Wykorzystanie algorytmu Kruskala do generowania labiryntu

Opis problemu

Projekt rozwiązuje problem wygenerowania labiryntu o zadanych rozmiarach, *M x N* (prostokąt złożony z siatki kwadratowej o n wierszach i m kolumnach).

Opis rozwiązania

Generowanie takiego labiryntu na siatce kwadratowej może być uciążliwe, dlatego do tego zadania wykorzystany został algorytm Kruskala. Znalezieniu on drzewa rozpinającego dla danego grafu, wybierając losowo krawędzie zamiast tych o najmniejszej wadze, jak w oryginalnym algorytmie Kruskala.

Oto pseudokod funkcji wykorzystanej w projekcie:

```
funkacja KruskalModified(G):
  F = Zbiór wszystkich krawędzi w grafie G
  K = Zbiór krawedzi wyznaczających drzewo rozpinające w gr
afie G
  Tasuj zbiór krawędzi F w sposób losowy
  Stwórz z każdego wierzchołka zbiór rozłączny
  Dla każdej krawędzi e w F:
      Sprawdź, czy dodanie krawędzi do drzewa stworzy cykl
      Jeżeli dodanie e do K nie tworzy cyklu w K:
          Dodaj e do K i połącz zbiory
  Zwróć K
```

Oszacowanie złożoności czasowej i pamięciowej algorytmu Kruskala

V — to liczba wierzchołków $(x \cdot y)$

E — to liczba krawędzi $(2((x-1) \cdot y))$

$2V \approx E$

Złożoność czasowa

1. Tworzenie tablicę wszystkich krawędzi: $\sim O(V^2)$
2. Losowanie tablicy krawędzi: $\sim O(E)$
3. Tworzenie struktury zbiorów rozłącznych: $\sim O(V)$
4. Tworzenie drzewa rozpinającego: $\sim O(E \cdot log^*(V))$

`log*()` — to logarytm iterowany, funkcja, która mówi nam, ile razy musimy zastosować funkcję logarytmu do liczby n, zanim osiągniemy wartość mniejszą lub równą 1.
Na przykład.
jeśli mamy liczbę
$n = 16$, musimy zastosować logarytm dwukrotnie, aby osiągnąć wartość mniejszą lub równą 1:

- $log_2(16) = 4$
- $log_2(4) = 2$
- $log_2(2) = 1$

Więc
$log^*16 = 3$
Jest to bardzo wolno rosnąca funkcja. Nawet dla bardzo dużych liczb.

W algorytmie została wykorzystana tzw. kompresja ścieżki która w tej implementacji polega na aktualizacji wskaźników rodzica każdego wierzchołka na ścieżce do korzenia podczas wykonywania operacji. [Źródło]

Więc finalnie:

$$O(V^2) + O(E) + O(V) + O(E \cdot log^*(V)) \approx$$
$$O(V^2) + O(\tfrac{1}{2}V) + O(\tfrac{1}{2}V) + O(\tfrac{1}{2}V \cdot log^*(V)) \approx$$
$$O(V^2) + O(V) + O(\tfrac{1}{2}V \cdot log^*(V)) \approx \mathbf{O(V^2)}$$

Powodowane jest to głównie przez potrzebę skopiowania wszystkich istniejących krawędzi, aby potem ustawić je w kolejności losowej.

Złożoność pamięciowa

1. Tablica wszystkich krawędzi: $\sim O(E)$
2. Tablica zbiorów rozłącznych: $\sim O(V)$
3. Tablica drzewa rozpinającego: $\sim O(E)$

Łączna złożoność pamięciowa algorytmu wynosi więc

$$O(E + V + E) = O(E + V)$$

Opis użytych struktur

Projekt oparty jest na implementacji tworzonej podczas zajęć. Głównie wykorzystywane są struktury `GraphAsMatrix`, `Edge`, `Vertex`.

Klasa `Labyrinth`

Służy jako reprezentacja labiryntu w kodzie. Posiada proste atrybuty i metody takie jak rozmiar *M x N*, graf na którego podstawie powstaje labirynt, tablicę z krawędziami labiryntu czy metodę służącą do wyświetlania wizualizacji labiryntu.

```
class Labyrinth
{
private:
  int x;
  int y;
  GraphAsMatrix* graph;
  std::vector<Edge*> labyrinthEdges;

  void generateEdges();
```

```
public:
  Labyrinth(int x, int y);
  ~Labyrinth();

  void Print();
};
```

Konstruktor

Tworzy on graf oraz generuje siatkę podstawowych krawędzi dla danych wymiarów labiryntu.
Następnie jest wywoływana metoda `KruskalModified()` zwracająca tablicę krawędzi.

Na samym końcu prezentowana jest nam liczba uzyskanych krawędzi przy pomocy powyższego algorytmu wraz z ich usunięciem z grafu — Krawędzie między wierzchołkami wybrane do drzewa rozpinającego graf reprezentują „brak ścian". Pozostałe krawędzie grafu reprezentują ściany.

Metoda `generateEdges()`

Służy do wypełnienia grafu `graph` wszystkimi możliwymi krawędziami tworząc siatkę dla danych wymiarów labiryntu.

Metoda `Print()`

Jej zadaniem jest wypisanie wizualizacji ścieżek labiryntu. W tej reprezentacji spacje służą jako ściany. Znaki reprezentujące wierzchołki oraz krawędzie są zapisywane w macierzy a następnie wypisywane.

Oszacowanie złożoności czasowej i pamięciowej podstawowych operacji klasy `Labyrinth`

Złożoność czasowa

- Metoda `generateEdges()`: $\sim O(V)$

- Metoda `Print()` : $\sim O(R \cdot C)$ (gdzie *R=2y−1* i *C=2x−1*, generowanie macierzy ze znakami do wizualizacji)
- Konstruktor: $\sim O(V^2) + O(V) + O(V^2) + O(V) \approx O(V^2)$

  (tworzenie grafu + `generateEdges()` + `KruskalModified()` + pętla po tablicy z krawędziami labiryntu)

Złożoność pamięciowa

- Metoda `generateEdges()`: $\sim O(1)$ (Nie tworzy żadnych dodatkowych struktur danych, więc złożoność pamięciowa wynosi 1, jedynie zmienia wartości w istniejących już strukturach)
- Metoda `Print()` : $\sim O(R \cdot C)$ (gdzie *R=2y−1* i *C=2x−1*, generowanie macierzy ze znakami do wizualizacji)
- Konstruktor: $\sim O(V) + O(E + V)$

  (tworzony jest graf o liczbie wierzchołków równej iloczynowi rozmiarów labiryntu + `KruskalModified()`)

Uruchamianie programu oraz użytkowanie

Program można uruchomić komendą: `make run ARGS="x y"`,
gdzie x oraz y oznaczają rozmiar labiryntu do wygenerowania.

Kompiluje on pliki ze zdefiniowanymi metodami oraz generuje jeden prosty labirynt.

Przykładowe wywołanie programu dla rozmiaru labiryntu *5×5*.

```
---------- ZMODYFIKOWANY ALGORYTM KRUSKALA ----------

A) KRAWĘDZIE JAKO DROGI W LABIRYNCIE

v0: 12, v1: 17, waga: 0
v0: 17, v1: 18, waga: 0
v0: 21, v1: 22, waga: 0
v0: 19, v1: 24, waga: 0
v0: 5, v1: 10, waga: 0
v0: 3, v1: 4, waga: 0
```

```
v0: 16, v1: 21, waga: 0
v0: 13, v1: 14, waga: 0
v0: 10, v1: 11, waga: 0
v0: 0, v1: 5, waga: 0
v0: 1, v1: 6, waga: 0
v0: 5, v1: 6, waga: 0
v0: 13, v1: 18, waga: 0
v0: 10, v1: 15, waga: 0
v0: 7, v1: 12, waga: 0
v0: 7, v1: 8, waga: 0
v0: 15, v1: 16, waga: 0
v0: 3, v1: 8, waga: 0
v0: 14, v1: 19, waga: 0
v0: 23, v1: 24, waga: 0
v0: 1, v1: 2, waga: 0
v0: 17, v1: 22, waga: 0
v0: 9, v1: 14, waga: 0
v0: 15, v1: 20, waga: 0

 B) WIZUALIZACJA

+   +---+   +---+
|   |       |
+---+   +---+   +
|       |       |
+---+   +   +---+
|       |   |   |
+---+   +---+   +
|   |   |       |
+   +---+   +---+
```