Algorytmy i Struktury Danych II
Projekt – Generowanie Labiryntu

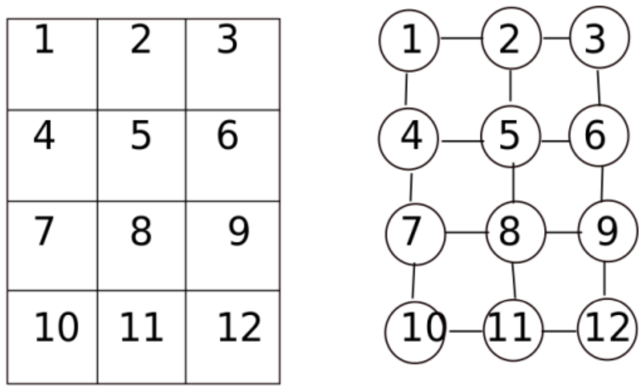
Jakub Ostrowski

Grupa 5

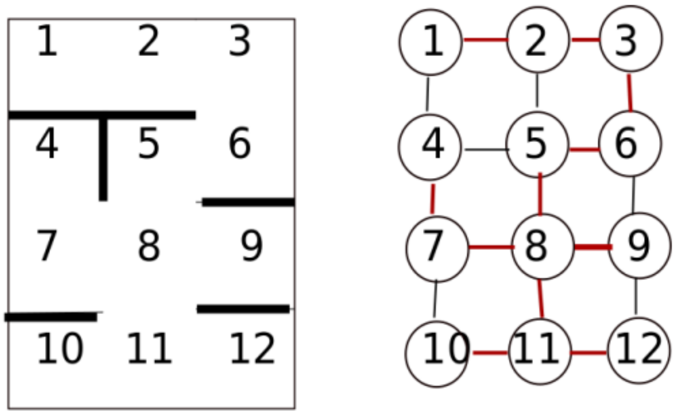
27.05.2022

1. Opis problemu

- Wykorzystać algorytm Kruskala do generowania labiryntu
- Na wejściu dostajemy rozmiar labiryntu, np. n*m (prostokąt złożony z siatki kwadratowej o n wierszach i m kolumnach)
- Generujemy graf reprezentujący „przyleganie” komórek siatki, np. dla n=4 i m=3 będzie to:



- Graf nie posiada wag
- Na danym grafie wykonujemy zmodyfikowany algorytm Kruskala znajdujący drzewo rozpinające graf, wybierając w kolejnych krokach losowe krawędzie
- Krawędzie między wierzchołkami wybrane do drzewa rozpinającego graf reprezentują „brak ścian” między komórkami reprezentującymi te wierzchołki. Pozostałe krawędzie grafu reprezentują ściany.



- Wykonać wizualizację labiryntu, np. w trybie tekstowym

2. Opis rozwiązania (pseudokod)

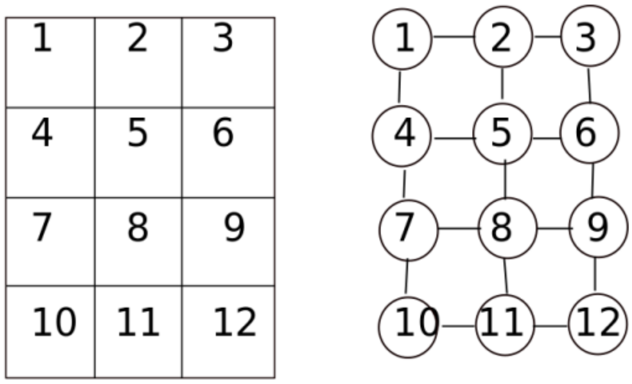
- Interpretacja zadania

a) Połączenia pomiędzy poszczególnymi wierzchołkami są rozumiane jako „brak ściany” w labiryncie
b) Brak połączenia pomiędzy poszczególnymi wierzchołkami są rozumiane jako „ściany” w labiryncie
c) Siatka połączeń odpowiada krawędziom łączącym n*m wierzchołków w grafie, które stworzą siatkę n*m, gdzie n i m to odpowiednio wysokość i szerokość
d) Łączna ilość połączeń w siatce wynosi **n*(m-1)+(n-1)*m**, ze względu na ilość krawędzi w wierszu i kolumnie
e) Krawędzie poziome odpowiadają wierzchołkom **i** oraz **i+1**, gdzie i jest numerem wierzchołka oraz **i mod m = m-1**
f) Krawędzie pionowe odpowiadają wierzchołkom **i** oraz **i+m**, gdzie i jest numerem wierzchołka oraz **i <= m*n**
g) Labirynt jest reprezentowany przez minimalne drzewo rozpinające daną siatkę n*m grafu, gdzie aby połączyć **n*m** wierzchołków grafu potrzebujemy **(n*m)-1** krawędzi

- Schemat rozwiązania:

a) Najpierw musimy utworzyć połączenia wierzchołków (krawędzie), które tworzyłyby siatkę n x m grafu.
b) Dodajemy wszystkie krawędzie do wektora
c) W celu uzyskania losowości – mieszamy w losowy sposób położenie wszystkich elementów w wektorze
d) Tworzymy wektor zawierający wektory, które będą symbolizowały zbiory wierzchołków – na początku utworzone jest n*m wektorów zawierające tylko odpowiadający im wierzchołek
e) Jeśli dwa wierzchołki będą połączone poprzez krawędź – ich zbiory połączą się (zbiór zawierający drugi wierzchołek zostanie przeniesiony do zbioru zawierającego pierwszy wierzchołek)
f) Tworzymy pętle, w której pobieramy ostatnią krawędź z wektora zawierającego pomieszane elementy, następnie próbujemy wykonać połączenie wierzchołków i zbiorów jeśli dane dwa wierzchołki nie są jeszcze w tym samym zbiorze. W przypadku połączenia, dodajemy krawędź do grafu rozwiązania. Bez względu na przypadek – usuwamy daną krawędź z wektora
g) Po wykonanej pętli – graf rozwiązania zawiera n*m-1 krawędzi, które tworzą drzewo rozpinające danej siatki n*m grafu

- Omówienie przykładu – siatka: n=4 i m=3

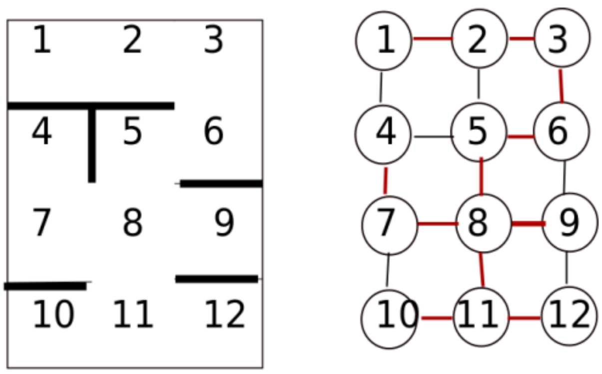


Na początku generujemy siatkę połączeń, którą umieścimy w wektorze.
Dla siatki z przykładu będzie to następujące 4*(3-1)+(4-1)*3=17 krawędzi:

0) (1,2)
1) (2,3)
2) (1,4)
3) (2,5)
4) (3,6)
5) (4,5)
6) (5,6)
7) (4,7)
8) (5,8)
9) (6,9)
10) (7,8)
11) (8,9)
12) (7,10)
13) (8,11)
14) (9,12)
15) (10,11)
16) (11,12)

Następnie mieszamy elementy wektora w losowy sposób, otrzymując przykładowy wynik
Np.:

0) (1,4)
1) (7,10)
2) (9,12)
3) (6,9)
4) (2,5)
5) (4,5)
6) (1,2)
7) (2,3)
8) (4,7)
9) (3,6)
10) (5,6)
11) (7,8)
12) (5,8)
13) (8,9)
14) (8,11)
15) (11,12)
16) (10,11)



Następnie bierzemy kolejne krawędzie od końca i działamy na odpowiadającym jej wierzchołkom zbiorach.

Początkowe rozmiary 4*3=12 zbiorów:
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1

Zbiory łączymy, gdy 2 wierzchołki danej krawędzi nie znajdują się w tym samym zbiorze.
Łączenie wierzchołków: (Krawędź : rozmiary zbiorów)

- (10,11): 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 0, 1 – zawartość zbioru 11 przechodzi do 10
- (11,12): 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 0, 0 – zawartość zbioru 12 przechodzi do 10 gdzie jest element 11
- (8,11): 1, 1, 1, 1, 1, 1, 1, 4, 1, 0, 0, 0 – zawartość zbioru 10 gdzie jest 11 przechodzi do 8
- (8,9): 1, 1, 1, 1, 1, 1, 1, 5, 0, 0, 0, 0 – zawartość zbioru 9 przechodzi do 8
- (5,8): 1, 1, 1, 1, 6, 1, 1, 0, 0, 0, 0, 0 – zawartość zbioru 8 przechodzi do 5
- Itd.

Gdy dla danej krawędzi łączymy zbiory – dodajemy dane krawędzie do grafu wynikowego o 3*4=12 wierzchołkach. Po połączeniu wszystkich wierzchołków , graf zawierać będzie 4*3-1=11 krawędzi utworzy drzewo rozpinające daną siatkę grafu.

- Pseudokod

```
Kruskal_labirynt(n, m):
    Stwórz wektor edges zawierający krawędzie siatki n*m grafu
    Pomieszaj losowo elementy wektora edges – edges.random_shuffle

    Stwórz wektor All_sets symbolizujący zbiory
    Dodaj n*m zbiorów zawierających tylko odpowiadający im wierzchołek do All_sets

    Stwórz graf wynikowy result o n*m wierzchołkach

    for(i=edges.size()-1; edges.size()>0; i--):
        Jeśli dwa wierzchołki danej krawędzi nie zawierają się w tym samym zbiorze w
        All_sets:
            połącz te zbiory i dodaj tę krawędź do grafu wynikowego result
        else
            continue

        edges.pop_back()
```

3. Opis użytych struktur danych

- Klasa Vertex – wierzchołek: każdy wierzchołek posiada unikalny numer (number)

```cpp
class Vertex {
    int number;
    public:
        int weight;
        std::string label;
        Vertex(int n) { number = n; }
        int Number() const { return number; }
};
```

- Klasa Edge – krawędź: każda krawędź łączy dwa wierzchołki (v0 i v1)

```cpp
class Edge {
    protected:
        Vertex* v0;
        Vertex* v1;
    public:
        int weight;
        std::string label;
        Edge (Vertex *V0, Vertex* V1) {
            v0 = V0;
            v1 = V1;
        }
        Vertex* V0 (){ return v0; };
        Vertex* V1 (){ return v1; };
        Vertex* Mate (Vertex *v) {
            if(v->Number() == v0->Number())
                return v1;
            return v0;
        }
};
```

- Klasa CountingVisitor – wizytator zliczający odwiedzone wierzchołki w grafie

```cpp
class CountingVisitor : Visitor<Vertex>{
    public:
        int visited_Vertices;
        CountingVisitor(): visited_Vertices(0) {}
        void Visit(Vertex& element) { visited_Vertices++; }
        void Zeruj() { visited_Vertices = 0; }
        bool IsDone() { return false; }
};
```

- Klasa Iterator – służy do poruszania się po grafie

```cpp
template <typename T>
class Iterator {
    public:
        virtual ~Iterator (){;}
        Iterator(){;}
        virtual bool IsDone() = 0;
        virtual T & operator*() = 0;
        virtual void operator++() = 0;
};
```

- Klasa GraphAsMatrix implementująca graf za pomocą listy sąsiedztwa

```cpp
class GraphAsMatrix {
    std::vector<Vertex*> vertices;
    std::vector<std::vector<Edge*>> adjacencyMatrix;
    bool isDirected;
    int numberOfVertices;
    int numberOfEdges = 0;

    class AllEdgesIter: public Iterator<Edge> {…

    class EmanEdgesIter: public Iterator<Edge> {…

    public:
        Iterator<Edge> &Edges() { return *new AllEdgesIter(*this); }
        Iterator<Edge> &EmanatingEdges(int v) { return *new EmanEdgesIter(*this, v); }

        GraphAsMatrix (int n, bool b) {…

        int NumberOfVertices() {…

        bool IsDirected() {…

        int NumberOfEdges(){…

        bool IsEdge(int u, int v) {…

        void MakeNull() {…

        void AddEdge (int u, int v) {…

        void AddEdge (Edge* edge) {…

        Edge* SelectEdge (int u, int v) {…

        Vertex* SelectVertex(int v) {…

        void Print_Edges() {…

        void DFS_visitor(CountingVisitor *visitor, Vertex *v, std::vector<bool> &visited) {…

        bool IsConnected() {…
};
```

4. **Oszacowanie złożoności czasowej i pamięciowej użytych struktur danych i podstawowych operacji na strukturach danych**

GraphAsMatrix

- Złożoność czasowa:
  Dodanie krawędzi – O(1)
- Złożoność pamięciowa:
  Przechowanie krawędzi w macierzy n*m – O(n*m)

5. **Oszacowanie złożoności czasowej i pamięciowej głównych algorytmów wykorzystanych w projekcie**

Kruskal_labirynt – utworzenie labiryntu

- Funkcja Create_mesh odpowiadająca za dodanie krawędzi do wektora, które tworzyłyby siatkę połączeń o wymiarach n*m grafu
  Złożoność czasowa – O(n*m)
  Złożoność pamięciowa – O(n*m)
- Funkcja RandomizeEdges odpowiadająca za pomieszanie krawędzi w wektorze
  Złożoność czasowa – O(n*m)
- Funkcja CreateSets odpowiadająca za utworzenie wektorów symbolizujących zbiory
  Złożoność czasowa – O(n*m)
  Złożoność pamięciowa – O(n*m)
- Funkcja ConnectVertices odpowiadająca za łączenie wierzchołków, które nie znajdują się w tym samym zbiorze
  Złożoność czasowa – O(n*m)
  Złożoność pamięciowa – O(n*m)

Ze względu na użycie funkcji ConnectVertices w pętli – złożoność czasowa całego algorytmu wyniesie: O((n*m)^2).

DFS – sprawdzenie czy wynikowy graf jest spójny
Złożoność czasowa – O(n*m)
Złożoność pamięciowa – O(n*m)

6. **Dokumentacja użytkowa: jak uruchomić program, jak wprowadzać dane**

- Kompilacja: g++ Labirynt_Kruskal.cpp -o Labirynt.x
- Uruchomienie: ./Labirynt.x a b c

Gdzie:

a) height - odpowiada za wysokość labiryntu, czyli pionową ilość wierzchołków w siatce tworzonego grafu.
b) width - odpowiada za szerokość labiryntu, czyli poziomą ilość wierzchołków w siatce tworzonego grafu.

c) delay - odpowiada za tryb wyświetlania, gdy jest równa 0 – program wyświetli od razu wygenerowany labirynt, natomiast w przypadku 1, program wyświetli proces powstawania połączeń pomiędzy kolejnymi wierzchołkami z zadanym odstępem czasowym delay_time_

d) delay_time_ - odpowiada za odstęp czasowy pomiędzy kolejnymi połączeniami w trybie wyświetlania stopniowego – jest zdefiniowaną stałą na początku programu (odstęp w milisekundach)

- Przykładowe uruchomienie: ./Labirynt.x 10 20 0