Dział: Projektowanie efektywnych algorytmów	**Data wykonania ćwiczenia** 31.01.2023
Prowadzący ćwiczenie: *Dr Beata Laszkiewicz*	**Data oddania sprawozdania**
Wykonujący ćwiczenie: • Marek Wala 262841	
Tytuł ćwiczenia: BADANIE EFEKTYWNOŚCI ALGORYTMÓW GRAFOWYCH W ZALEŻNOŚCI OD ROZMIARU INSTANCJI I SPOSOBU PAMIĘTANIA GRAFU	
Uwagi prowadzącego •	

1. Treść zadania:

Należy zaimplementować oraz dokonać pomiaru czasu działania wybranego algorytmu grafowego rozwiązującego następujący problem: Wyznaczanie najkrótszej ścieżki w grafie – algorytm Dijkstry (max. liczba punktów: 15 z użyciem gotowych struktur, 20+ z użyciem samodzielnie napisanych struktur). Algorytm należy zaimplementować dla obu poniższych reprezentacji grafu w pamięci komputera: - reprezentacja macierzowa, - reprezentacja listowa.

2. Podstawy teoretyczne:

Dijkstra to algorytm wyznaczania najkrótszej ścieżki pomiędzy dwoma wierzchołkami w grafie skierowanym lub nieskierowanym, w którym wagę każdej krawędzi można interpretować jako odległość. Algorytm opiera się na idei użycia kolejki priorytetowej do przechowywania wierzchołków i aktualizowania odległości do nich w czasie rzeczywistym. Kroki algorytmu Dijkstry:

- Ustaw początkowy wierzchołek jako "odwiedzony" i ustaw jego odległość jako 0.
- Dodaj wszystkie sąsiednie wierzchołki do kolejki priorytetowej, a ich odległość ustaw jako suma odległości do początkowego wierzchołka i wagi krawędzi łączącej te wierzchołki.
- Pobierz wierzchołek z kolejki priorytetowej o najniższej wartości i ustaw jako "odwiedzony".
- Dla każdego sąsiada "nieskończonego" wierzchołka, oblicz sumę odległości do bieżącego wierzchołka i wagi krawędzi łączącej te wierzchołki, a następnie aktualizuj odległość do tego wierzchołka, jeśli jest ona mniejsza niż jego obecna wartość.
- Powtórz kroki 3-4, aż wszystkie wierzchołki zostaną odwiedzone.
- Gdy wszystkie wierzchołki zostały odwiedzone, odległość do wszystkich wierzchołków jest znana i można znaleźć najkrótszą ścieżkę.

Algorytm Dijkstry jest efektywny, ponieważ korzysta z kolejki priorytetowej, co zapewnia logarytmiczny czas wyszukiwania najmniejszej wartości.

3. Kod implementacji jest dostępny i zawiera dużą ilość komentarzy. W czasie implementacji algorytmów napotkałem wiele problemów. Wykorzystane biblioteki standardowe: iostream, fstream, string, vector oraz algorithm.

W programie zdefiniowano następujące struktury danych:

- **Krawedz** - przechowuje informację o połączeniu dwóch wierzchołków w grafie i ich wadze
- **GrafMacierz** - reprezentacja grafu przy użyciu macierzy sąsiedztwa. Składa się z liczby wierzchołków, liczby krawędzi i wektora wektorów, który przechowuje informację o wagach pomiędzy wierzchołkami
- **GrafLista** - reprezentacja grafu przy użyciu listy sąsiedztwa. Składa się z liczby wierzchołków, liczby krawędzi i wektora wektorów, który przechowuje informację o połączeniach pomiędzy wierzchołkami

Program zawiera następujące funkcje:

- **wczytajGrafMacierz** - wczytuje dane grafu z pliku i zwraca jego reprezentację w postaci macierzy sąsiedztwa
- **wczytajGrafLista** - wczytuje dane grafu z pliku i zwraca jego reprezentację w postaci listy sąsiedztwa
- **wygenerujGrafMacierz** - generuje graf z losowymi połączeniami i zwraca jego reprezentację w postaci macierzy sąsiedztwa
- **wygenerujGrafLista** - generuje graf z losowymi połączeniami i zwraca jego reprezentację w postaci listy sąsiedztwa
- **dijkstraMacierz i dijkstraLista** to implementacje algorytmu Dijkstry w przypadku grafów opisanych jako macierz sąsiedztwa i lista sąsiedztwa odpowiednio. Funkcja otrzymuje jako argumenty graf, punkt początkowy i punkt końcowy, a następnie szuka najkrótszej ścieżki pomiędzy nimi. W obu przypadkach najpierw ustawiana jest odległość od wierzchołka początkowego na 0, a następnie w głównej pętli algorytmu w każdym kroku znajdujemy najbliższy nieodwiedzony wierzchołek i aktualizujemy odległości do sąsiednich wierzchołków. Gdy nie ma już nieodwiedzonych wierzchołków, pętla jest zakończona. Następnie kolejność wierzchołków na znalezionej ścieżce jest uzupełniana, a następnie odwracana i zwracana jako wynik funkcji.

Przykładowe problemy:
- kod z funkcją relaks nie zawsze zwracał poprawne wyniki
- środowisko programistyczne nie chciało współpracować z plikami, błąd ostream

4. Wyniki czasowe dla poszczególnych reprezentacji danych oraz różnej ich liczbie i zagęszczeniu.

Porównanie listowej i macierzowej reprezentacji grafu

Rozmiar Grafu	Gęstość	Macierzowa Reprezentacja	Listowa Reprezentacja
100	25%	458939 ns	311310 ns
100	50%	538392 ns	384371 ns
100	75%	500772 ns	449439 ns
100	90%	473371 ns	493602 ns
150	25%	1068924 ns	737549 ns
150	50%	1178024 ns	837724 ns
150	75%	1077528 ns	990097 ns
150	90%	1027036 ns	1063742 ns
200	25%	1717124 ns	1112659 ns
200	50%	1980689 ns	1427546 ns
200	75%	1852236 ns	1682314 ns
200	90%	1785767 ns	1838814 ns
300	25%	3791951 ns	2371719 ns
300	50%	4494314 ns	3140059 ns
300	75%	4216268 ns	3727147 ns
300	90%	4066446 ns	4257522 ns
500	25%	10607639 ns	6639522 ns
500	50%	12382927 ns	8786918 ns
500	75%	12333039 ns	11122052 ns
500	90%	12252661 ns	13188237 ns

5. Środowisko testowe:
- Środowisko programistyczne Xcode system MacOs
- i7 z 2017r.

6. Wnioski:
- W czasie wykonywania operacji na grafie reprezentowanym jako macierz, średni czas wzrasta wraz z wzrostem gęstości grafu.
- Na grafie reprezentowanym jako lista, średni czas wzrasta wraz ze wzrostem gęstości grafu, ale jest mniejszy niż średni czas dla macierzowej reprezentacji grafu.
- Średni czas rośnie wraz ze wzrostem rozmiaru grafu.
- W każdym badanym przypadku, średni czas dla listowej reprezentacji grafu był mniejszy niż średni czas dla macierzowej reprezentacji grafu.

7. Źródła:
Wikipedia, pseudokody z Politechniki Śląskiej autor nieznany