Dokumentacja rozwiązania

Grzegorz Janysek

24 maja 2022

1 Problem i rozwiązanie

Celem jest wygenerowanie grafu reprezentującego labirynt, a następnie wyświetlenie wizualizacji tego grafu. Labirynt ma nie zawierać cykli i ma być spójny. Aby to zapewnić znajdywane jest losowe drzewo rozpinające graf początkowy będący siatką prostokątną wierzchołków.

Jako argumenty program otrzymuje rozmiar generowanego labiryntu: n kolumn na m wierszy.

```
fn main(szerokosc, wysokosc) {
  graf = wygeneruj_graf(szerokosc, wysokosc);
  znajdz_drzewo_rozpinajace(graf);
  wizualizuj(graf);
}
```

Na podstawie argumentów tworzony jest graf nieskierowany o $n*m$ wierzchołkach i krawędziach pomiędzy każdą parą sąsiadujących wierzchołków. W sposób taki, aby każdy z wierzchołków poza tymi które znajdują sie z krawędzi prostokątnej siatki miał czterech sąsiadów: na górze, na dole, po lewej i po prawej stronie.

```
fn wygeneruj_graf(szerokosc, wysokosc) {
  graf = Graph::new();

  for i in 0..(szerokosc * wysokosc) {
    // dodaj wierzcholek o grupie 'i'
    graf.graph.add_node(i);
  }

  // dodaj poziome krawedzie
  for x in 1..szerokosc {
    for y in 0..wysokosc {
      graf.add_edge(
        node_at(x, y),
        node_at(x - 1, y),
        // oznacz krawedz jako
        // nie nalezaca do drzewa
        false,
      )
    }
  }

  // dodaj poziome krawedzie
  for x in 0..szerokosc {
    for y in 1..wysokosc {
      graf.add_edge(
        node_at(x, y),
        node_at(x, y - 1),
        // oznacz krawedz jako
        // nie nalezaca do drzewa
        false,
      )
    }
  }

  return graf
}
```

Następnie znajdywane jest pseudolosowe drzewo rozpinające graf. Użyty do tego został zmodyfikowany algorytm Kruskala, różniący się wyborem krawędzi. Algorytm iteruje po krawędziach grafu w losowej kolejności. Jeżeli natrafi na krawędź łączącą wierzchołki w różnych grupach tj. należących do rozdzielnych poddrzew grafu: łączy te wierzchołki w jedno drzewo scalając grupy do których należą i oznaczając krawędź jak należącą do drzewa. Oznaczone w ten sposób krawędzie tworzą zakończeniu iteracji drzewo rozpinające graf.

```
fn znajdz_drzewo_rozpinajace(graf) {
  // zapewnia losowa kolejnosc itreacji
  edges = graf.edges().shuffle()

  for e in edges {
    // jezeli krawedz laczy
    // wierzcholki o roznych grupach
    if e.a.grupa != e.b.grupa {
      // zlacz te grupy...
      merge(e.a.grupa, e.b.grupa)
      // ...i oznacz krawedz
      // jako nalezaca do drzewa
      e.value = true
    }
  }
}
```

2 Użyte struktury danych

W programie wykorzystano grafu jako struktury przechowującej wierzchołki labiryntu. Implementacja grafu używa macierzy sąsiedztwa. Pozwala ona na reprezentację grafów skierowanych oraz nieskierowanych. Struktura jest generyczna i umożliwia na para-

metryzowanie typów wag krawędzi jak i wierzchołków. Zaimplementowano podstawowe metody CRUD, iteratory i referencje dla wierzchołków i krawędzi grafu.

3 Oszacowanie złożoności grafu

Złożoność pamięciowa ograniczona jest rozmiarem macierzy sąsiedztwa i wynosi $O(n^2)$. Złożoności czasowe operacji na grafie kształtują się następująco:

operacja	złożoność
dodanie wierzchołka	$O(n)$
odczytanie wierzchołka	$O(1)$
zmiana wagi wierzchołka	$O(1)$
usunięcie wierzchołka	$O(n)$
dodanie krawędzi	$O(1)$
odczytanie krawędzi	$O(1)$
zmiana wagi krawędzi	$O(1)$
usunięcie krawędzi	$O(1)$
odczytanie sąsiadów wierzchołka	$O(n)$
odczytanie wierzchołków krawędzi	$O(1)$

4 Oszacowanie złożoności zmodyfikowaego algorytmu Kruskala

n jest ilością wierzchołków. Operacja utworzenia listy krawędzi o losowej kolejności działa w czasie $O(n)$.

Porównanie grup wierzchołki odbywa się w czasie $O(1)$.

Scalenie grup wierzchołków dla dedykowanej do tego zadania struktury zbiorów rozłącznych ma złożoność czasową $O(log\ n)$. W omawianej implementacji nie jest ona jednak wykorzystywana co pogarsza złożoność tej operacji do $O(n)$. Jest to wynikiem sposobu łączenia grup a i b, polegającego na iteracji po wszystkich wierzchołkach V takich że grupa V to b a następnie zmiany grupy V na a.

Porównanie oraz scalanie grup wymagane jest dla każdej z krawędzi, których to liczba dąży do $2n$. Daje to całkowitą złożoność czasową wynoszącą $O(n^2)$

Złożoność pamięciowa dyktowana jest rozmiarem (tworzonej w początkowym kroku) listy krawędzi i wynosi $O(n)$.

5 Dokumentacja użytkowa

Opisane w dalszej części komendy wymagają łańcucha narzędzi języka Rust.

5.1 Testowanie

Uruchomienie zautomatyzowanych testów biblioteki zawierającej implementację grafu:

```
$ cargo test
```

5.2 Kompilacja

Kompilacja biblioteki grafu oraz programu generującego labirynt. Plik wykonywalny znajduje się pod ścieżką `target/release/maze`:

```
$ cargo build --release
```

5.3 Uruchomienie programu

Uruchomienie program generującego labirynt z domyślnymi argumentami:

```
$ cargo run --release
```

Wyświetlenie pomocy do programu:

```
$ cargo run --release -- -w 30 -h 5
```

Uruchomienie programu z argumentami określającymi wielkość labiryntu:

```
$ cargo run --release -- -w 30 -h 5
```