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: Григоренко Родіон Ярославович

Перевірив *Сопов О. О.*

(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

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 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·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

ВИКОНАННЯ

Псевдокод алгоритмів

АЛГОРИТМ IDS

public bool IDS(Node root, Node goal, int maxDepth)

ПОЧАТОК

ДЛЯ(int i = 0; i < maxDepth; i++)

ЯКЩО (DFS(root, goal, i))

```
                    return true;
        return false;
        КІНЕЦЬ
public bool DFS(Node root, Node goal, int depth)
ПОЧАТОК
                                ЯКЩО (root == goal)
                            return true;
        ЯКЩО (depth == 0)
                return false;

        ДЛЯ КОЖНОГО child в AllNodes
                ЯКЩО (child == root)
                        continue;
                ЯКЩО (DFS(child, goal, depth - 1))
                        Path.Add(child);
                        return true;
         return false;
  КІНЕЦЬ
                            АЛГОРИТМ A*
public bool PathFinding(Node startNode, Node endNode)
ПОЧАТОК

OpenList.Add(startNode);

ДЛЯ КОЖНОГО node В AllNodes
        node.Gcost = int.MaxValue;
        node.CalculateFcost();
```

```
            node.CameFromNode = null;

SetManhattanDistances(endNode);

startNode.Gcost = 0;

ПОКИ (OpenList.Count > 0)
                        Node currentNode = GetLowestFcostNode(OpenList);

        ЯКЩО (currentNode == endNode)
                CalculatePath(endNode);
                Time = (int)sw.ElapsedMilliseconds;
                UnityEngine.Debug.Log(Time);
                return true;

        OpenList.Remove(currentNode);
        ClosedList.Add(currentNode);

                ДЛЯ КОЖНОГО node В currentNode.Children
                        ЯКЩО (ClosedList.Contains(node))
                                continue;

                        int tentativeGcost = currentNode.Gcost + 1;

                        ЯКЩО (tentativeGcost < node.Gcost)
                                node.CameFromNode = currentNode;
                                node.Gcost = tentativeGcost;
```

node.CalculateFcost();

ЯКЩО (!OpenList.Contains(node))

OpenList.Add(node);

return false;

Програмна реалізація

Вихідний код

Node.cs

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

public class Node
{
    public List<Node> Children = new List<Node>();
    public Node CameFromNode;
    public Transform Cell;
    public bool IsWall;
    public int Gcost;
    public int Hcost;
```

```
    public int Fcost;
    public int ManhattanDistance;

    public Node(Transform cell)
    {
        Cell = cell;
    }

    public void CalculateFcost()
    {
        Fcost = Gcost + Hcost;
    }
}
```

Cell.cs

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

public class Cell : MonoBehaviour
{
    public bool IsWall = false;
    public bool IsStartNode = false;
    public bool IsGoalNode = false;
```

```
    public void OnMouseDown()
    {
        if (!GameManager.instance.ChoosingStartNode &&
!GameManager.instance.ChoosingGoalNode)
        {
            IsWall = !IsWall;
            RefreshColor();
        }
        else if (GameManager.instance.ChoosingStartNode)
        {
            GameManager.instance.LabirynthGenerator.SetStartCell(this);
            IsStartNode = true;
            IsGoalNode = false;
            IsWall = false;
            RefreshColor();
        }
        else if (GameManager.instance.ChoosingGoalNode)
        {
            GameManager.instance.LabirynthGenerator.SetGoalCell(this);
            IsGoalNode = true;
            IsStartNode = false;
            IsWall = false;
            RefreshColor();
        }
    }
```

```
    public void RefreshColor()
    {
        if (IsWall)
        {
            GetComponentInChildren<SpriteRenderer>().color =
Color.black;
        }
        else
        {
            GetComponentInChildren<SpriteRenderer>().color =
Color.white;
        }


        if (IsStartNode)
        {
            GetComponentInChildren<SpriteRenderer>().color =
Color.green;
        }
        else if (IsGoalNode)
        {
            GetComponentInChildren<SpriteRenderer>().color =
Color.blue;
        }
    }
}
```

GameManager.cs

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

public class GameManager : MonoBehaviour
{
    public static GameManager instance;

    public LabirynthGenerator LabirynthGenerator;
    public bool ChoosingStartNode = false;
    public bool ChoosingGoalNode = false;
    private void Start()
    {
        instance = this;
    }

    public void ChooseStartNode()
    {
        ChoosingStartNode = true;
        ChoosingGoalNode = false;
    }
```

```
    public void ChooseGoalNode()
    {
        ChoosingStartNode = false;
        ChoosingGoalNode = true;
    }

    public void ChooseWalls()
    {
        ChoosingStartNode = false;
        ChoosingGoalNode = false;
    }


}
```

LabirynthGenerator.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Diagnostics;
using TMPro;
public class LabirynthGenerator : MonoBehaviour
{
    private IDS_Alrorithm IDS = new IDS_Alrorithm();
```

```
    private AStarAlgorithm AStar = new AStarAlgorithm();
    [SerializeField] Transform _grid;
    [SerializeField] private GameObject _nodePrefab;
    private List<Transform> _allNodesTransforms = new
List<Transform>();
    [SerializeField] private Transform _startCell;
    [SerializeField] private Transform _goalCell;
    private Node _startNode;
    private Node _goalNode;

    [SerializeField] TextMeshProUGUI _time;
    private Stopwatch sw = new Stopwatch();

    [SerializeField] TextMeshProUGUI _iterations;
    [SerializeField] TextMeshProUGUI _statesAmount;
    [SerializeField] TextMeshProUGUI _deadEnds;
    [SerializeField] TextMeshProUGUI _statesInMemory;



    public void GenerateBlankGrid()
    {
        for (int i = 0; i < 20; i++)
        {
            for (int j = 0; j < 20; j++)
            {
```

```
            var node = Instantiate(_nodePrefab, new Vector3(j * 1.1f, i
* 1.1f, 0), Quaternion.identity, _grid);
            _allNodesTransforms.Add(node.transform);
        }
    }
}

public void SetNodesCells(Algorithm algo)
{
    algo.AllNodes = new List<Node>();
    foreach (var cell in _allNodesTransforms)
    {
        algo.AllNodes.Add(new Node(cell));
    }
}


private void Start()
{
    GenerateBlankGrid();
    SetNodesCells(IDS);
    SetNodesCells(AStar);
    IDS.SetNodesChildren();
    AStar.SetNodesChildren();

}
```

```
public void FindPathByIDS()
{
    IDS.SetWalls();
    sw.Restart();
    IDS.SetNodesChildren();
    _startNode = IDS.FindNodeByCell(_startCell);
    _goalNode = IDS.FindNodeByCell(_goalCell);
    IDS.IDS(_startNode, _goalNode, 20);
    _time.text = sw.ElapsedMilliseconds.ToString();
    _iterations.text = IDS.Iterations.ToString();
    _statesAmount.text = IDS.StatesAmount.ToString();
    _statesInMemory.text = IDS.StatesInMemory.ToString();
    _deadEnds.text = IDS.DeadEnds.ToString();
    ShowPath(IDS);

}

public void FindPathByAStar()
{
    AStar.SetWalls();
    AStar.SetNodesChildren();
    _startNode = AStar.FindNodeByCell(_startCell);
    _goalNode = AStar.FindNodeByCell(_goalCell);
    AStar.SetManhattanDistances(_goalNode);
```

```
        AStar.PathFinding(_startNode, _goalNode, sw);
        _time.text = AStar.Time.ToString();
        _iterations.text = AStar.Iterations.ToString();
        _statesAmount.text = AStar.StatesAmount.ToString();
        _statesInMemory.text = AStar.StatesInMemory.ToString();
        _deadEnds.text = AStar.DeadEnds.ToString();
        ShowPath(AStar);
    }

    public void ShowPath(Algorithm algo)
    {
        foreach(var node in algo.Path)
        {

node.Cell.GetChild(0).GetComponent<SpriteRenderer>().color =
Color.red;
        }
    }

    public void SetStartCell(Cell cell)
    {
        foreach(var node in AStar.AllNodes)
        {
            if (node.Cell == cell)
                continue;
            node.Cell.GetComponent<Cell>().IsStartNode = false;
```

```
            node.Cell.GetComponent<Cell>().RefreshColor();
        }
        _startCell = cell.transform;
    }

    public void SetGoalCell(Cell cell)
    {
        foreach (var node in AStar.AllNodes)
        {
            if (node.Cell.GetComponent<Cell>() == cell)
                continue;
            node.Cell.GetComponent<Cell>().IsGoalNode = false;
            node.Cell.GetComponent<Cell>().RefreshColor();
        }
        _goalCell = cell.transform;
    }

}
```

Algorithm.cs

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

```
public class Algorithm
{
    public List<Node> AllNodes = new List<Node>();

    public List<Node> Path = new List<Node>();

    public long StatesAmount = 0;
    public long Iterations = 0;
    public long DeadEnds = 0;
    public long StatesInMemory = 0;

    public void SetNodesChildren()
    {
        for (int i = 0; i < 20; i++)
        {
            for (int j = 0; j < 20; j++)
            {
                AllNodes[20 * i + j].Children = new List<Node>();
                if (AllNodes[20 * i + j].IsWall)
                    continue;
                if (j + 1 < 20 && !AllNodes[20 * i + j + 1].IsWall)
                    AllNodes[20 * i + j].Children.Add(AllNodes[20 * i + j +
1]);
                if (i + 1 < 20 && !AllNodes[20 * (i + 1) + j].IsWall)
                    AllNodes[20 * i + j].Children.Add(AllNodes[20 * (i + 1)
+ j]);
```

```
            if (j - 1 >= 0 && !AllNodes[20 * i + j - 1].IsWall)
                AllNodes[20 * i + j].Children.Add(AllNodes[20 * i + j - 1]);
            if (i - 1 >= 0 && !AllNodes[20 * (i - 1) + j].IsWall)
                AllNodes[20 * i + j].Children.Add(AllNodes[20 * (i - 1) + j]);
        }
    }
}

public void SetWalls()
{
    foreach (var node in AllNodes)
    {
        if (node.Cell.GetComponent<Cell>().IsWall)
        {
            node.IsWall = true;
        }
        else
        {
            node.IsWall = false;
        }
    }
}
public Node FindNodeByCell(Transform cell)
{
```

```
        for (int i = 0; i < AllNodes.Count; i++)
        {
            if (AllNodes[i].Cell == cell)
                return AllNodes[i];
        }
        throw new System.Exception("Can't find the node");
    }

    public void SetManhattanDistances(Node goalNode)
    {
        Vector2 goalCoords = new
Vector2(AllNodes.IndexOf(goalNode)%20,
AllNodes.IndexOf(goalNode) / 20);
        for (int i = 0; i < 20; i++)
        {
            for (int j = 0; j < 20; j++)
            {
                AllNodes[20 * i + j].Hcost = Mathf.Abs((int)goalCoords.x -
j) + Mathf.Abs((int)goalCoords.y - i);
            }
        }
    }

}
```

IDS_Algorithm.cs

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

public class IDS_Alrorithm : Algorithm
{

    public bool IDS(Node root, Node goal, int maxDepth)
    {
        for (int i = 0; i < maxDepth; i++)
        {
            if(DFS(root, goal, i))
            {
                return true;
            }
        }

        return false;
    }

    public bool DFS(Node root, Node goal, int depth)
    {
        if (root == goal)
        {
            StatesInMemory++;
```

```
    return true;
}


if (depth == 0)
{
    DeadEnds++;
    return false;
}

foreach(var child in root.Children)
{
    StatesAmount++;
    Iterations++;
    if (child == root)
        continue;
    if(DFS(child, goal, depth - 1))
    {
        Path.Add(child);
        StatesInMemory++;
        return true;

    }
}

DeadEnds++;
```

```
        return false;
    }


}
```

AStarAlgorithm.cs

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

public class AStarAlgorithm : Algorithm
{
    public List<Node> ClosedList = new List<Node>();
    public List<Node> OpenList = new List<Node>();
    public int Time;

    public bool PathFinding(Node startNode, Node endNode,
Stopwatch sw)
    {
        sw.Start();
        OpenList.Add(startNode);
```

```
foreach (var node in AllNodes)
{
    node.Gcost = int.MaxValue;
    node.CalculateFcost();
    node.CameFromNode = null;
}
SetManhattanDistances(endNode);

startNode.Gcost = 0;

while (OpenList.Count > 0)
{
    Iterations++;
    Node currentNode = GetLowestFcostNode(OpenList);
    if (currentNode == endNode)
    {
        CalculatePath(endNode);
        Time = (int)sw.ElapsedMilliseconds;
        UnityEngine.Debug.Log(Time);
        return true;
    }

    OpenList.Remove(currentNode);
    ClosedList.Add(currentNode);

    foreach (var node in currentNode.Children)
```

```
        {
            StatesAmount++;
            StatesInMemory++;
            if (ClosedList.Contains(node))
                continue;

            int tentativeGcost = currentNode.Gcost + 1;
            if (tentativeGcost < node.Gcost)
            {
                node.CameFromNode = currentNode;
                node.Gcost = tentativeGcost;
                node.CalculateFcost();

                if (!OpenList.Contains(node))
                {
                    OpenList.Add(node);
                }
            }
        }
    }

    DeadEnds++;
    return false;

}
```

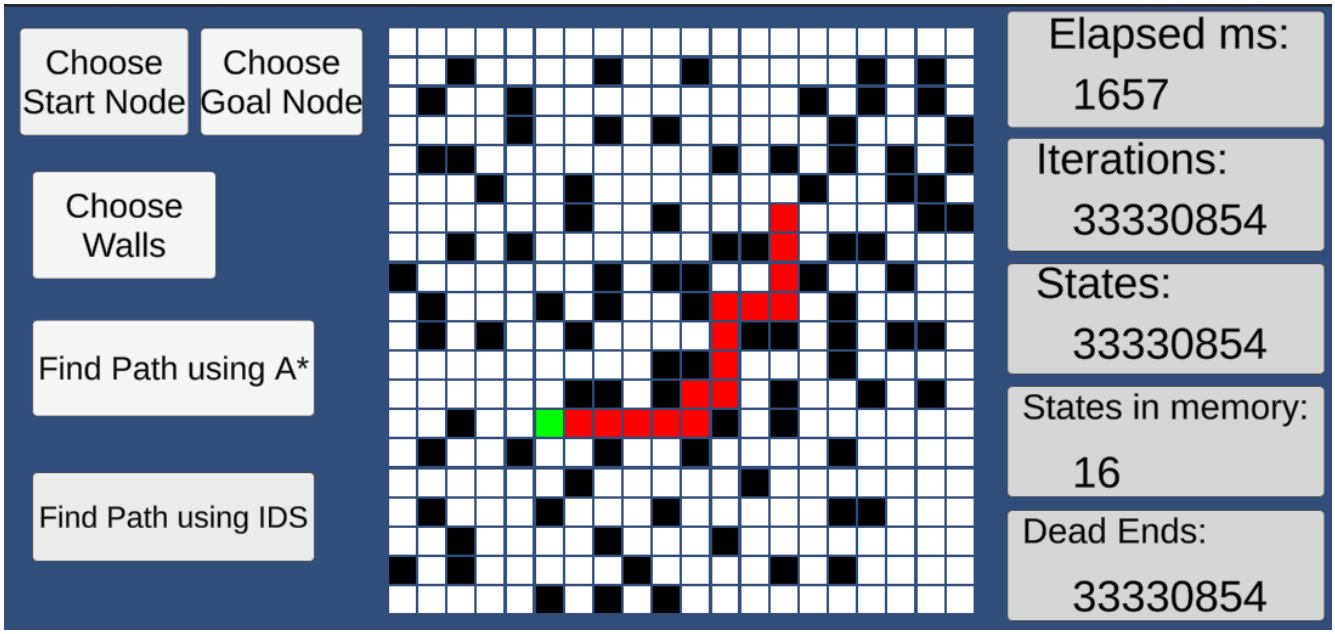
```
    private void CalculatePath(Node endNode)
    {
        Node currentNode = endNode;
        while (currentNode.CameFromNode != null)
        {
            Path.Add(currentNode);
            currentNode = currentNode.CameFromNode;
        }
        Path.Reverse();
    }

    private Node GetLowestFcostNode(List<Node> openList)
    {
        Node lowestFcostNode = openList[0];
        for (int i = 1; i < openList.Count; i++)
        {
            if (openList[i].Fcost < lowestFcostNode.Fcost)
            {
                lowestFcostNode = openList[i];
            }
        }
        return lowestFcostNode;
    }
}
```

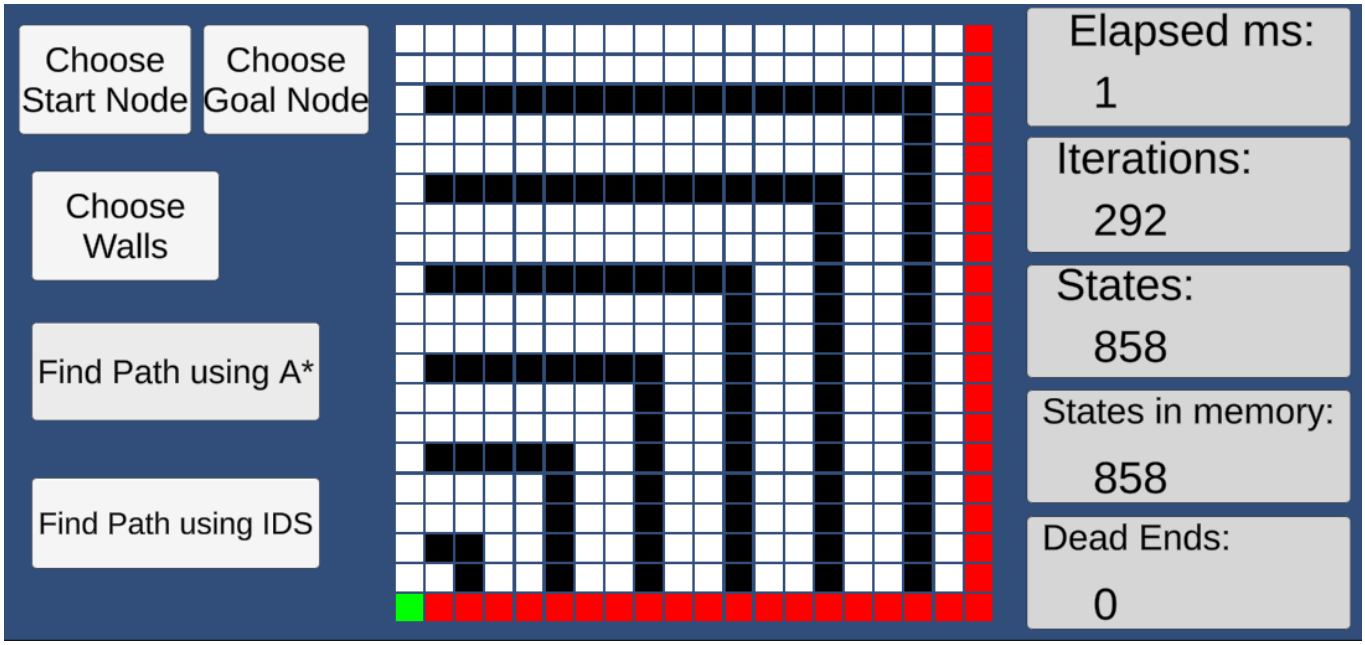
Приклади роботи

На картинках представлені приклади роботи обох алгоритмів.

IDS



A*



Дослідження алгоритмів

У таблицях наведені дані оцінювання обох алгоритмів.

IDS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам'яті	Час в мілісекундах
0:0 5:5	86329	86329	86329	13	5
1:3 7:1	26382	26382	26382	11	1
7:3 2:7	1513027	1513027	1513027	12	74
3:7 6:1	69732	69732	69732	10	3
9:8 11:12	7493	7493	7493	9	1
0:0 2:2	141345	141345	141345	15	7
1:2 8:5	29004	29004	29004	11	1

0:0 8:8	154171312	154171312	154171312	17	7029
1:5 6:0	2377264	2377264	2377264	13	115
3:7 5:2	8519	8519	8519	8	0
7:5 2:3	6921684680	6921684700	6921684680	0	330005
8:2 1:8	129962256	129962256	129962256	16	6178
4:3 8:8	349212	349212	349212	10	16
2:2 6:5	4493	4493	4493	10	0
1:5 7:1	182410	182410	182410	11	9
1:1 6:3	2182	2182	2182	8	0
3:3 8:7	820902	820902	820902	14	42
3:6 9:1	955316	955316	955316	12	45
12:5 0:3	2046452840	2046452840	2046452840	17	90071
0:0 5:2	4707	4707	4707	10	0

Середня кількість ітерацій: 462 942 470.25

Середня кількість глухих кутів: 462 942 470.25

Середня кількість згенерованих станів: 462 942 470.25

Середня кількість станів у пам’яті: 11.35

Середній час в мілісекундах: 21 680.1

A*

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті	Час в мілісекундах

0:0 19:19	345	0	1106	1106	1
1:9 12:3	42	0	131	131	0
16:0 0:16	179	0	559	559	1
0:0 19:19	182	1	674	674	0
5:1 11:14	68	0	197	197	0
1:5 8:4	76	0	234	234	0
1:4 11:5	59	0	185	185	0
8:9 10:7	266	0	952	952	1
2:1 13:9	77	0	233	233	0
3:5 9:15	49	0	147	147	0
1:7 16:5	43	0	131	131	0
0:0 19:19	238	0	811	811	1
3:12 12:2	260	0	917	917	1
2;1 16:16	110	0	316	316	0
5:3 1:9	59	0	184	184	0
11:4 5:14	79	0	255	255	0
7:5 9:11	18	0	46	46	0
5:5 11:5	197	0	697	697	1
17:12 1:3	103	0	318	318	0
0:0 19:19	310	0	1016	1016	1

Середня кількість ітерацій: 138

Середня кількість глухих кутів: 0.05

Середня кількість згенерованих станів: 455.45

Середня кількість станів у пам’яті: 455.45

Середній час в мілісекундах: 0.35

ВИСНОВОК

При виконанні даної лабораторної роботи Я розглянув наступні алгоритми: алгоритм неінформативного пошуку – пошук з ітеративним заглибленням(IDS), а також алгоритм інформативного пошуку – пошук A*.

В ході виконання роботи дослідив дані алгоритми та провів експерименти по 20 серій для кожного алгоритма.

Їх результати:

A*:

Середня кількість ітерацій: 138

Середня кількість глухих кутів: 0.05

Середня кількість згенерованих станів: 455.45

Середня кількість станів у пам’яті: 455.45

Середній час в мілісекундах: 0.35

IDS:

Середня кількість ітерацій: 462 942 470.25

Середня кількість глухих кутів: 462 942 470.25

Середня кількість згенерованих станів: 462 942 470.25

Середня кількість станів у пам’яті: 11.35

Середній час в мілісекундах: 21 680.1

Порівнюючи алгоритми можна сказати, що A* потребує значно менше ітерацій та генерує значно менше станів, проте зберігає у пам’яті трохи більше станів ніж IDS.

Також алгоритм A* працює набагато швидше. Більшість експериментів була виконана за менше ніж 1 мс. Також A* може зайти в глухий кут лише якщо розв’язку в задачі нема, тобто ціль в лабіринті недосяжна за перешкодами, тоді як кількість глухих кутів у алгоритмі IDS зазвичай дорівнює кількості згенерованих станів.

Отже, можу зробити висновок, що якщо глибина пошуку не перевищує 10-20, то можна використовувати обидва алгоритми, IDS в цьому випадку може знайти розв’язок за прийнятний час, проте якщо глибина більша, то в використанні алгоритму ітеративного заглиблення IDS немає сенсу, A* в більшості характеристик переважає IDS, він швидший, потребує менше машинних ресурсів, і усі його зарактеристики в рази кращі за відповідні характеристики алгоритму IDS.