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 Поліщук Валерій* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе Ілля Елдарійович* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – розглянути та дослідити алгоритми неінформативного, інформативного та локального пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

2 ЗАВДАННЯ

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту (таблиця 2.1) за допомогою алгоритму неінформативного пошуку **АНП**, алгоритму інформативного пошуку **АІП,** що використовує задану евристичну функцію Func, або алгоритму локального пошуку **АЛП та бектрекінгу,** що використовує задану евристичну функцію Func.

Програму реалізувати на довільній мові програмування.

Увага! Алгоритм неінформативного пошуку **АНП,** реалізовується за принципом «AS IS», тобто так, як є, без додаткових модифікацій (таких як перевірка циклів, наприклад).

Провести серію експериментів для вивчення ефективності роботи алгоритмів. Кожний експеримент повинен відрізнятись початковим станом. Серія повинна містити не менше 20 експериментів для кожного алгоритму. Початковий стан зафіксувати у таблиці експериментів. За проведеними серіями необхідно визначити:

- середню кількість етапів (кроків), які знадобилось для досягнення розв’язку (ітерації);
- середню кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе;
- середню кількість згенерованих станів під час пошуку;
- середню кількість станів, що зберігаються в пам’яті під час роботи програми.

Передбачити можливість обмеження виконання програми за часом (30 хвилин) та використання пам’яті (1 Гб).

Використані позначення:

- **8-ферзів** – Задача про вісім ферзів полягає в такому розміщенні восьми ферзів на шахівниці, що жодна з них не ставить під удар один одного. Тобто, вони не повинні стояти в одній вертикалі, горизонталі чи діагоналі.
- **8-puzzle** – гра, що складається з 8 однакових квадратних пластинок з нанесеними числами від 1 до 8. Пластинки поміщаються в квадратну коробку, довжина сторони якої в три рази більша довжини сторони пластинок, відповідно в коробці залишається незаповненим одне квадратне поле. Мета гри – переміщаючи пластинки по коробці досягти впорядковування їх по номерах, бажано зробивши якомога менше переміщень.

- **Лабіринт** – задача пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху. Структура лабіринту зчитується з файлу, або генерується програмою.
- **LDFS** – Пошук вглиб з обмеженням глибини.
- **BFS** – Пошук вшир.
- **IDS** – Пошук вглиб з ітеративним заглибленням.
- **A*** – Пошук A*.
- **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
- **F1** – кількість пар ферзів, які б'ють один одного з урахуванням видимості (ферзь A може стояти на одній лінії з ферзем B, проте між ними стоїть ферзь C; тому A не б'є B).
- **F2** – кількість пар ферзів, які б'ють один одного без урахування видимості.
- **H1** – кількість фішок, які не стоять на своїх місцях.
- **H2** – Манхетенська відстань.
- **H3** – Евклідова відстань.
- **COLOR** – Задача розфарбування карти самостійно обраної країни, не менше 20 регіонів (областей). Необхідно розфарбувати карту не більше ніж у 4 різні кольори. Мається на увазі приписування кожному регіону власного кольору так, щоб кольори сусідніх регіонів відрізнялись. Використовувати евристичну функцію, яка повертає кількість пар суміжних вузлів, що мають однаковий колір (тобто кількість конфліктів). Реалізувати алгоритм пошуку із поверненнями (backtracking) для розв'язання поставленої задачі. Для підвищення швидкодії роботи алгоритму використати евристичну функцію, а початковим станом вважати випадкову вершину.
- **HILL –** Пошук зі сходженням на вершину з використанням із використанням руху вбік (на 100 кроків) та випадковим перезапуском (кількість необхідних разів запуску визначити самостійно).
- **ANNEAL** – Локальний пошук із симуляцією відпалу. Робоча характеристика – залежність температури T від часу роботи алгоритму t. Можна розглядати лінійну залежність: $T = 1000 - k \cdot t$, де k – змінний коефіцієнт.
- **BEAM** – Локальний променевий пошук. Робоча характеристика – кількість променів k. Експерименти проводи із кількістю променів від 2 до 21.
- **MRV** – евристика мінімальної кількості значень;
- **DGR** – ступенева евристика.

Таблиця 2.1 – Варіанти алгоритмів

№	Задача	АНП	АІП	АЛП	Func
1	Лабіринт	LDFS	A*		H2
2	Лабіринт	LDFS	RBFS		H3
3	Лабіринт	BFS	A*		H2
4	Лабіринт	BFS	RBFS		H3
5	Лабіринт	IDS	A*		H2
6	Лабіринт	IDS	RBFS		H3
7	8-ферзів	LDFS	A*		F1
8	8-ферзів	LDFS	A*		F2
9	8-ферзів	LDFS	RBFS		F1
10	8-ферзів	LDFS	RBFS		F2
11	8-ферзів	BFS	A*		F1
12	8-ферзів	BFS	A*		F2
13	8-ферзів	BFS	RBFS		F1
14	8-ферзів	BFS	RBFS		F2
15	8-ферзів	IDS	A*		F1
16	8-ферзів	IDS	A*		F2
17	8-ферзів	IDS	RBFS		F1
18	Лабіринт	LDFS	A*		H3
19	8-puzzle	LDFS	A*		H1
20	8-puzzle	LDFS	A*		H2
21	8-puzzle	LDFS	RBFS		H1
22	8-puzzle	LDFS	RBFS		H2
23	8-puzzle	BFS	A*		H1
24	8-puzzle	BFS	A*		H2
25	8-puzzle	BFS	RBFS		H1
26	8-puzzle	BFS	RBFS		H2
27	Лабіринт	BFS	A*		H3

28	8-puzzle	IDS	A*		H2
29	8-puzzle	IDS	RBFS		H1
30	8-puzzle	IDS	RBFS		H2
31	COLOR			HILL	MRV
32	COLOR			ANNEAL	MRV
33	COLOR			BEAM	MRV
34	COLOR			HILL	DGR
35	COLOR			ANNEAL	DGR
36	COLOR			BEAM	DGR

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

LDFS

State? LDFS(State root, int depth, ref int iterations, int depth_limit, ref int angles, ref int state_count)

```
iterations++
Якщо (root == finalState)
     То
          max_depth = depth;
           повернути root
Все Якщо

childs = root.GetChilds();

Якщо (depth >= depth_limit)
     То
          angles += childs.Count;
           повернути null;
Все Якщо

state_count += childs.Count;

Для кожного елемента child у списку childs
     stack.Push(child)
     result = DFS(child, depth + 1, ref iterations, depth_limit, ref
     angles, ref state_count);
     Якщо result != null
          То
               повернути result
     Все Якщо
     stack.Pop()
Все для кожного
повернути null
```

State.GetChilds

```
parent_matrix = this.Matrix
childs = new List<State>
cords = GetZeroCord()
i = cords.Item1
j = cords.Item2
Повторити для i від 0 до 4
    newI = i + offsetI[k]
    newJ = j + offsetJ[k]
    Якщо (newI >= 3 або newI < 0 або newJ >= 3 або  newJ < 0)
        То
            Продовжити
    Все Якщо
    child = new State(parent_matrix.Clone)
    tmp = child[newI, newJ]
    child[newI, newJ] = 0
    child[i,j] = tmp
    childs.Add(child)
Все повторити
Повернтуи childs
```

RBFS

(ExtendedState?, int) Search(ExtendedState node, int f_limit, ref int iterations, ref int angles, ref int state_count)

```
iterations++
Якщо (node == finalState)
   То
       Повернути (node, node.FunctionValue)
Все якщо
List successors = node.GetChildsEx()
state_count += successors.Count
Якщо (successors.Count==0)
   То
       Повернути (null, int.MaxValue)
Все якщо
Поки (successors.Count>=1)
   Відсортувати successors за FunctionValue
   bestNode = successors.First()
```

```
  Якщо (bestNode.FunctionValue>f_limit)
       То
            angles++
            Повернути (null, bestNode.FunctionValue)
  Все якщо
  alternative = successors[1].FunctionValue
  (result, bestNode.FunctionValue) = Search(bestNode, Math.Min(f_limit,
  alternative), ref iterations, ref angles, ref state_count)
  Якщо (result != null)
       То
            finalResult= result
            Зупинити
  Все якщо
Все поки
Повернути (finalResult, 0)
```

ExtendedState.GetChildsEx

```
  parent_matrix = this.Matrix
  childs = new List<State>
  cords = GetZeroCord()
  i = cords.Item1
  j = cords.Item2
  Повторити для i від 0 до 4
       newI = i + offsetI[k]
       newJ = j + offsetJ[k]
       Якщо (newI >= 3 або newI < 0 або newJ >= 3 або  newJ < 0)
            То
                 Продовжити
       Все Якщо
       childMatrix = parent_matrix.Clone()
       tmp = childMatrix[newI, newJ]
       childMatrix[newI, newJ] = 0
       childMatrix[i,j] = tmp
       child = new ExtendedState(childMatrix, 1, this)
       childs.Add(child)
  Все повторити
  Повернтуи childs
```

3.2 Програмна реалізація

3.2.1 Вихідний код

LDFS

```
internal class LDFS
   {
      static readonly State finalState = new State(new int[,] { {1,2,3},
                                                 {4,5,6},
                                                 {7,8,0}  });

      static bool solved = false;
      static int iterator = 0;
      private static Stack<State> stack = new Stack<State>();
      private static List<State> visited = new List<State>();
      private static int max_depth;


      public static void PrintPath(ref int state_in_mem)
      {
         int i = 1;
         int count = stack.Count;

         foreach (var item in stack.Reverse())
         {
            item.Print();
            Console.WriteLine();
            if (i!=count)
            {
               List<State> list = item.GetChilds();
               state_in_mem += list.Count;
            }
            i++;
         }
      }

      public static void Solve(State root, int depth_limit)
      {
         int iterations = 0;
         int angles = 0;
         int states = 1;
         stack.Push(root);
         State? result = DFS(root, 0, ref iterations, depth_limit, ref angles, ref states);
         if (result is null)
         {
            Console.WriteLine("result not found");
         }
         else
         {
            int state_in_mem = 1;
            PrintPath(ref state_in_mem);
            Console.WriteLine($"---------{max_depth} depth, {iterations} iterations, {angles} angles, {states} total states count, {state_in_mem} states in memory");
         }

      }

      public static State? DFS(State root, int depth, ref int iterations, int depth_limit, ref int angles, ref int state_count)
      {

         iterations++;
```

```
            // check if matrix is final matrix.
            if (root == finalState)
            {

                max_depth = depth;
                return root;
            }

            List<State> childs = root.GetChilds();

            if (depth >= depth_limit)
            {
                angles += childs.Count;
                return null;
            }

            state_count += childs.Count;

            foreach (var child in childs)
            {
                stack.Push(child);
                // recursive call to dfs.
                State? result = DFS(child, depth + 1, ref iterations, depth_limit, ref angles, ref state_count);

                if (result is not null)
                {
                    return result;
                }
                stack.Pop();
            }
            return null;

        }
                }
```

```
internal class State : ICloneable
    {
        public int[,] Matrix { get; set; }

        public State(int[,] matrix)
        {
            if (matrix.GetLength(0)!=3 || matrix.GetLength(1) != 3)
            {
                throw new ArgumentOutOfRangeException();
            }
            Matrix = matrix;
        }

        static protected readonly int[] offsetI = { -1, 0, 0, 1 };

        static protected readonly int[] offsetJ = { 0, -1, 1, 0 };

        public static bool operator ==(State? a, State? b)
        {
            if ((a is null && b is not null) || (a is not null && b is null))
            {
                return false;
            }
            if (a is null && b is null)
            {
```

```
            return true;
        }
        bool result = true;
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                if (a[i, j] != b[i, j])
                {
                    result = false;
                }
            }
        }
        return result;
    }

    public static bool operator !=(State? a, State? b)
    {
        if (a==b)
        {
            return false;
        }
        else
        {
            return true;
        }
    }

    public int this[int x, int y]
    {
        get
        {
            return Matrix[x, y];
        }
        protected set
        {
            Matrix[x, y] = value;
        }
    }


    public void Print()
    {
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                Console.Write(Matrix[i, j] + "  ");
            }
            Console.WriteLine();
        }
    }

    public  List<State> GetChilds()
    {
        int[,] parent_matrix = this.Matrix;
        List<State> childs = new List<State>();
        var cords = GetZeroCord();
        int i = cords.Item1;
        int j = cords.Item2;
        int newI;
        int newJ;
        int tmp;
        State child;
        for (int k = 0; k < 4; k++)
        {
```

```
            newI = i + offsetI[k];
            newJ = j + offsetJ[k];
            if (newI >= 3 || newI < 0 || newJ >= 3 || newJ < 0)
            {
                continue;
            }

            child = new State((int[,])parent_matrix.Clone());
            tmp = child[newI, newJ];
            child[newI, newJ] = 0;
            child[i,j] = tmp;
            childs.Add(child);
        }
        return childs;
    }


    public (int, int) GetZeroCord()
    {
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                if (Matrix[i,j]==0)
                {
                    return (i, j);
                }
            }
        }

        return (-1, -1);
    }

    public object Clone()
    {
        State result = new State((int[,])Matrix.Clone());
        return result;
    }
            }
```

RBFS

```
internal class RBFS
    {
        static readonly State finalState = new State(new int[,] { {1,2,3},
                                                  {4,5,6},
                                                  {7,8,0}  });


        private static ExtendedState? finalResult = null;

        public static void Start(int[,] matrix)
        {
            int iterations = 0;
            int states_count = 1;

            int angles = 0;
            (ExtendedState? node, int f ) = Search(new ExtendedState(matrix, 0, null), int.MaxValue, ref iterations, ref angles, ref states_count);
            if (node is null)
            {
```

```
        Console.WriteLine("not solved");
    }
    else
    {
        int state_in_memory = 1;
        PrintResult(FindSolution(node, ref state_in_memory), iterations, states_count, angles, state_in_memory);
    }
}


private static List<ExtendedState> FindSolution(ExtendedState node, ref int state_in_memory)
{
    List<ExtendedState> result= new List<ExtendedState>();
    result.Add(node);
    while (node.Parent != null)
    {
        node = node.Parent;
        List<ExtendedState> list = node.GetChildsEx();
        state_in_memory += list.Count;
        result.Add(node);
    }
    result.Reverse();
    return result;
}

private static void PrintResult(List<ExtendedState> result, int iterations, int states_count, int angles, int state_in_memory)
{
    foreach (var item in result)
    {
        item.Print();
        Console.WriteLine();
        Console.WriteLine();
    }

    Console.WriteLine($"---------{iterations} iterations, {angles} angles, {states_count} total states count, {state_in_memory} states in memory");
}

public static (ExtendedState?, int) Search(ExtendedState node, int f_limit, ref int iterations, ref int angles, ref int state_count)
{
    iterations++;

    if (node == finalState)
    {
        return (node, node.FunctionValue);
    }

    List<ExtendedState> successors = node.GetChildsEx();

    state_count += successors.Count;

    if (successors.Count==0)
    {
        return (null, int.MaxValue);
    }

    while (successors.Count>=1)
    {
        successors.Sort(delegate (ExtendedState c1, ExtendedState c2) { return c1.FunctionValue.CompareTo(c2.FunctionValue); });
        ExtendedState bestNode = successors.First();

        if (bestNode.FunctionValue>f_limit)
        {
            angles++;
            return (null, bestNode.FunctionValue);
        }
```

```
            int alternative = successors[1].FunctionValue;

            (ExtendedState? result, bestNode.FunctionValue) = Search(bestNode, Math.Min(f_limit, alternative), ref iterations, ref angles, ref state_count);

            if (result != null)
            {
                finalResult= result;
                break;
            }
        }

        return (finalResult, 0);

    }
            }




internal class ExtendedState : State
    {
        public ExtendedState(int[,] matrix, int path_cost, ExtendedState? parent) : base(matrix)
        {
            if (matrix.GetLength(0) != 3 || matrix.GetLength(1) != 3)
            {
                throw new ArgumentOutOfRangeException();
            }
            Matrix = matrix;
            Parent = parent;
            PathCost = GetPathCost(parent, path_cost);
            FunctionValue = GetFunctionValue(matrix, PathCost);
        }

        public int FunctionValue { get; set; }
        public ExtendedState? Parent { get; set; }
        public int PathCost { get; set; }


        private static int GetPathCost(ExtendedState? parent, int path_cost)
        {
            if (parent == null)
            {
                return path_cost;
            }
            else
            {
                return parent.PathCost + path_cost;
            }
        }

        private static int GetFunctionValue(int[,] matrix, int ready_path_cost)
        {
            int value = 0;
            int k = 1;

            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    if (matrix[i, j] != k && matrix[i, j] != 0)
                    {
                        value++;
                    }
                    k++;
                }
```

```
    }
    return value + ready_path_cost;
}

public  List<ExtendedState> GetChildsEx()
{

    int[,] parent_matrix = this.Matrix;
    List<ExtendedState> childs = new List<ExtendedState>();
    var cords = GetZeroCord();
    int i = cords.Item1;
    int j = cords.Item2;
    int newI;
    int newJ;
    int tmp;
    ExtendedState child;
    for (int k = 0; k < 4; k++)
    {
        newI = i + offsetI[k];
        newJ = j + offsetJ[k];
        if (newI >= 3 || newI < 0 || newJ >= 3 || newJ < 0)
        {
            continue;
        }

        int[,] childMatrix = (int[,])parent_matrix.Clone();
        tmp = childMatrix[newI, newJ];
        childMatrix[newI, newJ] = 0;
        childMatrix[i, j] = tmp;
        child = new ExtendedState(childMatrix, 1, this);
        childs.Add(child);
    }
    return childs;
}
          }
```

3.2.2 Приклади роботи

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

Рисунок 3.1 – Алгоритм LDFS

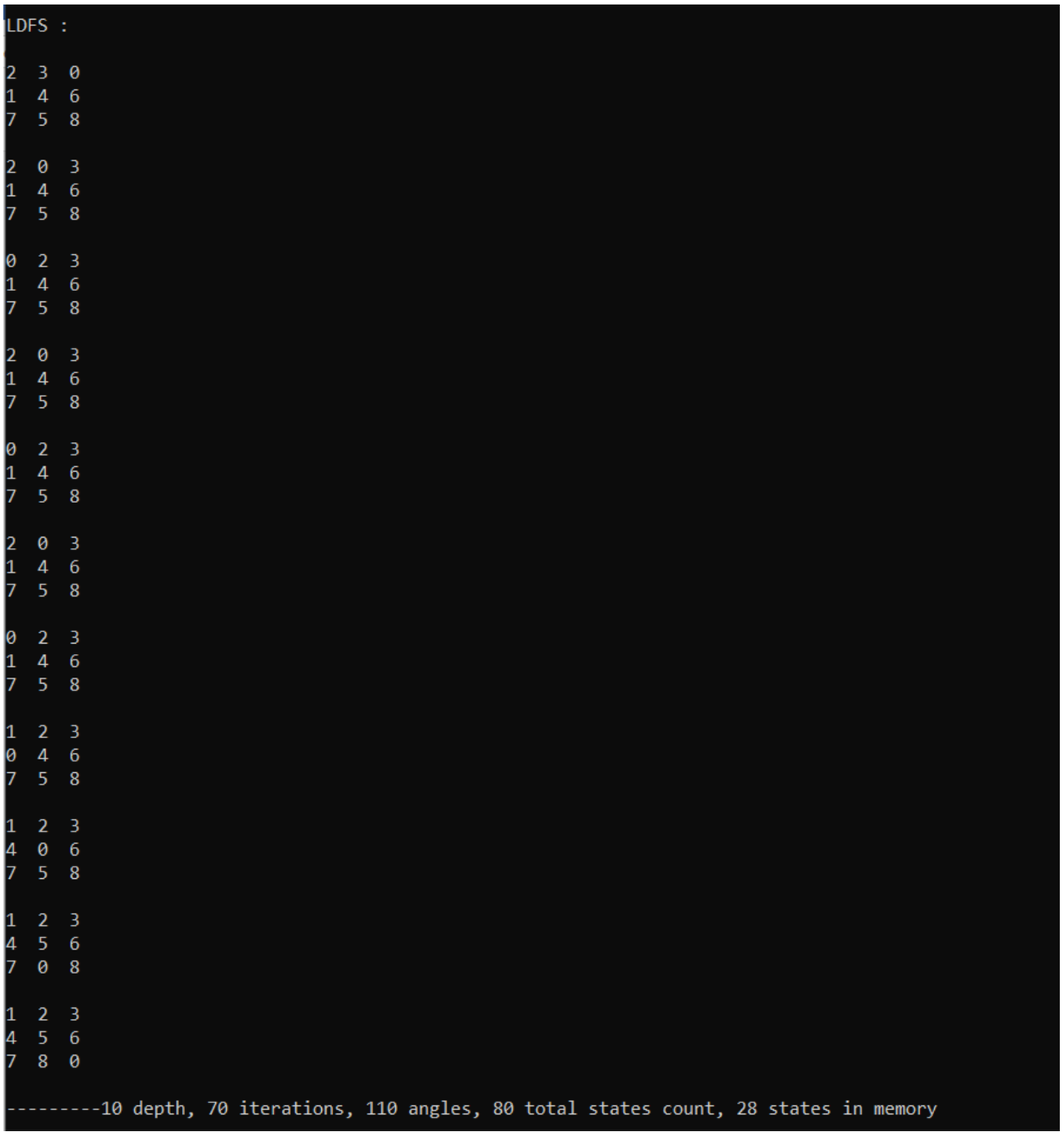
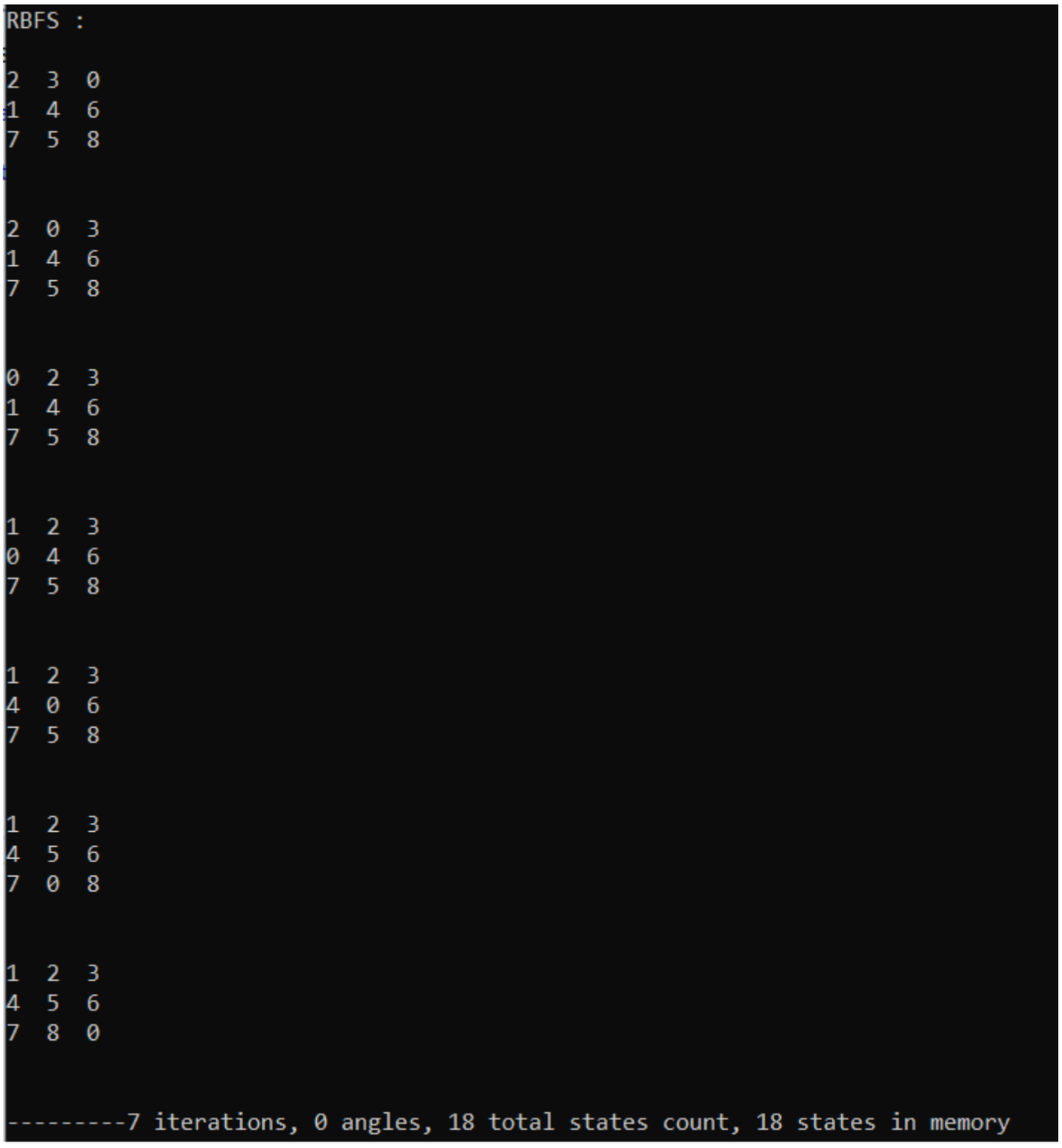


Рисунок 3.2 – Алгоритм RBFS



3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму LDFS, задачі 8 puzzle для 20 початкових станів з обмеженням глибини L = 15.

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS

Початкові стани	Ітерації	Кількість глухих кутів	Всього станів	Всього станів у пам’яті
1 3 0 8 2 7 4 6 5	355561	226262	355573	42
0 5 3 2 1 7 4 8 6	663172	422014	663184	40
2 3 0 1 5 7 4 8 6	83076	52862	83088	40
4 1 3 7 0 8 5 6 2	3228043	2054206	3228053	44
2 3 6 4 1 5 0 7 8	77446	49278	77459	38
4 1 3 7 0 2 8 6 5	357322	227382	357335	42
2 3 6 1 5 8 0 4 7	811213	516222	811222	36
4 1 3 2 8 5 0 7 6	97447	62006	97461	40
0 5 3 4 1 6	386802	246142	386813	40

7 2 8				
1 3 0 8 2 5 4 7 6	4771	3030	4786	40
2 4 0 5 3 1 7 8 6	4553164	2897462	4553178	42
1 6 2 4 5 3 7 8 0	199916	127214	199928	40
1 3 0 4 6 5 7 2 8	191581	121910	191592	40
1 5 2 8 7 3 0 4 6	87151	55454	87165	40
3 5 0 1 4 8 7 6 2	664668	422966	664680	42
1 6 0 7 3 2 5 4 8	315069	200494	315080	42
1 3 5 7 0 2 8 4 6	190136	120990	190151	42
1 8 2 0 4 3 7 6 5	24562	16112	24578	43
1 3 7	1363954	867966	1363967	42

5 0 2 4 8 6				
0 2 3 1 5 6 4 7 8	209	126	224	36

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

Таблиця 3.2 – Характеристики оцінювання RBFS

Початкові стани	Ітерації	Кількість глухих кутів	Всього станів	Всього станів у пам’яті
1 3 0 8 2 7 4 6 5	1475	1462	4316	37
0 5 3 2 1 7 4 8 6	1067	1054	2796	35
2 3 0 1 5 7 4 8 6	3343	3330	8584	35
4 1 3 7 0 8 5 6 2	1318	1305	3814	37
2 3 6 4 1 5 0 7 8	52	41	134	28
4 1 3 7 0 2 8 6 5	89	78	252	30

2 3 6 1 5 8 0 4 7	26	15	66	26
4 1 3 2 8 5 0 7 6	12	3	33	25
0 5 3 4 1 6 7 2 8	4036	4023	11441	35
1 3 0 8 2 5 4 7 6	12	3	33	25
2 4 0 5 3 1 7 8 6	13893	13878	38250	42
1 6 2 4 5 3 7 8 0	53	44	134	25
1 3 0 4 6 5 7 2 8	967	956	2814	30
1 5 2 8 7 3 0 4 6	38	27	102	30
3 5 0 1 4 8 7 6 2	5171	5156	15174	42
1 6 0 7 3 2	214	201	613	37

5 4 8				
1 3 5 7 0 2 8 4 6	82	71	233	32
1 8 2 0 4 3 7 6 5	89	79	260	28
1 3 7 5 0 2 4 8 6	2495	2482	6719	37
0 2 3 1 5 6 4 7 8	5	0	11	11

ВИСНОВОК

При виконанні даної лабораторної роботи я розглянув 2 алгоритми для вирішення задачі 8 puzzle, а саме : алгоритм LDFS неінформативного пошуку та алгоритм RBFS інформативного пошуку з використанням евристичної функції H1 - кількість фішок, які не стоять на своїх місцях.

Я виконав програмну реалізацію цих алгоритмів на мові програмування C#.

Потім я протестував алгоритми на 20 різних початкових станах та записав аналітичні дані у таблицю. Проаналізувавши результати досліджень можна зробити висновок :

Алгоритм LDFS є неповним (при L<d) та неоптимальним (при L>d). Крім того, час його виконання у більшості випадків є набагато більшим ніж час виконання алгоритму RBFS, адже це алгоритм неінформативного пошуку.

Алгоритм RBFS в свою є оптимальним та повним і займає лінійний простір.

Тож, можна з впевненістю стверджувати, що алгоритм RBFS є кращим у всіх його аспектах. Також треба зазначити, що усі алгоритми інформативного пошуку будуть мати перевагу над алгоритмами неінформативного пошуку.

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%.