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»

“Проектування і аналіз алгоритмів зовнішнього сортування”

Виконав(ла) ІП-14 Ковалик Назар Миколайович 13.12.2022
(шифр, прізвище, ім’я, по батькові)

Перевірив _____
(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.

• LDFS – Пошук вглиб з обмеженням глибини.

• A* – Пошук A*.

• F2 – кількість пар ферзів, які б’ють один одного без урахування видимості.

№	Задача	АНП	АІП	АЛП	Func
8	8-ферзів	LDFS	A*		F2

3. ВИКОНАННЯ

1. Програмна реалізація алгоритму

1. Вихідний код

AZIROCHKA

Main.cs

```
using pa2;

var (queensCount, fieldSize) = ConsoleHelper.GetDataFromConsole();
var generatedQueens = Search.GenerateInitialQueens(queensCount, fieldSize);
var startTime = DateTime.Now;
ConsoleHelper.Draw(generatedQueens, fieldSize);
var solution = Search.FindThePath(queensCount, fieldSize);
Console.WriteLine((DateTime.Now - startTime).TotalMilliseconds);
ConsoleHelper.Draw(solution.Queens, fieldSize);
```

Queen.cs

```
namespace pa2;

public class Queen
{
    public int X;
    public int Y;

    public Queen(int x, int y)
    {
        X = x;
        Y = y;
    }

    public override string ToString()
    {
        return $"{X},{Y}";
    }
}
```

Search.cs

```
using pa.Common;

namespace pa2;

public static class Search
{
    public static State FindThePath(int queensCount, int fieldSize)
    {
                    while (true)
                    {
                            int? minIdx = null;
                            var minVal = int.MaxValue;
```

```
                        for (var i = 0; i < ChessMap.States.Count; i++)
                        {
                                var state = ChessMap.States[i];
                                var val = state.Depth + state.Actions[0].HitsCount;
                                if (val < minVal)
                                {
                                        minVal = val;
                                        minIdx = i;
                                }
                        }

                        if (minIdx is not null)
                        {
                                var possibleSolution = ChessMap.States[(int)minIdx].CreateDirevative((int)minIdx,
                                        queensCount, fieldSize);
                                if (possibleSolution is not null)
                                {
                                        return possibleSolution;
                                }
                        }
                }
        }

        public static List<Queen> GenerateInitialQueens(int queensCount, int fieldSize)
    {
                if (!ConsoleHelper.IsInRange((uint)queensCount, (uint)fieldSize))
      {
                        throw new InvalidDataException();
      }

                var newState = new State();
                var queens = newState.Queens;
                var map = newState.Map = new List<int>(8 * 8);
                for (int j = 0; j < 64; j++)
                {
                        map.Add(0);
                }

                var i = queensCount;
                while (i > 0)
                {
                        var x = queensCount - i;
                        var y = Random.Shared.Next(0, fieldSize - 1);

                        ChessMap.UpgradeMap(map, x, y, 1, fieldSize);
                        queens.Add(new(x, y));
                        i--;
                }

                newState.GenerateActions(queensCount, fieldSize);
                ChessMap.States.Add(newState);
                return queens;

        }


}

namespace pa2;

public class State
{
        public int Depth = 0;
        public List<int> Map = new();
        public List<Action> Actions = new();
        public List<Queen> Queens = new();
        public Action? FirstPreviousAction = null;

        public bool IsSolution(int queensCount)
        {
                for (int q = 0; q < queensCount; q++)
```

```
                {
                        int x = Queens[q].X;
                        int y = Queens[q].Y;
                        if (Map[y * 8 + x] >= 2)
    {
                                return false;
    }
                }

                return true;
        }

        public void GenerateActions(int queensCount, int fieldSize)
        {
                for (int q = 0; q < queensCount; q++)
                {
                        int x = Queens[q].X;
                        int y = Queens[q].Y;
                        for (int i = 0; i < fieldSize; i++)
                        {
                                if (y != i)
                                {
                                        Actions.Add(new(Map[i * 8 + x], q, x, i));
                                }
                        }
                }

                Actions.Sort((x, y) => x.HitsCount - y.HitsCount);
        }

public State? CreateDirevative(int index,int queensCount, int fieldSize)
        {
                var action = Actions[0];
                Actions.RemoveAt(0);
                var newState = new State();
                newState.Depth = Depth + 1;
                newState.FirstPreviousAction = action;
                foreach(var item in Queens)
    {
                        newState.Queens.Add(new(item.X, item.Y));
    }

                newState.Map = new(Map);
                ChessMap.UpgradeMap(newState.Map, newState.Queens[action.QueenNumber], -1, fieldSize);
                newState.Queens[action.QueenNumber].X = action.Ox;
                newState.Queens[action.QueenNumber].Y = action.Oy;
                var sorted = new List<Queen>(newState.Queens);

                sorted.Sort((q, b) => (q.X + q.Y * fieldSize)
                - (b.X + b.Y * fieldSize));

                string hash = string.Join("-", sorted);
                if (!ChessMap.Set.Contains(hash))
                {
                        ChessMap.Set.Add(hash);
                        ChessMap.UpgradeMap(newState.Map, action.Ox, action.Oy, 1,fieldSize);
                        if (newState.IsSolution(fieldSize))
                        {
                                return newState;
                        }

                        newState.GenerateActions(queensCount, fieldSize);
                        ChessMap.States.Add(newState);
                }

                if (Actions.Count == 0)
    {
                        ChessMap.States.RemoveAt(index);
    }

                return null;
```

```
            }
}

namespace pa2;

public class Constants
{
    public const int QUEENS_COUNT_MAX = 8;
    public const int QUEENS_COUNT_MIN = 2;
    public const int FIELD_SIZE_MIN = 3;
    public const int FIELD_SIZE_MAX = 8;
}
using pa2;

namespace pa2;

public class ConsoleHelper
{
    public static void Draw(List<Queen> queens, int fieldSize)
    {
        for (int i = 0; i < fieldSize; i++)
        {
            for (int j = 0; j < fieldSize; j++)
            {
                if (queens.Any(q => q.X == j && q.Y == i))
                {
                    Console.Write("Q");
                }
                else
                {
                    Console.Write("*");
                }

                Console.Write('\t');
            }

            Console.WriteLine('\n');
        }
    }
    public static (int, int) GetDataFromConsole()
    {
        Console.WriteLine("Welcome to the lab 2");
        while (true)
        {
            Console.WriteLine("Enter the queens count: ");
            var queensAsString = Console.ReadLine();
            Console.WriteLine("Enter the field size: ");
            var fieldSizeAsString = Console.ReadLine();
            if (IsNullOrWhiteSpace(queensAsString, fieldSizeAsString))
            {
                continue;
            }

            if (!uint.TryParse(queensAsString?.Trim(), out var queensCount))
            {
                Console.WriteLine($"Wrong format of {queensAsString}");
            }

            if (!uint.TryParse(fieldSizeAsString?.Trim(), out var fieldSize))
            {
                Console.WriteLine($"Wrong format of {fieldSizeAsString}");
            }

            if (IsInRange(queensCount, fieldSize))
            {
                return ((int)queensCount, (int)fieldSize);
            }

            Console.WriteLine($"Max values are: {Constants.FIELD_SIZE_MAX}" +
                $" and {Constants.QUEENS_COUNT_MAX}");
        }
```

```
    }

    public static bool IsInRange(uint queensCount, uint fieldSize)
    {
        bool queensInRange = queensCount >= Constants.QUEENS_COUNT_MIN &&
            queensCount <= Constants.FIELD_SIZE_MAX;
        bool fieldSizeInRange = fieldSize >= Constants.FIELD_SIZE_MIN &&
            fieldSize <= Constants.FIELD_SIZE_MAX;
        return queensInRange && fieldSizeInRange;
    }

    private static bool IsNullOrWhiteSpace(params string?[] values)
    {
        return values.Any(x => string.IsNullOrWhiteSpace(x));
    }
}
namespace pa2;

public static class ChessMap
{
        public static HashSet<string> Set = new();
        public static List<State> States = new();
        public static void UpgradeMap(List<int> map, Queen queen, int value, int fieldSize)
    {
                UpgradeMap(map, queen.X, queen.Y, value, fieldSize);
    }

    public static void UpgradeMap(List<int> map, int x, int y, int value, int fieldSize)
    {

                map[y * 8 + x] += value;
                for (int i = 0; i < fieldSize; i++)
                {
                        if (x != i)
                        {
                                map[y * 8 + i] += value;
                        }
                }

                for (int i = 0; i < fieldSize; i++)
                {
                        if (y != i)
                        {
                                map[i * 8 + x] += value;
                        }
                }

                int z = y - x;
                for (int i = 0; i < fieldSize; i++)
                {
                        if (x != i && z + i >= 0 && z + i < fieldSize)
                        {
                                map[(z + i) * 8 + i] += value;
                        }
                }

                z = y + x;
                for (int i = 0; i < fieldSize; i++)
                {
                        if (x != i && z - i >= 0 && z - i < fieldSize)
                        {
                                map[(z - i) * 8 + i] += value;
                        }
                }
        }
}
```

Action.cs

```
namespace pa2;

public record Action(int HitsCount, int QueenNumber,int Ox, int Oy);
```

LDFS

```
namespace pa2.Ldfs;

public class Queen
{
    public Queen(int x, int y, int wrong)
    {
        X = x;
        Y = y;
        Wrong = wrong;
    }

    public int X { get; set; }
    public int Y { get; set; }
    public int Wrong { get; set; }

    public void Deconstruct(out int x, out int y, out int wrong)
    {
        x = X;
        y = Y;
        wrong = Wrong;
    }
}
```

```
using pa2.Ldfs;


var (queensCount, fieldSize) = ConsoleHelper.GetDataFromConsole();
var ldfs = new LdfsSearch(queensCount, fieldSize);
ldfs.GenerateInitialQueens();
Console.WriteLine("The generated board: ");
ConsoleHelper.Draw(ldfs.Queens, 6);
var solution = ldfs.Search();
Console.WriteLine("The found solution");
ConsoleHelper.Draw(solution, 6);
```

```
namespace pa2.Ldfs;

public class LdfsSearch
{
    private int _totalWrong = 0;
    private bool _isSolved = false;
    private readonly int _maxDepth = 32;
    private readonly List<Queen> _queens = new();
    private readonly int[][] _playfield = new int[8][];

            public List<Queen> Queens => _queens;
            public int[][] PlayField => _playfield;

    private readonly int _fieldSize;
    private readonly int _queensCount;

    public LdfsSearch(int fieldSize, int queensCount)
    {
        _fieldSize = fieldSize;
        _queensCount = queensCount;

        for (int i = 0; i < 8; i++)
        {
```

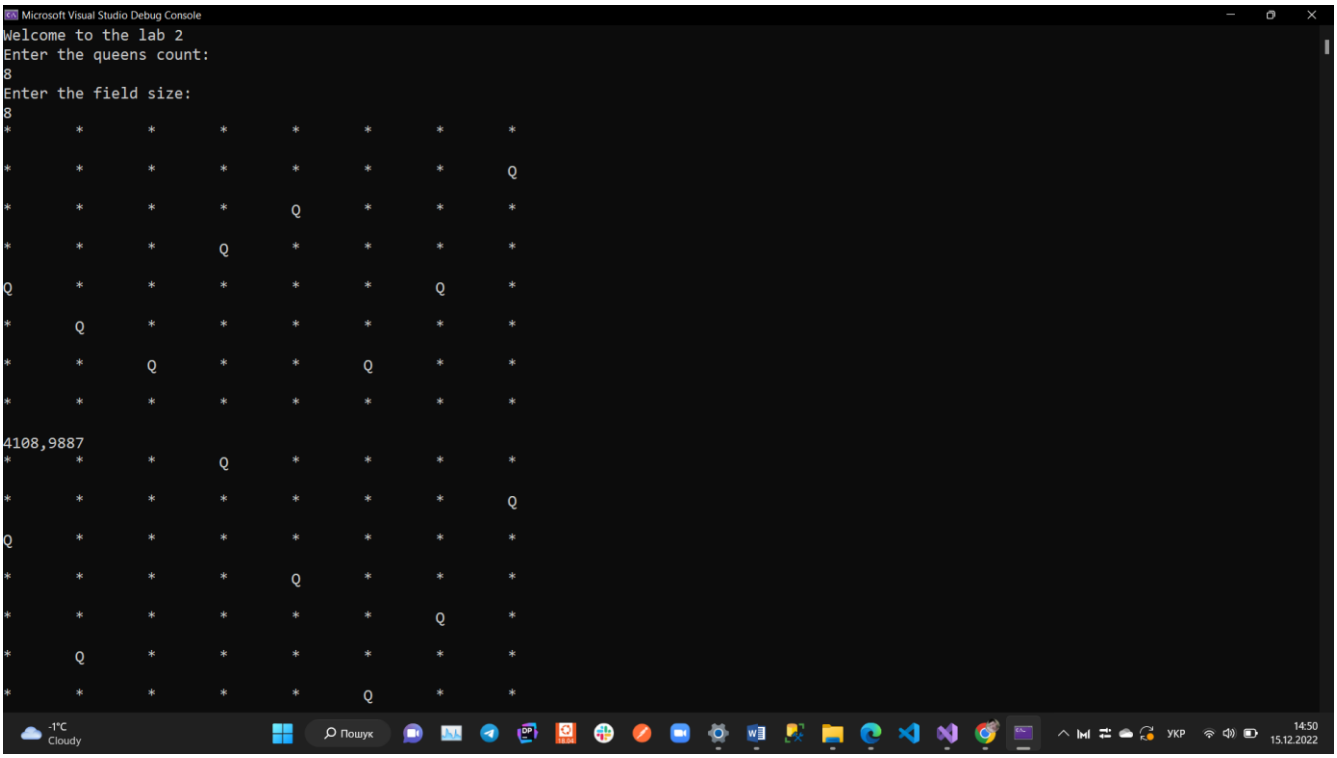
```
            _playfield[i] = new int[8];
        }
    }

    public List<Queen> Search()
    {
        for (var q = 0; q < _queensCount; q++)
        {
            var x = _queens[q].X;
            var y = _queens[q].Y;
            for (var i = 0; i < _fieldSize; i++)
            {
                if (y != i)
                {
                    MoveRec(q, x, i, 0);
                }
            }
        }

        if (_isSolved)
        {
            return _queens;
        }

        throw new Exception("The solution was not found");
    }

    public void GenerateInitialQueens()
    {
        var add = _queensCount;
        while (add > 0)
        {
            var x = _queensCount - add;
            var y = Random.Shared.Next(0, _fieldSize - 1);
            _playfield[y][x] = 1;
            var wrong = 0;
            for (var i = 0; i < _queens.Count; i++)
            {
                var x2 = _queens[i].X;
                var y2 = _queens[i].Y;
                if (x == x2 || y == y2 || x + y == x2 + y2 || x - y == x2 - y2)
                {
                    _queens[i].Wrong++;
                    wrong++;
                    _totalWrong++;
                }
            }

            _totalWrong += wrong + 1;
            _queens.Add(new(x, y, wrong + 1));
            add--;
        }
    }

        private void MoveRec(int queen, int x, int y, int depth)
        {
                if (depth >= _maxDepth || _isSolved)
                {
                        //ThreadPool.QueueUserWorkItem(_ => Console.Write(depth));
                        return;
                }

                if (_playfield[y][x] == 1)
                {
                        return;
                }

                var (oldx, oldy, oldw) = _queens[queen];
                _queens[queen].X = x;
                _queens[queen].Y = y;
                _playfield[y][x] = 1;
```

```
                _playfield[oldy][oldx] = 0;
                _totalWrong -= oldw;
                UpdateWrong(queen, x, y, oldx, oldy, out var wrong);
                _queens[queen].Wrong = wrong + 1;
                _totalWrong += wrong + 1;
                if (_totalWrong == _queensCount)
                {
                        _isSolved = true;
                        return;
                }

                depth++;
                for (var q = 0; q < _queensCount; q++)
                {
                        var (x1, y1, _) = _queens[q];
                        if (q == queen)
                        {
                                continue;
                        }

                        for (var i = 0; i < _fieldSize; i++)
                        {
                                if (y1 != i)
                                {
                                        MoveRec(q, x1, i, depth);
                                }
                        }
                }

                if (_isSolved)
                {
                        return;
                }

                _queens[queen].X = oldx;
                _queens[queen].Y = oldy;
                _queens[queen].Wrong = oldw;
                _playfield[y][x] = 0;
                _playfield[oldy][oldx] = 1;
                _totalWrong -= wrong + 1;
                _totalWrong += oldw;


                UpdateWrong(queen, oldx, oldy, x, y, out _);
        }

        private void UpdateWrong(int queen, int x0, int y0, int x1,int y1, out int  wrong)
{
                wrong = 0;
                for (var i = 0; i < _queensCount; i++)
                {
                        if (i == queen)
                        {
                                continue;
                        }

                        var (x2, y2, _) = _queens[i];
                        if (AreEqualDirections(x0, y0, x2, y2))
                        {
                                _queens[i].Wrong++;
                                ++wrong;
                                _totalWrong++;
                        }

                        if (AreEqualDirections(x1, y1, x2, y2))
                        {
                                _queens[i].Wrong--;
                                _totalWrong--;
                        }
                }
        }
```

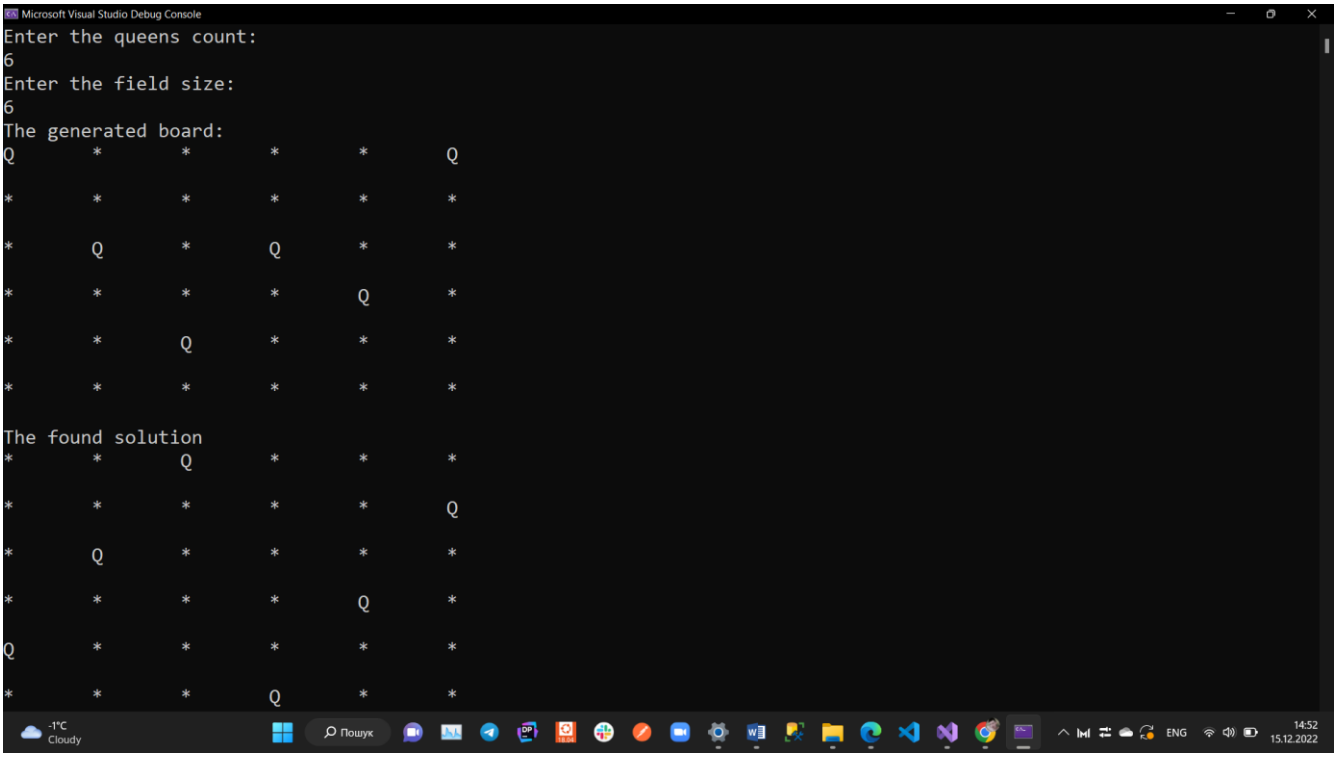
```
    private static bool AreEqualDirections(int x1, int y1, int x2, int y2) =>
                        x1 == x2 || y1 == y2 || x1 + y1 == x2 + y2 || x1 - y1 == x2 - y2;
}
```

2. Виконання

A зірочка:



LDFS



3. Дослідження алгоритмів

	LDFS	A*
Повний	ні якщо L<d	так
Час	O(b ^ L)	
Простір	O(b * L)	
Оптимальний	ні якщо L>d	
Основна ідея (глобально)	Модифікація DFS з доданим лімітом. Алгоритм DFS передбачає якнайшвидше заглиблення до кінцевих вузлів, їх обробку та повернення на 1 рівень вгору тільки коли всі гілки були оброблені. Пошук в глибину може бути розглянуто як окремий випадок пошуку з обмеженням глибину при якому L=∞	Цей алгоритм на кожному кроці обирає для обробки вузол, що має найменшу суму пройденого шляху до нього та евристичної функції, що зазвичай є відстанню до цілі.
Основна ідея (тут)	Для знаходження шляху відбувається перебір всіх можливих комбінацій ходів на кожному з кроків до знаходження першого вирішення. Кількість кроків дорівнює ліміту.	На кожному кроці можна здвинути один з 8ми ферзів на будь-яку допустиму кількість клітинок до краю у одному з 8ми напрямів. Програма зберігає всі розгорнуті стани і з них на кожному кроці обирає який наступний стан розгорнути обираючи дію з мінімальною сумою кількості видимих ферзів з клітинки та глибини стану.

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

Таблиця 3.1

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	451463	383469	178462987637760	1
Стан 2	143043	130364	178462987637760	1

Станн 3	7468910	7221266	178462987637760	1

Стан 4	2749096	2594734	178462987637760	1
Стан 5	1567190	1722785	178462987637760	1
Стан 6	378522	360907	178462987637760	1
Стан 7	1678921	1964462	178462987637760	1
Стан 8	924684	885030	178462987637760	1
Стан 9	648433	831504	178462987637760	1
Стан 10	81334	79669	178462987637760	1
Стан 11	1989270	1948618	178462987637760	1
Стан 12	2285188	2239036	178462987637760	1
Стан 13	191950	481905	178462987637760	1
Стан 14	285306	274576	178462987637760	1
Стан 15	539012	625565	178462987637760	1
Стан 16	248740	243456	178462987637760	1
Стан 17	47921	46938	178462987637760	1
Стан 18	391002	383017	178462987637760	1
Стан 19	188076	184233	178462987637760	1
Стан 20	925029	906147	178462987637760	1

В таблиці 3.1 наведені характеристики оцінювання алгоритму A* задачі 8-ферзів для 20 початкових станів.

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	6617	0	158077	3319
Стан 2	5959	0	16015	259
Стан 3	24605	0	457693	3665
Стан 4	25589	0	656431	4530
Стан 5	3856	0	238177	5054
Стан 6	5168	0	145163	3312
Стан 7	12639	0	430522	6740
Стан 8	35730	0	717149	5875
Стан 9	513	0	11034	229
Стан 10	46580	0	13039	295
Стан 11	42612	0	198273	3218
Стан 12	56379	0	14482	440
Стан 13	63594	0	13955	252
Стан 14	42785	0	205291	4136

Стан 15	56102	0	325004	5188
Стан 16	76332	0	31743	316
Стан 17	95034	0	254266	5159
Стан 18	98551	0	227556	5896
Стан 19	134742	0	334088	5681
Стан 20	147517	0	261503	3307

4. ВИСНОВОК

При виконанні лабораторної роботи було ознайомлено з різними алгоритмами пошуку, а також детально розглянуто неінформативний алгоритм пошуку в глибину з обмеженням глибини LDFS та інформативний алгоритм пошуку A*. Для них був написаний код та були проведені заміри обох алгоритмів. В результаті було виявлено, що випадку проблеми 8 ферзів, алгоритм LDFS, не дивлячись на те, що він іноді може знайти вирішення дуже швидко, через те що він перебирає всі можливі комбінації кроків у багатьох випадках займає занадто багато часу. У порівнянні з ним, A* працює швидше і може знайти шлях за адекватну кількість часу для більшої кількості початкових станів, але в обмін на це витрачає багато оперативної пам’яті.

5. КРИТЕРІЇ ОЦІНЮВАННЯ

У випадку 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%.