Черкаський національний університет імені Богдана Хмельницького

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗОВАНИХ СИСТЕМ

КУРСОВА РОБОТА

з дисципліни "Програмування та Алгоритмічні мови"

НА ТЕМУ «ВІЗУАЛІЗАЦІЯ «ХВИЛЬОВОГО» ВАРІАНТУ BFS»

Студента 2 курсу, групи КС-202

напряму підготовки «Програмна інженерія»
спеціальності «Інженерія програмного
Забезпечення»

Невмитого Олега Миколайовича

Керівник викл. кафедри ПЗАС

Порубльов І.М.

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна шкала: _____

Кількість балів: _____Оцінка: ECTS _____

Члени комісії

_____ _____

(підпис) **(прізвище та ініціали)**

_____

(підпис) **(прізвище та ініціали)**

_____ _____

(підпис) **(прізвище та ініціали)**

м. Черкаси – 2021 рік

Зміст

Вступ

Мета цієї курсової роботи - написати програму для візуалізації алгоритму пошуку в ширину.

Пошук в ширину - це метод обходу графа, який працює як з орієнтованими, так і не орієнтованими графами. [2]

Граф – це сукупність об’єктів зв’язаних між собою. Ці об’єкти називають вершинами (або вузли), зв’язані вони між собою ребрами.

Ребра графа можуть бути напрямлені, або не напрямлені. Наприклад у нас є місто **А** і місто **В –** це наші вершини, ребра у міст можна розглянути як дороги.

Якщо з міста А в місто Б веде дорога, в зворотню сторону якої їхати не можна, то це напрямлений граф, а коли ми можемо по тій самій дорозі їхати і в одну, і в зворотню сторону - ненапрямлений. Якщо дві вершини з’єднані вони називаються суміжними, в інакшому разі – не суміжними.

Матриця суміжності – це спосіб подання графу в виді матриці, тобто двовимірниного масиву, де номер колонки та рядка є номерами вершини.

Графи використовуються в різних сферах науки, можна навести декілька прикладів:

1. Файлова система комп'ютера. Ієрархія файлів в багатьох операційних системах має вигляд дерева, яке є окремим випадком графу.
2. Молекули всіх хімічних речовин можна зобразити у вигляді графу, де атоми є вершинами, а зв'язки між ними – ребрами
3. Соціальну мережу також можна подати у вигляді графу, де кожна людина чи соціальна група є вершиною, а зв'язки між ними — ребрами
4. А ось розробка програмного забезпечення та комп'ютерні науки є однією з тих галузей, де графи застосовуються найчастіше. Графи також є зручними для зображення структур даних, блок-схем, потоків даних, схем баз даних та баз знань, скінченних автоматів,

схем комп'ютерних мереж та окремих сайтів, схем викликів підпрограм тощо. Також графи широко використовуються в багатьох алгоритмах пошуку та сортування.[3]

Розділ 1. Огляд алгоритмів роботи програми та інструментарію для візуалізації пошуку в ширину

1.1. Вибір структур даних

В ході реалізації програми було використано декілька структур даних. Для початку масив, в ньому ми будемо зберігати нашу матрицю. Для найшокоротшого шляху, який складається із координат (іншими словами індексу елементів матриці) ми будемо використовувати список. І оскільки у нас саме пошук в ширину нам потрібна структура даних – черга (queue), оскількі він працює за принципом послідовного пошуку, в ній також ми будемо зберігати історію проходження BFS для візуалізації.

Масив (array) - це впорядкована структура однотипних даних з фіксованою кількістю, які зберігаються послідовно в комірках оперативної пам’яті, мають порядковий номер, який начинається з нуля. Дана структура ефективна для швидкого доступу до елемента по номеру, але не ефективні при додаванні та видаленні елементів.

Зв`язний список (linked list) - це структура даних, в якій елементи лінійно впорядковані, але порядок визначається не номерами елементів (як в масивах), а посиланнями, що входять у склад елементів списку та вказують на наступний елемент.

Черга (queue) - динамічна структура даних, що працює за принципом «перший прийшов — перший пішов» (англ. FIFO — first in, first out). У черги є голова

(англ. head) та хвіст (англ. tail). Елемент, що додається до черги, опиняється в її хвості. Елемент, що видаляється з черги, знаходиться в її голові.

1.2. Вибір графічної бібліотеки

При виборі графічної бібліотеки я вирішив, що найоптимальніший варіант - Windows Forms, оскільки для візуалізації пошуку в ширину достатньо інструментарію, що надається цим інтерфейсом програмування додаткі.

Windows Forms - Це інтерфейс программування додатків (API), що відповідає за графічний інтерфейс користувача і є частиною Microsoft .NET Framework. Цей інтерфейс спрощує доступ до елементів інтерфейсу Microsoft Windows за рахунок створення обгортки для існуючого Win32 API в керованому коді. Причому керований код класи, що реалізують API для Windows Forms, не залежать від мови розробки. Тобто програміст однаково може використовувати Windows Forms як при написанні програмного забезпечення на C#, C++.

1.3.Опис алгоритму

В основі реалізації завдання лежить алгоритм пошуку в ширину.

Пóшук у ширинý — алгоритм пошуку на графі. Якщо задано граф G = (V, E) та початкову вершину s, алгоритм пошуку в ширину систематично обходить всі досяжні із s вершини. На першому кроці вершина s позначається, як пройдена, а в список додаються всі вершини, досяжні з s без відвідування проміжних вершин. На кожному наступному кроці всі поточні вершини списку відмічаються, як пройдені, а новий список формується із вершин, котрі є ще не пройденими сусідами поточних вершин списку. Для реалізації списку вершин найчастіше використовується черга та принцип FIFO (перший прийшов, перший вийшов). Виконання алгоритму продовжується до досягнення шуканої вершини або до того часу, коли на певному кроці в список не включається жодна вершина. Другий випадок означає, що всі вершини, доступні з початкової, уже відмічені, як пройдені, а шлях до цільової вершини не знайдений.

1.4. Висновок до першого розділу

В цьому розділі було обрано структури даних (масив, список, черга), графічну бібліотеку (Windows Forms) та описано головний алгоритм для реалізації ((Breadth-first Search).

Розділ 2. Проектування програми візуалізіції пошуку в ширину

2.1. Функції для реалізації

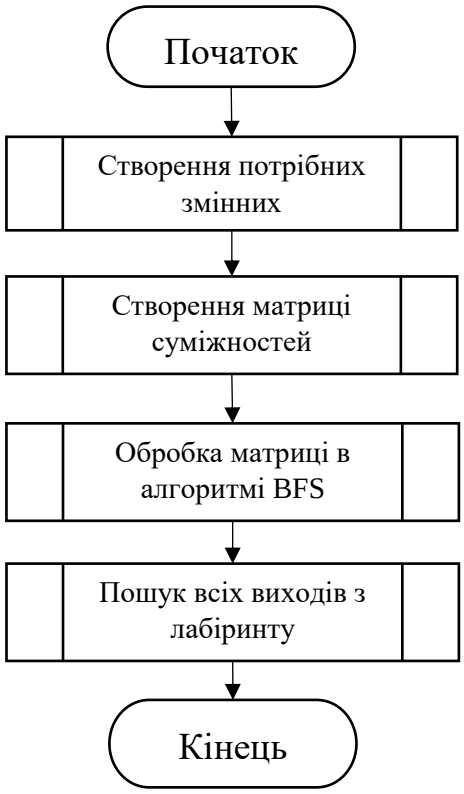
Наш програмний продукт повинен виконувати свою роботу так:
Програма читає з текстового файлу символьно-графічну версію лабіринту. Ми цей лабіринт перетворюємо в матрицю суміжностей, щоб далі працювати з цим лабіринтом в головному алгоритмі – пошук в ширину. Пошу в ширину відмітить довжину кроку, та взагалі можна чи туди йти. Після відмітки нам потрібно пройтися по краям лабіринту та виявити координати всіх виходів.

Щоб намалювати лабіринт графічно, ми повині зберегти історію кроків в черзі, для того щоб відображати його послідовно як це відбувається в BFS.
Спочатку ми малюємо рамку лабиринту, наші стіни, вони позначені в матриці кроком -1, це означає, що там немає проходу. Ми малюємо кроки послідновно, відмічаючи найкоротший шлях червоним кольором, а всі інші – чорним. Коли ми дісталися виходу ми закінчуємо малювання.

2.2. Блок схема алгоритму візуалізації пошуку в ширину

Отже, вся програма складається з створення потрібних змінних, формування матриці для обробки лабіринту в алгоритмі BFS, пошуку найкоротшого шляху та відображення цього графічно.

Блок-схема програмного продукту в класі Maze (малюнок 1).



Малюнок 1. Узагальнена блок схема класу Maze

Блок-схема програмного продукту в класі Form (малюнок 2).

Малюнок 2. Узагальнена блок схема класу Form

2.3. Висновок до другого розділу

У другому розділі був узагальнено описаний функціонал, який нам потрібно реалізувати, та зроблена блок-схема до його реалізації.

Розділ 3. Реалізація візуалізації пошуку в ширину.

3.1. Реалізація завдання

Логіка нашої програми описана в файлі Form.cs. За допомогою засобів Win Forms ми обробляємо подія натискання на кнопку render, де відбувається виклик головних методів (Лістинг 1).

```
private void button1_Click(object sender, EventArgs e)
    {
```

```
        string path = "maze.txt";
        _maze = new Maze("*", ".", "S", path);
        _renderer = new WithTimerRenderer(_graphics, _maze);
        _renderer.ShowWalls();
        pictureBox1.Image = _bm;
        timer1.Start();
    }
```

1-й етап:

Програма починається з класу Maze.

Створення зммінних які ми будемо використовувати для роботи з матрицею (Лістинг 2).

```
        //координати гравця, звідки буде почато пошук(старт)
    private Point _player;
    //graph[i,j] - index це вершина
    //значення це довжина кроку від старту
    private int[,] _matrix;
    //координати виходів
    private List<Point> _exits = new List<Point>();
    //історія роботи алгоритму пошуку в ширину
    //потрібно щоб правильно відмалювати
    private Queue<Point> _BFSHistory = new Queue<Point>();
```

Лістинг 2. Змінні в класі Maze.

Далі йде створення матриці (Лістинг 3). Ми ініціалізуємо масив на кількість елементів, отримавши розміри прочитані с текстового документу. Потів в цей

масив записуємо в комірки масива “-1” – сюди не можна йти (стіна), “0” – комірки в які можна проходити.

```
private void CreateAdjacencyMatrix(string side, string freeWave, string player,
string filePath)
{
    int[] size = GetMazeSizeFromTxt(filePath);

    _matrix = new int[size[0], size[1]];

    using (StreamReader reader = new StreamReader(filePath))
    {
        int i = 0;
        string line = "";
        while ((line = reader.ReadLine()) != null)
        {
            for (int j = 0; j < line.Length; j++)
            {
                string currentCell = Convert.ToString(line[j]);
                if (currentCell == side)
                    _matrix[i, j] = -1;
                else if (currentCell == freeWave)
                    _matrix[i, j] = 0;
                if (currentCell == player)
                    _player = new Point(i, j);
            }

            i++;
        }
```

```
        }
    }
```

Лістинг 3. Метод створення матриці в класі Maze

Далі у нас йде створення екземпляра класу WithTimerRenderer, це допоміжний клас для малювання лабіринту.

В лістингу 4 ми відмічаємо граф пошуком в ширину і кладемо його в поле _markedGraph, отримуємо історію пошуку в поле _nextSteps, та координати першого найденого найкоротшого шляху в поле shortestPath.

```
private Point[] _shortestPath;
    private Queue<Point> _nextSteps;
    private int[,] _markedGraph;
    private Graphics _graphics;
    private Maze _maze;
    private Font _font = new Font("Arial", 18);
    private Brush _brushForMarkers = Brushes.Green;
    private Brush _brushForShortestPath = Brushes.Red;

    public WithTimerRenderer(Graphics graphics, Maze maze)
    {
      _graphics = graphics;
      _maze = maze;
      _markedGraph = _maze.Bfs();
      _nextSteps = _maze.GetPathesForRenderer();
      _shortestPath = _maze.GetShortestPath();
     }
```

Лістинг 4. Змінні та конструктор класу WithTimerRenderer

Метод Bfs (лістинг 5) робить ітерації поки наша черга не залишиться пустою, в одній ітерації ми находимо чотири точки навкого поточної (ліву, праву, верхню, нижню) і добавляємо їх в чергу якщо там помітка нуль.

```
public int[,] Bfs()
    {
```

```
Queue<Point> queue = new Queue<Point>();
queue.Enqueue(_player);
_BFSHistory.Enqueue(_player);
while (queue.Count != 0)
{
   Point currentNode = queue.Peek();
   queue.Dequeue();
   Point[] points = GetNearestElements(currentNode);
   for (int i = 0; i < points.Length; i++)
   {
      if (CoordinateExistence(points[i]))
      {
         if (_matrix[points[i].X, points[i].Y] == 0 && points[i] != _player)
         {
            if (IsEdgeOfTheMaze(points[i]))
            {
               _exits.Add(points[i]);
            }
            _matrix[points[i].X, points[i].Y] = _matrix[currentNode.X,
currentNode.Y] + 1;
            _BFSHistory.Enqueue(points[i]);
            queue.Enqueue(points[i]);
         }
      }
   }
}
return _matrix;}
```

Лістинг 6. Метод пошуку в ширину (клас Maze).

Після виклику пошуку в ширину ми запускаємо метод ShowWalls класу WithTimerRenderer (лістинг 7), який проходить по нашій відміченій матриці і малює стіни по тим координатам де помітка “-1”.

```
public void ShowWalls()
{
    for (int i = 0; i < _markedGraph.GetLength(0); i++)
    {
        for (int j = 0; j < _markedGraph.GetLength(1); j++)
        {
            if (_markedGraph[i, j] == -1)
                _graphics.FillRectangle(Brushes.Black, j * 50, i * 50, 50, 50);
        }
    }
}
```

Лістинг 7. Метод ShowWalls класу WithTimerRenderer

Запускаємо таймер (код таймера на лістингу 8), який кожну ітерацію викликає метод ShowPathLengths (лістинг 9), який малює позначки довжини шляху від початкової точки. Після цього в нашому таймері йде перевірка того, дісталися ми виходу чи ні, якщо так, то ми закінчуємо малювання, стираємо старий ренден малюємо заново, залишаючи лише помітки найкоротшого шляху.

```
private void timer1_Tick(object sender, EventArgs e)
{
    bool endRenderPathes = _renderer.ShowPathLengths();

    if (endRenderPathes)
    {
        pictureBox1.Image = null;
```

```
            _graphics.Clear(Color.White);
            _renderer.ShowWalls();
            _renderer.ShowShortestPath();
            pictureBox1.Image = _bm;
            timer1.Stop();
        }
        pictureBox1.Image = _bm;
    }
```

Лістинг 8. Логіка таймеру в класі Form.

```
public bool ShowPathLengths()
    {
        while (_nextSteps.Count != 0)
        {
            Point currentNode = _nextSteps.Peek();

            _nextSteps.Dequeue();
            if (IsShortestPath(currentNode))
            {
                _graphics.DrawString(
                        Convert.ToString(_markedGraph[currentNode.X,
currentNode.Y]),
                        _font, _brushForShortestPath,
                        currentNode.Y * 50, currentNode.X * 50, new StringFormat()
                       );
            }
            else
            {
```

```
            _graphics.DrawString(
                                    Convert.ToString(_markedGraph[currentNode.X,
             currentNode.Y]),
                        _font, _brushForMarkers,
                        currentNode.Y * 50, currentNode.X * 50, new StringFormat()
                       );
        }
        if (IsFoundEndOfShortestPath(currentNode))
            break;

        return false;
    }

    return true;
}
```

Лістинг 9. Метод ShowPathLengths класу WithTimerRenderer.

3.2. Висновок до третього розділу

В цьому розділі ми розглянули реалізацію сновних функцій для виконання плану із другого розділу. А саме розглянули як реалізувати наш основний основний алгоритм “пошук в ширину”, так як можна було його викроститати для візуалізації на формах.

Висновки

Для досягнення мети курсової роботи було вирішено намічені завдання. Ми вибрали структури даних які будемо використовувати, обрали графічну бібліотеку та вияснили як працює пошук в ширину.

Далі ми спроектували реалізація, вияснивши які функції нам потрібно реалізувати та зробишки блок схему загального алгоритму роботи програми.

В кінці ми реалізували алгоритм пошуку в ширину, зрозуміли як він працює та як його можна використати для візуального відображення.

Список інформаційних джерел

1. Microsoft Visual Studio - https://uk.wikipedia.org/wiki/Microsoft_Visual_Studio
2. Пошук у ширину - https://uk.wikipedia.org/wiki/Пошук_у_ширину
3. Граф - https://uk.wikipedia.org/wiki/Граф_(математика)
4. Черга - https://uk.wikipedia.org/wiki/Черга_(структура_даних)