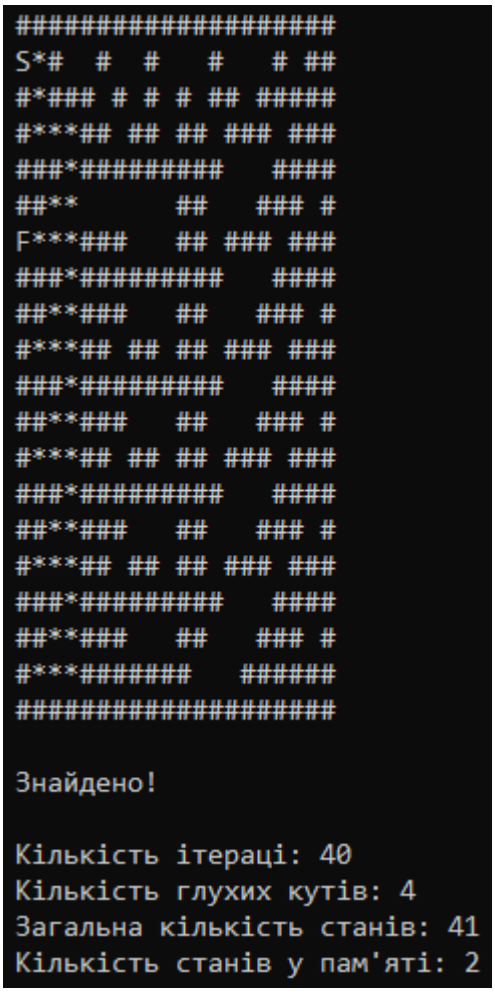

Рисунок 3.1 – Алгоритм LDFS

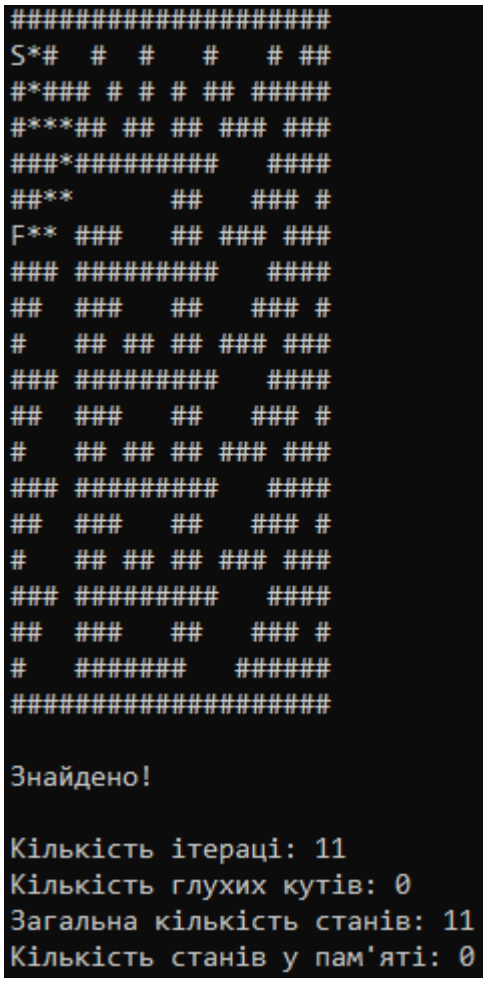

Рисунок 3.2 – Алгоритм RBFS

1.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму LDFS, задачі пошуку шляху у лабіринті для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання LDFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	24	1	25	9
Стан 2	40	5	41	4
Стан 3	50	4	51	12
Стан 4	34	3	35	13
Стан 5	40	4	41	2
Стан 6	28	1	29	5
Стан 7	40	4	41	10
Стан 8	41	5	42	4
Стан 9	42	1	43	5
Стан 10	24	1	25	1
Стан 11	24	0	25	10
Стан 12	33	4	34	3
Стан 13	31	3	32	4
Стан 14	35	2	36	8
Стан 15	19	0	20	6
Стан 16	49	6	50	12
Стан 17	40	5	41	9
Стан 18	25	3	26	4
Стан 19	34	2	35	7
Стан 20	40	4	41	1

В таблиці 3.2 наведені характеристики оцінювання алгоритму RBFS, задачі пошуку шляху у лабіринті для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання RBFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	22	0	23	11
Стан 2	26	1	27	9
Стан 3	29	0	30	18
Стан 4	37	3	38	20
Стан 5	11	0	12	3
Стан 6	22	0	23	9
Стан 7	30	0	31	16
Стан 8	24	1	25	6
Стан 9	25	0	26	9
Стан 10	23	0	24	2
Стан 11	24	0	25	10
Стан 12	29	1	30	7
Стан 13	23	0	24	9
Стан 14	19	0	20	10
Стан 15	19	0	20	6
Стан 16	32	1	33	20
Стан 17	23	0	24	15
Стан 18	24	1	25	8
Стан 19	30	1	31	12
Стан 20	11	0	12	2

Висновок

При виконанні даної лабораторної роботи було розглянуто виконання алгоритмів інформативного та неінформативного пошуків на основі алгоритмів RBFS та LDFS. Для реалізації інформативного пошуку була використана евристична функція знаходження Евклідової відстані між поточної точкою лабіринту та кінцевою координатою лабіринту.

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%.