

**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-12 Бондарчук Анастасія  
(шифр, прізвище, ім'я, по батькові)

**Перевірив**

Головченко М. М.  
(прізвище, ім'я, по батькові)

Київ 2022

## ЗМІСТ

<b>1</b>	<b>МЕТА ЛАБОРАТОРНОЇ РОБОТИ .....</b>	<b>3</b>
<b>2</b>	<b>ЗАВДАННЯ .....</b>	<b>4</b>
<b>3</b>	<b>ВИКОНАННЯ.....</b>	<b>8</b>
3.1	ПСЕВДОКОД АЛГОРИТМІВ.....	8
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ.....	9
3.2.1	<i>Вихідний код.....</i>	<i>9</i>
3.2.2	<i>Приклади роботи .....</i>	<i>18</i>
3.3	ДОСЛІДЖЕННЯ АЛГОРИТМІВ.....	19
	<b>ВИСНОВОК .....</b>	<b>37</b>
	<b>КРИТЕРІЇ ОЦІНЮВАННЯ .....</b>	<b>38</b>

## 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 **АПІ**, 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(Cell source, Cell destination, Maze maze)returns List<(int, int)>

    int[] rowsMove = { -1, 0, 0, 1 };
    int[] colsMove = { 0, -1, 1, 0 };
    if (maze.Matrix[source.X,source.Y] != 1 || maze.Matrix[destination.X,
destination.Y] != 1):
        return new List<(int, int)>();
    end if;
    bool[,] visitedCells = new bool[maze.Matrix.GetLength(0),
maze.Matrix.GetLength(1)];
    visitedCells[source.X, source.Y] = true;
    List<Cell> Queue = new List<Cell> {source};
    while (Queue.Count != 0):
        Cell current = Queue[0];
        if (current.X == destination.X && current.Y == destination.Y)
            return destination.Path = current.Path;
        end if;
        Queue.RemoveAt(0);
        for (int i = 0; i < 4; i++)
            int row = current.X + rowsMove[i];
            int col = current.Y + colsMove[i];
            if (maze.IsValid(row, col) && maze.Matrix[row, col] == 1 &&
!visitedCells[row, col]):
                visitedCells[row, col] = true;
                Cell NeighborCell = new Cell(row, col)
                {Path = new List<(int, int)>(current.Path)};
                NeighborCell.Path.Add((NeighborCell.X, NeighborCell.Y));
                Queue.Add(NeighborCell);
            end if;
        end for;
    end while;
    Queue.Clear();
    return new List<(int, int)>();
end.
```



```

AStar(Cell source, Cell destination, Maze maze) returns List<(int, int)>
    int[] rowsMove = { -1, 0, 0, 1 };
    int[] colsMove = { 0, -1, 1, 0 };
    if (maze.Matrix[source.X,source.Y] != 1 || maze.Matrix [destination.X,
destination.Y] != 1):
        return new List<(int, int)>();
    end if;
    bool[,] visitedCells = new bool[maze.Matrix.GetLength(0),
maze.Matrix.GetLength(1)];
    visitedCells[source.X, source.Y] = true;
    List<Cell> PriorityQueue = new List<Cell>();
    PriorityQueue.Add(source);
    while (PriorityQueue.Count != 0):
        Cell current = PriorityQueue[0];
        maze.ManhattanDistance(current);
        if (current.X == destination.X && current.Y == destination.Y):
            return destination.Path = current.Path;
        end if;
        PriorityQueue.RemoveAt(0);
        for (int i = 0; i < 4; i++):
            int row = current.X + rowsMove[i];
            int col = current.Y + colsMove[i];
            if (maze.IsValid(row, col) && maze.Matrix[row, col] == 1 &&
!visitedCells[row, col])
                visitedCells[row, col] = true;
                Cell NeighborCell = new Cell(row, col)
                {Path = new List<(int, int)>(current.Path)};
                NeighborCell.Path.Add((NeighborCell.X, NeighborCell.Y));
                maze.ManhattanDistance(NeighborCell);
                AddToQueue(PriorityQueue, NeighborCell);
            end if;
        end for;
    end while;
    PriorityQueue.Clear();
    return new List<(int, int)>();
end.

```

## 3.2 Програмна реалізація

### 3.2.1 Вихідний код

```
using System;
```

```

using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;

namespace lab2
{
    internal class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Number of rows: ");
            int Row = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("\nNumber of cols: ");
            int Col = Convert.ToInt32(Console.ReadLine());
            Random rand = new Random();
            int[,] matrix = new int[Row, Col];
            for (int i = 0; i < Row; i++)
            {
                for (int j = 0; j < Col; j++)
                {
                    int temp = rand.Next(-5, 20);
                    matrix[i, j] = temp < 0 ? 0 : 1;
                }
            }

            matrix[0, 0] = matrix[Row-1, Col-1] = 1;
            Cell source = new Cell();
            Cell destination = new Cell(Row-1, Col-1);
            Maze maze = new Maze(matrix.GetLength(0), matrix.GetLength(1),
matrix, source, destination);
            maze.ManhattanDistance(source);
            var sw = Stopwatch.StartNew();
            List<(int, int)> result = FindWayAlgorithm.BFS(source,
destination, maze);
            sw.Stop();
            maze.PrintMaze();
            Console.WriteLine();
            if (result.Count > 0)
            {
                Console.WriteLine($"The          shortest          path          is:
{result.Count}.");
                // Console.WriteLine("Path is: ");
                // foreach (var cell in result)

```

```

        // {
        //     if (result.IndexOf(cell) == result.Count-1)
        //     {
        //         Console.Write($"({cell})");
        //     }
        //     else
        //     {
        //         Console.Write($"({cell}) -> ");
        //     }
        // }
    }
    else
    {
        Console.WriteLine("Path doesn't exist.");
    }

    Console.WriteLine($"Time of working the BFS algorithm:
{sw.Elapsed.TotalSeconds}");

    result.Clear();
    sw = Stopwatch.StartNew();
    result = FindWayAlgorithm.AStar(source, destination, maze);
    sw.Stop();
    Console.WriteLine();
    if (result.Count > 0)
    {
        Console.WriteLine($"The shortest path is:
{result.Count}.");

        // foreach (var cell in result)
        // {
        //     if (result.IndexOf(cell) == result.Count-1)
        //     {
        //         Console.Write($"({cell})");
        //     }
        //     else
        //     {
        //         Console.Write($"({cell}) -> ");
        //     }
        // }
    }
    else
    {
        Console.WriteLine("Path doesn't exist.");
    }
}

```

```

        Console.WriteLine($"Time of working the AStar algorithm:
{sw.Elapsed.TotalSeconds}");
        result.Clear();
        Console.ReadLine();
    }

}

using System;
using System.Collections.Generic;
using System.ComponentModel;
using Microsoft.SqlServer.Server;

namespace lab2
{
    public class FindWayAlgorithm
    {
        public static List<(int, int)> BFS(Cell source, Cell destination,
Maze maze)
        {
            int[] rowsMove = { -1, 0, 0, 1 };
            int[] colsMove = { 0, -1, 1, 0 };
            if (maze.Matrix[source.X,source.Y] != 1 ||
maze.Matrix[destination.X,destination.Y] != 1)
            {
                return new List<(int, int)>();
            }
            bool[,] visitedCells = new bool[maze.Matrix.GetLength(0),
maze.Matrix.GetLength(1)];
            visitedCells[source.X, source.Y] = true;
            List<Cell> Queue = new List<Cell> {source};
            while (Queue.Count != 0)
            {
                Cell current = Queue[0];
                if (current.X == destination.X && current.Y ==
destination.Y)
                {
                    return destination.Path = current.Path;
                    // return current.Distance;
                }
                Queue.RemoveAt(0);
                for (int i = 0; i < 4; i++)
                {

```

```

        int row = current.X + rowsMove[i];
        int col = current.Y + colsMove[i];
        if (maze.IsValid(row, col) && maze.Matrix[row, col] ==
1 && !visitedCells[row, col])
        {
            visitedCells[row, col] = true;
            Cell NeighborCell = new Cell(row, col)
            {
                Path = new List<(int, int)>(current.Path)
            };
            NeighborCell.Path.Add((NeighborCell.X,
NeighborCell.Y));

            Queue.Add(NeighborCell);
        }
    }
    Queue.Clear();
    return new List<(int, int)>();
}

public static List<(int, int)> AStar(Cell source, Cell destination,
Maze maze)
{
    int[] rowsMove = { -1, 0, 0, 1 };
    int[] colsMove = { 0, -1, 1, 0 };

    if (maze.Matrix[source.X, source.Y] != 1 ||
maze.Matrix[destination.X, destination.Y] != 1)
    {
        return new List<(int, int)>();
    }

    bool[,] visitedCells = new bool[maze.Matrix.GetLength(0),
maze.Matrix.GetLength(1)];
    visitedCells[source.X, source.Y] = true;
    List<Cell> PriorityQueue = new List<Cell>();
    PriorityQueue.Add(source);
    while (PriorityQueue.Count != 0)
    {
        Cell current = PriorityQueue[0];
        maze.ManhattanDistance(current);
    }
}

```

```

        if (current.X == destination.X && current.Y ==
destination.Y)
        {
            return destination.Path = current.Path;
        }
        PriorityQueue.RemoveAt(0);
        for (int i = 0; i < 4; i++)
        {
            int row = current.X + rowsMove[i];
            int col = current.Y + colsMove[i];
            if (maze.IsValid(row, col) && maze.Matrix[row, col] ==
1 && !visitedCells[row, col])
            {
                visitedCells[row, col] = true;
                Cell NeighborCell = new Cell(row, col)
                {
                    Path = new List<(int, int)>(current.Path)
                };
                NeighborCell.Path.Add((NeighborCell.X,
NeighborCell.Y));

                maze.ManhattanDistance(NeighborCell);
                AddToQueue(PriorityQueue, NeighborCell);
            }
        }
        PriorityQueue.Clear();
        return new List<(int, int)>();
    }

    public static void AddToQueue(List<Cell> Queue, Cell cell)
    {
        if (Queue.Count > 0)
        {
            int index = 0;
            while (index < Queue.Count && cell.MDistance +
cell.Path.Count > Queue[index].MDistance + Queue[index].Path.Count)
            {
                ++index;
            }
            if (index < Queue.Count - 1)
            {

```

```

        Queue.Insert(index, cell);
    }
    else
    {
        Queue.Add(cell);
    }
}
else
{
    Queue.Add(cell);
}
}

}
}
using System;

namespace lab2
{
    public class Maze
    {
        public int Row;
        public int Col;
        public int[,] Matrix;
        public Cell Source;
        public Cell Destination;

        public Maze(int row, int col, int[,] matrix, Cell source, Cell
destination)
        {
            Row = row;
            Col = col;
            Matrix = matrix;
            Source = source;
            Destination = destination;
        }
        public Maze()
        {
            Row = 10;
            Col = 10;
            Matrix = new int[,]
            {
                {1, 0, 1, 1, 0, 0, 0, 1, 0, 0},

```

```

        {1, 1, 1, 0, 1, 1, 1, 0, 0, 0},
        {0, 1, 1, 1, 1, 1, 1, 1, 0, 1},
        {1, 1, 0, 1, 1, 1, 0, 1, 1, 1},
        {0, 1, 0, 1, 0, 1, 0, 0, 0, 1},
        {1, 1, 1, 1, 1, 0, 0, 0, 1, 1},
        {0, 1, 0, 0, 0, 1, 1, 1, 1, 0},
        {0, 1, 0, 0, 0, 1, 1, 0, 0, 0},
        {0, 1, 0, 0, 0, 1, 0, 0, 0, 0},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
    };

    Source = new Cell(0, 0);
    Destination = new Cell(9, 9);
}

public bool IsValid(int row, int col)
{
    return ((row >= 0) && (row < Row) && (col >= 0) && (col < Col));
}

public void ManhattanDistance(Cell current)
{
    int x = Math.Abs(Destination.X - current.X);
    int y = Math.Abs(Destination.Y - current.Y);
    current.MDistance = x + y;
}

public void PrintMaze()
{
    Console.Write(" ");
    for (int num = 0; num < Matrix.GetLength(1); num++)
    {
        Console.Write("=");
    }
    Console.WriteLine();
    for (int i = 0; i < Matrix.GetLength(0); i++)
    {
        Console.Write("||");
        for (int j = 0; j < Matrix.GetLength(1); j++)
        {
            if (Matrix[i, j] == 0)
            {
                Console.Write("x");
            }
            else if (Matrix[i, j] == 1)

```



```

        {
            if (i == Source.X && j == Source.Y)
            {
                Console.Write("S");
            }
            else if (i == Destination.X && j == Destination.Y)
            {
                Console.Write("D");
            }
            else
            {
                Console.Write(" ");
            }
        }

        Console.Write("||");
        Console.WriteLine();
    }
    Console.Write(" ");
    for (int num = 0; num < Matrix.GetLength(1); num++)
    {
        Console.Write("=");
    }
}

}

using System;
using System.Collections.Generic;

namespace lab2
{
    public class Cell
    {
        public int X;
        public int Y;
        public int MDistance;
        public List<(int, int)> Path;

        public Cell(int x, int y)
        {
            X = x;
            Y = y;
            MDistance = Int32.MaxValue;

```

```

        Path = new List<(int, int)>{(X, Y)};
    }

    public Cell()
    {
        X = 0;
        Y = 0;
        MDistance = Int32.MaxValue;
        Path = new List<(int, int)>{(X, Y)};
    }
}

```

### 3.2.2 Приклади роботи

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

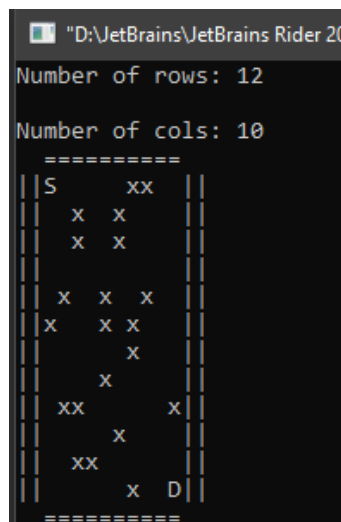


Рисунок 3.1 – Випадковим чином заданий лабіринт

```

The shortest path is: 21.
Path is: ((0, 0)) -> ((0, 1)) -> ((0, 2)) -> ((0, 3)) -> ((0, 4)) -> ((1, 4)) -> ((2, 4)) -> ((3, 4)) -> ((3, 5)) -> ((3, 6)) -> ((3, 7)) -> ((3, 8)) ->
((4, 8)) -> ((5, 8)) -> ((6, 8)) -> ((7, 8)) -> ((8, 8)) -> ((9, 8)) -> ((9, 9)) -> ((10, 9)) -> ((11, 9))
Time of working the BFS algorithm: 0,0024031

```

Рисунок 3.2 – Алгоритм BFS

```

The shortest path is: 21.
((0, 0)) -> ((0, 1)) -> ((1, 1)) -> ((2, 1)) -> ((3, 1)) -> ((3, 2)) -> ((4, 2)) -> ((5, 2)) -> ((6, 2)) -> ((7, 2)) -> ((7, 3)) -> ((8, 3)) -> ((9, 3)) ->
((9, 4)) -> ((10, 4)) -> ((10, 5)) -> ((10, 6)) -> ((10, 7)) -> ((11, 7)) -> ((11, 8)) -> ((11, 9))
Time of working the AStar algorithm: 0,0005848

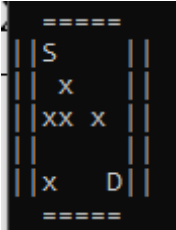
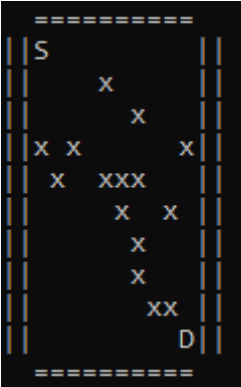

```

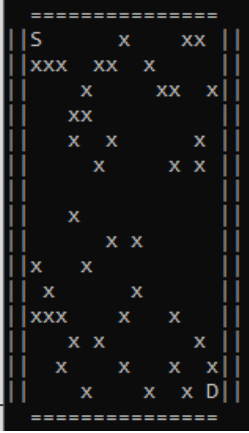
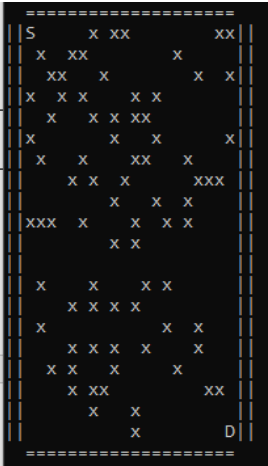

Рисунок 3.3 – Алгоритм A-Star

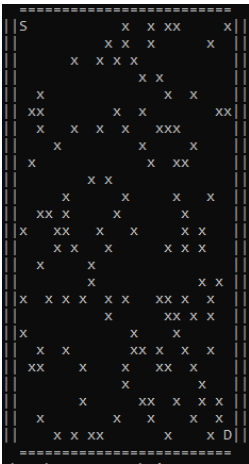

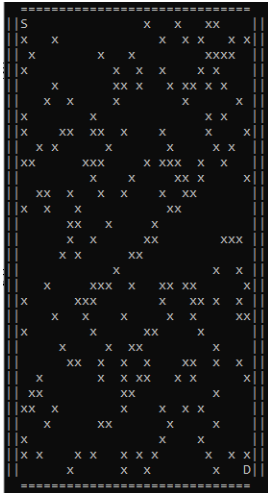
### 3.3 Дослідження алгоритмів

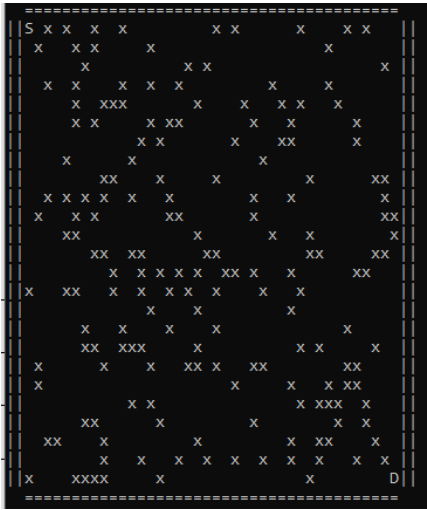
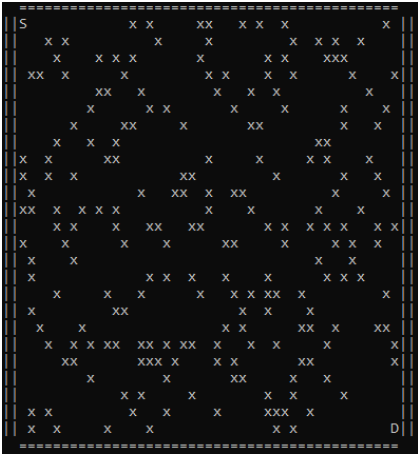
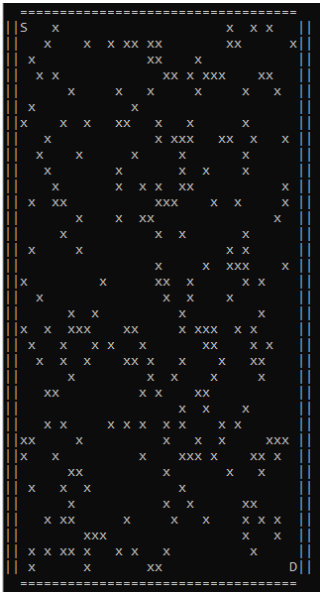
В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS, задачі пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху для 20 початкових станів.

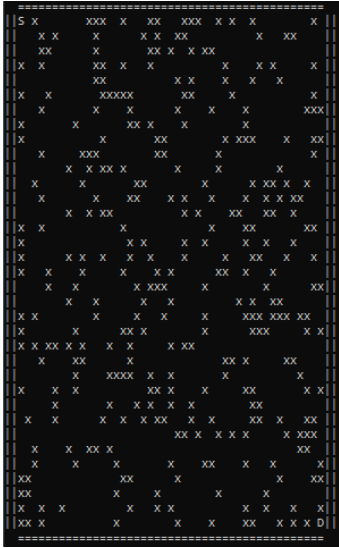
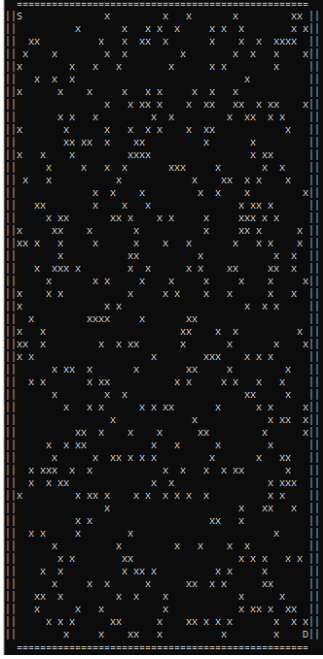
Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

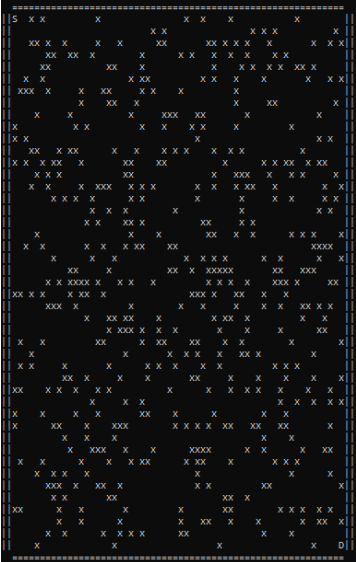
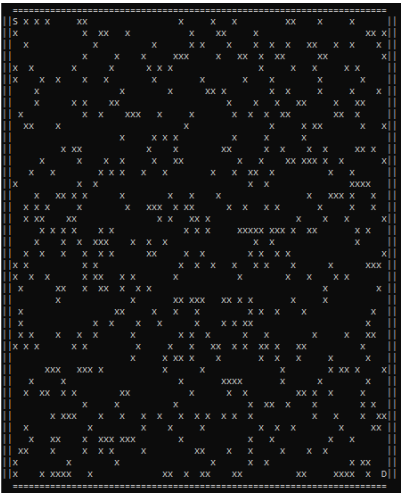
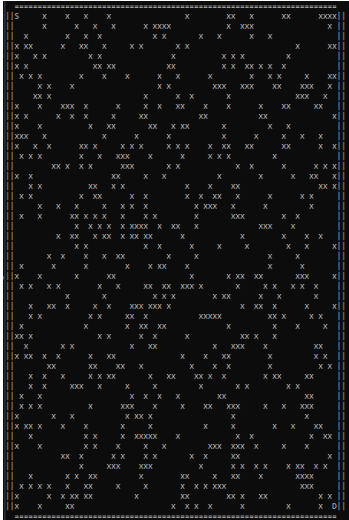
Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам'яті
	20	1	76	19
	85	1	336	84
	160	0	636	159

	181	0	720	180
	321	0	1280	320
	350	1	1396	349

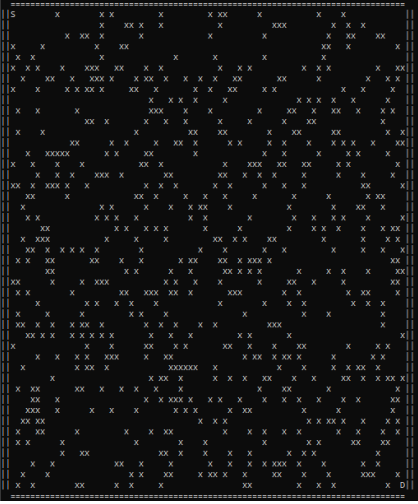
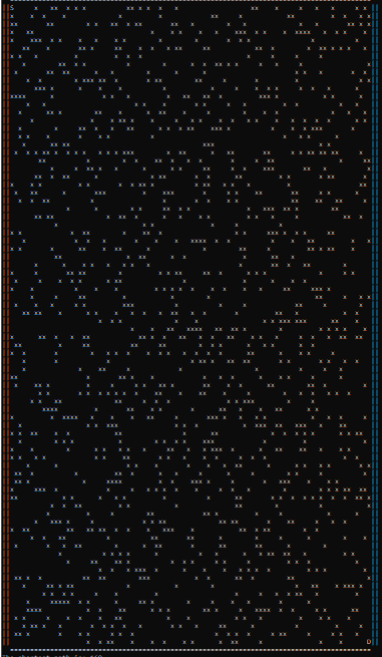
	506	0	2020	505
	363	0	1448	363
	702	0	2804	701

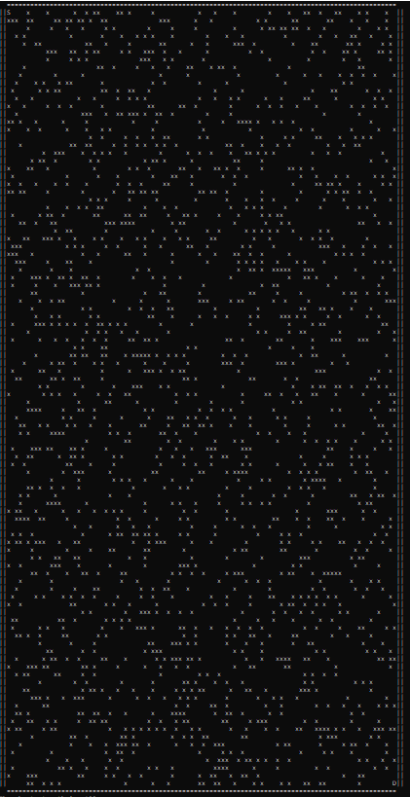
	793	0	3168	792
	908	0	3628	907
	996	1	3980	995

	1230	0	4916	1229
	2038	0	8148	2037

	2172	0	8684	2171
	2256	0	9020	2255
	2781	0	11120	2780



	2865	0	11456	2868
	5740	0	22956	5739

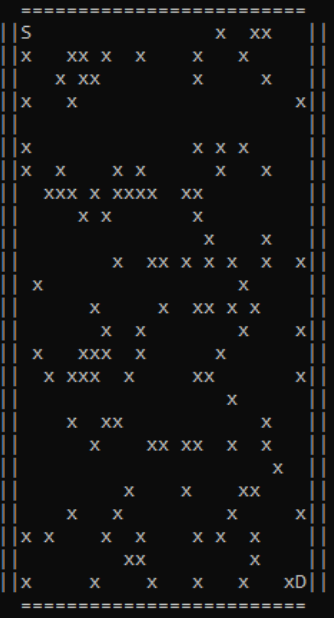
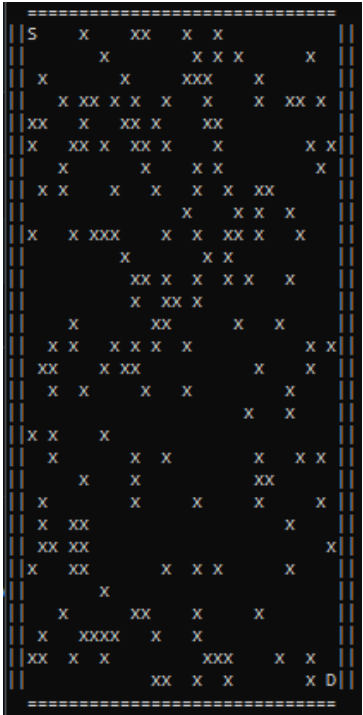
	7973	0	31888	7972
---	------	---	-------	------

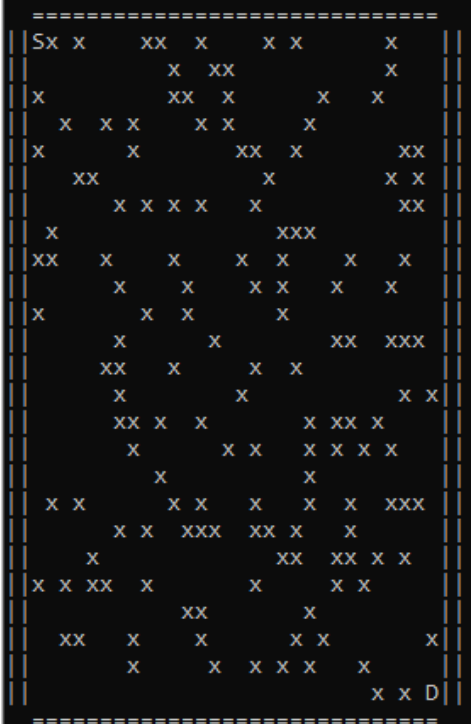
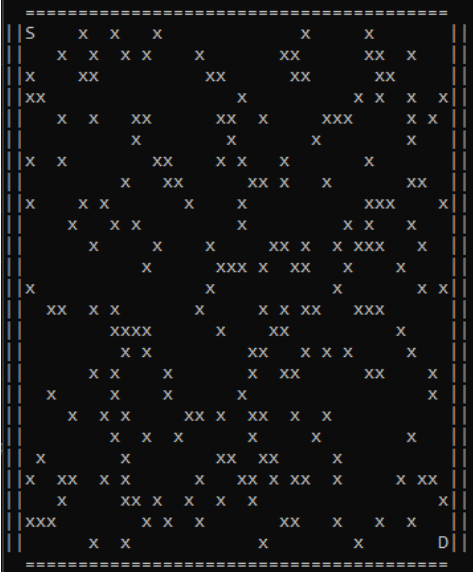
В таблиці 3.2 наведені характеристики оцінювання алгоритму A\*, задачі пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху для 20 початкових станів.

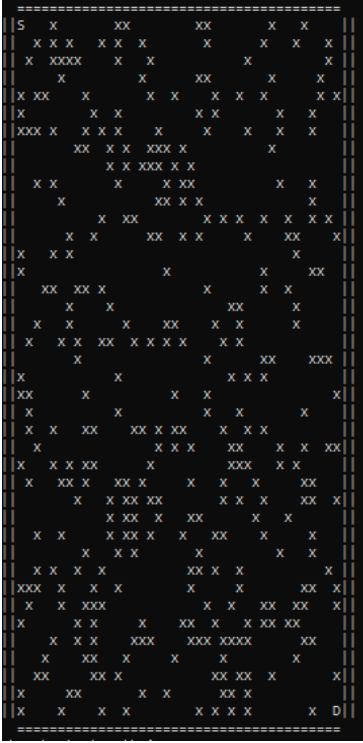
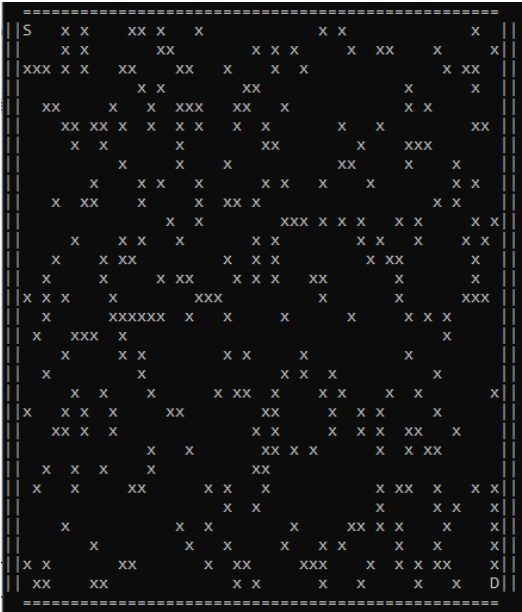
Таблиця 3.3 – Характеристики оцінювання A\*

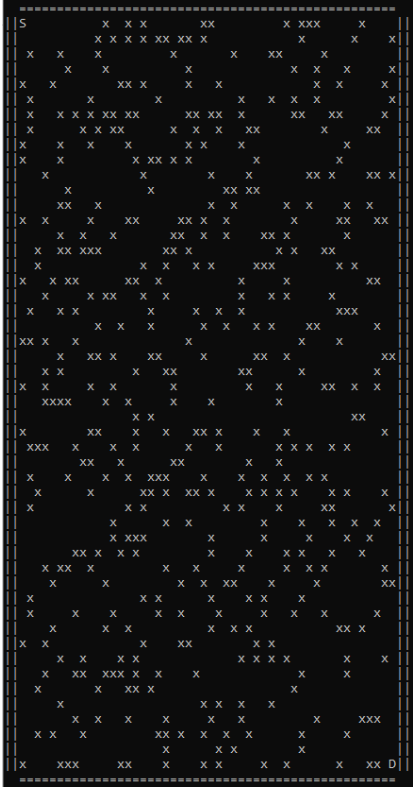
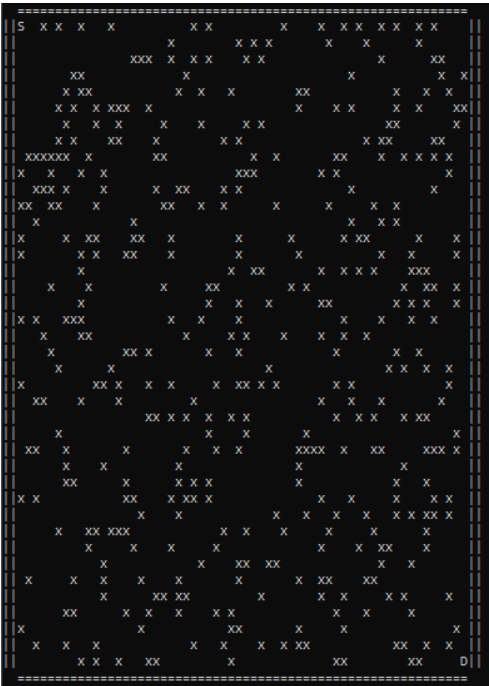
Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам'яті
	11	0	40	17
	21	1	80	36
	45	0	176	77

	50	0	196	88
	54	0	212	96

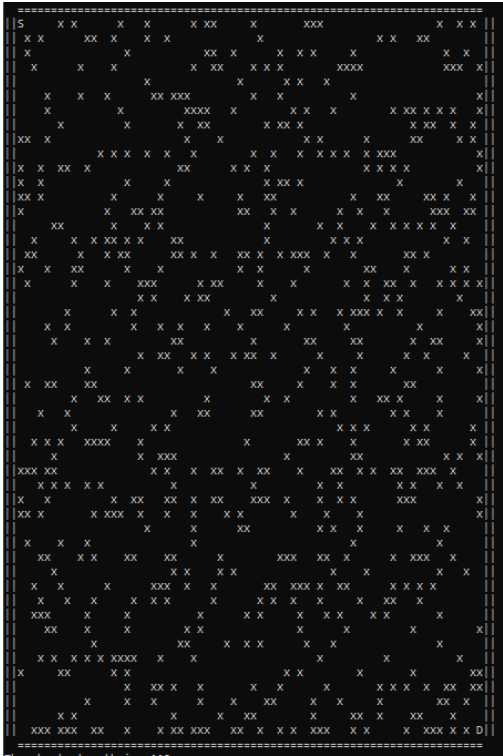
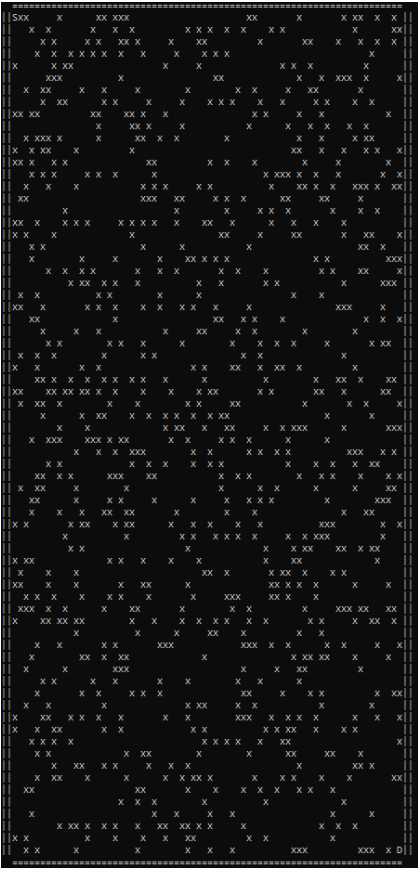
	91	0	360	151
	115	0	456	192

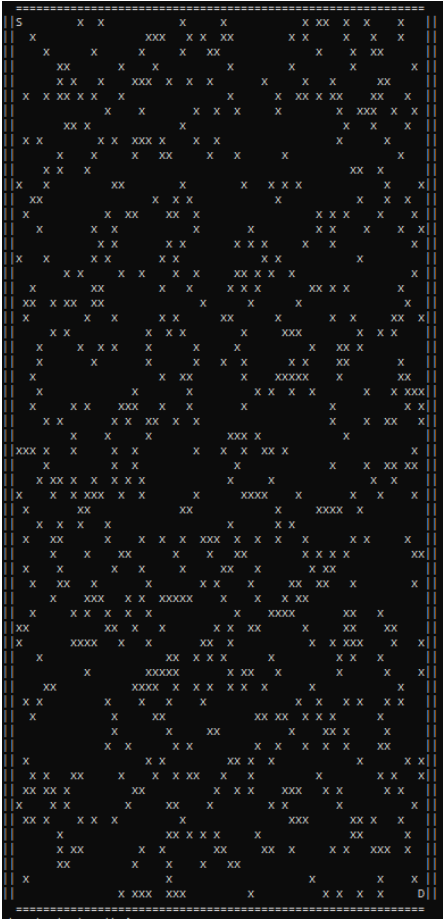
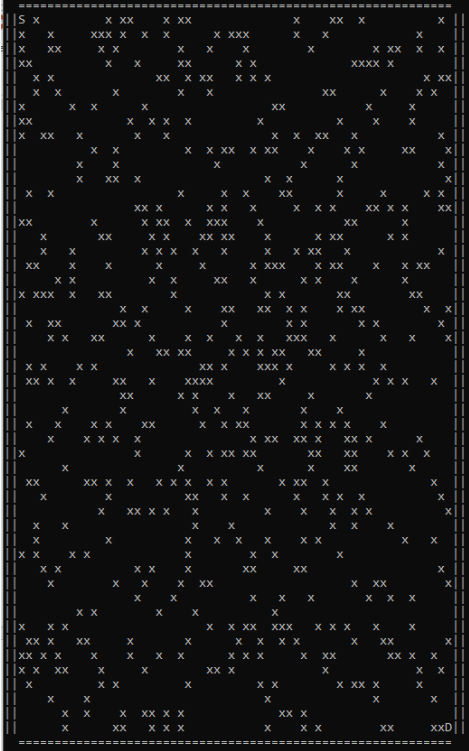
	163	1	648	248
	210	0	836	327

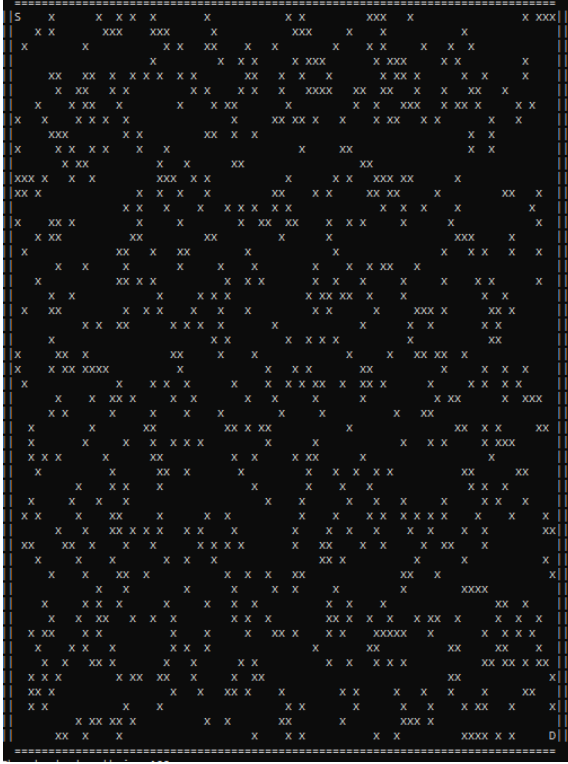
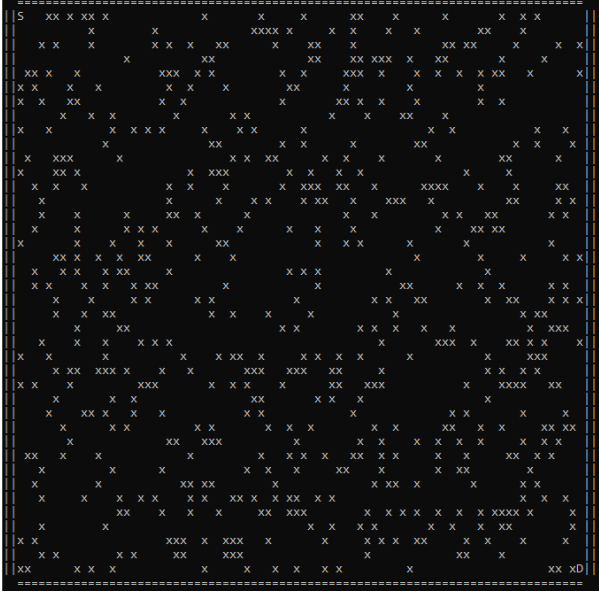
	225	0	896	349
	227	0	904	348

	266	0	1060	421
	274	0	1092	437



	534	0	2132	807
	580	0	2316	910

	677	0	2704	1033
	791	0	3160	1188

	940	0	3756	1387
	1111	0	4440	1406



## ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто два алгоритми вирішення задачі пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху. Було розроблено та використано два алгоритми: BFS та  $A^*$ (AStar). За результатами роботи та тестування можна стверджувати, що алгоритм  $A^*$ (AStar) є більш оптимальним та швидкодійним, використовує меншу кількість пам'яті та знаходить рішення за меншу кількість ітерацій. Винятком може слугувати «незручність» сформованого лабіринту для враховування евристичної функції. BFS є більш стабільним, хоч і потребує майже вдвічі більших обсягів пам'яті та набагато більше часу для знаходження рішення.

## 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%.