

**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**

Головченко М.М.  
(прізвище, ім'я, по батькові)

Київ 2022

## ЗМІСТ

<b>1</b>	<b>МЕТА ЛАБОРАТОРНОЇ РОБОТИ .....</b>	<b>3</b>
<b>2</b>	<b>ЗАВДАННЯ .....</b>	<b>4</b>
<b>3</b>	<b>ВИКОНАННЯ.....</b>	<b>8</b>
3.1	ПСЕВДОКОД АЛГОРИТМІВ.....	8
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ.....	9
3.2.1	<i>Вихідний код.....</i>	<i>9</i>
3.2.2	<i>Приклади роботи .....</i>	<i>23</i>
3.3	ДОСЛІДЖЕННЯ АЛГОРИТМІВ.....	24
	<b>ВИСНОВОК .....</b>	<b>29</b>
	<b>КРИТЕРІЇ ОЦІНЮВАННЯ .....</b>	<b>30</b>

## 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 **АПІ**, 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 відрізнятися 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 впорядкування 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 А може стояти на одній лінії з ферзем В, проте між ними стоїть ферзь С; тому А не б'є В).
- **F2** – кількість пар ферзів, які б'ють один одного без урахування видимості.
- **H1** – кількість фішок, які не стоять на своїх місцях.
- **H2** – Манхетенська відстань.
- **H3** – Евклідова відстань.
- **COLOR** – Задача розфарбування карти самостійно обраної країни, не менше 20 регіонів (областей). Необхідно розфарбувати карту не більше ніж у 4 різні кольори. Мається на увазі приписування кожному регіону власного кольору так, щоб кольори сусідніх регіонів відрізнялись. Використовувати евристичну функцію, яка повертає кількість пар суміжних вузлів, що мають однаковий колір (тобто кількість конфліктів). Реалізувати алгоритм пошуку із поверненнями (backtracking) для розв'язання поставленої задачі. Для

підвищення швидкодії роботи алгоритму використати евристичну функцію, а початковим станом вважати випадкову вершину.

- **HILL** – Пошук зі сходженням на вершину з використанням із використанням руху вбік (на 100 кроків) та випадковим перезапуском (кількість необхідних разів запуску визначити самостійно).

- **ANNEAL** – Локальний пошук із симуляцією відпалу. Робоча характеристика – залежність температури  $T$  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	АП	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

## 3.1 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&lt;PuzzleBoard, int&gt;()

closed = HashSet&lt;PuzzleState&gt;()

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.Enqueue(child, child.GetOutlay())
    end for each
end while
return null

```

## 3.2 Програмна реалізація

### 3.2.1 Вихідний код

```

public enum MovingDirection
{
    Up = 1,
    Right = 2,
    Down = 3,
    Left = 4
}

```

```

public class LDFSuzzleBoard : IPuzzleBoard
{
    public IPuzzleBoard Parent { get; set; }
    public List<IPuzzleBoard> Children { get; private set; } = new();
    public int Depth { get; set; }
    public int[][] PuzzleState { get; set; }
    public MovingDirection PuzzleLastMovingDirection { get; set; }
    protected readonly IEightPuzzleManager PuzzleManager;
    protected readonly IEightPuzzleValidator PuzzleValidator;

    public LDFSuzzleBoard
        (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);
    }

    public 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}";
    }
}

```

```

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;
    }
}

```

```

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();
    }
}

```

```

public static class IoCContainerExtensions
{
    /// <exception cref="ArgumentException"></exception>
    /// <exception cref="InvalidDataException"></exception>
    public static IServiceCollection ConfigureServices
        (this IServiceCollection services, bool debug, int[][]
goalState, int puzzleValueToMove,
        PuzzleSortAlgorithmType sortingAlgorithmType)
    {
        var puzzleValidator = services
            .AddSingleton<IEightPuzzleValidator>((_) => new
EightPuzzleValidator(goalState, puzzleValueToMove))
            .BuildServiceProvider()
            .GetService<IEightPuzzleValidator>();

        var puzzleManager = services
            .AddSingleton<IEightPuzzleManager>((_) => new
EightPuzzleManager(puzzleValidator))

```

```

        .BuildServiceProvider()
        .GetRequiredService<IEightPuzzleManager>();

        int[][] generatedPuzzleField;

        if (debug)
        {
            // generatedPuzzleField = new[] { new [] {1, 2, 3}, new
            [] {4, 0, 6}, new [] {5, 7, 8}};

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

            if
(!puzzleValidator!.ValidateFieldCorrectness(generatedPuzzleField))
            {
                throw new InvalidDataException("The generated puzzle
field is wrong");
            }
        }
        else
        {
            generatedPuzzleField =
puzzleManager.GenerateStartState();
        }

        int depthLimit;

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

                services
                    .AddSingleton<IEightPuzzleSolving>((_) =>
                        new LdfsSolver(puzzleValidator, depthLimit));
                break;
            case PuzzleSortAlgorithmType.Astar:
                depthLimit = 30;
                services
                    .AddSingleton<IEightPuzzleSolving>((_) =>
                        new AstarSolver(puzzleValidator,
depthLimit));

```

```

        break;
    default:
        throw new ArgumentException("Specified wrong
algorithm type");
    }

    services
        .AddSingleton<IPuzzleBoard>((_) =>
            new LDFSboard(generatedPuzzleField, 0, null,
                default, puzzleManager, puzzleValidator))
        .AddSingleton<IAstarPuzzleBoard>((_) =>
            new AstarLdfboard
                (generatedPuzzleField, 0, null,
                    default, puzzleManager, puzzleValidator));

    return services;
}
}

```

```

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;
    }
}

```

```

public class EightPuzzleValidator : IEightPuzzleValidator
{
    public int PuzzlesCountInRow { get; } = 3;
    public int[][] GoalState { get; }
    public int PuzzlesCount { get; } = 9;
    public int PuzzleValueToMove { get; }

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

    public bool ValidatePuzzleMoving(IPuzzleBoard puzzleBoard,
        MovingDirection movingDirection)
    {
        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;
}

public bool ValidateGoalStateReaching(int[][] fieldState)
{
    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;
}

```

```

    public bool ValidateFieldCorrectness(IEnumerable<int[]>
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);
    }
}

```

```

public enum PuzzleSortAlgorithmType
{
    LDfs,
    Astar
}

```

```

public class LdfsSolver : IEightPuzzleSolving, IDepthLimited
{
    private readonly IEightPuzzleValidator _puzzleValidator;
}

```

```

        public int DepthLimit { get; }

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

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

            if (puzzleBoard.Depth == DepthLimit)
            {
                return null;
            }

            puzzleBoard.GenerateChildren();

            return puzzleBoard.Children
                .Select(SolveEightPuzzle)
                .FirstOrDefault(result => result != null);
        }
    }
}

```

```

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

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

```

```

public IPuzzleBoard SolveEightPuzzle(IPuzzleBoard puzzleBoard)
{
    _open.Enqueue(puzzleBoard as IAstarPuzzleBoard, 0);

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

        if
(_puzzleValidator.ValidateGoalStateReaching(currentBoard.PuzzleState))
        {
            return currentBoard;
        }
        else if (currentBoard.Depth == DepthLimit)
        {
            return null;
        }

        currentBoard.GenerateChildren();

        foreach (var child in
currentBoard.Children.Cast<IAstarPuzzleBoard>())
        {
            if (_closed.Contains(child.PuzzleState))
            {
                continue;
            }

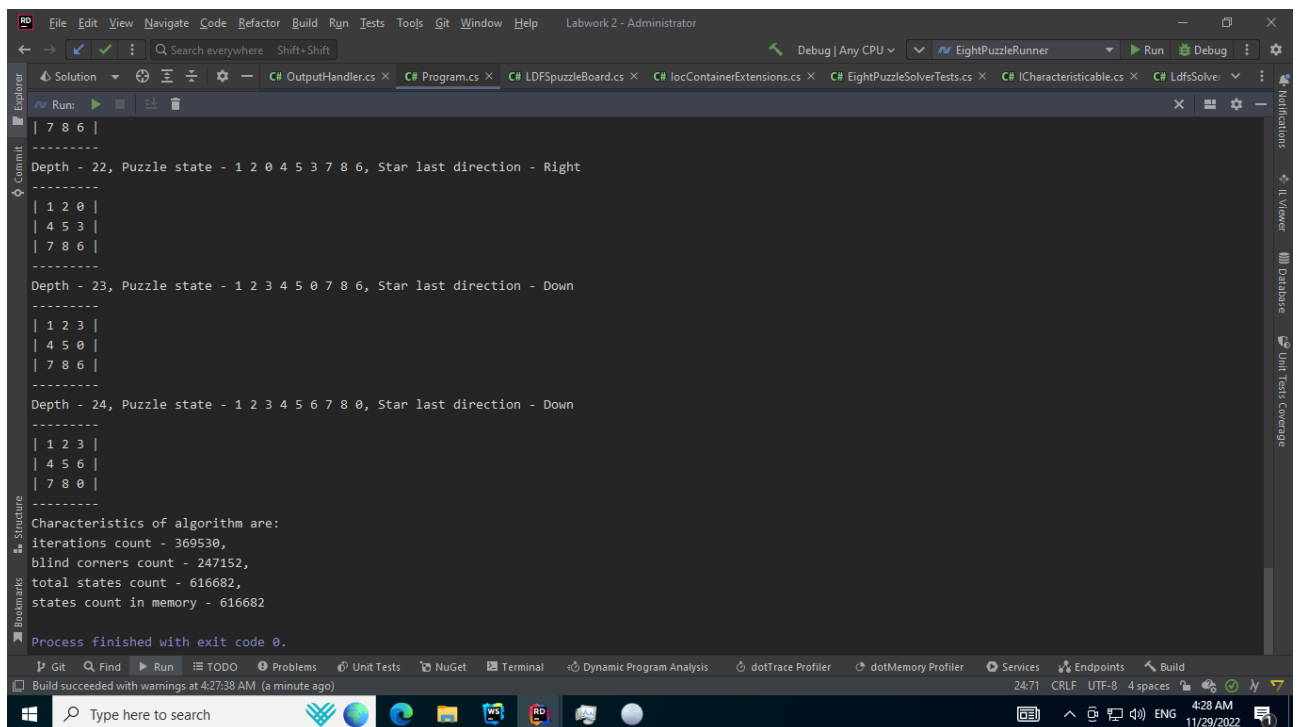
            _open.Enqueue(child, child.GetOutlay());
        }
    }

    return null;
}

```

### 3.2.2 Приклади роботи

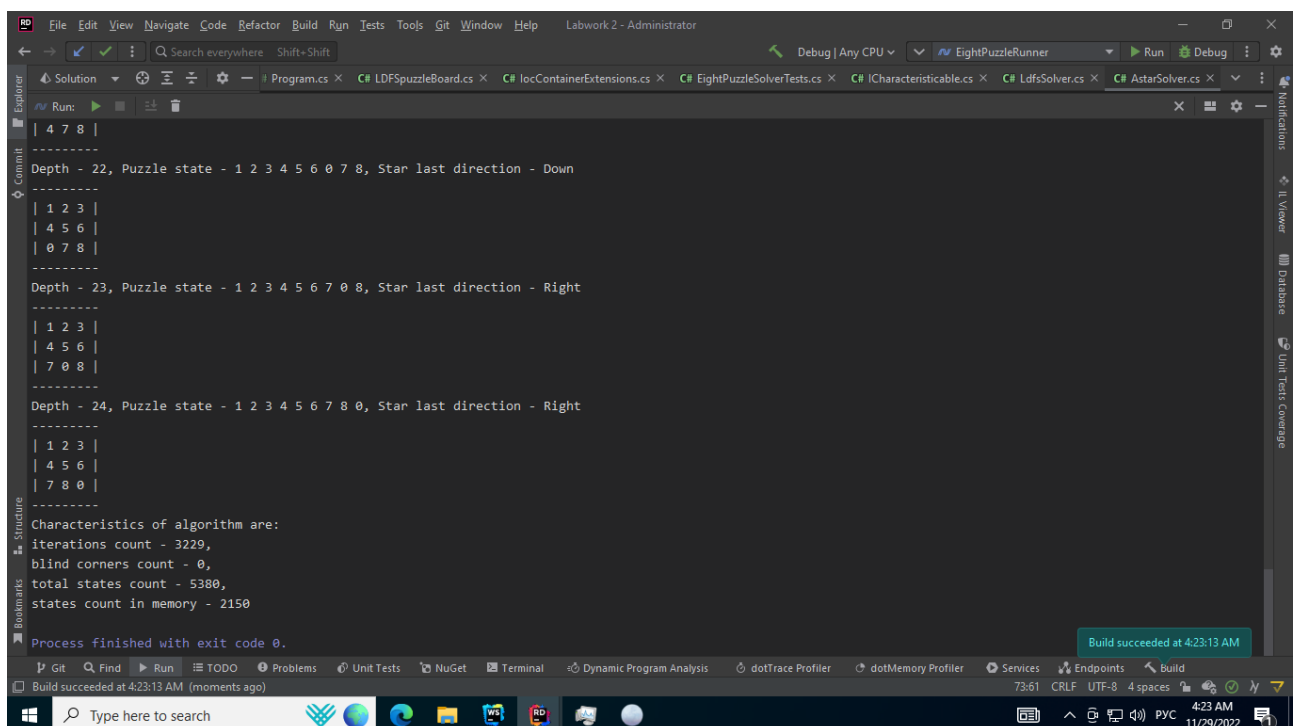
На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.



The screenshot shows the Visual Studio IDE with the 'EightPuzzleRunner' application running in debug mode. The console output displays the puzzle state at different depths and the final characteristics of the LDFS algorithm.

```
File Edit View Navigate Code Refactor Build Run Tests Tools Git Window Help Labwork 2 - Administrator
Debug | Any CPU | EightPuzzleRunner | Run | Debug
C# OutputHandler.cs | C# Program.cs | C# LDFSpuzeBoard.cs | C# locContainerExtensions.cs | C# EightPuzzleSolverTests.cs | C# ICharacteristicable.cs | C# LdfsSolver
Run:
| 7 8 6 |
-----
Depth - 22, Puzzle state - 1 2 0 4 5 3 7 8 6, Star last direction - Right
-----
| 1 2 0 |
| 4 5 3 |
| 7 8 6 |
-----
Depth - 23, Puzzle state - 1 2 3 4 5 0 7 8 6, Star last direction - Down
-----
| 1 2 3 |
| 4 5 0 |
| 7 8 6 |
-----
Depth - 24, Puzzle state - 1 2 3 4 5 6 7 8 0, Star last direction - Down
-----
| 1 2 3 |
| 4 5 6 |
| 7 8 0 |
-----
Characteristics of algorithm are:
iterations count - 369530,
blind corners count - 247152,
total states count - 616682,
states count in memory - 616682
Process finished with exit code 0.
Build succeeded with warnings at 4:27:38 AM (a minute ago)
```

Рисунок 3.1 – Алгоритм LDFS



The screenshot shows the Visual Studio IDE with the 'EightPuzzleRunner' application running in debug mode. The console output displays the puzzle state at different depths and the final characteristics of the A\* algorithm.

```
File Edit View Navigate Code Refactor Build Run Tests Tools Git Window Help Labwork 2 - Administrator
Debug | Any CPU | EightPuzzleRunner | Run | Debug
C# Program.cs | C# LDFSpuzeBoard.cs | C# locContainerExtensions.cs | C# EightPuzzleSolverTests.cs | C# ICharacteristicable.cs | C# LdfsSolver.cs | C# AstarSolver.cs
Run:
| 4 7 8 |
-----
Depth - 22, Puzzle state - 1 2 3 4 5 6 0 7 8, Star last direction - Down
-----
| 1 2 3 |
| 4 5 6 |
| 0 7 8 |
-----
Depth - 23, Puzzle state - 1 2 3 4 5 6 7 0 8, Star last direction - Right
-----
| 1 2 3 |
| 4 5 6 |
| 7 0 8 |
-----
Depth - 24, Puzzle state - 1 2 3 4 5 6 7 8 0, Star last direction - Right
-----
| 1 2 3 |
| 4 5 6 |
| 7 8 0 |
-----
Characteristics of algorithm are:
iterations count - 3229,
blind corners count - 0,
total states count - 5380,
states count in memory - 2150
Process finished with exit code 0.
Build succeeded at 4:23:13 AM
Build succeeded at 4:23:13 AM (moments ago)
```

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 3 4 8	641314	427235	1068551	1068551
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 5 0 2 1 7 6	314	0	541	226
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 2 0 6 7 5 1	717	0	1214	496
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%.