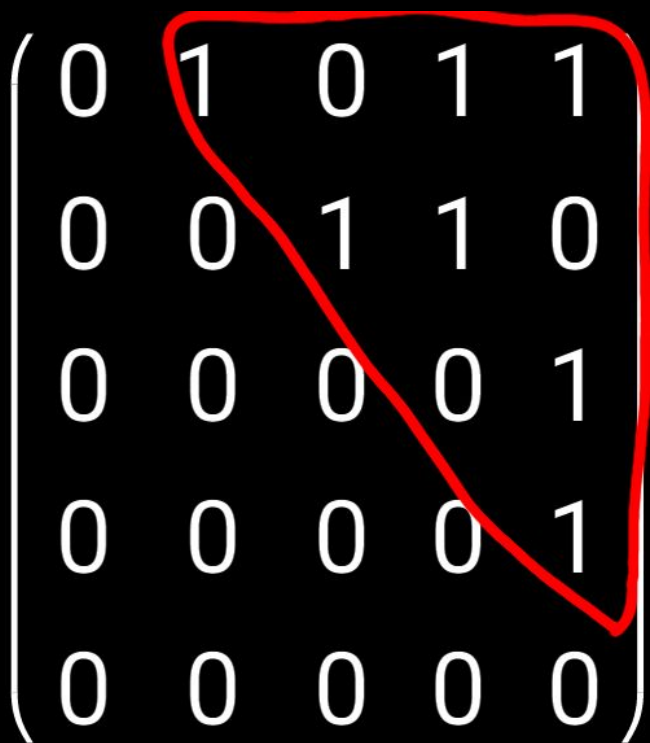


# Sprawozdanie

algorytmy grafowe

# Dane wejściowe


$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Grafy tworzone są poprzez utworzenie macierzy sąsiedztwa, która zawiera krawędzie jedynie powyżej głównej przekątnej, zapewnia to acykliczność. Zaczynamy od macierzy zer o rozmiarze  $n \times n$ . Aby zapewnić spójność generujemy permutację wierzchołków i tworzymy krawędzie między wierzchołkami leżącymi obok siebie w tej permutacji. Kierunek krawędzi prowadzi do wierzchołka o mniejszym indeksie. Następnie wypełniamy macierz '1' powyżej głównej przekątnej aż do uzyskaniażądanego współczynnika wypełnienia grafu krawędziami. Finalnie przekształcamy macierz do żądanej reprezentacji. (Przekształcenie jest wykonywane tylko na potrzeby tworzenia grafu algorytmy działają z konkretnymi reprezentacjami)

Do testów algorytmów przyjęliśmy  $n$ 'y zaczynające się od 100 z krokiem 100 do 1000 włącznie oraz współczynnik wypełnienia grafu 0.5

# Testy algorytmów

W ramach testów algorytmów dwudziestokrotnie zmierzaliśmy czas ich działania dla każdego rozmiaru danych dla każdej reprezentacji. Odrzuciliśmy skrajne wyniki (25% najwyższych i 25% najniższych), a z pozostałych wyników obliczyliśmy średnią oraz odchylenie standardowe - te dane są przedstawione na wykresach. W opisach algorytmów  $n$  będzie oznaczało ilość wierzchołków, a  $m$  ilość krawędzi. Czas mierzony jest za pomocą modułu time, dokładność pomiaru to  $\pm 1\mu s$ .

<del>2.2</del>	2.3	2.3	2.4	2.4	2.4	2.45	2.46	2.5	<del>2.55</del>
----------------	-----	-----	-----	-----	-----	------	------	-----	-----------------

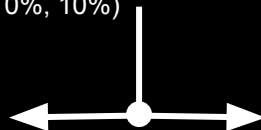
przykładowe wyniki



2.3	2.3	2.4	2.4	2.4	2.45	2.46	2.5
-----	-----	-----	-----	-----	------	------	-----

wyniki po odrzuceniu skrajnych wartości (10%, 10%)

$\sim 2.401$  ← średnia



odchylenie standardowe →  $\sim 0.067$

# Lista następników

Lista następników to struktura danych, która zawiera informacje o bezpośrednich następnikach każdego wierzchołka w grafie skierowanym.

Złożoność pamięciowa listy następników w grafie skierowanym zależy od liczby krawędzi w grafie oraz liczby wierzchołków. Jeśli graf ma  $n$  wierzchołków i  $m$  krawędzi, to pamięć potrzebna do przechowywania listy następników wynosi  $O(n + m)$ .

Znalezienie pojedynczej krawędzi wymaga sprawdzenia całej listy następników określonego wierzchołka, z którego może wychodzić  $n-1$  krawędzi, dlatego złożoność wynosi  $O(n-1) \Rightarrow O(n)$ .

Znalezienie wszystkich następników danego wierzchołka również wymaga przejrzenia całej listy dotyczącej odpowiadającego wierzchołka, stąd złożoność  $O(n-1) \Rightarrow O(n)$ .

```
0: [1, 2, 3, 4, 5, 6, 7, 8, 9]
1: [3, 5, 6]
2: [6, 9]
3: [5, 6, 9]
4: [5]
5: [8]
6: [7]
7: [9]
8: []
9: []
```

# Macierz sąsiedztwa

Jest to dwuwymiarowa tablica o wymiarach  $n \times n$ , gdzie  $n$  to liczba wierzchołków w grafie. W macierzy sąsiedztwa wartość  $M[i][j]$  określa, czy istnieje krawędź między wierzchołkiem  $i$  a wierzchołkiem  $j$ . Jeśli krawędź między  $i$  a  $j$  istnieje, to wartość  $M[i][j]$  wynosi 1, w przeciwnym przypadku wynosi 0. Macierz jest wielkości  $n \times n$  zatem jej złożoność pamięciowa to  $O(n^2)$ .

Ponieważ każda krawędź jest reprezentowana przez pojedyncze pole w macierzy, złożoność czasowa sprawdzenia krawędzi jest stała i wynosi  $O(1)$ .

Aby znaleźć wszystkich sąsiadów danego wierzchołka w macierzy sąsiedztwa, należy przejrzeć pojedynczy wiersz lub kolumnę odpowiadającą wierzchołkowi  $i$  i znaleźć wszystkie pola, których wartość wynosi 1. Sprawdzenie tylko danego wiersza (lub kolumny) wymaga sprawdzenia  $n$  pól, dlatego złożoność wynosi  $O(n)$ .

```
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 1, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 1, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

# Lista krawędzi

Jest to reprezentacja grafu polegająca na przechowywaniu listy uporządkowanych par złożonych z wierzchołka z którego wychodzi krawędź oraz wierzchołka do którego prowadzi. Ilość wymaganej pamięci jest proporcjonalna do ilości krawędzi w grafie. W przypadku grafu pełnego skierowanego jest to  $n(n-1)$  zatem złożoność pamięciowa to w ogólności -  $O(m)$ , pesymistycznie -  $O(n^2)$ .

Znalezienie pojedynczej krawędzi wymaga przejrzania całej listy, odbywa się to bez dodatkowej pamięci ze złożonością czasową  $O(m)$ , pesymistycznie  $O(n^2)$ .

Wyszukanie wszystkich następników wężła również wymaga przejrzania całej listy, odbywa się to bez dodatkowej pamięci ze złożonością czasową  $O(m)$ , pesymistycznie  $O(n^2)$ .

Reprezentacja ta jest efektywna jedynie w przypadku rzadkich grafów, jeśli  $m \gg n$  to w większości przypadków lepiej spisać listę następników - optymalizuje ona ilość zajmowanej pamięci i zagwarantuje liniową złożoność zależną od ilości węzłów dla wyszukiwania krawędzi bądź następników wężła.

```
edges
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(0, 5)
(0, 6)
(0, 7)
(0, 8)
(0, 9)
(1, 3)
(1, 5)
(1, 6)
(2, 6)
(2, 9)
(3, 5)
(3, 6)
(3, 9)
(4, 5)
(5, 8)
(6, 7)
(7, 9)
```

# BFS

Algorytm przeszukuje graf zaczynając od jednego z wierzchołków. Najpierw przechodzi do najbliższych sąsiadów aktualnego wierzchołka, po przetworzeniu wszystkich przechodzi do najbliższych sąsiadów sąsiadów tego wierzchołka itd. aż nie zostaną odwiedzone wszystkie wierzchołki do których prowadzą ścieżki. Jeśli do wierzchołka nie prowadziła żadna ścieżka bfs zostanie wywołany dla tego wierzchołka.

Algorytm realizowany jest przez użycie stosu. Wrzucamy wierzchołek od którego zaczynamy na stos, po czym dopóki stos nie jest pusty zdejmujemy wierzchołek ze stosu, przetwarzamy go i zaznaczamy jako odwiedzony, po czym wrzucamy na stos wszystkie nieodwiedzone następni.

BFS wymaga dodatkowej pamięci (nie wliczając reprezentacji grafu), potrzebna jest pamięć na stos. Pesymistycznie na stosie będzie przechowywane  $n-1$  wierzchołków gdy wierzchołek startowy będzie poprzednikiem wszystkich innych wierzchołków co skutkuje złożonością  $O(n)$ .

# BFS

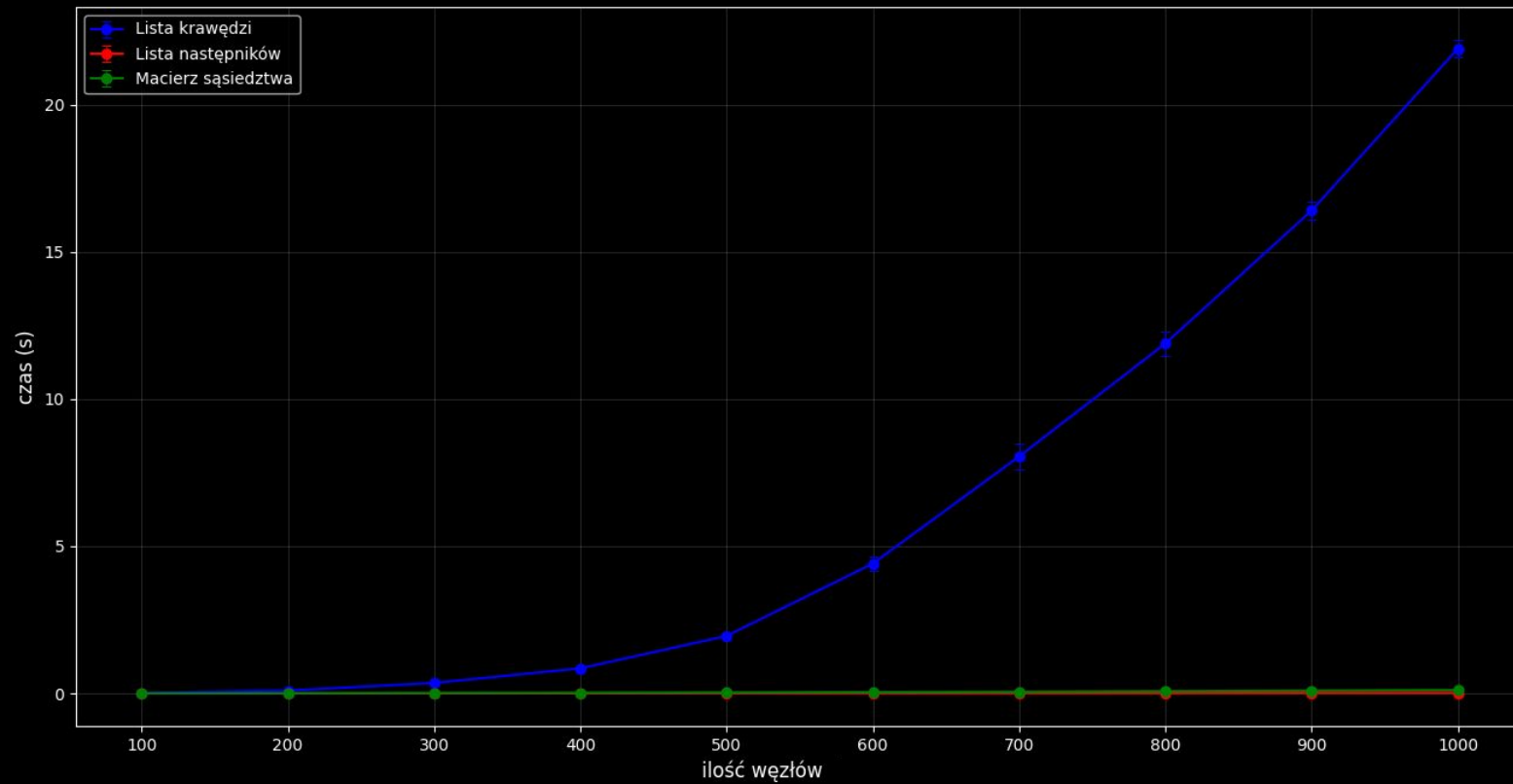
Inicjowanie tablicy odwiedzonych wierzchołków odbywa się w czasie liniowym zależnym od  $n$ , Każdy wierzchołek jest odwiedzany co najwyżej raz, ponieważ tylko za pierwszym razem, gdy jest osiągnięty jego wartość odwiedzony posiada wartość False, zatem każdy wierzchołek jest dodany do kolejki co najwyżej raz.

Przeszukiwanie grafu wszerz wymaga dla każdego wierzchołka znalezienia i przejrzenia listy jego następników czego czas będzie się różnił w zależności od reprezentacji grafu. Sumarycznie będzie to:

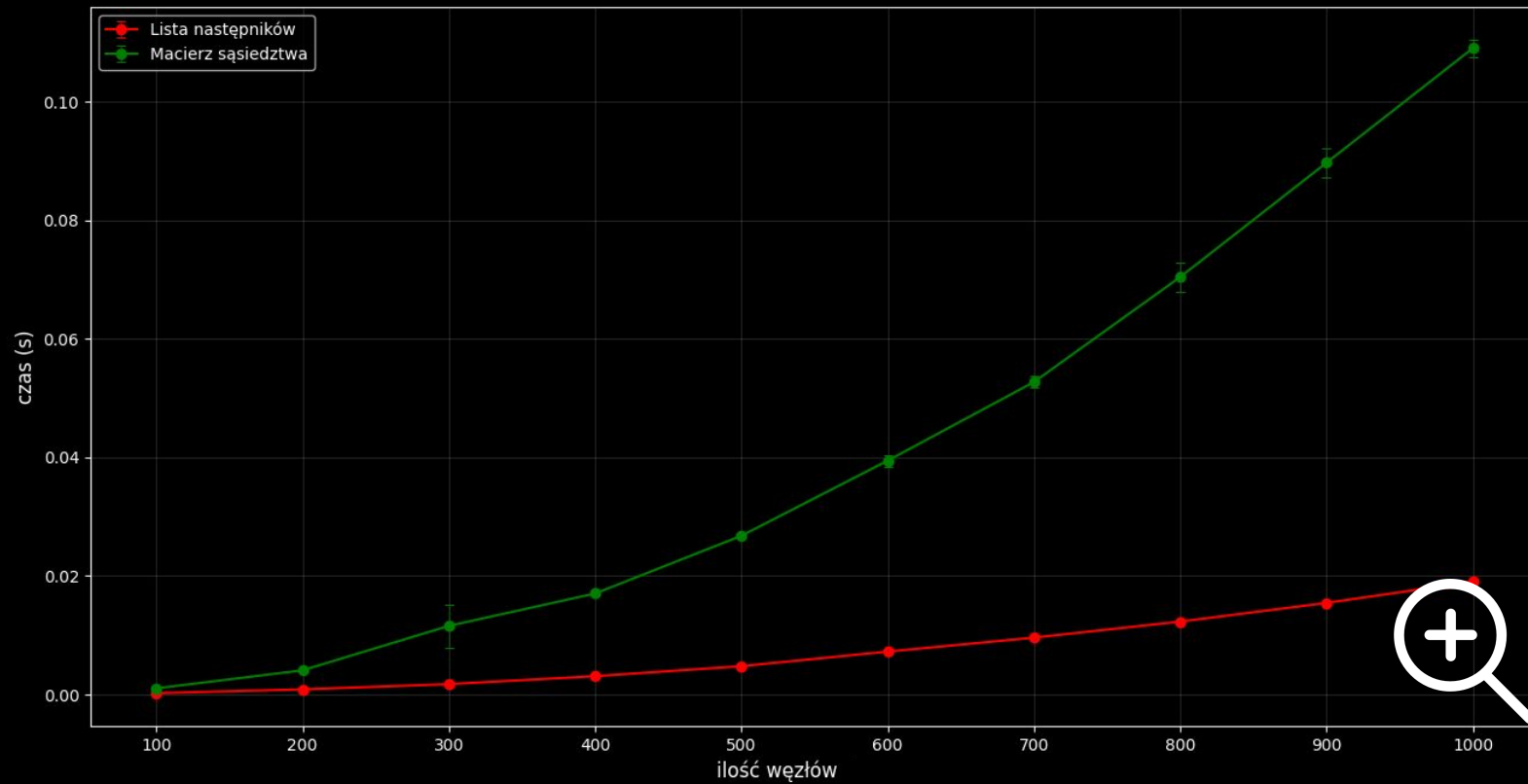
$O(n^2)$  dla listy następników  $O(n^2)$  dla macierzy sąsiedztwa i  $O(n^3)$  dla listy krawędzi



## bfs



bfs



# BFS

Zgodnie z oczekiwaniami BFS działając z macierzą sąsiedztwa i listą następników spisuje się lepiej niż działając z listą krawędzi. W przypadku listy następników algorytm spisuje się jednak najlepiej, spowodowane jest to natychmiastowym dostępem do listy następników wymagane jest jedynie jej przejrzanie, macierz sąsiedztwa w celu znalezienia listy następników natomiast wymaga przejrzania  $n$  indeksów w macierzy co wyjaśnia jej gorszą wydajność.

# DFS

Depth-First Search jest to algorytm przeszukujący graf w głąb. Algorytm DFS zwykle zaczyna od wybranego wierzchołka startowego i rekurencyjnie odwiedza sąsiadów danego wierzchołka, zaczynając od pierwszego napotkanego, a następnie przechodząc do kolejnych. Algorytm zapamiętuje odwiedzone wierzchołki i zaznacza je, aby nie odwiedzać ich ponownie, co zapobiega zapętłaniu się w przypadku grafów cyklicznych.

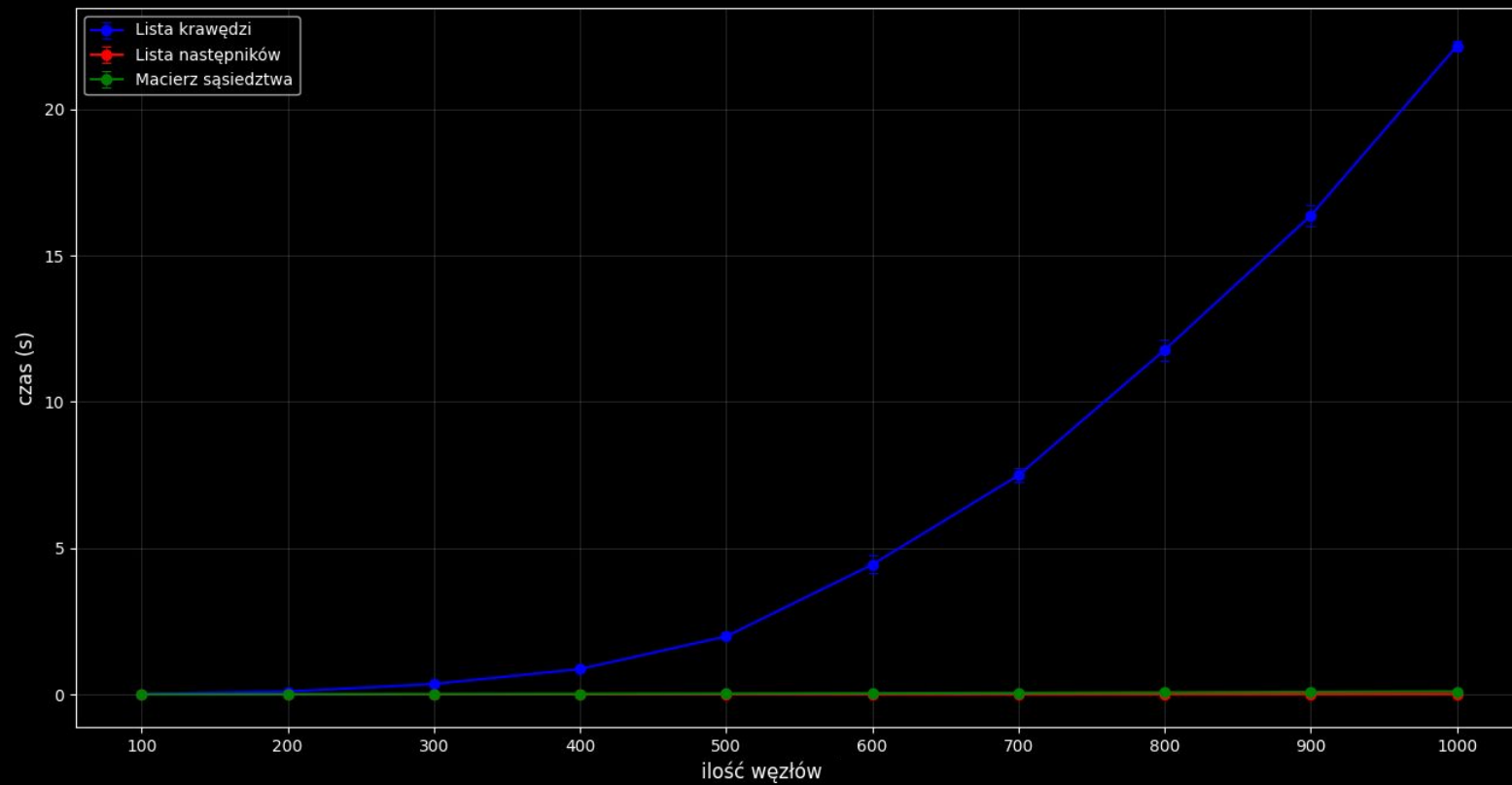
Złożoność czasowa DFS zależy od konkretnego grafu i implementacji algorytmu. Jednak ogólnie rzecz biorąc, złożoność czasowa DFS dla grafu o  $V$  wierzchołkach i  $E$  krawędziach wynosi  $O(V+E)$ . Będzie się ona jednak różnić w zależności od reprezentacji danych:

Dla listy następników będzie to  $O(n^2)$ ;

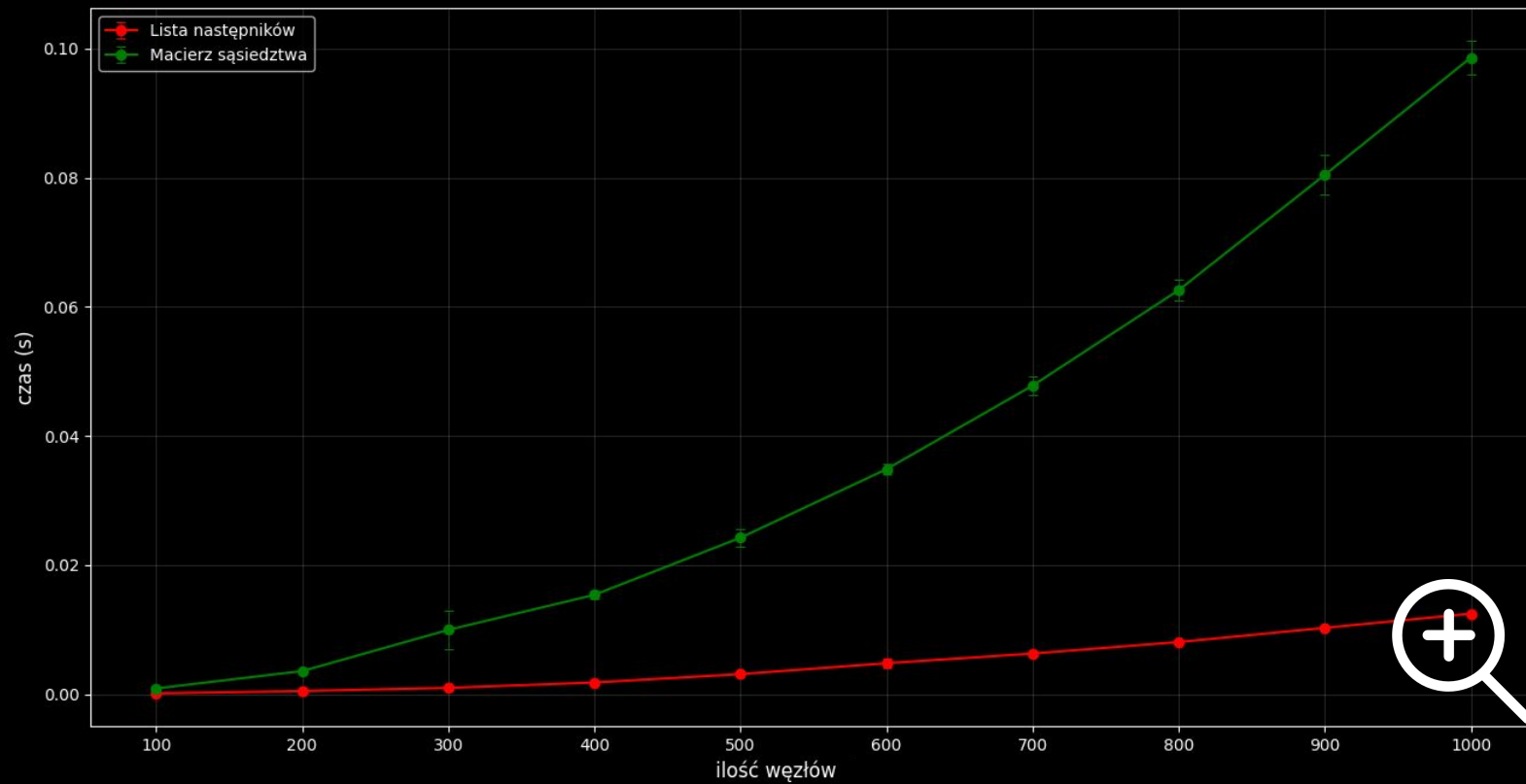
dla macierzy sąsiedztwa -  $O(n^2)$ ;

a dla listy krawędzi -  $O(n^3)$ .

## dfs



dfs



# Sortowanie topologiczne

Grafy skierowane bez cykli, zwane również grafami acyklicznymi można sortować topologicznie. Sortowanie topologiczne polega na uporządkowaniu wierzchołków grafu w taki sposób, że jeśli istnieje droga z wierzchołka A do wierzchołka B, to wierzchołek A występuje przed wierzchołkiem B w porządku topologicznym.

Graf cykliczny to taki graf, w którym można przejść po kilku krawędziach i wrócić do punktu startowego. Taki graf nie nadaje się do sortowania topologicznego, ponieważ nie można jednoznacznie określić kolejności wierzchołków, gdyż kilka wierzchołków jest ze sobą wzajemnie zależnych i nie można stwierdzić, który powinien być pierwszy. Jeśli wierzchołki A i B są we wspólnym cyklu to istnieje droga zarówno z A do B i z B do A, przez co nie można ustalić ich porządku topologicznego.

# Sortowanie topologiczne (w głąb)

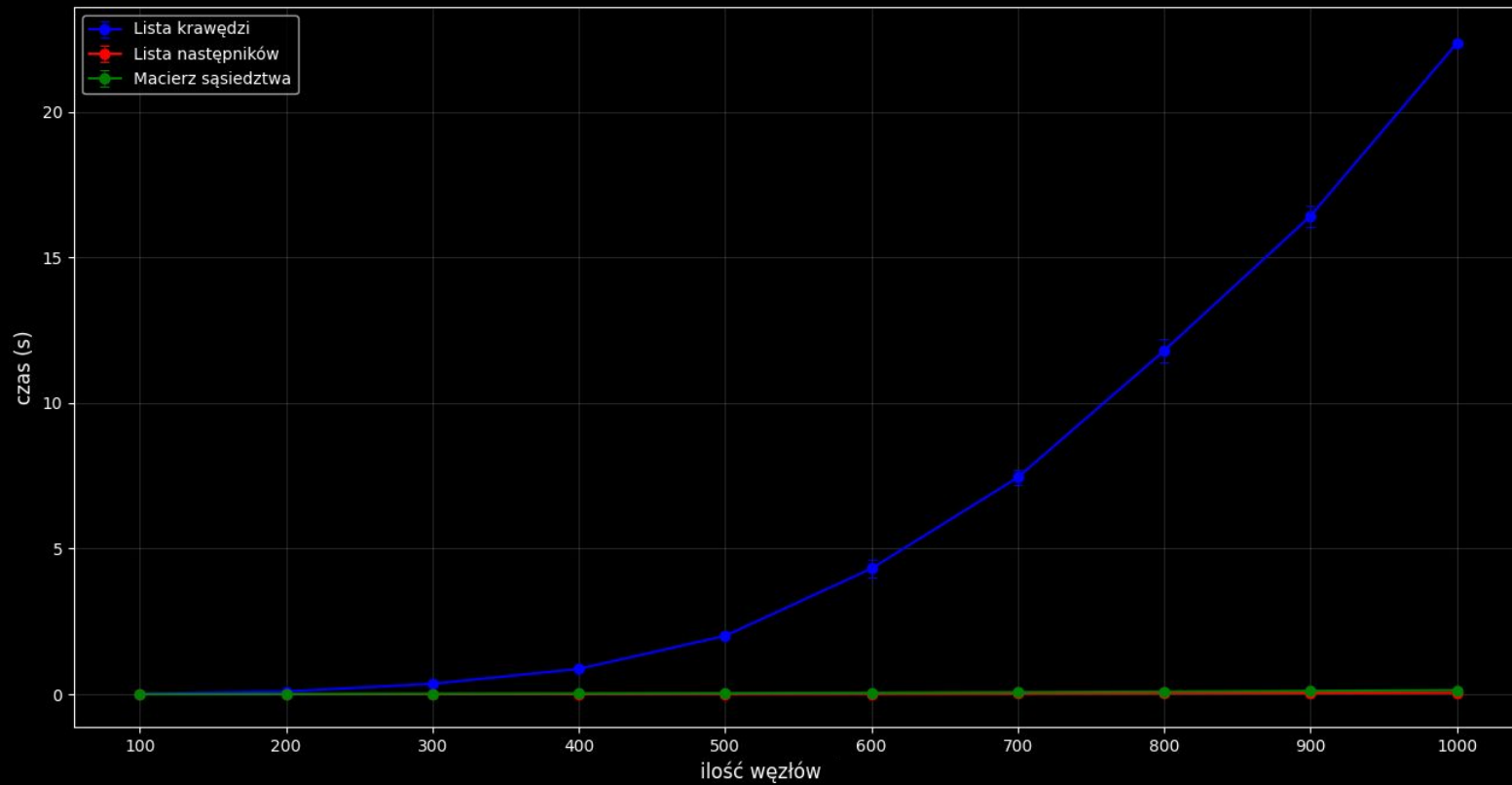
Sortowanie topologiczne w głąb opiera się na algorytmie DFS. Należy rozpocząć ten algorytm dla losowego wierzchołka. W trakcie przeszukiwania, gdy odwiedzimy dany wierzchołek, umieszczamy go na stosie oraz oznaczamy jako odwiedzony.

Następnie przeszukujemy wierzchołki sąsiednie i dla każdego nieodwiedzonego wywołujemy rekurencyjnie algorytm DFS, wpisujemy na stos oraz oznaczamy jako odwiedzony. Algorytm kontynuuje przeszukiwanie grafu, wybierając kolejne nieodwiedzone wierzchołki aż dotrze do wierzchołka nieposiadającego

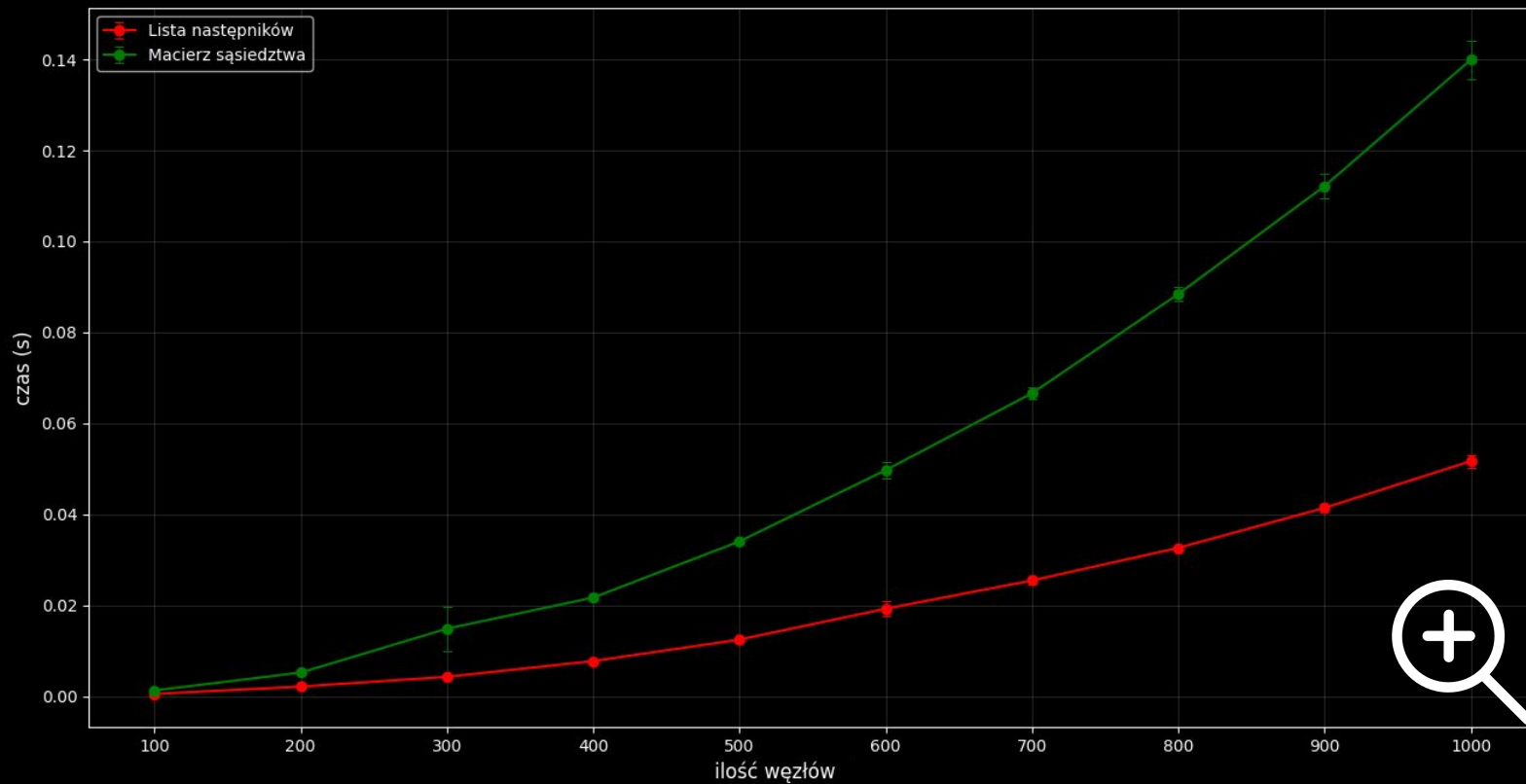
następników. Wtedy cofając się po ścieżce, zgodnie z końowymi czasami przetworzenia wierzchołków, spisujemy je ze stosu na listę wynikową. Dzieje się to do momentu aż wszystkie wierzchołki w grafie nie zostaną odwiedzone.



## depth\_sort



## depth\_sort



# Sortowanie topologiczne (wszerz)

W acyklicznym grafie skierowanym musi wystąpić przynajmniej jeden wierzchołek o stopniu wchodzącym zero. Usunięcie wierzchołka z grafu nigdy nie powoduje, że staje się on grafem cyklicznym. Zatem po każdej operacji usunięcia wierzchołka w grafie pojawia się przynajmniej jeden wierzchołek o stopniu wchodzącym zero. Korzystając z tej własności jesteśmy w stanie posortować graf topologicznie, usuwając wierzchołki aż otrzymamy graf pusty.

Zaczynamy od obliczenia stopni wejściowych wierzchołków, będzie to wymagało przejrzania wszystkich krawędzi, stąd wystąpią różnice w złożoności czasowej w zależności od implementacji. Ten etap zajmie  $O(m)$ , pesymistycznie  $O(n^2)$  dla listy krawędzi i listy następników i  $O(n^2)$  dla macierzy sąsiedztwa.

W dalszym kroku dodamy do kolejki wszystkie wierzchołki o stopniu wejściowym równym zero, wyszukanie ich w tablicy odbywa się w czasie liniowym.

Ostatecznie dopóki kolejka nie jest pusta pobieramy z niej węzeł, usuwamy go i aktualizujemy stopnie wejściowe jego następników, jeśli wyzerujemy stopień węzła dodajemy go do kolejki. Etap ten odbędzie się dla każdego węzła.

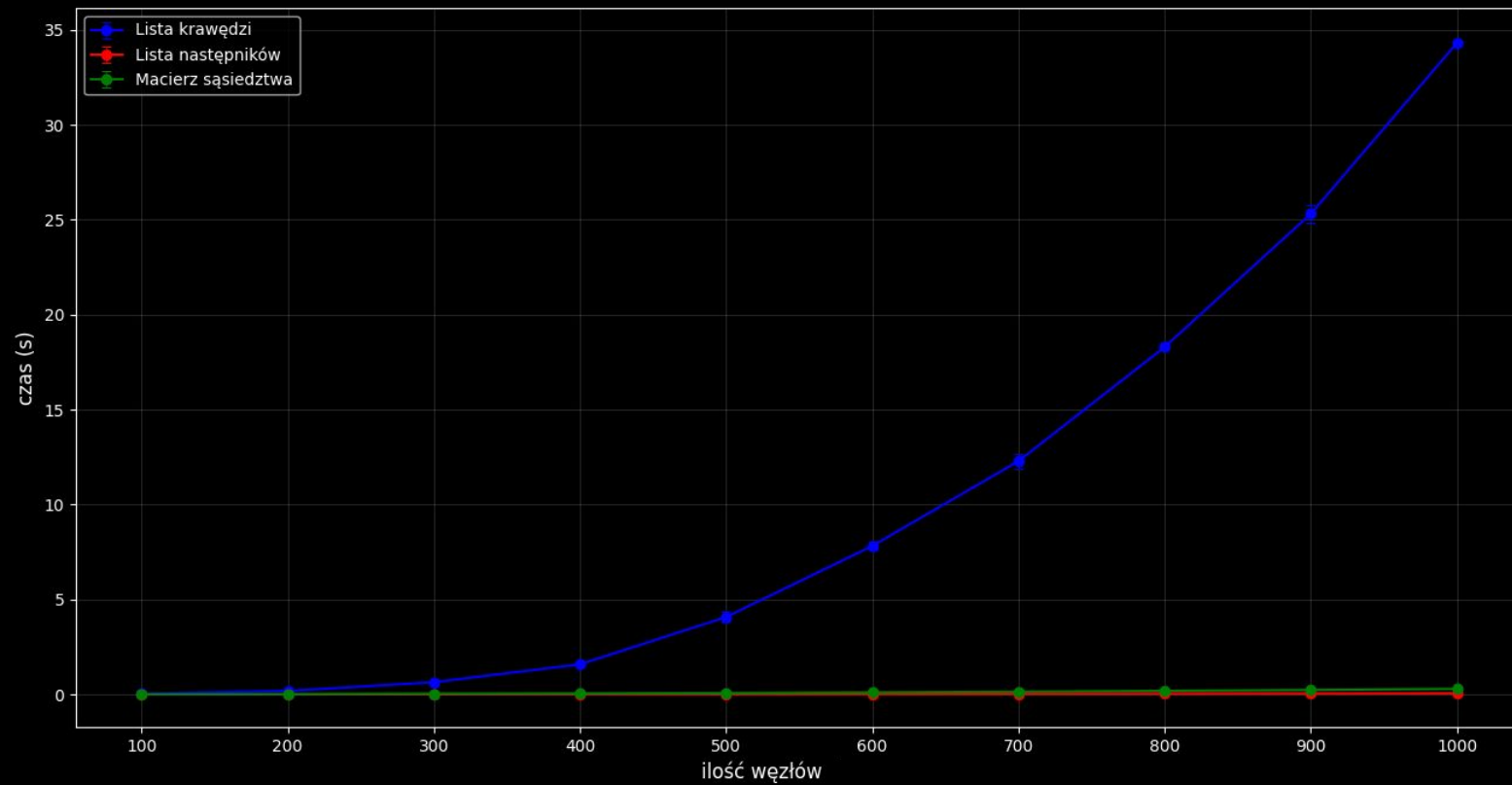
Reasumując musimy znaleźć i przejrzeć listę następników dla każdego węzła. Wystąpią różnice ze względu na implementację. Etap ten zajmie  $O(n^2)$  dla listy następników i tablicy sąsiedztwa i  $O(nm)$ , pesymistycznie  $O(n^3)$  dla listy krawędzi.

# Sortowanie topologiczne (wszerz)

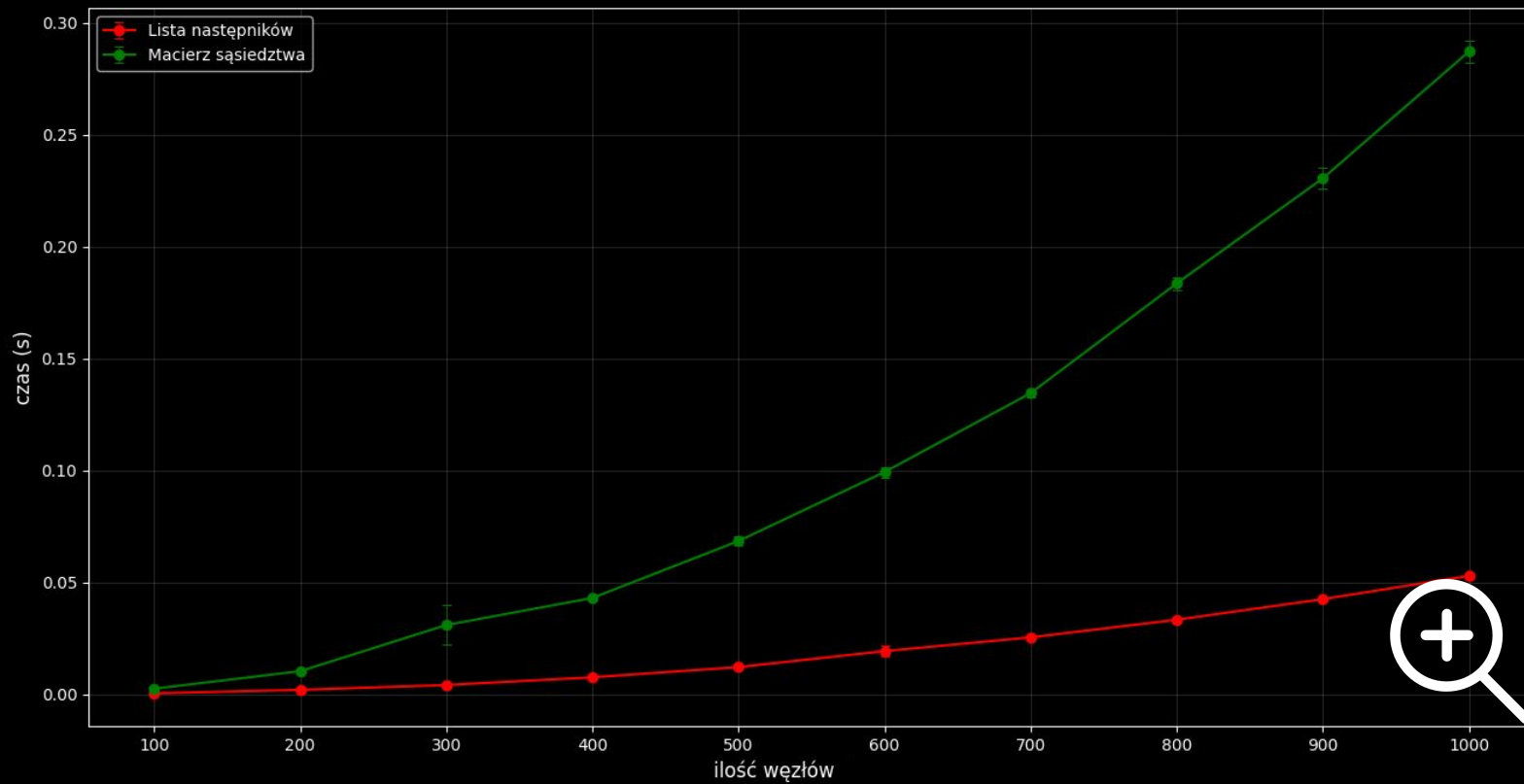
Podsumowując złożoność czasowa sortowania topologicznego wszerz to:  $O(n^2)$  dla listy następników  $O(n^2)$  dla macierzy sąsiedztwa i  $O(n^3)$  dla listy krawędzi.

Algorytm wymaga dodatkowej pamięci (nie wliczając reprezentacji grafu), potrzebna jest pamięć na stos. Pesymistycznie na stosie będzie przechowywane  $n-1$  wierzchołków gdy wierzchołek startowy będzie jedynym poprzednikiem wszystkich innych wierzchołków co skutkuje złożonością  $O(n)$ .

## breadth\_sort



## breadth\_sort



# Sortowanie topologiczne (wszerz)

Zgodnie z oczekiwaniami algorytm działając z macierzą sąsiedztwa i listą następników spisuje się lepiej niż działając z listą krawędzi. W przypadku listy następników algorytm spisuje się jednak najlepiej, spowodowane jest to natychmiastowym dostępem do listy następników wymagane jest jedynie jej przejrzanie, macierz sąsiedztwa w celu znalezienia listy następników natomiast wymaga przejrzania  $n$  indeksów w macierzy co wyjaśnia jej gorszą wydajność.