Черкаський національний університет імені Богдана Хмельницького
Кафедра програмного забезпечення автоматизованих систем

КУРСОВА РОБОТА

з дисципліни «Програмування та алгоритмічні мови» НА ТЕМУ «Генератор лабіринтів»

Студента 2 курсу групи КС-211
Галузі знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення (програмна інженерія)»
Ягудіна К.О.

Керівник: викладач кафедри ПЗАС
Порубльов І.М

Національна шкала: _____

Кількість балів:___Оцінка: ESTC_____

Члени комісії _____ _____
(підпис) (прізвище та ініціали)
_____ _____
(підпис) (прізвище та ініціали)
_____ _____
(підпис) (прізвище та ініціали

м. Черкаси – 2022

Зміст

Вступ

Лабіринти, відома майже кожному, ще з дитинства структура, яка зазвичай має прямокутну форму і поділена на квадрати, між деякими сусідніми квадратами відсутня стінка. Тривіальним використанням лабіринту є гра, мета якої знайти шлях від однієї точки до іншої. Але на цьому їх використання не обмежується, на основі лабіринтів можуть будуватися цілі ігрові локації, наприклад печери, болота, проектуватися «живі» лабіринти, у програмуванні лабіринти використовуються для візуалізації та аналізу алгоритмів, найчастіше пов’язаних із деревами.

Лабіринти мають дуже широку класифікацію [6], але для нашої роботи доречно виділити теселяцію і маршрутизацію. Теселяція - класифікація геометричної сітки лабіринту, у нашому завданні ми будемо розглядати звичну нам ортогональну (прямокутна сітка) таселяцію, але вона може бути, як конкретною фігурою, наприклад дельта (трикутник), сігма (шестикутник), тета (коло), так і аморфною crack-лабіринт. Маршрутизація – класифікація, яка пов’язана з генерацією проходів у лабіринті. Виділяють ідеальні - лабіринти без недосяжних областей, внутрішніх циклів, одномаршрутні - лабіринти без розгалужень, плетені - лабіринти без тупиків.

Мета курсової роботи – підбір та реалізація алгоритму для генерації ідеального ортогонального лабіринту, зображення лабіринту на екрані та збереження лабіринту у текстовий документ з подальшої можливістю відкриття цього лабіринту у програмі. Користувачу надається набір із елементів керування, пов’язані з характеристиками лабіринту, візуальним відображенням роботи алгоритму та збереженням лабіринту у текстовий файл.

Задачами роботи є:

1. Проаналізувати інтернет джерела, дослідити та обрати алгоритми побудови ідеальних лабіринтів.
2. Спроектувати програму, побудувати блок схему обраного алгоритму.

3. Скласти структурну схему програмної частини для виконання всіх вимог завдання.
4. Розробити та реалізувати програмний продукт включно з користувальницьким інтерфейсом.
5. Зробити висновки.

Для розуміння роботи введемо поняття:

1. Клітинка – найменша складова лабіринту квадратної форми, кожна клітинка має 4 стінки, якими вона відділена від сусідніх клітинок.
2. Стінка – горизонтальний або вертикальний відрізок, який розділяє дві сусідні клітинки.

Розділ 1. Дослідження та вибір алгоритмів для генерації ідеального лабіринту та його перевірки.

1.1 Алгоритм генерації ідеального лабіринту.

Існує велика кількість алгоритмів для генерації ідеальних лабіринтів, найпопулярнішими прикладами таких алгоритмів є recursive backtracker, алгоритм Краскала, алгоритм Пріма та алгоритм Олдоса-Бродера [2], кожен із яких має свій унікальний підхід до генерації лабіринтів.

Для свого проекту я обрав алгоритм Краскала. Алгоритм Краскала – це алгоритм який створює мінімальне кістякове дерево (MST) у зваженому графі [3], для генерації лабіринту я буду використовувати рандомізовану [1, ст. 168] версію цього алгоритму. Стандартна реалізація цього алгоритм працює так:

- *Створити ліс (масив з дерев) F, у якому кожна вершина графа є окремим деревом.*
- *Створити відсортований за вагою ребер масив S з усіма ребрами графа.*
- *Поки S не пустий і F ще не кістяковий (spanning).*
    - *Убираємо ребро з найменшою вагою з масиву S.*
    - *Якщо убране ребро об'єднує два різні дерева, зливаємо два дерево у одне.*

Псевдокод імплементації алгоритму з використанням disjoint-set структури даних:

```
algorithm Kruskal(G) is
 F:= ∅
 for each v ∈ G.V do
        MAKE-SET(v)
 for each (u, v) in G.E ordered by weight (u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
               F:= F ∪ {(u, v)} ∪ {(v, u)}
               UNION(FIND-SET(u), FIND-SET(v))
 return F
```

Радномізована версія буде відрізнятися другим кроком у алгоритмі, тепер масив ребер буде відсортований не за вагою, а у випадковому порядку. Якщо вибране випадковим чином ребро об'єднує два різні дерева, які нагадую є вершинами, а у випадку лабіринту клітинками, зливаємо ці дерева в одне і знищуємо стінку між клітинками. У випадку коли ребро об'єднує два дерева, які вже злиті в одне дерево, нічого не змінюємо, це означає, що між цими клітинками лабіринту вже є шлях [2, с1.].

Псевдокод імплементації рондомізованого алгоритму з використанням disjoint-set структури даних:

```
algorithm RandomizedKruskal(G) is
 for each v ∈ G.V do
        MAKE-SET(v)
 for each (u, v) in G.E ordered by random do
        if FIND-SET(u) ≠ FIND-SET(v) then
               UNION(FIND-SET(u), FIND-SET(v))
               BREAK-WALL(u, v)
 return void
```

Приклад роботи алгоритму:

Для прикладу скористуємося простою сіткою 3 на 3. У кожної клітинки є літера, яка вказує, на те, якому сету вона належить.

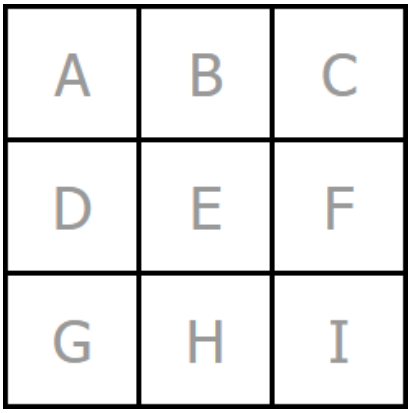


Рисунок 1.1 Початковий стан лабіринту

Виберемо випадкове ребро, нехай це буде ребро між вершинами (2, 2) та (2, 3). Вершини знаходяться у різних сетах, тому об’єднуємо ці два сети і руйнуємо стінку між цими вершинами.

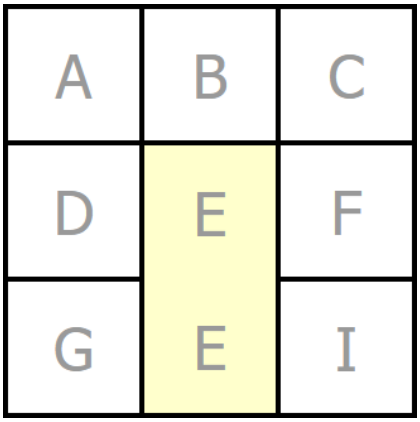
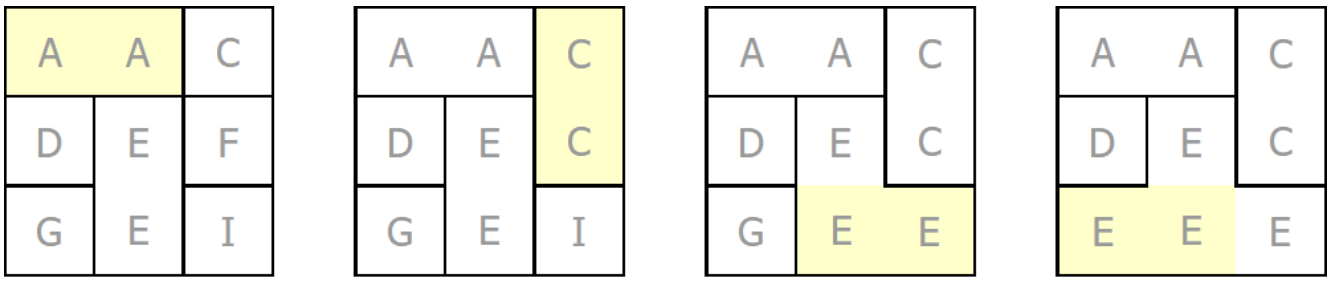


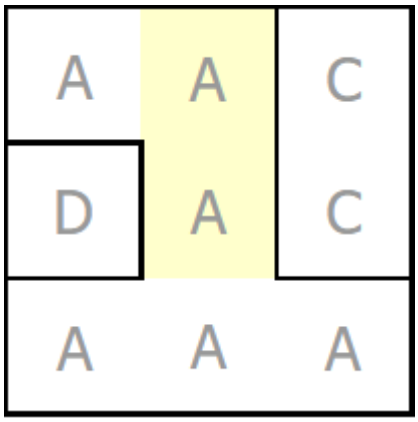
Рисунок 1.2 Стан лабіринту після об’єднання вершин

Виконаємо цю операцію ще декілька разів, для того щоб показати, як відбувається зливання дерев.



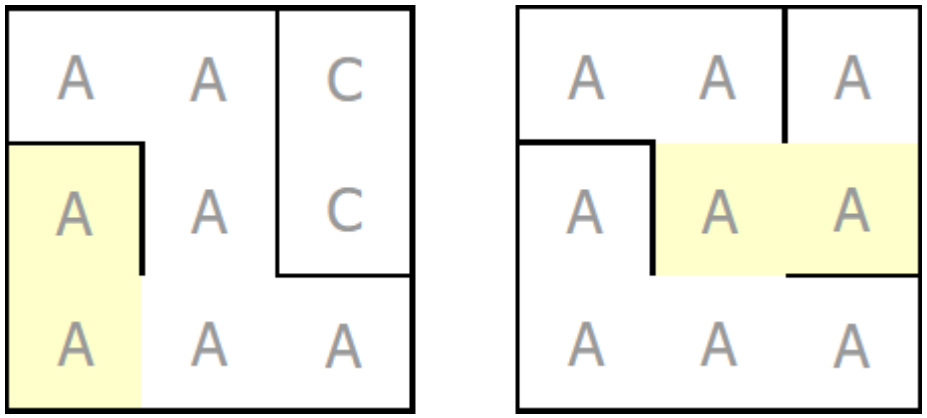
1.3 Стани лабіринтів після декількох послідовних ходів

Виберемо ребро між вершинами (1, 2) та (2, 2). Два дерева A та E, об'єднуються в один сет A, це означає, що з будь-якої клітинки у сеті A можна потрапити до будь-якої іншої клітинки у сеті A.



1.4 Стан лабіринту після злиття двох дерев.

Закінчуємо генерацію лабіринту генерацію лабіринту аналогічними кроками.

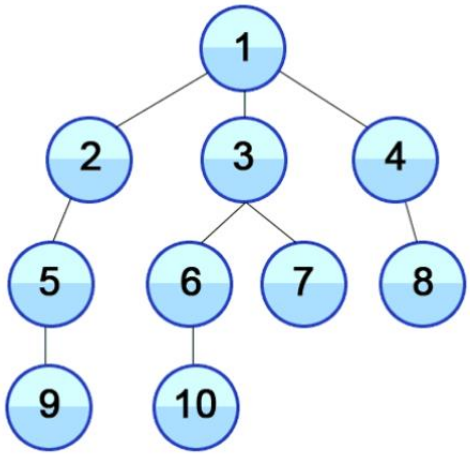


1.5 Стан лабіринту після завершення виконання алгоритму

Алгоритм завершує свою роботу, після того, як перебирає всі ребра, таким чином ми отримаємо один результуючий сет і ідеальний лабіринт.

1.2 Алгоритм перевірки ідеального лабіринту.

Для того, щоб перевірити правильність генерації лабіринту, особливо великого розміру, потрібен алгоритм, який буде шукати шлях між двома обраними клітинками. Я обрав алгоритм пошуку в ширину (BFS), цей алгоритм призначений для пошуку вузла (node), який задовольняє вимогу, у дерево-видній структурі даних. Основна ідея алгоритм полягає в тому, щоб на кожному рівні глибини перевірити всі вузли (nodes), а потім перейти на наступний рівень у глибину. [5]



1.5 Приклад послідовності обходу вузлів BFS

Псевдокод реалізації алгоритму пошуку в ширину для графа, з збереженням найкоротшого шляху.

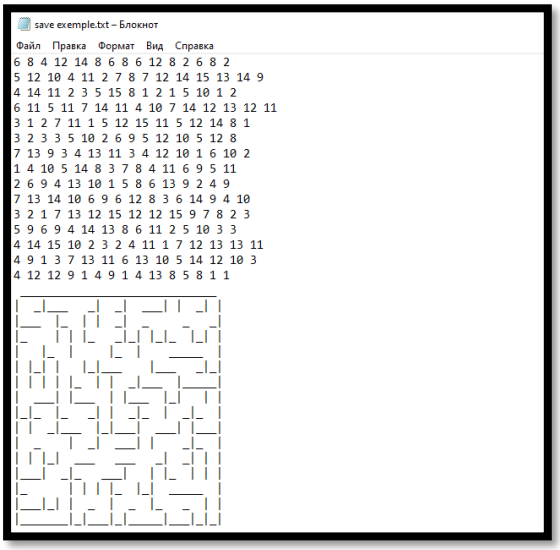
```
procedure BFS(G, root) is
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
           if w is not labeled as explored then
                label w as explored
                w.parent := v
                Q.enqueue(w)
```

За допомогою модифікацію цього алгоритму, яка надає можливість обирати початкову клітинку і клітинку, яку треба знайти, буде можливо перевірити лабіринт, на наявність шляху між двома клітинками.

1.3 Алгоритм збереження ідеального лабіринту.

Ціль завдання зберегти лабіринт у текстовому файлі, у форматі зручному, як для програми, так і для користувача. Було вирішено розбити цю задачу на дві окремі, як зберегти зручно для користувача і як для програми. Найпростіше було вирішити питання зберігання у зручному форматі для програми, для цього було прийнято рішення зберігати масив розміру n^2, де $n -$ довжина сторони лабіринту з у кожній комірці масиву буде зберігатися статус відповідної клітинки лабіринту (детальніше про статус клітинки у «Розділі 3»). Але це не дуже комфортно для користувача, тому додатково я буду вимальовувати лабіринт, за у текстовому файлі за допомогою символів з таблиці ASCII.

Приклад збереження лабіринту з майбутньої реалізації.



1.6 Приклад збереження лабіринту

Розділ 2. Проектування користувацького інтерфейсу та роботи алгоритмів.

2.1 Користувацький інтерфейс програми.

Програмний продукт має мати графічний інтерфейс для взаємодії з користувачем, елементи керування виконанням алгоритму та працювати з файловою системою для збереження та відкриття файлів.

У продукті мають бути наявні такі елементи:

1) Полотно для вимальовування лабіринту.
2) Поля для вводу розміру лабіринту та регулювання швидкості анімації.
3) Поле для вводу кількості стінок, для додаткового видалення, після виконання основного лабіринту.
4) Кнопки керування виконанням алгоритму (“Start”, “Stop”, “Finish”, “Next”, “Reset”).
5) Кнопка для переходу у режим BFS.
6) Поле для вводу назви файлу, для збереження лабіринту і кнопка “Save”.

7) Випадаючий список з файлів, для відкриття у програмі в вигляді лабіринту та кнопка “Open”.

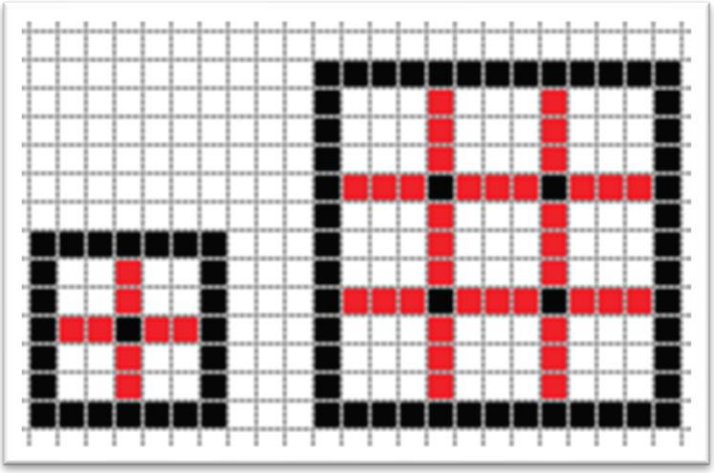
2.2 Проектування роботи алгоритмів.

Для облегшення виконання переключення між режимами генерації лабіринту та BFS, а також майбутнього розширення програми, я обрав трохи видозмінений паттерн «стратегія». Замість спільного інтерфейсу буде використовуватися наслідування. Ми будемо мати клас Maze, який буде мати загальні утіліті методи та віртуальні методи для візуалізації роботи алгоритмів, та його нащадків Kruskal та BFS, які будуть мати методи для виконання алгоритму та перевизначать логіку методів анімації.

Дефолтним значенням розміру лабіринту буде 10, кількості додаткових стінок для видалення 0, а інтервал таймера 1000 ms. При введенні нових даних у поля розміру лабіринту, та кількості стінок програма буде зупинятися та ініціалізувати новий лабіринт з отриманих вхідних даних. Генерації лабіринту буде винесена в окремий метод. Логіка дій при натискання на кнопки куревання, буде оброблятися за допомогою публічних полів-перемикачів (public bool isFinished і т.п.), які знаходяться у батьківському класі Maze, наприклад натискання на кнопку “Save” і “BFS”, буде можливим тільки тоді, коли поле-перемикач isFinished матиме значення true. Зберігання файлів буде відбуватися в спеціальній папці у директорії проекту, задля того, щоб при перенесенні програми на інший комп’ютер, зменшити ризик IOException(-ів), з цієї папки також будуть автоматично підтягуватися файли для спроби відкрити у вигляді лабіринту. Вимальовування лабіринту на полотні буде відбуватися так, спочатку ми малюємо пустий прямокутник з усіма стінками, і потім для кожних клітинки визиваємо метод, який буде малювати її відповідно до її статусу. Зміна швидкості анімації буде відбуватися динамічно. Програма написана для багаторазового використання функціоналу під час однієї сесії і не крашиться без безпосереднього втручання користувача у файлову систему або створення виключних ситуацій під час динамічної зміни точок BFS (детальніше у Розділі 3).

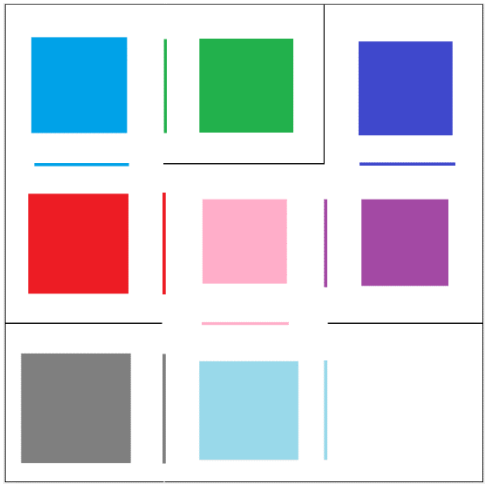
2.3 Максимальна кількість додатково видалених стінок.

Ця тема була спеціально винесена в окремий розділ, щоб представити математичне доведення, що після виконання алгоритму Краскала на лабіринті розміру n на n максимальна кількість додатково видалених стінок дорівнює $(n-1)^2$. Для початку виведемо формулу для визначення кількості внутрішніх стінок у лабіринті розміру n на n. Для прикладу візьмемо лабіринт розміру 3 на 3, та 2 на 2.



2.1 Лабіринти 2 на 2 та 3 на 3

Бачимо, що в обох випадках ми маємо $n-1$ вертикальних і горизонтальних ліній, кожна з яких має n стінок, ця пропорція зберігається на всіх лабіринтах у яких довжина дорівнює висоті (тобто вони є квадратами). Знаючи це ми можемо порахувати загальну кількість внутрішніх стінок $N_{\text{заг}} = n * (n-1) + n * (n-1) = 2\big(n * (n-1)\big) = 2(n^2 - n)$. Тепер повернемося до нашого алгоритму, він генерує ідеальний лабіринт, це означає, що між будь-якими двома клітинками є рівно один шлях, математично це означає, що у клітинки буде видалена рівно 1 «своя» стінка, крім останньої клітинки, на якій закінчується виконання алгоритму, я написав саме «своя» стінка, бо у клітинки можуть бути відсутні декілька стінок, але вони були видалені, як «свої» стінки сусідніх клітинок.

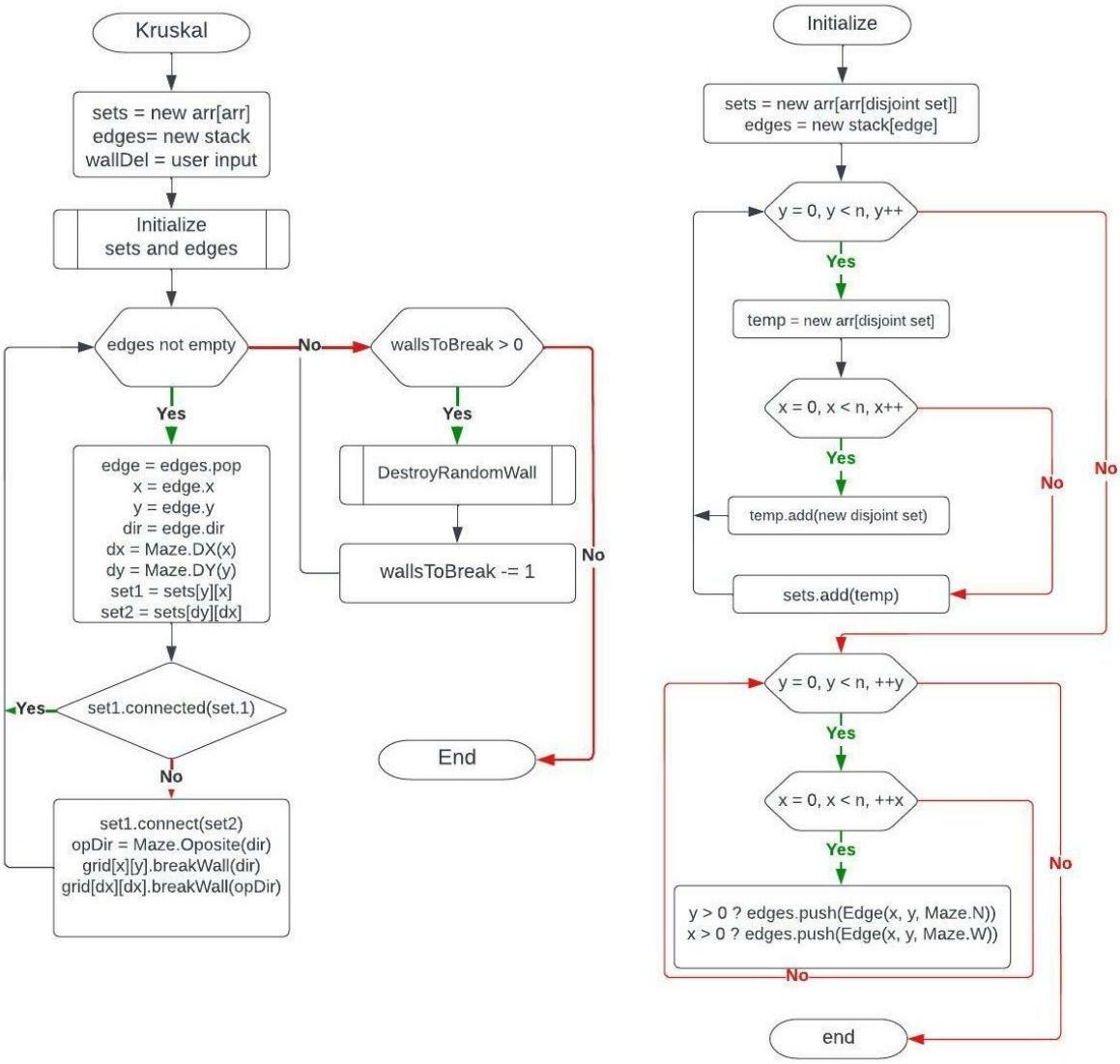


2.2 Приклад видалення стінок, кольорові стінки видалені

Це означає, що алгоритм видалить $N_{\text{вид}} = n^2 - 1$ стінок, тепер знаючи скільки загалом стінок у лабіринті і скільки видалить алгоритм ми можемо порахувати, скільки стінок залишиться після генерації. $N_{\text{зал}} = N_{\text{заг}} - N_{\text{вид}} = 2(n^2 - n) - (n^2 - 1) = 2n^2 - 2n - (n^2 - 1) = n^2 - 2n + 1 = (n - 1)^2$. Таким чином ми доводимо, що кількість стінок, які залишаться після виконання алгоритму і можуть бути видалені дорівнює $(n - 1)^2$.

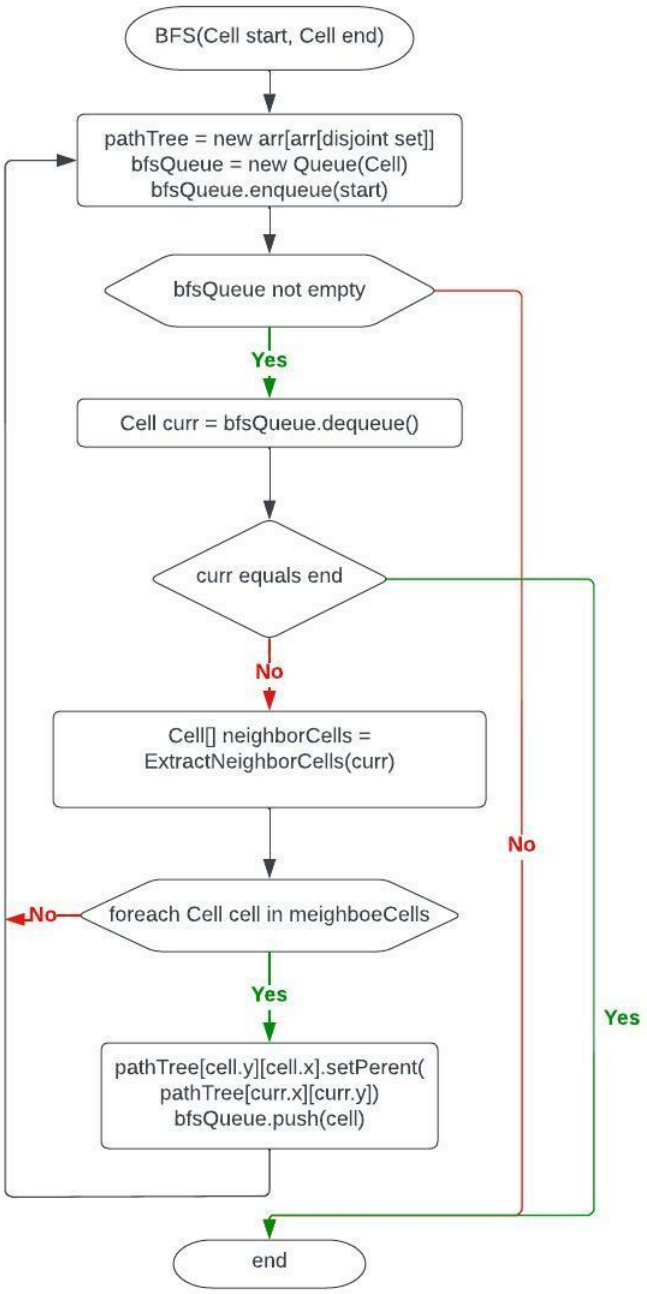
2.3 Блок-схема роботи алгоритмів.

Модифікований алгоритм Краскала для генерації лабіринту:



2.3 Блок схема роботи модифікованого алгоритму Краскала

Блок схема роботи модифікованого для лабіринтів BFS з відновленням шляху.

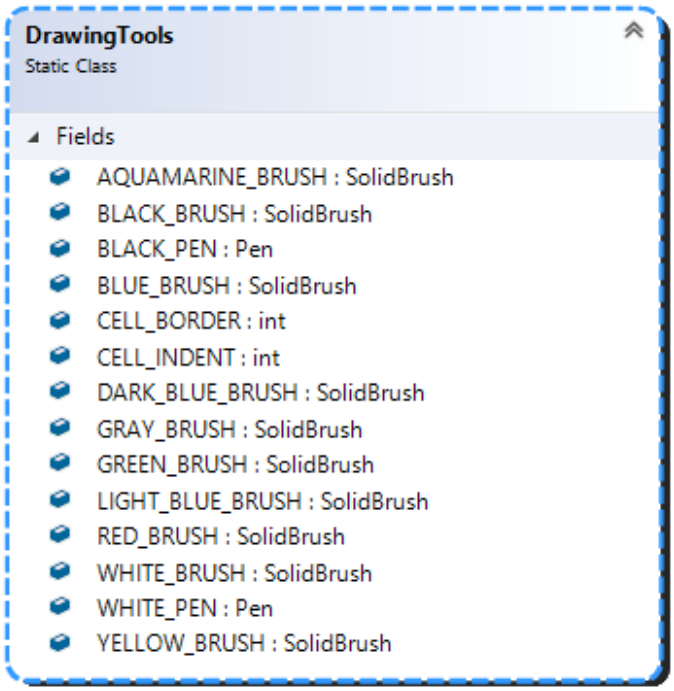


2.4 Блок схема роботи модифікованого алгоритму BFS

Розділ 3. Реалізація програмного продукту.

3.1 Клас DrawingTools.

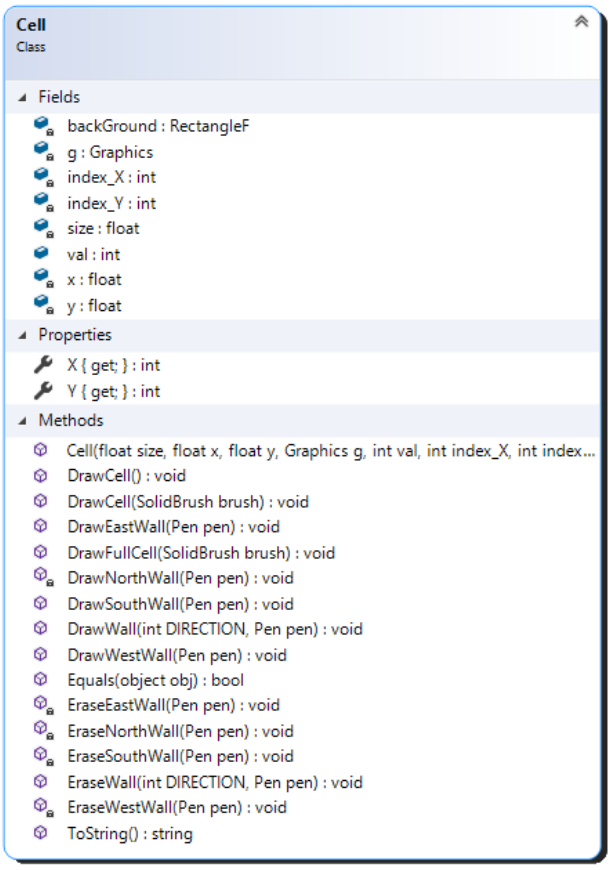
Так, як при анімації алгоритмів нам прийдеться часто звертатися до класів пов'язаним з малюванням (`Pen, SolidBrush and etc.`), було прийнято рішення створити і винести ці всі утіліті інструменти в окремий клас (`Додаток А`).



3.1 утіліті клас `DrawingTools`

3.2 Клас Cell.

Почнемо з найменшої частини реалізації, клітинки. У попередніх розділах я згадував «статус» клітинки, цю тему важко пояснити без згадування класу Maze, і клас Maze без класу Cell, тому вони будуть іти послідовно. Почнемо з ознайомлення з UML схемою.



3.2 UML схема класу Cell

Почнемо з поля `public int val`, це поле і є тим самим «статусом» клітинки, воно буде зберігати в собі інформацію про наявність у клітинки стінок, і інші властивості, які з'являться при реалізації алгоритмів. Принцип роботи полягає в двійковому представленні числа, для пояснення візьмемо перші чотири біти числа, кожен біт відповідає своєму числовому значенні $0_8 0_4 0_2 0_1$, саме сюди ми будемо записувати наші властивості. Для того щоб записування працювало правильно, наші властивості будуть мати значення різних степенів 2, наприклад зараз «статус» нашої клітинки пустий і дорівнює 0, $0_{10} = 0000_2$, ми хочемо записати у статус клітинки властивість `VISITED`, яка буде записуватися у перший біт «статусу», перший біт двійкового числа дорівнює $0001_2 = 1_{10}$, ініціалізуємо властивість `VISITED` із значенням 1, `public conts int VISITED = 1`. Тепер для того, щоб додати цю властивість виконуємо операцію побітового «або» `val = val | VISITED`, результат отримаємо 0001_2, тепер припустимо що ми маємо ще одну властивість, яка буде записуватися у третій біт числа $0100_2 = 4_{10}$, `public conts

`int CHECKED = 4`, щоб додати властивість ми знов виконуємо операцію побітового «або» `val = val | VISITED`, результатом буде $0101_2 = 5_{10}$, для перевірки «статусу» клітинки на наявність властивості використовуємо операцію побітового «і», якщо результат відмінний від 0, «статус» має цю властивість `val & VISITED`, результат $0001_2 = 1_{10}$ властивість присутня, для видалення властивості використовуємо звичайну операцію віднімання, наприклад ми хочемо видалити властивість `CHECKED`, використовуємо оператор віднімання `val = val - CHECKED`, результатом буде $0001_2 = 1_{10}$, тепер перевірка на наявність властивості `CHECKED` буде дорівнювати 0, це означає, що властивість відсутня `val & VISITED`, результат $0000_2 = 10_2$. Поля `backGround, g, x, y` використовуються в методах для вимальовування клітинки (Draw\Erase), `x i y` координати верхнього лівого кута, `size` довжина сторони. Поля index_X і `index_Y` зберігають індекси клітинки у лабіринті. У клітинки наявна велика кількість методів для вимальовування, але більшість із них приватна і викликаються через два загальних метода `EraseWall` і `DrawWall`.

```
public void DrawWall(int DIRECTION, Pen pen)
{
    switch (DIRECTION)
    {
        case Maze.S:
            DrawSouthWall(pen);
            return;
        case Maze.N:
            DrawNorthWall(pen);
            return;
        case Maze.W:
            DrawWestWall(pen);
            return;
        case Maze.E:
            DrawEastWall(pen);
            return;
    }
}
```

```
public void EraseWall(int DIRECTION, Pen pen)
{
    switch (DIRECTION)
    {
        case Maze.S:
            EraseSouthWall(pen);
            return;
        case Maze.N:
            EraseNorthWall(pen);
            return;
        case Maze.W:
            EraseWestWall(pen);
            return;
        case Maze.E:
            EraseEastWall(pen);
            return;
    }
}
```

3.3 Методи `EraseWall` та `DrawWall`

Координати та індекси клітинці приходять через конструктор, розрахунок формул для вимальовування елементів клітинки знаходяться у відповідних методах.

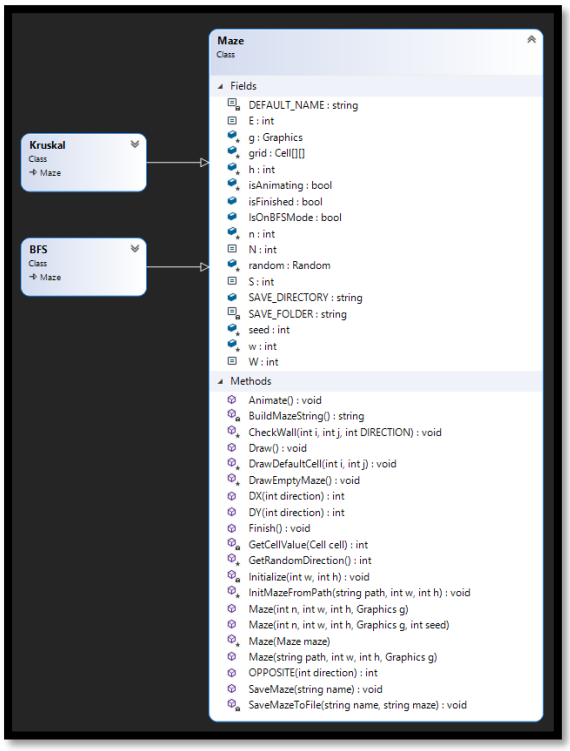
```
/// <summary> Draw West wall of the Cell using given pen
1 reference
public void DrawWestWall(Pen pen)
{
    g.DrawLine(pen, x, y, x, y + size);
}

/// <summary> Erase North wall of the Cell using given pen
1 reference
private void EraseNorthWall(Pen pen)
{
    g.DrawLine(pen, x + CELL_INDENT, y, x + size - CELL_INDENT, y);
}
```

3.4 Приклад методів для вимальовування елементів клітинки

3.3 Батьківський клас Maze.

Батьківський клас Maze представляє обертку для реалізації класів Kruskal та BFS, він містить утіліті поля та методи, а також віртуальні методи реалізації анімації. Почнемо ознайомлення з UML схеми.



3.5 UML схема класу Maze

Поля `E, N, S, W` позначають напрямок і є властивостями клітинок, які вказують на стінки, які треба видалити.

```
public const int N = 1;
public const int S = 2;
public const int E = 4;
public const int W = 8;
```

3.6 Константи, які позначають напрямок

DX, DY, OPPOSITE є утільними switch\case методами. DX та DY повертають значення, на яке потрібно зробити зсув по осям за заданим напрямком, OPPOSITE повертає протилежний напрямок. Наприклад метод DX.

```
public static int DX(int direction)
{
            switch (direction)
            {
                case Maze.E:
                    return +1;
                case Maze.W:
                    return -1;
                case Maze.N:
                case Maze.S:
                    return 0;
            }
            return -1;
}
```

Двох вимірний масив grid і є нашим лабіринтом, він ініціалізується під час виклику метода Initialize. Параметри w, h, n передаються через конструктор.

```
private void Initialize(int w, int h)
        {
            this.w = w;
            this.h = h;

            grid = new Cell[n][];

            float cellSize = (float)(w * 1.0) / n;

            for (int j = 0; j < n; j++)
            {
                grid[j] = new Cell[n];
                float y = j * cellSize;

                for (int i = 0; i < n; i++)
                {
                    float x = i * cellSize;
                    grid[j][i] = new Cell(cellSize, x, y, g, 0, i, j);
                }
            }
}
```

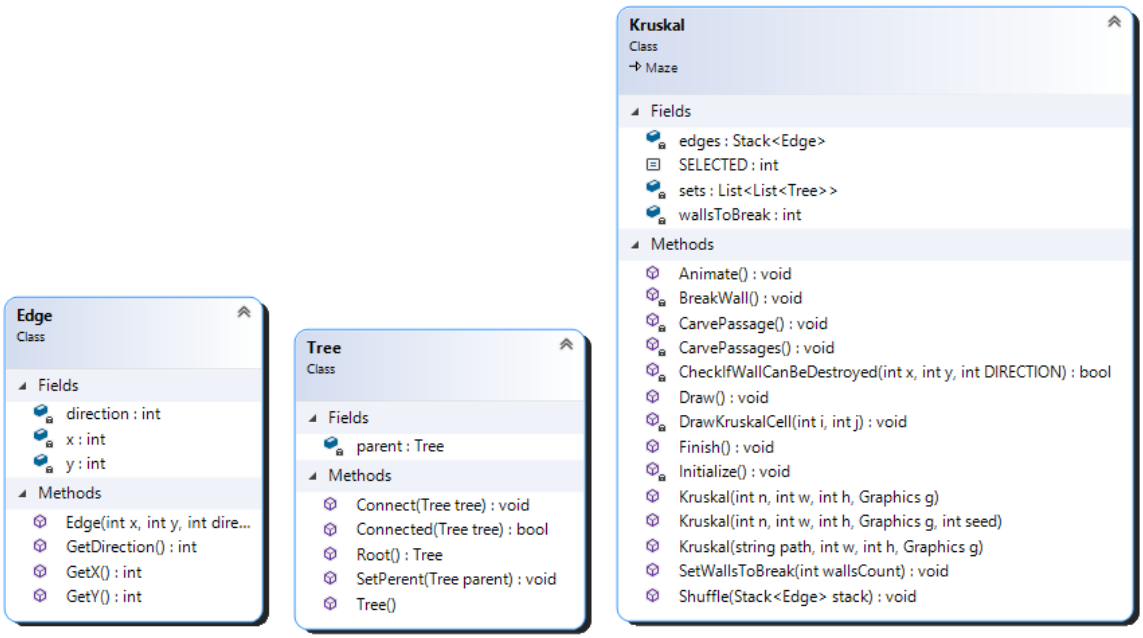
Кожній клітинці ми передаємо її розмір, координати, початковий «статус», та індекси у лабіринті. Розмір клітинки рахуємо, як загальну ширину поділити на кількість клітинок.

Клас має чотири конструктори, два для генерації лабіринту під час роботи програми (один з конструкторів передбачає передачу власного `seed` для `Random`), один для читанні з файла та конструктор копіювання.

Методи `Animate`, `Finish`, `Draw` мають модифікатор `virtual` і перевизначаються у дочірніх класах. Метод Draw (`Додаток Б`) має тіло, яке реалізує вимальовування рамки лабіринту та клітинок. Вимальовування клітинок відбувається через її «статус», де значення один у відповідному біті означає відсутність стінки, а нуль наявність.

3.4 Нащадок клас Kruskal.

Клас `Kruskal` є нащадком класу `Maze`, він перевизначає методи для анімації, та його основною особливістю є реалізація алгоритму генерації лабіринту.



3.7 UML схема класу `Kruskal` та класів які він використовує

Першої важливою частиною є поява свого метода `Initialize()`, який визивається після виконання батьківського `Initialize()`, у ньому

ініціалізуються `edges` і `sets`, а також `edges` перемішуються у випадковому порядку.

```
private void Initialize()
{
            sets = new List<List<Tree>>();
            for(int y = 0; y < n; ++y )
            {
                List<Tree> temp = new List<Tree>();
                for(int x = 0; x < n; ++x)
                {
                    temp.Add(new Tree());
                }
                sets.Add(temp);
            }

            edges = new Stack<Edge>();
            for (int y = 0; y < n; ++y)
            {
                for (int x = 0; x < n; ++x)
                {
                    if (y > 0) { edges.Push(new Edge(x, y, Maze.N)); }
                    if (x > 0) { edges.Push(new Edge(x, y, Maze.W)); }
                }
            }
            Shuffle(edges);
}
```

З'являється метод `CarvePassage (Додаток В)` для покрокової анімації лабіринту, а також його версія для виконання всього алгоритму одразу закінчення генерації `CarvePassages` їх відмінністю є те, що деякі `if` у `CarvePassage` замінені на `while` `CarvePassages` із-за чого, він одразу виконує алгоритм до кінця, а не один його крок.

Бачимо, що у самому класі з'явилася нова властивість для клітинки `SELECTED` це потрібно для анімації алгоритму і використовується у методі `DrawKruskalCell.`

```
private void DrawKruskalCell(int i, int j)
{
            if (grid[j][i].val == 0)
            {
                grid[j][i].DrawCell();
            }
            else if ((grid[j][i].val & SELECTED) != 0)
            {
                grid[j][i].DrawCell(YELLOW_BRUSH);
                grid[j][i].val -= SELECTED;

            }
            else
            {
                grid[j][i].DrawCell(WHITE_BRUSH);
            }
}
```

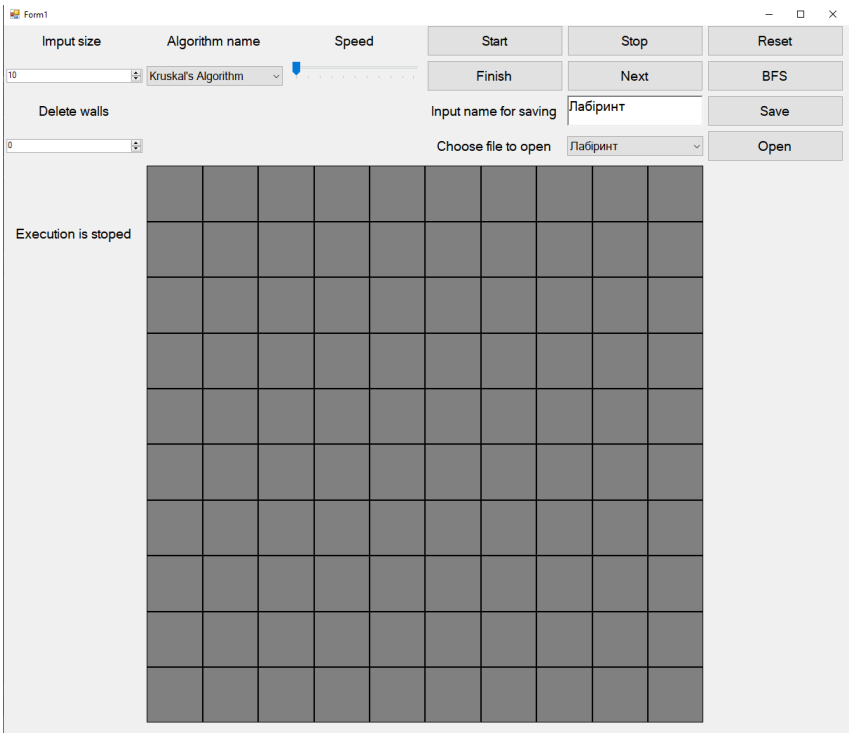
Клас перевизначає методи, для того щоб взаємодіяти з користувачем.

```
public override void Animate()
{
        CarvePassage();
        Draw();
}

 public override void Finish()
{
        CarvePassages();
        Draw();
}
```

3.5 Взаємодія з користувачем

Користувачу надається доступ до візуального інтерфейсу.

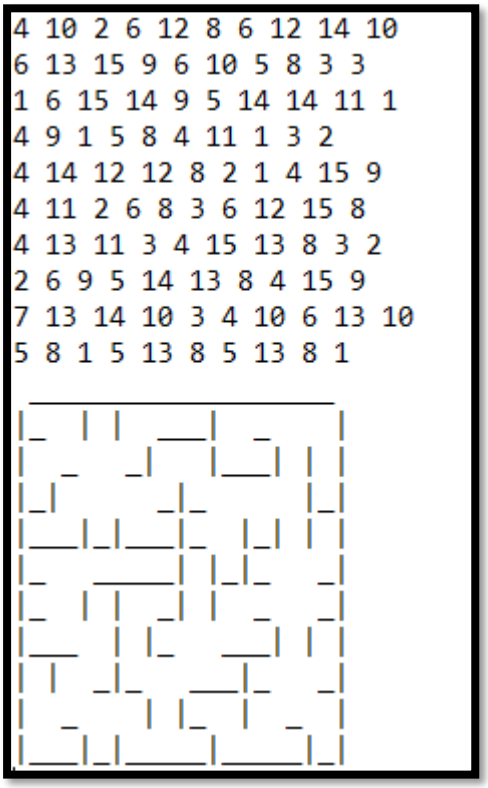


3.8 Візуальний інтерфейс програми

Інтерфейс простий і зрозумілий інтуїтивно, для всіх елементів керування написані event listeners, з логікою, яка забезпечує, те що програма не зламається під час виконання. Після завершення генерації лабіринту, відкриваються функції збереження та BFS, у режимі BFS натисканням лівої кнопки на клітинку лабіринту змінюється позиція старту, натисканням правої, позиція клітинки, яку треба знайти. Зберігання відбувається у наперед визначену папку, `maze saves`, яка знаходиться у корневій директорії проекту, з неї ж і підтягуються файли для відкриття у програмі.

Висновок

Для реалізації цієї програми були вивчені та використані методи генерації ідеальних лабіринтів, алгоритми пошуку у лабіринтах, структури даних «дерева» та «disjoint set». Була проведена робота з графічним інтерфейсом додання та обробка незвичайних “`Click Action Listeners`” для вибору клітинки в режимі BFS. Лабіринт зберігається у форматі, одночасно зрозумілому і користувачу і програмі.



Цю програма може легко розширюватися, все що потрібно зробити для додання нового алгоритму, це унаслідувати клас Maze та перевизначити методи, які відповідаю за анімацію, та вимальовування лабіринту. Таким чином ця програма може стати чудовим навчальним інструментом для візуалізації роботи алгоритмів генерації лабіринтів, а при деяких модифікаціях іще й алгоритмів пошуку. Саме цим я планую займатися у вільний час, і зробити так, щоб у ній можна було подивитися роботу всіх алгоритмів, для генерації ідеального лабіринту.

Список літератури та інтернет джерел

1. Mazes for Programmers Jamis Buck August 4, 2005 [Текст] розділ 10 ст. 168.
2. Maze generation: Algorithm Recap [електронний документ] https://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap .
3. Kruskal's algorithm [електронний документ] https://en.wikipedia.org/wiki/Kruskal%27s_algorithm.
4. Maze Generation: Kruskal's Algorithm [електронний документ] https://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm.
5. Breadth-first search [електронний документ] https://en.wikipedia.org/wiki/Breadth-first_search.
6. Класифікація, генерація, пошук рішень лабіринтів [електронний документ] https://habr.com/ru/post/445378/ .

Додаток А. Клас `DrawingTools`

```
class DrawingTools
{
    public static readonly int CELL_BORDER = 2;
    public static readonly int CELL_INDENT = CELL_BORDER / 2;
    public static readonly Pen WHITE_PEN = new Pen(Color.White, CELL_BORDER);
    public static readonly Pen BLACK_PEN = new Pen(Color.Black, CELL_BORDER);
    public static readonly SolidBrush BLACK_BRUSH = new SolidBrush(Color.Black);
    public static readonly SolidBrush WHITE_BRUSH = new SolidBrush(Color.White);
    public static readonly SolidBrush GRAY_BRUSH = new SolidBrush(Color.Gray);
    public static readonly SolidBrush RED_BRUSH = new SolidBrush(Color.Red);
    public static readonly SolidBrush YELLOW_BRUSH = new SolidBrush(Color.Yellow);
    public static readonly SolidBrush BLUE_BRUSH = new SolidBrush(Color.Blue);
    public static readonly SolidBrush LIGHT_BLUE_BRUSH = new SolidBrush(Color.LightBlue);
    public static readonly SolidBrush DARK_BLUE_BRUSH = new SolidBrush(Color.DarkBlue);
    public static readonly SolidBrush GREEN_BRUSH = new SolidBrush(Color.Green);
    public static readonly SolidBrush AQUAMARINE_BRUSH = new SolidBrush(Color.Aquamarine);
}
```

Додаток Б. Метод Draw

```
public virtual void Draw()
{
            DrawEmptyMaze();

            for (int j = 0; j < n; j++)
            {
                for (int i = 0; i < n; i++)
                {
                    DrawDefaultCell(i, j);

                    CheckWall(i, j, N);

                    CheckWall(i, j, E);

                    CheckWall(i, j, S);

                    CheckWall(i, j, W);
                }
            }
}

protected void CheckWall(int i, int j, int DIRECTION)
{
         if ((grid[j][i].val & DIRECTION) != 0)
         {
             grid[j][i].EraseWall(DIRECTION, WHITE_PEN);
         }
         else
         {
             grid[j][i].DrawWall(DIRECTION, BLACK_PEN);
         }
}
```

Додаток В. Метод CarvePassage

```
private void CarvePassage()
{
            if(edges.Count > 0)
            {
                Edge temp = edges.Pop();

                int x = temp.GetX();
                int y = temp.GetY();
                int direction = temp.GetDirection();

                int dx = x + Maze.DX(direction);
                int dy = y + Maze.DY(direction);

                Tree set1 = sets[y][x];
                Tree set2 = sets[dy][dx];

                if (!set1.Connected(set2))
                {
                    set1.Connect(set2);

                    //Destroy wall
                    grid[y][x].val |= direction;
                    grid[dy][dx].val |= Maze.OPPOSITE(direction);

                    //Color cells for visualization
                    grid[y][x].val |= SELECTED;
                    grid[dy][dx].val |= SELECTED;
                }
            }
            else if (wallsToBreak > 0)
            {
                BreakWall();
            }
            else
            {
                isFinished = true;
            }
}
```