# Algorytmy grafowe

Eryk Andrzejewski - 145277

14 czerwca 2020

## Spis treści

# 1 Wstęp

Sprawozdanie to poświęcone jest algorytmom grafowym. Zawarłem w nim ogólny opis implementacji kilku różnych reprezentacji grafów oraz podstawowych algorytmów, które można na tych grafach wykonać. Zawiera również pewne przemyślenia, spostrzeżenia oraz wyniki testów wydajności poszczególnych algorytmów dla różnych reprezentacji grafów. Skromnemu gronu osób, które to czyta, życzę miłej lektury :-)

## 1.1 Wymagania projektu

Celem projektu było zaimplementowanie, w wybranym języku programowania, trzech reprezentacji grafów: macierzy sąsiedztwa, macierzy incydencji (macierzy krawędzi) oraz listy sąsiedztwa. Należało zaprogramować również algorytmy przeszukiwania grafu: DFS oraz BFS dla wszystkich trzech, wcześniej wymienionych, reprezentacji. Podobnie miała się rzecz z sortowaniem topologicznym skierowanego grafu acyklicznego w wersjach BFS oraz DFS – również mieliśmy dokonać implementacji dla wszystkich tych reprezentacji.

Kolejne wymagania dotyczyły generowania grafów Hamiltona i Eulera oraz grafów, które nie należą do którejś z tych grup. Po wygenerowaniu takiego grafu, należało w nim znaleźć przynajmniej jeden cykl każdego typu (o ile istnieje).

## 1.2 Ogólny opis realizacji

Projekt wykonałem w języku C++. Dołożyłem wszelkich starań, aby kod był czytelny, rozszerzalny i wygodny w użyciu.

Kluczową decyzją, podjętą na etapie projektowania, było wyodrębnienie pewnej warstwy abstrakcji, za którą kryją się implementacje poszczególnych reprezentacji grafu. Graf jest abstrakcyjną strukturą – użytkowników nie interesują wewnętrzne szczegóły implementacji, a jedynie informacje o połączeniach poszczególnych wierzchołków. Wobec tego wybór implementacji grafu nie powinien mieć żadnego związku z realizacją rozmaitych algorytmów grafowych (przeszukiwanie BFS, DFS itd.). Graf powinien *działać*, niezależnie od tego, co jest *pod maską*.

Mówiąc konkretniej, utworzyłem klasę abstrakcyjną grafu (`Graph`) z metodami czysto wirtualnymi, których stosowanie pozwala dokonywać wszelkich dozwolonych operacji na grafie. Każda z realizacji grafu posiada osobną

klasę, która dziedziczy po klasie `Graph` i implementuje brakujące metody. Zważywszy na to, że grafy możemy podzielić na skierowane i nieskierowane, oraz wyróżniając wszystkie trzy reprezentacje, daje nam to w sumie sześć różnych klas.

Takie podejście pozwala znacznie zmniejszyć rozmiary kodu, uczynić go prostszym w zarządzaniu – algorytmy grafowe implementujemy raz, a nie sześć razy. Do tego, taka warstwa abstrakcji pozwala uczynić kod znacznie bardziej czytelnym (wywołanie metody `addEdge` jest z pewnością bardziej czytelne, niż odwoływanie się do szczegółów implementacji jakiejś macierzy lub listy).

Poszczególne algorytmy grafowe posiadają również własne klasy, które implementują określone interfejsy. Dzięki temu możliwe jest skorzystanie z polimorfizmu i przykładowo wykonanie sortowania topologicznego przy pomocy różnych algorytmów, np. BFS i DFS. Nie będę jednak tego wszystkiego dokładnie omawiał w tym miejscu, gdyż miał być to tylko pewien zarys projektu.

Dodatkowo wspomnę, że w projekcie znajdują się również dodatkowe fragmenty kodu, niepowiązane zbyt mocno z tematem, ale jednak potrzebne lub przynajmniej bardzo przydatne do realizacji tego projektu. Mogę wyróżnić w tym miejscu klasę `Matrix`, która implementuje bardzo prostą macierz, wykorzystywaną przy implementacji grafów skierowanych i nieskierowanych, opartych o macierz sąsiedztwa. W pliku `io.hpp` znajdują się przeładowane operatory `«`, które pozwalają w wygodny sposób wypisywać kontenery z STL (`std::vector`, `std::list` itp.) przy użyciu obiektów klasy `std::ostream`, np. `std::cout`. Plik `random.hpp` zawiera funkcje – nakładki, stanowiące prostszy interfejs do losowania liczb, niż korzystanie z funkcji `rand()`, czy biblioteki `<random>` z C++11.

# 2 Szczegóły implementacji

## 2.1 Reprezentacje grafów

Jak powiedziałem we wstępie do tego sprawozdania, w projekcie została wyodrębniona pewna warstwa abstrakcji. Dzięki temu, każdy graf na pewno posiada następujące podstawowe metody:

- `void addEdge(int startVertex, int endVertex)`
- `void removeEdge(int startVertex, int endVertex)`
- `bool containsEdge(int startVertex, int endVertex)`
- `std::list<int> getPredecessors(int vertex)`
- `std::list<int> getSuccessors(int vertex)`
- `std::size_t getIndegree(int vertex)`
- `std::size_t getOutdegree(int vertex)`

Metod tych jest więcej, ale nie są na tyle warte uwagi, jak te powyżej – mógłbym oczywiście opisywać wszystko bardzo dokładnie, ale sprawozdanie by trochę spuchło, a ja musiałbym studiować na trzy etaty ;).

Powyższe metody pozwalają rozszerzać graf o nowe krawędzie, usuwać istniejące. Możemy również sprawdzić, czy graf zawiera krawędź pomiędzy dwoma wierzchołkami, a także pobrać listy wierzchołków, do jakich (lub od jakich) występuje powiązanie z danym wierzchołkiem.

Interfejs dla grafów skierowanych i nieskierowanych jest ujednolicony. W przypadku grafu nieskierowanego należy się jednak spodziewać, że kolejność połączeń nie ma znaczenia, a metoda `getPredecessors` zwróci identyczną listę, jak `getSuccessors`. Dodatkowo, grafy nieskierowane mają na przykład dodatkową metodę `getDegree`, która w ich przypadku robi to samo, co `getIndegree` oraz `getOutdegree`, ale czytelniej.

Rozróżnienie grafów skierowanych i nieskierowanych na poziomie architektury kodu było istotne, aby, dla przykładu, kompilator nie dopuścił do sytuacji, gdy algorytm sortowania topologicznego miałby wywołać się dla grafu nieskierowanego. Osiągnąłem to w dość prosty sposób, wprowadzając dodatkowy poziom dziedziczenia (klasa `UndirectedGraph` dziedziczy po

klasie `Graph`, a implementacje kolejnych reprezentacji dziedziczą dopiero po klasie `UndirectedGraph`).

Aby unikać niepotrzebnego zamieszania z rozszerzaniem poszczególnych struktur danych (w przypadku list nie jest to duży problem, ale w przypadku macierzy już tak), poczyniłem założenie, iż liczba wierzchołków w grafie jest stała – jest ich przez cały okres istnienia obiektu dokładnie tyle, ile przekażemy konstruktorowi w momencie jego tworzenia. Założenie takie okazało się istotne i wpłynęło na ogólną ocenę reprezentacji grafów (projekt miał swoje konkretne wymagania, i do ich realizacji pewne reprezentacje były lepsze, a inne gorsze – co do tego, które z nich były które, miałem od początku pewne przypuszczenia, ale w celu ich zweryfikowania przeprowadziłem testy).

Teraz skupię się na krótkim omówieniu każdej z reprezentacji grafów.

### 2.1.1 Macierz sąsiedztwa

Macierz sąsiedztwa jest jedną z najprostszych i najskuteczniejszych reprezentacji grafu w komputerze. Jest to macierz kwadratowa o wymiarach $N \times N$, przy założeniu że $N$ to liczba wierzchołków. W projekcie jest reprezentowana przez klasy `DirectedAdjacencyMatrixGraph`
oraz `UndirectedAdjacencyMatrixGraph`, które pośrednio dziedziczą po klasie `Graph`.

Obecność krawędzi pomiędzy dwoma wierzchołkami (oznaczonymi dalej jako $P$ i $Q$) sygnalizuje wartość w komórce macierzy ($P$, $Q$). Wartość ta oznacza liczbę krawędzi pomiędzy tymi wierzchołkami. W całym projekcie zakładałem, że operuję tylko na grafach prostych, a nie na multigrafach (czyli pomiędzy dwoma wierzchołkami istnieje maksymalnie jedna krawędź, a dodatkowo pomijam pętle własne), ale taka reprezentacja (i implementacja) grafu również nadaje się do multigrafów. Warto tylko wspomnieć, że przy tej reprezentacji nie bylibyśmy w stanie w jakiś sposób etykietować różnych krawędzi incydentnych z tymi samymi wierzchołkami.

Implementacja metody dodającej krawędź sprowadza się do inkrementacji odpowiedniej komórki macierzy, metody usuwającej – dekrementacji, a metody sprawdzającej istnienie krawędzi – do porównania odpowiedniej komórki z zerem. Operacje te mają więc złożoność $O(1)$, czas wykonania nie zależy od rozmiaru macierzy i to ogromna zaleta tej reprezentacji – niestety kosztem pamięci, bo złożoność pamięciowa jest rzędu $O(V^2)$. Pobranie listy następników lub poprzedników wymaga już złożoności liniowej $O(V)$, gdyż trzeba przeiterować po całym wierszu, bądź kolumnie, w celu sprawdzenia

połączenia innych wierzchołków, z obecnym.

W ramach ciekawostki warto dodać, że iterowanie w wierszu jest z reguły wydajniejsze, niż w kolumnie (pewien ciągły obszar pamięci, taki jak wiersz, może zostać zcache'owany, a nie można powiedzieć tego samego o kolumnie), także nawet w kontekście grafu nieskierowanego, wybór przeznaczenia dla wierszy i kolumn może mieć znaczenie.

### 2.1.2 Macierz krawędzi

Kolejną reprezentacją, która również korzysta z macierzy jako formy przechowywania danych, jest macierz krawędzi (macierz incydencji). W tym przypadku jednak musimy mieć na uwadze, że macierz ta musi być rozszerzalna. Zamiast zastosować klasę `Matrix`, jak w przypadku poprzedniej reprezentacji, zastosowałem po prostu `std::list<std::vector<int»`. Motywacją takiego wyboru jest fakt, że lista krawędzi nie potrzebuje dostępu dowolnego, za to powinna być w prosty sposób rozszerzalna i pozwalać na szybkie i bezbolesne usuwanie elementów – sytuacja więc stworzona do użycia kontenera `std::list`, realizującego zwyczajną *linked-listę*. Z kolei każdy wiersz ma stałą szerokość, choć nieznaną w czasie kompilacji, wymaga szybkiego i swobodnego dostępu do poszczególnych jego elementów. Potrzebny nam więc będzie zaalokowany dynamicznie ciągły obszar pamięci, a do tego zastosowania najlepiej pasuje `std::vector`.

Nieco zmodyfikowałem oryginalną koncepcję, przez to w mojej implementacji kolumny oznaczają wierzchołki, a wiersze – krawędzie. Motywacją do takiej formy implementacji była większa wydajność i prostota rozszerzania o wiersze (nadal zachowujemy założenie, że liczba wierzchołków, czyli liczba kolumn jest stała).

W projekcie, reprezentacji tej odpowiadają klasy
`DirectedIncidenceMatrixGraph` oraz `UndirectedIncidenceMatrixGraph`.

Taka forma reprezentacji jest jednak według mnie gorsza, niż pozostałe (przynajmniej w zakresie tego projektu, bo na upartego mógłbym wskazać pewne jej zalety). Wykonanie prawie wszystkich podstawowych operacji (poza dodaniem wiersza z krawędzią, które przy odpowiedniej implementacji kontenera listy powinno być $O(1)$) wymaga przeiterowania po wszystkich wierszach macierzy (krawędziach grafu). Poza tym, trzeba jeszcze przeiterować wewnątrz każdego wiersza (co ma złożoność $O(V)$, bo trzeba pamiętać że w skrajnym przypadku wierzchołek końcowy danej krawędzi może znajdować się w ostatniej kolumnie).

Operacje te mają więc złożoność $O(V \cdot E)$. Im więcej krawędzi będzie w grafie, tym dłużej będą się wykonywały podstawowe operacje (przynajmniej w przypadku pesymistycznym). Ta reprezentacja zajmuje również dużo pamięci, której złożoność możemy opisać poprzez $O(V \cdot E)$. Uzasadnienie takiej złożoności jest banalnie proste – macierz ma szerokość $V$ i wysokość $E$, wobec czego wszystkich komórek jest dokładnie $V \cdot E$.

Zaletą tej reprezentacji (której moglibyśmy uraczyć, np. w przypadku zajmowania się multigrafami) jest rozróżnialność krawędzi. Moglibyśmy, przykładowo, każdej z nich przypisać jakąś etykietę, albo wagę.

### 2.1.3 Lista następników

Trzecią reprezentacją, zaimplementowaną w projekcie, jest lista następników (lista sąsiedztwa). Jest to chyba najprostsza i najbardziej intuicyjna reprezentacja. Graf jest tablicą list, opisujących poszczególne wierzchołki. Te wewnętrzne listy zawierają odniesienia do wierzchołków, z którymi dany wierzchołek ma połączenie (w przypadku grafu skierowanego, bardziej zasadne jest mówienie o liście następników).

Mówiąc bardziej technicznie, rolę kontenera pełni obiekt `std::vector<std::list<int>>`. W tym przypadku sytuacja jest odwrotna, niż w przypadku macierzy incydencji. Liczba wierzchołków jest stała, więc nie będzie często rozszerzana, za to często może być potrzeba losowego dostępu, co motywuje użycie `std::vector`. Natomiast wewnętrzne listy będą służyły wyłącznie do iterowania po nich, dołączania nowych elementów oraz usuwania starych, dlatego do ich realizacji użyłem `std::list`.

Każda z wewnętrznych list przechowuje następników danego wierzchołka. Rozmiar listy przypisanej do każdego wierzchołka jest równy stopniowi wyjściowemu tego wierzchołka. Całkowite zużycie pamięci zależy od sumy rozmiarów wszystkich list wewnętrznych, co możemy sprowadzić do sumy stopni wszystkich wierzchołków. Z elementarnych własności grafu wiemy, że suma stopni wszystkich wierzchołków grafu zależy liniowo od liczby krawędzi (w przypadku grafu nieskierowanego jest to dwukrotność tej liczby). Wobec tego ta reprezentacja grafu ma złożoność pamięciową $O(E)$.

Takie rozwiązanie jest oszczędne pamięciowo, ale w części przypadków wolniejsze niż w przypadku macierzy sąsiedztwa. Dodawanie krawędzi, przy odpowiedniej wewnętrznej implementacji listy, będzie rzędu $O(1)$, ale znajdywanie krawędzi i ich usuwanie to operacje liniowe o złożoności, którą pozwoliłem sobie zapisać w takiej postaci: $O(deg(V_n))$. Aby znaleźć dany wierz-

chołek wśród wszystkich następników, musimy bowiem przeiterować po całej wewnętrznej liście, której rozmiar jest równy stopniowi danego wierzchołka.

Nieco ciekawiej zaczyna się robić w przypadku grafu skierowanego i metody `getPredecessors`, ona będzie miała złożoność $O(E)$ – algorytm musi przejść po całym grafie i sprawdzić, czy przypadkiem któryś z wierzchołków nie ma szukanego wierzchołka jako następnika.

Jesteśmy w stanie natychmiast określić stopień danego wierzchołka i jego następników (w przypadku grafu skierowanego – tylko stopień wychodzący, poszukiwanie poprzedników nie odbywa się natychmiast, ale jest zależne liniowo od liczby krawędzi w grafie, jak wspomniałem przed chwilą).

Trzeba jednak mieć na uwadze, że jeżeli nie poczyniłbym założenia o stałej liczbie wierzchołków, ta reprezentacja mogłaby być nawet skuteczniejsza niż macierz sąsiedztwa, której rozszerzanie o dodatkowe wierzchołki mogłoby być dość czasochłonne.

### 2.1.4 Podsumowanie

Jak widać, każda z tych reprezentacji ma swoje wady i zalety i może znaleźć zastosowania w różnych sytuacjach. Uważam jednak, że przy założeniach tego projektu, najlepiej sprawdzi się macierz sąsiedztwa. W przypadku większości operacji elementarnych ma ona największą wydajność, a przecież to jest dziś kluczowe. Pamięć jest zasobem tanim i stosunkowo prostym do zdobycia, natomiast czasu nikt nikomu nie zwróci.

Macierz sąsiedztwa przegrywa z listą następników, jeżeli chodzi o zużycie pamięci – w większości wypadków nie powinno mieć to wielkiego znaczenia, z drugiej strony może mieć, kiedy zaczynamy myśleć o naprawdę dużych grafach. Ale być może w tak dużych grafach wydajność macierzy sąsiedztwa okaże się jeszcze większą zaletą?

Nieco trudno było mi wskazać zalety macierzy incydencji. Jedyne, co na ten moment przychodzi mi do głowy, to rozróżnialność krawędzi, którym możemy przypisywać rozmaite wagi, etykiety. Taka reprezentacja mogłaby się więc sprawdzić w przypadku pewnych szczególnych i niedużych multigrafów.

Moje przewidywania znajdą potwierdzenie lub zaprzeczenie w wynikach testów, które zostaną omówione niżej.

## 2.2 Generowanie grafów

Bardzo istotną częścią projektu jest moduł generujący grafy. Pozwala to zautomatyzować testowanie rozmaitych algorytmów, oraz wykonywanie testów wydajnościowych.

Każdy generator jest klasą, która dziedziczy po klasie abstrakcyjnej `Generator` i implementuje metodę `generate()`. Mówiąc ściślej, generator jest szablonem klasy, dzięki temu możliwe jest używanie go dla różnych implementacji grafu. Metoda `generate()` zwraca utworzony i odpowiednio wypełniony obiekt grafu danej implementacji (typ klasy jest przekazywany poprzez szablon). Metoda ta przyjmuje dwa argumenty: rozmiar grafu (wyrażany w liczbie wierzchołków) oraz stopień nasycenia (procentowa liczba krawędzi w stosunku do maksymalnej dla danego typu grafu).

Projekt zawiera kilka klas generujących. Nie będę ich w tym miejscu jakoś bardzo szczegółowo omawiał, ograniczę się do kilku najistotniejszych słów.

Jednym z generatorów jest `IterativeGenerator`, który generuje graf w sposób deterministyczny i czyni to *po kolei* (kolejne krawędzie łączą dany wierzchołek z kolejnymi). Dla grafu skierowanego powstanie graf acykliczny, co jest ważne w przypadku testowania algorytmów sortowania topologicznego.

Kolejnym uniwersalnym generatorem jest `RandomGenerator`, który łączy w grafie dowolne wierzchołki, które jeszcze nie zostały połączone. Zastosowanie losowości pozwala wykrywać rozmaite błędy w implementacjach algorytmów.

Projekt posiada na chwilę obecną jeszcze cztery generatory, które są jednak w pewnym stopniu ze sobą powiązane. O dowolnym grafie nieskierowanym możemy powiedzieć to, że zawiera cykl Hamiltona, albo go nie zawiera – podobnie jest z cyklem Eulera. Generatory te tworzą więc grafy o zadanej liczbie wierzchołków i stopniu nasycenia, dla wszystkich czterech przypadków (graf zawiera oba cykle, nie zawiera żadnego, albo zawiera po jednym). Tworzenie tych generatorów sprawiło mi najwięcej kłopotu.

Wygenerowanie grafu Hamiltona jest prostą sprawą. Wystarczy połączyć wszystkie wierzchołki w cykl (składający się z $N$ krawędzi, jeżeli mamy $N$ wierzchołków w grafie), a następnie możemy dokładać dowolną liczbę krawędzi i nie zepsujemy cyklu. Sprawienie, żeby graf nie miał na pewno cyklu Hamiltona jest również bardzo proste. Wystarczy już na samym początku wyizolować jeden z wierzchołków (w moim przypadku ostatni – traktowałem

graf, jak gdyby miał o jeden wierzchołek mniej). Graf z izolowanym wierzchołkiem na pewno nie jest grafem Hamiltona, gdyż nie jesteśmy w stanie w żaden sposób do tego wierzchołka dotrzeć (albo z niego dotrzeć do pozostałych).

Generowanie grafu Eulera sprawiło mi zdecydowanie więcej trudności, choć ostatecznie zastosowałem metodę, która okazała się być całkiem prosta. W pierwszym podejściu spróbowałem tego dokonać metodą siłową, z nawrotami. Po prostu dokładałem kolejne krawędzie do grafu i spróbowałem skonstruować $N$-krawędziowy cykl. Jeżeli było to niemożliwe, cofałem się do poprzedniego wierzchołka. Taka metoda okazała się skuteczna dla pewnych konkretnych kombinacji liczby wierzchołków i stopnia nasycenia. Zdarzają się jednak takie pary, dla których wygenerowanie grafu Eulera nie jest możliwe. W takim wypadku czas generacji był bardzo długi – a przecież nie o to chodziło. Kolejne podejście opierało się na podziale sumy stopni wszystkich wierzchołków (dwukrotności liczby krawędzi) na poszczególne wierzchołki, tak aby każdy z tych wierzchołków miał parzysty stopień. Jednak, jak się okazało, nie było to takie proste, gdyż taki podział (w anglojęzycznej literaturze spotkałem się z terminem *degree sequence*) musi spełniać odpowiednie warunki (mówi o tym twierdzenie Erdős–Gallai).

Ostatecznie zastosowałem metodę, o której napomknięto w treści zadania – choć nie ukrywam, że nie zrozumiałem na samym początku tego zamysłu. Najpierw wygenerowałem minimalny cykl Hamiltona (zapewnił on spójność grafu). Następnie z pozostałych krawędzi, które miały zostać wstawione do grafu, zacząłem układać trzy-krawędziowe cykle (dodanie cyklu nie psuje parzystości stopni wierzchołków). Taka metoda pozwala na wygenerowanie grafu Eulera o liczbie zadanych krawędzi, z maksymalnym błędem bezwzględnym wynoszącym 1.

Podobnie wygenerowałem graf, który jest wyłącznie grafem Eulera. Najpierw wygenerowałem cykl zawierający $N - 1$ wierzchołków, by zapewnić spójność grafu, a następnie wstawiałem takie trzy-elementowe cykle – pomijając jednak jeden z wierzchołków, który jest izolowany.

Wygenerowanie grafu, który nie jest grafem Eulera jest bardzo proste, jeżeli potrafimy już wygenerować graf Eulera. Wystarczy dodać krawędź pomiędzy dowolnymi dwoma wierzchołkami o parzystym stopniu, które jeszcze nie zostały połączone. W ten sposób zepsujemy ich parzystość.

Na koniec tej części wspomnę jeszcze tylko, że tworzenie generatora grafów zainspirowało mnie do matematycznych rozważań i poszukiwań na temat grafu Eulera. Zainteresowały mnie następujące problemy:

- Ile jest możliwych różnych cyklów Eulera w grafie pełnym $K_n$?
- Czy możliwe jest utworzenie grafu o dokładnie N wierzchołkach i E krawędziach, który jest grafem Eulera?

Rozwiązanie pierwszeg z problemów znalazłem już w pewnej pracy: http://www.algana.co.uk/publications/Counting.pdf

Drugi nadal jednak *chodzi mi po głowie*. Zastanawiam się, czy jest możliwe sformułowanie jakiegoś warunku, który pozwoliłby odpowiedzieć na to pytanie i jednocześnie nie wymagał sprawdzenia, w skrajnym przypadku, wszystkich możliwych kombinacji (co miałoby złożoność $O(V!)$).

## 2.3 Przeszukiwanie grafu

Przeszukiwanie grafu jest podstawową operacją, jaką możemy na nim wykonać. Polega to na tym, że zaczynamy od pewnego wierzchołka i staramy się dotrzeć do wszystkich możliwych wierzchołków. Dzięki temu możemy przykładowo zweryfikować, czy graf jest spójny (lista wygenerowana przez algorytm przeszukiwania grafu powinna mieć rozmiar równy liczbie wierzchołków w grafie).

W tym projekcie zaimplementowałem dwa rodzaje przeszukiwania grafu: BFS (z ang. *Breath First Search*) oraz DFS (z ang. *Deep First Search*).

Pierwszy z algorytmów jest bardzo intuicyjny, nie jest też rekurencyjny. Zaczynamy w pewnym wierzchołku, od razu możemy go uznać za osiągnięty (i wpisać do wynikowej listy). Następnie dodajemy wszystkie jego następniki, które nie zostały jeszcze osiągnięte, do pewnej kolejki. Podczas dotarcia do każdego z wierzchołków, również wpisujemy go na listę, oznaczamy jako osiągnięty i ponownie dodajemy jego nieodwiedzone następniki do kolejki. Proces ten powtarzamy tak długo, aż kolejka nie będzie pusta.

Drugi z algorytmów (DFS) ma zupełnie inne podejście, moim zdaniem trochę mniej intuicyjne, ale zdecydowanie prostsze w implementacji, wykorzystujące rekurencję. Znajdując się w jakimś wierzchołku, wykonujemy algorytm przeszukiwania rekurencyjnie, po wszystkich nieodwiedzonych jeszcze wierzchołkach. Przy odwiedzeniu wierzchołka, oznaczamy go jako odwiedzony i dodajemy do listy. Jeżeli *zakopiemy się* tak głęboko, że z danego wierzchołka nie prowadzą już żadne krawędzie do nieodwiedzonych wierzchołków, wykorzystujemy cechę rekurencji, która w momencie zakończenia wykonania instancji funkcji, *cofa się* i przechodzi do kolejnego wierzchołka.

W przypadku algorytmów, które operują już na pewnej warstwie abstrakcji, nie możemy podać złożoności wprost, gdyż będzie ona różna dla różnych reprezentacji grafu.

Algorytm BFS rozpoczyna działanie od pewnego wierzchołka grafu. Później, co logiczne, musi przejść po wszystkich pozostałych wierzchołkach (jest to przypadek pesymistyczny, gdyż może się zdarzyć że graf nie będzie spójny, albo będzie posiadał izolowane wierzchołki, wtedy algorytm BFS nie pozwoli nam dostać się do każdego wierzchołka). Do implementacji wykorzystano kolejkę, która jest w zasadzie listą – dodawanie i zdejmowanie elementów jest w czasie $O(1)$. Jedyną metodą grafową, którą wykorzystuję, jest `getSuccessors`, która w przypadku listy sąsiedztwa jest niemal natychmiastowa – odpowiada reprezentacji, jaką przechowuje graf. W przypadku macierzy sąsiedztwa, zdobycie listy następników jest operacją liniową, zależną od liczby wierzchołków w grafie. W przypadku macierzy incydencji, w pesymistycznym przypadku, może mieć to złożoność nawet $O(V \cdot E)$. Wobec powyższego, złożoności algorytmu BFS dla poszczególnych reprezentacji będą następujące:

- macierz sąsiedztwa – $O(V^2)$
- lista sąsiedztwa – $O(V)$
- macierz incydencji – $O(V^2 \cdot E)$

Algorytm DFS działa na bardzo podobnej zasadzie, jak algorytm BFS. Również musi przebyć jednokrotnie wszystkie wierzchołki, również wykorzystuje wyłącznie metodę `getSuccessors`. Robi dokładnie to samo co BFS, z tym że wykorzystuje rekurencję, zamiast dedykowanej kolejki. Wobec tego złożoności obliczeniowe tego algorytmu dla wszystkich reprezentacji będą identyczne, jak w przypadku BFS.

## 2.4 Sortowanie topologiczne

Sortowanie topologiczne jest operacją, którą możemy wykonać na grafie skierowanym i acyklicznym. Zamysł jest taki, żeby wszyscy poprzednicy poszczególnych wierzchołków zostali wypisani przed ich następnikami. Jeżeli graf skierowany jest acykliczny, to na pewno istnieją wierzchołki o stopniu wejściowym równym zero (bez poprzedników) i to od nich należy rozpocząć wypisywanie. Jeżeli usuniemy te wierzchołki (a mówiąc ściślej, usuniemy

wszystkie krawędzie usunięte z tymi wierzchołkami) to pojawią się kolejne wierzchołki o stopniu wejściowym równym zero i tak dalej.

Z powyżej opisanej koncepcji korzysta algorytm BFS. Na samym początku inicjalizujemy pewną kolejkę wszystkimi wierzchołkami o stopniu wchodzącym równym zero. Następnie, dopóki ta kolejka nie jest pusta, pobieramy z niej wierzchołek i umieszczamy go w liście wynikowej. Następnie, usuwamy krawędź łączącą wierzchołek z każdym z jego następników, a następnie dodajemy dany wierzchołek do kolejki, jeżeli tylko jego nowy stopień wejściowy jest równy zero.

Innym algorytmem sortowania topologicznego, jest algorytm DFS. Podobnie jak przy przeszukiwaniu grafu, jest on oparty o rekurencję. Wchodzi w głąb grafu i oznacza odpowiednie wierzchołki jako odwiedzone, a po wykonaniu się procedur rekurencyjnych dla następnika danego wierzchołka, wstawia go na początek listy wynikowej. Możliwe jest rozszerzenie kolorowania grafu z dwóch wartości (nieodwiedzony, odwiedzony) do trzech wartości (nieodwiedzony, przetwarzany, odwiedzony). Pozwala to na wykrycie cykli w grafie. W swojej implementacji również zastosowałem taką metodę.

Ocena złożoności obliczeniowych w przypadku algorytmów sortowania topologicznego znowu nie jest taka oczywista. Każda z reprezentacji grafu ma inną złożoność operacji elementarnych. Tym razem algorytmy różnią się nieco pod względem działania – na przykład algorytm BFS usuwa krawędzie, czego nie robi algorytm DFS. Usuwanie krawędzi w reprezentacjach bazujących na listach (lista następników, macierz krawędzi) może być bardzo czasochłonne, co potwierdzają wyniki benchmarku, które pojawią się w dalszej części tego sprawozdania.

W algorytmie BFS znajdują się dwie rozłączne pętle. Pierwsza z nich iteruje po wszystkich wierzchołkach i sprawdza wejściowy stopień każdego z nich – co jest operacją bardzo czasochłonną. Dla macierzy sąsiedztwa jest to operacja liniowa o złożoności $O(V)$, dla listy następników – $O(E)$, dla macierzy incydencji – również $O(E)$. Cała pętla ma dla macierzy sąsiedztwa złożoność $O(V^2)$, dla listy następników i dla macierzy incydencji – $O(V \cdot E)$.

Druga pętla (wraz z zagnieżdżoną w niej pętlą) również musi przejść po wszystkich krawędziach, usuwając je i sprawdzając stopnie wierzchołków z nią incydentnych. Usuwanie jest również operacją nieco czasochłonną, ale nie aż tak jak sprawdzanie stopnia wejściowego. Dla macierzy sąsiedztwa jest to $O(1)$, dla listy sąsiedztwa w przypadku pesymistycznym $O(deg(V_n))$, a dla macierzy incydencji już $O(E)$. Dla wszystkich reprezentacji, sprawdzenie stopnia wejścowego ma większą złożoność. Dlatego złożoność ciała pętli

dla macierzy sąsiedztwa wynosi $O(V)$, a dla listy następników i macierzy incydencji – $O(E)$. Cała druga pętla będzie więc miała złożoność $O(V \cdot E)$ dla macierzy sąsiedztwa, $O(E^2)$ dla macierzy incydencji i listy sąsiedztwa.

Z racji, że mieliśmy dwie rozłączne pętle, ostateczna złożoność będzie sumą złożoności składowych. Prezentuje się ona następująco dla poszczególnych reprezentacji:

- macierz sąsiedztwa – $O(V^2 + V \cdot E)$
- lista sąsiedztwa – $O(E^2 + V \cdot E)$
- macierz incydencji – $O(E^2 + V \cdot E)$

Algorytm DFS musi z kolei jednokrotnie przejść po każdym wierzchołku (rekurencyjnych wywołań pewnej metody jest więcej, ale kończą się one, jeżeli wierzchołek został oznaczony jako odwiedzony). Jedyną operacją wykonywaną na grafie jest pobranie listy następników. Dla macierzy sąsiedztwa jest to operacja o złożoności $O(V)$, dla listy sąsiedztwa – $O(1)$, a dla macierzy incydencji w pesymistycznym przypadku $O(V \cdot E)$. Podsumowując, złożoność mojej implementacji algorytmu DFS będzie wynosiła:

- macierz sąsiedztwa – $O(V^2)$
- lista sąsiedztwa – $O(V)$
- lista sąsiedztwa – $O(V^2 \cdot E)$

## 2.5 Poszukiwanie cyklów Eulera i Hamiltona

Do wyszukiwania cyklów Eulera i Hamiltona w grafie nieskierowanym przygotowałem implementacje algorytmów z nawracaniem. Algorytm stara się wyznaczyć cykl w najprostszy możliwy sposób (biorąc zawsze pierwsze wolne wierzchołki jako kolejne wyrazy ciągu opisującego cykl). Jeżeli się to uda – kończy działanie. Jeśli nie – wykonuje nawrót poprzez cofnięcie się do poprzedniego wierzchołka.

Moją intencją od początku było ulepszenie oryginalnej koncepcji tak, aby użytkownik mógł znaleźć dowolną liczbę cyklów. W większości wypadków byłby to pewnie tylko jeden, ale w niektórych sytuacjach byłyby to wszystkie cykle. Byłoby to łatwiejsze, gdybym miał do dyspozycji operator yield. Język C++ jednak takiego nie wspiera, wobec czego zrezygnowałem z rekurencji. Kod wyszedł może nieco dłuższy, niż w oryginalnym zamyśle, ale zdaje się działać poprawnie. Liczba wszystkich wygenerowanych cyklów (zarówno Hamiltona, jak i Eulera) zgadzała się z teorią, dla wszystkich grafów, które sprawdziłem.

Algorytm poszukujący cyklu Hamiltona oraz Eulera różnią się nieznacznie. Ten pierwszy oznacza pewne wierzchołki jako odwiedzone i stara się znaleźć cykl składający się z $|V|$ krawędzi. Przy nawrotach wierzchołki są z powrotem ustawiane jako nieodwiedzone.

Ten drugi z kolei, przy każdym zagłębieniu się usuwa pewną krawędź i stara się znaleźć cykl o $|E|$ krawędziach (tyle, ile jest w grafie). Przy nawrotach krawędzie te są ponownie dodawane.

Taka strategia wymusiła na mnie konieczność zapamiętywania listy następników na każdym poziomie zagłębienia. Może się to wydawać sporym zużyciem pamięci, ale trzeba zdawać sobie sprawę, że rekurencyjne wywołania robią dokładnie to samo. Każda instancja funkcji przechowuje swoje zmienne lokalne na stosie. W momencie rekurencyjnego wywoływania funkcji, coraz więcej stosu jest zajmowane, co w skrajnym przypadku mogłoby prowadzić do jego przepełnienia.

Ocena złożoności obliczeniowej jest według mnie jeszcze trudniejsza niż poprzednio. W przypadku grafu Hamiltona sprawa może jeszcze nie wydaje się taka trudna. Znalezienie jakiegokolwiek cyklu i tym samym ocena, czy graf taki cykl w ogóle posiada, ma pesymistyczną złożoność $O(V!)$ (nie jestem specjalistą z dziedziny teoretycznej algorytmiki, ale nazwałbym taką złożonością *abstrakcyjną*, gdyż operuje ona na pewnej abstrakcji i nie uwzględnia ona złożoności obliczeniowych operacji elementarnych, uznając że wynoszą one

$O(1)$). W przypadku algorytmu poszukującego cykl Eulera sytuacja się komplikuje. Z jednej strony, znalezienie wszystkich możliwych cyklów może mieć bardzo z góry oszacowaną pesymistyczną złożoność $O(E!)$, z drugiej wiemy że jest to z reguły łatwy proces, a odpowiedzieć sobie na pytanie – czy graf zawiera cykl Eulera, możemy w relatywnie krótkim czasie, bo (znowu operując na pewnej abstrakcji) $O(V)$ (trzeba przeiterować po wszystkich wierzchołkach i sprawdzić ich stopnie).

# 3 Testy wydajności algorytmów

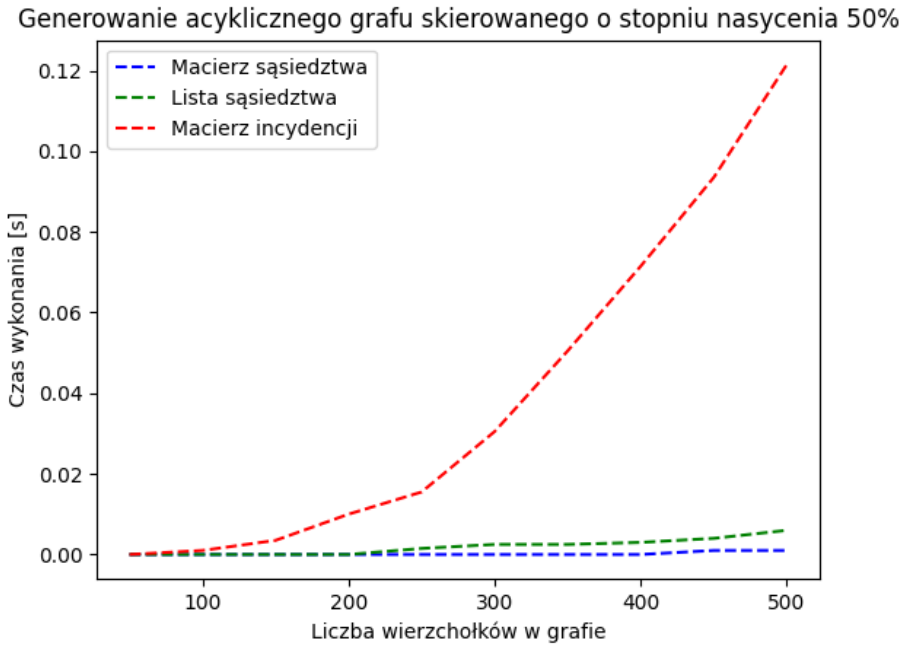
W celu przetestowania wydajności wszystkich algorytmów napisałem program, korzystający z przygotowanej przeze mnie biblioteki *graphs* (to biblioteka, która gromadzi wszystkie omówione powyżej algorytmy i reprezentacje grafów). Wykorzystałem również skrypty do automatyzacji, przygotowane w Pytohnie na potrzeby tego przedmiotu: *benchmark* – który wykonuje automatycznie wszystkie testy, na podstawie konfiguracji zawartej w pliku JSON, oraz *plotter* – który rysuje wykresy na podstawie wyników benchmarku.

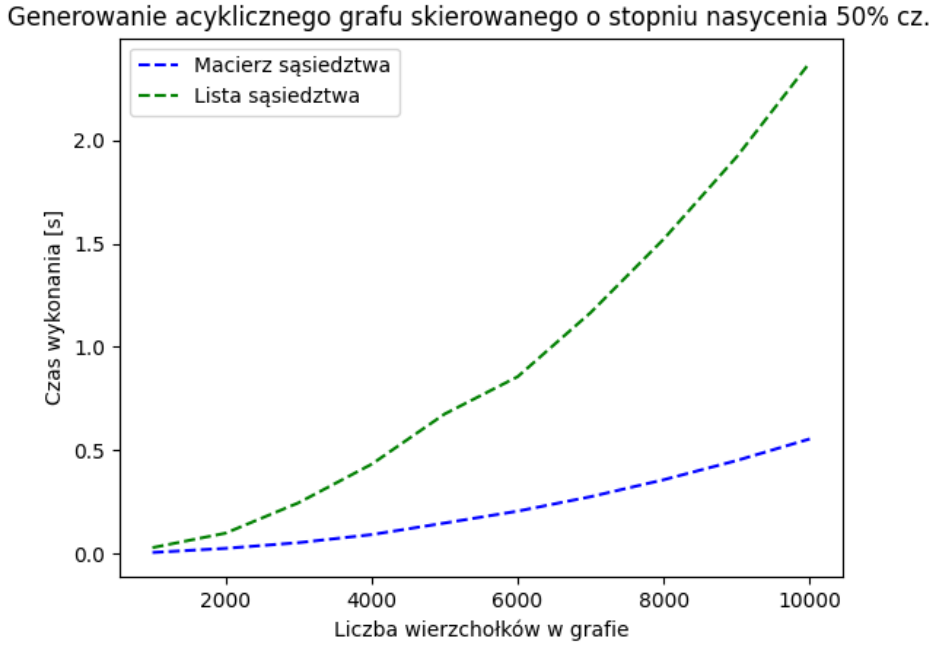
Dołożyłem wszelkich starań, aby wyniki były jak najbardziej rzetelne i mam nadzieję, że takie są. Wygenerowanych wykresów jest stosunkowo dużo, jednak ze względu na pewne problemy z LaTeX, nie udało mi się podzielić tego na podrozdziały (obrazki *lubią* bezładnie *skakać* po stronach).

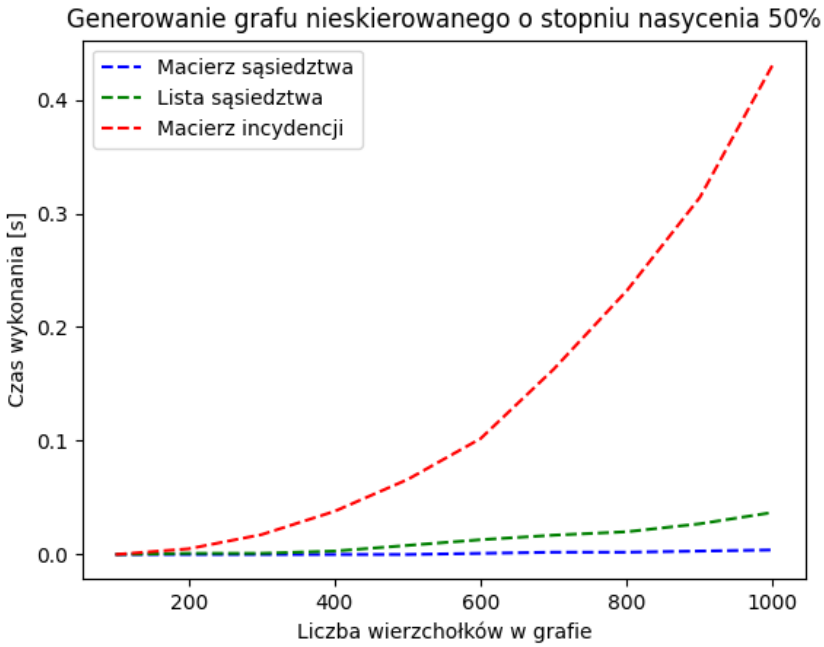
Niektóre wykresy podzieliłem na dwie części, ponieważ często zdarzało się, że macierz incydencji była zdecydowanie powolniejsza, niż pozostałe reprezentacje, czym zakłócała inne wyniki. Dlatego jedna część prezentuje całościowe wyniki dla trzech reprezentacji, a druga prezentuje wyniki na większych grafach, ale tylko dla listy sąsiedztwa i macierzy sąsiedztwa.

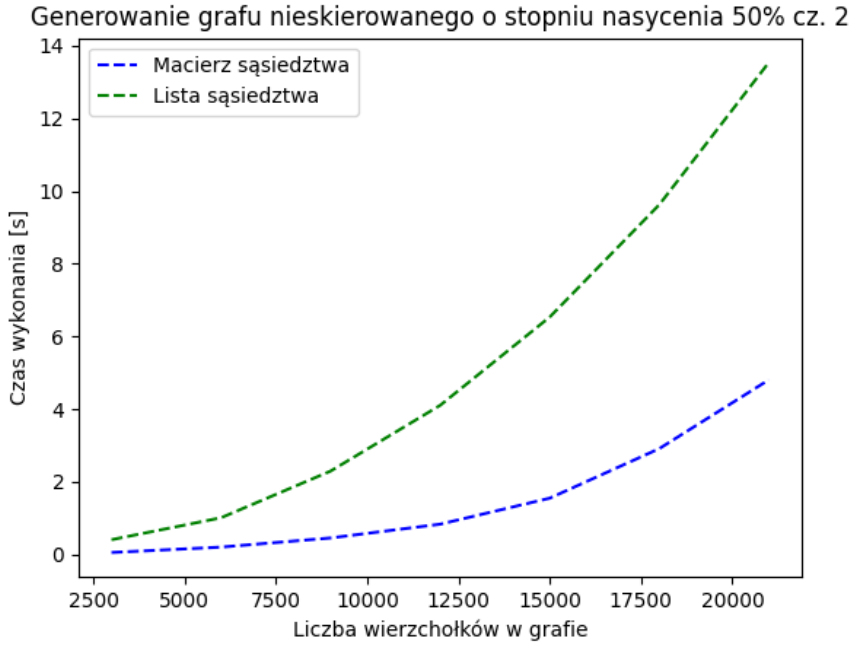
## 3.1 Generowanie grafów

W tej sekcji znajdują się wyniki benchmarku dla algorytmów generujących acykliczny graf skierowany oraz graf nieskierowany.


Generowanie acyklicznego grafu skierowanego o stopniu nasycenia 50%
Macierz sąsiedztwa
Lista sąsiedztwa
Macierz incydencji
Czas wykonania [s]
0.12
0.10
0.08
0.06
0.04
0.02
0.00
100
200
300
400
500
Liczba wierzchołków w grafie

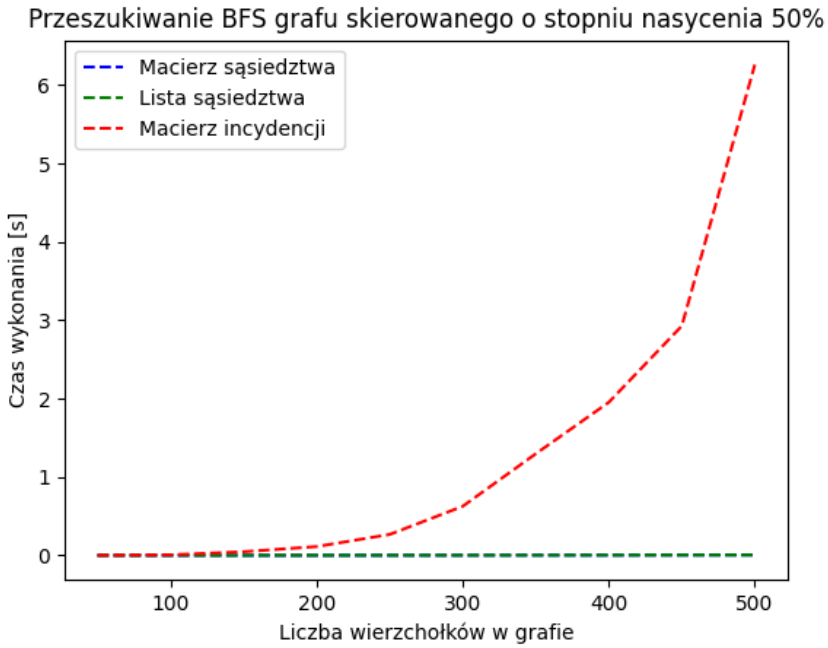

Generowanie acyklicznego grafu skierowanego o stopniu nasycenia 50% cz.
Macierz sąsiedztwa
Lista sąsiedztwa
Czas wykonania [s]
2.0
1.5
1.0
0.5
0.0
2000
4000
6000
8000
10000
Liczba wierzchołków w grafie

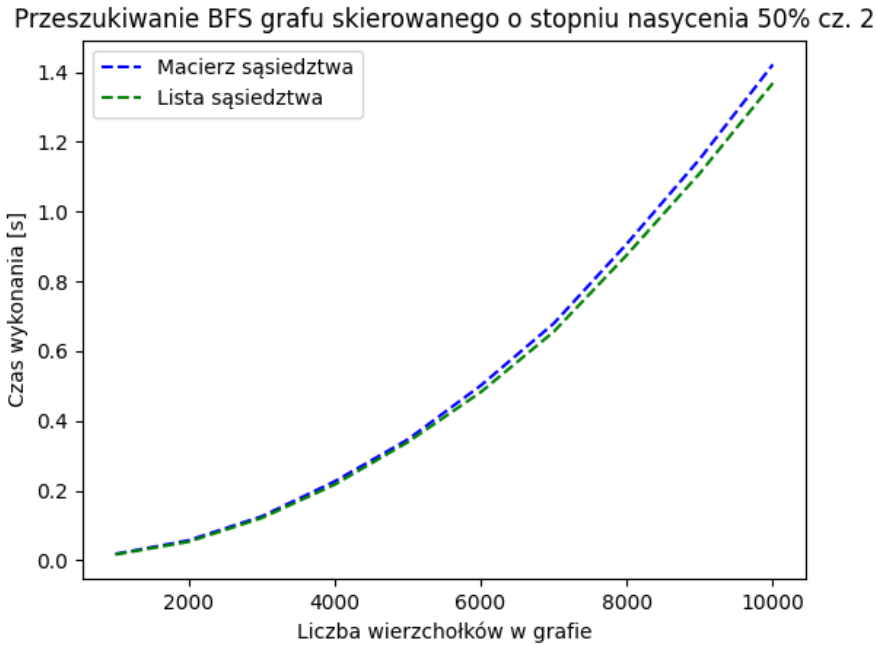

Generowanie grafu nieskierowanego o stopniu nasycenia 50%
Macierz sąsiedztwa
Lista sąsiedztwa
Macierz incydencji
Czas wykonania [s]
0.0
0.1
0.2
0.3
0.4
200
400
600
800
1000
Liczba wierzchołków w grafie

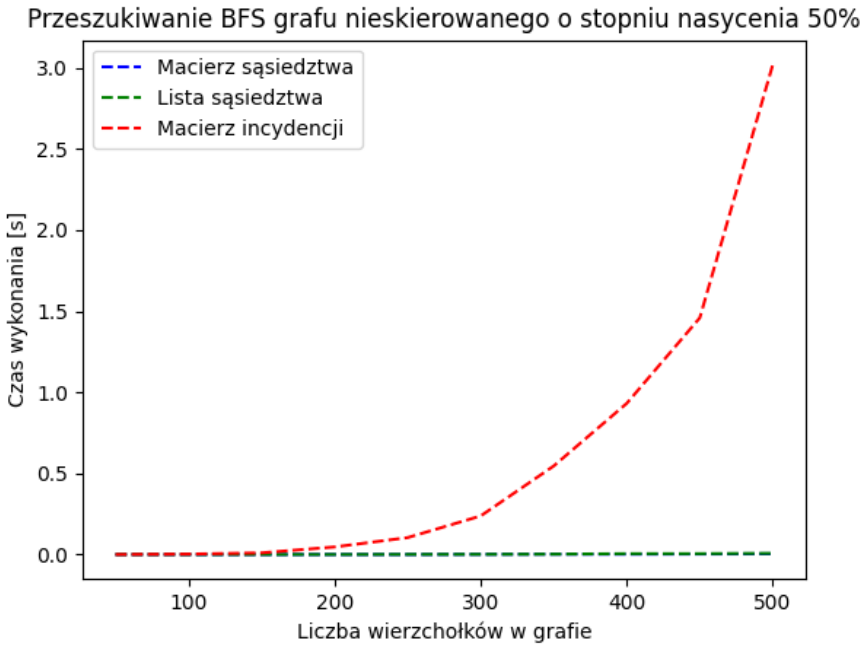

Generowanie grafu nieskierowanego o stopniu nasycenia 50% cz. 2
Macierz sąsiedztwa
Lista sąsiedztwa
Czas wykonania [s]
0
2
4
6
8
10
12
14
2500
5000
7500
10000
12500
15000
17500
20000
Liczba wierzchołków w grafie

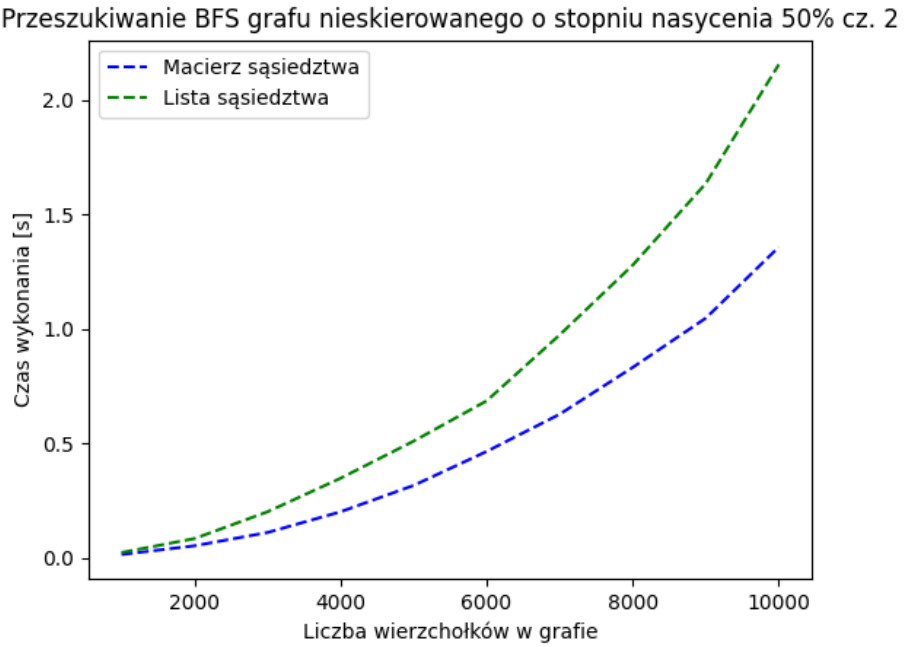
## 3.2 Przeszukiwanie grafu

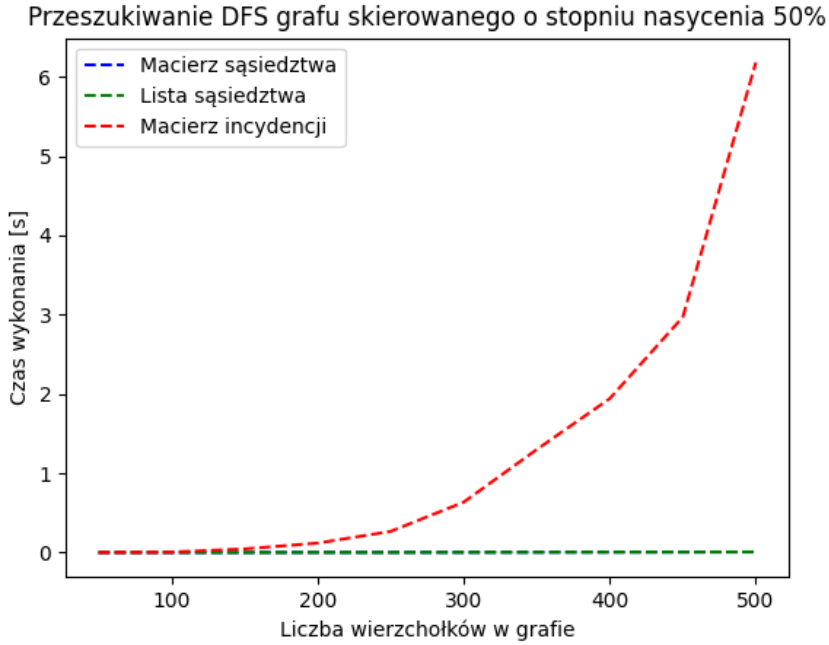
W tej sekcji przedstawię wyniki testów algorytmów DFS oraz BFS przeszukujących grafy skierowane oraz nieskierowane.

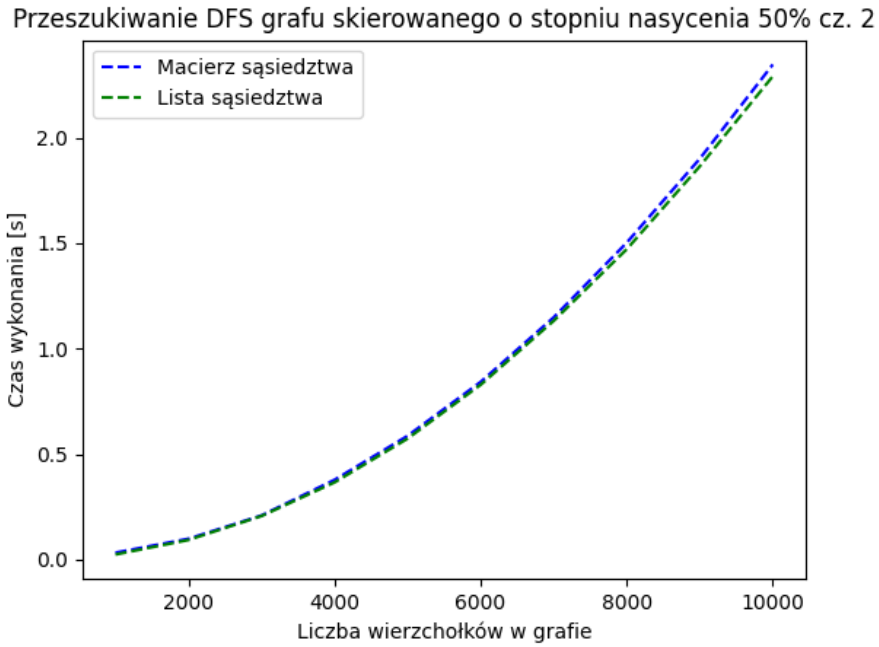


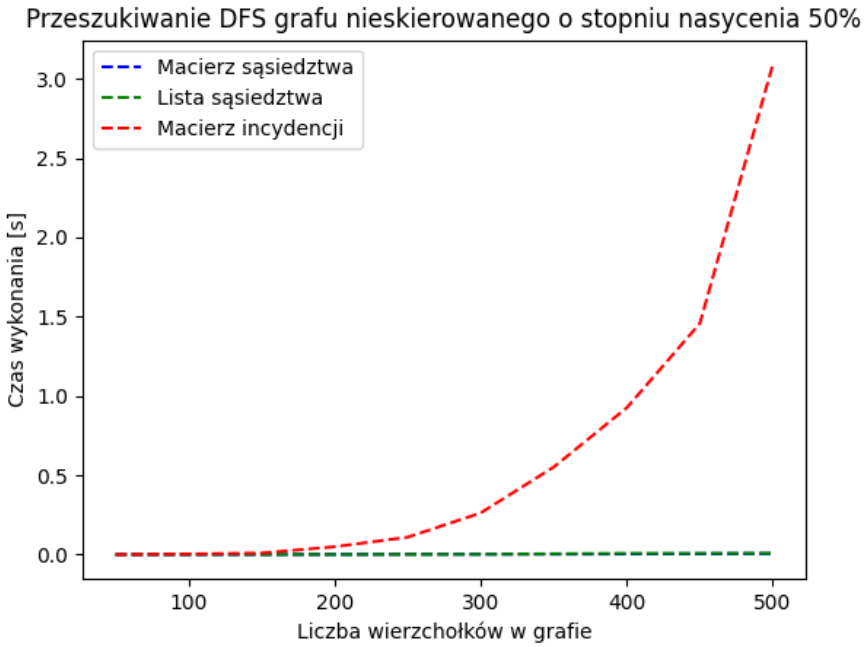

Przeszukiwanie BFS grafu skierowanego o stopniu nasycenia 50% cz. 2
Macierz sąsiedztwa
Lista sąsiedztwa
Czas wykonania [s]
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
2000
4000
6000
8000
10000
Liczba wierzchołków w grafie

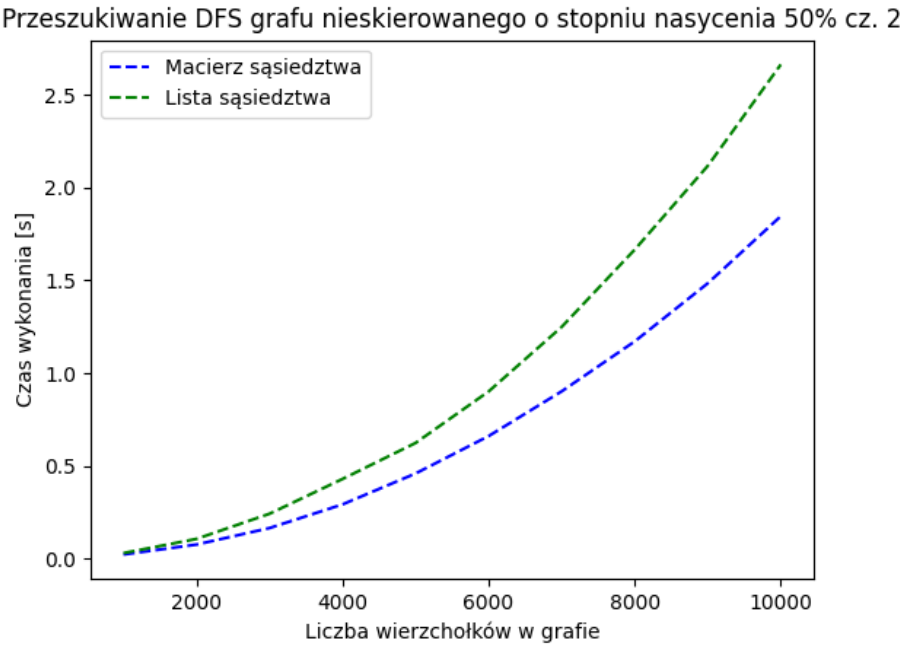

Przeszukiwanie BFS grafu nieskierowanego o stopniu nasycenia 50%
Macierz sąsiedztwa
Lista sąsiedztwa
Macierz incydencji
Czas wykonania [s]
0.0
0.5
1.0
1.5
2.0
2.5
3.0
100
200
300
400
500
Liczba wierzchołków w grafie


Przeszukiwanie BFS grafu nieskierowanego o stopniu nasycenia 50% cz. 2
Macierz sąsiedztwa
Lista sąsiedztwa
Czas wykonania [s]
0.0
0.5
1.0
1.5
2.0
2000
4000
6000
8000
10000
Liczba wierzchołków w grafie


Przeszukiwanie DFS grafu skierowanego o stopniu nasycenia 50%
Macierz sąsiedztwa
Lista sąsiedztwa
Macierz incydencji
Czas wykonania [s]
0
1
2
3
4
5
6
100
200
300
400
500
Liczba wierzchołków w grafie

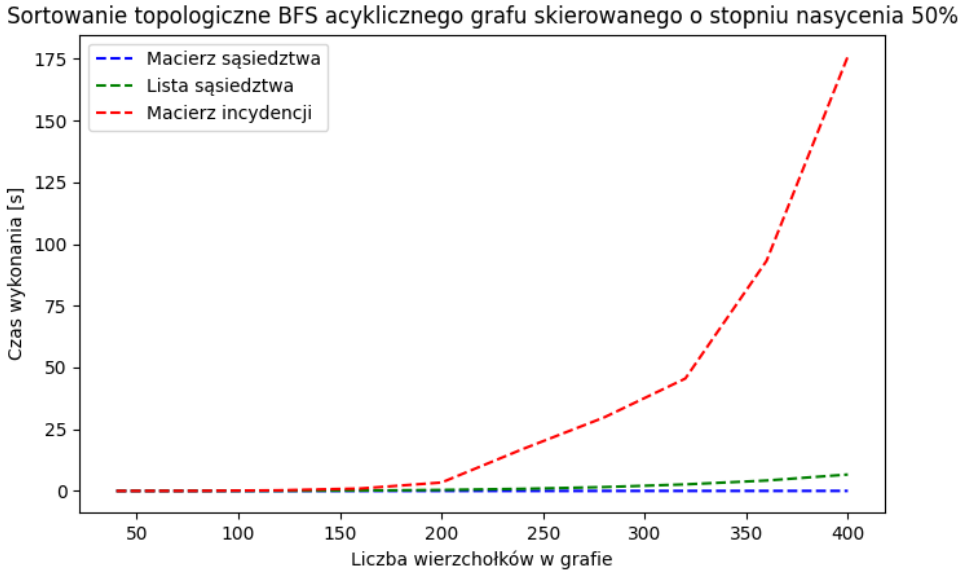

Przeszukiwanie DFS grafu skierowanego o stopniu nasycenia 50% cz. 2
Macierz sąsiedztwa
Lista sąsiedztwa
Czas wykonania [s]
0.0
0.5
1.0
1.5
2.0
2000
4000
6000
8000
10000
Liczba wierzchołków w grafie

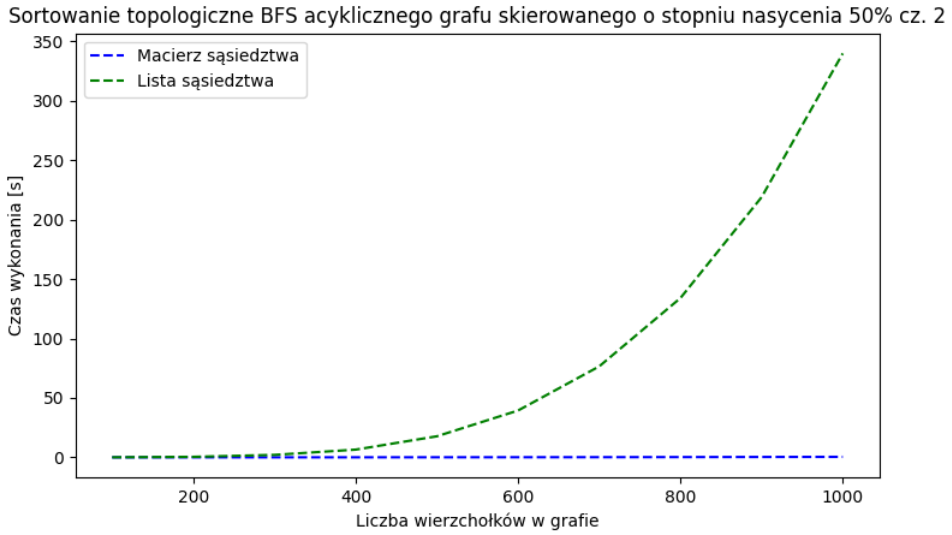

Przeszukiwanie DFS grafu nieskierowanego o stopniu nasycenia 50%
Macierz sąsiedztwa
Lista sąsiedztwa
Macierz incydencji
Czas wykonania [s]
0.0
0.5
1.0
1.5
2.0
2.5
3.0
100
200
300
400
500
Liczba wierzchołków w grafie

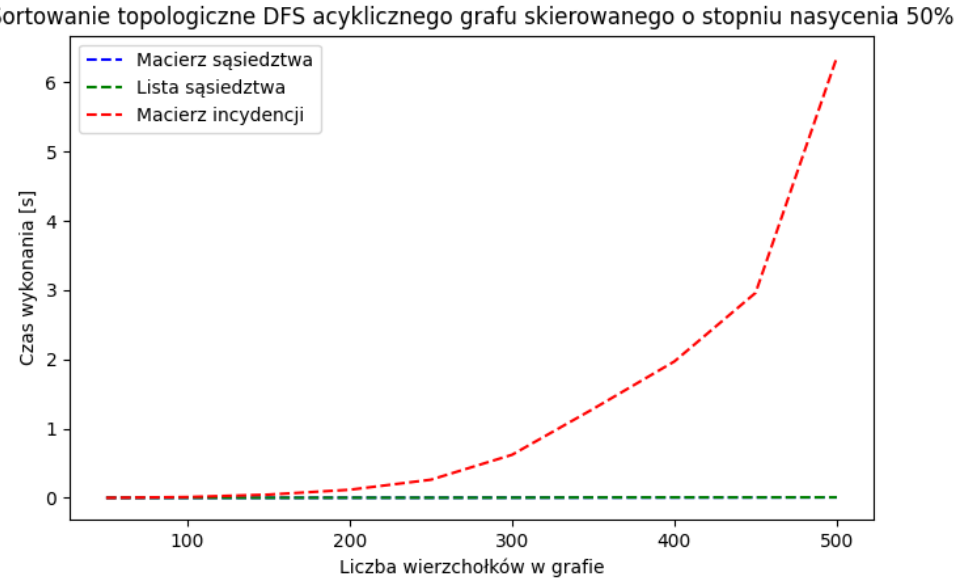


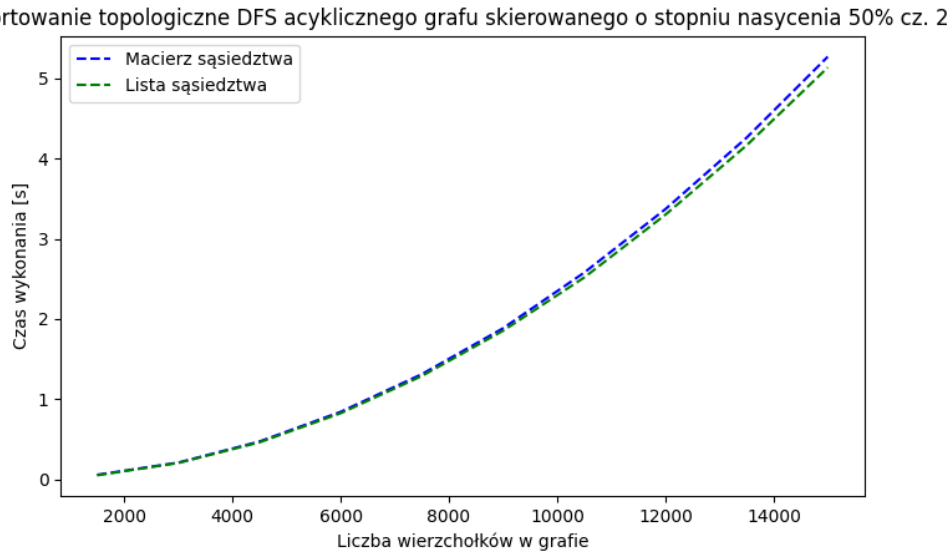

## 3.3 Sortowanie topologiczne

W tej części przedstawię wyniki testów algorytmów sortowania topologicznego BFS oraz DFS dla acyklicznego grafu skierowanego.


Sortowanie topologiczne BFS acyklicznego grafu skierowanego o stopniu nasycenia 50%
Macierz sąsiedztwa
Lista sąsiedztwa
Macierz incydencji
Czas wykonania [s]
Liczba wierzchołków w grafie
0
25
50
75
100
125
150
175
50
100
150
200
250
300
350
400

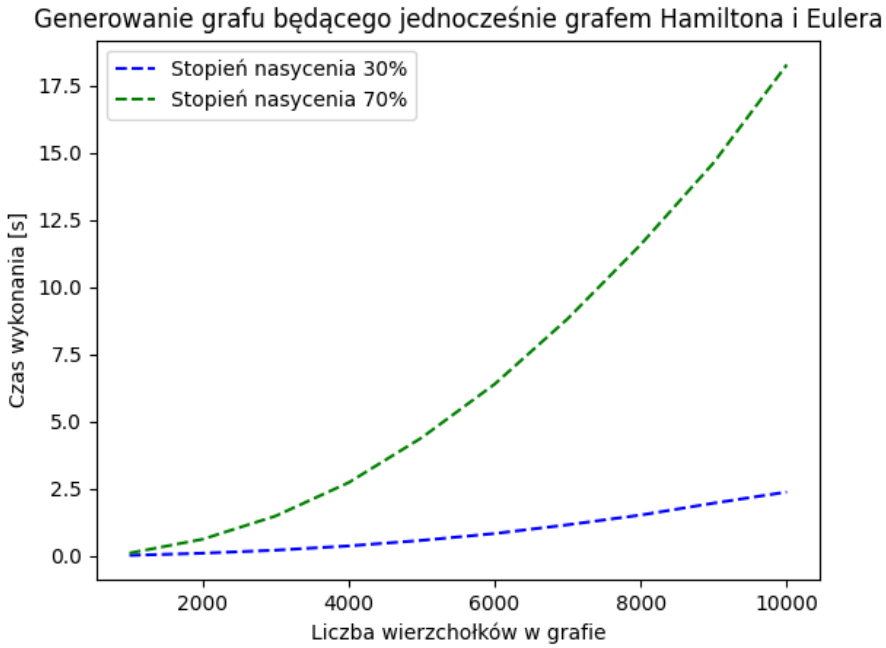

Sortowanie topologiczne BFS acyklicznego grafu skierowanego o stopniu nasycenia 50% cz. 2
Macierz sąsiedztwa
Lista sąsiedztwa
Czas wykonania [s]
Liczba wierzchołków w grafie
0
50
100
150
200
250
300
350
200
400
600
800
1000

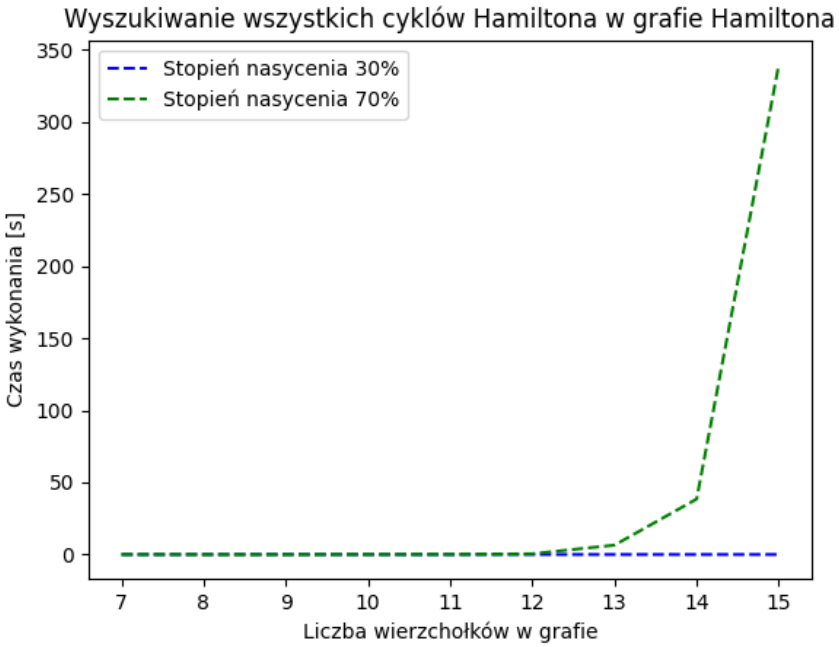

Sortowanie topologiczne DFS acyklicznego grafu skierowanego o stopniu nasycenia 50%
Macierz sąsiedztwa
Lista sąsiedztwa
Macierz incydencji
Czas wykonania [s]
0
1
2
3
4
5
6
100
200
300
400
500
Liczba wierzchołków w grafie

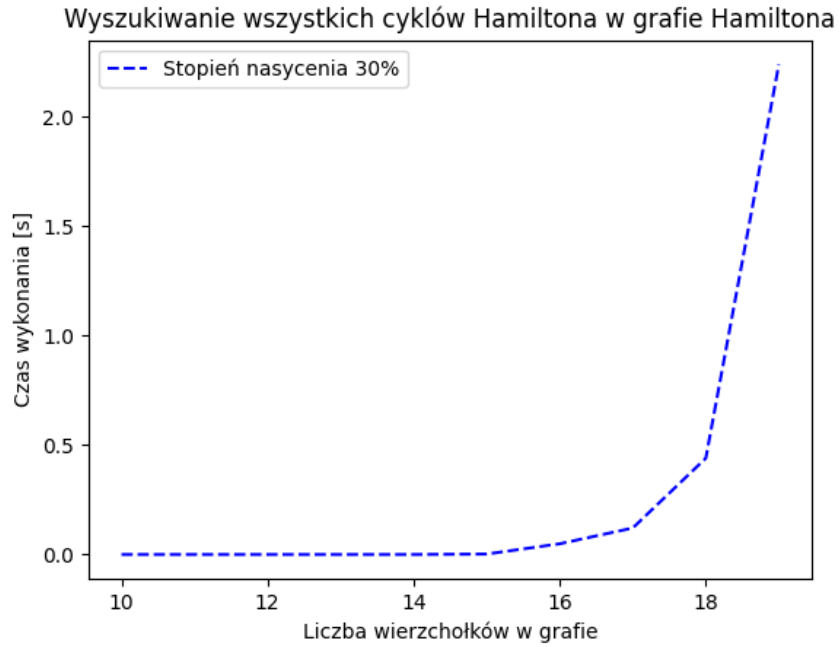

Sortowanie topologiczne DFS acyklicznego grafu skierowanego o stopniu nasycenia 50% cz. 2
Macierz sąsiedztwa
Lista sąsiedztwa
Czas wykonania [s]
0
1
2
3
4
5
2000
4000
6000
8000
10000
12000
14000
Liczba wierzchołków w grafie

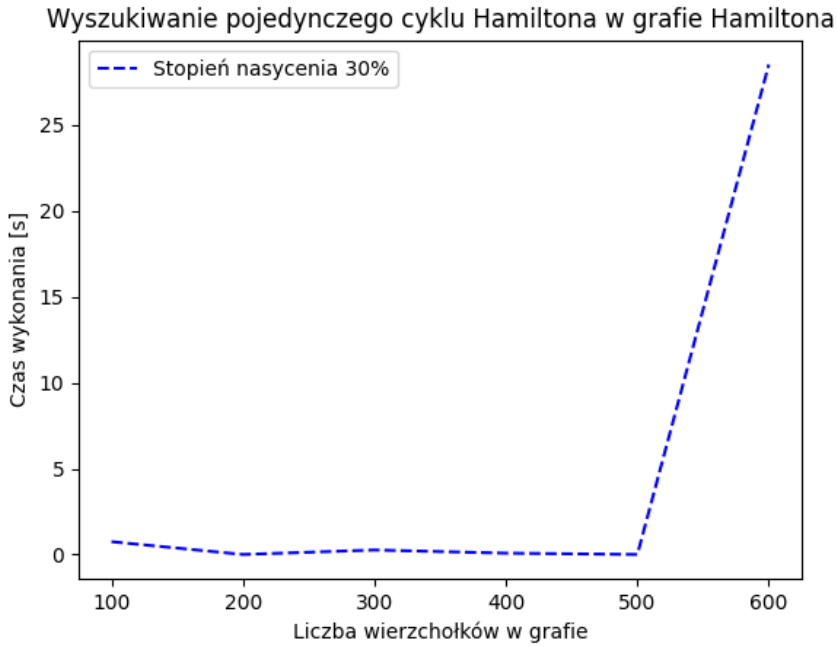
## 3.4 Graf Eulera i Hamiltona

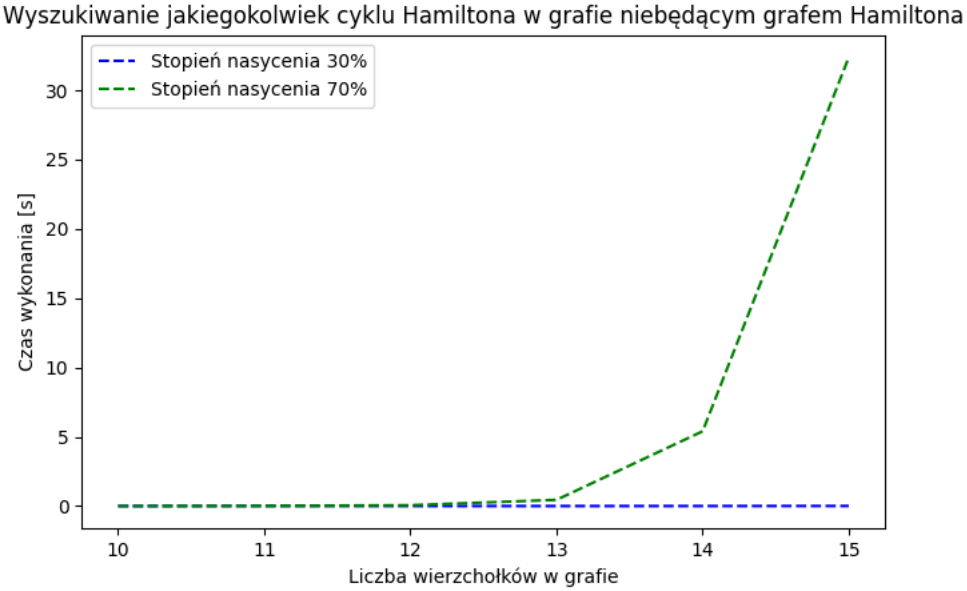
W tej części przedstawię wyniki testów algorytmów generujących grafy Eulera i Hamiltona oraz wyszukujących w nich cykle Eulera i Hamiltona.

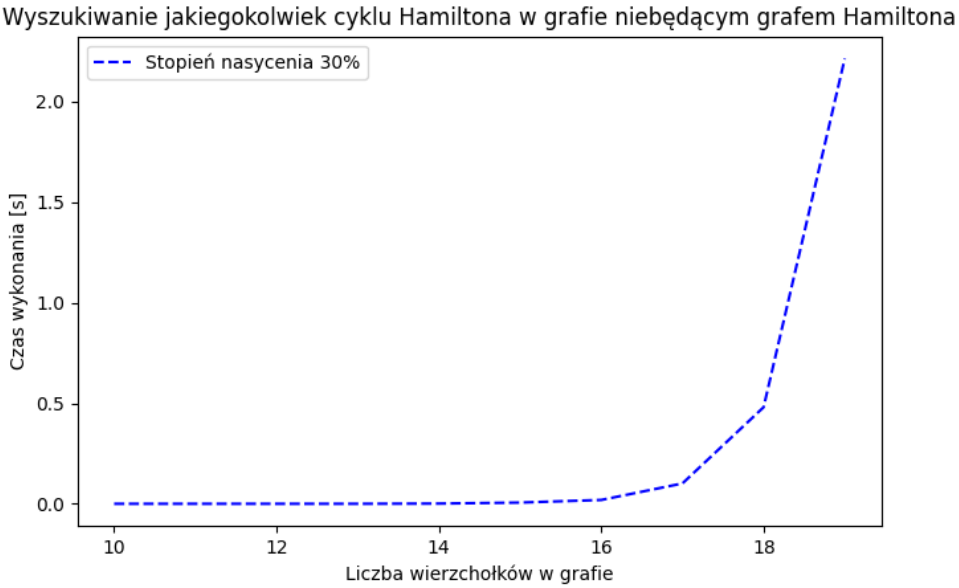


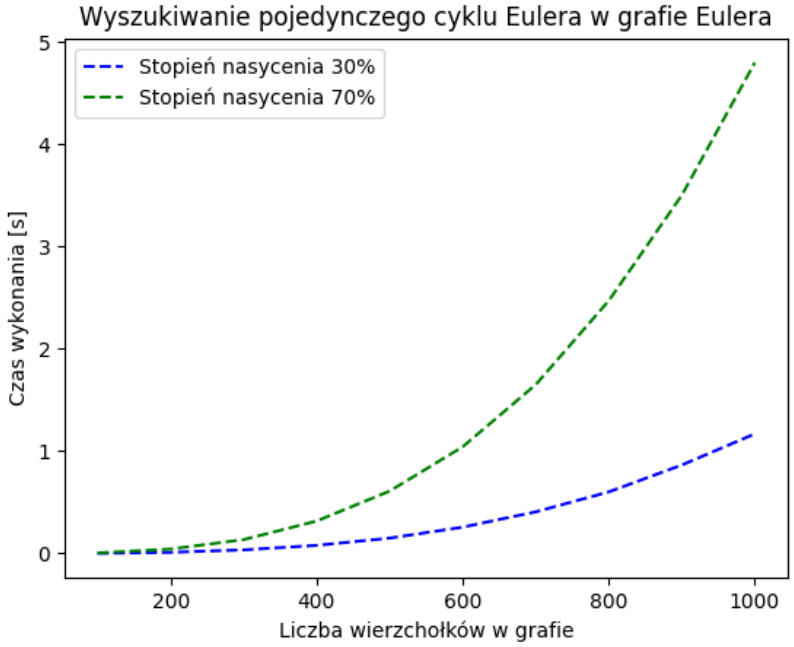

Wyszukiwanie wszystkich cyklów Hamiltona w grafie Hamiltona
Stopień nasycenia 30%
Stopień nasycenia 70%
Czas wykonania [s]
Liczba wierzchołków w grafie
0
50
100
150
200
250
300
350
7
8
9
10
11
12
13
14
15

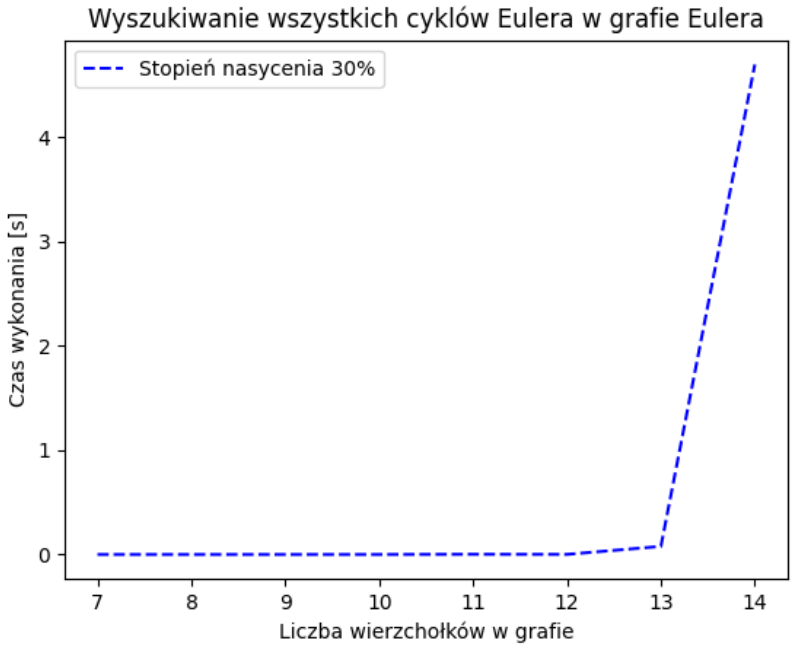

Wyszukiwanie wszystkich cyklów Hamiltona w grafie Hamiltona
Stopień nasycenia 30%
Czas wykonania [s]
Liczba wierzchołków w grafie
0.0
0.5
1.0
1.5
2.0
10
12
14
16
18


Wyszukiwanie pojedynczego cyklu Hamiltona w grafie Hamiltona
Stopień nasycenia 30%
Czas wykonania [s]
0
5
10
15
20
25
100
200
300
400
500
600
Liczba wierzchołków w grafie


Wyszukiwanie jakiegokolwiek cyklu Hamiltona w grafie niebędącym grafem Hamiltona
Stopień nasycenia 30%
Stopień nasycenia 70%
Czas wykonania [s]
0
5
10
15
20
25
30
10
11
12
13
14
15
Liczba wierzchołków w grafie


Wyszukiwanie jakiegokolwiek cyklu Hamiltona w grafie niebędącym grafem Hamiltona
Stopień nasycenia 30%
Czas wykonania [s]
0.0
0.5
1.0
1.5
2.0
10
12
14
16
18
Liczba wierzchołków w grafie


Wyszukiwanie pojedynczego cyklu Eulera w grafie Eulera
Stopień nasycenia 30%
Stopień nasycenia 70%
Czas wykonania [s]
0
1
2
3
4
5
200
400
600
800
1000
Liczba wierzchołków w grafie


Wyszukiwanie wszystkich cyklów Eulera w grafie Eulera
Stopień nasycenia 30%
Czas wykonania [s]
0
1
2
3
4
7
8
9
10
11
12
13
14
Liczba wierzchołków w grafie

# 4 Podsumowanie

Na końcu tego sprawozdania pozwoliłem sobie na pewne podsumowanie i wnioski, które nasunęły mi się podczas realizacji całego projektu i samego sprawozdania.

- Nie ma reprezentacji idealnej, za to może być reprezentacja zła – w niektórych zastosowaniach macierz sąsiedztwa się sprawdzi lepiej, w innych lista następników. Za to nie znalazłem dobrego zastosowania dla macierzy incydencji, poza tym o którym wspominałem wcześniej (rozróżnialność krawędzi).

- Algorytm sortowania topologicznego BFS, mimo swojej teoretycznej prostoty, okazał się bardzo powolny, wyraźnie wolniejszy niż sortowania topologicznego DFS. Powodem tego było wykorzystanie metody `getIndegree`, której złożoność jest duża dla każdej z implementacji reprezentacji grafu

- Prostota abstrakcyjnego algorytmu nie musi implikować niskiej jego złożoności – wszystko zależy jeszcze od złożoności wewnętrznych wywołań funkcji

- Obliczeniowo, trudniej jest znaleźć cykl Hamiltona, niż obwód Eulera. Z drugiej strony, zdecydowanie trudniej było mi wygenerować graf Eulera, niż graf Hamiltona. Odniosłem więc wrażenie, że pewne problemy chodzą parami i każde pojęcie ma swoją jaśniejszą i ciemniejszą stronę ;)

- Nie da się zbudować grafu Eulera dla każdej pary $(V, E)$, nawet jeśli $V <= E <= \frac{V(V-1)}{2}$. Z tego powodu, przygotowany przeze mnie algorytm generacji z nawrotami jest nieco mało użyteczny.

- Szablony w języku C++ to jest piękne i bardzo rozbudowane narzędzie, które znacznie uprościło mi pracę.

- Warto zadbać o testowanie projektu już na wczesnym jego etapie. Osobiście dodałem testy jednostkowe dopiero pod koniec pracy nad projektem, ale i tak pozwoliły mi one upewnić się, że wszystko działa jak należy i pomogły przy drobnej przebudowie projektu i dodaniu wsparcia dla obsługi błędów.

Mógłbym oczywiście udawać, że wszystko poszło po mojej myśli, ale uważam że nie o to chodzi. Część problemów i wad postarałem się wyeliminować, jednak niektóre nadal czekają na poprawę. Na pewną mało przyjemną dolegliwość cierpi losowy generator grafu (wykorzystywany np. podczas generacji grafów Hamiltona i Eulera). Mianowicie, w obecnej wersji działa on tak, że losuje trójkę wierzchołków, sprawdza czy taki trójkąt może zostać dodany do grafu, jeżeli tak – to dodaje, jeżeli nie – ponawia losowanie. Jeżeli w grafie jest stosunkowo mało krawędzi, nie stanowi to problemu. Jeżeli jednak graf jest mocno zapełniony, takie losowanie musi być często powtarzane. Uprzykrzyło mi to dość mocno życie podczas próby generowania grafów Hamiltona i Eulera o stopniu nasycenia 70% na potrzeby testów. Jest to więc jedna z rzeczy, którą należałoby poprawić.

Z kolei jestem wyjątkowo zadowolony z ogólnej struktury projektu, wydaje mi się ona być całkiem wygodna w obsłudze – na pewno pozwoliła bezproblemowo przygotować pokazowy program *example* oraz program testujący *benchmark*.

Na realizację całego projektu, wszystkich testów oraz tego sprawozdania poświęciłem bardzo dużo czasu. Liczę więc, że jest wykonane jak należy.

Dziękuję za uwagę!