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-13 Макарчук Лідія Олександрівна* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олексій Олександрович* _____
(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 (1Гб).

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, щоскладається 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

16 варіант

№	Задача	АНП	АІП	АЛП	Func
16	8-ферзів	IDS	A*		F2

3.1 Псевдокод алгоритмів

IDS(initialNode)

1. ПОЧАТОК
2. Цикл для depth від 0 до ∞:
   2.1. result = DLS(initialNode, depth)
   2.2. ЯКЩО result != Indicator.cutoff ТО ПОВЕРНУТИ result
3. КІНЕЦЬ

DLS(node, limit)

1. ПОЧАТОК
2. Присвоїти cutoff_occurred = false
3. ЯКЩО node.state.goal() == true ТО
   3.1. solution = node.state
   3.2. ПОВЕРНУТИ Indicator.goal
4. ЯКЩО currDepth == limit ТО ПОВЕРНУТИ Indicator.cutoff
5. successors = Expand(node)
6. Цикл проходу по всіх нащадках successor у масиві successors:
   6.1. Присвоїти result = DLS(successor, limit)
   6.2. ЯКЩО result == Indicator.cutoff ТО Присвоїти cutoff_occurred = true
   6.3. ІНАКШЕ ЯКЩО result != Indicator.failure ТО ПОВЕРНУТИ result
7. ЯКЩО cutoff_occurred == true ТО ПОВЕРНУТИ Indicator.cutoff
8. ІНАКШЕ ПОВЕРНУТИ Indicator.failure

9. КІНЕЦЬ

Expand(node)

1. ПОЧАТОК
2. Нащадки successors = ∅
3. Цикл проходу по всіх нащадках node:
    3.1. Створити новий вузел s
    3.2. s.state = новий стан
    3.3. s.parent = node
    3.4. s.depth = node.depth + 1
    3.5. s.action = [column, oldRow, newRow]
    3.6. додати s до successors
4. ПОВЕРНУТИ successors
5. КІНЕЦЬ

A*(initialNode)

1. ПОЧАТОК
2. Список вузлів для розгортання openList = ∅
3. node.score = F2(node.state)
4. Додати початковий вузол node до openList
5. ПОКИ openList != ∅
    5.1. current = вузол із openList з найменшим значенням score
    5.2. ЯКЩО current.state.goal() == true ТО
        5.2.1. solution = current.state
        5.2.2. ПОВЕРНУТИ true
    5.3. Видалити current із openList
    5.4. successors = Expand(node)
    5.5. Цикл проходу по всіх нащадках successor у масиві successors:
        5.5.1. successor.score = F2(successor.state)
        5.5.2. Додати вузол successor до openList

6. КІНЕЦЬ

F2(state)

1. ПОЧАТОК
2. Присвоїти sum = 0
3. sum = sum + кількість пар ферзів в однакових рядках
4. sum = sum + кількість пар ферзів в однакових стовпцях
5. sum = sum + кількість пар ферзів в однакових головних діагоналях
6. sum = sum + кількість пар ферзів в однакових побічних діагоналях
7. ПОВЕРНУТИ sum
8. КІНЕЦЬ

3.2 Програмна реалізація

3.2.1 Вихідний код

//Модуль SolutionTree

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Linq;
using System.Diagnostics;

namespace laba2
{
   internal class SolutionTree
   {/*  class for algorithms implementation  */

      const long bytesInOneGB = 1024 * 1024 * 1024;
      const long msIn30Minutes = 1800000;
      Stopwatch innerTimer;

      /* some variables for statistic */
      private long iterations; //how many iterations of the algotithm were made
      long nodesSaved;   //how many nodes the program holds at the same time
      private long totalNodesCreated;  //how many nodes have been created during the algorithm working
      private List<int[]> path;   //stores all actions

      /* attributes for solving the problem */
      private int N;
      private Node root;
      private State solution; //solution state

      public State Solution { get => solution; }
      enum Indicator
      {
         goal,   //solution is found
         cutoff, //the solution may exist but not at this depth
```

```
            failure //the solution is not exist
        }
        /*   Constructor   */
        public SolutionTree(int N, State initialState)
        {
            this.N = N;
            this.iterations = 0;
            this.totalNodesCreated = 1;
            this.path = new List<int[]>();
            this.nodesSaved = 1;

            Node.InitializeMemoryUsage(N);
            root = new Node(initialState, null, 0, new int[] {-1, -1, -1}); //create the first (root) node of the tree

            innerTimer = new Stopwatch();
            innerTimer.Start();
        }
        /*  getters/setters */
        public long Iterations => iterations;
        public long TotalNodesCreated => totalNodesCreated;
        public List<int[]> Path => path;
        public long NodesSaved => nodesSaved;

        /*   algorithms   */

        private List<Node> Expand(ref Node node)
        {
            int currentDepth = node.Depth;
            List<Node> successors = new List<Node>();
            if (currentDepth == N)    //if node cannot have successors
                return successors;
            int currentRow = node.getState.board[currentDepth];
            for (int row = 0; row < N; row++)
            {
                //if(node.getState.IsSafe(row, node.Depth))   //if new position is safe then create a new state with placed queen here
                //{
                State newState = new State(node.getState);
                newState.PlaceQueen(row, currentDepth);   //node.Depth is a column index
                successors.Add(new Node(newState, node, currentDepth + 1, new int[] { currentDepth, currentRow, row }));
                //}
            }
            return successors;
        }
        private Indicator DLS(Node node, int limit, ref long currNodesInMemory)  //Depth-Limited-Search
        {
            iterations++;
            bool cutoff_occurred = false;
            if (node.getState.IsGoal()) //check if node has the goal state
            {
                solution = new State(node.getState);
                getPath(node);
                return Indicator.goal;
            }
            if (node.Depth == limit)    //if depth == limit stop search
            {
                currNodesInMemory--;
                return Indicator.cutoff;
            }
            List<Node> successors = Expand(ref node);

            /*statistic*/
            totalNodesCreated += successors.Count;  //statistic
            currNodesInMemory += successors.Count;
            if (currNodesInMemory > nodesSaved) nodesSaved = currNodesInMemory;
            if (currNodesInMemory * Node.memoryUsageInBytes > bytesInOneGB || innerTimer.ElapsedMilliseconds >
msIn30Minutes) //memory&time restriction
```

```
            return Indicator.failure;
        /*statistic*/

        for (int i = 0; i < successors.Count; i++)  //for every node in successors
        {
            Indicator result = DLS(successors[i], limit, ref currNodesInMemory);
            if (result == Indicator.cutoff)
                cutoff_occurred = true;
            else if(result != Indicator.failure)    //if is goal
                return result;
        }
        currNodesInMemory--;
        if (cutoff_occurred)
            return Indicator.cutoff;
        else
            return Indicator.failure;
    }

    public bool IDS()   //Iterative-Deepening-Search
    {
        for (int i=0; i<=N; i++)
        {
            long currNodeSaved = 0;
            Indicator result = DLS(root, i, ref currNodeSaved);
            if (result == Indicator.goal)
                return true;
            else if (result == Indicator.failure)
                return false;
        }
        return false;
    }

    public bool AStar()
    {
        PriorityQueue<Node, int> openList = new PriorityQueue<Node, int>(); //stores nodes that should be expanded
        openList.Enqueue(root, root.getState.F2()); //add initial node to opneList
        while(openList.Count > 0)   //while open list is not empty
        {
            iterations++;
            Node current = openList.Dequeue(); //get node with min value F2
            if(current.getState.IsGoal())
            {
                solution = new State(current.getState);
                getPath(current);
                return true;
            }
            List<Node> successors = Expand(ref current);
            totalNodesCreated += successors.Count;
            if (openList.Count * Node.memoryUsageInBytes > bytesInOneGB || innerTimer.ElapsedMilliseconds >
msIn30Minutes)//memory&time restriction
                return false;
                for (int i = 0; i < successors.Count; i++)
            {
                openList.Enqueue(successors[i], successors[i].getState.F2());
            }
            if (openList.Count > nodesSaved)
                nodesSaved = openList.Count;
        }
        return false;
    }

  }
}
```

//Модуль Node

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace laba2
{
   internal class Node
   {
      /*  class for building searching solution tree  */
      private State state;    //current state of the board; indicies represent rows, numbers represent columns of a board
      private Node? parent;
      private int depth;  //how many states are from initial one
      private int[] action;   //[0] - column number, [1] - old row, [2] - new row

      public Node(State currState, Node? parent, int depth, int[] action)
      {
         this.state = new State(currState);
         this.depth = depth;
         this.parent = parent;
         this.action = action;
      }

      public State getState => state;
      public int Depth => depth;
      public Node? Parent => parent;
      public int[] Action => action;
   }
}
```

//Модуль State

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace laba2
{
   internal class State
   {
      public int[] board;
      public State(int N)
      {
         board = new int[N];
      }
      public State(int[] board)
      {
         this.board = board;
      }
      public State(State state)
      {
         this.board = new int[state.board.Length];
         Array.Copy(state.board, this.board, state.board.Length);
      }

      public bool IsGoal()  //check if all the queens are in correct places
      {
         for (int j = 1; j< board.Length;j++)    // check columns from the end
         {
            for(int k=j-1;k>=0;k--) // with every column before j one
            {
               if(board[j]== board[k])  //if in the same row
                  return false;
```

```
                int diff = j - k;   //how far the column k is from column j
                if (board[j] - diff == board[k] || board[j] + diff == board[k])   //check diagonals
                    return false;
            }
        }
        return true;
    }
    public bool IsSafe(int row, int columnIndex)   //checks if new queen does not break the rules in compare with previous
located
    {
        for (int k = columnIndex - 1; k >= 0; k--)
        {
            if (row == board[k])  //if in the same row
                return false;
            int diff = columnIndex - k;
            if (row - diff == board[k] || row + diff == board[k])   //check diagonals
                return false;
        }
        return true;
    }
    public void PlaceQueen(int row, int columnIndex)
    {
        board[columnIndex] = row;
    }
    public int F2()
    {
        int sum = 0;
        int[] numberOfQueenInRow = new int[board.Length];
        for(int i=0; i<board.Length; i++)
        {
            numberOfQueenInRow[board[i]]++;
        }
        for(int i=0; i< numberOfQueenInRow.Length;i++)  //calculate attacking pairs in rows
        {
            if (numberOfQueenInRow[i] > 1)
                sum += CalculateCombinations(numberOfQueenInRow[i]);
        }
        for (int j = board.Length-1; j > 0; j--)    // check columns from the end
        {
            for (int k = j - 1; k >= 0; k--) // check with in column before j
            {
                int diff = j - k;   //how far the column k is from column j
                if (board[j] - diff == board[k])   //check primary diagonals
                    sum++;
                if (board[j] + diff == board[k])    //check secondary diagonals
                    sum++;
            }
        }
        return sum;
    }
    private static int CalculateCombinations(int n)  //calculate combinations C(n, r)
    {
        const int pairOfQueens = 2;
        return n*(n-1) / pairOfQueens;
    }
  }
}
```

3.2.2 Приклади роботи

На рисунках 3.1.1, 3.1.2 і 3.2.1, 3.2.2 показані приклади роботи програми для різних алгоритмів пошуку.

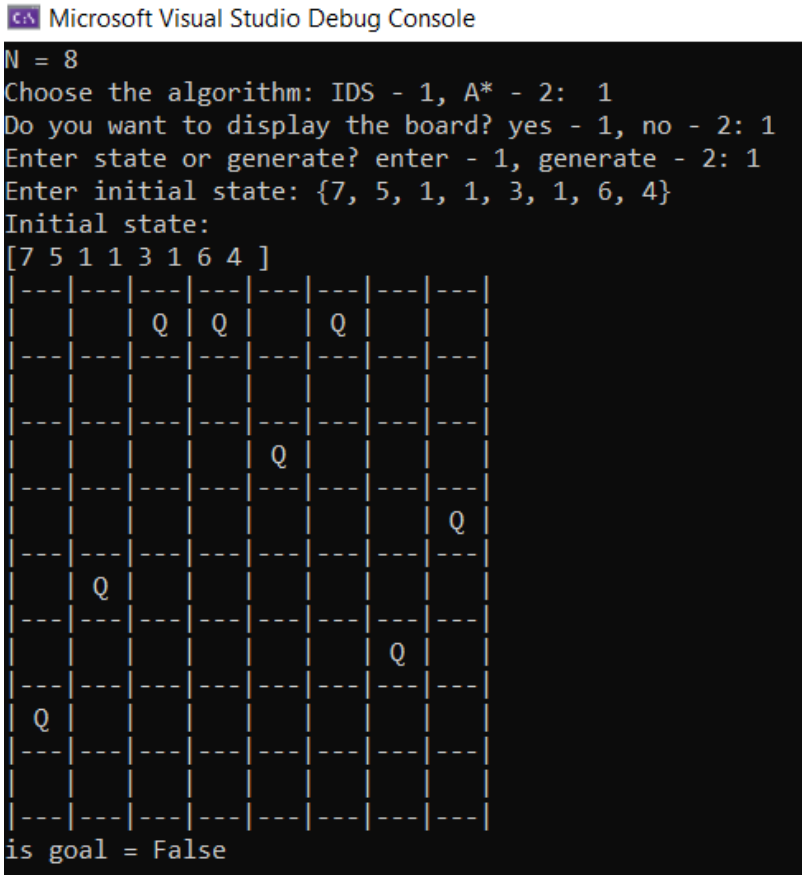

Рисунок 3.1.1 – Алгоритм IDS

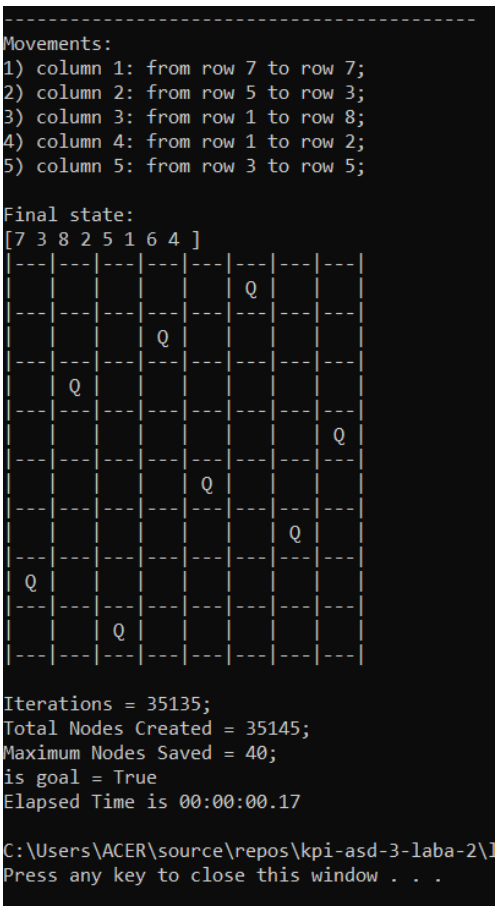

Рисунок 3.1.2 – Алгоритм IDS

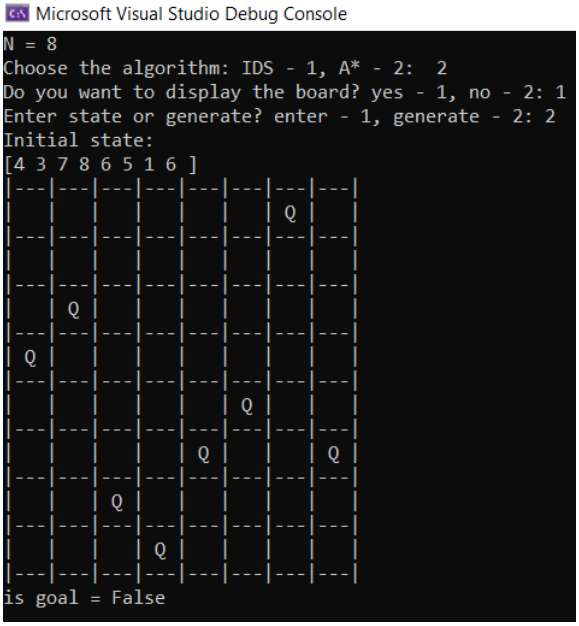

Рисунок 3.2.1 – Алгоритм A*

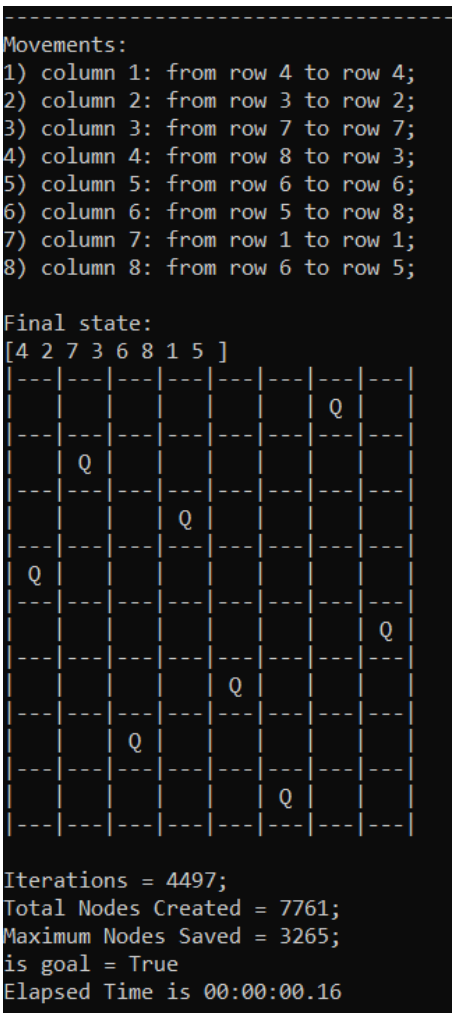

Рисунок 3.2.2 – Алгоритм A*

3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму IDS, задачі про вісім ферзів для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму IDS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1 {7, 5, 1, 1, 3, 1, 6, 4}	35135	0	35145	40
Стан 2 {7, 6, 1, 1, 5, 1, 4, 8}	27709	0	27721	40
Стан 3 {6, 1, 5, 4, 3, 1, 3, 6}	20536	0	20545	40
Стан 4 {1, 7, 6, 6, 1, 4, 5, 4}	528088	0	528105	56
Стан 5 {6, 4, 1, 2, 2, 7, 4, 3}	584585	0	584601	56
Стан 6 {3, 2, 1, 4, 3, 6, 3, 6}	158018	0	158033	48
Стан 7 {7, 6, 4, 2, 3, 3, 2, 7}	227146	0	227161	48
Стан 8 {7, 6, 1, 5, 4, 1, 7, 6}	972534	0	972553	56
Стан 9 {3, 1, 2, 3, 3, 5, 7, 2}	17868	0	17881	40
Стан 10 {8, 4, 4, 2, 4, 6, 3, 4}	528088	0	528105	56
Стан 11 {3, 6, 7, 8, 3, 3, 2, 6}	140747	0	140761	48
Стан 12 {4, 2, 3, 3, 6, 2, 2, 2}	1143426	0	1143441	56
Стан 13 {8, 4, 7, 6, 6, 5, 7, 7}	1301813	0	1301833	56
Стан 14 {6, 6, 3, 8, 5, 6, 4, 5}	564047	0	564065	56
Стан 15 {5, 4, 7, 6, 1, 2, 4, 3}	584585	0	584601	56
Стан 16 {1, 8, 5, 2, 5, 5, 8, 8}	1485548	0	1485569	56
Стан 17 {3, 1, 7, 2, 3, 7, 2, 5}	70508	0	70521	48
Стан 18 {1, 3, 4, 3, 2, 3, 7, 6}	972534	0	972553	56
Стан 19 {2, 2, 5, 6, 7, 2, 8, 2}	185729	0	185745	48
Стан 20 {2, 6, 7, 1, 6, 7, 5, 2}	18156	0	18169	40

В таблиці 3.2 наведені характеристики оцінювання алгоритму A*, задачі про вісім ферзів 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
Стан 1 {3, 2, 4, 5, 6, 4, 8, 5}	20	0	153	134
Стан 2 {8, 6, 4, 1, 6, 4, 2, 7}	601	0	4801	4201
Стан 3 {7, 6, 5, 3, 1, 6, 8, 2}	3	0	17	15
Стан 4 {1, 8, 7, 2, 2, 2, 2, 4}	160	0	1217	1058
Стан 5 {5, 6, 1, 6, 5, 7, 2, 7}	106	0	841	736
Стан 6 {2, 7, 1, 8, 6, 6, 5, 5}	612	0	4033	3422
Стан 7 {3, 7, 3, 3, 3, 7, 7, 3}	276	0	2089	1814
Стан 8 {1, 6, 1, 7, 2, 4, 7, 3}	297	0	2369	2073
Стан 9 {5, 3, 1, 3, 7, 8, 7, 6}	113	0	721	609
Стан 10 {4, 6, 5, 2, 6, 1, 5, 8}	661	0	5281	4621
Стан 11 {5, 2, 2, 8, 8, 2, 4, 8}	219	0	1745	1527
Стан 12 {7, 1, 8, 7, 4, 4, 2, 5}	50	0	393	344
Стан 13 {3, 3, 2, 7, 7, 1, 4, 8}	166	0	1321	1156
Стан 14 {3, 6, 8, 7, 7, 6, 1, 6}	558	0	4273	3716
Стан 15 {7, 3, 7, 6, 8, 5, 2, 6}	200	0	1593	1394
Стан 16 {6, 2, 6, 7, 7, 3, 2, 8}	69	0	545	477
Стан 17 {1, 1, 7, 3, 2, 3, 4, 6}	175	0	1361	1187
Стан 18 {6, 6, 4, 3, 8, 1, 2, 1}	135	0	1017	883
Стан 19 {7, 1, 3, 6, 8, 5, 3, 1}	895	0	7153	6259
Стан 20 {6, 4, 8, 2, 7, 3, 4, 4}	189	0	1449	1261

В таблиці 3.3 наведені середні значення характеристик оцінювання алгоритмів IDS та A* для задачі про вісім ферзів.

Таблиця 3.3 – Середні значення характеристик оцінювання алгоритмів IDS та A*

Назва алгоритму	Середня к-сть ітерацій	Середня к-сть гл. кутів	Середня к-сть всіх станів	Середня к-сть всіх станів у пам’яті
IDS	478340	0	478355,4	50
A*	275,25	0	2118,6	1844,35

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми неінформативного та інформативного пошуку на прикладі задачі про 8 ферзів. Для кожного алгоритму було проведено по серії експериментів, після чого був зроблений порівняльний аналіз ефективності використання алгоритмів. Проаналізувавши алгоритми за всіма критеріями, можна зробити висновок, що алгоритм інформативного пошуку A* справляється із задачею швидше, ніж алгоритм неінформативного пошуку IDS. Це можна пов’язати з тим, що A* намагається розгорнути в першу чергу вузол, який є потенційно кращим за інші. Такий підхід дає змогу зменшити кількість вузлів, що переглядаються та знайти розв’язок задачі швидше. Проте кількість вузлів, що зберігає у пам’яті IDS, є набагато меншою, ніж кількість вузлів, що зберігає A*, тому за використанням пам’яті IDS буде кращим, ніж A*.

Таким чином при виборі алгоритму потрібно вирішувати, що є більш критичним у даній ситуації: пам'ять чи час, і відповідно до цього використовувати IDS або ж A*.

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%.