

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 № 1 з дисципліни
«Проектування алгоритмів»

«Неінформативний, інформативний та локальний пошук»

Виконав(ла)

IT-01 Бардін В. Д.
(шифр, прізвище, ім'я, по батькові)

Перевірив

Камінська П. А.
(прізвище, ім'я, по батькові)

Київ 2021

ЗМІСТ

1	МЕТА ЛАБОРАТОРНОЇ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ.....	6
3.1	ПСЕВДОКОД АЛГОРИТМІВ.....	6
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ.....	7
3.2.1	<i>Вихідний код.....</i>	<i>7</i>
3.2.2	<i>Приклади роботи</i>	<i>13</i>
3.3	ДОСЛІДЖЕННЯ АЛГОРИТМІВ	14
	ВИСНОВОК	23

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 **АП**, 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. 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 (512 Мб)

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*.

Таблиця 2.1 – Варіанти алгоритмів

№	Задача	АНП	АПП	Func
1	8-ферзів	LDFS	A*	F1

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

```
- LDFS
func Ldfs(limit, node) {
    if (node.board is valid) {
        return
    }

    // Check if depth limit reached
    if (node.depth >= limit) {
        return depthReachedFailure
    }

    // outer variables that keeps some meta-information
    _totalSteps++;
    _depth = node.depth;

    // Move queen if possible
    if (node.contains pos update) {
        node.board.MoveQueen()

        if (node.board is valid) {
            return
        }
    }

    // SubNodes for this node wasn't generated yet, so generate them
    if (node.subNodes is null) {
        node.subNodes = GenerateSubNodes()
    }

    // Select a next node to check
    if (node.subNodes.count > 0) {
        subNode = node.subNodes.First()
        if (subNode is not null) {
            Ldfs(limit, subNode)
            return
        }
    }

    // Dead end reached. Use a backtrack to check another path
    RollbackChanges(node)
    _backtracks++;
    if (node.parentNode is null) {
        return solutionNotFound
    }

    Ldfs(limit, node.parentNode)
}
```

```

- A*
// board is an input board that will be used as an input state
func AStar(board) {
    // A priority queue, where priority is an integer number. Queue has a custom
    // comparer that return nodes from smaller to bigger number
    openNodes = {start}
    // A list that contains checked board states
    closeNodes = {}

    while (board is not valid) {
        // Get a nearest node, as a priority queue is used we can just use
        // .Dequeue() method and it'll return as a required node
        node = openNodes.Dequeue()

        if (node contains pos update) {
            node.board.MoveQueen()
            closeNodes.Add(node.board.positions)
        }

        // New nodes that should be check while solving the board
        subNodes = {}
        foreach (subNode in GenerateSubNodes()) {
            boardFingerprint = subNode.board.positions

            if (!closeNodes.Contains(boardFingerprint)) {
                subNodes.Add(subNode)
            }
        }

        foreach (subNode in subNodes) {
            subNode.Weight = node.weight + Heuristic(subNode)
            openNodes.Enqueue(subNode, subNode.weight)
        }

        if (openNodes.count is 0) {
            return SolutionNotFound
        }
    }

    return node
}

```

3.2 Програмна реалізація

3.2.1 Вихідний код

```

- LDFS
public class LdfsSolver
{
    private int _depth;
    private int _totalSteps;
    private int _backtracks;
}

```

```

public LdfsSolutionInfo Solve(Board board)
{
    if (board.ValidateLite())
    {
        return new LdfsSolutionInfo()
        {
            Depth = _depth,
            TotalSteps = _totalSteps,
            BackTracks = _backtracks,
            SolvedBoard = board,
        };
    }

    var topNode = new LdfsNode()
    {
        Depth = 0,
        Board = board,
        ParentLdfsNode = null,
        PositionUpdate = null,
    };

    try
    {
        Ldfs(1000, topNode);
    }
    catch (Exception)
    {
        throw new DepthLimitReached(
            new LdfsSolutionInfo
            {
                Depth = _depth,
                TotalSteps = _totalSteps,
                BackTracks = _backtracks,
                SolvedBoard = topNode.Board,
            });
    }

    return new LdfsSolutionInfo()
    {
        Depth = _depth,
        TotalSteps = _totalSteps,
        BackTracks = _backtracks,
        SolvedBoard = topNode.Board,
    };
}

private void Ldfs(int limit, LdfsNode node)
{
    if (node.Board.Validate())
    {
        return;
    }

    if (node.Depth >= limit)
    {

```



```

        throw new Exception("Depth limit was reached but solutions wasn't
found!");
    }

    _totalSteps++;
    _depth = node.Depth;

    if (node is { PositionUpdate: not null })
    {
        node.Board.Move(node.PositionUpdate);
        node.Board.InvalidPositions.Clear();

        if (node.Board.Validate())
        {
            return;
        }
    }

    // If nodes wasn't generated yet for the current node, it's a time to
generate some
    if (node.Nodes is null)
    {
        var subNodes = GetPossiblePositions(node);
        if (subNodes is { Count: > 0 })
        {
            node.Nodes = subNodes;
        }
    }

    // Try to select a sub-node to check next for the given node
    if (node.Nodes is { Count: > 0 })
    {
        var nextSubNode = node.Nodes.First();
        if (nextSubNode is not null)
        {
            Ldfs(limit, nextSubNode);
            return;
        }
    }

    RollbackChanges(node);
    _backtracks++;
    if (node.ParentLdfsNode is null) throw new Exception("Solutions not
found");

    Ldfs(limit, node.ParentLdfsNode);
}

private List<LdfsNode> GetPossiblePositions(LdfsNode node)
{
    var safeCells = node.Board.GetSafeCellsExt();

    var possibleStates = safeCells.Select(safeCell =>
        new LdfsNode()
        {

```

```

        Depth = ++node.Depth,
        ParentLdfsNode = node,
        Board = node.Board,
        PositionUpdate = new QueenPositionUpdate()
        {
            NewPosition = safeCell.New,
            CurrentPosition = safeCell.Current,
        },
    })
    .ToList();

    return possibleStates;
}

private void RollbackChanges(LdfsNode node)
{
    if (node is { ParentLdfsNode: { Nodes: { Count: > 0 } } })
    {
        node.ParentLdfsNode.Nodes.Remove(node);
    }

    if (node.PositionUpdate is null) return;
    node.Board.MoveBack(node.PositionUpdate);
}
}

- A*
public class AStarSolver
{
    private readonly PriorityQueue<AStarNode, int> _solutionNodes;
    private readonly List<List<Position>> _closedStates;
    private readonly DateTime _stopDateTime;

    public AStarSolver()
    {
        _solutionNodes = new PriorityQueue<AStarNode, int>(new NodesComparer());
        _closedStates = new List<List<Position>>();
        _stopDateTime = DateTime.Now.AddMinutes(30);
    }

    public AStarSolutionInfo Solve(Board board)
    {
        if (board.Validate())
        {
            return new AStarSolutionInfo()
            {
                SolvedBoard = board,
            };
        }

        try
        {
            var solNode = FindSolutionPath(board);
            return new AStarSolutionInfo()
            {
                OpenStates = _solutionNodes.Count,
            };
        }
        catch { }
    }
}

```

```

        ClosedStates = _closedStates.Count,
        SolvedBoard = solNode.Board,
    };
}
catch (TimeoutException)
{
    return new AStarSolutionInfo()
    {
        OpenStates = _solutionNodes.Count,
        ClosedStates = _closedStates.Count,
    };
}
}

private AStarNode FindSolutionPath(Board board)
{
    var node = new AStarNode()
    {
        Weight = 0,
        Board = new Board(board),
        // It's a top node that represents the beginning state of the board
        PositionUpdate = null,
    };

    _solutionNodes.Enqueue(node, node.Weight);

    while (!node.Board.Validate())
    {
        if (_stopDateTime <= DateTime.Now)
        {
            throw new TimeoutException();
        }

        node = _solutionNodes.Dequeue();

        if (node is { PositionUpdate: not null })
        {
            node.Board.Move(node.PositionUpdate);
            _closedStates.Add(new List<Position>(node.Board.Positions));
        }

        // Generate and filter subNodes
        var subNodes = new List<AStarNode>();
        foreach (var subNode in GenerateSubNodes(node.Board))
        {
            // This can be optimized with creating a hash code and saving it
            // instead of a list of positions
            node.Board.Move(subNode.PositionUpdate);
            var state = new List<Position>(node.Board.Positions);
            node.Board.MoveBack(subNode.PositionUpdate);

            var cState = _closedStates.FirstOrDefault(x =>
state.All(x.Contains));
            if (cState is null)
            {

```

```

        subNodes.Add(subNode);
    }
}

foreach (var sNode in subNodes)
{
    sNode.Weight = Heuristic(sNode, node.Weight);
    _solutionNodes.Enqueue(sNode, sNode.Weight);
}

if (_solutionNodes.Count is 0)
{
    throw new Exception("Solution wasn't found! Checked states: " +
_closedStates.Count);
}
}

return node;
}

private List<AStarNode> GenerateSubNodes(Board board)
{
    var safeCells = board.GetSafeCellsExt();

    return safeCells.Select(x => new AStarNode()
    {
        Weight = 0,
        Board = new Board(board),
        PositionUpdate = new QueenPositionUpdate()
        {
            CurrentPosition = x.Current,
            NewPosition = x.New,
        },
    }).ToList();
}

private int Heuristic(AStarNode node, int currentWeight)
{
    var weight = currentWeight;

    node.Board.Move(node.PositionUpdate);
    foreach (var pos in node.Board.Positions)
    {
        // -1 requires cause method returns a total number of the queens at
        the row, col or diagonal
        weight += node.Board.GetQueensAmountOnRow(pos.Row) - 1;
        weight += node.Board.GetQueensAmountOnColumn(pos.Column) - 1;
        weight += node.Board.GetQueensAmountOnDiagonals(pos.Row, pos.Column)
- 1;
    }
    node.Board.MoveBack(node.PositionUpdate);

    return weight;
}
}

```

3.2.2 Приклади роботи

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

```
##### LDFS #####
Start date & time: Чт 23.09.21 12:49:56

-----
.  X  .  .  X  X  .  .
.  .  .  .  .  .  .  .
X  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .
.  .  .  X  .  .  X  .
.  .  .  .  .  .  .  .
.  .  .  .  .  .  X  .
.  .  X  .  .  .  .  .
-----

.  .  .  .  .  X  .  .
.  .  X  .  .  .  .  .
X  .  .  .  .  .  .  .
.  .  .  .  .  .  .  X
.  .  .  X  .  .  .  .
.  X  .  .  .  .  .  .
.  .  .  .  .  .  X  .
.  .  .  .  X  .  .  .
-----

Complete date & time: Чт 23.09.21 12:49:56
Depth: 10
Backtracks: 0
Total steps: 11
-----
```

Рисунок 3.1 – Алгоритм LDFS

```
##### AStar #####
Start date & time: Чт 23.09.21 12:49:51

-----
.  X  .  .  X  X  .  .
.  .  .  .  .  .  .  .
X  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .
.  .  .  X  .  .  X  .
.  .  .  .  .  .  .  .
.  .  .  .  .  .  X  .
.  .  X  .  .  .  .  .
-----

.  .  .  X  .  .  .  .
.  X  .  .  .  .  .  .
.  .  .  .  X  .  .  .
.  .  .  .  .  .  .  X
.  .  .  .  .  X  .  .
X  .  .  .  .  .  .  .
.  .  X  .  .  .  .  .
.  .  .  .  .  .  X  .
-----

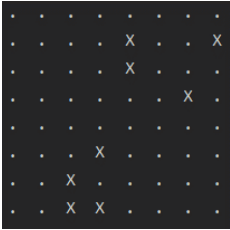
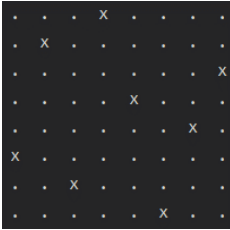
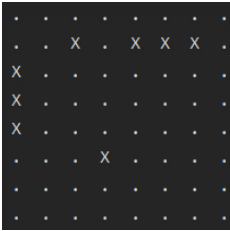
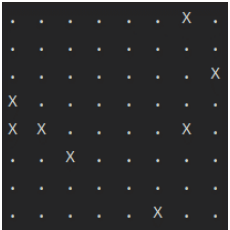
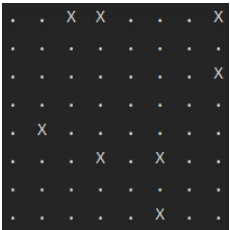
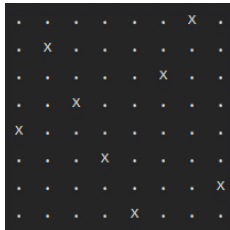
Complete date & time: Чт 23.09.21 12:49:56
Open nodes: 4854
Checked nodes: 2432
-----
```


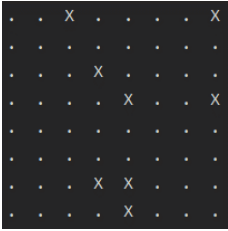
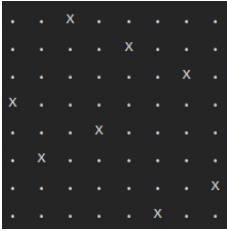
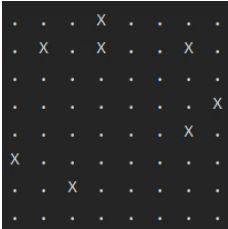
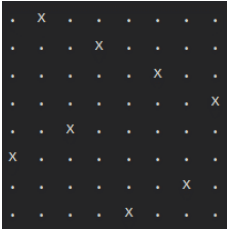
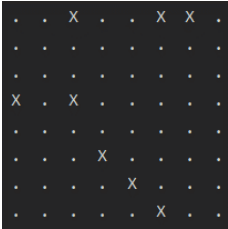
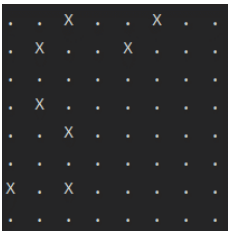
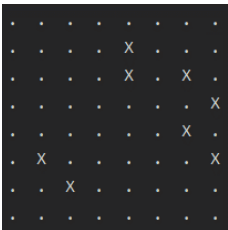
Рисунок 3.2 – Алгоритм A*

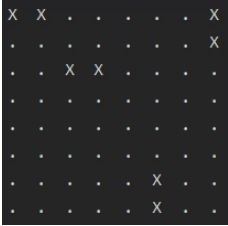
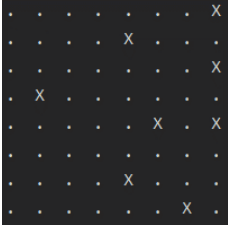
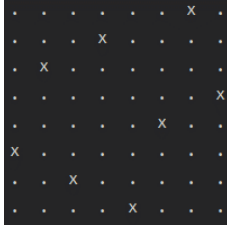
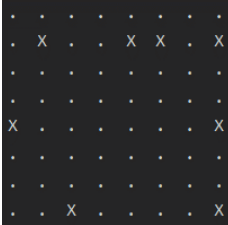
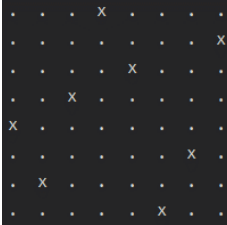
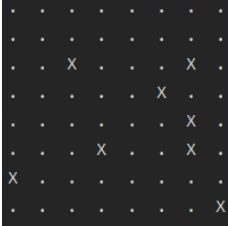
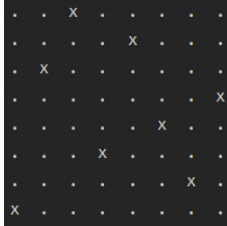
3.3 Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання алгоритму **LDFS**

Початкові стани	Кінцевий стан	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам'яті
Стан 1 		171	44	332	117
Стан 2 	Cutoff	1798	623	3420	999
Стан 3 	Cutoff	2192	812	4003	999
Стан 4 		109	22	217	86

Стан 5 	Cutoff	1950	744	3693	999
Стан 6 		115	26	227	86
Стан 7 		43	11	73	29
Стан 8 	Cutoff	1303	305	2606	999
Стан 9 	Cutoff	1284	284	2567	999
Стан 10 	Cutoff	1488	488	2975	999

Стан 17 	Cutoff	1286	287	2570	999
Стан 18  		137	33	273	103
Стан 19  		58	10	115	47
Стан 20  		80	17	159	62

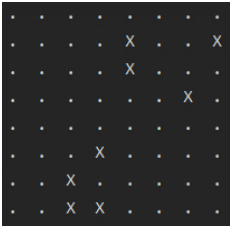
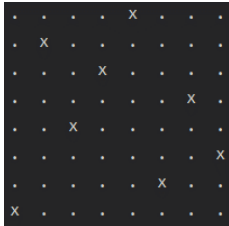
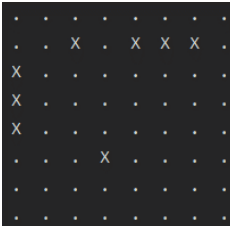
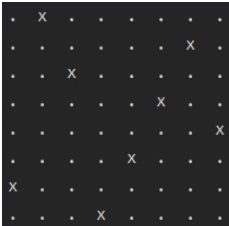
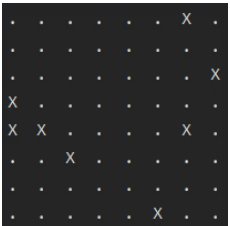
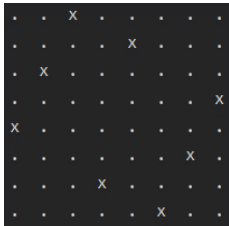
За допомогою LDFS вдалося розв'язати 11 дошок з 20. Очікувана ефективність не інформативного пошуку в загальному випадку складає приблизно 14%, у конкретно цьому випадку отримано ефективність в 55%, що є дуже гарним результатом.

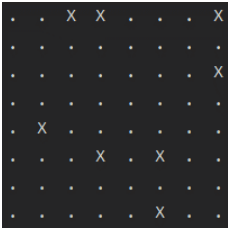
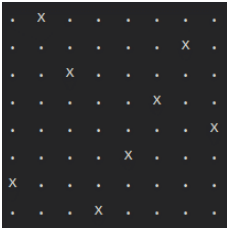
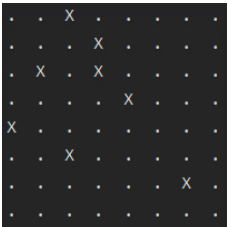
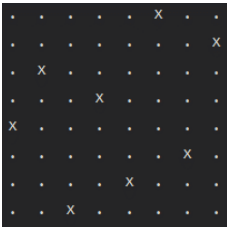
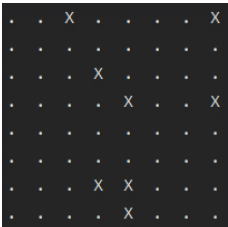
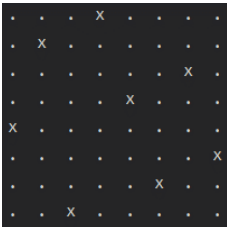
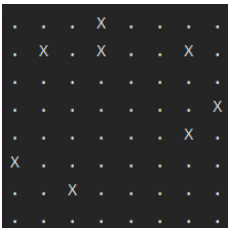
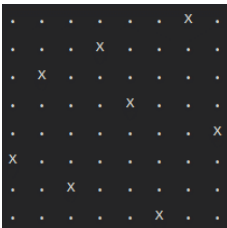
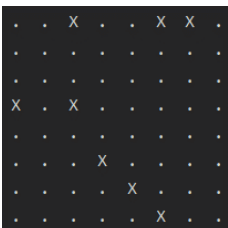
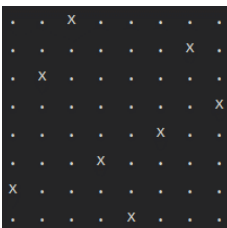
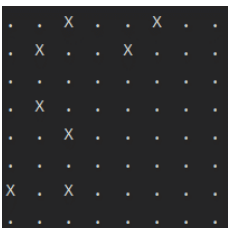
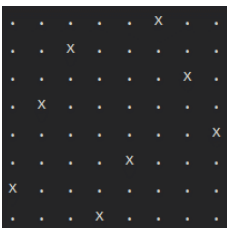
Табл. 3.2 — Зведена інформація з табл. 3.1


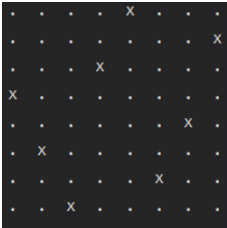
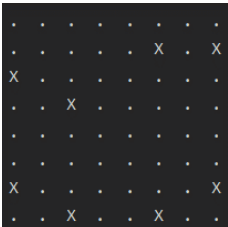
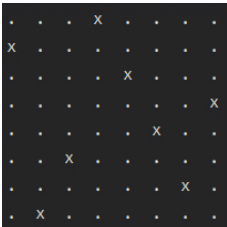
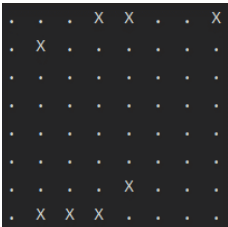
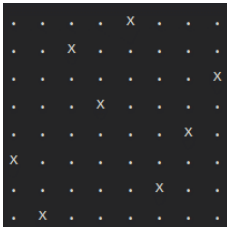
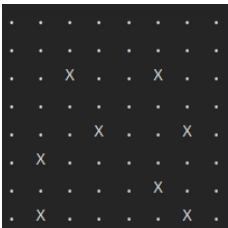
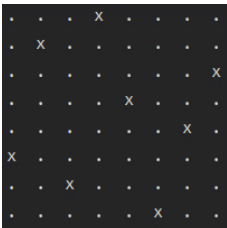
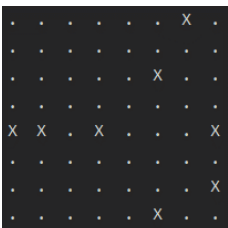
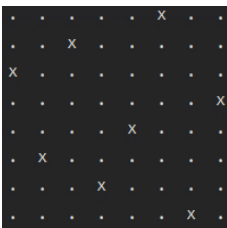
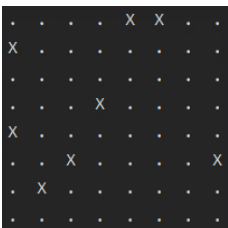
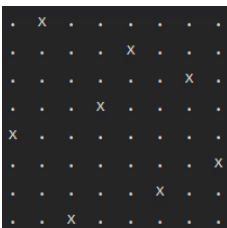
	Середня кіл-сть ітерацій	Середня кіл-сть глухих кутів	Середня кіл-сть станів
Розв'язані дошки	93	21	182
Нерозв'язані дошки	1551	467	3017
Всі дошки	749	222	1458

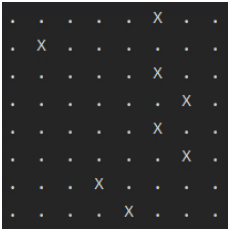
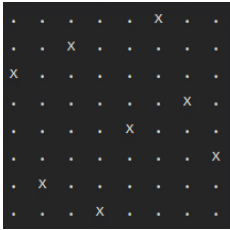
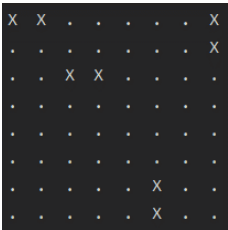
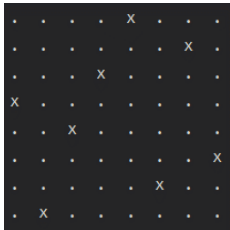
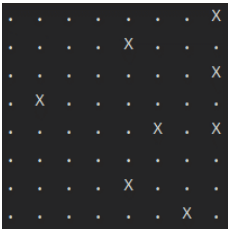
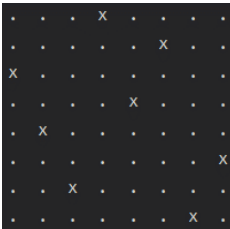
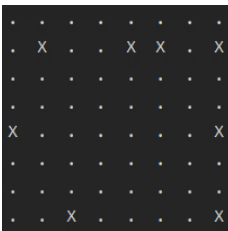
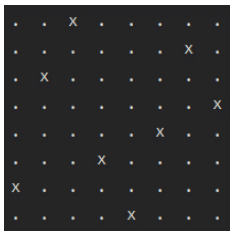
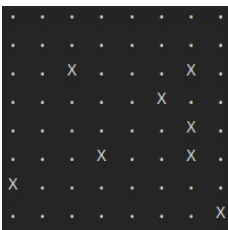
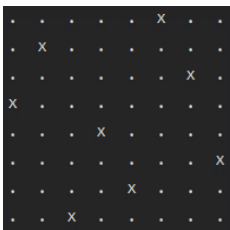
В таблиці 3.3 наведені характеристики оцінювання алгоритму A^* , задачі 8-ферзів для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання A^*

Початкові стани	Кінцевий стан	Ітерації	Всього станів	Всього станів у пам'яті
Стан 1 		5847	17003	11156
Стан 2 		1131	3420	2786
Стан 3 		4617	13568	8951

Стан 4 		3120	9626	6506
Стан 5 		1517	5049	3532
Стан 6 		2493	7609	5116
Стан 7 		1085	3448	2363
Стан 8 		3841	11896	8055
Стан 9 		5837	17492	11655

Стан 10 		171	766	595
Стан 11 		1128	3874	2746
Стан 12 		5648	17526	11878
Стан 13 		696	2549	1853
Стан 14 		492	2167	1675
Стан 15 		2248	7682	5434

Стан 16 		1534	5043	3509
Стан 17 		2435	7480	5045
Стан 18 		2423	7182	4759
Стан 19 		4508	13941	9433
Стан 20 		1896	6520	4624

A* на відміну від LDFS зміг розв'язати всі 20 дошок, що доводить той факт, що інформативний пошук у більшій кіл-сті випадків дає правильний результат. До того ж, A* дає оптимальний результат. Якщо поглянути на таблицю 3.4, то можна побачити, що він генерує і перевіряє більше станів проте через те, що цей алгоритм ітеративний, а не рекурсивний це не є проблемою.

Табл. 3.4 — Зведена інформація з табл. 3.3

	Середня кіл-сть ітерацій	Середня кіл-сть глухих кутів	Середня кіл-сть станів
Розв’язані дошки	2633	8192	5584

У таблиці 3.5 наведено данні по середній кількості згенерованих станів, ітерацій та глухих кутів.

З таблиці стає очевидно, що LDFS генерує менше станів і в цілому працює менше, бо йому не потрібно на кожній ітерації розраховувати вагу нових вершин і перевіряти чи вже були перевірені такі розстановки. Але це має і побічний ефект — цей алгоритм не гарантує те, що всі дошки будуть вирішені. У той самий час, A^* , який генерує і перевіряє більше станів у загальному випадку буде пріоритетнішим, бо має більший відсоток вдало розв’язаних шахівниць.

Табл. 3.5 — порівняння алгоритмів LDFS та A^*

	Середня кіл-сть ітерацій	Середня кіл-сть глухих кутів	Середня кіл-сть станів
A^*	2633	8192	5584
LDFS	749	222	1458

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми інформативного та неінформативного пошуку, на прикладі класичної задачі 8-ферзів.

Для розв'язання задачі в якості неінформативного алгоритму пошуку було використано Limited Depth-First Search. Він дозволяє шукати рішення задачі в графі з максимальною просторовою складністю $O(|V|)$ та case performance $O(|V| + |E|)$. Проте, через те, що в більшості імплементацій цей алгоритм рекурсивний, інколи виникають проблеми з переповненням стеку і тому, дуже важливо встановити оптимальне обмеження глибини, бо відсоток розв'язаних дошок напряду залежить від того максимальної глибини, на яку може заглибитись алгоритм.

В якості інформативного алгоритму використано A^* . Він набагато ефективніший за Limited Depth-First Search в плані використання пам'яті, проте, як показали практичні дослідження він працює довше за LDFS, але може розв'язувати дошки які не зміг вирішити його конкурент. Цей алгоритм має просторову складність $O(|V|)$ та case performance $O(|E|)$, що краще за показники LDFS.

На мою думку, алгоритми неінформативного пошуку можна використовувати для попереднього аналізу, проте вони використовують більше пам'яті і не гарантують результат. У той самий час алгоритми інформаційного пошуку мають більше шансів розв'язати задачу, але немає гарантій, що вони зможуть швидко знайти рішення.