Politechnika
Wrocławska

W4N

Badanie efektywności algorytmów grafowych w zależności od rozmiaru instancji oraz sposobu reprezentacji grafu w pamięci komputeraą

Algorytmy i złożoność obliczeniowa - Projekt

Autor:
Filip Kwiek 280947

Termin zajęć:
Wtorek np. 11:15

Prowadzący:
Dr. inż. Jarosław Mierzwa

Spis treści

1 Wstęp

Celem projektu było zbadanie efektywności wybranych grafowych:

- Algorytmów minimalnego drzewa rozpinającego (MST)
- Algorytmów najkrótszej ścieżki
- Algorytmów maksymalnego przepływu

W projekcie dla każdego problemu badam czas wykonania algorytmu w zależności od rozmiaru instancji oraz sposobu reprezentacji grafu w pamięci komputera. Wszystkie algorytmy zostały zaimplementowane w języku C++ i przetestowane na losowo wygenerowanych grafach o różnych gęstościach oraz reprezentacjach.

Wybrane algorytmy grafowe:

- Algorytm Kruskala
- Algorytm Prima
- Algorytm Dijkstry
- Algorytm Bellmana-Forda
- Algorytm Forda-Fulkersona
- Algorytm Edmondsa-Karpa

2 Technologie użyte w projekcie

Projet został wykonany w języku c++ w wersji 20. Do kompilacji użyto kompilatora g++ w wersji 14.2.1, a do zbudowania oraz zarządzania strukturą wykorzystano system CMake w wersji 3.30 Do zapisu oraz odczytu plików wykorzysano bibliotekę *fstream*, natomiast do pomiaru czasu bibliotekę *chrono*.

3 Plan eksperymentu

W ramach przygotowań, cały plan oparty został na założeniu, iż grafy są generowane losowo niezależnie dla każdej reprezetacji, jednocześnie mając stały rozmiar 1000 węzłów.

Eksperyment będzie polegał na zmierzeniu czasu wykonywania algorytmu na 5 reprezentatywnych gęstościach grafów:

- 20%
- 40%
- 60%
- 80%
- 100%

Oraz na 2 reprezentacjach grafów:

- Lista sąsiedztwa
- Macierz sąsiedztwa

Dla każdej gęstości oraz reprezentacji grafu zostało wykonane 10 pomiarów czasu, a dane obejmują ich średnią z każdego pomiaru.

Podczas pomiarów, dane były zapisywane do plików csv. Ze względu na ich ilość, nie zostały one wrzucone razem z kodem, ale istnieje możliwość ich samodzielnej generacji w programie. Przy opisie grafów, nie uwzględniam ich złożoności pamięciowej ponieważ zależy ona od ich reprezentacji, a nie od gęstości grafu.

Złożoności pamięciowe dla obu reprezentacji są następujące:

- Macierz sąsiedztwa: $O(V^2)$, gdzie V to liczba wierzchołków w grafie.
- Lista sąsiedztwa: $O(V + E)$, gdzie E to liczba krawędzi w grafie.

4 Użyty sprzęt

- Procesor: Amd Ryzen 5 3550H 2.1GHz
- Pamięć RAM: 32GB DDR4
- System operacyjny: EndeavourOS (Arch Linux) - kernel 6.14.7-zen2-1-zen

Podczas pomiarów sprzęt był używany na zasilaniu sieciowym, a proces nie był przypisany do jednego rdzenia, przez aplikację *irqbalance*, która rozbija na siłę procesy obciążające jeden rdzeń na wszystkie rdzenie procesora. Ponadto taktowanie procesora nie było ustawione na stałe.

5 Problem MST

Problem minimalnego drzewa rozpinającego (MST) polega na znalezieniu podzbioru krawędzi grafu, który tworzy drzewo rozpinające i ma najmniejszą możliwą sumę wag krawędzi. W kontekście grafów, drzewo rozpinające to podgraf, który zawiera wszystkie wierzchołki grafu i jest drzewem (czyli nie ma cykli). Minimalne drzewo rozpinające to takie drzewo, które ma najmniejszą możliwą sumę wag krawędzi spośród wszystkich możliwych drzew rozpinających. Istnieją trzy popularne algorytmy do rozwiązania tego problemu:

- Algorytm Kruskala
- Algorytm Prima
- Algorytm Boruvki

W moim eksperymencie, zastosowałem dwa pierwsze algorytmy, ponieważ są one najpopularniejsze i najczęściej stosowane w praktyce.

5.1 Algorytmn Kruskala

Algorytm Kruskala jest jednym z najpopularniejszych algorytmów do znajdowania minimalnego drzewa rozpinającego w grafie. Działa on na zasadzie sortowania krawędzi grafu według ich wag i dodawania ich do drzewa rozpinającego, o ile nie tworzą cyklu. Algorytm ten jest szczególnie efektywny w przypadku grafów rzadkich.

- Złożoność czasowa: $O(E \log E)$, gdzie E to liczba krawędzi w grafie.

Moja imlementacja zakłada użycie struktury setów rozłącznych do sprawdzania cykli, co pozwala na efektywne zarządzanie zbiorami wierzchołków.

5.1.1 Lista kroków

1. Posortuj krawędzie grafu według ich wag.
2. Inicjalizuj zbiór rozłącznych zbiorów dla każdego wierzchołka.
3. Iteruj przez posortowane krawędzie:
    - Jeśli krawędź łączy dwa różne zbiory, dodaj ją do drzewa rozpinającego i połącz te zbiory.
4. Zwróć drzewo rozpinające.
5. Zwróć sumę wag krawędzi w drzewie rozpinającym.

5.2 Algorytm Prima

Algorytm Prima jest kolejnym popularnym algorytmem do znajdowania minimalnego drzewa rozpinającego w grafie. Działa on na zasadzie iteracyjnego dodawania krawędzi do drzewa rozpinającego, zaczynając od dowolnego wierzchołka i wybierając zawsze krawędź o najmniejszej wadze, która łączy drzewo z wierzchołkiem spoza drzewa.

- Złożoność czasowa: $O(E \log V)$, gdzie E to liczba krawędzi, a V to liczba wierzchołków w grafie.

Moja implementacja zakłada użycie kopców minimalbych, które pozwalają na efektywne zarządzanie krawędziami i ich wagami. Ulepsza to wydajność algorytmu, ponieważ znajdywanie krawędzi o najmniejszej wadze w najgorszym wypadku będzie wynosić $O(\log V)$, a nie $O(V)$, jak w przypadku prostego algorytmu.

5.2.1 Lista kroków

1. Inicjalizuj drzewo rozpinające jako puste.
2. Wybierz dowolny wierzchołek jako wierzchołek startowy i dodaj go do drzewa rozpinającego.
3. Utwórz kopiec minimalny, który będzie przechowywał krawędzie łączące drzewo z wierzchołkami spoza drzewa.
4. Iteruj, dopóki kopiec nie jest pusty:
    - Wyciągnij krawędź o najmniejszej wadze z kopca.
    - Jeśli krawędź łączy drzewo z wierzchołkiem spoza drzewa, dodaj ją do drzewa rozpinającego i dodaj sąsiadujące krawędzie do kopca.
5. Zwróć drzewo rozpinające.
6. Zwróć sumę wag krawędzi w drzewie rozpinającym.

6 Problem najkrótszej ścieżki

Problem najkrótszej ścieżki polega na znalezieniu najkrótszej ścieżki pomiędzy dwoma wierzchołkami w grafie. Istnieje wiele algorytmów do rozwiązania tego problemu, ale dwa z nich są szczególnie popularne:

- Algorytm Dijkstry
- Algorytm Bellmana-Forda

6.1 Algorytm Dijkstry

Algorytm Dijkstry jest jednym z najpopularniejszych algorytmów do znajdowania najkrótszej ścieżki w grafie z dodatnimi wagami krawędzi. Działa on na zasadzie iteracyjnego znajdowania najkrótszej ścieżki do każdego wierzchołka, zaczynając od wierzchołka źródłowego i aktualizując odległości do sąsiadujących wierzchołków.

- Złożoność czasowa: $O(E \log V)$, gdzie E to liczba krawędzi, a V to liczba wierzchołków w grafie.

Algorytm Dijkstry działa tylko dla grafów z dodatnimi wagami krawędzi. W przypadku grafów z ujemnymi wagami krawędzi, należy użyć algorytmu Bellmana-Forda.

6.1.1 Lista kroków

1. Inicjalizuj odległości do wszystkich wierzchołków jako nieskończoność, a odległość do wierzchołka źródłowego jako 0.
2. Utwórz kopiec minimalny, który będzie przechowywał wierzchołki i ich odległości.
3. Iteruj, dopóki kopiec nie jest pusty:
    - Wyciągnij wierzchołek o najmniejszej odległości z kopca.

- Dla każdego sąsiada tego wierzchołka, jeśli nowa odległość jest mniejsza niż obecna, zaktualizuj odległość i dodaj sąsiada do kopca.

4. Zwróć tablicę odległości do wszystkich wierzchołków.
5. Zwróć najkrótszą ścieżkę do wierzchołka docelowego.

6.2 Algorytm Bellmana-Forda

Algorytm Bellmana-Forda jest algorytmem do znajdowania najkrótszej ścieżki w grafie, który może zawierać krawędzie o ujemnych wagach. Działa on na zasadzie iteracyjnego relaksowania krawędzi grafu, co pozwala na znalezienie najkrótszej ścieżki do każdego wierzchołka.

- Złożoność czasowa: $O(V \cdot E)$, gdzie V to liczba wierzchołków, a E to liczba krawędzi w grafie.

Algorytm Bellmana-Forda jest bardziej uniwersalny niż algorytm Dijkstry, ponieważ może obsługiwać grafy z ujemnymi wagami krawędzi, ale jest mniej wydajny w przypadku grafów z dodatnimi wagami krawędzi.

6.2.1 Lista kroków

1. Inicjalizuj odległości do wszystkich wierzchołków jako nieskończoność, a odległość do wierzchołka źródłowego jako 0.
2. Iteruj $V - 1$ razy (gdzie V to liczba wierzchołków):
   - Dla każdej krawędzi (u, v) z wagą w, jeśli odległość do $u + w$ jest mniejsza niż odległość do v, zaktualizuj odległość do v.
3. Sprawdź, czy istnieją cykle o ujemnej wadze:
   - Jeśli po $V - 1$ iteracjach można jeszcze zaktualizować jakąkolwiek krawędź, to graf zawiera cykl o ujemnej wadze.
4. Zwróć tablicę odległości do wszystkich wierzchołków.
5. Zwróć najkrótszą ścieżkę do wierzchołka docelowego.

7 Problem maksymalnego przepływu

Problem maksymalnego przepływu polega na znalezieniu maksymalnego przepływu w sieci przepływowej, która jest grafem skierowanym z dodatnimi wagami krawędzi. Istnieją dwa popularne algorytmy do rozwiązania tego problemu:

- Algorytm Forda-Fulkersona
- Algorytm Edmondsa-Karpa

7.1 Algorytm Forda-Fulkersona (przeszukiwanie grafu w głąb)

Algorytm Forda-Fulkersona jest jednym z najpopularniejszych algorytmów do znajdowania maksymalnego przepływu w sieci przepływowej. Działa on na zasadzie iteracyjnego znajdowania ścieżek powiększających przepływ w sieci i aktualizowania przepływu, aż nie będzie już możliwe znalezienie kolejnej ścieżki powiększającej.

- Złożoność czasowa: $O(E \cdot f)$, gdzie E to liczba krawędzi w grafie, a f to maksymalny przepływ.

Moja implementacja zakłada użycie algorytmu przeszukiwania w głąb (DFS) do znajdowania ścieżek powiększających przepływ, co pozwala na efektywne zarządzanie przepływem w sieci.

7.1.1 Lista kroków

1. Inicjalizuj przepływ do wszystkich krawędzi jako 0.
2. Iteruj, dopóki istnieje ścieżka powiększająca przepływ:
   - Znajdź ścieżkę powiększającą przepływ w sieci za pomocą algorytmu DFS.
   - Oblicz minimalną pojemność na tej ścieżce.
   - Zaktualizuj przepływ wzdłuż tej ścieżki.
3. Zwróć maksymalny przepływ.

7.2 Algorytm Edmondsa-Karpa (przeszukiwanie grafu wszerz)

Algorytm Edmondsa-Karpa jest ulepszoną wersją algorytmu Forda-Fulkersona, która wykorzystuje algorytm przeszukiwania wszerz (BFS) do znajdowania ścieżek powiększających przepływ. Działa on na zasadzie iteracyjnego znajdowania ścieżek powiększających przepływ w sieci i aktualizowania przepływu, aż nie będzie już możliwe znalezienie kolejnej ścieżki powiększającej.

- Złożoność czasowa: $O(V \cdot E^2)$, gdzie V to liczba wierzchołków, a E to liczba krawędzi w grafie.

Moja implementacja zakłada użycie algorytmu przeszukiwania wszerz (BFS) do znajdowania ścieżek powiększających przepływ, co pozwala na efektywne zarządzanie przepływem w sieci.

7.2.1 Lista kroków

1. Inicjalizuj przepływ do wszystkich krawędzi jako 0.
2. Iteruj, dopóki istnieje ścieżka powiększająca przepływ:
   - Znajdź ścieżkę powiększającą przepływ w sieci za pomocą algorytmu BFS.
   - Oblicz minimalną pojemność na tej ścieżce.
   - Zaktualizuj przepływ wzdłuż tej ścieżki.
3. Zwróć maksymalny przepływ.

8 Przebieg eksperymentu

Dla każdego algorytmu zastosowano dwie opisane wcześniej reprezentacje oraz gęstości grafów. Każdy algorytm był wykonywany na osobno losowym generowanym grafie, niezależnie od reprezentacji. Każdy z nich został uruchomiony 20 razy (10 dla listy sąsiedztwa i 10 dla macierzy sąsiedztwa), a czas wykonania został uśredniony. Wyniki przedstawiono na wykresach, gdzie oś X to rozmiar tablicy, a oś Y to czas wykonania algorytmu w milisekundach.

8.1 Problem MST

8.1.1 Tabele

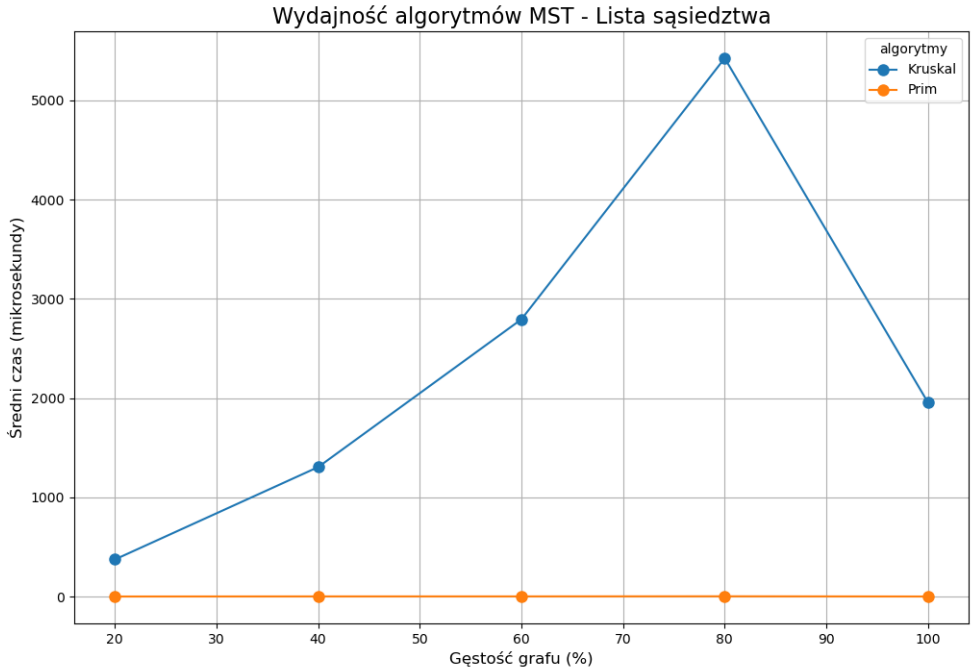
Density	Kruskal	Prim
20	374	1
40	1,305	2
60	2,794	2
80	5,425	3
100	1,956	1

Tabela 1: Tabela średnich wyników w milisekundach dla algorytmu Kruskala i Prima dla Listy sąsiedztwa

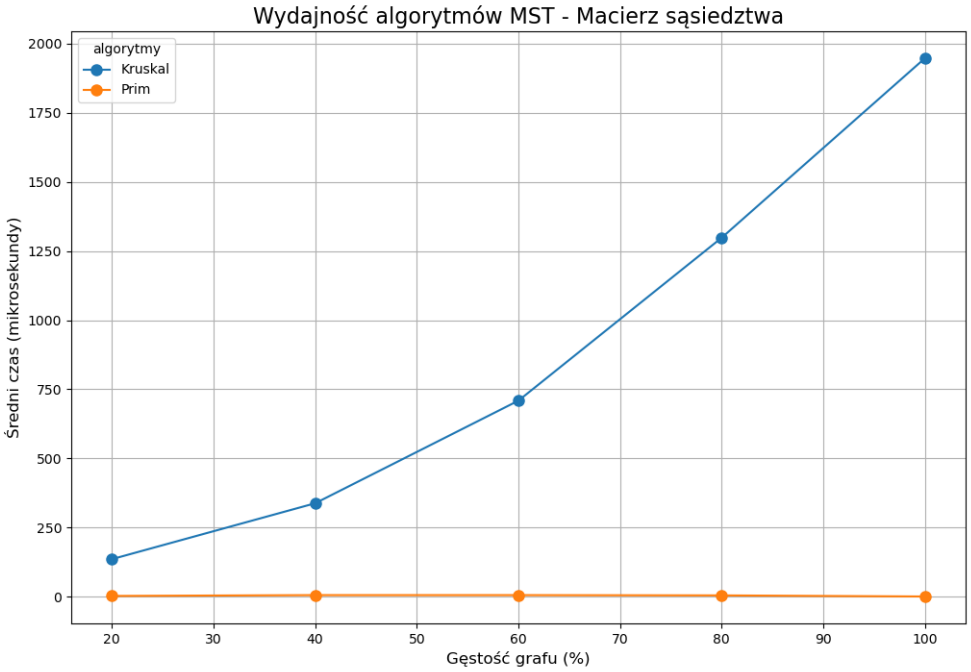
Density	Kruskal	Prim
20	136	3
40	338	6
60	709	6
80	1,298	5
100	1,947	1

Tabela 2: Tabela średnich wyników w milisekundach dla algorytmu Kruskala i Prima dla Macierzy sąsiedztwa

8.1.2 Wykresy Typu 1 dla Kruskala i Prima

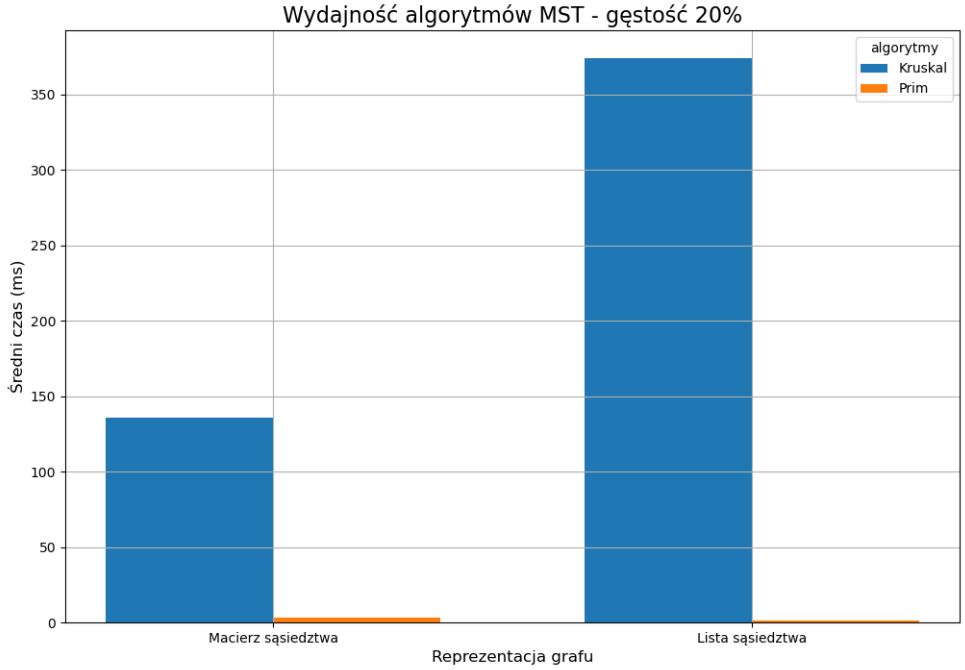


Rysunek 1: Czasy wykonania algorytmów Kruskala i Prima dla Listy sąsiedztwa

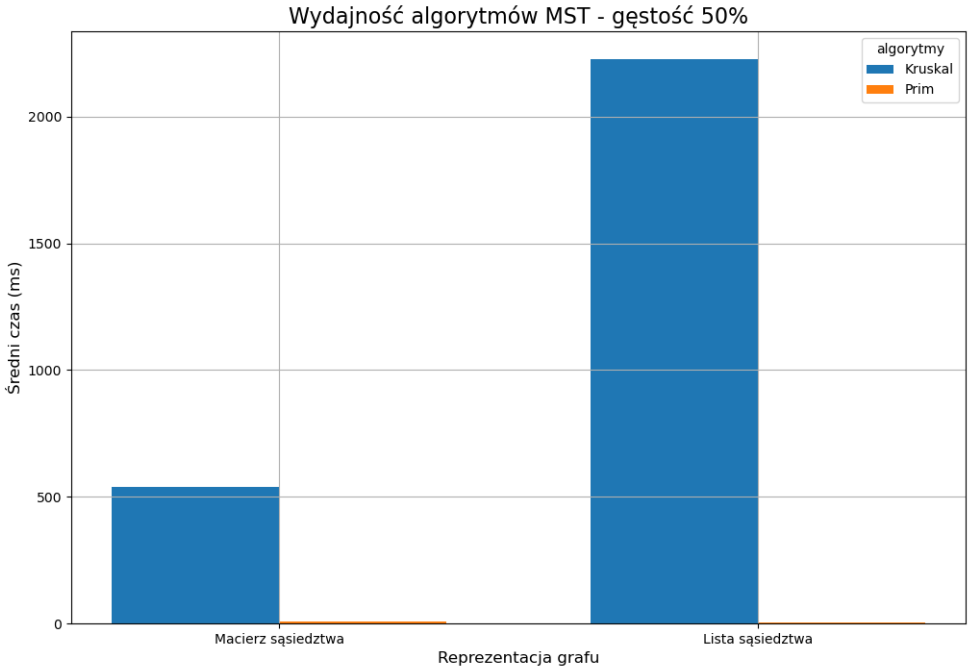


Rysunek 2: Czasy wykonania algorytmów Kruskala i Prima dla Macierzy sąsiedztwa

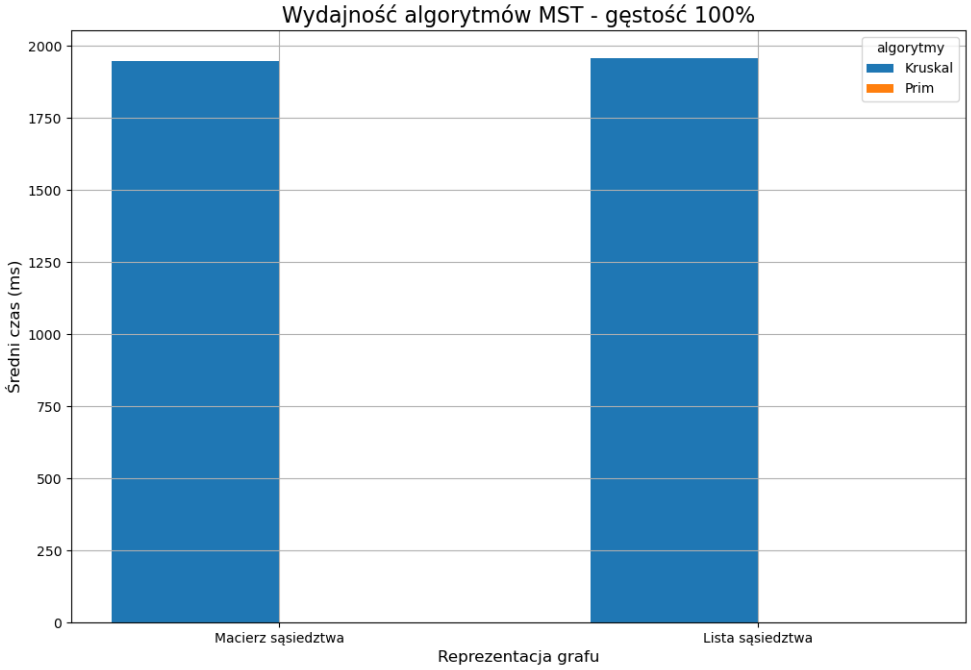
8.1.3 Wykresy Typu 2 dla Kruskala i Prima



Rysunek 3: Czasy wykonania algorytmów Kruskala i Prima dla gęstości 20%



Rysunek 4: Czasy wykonania algorytmów Kruskala i Prima dla gęstości 50%



Rysunek 5: Czasy wykonania algorytmów Kruskala i Prima dla gęstości 100%

8.1.4 Wnioski

Analizując table oraz wykresy możemy zobaczyć drastyczną różnicę w czasie wykonania algorytmu Kruskala i Prima. Algory

8.2 Problem najkrótszej ścieżki

8.2.1 Tabele

Density	Dijkstra	Bellman-Ford
20	0	251
40	1	521
60	1	781
80	1	958
100	1	968

Tabela 3: Tabela średnich wyników w milisekundach dla algorytmu Dijkstry i Bellmana-Forda dla Listy sąsiedztwa

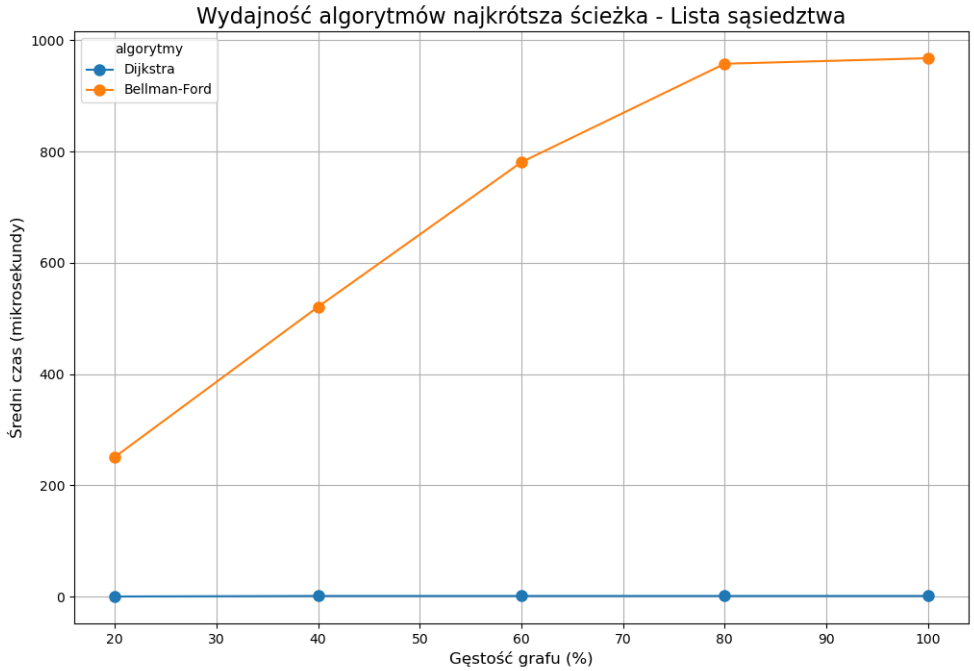
8.2.2 Wnioski

W przypadku sortowania przez kopcowanie, czas wykonania algorytmu rośnie w sposób zbliżony do liniowego i jest również zależny od typu danych. Bardzo podobnie do sortowania szybkiego, czas wykonania przypadku tablicy juz posortowanej jest najkrótszy, a czas wykonania algorytmu dla tablicy posortowanej malejąco jest średnio najdłuższy, ale zbliżony do czasu sortownaia tablicy losowej. W przypadku sortowania przez kopcowanie, czas wykonania algorytmu dla $float$ jest krótszy niż dla int. Na Obu wykresach możemy również zauważyć, że czas wykonania algorytmu dla tablicy posortowanej w 33% jest krótszy niż dla tablicy posortowanej w 66%

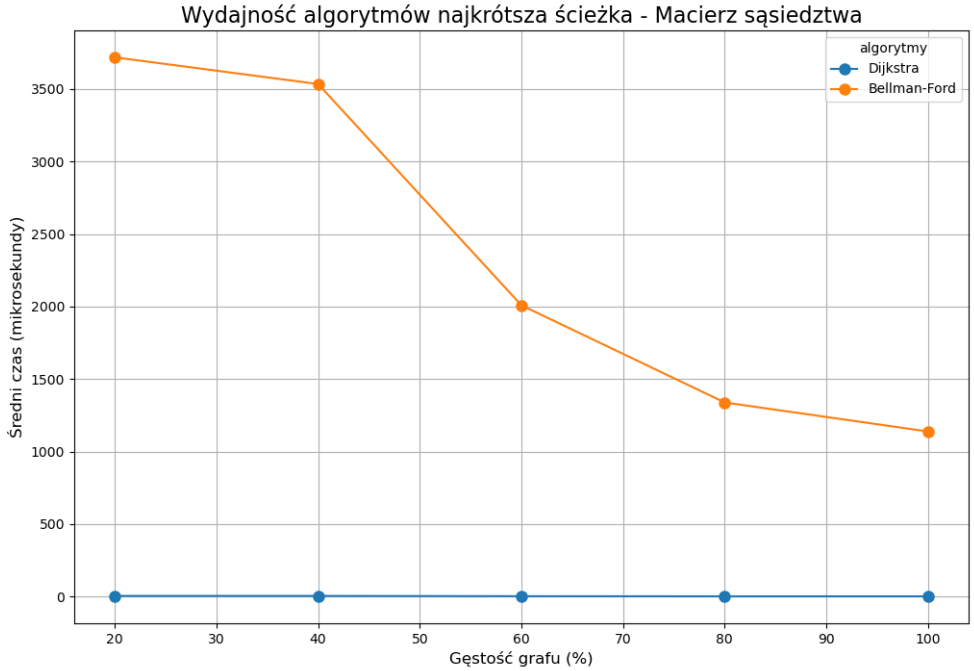
Density	Dijkstra	Bellman-Ford
20	5	3,719
40	5	3,535
60	3	2,009
80	2	1,339
100	2	1,139

Tabela 4: Tabela średnich wyników w milisekundach dla algorytmu Dijkstry i Bellmana-Forda dla Macierzy sąsiedztwa

8.2.3 Wykresy Typu 1 dla Dijkstry i Bellmana-Forda

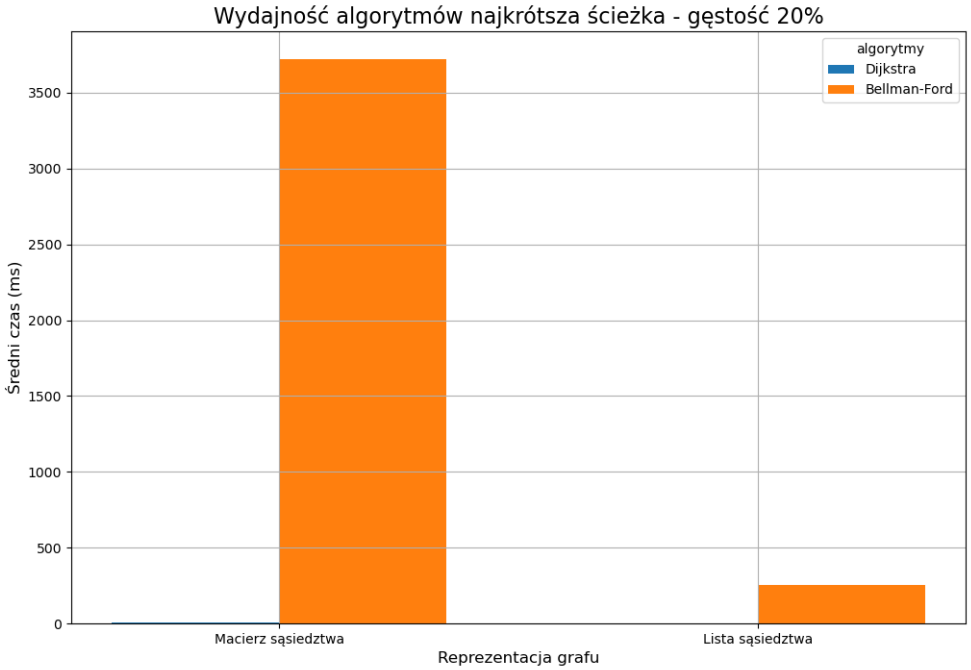


Rysunek 6: Czasy wykonania algorytmów Dijkstry i Bellmana-Forda dla Listy sąsiedztwa

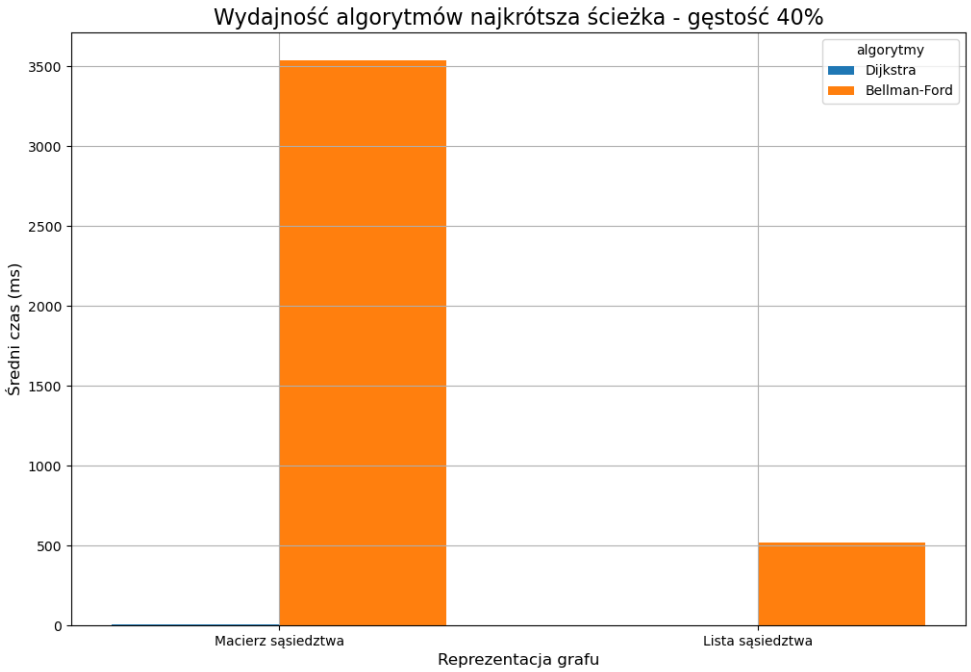


Rysunek 7: Czasy wykonania algorytmów Dijkstry i Bellmana-Forda dla Macierzy sąsiedztwa

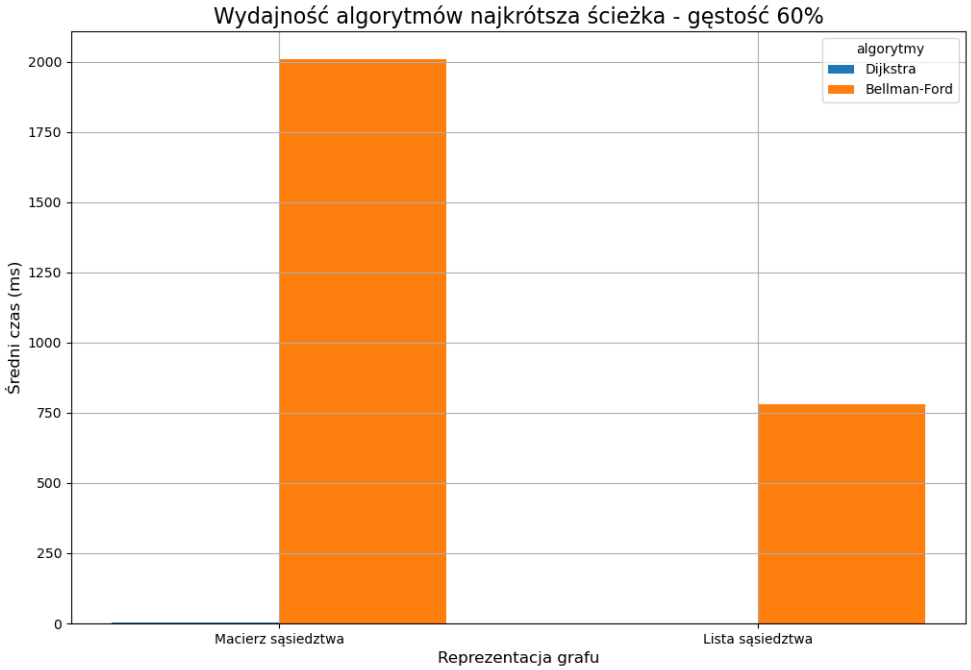
8.2.4 Wykresy Typu 2 dla Dijkstry i Bellmana-Forda



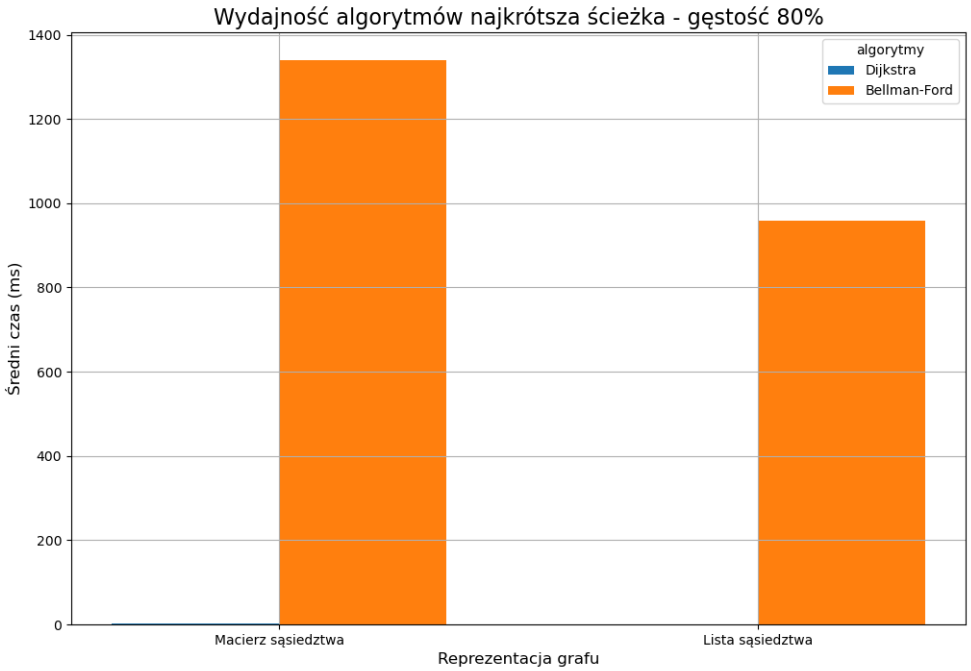
Rysunek 8: Czasy wykonania algorytmów Dijkstry i Bellmana-Forda dla gęstości 20%



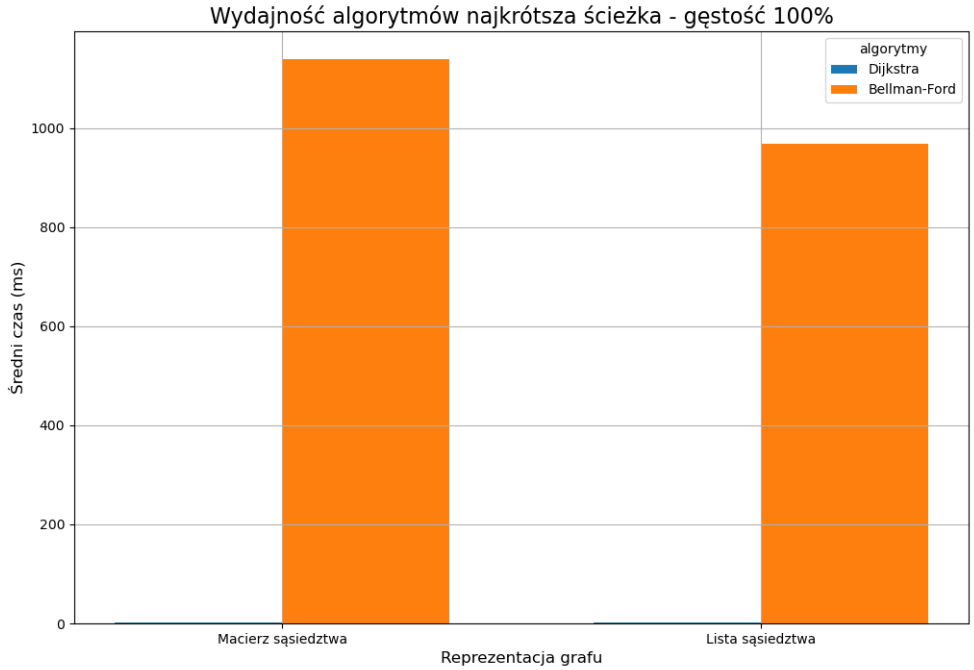
Rysunek 9: Czasy wykonania algorytmów Dijkstry i Bellmana-Forda dla gęstości 40%



Rysunek 10: Czasy wykonania algorytmów Dijkstry i Bellmana-Forda dla gęstości 60%



Rysunek 11: Czasy wykonania algorytmów Dijkstry i Bellmana-Forda dla gęstości 80%



Rysunek 12: Czasy wykonania algorytmów Dijkstry i Bellmana-Forda dla gęstości 100%

8.2.5 Wnioski

W przypadku sortowania przez kopcowanie, czas wykonania algorytmu rośnie w sposób zbliżony do liniowego i jest również zależny od typu danych. Bardzo podobnie do sortowania szybkiego, czas wykonania przypadku tablicy juz posortowanej jest najkrótszy, a czas wykonania algorytmu dla tablicy posortowanej malejąco jest średnio najdłuższy, ale zbliżony do czasu sortownaia tablicy losowej. W przypadku sortowania przez kopcowanie, czas wykonania algorytmu dla $float$ jest krótszy niż dla int. Na Obu wykresach możemy również zauważyć, że czas wykonania algorytmu dla tablicy posortowanej w 33% jest krótszy niż dla tablicy posortowanej w 66%.

8.3 Problem maksymalnego przepływu

8.3.1 Tabele

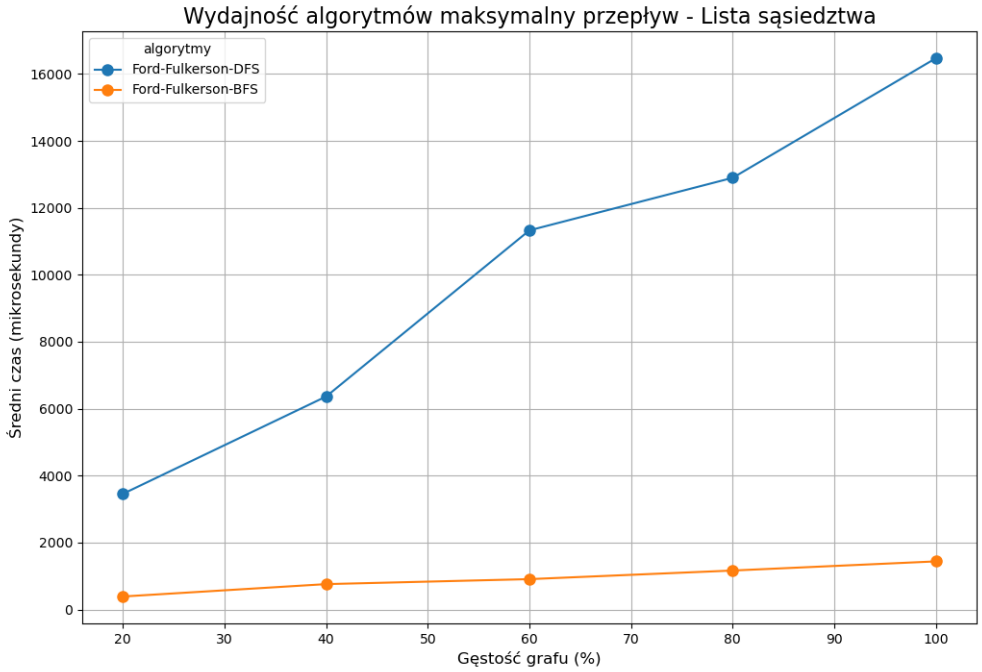
Density	Ford-Fulkerson-DFS	Ford-Fulkerson-BFS
20	3,455	388
40	6,364	758
60	11,328	907
80	12,898	1,164
100	16,472	1,437

Tabela 5: Tabela średnich wyników w milisekundach dla algorytmu Forda-Fulkersona i Edmondsa-Karpa dla Listy sąsiedztwa

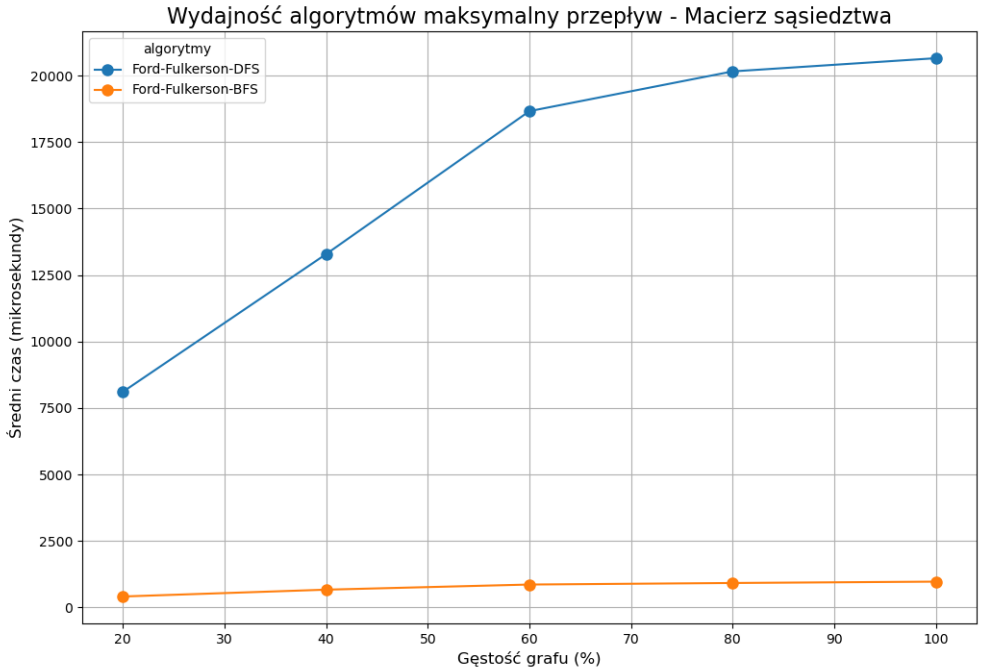
Density	Ford-Fulkerson-DFS	Ford-Fulkerson-BFS
20	8,109	411
40	13,289	668
60	18,665	861
80	20,160	919
100	20,656	970

Tabela 6: Tabela średnich wyników w milisekundach dla algorytmu Forda-Fulkersona i Edmondsa-Karpa dla Macierzy sąsiedztwa

8.3.2 Wykresy Typu 1 dla Forda-Fulkersona i Edmondsa-Karpa

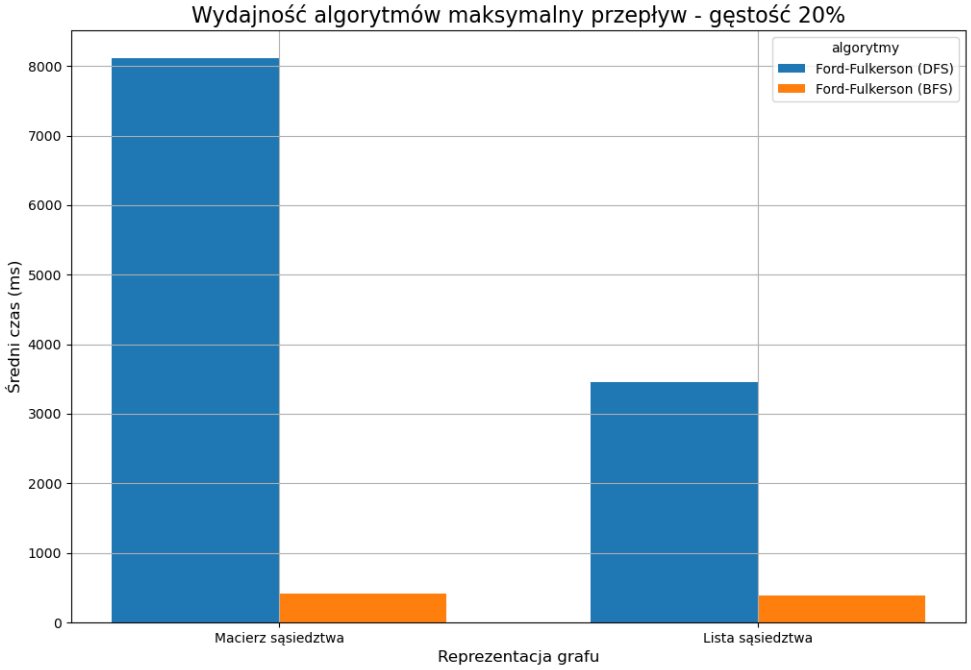


Rysunek 13: Czasy wykonania algorytmów Forda-Fulkersona i Edmondsa-Karpa dla Listy sąsiedztwa

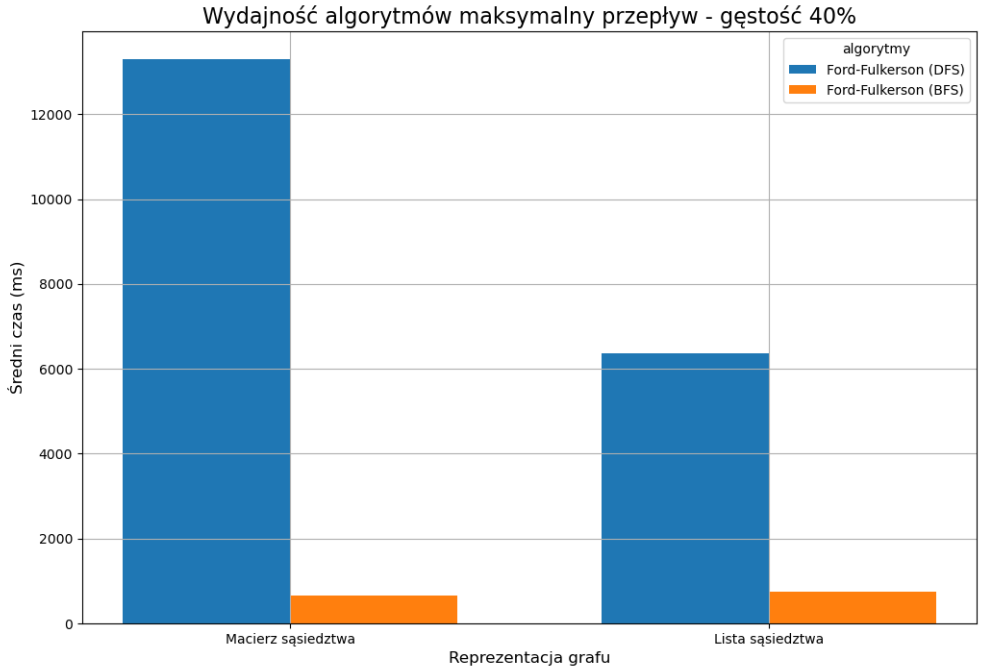


Rysunek 14: Czasy wykonania algorytmów Forda-Fulkersona i Edmondsa-Karpa dla Macierzy sąsiedztwa

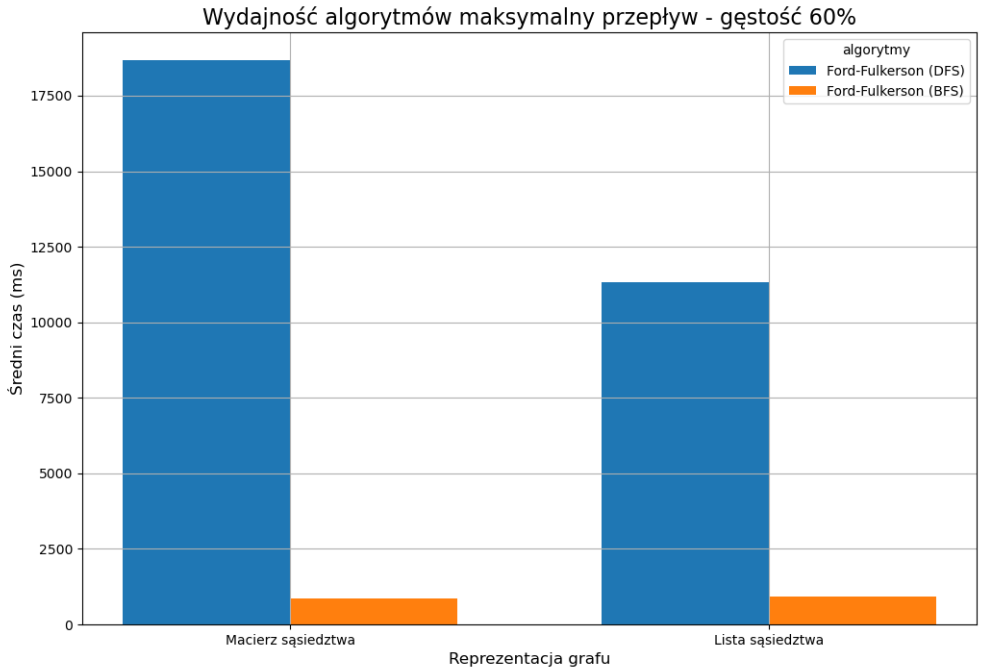
8.3.3 Wykresy Typu 2 dla Forda-Fulkersona i Edmondsa-Karpa



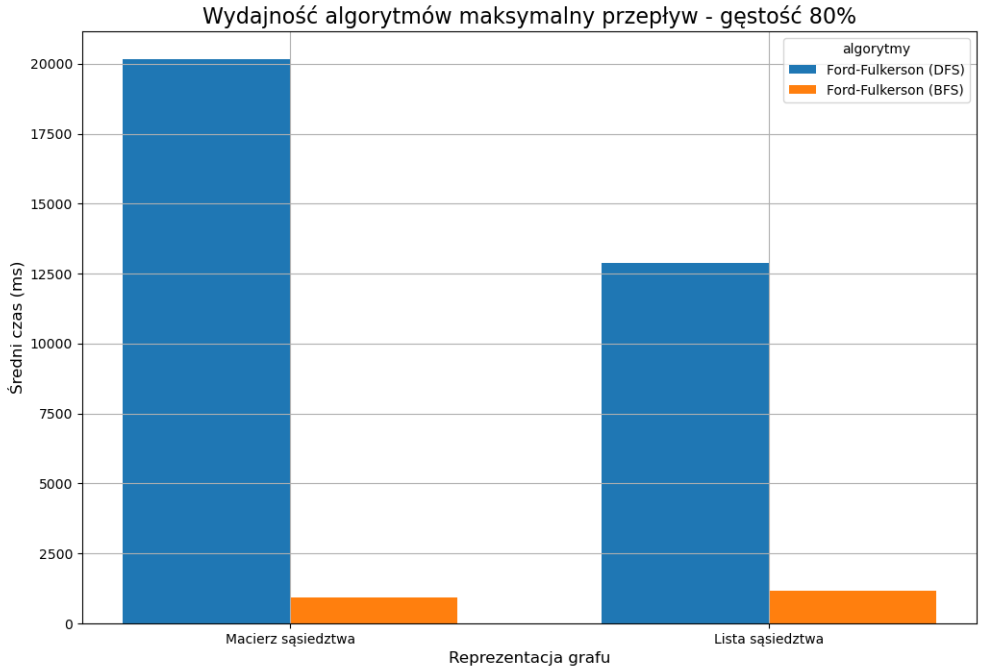
Rysunek 15: Czasy wykonania algorytmów Forda-Fulkersona i Edmondsa-Karpa dla gęstości 20%



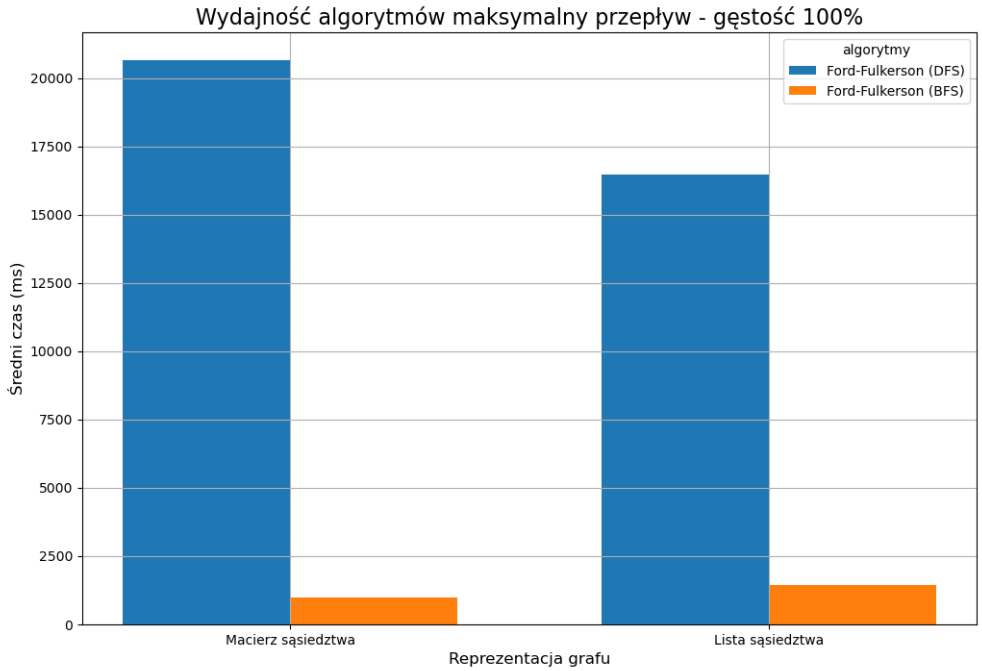
Rysunek 16: Czasy wykonania algorytmów Forda-Fulkersona i Edmondsa-Karpa dla gęstości 40%



Rysunek 17: Czasy wykonania algorytmów Forda-Fulkersona i Edmondsa-Karpa dla gęstości 60%



Rysunek 18: Czasy wykonania algorytmów Forda-Fulkersona i Edmondsa-Karpa dla gęstości 80%



Rysunek 19: Czasy wykonania algorytmów Forda-Fulkersona i Edmondsa-Karpa dla gęstości 100%

9 Podsumowanie

Na podstawie eksperymentu można zauważyć, iż czasy wykonania algorytmów dla różnych reprezentacji oraz gęstości potrafią drastycznie się różnić. W przypadku problemu MST algorytm Prima jest zdecydowanie szybszy niż algorytm Kruskala, co jest spowodowane tym, że algorytm Prima działa w czasie $O(E \log V)$, podczas gdy algorytm Kruskala w czasie $O(E \log E)$. Jednakże ta różnica nie powinna być rzędu setek i może to wynikać z różnych zaburzeń wynikających z metody pomiaru czasu wykonania algorytmu, takich jak obciążenie procesora, czy też inne procesy działające w tle. Podobne różnice widać w przypadku problemów najkrótszej ścieżki oraz maksymalnego przepływu, gdzie algorytm Dijkstry jest szybszy niż algorytm Bellmana-Forda, a algorytm Edmondsa-Karpa jest szybszy niż algorytm Forda-Fulkersona. Te różnice również sięgają podobnej skali co w wypadku problemu MST.

Kolejne ciekawe zjawiska można także zauważyć w wykonaniu algorytmu Kruskala i Prima na różnych reprezentacjach grafu. W przypadku listy sąsiedztwa czas wykonania algorytmu kruskala, czas ten rośnie dla coraz większych gęstości grafu, by potem spaść dla 100% gęstości. Takie zjawisko nie występuje w przypadku macierzy sąsiedztwa, gdzie czas wykonania algorytmu Kruskala rośnie wraz z gęstością grafu. Podobne zjawisko występuje dla algorytmu Bellmana-Forda, gdzie gdy gęstość grafu rośnie, czas wykonania spada.

Wyniki jasno pokazują, które algorytmy sprawdzą się najlepiej w każdym wymienionym problemie i poza przypadkami tzw. granicznymi, w większości będą one używane.

10 Literatura

- Opracowaie tabel: praca własna, dane obrobione przez skrypty w języku *Python*
- Dokumentacja chrono: `https://cplusplus.com/reference/chrono/`
- Dokumentacja fstream `https://cplusplus.com/reference/fstream/`
- Opracowanie algorytmów MST `https://eduinf.waw.pl/inf/alg/001_search/0141.php`
- Opracowanie algorytmów SSP
  `https://eduinf.waw.pl/inf/alg/001_search/0138.php`
  `https://eduinf.waw.pl/inf/alg/001_search/0138a.php`
- Opracowanie algorytmów Maksymalnego przepływu:
  `https://eduinf.waw.pl/inf/alg/001_search/0125.php`
  `https://eduinf.waw.pl/inf/alg/001_search/0126.php`