Dokumentacja projektu z AiSD II

Autor: Szymon Fortuna

Wybrany temat projektu: 3

Opis problemu

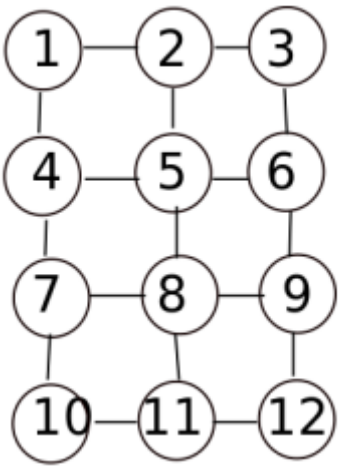
Problem dotyczy generowania i wizualizacji prostokątnego labiryntu o podanej wysokości H i szerokości W z częściowo pseudolosowymi krawędziami. Częściowo, ponieważ chcemy wymusić acykliczność w labiryncie, tzn. aby pomiędzy dowolnymi dwoma polami tego labiryntu istniała dokładnie jedna ścieżka niezawierająca powtarzających się pól.

Opis rozwiązania

1. Generowanie siatki:

```
GraphAsMatrix graph = graph(H * W, nieskierowany)
for (i = 0, 1, …, graph.numberOfVertices()){
     if (i + W < graph.numberOfVertices())
          graph.AddEdge(i, i + W)
     if ((i + 1) % W != 0)
          graph.AddEdge(i, i + 1)
}
```

Czego wynikiem będzie graf o następującej strukturze:



z tą różnicą, że w mojej implementacji wierzchołki są numerowane od 0 do H * W – 1, a nie od 1 do H * W, jak powyżej.

2. Znalezienie drzewa rozpinającego graf przy użyciu zmodyfikowanego algorytmu Kruskala:

```
SetPartition partition = partition (graph.Vertices)

while (nie wszystkie wierzchołki znajdują się w tym samym podzbiorze podziału){

    Edge* randomEdge = krawędź wylosowana w taki sposób, aby nie doprowadzić do cyklu

    spanningTree.AddEdge(randomEdge)

    set first = partition.find(randomEdge.V0())

    set second = partition.find(randomEdge.V1())

    partition.join(first, second)

}
```

3. Wypisanie danych dotyczących drzewa rozpinającego do pliku tekstowego:

```
otwórz plik .txt i sprawdź poprawność otwarcia

wypisz(plik, H)

wypisz(plik, W)

for (e : spanningTree.Edges){

    wypisz(plik, e.V0().Number())
```

```
    wypisz(plik, e.V1().Number())
}
```

4. Klasa SetPartition zawiera pole partition będące wektorem zbiorów oraz zmienną całkowitoliczbową maxPower, przechowującą moc najliczniejszego podzbioru w podziale. Konstruktor klasy SetPartition:

```
SetPartition(collection){
    for (c : collection){
        set singleton
        singleton.insert(element)
        partition.push_back(singleton)
    }
    maxPower = 1
}
```

5. Operacja find(element) znajduje podzbiór, do którego należy wskazany element:

```
set find(element){
    for (s : partition){
        for (e : s){
            if (e == element)
                return s
        }
    }
    wypisz(„Nie znaleziono szukanego elementu”)
    return empty_set
}
```

Zbiór pusty jest polem klasy SetPartition używanym tylko w powyższej metodzie na wypadek, gdyby metoda find została wywołana z argumentem, który nie należy do podziału, jednak w mojej aplikacji taka sytuacja nie powinna występować.

6. Metoda `join(first, second)` łączy dwa podzbiory w jeden, usuwa jeden z nich i w razie potrzeby aktualizuje licznik `maxPower`:

```
void join(first, second){
    if (first.size() + second.size() > maxPower)
        maxPower = first.size() + second.size()
    for (element : second)
        first.insert(element)
    partition.erase(second)
}
```

7. Wyświetlanie labiryntu opiera się na następującej zasadzie:

```
edges = [odczytaj(plik)]
H = edges[0]
W = edges[1]
wypisz tabelę o wymiarach H x W, wypełniając ją
wartościami od 0 do H * W – 1, domyślnie ustawiając ścianę
labiryntu pomiędzy każdą możliwą parą
for (i = 3, 5, …, edges.length – 1){
    if (edges[i] – edges[i – 1] == 1)
        usuń prawą i lewą ścianę pomiędzy komórkami
edges[i – 1] i edges[i]
    if (edges[i] – edges[i – 1] == W)
        usuń dolną i górną ścianę pomiędzy komórkami
edges[i – 1] i edges[i]
}
```

Opis użytych struktur danych

1. W projekcie użyłem implementacji grafu z zajęć (`Vertex, Edge, GraphAsMatrix` oraz wizytatory i iteratory), dodając kilka pól i metod:

a) `edges` – wektor przechowujący wszystkie krawędzie
b) `SelectEdge(int n)` – dotychczas mogliśmy wybrać krawędź na podstawie numerów jej wierzchołków, to przeciążenie ma zastosowanie w przypadku losowego wyboru krawędzi

2. Klasa `SetPartition` odpowiada za podział zbioru. Podział reprezentowany jest za pomocą wektora podzbiorów. Przechowywana jest także moc najliczniejszego podzbioru.
3. Klasa `Labirynth` służy do zainicjowania początkowej siatki oraz znalezienia drzewa rozpinającego. Posiada też metodę, która zapisuje krawędzie drzewa do pliku tekstowego.

Oszacowanie złożoności czasowej i pamięciowej użytych struktur danych i podstawowych operacji na tych strukturach

Oznaczmy ilość pól labiryntu jako n.

Złożoność pamięciowa klasy reprezentującej graf jest zdominowana przez macierz sąsiedztwa o rozmiarze n x n. Zatem złożoność pamięciowa (a zatem też czasowa konstruktora) wynosi $O(n^2)$. Natomiast operacje, z których korzystam w projekcie to:

a) przechodzenie po wszystkich wierzchołkach – O(n)
b) przechodzenie po wszystkich krawędziach – $O(n^2)$
c) pozostałe operacje typu wstawienie, znalezienie odpowiedniej krawędzi zajmują O(1)

Złożoność pamięciowa struktury `SetPartition` wynosi O(n), ponieważ każdy wierzchołek jest przechowywany w danym momencie tylko raz, w odpowiednim dla siebie zbiorze. Natomiast złożoności czasowe metod tej klasy wyglądają tak:

a) konstruktor – O(n), ponieważ iteruje po wszystkich elementach podziału (w tym przypadku pól labiryntu) i umieszcza je w odpowiednich podzbiorach
b) operacja `join` iteruje po elementach drugiego zbioru i wstawia je do pierwszego, w najgorszym przypadku tych elementów będzie n – 1, zaś w najbardziej optymistycznym tylko 1, zatem złożoność tej metody wynosi w poszczególnych notacjach O(n) i Ω(1)

c) operacja `find` przegląda podział aż do znalezienia szukanej wartości, więc w najgorszym przypadku będzie to O(n), a w najlepszym Ω(1)

Struktura labiryntu zajmuje pamięciowo $O(n^2)$ ze względu na przechowywanie grafu bazowego i jego drzewa rozpinającego. Z kolei operacje:

a) konstruktor – O(n), bo iteruje po wszystkich wierzchołkach w celu dodania odpowiednich krawędzi; dodawanie krawędzi jest O(1)
b) zmodyfikowany algorytm Kruskala potrzebuje czasu liniowego, żeby przygotować ciąg wierzchołków, który zostanie przekazany do konstruktora `SetPartition`. Modyfikacja klasy `GraphAsMatrix` polegająca na udzieleniu przyjaźni tej funkcji mogłaby zaoszczędzić ten czas. Główną częścią tej metody jest pętla `while`. Wygenerowanie odpowiedniej krawędzi (pętla `do while`) zajmuje $O(n^3)$, ponieważ należy sprawdzać, czy odpowiednia krawędź nie została już wylosowana, czy nie doprowadzi do cyklu, a złożoność warunku wynosi O(n). Następne 4 linijki kodu mają złożoność odpowiednio O(1), O(n), O(n) i O(n), złożoność warunku głównej pętli jest stała
c) zapisanie labiryntu do pliku tekstowego jest zależne liniowo od ilości krawędzi w otrzymanym drzewie rozpinającym

Oszacowanie złożoności czasowej i pamięciowej głównych algorytmów wykorzystanych w projekcie

Odczytanie danych z pliku tekstowego jest rzędu O(e), gdzie e to liczba krawędzi drzewa.

Wypisanie siatki bez krawędzi wynosi O(n).

Zwizualizowanie faktycznej struktury labiryntu jest w czasie O(e), bo algorytm musi przejrzeć wszystkie krawędzie i w czasie stałym usunąć każdą ze ścian, które należy usunąć.

Dokumentacja użytkowa: jak uruchomić program, jak wprowadzać dane?

Projekt został podzielony na dwa moduły:

a) zbiór plików nagłówkowych, implementacji zadeklarowanych w nich funkcji oraz funkcja `main()` napisane w języku C++
b) prosta strona internetowa HTML wraz z formatowaniem CSS i plikiem skryptów napisanych w JavaScript

Pierwszym etapem uruchomienia projektu jest kompilacja modułu napisanego w C++. Można to zrobić poleceniem `g++ *.cpp -o graph.exe`, a następnie uruchomić komendą `./graph.exe`. Po uruchomieniu należy podać na standardowe wejście szerokość i wysokość labiryntu. Program powinien wygenerować plik tekstowy o nazwie *edges_to_visualize.txt* zrozumiały dla skryptu, poinformować o tym i zakończyć działanie.

Następnym etapem jest otworzenie w przeglądarce pliku *index.html* i w polu formularza wskazanie ww. pliku. Wówczas na stronie powinien zostać wygenerowany labirynt. Aby wyświetlić nowy labirynt na podstawie nowych danych należy odświeżyć stronę. Skrypt zakłada, że przekazany plik jest w poprawnym formacie.

Przykładowe wyjście programu w C++:

```
PS D:\Studia\Algorytmy i struktury danych II\Graf\Labyrinth-with-modified-Kruskal-algorithm> ./graph.exe
Podaj wysokosc siatki: 4
Podaj szerokosc siatki: 8
Zapisano plik
```

Przykładowy zrzut ekranu z działania strony: