AISDE – ćwiczenie 5

Algorytmy grafowe

Wstęp

Ćwiczenie ma na celu zapoznanie się z przykładowymi implementacjami wybranych algorytmów grafowych. Analizowane będą dwa rodzaje algorytmów: znajdujące najkrótszą drogę w grafie skierowanym (digrafie) oraz tworzące najmniejsze drzewo rozpinające w grafie nieskierowanym

Zakłada się znajomość:

- podstawowej terminologii używanej w teorii grafów,
- algorytmów: Dijkstry, Floyda, Kruskala, Borůvki i Prima,
- podstaw języka C++,
- niniejszej instrukcji; sekcja *Dodatek* jest nieobowiązkowa, lecz ułatwia zrozumienie kodu np. w celu jego modyfikacji lub oszacowania teoretycznej złożoności obliczeniowej i pamięciowej zaimplementowanych algorytmów.

Wskazane jest także zapoznanie się z kodem programów wykorzystywanych w ćwiczeniu, zwłaszcza z treścią funkcji: *Floyd*, *DijkstraMatrix*, *DijkstraList*, *DijkstraMatrixPQ*, *DijkstraListPQ* i *DFS* (plik *MinpathMain.cpp*) oraz *Kruskal* i *Boruvka* (plik *MST_Algorithms.cpp*).

W całej instrukcji liczba wierzchołków grafu jest oznaczona jako *m*, zaś liczba krawędzi – jako *n*.

Wykorzystywane narzędzia

W ćwiczeniu będą wykorzystywane następujące programy:

- *graphgen* – generator losowych grafów nieskierowanych,
- *digraphgen* – generator losowych grafów skierowanych,
- *mst* – program tworzący najmniejsze drzewo rozpinające (ang. *Minimum Spanning Tree* – stąd nazwa) w grafie **nieskierowanym**,
- *minpath* – program znajdujący najkrótsze ścieżki w grafie **skierowanym**.

Działanie tych narzędzi jest krótko omówione poniżej, natomiast szczegóły ich budowy – w końcowym punkcie **Dodatek**. Wszystkie programy są dostępne w postaci plików źródłowych. Wpisanie polecenia *make* w katalogu projektu powoduje pojawienie się w nim wspomnianych czterech plików wykonywalnych. Przyjęta została zasada, że każda klasa jest zdefiniowana w osobnym pliku źródłowym o nazwie identycznej jak nazwa klasy. Wyjątkiem są klasy szablonowe *List* i *ListElement*, których ciała są z konieczności definiowane w plikach nagłówkowych. Po modyfikacji dowolnego pliku źródłowego należy dokonać rekompilacji programem *make*. Z kolei po modyfikacji pliku nagłówkowego należy najpierw usunąć

wszystkie produkty kompilacji poleceniem *make clean*, a następnie powtórnie skompilować cały projekt poleceniem *make*.

Generator grafów nieskierowanych – *graphgen*

Jest to prosty generator losowych grafów nieskierowanych. Polecenie:

```
./graphgen m n [nazwa_pliku_wynikowego]
```

spowoduje stworzenie grafu o *m* wierzchołkach i *n* gałęziach. Zostanie on zapisany w pliku o podanej nazwie. Jeśli nazwa zostanie pominięta, graf będzie zapisany w pliku *graf.txt*.

UWAGA! Jeżeli argument *n* przekracza liczbę krawędzi pełnego grafu nieskierowanego o *m* wierzchołkach, zostanie wygenerowany taki właśnie graf pełny, o czym użytkownik jest informowany. Należy to brać pod uwagę podczas analizy zależności czasu wykonania algorytmów grafowych od liczby krawędzi. Natomiast próba wygenerowania grafu, w którym $n < m - 1$, zakończy się błędem, ponieważ graf taki byłby niespójny.

Każdy graf wygenerowany przez program graphgen jest spójny – zawiera ścieżkę Hamiltona prowadzącą przez wierzchołki o kolejnych numerach. Stopnie wszystkich wierzchołków grafu są w przybliżeniu równe. Wagi krawędzi to wartości bezwzględne zmiennej losowej o rozkładzie normalnym o zerowej wartości oczekiwanej, zaokrąglone w górę do najbliższej liczby naturalnej. Graf wynikowy zostaje zapisany do pliku tekstowego. Każdy wiersz takiego pliku zawiera opis jednej krawędzi:

```
nr_wierzchołka_1 <sp> nr_wierzchołka_2 <sp> waga_krawędzi
```

gdzie **<sp>** oznacza spację lub znak tabulacji. Na przykład wiersz o zawartości:

```
2 10 5
```

opisuje krawędź o wadze 5 wychodzącą z wierzchołka 2 i dochodzącą do wierzchołka 10. Numeracja wierzchołków rozpoczyna się od zera. Ponieważ graf jest nieskierowany, kolejność wierzchołków w opisie krawędzi nie ma znaczenia – jako pierwszy wypisywany jest wierzchołek o mniejszym indeksie.

Generator grafów skierowanych – *digraphgen*

Działanie tego programu jest bardzo podobne do poprzedniego. Uruchamiany jest poleceniem:

```
./digraphgen m n [nazwa_pliku_wynikowego]
```

Domyślna nazwa pliku wyjściowego to *digraf.txt*. Wagi krawędzi mają rozkład równomierny na przedziale od 1 do 100. Format tworzonego pliku jest identyczny, jak dla programu *graphgen*. Ponieważ jednak w tym przypadku mamy do czynienia z grafem skierowanym, zakładamy, że pierwszy z wierzchołków opisujących krawędź jest jej początkiem, a drugi – końcem.

UWAGA! Jeżeli argument *n* przekracza liczbę krawędzi pełnego grafu skierowanego o *m* wierzchołkach,

zostanie wygenerowany taki właśnie graf pełny, o czym użytkownik jest informowany. Należy to brać pod uwagę podczas analizy zależności czasu wykonania algorytmów grafowych od liczby krawędzi. Natomiast próba wygenerowania grafu, w którym $n < m$, zakończy się błędem, ponieważ graf taki byłby niespójny. Każdy graf wygenerowany przez program `digraphgen` jest spójny – zawiera cykl Hamiltona prowadzący przez wierzchołki o kolejnych numerach, tj. $0 \to 1 \to 2 \to \ldots \to (m–1) \to 0$.

Program wyznaczający najmniejsze drzewo rozpinające – *mst*

Program ten zawiera przykładowe implementacje algorytmów Kruskala i Borůvki, wyznaczających najmniejsze drzewo rozpinające grafu nieskierowanego. Wywołanie ma postać:

```
./mst nazwa_pliku_z_grafem_nieskierowanym
```

Graf wejściowy może być wygenerowany przez program *graphgen*. Program *mst* podaje czasy wykonania obu algorytmów. Dla grafów o liczbie krawędzi nie przekraczającej wartości MAX_PRINTABLE_E (zdefiniowanej w pliku *MST_Utils.h*) wyświetla także krawędzie tworzące znalezione drzewo. Są one wypisywane w takiej kolejności, w jakiej były dodawane do drzewa, co może być pomocne przy analizowaniu algorytmu.

Program wyznaczający najkrótsze ścieżki w grafie skierowanym – *minpath*

Program *minpath* wyznacza odległości pomiędzy parami węzłów digrafu odczytanego z pliku. Możliwe są dwa sposoby wywołania:

```
./minpath nazwa_pliku_z_digrafem v w
```

```
./minpath nazwa_pliku_z_digrafem
```

Pierwszy sposób pozwala wyznaczyć długość drogi od wierzchołka *v* do wierzchołka *w*, natomiast drugi – macierz odległości pomiędzy wszystkimi parami wierzchołków. Element w wierszu *i* i kolumnie *j* tej macierzy odpowiada długości drogi prowadzącej od wierzchołka *i* do wierzchołka *j*. Numeracja wierszy i kolumn zaczyna się od zera. Znak „minus” w jakiejś pozycji oznacza brak drogi pomiędzy odpowiednimi wierzchołkami. Wyświetlanie działa tylko w przypadku, gdy liczba wierzchołków grafu nie przekracza wartości stałej MAX_PRINTABLE_V, zdefiniowanej w pliku *MinpathUtils.h*. Niezależnie od sposobu wywołania programu, wyświetla on czas wykonania każdego z algorytmów.

Przebieg ćwiczenia

Ćwiczenie należy rozpocząć od prawidłowej inicjalizacji generatorów zmiennych losowych w programach tworzących grafy. W tym celu stałej SEED, zdefiniowanej w plikach *Graphgen.cpp* i *Digraphgen.cpp*, należy nadać wartość swojego numeru albumu.

UWAGA! W oryginalnej wersji programu *minpath* uruchamiane są po kolei wszystkie wymienione w instrukcji algorytmy. Jeżeli użytkownik zamierza pominąć któryś z nich, np. z uwagi na nieakceptowalnie długi czas wykonania, może tego dokonać wykomentowując w pliku *MinpathMain.cpp* wywołanie funkcji *FindShortestPath* dla tego algorytmu.

Przykładowe zadania do wykonania

1. Dla grafu **skierowanego** o kilku wierzchołkach i kilku-kilkunastu krawędziach obserwujemy efekty działania każdego z algorytmów poszukujących najkrótszej drogi między wszystkimi parami wierzchołków. Jeżeli różne algorytmy dostarczają różnych wyników, należy wyjaśnić przyczynę. Graf, wraz z numerami wierzchołków i wagami krawędzi, należy narysować sprawozdaniu i zaznaczyć na jedną z dróg znalezionych przez algorytmy.

2. Dla grafu **nieskierowanego** o kilku wierzchołkach i kilku-kilkunastu krawędziach obserwujemy efekty działania każdego z algorytmów wyznaczających najmniejsze drzewo rozpinające (MST). Jeżeli różne algorytmy dostarczają różnych wyników, należy wyjaśnić przyczynę. Graf, wraz z numerami wierzchołków i wagami krawędzi, należy narysować sprawozdaniu a następnie wyróżnić na nim krawędzie znalezionego drzewa rozpinającego.

3. Dla wskazanych przez prowadzącego algorytmów poszukujących najkrótszych dróg w grafie **skierowanym** badamy zależność czasu obliczeń od liczby wierzchołków m tego grafu przy ustalonej liczbie krawędzi n i vice versa. Minimalna liczba wierzchołków/krawędzi użyta w analizie powinna pozwalać uzyskać czasy rzędu dziesiątek milisekund. Mniejsze czasy są podatne na fluktuacje, ponadto na wielu maszynach obserwowane czasy są skwantowane co jedną milisekundę (a na niektórych maszynach nawet co 10 milisekund). Maksymalna liczba wierzchołków/krawędzi jest ograniczona wyłącznie cierpliwością osoby wykonującej ćwiczenie. Zaleca się prowadzenie badań **przynajmniej** do osiągnięcia czasów rzędu kilkunastu/kilkudziesięciu sekund. Liczba krawędzi n grafu skierowanego musi spełniać oczywiste warunki:

$$m \le n \le m(m-1).$$

Górne ograniczenie wynika z liczby krawędzi digrafu pełnego, zaś dolne z wymagania spójności grafu. Sugeruje się, by najmniejsza i największa liczba krawędzi użyta w eksperymencie były zbliżone odpowiednio do dolnego i górnego ograniczenia. W sprawozdaniu należy zamieścić wykresy z wynikami eksperymentów, tj. czasami obliczeń dla różnych wartości m i n. (Wyniki te należy również umieścić w tabelach na końcu sprawozdania.) Skala na obu osiach powinna pozwolić łatwo oszacować postać funkcyjną zależności czasu działania od m lub n. W zależności od spodziewanej klasy złożoności można zastosować na jednej lub obu osiach skalę logarytmiczną. Przy spodziewanej złożoności $O(f(m, n))$ można także spróbować wykreślić czas w funkcji $f(m)$ lub $f(n)$. Następnie proszę określić złożoność czasową każdej z przebadanych implementacji algorytmu. **Należy porównać ją z teoretyczną złożonością wyznaczoną na podstawie analizy wykorzystywanych struktur danych i algorytmów**.

UWAGA! W punkcie tym program *minpath* trzeba uruchamiać w trybie „każdy do każdego”, czyli bez podawania wierzchołka startowego i docelowego. Przy takim ustawieniu algorytm Dijkstry zostanie uruchomiony *m* razy, startując za każdym razem od innego spośród *m* wierzchołków i znajdując odległość do wszystkich pozostałych. Dzięki temu łatwiej będzie porównać różne wersje algorytmu Dijkstry z algorytmem Floyda, który z zasady znajduje drogi między wszystkimi parami wierzchołków. Dodatkową korzyścią jest uzyskanie większych – a zatem mniej podatnych na fluktuacje – czasów działania. Przy szacowaniu złożoności obliczeniowej algorytmów **trzeba jednak pamiętać, że zmierzony czas odpowiada *m*-krotnemu wywołaniu algorytmu Dijkstry i jednokrotnemu wywołaniu algorytmu Floyda**.

4. Dla wskazanych przez prowadzącego algorytmów wyznaczających najmniejsze drzewo rozpinające grafu **nieskierowanego** badamy zależność czasu obliczeń od liczby wierzchołków *m* tego grafu przy ustalonej liczbie krawędzi *n* i vice versa. Ponieważ algorytmy wyznaczania MST są bardzo szybkie, rozmiary największych grafów użytych w tym punkcie mogą (a nawet powinny) osiągać kilka/kilkadziesiąt tysięcy wierzchołków i kilkanaście/kilkadziesiąt milionów krawędzi. **Sama generacja grafów tej wielkości i ich zapis na dysk potrwa o wiele dłużej, niż wyznaczanie MST – proszę uzbroić się w cierpliwość.** Wszystkie uwagi dotyczące użytej liczby krawędzi i wierzchołków, sposobu wykonywania wykresów itp. – jak w opisie przykładowego zadania 3.

5. Modyfikujemy dowolną z wersji algorytmu Dijkstry, tak by wypisywała na ekran znalezioną najkrótszą drogę, tj. sekwencję wierzchołków od początkowego do końcowego. Kod należy krótko opisać i **koniecznie zamieścić wyniki testowania** dla niewielkiego grafu.

6. Ulepszamy algorytm Floyda. Testujemy poprawność działania naszej wersji na grafie na tyle małym, by macierz najkrótszych dróg została wypisana na ekran. Następnie badamy zależność czasu obliczeń od liczby wierzchołków i krawędzi i porównujemy z oryginalną implementacją oraz z jedną z implementacji algorytmu Dijkstry. Wszystkie uwagi dotyczące użytej liczby krawędzi i wierzchołków, sposobu wykonywania wykresów itp. – jak w opisie przykładowego zadania 3.

7. Jak można zmodyfikować funkcję *Kruskal*(), by zmniejszyć jej złożoność obliczeniową? Proszę wprowadzić tę modyfikację, przetestować kod (aby uzyskać pewność, że tworzy drzewo o identycznej wadze sumarycznej, jak wersja oryginalna) i porównać z oryginałem pod względem czasu wykonania. Na tej podstawie oszacować złożoność obliczeniową ulepszonej wersji i porównać ją ze złożonością oszacowaną teoretycznie.

Sprawozdanie

Sprawozdanie powinno zawierać następujące elementy:

- Opis przeprowadzonych badań wraz z uzasadnieniem wyboru użytych parametrów, np. liczby wierzchołków i krawędzi grafu.
- Porównanie złożoności czasowej użytych algorytmów: wykresy w odpowiedniej skali, zależności

czasu obliczeń i zajętości pamięci od liczby wierzchołków i krawędzi (notacja *O*). Należy porównać zaobserwowaną złożoność z teoretyczną, tj. oszacowaną na podstawie analizy kodu. Należy zastanowić się, w jakich sytuacjach uwidaczniają się mocne strony poszczególnych algorytmów (lub wersji tego samego algorytmu).

- Listingi zmodyfikowanych funkcji (lub ich najistotniejszych fragmentów) z **wyraźnie wyróżnionymi** własnymi modyfikacjami. Zmodyfikowane funkcje powinny być najpierw przetestowane pod względem **poprawności** działania, zaś wpływ modyfikacji na działanie (czas wykonania, złożoność obliczeniową i pamięciową itp.) powinien zostać opisany.

Dodatek

Poniżej opisane są w skrócie ważniejsze pliki, klasy i funkcje tworzące programy *minpath* i *mst*.

Program *minpath* – najważniejsze klasy

AbstractGraph – klasa abstrakcyjna opisująca interfejs konkretnych klas reprezentujących graf skierowany. Dzięki polimorfizmowi funkcje operujące na różnych reprezentacjach grafów mogą mieć wspólny interfejs, co znacznie upraszcza kod korzystający z tych funkcji.

MatrixDigraph – klasa dziedzicząca po *AbstractGraph*. Implementacja grafu skierowanego jako macierzy przyległości (sąsiedztwa).

ListDigraph – klasa dziedzicząca po *AbstractGraph*. Jest to również graf skierowany, lecz zaimplementowany jako `m`-elementowa tablica wskaźników na listy (klasa *List*) krawędzi (wskaźników na obiekty klasy *HalfEdge*). Obie klasy są opisane poniżej.

HalfEdge – krawędź opisana parą liczb naturalnych: indeksem wierzchołka końcowego i wagą. Wierzchołek początkowy jest dany niejawnie – odpowiada indeksowi tablicy, pod którym znajduje się wskaźnik na listę, do której należy dany obiekt *HalfEdge*.

List – klasa implementująca listę jednokierunkową obiektów klasy *ListElement*. Obie klasy są szablonami, dzięki czemu element może przechowywać obiekt dowolnego typu (w tym także typy prymitywne, np. *int*). Lista posiada wskaźnik (*current*) na bieżący element, początkowo wskazujący na element pierwszy. Funkcja *getNext*() pobiera wskazywany element (bez usuwania go z listy) i przesuwa wskaźnik o jedną pozycję. Funkcja *hasMoreElements*() pozwala sprawdzić, czy wskaźnik nie wykroczył poza listę, natomiast *rewind*() cofa go na jej początek, co pozwala na kolejne przejście listy. Nowy element można dodać do końca listy funkcją *append*(), której argumentem może być obiekt dowolnej klasy lub typ prymitywny.

Vertex – struktura reprezentująca wierzchołek grafu, wykorzystywana przez algorytm Dijkstry. Zawiera parę liczb naturalnych: *index* – indeks (numer) wierzchołka i *dist* – jego wagę, tzn. odległość od wierzchołka początkowego.

DistanceTable – kolejka priorytetowa, przechowująca informacje o kolejnych wierzchołkach: ich wagi,

czyli odległości od wierzchołka początkowego (tablica *dists*) oraz flagi informujące o tym, czy algorytm Dijkstry traktuje te wagi jako tymczasowe, czy ustalone (tablica *fixed*). Metoda *getNearest*() zwraca indeks (pozycję w tablicy) wierzchołka o najmniejszej spośród nieustalonych wag.

HeapPQ – kolejka priorytetowa wierzchołków zaimplementowana z użyciem kopca. Kluczem jest odległość danego wierzchołka od wierzchołka początkowego. Tablica *vertexHeap* przechowuje wskaźniki na struktury *Vertex*, przy czym są one uporządkowane jako kopiec minimalny, tzn. korzeń zawiera wskaźnik na element o najmniejszym kluczu. Tablica *offsets* zawiera pozycje kolejnych wierzchołków znajdujących się w *vertexHeap*. Odwołanie do *k*-go elementu tablicy *offsets* umożliwia więc znalezienie w kopcu wierzchołka o indeksie *k* w czasie *O*(1). Metoda *removeNearest*() zwraca element o najmniejszym kluczu i usuwa go z kopca. Metoda *decreaseKey*() pozwala na modyfikację klucza wierzchołka o podanym indeksie. Po każdej z tych operacji automatycznie przywracana jest własność kopca.

MinpathMain – główna klasa programu. Zawiera funkcje implementujące cztery wersje algorytmu Dijkstry, algorytm Floyda oraz algorytm przeszukujący całą przestrzeń rozwiązań, tzn. testujący wszystkie możliwe drogi. Te funkcje to:

DijkstraMatrix – implementacja podstawowa. Operuje na grafie zapisanym w postaci macierzy sąsiedztwa (obiekcie klasy *MatrixDigraph*), zaś wagi wierzchołków przechowuje w tablicy (wewnętrznej tablicy obiektu klasy *DistanceTable*).

DijkstraMatrixPQ – również wykorzystuje graf w postaci macierzy sąsiedztwa, lecz tymczasowe wagi wierzchołków przechowuje w kolejce priorytetowej zaimplementowanej jako kopiec – obiekcie klasy *HeapPQ*. Po nadaniu wierzchołkowi wagi ustalonej jest on usuwany z kolejki.

DijkstraList – funkcja operująca na grafie zapisanym w postaci tablicy list krawędzi (obiekcie klasy *ListDigraph*). Wagi wierzchołków są przechowywane w tablicy – identycznie, jak w przypadku funkcji *DijkstraMatrix*.

DijkstraListPQ – operuje na obiekcie klasy *ListDigraph*, natomiast tymczasowe wagi wierzchołków przechowuje w kopcu.

Floyd – typowa implementacja, operująca na macierzy sąsiedztwa. **UWAGA!** Funkcja nadpisuje macierz przyległości grafu wejściowego, umieszczając w niej ostatecznie macierz odległości między wierzchołkami. Dlatego do badania tej funkcji potrzebna jest osobna instancja grafu.

DFS – funkcja znajdująca wszystkie możliwe drogi w grafie poprzez przeszukiwanie w głąb.

Dzięki temu, że obie reprezentacje grafu (tablicowa i listowa) dziedziczą po *AbstractGraph*, wszystkie wersje algorytmu Dijkstry mają wspólny interfejs: otrzymują wskaźnik na *AbstractGraph*, indeksy wierzchołków początkowego i końcowego oraz flagę *findAllPaths* decydującą, czy mają być szukane najkrótsze drogi między zadaną parą, czy między wszystkimi parami wierzchołków. Identyczny interfejs dla funkcji *Floyd* i *DFS* zapewniają funkcje „opakowujące” *RunFloyd* i *RunDFS*.

Dodatkowo plik zawiera funkcje pomocnicze: *FindShortestPathDijkstra*() oraz *FindShortestPath*() dla

algorytmów Floyda i DFS. Pozwalają one na pomiar czasu wykonania algorytmu i wyświetlenie informacji podanych na wstępie tego punktu: komentarzy, czasu wykonania i długości znalezionej drogi (dróg).

Program *mst* – najważniejsze klasy i pliki

Klasa *Forest* – implementuje abstrakcyjną strukturę danych przechowującą rozłączne podzbiory, a znaną powszechnie jako Disjoint Subsets lub Union-Find. Dla struktury takiej zdefiniowane muszą być dwie standardowe operacje: określenia podzbioru, do którego należy dany element (Find) oraz łączenia dwóch podzbiorów (operacja Union). W przypadku algorytmów MST wspomniane podzbiory odpowiadają tworzonym poddrzewom, zaś ich elementami są wierzchołki grafu. Sama struktura jest zaś niezbędna, by podczas dodawania do MST kolejnych gałęzi ustrzec się przed powstaniem cyklu. Takie „bezpieczne" dodawanie gałęzi realizuje metoda *addEdge*(*edge*). Operacja ta kończy się powodzeniem tylko wtedy, gdy spowoduje połączenie dwóch drzew – w skrajnym przypadku składających się z pojedynczych wierzchołków. W przeciwnym przypadku funkcja zwraca wartość *false*, a gałąź *edge* nie zostaje dodana, gdyż spowodowałoby to powstanie cyklu w którymś z istniejących drzew.

Klasa *Edge* – krawędź grafu nieskierowanego jako trójka liczb naturalnych: indeksy wierzchołków i waga krawędzi.

Klasa *Graph* – implementacja grafu nieskierowanego jako tablicy wskaźników na obiekty typu *Edge.*

Plik MST_Algorithms – zawiera funkcje realizujące algorytmy Kruskala i Borůvki. Każda z tych funkcji operuje na obiekcie klasy Graph i zwraca wskaźnik na tablicę wskaźników do obiektów klasy Edge – krawędzi wchodzących w skład znalezionego drzewa.

Plik *MST_Utils* – zawiera m.in. funkcje *FindMST* i *PrintMST.* Pierwsza z nich „opakowuje" algorytm MST (przekazany jako wskaźnik na odpowiednią funkcję), mierząc czas jego wykonania i wyświetlając informacje dodatkowe. Może także wypisywać stworzone drzewo krawędź po krawędzi (za pomocą *PrintMST*), jeśli użytkownik wywoła ją z ustawioną flagą *printTree* i jeśli liczba krawędzi drzewa MST nie przekracza MAX_PRINTABLE_E.

Ostatnia modyfikacja 15.05.2018

Opracował dr inż. Dominik Kasprowicz

dkasprow@imio.pw.edu.pl