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 Дейнега Владислав Миколайович* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов О.О.* _____
(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

```
ПОЧАТОК IDS
    max_depth = 0
    ПОКИ ПРАВДА
        Result = _DLS(max_depth)
        ЯКЩО result == true ТО повернути result
        ІНАКШЕ ЯКЩО result == filure ТО повернути result
        Max_depth++
    ПОВТОРИТИ
КІНЕЦЬ
ПОЧАТОК DLS
    Повернути RECURSIVE_DLS (max_depth, 0, _start)
КІНЕЦЬ
ПОЧАТОК RECURSIVE_DLS
    cutoff_occurred = false
    ЯКЩО depth >= max_depth ТО
        res.succes = false
        res.cutOff = true;
        res.failure = false;
        повернути  res;
    children = _find_children(cur)
    ЯКЩО children.Count == 0 ТО
        res.succes = false
        повернути  res
    ДЛЯ КОЖНОГО childe У children
        ЯКЩО childe == finish ТО
            res.succes = true
            повернути  res
        ІНАКШЕ
```

```
                    Res = RECURSIVE_DLS (max_depth, depth+1, childe)
                    ЯКЩО Res.succes == true ТО
                        повернути  res
                    ІНАШЕ ЯКЩО res.cutOff == true ТО
                        cutoff_occurred = true
        ПОВТОРИТИ
        ЯКЩО cutoff_occurred == true ТО
            res.succes = false
            повернути  res
        ІНАКШЕ
            res.failure = true
            повернути  res
КІНЕЦЬ



ПОЧАТОК RBFS
        res = _recursive_RDFS(_start, fLimit, 0)
        Повернути res
КІНЕЦЬ
ПОЧАТОК recursive_RDFS
        children = _find_children(cur)
        ЯКЩО childe == finish ТО
            res.succes = true
            повернути  res
        ЯКЩО children.Count == 0 ТО
            res.succes = false
            повернути  res
        ЯКЩО children[0].f == double.MaxValue ТО
            ДЛЯ КОЖНОГО child З children
```

```
            child.f = _funk(child, g + 1)
            ПОВТОРИТИ
        ПОКИ ПРАДА
            int best = 0;
             int alt = 0;
            find_best(children, ref best)
            ЯКЩО fLimit < children[best].f ТО
                result.fLimit = children[best].f
                повернути result
            ЯКЩО children.Count != 1 ТО
                find_alternative(children, ref best, ref alt)
                fLimit = Math.Min(fLimit, children[alt].f)
            result = _recursive_RDFS(children[best], fLimit, g+1)
            children[best].f = result.fLimit
            ЯКЩО result.succes != false ТО
                повернути result
        ПОВТОРИТИ
    КІНЕЦЬ
```

3.2 Програмна реалізація

3.2.1 Вихідний код

Maze.css

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



namespace Lab
{
```

```
class Maze
{
    private long _byteInGig = 1024 * 1024 * 1024;
    private long _msIn30Min = 1800000;
    private long _used_mem = sizeof(int) * 2 + sizeof(double) + sizeof(bool) * 6;
    private class Node
    {
        public bool wached = false;
        public bool isPath = false;

        public bool top = true;
        public bool right = true;
        public bool bottom = true;
        public bool left = true;

        public int[] corde;

        public double f = double.MaxValue;
    }

    private Node _start = new Node();
    private Node _finish = new Node();
    private Node[,] _maze;
    private int _size;

    private Stopwatch time;

    public int[] GetFinishCorde()
    {
        return _finish.corde;
    }

    public int[] GetStartCorde()
    {
        return _start.corde;
    }

    //-----------Maze-----------//
    private List<Node> _get_neighbours(int y_cur, int x_cur)
    {
        List<Node> neighbours = new List<Node>();
        int[] cord_t = new int[2] { y_cur - 1, x_cur };
        int[] cord_l = new int[2] { y_cur, x_cur - 1 };
        int[] cord_b = new int[2] { y_cur + 1, x_cur };
        int[] cord_r = new int[2] { y_cur, x_cur + 1 };
```

```
            List<int[]> temp = new List<int[]>() { cord_t, cord_l, cord_b, cord_r };

            for (int i = 0; i < 4; i++)
            {
                if (temp[i][0] >= 0 && temp[i][0] < _size && temp[i][1] >= 0 && temp[i][1]
< _size)
                {
                    if (_maze[temp[i][0], temp[i][1]].wached == false)
                    {
                        neighbours.Add(_maze[temp[i][0], temp[i][1]]);
                    }
                }
            }
            return neighbours;
        }

        private void _dell_wall(Node fir, Node sec)
        {
            if (fir.corde[0] < sec.corde[0])
            {
                _maze[fir.corde[0], fir.corde[1]].bottom = false;
                _maze[sec.corde[0], sec.corde[1]].top = false;
                return;
            }
            if (fir.corde[0] > sec.corde[0])
            {
                _maze[fir.corde[0], fir.corde[1]].top = false;
                _maze[sec.corde[0], sec.corde[1]].bottom = false;
                return;
            }
            if (fir.corde[1] < sec.corde[1])
            {
                _maze[fir.corde[0], fir.corde[1]].right = false;
                _maze[sec.corde[0], sec.corde[1]].left = false;
                return;
            }
            if (fir.corde[1] > sec.corde[1])
            {
                _maze[fir.corde[0], fir.corde[1]].left = false;
                _maze[sec.corde[0], sec.corde[1]].right = false;
                return;
            }
        }

        private void _make_false()
        {
```

```
    for (int i = 0; i < _size; i++)
    {
        for (int j = 0; j < _size; j++)
        {
            _maze[i, j].wached = false;
        }
    }
}

public Maze(int size)
{
    this._size = size;

    _maze = new Node[this._size, this._size];

    _create_empty_maze();

    time = new Stopwatch();
    time.Start();
}

private void _create_empty_maze()
{
    Random rand = new Random();
    //bool end;
    //if (rand.Next(21) <= 5)
    //{
    //    end = false;
    //    _finish.corde = new int[2] { int.MaxValue, int.MaxValue };
    //}
    //else
    //{
    //    end = true;
    //}

    if (rand.Next(2) == 0)
    {
        _start.corde = new int[2] { rand.Next(_size), 0 };
        //if(end == true)
        _finish.corde = new int[2] { rand.Next(_size), _size - 1 };
    }
    else
    {
        _start.corde = new int[2] { 0,rand.Next(_size) };
        //if (end == true)
        _finish.corde = new int[2] { _size - 1, rand.Next(_size) };
```

```
                    }

                    for (int i = 0; i < this._size; i++)
                    {
                        for (int j = 0; j < this._size; j++)
                        {
                            if (i == _start.corde[0] && j == _start.corde[1])
                            {
                                _maze[i, j] = _start;
                            }
                            else if (i == _finish.corde[0] && j == _finish.corde[1])
                            {
                                _maze[i, j] = _finish;
                            }
                            else
                            {
                                _maze[i, j] = new Node();
                                _maze[i, j].corde = new int[2] { i, j };
                            }
                        }
                    }
                }

                public void generate_maze()
                {
                    Stack<Node> node_stack = new Stack<Node>();
                    Node current = _start;
                    _start.wached = true;
                    Node neighbour_node;
                    int unwached = _size * _size-1;
                    Random rand = new Random();
                    while (unwached != 0)
                    {

                        List<Node>      neighbours      =      _get_neighbours(current.corde[0],
current.corde[1]);
                        if (neighbours.Count != 0)
                        {
                            if  (current.corde[0]  ==  _finish.corde[0]  &&  current.corde[1]  ==
_finish.corde[1])
                            {
                                _maze[current.corde[0], current.corde[1]].wached = true;
                                current = node_stack.Pop();
                            }
                            else
                            {
```

```
                            int index = rand.Next(neighbours.Count);
                            neighbour_node = neighbours[index];
                            _maze[neighbour_node.corde[0],  neighbour_node.corde[1]].wached  =
true;
                            unwached -= 1;
                            _dell_wall(current, neighbour_node);

                            node_stack.Push(current);
                            current = neighbour_node;
                        }
                    }
                    else if (node_stack.Count > 0)
                    {
                        current = node_stack.Pop();
                    }
                }

                _make_false();
            }

            public void draw_maze()
            {
                Console.ForegroundColor = ConsoleColor.Red;
                for (int i = 0; i < _size + _size + 1; i++)
                {
                    Console.Write("--");
                }
                Console.ResetColor();
                Console.WriteLine();

                for (int i = 0; i < _size; i++)
                {
                    // side wall
                    Console.ForegroundColor = ConsoleColor.Red;
                    Console.Write('|');
                    Console.ResetColor();
                    for (int j = 0; j < _size; j++)
                    {
                        if (_start.corde[0] == i && _start.corde[1] == j)
                        {
                            Console.ForegroundColor = ConsoleColor.Green;
                            Console.Write(" S ");
                            Console.ResetColor();
                        }
                        else if (_finish.corde[0] == i && _finish.corde[1] == j)
                        {
```

```
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write(" E ");
            Console.ResetColor();
        }
        else if (_maze[i, j].isPath == true)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write(" P ");
            Console.ResetColor();
        }
        else
            Console.Write(" * ");
        if (_maze[i, j].right == true)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write("|");
            Console.ResetColor();
        }
        else
        {
            Console.Write("*");
        }
    }
    Console.WriteLine();
    // bottom wall
    if (i != _size - 1)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.Write('|');
        Console.ResetColor();
        for (int j = 0; j < _size; j++)
        {
            if (_maze[i, j].bottom == true)
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.Write("---");
                Console.ResetColor();
            }
            else
            {
                Console.Write(" * ");
            }
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write("|");
            Console.ResetColor();
        }
```

```
                Console.WriteLine();
            }
        }

        for (int i = 0; i < _size + _size + 1; i++)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write("--");
            Console.ResetColor();
        }

        Console.WriteLine();
    }

    //-----------DLS-----------//
    private Result _DLS(int max_depth, Result res)
    {
        return _recursive_DLS(max_depth, 0, _start, res);
    }

    private Result _recursive_DLS(int max_depth, int depth, Node cur, Result res)
    {
        if (res.cStateInMemory * _used_mem > _byteInGig || time.ElapsedMilliseconds >
_msIn30Min)
        {
            res.failure = true;
            res.cutOff = false;

            return res;
        }

        bool cutoff_occurred = false;
        if (depth >= max_depth)
        {
            res.succes = false;
            res.cutOff = true;
            res.failure = false;
            res.cStateInMemory--;
            //res.corner++;
            return res;
        }
        List<Node> children = _find_children(cur);
        if (children.Count == 0)
        {
            res.succes = false;
            res.cStateInMemory--;
```

```
                    return res;
                }

                res.cState += children.Count;
                res.cStateInMemory += children.Count;

                foreach (Node childe in children)
                {
                    if   (childe.corde[0]   ==   _finish.corde[0]   &&   childe.corde[1]   ==
_finish.corde[1])
                    {
                        cur.isPath = true;
                        res.succes = true;
                        //res.cStateInMemory++;
                        return res;
                    }
                    else
                    {
                        //depth++;
                        res.iterations++;
                        cur.isPath = true;
                        res = _recursive_DLS(max_depth, depth+1, childe, res);
                        if (res.succes)
                        {
                            return res;
                        }
                        else if (res.cutOff == true)
                        {
                            cutoff_occurred = true;
                        }

                       // cur.isPath = false;
                    }
                }
                res.cStateInMemory--;
                //depth--;
                cur.isPath = false;
                if (cutoff_occurred == true)
                {
                    res.succes = false;
                    return res;
                }
                else
                {
                    res.failure = true;
                    res.succes = false;
```

```
                return res;
            }
        }

        public Result IDS()
        {
            int max_depth = 0;
            Result result = new Result();
            while (true)
            {
                result = _DLS(max_depth, result);
                if (result.succes != false)
                {
                    time.Stop();
                    result.time = time.ElapsedMilliseconds;
                    return result;
                }
                else if(result.failure == true)
                {
                    time.Stop();
                    result.time = time.ElapsedMilliseconds;
                    result.corner++;
                    return result;
                }
                result.corner++;
                result.cutOff = false;
                result.failure = false;
                result.cStateInMemory = 0;
                max_depth++;
            }
        }

        //-----------RBFS-----------//
        private double _funk(Node cur, int g)
        {
            return _h(cur) + g;
        }

        private double _h(Node node)
        {
            if (node.corde[0] == _finish.corde[0] && node.corde[1] == _finish.corde[1])
            {
                return 0;
            }
            return  Math.Sqrt(Math.Pow(2,_finish.corde[0]  -  node.corde[0])  +  Math.Pow(2,
_finish.corde[1] - node.corde[1]));
```

```
        }

        private void _find_best(List<Node> children, ref int best)
        {
            for (int i = 0; i < children.Count; i++)
            {
                if (children[best].f > children[i].f)
                {
                    best = i;
                }
            }
        }

        private void _find_alternative(List<Node> children, ref int best, ref int alt)
        {
            for (int i = 0; i < children.Count; i++)
            {
                if ((children[alt].f > children[i].f && i != best) || children[alt].f ==
children[best].f)
                {
                    alt = i;
                }
            }
        }

        private List<Node> _find_children(Node node)
        {
            List<Node> children = new List<Node>();
            if (node.top == false && _maze[node.corde[0] - 1, node.corde[1]].isPath != true)
            {
                children.Add(_maze[node.corde[0] - 1, node.corde[1]]);
            }
            if (node.left == false && _maze[node.corde[0], node.corde[1] - 1].isPath !=
true)
            {
                children.Add(_maze[node.corde[0], node.corde[1] - 1]);
            }
            if (node.bottom == false && _maze[node.corde[0] + 1, node.corde[1]].isPath !=
true)
            {
                children.Add(_maze[node.corde[0] + 1, node.corde[1]]);
            }
            if (node.right == false && _maze[node.corde[0], node.corde[1] + 1].isPath !=
true)
            {
                children.Add(_maze[node.corde[0], node.corde[1] + 1]);
```

```
    }
    return children;
}

private Result _recursive_RDFS(Node cur,  double fLimit, int g, Result result)
{

    result.cStateInMemory++;
    cur.isPath = true;
    List<Node> children = _find_children(cur);

    if (cur.corde[0] == _finish.corde[0] && cur.corde[1] == _finish.corde[1])
    {
        result.succes = true;
        //result.cStateInMemory++;
        return result;
    }
    if (children.Count == 0)
    {
        cur.isPath = false;
        result.cStateInMemory--;
        result.fLimit = int.MaxValue;
        return result;
    }

    if (children[0].f == double.MaxValue)
    {
        foreach (Node child in children)
        {
            child.f = _funk(child, g + 1);
        }
    }

    result.cState+= children.Count;

    while (true)
    {
        result.iterations++;

        int best = 0;
        int alt = 0;
        _find_best(children, ref best);

        if (fLimit < children[best].f)
        {
            cur.isPath = false;
```

```
                    result.fLimit = children[best].f;
                    result.cStateInMemory--;
                    return result;
                }

                if (children.Count != 1)
                {
                    _find_alternative(children, ref best, ref alt);
                    fLimit = Math.Min(fLimit, children[alt].f);
                }

                result = _recursive_RDFS(children[best], fLimit, g+1, result);
                children[best].f = result.fLimit;

                if (result.succes != false)
                {
                    //result.cStateInMemory++;
                    return result;
                }
            }
        }

        public Result RDFS()
        {
            double fLimit = int.MaxValue;
            _start.f = 0;
            Result res = new Result();
            res = _recursive_RDFS(_start, fLimit, 0, res);

            if (res.succes == false)
            {
                res.corner++;
            }

            time.Stop();
            res.time = time.ElapsedMilliseconds;
            return res;
        }
    }
}
```

Program.css

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

```
using System.Threading.Tasks;

namespace Lab
{
    class Program
    {
        static void Main(string[] args)
        {
            Result res;
            string answ;
            int num, count, size;

            Console.WriteLine("Which algoritm you want? ids/rbfs");
            answ = Console.ReadLine();

            Console.WriteLine("How much maze you want generate?");
            count = Convert.ToInt32(Console.ReadLine());

            Console.WriteLine("Maze size?");
            size = Convert.ToInt32(Console.ReadLine());
            switch (answ)
            {
                case "ids":
                    for (int i = 0; i < count; i++)
                    {
                        num = i + 1;
                        Console.WriteLine();
                        Console.WriteLine("Maze №" + num);

                        Maze maze = new Maze(size);
                        maze.generate_maze();

                        res = maze.IDS();
                        maze.draw_maze();

                        Console.WriteLine("State - " + maze.GetStartCorde()[0] + ", " +
maze.GetStartCorde()[1] + "; " + maze.GetFinishCorde()[0] + ", " +
maze.GetFinishCorde()[1]);
                        Console.WriteLine("Iterations - " + res.iterations);
                        Console.WriteLine("Dead end - " + res.corner);
                        Console.WriteLine("States count - " + res.cState);
                        Console.WriteLine("States in memory - " + res.cStateInMemory);
                        Console.WriteLine("Time - " + res.time);
                    }
                    break;
                case "rbfs":
                    for (int i = 0; i < count; i++)
                    {
                        num = i + 1;
                        Console.WriteLine();
                        Console.WriteLine("Maze №" + num);

                        Maze maze = new Maze(size);
                        maze.generate_maze();

                        res = maze.RDFS();
                        maze.draw_maze();

                        Console.WriteLine("State - " + maze.GetStartCorde()[0] + ", " +
maze.GetStartCorde()[1] + "; " + maze.GetFinishCorde()[0] + ", " +
maze.GetFinishCorde()[1]);
                        Console.WriteLine("Iterations - " + res.iterations);
                        Console.WriteLine("Dead end - " + res.corner);
                        Console.WriteLine("States count - " + res.cState);
                        Console.WriteLine("States in memory - " + res.cStateInMemory);
```

```
                    Console.WriteLine("Time - " + res.time);
                }
                break;
            default:
                Console.WriteLine("Error");
                break;
        }

        Console.ReadLine();
    }
}
   }
```

Result.css

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

namespace Lab
{
    class Result
    {
        public bool succes = false;
        public double fLimit;
        public bool cutOff = false;
        public bool failure = false;

        public int corner = 0;
        public int iterations = 0;
        public int cState = 0;
        public int cStateInMemory = 0;

        public float time;
    }
   }
```

3.2.2 Приклади роботи

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

Рисунок 3.1 – Алгоритм IDS

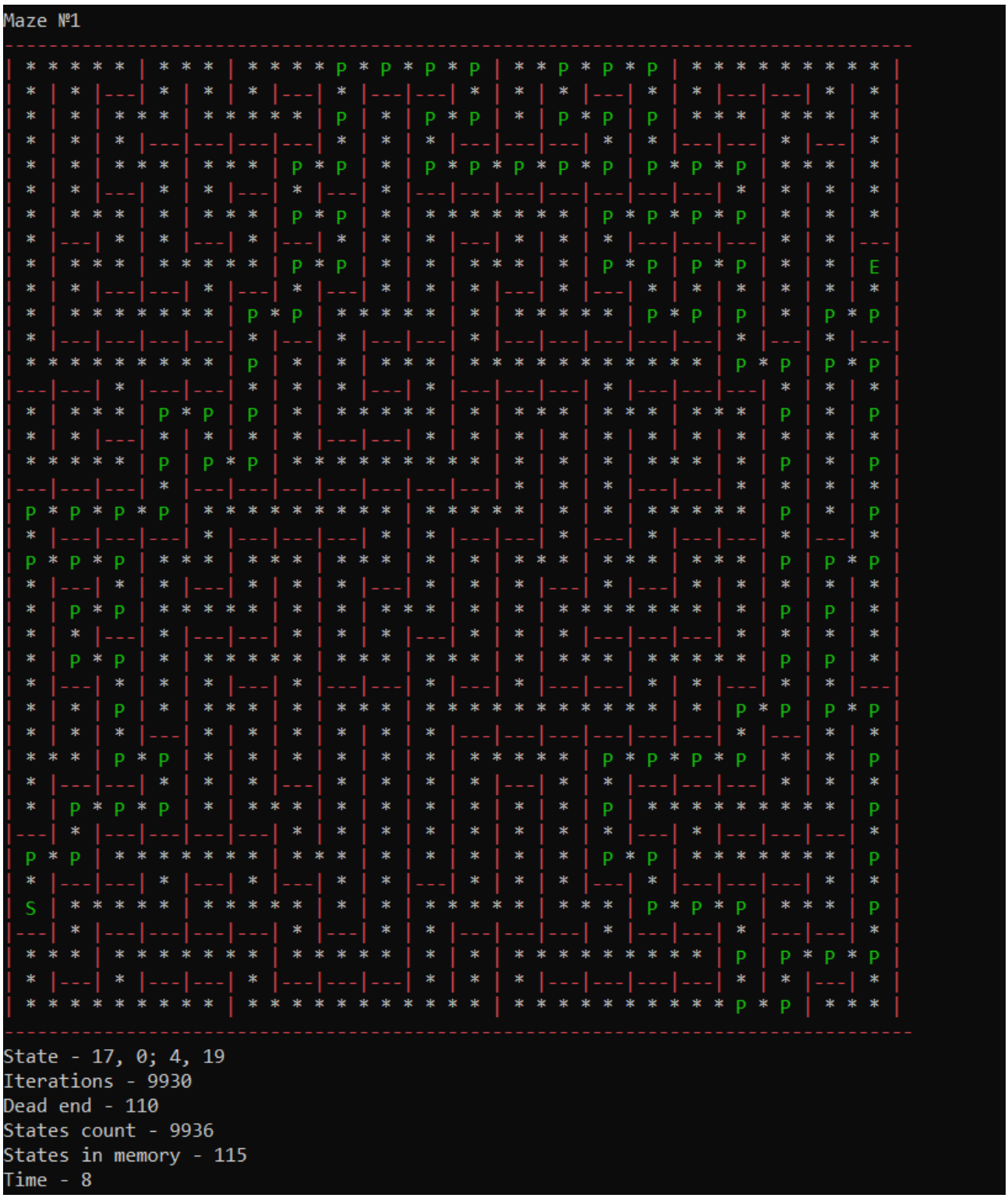
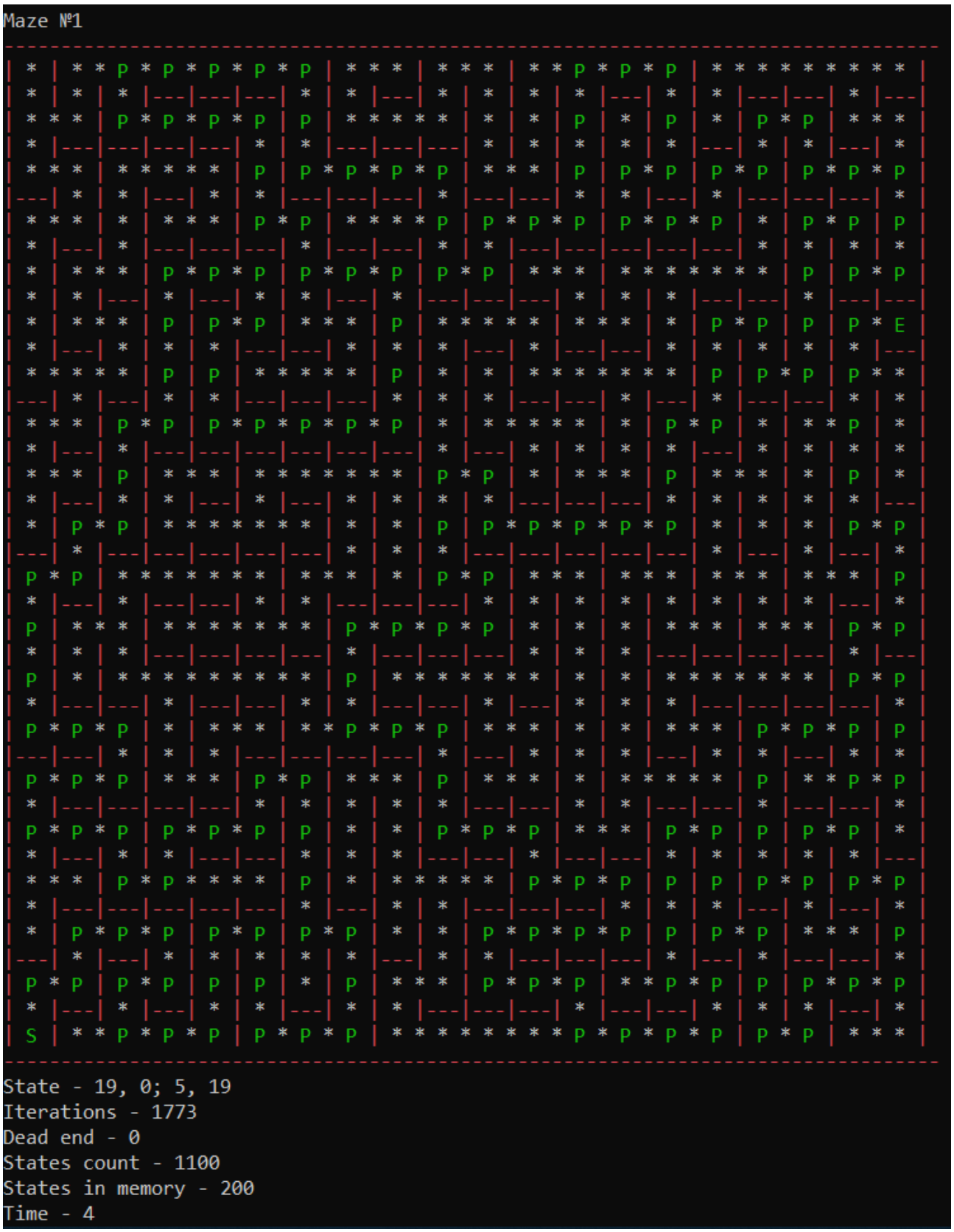


Рисунок 3.2 – Алгоритм RBFS



3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму IDS, задачі з лабіринтом для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритм IDS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
0, 17; 19, 3	10212	123	10217	127
0, 8; 19, 1	39033	226	39043	235
15, 0; 16, 19	12417	144	12422	148
0, 0; 19, 18	8255	115	8258	117
6, 0; 19, 19	19517	172	19525	179
5, 0; 7, 19	22023	161	22030	167
0, 4; 19, 12	5747	95	5748	95
0, 15; 19, 10	6631	110	6633	111
0, 0; 11, 19	29199	190	29206	196
0, 14; 19, 3	14336	142	14343	148
11, 0; 3, 19	55527	271	55541	284
0, 10; 19, 8	10615	125	10622	131
0, 10; 19, 16	22640	165	22645	169
0, 1; 19, 14	8481	106	8484	108
18, 0; 6, 19	22047	173	22053	178
0, 11; 19, 17	9877	127	9881	130
16, 0; 18, 19	41570	241	41580	250
0, 4; 19, 13	5458	98	5459	98
0, 15; 19, 11	26671	181	26678	187
0, 0; 13, 19	19641	160	19644	162

В таблиці 3.2 наведені характеристики оцінювання алгоритму RBFS, задачі з лабіринтом для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання RBFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пом’яті
8, 0; 4, 19	3624	0	2142	256
0, 13; 19, 5	2558	0	1532	210
0, 7; 19, 16	459	0	284	67
12, 0; 18, 19	70	0	69	58
3, 0; 16, 19	279	0	234	139
14, 0; 14, 19	123	0	110	78
0, 6; 19, 12	1002	0	641	174
0, 11; 19, 13	281	0	219	110
16, 0; 15, 19	129	0	124	105
7, 0; 13, 19	78	0	80	72
0, 12; 19, 14	918	0	599	174
0, 12; 19, 2	558	0	387	132
0, 3; 19, 0	54	0	56	53
8, 0; 1, 19	148	0	129	75
19, 0; 19, 19	85	0	83	68
11, 0; 17, 19	1109	0	709	182
0, 16; 19, 19	120	0	114	87
1, 0; 0, 19	426	0	290	89
12, 0; 18, 19	314	0	251	140
0, 17; 19, 0	1287	0	799	161

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми неінформативного IDS пошуку та інформативного пошуку RBFS. У результаті тестів вийшло

IDS

Середнє значення кількості ітерацій - 19494.85

Середнє значення кількості глухіх кутів - 151.25

Середнє значення всіх станів - 19500.6

Середнє значення станів у пам’яті - 160

RBFS

Середнє значення кількості ітерацій - 681

Середнє значення кількості глухіх кутів - 0

Середнє значення всіх станів - 442

Середнє значення станів у пам’яті - 121

У підсумку можна сказати, що RBFS набагато швидше IDS, через підхід до пошуку шляху. RBFS, на відміну від IDS, не обходить дерево у глубину а намагається знайти кращий шлях з допомогою евристичної функції. На невеликих обсягах даних різниця у роботі алгоритмів не помітна, але прибільщих обсягах IDS буде працювати набагато повыльныше.

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%.