# Algorytmy i złożoność obliczeniowa

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

Kasper Radom 264023

16 VI 2024

## Spis treści

# 1 Sformułowanie zadania

Celem zadnania jest badanie efektywności algorytmów grafowych w zależności od rozmiaru instancji oraz sposobu reprezentacji grafu w pamięci komputera. Rozwazane algorytmy można podzielić na dwie grupy, algorytmy wyszukujące minimalne drzewo rozpinające; algorytm Prima i Kruskala oraz algorytmy wyszukujące najkrótszą ścieżkę między dwoma wierzchołkami: algorytm Dijkstry i Forda-Bellmana. Każdy algorytm zaimplementowamy jest dla wczytywania grafu osobno z dwóch reprezentacji: macierz incydencji, lista następników.

# 2 Opis reprezentacji

## 2.1 Macierz incydencji

Graf można przedstawić na wiele spowobów. Ludzka percepcja i wyobrażenie o świecie oparte są głównie o zmysł wzroku. Z tego powodu czlowiekowi najłatwiej jest wykonywać operacje na grafie z użyciem reprezentacji graficznej. Komputer jednak niema tego przywileju. Musi mieć zapisaną w pamięci postać składającą się z liczb i zaimplementowamych struktur. Reprezentacja w postaci macierzy incydencji jest jednym z pierwszych przychodzących na myśl, gdy stajemy przed zadaniem przedstawienia grafu w formie czytelnej dla człowieka jak i komputera. Ten kompromis pomaga programiście łatwo zaimplementować tę reprezentacje i łatwo na niej operować. Idea tej reprezentacji polega na stworzeniu listy krawędzi (kolumnu macierzy) oraz wierzchołków (rzędy krawędzi) grafie. W każdej kolumnie, na przecięciu z rzędem przypisanym do wierzchołka incydentnego z krawędzią wpisana jest waga krawędzi. W przypadku grafu skierowanego przy wierzchołku z którego krawędź wychodzi wpisana jest normalna waga natomiast przy wierzchołku do którego krawędź wchodzi wpisana jest ta sama wage, tylko ujemna. W reszcie komurek znajdują są zera lub inne znaki oznaczające, że krawędź nie jest incydentna z wierzchołkiem.

### 2.2 Lista następników

W odpowiedzi na pojawiający wię problem rozmiarów macierzy incydencji, który w przypadku grafu pełnego ma złożoność pamięciową $O(n^3)$, używana jest również lista następników. Jej złożoność pamięciowa już wynosi $O(n^2)$. Każdemu wierzchołkowi przypisane są dwie listy. Pierwsza zawiera następników, druga natomiast wagi krawędzi incydentnych z poszczególnymi wierzchołkami. Kolejność wpisywania elementów do list jest istotna. Dany następnik na pozycji $n$ w liście pierwszej oznacza, że w liście drógiej na tej samej pozycji $n$ znajdziemy wagę krawędzi do niego wiodącej.

## 3 Opis algorytmów

### 3.1 Wyznaczanie minimalnego drzewa rozpinającego (MST)

Minimalne drzewo rozpinające (ang. Minimum Spanning Tree, MST) to podzbiór krawędzi grafu, który łączy wszystkie wierzchołki tego grafu, tworząc drzewo o minimalnej łącznej wadze. Znalezienie MST jest kluczowym problemem w teorii grafów i ma liczne zastosowania w różnych dziedzinach, takich jak sieci komputerowe, projektowanie układów scalonych czy optymalizacja sieci transportowych. W kontekście tego sprawozdania, rozważane są dwa klasyczne algorytmy do wyznaczania minimalnego drzewa rozpinającego: algorytm Prima oraz algorytm Kruskala. Oba algorytmy są fundamentalne w teorii grafów i różnią się podejściem do problemu.

### 3.1.1 Algorytm Prima

Algorytm Prima rozpoczyna od wybranego wierzchołka i w każdym kroku dodaje do drzewa krawędź o minimalnej wadze, która łączy wierzchołek spoza drzewa z wierzchołkiem już w drzewie. Kroki algorytmu są następujące:

1. Inicjalizacja: Wybierz dowolny wierzchołek jako początkowy i dodaj go do drzewa.
2. Znajdź najmniejszą krawędź łączącą wierzchołek w drzewie z wierzchołkiem poza drzewem.
3. Dodaj wybraną krawędź i nowy wierzchołek do drzewa.
4. Powtarzaj kroki 2-3, aż wszystkie wierzchołki będą w drzewie.

Algorytm Prima jest szczególnie efektywny dla grafów o dużej liczbie krawędzi, zwłaszcza gdy implementuje się go z wykorzystaniem kolejki priorytetowej. Jego złośoność obliczeniowa wynosi: $O(|V|^2)$ gdzie $|V|$ jest liczbą wierzchołków rozpatrywanym grafie.

### 3.1.2 Algorytm Kruskala

Algorytm Kruskala działa na zasadzie dodawania krawędzi o najmniejszej wadze do drzewa, unikając przy tym tworzenia cykli. Kroki algorytmu są następujące:

1. Inicjalizacja: Posortuj wszystkie krawędzie grafu w porządku rosnącym według ich wag.
2. Dodaj do drzewa kolejne krawędzie w tej kolejności, upewniając się, że dodanie krawędzi nie utworzy cyklu.
3. Powtarzaj krok 2, aż drzewo będzie zawierać $n-1$krawędzi, gdzie $n$ to liczba wierzchołków.

Algorytm Kruskala jest często efektywny w grafach rzadkich, a jego implementacja wymaga efektywnego zarządzania zbiorami wierzchołków, co można osiągnąć za pomocą struktury danych "Zbiory rozłączne" (ang. Disjoint Sets). Jego złośoność obliczeniowa wynosi: $O(|E|*log(|V|)$ gdzie $|E|$ to liczba krawędzi, a $|V|$ jest liczbą wierzchołków rozpatrywanym grafie.

## 3.2 Wyznaczanie najkrótszej ścieżki w grafie

Problem wyznaczania najkrótszej ścieżki w grafie jest jednym z fundamentalnych zagadnień w teorii grafów. Znalezienie najkrótszej drogi między dwoma wierzchołkami jest istotne w wielu praktycznych zastosowaniach, takich jak nawigacja, sieci komputerowe, analiza dróg i inne.

### 3.2.1 Algorytm Dijkstry

Algorytm Dijkstry służy do znajdowania najkrótszych ścieżek od jednego wierzchołka do wszystkich innych w grafie o nieujemnych wagach krawędzi. Działa on na zasadzie "rozpędzania się" od wierzchołka startowego, wybierając w każdym kroku wierzchołek o najmniejszym koszcie dotarcia i aktualizując koszty dojścia do sąsiadów.

1. Inicjalizacja: Ustaw koszt dojścia do wierzchołka startowego na 0, a do wszystkich innych wierzchołków na nieskończoność lub inną wartość oznaczającą, że wierzchołek nie byl odwiedzony, w przypadku poniższej implementacji, -1. Ustaw wszystkie wierzchołki jako nieodwiedzone.

2. Wybierz nieodwiedzony wierzchołek o najmniejszym koszcie dojścia.

3. Zaktualizuj koszty dojścia do sąsiadów wybranego wierzchołka, jeśli możliwe jest dotarcie do nich krótszą ścieżką przez ten wierzchołek.

4. Oznacz wierzchołek jako odwiedzony.

5. Powtarzaj kroki 2-4, aż wszystkie wierzchołki zostaną odwiedzone lub nie ma już wierzchołków o skończonym koszcie dojścia.

Jego złośoność obliczeniowa wynosi: $O(|E| + |V| * log(|V|)$ gdzie $|E|$ to liczba krawędzi, a $|V|$ jest liczbą wierzchołków rozpatrywanym grafie.

### 3.2.2 Algorytm Bellmana-Forda

Algorytm Bellmana-Forda jest bardziej ogólny niż algorytm Dijkstry, ponieważ pozwala na uwzględnienie krawędzi o ujemnych wagach. Jest on jednak wolniejszy i ma większą złożoność czasową. Algorytm Bellmana-Forda działa na zasadzie relaksacji wszystkich krawędzi grafu w każdym z $n-1$ kroków, gdzie $n$ to liczba wierzchołków.

1. Inicjalizacja: Ustaw koszt dojścia do wierzchołka startowego na 0, a do wszystkich innych wierzchołków na nieskończoność (lub jak w tym przypadku -1)

2. Przez $n-1$ razy zrelaksuj wszystkie krawędzie: dla każdej krawędzi (u, v) o wadze w, jeśli koszt dojścia do v przez u jest mniejszy niż aktualny koszt dojścia do v, zaktualizuj koszt dojścia do v.

3. Sprawdź, czy istnieje cykl o ujemnej wadze: jeśli w $n$-tym kroku możliwa jest dodatkowa relaksacja, graf zawiera cykl o ujemnej wadze.

Jego złośoność obliczeniowa wynosi: $O(|E|*|V|)$ gdzie $|E|$ to liczba krawędzi, a $|V|$ jest liczbą wierzchołków rozpatrywanym grafie.

# 4 Plan eksperymentu

W celu porównania efektywności algorytmów na dwóch reprezentacjach, każdy algorytm wykona się na 7 różnych rozmiarach instancji grafu. W algorytmie Prima i Kruskala testowe tozmiary to: 100, 200, 300, 400, 500, 600 i 700 wierzchołków. Natomiast z racji tego, że wyznaczanie najkrótszej ścieżki okazało się szybszą operacją. Z tego powodu w algorytmie Bellmana-Forda oraz Dijkstry rozmiary grafu to 300, 400, 500, 600, 700, 800 oraz 900 wierzchołków. Eksperymenty zostały również przeprowadzone dla różnych gęstości grafu; 25, 50 i 99 procent wszystkich możliwych krawędzi.

Bardzo czasochłonne okazało się samo generowanie grafu. Z tego powodu plan eksperymentów został tak zmienionym aby dla jednej wygenerowanej instancji grafu o danym rozmiarze i danej gęstości wykonywane były testy dla każdego z odpowiadającej mu grupy algorytmów. Wyniki zapisane są w pliku csv.

## 4.1 Generowania grafu

Algorytm pozwala na tworzenie grafów skierowanych *directed* i nieskierowanych *undirected* o określonej liczbie wierzchołków, gęstości krawędzi i maksymalnej wadze krawędzi. Parametry te podawane są jako argumenty metody *randomGraph*.

Funkcja *randomGraph* rozpoczyna od wyczyszczenia aktualnych reprezentacji grafu za pomocą *clearRepresentations*, a następnie ustawia liczbę wierzchołków na podstawie przekazanego argumentu vertices. W zależności od typu grafu *type*, czyli czy jest on nieskierowany *'U'*, czy skierowany, oblicza liczbę krawędzi *edges* na podstawie gęstości *density*. W przypadku grafu nieskierowanego liczba krawędzi jest obliczana jako $\frac{density}{100.0}\frac{vertices(vertices-1)}{2}$ , a dla grafu skierowanego jako $\frac{density}{100.0} * vertices * (vertices - 1)$. Następnie, funkcja wywołuje *makeMatrix* i *makeList*, aby utworzyć odpowiednie struktury danych do przechowywania macierzy incydencji i listy sąsiedztwa. Proces generowania grafu rozpoczyna się od utworzenia drzewa spinającego *spanning tree*. Tworzone są dwie tablice: *visited* dla odwiedzonych wierzchołków i *unvisited* dla nieodwiedzonych. Na początku, do tablicy *unvisited* dodawane są wszystkie wierzchołki oprócz pierwszego, który dodawany jest do tablicy *visited*. Pętla *while* działa dopóki wszystkie wierzchołki nie zostaną odwiedzone. W każdym kroku losowane są dwa wierzchołki: *from* z tablicy *visited* oraz *to* z tablicy *unvisited*. Jeśli te dwa wierzchołki są różne, losowana jest waga krawędzi (od 1 do *maxWeight*) i dodawana jest krawędź między wierzchołkami za pomocą funkcji *addEdge*. Wierzchołek to jest następnie przenoszony z tablicy *unvisited* do *visited*. Jeśli graf jest skierowany, dla każdej dodanej krawędzi dodawana jest również krawędź w przeciwnym kierunku z losową wagą (od 1 do *maxWeight*), aby zapewnić ścieżki między dowolnymi dwoma wierzchołkami. Po utworzeniu drzewa spinającego, algorytm dodaje resztę krawędzi, aż do osiągnięcia liczby krawędzi określonej przez *density*. Losowane są dwa różne wierzchołki z tablicy *visited*. Jeśli między tymi wierzchołkami nie istnieje jeszcze krawędź, losowana jest waga krawędzi (od 1 do *maxWeight*) i krawędź jest dodawana za pomocą funkcji *addEdge*. Na koniec, tablice *visited* i *unvisited* są usuwane z pamięci, co kończy proces generowania grafu.

```
void Graph::randomGraph(int vertices, int density,
    int maxWeight) {
  clearRepresentations();
  this->vertices = vertices;
  if(type == 'U'){
      this->edges = int(density/100.0 * vertices *
          (vertices-1) / 2);
  }else{
      this->edges = int(density/100.0 * vertices *
          (vertices-1));
  }
  makeMatrix();
  makeList();
  //tworzenie drzewa spinającego
  Table* visited = new Table();
  Table* unvisited = new Table();
  for(int v=1; v<vertices; v++){
      unvisited->push(v);
  }
  visited->push(0);
  int from, to, weight, vertex, edgeId = 0;
  while (unvisited->getSize()!=0) {
      from = rand() % (visited->getSize());
      to = rand() % (unvisited->getSize());
      if(from == to)continue;
      weight = rand() % (maxWeight)+1;
      vertex = unvisited->get(to);
      addEdge(visited->get(from), vertex, weight,
         edgeId);
      edgeId++;
      if(type == 'D'){//podwojenie drzewa aby
         zapewnic sciezki miedzy dwoma dowolnymi
         wierzcholkami
          weight = rand() % (maxWeight)+1;
          addEdge(vertex, visited->get(from),
             weight, edgeId);
          edgeId++;
      }
```

```
            visited->push(vertex);
            unvisited->remove(to);
        }
        //dodawanie reszty krawędzi
        while(edgeId<edges){
            from = rand() % (visited->getSize());
            to = rand() % (visited->getSize());
            if(from != to and list[from][0]->find(to) ==
                -1) {
                weight = rand() % (maxWeight) + 1;
                addEdge(from, to, weight, edgeId);
                edgeId++;
            }
        }

        delete visited;
        delete unvisited;
    }
```

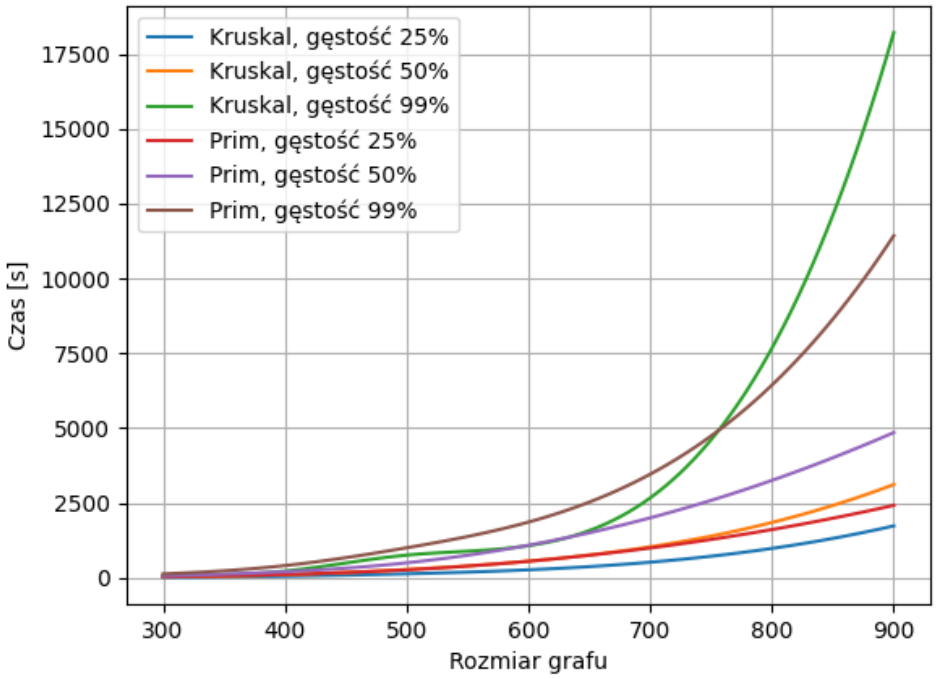
## 4.2 Pomiary czasu

Do pomiaru czasu wykonania algorytmów, powstała klasy *Time*, która implementuje prosty stoper umożliwiający mierzenie czasu w milisekundach. Klasa ta korzysta z wysokorozdzielczego zegara dostępnego w standardowej bibliotece C++ *¡chrono¿*. Poniżej znajduje się szczegółowy opis sposobu pomiaru czasu przy użyciu tej klasy:

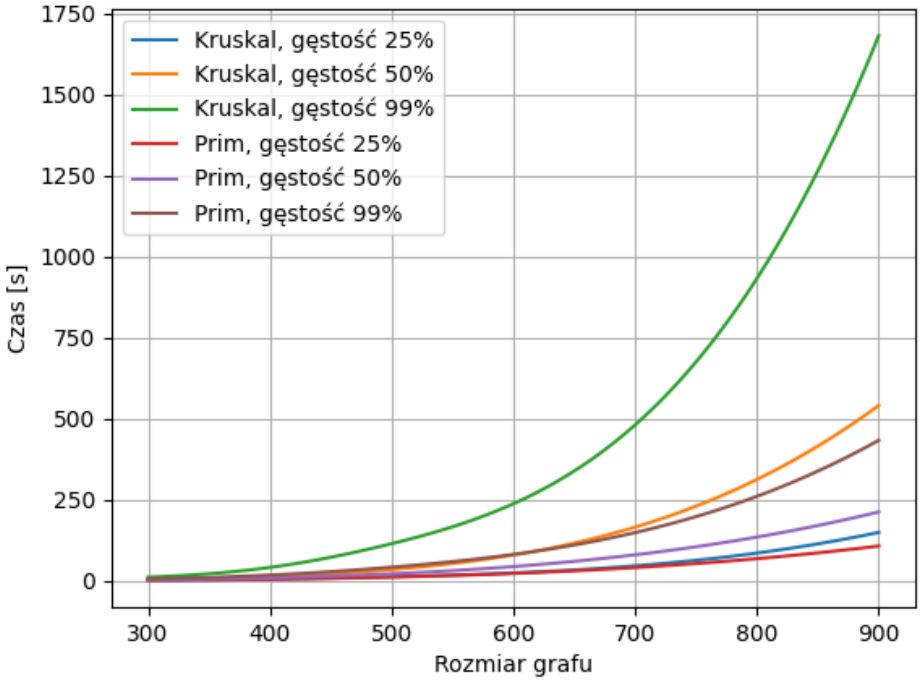
```
#include "Time.h"

//włączanie "stopera", zapisywanie obecnego czasu
   do zmiennej startTime
void Time::start() {
    startTime = high_resolution_clock::now();
}

//wyłączanie "stopera", zapisywanie obecnego czasu
    do zmiennej stopTime
void Time::stop() {
    endTime = high_resolution_clock::now();
}

//odejmowanie czasu początkowego i koncowego w
   celu obliczenia czasu trwania algorytmu
long Time::getTimeMiliseconds() {
    stop();
    return duration_cast<milliseconds>(endTime -
       startTime).count();
}

long Time::getTimeSeconds() {
    stop();
    return duration_cast<seconds>(endTime -
       startTime).count();
}
```

# 5 Wyniki eksperymentu

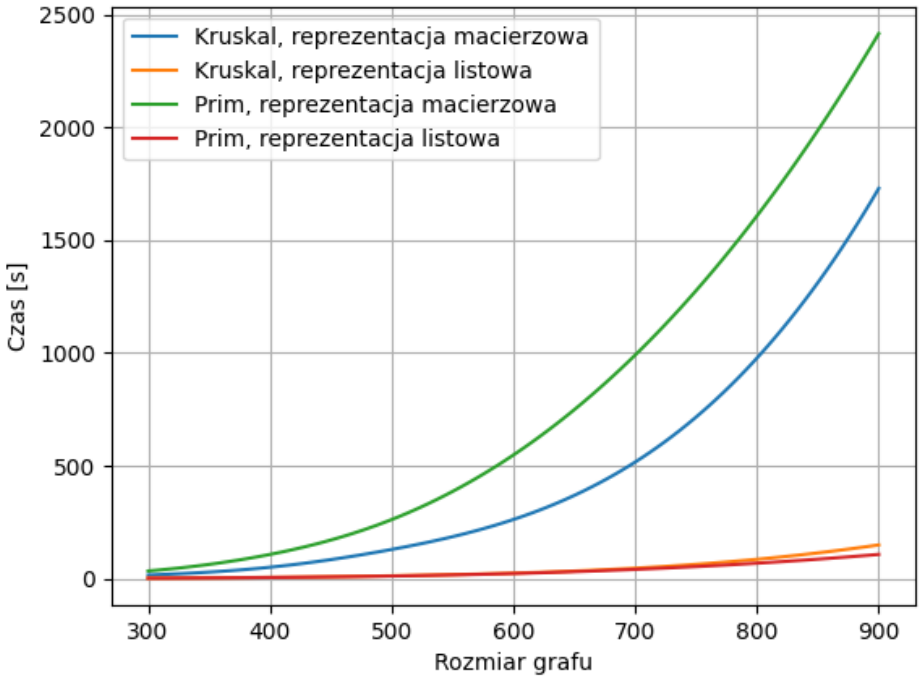
## 5.1 Wyznaczanie minimalnego drzewa rozpinającego (MST)



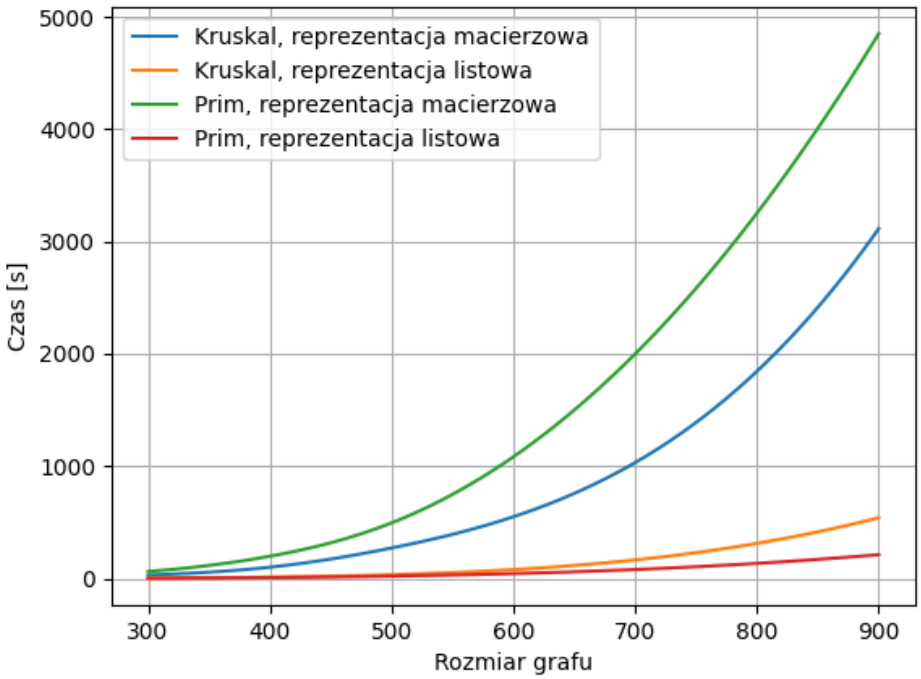

Rysunek 1: Porównanie gęstości grafów przy reprezentacji z macierzą incydencji



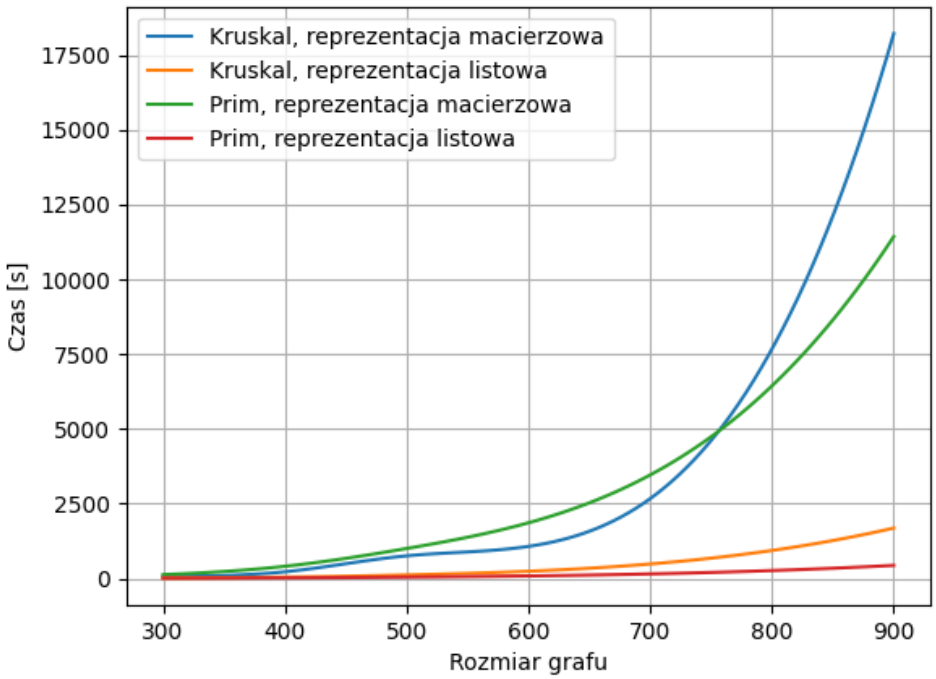

Rysunek 2: Porównanie gęstości grafów przy reprezentacji z listą następników




Rysunek 3: Porównanie reprezentacji grafów dla gęstości 25%

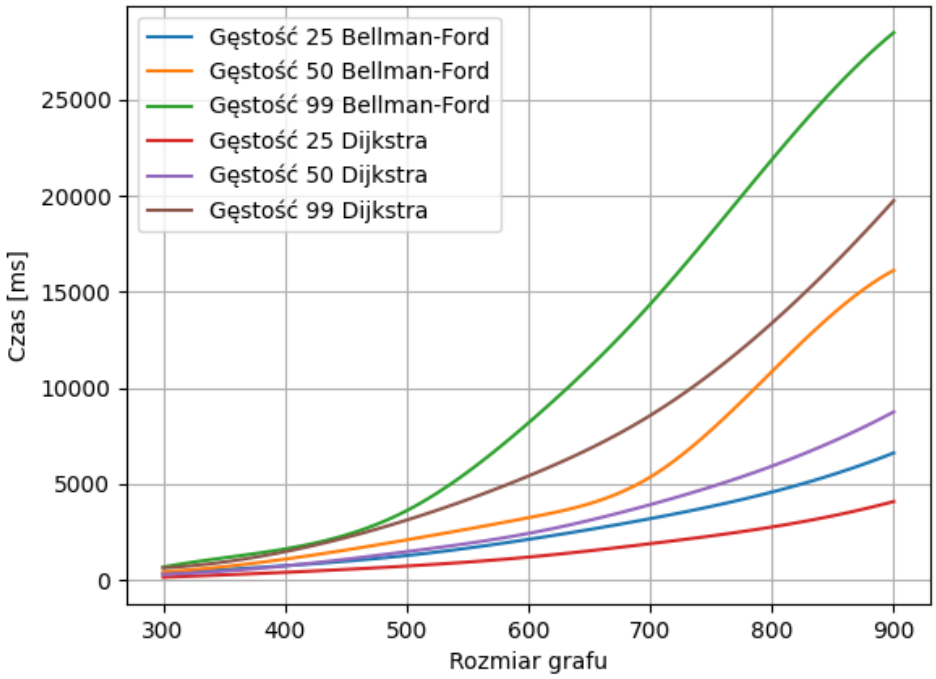



Rysunek 4: Porównanie gęstości grafów dla gęstości 50%

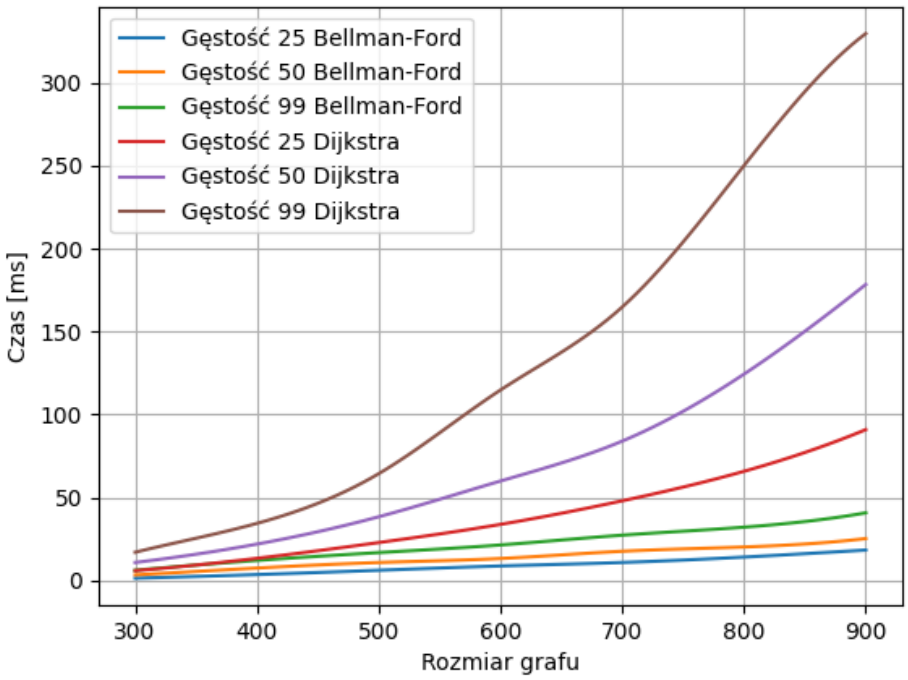



Rysunek 5: Porównanie gęstości grafów dla gęstości 99%

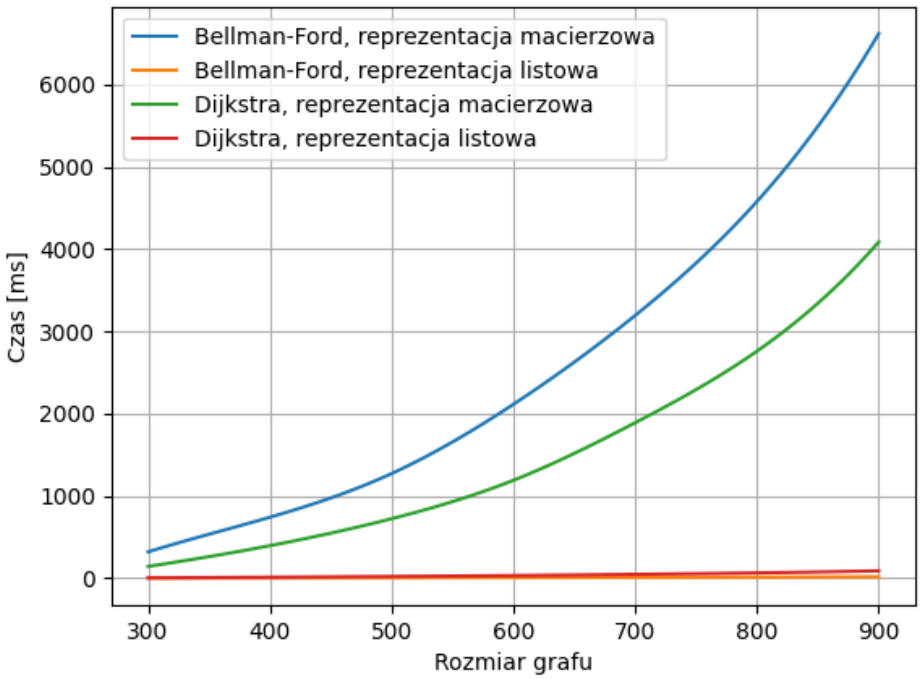
## 5.2 Wyznaczanie najkrótszej ścieżki w grafie



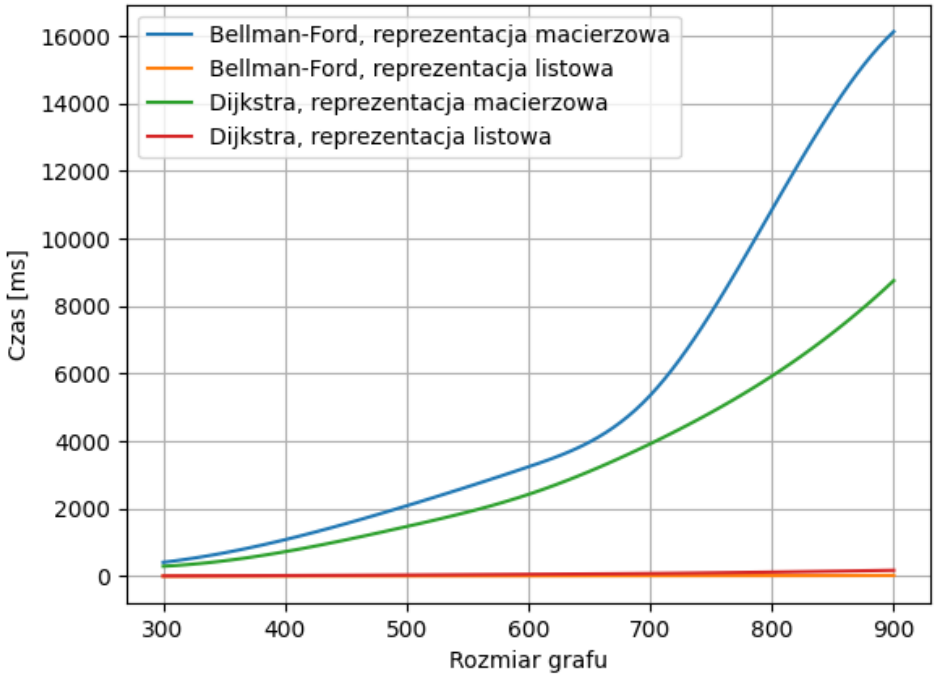

Rysunek 6: Porównanie gęstości grafów przy reprezentacji z macierzą incydencji



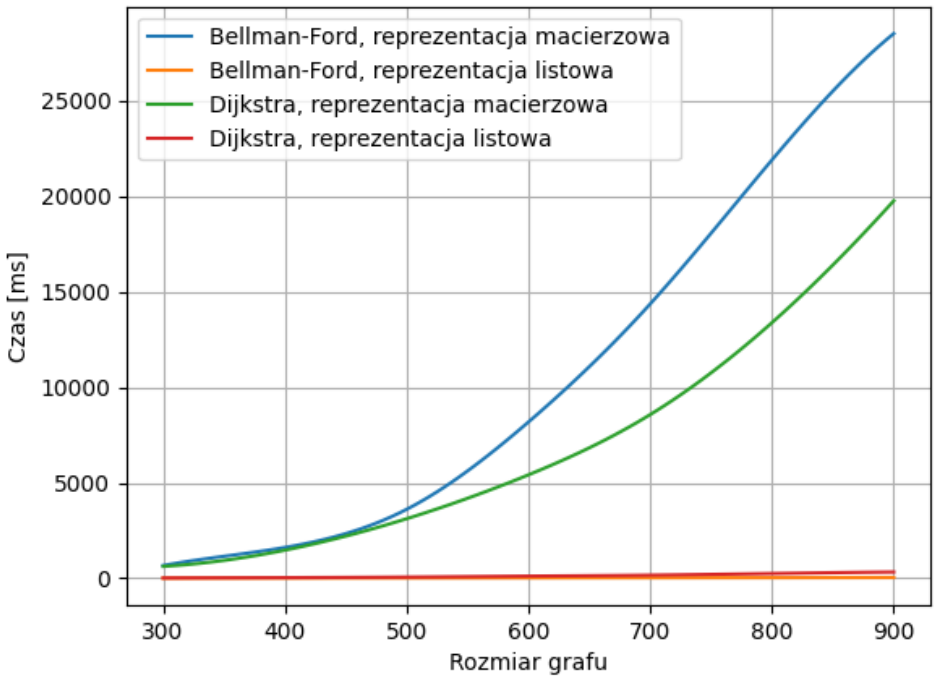

Rysunek 7: Porównanie gęstości grafów przy reprezentacji z listą następników




Rysunek 8: Porównanie reprezentacji grafów dla gęstości 25%




Rysunek 9: Porównanie gęstości grafów dla gęstości 50%




Rysunek 10: Porównanie gęstości grafów dla gęstości 99%

# 6 Podsumowanie

Przeglądając dane zebrane z eksperymentów i porównując wynki dla poszczególnych odpowiadających sobie reprezentacji, nasuwa sie pytanie czy algorytmy zaimplementowane dla macierzy były zaprojektowane efektywnie. Ich czas jest zdecydowanie dłuższy niż tych opierających siie na liście. Drugim wnioskiem jaki można wysnuć z powyższego testu, jest teza, że reprezentacja macierzowa jest mniej efektywna zarówno czasowo jak i pamięciowo.

# Literatura


[1] Algorytmy znajdujące minmalne drzewo rozpinające.

[2] Algorytm Bellmana-Forda.

[3] Algorytm Dijkstry.

[4] Adrian Horzyk ,AGH, *Algorithms*. WSTĘP DO INFORMATYKI, Grafy i struktury grafowe Wykład