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-15, Плугатирьов Д.В.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе І.Е.* _____
(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: $T = 1000 - k \cdot t$, 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

LDFS

```
SolveLDFS(puzzleBoard) returns PuzzleBoard
    if ValidateGoalState(puzzleBoard) then
        return puzzleBoard
    end if
    if puzzleBoard.Depth == depthLimit then
        return null
    end if
    puzzleBoard.GenerateChildren()
    for each child in puzzleBoard.Children do
        result = SolveLDFS(child)
        if result != null then
            return result
        end if
    end for each
```

A*

```
SolveAstar(puzzleBoard) return PuzzleBoard
    open = PriorityQueue<PuzzleBoard, int>()
    closed = HashSet<PuzzleState>()
    open.Enqueue(puzzleBoard, 0)
    while open.Count != 0 do
        currentBoard = open.Dequeue()
        closed.Add(currentBoard.PuzzleState)
        if ValidateGoalState(currentBoard.PuzzleState) then
            return currentBoard
        end if
        if currentBoard.Depth == depthLimit then
```

```
            return null
        end if
        currentBoard.GenerateChildren()
        for each child in currentBoard.Children do
            if closed.Contains(child.PuzzleState) then
                continue
            end if
            open.Equeue(child, child.GetOutlay())
        end for each
    end while
    return null
```

3.2 Програмна реалізація

3.2.1 Вихідний код

```
using EightPuzzleLogic.Boards;

namespace EightPuzzleLogic.Algorithms
{
    public class AstarSolver : IEightPuzzleSolving, ICharacteristicable
    {
        private readonly PriorityQueue<IAstarPuzzleBoard, int> _open =
new();
        private readonly HashSet<int[][]> _closed = new();
        private readonly IEightPuzzleValidator _puzzleValidator;

        public int IterationsCount { get; set; }
        public int BlindCornersCount { get; set; }
        public int OverallStatesCount { get; set; }
        public int StatesCountInMemory { get; set; }

        public AstarSolver(IEightPuzzleValidator puzzleValidator)
        {
            _puzzleValidator = puzzleValidator;
        }

        /// <exception cref="InsufficientMemoryException"></exception>
```

```
        /// <exception cref="ArgumentNullException"></exception>
        public IPuzzleBoard SolveEightPuzzle(IPuzzleBoard puzzleBoard)
        {
            if (puzzleBoard is null)
            {
                throw new ArgumentNullException(nameof(puzzleBoard));
            }

            _open.Enqueue(puzzleBoard as IAstarPuzzleBoard, 0);

            while (_open.Count != 0)
            {
                StatesCountInMemory--;
                var currentBoard = _open.Dequeue();
                _closed.Add(currentBoard.PuzzleState);

                IterationsCount++;

                if
(_puzzleValidator.ValidateGoalStateReaching(currentBoard.PuzzleState))
                {
                    return currentBoard;
                }

                currentBoard.GenerateChildren();

                long bytesCountInGb = (long)Math.Pow(1024, 3);

                if (GC.GetTotalMemory(false) >= bytesCountInGb)
                {
                    throw new InsufficientMemoryException("The available
memory has been exceeded");
                }

                foreach (var child in
currentBoard.Children.Cast<IAstarPuzzleBoard>())
                {
                    OverallStatesCount++;

                    if (_closed.Contains(child.PuzzleState))
                    {
                        continue;
                    }
```

```
                    StatesCountInMemory++;

                    _open.Enqueue(child, child.GetOutlay());
                }
            }

            return null;
        }
    }
}
```

```
namespace EightPuzzleLogic.Algorithms
{
    public interface ICharacteristicable
    {
        public int IterationsCount { get; set; }
        public int BlindCornersCount { get; set; }
        public int OverallStatesCount { get; set; }
        public int StatesCountInMemory { get; set; }
    }
}
```

```
namespace EightPuzzleLogic.Algorithms;

public interface IDepthLimited
{
    public int DepthLimit { get; }
}
```

```
using EightPuzzleLogic.Boards;

namespace EightPuzzleLogic.Algorithms
{
    public interface IEightPuzzleSolving
    {
        public IPuzzleBoard SolveEightPuzzle(IPuzzleBoard puzzleBoard);
    }
}
```

```
using EightPuzzleLogic.Boards;
```

```
namespace EightPuzzleLogic.Algorithms
{
    public class LdfsSolver : IEightPuzzleSolving, IDepthLimited,
ICharacteristicable
    {
        private readonly IEightPuzzleValidator _puzzleValidator;

        private int _depthLimit;
        /// <exception cref="ArgumentOutOfRangeException"></exception>
        public int DepthLimit {
            get => _depthLimit;
            set
            {
                if (value < 2)
                {
                    throw new ArgumentOutOfRangeException(nameof(value),
                        "Depth limit mustn't be less than 2");
                }

                _depthLimit = value;
            }
        }

        public int IterationsCount { get; set; }
        public int BlindCornersCount { get; set; }
        public int OverallStatesCount { get; set; }
        public int StatesCountInMemory { get; set; }

        public LdfsSolver(IEightPuzzleValidator puzzleValidator, int
depthLimit = 27)
        {
            _puzzleValidator = puzzleValidator;
            DepthLimit = depthLimit;
        }

        /// <exception cref="InsufficientMemoryException"></exception>
        /// <exception cref="ArgumentNullException"></exception>
        public IPuzzleBoard SolveEightPuzzle(IPuzzleBoard puzzleBoard)
        {
            if (puzzleBoard is null)
            {
                throw new ArgumentNullException(nameof(puzzleBoard));
            }
```

```
            IterationsCount++;

            if
(_puzzleValidator.ValidateGoalStateReaching(puzzleBoard.PuzzleState))
            {
                return puzzleBoard;
            }

            if (puzzleBoard.Depth == DepthLimit)
            {
                BlindCornersCount++;

                return null;
            }

            puzzleBoard.GenerateChildren();

            long bytesCountInGb = (long)Math.Pow(1024, 3);

            if (GC.GetTotalMemory(false) >= bytesCountInGb)
            {
                throw new InsufficientMemoryException("The available
memory has been exceeded");
            }

            foreach (IPuzzleBoard child in puzzleBoard.Children)
            {
                OverallStatesCount++;
                StatesCountInMemory++;

                var result = SolveEightPuzzle(child);

                if (result is not null)
                {
                    return result;
                }
            }

            return null;
        }
    }
}
```

```
namespace EightPuzzleLogic.Algorithms;

public enum PuzzleSortAlgorithmType
{
    LDFS,
    Astar
}
```

```
namespace EightPuzzleLogic.Boards;
using System;

public class AstarLdfSpuzzleBoard : LDFSpuzzleBoard, IAstarPuzzleBoard
{
    public AstarLdfSpuzzleBoard(int[][] puzzleState, int depth,
IPuzzleBoard parent,
        MovingDirection puzzleMovingDirection, IEightPuzzleManager
puzzleManager, IEightPuzzleValidator puzzleValidator)
        : base(puzzleState, depth, parent, puzzleMovingDirection,
puzzleManager, puzzleValidator)
    {
    }

    public override IPuzzleBoard Clone()
    {
        var newField = new[] { PuzzleState[0].Clone() as int[],
PuzzleState[1].Clone() as int[],
            PuzzleState[2].Clone() as int[] };

        return new AstarLdfSpuzzleBoard(newField, Depth, Parent,
PuzzleLastMovingDirection,
            PuzzleManager, PuzzleValidator);
    }

    public int GetDistanceToGoal()
    {
        var distanceToGoalPlace = 0;

        for (int i = 0; i < PuzzleValidator.PuzzlesCountInRow; i++)
        {
            for (int j = 0; j < PuzzleValidator.PuzzlesCountInRow; j++)
            {
                if (PuzzleState[i][j] ==
PuzzleValidator.PuzzleValueToMove)
```

```
                {
                    continue;
                }

                var (puzzleDiv, puzzleMod) = (
                        (PuzzleState[i][j] - 1) /
PuzzleValidator.PuzzlesCountInRow,
                        (PuzzleState[i][j] - 1) %
PuzzleValidator.PuzzlesCountInRow
                    );

                distanceToGoalPlace += Math.Abs(puzzleDiv - i) +
Math.Abs(puzzleMod - j);
            }
        }

        return distanceToGoalPlace;
    }

    public int GetOutlay()
    {
        return GetDistanceToGoal() + Depth;
    }
}
```

```
namespace EightPuzzleLogic.Boards
{
    public interface IAstarPuzzleBoard : IPuzzleBoard
    {
        public int GetOutlay();
        public int GetDistanceToGoal();
    }
}
```

```
namespace EightPuzzleLogic.Boards
{
    public interface IPuzzleBoard
    {
        public IPuzzleBoard Parent { get; set; }
        public List<IPuzzleBoard> Children { get; }
        public int Depth { get; set; }
        public int[][] PuzzleState { get; set; }
        public MovingDirection PuzzleLastMovingDirection { get; set; }
```

```
        public IPuzzleBoard Clone();
        public void GenerateChildren();
        public Tuple<int, int> GetCoordinatesOfPuzzle(int puzzleValue);
    }
}
```

```
using EightPuzzleLogic.Extensions;

namespace EightPuzzleLogic.Boards
{
    public class LDFSpuzzleBoard : IPuzzleBoard
    {
        public IPuzzleBoard Parent { get; set; }
        public List<IPuzzleBoard> Children { get; private set; } = new();

        private int _depth;
        /// <exception cref="ArgumentOutOfRangeException"></exception>
        public int Depth {
            get => _depth;
            set
            {
                if (value < 0)
                {
                    throw new ArgumentOutOfRangeException(nameof(value),
"Depth mustn't be less than 0");
                }

                _depth = value;
            }
        }

        public int[][] PuzzleState { get; set; }
        public MovingDirection PuzzleLastMovingDirection { get; set; }
        protected readonly IEightPuzzleManager PuzzleManager;
        protected readonly IEightPuzzleValidator PuzzleValidator;

        public LDFSpuzzleBoard
            (int[][] puzzleState, int depth, IPuzzleBoard parent,
MovingDirection puzzleMovingDirection,
                IEightPuzzleManager puzzleManager, IEightPuzzleValidator
puzzleValidator)
        {
```

```
            PuzzleState = puzzleState;
            Depth = depth;
            Parent = parent;
            PuzzleLastMovingDirection = puzzleMovingDirection;
            PuzzleManager = puzzleManager;
            PuzzleValidator = puzzleValidator;
        }

        public virtual IPuzzleBoard Clone()
        {
            var newField = new[] { PuzzleState[0].Clone() as int[],
PuzzleState[1].Clone() as int[],
                PuzzleState[2].Clone() as int[] };

            return new LDFSpuzzleBoard
                (newField, Depth, Parent, PuzzleLastMovingDirection,
PuzzleManager, PuzzleValidator);
        }

        public Tuple<int, int> GetCoordinatesOfPuzzle(int puzzleValue)
        {
            for (var x = 0; x < PuzzleValidator.PuzzlesCountInRow; ++x)
            {
                for (var y = 0; y < PuzzleValidator.PuzzlesCountInRow;
++y)
                {
                    if (PuzzleState[x][y] == puzzleValue)
                    {
                        return new Tuple<int, int>(x, y);
                    }
                }
            }

            return new Tuple<int, int>(-1, -1);
        }

        private IPuzzleBoard TryGetChild(MovingDirection movingDirection)
        {
            var child = PuzzleManager.MovePuzzle(this, movingDirection);

            if (child is null)
            {
                return null;
```

```
            }

            child.PuzzleLastMovingDirection = movingDirection;
            child.Parent = this;
            child.Depth++;

            return child;
        }

        public void GenerateChildren()
        {
            var children = new List<IPuzzleBoard>
            {
                TryGetChild(MovingDirection.Right),
                TryGetChild(MovingDirection.Left),
                TryGetChild(MovingDirection.Up),
                TryGetChild(MovingDirection.Down)
            };

            Children = children.Where(child => child is not
null).ToList();
        }

        /// <exception cref="ArgumentNullException"></exception>
        public override string ToString()
        {
            return $"Depth - {Depth}, Puzzle state - {string.Join(" ",
PuzzleState.To1Dimension())}, "
                   + $"Star last direction -
{PuzzleLastMovingDirection}";
        }
    }
}
```

```
namespace EightPuzzleLogic.Boards
{
    public enum MovingDirection
    {
        Up = 1,
        Right = 2,
        Down = 3,
        Left = 4
    }
```

```
}
```

```
namespace EightPuzzleLogic.Extensions;

public static class DataStructureExtensions
{
    /// <exception cref="ArgumentNullException"></exception>
    public static T[] To1Dimension<T>(this T[][] twoDimensionalArray)
    {
        if (twoDimensionalArray is null)
        {
            throw new ArgumentNullException
                (nameof(twoDimensionalArray), "The input array mustn't be
null");
        }

        return twoDimensionalArray.SelectMany(subarray =>
subarray).ToArray();
    }
}
```

```
using EightPuzzleLogic.Algorithms;
using EightPuzzleLogic.Boards;
using Microsoft.Extensions.DependencyInjection;

namespace EightPuzzleLogic.Extensions
{
    public static class IocContainerExtensions
    {
        /// <exception cref="ArgumentException"></exception>
        /// <exception cref="InvalidDataException"></exception>
        public static IServiceCollection ConfigureServices
            (this IServiceCollection services, int[][]
generatedPuzzleField, object solver)
        {
            var puzzleValidator = services
                .BuildServiceProvider()
                .GetRequiredService<IEightPuzzleValidator>();

            var puzzleManager = services
                .BuildServiceProvider()
                .GetRequiredService<IEightPuzzleManager>();
```

```
            services
                .AddSingleton<IEightPuzzleSolving>((_) =>
(IEightPuzzleSolving)solver)
                .AddSingleton<ICharacteristicable>((_) =>
(ICharacteristicable)solver)
                .AddSingleton<IPuzzleBoard>((_) =>
                    new LDFSpuzzleBoard(generatedPuzzleField, 0, null,
default, puzzleManager,
                        puzzleValidator))
                .AddSingleton<IAstarPuzzleBoard>((_) =>
                    new AstarLdfSpuzzleBoard(generatedPuzzleField, 0,
null, default, puzzleManager,
                        puzzleValidator));

            return services;
        }

        public static IEightPuzzleValidator ConfigurePuzzleValidator
            (this ServiceCollection services, int[][] goalState, int
puzzleValueToMove = 0)
        {
            return services
                .AddSingleton<IEightPuzzleValidator>((_) => new
EightPuzzleValidator(goalState, puzzleValueToMove))
                .BuildServiceProvider()
                .GetRequiredService<IEightPuzzleValidator>();
        }

        public static IEightPuzzleManager ConfigurePuzzleManager
            (this ServiceCollection services, IEightPuzzleValidator
puzzleValidator)
        {
            return services
                .AddSingleton<IEightPuzzleManager>((_) => new
EightPuzzleManager(puzzleValidator))
                .BuildServiceProvider()
                .GetRequiredService<IEightPuzzleManager>();
        }
    }
}
```

```
using EightPuzzleLogic.Boards;
```

```
namespace EightPuzzleLogic
{
    public class EightPuzzleManager : IEightPuzzleManager
    {
        private readonly IEightPuzzleValidator _puzzleValidator;

        public EightPuzzleManager(IEightPuzzleValidator puzzleValidator)
        {
            _puzzleValidator = puzzleValidator;
        }

        public IPuzzleBoard MovePuzzle(IPuzzleBoard puzzleBoard,
MovingDirection movingDirection)
        {
            if (!_puzzleValidator.ValidatePuzzleMoving(puzzleBoard,
movingDirection))
            {
                return null;
            }

            var child = puzzleBoard.Clone();
            int tempPuzzle;
            var (xPuzzleCoordinate, yPuzzleCoordinate) = puzzleBoard
                .GetCoordinatesOfPuzzle(_puzzleValidator.PuzzleValueToMov
e);

            switch (movingDirection)
            {
                case MovingDirection.Left:
                    tempPuzzle =
child.PuzzleState[xPuzzleCoordinate][yPuzzleCoordinate - 1];
                    child.PuzzleState[xPuzzleCoordinate][yPuzzleCoordinat
e - 1] = _puzzleValidator.PuzzleValueToMove;
                    child.PuzzleState[xPuzzleCoordinate][yPuzzleCoordinat
e] = tempPuzzle;
                    break;
                case MovingDirection.Right:
                    tempPuzzle =
child.PuzzleState[xPuzzleCoordinate][yPuzzleCoordinate + 1];
                    child.PuzzleState[xPuzzleCoordinate][yPuzzleCoordinat
e + 1] = _puzzleValidator.PuzzleValueToMove;
                    child.PuzzleState[xPuzzleCoordinate][yPuzzleCoordinat
e] = tempPuzzle;
```

```
                    break;
                case MovingDirection.Up:
                    tempPuzzle = child.PuzzleState[xPuzzleCoordinate -
1][yPuzzleCoordinate];
                    child.PuzzleState[xPuzzleCoordinate -
1][yPuzzleCoordinate] = _puzzleValidator.PuzzleValueToMove;
                    child.PuzzleState[xPuzzleCoordinate][yPuzzleCoordinat
e] = tempPuzzle;
                    break;
                case MovingDirection.Down:
                    tempPuzzle = child.PuzzleState[xPuzzleCoordinate +
1][yPuzzleCoordinate];
                    child.PuzzleState[xPuzzleCoordinate +
1][yPuzzleCoordinate] = _puzzleValidator.PuzzleValueToMove;
                    child.PuzzleState[xPuzzleCoordinate][yPuzzleCoordinat
e] = tempPuzzle;
                    break;
                default:
                    return null;
            }

            return child;
        }

        public int[][] GenerateStartState()
        {
            var result = new[] { new
int[_puzzleValidator.PuzzlesCountInRow],
                new int[_puzzleValidator.PuzzlesCountInRow], new
int[_puzzleValidator.PuzzlesCountInRow] };
            var rd = new Random();
            var puzzlePositionInRow = 0;
            var zeroExists = false;

            for (var i = 0; i < _puzzleValidator.PuzzlesCountInRow; i++)
            {
                while (puzzlePositionInRow <
_puzzleValidator.PuzzlesCountInRow)
                {
                    var generatedPuzzle = rd.Next(0,
_puzzleValidator.PuzzlesCount);
```

```
                    if (generatedPuzzle ==
_puzzleValidator.PuzzleValueToMove)
                    {
                        continue;
                    }

                    if ((generatedPuzzle == 0) && !zeroExists)
                    {
                        puzzlePositionInRow++;
                        zeroExists = true;
                    }

                    if (Array.Exists(result, row =>
row.Contains(generatedPuzzle)))
                    {
                        continue;
                    }

                    if ((puzzlePositionInRow != 1) || (i != 1))
                    {
                        result[i][puzzlePositionInRow] = generatedPuzzle;
                    }

                    puzzlePositionInRow++;
                }

                puzzlePositionInRow = 0;
            }

            result[^2][_puzzleValidator.PuzzlesCountInRow - 2] =
_puzzleValidator.PuzzleValueToMove;

            return result;
        }
    }
}
```

```
using EightPuzzleLogic.Boards;

namespace EightPuzzleLogic;

public class EightPuzzleValidator : IEightPuzzleValidator
{
```

```
    public int PuzzlesCountInRow { get; } = 3;
    public int PuzzlesCount { get; } = 9;

    private int[][] _goalState;
    /// <exception cref="ArgumentException"></exception>
    public int[][] GoalState {
        get => _goalState;
        set
        {
            if (value.Length != PuzzlesCountInRow)
            {
                throw new ArgumentException("The state has incorrent
count of puzzles");
            }

            for (var i = 0; i < value.Length; i++)
            {
                if (value[i].Length != PuzzlesCountInRow)
                {
                    throw new ArgumentException("The state has incorrent
count of puzzles");
                }
            }

            _goalState = value;
        }
    }

    private int _puzzleValueToMove;
    /// <exception cref="ArgumentOutOfRangeException"></exception>
    public int PuzzleValueToMove {
        get => _puzzleValueToMove;
        set
        {
            if ((value < 0) || (value > 8))
            {
                throw new ArgumentOutOfRangeException(nameof(value),
                    "Specified value mustn't be less than 0 and bigger
then 8");
            }

            _puzzleValueToMove = value;
        }
```

```
    }

    public EightPuzzleValidator(int[][] goalState, int puzzleValueToMove)
    {
        GoalState = goalState;
        PuzzleValueToMove = puzzleValueToMove;
    }

    /// <exception cref="ArgumentNullException"></exception>
    public bool ValidatePuzzleMoving(IPuzzleBoard puzzleBoard,
MovingDirection movingDirection)
    {
        if (puzzleBoard is null)
        {
            throw new ArgumentNullException(nameof(puzzleBoard));
        }

        var (xPuzzleCoordinate, yPuzzleCoordinate) =
puzzleBoard.GetCoordinatesOfPuzzle(PuzzleValueToMove);

        switch (movingDirection)
        {
            case MovingDirection.Left:
                if ((yPuzzleCoordinate == 0)
                    || (puzzleBoard.PuzzleLastMovingDirection ==
MovingDirection.Right))
                {
                    return false;
                }
                break;
            case MovingDirection.Right:
                if ((yPuzzleCoordinate == (PuzzlesCountInRow - 1))
                    || (puzzleBoard.PuzzleLastMovingDirection ==
MovingDirection.Left))
                {
                    return false;
                }
                break;
            case MovingDirection.Up:
                if ((xPuzzleCoordinate == 0)
                    || (puzzleBoard.PuzzleLastMovingDirection ==
MovingDirection.Down))
                {
```

```
                    return false;
                }
                break;
            case MovingDirection.Down:
                if ((xPuzzleCoordinate == (PuzzlesCountInRow - 1))
                    || (puzzleBoard.PuzzleLastMovingDirection ==
MovingDirection.Up))
                {
                    return false;
                }
                break;
            default:
                return false;
        }

        return true;
    }

    /// <exception cref="ArgumentException"></exception>
    public bool ValidateGoalStateReaching(int[][] fieldState)
    {
        if (fieldState.Length != PuzzlesCountInRow)
        {
            throw new ArgumentException("The state has incorrent count of
puzzles");
        }

        for (var i = 0; i < fieldState.Length; i++)
        {
            if (fieldState[i].Length != PuzzlesCountInRow)
            {
                throw new ArgumentException("The state has incorrent
count of puzzles");
            }
        }

        for (int i = 0; i < PuzzlesCountInRow; i++)
        {
            for (int j = 0; j < PuzzlesCountInRow; j++)
            {
                if (fieldState[i][j] != GoalState[i][j])
                {
                    return false;
```

```
                }
            }
        }

        return true;
    }

    /// <exception cref="ArgumentNullException"></exception>
    public bool ValidateFieldCorrectness(IEnumerable<int[]>
fieldToValidate)
    {
        if (fieldToValidate is null)
        {
            throw new ArgumentNullException(nameof(fieldToValidate));
        }

        if ((fieldToValidate.Count() != PuzzlesCountInRow) ||
(fieldToValidate.Any(row => row.Length != 3)))
        {
            return false;
        }

        var sortedPuzzleState = fieldToValidate
            .SelectMany(row => row).ToList()
            .OrderBy(num => num).ToList();

        sortedPuzzleState.Remove(0);
        sortedPuzzleState.Add(0);

        var slicedSortedPuzzleState = new int[3][] {new
int[PuzzlesCountInRow],
            new int[PuzzlesCountInRow], new int[PuzzlesCountInRow]};
        var currentPositionInList = 0;

        for (var i = 0; i < PuzzlesCountInRow; i++)
        {
            for (var j = 0; j < PuzzlesCountInRow; j++)
            {
                slicedSortedPuzzleState[i][j] =
sortedPuzzleState[currentPositionInList++];
            }
        }
```

```
        return ValidateGoalStateReaching(slicedSortedPuzzleState);
    }
}
```

```
using EightPuzzleLogic.Boards;

namespace EightPuzzleLogic
{
    public interface IEightPuzzleManager
    {
        public int[][] GenerateStartState();
        public IPuzzleBoard MovePuzzle(IPuzzleBoard puzzleBoard,
MovingDirection movingDirection);
    }
}
```

```
using EightPuzzleLogic.Boards;

namespace EightPuzzleLogic;

public interface IEightPuzzleValidator
{
    public int PuzzlesCount { get; }
    public int[][] GoalState { get; }
    public int PuzzlesCountInRow { get; }
    public int PuzzleValueToMove { get; }

    public bool ValidatePuzzleMoving(IPuzzleBoard puzzleBoard,
MovingDirection movingDirection);
    public bool ValidateGoalStateReaching(int[][] fieldState);
    public bool ValidateFieldCorrectness(IEnumerable<int[]>
fieldToValidate);
}
```

```
using EightPuzzleLogic;
using EightPuzzleLogic.Boards;
using EightPuzzleLogic.Algorithms;

namespace EightPuzzleRunner
{
    public class IOhandler
    {
```

```
        private readonly IEightPuzzleValidator _puzzleValidator;

        public IOhandler(IEightPuzzleValidator puzzleValidator)
        {
            _puzzleValidator = puzzleValidator;
        }

        public static void CatchAlgorithmType(ref PuzzleSortAlgorithmType
algorithmType)
        {
            Console.Write("Enter the algorithm to use:\n"
                + "ldfs\n"
                + "a*\n"
                + ">>> ");

            bool errorOccured = false;

            do
            {
                errorOccured = false;
                var algorithmToUse = Console.ReadLine();

                switch (algorithmToUse.ToLower().Trim())
                {
                    case "a*":
                        algorithmType = PuzzleSortAlgorithmType.Astar;
                        break;
                    case "ldfs":
                        algorithmType = PuzzleSortAlgorithmType.LDFS;
                        break;
                    default:
                        errorOccured = true;
                        System.Console.WriteLine("You entered wrong
algorithm name. Try again");
                        break;
                }
            } while (errorOccured);
        }

        /// <exception cref="ArgumentNullException"></exception>
        public void PrintField(int[][] puzzleField)
        {
            if (puzzleField is null)
```

```
            {
                throw new ArgumentNullException(nameof(puzzleField),
                    "The puzzle field mustn't be null");
            }

            const int dashesCountInRow = 9;
            var currentPuzzlePositionInRow = 0;

            var horizontalLine = new string('-', dashesCountInRow);
            Console.WriteLine(horizontalLine);

            for (var i = 0; i < _puzzleValidator.PuzzlesCountInRow; i++)
            {
                var verticalLine = $"|
{puzzleField[i][currentPuzzlePositionInRow++]} "
                    + $"{puzzleField[i][currentPuzzlePositionInRow++]} "
                    + $"{puzzleField[i][currentPuzzlePositionInRow]} |";
                currentPuzzlePositionInRow = 0;

                Console.WriteLine(verticalLine);
            }

            Console.WriteLine(horizontalLine);
        }

        /// <exception cref="ArgumentNullException"></exception>
        public void PrintSortingSolutionStepByStep(IPuzzleBoard
resultPuzzleBoard)
        {
            if (resultPuzzleBoard == null)
            {
                throw new
ArgumentNullException(nameof(resultPuzzleBoard),
                    "Puzzle board mustn't be null");
            }

            PrintSortingSolutionRecursively(resultPuzzleBoard);
        }

        private void PrintSortingSolutionRecursively(IPuzzleBoard
puzzleBoard)
        {
            if (puzzleBoard is null)
```

```
            {
                return;
            }

            PrintSortingSolutionRecursively(puzzleBoard.Parent);
            Console.WriteLine(puzzleBoard);
            PrintField(puzzleBoard.PuzzleState);
        }

        /// <exception cref="ArgumentNullException"></exception>
        public void PrintSolvingCharacteristics(ICharacteristicable
characteristicable)
        {
            if (characteristicable is null)
            {
                throw new
ArgumentNullException(nameof(characteristicable));
            }

            System.Console.WriteLine($"Iterations count -
{characteristicable.IterationsCount}\n"
                + $"Blind corners count -
{characteristicable.BlindCornersCount}\n"
                + $"Overall states count -
{characteristicable.OverallStatesCount}\n"
                +$"States count in memory -
{characteristicable.StatesCountInMemory}");
        }
    }
}
```

```
using System.Diagnostics;
using EightPuzzleLogic;
using EightPuzzleLogic.Algorithms;
using EightPuzzleLogic.Boards;
using EightPuzzleLogic.Extensions;
using Microsoft.Extensions.DependencyInjection;

namespace EightPuzzleRunner;

public static class Program
{
    public static void Main()
```

```
    {
        Console.WriteLine("You are going to solve 8-puzzle problem on
randomly generated field:");

        var goalState = new[] { new [] {1, 2, 3}, new [] {4, 5, 6}, new
[] {7, 8, 0} };

        var serviceCollection = new ServiceCollection();
        var puzzleValidator =
serviceCollection.ConfigurePuzzleValidator(goalState);
        var puzzleManager =
serviceCollection.ConfigurePuzzleManager(puzzleValidator);

        var generatedPuzzleField = puzzleManager.GenerateStartState();

        PuzzleSortAlgorithmType sortingAlgorithmType = default;
        IOhandler.CatchAlgorithmType(ref sortingAlgorithmType);

        object solverObject;
        IServiceProvider serviceContainer;

        try
        {
            switch (sortingAlgorithmType)
            {
                case PuzzleSortAlgorithmType.LDFS:
                    int depthLimit = 27;

                    solverObject = new LdfsSolver(puzzleValidator,
depthLimit);
                    break;
                case PuzzleSortAlgorithmType.Astar:
                    solverObject = new AstarSolver(puzzleValidator);
                    break;
                default:
                    throw new ArgumentException("Specified wrong
algorithm type");
            }

            serviceContainer = serviceCollection
                .ConfigureServices(generatedPuzzleField, solverObject)
                .BuildServiceProvider();
        }
```

```
        catch (ArgumentOutOfRangeException ex)
        {
            System.Console.WriteLine(ex.Message);
            return;
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine(ex.Message);
            return;
        }
        catch (InvalidDataException ex)
        {
            Console.WriteLine(ex.Message);
            return;
        }

        IPuzzleBoard startingBoard;

        if (sortingAlgorithmType is PuzzleSortAlgorithmType.LDFS)
        {
            startingBoard = serviceContainer.GetService<IPuzzleBoard>();
        }
        else
        {
            startingBoard =
serviceContainer.GetService<IAstarPuzzleBoard>();
        }

        var outputHandler = new
IOhandler(serviceContainer.GetService<IEightPuzzleValidator>());

        Console.WriteLine("Created puzzle field:");
        outputHandler.PrintField(startingBoard!.PuzzleState);

        var timer = new Stopwatch();

        timer.Start();
        IPuzzleBoard resultBoard;
        var solver = solverObject as IEightPuzzleSolving;

        try
        {
            resultBoard = solver!.SolveEightPuzzle(startingBoard);
```

```
        }
        catch (ArgumentNullException ex)
        {
            System.Console.WriteLine(ex.Message);
            return;
        }
        catch (ArgumentException ex)
        {
            System.Console.WriteLine(ex.Message);
            return;
        }
        catch (InsufficientMemoryException ex)
        {
            System.Console.WriteLine(ex.Message);
            return;
        }

        timer.Stop();

        Console.WriteLine($"Solving of 8-puzzle took {timer.Elapsed}
time");

        if (resultBoard is null)
        {
            Console.WriteLine("Solution is not found");
        }
        else
        {
            Console.WriteLine("Solution step-by-step:");

            try
            {
                outputHandler.PrintSortingSolutionStepByStep(resultBoard)
;
                outputHandler.PrintSolvingCharacteristics(serviceContaine
r
                                .GetService<ICharacteristicable>());
            }
            catch (ArgumentNullException ex)
            {
                Console.WriteLine(ex.Message);
            }
            catch (ArgumentOutOfRangeException ex)
```

```
            {
                System.Console.WriteLine(ex.Message);
            }
        }
    }
}
```

3.2.2 Приклади роботи

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

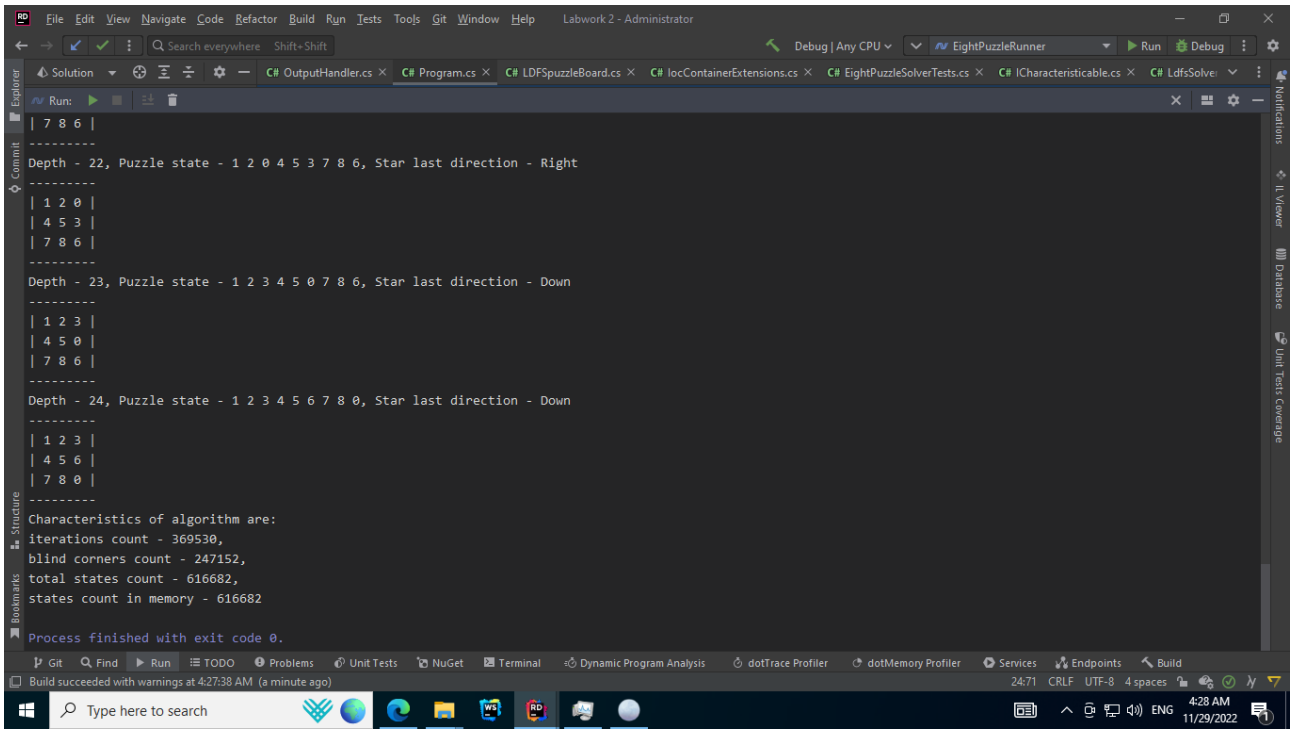

Рисунок 3.1 – Алгоритм LDFS

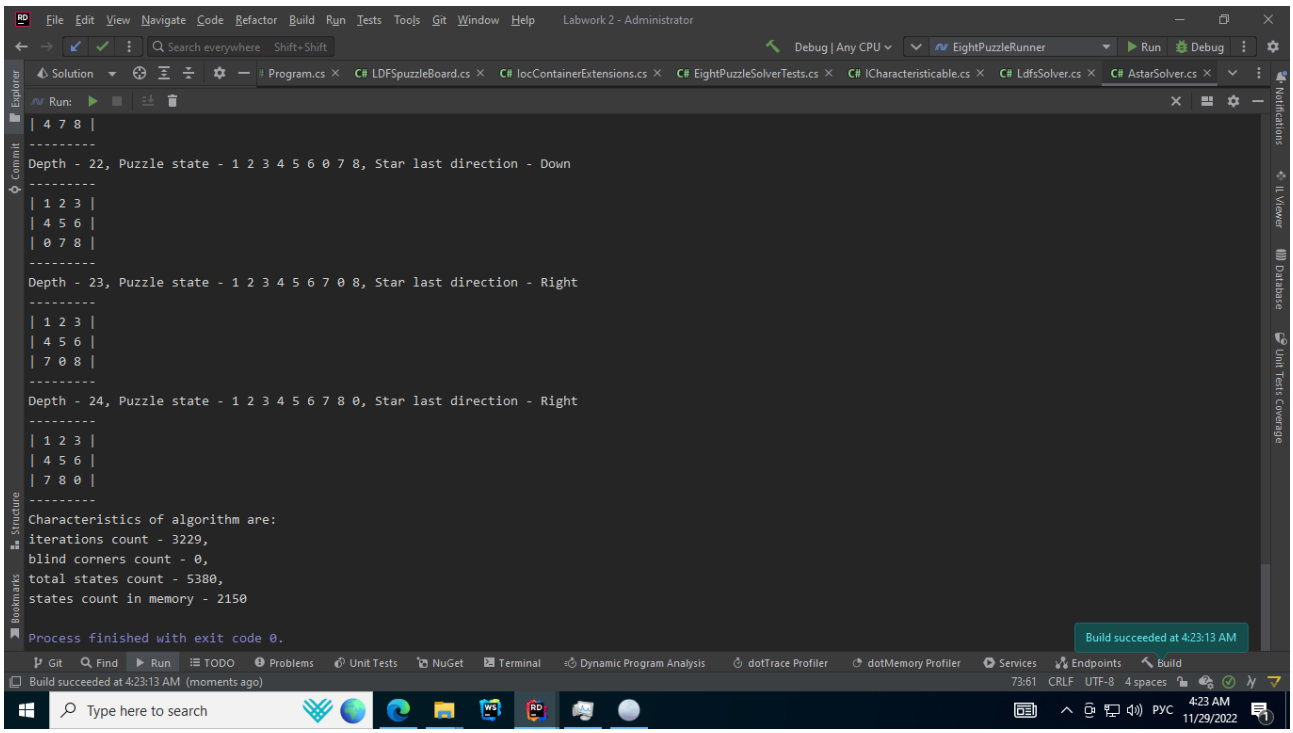

Рисунок 3.2 – Алгоритм A*

3.3 Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
7 1 4 8 0 2 5 6 3	369530	247152	616682	616682
6 1 7 4 0 5 8 3 2	38999	26074	65073	65073
1 5 7 6 0 2	641314	427235	1068551	1068551

3 4 8				
1 4 2 8 0 3 7 5 6	362819	242682	605501	605501
1 8 3 6 0 5 4 7 2	391212	261445	652657	652657
1 6 2 8 0 3 4 5 7	332105	222259	554364	554364
4 1 5 2 0 7 3 8 6	1147311	764529	1911840	1911840
1 6 3 8 0 7 5 2 4	412034	275074	687108	687108
4 1 5 6 0 8 2 3 7	765585	509571	1275156	1275156
1 3 2 7 0 8 6 5 4	37532	25155	62687	62687
4 3 1 6 0 8 5 7 2	731727	487190	1218917	1218917
3 8 4 5 0 7 1 6 2	1548744	1031143	2579887	2579887

1 7 5 8 0 3 6 4 2	465579	310909	776488	776488
8 3 2 1 0 4 7 5 6	30810	20682	51492	51492
4 2 3 5 0 8 7 6 1	232358	156155	388513	388513
1 2 3 8 0 6 4 5 7	359718	240681	600399	600399
5 2 1 3 0 4 7 6 8	35683	23965	59648	59648
2 5 8 3 0 6 7 1 4	219685	147567	367252	367252
1 6 8 3 0 5 2 4 7	132218	88980	221198	221198
5 8 4 2 0 6 7 1 3	1707606	1136379	2843985	2843985

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

Таблиця 3.2 – Характеристики оцінювання алгоритму A*

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
6 7 8 2 0 5 4 1 3	983	0	1635	651
7 8 3 6 0 2 4 1 5	3325	0	5567	2241
7 3 6 5 0 1 2 8 4	357	0	599	241
7 8 2 3 0 4 5 1 6	3127	0	5181	2053
4 3 5 2 0 7 8 1 6	81	0	141	59
6 7 2 4 0 3 5 8 1	2469	0	4184	1714
2 7 8 1 0 4 3 5 6	2412	0	4021	1608
7 2 6 1 0 4 5 8 3	632	0	1073	440
4 3 8	314	0	541	226

5 0 2 1 7 6				
3 2 7 5 0 6 1 8 4	2221	0	3721	1499
3 8 7 1 0 6 2 4 5	176	0	299	122
7 8 5 1 0 2 3 4 6	908	0	1508	599
5 1 3 8 0 6 4 2 7	197	0	337	139
6 3 1 7 0 8 5 4 2	2617	0	4397	1779
7 6 8 3 0 1 4 5 2	5162	0	8666	3503
1 8 5 3 0 4 7 6 2	563	0	944	380
3 8 4 6 0 5 7 2 1	2474	0	4170	1695
8 4 3	717	0	1214	496

2 0 6 7 5 1				
1 8 2 7 0 4 5 3 6	153	0	264	110
3 2 1 6 0 7 5 4 8	8995	0	15055	6059

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми інформативного та неінформативного пошуку на прикладі програми-знаходження рішення гри 8-puzzle.

В якості алгоритму неінформативного пошуку я реалізував LDFS. Від пошуку в глибину (DFS), він відрізняється обмеженням глибини для запобігання утворенню надвеликих об’ємів станів пазлової дошки.

Інформативний пошук я реалізував, застосувавши A*. Він працює за принципом відбору нащадку дошки на кожному рівні, котрий має найменшу вагу за евристикою. В якості евристики я використав Манхеттенську, котра полягає у сумі відстаней кожного пазла від його цільової позиції по вертикалі та горизонталі.

Провівши тестування алгоритмів, я помітив, що A* працює швидше, але видає рішення меншої кількості задач, ніж LDFS.

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%.