POLITECHNIKA WROCŁAWSKA **Wydział Informatyki i Telekomunikacji**	
Algorytmy i złożoność obliczeniowa	
Rok akademicki:	2024/2025
Autor projektu:	Daniel Gościński
Nr indeksu:	280878
Temat projektu:	BADANIE EFEKTYWNOŚCI ALGORYTMÓW GRAFOWYCH W ZALEŻNOŚCI OD ROZMIARU INSTANCJI I SPOSOBU PAMIĘTANIA GRAFU

Oświadczenie: Przekazując to sprawozdanie do oceny prowadzącemu zajęcia Autor oświadcza, że zostało ono przygotowane samodzielnie, bez udziału osób trzecich oraz że żadna jego część nie jest plagiatem

1. Spis treści

2. Wstęp teoretyczny

Graf jest strukturą matematyczną opisującą relację między obiektami. Składa się on z wierzchołków połączonych krawędziami. W ten sposób powstaje relacja między dwoma wierzchołkami. Wyróżniamy grafy skierowane oraz nieskierowane, różniące się rodzajem krawędzi. W przypadku grafów skierowanych krawędź posiada wyznaczony kierunek, wymuszając kierunek poruszania się w grafie. W przypadku grafów nieskierowanych po każdej krawędzi można poruszać się obustronnie. Na potrzeby projektu wykorzystane będą tylko grafy skierowane, w których każda krawędź posiada swoją wagę, którą można traktować jako cenę przejścia pomiędzy wierzchołkami. Istotnym pojęciem jest również pojęcie gęstości, oznaczająca stosunek liczby krawędzi do maksymalnej możliwej ilości krawędzi (wyrażana w procentach).

Następną rzeczą, jaką należy rozważyć, jest sposób zapisania grafu. W tym celu istnieją określone struktury pozwalające na sprawne poruszanie się programu po grafie. Na potrzeby projektu wyróżnione zostały dwa z nich, które zostały wykorzystane do utworzenia programu:

- Macierz sąsiedztwa: macierz kwadratowa o rozmiarze VxV (gdzie V oznacza liczbę wierzchołków). Implementowana poprzez wektor wektorów (lub tablicę tablic), w wyniku czego jesteśmy w stanie określić wierzchołki krawędzi poprzez odpowiednie indeksy, a wartość pod danymi indeksami oznaczać będzie wagę krawędzi. Komórki, które nie posiadają żadnej krawędzi, zawierają wartość umowną -1. W ten sposób przykładowo wartość 20 pod indeksami [1][5] oznaczać będzie krawędź między punktami 1 i 5 o wadze 20. Jest to struktura prosta w implementacji, jednak posiada ona widoczne wady. Największą wadą jest zapotrzebowanie na pamięć, która wynika z budowy struktury. Graf posiadający 100 wierzchołków będzie potrzebował aż 10 000 komórek pamięci, co może okazać się zbędne przy grafach o niskiej gęstości. Przykładowo graf 100 wierzchołkowy posiadający tylko 10 krawędzi oznaczać będzie tylko 10 niepustych komórek na 10 000, co oznacza dużą ilość dziur w strukturze, które i tak muszą istnieć.
- Lista sąsiedztwa: jest to wektor wektorów przechowujący pary dwóch liczb całkowitych. Pierwszy wymiar struktury jest wektorem o stałej długości V, a drugi wymiar to wektor o niezainicjalizowanej długości, który jest dynamicznie zwiększany, kiedy istnieje krawędź. W przypadku istnienia krawędzi wektor przechowuje parę {u,waga}, gdzie u oznacza drugi wierzchołek. W ten sposób możemy uzyskać krawędź, odczytując indeks wektora oraz zawartość pary. Przykładowo wartość 1,{5,20) oznacza, że między wierzchołkami 1 i 5 mamy krawędź o wadze 20. To, co należy zauważyć to fakt, że nie mamy określonej kolejności, w której przechowujemy pary w wektorze. Oznacza to że kolejność krawędzi dyktowana jest kolejnością zapisu.

Jeden z algorytmów wykonywanych na grafach jest algorytm Dijkstry. Jego celem jest wyznaczenie najkrótszej odległości pomiędzy wyznaczonym wierzchołkiem a resztą wierzchołków w grafie. Książkowa złożoność czasowa tego algorytmu wynosi O((V+E)logV), gdzie V to ilość wierzchołków a E to ilość krawędzi. Opis działania algorytmu Dijkstry opisany został w sekcji III.

3. Opis działania algorytmu

Przed opisem algorytmu należy zrozumieć struktury wykorzystane do zapisu grafu (Wytłumaczone w sekcji II).

Algorytm Dijkstry jest stosunkowo prosty w zrozumieniu. Należy zaznaczyć, że implementacja algorytmu będzie różnić się w zależności od wykorzystanej struktury grafu (w inny sposób przemieszczamy się po grafie). Poniższe wytłumaczenie opiera się o liście sąsiedztwa, a następnie zaprezentowany zostanie zmieniony pseudokod wykorzystujący macierz sąsiedztwa.

Jako dane wejściowe algorytm przyjmuje graf w postaci listy sąsiedztwa, ilość wierzchołków oraz wierzchołek startowy, od którego będzie wyznaczana ścieżka do innych punktów. Zaczynamy od utworzenia tablicy zawierającej dystanse i wypełniamy każdą komórkę nieskończonością (oznacza to, że nie mamy najkrótszej odległości do wierzchołka). Ustawiamy dystans[start] na zero (dystans do samego siebie wynosi 0). Tworzymy dodatkowo tablicę poprzedni (w celu odtworzenia najkrótszej ścieżki) oraz odwiedzony (aby oznaczyć odwiedzone wierzchołki). Inicjalizujemy również kolejkę priorytetową przechowującą pary, do której umieszczamy wierzchołek startowy o wadze krawędzi 0. Po początkowej inicjalizacji wykonujemy pętlę dopóki kolejka nie będzie pusta, a w niej wykonujemy kroki:

- Pobierz wierzchołek u o najmniejszym dystansie (wierzchołek U)
- Jeśli był on przetworzony, pomiń go
- Jeśli nie był przetworzony, oznacz go jako przetworzony
- Dla każdego sąsiada V wierzchołka U sprawdź, czy droga przez U do V jest krótsza niż aktualnie znana. Jeśli tak, zaktualizuj dystans[v] i poprzedni[v], a V z nowym dystansem dodaj do kolejki priorytetowej.

Po ukończeniu pętli uzyskamy wypełnione tablice dystans i poprzedni, dystans[v] zawiera najkrótszy dystans od wierzchołka początkowego do V, a tablica poprzedni wykorzystana może zostać do odtworzenia ścieżki.

```
Funkcja Dijkstra(lista, start, n):
    dystans ← tablica n elementów o wartości ∞
    poprzedni ← tablica n elementów o wartości -1
    odwiedzony ← tablica n elementów o wartości false

    dystans[start] ← 0

    pq ← pusta kolejka priorytetowa (min-kopiec)
    dodaj (0, start) do pq  // para (dystans, wierzchołek)

    dopóki pq nie jest puste:
        (d, u) ← usuń element o najmniejszym dystansie z pq
        jeśli odwiedzony[u] = true:
            kontynuuj do następnej iteracji
        oznacz odwiedzony[u] jako true

        dla każdej krawędzi (v, waga) w lista[u]:
            jeśli nie odwiedzony[v] oraz dystans[u] + waga < dystans[v]:
                dystans[v] ← dystans[u] + waga
                poprzedni[v] ← u
                dodaj (dystans[v], v) do pq

    zwróć (dystans, poprzedni)
```

Pseudokod algorytmu Dijkstry dla listy sąsiedztwa

Różnica implementacji wynika z innej struktury danych użytej jako reprezentację grafu

```
Funkcja Dijkstra(macierz, start, n):
    dystans ← tablica n elementów o wartości ∞
    poprzedni ← tablica n elementów o wartości -1
    odwiedzony ← tablica n elementów o wartości false
    dystans[start] ← 0

    Powtarzaj n razy:
      Znajdź nieodwiedzonego wierzchołka u, dla którego dystans[u] jest minimalny
      (jeśli taki nie istnieje, zakończ pętlę)
      odwiedzony[u] ← prawda

      Dla każdego wierzchołka v:
          Jeśli macierz[u][v] > 0 i odwiedzony[v] = fałsz:
              Jeśli dystans[u] + macierz[u][v] < dystans[v]:
                  dystans[v] ← dystans[u] + macierz[u][v]
                  poprzedni[v] ← u
    zwróć (dystans, poprzedni)
```

Pseudokod algorytmu Dijkstry dla macierzy sąsiedztwa

4. Plan eksperymentów

W celu zbadania algorytmu wykonany został eksperyment mierzący czas wykonania algorytmu Dijkstry. W tym celu utworzono funkcję testującą i mierzącą czasy. Badanie modyfikuje ilość wierzchołków grafu {100,500,1000,3000,5000} oraz gęstość grafu {25%,50%,75%}. Dla każdego rozmiaru i gęstości graf jest generowany testowany 20 razy w celu uśrednienia wyników.

5. Warunki badawcze

Badania zostały wykonane na komputerze o następujących podzespołach, następującej wersji systemu operacyjnego. Podano poniżej również wersję wykorzystanego kompilatora C++.

Procesor	AMD Ryzen 5 3600
Karta graficzna	NVIDIA GeForce GTX 1660 SUPER (6GB)
RAM	Crucial 16GB (2x8GB) 3600MHz CL16 Ballistix
Płyta główna	MSI B450-A PRO MAX (MS-7B86)
Windows	10 OS Build 19045.5753
Kompilator	g++ Wersja: (Rev3, Built by MSYS2 project) 13.2.0

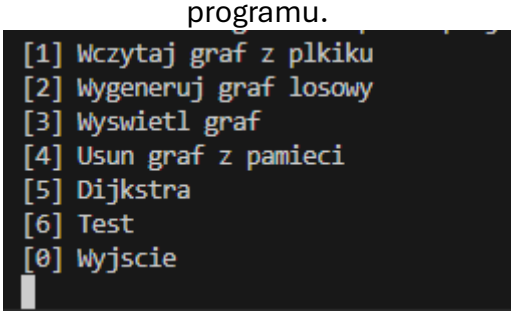
6. Opis programu

Uwaga: Wyniki czasu zapisywane są w pliku „wyniki_testu.txt". Przed uruchomieniem programu zaleca się sprawdzenie, czy pliki takie nie istnieją w folderze, w którym umieszczony jest program, oraz czy nie ma na tych plikach danych. Uruchomienie programu usunie wszystkie informacje znajdujące się w tych plikach.

Program kompilowany był przy pomocy polecenia konsoli

g++ .\grafy.cpp -o .\grafy.exe -g

Po uruchomieniu programu w konsoli wyświetli się lista możliwych funkcji programu.



Wyjaśnienie poszczególnych funkcji programu:

1. Wczytaj graf z pliku: pozwala na wczytanie grafu z pliku tekstowego. Plik tekstowy powinien znajdować się z tym samym folderze co program. Format pliku tekstowego powinien być następujący
   - W pierwszym wierszu zapisana jest liczba krawędzi oraz liczba wierzchołków (obie liczby rozdzielone są spacją)
   - W dalszych wierszach znajdują się opisy krawędzi zapisane następująco: początek krawędzi, koniec krawędzi, waga (liczby oddzielone spacją, każdy wiersz to jedna krawędź)

   Po wczytaniu pliku program wygeneruje graf w formie listowej oraz macierzowej, i będzie przechowywał go w swojej pamięci. Wczytanie nowego pliku usunie poprzedni graf i utworzy nowy.
2. Wygeneruj graf losowy: pozwala na wygenerowanie grafu o zadanej liczbie wierzchołków i zadanej gęstości grafu. Program poprosi o podanie liczby wierzchołków, gęstość grafu oraz maksymalną wagę krawędzi. Po podaniu danych program utworzy graf w obu strukturach. Tak samo jak pierwsza funkcja utworzenie

nowego grafu gdy w pamięci już istnieje graf spowoduje usunięcie poprzedniego obiektu z pamięci i nadpisanie go nowym.

3. Wyświetl graf: Pozwala na wypisanie grafu w postaci macierzy, listy lub obu. Wynik zostanie wyświetlony w konsoli.
4. Usuń graf z pamięci: usuwa istniejący już graf z pamięci programu
5. Dijkstra: pozwala na wykonanie algorytmu Dijkstry na istniejącym grafie w pamięci. Program poprosi o podanie wierzchołka startowego oraz docelowego, na podstawie których program utworzy ścieżkę o najmniejszej wadze.
6. Test: pozwala na wykonanie testu opisanego w sekcji IV

7. Opis implementacji

W tej sekcji opisany zostanie sposób, w jaki zaimplementowano funkcję testową.

Funkcja wykonuje pięciokrotnie pętlę, w której zmieniana jest liczba wierzchołków ze zbioru {100,500,1000,3000,5000}. Maksymalna waga krawędzi ustawiana jest na ilość wierzchołków + 10. Zagnieżdżona została druga pętla wykonująca się trzykrotnie zmieniająca gęstość w zbiorze {25,50,75}. Ostatnia zagnieżdżona pętla wykonuje faktyczny test poprzez poniższe czynności:

- Utwórz graf losowy
- Zmierz czas algorytmu dla macierzy
- Zmierz czas algorytmu dla listy
- Zapisz czasy

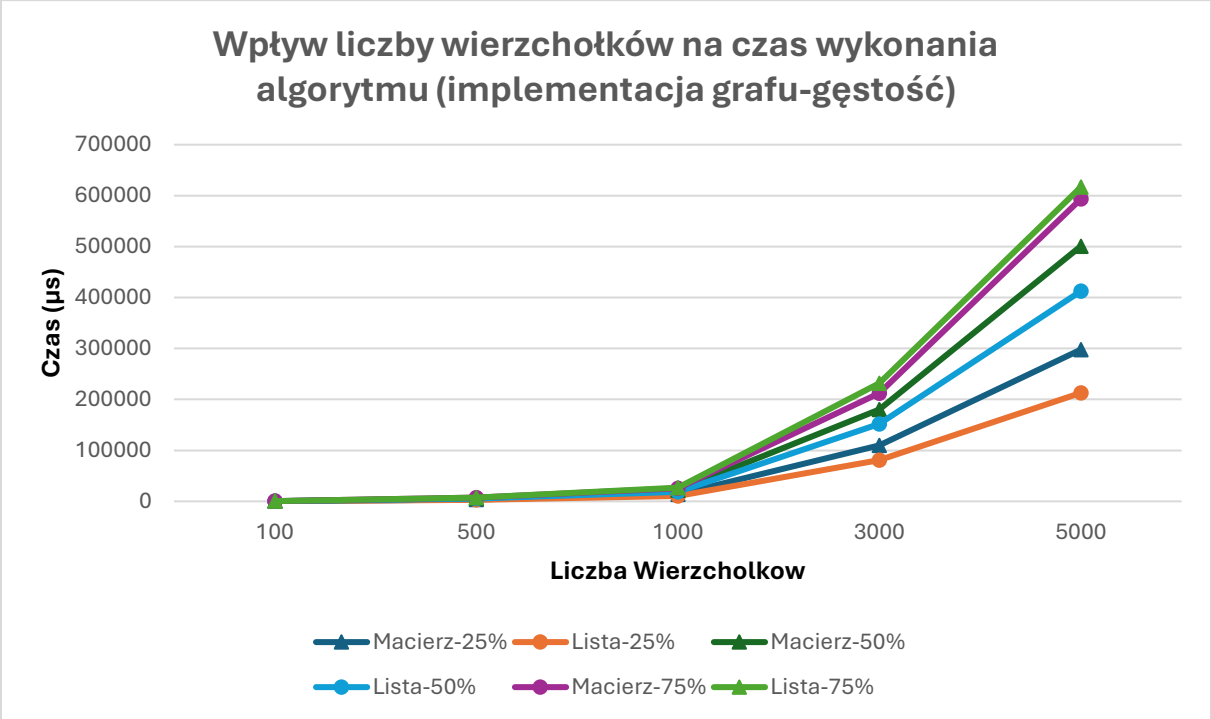
Pętla wykonuje się 20 razy w celu uśrednienia wyniku. Czasy uśrednione następnie są zapisywane do odpowiednich tablic, aby później wypisać wyniki do pliku wyniki_testu.txt

8. Tabele i wykresy

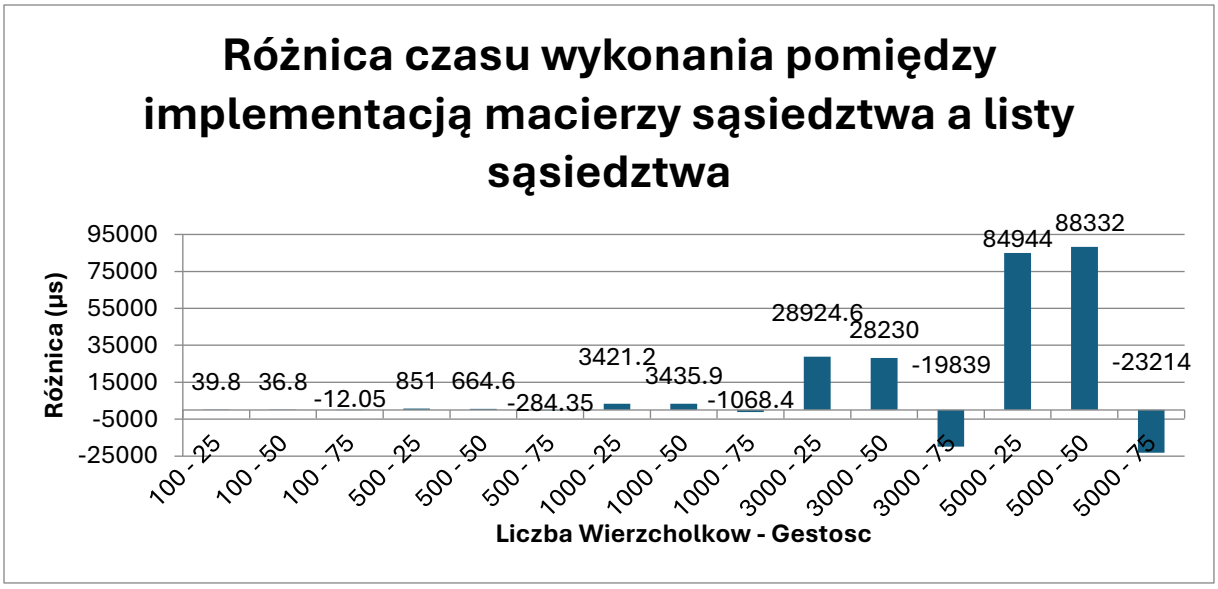
Po wykonaniu testów uzyskano wyniki, które w celu wizualizacji oraz łatwiejszej interpretacji umieszczono w tabelach oraz wykresach.

Liczba Wierzcholkow	Gęstość	Czas Macierz (µs)	Czas Lista (µs)
100	25	256.45	216.65
100	50	373.20	336.40
100	75	457.30	469.35
500	25	3849.30	2998.30
500	50	6017.90	5353.30
500	75	7284.65	7569.00
1000	25	13716.10	10294.90
1000	50	21833.10	18397.20
1000	75	25856.80	26925.20
3000	25	109680.00	80755.40
3000	50	180052.00	151822.00
3000	75	212111.00	231950.00
5000	25	297795.00	212851.00
5000	50	500835.00	412503.00
5000	75	593499.00	616713.00

Tabela 1: Tabela czasu wykonania algorytmu Dijkstry



Wykres 1: Wpływ liczby wierzchołków na czas wykonania algorytmu



Wykres 2: Różnica czasu wykonania pomiędzy implementacją grafu jako macierz sąsiedztwa a listy sąsiedztwa

Wykres 2 zawiera różnicę czasu uzyskanego przy implementacji z macierzą sąsiedztwa a implementacji z listą sąsiedztwa. Różnica ta wskazuje na efektywność jednego algorytmu nad drugim. Wartość dodatnia oznacza, że wersja wykorzystująca listę sąsiedztwa wykonała się szybciej niż wariant używający macierzy.

9. Dyskusja, interpretacja i wnioski

Analizując wyniki i porównując je z oczekiwanymi czasami, możemy powiedzieć, że obie implementacje utworzone na potrzeby programu działają w czasie zgodnym z wiedzą teoretyczną.

Zmiana gęstości grafu wpływa na czas wykonania algorytmu (przy tej samej ilości wierzchołków). W przypadku macierzy sąsiedztwa wpływ gęstości jest mniej zauważalny niż w przypadku listy dla małych grafów, ale jest on bardziej wyraźni dla dużych struktur.

Analizując drugi wykres, możemy zobaczyć różnicę w implementacjach. Przy małych rozmiarach grafu wartości są niezauważalne. Przy rozmiarach 1000,3000,5000 możemy zobaczyć, że dla grafów o stosunkowo małej gęstości (25% i 50%) wersja listowa jest szybsza od macierzowej, za to przy większych gęstościach (75%) macierz zaczyna być szybsza. Wynika to z faktu, że lista traci swoją efektywność przy dużej liczbie krawędzi, ponieważ przeszukanie sąsiadów staje się kosztowne.

Na tej podstawie możemy utworzyć wnioski, że dla algorytmów o dużej ilości wierzchołków i małej gęstości lepszą strukturą do zapisu grafu jest lista sąsiedztwa, za to dla dużej gęstości grafu bardziej opłacalną strukturą jest macierz sąsiedztwa. Przy małych rozmiarach grafów nie ma większego znaczenia w perspektywie opłacalności, więc można kierować się prostszą implementacją.

10. Podsumowanie

Algorytm Dijkstry jest stosunkowo prostym do zrozumienia algorytmem do znalezienia najkrótszej drogi między dwoma wierzchołkami w grafie skierowanym. Przy tworzeniu algorytmu należy zastanowić się nad rozmiarami grafów, na jakich algorytm będzie operował. Na podstawie tej wiedzy możemy dobrać odpowiednią strukturę do przechowywania grafu, w celu uzyskania jak najkrótszego czasu wykonania algorytmu.

11. Literatura

Dijkstra's Algorithm to find Shortest Paths from a Source to all

Notatki i materiały z wykładu „Algorytmy i Złożoność Obliczeniowa”

Applying Dijkstra’s Algorithm in Routing Process (Nitin Gupta, Kapil Mangla,Anand Kumar Jha ,Md. Umar)