

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 Куманецька Ірина
(шифр, прізвище, ім'я, по батькові)

Перевірив

Ахаладзе І. Е.
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1	МЕТА ЛАБОРАТОРНОЇ РОБОТИ	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ.....	8
3.1	ПСЕВДОКОД АЛГОРИТМІВ.....	8
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ.....	8
3.2.1	<i>Вихідний код.....</i>	<i>8</i>
3.2.2	<i>Приклади роботи</i>	<i>14</i>
3.3	ДОСЛІДЖЕННЯ АЛГОРИТМІВ	15
	ВИСНОВОК	26
	КРИТЕРІЇ ОЦІНЮВАННЯ	27

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. 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 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

```
BFS(graph[])
    cur_node := first state
    if cur_node is a solution
        return cur_node
    end if
    graph.Enqueue(cur_node)
    while true
        cur_node := graph.Dequeue()
        children := cur_node.GenChildren()
        for child in children
            graph.Enqueue(child)
            if child is a solution
                return child
            end if
        end for
    end while

RBFS()
    problem := first state
    return Search(problem, inf, 0)

Search(node, limit, iter)
    if node is a solution
        return node, 0
    end if
    successors := node.GenChildren()
    cost := []
    for child in successors
        cost.Add(child.CountBeaten()+iter)
    end for
    while true
        min := cost.Min()
        if min > limit
            return null, inf
        end if
        indBest := cost.IndexOf(min)
        best := successors[indBest]
        alt := cost.SecondMin()
        res, cost[indBest] := Search(best, Min(limit, alt), iter+1)
        if res != null
            return res, cost[indBest]
        end if
    end while
```

3.2 Програмна реалізація

3.2.1 Вихідний код

State.cs

```
using System;
using System.Collections.Generic;

namespace EightQueens
{
    public class State
    {
```



```

private byte[] Board { get; }

public State()
{
    Board = new byte[Const.Size];
    GenerateRandom();
}

public State(byte[] board)
{
    Board = board;
}

public List<State> GenerateChildren()
{
    List<State> children = new List<State>();

    for (int i = 0; i < Const.Size; i++)
    {
        for (byte j = 0; j < Const.Size; j++)
        {
            if (Board[i] != j)
            {
                byte[] child = new byte[Const.Size];
                for (int k = 0; k < Const.Size; k++)
                {
                    child[k] = Board[k];
                }
                child[i] = j;
                children.Add(new State(child));
            }
        }
    }

    return children;
}

private void GenerateRandom()
{
    Random random = new Random();

    for (int i = 0; i < Const.Size; i++)
    {
        byte col = (byte)random.Next(Const.Size);
        Board[i] = col;
    }
}

public int CountBeaten()
{
    return CheckColumns() + CheckLeftDiag() + CheckRightDiag();
}

private int CheckColumns()
{
    int pairs = 0;
    int count;

    for (byte col = 0; col < Const.Size; col++)
    {
        count = 0;
        for (int j = 0; j < Const.Size; j++)
        {
            if (Board[j] == col)
            {
                count++;
            }
        }
    }
}

```

```

    }

    if (count>1)
    {
        pairs += count - 1;
    }
}

return pairs;
}
private int CheckRightDiag()
{
    int pairs = 0;
    int count;

    for (int dif = 0; dif < Const.Size-1; dif++)
    {
        count = 0;
        for (int row = 0; row+dif < Const.Size; row++)
        {
            if (Board[row] == row+dif)
            {
                count++;
            }
        }

        if (count>1)
        {
            pairs += count - 1;
        }
    }

    for (int dif = 1; dif < Const.Size-1; dif++)
    {
        count = 0;
        for (int col = 0; col+dif < Const.Size; col++)
        {
            if (Board[col+dif] == col)
            {
                count++;
            }
        }

        if (count>1)
        {
            pairs += count - 1;
        }
    }

    return pairs;
}
private int CheckLeftDiag()
{
    int pairs = 0;
    int count;

    for (int sum = 1; sum < Const.Size; sum++)
    {
        count = 0;
        for (int row = 0; row < sum+1; row++)
        {
            if (Board[row] == sum-row)
            {
                count++;
            }
        }
    }
}

```

```

        }
    }

    if (count>1)
    {
        pairs += count - 1;
    }
}

for (int sum = Const.Size; sum < 2*Const.Size-2; sum++)
{
    count = 0;
    for (int row = sum-Const.Size+1; row < Const.Size; row++)
    {
        if (Board[row] == sum-row)
        {
            count++;
        }
    }

    if (count>1)
    {
        pairs += count - 1;
    }
}

return pairs;
}

public void PrintBoard()
{
    for (int i = 0; i < Const.Size; i++)
    {
        for (byte j = 0; j < Const.Size; j++)
        {
            if (Board[i] == j)
            {
                Console.Write("* ");
            }
            else
            {
                Console.Write("_ ");
            }
        }
        Console.WriteLine();
    }
    Console.WriteLine();
}

public bool Same(State other)
{
    bool res = true;
    for (int i = 0; i < Const.Size; i++)
    {
        if (Board[i] != other.Board[i])
        {
            res = false;
        }
    }

    return res;
}
}
}

```

BfsSolver.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

namespace EightQueens
{
    public class BfsSolver
    {
        private Queue<State> _boards = new();

        public void Solve()
        {
            State cur = new State();
            cur.PrintBoard();
            if (cur.CountBeaten() == 0)
            {
                return;
            }

            _boards.Enqueue(cur);
            bool solved = false;
            List<State> children;
            int iterations = 0;
            Stopwatch sw = new Stopwatch();
            sw.Start();

            while (!solved)
            {
                cur = _boards.Dequeue();
                children = cur.GenerateChildren();
                foreach (var child in children)
                {
                    _boards.Enqueue(child);
                    if (child.CountBeaten() == 0)
                    {
                        solved = true;
                        cur = child;
                        break;
                    }
                }
                iterations++;
            }

            sw.Stop();
            Console.WriteLine(sw.Elapsed);
            Console.WriteLine("Generated states: " + (iterations +
            _boards.Count));
            Console.WriteLine("Not used: " + _boards.Count);
            cur.PrintBoard();
        }
    }
}
```

Rbfs.cs

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace EightQueens
```

```

{
    public class Rbfs
    {
        private int _count;
        public void Solve()
        {
            State problem = new State();
            problem.PrintBoard();
            State solution = Search(problem, int.MaxValue, 0).Item1;
            if (solution == null)
            {
                Console.WriteLine("Solution not found.");
            }
            else
            {
                solution.PrintBoard();
            }
            Console.WriteLine("Generated states: " + (Const.Size*(Const.Size-
1)*_count+1));
        }

        private (State, int) Search(State node, int limit, int iter)
        {
            State res;
            _count++;

            if (node.CountBeaten() == 0)
            {
                Console.WriteLine("Moves: " + iter);
                return (node, 0);
            }

            List<State> successors = node.GenerateChildren();
            List<int> cost = new List<int>();

            foreach (var child in successors)
            {
                cost.Add(child.CountBeaten()+iter);
            }

            while (true)
            {
                int min = cost.Min();

                if (min > limit)
                {
                    return (null, Int32.MaxValue);
                }

                int indBest = cost.IndexOf(min);
                State best = successors[indBest];
                cost.RemoveAt(indBest);
                int alt = cost.Min();
                cost.Insert(indBest, min);
                (res, cost[indBest]) = Search(best, Math.Min(limit, alt), iter +
1);

                if (res != null)
                {
                    return (res, cost[indBest]);
                }
            }
        }
    }
}

```

3.2.2 Приклади роботи

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

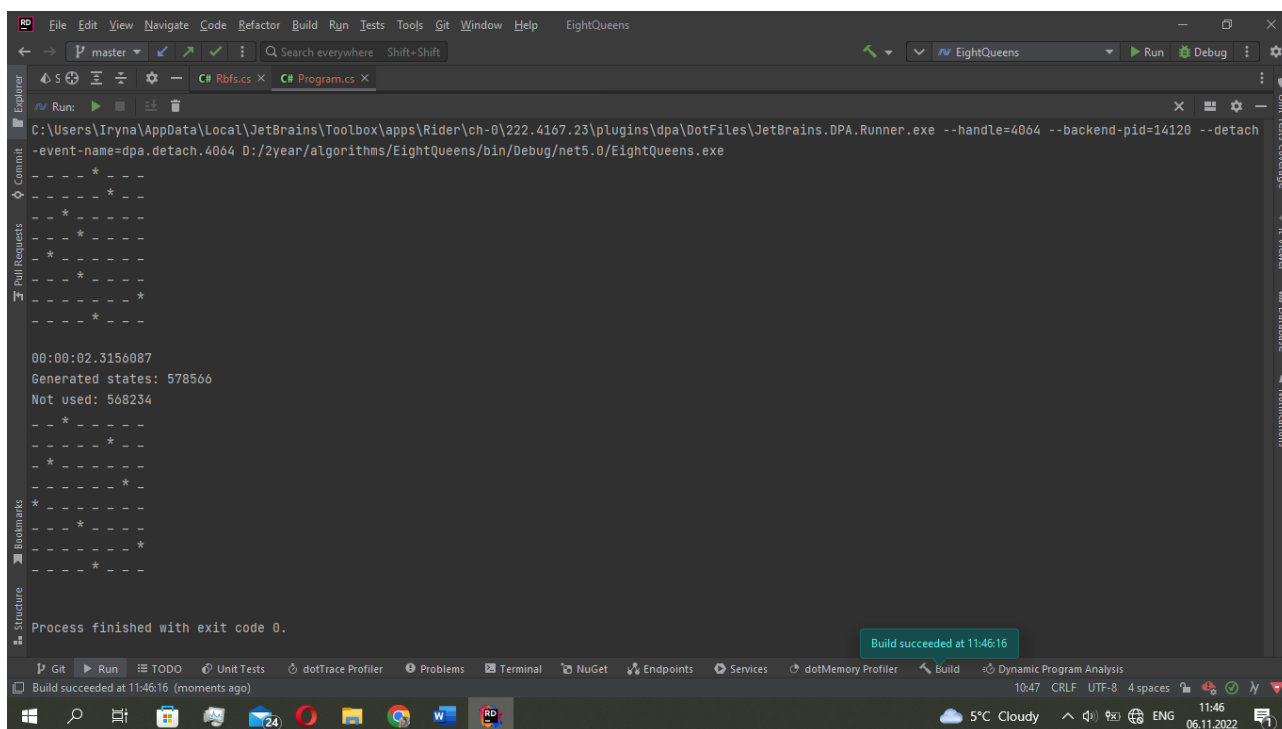


Рисунок 3.1 – Алгоритм BFS

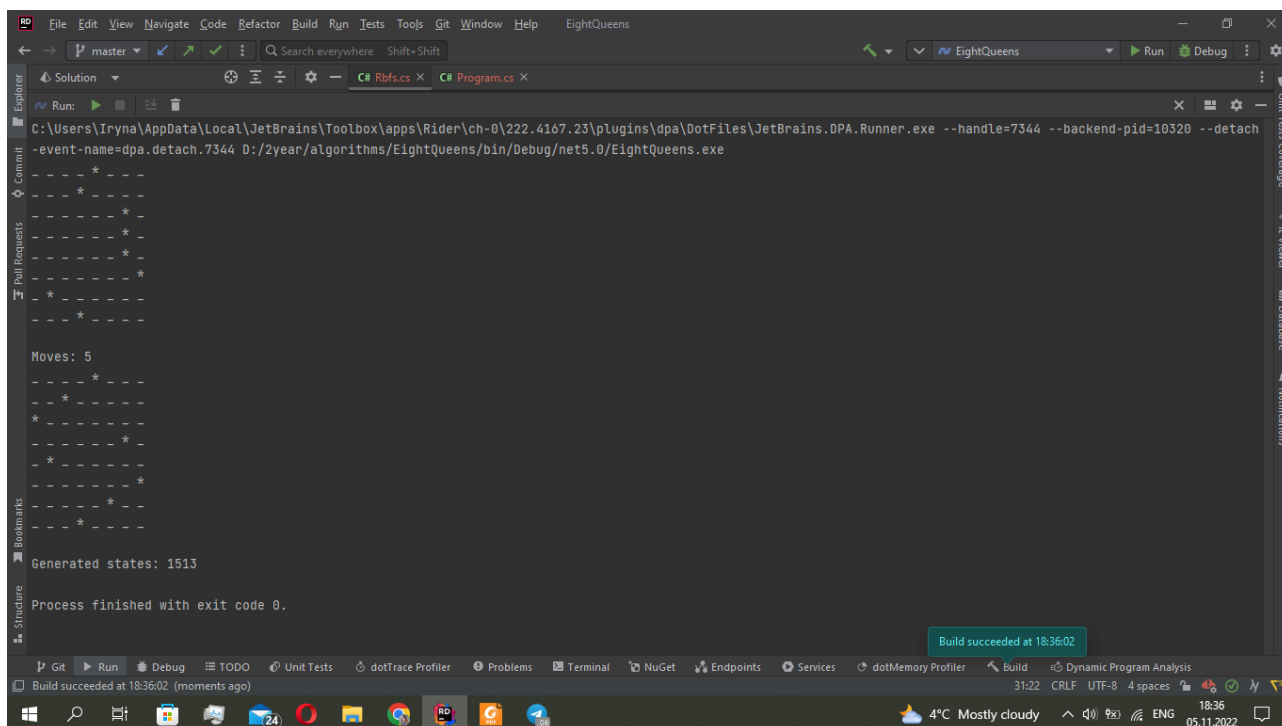
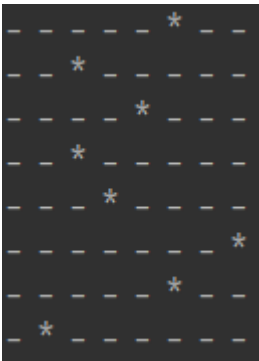
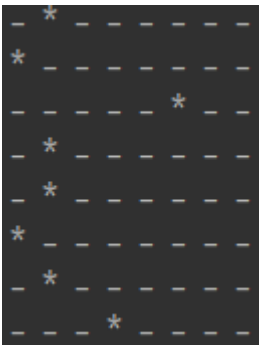
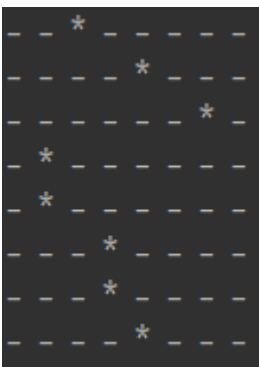


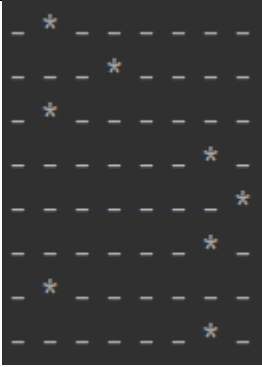
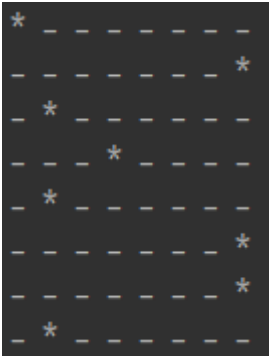

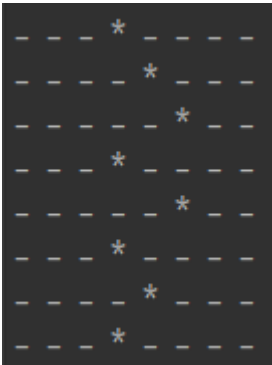
Рисунок 3.2 – Алгоритм RBFS



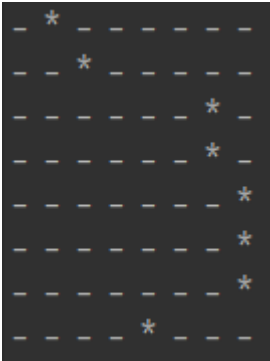

3.3 Дослідження алгоритмів

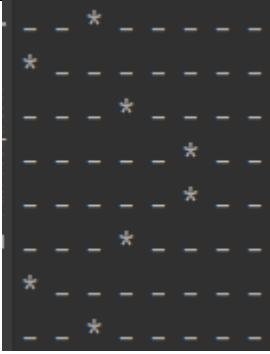


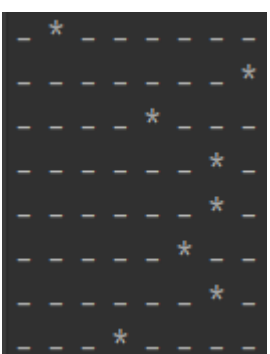
В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS задачі 8 ферзів для 20 початкових станів.

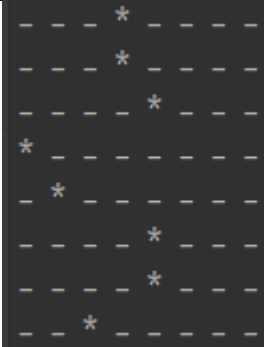
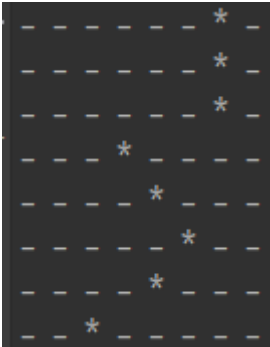

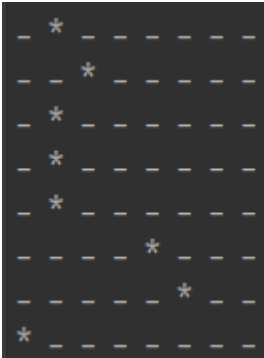
Таблиця 3.1 – Характеристики оцінювання BFS

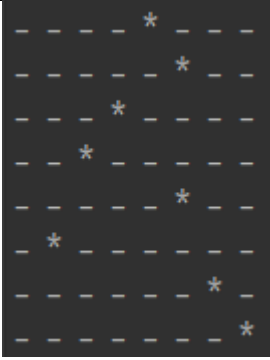
Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам'яті
Стан 1 	4	0	3586910	3522858
Стан 2 	5	0	43109178	42339371
Стан 3 	5	0	51028252	50117033
Стан 4	5	0	54195677	53227897

				
Стан 5 	5	0	41846388	41099131
Стан 6 	5	0	21847376	21457244
Стан 7 	5	0	32403891	31825250
Стан 8	4	0	927525	910962

				
Стан 9 	5	0	131115086	128773745
Стан 10 	4	0	952044	935043
Стан 11 	4	0	604011	593225
Стан 12	4	0	814345	799803

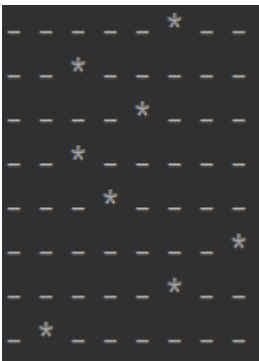
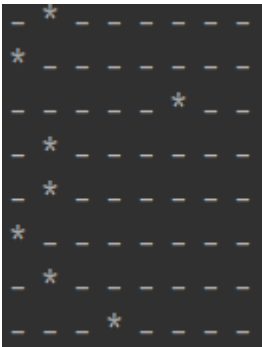
				
Стан 13 	5	0	41142742	40408050
Стан 14 	4	0	967789	950507
Стан 15 	4	0	3407473	3346625
Стан 16	4	0	942798	925692

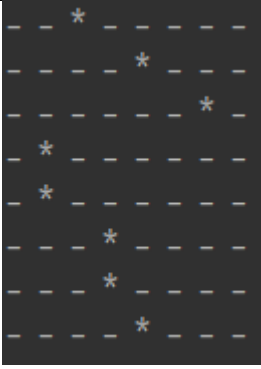
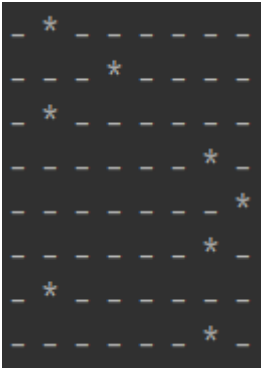
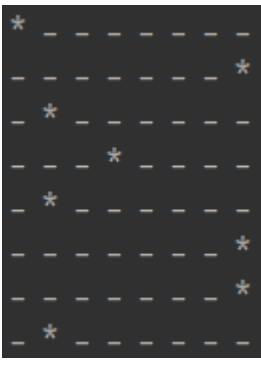

				
Стан 17 	5	0	13079118	12845562
Стан 18 	2	0	1723	1682
Стан 19 	4	0	21831705	21441853
Стан 20	4	0	747263	733919

				
---	--	--	--	--


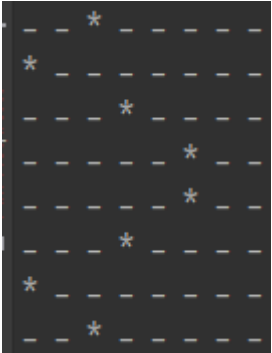


В таблиці 3.2 наведені характеристики оцінювання алгоритму RBFS задачі 8 ферзів для 20 початкових станів.

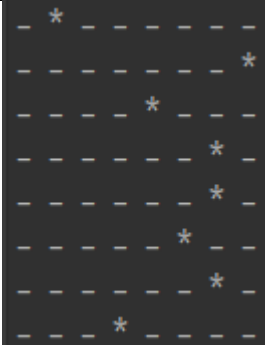
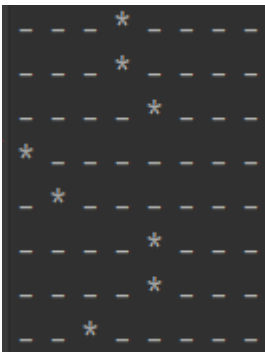
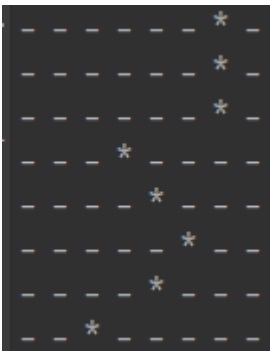

Таблиця 3.2 – Характеристики оцінювання RBFS

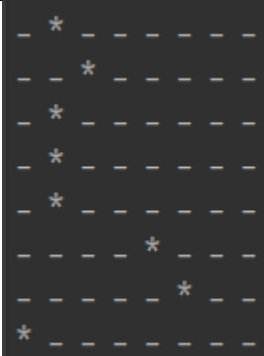

Початкові стани	Ітерації	К-сть кутів	гл. Всього станів	Всього станів у пам'яті
Стан 1 	5	17	1289	1289
Стан 2 	6	27	1905	1905
Стан 3	6	120	7113	7113

				
Стан 4 	6	236	13609	13609
Стан 5 	5	0	337	337
Стан 6 	5	0	337	337
Стан 7	7	187	10921	10921

				
Стан 8 	4	0	281	281
Стан 9 	5	384	21841	21841
Стан 10 	4	0	281	281
Стан 11	4	0	281	281

				
Стан 12 	4	0	281	281
Стан 13 	5	5	617	617
Стан 14 	5	1	393	393
Стан 15	5	89	5321	5321

				
Стан 16 	5	1	393	393
Стан 17 	5	0	337	337
Стан 18 	2	0	169	169
Стан 19	6	40	2633	2633

				
Стан 20 	5	5	617	617

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто розв'язання задачі 8 ферзів алгоритмом пошуку вшир та рекурсивним пошуком по першому співпадінню. Обидва алгоритми було записано у вигляді псевдокоду, реалізовано засобами мови C# та протестовано на 20 початкових станах. Зазначимо, що тестування проводилися на спрощених початкових станах, щоб алгоритм неінформативного пошуку виконувався за адекватний час (до 30 хв).

Для алгоритму пошуку вшир середня кількість кроків для розв'язування задачі – 4.35, а середня кількість згенерованих станів дорівнює 23 227 565. Для алгоритму рекурсивного пошуку за першим співпадінням середня кількість кроків дорівнює 4.95, а середня кількість згенерованих станів – 3458. При цьому загальна кількість глухих кутів під час роботи інформативного пошуку дорівнює 55.6.

Отже, RBFS в середньому розв'язує задачу за трохи більшу кількість кроків, ніж BFS, проте інформативний пошук генерує в тисячі разів менше станів, ніж неінформативний. Через це інформативний пошук працює в рази швидше та вимагає набагато менше пам'яті (в обох алгоритмах майже усі стани зберігаються в пам'яті одночасно). Незважаючи на періодичне потрапляння у глухий кут, використання інформативного пошуку по всіх параметрах видається більш ефективним.

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%.