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-13 Дойчев Костянтин* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олексій Олександрович* _____
(прізвище, ім'я, по батькові)

Київ 2023

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

3.1 Псевдокод алгоритмів

Алгоритм пошуку LDFS

```
ПОЧАТОК
    Рекурсивний пошук в глибину(Початковий вузол)
КІНЕЦЬ

ФУНКЦІЯ Рекурсивний пошук в глибину(node)
    bool cutoffOccured = false
    ЯКЩО кількість конфліктів у node дорівнює 0 ТО
        ПОВЕРНУТИ node, SUCCESS
    ВСЕ ЯКЩО
    ІНАКШЕ ЯКЩО глибина node дорівнює limit ТО
        ПОВЕРНУТИ null, CUTOFF
    КІНЕЦЬ ІНАКШЕ ЯКЩО
    Розкрити node.successors
    ДЛЯ successor В node.successors
        result = Рекурсивний пошук(successor, SOLVING, limit)
        ЯКЩО result[1] = CUTOFF ТО
            Встановити cutoffOccured значення true
        КІНЕЦЬ ЯКЩО
        ІНАКШЕ ЯКЩО result != FAILURE ТО
            ПОВЕРНУТИ result
        КІНЕЦЬ ІНАКШЕ
    КІНЕЦЬ ДЛЯ
    ЯКЩО cutoffOccured ТО
        ПОВЕРНУТИ null, CUTOFF
    КІНЕЦЬ ЯКЩО
    ПОВЕРНУТИ null, FAILURE
КІНЕЦЬ ФУНКЦІЇ
```

Алгоритм пошуку A Star

Пошук(node)

ПОЧАТОК

open = черга <node> з пріоритетом по вартості вузла

closed = множина <node>

Вставити node в open

ПОКИ довжина open не рівна 0 ТО

Витягнути з open елемент з найменшою вартістю current

ЯКЩО кількість конфліктів у current рівна 0 ТО

ПОВЕРНУТИ current, SUCCESS

КІНЕЦЬ ЯКЩО

Додати current у множину closed

Розкрити нащадки current.successors

ДЛЯ КОЖНОГО successor В current.successors

ЯКЩО множина closed НЕ містить стану successor ТО

Додати successor у чергу open

КІНЕЦЬ ЯКЩО

КІНЕЦЬ ДЛЯ

КІНЕЦЬ ПОКИ

ПОВЕРНУТИ null, FAILURE

КІНЕЦЬ

3.2 Програмна реалізація

3.2.1 Вихідний код

Program.cs

```
namespace Lab2;

class Program
{
```

```
    public static void Main(string[] args)
    {
        try
        {
            State startState = new State();
            Console.WriteLine("Start state:");
            Console.WriteLine(startState);
            foreach (var col in startState.field)
            {
                Console.Write($"{col} ");
            }
            Console.WriteLine();
            Node root = new Node(startState);
            SearchAlgorithm algorithm;
            Console.WriteLine("Choose LDFS or A*? [0/1]: ");
            int algoChoice = Int32.Parse(Console.ReadLine());
            if (algoChoice == 0)
            {
                Console.Write("Enter the limit for search: ");
                int limit = Int32.Parse(Console.ReadLine());
                algorithm = new LDFSAlgo(root, limit);
            }
            else
            {
                algorithm = new AStarAlgo(root);
            }

            Console.WriteLine(algorithm.DisplayResults());
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
        }
    }
}
```

State.cs

```
namespace Lab2;

public class State
{
    public byte[] field { get; set; }
    public static int Size = 8;
    public State()
    {
        field = new byte[Size];
        Random random = new Random();
        for (int i = 0; i < Size; i++)
            field[i] = (byte)random.Next(0, Size);
    }

    public State(State other, int column, byte newPosition)
    {
        field = new byte[other.field.Length];
        Array.Copy(other.field, field, other.field.Length);
        field[column] = newPosition;
    }

    public override string ToString()
    {
        string output = "";
        for (int i = 0; i < field.Length; i++)
```

```
        {
            for (int j = 0; j < field.Length; j++)
            {
                if (i == field[j])
                    output += "[Q]";
                else
                    output += "[ ]";
            }

            output += "\n";
        }

        return output;
    }

    public int F2()
    {
        int conflicts = 0;
        for (int i = 0; i < field.Length; i++)
        {
            for (int j = i + 1; j < field.Length; j++)
            {
                if (field[i] == field[j])
                    conflicts++;

                if (i - field[i] == j - field[j])
                    conflicts++;

                if (i + field[i] == j + field[j])
                    conflicts++;
            }
        }

        return conflicts;
    }
}
```

Node.cs

```
namespace Lab2;

public class Node
{
    private Node? parent;

    public State State { get; }
    public int Depth { get; }
    public Node[] Successors { get; set; }

    internal int Cost() => Depth + State.F2();

    public Node(State InitialState)
    {
        parent = null;
        State = InitialState;
        Depth = 0;
    }

    public Node(Node parentNode, int queenColumn, byte position)
    {
        parent = parentNode;
        Depth = parentNode.Depth + 1;
```

```
        State = new State(parentNode.State, queenColumn, position);
    }

    public override string ToString()
    {
        return State.ToString();
    }

    public static void Expand(Node node)
    {
        int index = 0;
        node.Successors = new Node[node.State.field.Length *
(node.State.field.Length - 1)];
        for (int i = 0; i < node.State.field.Length; i++)
        {
            for (byte j = 0; j < node.State.field.Length; j++)
            {
                if(node.State.field[i] == j)
                    continue;
                node.Successors[index++] = new Node(node, i, j);
            }
        }
    }
}
```

SearchAlgorighm.cs

```
namespace Lab2;

public abstract class SearchAlgorithm
{
    internal AlgoStatistics _statistics;

    public SearchAlgorithm()
    {
        _statistics = new AlgoStatistics();
    }

    public abstract Solution Search(Node problem);

    public string DisplayResults()
    {
        return _statistics.Results() + _statistics.Stats();
    }
}
```

LDFSAlgo.cs

```
namespace Lab2;

public class LDFSAlgo : SearchAlgorithm
{
    private HashSet<State> TotalStates;
    private HashSet<State> StatesInMemory;
    private readonly int _limit;
    public LDFSAlgo(Node node, int lim)
    {
        _limit = lim;
        TotalStates = new HashSet<State>();
        StatesInMemory = new HashSet<State>();
        _statistics.Stopwatch.Start();
```

```
        _statistics.Solution = Search(node);
        _statistics.Stopwatch.Stop();
        _statistics.StatesInMemory = StatesInMemory.Count;
        _statistics.TotalStates = TotalStates.Count;
    }


    public override Solution Search(Node node)
    {
        return RecursiveDLS(node);
    }

    private Solution RecursiveDLS(Node node)
    {
        ++_statistics.Iterations;
        if (_statistics.Stopwatch.ElapsedMilliseconds >=
AlgoStatistics.TimeLimit)
        {
            ++_statistics.DeadEnds;
            return new Solution(null, Status.TIME_EXCEED);
        }

        bool cutoffOccured = false;
        if (node.State.F2() == 0)
            return new Solution(node, Status.SUCCESS);
        if (node.Depth == _limit)
        {
            ++_statistics.DeadEnds;
            return new Solution(null, Status.CUTOFF);
        }
        TotalStates.Add(node.State);
        Node.Expand(node);
        foreach (var successor in node.Successors)
        {
            TotalStates.Add(successor.State);
            Solution result = RecursiveDLS(successor);

            if (result.status == Status.TIME_EXCEED)
                return result;

            if (result.status == Status.CUTOFF)
                cutoffOccured = true;

            else if (result.status != Status.FAILURE)
            {
                foreach (var succ in node.Successors)
                    StatesInMemory.Add(succ.State);

                return result;
            }
        }

        if (cutoffOccured)
            return new Solution(null, Status.CUTOFF);

        return new Solution(null, Status.FAILURE);
    }
}
```

AStarAlgo.cs

```
namespace Lab2;

public class AStarAlgo : SearchAlgorithm
{

    public AStarAlgo(Node problem)
    {
        _statistics = new AlgoStatistics();
        _statistics.Stopwatch.Start();

        _statistics.Solution = Search(problem);
        _statistics.Stopwatch.Stop();

    }

    public override Solution Search(Node problem)
    {
        if (_statistics.Stopwatch.ElapsedMilliseconds >=
AlgoStatistics.TimeLimit)
        {
            return new Solution(null, Status.TIME_EXCEED);
        }

        PriorityQueue<Node, int> open = new PriorityQueue<Node, int>();
        HashSet<State> closed = new HashSet<State>();
        open.Enqueue(problem, problem.State.F2());
        while (open.Count != 0)
        {
            ++_statistics.Iterations;
            Node current = open.Dequeue();
            if (current.State.F2() == 0)
            {
                _statistics.TotalStates = _statistics.StatesInMemory =
open.Count + closed.Count;
                return new Solution(current, Status.SUCCESS);
            }
            closed.Add(current.State);
            Node.Expand(current);
            foreach (var successor in current.Successors)
            {
                if(!closed.Contains(successor.State))
                    open.Enqueue(successor, successor.State.F2());
            }
        }

        return new Solution(null, Status.FAILURE);
    }

}
```

AlgoStatistics.cs

```
using System.Diagnostics;

namespace Lab2;

public class AlgoStatistics
{
    public Solution Solution { get; set; }
    public long Iterations { get; set; }
    public int DeadEnds { get; set; }
    public Stopwatch Stopwatch { get; }
```

```
    public long TotalStates { get; set; }
    public long StatesInMemory { get; set; }

    public static int TimeLimit = 1800000;

    public AlgoStatistics()
    {
        Iterations = 0;
        DeadEnds = 0;
        TotalStates = 0;
        StatesInMemory = 0;
        Stopwatch = new Stopwatch();
    }
    public string Results() => Solution.status switch
        {
            Status.SUCCESS => "Solution found successfully!\n" +
Solution.node.State,
            Status.CUTOFF => "The search was cut off.\n",
            Status.FAILURE => "The search has failed.\n",
            Status.TIME_EXCEED => "The search has exceeded the time limit.",
        };

    public string Stats() => $"Iterations: {Iterations}\nTime elapsed:
{Stopwatch.ElapsedMilliseconds} ms\nDead ends: {DeadEnds}\nTotal states:
{TotalStates}\nStates in memory: {StatesInMemory}" ;
}
```

Solution.cs

```
namespace Lab2;

public class Solution
{
    public Node? node { get; }
    public Status status { get; }

    public Solution(Node? node, Status status)
    {
        this.node = node;
        this.status = status;
    }
}
```

Status.cs

```
namespace Lab2;

public enum Status
{
    SUCCESS,
    CUTOFF,
    FAILURE,
    TIME_EXCEED
}
```

3.2.2 Приклади роботи

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

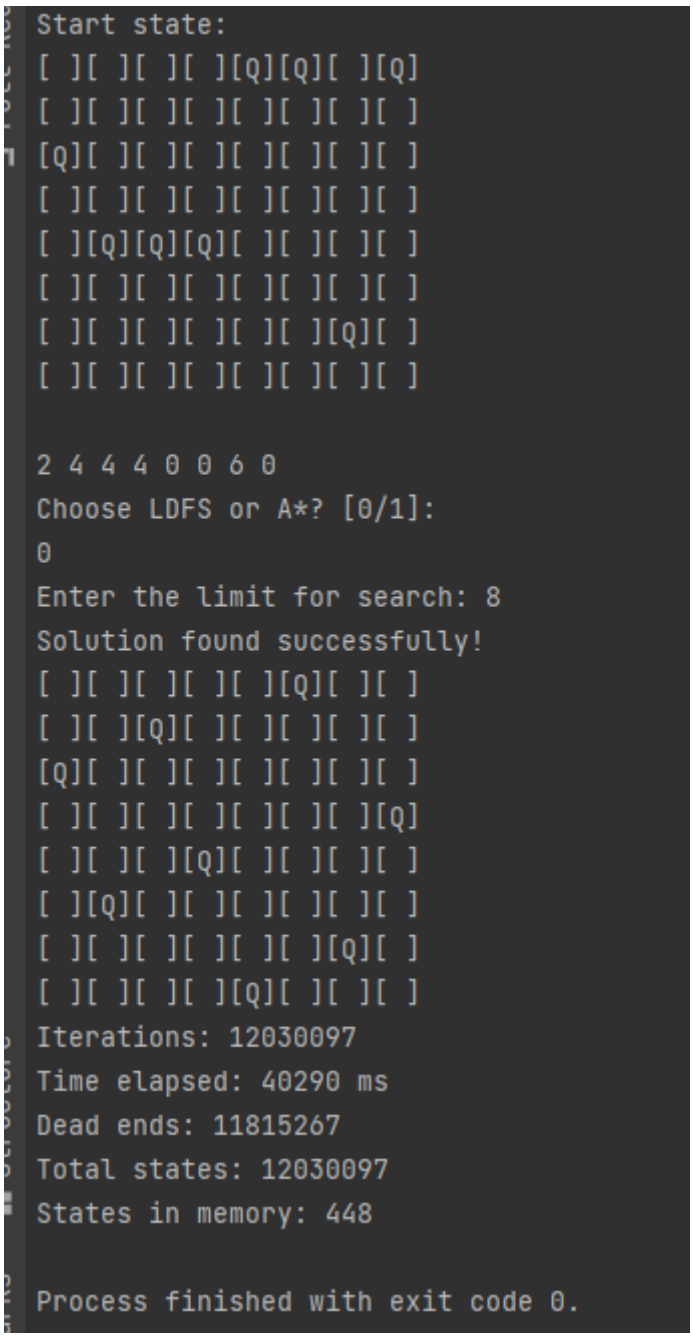

Рисунок 3.1 – LDFS

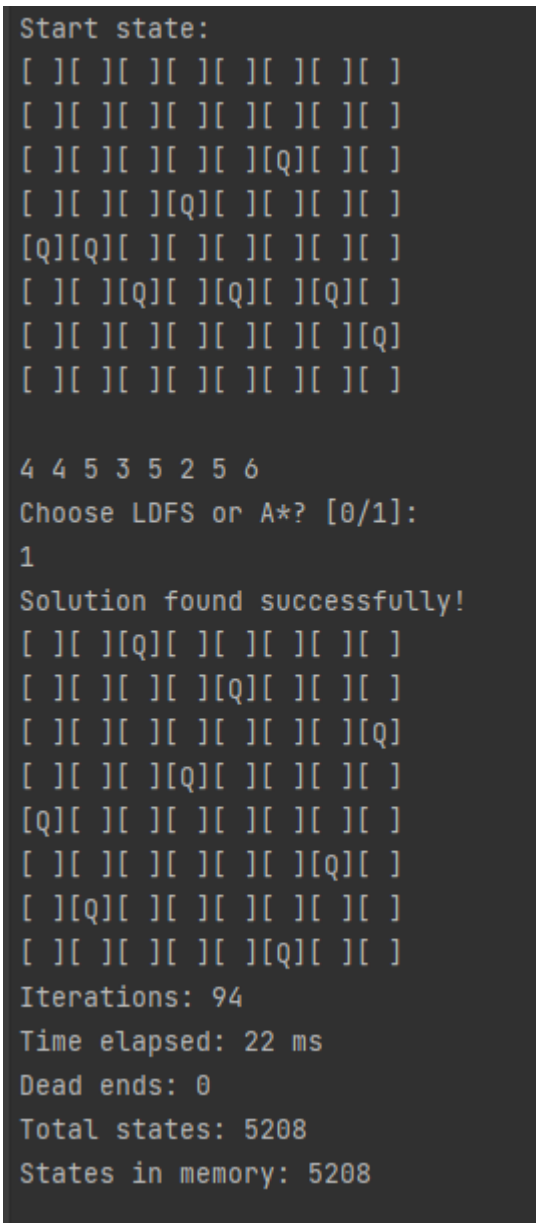

Рисунок 3.2 –A*

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму LDFS задачі 8 ферзів для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання LDFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
2 4 4 4 0 0 6 0	12030097	11815267	12030097	448
7 7 4 6 0 5 1 5	4224	4141	4224	448
4 7 6 0 2 7 2 7	749441	736051	749441	448
7 7 7 4 0 1 6 3	2221317	2181644	2221317	448
5 1 3 2 6 4 0 5	20407	20035	20407	448
3 5 6 1 7 7 7 6	13180379	12945009	13180379	448
6 0 6 4 2 7 4 4	1857826	1824644	1857826	448
4 0 7 0 6 5 3 3	12084712	11868907	12084712	448
3 2 5 3 1 4 4 2	32232717	31657126	32232717	448
0 6 0 3 2 2 0 3	32625118	32042520	32625118	448
7 3 7 1 2 3 3 7	32234987	31659355	32234987	448
6 0 1 0 6 3 2 7	31351848	30791987	31351848	448
5 0 7 1 6 2 7 5	21356953	20975572	21356953	448
2 0 6 2 0 2 3 5	257424	252820	257424	448
3 1 7 7 4 2 5 6	442287	434382	442287	448
6 5 0 4 1 4 5 7	491103	482327	491103	448
6 3 0 7 6 7 0 3	31737115	31170374	31737115	448
5 2 1 1 3 4 4 1	12045310	11830208	12045310	448
1 0 1 7 4 2 6 3	413782	406386	413782	448
1 2 3 1 1 6 5 1	32254379	31678401	32254379	448

Середня кількість ітерацій – 13479571.3

Середня кількість глухих кутів – 12238857.8

Середня кількість згенерованих станів - 13479571.3

Середня кількість станів у пам’яті – 448

В таблиці 3.2 наведені характеристики оцінювання алгоритму A* задачі 8 ферзів для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання A*

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
7 5 3 3 6 4 4 5	4	---	168	168
0 3 0 3 6 4 5 1	6	---	280	280
3 7 3 1 7 6 3 4	23	---	1232	1232
3 7 7 1 1 4 5 2	7	---	336	336
7 5 2 6 5 7 6 7	169	---	9408	9408
2 3 6 5 5 1 2 3	9	---	448	448
2 1 0 7 2 0 1 7	7	---	336	336
1 7 0 7 5 5 7 1	6	---	280	280
3 7 6 3 2 1 4 1	6	---	280	280
7 7 0 4 0 3 4 4	19	---	1008	1008
6 0 2 6 4 5 7 1	66	---	3640	3640
6 3 7 5 1 5 2 6	33	---	1792	1792
1 6 1 6 6 5 6 4	7	---	336	336
6 6 4 1 0 2 5 0	123	---	6832	6832
4 3 0 5 7 4 0 4	5	---	224	224
1 1 4 2 0 3 7 4	4	---	168	168
3 7 5 4 5 4 1 0	7	---	336	336
7 5 5 6 5 1 7 7	7	---	336	336
4 3 2 3 5 2 6 1	10	---	504	504
6 3 3 6 6 0 4 1	8	---	392	392

Середня кількість ітерацій – 26.3

Середня кількість глухих кутів ----

Середня кількість згенерованих станів – 1416.8

Середня кількість станів у пам’яті 1416.8

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто два алгоритми пошуку: LDFS та A Star.

LDFS – алгоритм неінформативного пошуку, не є повним. Для пошуку правильного рішення потребує багато часу і пам’яті.

A Star – алгоритм інформативного пошуку, що використовує евристичну функцію для оцінки вигіднішого нащадка для подальшого пошуку.

Порівняємо результати досліджень обох алгоритмів:

	LDFS	A Star
Середня кількість ітерацій	13479571.3	26.3
Середня кількість глухих кутів	12238857.8	----
Середня кількість згенерованих станів	13479571.3	1416.8
Середня кількість станів у пам’яті	448	1416.8

Бачимо, що A Star є набагато ефективнішим у вирішенні задач такого класу.

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%.