**Uniwersytet Jagielloński w Krakowie**

Wydział Fizyki, Astronomii i Informatyki Stosowanej

**Albert Surmacz**

Nr albumu: 1160401

# Badanie grafów permutacji z językiem Python

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Informatyki Stosowanej

Kraków 2021

## Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

## Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

*Bardzo dziękuję Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za wszelką pomoc udzieloną w trakcie powstawania tej pracy.*

**Streszczenie**

W pracy zebrano informacje na temat grafów permutacji. Graf nieskierowany nazywamy grafem permutacji wtedy, gdy jego wierzchołki reprezentują elementy permutacji, a krawędzie reprezentują pary elementów odwróconych przez tę permutację. Interesujące jest to, że niektóre problemy trudne dla grafów ogólnych, dla grafów permutacji są rozwiązywalne w czasie wielomianowym. Dodatkowo rozpoznanie grafu permutacji można wykonać w czasie wielomianowym.

Przedstawiono implementacje w języku Python wybranych algorytmów dla grafów permutacji. Opisano właściwości grafów permutacji, które posłużyły do implementacji wydajnych algorytmów grafowych. Implementacje przedstawiono wraz z opisami przebiegu algorytmów i analizą złożoności obliczeniowej. Zaimplementowano dwa algorytmy rozpoznawania grafów permutacji. Przygotowano generatory wybranych typów grafów permutacji, grafy losowe i grafy o założonych właściwościach. Ciekawym algorytmem jest sprawdzanie spójności dla grafu permutacji, czego nie znaleziono w innych publikacjach, a złożoność obliczeniowa zależna liniowo od liczby wierzchołków jest lepsza niż dla grafu ogólnego. Przedstawiono również algorytmy wyznaczania dopełnienia, największej kliki, oraz dwa algorytmy wyznaczania największeo zbioru niezależnego. Wykorzystują one dwie odmienne techniki programowania. Pierwszą jest metoda dziel i zwyciężaj, a druga to programowanie dynamiczne. Zaimplementowano przeszukiwanie grafu w głąb i wszerz, gdzie na wejściu graf ma postać permutacji.

Została również wykonana galeria przykładowych grafów permutacji. Interesującym jest porównanie grafów permutacji z grafami kołowymi. Przygotowano też tabelę porównującą ilości grafów o danej wielkości należących do różnych klas grafów, do której wykorzystano między innymi dane uzyskane przy pomocy zaimplementowanych algorytmów.

Poprawność wszystkich implementacji została potwierdzona testami jednostkowymi. Złożoności czasowe przedstawionych algorytmów zostały potwierdzone doświadczalnie przy pomocy narzędzi dostępnych w języku Python, a wyniki zilustrowane na wykresach.

**Słowa kluczowe:** grafy permutacji, problem kliki, zbiory niezależne, spójność, dopełnienie grafu, przeszukiwanie grafu

**English title:** Study of permutation graphs with Python


## Abstract

Information on permutation graphs was collected in this paper. An undirected graph is a permutation graph if its vertices represent the permutation elements, and whose edges represent pairs of elements that are reversed by the permutation. The interesting thing is that several problems that are hard for general graphs may be solved in polynomial time for permutation graphs. Furthermore, permutation graphs can be recognized in polynomial time.

Python implementations of selected graph algorithms for permutation graphs are presented. This work describes the properties of permutation graphs that were used to implement efficient graph algorithms. Implementations with descriptions of algorithms are presented with time complexity analysis. Two permutation graph recognition algorithms were implemented. There are implementations of permutation graphs generators, both random and with detailed properties. An interesting algorithm is a connectivity check for permutation graphs that had not been found in other publications and its time complexity linear in vertices is more efficient than for general simple graphs. This study presents also implementations of algorithms for finding the complement graph, the largest clique, and two different algorithms for finding the largest independent sets. These implementations use two different programming techniques: the divide and conquer method, and the dynamic programming. There are two methods for graph traversal, a depth-first search and a breadth-first search, where the permutation representation was used as an input.

A gallery of sample permutation graphs is also presented and compared with circle graphs. A table comparing the number of graphs belonging to different classes of graphs is also prepared. Data on permutation graphs comes from tests performed with implemented algorithms.

The correctness of all presented implementations was confirmed by unit tests. The time complexities of the presented algorithms have been confirmed empirically using the tools available in Python, and the results are illustrated in plots.



**Keywords:** permutation graphs, clique problem, independent sets, connectivity, complement graph, graph traversal

# Spis treści

# Spis tabel

# Spis rysunków

# Listings

# 1. Wstęp

Tematem niniejszej pracy jest prezentacja grafów permutacji. Interesujące w nich jest to, że do zapisu informacji o wierzchołkach i węzłach takiego grafu wystarcza ciąg liczb. Przedmiotem naszego zainteresowania są również związane z nimi algorytmy, które wykorzystują tę własność. Wyjątkowe w nich jest to, że pewne problemy grafowe, które dla grafów ogólnych są NP-trudne, dla grafów permutacji posiadają rozwiązania w postaci algorytmów działających w czasie wielomianowym. Również rozpoznanie grafów permutacji można wykonać w czasie wielomianowym, co w sumie daje możliwość szybkiego rozwiązywania pewnych problemów dla instancji grafów, które są grafami permutacji. Ta przewaga sprawia, że są warte do rozważenia przy pracy z grafami, ale również znajdują swoje specjalne zastosowania między innymi w problemach obliczeń równoległych [5].

Celem pracy jest implementacja wybranych algorytmów grafów permutacji w języku Python. Jest to przede wszystkim algorytm rozpoznania grafu permutacji, oraz inne algorytmy użyteczne podczas pracy z grafami permutacji. Przykładowo generowanie pewnych typów grafów permutacji, sprawdzanie spójności grafu, wyznaczanie dopełnienia grafu, przeszukiwanie wszerz oraz przeszukiwanie w głąb. Są to również algorytmy związane z typowymi problemami w teorii grafów, jak wyznaczanie największego zbioru niezależnego, oraz wyznaczanie największej kliki.

Praca została podzielona na części stopniowo rozwijające tematykę grafów permutacji. Rozdział 1 jest wprowadzeniem do niniejszej pracy. Rozdział 2 podaje podstawowe definicje związane z problematyką teorii grafów, oraz wprowadza w tematykę grafów permutacji. Zawiera również galerię wszystkich spójnych grafów permutacji z liczbą wierzchołków $n$ mniejszą od 5. Rozdział 3 przedstawia implementację grafów, która została wykorzystana do projektowania przedstawianych w tej pracy algorytmów. Rozdział 4 to zbiór algorytmów wraz z ich implementacjami w języku Python, oraz ze szczegółowymi opisami zawierającymi szacowania złożoności obliczeniowych zaprezentowanych rozwiązań. Rozdział 5 to podsumowanie całej pracy, oraz porównanie reprezentacji grafu poprzez permutację z reprezentacją macierzy sąsiedztwa. W dodatku A znajdują się wykresy prezentujące wyniki testów wydajnościowych zaimplementowanych algorytmów.

# 2. Teoria grafów

Teoria grafów to nauka z zakresu matematyki i informatyki. Zagłębiając się w literaturę, spotykamy się z różnymi autorami, którzy stosują odmienną terminologię. W tym rozdziale, dla jednoznaczności podamy definicje podstawowych pojęć wykorzystywanych w niniejszej pracy. Wzorujemy się głównie na opisach z podręczników Cormena [2], Wilsona [3] i Golumbica[5].

## 2.1. Podstawowe definicje

**Definicja:** Graf prosty $G = (V, E)$ definiujemy jako uporządkowaną parę składającą się z niepustego skończonego zbioru wierzchołków $V$, oraz ze skończonego zbioru krawędzi $E$ (par wierzchołków) łączących te wierzchołki [3]. Na rysunkach wierzchołki, inaczej nazywane węzłami, oznaczamy jako kółka z zapisaną w środku unikalną etykietą identyfikującą wierzchołek, natomiast krawędzie jako proste odcinki pomiędzy nimi.

**Definicja:** Graf nieskierowany (ang. *undirected graph)* to graf prosty, w którym zbiór krawędzi $E$ jest zbiorem nieuporządkowanych par wierzchołków [2]. Oznacza to, że każda krawędź łączy dwa różne wierzchołki, a zbiory $\{u, v\}$ oraz $\{v, u\}$ gdzie $u, v \in V$ i $u \neq v$ oznaczają tę samą krawędź.

**Definicja:** Graf nieskierowany jest spójny (ang. *connected*), jeżeli pomiędzy każdą parą wierzchołków grafu istnieje połączenie (ciąg krawędzi). Graf niespójny składa się z pewnej liczby spójnych podgrafów, nazywanych *składowymi spójnymi* (ang. *connected components*)[5].

**Definicja:** Graf pełny (ang. *complete graph*) to graf prosty nieskierowany, w którym każda para wierzchołków połączona jest krawędzią. Graf oznaczamy symbolem $K_n$, gdzie $n$ to liczba wierzchołków [3].

**Definicja:** Graf cykliczny (ang. *cycle graph*) o $n$ wierzchołkach zapisywany jako $C_n$ to graf prosty nieskierowany spójny, w którym każdy wierzchołek jest połączony z dwoma innymi wierzchołkami [3].

**Definicja:** Graf liniowy $P_n$ (ang. *path graph*) o $n$ wierzchołkach to graf otrzymany z grafu $C_n$ poprzez usunięcie jednej krawędzi [3].

**Definicja:** Dopełnienie (ang. *complement, inverse*) grafu $G$ zapisywane jako $\overline{G}$ jest grafem z identycznym zbiorem wierzchołków co graf $G$, oraz zbiorem krawędzi składającym się z tych par wierzchołków, pomiędzy którymi nie występują krawędzie w grafie $G$ [5].

**Definicja:** Zbiór niezależny (ang. *independent set*) dla grafu $G = (V, E)$ jest to podzbiór wierzchołków $S \subset V$ z taką właściwością, że dla każdej pary wierzchołków $(u, v)$ z tego zbioru nie istnieje krawędź w grafie $G$ łącząca te wierzchołki [5].

**Definicja:** Klika (ang. *clique*) jest to podzbiór $C$ zbioru wierzchołków grafu nieskierowanego, gdzie pomiędzy każdą parą wierzchołków z $C$ istnieje krawędź. Klikę składającą się z $k$ wierzchołków nazywamy $k$-kliką. Mówimy o klice, że jest maksymalna (ang. *maximal*) w danym grafie, jeżeli nie możemy dodać do niej kolejnego wierzchołka tak, by utworzyć większą klikę [5].

**Definicja:** Grafy izomorficzne $G_1$, $G_2$ są to takie grafy, dla których istnieje jednoznaczna odpowiedniość pomiędzy węzłami tych grafów, taka, że liczba krawędzi łączących dane dwa wierzchołki grafu $G_1$ jest równa liczbie krawędzi łączących odpowiadające im wierzchołki grafu $G_2$ [3].

**Definicja:** Reprezentacja macierzy sąsiedztwa (ang. *adjacency matrix representation*) to jeden ze sposobów zapamiętywania grafu w pamięci komputera. Jeśli graf ma $n$ wierzchołków oznakowanych liczbami ze zbioru $\{0, 1, ..., n-1\}$, to macierzą sąsiedztwa reprezentującą ten graf jest macierz $A$ o wymiarze $n \times n$, której wyraz $A[i, j]$ jest równy liczbie krawędzi łączących wierzchołek $i$ z wierzchołkiem $j$ [3]. W ten sposób można zapisać grafy proste i multigrafy (na diagonali zapisywana jest liczba pętli przy wierzchołkach). Dla grafów nieskierowanych macierz sąsiedztwa jest symetryczna.

Używając innej konwencji, można za pomocą macierzy sąsiedztwa zapisać graf prosty ważony, gdzie niezerowe elementy macierzy $A$ oznaczają wagi odpowiednich krawędzi. Zero oznacza brak krawędzi, co może być problemem dla algorytmów wykorzystujących krawędzie o wadze zero.

## 2.2. Grafy permutacji

**Definicja:** Graf permutacji (ang. *permutation graph*) jest to graf nieskierowany, którego wierzchołki reprezentują elementy permutacji, a krawędzie reprezentują pary elementów tworzących inwersję w permutacji [12]. W niniejszej pracy rozważamy tylko te permutacje, które zgodnie z podaną definicją tworzą grafy spójne, ponieważ typowe problemy z teorii grafów można rozwiązywać niezależnie dla każdej składowej spójnej. Na rysunkach od 2.1 do 2.17 przedstawiono galerię wszystkich spójnych grafów permutacji z liczbą wierzchołków $n$ równą $2, 3, 4$.

Z różnych permutacji można uzyskać izomorficzne grafy. Przykładowo, graf zaprezentowany na rysunku 2.14 to graf na bazie permutacji $(2, 3, 1, 0)$, natomiast pozostałe izomorficzne do niego grafy permutacji to $(3, 1, 2, 0)$ 2.15 oraz $(3, 2, 0, 1)$ 2.16. Cecha ta nie występuje dla grafów pełnych, ponieważ zawsze istnieje tylko jedna reprezentacja i jest to permutacja, w której wszystkie elementy są posortowane malejąco. Przykłady grafów pełnych znajdują się na rysunkach 2.1 [permutacja $(1, 0)$], 2.4 [permutacja $(2, 1, 0)$] oraz 2.17 [permutacja $(3, 2, 1, 0)$].

W tabeli 2.1 zestawiono porównanie, które obrazuje liczbę grafów permutacji wśród grafów ogólnych. Znajdujemy tam potwierdzenie, że nie wszystkie grafy są możliwe do zapisania jako permutacja. Sama liczba permutacji, która dla $n$ elementów wynosi $n!$, jest mniejsza niż liczba grafów o tej samej liczbie wierzchołków. Tabela pozwala nam również zobrazować sobie rozmiary klasy grafów permutacji. Liczba spójnych grafów permutacji jest to liczba permutacji wyznaczających grafy spójne. Należy pamiętać, że są to dane dla grafów, w których rozróżniamy wierzchołki, więc są wśród nich grafy izomorficzne.

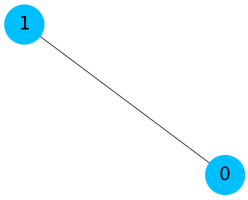
Tabela została utworzona na podstawie zbiorów z internetowej encyklopedii [11], za wyjątkiem danych o ilości spójnych grafów permutacji, które zostały wyznaczone przy pomocy zaimplementowanych funkcji.

Tabela 2.1. Porównanie liczby grafów etykietowanych o $n$ wierzchołkach z liczbą permutacji oraz grafów permutacji.

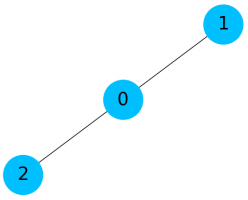
| $n$ | graf prosty | spójny graf prosty | permutacja | spójny graf permutacji |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 1 | 2 | 1 |
| 3 | 8 | 4 | 6 | 3 |
| 4 | 64 | 38 | 24 | 13 |
| 5 | 1024 | 728 | 120 | 71 |
| 6 | 32768 | 26704 | 720 | 461 |
| 7 | 2097152 | 1866256 | 5040 | 3447 |

**Definicja:** Graf kołowy (ang. *circle graph*) jest to graf nieskierowany, którego wierzchołki reprezentują cięciwy pewnego okręgu, a krawędzie łączą dwa wierzchołki, których odpowiednie cięciwy przecinają się [13].

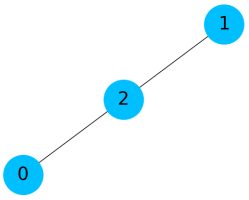
Grafy permutacji są szczególnym przypadkiem grafów kołowych. Właściwość, że istnieją grafy kołowe, które nie są grafami permutacji możemy łatwo zaobserwować przedstawiając interpretację graficzną grafów permutacji przedstawioną na rysunku 2.18. Grafy permutacji przedstawiamy jako ciąg liczb przedstawiających kolejne elementy permutacji. Zapisujemy je w jednym wierszu, a powyżej nich zapisujemy te same liczby lecz w posortowanej kolejności. Następnie pomiędzy takimi samymi elementami, które oznaczają wierzchołki grafu, rysujemy odcinki. Przecinające się odcinki wyznaczają krawędzie w tym grafie pomiędzy wierzchołkami, z których wychodzą te odcinki. Następnie zapisujemy te same elementy tym razem na okręgu, a narysowane proste wyznaczą cięciwy. Widzimy zatem, że każdy graf permutacji jest grafem kołowym. Jednak istnieją grafy kołowe, które nie są grafami permutacji. Zapisując graf permutacji na okręgu mamy to ograniczenie, że zapisujemy liczby w taki sposób, że zawsze można podzielić okrąg na dwie części i na obu połowach znajdą się te same liczby. Graf cykliczny $C_5$ jest najmniejszym grafem kołowym, który nie jest grafem permutacji. Na jego reprezentacji kołowej widzimy, że podział, taki jak w przypadku grafów permutacji nie jest możliwy. Graf kołowy znajdziemy na rysunku 2.19.



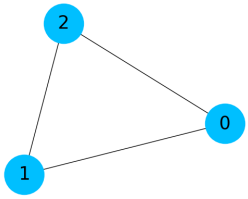

Rysunek 2.1. $(1, 0)$.



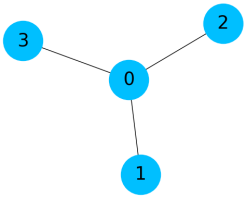

Rysunek 2.2. $(1, 2, 0)$.



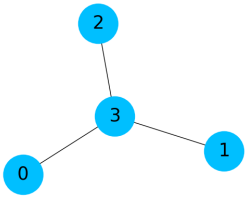

Rysunek 2.3. $(2, 0, 1)$.



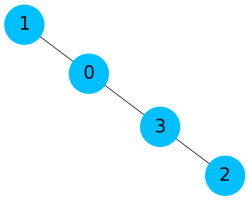

Rysunek 2.4. $(2, 1, 0)$.



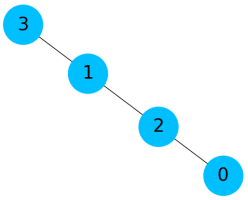

Rysunek 2.5. $(1, 2, 3, 0)$.



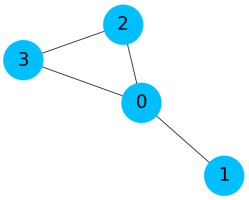

Rysunek 2.6. $(3, 0, 1, 2)$.



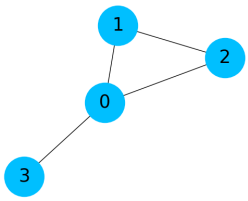

Rysunek 2.7. $(1, 3, 0, 2)$.



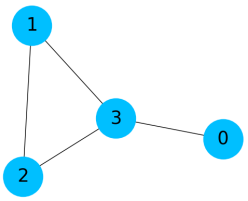

Rysunek 2.8. $(2, 0, 3, 1)$.



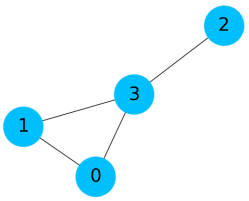

Rysunek 2.9. $(1, 3, 2, 0)$.



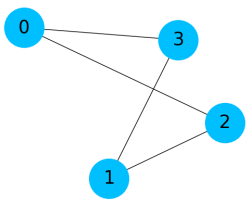

Rysunek 2.10. $(2, 1, 3, 0)$.



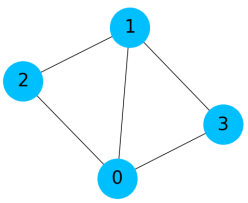

Rysunek 2.11. $(3, 0, 2, 1)$.



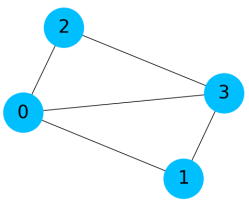

Rysunek 2.12. $(3, 1, 0, 2)$.



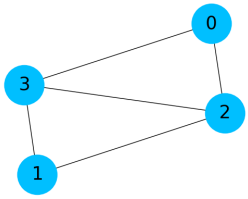

Rysunek 2.13. $(2, 3, 0, 1)$.



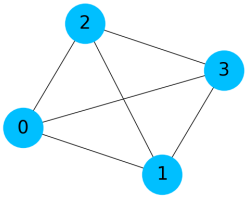

Rysunek 2.14. $(2, 3, 1, 0)$.



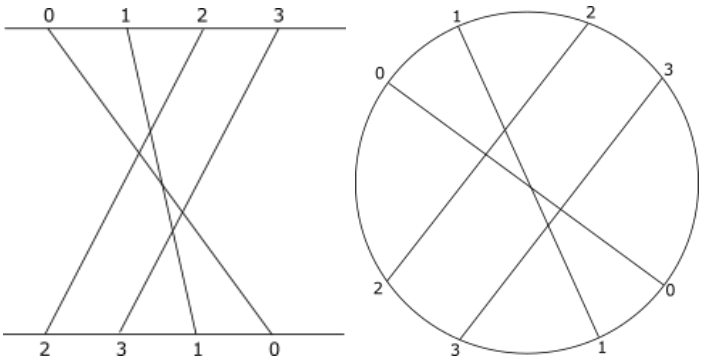

Rysunek 2.15. $(3, 1, 2, 0)$.



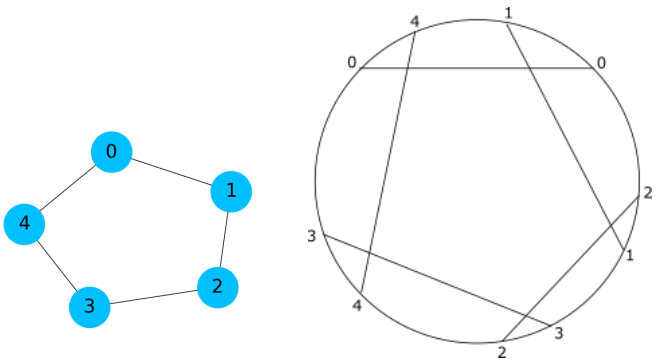

Rysunek 2.16. $(3, 2, 0, 1)$.




Rysunek 2.17. $(3, 2, 1, 0)$.




Rysunek 2.18. Graf permutacji $(2, 3, 1, 0)$ jako graf kołowy.




Rysunek 2.19. Graf cykliczny $C_5$ jako graf kołowy.

# 3. Implementacja grafów

Grafy permutacji będą reprezentowane jako lista Pythona z permutacją kolejnych liczb całkowitych od zera w górę. Narzędzia przygotowane w ramach tej pracy działają w Pythonie 2.7 i Pythonie 3.7+. Poniżej przedstawiamy przykładowe zastosowanie zaimplementowanych algorytmów.

Listing 3.1. Moduł usages.

```
#!/usr/bin/python

from graphs import Graph
from edges import Edge
from recognition import PermutationGraph
from is_connected_graph import is_connected_graph
from clique_and_iset import largest_clique, maximum_iset
from BFS import permBFS
from DFS import permDFS

# create a graph
N = 4  # number of nodes
G = Graph(N)
nodes = range(N)
edges = [Edge(2, 1), Edge(2, 0), Edge(1, 0), Edge(1, 3), Edge(0, 3)]
for node in nodes:
    G.add_node(node)
for edge in edges:
    G.add_edge(edge)

perm_graph_recog = PermutationGraph(G)

if perm_graph_recog.run():
    # find representation
    permutation = perm_graph_recog.get_perm_repr()

    is_connected = is_connected_graph(permutation)
    clique = largest_clique(permutation)
    iset = maximum_iset(permutation)

    starting_node = 0
    BFS = permBFS(permutation)
    order_BFS = BFS.BFS_order(starting_node)

    DFS = permDFS(permutation)
    order_DFS = DFS.DFS_order(starting_node)

    print("permutation = {}".format(permutation))
    print("is_connected = {}".format(is_connected))
    print("largest clique = {}".format(clique))
    print("maximum iset {}".format(iset))
```

```
    print("BFS order = {}".format(order_BFS))
    print("DFS order = {}".format(order_DFS))
else:
    print("not a permutation graph")
```

# 4. Algorytmy

Przedstawimy tutaj wybrane problemy z teorii grafów, które są interesujące ze względu na możliwość ich rozwiązywania w odmienny sposób dzięki wykorzystaniu reprezentacji grafu w postaci permutacji. Omówione zostaną przy tym złożoności obliczeniowe takich rozwiązań.

## 4.1. Rozpoznawanie grafów permutacji - algorytm siłowy

Aby wykorzystać potencjał grafów permutacji najistotniejszym algorytmem jest rozpoznanie, czy dany graf spełnia warunki ich definicji, oraz czy potrafimy wyznaczyć odpowiednią permutację. Mając tę pewność i wyznaczoną permutację, możemy stosować specjalne algorytmy dla tej rodziny grafów, które mogą okazać się wydajniejsze niż algorytmy ogólne. Przedstawimy dwa sposoby rozwiązania tego problemu. Pierwszym z nich jest algorytm siłowy (ang. *brute force*) sprawdzający wszystkie możliwe sposoby przyporządkowania liczb do wierzchołków grafu.

**Dane wejściowe:** Dowolny graf nieskierowany.

**Problem:** Rozpoznanie czy jest to graf permutacji.

**Opis algorytmu:** Algorytm rozpoczynamy od przygotowania wszystkich możliwych permutacji o liczbie elementów równej liczbie węzłów badanego grafu. Następnie porównujemy krawędzie badanego grafu z interpretacją badanych permutacji. Sprawdzamy każde możliwe odwzorowanie etykiet na wierzchołki badanego grafu. Jeśli znajdziemy różnicę, przechodzimy do kolejnego odwzorowania. Jeśli okaże się, że badaliśmy graf permutacji, skończymy w momencie znalezienia permutacji i odwzorowania, które wyznaczą takie same krawędzie jak w danym grafie. Natomiast jeśli nim nie jest, algorytm zakończy się dopiero po zweryfikowaniu każdej możliwej permutacji.

**Złożoność:** Algorytm jest mało wydajny, ponieważ za każdym razem, gdy badamy graf o $n$ wierzchołkach, który nie jest grafem permutacji, sprawdza wszystkie możliwe permutacje, których jest $n!$, oraz wszystkie odwzorowania tych permutacji na wierzchołki grafu, których jest również $n!$. Zatem złożoność czasowa wynosi $O(n! \cdot n!)$.

**Uwagi:** Duża złożoność obliczeniowa tego algorytmu powoduje, że stosowanie go w celu możliwości późniejszego stosowania wydajnych algorytmów nie jest opłacalne.

Listing 4.1. Moduł is_permutation_graph.

```
import itertools
from graphs import Graph
from edges import Edge
from is_edge import is_edge

def is_correct_perm(graph, permutation):
    ''' Czy dany graf mozna zapisac dana permutacja. '''
    size = len(permutation)
    for keys in itertools.permutations(range(size)):
        is_correct = True
        zip_iterator = zip(keys, permutation)
        mapping_dict = dict(zip_iterator)
        for a in range(size):
            for b in range(a + 1, size):
                if is_edge(permutation, a, b) != graph.has_edge(
                        Edge(mapping_dict[a], mapping_dict[b])):
                    is_correct = False
        if is_correct:
            return keys
    return []

def is_permutation_graph(graph):
    ''' Czy graf mozna zapisac jako permutacje. '''
    for permutation in itertools.permutations(range(graph.v())):
        key = is_correct_perm(graph, permutation)
        if key:
            return permutation, key
    return []
```

## 4.2. Rozpoznawanie grafów permutacji - algorytm wielomianowy

Grafy permutacji nie byłyby tak atrakcyjne, gdyby nie istniał algorytm rozpoznawania ich w czasie wielomianowym. Na szczęście taki algorytm został podany przez autorów takich jak Pnueli, Lempel i Eeven [6]. Działanie opisanego algorytmu opiera się na twierdzeniu, które mówi, że graf G jest grafem permutacji wtedy i tylko wtedy, gdy $G$ i $\bar{G}$ są grafami porównywalności (ang. *comparability graph*) [6].

**Dane wejściowe:** Dowolny graf nieskierowany.

**Problem:** Rozpoznanie czy jest to graf permutacji.

**Opis algorytmu:** Algorytm rozpoczynamy od próby wyznaczenia grafu porównywalności. W tym celu wybieramy dowolną krawędź i nadajemy jej dowolny kierunek. Następnie sprawdzamy, czy utworzony w ten sposób graf jest stabilny, co oznacza, że nie ma w nim krawędzi w relacji $\Gamma$ zdefiniowanej następująco. Jeżeli $i \neq k$ oraz $(i,j)$, $(j,k) \in E$, to $(i,j)\Gamma(j,k)$ wtedy i tylko wtedy, gdy $(i,k) \notin E$. Oznacza to konieczność sprawdzania kolejnych

krawędzi grafu w celu identyfikacji tej relacji. W przypadku znalezienia takiej krawędzi nadajemy jej odpowiedni kierunek i powtarzamy sprawdzanie krawędzi.

Gdy już sprawdzimy wszystkie krawędzie, wykonujemy test na podgrafie utworzonym z krawędzi skierowanych badanego grafu. Test polega na sprawdzaniu zależności pomiędzy zbiorami wierzchołków do których możemy się dostać z danego wierzchołka. Niech $V(i)$ oznacza zbiór wierzchołków do którego wychodzą krawędzie skierowane rozpoczynające się w wierzchołku $i$. Aby spełnić warunki testu, dla każdego wierzchołka musi zachodzić zależność $V(i) \supset W(i) = \bigcup_{j \in V(i)} V(j)$. Jeśli graf nie spełnia warunków tego testu, kończymy algorytm z pewnością, że badany graf nie jest grafem permutacji. W przeciwnym przypadku kontynuujemy algorytm powtarzając opisane wcześniej kroki dla pozostałych krawędzi nieskierowanych tego grafu. Jeśli dojdziemy do momentu, w którym już takich nie ma, czyli graf jest grafem porównywalności, należy powtórzyć cały algorytm dla dopełnienia grafu. Jeśli również dla dopełnienia grafu przebrniemy przez cały algorytm, oznacza to, że badany graf jest grafem permutacji.

**Złożoność:** Złożoność przedstawionego algorytmu jest wielomianowa. Algorytm przetwarza każdą krawędź grafu, jednak liczba krawędzi badanego grafu nie ma bezpośredniego znaczenia, ponieważ i tak oprócz grafu sprawdzamy również jego dopełnienie. Zatem złożoność obliczeniowa rośnie proporcjonalnie do liczby wszystkich możliwych krawędzi w grafie o danej liczbie węzłów. Testy potwierdzające wielomianową złożoność znajdują się w dodatku A.

**Uwagi:** Do wyznaczenia permutacji reprezentującej dany graf permutacji wykorzystujemy utworzone w tym algorytmie zbiory wierzchołków $V(i)$.

## 4.3. Generowanie przypadkowych grafów permutacji

Do badań nad grafami permutacji wykorzystujemy funkcję, która pozwala generować losowe grafy permutacji. Implementacja funkcji jest przedstawiona na listingu 4.2.

**Dane wejściowe:** Liczba wierzchołków grafu permutacji.

**Problem:** Generowanie losowej permutacji.

**Opis algorytmu:** Algorytm rozpoczynamy od przygotowania listy o rozmiarze równym parametrowi *size* zawierającej kolejne liczby począwszy od 0. Następnie korzystając z metody random.shuffle(), która implementuje algorytm Fishera i Yatesa, mieszamy kolejność elementów listy. Otrzymaną listę zwracamy jako wynik.

**Złożoność:** Złożoność jest zależna od zastosowanego algorytmu mieszania liczb w permutacji. W przypadku algorytmu Fishera i Yatesa złożoność czasowa wynosi $O(n)$.

**Uwagi:** Jako wynik możliwa jest do uzyskania każda permutacja. Oznacza to, że grafy mogą być spójne lub niespójne, a także kilka permutacji może reprezentować izomorficzne grafy.

Listing 4.2. Moduł generation_of_permutations, graf przypadkowy.

```
import random

def generate_random_permutation(size):
    ''' Generacja losowej permutacji. '''
    permutation = list(range(size))
    random.shuffle(permutation)
    return permutation
```

## 4.4. Generowanie wybranych grafów permutacji

Do badań nad grafami permutacji stosujemy nie tylko generację losowych grafów, ale również generowanie grafów permutacji o założonych właściwościach. Przygotowaliśmy funkcje pozwalające na generowanie grafów dwudzielnych: graf liniowy $P_n$, graf gwiazda $K_{1,p}$, graf dwudzielny pełny $K_{p,q}$. Głównie interesujące jest wyznaczanie grafu liniowego, którego implementacja została przedstawiona na listingu 4.3. Algorytm polega na powiększaniu listy wyznaczającej permutację o kolejne liczby przy zachowaniu spójności grafu.

Listing 4.3. Moduł generation_of_permutations, graf liniowy.

```
def swap(L, i, j):
    L[i], L[j] = L[j], L[i]

def make_path_perm(n):
    if n < 1:
    raise ValueError("no nodes")
    elif n == 1:
        return [0]
    elif n == 2:
        return [1, 0]
    else:
        perm = make_path_perm(n - 2)
        perm.append(n - 2)
        perm.append(n - 1)
        swap(perm, -2, -1)
        swap(perm, -3, -2)
        return perm
```

## 4.5. Testowanie spójności grafu permutacji

Wykorzystując funkcję generowania losowych permutacji nie mamy pewności, że otrzymana permutacja będzie odpowiadała grafowi spójnemu. W naszych badaniach szczególnie interesują nas grafy spójne, dlatego potrzebna jest funkcja testująca spójność grafu.

**Dane wejściowe:** Dowolny graf permutacji.

**Problem:** Badanie spójności grafu.

**Opis algorytmu:** Algorytm polega na sprawdzaniu, czy kolejne podciągi utworzone z początkowych liczb permutacji zawierają element o wartości większej od rozmiaru tego podciągu. Jeśli znajdziemy taki podciąg, który nie ma takiej liczby, oznacza to, że te węzły nie posiadają żadnej krawędzi do węzłów poza tym podciągiem, czyli graf nie jest spójny.

**Złożoność:** Złożoność obliczeniowa algorytmu jest liniowa $O(n)$, ponieważ mamy jedno przejście wzdłuż permutacji i dwa testy dla każdej pozycji. Złożoność została potwierdzona w testach, których wyniki znajdziemy w dodatku A.

**Uwagi:** Uzyskiwana złożoność obliczeniowa jest korzystniejsza niż złożoność algorytmów dla grafu ogólnego (wtedy korzysta się z BFS lub DFS).

Listing 4.4. Testowanie spójności grafu permutacji.

```
def is_connected_graph(permutation):
    '''Test czy permutacja reprezentuje graf spojny.'''
    size = len(permutation)
    maxi = 0
    for i in range(size):
        if permutation[i] > maxi:
            maxi = permutation[i]
        if i != size - 1 and maxi == i:
            return False
    return True
```

## 4.6. Wyznaczanie dopełnienia grafu permutacji

Dopełnieniem grafu $G$ jest graf $\bar{G}$, który ma krawędzie tylko pomiędzy parami tych wierzchołków, pomiędzy którymi nie istnieje krawędź w grafie $G$. W języku permutacji oznacza to, że liczby tworzące inwersję w grafie $G$, nie będą tworzyły inwersji w grafie $\bar{G}$ i na odwrót, liczby tworzące inwersję w grafie $\bar{G}$, nie będą tworzyły inwersji w gafie $G$. Stąd widać, że wystarczy odwrócić kolejność liczb w permutacji związanej z $G$, aby otrzymać permutację związaną z $\bar{G}$.

Funkcja wyznaczająca dopełnienie grafu permutacji jest przedstawiona na listingu 4.5. Złożoność obliczeniowa funkcji jest rzędu $O(n)$.

Listing 4.5. Wyznaczanie dopełnienia grafu permutacji.

```
def complement_graph(permutation):
    '''Wyznaczanie dopelnienia grafu permutacji.'''
    return permutation[::-1]
```

## 4.7. Wyznaczanie największego zbioru niezależnego - dziel i zwyciężaj

Wyznaczanie największego zbioru niezależnego dla grafów permutacji sprowadza się do znalezienia najdłuższego podciągu rosnącego (ang. *Longest Increasing Subsequence, LIS*) w permutacji reprezentującej graf, co zostało zaprezentowane na listingu 4.8.

Podamy dwa rozwiązania tego problemu. Pierwsze z nich stosuje algorytm wyszukiwania binarnego (ang. *binary search*), opierającego się na metodzie dziel i zwyciężaj (ang. *divide and conquer*). Drugie rozwiązanie stosuje technikę programowania dynamicznego (ang. *dynamic programming*), w której do obliczenia pewnej wartości używamy wartości obliczonych już wcześniej. Odmienne sposoby uzyskują różne złożoności obliczeniowe.

**Dane wejściowe:** Dowolny graf permutacji.

**Problem:** Wyznaczanie najdłuższego rosnącego podciągu przy wykorzystaniu wyszukiwania binarnego.

**Opis algorytmu:** Algorytm polega na przechodzeniu po kolejnych elementach permutacji i przy użyciu wyszukiwania binarnego wyznaczania takiej liczby *mid*, żeby dany element mógł przedłużyć podciąg o długości *mid*.

**Złożoność:** Za $n$ przyjmujemy rozmiar badanej permutacji. Złożoność czasowa wyszukiwania binarnego wynosi $O(\log n)$. W zaprezentowanym algorytmie wykonujemy go na każdym elemencie, zatem w sumie złożoność czasowa jest równa $O(n \log n)$. Wyniki testów znajdują się w dodatku A.

**Uwagi:** Zwracana lista zawiera liczby w rosnącym porządku ze względu na zastosowaną metodę. W ogólnym przypadku kolejność elementów największego zbioru niezależnego nie ma znaczenia.

Listing 4.6. Najdłuższy rosnący podciąg (wyszukiwanie binarne).

```
def longest_increasing_subsequence(permutation):
    '''LIS wersja z wyszukiwaniem binarnym.'''
    size = len(permutation)
    P = [0 for i in range(size)]
    M = [1 for i in range(size + 1)]
    L = 0
    for i in range(size):
        lo = 1
        hi = L
        while lo <= hi:
            mid = int(ceil((lo + hi) / 2.0))    # Py2 vs Py3
            if permutation[M[mid]] < permutation[i]:
                lo = mid + 1
            else:
                hi = mid - 1
        newL = lo
        P[i] = M[newL - 1]
        M[newL] = i
```

```
        if newL > L:
            L = newL
    S = [0 for i in range(L)]
    k = M[L]
    for i in range(L - 1, -1, -1):
        S[i] = permutation[k]
        k = P[k]
    return S
```

## 4.8. Wyznaczanie największego zbioru niezależnego - programowanie dynamiczne

**Dane wejściowe:** Dowolny graf permutacji.

**Problem:** Wyznaczanie najdłuższego rosnącego podciągu przy wykorzystaniu programowania dynamicznego.

**Opis algorytmu:** Działanie algorytmu polega na przechodzeniu permutacji wykorzystując dwa indeksy, początku i końca badanego w danej chwili fragmentu oraz zapisywaniu w pomocniczych listach poprzedników danego elementu dla dotychczasowo znalezionych rosnących podciągów i ich rozmiarów. Gdy zbadamy już całą permutację wybierany jest najdłuższy z nich, który zostaje przepisany do listy zwracanej w wyniku.

**Złożoność:** Złożoność czasowa algorytmu wynosi $O(n^2)$, gdzie $n$ to rozmiar permutacji. Wynika to z tego, że dla każdego elementu musimy sprawdzić wszystkie poprzednie elementy.

**Uwagi:** Nie jest to najwydajniejsze rozwiązanie, ale interesujące ze względu na możliwość zastosowania techniki programowania dynamicznego.

Listing 4.7. Najdłuższy rosnący podciąg (programowanie dynamiczne).

```
def longest_increasing_subsequence_dynamic_programming(permutation):
    '''LIS wersja z programowaniem dynamicznym.'''
    n = len(permutation)
    lis = [1] * n
    prev = [None] * n
    for i in range(1, n):
        for j in range(0, i):
            if permutation[i] > permutation[j] and lis[i] < lis[j] + 1:
            lis[i] = lis[j] + 1
            prev[i] = j
    maxId = lis.index(max(lis))

    i = maxId
    result = [permutation[maxId]]
    while prev[i] is not None:
        i = prev[i]
        result.append(permutation[i])
    result.reverse()
    return result
```

Listing 4.8. Wyznaczanie największego zbioru niezależnego.

```
def maximum_iset(permutation):
    '''Wyznaczanie najwiekszego zbioru niezaleznego w grafie permutacji.'''
    return longest_increasing_subsequence(permutation)
```

## 4.9. Wyznaczanie największej kliki w grafie permutacji

Wyznaczanie największej kliki dla grafu permutacji sprowadza się do znalezienia największego malejącego podciągu (ang. *Longest Decreasing Subsequence, LDS*) w permutacji reprezentującej graf. Jest to równoważne ze znalezieniem LIS dla dopełnienia tego grafu. Zatem do wyznaczenia LDS stosujemy przedstawione wcześniej funkcje wyznaczania dopełnienia grafu oraz do znajdywania LIS.

Implementacja jest przedstawiona na listingach 4.9 i 4.10.

**Dane wejściowe:** Dowolny graf permutacji.

**Problem:** Wyznaczenie największego malejącego podciągu w permutacji.

**Opis algorytmu:** Algorytm rozpoczynamy od wyznaczenia dopełnienia grafu. Następnie znajdujemy LIS dla tego dopełnienia oraz odwracamy jego kolejność. Otrzymaną w ten sposób listę zwracamy jako wynik.

**Złożoność:** Złożoność czasowa algorytmu zależy od zastosowanego algorytmu wyszukiwania LIS. W zaprezentowanym przypadku złożoność wynosi $O(n \log n)$. Testy zostały umieszczone w dodatku A.

**Uwagi:** Problem ten można rozwiązać bez korzystania z właściwości dopełnienia grafu implementując funkcję analogiczną do wyznaczania LIS. Należy również pamiętać, że kolejność elementów największej kliki nie ma znaczenia, a otrzymywana lista w porządku malejącym wynika ze sposobu prezentowanej implementacji.

Listing 4.9. Wyznaczanie największego malejącego podciągu.

```
def longest_decreasing_subsequence(permutation):
    '''Wyznaczanie LDS przy wykorzystaniu LIS i dopelnienia grafu.'''
    return longest_increasing_subsequence(
        complement_graph(permutation))[::-1]
```

Listing 4.10. Wyznaczanie największej kliki.

```
def largest_clique(permutation):
    '''Wyznaczanie najwiekszej kliki w grafie permutacji.'''
    return longest_decreasing_subsequence(permutation)
```

## 4.10. Sprawdzanie istnienia krawędzi w grafie permutacji

Aby sprawdzić, czy graf zawiera krawędź pomiędzy dwoma wybranymi węzłami, wczytujemy permutację oraz węzły, które będziemy sprawdzać. W implementacji wykorzystujemy niejawnie permutację odwrotną, ponieważ znajdujemy pozycje węzłów w permutacji. Na końcu wystarczy sprawdzić, czy dwa węzły tworzą inwersję. Ważne jest, aby zastosować poprawną kolejność argumentów funkcji, bo to wpływa na wykrywanie inwersji.

Implementacja jest przedstawiona na listingu 4.11. Czas pracy funkcji jest liniowy $O(n)$, ponieważ mamy pojedyncze przejście wzdłuż permutacji.

Listing 4.11. Sprawdzanie istnienia krawędzi w grafie permutacji.

```
def is_edge(permutation, a, b):
    '''Test istnienia krawedzi w grafie permutacji.'''
    if a > b:
        a, b = b, a
    for k, item in enumerate(permutation): # O(n) time, O(1) memory
        if item == a:
            left = k
        if item == b:
            right = k
    return left > right
```

## 4.11. Przeszukiwanie wszerz grafu permutacji

Przeszukiwanie wszerz (ang. *breadth-first search*) to dobrze znana metoda przechodzenia przez graf. Jednak jej implementacja różni się w zależności od zastosowanej reprezentacji grafu.

**Dane wejściowe:** Dowolny graf oraz węzeł, od którego zaczniemy przeszukiwanie.

**Problem:** Przeszukiwanie grafu wszerz.

**Opis algorytmu:** Algorytm rozpoczynamy od stworzenia listy, której elementy odpowiadają za przechowywanie informacji o tym, czy dany wierzchołek został już odwiedzony. Początkowo lista ma same wartości False. Tworzymy też kolejkę, która służy do przechowywania kolejnych kandydatów na węzły do odwiedzenia. Dodajemy do niej startowy węzeł i oznaczamy go jako odwiedzony. Następnie wyciągamy wierzchołek z początku kolejki oraz dodajemy wierzchołki, które można bezpośrednio z niego odwiedzić. Aby to zrobić przeszukujemy całą permutację wyznaczającą badany graf. Szukamy nieodwiedzonych sąsiadów danego węzła. Wykorzystujemy do tego przedstawioną wcześniej funkcję do sprawdzania istnienia krawędzi. Za każdym razem, gdy zostaną spełnione warunki, oznaczamy znaleziony węzeł jako odwiedzony. W dalszym ciągu algorytm powtarza te czynności do momentu, gdy kolejka będzie już pusta.

**Złożoność:** Złożoność czasowa algorytmu wynosi $O(n^2)$, gdzie $n$ oznacza rozmiar permutacji. Powodem kwadratowej złożoności jest konieczność sprawdzania każdego elementu permutacji tyle razy ile elementów w sumie znajdzie się w kolejce, a dla grafu spójnego liczba ta wyniesie dokładnie $n$. Testy znajdują się w dodatku A.

**Uwagi:** Algorytm działa dla dowolnej permutacji.

Listing 4.12. Moduł BFS.

```
import collections

class permBFS():

    def __init__(self, perm):
        self.size = len(perm)
        self.perm = perm
        self.position = list(perm)  # temporary, O(n) memory
        for k, item in enumerate(perm):  # O(n) time
            self.position[item] = k  # for testing edges

    def has_edge(self, i, j):
        if i > j:
            i, j = j, i
        return self.position[i] > self.position[j]

    def BFS_order(self, node):
        order = []
        visited = [False] * self.size
        queue = collections.deque()
        queue.appendleft(node)
        visited[node] = True

        while queue:
            source = queue.pop()
            order.append(source)
            for target in self.perm:
                if not visited[target]:
                    if self.has_edge(source, target):
                        queue.appendleft(target)
                        visited[target] = True
        return order
```

## 4.12. Przeszukiwanie w głąb grafu permutacji

Przeszukiwanie w głąb (ang. *depth-first search*) to druga dobrze znana metoda przechodzenia przez graf. W tym przypadku implementacja również różni się w zależności od zastosowanej reprezentacji grafu. Listing 4.13 przedstawia implementację algorytmu rekurencyjnego. Rekurencja to technika, w której w celu rozwiązania danego problemu algorytm wywołuje samego siebie do rozwiązywania mniejszych podproblemów [2].

**Dane wejściowe:** Dowolny graf oraz węzeł, od którego zaczniemy przeszukiwanie.

**Problem:** Przeszukiwanie grafu w głąb.

**Opis algorytmu:** Algorytm polega na przeszukiwaniu nieodwiedzonych sąsiadów danego wierzchołka, następnie oznaczaniu ich jako odwiedzone oraz wywoływaniu samego siebie. Wywołanie to następuje dla tej samej permutacji, ale rozpoczyna się od znalezionego uprzednio sąsiada. Jeśli dany wierzchołek nie ma już takich sąsiadów następuje nawrót (ang. *backtracking*), co oznacza, że algorytm cofa się do poprzedniego wierzchołka. Po powrocie algorytm kontynuuje poszukiwanie sąsiadów do odwiedzenia. Algorytm kończy się w chwili powrotu do wierzchołka startowego, kiedy nie będzie już nieodwiedzonych sąsiadów tego wierzchołka.

**Złożoność:** Złożoność czasowa algorytmu wynosi $O(n^2)$, gdzie $n$ oznacza rozmiar permutacji. Zostało to potwierdzone w dodatku A.

**Uwagi:** Algorytm rekurencyjny dla problemu z dużym grafem, w którym będziemy mieli zbyt dużo wywołań rekurencyjnych, może spowodować błąd pamięci. Błąd może wystąpić na poziomie interpretera Pythona (wyjątek RuntimeError), a jeżeli przesuniemy ograniczenie, to błąd wystąpi na poziomie jądra systemu operacyjnego.

Listing 4.13. Moduł DFS.

```
class permDFS():

    def __init__(self, perm):
        self.size = len(perm)
        self.perm = perm
        self.visited = set()
        self.position = list(perm)
        for k, item in enumerate(perm):
            self.position[item] = k

    def has_edge(self, i, j):
        if i > j:
            i, j = j, i
        return self.position[i] > self.position[j]

    def visit(self, node, order=None):
        self.visited.add(node)
        if order is None:
            order = [node]
        for target in self.perm:
            if self.has_edge(node, target):
                if target not in self.visited:
                    order.append(target)
                    self.visit(target, order)
        return order

    def DFS_order(self, node, order=None):
        order = self.visit(node, order)
        self.visited.clear()
        return order
```

# 5. Podsumowanie

Celem niniejszej pracy było przedstawienie grafów permutacji i zbadanie ich właściwości. Przytoczono odpowiednie definicje z ogólnej teorii grafów, jak i zagadnienia dotyczące samych grafów permutacji. W pracy znajduje się również porównanie grafów permutacji z grafami kołowymi, oraz galeria przykładowych grafów permutacji z liczbą wierzchołków co najwyżej pięć.

Przedstawione zostały również implementacje szybkich rozwiązań dla problemów, rozwiązywalnych na grafach permutacji, które są trudniej wykonywalne dla grafu ogólnego. Wykonano je w języku Python. Dla potwierdzenia opisanych złożoności wykonano testy sprawdzające wydajność tych algorytmów. Wyniki testów zostały przedstawione na wykresach.

Warto zauważyć podobieństwa i różnice między reprezentacją grafu jako permutacji, a reprezentacją macierzy sąsiedztwa. W tabeli 5 porównano te reprezentacje. Pamięć wykorzystana do przechowywania grafu jest mniejsza dla grafu permutacji, ale na przykład macierz ma przewagę w szybkości sprawdzania istnienia danej krawędzi. Algorytmy przeszukiwania grafu mimo odmiennych implementacji uzyskują podobną złożoność obliczeniową. Złożoności czasowe algorytmów takich jak wyznaczanie największej kliki, wyznaczanie największego zbioru niezależnego, badanie spójności grafu, czy wyznaczanie dopełnienia grafu, można wykonać szybciej implementując graf jako permutację niż jako macierz sąsiedztwa. Należy jednak pamiętać, że reprezentacja przez permutację, w przeciwieństwie do reprezentacji macierzy sąsiedztwa, nie umożliwia przedstawienia każdego grafu.

Na koniec warto podkreślić oszczędność pamięci oferowaną przez reprezentację przez permutację, oraz możliwość szybkiego rozwiązywania problemów trudnych dla ogólnych grafów. Zadziwiająca jest złożoność $O(n)$ algorytmu sprawdzającego spójność grafu. Algorytm w pełni wykorzystuje zakodowaną w inwersjach informację o krawędziach grafu. Dość nieoczekiwanie nie znaleźliśmy tego algorytmu w literaturze.

Tabela 5.1. Porównanie reprezentacji przez permutację i reprezentacji macierzy sąsiedztwa. Liczba $n$ oznacza liczbę wierzchołków grafu, a jednocześnie długość permutacji, liczbę wierszy i liczbę kolumn macierzy sąsiedztwa.

| Nazwa problemu | Permutacja | Macierz sąsiedztwa |
|---|---|---|
| zajętość pamięci | $O(n)$ | $O(n^2)$ |
| sprawdzanie istnienia krawędzi | $O(n)$ | $O(1)$ |
| wyznaczanie dopełnienia grafu | $O(n)$ | $O(n^2)$ |
| przeszukiwanie wszerz | $O(n^2)$ | $O(n^2)$ |
| przeszukiwanie w głąb | $O(n^2)$ | $O(n^2)$ |
| wyznaczanie największej kliki | $O(n \log n)$ | $O(n^2)$ |
| wyznaczanie największego zbioru niezależnego | $O(n \log n)$ | $O(n^2)$ |
| badanie spójności grafu | $O(n)$ | $O(n^2)$ |

# A. Testy algorytmów

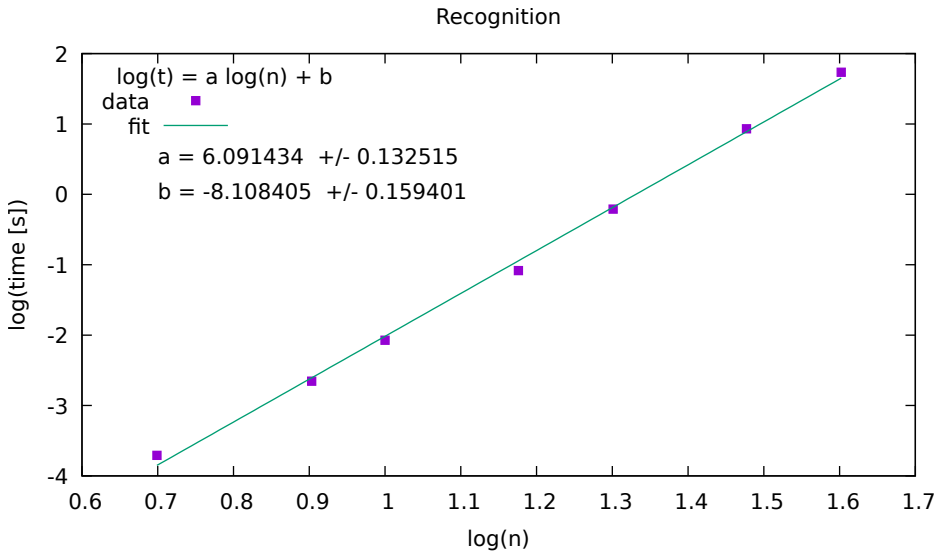
Poprawność zaimplementowanych algorytmów została przetestowana za pomocą testów jednostkowych przy użyciu modułu unittest [9].

Przeprowadzone zostały również testy wydajności. Wykonano je za pomocą modułu timeit [10], który wykorzystany został do dokonania pomiarów pojedynczych wywołań badanych funkcji. Testy przeprowadzano dla każdych badanych danych po trzy razy, a przedstawione w niniejszym rozdziale wykresy prezentują najszybsze z nich.

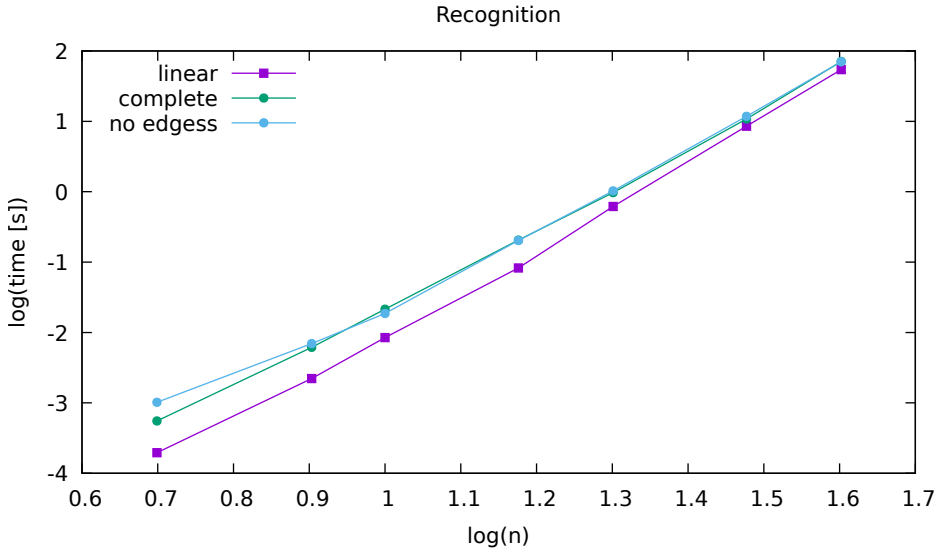
## A.1. Testy rozpoznawania grafów permutacji

W celu przetestowania wydajności zmierzono czasy wykonania algorytmu. Badano wywołania dla grafów pełnych o coraz większych liczbach wierzchołków. Uzyskane wyniki znajdują się na wykresie A.1. Współczynnik dopasowanej krzywej $a = 6.09(14)$ świadczy o wielomianowym czasie badanego algorytmu.

Wykres A.2 przedstawia porównanie wydajności dla odmiennych klas grafów. Możemy na nim zauważyć, że dla grafu pełnego i grafu bez krawędzi algorytm jest wykonywany w podobnym czasie.



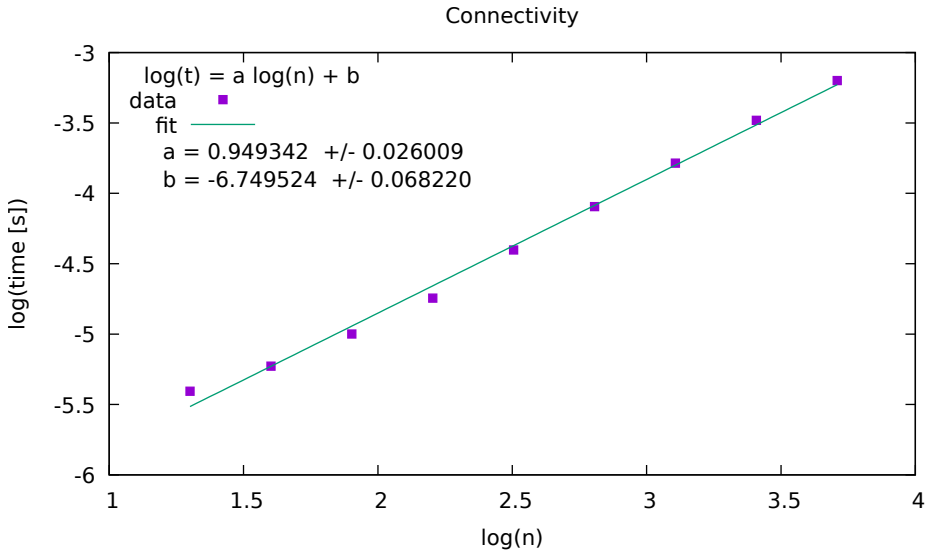

Rysunek A.1. Wykres wydajności algorytmu rozpoznania grafu permutacji.



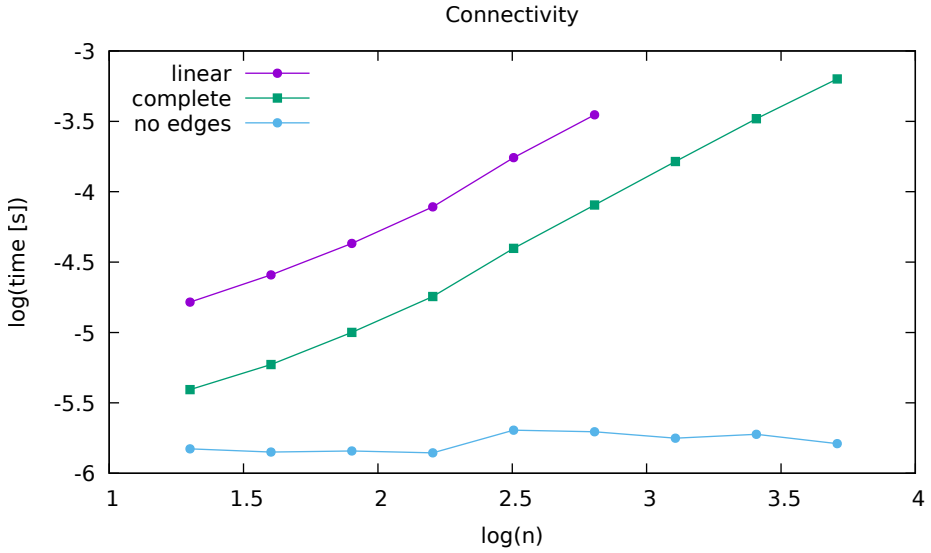

Rysunek A.2. Porównanie wydajności algorytmu rozpoznania grafu permutacji dla grafu liniowego, grafu pełnego i wierzchołków izolowanych.

## A.2. Testy sprawdzania spójności grafów permutacji

Algorytm sprawdzania spójności grafu permutacji przetestowano na grafach pełnych, a uzyskane wyniki przedstawiono na wykresie A.3. Współczynnik $a = 0.949(26)$ mówi, że jest to potwierdzona zależność liniowa. Porównano również czasy dla grafów liniowych i wierzchołków izolowanych. Uzyskane rezultaty przedstawiono na wykresie A.4. Wynika z nich, że graf pełny jest łatwiejszym przypadkiem niż graf liniowy, a dla grafu bez krawędzi algorytm działa w czasie stałym niezależnie od ilości wierzchołków.



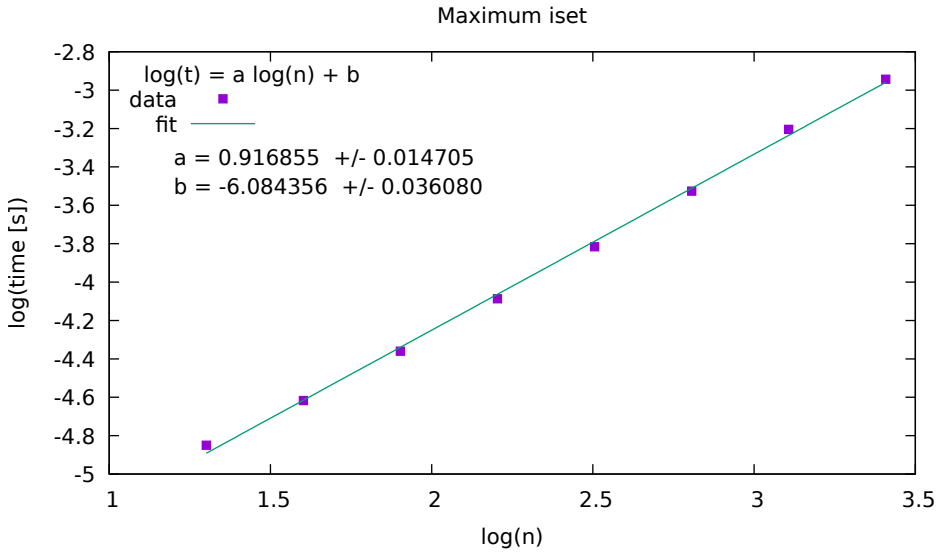

Rysunek A.3. Wykres wydajności algorytmu badania spójności grafu permutacji.



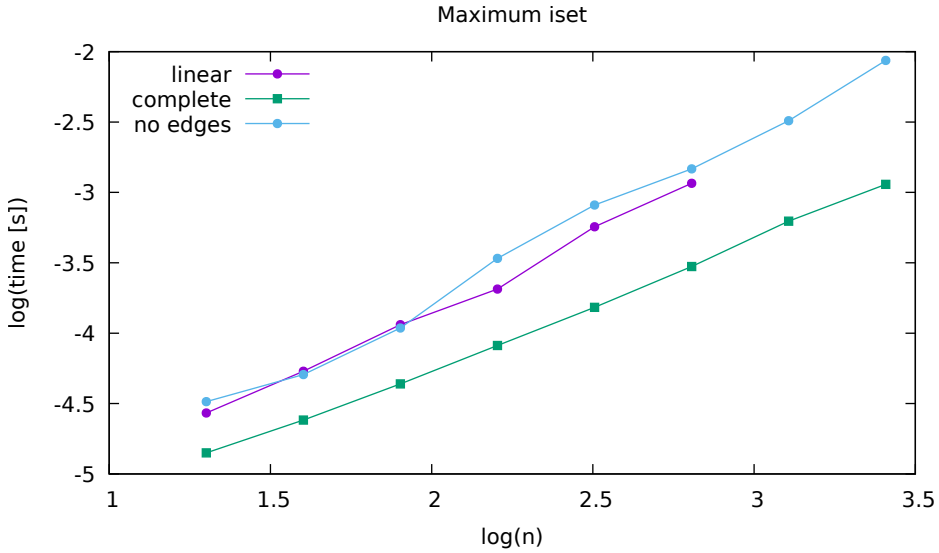

Rysunek A.4. Porównanie wydajności algorytmu badania spójności grafu permutacji dla grafu liniowego, grafu pełnego i wierzchołków izolowanych.

## A.3. Testy wyznaczania największego zbioru niezależnego

Znajdywanie największego zbioru niezależnego zostało przetestowane na coraz to większych grafach pełnych. Wyniki znajdują się na wykresie A.5. Wartość współczynnika prostej dopasowanej do danych pomiarowych wyniosła $a = 0.917(15)$. Możemy potwierdzić zakładaną wcześniej złożoność $O(n \log n)$. Podobnie jak dla uprzednich algorytmów wykonano porównanie dla konkretnych grafów A.6. Analogicznie do poprzednich algorytmów pomiary czasów dla grafu liniowego są większe niż dla grafu pełnego o tej samej liczbie wierzchołków. Warto zauważyć, że tym razem graf bez wierzchołków zajmuje jeszcze więcej czasu. Rozmiar grafów liniowych został ograniczony ze względu na wyjątek przekroczenia dozwolonej głębokość rekurencji występujący przy ich generowaniu.



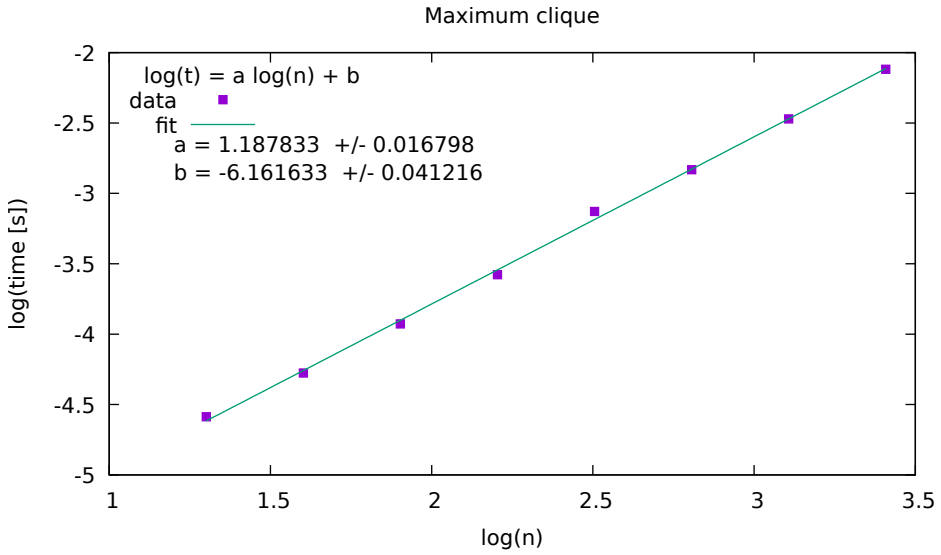

Rysunek A.5. Wykres wydajności algorytmu wyznaczania największego zbioru niezależnego dla grafu pełnego.



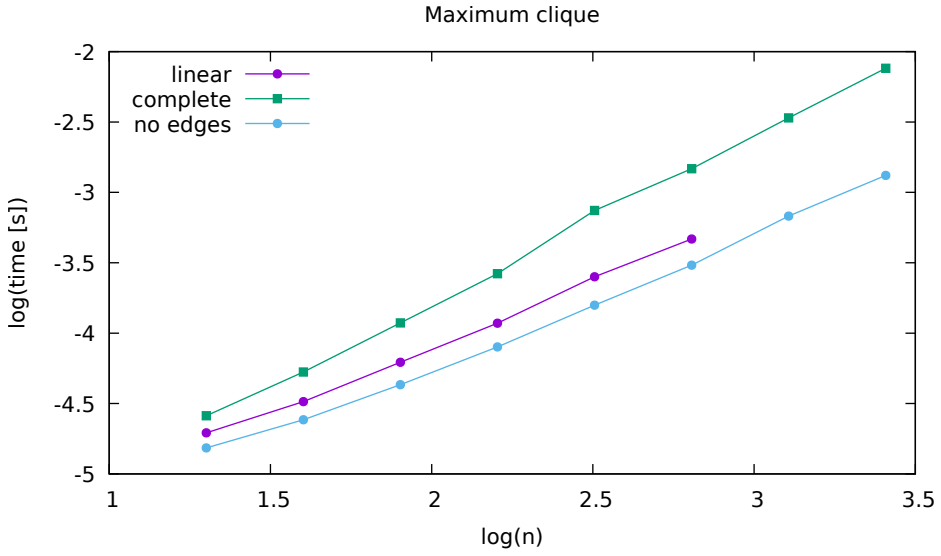

Rysunek A.6. Porównanie wydajności algorytmu wyznaczania największego zbioru niezależnego dla grafu liniowego, grafu pełnego i wierzchołków izolowanych.

## A.4. Testy wyznaczania największej kliki

Testy algorytmu wyznaczania największej kliki przeprowadzono analogicznie do testów wyznaczania największego zbioru niezależnego. Wyniki przedstawiono na wykresach: A.7 i A.8. Wartość współczynnika $a$ wyniosła 1.188(17). Jest to wynik większy niż dla algorytmu dla zbioru niezależnego, ale potwierdza on teoretyczną złożoność czasową $O(n \log n)$. Warto również zauważyć, że tym razem grafy pełne są najtrudniejszymi, a izolowane wierzchołki najłatwiejszymi przypadkami dla tego algorytmu.



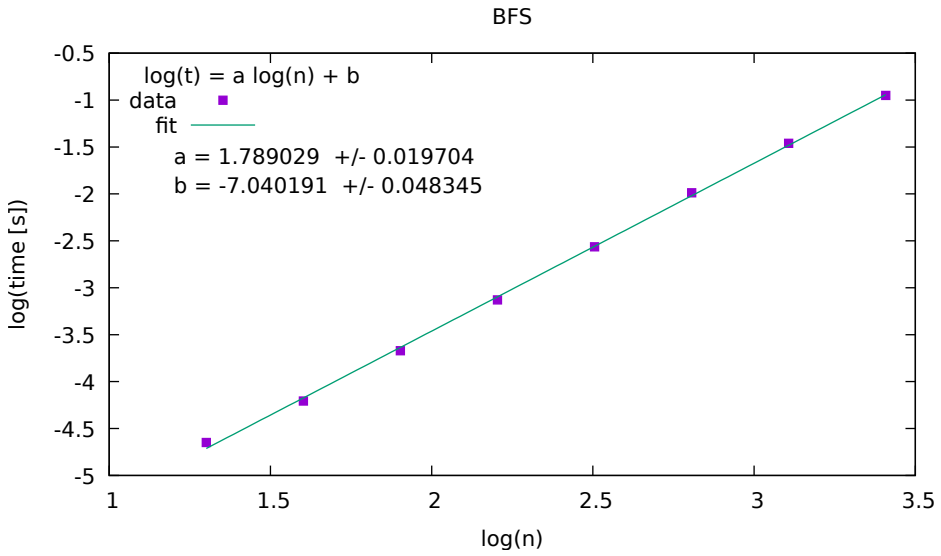

Rysunek A.7. Wykres wydajności algorytmu wyznaczania największej kliki.



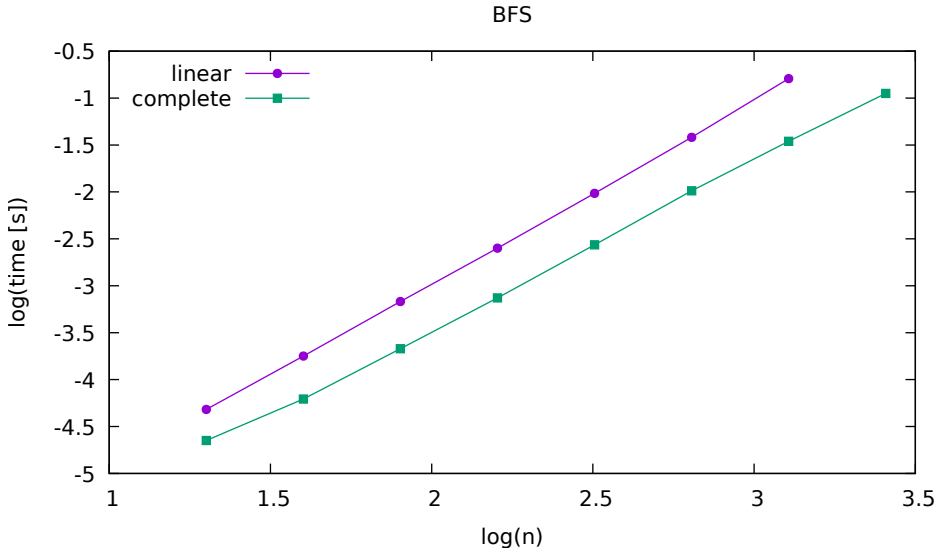

Rysunek A.8. Porównanie wydajności algorytmu wyznaczania największej kliki dla grafu liniowego, grafu pełnego i wierzchołków izolowanych.

## A.5. Testy przeszukiwania grafów permutacji

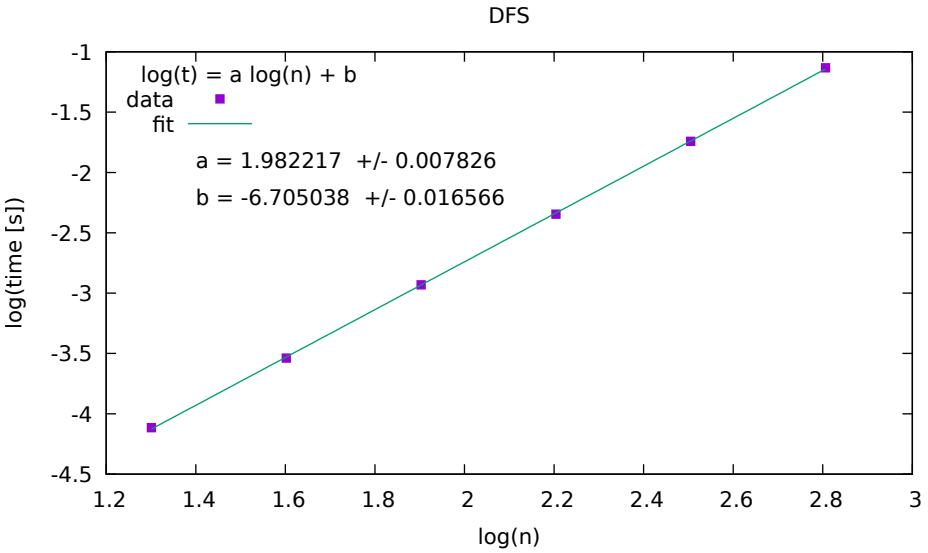
W celu sprawdzenia wydajności algorytmów przeszukiwań grafu. Przeprowadzono testy algorytmów BFS i DFS rozpoczynając przeszukiwanie za każdym razem od wierzchołka 0. Pojedyncze wywołania algorytmów wykonywano na grafach pełnych i liniowych. Wyniki zaprezentowano na wykresach A.9 i A.10 dla przeszukiwania wszerz oraz A.11 i A.11 dla przeszukiwania w głąb. Otrzymane wartości współczynników to $a = 1.79(2)$ dla BFS oraz $a = 1.98(1)$ dla DFS. Świadczą one o kwadratowej złożoności czasowej zaimplementowanych algorytmów.



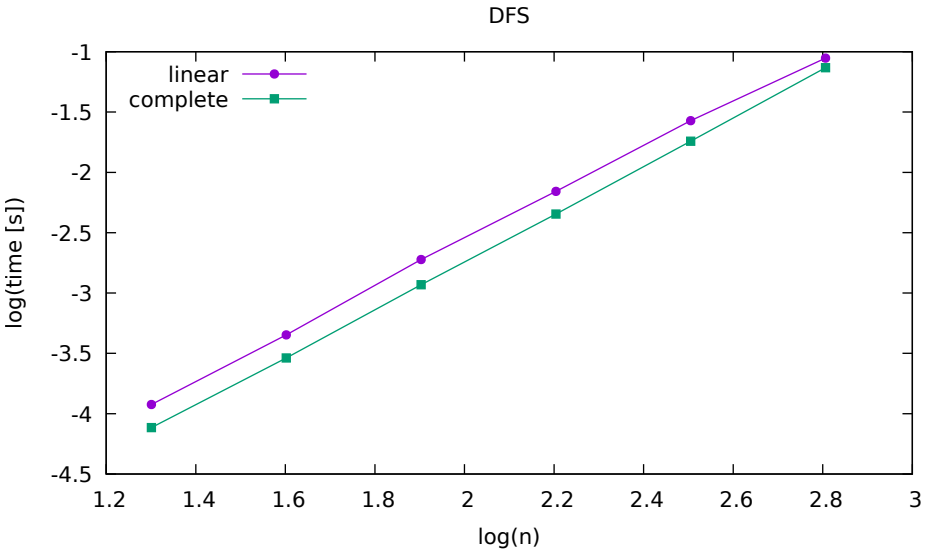

Rysunek A.9. Wykres wydajności algorytmu BFS dla grafu pełnego.




Rysunek A.10. Porównanie wydajności BFS dla grafu pełnego i liniowego.




Rysunek A.11. Wykres wydajności algorytmu DFS.




Rysunek A.12. Porównanie wydajności DFS dla grafu pełnego i liniowego.

# Bibliografia


[1] Andrzej Kapanowski, graphs-dict, GitHub repository, 2021, `https://github.com/ufkapano/graphs-dict/`.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[3] Robin J. Wilson, *Wprowadzenie do teorii grafów*, Wydawnictwo Naukowe PWN, Warszawa 1998.

[4] Jacek Wojciechowski, Krzysztof Pieńkosz, *Grafy i sieci*, Wydawnictwo Naukowe PWN, Warszawa 2013.

[5] Martin Charles Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[6] A. Pnueli, A. Lempel, S. Even, *Transitive orientation of graphs and identification of permutation graphs*, Canadian Journal of Mathematics, 1971.

[7] Ross M. McConnell, Jeremy P. Spinrad, *Modular decomposition and transitive orientation*, Discrete Mathematics, 1999.

[8] Python Programming Language - Official Website, `https://www.python.org/`.

[9] Python Docs, unittest, 2021, `https://docs.python.org/3/library/unittest.html`.

[10] Python Docs, timeit, 2021, `https://docs.python.org/3/library/timeit.html`.

[11] The On-Line Encyclopedia of Integer Sequences, 2021, `https://oeis.org/`.

[12] Wikipedia, Permutation graph, 2021, `https://en.wikipedia.org/wiki/Permutation_graph`.

[13] Wikipedia, Circle graph, 2021, `https://en.wikipedia.org/wiki/Circle_graph`.