Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Łukasz Malinowski

Nr albumu: 1052891

Implementacja wybranych algorytmów dla grafów bez wag w języku Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dr hab. Andrzej Kapanowski
Instytut Fizyki UJ

Kraków 2014

3423994616(1)

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

2053623312(2)

Pragnę złożyć serdeczne wyrazy podziękowania dla Pana Profesora Andrzeja Kapanowskiego za nieocenioną pomoc, wyrozumiałość, poświęcony czas oraz wielkie zaangażowanie.

2903229453(3)

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów i struktur danych dla algorytmów bez wag. Zdefiniowano wspólny interfejs dla grafów prostych i multigrafów, oparty na trzech klasach: Edge (dla krawędzi skierowanych), Graphs i MultiGraph. Stworzono sześć różnych implementacji realizujących interfejs grafów, łącznie z generatorami pewnych typów grafów.

W pracy zaimplementowano dwa najważniejsze algorytmy przeszukiwania grafów: przeszukiwanie wszerz i przeszukiwanie w głąb. Zaimplementowano dwa różne algorytmy sortowania topologicznego grafów acyklicznych skierowanych. Pokazano algorytmy wyznaczania składowych spójnych grafu nieskierowanego i silnie spójnych składowych grafu skierowanego. Zaimplementowano dwa sposoby wyznaczania domknięcia przechodniego, oraz kilka algorytmów do kolorowania wierzchołków i krawędzi grafu.

Dla grafów dwudzielnych stworzono implementację algorytmu sprawdzającego dwudzielność grafu, oraz implementacje trzech algorytmów wyznaczania maksymalnego skojarzenia w grafach dwudzielnych: algorytm ścieżek powiększających, algorytm Hopcrofta-Karpa, a także algorytm oparty o metodę Forda-Fulkersona.

Każdemu algorytmowi odpowiada osobna klasa, oraz odpowiedni zestaw testów jednostkowych, przygotowanych z wykorzystaniem standardowych narzędzi wspomagających sprawdzanie kodu w Pythonie. Ponadto dla wybranych implementacji wykonano eksperymenty komputerowe sprawdzające zgodność rzeczywistej wydajności kodu z przewidywaniami teoretycznymi.

Słowa kluczowe: grafy, multigrafy, przeszukiwanie wszerz, przeszukiwanie w głąb, sortowanie topologiczne, składowe spójne, silnie spójne składowe, grafy dwudzielne, maksymalne skojarzenie, domknięcie przechodnie, kolorowanie grafu, grafy eulerowskie, grafy hamiltonowskie

Abstract

Python implementations of selected graph algorithms and data structures for unweighted graphs are presented. Graphs interface is defined which is suitable for simple graphs and for multigraphs. The interface is based on three classes: the Edge class (for directed edges), the Graph class, and the MultiGraph class. Some graph generators are also included.

The algorithms are implemented using a unified approach. There are separate classes devoted to different algorithms. All algorithms were tested by means of the standard Python unit testing framework. Additional computer experiments were done in order to compare real and theoretical computational complexity.

Two algorithms for traversing of a graph are presented: breadth-first search and depth-first search. Two algorithms for finding a topological ordering of a dag are shown. Algorithms to compute the connected components of an undirected graph and to compute the strongly connected components of a directed graph are presented. There are two algorithms for computing the transitive closure of a graph, several algorithms for a vertex coloring and for an edge coloring. There are also algorithms connected with Eulerian graphs and Hamiltonian graphs.

For the case of bipartite graphs, there are algorithms for bipartiteness testing and for finding a maximum bipartite matching: the augmenting path algorithm, the Hopcroft-Karp algorithm, and the algorithm based on the Ford-Fulkerson method.

Keywords: graphs, multigraphs, breadth-first search, depth-first search, topological sorting, connected components, strongly connected components, bipartite graphs, maximum bipartite matching, transitive closure, graph coloring, Eulerian graphs, Hamiltonian graphs

Spis treści

8715576825(6)

2527807097(7)

3154200972(8)

1. Wstęp

Tematem niniejszej pracy jest *Implementacja wybranych algorytmów dla grafów bez wag w języku Python*. Praca skupia się na przedstawieniu w przejrzysty sposób algorytmów grafowych, ich implementacji w języku Python, oraz analizie ich wydajności. Wraz z rozwojem informatyki, oraz jej głównej gałęzi algorytmiki, coraz szybciej możemy obliczać interesujące nas zagadnienia związane z grafami. Przy implementacji algorytmów grafowych niewątpliwie duże znaczenie ma w jakim języku tworzymy implementację. Niektóre języki pozwalają na mniej, bądź bardziej wydajne implementacje, poprzez wewnętrzne optymalizacje związane z pamięcią, operacjami na zmiennych, przechowywaniem danych w różnych strukturach, oraz różnicami w implementacji kontenerów w bibliotece standardowej. W pracy został użyty język Python, który łączy w sobie przejrzystość, czytelność, oraz wydajność pisanych w nim aplikacji.

Tematyka pracy była już wcześniej podejmowana w wielu książkach, między innymi w klasycznym podręczniku Cormena [1], jednak rzadko kiedy autorzy skupiali się na implementacji w konkretnym języku programowania. Częściej publikacje mają charakter bardziej teoretyczny i skupiają się na matematycznej stronie teorii grafów [2], [3]. Jednym z wyjątków jest książka Sedgewicka, prezentująca implementacje w C++ [4].

Praca podzielona została na sześć rodziałów. Na początku zostaną omówione podstawowe pojęcia związane z grafami oraz językiem Python, który został wykorzystany do przedstawienia implementacji algorytmów. Następnie opisane są pojęcia związane z teorią grafów, które zostały wykorzystane w różnych cześciach pracy. W następnej kolejności znajduje się rozdział poświęcony algorytmom i ich implementacjom w języku Python. Praca nie zawiera dowodów poprawności algorytmów, ponieważ są one znane w literaturze zamieszczonej w bibliografii. Na końcu pracy znajdują się dodatki zawierające kody źródłowe podstawowych klas grafowych, oraz omówienie testów komputerowych. Praca ta oczywiście nie wyczerpuje całości tematu, a stanowi jedynie solidną bazę do dalszej analizy algorytmów grafowych. Przyjęte konwencje prezentacji algorytmów ułatwiają zapisywanie, analizę, rozumienie i rozwijanie nowych ideii.

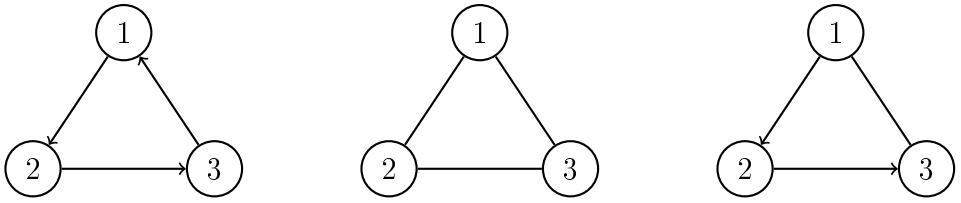
1.1. Grafy

Grafem nazywamy parę składającą się ze zbioru obiektów (wierzchołków) oraz ze zbioru połączeń między tymi obiektami (krawędzie). W celu oznaczenia wierzchołków w grafie najczęściej stosowane są liczby naturalne albo etykiety tekstowe. Krawędzie grafu obrazują relacje pomiędzy jego wierzchołkami i często posiadają dodatkowe atrybuty, takie jak waga, kolor lub

1639995858(9)

długość. Dzięki grafom można łatwiej analizować różne relacje i procesy w układach biologicznych, fizycznych, społecznych i informatycznych. Zapisanie badanego problemu w języku teorii grafów ułatwia korzystanie ze znanych rozwiązań, dokładnych lub przybliżonych.

Działem matematyki i informatyki, zajmującym się badaniem własności grafów, jest teoria grafów 3. Grafy możemy podzielić ze względu na typ krawędzi. W *grafach skierowanych* krawędzie posiadają określony kierunek, jest wierzchołek początkowy i wierzchołek końcowy dla każdej krawędzi. W *grafach nieskierowanych* krawędzie nie mają określonego kierunku, natomiast w *grafach mieszanych* występują oba rodzaje krawędzi.



(a) graf skierowany (b) graf nieskierowany (c) graf mieszany

Rysunek 1.1. (a) Graf skierowany - krawędzie łączące wierzchołki mają określony kierunek. (b) Graf nieskierowany - krawędzie łączące wierzchołki nie mają określonego kierunku. (c) Graf mieszany - zawiera oba rodzaje krawędzi.

1.2. Multigrafy

Multigrafy są uogólnieniem grafów, które często jest potrzebne w zastosowaniach. Dopuszcza się istnienie pętli (krawędź łącząca wierzchołek z nim samym) oraz krawędzi wielokrotnych (kilka krawędzi ma ten sam wierzchołek początkowy i końcowy). W pracy będziemy używać nazwy multigraf tylko tam, gdzie dopuszczalne są pętle i krawędzie wielokrotne. W przeciwnym wypadku będziemy korzystać z nazwy graf lub graf prosty.

1.3. Organizacja pracy

Rozdział 1 zawiera wstęp do pracy magisterskiej. Rozdział 2 zawiera wprowadzenie do języka Python. Zostały w nim przedstawione podstawowe struktury danych oraz cechy języka, które należy poznać, aby zrozumieć implementacje algorytmów. Rozdział 3 wprowadza podstawowe pojęcia z zakresu teorii grafów. Do tych pojęć często odwołujemy się w innych częściach pracy. Rozdział 4 opisuje zagadnienia związane z grafami istotne dla ich implementacji, oraz interfejs użyty przy tworzeniu obiektów grafów, które później zostały użyte przy implementacji algorytmów. Rozdział 5 przedstawia opis oraz implementację wybranych algorytmów grafowych. W rozdziale tym można również napotkać uwagi przydatne przy implementacji algorytmów.

Rozdział 6 jest ostatnim rozdziałem zawierającym podsumowanie pracy. Dodatkowo na końcu pracy znajdują się dodatki zawierające kod klas grafowych oraz wyniki testów algorytmów.

2206782925(11)

2. Wprowadzenie do Pythona

Python jest obiektowo-zorientowanym językiem programowania wysokiego poziomu [5]. Oznacza to, że jego składnia oraz słowa kluczowe zostały stworzone tak, aby były w jak największym stopniu zrozumiałe i wygodne dla człowieka, a nie dla maszyny. Python został stworzony zgodnie z paradygmatem programowania obiektowego (ang. *object-oriented programming*), w mniejszym stopniu wspiera również programowanie proceduralne i funkcyjne. W języku tym wszystko jest obiektem, a określone zachowania zdefniowane dla obiektów nazwane są metodami. Python zawiera rozbudowany zbiór pakietów nazywany *biblioteką standardową*, który znacząco ułatwia wydajne tworzenie oprogramowania. Język cechuje czytelna i klarowna składnia. W celu poprawienia czytelności stosowany jest system wcięć do zaznaczenia bloku instrukcji. Dla usprawnienia zarządzania pamięcią, oraz eliminacji trudnych do wykrycia błędów programistycznych, zastosowano automatyczne zarządzanie pamięcią za pomocą tzw. odśmiecania pamięci (ang. *garbage collection*). Język posiada typy dynamiczne, czyli zmienne w programie nie mają określonych typów, natomiast mogą przechowywać łącza do obiektów różnych typów. Ten sam kod może być wykorzystywany na różnych platformach, ponieważ Python nie jest zależny od konkretnego systemu operacyjnego. Interpretery języka Python dostępne są dla większości popularnych systemów operacyjnych.

2.1. Typy i struktury danych

Python posiada wiele wbudowanych typów oraz struktur danych. W programie nie ma potrzeby definiowania typu danej zmiennej, jak w językach z typami statycznymi, ponieważ Python jest językiem dynamicznie typowanym. Rozpoznaje on typy obiektów i przypisuje do zmiennej łącze do obiektu. Wszystkie zmienne w Pythonie przenoszone są przez referencję, a nie przez wartość. Oznacza to, że funkcja otrzymuje łącze do oryginału obiektu, a nie do jego niezależnej kopii. Wszystkie operacje wykonane na obiekcie wewnątrz funkcji będą widoczne po wyjściu z funkcji. Oczywiście w razie potrzeby można łatwo wykonać pełną, niezależną kopię każdego obiektu (moduł copy), ale zwykle nie jest to potrzebne.

2.1.1. Typy proste

Wśród typów prostych można wyróżnić typy numeryczne i tekstowe. Pomiędzy typami numerycznymi następuje niejawna automatyczna konwersja, dlatego nie trzeba jawnie konwertować liczb całkowitych na liczby zmiennoprzecinkowe. Język Python jest językiem silnie typowalnym oznacza to, że

1634434094(12)

tam gdzie niejawna konwersja mogłaby powodować utratę precyzji albo niejednoznaczności, należy samemu dokonać jawnej konwersji. Tekst zawierający liczbę musi być więc jawnie konwertowany na liczbę i odwrotnie, liczba musi być jawnie konwertowana na tekst. Poniższa tabelka zwiera typy podstawowe w Pythonie.

Tabela 2.1. Typy proste w Pythonie.

Typ	**Opis**	**Przykład**
None	obiekt pusty (odpowiednik null)	None
bool	wartość logiczna	True, False
int	liczba całkowita	875, −3
long	liczba całkowita długa	123456789000L
float	liczba zmiennoprzecinkowa	13.128, 2e3
complex	liczba zespolona	4+2.5j, −7J
str	tekst	'tekst', "tekst"

2.1.2. Kolekcje

Kolekcje w Pythonie zostały zaprojektowane tak, żeby można było je elastycznie używać. Python nie posiada tablicy o stałym rozmiarze tylko listę, którą można dowolnie rozszerzać. Szeroki zbiór kolekcji wbudowanych, takich jak lista, krotka, zbiór, czy słownik, umożliwia bardzo efektywne i szybkie pisanie programów. W celu użycia dowolnej kolekcji wystarczy do nazwy zmiennej przypisać obiekt kolekcji, co ilustruje przykład poniżej.

Listing 2.1. Tworzenie kolekcji.

```
lista = list()
krotka = tuple()
zbior = set()
slownik = dict()
```

Podstawowe kolekcje, które zostały użyte podczas pisania pracy, wraz z przykładem użycia, prezentuje tabela 2.2.

Tabela 2.2. Kolekcje w Pythonie.

Typ	**Opis**	**Przykład**
list	lista (zmienna długość i zawartość)	[1, "a", True]
tuple	krotka (niezmienna)	(1, "a", True)
set	zbiór (zmienny)	set ([1, "a", True])
frozenset	zbiór (niezmienny)	frozenset ([1, "a", True])
dict	słownik (zmienny)	{'a': 1.2, 1: True}

2.2. Składnia języka

2.2.1. System wcięć

Aby zapewnić czytelność kodu oraz ustandaryzować sposób pisania programów, Python zamiast nawiasów ograniczających ({ }) stosuje system wcięć.

6201674549(13)

Instrukcje kończą się wraz z końcem linii, chyba że jawnie złamiemy wiersz znakiem ukośnika wstecznego \ (ang. *backslash*).

Listing 2.2. System wcięć w Pythonie.

```
for x in xrange(20):
    if x < 10:
        if x == 5:
            print "x = 5"
        else:
            print "x < 10 and x != 5"
    else:
        if x == 15:
            print "x=15"
        else:
            print "x >= 10 and x != 15"
```

2.2.2. Operatory

Podstawowe operatory logiczne w Pythonie to koniunkcja **and** oraz alternatywa **or**. Działają w ten sposób, że zwracają wartość ostatniego obliczonego wyrażenia. W przypadku wyrażenia 1 **and** 0 **or** 5 wynikiem będzie 5. Python posiada również operatory porównania (==, !=, <, >, <=, >=, **is**), negacji **not** oraz zawierania **in**. Operatory te zwracają wartość logiczną True lub False.

2.2.3. Komentarze

Kiedy chcemy dodać luźne uwagi w kodzie programu, które nie będą interpretowane przez interpreter języka, można użyć jednego z dwóch typów kometarzy jakie posiada Python: komentarz jednoliniowy lub komentarz wieloliniowy. Komentarz jednoliniowy tworzy się przez wstawienie przed zawartością komentarza znaku hash (#). Cały tekst znajdujący się po tym znaku, aż do końca linii, nie jest interpretowany. Komentarz wieloliniowy tworzy się przez umieszczenie komentarza pomiędzy potrójnymi apostrofami (''') lub cudzysłowami (""""). Kiedy używamy komentarza wieloliniowego na początku modułu, klasy lub funkcji, to pełni on dodatkowo fukcję dokumentacyjną. Aby zobaczyć opis modułu, klasy lub funkcji, który stworzyliśmy przez dodanie komentarza wieloliniowego, należy skorzystać z atrybutu __doc__. Python wspiera także automatyczne generowanie dokumentacji dla programu.

2.3. Instrukcje sterujące

Python, podobnie jak większość języków programowania, posiada instrukcje sterujące przebiegiem wykonania programu. Do podstawowych instrukcji sterujących opisanych w tym paragrafie należą pętle oraz instrukcje warunkowe.

6834589454(14)

2.3.1. Instrukcja warunkowa

Python posiada instrukcję warunkową **if**, która umożliwia sterowanie przebiegiem wykonania programu. Jako wyrażenie warunkowe niekoniecznie musi zostać użyta wartość logiczna, jak w innych językach. Każda liczba z wyjątkiem zera jest traktowana jako prawda, a zero jako fałsz. W Pythonie obiekt None, a także puste kolekcje list (), set (), dict () interpretowane są jako fałsz.

Listing 2.3. Instrukcja warunkowa **if** w Pythonie.

```
if expression_1:
    instructions
elif expression_2:    # opcjonalnie
    instructions
else:    # opcjonalnie
    instructions
```

2.3.2. Pętle

Do powtarzania tej samej czynności wiele razy Python udostępnia dwa rodzaje pętli, pętlę **while** oraz pętlę **for**. Pętla **for** działa inaczej niż w innych językach. Nie zwiększamy jawnie indeksu iteratora, tylko iterujemy po kolekcji obiektów. Fukcja xrange(n) tworzy generator liczb z zakresu od 0 do $n-1$, po którym następnie przechodzimy. W pętlach można również zastosować instrukcje **break** (natychmiastowe wyjście z pętli) oraz **continue** (kontynuowanie pętli).

Listing 2.4. Pętla **for** od 0 do 9 w Pythonie.

```
for i in xrange(10):
    instructions
```

Pętla **while** działa podobnie jak w innych językach. Jeśli wyrażenie warunkowe jest prawdziwe, to zawartość pętli jest wykonywana, w przeciwnym wypadku program nie wykonuje pętli.

Listing 2.5. Pętla **while** w Pythonie.

```
while expression:
    instructions
```

2.4. Funkcje

Funkcje w Pythonie definiujemy za pomocą słowa kluczowego **def**. W definicji funkcji nie podajemy ani zwracanego typu, ani czy w ogóle funkcja zwraca jakąś wartość. Jeśli chcemy, żeby funkcja zwracała jakąś wartość, wewnątrz funkcji musimy użyć słowa kluczowego **return**, a po nim wartość, która ma być zwrócona. Do funkcji możemy przekazać także dowolną ilość argumentów, można również stworzyć funkcję, która przyjmuje nieokreśloną liczbę argumentów.

2183886043(15)

Listing 2.6. Przykład funkcji w Pythonie.

```
def function(a, b):
    return a * b
```

Wygodną funkcjonalnością oferowaną przez Pythona są argumenty domyślne dla funkcji. W definicji funkcji po nazwie argumentu wystarczy przypisać do niego jego wartość domyślną. Jeżeli funkcja zostanie wywołana bez danego argumentu, to zostanie mu przypisana jego wartość domyślna.

Przy wywoływaniu funkcji możemy podawać argumenty w takiej kolejności, w jakiej zostały podane w definicji funkcji, ale można zmienić ich kolejność poprzedzając przekazywaną wartość argumentu jego nazwą oraz znakiem równości.

Listing 2.7. Przykład funkcji z argumentami domyślnymi.

```
def function(a=5, b=10):
    return a * b

# argument b przyjmuje wartosc domyslna 10
result = function(20)    # wynik to 200

# argument a przyjmuje wartosc domyslna 5
result = function(b=20)    # wynik to 100
```

2.5. Moduły

Moduły w języku Python to pliki z rozszerzeniem .py. Każdy moduł powinien zawierać odrębną funkcjonalność programu. Moduły importujemy do swojego programu za pomocą słowa kluczowego **import**. Każdy moduł posiada atrybut __name__, który w zależności czy moduł jest uruchomiony czy też importowany wewnątrz jakiegoś innego modułu ustawiany jest jako wartość "__main__" lub jako nazwa modułu.

Listing 2.8. Przykład importowania modułów w Pythonie.

```
import importlib
import os, sys
from collections import deque
```

Kiedy chcemy przejrzeć zawartość modułu możemy użyć funkcji dir. Wyświetli ona wszystkie funkcje i klasy dostępne w module.

Listing 2.9. Przykład użycia funkcji dir.

```
>>> import io
>>> dir(io)
['BlockingIOError', 'BufferedIOBase', 'BufferedRWPair',
'BufferedRandom', 'BufferedReader', 'BufferedWriter',
'BytesIO', 'DEFAULT_BUFFER_SIZE', 'FileIO', 'IOBase',
'IncrementalNewlineDecoder', 'OpenWrapper', 'RawIOBase',
'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'StringIO',
'TextIOBase', 'TextIOWrapper', 'UnsupportedOperation',
'__all__', '__author__', '__builtins__', '__doc__',
```


2154796265(16)

```
'__file__', '__name__', '__package__', '_io', 'abc',
'open']
```

2.6. Obiektowość w Pythonie

Python jest zorientowanym obiektowo językiem programowania. Wspiera dziedziczenie, polimorfizm, abstrakcje i enkapsulacje.

2.6.1. Klasy

Klasa to szablon według którego będą tworzone obiekty danego typu. Klasy tworzy się za pomocą słowa kluczowego **class**, a po nim podajemy nazwę klasy. Python posiada także metody specjalne dla klasy, jedną z nich jest metoda __init__ inicjalizująca obiekt danej klasy. Metoda ta może posiadać argumenty, które zostaną przekazane do obiektu podczas jego tworzenia. Każda metoda wewnątrz klasy, jeśli nie jest to metoda statyczna, posiada jako pierwszy argument zmienną **self**, która jest referencją do obiektu, na którym wykonywana jest metoda. Pola w klasach nie posiadają modyfikatora dostępu. Wszystkie pola są domyślnie publiczne i dostępne z każdego poziomu.

Listing 2.10. Przykład deklaracji klasy w Pythonie.

```
class MyClass:
    """Opis klasy."""

    def __init__(self):
        """Inicjalizacja obiektu."""
        pass
```

2.6.2. Obiekty

Obiekty są zwykle instancjami pewnych klas. W obiekcie przechowywane są atrybuty, do których możemy się odwoływać w trakcje życia obiektu. W Pythonie nie musimy jawnie niszczyć obiektu, robi to za nas odśmiecacz pamięci, kiedy nie ma już żadnych odniesień do istniejącego obiektu. Obiekty w Pythonie tworzymy w bardzo prosty sposób, poprzez wywołanie nazwy klasy, z ewentualnymi argumentami. Referencję do obiektu zwykle zapisujemy w jakiejś zmiennej.

Listing 2.11. Przykład tworzenia obiektu w Pythonie.

```
instance = MyClass()
```

2.6.3. Dziedziczenie

Dziedziczenie jest jedną z cech obiektowości, która pozwala na współdzielnie przez klasy funkcjonalności, oraz na ominięciu powielania kodu przy

3360289667(17)

tworzeniu klas. Python zapewnia bardzo prosty sposób dziedziczenia, wystarczy nazwę klasy, z której chcemy oddziedziczyć, umieścić w nawiasach po nazwie klasy dziedziczącej.

Listing 2.12. Przykład dziedziczenia w Pythonie.

```
class A:
    def __init__(self):
        pass

class B(A):    # klasa B dziedziczy z klasy A
    def __init__(self):
        pass
```

Python obsługie również dziedziczenie wielokrotne. Pozwala to na jeszcze większe współdzielenie funkcjonalności przez klasy.

Listing 2.13. Przykład dziedziczenia wielokrotnego w Pythonie.

```
class A:
    def __init__(self):
        pass

class B:
    def __init__(self):
        pass

class C(A, B):    # dziedziczenie z klas A i B
    def __init__(self):
        pass
```

2.6.4. Wyjątki

Wyjątki są bardzo wygodną i nowoczesną metodą sygnalizacji błędów i sytuacji nietypowych w programie. Python wspiera obsługę wyjątków. W Pythonie wyróżniamy kilka typów wyjątków: informujące o błędach składni, wyjątki wbudowane oraz definiowane przez użytkownika. Wyjątki są obecnie klasami, a ich instancje są zwykłymi obiektami, które możemy przechwytywać i wykonywać na nich pewne operacje. Najczęstszą operacją jest pobranie informacji o błędzie z wyjątku. Jeśli podczas pisania programu użyjemy niedozwolonej składni języka, to otrzymamy wyjątek SyntaxError.

Listing 2.14. Przykład błędu składni w Pythonie.

```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                     ^
SyntaxError: invalid syntax
```

Kiedy chcemy zdefiniować własny wyjątek, wystarczy stworzyć klasę dziedziczącą po klasie Exception.

Listing 2.15. Przykład tworzenia wyjątku w Pythonie.

```
class MyError(Exception):
```


3912986238(18)

```
    def __init__(self):
        pass

    def __str__(self):
        return "MyError"
```

Aby rzucić wyjątek, należy użyć słowa kluczowego **raise**. Jeśli otoczymy segment kodu, w którym użyto **raise**, blokiem **try**, **except**, to będziemy mogli przechwycić rzucony wyjątek.

Listing 2.16. Przykład obsługiwania wyjątku w Pythonie.

```
try:
    raise MyError()
except MyError as exception:
    print "Exception:", exception
```

2.7. Testowanie

Do tworzenia testów jednostkowych w Pythonie służy moduł unittest. Podstawowy test weryfikujący poprawność działania danej cześci programu nazywamy TestCase. Natomiast grupa TestCase tworzy TestSuite. Do testowania zależności, jakie zachodzą pomiędzy obiektami, używa się funkcji assertEqual, która sprawdza czy wartość podana w pierwszym argumencie jest identyczna z wartością podaną w drugim argumencie. Do zainicjalizowania początkowych danych potrzebnych do testów należy w klasie testującej umieścić metodę setUp. Do działań czyszczących stosuje się metodę tearDown. Nazwy klas i metod testujących należy tworzyć zgodnie z konwencją.

Listing 2.17. Przykład testów w Pythonie.

```
import unittest

class TestValue(unittest.TestCase):

    def setUp(self):
        self.value = 100

    def test_int(self):
        self.assertEqual(self.value, 100)

    def test_bool(self):
        self.assertTrue(self.value)

    def tearDown(self):
        pass

if __name__ == "__main__":
    unittest.main()    # uruchomienie testow
```


1347676236(19)

2.8. Wydajność

Dużą zaletą Pythona jest jego wysoka wydajność w porównaniu z innymi językami skryptowymi. Większość podstawowych modułów Pythona napisana jest w języku C, dlatego tworzenie nawet dużych projektów w Pythonie ma sens. Dla zwiększenia wydajności Python może skompilować pliki źródłowe do bajtkodu (pliki z rozszerzeniem .pyc) i tym samym przyspieszyć ich wykonywanie. Często tworzona aplikacja ma pewien krytyczny fragment, który wystarczy napisać w języku C/C++, a całą resztę sterowania można wygodnie budować w Pythonie.

Przy tworzeniu wydajnych programów przydaje się wiedza o implementacji danych obiektów wbudowanych, czy mechanizmów Pythona. Z drugiej strony zamiast polegać na intuicji warto przeprowadzać testy praktyczne różnych realizacji obliczeń, aby wybrać optymalne rozwiązanie. Jest to też istotne z powodu szybkiego rozwoju języka i ulepszania wielu jego elementów.

3. Teoria grafów

W różnych podręcznikach dotyczących teorii grafów można spotkać nieco różniące się definicje, dlatego podamy definicje pojęć używanych w niniejszej pracy. Wiodącym źródłem będzie książka Cormena [1].

3.1. Multigrafy skierowane

Multigraf (ang. *multigraph*) jest to uporządkowana para $G = (V, E)$, gdzie V to zbiór wierzchołków (skończony i niepusty), a E to wielozbiór (multizbiór, rodzina) krawędzi. Jest istotne, że elementy wielozbioru mogą się powtarzać, choć ich kolejność nie jest ustalona. Krawędzią nazywamy uporządkowanę parę (s, t) dwóch wierzchołków ze zbioru V. Tak określona krawędź jest skierowana z s do t, a multigraf G nazywamy *skierowanym*. Krawędź (s, t) jest *wychodząca* z wierzchołka s i jest *wchodząca* do wierzchołka t. Wierzchołek t jest *sąsiedni* względem wierzchołka s, jeżeli w multigrafie istnieje krawędź (s, t) (relacja sąsiedztwa). *Pętla* jest to krawędź postaci (s, s).

Multigraf $G' = (V', E')$ jest *podmultigrafem* multigrafu $G = (V, E)$, jeżeli V' jest podzbiorem V, oraz E' zawiera się w E. Symbole $|V|$ i $|E|$ oznaczają odpowiednio liczbę wierzchołków i liczbę krawędzi multigrafu (z powtórzeniami). W zapisie z notacją O będziemy dla prostoty pisać V i E.

Stopniem wyjściowym wierzchołka s nazywamy liczbę krawędzi wychodzących z s i oznaczamy przez $\mathrm{outdeg}(s)$. *Stopniem wejściowym* wierzchołka s nazywamy liczbę krawędzi wchodzących do s i oznaczamy przez $\mathrm{indeg}(s)$.

Lemat o uściskach dłoni (multigrafy skierowane): Dany jest multigraf skierowany $G = (V, E)$. Spełniona jest zależność

$$\sum_{s \in V} \mathrm{indeg}(s) = |E| = \sum_{s \in V} \mathrm{outdeg}(s). \tag{3.1}$$

3.2. Multigrafy nieskierowane

Często definiuje się krawędzie *nieskierowane* jako dwuelementowe podzbiory zbioru V, np. $\{s, t\}$. Można powiedzieć, że w wielozbiorze E istnieją jednocześnie dwie krawędzie skierowane (s, t) i (t, s), które są liczone jako jedna krawędź nieskierowana. Pętla nieskierowana jest wtedy reprezentowana przez (s, s), czy raczej przez mały wielozbiór $\{s, s\}$. Multigraf zawierający tylko krawędzie nieskierowane nazywamy *multigrafem nieskierowanym*. Krawędź $\{s, t\}$ jest *incydentna* z wierzchołkami s i t. W multigrafie nieskierowanym relacja sąsiedztwa jest symetryczna.

3487346537(21)

3.3. Grafy regularne

Stopień wierzchołka $\deg(s)$ w multigrafie nieskierowanym jest to liczba krawędzi incydentnych z wierzchołkiem s, przy czym pętle są liczone podwójnie. *Graf regularny stopnia* n (inaczej: graf n-regularny) jest to graf nieskierowany, w którym wszystkie wierzchołki są stopnia n. Szczególnym przypadkiem grafów regularnych są *grafy kubiczne* (grafy 3-regularne). Ciekawe grafy kubiczne to graf Wagnera ($|V| = 8$, $|E| = 12$) i graf Petersena ($|V| = 10$, $|E| = 15$).

Lemat o uściskach dłoni (multigrafy nieskierowane): Dany jest multigraf nieskierowany $G = (V, E)$. Spełniona jest zależność

$$\sum_{s \in V} \deg(s) = 2|E|. \tag{3.2}$$

Jako wniosek otrzymujemy fakt, że w dowolnym multigrafie nieskierowanym liczba wierzchołków o nieparzystych stopniach jest parzysta.

3.4. Ścieżki

Ścieżką (drogą) P z s do t w multigrafie $G = (V, E)$ nazywamy sekwencję wierzchołków $(v_0, v_1, v_2, \ldots, v_n)$, gdzie $v_0 = s$, $v_n = t$, oraz (v_{i-1}, v_i) $(i = 1, \ldots, n)$ są krawędziami z E. Długość ścieżki P wynosi n. Ścieżka składająca się z jednego wierzchołka ma długość zero. Jeżeli istnieje ścieżka z s do t, to mówimy, że t jest *osiągalny* z s. *Ścieżka prosta* to ścieżka, w której wszystkie wierzchołki są różne.

3.5. Cykle

Cykl jest to ścieżka, w której pierwszy i ostatni wierzchołek są takie same, $v_0 = v_n$. *Cykl prosty* jest to cykl, w którym wszystkie wierzchołki są różne, z wyjątkiem ostatniego. Wydaje się, że w literaturze ścieżki typu (s) i (s, t, s) [multigraf nieskierowany z pojedyńczą krawędzią $\{s, t\}$] nie są uważane za cykle proste. Natomiast multigraf z pętlą [skierowaną (s, s) lub nieskierowaną $\{s, s\}$] lub ścieżką (s, t, s) [multigraf nieskierowany z krawędią wielokrotną $\{s, t\}$ lub multigraf skierowany z krawędziami (s, t) i (t, s)] będzie uważany za cykliczny.

Multigraf niezawierający cykli prostych nazywamy *acyklicznym*, a w świetle poprzednich rozważań będzie to *graf acykliczny*. Graf skierowany acykliczny nazywamy *dagiem* (ang. *directed acyclic graph*).

3.6. Spójność

Multigraf nieskierowany jest *spójny* (ang. *connected*), jeżeli każdy wierzchołek jest osiągalny ze wszystkich innych wierzchołków. Multigraf skierowany jest *silnie spójny* (ang. *strongly connected*), jeśli każde dwa wierzchołki są osiągalne jeden z drugiego.

1714247437(22)

3.7. Grafy hamiltonowskie

Ścieżka Hamiltona to ścieżka przechodząca przez każdy wierzchołek multigrafu dokładnie raz. *Cykl Hamiltona* jest to cykl prosty przebiegający przez wszystkie wierzchołki multigrafu dokładnie jeden raz, oprócz ostatniego wierzchołka. *Graf hamiltonowski* to graf zawierający cykl Hamiltona. *Graf półhamiltonowski* to graf zawierający ścieżkę Hamiltona.

Problem znajdowania ścieżki Hamiltona jest NP-zupełny. Nieformalnie można powiedzieć, że graf jest hamiltonowski, jeżeli tylko ma on odpowiednio dużo krawędzi w stosunku do ilości wierzchołków.

3.8. Grafy eulerowskie

Ścieżka Eulera jest to ścieżka przechodząca przez każdą krawędź multigrafu dokładnie raz. *Cykl Eulera* jest to cykl przechodzący przez wszystkie krawędzie multigrafu dokładnie jeden raz. *Graf eulerowski* to graf zawierający cykl Eulera. *Graf półeulerowski* to graf zawierający ścieżkę Eulera.

Jeżeli wszystkie wierzchołki grafu nieskierowanego mają stopień parzysty, to da się skonstruować cykl Eulera. Jeżeli najwyżej dwa wierzchołki mają stopień nieparzysty, to istnieje ścieżka Eulera.

Graf skierowany jest eulerowski, jeżeli jest silnie spójny, oraz w każdym wierzchołku stopień wchodzący jest równy stopniowi wychodzącemu. Graf skierowany będzie półeulerowski, gdy wszystkie wierzchołki z wyjątkiem dwóch mają takie same stopnie wychodzące i wchodzące, w jednym z tych dwóch wierzchołków stopień wychodzący jest o jeden większy niż wchodzący a w drugim odwrotnie. *Algorytm Fleury'ego* pozwala na odszukanie cyklu Eulera w grafie eulerowskim.

3.9. Drzewa i las

Drzewo (ang. *tree*) jest to graf prosty nieskierowany, spójny i acykliczny. *Drzewo rozpinające* (ang. *spanning tree*) multigrafu G jest to drzewo, które zawiera wszystkie wierzchołki multigrafu G i jest podgrafem G. *Las* jest to niespójny graf nieskierowany i acykliczny, czy suma rozłącznych drzew.

Drzewo ukorzenione jest to drzewo, w którym wyróżniono jeden wierzchołek, zwany *korzeniem* (ang. *root*). Dla dowolnej ścieżki prostej rozpoczynającej się od korzenia stosuje się takie pojęcia, jak przodek, potomek, rodzic lub ojciec, dziecko lub syn. Krawędzie w drzewie ukorzenionym mogą być zorientowane w kierunku od korzenia (drzewo zstępujące) lub do korzenia (drzewo wstępujące).

3.10. Grafy dwudzielne

Graf dwudzielny (ang. *bipartite graph*) jest to graf prosty nieskierowany $G = (V, E)$, którego zbiór wierzchołków V można podzielić na dwa rozłączne zbiory V_1 i V_2 tak, że krawędzie nie łączą wierzchołków tego samego zbioru.

3740193878(23)

Niech $|V_1| = r$ i $|V_2| = s$, czyli $|V| = r + s$. Jeżeli pomiędzy wszystkimi parami wierzchołków należących do różnych zbiorów istnieje krawędź, to taki graf nazywamy *pełnym grafem dwudzielnym* $K_{r,s}$. Liczba krawędzi dla $K_{r,s}$ wynosi $|E| = rs$.

Twierdzenie: Niech $G = (V, E)$ będzie grafem dwudzielnym i niech $V = V_1 \cup V_2$ będzie podziałem wierchołków G. Jeżeli G ma cykl Hamiltona, to $|V_1| = |V_2|$. Jeżeli G ma ścieżkę Hamiltona, to $||V_1| - |V_2|| \leqslant 1$. Dla grafów dwudzielnych pełnych zachodzą też implikacje w lewo [6].

3.11. Skojarzenia

Skojarzeniem (ang. *matching*) w grafie nieskierowanym $G = (V, E)$ nazywamy podzbiór krawędzi M, taki że krawędzie z M są parami rozłączne (nie mają wspólnych wierzchołków). *Najliczniejszym skojarzeniem* (ang. *maximum-cardinality matching*) nazywamy każde skojarzenie o maksymalnej liczności. Problem znalezienia najliczniejszego skojarzenia jest w ogólności dość trudny, często rozwarza się go w odniesieniu do grafów dwudzielnych. W algorytmach korzysta się często z pojęcia ścieżki powiększającej (ang. *augmenting path*). Jest to ścieżka zawierająca na przemian krawędzie należące i nie należące do skojarzenia, przy czym ścieżka zaczyna się i kończy wierzchołkami nie pokrytymi w danym skojarzeniu.

Twierdzenie Berge'a o ścieżkach powiększających: Skojarzenie M jest maksymalne, gdy nie istnieje względem niego żadna ścieżka powiększająca.

4. Implementacja multigrafów

Na ogół uwagi o multigrafach dotyczą też grafów, chyba że osobno wspominamy o grafach.

4.1. Obiekty związane z multigrafami

Wierzchołek: W najprostszej sytuacji wierzchołki są liczbami całkowitymi od 0 do $n-1$, gdzie n jest liczbą wierzchołków multigrafu. W implementacjach słownikowych wierzchołki zwykle są stringami jednowierszowymi, ale w ogólności mają być obiektami hashowalnymi (jako klucze w słownikach) z pewnym porządkiem umożliwiającym sortowanie.

Krawędź: Krawędzie są instancjami klasy Edge(s, t, weight=1), przy czym należy podać wierzchołek początkowy s, wierzchołek końcowy t, ewentualnie wagę w. Dla krawędzi edge mamy dostęp do wierzchołków za pomocą atrybutów edge.source i edge.target. Krawędzie są skierowane, przy czym grafy nieskierowane traktują Edge(s, t) i Edge(t, s) jako tę samą krawędź. Krawędzie są hashowalne i można je porównywać.

Multigraf, graf: Multigrafy są instancjami klasy MultiGraph(n), gdzie n jest liczbą wierzchołków. Grafy proste są instancjami klasy Graph(n). Interfejs obu klas jest wspólny, choć inna jest interpretacja atrybutu G.weight(edge). W implementacjach słownikowych parametr n jest ignorowany i służy do zapewnienia kompatybilności z implementacjami macierzowymi. Opcjonalnym parametrem klas jest directed=True dla multigrafów skierowanych, oraz directed=False dla multigrafów nieskierowanych. Domyślnie tworzone są obiekty nieskierowane (brak parametru directed).

Ścieżka, cykl: Ścieżki i cykle są listami Pythona zawierającymi kolejne wierzchołki.

Drzewo swobodne (nieskierowane): Drzewo swobodne jest grafem prostym nieskierowanym, więc do jego przechowywania można wykorzystać obie klasy: MultiGraph i Graph.

Drzewo ukorzenione (skierowane): Drzewo ukorzenione powstaje naturalnie w wielu algorytmach, np. w algorytmach przeszukiwania multigrafów. Wygodnym sposobem przedstawienia takiego drzewa jest słownik, gdzie kluczami są dzieci, a wartościami ich rodzic. Rodzicem korzenia jest wartość

1522004738(25)

None. Innym sposobem przechowywania drzewa ukorzenionego jest klasa dla grafu prostego skierowanego.

Las: Las drzew swobodnych można przedstawić jako graf prosty nieskierowany, a las drzew ukorzenionych jako słownik z wieloma korzeniami (lub graf prosty skierowany).

Skojarzenie: Z definicji skojarzenie możemy reprezentować jako zbiór krawędzi, przy czym dla krawędzi Edge(s, t) powinien dla jednoznaczności zachodzić warunek $s < t$ (reprezentant krawędzi nieskierowanej).

Inna wydajna reprezentacja to słownik pair, zawierający jako klucze wszystkie wierzchołki grafu. Jeżeli wierzchołek v nie jest pokryty przez skojarzenie, to pair[v] = None. Jeżeli krawędź Edge(s, t) należy do skojarzenia, to ustawiamy pair[s] = t oraz pair[t] = s. Wykorzystanie pamięci wynosi wtedy $O(V)$.

4.2. Interfejs multigrafów

Dla operacji na multigrafach i grafach prostych zaproponowano interfejs przestawiony w tabeli 4.1.

4.3. Sposoby reprezentacji grafów

W pracy przygotowano kilka reprezentacji grafów.

— Moduł graphs1 - lista list z bool.
— Moduł graphs2 - lista z listami sąsiedztwa.
— Moduł graphs3 - słownik ze zbiorami sąsiadów.
— Moduł graphs4 - słownik z listami sąsiadów.
— Moduł graphs5 - słownik słowników z wagami.
— Moduł graphs6 - lista list z wagami.

Tabela 4.1. Interfejs multigrafów i grafów prostych. G i T są multigrafami, n jest liczbą całkowitą dodatnią.

Operacja	**Znaczenie**	**Metoda**
G = MultiGraph(n)	multigraf nieskierowany	__init__
G = MultiGraph(n, directed=True)	multigraf skierowany	__init__
G.is_directed()	czy jest skierowany	is_directed
G.v()	liczba wierzchołków	v
G.e()	liczba krawędzi	e
G.add_node(node)	dodanie wierzchołka	add_node
G.del_node(node)	usunięcie wierzchołka	del_node
G.has_node(node)	czy istnieje wierzchołek	has_node
G.add_edge(edge)	dodanie krawędzi	add_edge
G.del_edge(edge)	usunięcie krawędzi	del_edge
G.has_edge(edge)	czy istnieje krawędź	has_edge
G.weight(edge)	liczba krawędzi równoległych	weight
G.degree(node)	stopień wierzchołka (G nieskierowany)	degree
G.indegree(node)	stopień wejściowy wierzchołka	indegree
G.outdegree(node)	stopień wyjściowy wierzchołka	outdegree
G.iternodes()	iterator wierzchołków	iternodes
G.iteredges()	iterator krawędzi	iteredges
G.iteradjacent(node)	iterator wierzchołków sąsiednich	iteradjacent
G.iteroutedges(node)	iterator krawędzi wychodzących	iteroutedges
G.iterinedges(node)	iterator krawędzi przychodzących	iterinedges
G.show()	wyświetlanie małych multigrafów	show
G.copy()	zwraca kopię multigrafu	copy
G.transpose()	zwraca multigraf transponowany	transpose
G == T	porównywanie multigrafów	__eq__
G != T	porównywanie multigrafów	__ne__
G.add_graph(T)	dodawanie multigrafów	add_graph

1714846265(27)

5. Algorytmy i ich implementacje

Istnieje duża część problemów informatycznych dla których naturalnym sposobem reprezentacji są grafy, a metody rozwiązywania tych problemów są opisywane przez algorytmy grafowe. W tym rozdziale przedstawimy wybrane algorytmy dla grafów bez wag. Dla każdego algorytmu zamieszczamy jego implementację w języku Python.

5.1. Przeszukiwanie grafów

Dwa podstawowe algorytmy przeszukiwania grafów to przeszukiwanie wszerz (ang. *breadth-first search*, BFS) i przeszukiwanie w głąb (ang. *depth-first search*, DFS). Pewne algorytmy są modyfikacjami właśnie tych dwóch algorytmów.

5.1.1. Przeszukiwanie wszerz (BFS)

Algorytm przeszkuiwania wszerz jest jednym z podstawowych algorytmów operujących na grafach. Odwiedza wierzchołki grafu w taki sposób, że najpierw zostają odwiedzone wierzchołki o miejszej liczbie krawędzi na ścieżce łączacej je z wierzchołkiem źródłowym. Algorytm przeszukiwania wszerz jest często wykorzystywany w innych algorytmach, dlatego jego zrozumienie jest bardzo ważne. Działa on zarówno dla grafów skierowanych jak i nieskierowanych.

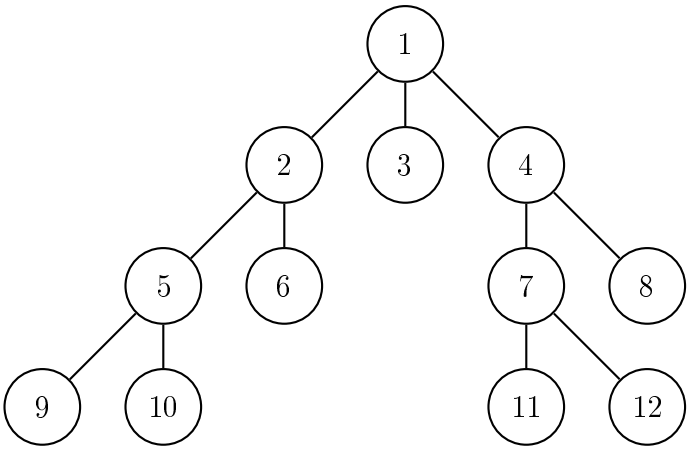
Dane wejściowe: Dowolny multigraf, opcjonalnie wierzchołek początkowy.

Problem: Przeszukiwanie grafu wszerz.

Opis algorytmu: Algorytm rozpoczynamy od dowolnego wierzchołka grafu. Na początku wszystkie wierzchołki grafu oznaczamy jako nieodwiedzone. Następnie początkowy wierzchołek wstawiamy do kolejki przechowującej wierzchołki, które jeszcze należy odwiedzić. Pobieramy wierzchołki z początku kolejki tak długo, aż kolejka nie będzie pusta. Dla każdego pobranego wierzchołka oznaczamy go jako odwiedzony oraz przeglądamy jego listę sąsiadów, a sąsiadów jeszcze nieodwiedzonych wstawiamy do kolejki.

Jeżeli na początku podano wierzchołek początkowy, to algorytm zbada spójną składową zawierającą ten wierzchołek. W przeciwnym wypadku algorytm będzie przeszukiwać kolejne spójne składowe, wybierając losowy wierzchołek z nowej składowej spójnej.

5990632195(28)



Rysunek 5.1. Kolejność odwiedzania wierzchołków w algorytmie przeszukiwania wszerz.

Złożoność: Złożoność czasowa algorytmu jest uzależniona jest od sposobu reprezentacji grafu. Jeśli graf jest reprezentowany za pomocą macierzy sąsiedztwa, to złożoność czasowa wynosi $O(V^2)$, ponieważ tyle czasu zajmuje przeglądanie sąsiadów. Natomiast przy reprezentacji grafu za pomocą list sąsiedztwa złożoność czasowa wynosi $O(V + E)$.

Złożoność pamięciowa algorytmu wynosi $O(V)$, ponieważ w pesymistycznym przypadku musimy przechowywać w kolejce wszystkie wierzchołki jednocześnie.

Uwagi: Dla algorytmu przeszukiwania wszerz wydajniejszą implementacją jest lista sąsiedztwa.

Listing 5.1. Moduł bfs.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# Python Algorithms: Mastering Basic Algorithms in
# the Python Language
# Copyright (C) 2010 by Magnus Lie Hetland
#
# http://pl.wikipedia.org/wiki/Przeszukiwanie_wszerz

from collections import deque
from main.structures.edges import Edge

class Bfs(object):
    """Algorytm przeszukiwania grafu wszerz."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktory bedzie
        przeszukiwany."""
        self.graph = graph
        self.previous = dict()
```


8489096839(29)

```
    def run(self, source=None, pre_action=None, post_action=None):
        """Metoda uruchamiajaca algorytm przeszkiwania wszerz.
        Opcjonalnymi argumentami metody jest wierzcholek, od
        ktorego zostanie rozpoczete przeszukiwanie oraz pre i post
        akcja wywolywana dla kazdego wierzcholka. Wynik zapisywany
        jest w zmiennej self.previous."""
        if source is not None:  # z source sprawdzamy jeden
            self._visit(source, pre_action, post_action)
        else:  # bez source sprawdzamy wszystkie
            for source in self.graph.iternodes():
                if source not in self.previous:
                    self._visit(source, pre_action, post_action)

    def _visit(self, source, pre_action, post_action):
        """Metoda odwiedzajaca kazdy wierzcholek. Jesli
        graf nie jest spojny zostanie wywolana kilka razy."""
        queue = deque()
        self.previous[source] = None
        queue.append(source)
        if pre_action:  # po queue.append
            pre_action(source)
        while queue:
            parent = queue.popleft()
            for child in self.graph.iteradjacent(parent):
                if child not in self.previous:
                    self.previous[child] = parent
                    queue.append(child)
                    if pre_action:  # po queue.append
                        pre_action(child)
            if post_action:  # akcja przy opuszczaniu parent
                post_action(parent)

    def to_dag(self):
        """Metoda zwracajaca skierowany acykliczny graf dwudzielny
        na podstawie listy poprzednikow."""
        dag = self.graph.__class__(self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            if self.previous[node] is not None:
                dag.add_edge(Edge(self.previous[node], node))
        return dag
```

5.1.2. Przeszukiwanie w głąb (DFS)

Algorytm przeszkuiwania w głąb jest obok algorytmu przeszukiwania wszerz jednym z podstawowych algorytmów operujących na grafach. Odwiedza wierzchołki grafu w taki sposób, że zanim wróci do wierzchołka, z którego dany wierzchołek został odwiedzony, musi przeszukać wszystkie sąsiadujące z nim wierzchołki. Algorytm przeszukiwania w głąb jest często wykorzystywa-

3737140139(30)

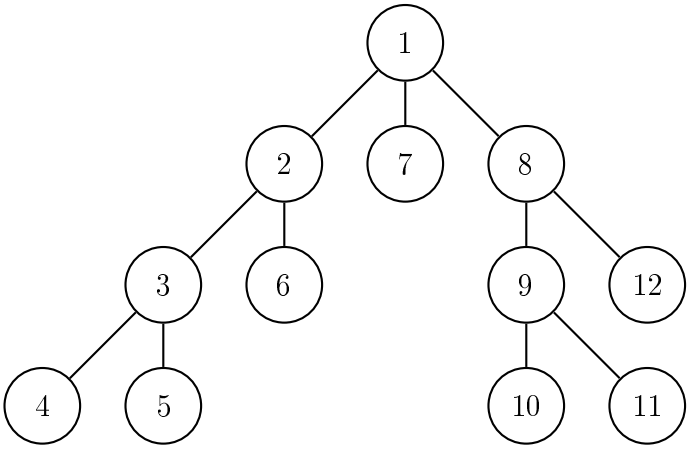
ny w innych algorytmach, dlatego jego zrozumienie jest bardzo ważne. Działa on zarówno dla grafów skierowanych jak i nieskierowanych.

Dane wejściowe: Dowolny multigraf, opcjonalnie wierzchołek początkowy.

Problem: Przeszukiwanie grafu w głąb.

Opis algorytmu: Algorytm rozpoczynamy od dowolnego wierzchołka grafu. Na początku wszystkie wierzchołki grafu oznaczamy jako nieodwiedzone. Następnie przechodzimy do początkowego wierzchołka i sprawdzamy, czy posiada jeszcze nieodwiedzonych sąsiadów. Jeśli posiada to przechodzimy to pierwszego nieodwiedzonego sąsiada i powtarzamy całą procedurę. Natomiast jeśli wszyscy sąsiedzi pewnego wierzchołka t są zbadani, albo nie ma on sąsiadów, algorytm wraca do wierzchołka s, z którego t został odwiedzony.

Jeżeli na początku podano wierzchołek początkowy, to algorytm zbada spójną składową zawierającą ten wierzchołek. W przeciwnym wypadku algorytm będzie przeszukiwać kolejne spójne składowe, wybierając losowy wierzchołek z nowej składowej spójnej.



Rysunek 5.2. Kolejność odwiedzania wierzchołków w algorytmie przeszukiwania w głąb.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V + E)$ w reprezentacji list sąsiedztwa, na co składa się początkowa inicjalizacja i przechodzenie po wszystkich sąsiadach każdego wierzchołka. W reprezentacji macierzowej złożoność wynosi natomiast $O(V^2)$.

Złożoność pamięciowa algorytmu wynosi $O(V)$, ponieważ w pesymistycznym przypadku, kiedy graf jest ścieżką zapamiętujemy poprzednika każdego wierzchołka.

Uwagi: Dla algorytmu przeszukiwania w głąb wydajniejszą implementacją jest lista sąsiedztwa.

Listing 5.2. Moduł dfs.

```
#!/usr/bin/python
```


2858555977(31)

```
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# Python Algorithms: Mastering Basic Algorithms in
# the Python Language
# Copyright (C) 2010 by Magnus Lie Hetland
#
# http://pl.wikipedia.org/wiki/Przeszukiwanie_w_g%C5%82%C4%85b

from collections import deque
from main.structures.edges import Edge

class DfsIterative(object):
    """Iteracyjny algorytm przeszukiwania grafu w glab."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktory bedzie
        przeszukiwany."""
        self.graph = graph
        self.previous = dict()

    def run(self, source=None, pre_action=None, post_action=None):
        """Metoda uruchamiajaca algorytm przeszkiwania w glab.
        Opcjonalnymi argumentami metody jest wierzcholek, od
        ktorego zostanie rozpoczete przeszukiwanie, oraz pre i post
        akcja wywolywana dla kazdego wierzcholka. Wynik zapisywany
        jest w zmiennej self.previous."""
        if source is not None:  # z source sprawdzamy jeden
            self._visit(source, pre_action, post_action)
        else:  # bez source sprawdzamy wszystkie
            for node in self.graph.iternodes():
                if node not in self.previous:
                    self._visit(node, pre_action, post_action)

    def _visit(self, source, pre_action, post_action):
        """Metoda odwiedzajaca kazdy wierzcholek. Jesli
        graf nie jest spojny zostanie wywolana kilka razy."""
        stack = deque()
        self.previous[source] = None
        stack.append(source)
        if pre_action:  # po stack.append
            pre_action(source)
        while stack:
            parent = stack.pop()
            for child in self.graph.iteradjacent(parent):
                if child not in self.previous:
                    self.previous[child] = parent
                    stack.append(child)
                    if pre_action:  # po stack.append
                        pre_action(child)
            if post_action:  # akcja przy opuszczaniu parent
                post_action(parent)

    def to_dag(self):
        """Metoda zwracajaca skierowany acykliczny graf dwudzielny
        na podstawie listy poprzednikow."""
```


3021218606(32)

```
        dag = self.graph.__class__(self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            if self.previous[node] is not None:
                dag.add_edge(Edge(self.previous[node], node))
        return dag

class DfsRecursive(object):
    """Rekurencyjny algorytm przeszukiwania grafu w glab."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktory bedzie
        przeszukiwany."""
        self.graph = graph
        self.previous = dict()

    def run(self, source=None, pre_action=None, post_action=None):
        """Metoda uruchamiajaca algorytm przeszkiwania w glab.
        Opcjonalnymi argumentami metody jest wierzcholek, od
        ktorego zostanie rozpoczete przeszukiwanie oraz pre i post
        akcja wywolywana dla kazdego wierzcholka. Wynik zapisywany
        jest w zmiennej self.previous."""
        if source is not None:
            self.previous[source] = None  # before visit
            self._visit(source, pre_action, post_action)
        else:
            for source in self.graph.iternodes():
                if source not in self.previous:
                    self.previous[source] = None  # before visit
                    self._visit(source, pre_action, post_action)

    def _visit(self, source, pre_action, post_action):
        """Metoda rekurencyjnie odwiedzajaca kazdy wierzcholek."""
        if pre_action:  # akcja przy wejsciu do source
            pre_action(source)
        for child in self.graph.iteradjacent(source):
            if child not in self.previous:
                self.previous[child] = source  # before visit
                self._visit(child, pre_action, post_action)
        if post_action:  # akcja przy opuszczaniu source
            post_action(source)

    def to_dag(self):
        """Metoda zwracajaca skierowany acykliczny graf dwudzielny
        na podstawie listy poprzednikow."""
        dag = self.graph.__class__(self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            if self.previous[node] is not None:
                dag.add_edge(Edge(self.previous[node], node))
        return dag
```

5.2. Sortowanie topologiczne

W rozwiązywaniu problemów informatycznych często zachodzi potrzeba posortowania pewnych obiektów wedle liniowego porządku. Przykładem może być potrzeba posortowania zadań powiązanych ze sobą w taki sposób, żeby

2915443750(33)

najpierw zostało wykonane to, które nie zależy od żadnego innego, a dopiero później te zadania, które uzależniają swoje wykonanie od innych. Z rozwiązaniem tego problemu przychodzi właśnie algorytm sortowania topologicznego. Wierzchołkami grafu w tym przykładzie są zadania do wykoniania, a krawędzie skierowane to powiązania pomiędzy tymi zadaniami. Procedura sortowania topologicznego może wykorzystywać do swojego działania zarówno algorytm przeszukiwania w głąb, albo być wykonana za pomocą usuwania wierzchołków niezależnych grafu, czyli takich, do których nie prowadzi żadna krawędź. Dla danego grafu może istnieć kilka różnych poprawnych rozwiązań problemu sortowania topologicznego.

5.2.1. Sortowanie topologiczne na bazie DFS

Jedną z możliwości napisania algorytmu sortowania topologicznego jest wykorzystanie algorytmu przeszukiwania w głąb. W tym celu wystarczy użyć klasycznego algorytmu przeszukiwania w głąb i podczas wychodzenia z przetworzonych wierzchołków umieszczać je na na początku listy.

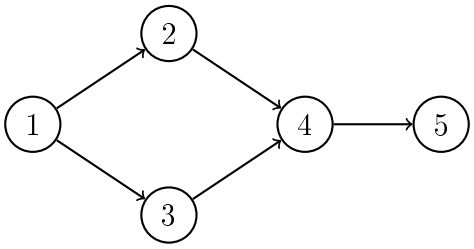
Dane wejściowe: Acykliczny graf skierowany (dag).

Problem: Sortowanie topologiczne wierzchołków dagu.

Opis algorytmu: Algorytm bazuje na algorytmie przeszukiwania grafu w głąb. Dodatkowo podczas opuszczania odwiedzanych wierzchołków każdy z nich dodajemy na początek listy, zawierającej posortowane topologicznie wierzchołki.

Rozpoczynamy od wierzchołka początkowego. Następnie dodajemy go na początek listy, która przechowuje posortowane topologicznie wierzchołki i przechodzimy do dowolnego nieodwiedzonego jeszcze wierzchołka sąsiadującego z danym. Procedurę tą powtarzamy tak długo, aż dany wierzchołek nie posiada nieodwiedzonych sąsiadów. Jeśli odwiedziliśmy wszystkie wierzchołki sąsiadujące z danym wierzchołkiem, to wracamy do wierzchołka z którego ten wierzchołek został odwiedzony. Algorytm kończy się w momencie, w którym nie ma już żadnych nieodwiedzonych wierzchołków. Otrzymaną lista zawiera wierzchołki posortowane topologicznie.

Przykład: Wynik działania algorytmu dla grafu poniżej to lista wierzchołków w kolejności [1, 2, 3, 4, 5] lub [1, 3, 2, 4, 5] (dwa rozwiązania).



Złożoność: Złożoność czasowa algorytmu wynosi tyle samo, co dla algorytmu DFS, czyli $O(V + E)$ dla reprezentacji list sąsiedztwa lub $O(V^2)$ dla reprezentacji macierzowej.

2433109112(34)

Złożoność pamięciowa algorytmu wynosi $O(V)$, czyli również tyle samo co dla algorytmu DFS.

Listing 5.3. Moduł topsort_dfs.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# Wprowadzenie do algorytmow
# Cormen Thomas H., Leiserson Charles E.,
# Rivest Ronald, Stein Clifford
#
# http://en.wikipedia.org/wiki/Topological_sorting

from main.algorithms.dfs import DfsRecursive

class TopologicalSortUsingDfs(object):
    """Algorytm sortowania topologicznego wykorzystujacy
    algorytm dfs."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktorego
        wierzcholki zostana przeszukane."""
        self.graph = graph
        self.order = list()

    def run(self):
        """Metoda uruchamiajaca algorytm sortowania
        topologicznego. Wynik zostanie zapisany w
        zmiennej self.order."""
        DfsRecursive(self.graph).run(
            post_action=lambda node: self.order.append(node))
        self.order.reverse()
```

5.2.2. Sortowanie topologiczne na bazie usuwania wierzchołków niezależnych

Inną możliwością zaimplementowania sortowania topologicznego jest stopniowe usuwanie wierzchołków, które nie posiadają krawędzi wchodzących wraz z krawędziami, które z nich wychodzą.

Dane wejściowe: Acykliczny graf skierowany (dag).

Problem: Sortowanie topologiczne wierzchołków dagu.

Opis algorytmu: Rozpoczynamy od stworzenia zbioru A wszystkich wierzchołków o stopniu wchodzącym 0, czyli takich, do których nie prowadzi żadna krawędź. Następnie wyciągamy dowolny wierzchołek ze zbioru A, dodajemy go do listy wierzchołków posortowanych topologicznie i przechodzimy po jego

6880059843(35)

sąsiadach, usuwając krawędzie ich łączące z grafu. Jeśli do sąsiada nie prowadzi żadna krawędź, to dodajemy ten wierzchołek do zbioru A. Algorytm powtarzamy, aż w zbiór A stanie się pusty.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V+E)$ dla reprezentacji list sąsiedztwa lub $O(V^2)$ dla reprezentacji macierzowej.

Listing 5.4. Moduł topsort_edges.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# Wprowadzenie do algorytmow
# Cormen Thomas H., Leiserson Charles E.,
# Rivest Ronald, Stein Clifford
#
# http://en.wikipedia.org/wiki/Topological_sorting

class TopologicalSortThroughErasingEdges(object):
    """Algorytm sortowania topologicznego wykorzystujacy
    usuwanie wierzcholkow niezaleznych."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktorego
        wierzcholki zostana przeszukane."""
        self.graph = graph
        self.order = list()

    def run(self):
        """Metoda uruchamiajaca algorytm sortowania
        topologicznego. Wynik zostanie zapisany w
        zmiennej self.order."""
        inedges = dict((node, 0) for node in self.graph.iternodes())
        Q = set()
        for edge in self.graph.iteredges():
            inedges[edge.target] += 1
        for node in self.graph.iternodes():
            if inedges[node] == 0:
                Q.add(node)
        while Q:
            source = Q.pop()
            self.order.append(source)
            # usuwanie wszystkich zewnetrznych krawedzi
            for target in self.graph.iteradjacent(source):
                inedges[target] -= 1
                if inedges[target] == 0:
                    Q.add(target)
```

5.3. Spójność

Definicje związane ze spójnością grafów zostały podane w rozdziale 3.6. Dla grafu nieskierowanego wyznacza się składowe spójne. Można to zrobić

5878398478(36)

łatwo i szybko, więc nie warto rezygnować ze sprawdzenia spójności nieznanego grafu, szczególnie w algorytmach wymagających spójnego grafu. Do znajdowania składowych spójnych można wykorzystać BFS lub DFS, gdzie sukcesywnie będziemy eksplorować kolejne składowe. Jeżeli struktura grafu ma się zmieniać dynamicznie, to lepszym rozwiązaniem jest skorzystanie ze struktury danych zbiorów rozłącznych.

Dla grafów skierowanych wyznacza się silnie spójne składowe, co jest klasycznym zastosowaniem DFS [1].

5.3.1. Składowe spójne w grafach nieskierowanych

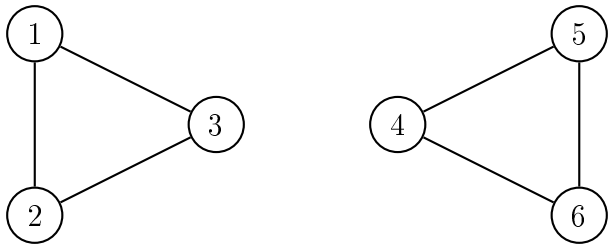
Algorytm znajdowania składowych spójnych grafu jest kolejnym algorytmem, który opiera się na algorytmie DFS badź BFS. Wykorzytujemy w nim fakt, że algorytm DFS i BFS odwiedzają wszystkie wierzchołki w składowej spójnej grafu.

Dane wejściowe: Graf nieskierowany G.

Problem: Wyznaczenie składowych spójnych G.

Opis algorytmu: Dla dowolnego wierzchołka uruchamiamy algorytm BFS lub DFS, a odwiedzone wierzchołki zaznaczamy jako należące do jednej składowej spójnej. Jeżeli pozostaną jeszcze jakieś nieodwiedzone wierzchołki grafu, to czynności powtarzamy, zaznaczając przynależność wierzchołków do innej składowej spójnej.

Przykład: Wynik działania algorytmu dla grafu poniżej są dwa zbiory wierzchołków [1, 2, 3] oraz [4, 5, 6].



Złożoność: Złożoność czasowa wynosi tyle samo, co dla algorytmu DFS.

Listing 5.5. Moduł cc.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# Wprowadzenie do algorytmow
# Cormen Thomas H., Leiserson Charles E.,
# Rivest Ronald, Stein Clifford

from main.algorithms.bfs import Bfs
```


1798132156(37)

```
from main.algorithms.dfs import DfsIterative

class ConnectedComponents(object):
    """Algorytm wyszukiwania spojnych skladowych grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem."""
        if graph.is_directed():
            raise ValueError("Graf jest skierowany.")
        self.graph = graph
        self.connected_components = []

    def run(self):
        """Metoda uruchamiajaca algorytm. Wynik przechowywany
        jest w zmiennje self.connected_components."""
        visited = set()
        dfs = DfsIterative(self.graph)
        for source in self.graph.iternodes():
            if source in visited:
                continue
            component = set()
            dfs.run(source, pre_action=lambda node: component.add(
                node))
            self.connected_components.append(component)
            visited.update(component)

class ConnectedComponentsBFS:
    """Algorytm wyszukiwania spojnych skladowych grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem."""
        if graph.is_directed():
            raise ValueError("Graf jest skierowany.")
        self.graph = graph
        self.connected_components = dict(
            (node, None) for node in self.graph.iternodes())
        self.n_cc = 0

    def run(self):
        """Metoda uruchamiajaca algorytm. Wynik przechowywany
        jest w zmiennje self.connected_components."""
        algorithm = Bfs(self.graph)
        for source in self.graph.iternodes():
            if self.connected_components[source] is None:
                algorithm.run(source, pre_action=lambda
                    node: self.connected_components.__setitem__(
                    node, self.n_cc))
                self.n_cc += 1

class ConnectedComponentsDFS:
    """Algorytm wyszukiwania spojnych skladowych grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem."""
        if graph.is_directed():
            raise ValueError("Graf jest skierowany.")
        self.graph = graph
```


1566154460(38)

```
        self.connected_components = dict(
            (node, None) for node in self.graph.iternodes())
        self.n_cc = 0

    def run(self):
        """Metoda uruchamiajaca algorytm. Wynik przechowywany
        jest w zmiennje self.connected_components."""
        dfs = DfsIterative(self.graph)
        for source in self.graph.iternodes():
            if self.connected_components[source] is None:
                dfs.run(source, pre_action=lambda
                    node: self.connected_components.__setitem__(
                    node, self.n_cc))
                self.n_cc += 1
```

5.3.2. Silnie spójne składowe w grafach skierowanych

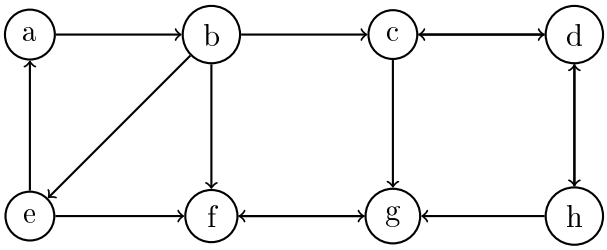
Algorytm wyszukiwania silnie spójnych składowych opiera się na podwójnym zastosowaniu algorytmu DFS dla przekazanego grafu oraz dla grafu transponowanego w stosunku do niego.

Dane wejściowe: Graf skierowany G.

Problem: Wyznaczenie silnie spójnych składowych G.

Opis algorytmu: Na początku wykonujemy algorytm DFS w celu wyznaczenia czasów przetworzenia wierzchołków grafu $G = (V, E)$. Następnie wykonujemy algorytm DFS dla grafu transponowanego $G^T = (V, E^T)$ (krawędzie mają odwrócone kierunki), przy czym rozważamy wierzchołki w kolejności malejących czasów przetworzenia z pierwszego etapu. Otrzymane drzewa przeszukiwania w głąb reprezentują szukane silnie spójne składowe.

Przykład: Wynik działania algorytmu dla grafu poniżej są trzy zbiory wierzchołków [a, b, c], [f, g] oraz [c, d, h].



Złożoność: Złożoność czasowa algorytmu wynosi $O(V+E)$ dla reprezentacji list sąsiedztwa, ponieważ dwa razy wykonywany jest algorytm DFS.

Listing 5.6. Moduł scc.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
```


3009245838(39)

```
#
# Wprowadzenie do algorytmow
# Cormen Thomas H., Leiserson Charles E.,
# Rivest Ronald, Stein Clifford

from main.algorithms.dfs import DfsRecursive

class StronglyConnectedComponents(object):
    """Algorytm wyszukiwania silnie spojnych skladowych
    dla grafu skierowanego grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem."""
        self.graph = graph
        self.scc = []

    def run(self):
        """Metoda uruchamiajaca algorytm. Wynik przechowywany
        jest w zmiennje self.scc."""
        processing_order = []
        dfs = DfsRecursive(self.graph)
        # uzycie post_action
        dfs.run(
            post_action=lambda node: processing_order.append(node))
        processing_order.reverse()
        dfs = DfsRecursive(self.graph.transpose())
        visited = set()
        for source in processing_order:
            if source in visited:
                continue
            component = set()
            dfs.run(source,
                    pre_action=lambda node: component.add(node))
            self.scc.append(component)
            visited.update(component)

class StronglyConnectedComponents2:
    """Algorytm wyszukiwania silnie spojnych skladowych
    dla grafu skierowanego grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem."""
        if not graph.is_directed():
            raise ValueError("graph is not directed")
        self.graph = graph
        self.scc = dict(
            (node, None) for node in self.graph.iternodes())
        self.n_scc = 0

    def run(self):
        """Metoda uruchamiajaca algorytm. Wynik przechowywany
        jest w zmiennje self.scc."""
        dfs = DfsRecursive(self.graph)
        processing_order = []
        # uzycie post_action
        dfs.run(
            post_action=lambda node: processing_order.append(node))
```


2206568130(40)

```
        processing_order.reverse()
        dfs = DfsRecursive(self.graph.transpose())
        for source in processing_order:
            if self.scc[source] is None:
                dfs.run(source, pre_action=lambda
                    node: self.scc.__setitem__(node, self.n_scc))
                self.n_scc += 1
```

5.4. Skojarzenia w grafach dwudzielnych

Pojęcie grafu dwudzielnego zostało zdefiniowane w rozdziale 3.10, dotyczącym teorii grafów. Jednym z problemów związanych z dwudzielnością grafów jest odpowiedź na pytanie, czy dany graf jest grafem dwudzielnym. W rozdziale tym zostanie omówiony algorytm sprawdzania dwudzielności grafu, oparty na algorytmie przeszukiwania wszerz, oraz trzy algorytmy znajdowania największego skojarzenia w grafie dwudzielnym. Skojarzenia zdefiniowano w rozdziale 3.11.

5.4.1. Sprawdzanie dwudzielności grafu na bazie BFS

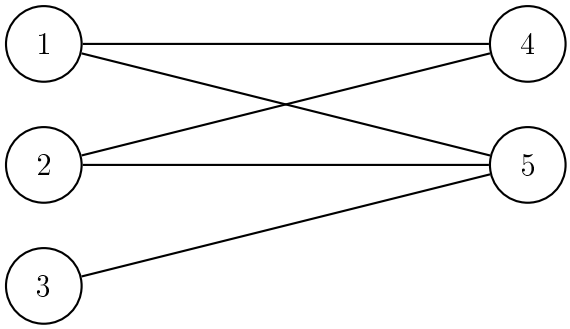
Sprawdzanie dwudzielności grafu za pomocą algorytmu przeszukiwania wszerz polega na stopniowym przechodzeniu wierzchołków i nadawaniu im kolorów niebieskiego albo czerwonego w taki sposób, żeby dwa sąsiednie wierzchołki nie miały tego samego koloru.

Dane wejściowe: Graf skierowany lub nieskierowany G.

Problem: Sprawdzenie, czy graf G jest grafem dwudzielnym, oraz pokolorowanie wierzchołków w przypadku grafu dwudzielnego.

Opis algorytmu: Algorytm bazuje na algorytmie przeszukiwania grafu wszerz BFS. Podczas wykonywania algorytmu BFS przypisujemy każdemu wierzchołkowi kolor czerwony (liczba 1) lub niebieski (liczba -1), w zależności od tego, jaki kolor miał wierzchołek prowadzący do danego. Kolor zostaje nadany tak, aby nie istniała krawędź składająca się z wierzchołków tego samego koloru. Jeśli w grafie pojawi się taka krawędź, to graf nie jest grafem dwudzielnym. Wynik działania algorytmu zapisywany jest w zmiennej logicznej `is_bipartite`.

1914030660(41)



Rysunek 5.3. Przykład grafu dwudzielnego.

Złożoność: Złożoność pamięciowa algorytmu wynosi $O(V)$, ponieważ zapamiętujemy tylko kolory odwiedzanych wierzchołków w słowniku color. Złożoność czasowa wynosi tyle samo co dla algorytmu BFS.

Uwagi: Zamiast algorytmu BFS można wykorzystać algorytm DFS. Algorytm działa prawidłowo w przypadku grafów spójnych i niespójnych.

Listing 5.7. Moduł bipartite.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://edu.i-lo.tarnow.pl/inf/utils/002_roz/ol022.php

from collections import deque

class Bipartite(object):
    """Algorytm badajacy dwudzielnosc grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, na ktorym
        bedzie sprawdzana dwudzielnosc."""
        self.graph = graph
        self.is_bipartite = True
        self.color = dict(
            ((node, None) for node in self.graph.iternodes()))

    def run(self):
        """Metoda uruchamiajaca algorytm sprawdzajacy
        dwudzielnosc grafu. Wynik zapisywany jest w
        zmiennej self.is_bipartite."""
        for source in self.graph.iternodes():
            if self.color[source] is None:
                self._visit(source)

    def _visit(self, source):
        """Metoda odwiedzajaca kazdy wierzcholek i
        nadajaca kolory poszczegolnym wierzcholkom."""
        queue = deque()
        self.color[source] = 1
        queue.append(source)
```


1670112784(42)

```
        while queue:
            parent = queue.popleft()
            for child in self.graph.iteradjacent(parent):
                if self.color[child] is None:
                    self.color[child] = -self.color[parent]
                    queue.append(child)
                else:  # child byl odwiedzony
                    if self.color[parent] == self.color[child]:
                        self.is_bipartite = False
                        return
```

5.4.2. Maksymalne skojarzenie w grafie dwudzielnym - metoda Forda-Fulkersona

Sposób wyszukiwania maksymalnego skojarzenia metodą Forda-Fulkersona opiera się na stworzeniu sieci przepływowej, gdzie waga krawędzi to jeden i wyszukaniu maksymalnego przepływu w tej sieci.

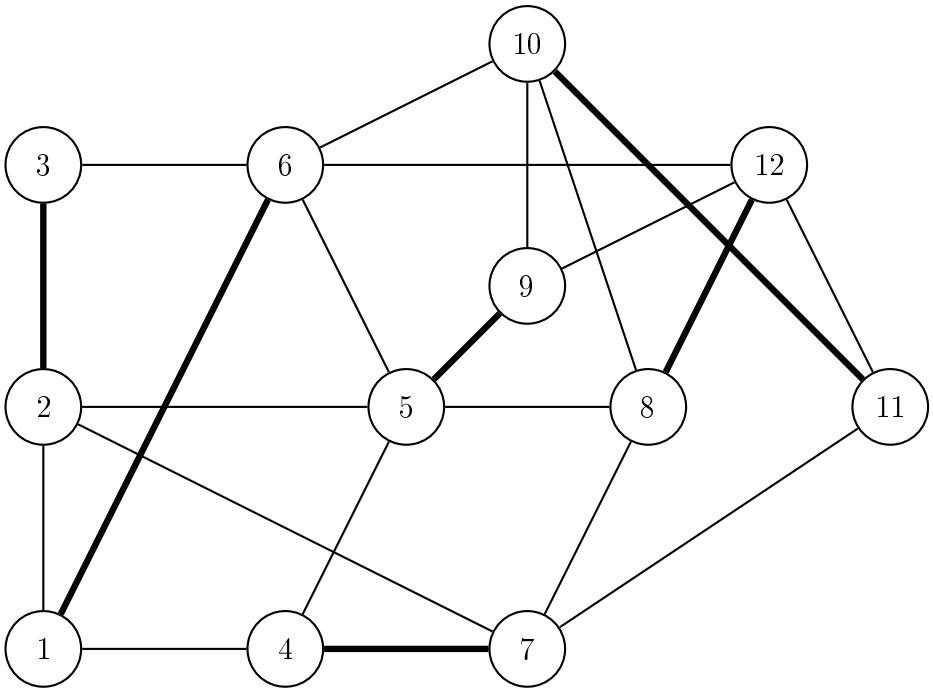
Dane wejściowe: Spójny nieskierowany graf dwudzielny G.

Problem: Znalezienie najliczniejszego skojarzenia w G.

Opis algorytmu: Algorytm stosuje metodę Forda-Fulkersona. Konstruujemy sieć przepływową (graf skierowany) $G' = (V', E')$ przez dodanie źródła s i ujścia t, $V' = V \cup \{s, t\}$. Niech $V = V_1 \cup V_2$ będzie podziałem grafu dwudzielnego G. Zbiór krawędzi skierowanych budujemy z trzech zbiorów: $E' = \{(s, u) : u \in V_1\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in V_2\}$.
Każdej krawędzi z E' przypisujemy jednostkową przepustowość. Maksymalny przepływ znajdujemy metodą Forda-Fulkersona. Przepływ w sieci G' odpowiada najliczniejszemu skojarzeniu w G.

Przykład: Wynik działania algorytmu dla grafu poniżej jest zbiór krawędzi (1,6), (2,3), (4,7), (5,9), (8,12), (10,11).

2221402386(43)



Rysunek 5.4. Maksymalne skojarzenie w grafie.

Złożoność: Złożoność czasowa algorytmu wynosi $O(VE)$.

Listing 5.8. Moduł matching_ff.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# Wprowadzenie do algorytmow
# Cormen Thomas H., Leiserson Charles E.,
# Rivest Ronald, Stein Clifford

from main.structures.edges import Edge
from bipartite import Bipartite
from fordfulkerson import FordFulkerson

class MatchingFordFulkerson:
    """Algorytm znajdujacy maksymalne skojarzenie w grafie
    dwudzielnym za pomoca algorytmu Forda Fulkersona."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, w ktory bedzie
        szukane maksymalne skojarzenie."""
        self.graph = graph
        self.pair = dict(
            (node, None) for node in self.graph.iternodes())
        self.cardinality = 0
        algorithm = Bipartite(self.graph)
        algorithm.run()
        self.v1 = set()
        self.v2 = set()
        for node in self.graph.iternodes():
```


5687531086(44)

```
            if algorithm.color[node] == 1:
                self.v1.add(node)
            else:
                self.v2.add(node)

    def run(self):
        """Metoda uruchamiajaca algorytm szukania maksymalnego
        skojarzenia. Wynik zapisywany jest w zmiennej
        self.cardinality oraz self.pair."""
        size = self.graph.v()
        # tworzenie sieci przeplywowej
        network = self.graph.__class__(size + 2, directed=True)
        self.source = size
        self.sink = size + 1
        network.add_node(self.source)
        network.add_node(self.sink)
        for node in self.graph.iternodes():
            network.add_node(node)
        for node in self.v1:
            network.add_edge(Edge(self.source, node))
        for edge in self.graph.iteredges():
            if edge.source in self.v1:
                network.add_edge(Edge(edge.source, edge.target))
            else:
                network.add_edge(Edge(edge.target, edge.source))
        for node in self.v2:
            network.add_edge(Edge(node, self.sink))
        algorithm = FordFulkerson(network)
        algorithm.run(self.source, self.sink)
        for source in self.v1:
            for target in self.v2:
                if algorithm.flow[source][target] == 1:
                    self.pair[source] = target
                    self.pair[target] = source
        self.cardinality = algorithm.max_flow
```

5.4.3. Maksymalne skojarzenie w grafie dwudzielnym - ścieżka powiększająca

Innym podejściem do szukania maksymalnego skojarzenia w grafie dwudzielnym jest metoda ścieżki powiększającej. Metoda ścieżki powiększającej polega na stopniowym powiększaniu skojarzenia przez wyszukiwanie tzw. ścieżki powiększającej 3.11.

Dane wejściowe: Spójny nieskierowany graf dwudzielny G.

Problem: Znalezienie najliczniejszego skojarzenia w G.

Opis algorytmu: Algorytm bazuje na wyszukiwania ścieżki powiększającej. Czynność szukania ścieżki powiększającej powtarzamy aż do momentu, kiedy nie uda się znaleźć żadnej ścieżki. Wtedy kończymy działanie algorytmu.

Złożoność: Złożoność czasowa wynosi $O(VE)$.

2767407954(45)

Listing 5.9. Moduł matching_ap.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://wazniak.mimuw.edu.pl/index.php?title=
# Zaawansowane_algorytmy_i_struktury_danych/Wyklad_7

from bipartite import Bipartite
from main.structures.edges import Edge

class MatchingUsingAugmentingPath(object):
    """Algorytm znajdujacy maksymalne skojarzenie w grafie
    dwudzielnym. Algorytm wykorzystuje sciezke powiekszajaca."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, w ktory bedzie
        szukane maksymalne skojarzenie."""
        self.graph = graph
        self.pair = dict(
            (node, None) for node in self.graph.iternodes())
        self.cardinality = 0
        algorithm = Bipartite(self.graph)
        algorithm.run()
        self.v1 = set()
        self.v2 = set()
        for node in self.graph.iternodes():
            if algorithm.color[node] == 1:
                self.v1.add(node)
            else:
                self.v2.add(node)

    def run(self):
        """Metoda uruchamiajaca algorytm szukania maksymalnego
        skojarzenia. Wynik zapisywany jest w zmiennej
        self.cardinality oraz self.pair."""
        while True:
            augmenting_path = self._find_augmenting_path()
            if augmenting_path:
                self.pair.update(self._fill_pairs())
                self.cardinality += 1
            else:
                break

    def _find_augmenting_path(self):
        """Metoda szukajaca sciezki powiekszajacej."""
        self.v1free = [vertex for vertex in self.v1 if
                       self.pair[vertex] is None]
        self.v2free = [vertex for vertex in self.v2 if
                       self.pair[vertex] is None]
        for v1 in self.v1free:
            self.visited_e = list()
            self.visited_v = list()
            last_v = self._visite(v1)
            if last_v in self.v2free:
                return True
```


3241325199(46)

```
        return False

    def _visite(self, parent):
        """Metoda oparta na dfs'ie szukajaca maksymalnego
        skojarzenia."""
        self.visited_v.append(parent)
        children = self.graph.iteradjacent(parent)
        for child in children:
            if Edge(parent, child) not in self.visited_e:
                if self._is_entering(child, parent) or \
                        self._is_outgoing(child, parent):
                    self.visited_e.append(Edge(parent, child))
                    self.visited_e.append(Edge(child, parent))
                    size = len(self.visited_v)
                    last = self._visite(child)
                    if last in self.v2free:
                        return last
                    else:
                        self.visited_v = self.visited_v[:size]
        return parent

    def _is_entering(self, child, parent):
        return child in self.v2 and self.pair[child] != parent

    def _is_outgoing(self, child, parent):
        return child in self.v1 and self.pair[child] == parent

    def _fill_pairs(self):
        """Metoda zwracajaca nowe skojarzenie."""
        path = self._remove_cycles(self.visited_v)
        return self._create_pairs(path)

    @classmethod
    def _create_pairs(cls, path):
        """Tworzenie wierzcholkow nalezacych do skojarzenia."""
        pair = dict()
        i = 0
        while i < len(path):
            first = path[i]
            second = path[i + 1]
            pair[first] = second
            pair[second] = first
            i += 2
        return pair

    @classmethod
    def _remove_cycles(cls, path):
        """Usuwanie cykli ze sciezki."""
        new_path = list()
        for item in path:
            if path.count(item) > 1:
                d = path[::-1]
                size = len(path)
                i = size - d.index(item) - 1
                new_path.extend(path[i:])
                return cls._remove_cycles(new_path)
            else:
```


2687162853(47)

```
                new_path.append(item)
        return new_path
```

5.4.4. Maksymalne skojarzenie w grafie dwudzielnym - algorytm Hopcrofta-Karpa

Najmniej złożonym czasowo algorytmem wyszukiwania maksymalnego skojarzenia w grafie dwudzielnym jest algorytm Hopcrofta-Karpa. Algorytm ten także wykorzystuje ścieżki powiększające 3.11 ale w celu przyspieszenia nie szuka ich pojedyńczo, tylko wyszukuje kilka na raz.

Dane wejściowe: Spójny nieskierowany graf dwudzielny G.

Problem: Znalezienie najliczniejszego skojarzenia w G.

Opis algorytmu: Algorytm w pętli zwiększa rozmiar skojarzenia przez znajdowanie ścieżek powiększających. Zamiast poszukiwać pojedyńczych ścieżek, algorytm znajduje maksymalny zbiór najkrótszych ścieżek.

Złożoność: Złożoność czasowa wynosi $O(\sqrt{V}E)$.

Listing 5.10. Moduł matching_hk.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://wazniak.mimuw.edu.pl/index.php?title=
# Zaawansowane_algorytmy_i_struktury_danych/Wyklad_7
#
# http://en.wikipedia.org/wiki/Hopcroft%E2%80%93Karp_algorithm

from collections import deque
from bipartite import Bipartite

class HopcroftKarp:
    """Algorytm Hopcroft-Karp znajdujacy maksymalne
    skojarzenie w grafie dwudzielnym."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, w ktory bedzie
        szukane maksymalne skojarzenie."""
        self.graph = graph
        self.pair = dict((node, None) for node in self.graph.iternodes())
        self.dist = dict()
        self.cardinality = 0
        algorithm = Bipartite(self.graph)
        algorithm.run()
        self.v1 = set()
        self.v2 = set()
        for node in self.graph.iternodes():
            if algorithm.color[node] == 1:
                self.v1.add(node)
```


1558544567(48)

```
            else:
                self.v2.add(node)
        self.queue = deque()    # for nodes from self.v1

    def run(self):
        """Metoda uruchamiajaca algorytm szukania maksymalnego
        skojarzenia. Wynik zapisywany jest w zmiennej
        self.cardinality oraz self.pair."""
        while self._bfs_stage():
            for node in self.v1:
                if self.pair[node] is None and self._dfs_stage(node):
                    self.cardinality = self.cardinality + 1
                    #print self.pair

    def _bfs_stage(self):
        """Metoda uruchamiajaca etap bfs algorytmu."""
        for node in self.v1:
            if self.pair[node] is None:
                self.dist[node] = 0
                self.queue.append(node)
            else:
                self.dist[node] = float("inf")
        self.dist[None] = float("inf")
        while self.queue:
            node = self.queue.popleft()
            if self.dist[node] < self.dist[None]:
                for target in self.graph.iteradjacent(node):
                    if self.dist[self.pair[target]] == float("inf"):
                        self.dist[self.pair[target]] = self.dist[node] + 1
                        self.queue.append(self.pair[target])
        return self.dist[None] != float("inf")

    def _dfs_stage(self, node):
        """Metoda uruchamiajaca etap dfs algorytmu."""
        if node is not None:
            for target in self.graph.iteradjacent(node):
                if self.dist[self.pair[target]] == self.dist[node] + 1:
                    if self._dfs_stage(self.pair[target]):
                        self.pair[target] = node
                        self.pair[node] = target
                        return True
            self.dist[node] = float("inf")
            return False
        return True
```

5.5. Domknięcie przechodnie

Mamy dany graf skierowany $G = (V, E)$. *Domknięciem przechodnim* (ang. *transitive closure*) grafu G nazywamy graf skierowany $G^+ = (V, E^+)$, w którym E^+ jest takim zbiorem krawędzi (s, t), że istnieje ścieżka z wierzchołka s do wierzchołka t w grafie G. Inaczej mówiąc, chcemy zbudować strukturę danych, za pomocą której szybko będzie można odpowiedzieć na pytanie dotyczące osiągalności (czy można jakoś dotrzeć z s do t). Takie problemy

3061584962(49)

pojawiają się np. w bazach danych, arkuszach kalkulacyjnych (aktualizacja komórek po zmianie danych).

Domknięcie przechodnie grafu zawsze istnieje. Ogólnie domknięcie przechodnie określa się dla relacji binarnej w pewnym zbiorze. Jednym z algorytmów pozwalających na wyznaczenie domknięcia przechodniego jest algorytm bliski algorytmowi Floyda-Warshalla. Dla grafów rzadkich lepszy może okazać się algorytm oparty o BFS lub DFS. W naszych implementacjach dozwolone są również grafy nieskierowane, dla których przyjmujemy, że istnieją krawędzie skierowane w obu kierunkach.

5.5.1. Domknięcie przechodnie na bazie algorytmu Floyda-Warshalla

Dane wejściowe: Graf skierowany lub nieskierowany.

Problem: Wyznaczenie domknięcia przechodniego grafu.

Opis algorytmu: Algorytm bazuje na algorytmie Floyda-Warshalla wyznaczania ścieżek pomiędzy wszystkimi parami wierzchołków. Jednak operacje arytmetyczne są zamienione na operacje logiczne, przez co na pewnych komputerach dostajemy większą szybkość i zmniejszone wymagania pamięciowe. W wyniku działania algorytmu otrzymujemy macierz T, gdzie T[s][t]=True, jeżeli istnieje ścieżka z s do t. W przeciwnym wypadku dostajemy wartość False.

Złożoność: Złożoność czasowa wynosi $O(V^3)$, tak samo jak w algorytmie Floyda-Warshalla.

Listing 5.11. Moduł closure.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://pl.wikipedia.org/wiki/Algorytm_Floyda-Warshalla

class TransitiveClosure(object):
    """Algorytm wyszukujacy zamkniecie przechodnie grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, na ktory bedzie
        wywolywany algorytm."""
        self.graph = graph
        self.connections = dict()
        for source in self.graph.iternodes():
            self.connections[source] = dict()
            for target in self.graph.iternodes():
                self.connections[source][target] = False
            self.connections[source][source] = True
        for edge in self.graph.iteredges():
            self.connections[edge.source][edge.target] = True
```


1518358447(50)

```
            if not self.graph.is_directed():
                self.connections[edge.target][edge.source] = True

    def run(self):
        """Metoda uruchomiajaca algorytm. Wynik algorytmu
        przechowywany jest w zmiennej self.connections."""
        for node in self.graph.iternodes():
            for source in self.graph.iternodes():
                for target in self.graph.iternodes():
                    if not self.connections[source][target]:
                        self.connections[source][target] = \
                            self.connections[source][node] \
                            and self.connections[node][target]
```

5.5.2. Domknięcie przechodnie na bazie BFS

Dane wejściowe: Graf skierowany lub nieskierowany.

Problem: Wyznaczenie domknięcia przechodniego grafu.

Opis algorytmu: Dla każdego wierzchołka s grafu uruchamiamy algorytm BFS, a przy każdym napotkanym wierzchołku t zaznaczamy istnienie połączenia od s do t.

Złożoność: Złożoność czasowa wynosi w reprezentacji list sąsiedztwa wynosi $O(V^2+VE)$, ponieważ dla każdego wierzchołka wykonujemy algorytm BFS o złożoności $O(V+E)$. W reprezentacji macierzowej złożoność wyniesie $O(V^3)$.

Listing 5.12. Moduł closure_bfs.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://pl.wikipedia.org/wiki/Algorytm_Floyda-Warshalla

from main.algorithms.bfs import Bfs

class TransitiveClosureBfs(object):
    """Algorytm wyszukujacy zamkniecie przechodnie grafu
    na bazie BFS."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, na ktory bedzie
        wywolywany algorytm."""
        self.graph = graph
        self.connections = dict()
        for source in self.graph.iternodes():
            self.connections[source] = dict()
            for target in self.graph.iternodes():
                self.connections[source][target] = False
            self.connections[source][source] = True
        for edge in self.graph.iteredges():
```


2671754076(51)

```
            self.connections[edge.source][edge.target] = True
            if not self.graph.is_directed():
                self.connections[edge.target][edge.source] = True

    def run(self):
        """Metoda uruchomiajaca algorytm. Wynik algorytmu
        przechowywany jest w zmiennej self.connections."""
        for source in self.graph.iternodes():
            algorithm = Bfs(self.graph)
            algorithm.run(source, pre_action=lambda node:
            self.connections[source].__setitem__(node, True))
```

5.6. Grafy eulerowskie

Pojęcie grafu eulerowskiego zostało wyjaśnione w rodziale 3.8, dotyczącym teorii grafów. Z grafami eulerowskim związana jest również pewna klasa algorytmów, które pozwalają na wyszukanie ścieżki Eulera lub cyklu Eulera.

Jednym z głównych problemów związanych z grafami eulerowskim jest tzw. *problem chińskiego listonosza* (ang. *Chinese Postman Problem*, CPP). Listonosz roznosząc listy musi przejść przez wszystkie ulice w swojej dzielnicy co najmniej jeden raz i wrócić na pocztę. Ponieważ jest człowiekiem leniwym (nie odnosi się to do pozostałych listonoszy, tylko do tego konkretnego), chciałby mieć jak najkrótszą do przejścia trasę. W grafach eulerowskich problem ten sprowadza się do wyznaczenia cyklu Eulera, gdzie krawędzie to ulice jakimi przemieszcza się listonosz, a wierzchołki to skrzyżowania tych ulic. W tym rodziale zostaną omówione dwa algorytmy znajdowania cyklu Eulera oraz jeden algorytm znajdowania ścieżki Eulera.

5.6.1. Algorytm Fleury'ego

Algorytm Fleury'ego pozwala znaleźć cykl Eulera w grafie eulerowskim. Jego zaletą jest prosty zapis oraz intuicyjność. Natomiast problematyczną kwestią algorytmu jest implementacja testu sprawdzającego, czy krawędź jest mostem, czyli krawędzią rozspajającą graf w przypadku jej usunięcia.

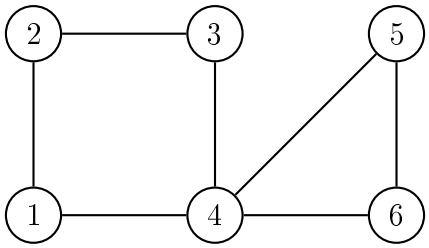
Dane wejściowe: Graf eulerowski.

Problem: Znalezienie cyklu Eulera w grafie eulerowskim.

Opis algorytmu: Pierwszym krokiem jest wybranie wierzchołka początkowego, z którego zamierzamy rozpocząć znajdowanie cyklu. Następnie wybieramy dowolną krawędź wychodzącą z tego wierzchołka pamiętając, że most wybieramy w ostateczności. Zapamiętujemy, że dana krawędź została odwiedzona, bądź usuwamy ją z grafu i przechodzimy do następnego wierzchołka. Następnie powtarzamy całą procedurę aż przejdziemy do wierzchołka, który nie będzie miał już żadnych krawędzi do odwiedzenia.

Przykład: Dla przedstawionego poniżej grafu, cyklem Eulera jest ścieżka przechodząca przez wierzchołki w następującej kolejności: 1, 2, 3, 4, 5, 6, 4, 1.

2654667643(52)



Złożoność: Złożoność czasowa w dużej mierze zależy od implementacji testu sprawdzającego, czy krawędź jest mostem. Samo przechodzenie i oznaczanie krawędzi jako odwiedzone ma złożoność liniową w stosunku do liczby krawędzi $O(E)$. Najszybsza implementacja algorytmu opisana przez Mikkel'a Thorup'a pozwala uzyskać złożoność $O(E \log^3 E \log \log E)$. Natomiast złożoność algorytmu przedstawionego w pracy wynosi $O(EV^2)$, ponieważ test sprawdzania, czy krawędź jest mostem, opiera się na algorytmie przeszukiwania w głąb, którego złożoność dla macierzy sąsiedztwa wynosi $O(V^2)$.

Uwagi: Największą uwagę należy zwrócić na implementację testu sprawdzającego czy krawędź jest mostem, ponieważ w dużej mierze ona wypływa na złożoność algorytmu. Do algorytmu można także dodać warunki sprawdzające, czy graf jest grafem eulerowskim 3.8, żeby nie wykonywać algorytmu w przypadku, kiedy wynik i tak będzie niepoprawny.

Listing 5.13. Moduł euler_fleury.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://www.austincc.edu/powens/+Topics/HTML/05-6/05-6.html

from main.algorithms.dfs import DfsIterative

class Fleury(object):
    """Algorytm wyszukujacy cykl Eulera w grafie."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem na, ktorym
        zostanie wywolany algorytm szukania cyklu Eulera."""
        self.graph = graph

    def run(self, source):
        """Metoda uruchamiajaca algorytm szukajacy
        cyklu Eulera. Wynik algorytmu, czyli sciezka
        jaka nalezy przejsc aby uzyskac cykl Eulera
        zapisywany jest w zmiannej self.current_trail."""
        node = source
        self.current_trail = [node]
        graph_copy = self.graph.copy()
        while list(graph_copy.iteradjacent(node)):
            for edge in list(graph_copy.iteroutedges(node)):
                if not self._is_bridge(edge, graph_copy):
                    break
```


7427137831(53)

```
            graph_copy.del_edge(edge)
            self.current_trail.append(edge.target)
            node = edge.target

    def _is_bridge(self, edge, graph):
        """Metoda sprawdza, czy podana krawedz w grafie
        przekazanym jako drugi argument jest mostem,
        czyli krawedzia rozpojniajaca graf w przypadku
        jej usuniecia."""
        dfs = DfsIterative(graph)
        list1 = list()
        list2 = list()
        dfs.run(edge.source,
                pre_action=lambda node: list1.append(node))
        graph.del_edge(edge)
        dfs.run(edge.source,
                pre_action=lambda node: list2.append(node))
        graph.add_edge(edge)
        return len(list1) != len(list2)
```

5.6.2. Algorytm znajdowania cyklu Eulera (ze stosem)

Kolejny algorytm pozwala na znajdowanie cyklu Eulera w grafie eulerowskim. Do swojego działania wykorzystuje stos, na którym przechowuje wierzchołki. Dzięki wykorzystaniu stosu można pominąć problematyczny aspekt szukania mostu dla algorytmu Fleury'ego 5.6.1. Chociaż zaletą algorytmu Fleury'ego była prostota zapisu i intuicyjność, która umożliwiała łatwe zrozumienie algorytmu, to algorytm ze stosem jest łatwiejszy do implemetacji. Algorytm wykorzystuje do swojego działania oprócz stosu również algorytm przeglądania w głąb, który został omówiony wcześniejszym rodziale 5.1.2.

Dane wejściowe: Graf eulerowski G.

Problem: Znalezienie cyklu Eulera dla grafu G.

Opis algorytmu: Algorytm rozpoczynamy od wybrania dowolnego wierzchołka początkowego, z którego zamierzamy rozpocząć znajdowanie cyklu. Następnie, jeśli istnieją nieodwiedzone dotąd krawędzie incydentne z bieżącym wierzchołkiem, to wybieramy dowolną z nich, przechodzimy do kolejnego wierzchołka i odkładamy aktualny na stos. W przeciwnym razie, jeśli żadne krawędzie nieodwiedzone nie istnieją, to przenosimy wierzchołek do rozwiązania i zdejmujemy go ze stosu. Następnie przechodzimy do wierzchołka, który jest na szczycie stosu i powtarzamy całą procedurem dla niego. Algorytm powtarzamy dotąd, aż stos będzie pusty.

Złożoność: Złożoność czasowa uzależniona jest od liczby krawędzi oraz wierzchołków w grafie, ponieważ zarówno każdą krawędź jak i wierzchołek musimy dowiedzić i wynosi ona $O(V + E)$.

Uwagi: Tak samo jak dla poprzedniego algorytmu znajdowania cyklu Eulera ważnym elementem przed przystąpieniem do głównej części algorytmu jest

1641492666(54)

sprawdzenie warunków koniecznych, które graf musi spełniać aby posiadał taki cykl.

Listing 5.14. Moduł euler_cycle.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://www.algorytm.org/algorytmy-grafowe/cykl-eulera.html

from euler_path import EulerPath

class EulerCycle(object):
    """Algorytm wyszukujacy cykl Eulera w grafie."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem na, ktorym
        zostanie wywolany algorytm szukania cyklu Eulera."""
        self.graph = graph
        self.cycle = []

    def run(self, source):
        """Metoda uruchamiajaca algorytm szukajacy
        cyklu Eulera. Wynik algorytmu, czyli sciezka
        jaka nalezy przejsc aby uzyskac cykl Eulera
        zapisywany jest w zmiannej self.cycle."""
        if self.graph.is_directed():
            if not self._check_directed_condition():
                return
        else:
            if not self._check_undirected_condition():
                return
        euler_path = EulerPath(self.graph)
        euler_path.run(source)
        if euler_path.path[0] == euler_path.path[-1]:
            self.cycle = euler_path.path

    def _check_directed_condition(self):
        """Warunek istnienia cyklu Eulera dla grafu skierowanego."""
        for source in self.graph.iternodes():
            if len(list(self.graph.iteroutedges(source))) != len(
                    list(self.graph.iterinedges(source))):
                return False
        return True

    def _check_undirected_condition(self):
        """Warunek istnienia cyklu Eulera dla grafu nieskierowanego."""
        for source in self.graph.iternodes():
            if len(list(self.graph.iteradjacent(source))) % 2 != 0:
                return False
        return True
```


2290273169(55)

5.6.3. Algorytm znajdowania ścieżki Eulera

Następnym zagadnieniem związanym z grafami eulerowskimi jest szukanie ścieżki Eulera. Ścieżka Eulera różni się od cyklu tym, że wierzchołek początkowy, z którego rozpoczynamy wyszukiwanie nie musi zgadzać się z wierzchołkiem końcowym, jak miało to miejsce w przypadku cyklu Eulera. Należy również pamiętać, że cykl Eulera także jest szczególnym przypadkiem ścieżki Eulera. Problem ten więc zawiera się w problemie znajdowania cyklu Eulera, dlatego też do rozwiązania go można wykorzystać dowolny algorytm znajdujący cykl Eulera opisany w tym rozdziale.

Dane wejściowe: Graf eulerowski G.

Problem: Znalezienie ścieżki Eulera dla grafu G.

Opis algorytmu: Do znajdowania cyklu wykorzystujemy algorytm Fleury'ego, bądź algorytm znajdowania cyklu Eulera ze stosem.

Złożoność: Złożoność czasowa zależy od wybranego algorytmu.

Uwagi: Tak samo jak dla poprzednich algorytmów znajdowania cyklu Eulera ważnym elementem przed przystąpieniem do głównej części algorytmu jest sprawdzenie warunków koniecznych, które graf musi spełniać, aby posiadał ścieżkę Eulera. Warunki dla wyszukiwania ścieżki są mniej restrykcyjne niż dla cyklu, ale sprawdzenie ich i tak pozwala zaoszczędzić czas, jeśli wejściowy graf ich nie spełnia.

Listing 5.15. Moduł euler_path.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0135.php

class EulerPath(object):
    """Algorytm wyszukujacy sciezke Eulera w grafie."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem na, ktorym
        zostanie wywolany algorytm szukania sciezki Eulera."""
        self.graph = graph
        self.path = []
        self.visited = set()

    def run(self, source):
        """Metoda uruchamiajaca algorytm szukajacy
        sciezki Eulera. Wynik algorytmu, czyli sciezka
        jaka nalezy przejsc aby uzyskac sciezke Eulera
        zapisywany jest w zmiannej self.path."""
        self._visit(source)

    def _visit(self, source):
```


7653804201(56)

```
        """Metoda odwiedzajaca kazda krawedz grafu spojnego."""
        stack = [source]
        while stack:
            parent = stack[-1]
            edges = list(self.graph.iteroutedges(parent))
            for edge in edges:
                if edge not in self.visited:
                    self.visited.add(edge)
                    if not self.graph.is_directed():
                        self.visited.add(~edge)
                    stack.append(edge.target)
                    break
                elif edge == edges[-1]:
                    self.path.append(stack.pop())
        self.path.reverse()
```

5.7. Grafy hamiltonowskie

Definicja grafu hamiltonowskiego została omówiona w rodziale dotyczącym teorii grafów 3.7. Z grafami hamiltonowskimi wiąże się również wiele zagadnień algorytmicznych. Jednym z badanych problemów jest znajdowanie ścieżki lub cyklu Hamiltona. Problem ten jest analogiczny jak dla grafów eulerowskich z tym, że odnosi się do wierzchołków a nie do krawędzi, oraz jego rozwiązanie jest znacznie trudniejsze. Należy więc znaleźć taką ścieżkę, która odwiedza wszystkie wierzchołki grafu tylko raz i w zależności czy rozpatrujemy cykl czy ścieżkę, wierzchołek końcowy musi być taki sam jak początkowy lub nie. Problem znajdowania ścieżki Hamiltona jest problemem NP-zupełnym.

Problem NP-zupełny, czyli problem niedeterministyczny wielomianowy zupełny (ang. *nondeterministic polynomial*), jest to w teorii obliczeniowej problem decyzyjny, którego rozwiązanie można zweryfikować w czasie wielomianowym, oraz wszystkie problemy w klasie NP są redukowalne do niego w czasie wielomianowym. Do dzisiaj nie wiadomo, czy problemy NP-zupełne można rozwiązać w czasie wielomianowym, jest to więc problem otwarty.

5.7.1. Algorytm znajdowania cyklu Hamiltona

Przedstawiony przez ze mnie algorytm znajdowania cyklu Hamiltona oparty jest na rekurencyjnym wywoływaniu algorytmu DFS. Istnieją inne bardziej wydajne algorytmy znajowania cyklu Hamiltona, jeden z nich znaleźć pod adresem [7]. Na początku należy również sprawdzić warunki dla grafu hamiltonowskiego, ponieważ algorytm dla innych grafów będzie działał błędnie. Istnieją grafy, które zawsze są hamiltonowskie. Są to grafy pełne z conajm-

1225146750(57)

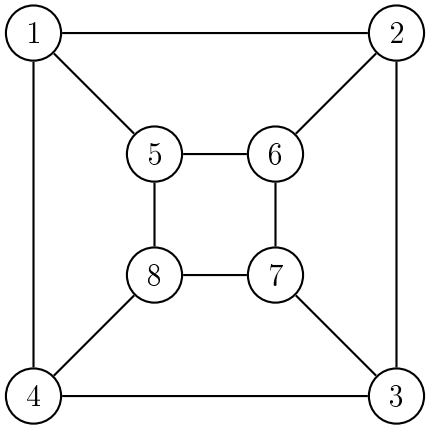
niej trzema wierzchołkami oraz grafy opisujące wielościany foremne. Dla tych grafów można pominąć sprawdzanie warunków początkowych w algorytmie.

Dane wejściowe: Graf hamiltonowski G.

Problem: Znalezienie cyklu Hamiltona dla grafu G.

Opis algorytmu: Algorytm startuje od dowolnego wierzchołka, który wstawiamy do kolejki. Następnie sprawdzamy, czy rozmiar kolejki to liczba wierzchołków grafu, jeśli tak to należy sprawdzić, czy ostatni wierzchołek ma krawędź łączącą go z wierzchołkiem początkowym. W przypadku kiedy istnieje taka krawędź oznacza to, że znaleźliśmy cykl Hamiltona i kończymy wykonywanie algorytmu. Natomiast w przeciwnym wypadku zaznaczamy wierzchołek jako odwiedzony i przechodzimy do dowolnego jego sąsiada, powtarzając całą procedurę algorytmu.

Przykład: Dla przedstawionego poniżej grafu (sześcian, 4-pryzma, kostka Q_3), cyklem Hamiltona jest ścieżka przechodząca przez wierzchołki w następującej kolejności: 1, 5, 6, 7, 8, 4, 3, 2, 1.



Złożoność: Algorytm w pesymistycznym przypadku ma złożoność $O(V!)$ co oznacza, że działa w czasie wykładniczym.

Uwagi: Algorytm działa tylko dla grafów hamiltonowskich. Należy więc sprawdzić, czy graf spełnia odpowiednie założenia. Należy również pamiętać, że algorytm ma dużą złożoność czasową, więc jego stosowanie dla grafów z powyżej dwudziestoma wierzchołkami nie jest zalecane.

Listing 5.16. Moduł hamilton.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu na podstawie zrodel:
#
# http://edu.i-lo.tarnow.pl/inf/utils/002_roz/ol025.php
```


2335825832(58)

```
from collections import deque

class HamiltonCircuit(object):
    """Algorytm wyszukujacy cykl Hamiltona w grafie."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, na
        ktorym zostanie wywolany algorytm szukania
        cyklu Hamiltona."""
        self.graph = graph
        self.current_trail = list()
        self.queue = deque()
        self.visited = set()

    def run(self, source):
        """Metoda uruchamiajaca algorytm szukajacy
        cyklu Hamiltona. Wynik algorytmu, czyli sciezka
        jaka nalezy przejsc aby uzyskac cykl Hamiltona
        zapisywany jest w zmiannej self.current_trail."""
        self.start = source
        self._hamilton_dfs(source)

    def _hamilton_dfs(self, source):
        """Metoda uruchamia rekurencyjny algorytm dfs
        z podanego wierzcholka."""
        if self.current_trail:
            return
        self.queue.append(source)
        if len(self.queue) == self.graph.v():
            if self.start in self.graph.iteradjacent(self.queue[-1]):
                self.current_trail = list(self.queue)
        else:
            self.visited.add(source)
            for node in self.graph.iteradjacent(source):
                if node not in self.visited:
                    self._hamilton_dfs(node)
            self.visited.discard(source)
        self.queue.pop()
```

5.8. Kolorowanie wierzchołków

Kolorowanie grafów (ang. *graph coloring*) jest bardzo ważnym zagadnieniem związanym z algorytmami grafowymi. Problem kolorowania grafu odnosi się zarówno do wierzchołków, jak i do krawędzi grafu. W przypadku wierzchołków mówimy o nadaniu kolorów wierzchołkom grafu w taki sposób, żeby dwa sąsiednie wierzchołki nie miały tego samego koloru. Minimalną liczbę kolorów użytą do pokolorowania grafu G nazywamy **liczbą chromatyczną** grafu i oznaczamy $\chi(G)$. Ze względu na szerokie zastosowanie optymalnego kolorowania grafu we współczesnej informatyce prowadzone są rozległe badania na ten temat. Problem kolorowania grafów należy do problemów NP-zupełnych, podobnie jak szukanie cyklu Hamiltona, dlatego też w praktycznych zastosowaniach używane są algorytmy przybliżone. Algoryt-

3658136763(59)

my przybliżone nie gwarantują znalezienia optymalnego rozwiązania, jednak działają w czasie wielomianowym.

Poniżej zostanie opisanych pięć algorytmów kolorowania wierzchołków grafu. Pierwszy algorytm jest algorytmem dokładnym, sprawdza wszystkie możliwości i wybiera optymalne rozwiązanie. Pozostałe algorytmy są algorytmami przybliżonymi i działają w czasie wielomianowym. W algorytmach przybliżonych wyróżnia się najmniejszy graf dość trudny i najmniejszy graf trudny. Najmniejszy graf dość trudny to graf, dla którego algorytm może znaleźć nieoptymalne rozwiązanie, natomiast najmniejszy graf trudny to graf, dla którego algorytm zawsze znajduje nieoptymalne rozwiązanie. Kolorowanie wierzchołków grafu było stosowane we wczesniejszym rozdziale do sprawdzania dwudzielności grafu 5.4. Jeśli wierzchołki grafu spójnego da się pokolorować dwoma kolorami, to jest on grafem dwudzielnym.

5.8.1. Dokładny algorytm kolorowania wierzchołków grafu

Dane wejściowe: Dowolny multigraf G.

Problem: Kolorowanie wierzchołków G.

Opis algorytmu: Algorytm składa się z dwóch faz. W pierwszej fazie generujemy kolejne kombinacje kolorów dla wierzchołków. Kombinacje kolorów najłatwiej generuje się za pomocą licznika. Metoda ta polega na stworzeniu licznika, którego cyfry reprezentują kolor danego wierzchołka. Ilość cyfr w liczniku to ilość wierzchołków. Na początku licznik może przyjmować tylko dwie cyfry zero i jeden. Jeśli wszystkie kombinacje z dwoma cyframi zostaną sprawdzone to sukcesywnie dodaje się kolejne cyfry. W kolejnej fazie sprawdzamy, czy kombinacja jest poprawna i żadne dwa wierzchołki połączone krawędzią nie mają tego samego koloru. Jeśli jest poprawna, to kończymy algorytm, w przeciwnym wypadku sprawdzamy kolejne kolorowanie wierzchołków.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(2^V)$, ponieważ opiera się na sprawdzaniu wszystkich kombinacji kolorowania wierzchołków.

Uwagi: Algorytm ma złożoność wykładniczą co sprawia, że nawet dla małej liczby wierzchołków jego wykonanie zajmuje dużo czasu.

Listing 5.17. Moduł coloring.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu sprawdzania dwudzielnosci grafu na
# podstawie artykulow ze stron:
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0142.php
# http://pl.wikipedia.org/wiki/Kolorowanie_grafu
# kaims.pl/~deren/ag/wyklady/14_kolorowanie.pdf

from heapq import heappush, heappop
```


2823201667(60)

```
class VerticesColoring(object):
    """Algorytm kolorowania grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktorego wierzcholkom
        zostana nadane kolory."""
        self.graph = graph

    def run(self):
        """Metoda uruchamiajaca algorytm kolorowania wierzcholkow
        grafu. Kolory to liczby calkowite od zera. Wynik
        zapisywany jest w zmiennej self.color."""
        self.color = dict(
            ((node, 0) for node in self.graph.iternodes()))
        self.number_of_color = 1
        is_colored = False
        while not is_colored:
            is_colored = True
            for source in self.graph.iternodes():
                for target in self.graph.iteradjacent(source):
                    if self.color[source] == self.color[target]:
                        is_colored = False
                        break
            if not is_colored:
                roll = False
                for node in self.color:
                    if self.color[node] == self.number_of_color - 1:
                        roll = True
                        self.color[node] = 0
                    else:
                        roll = False
                        self.color[node] += 1
                        break
                if roll:
                    self.number_of_color += 1
                    self.color = dict(((node, 0) for node in
                                       self.graph.iternodes()))
```

5.8.2. Przybliżony algorytm kolorowania wierzchołków grafu S (ang. *sequential*)

Dane wejściowe: Dowolny multigraf G.

Problem: Kolorowanie wierzchołków G.

Opis algorytmu: Uporządkowane w dowolny sposób wierzchołki grafu odwiedzamy kolejno nadając im kolory w sposób zachłanny, czyli wybieramy optymalne rozwiązanie lokalne dla danego wierzchołka. Sprawdzamy kolory sąsiadów wierzchołka, a następnie wybieramy pierwszy kolor, który nie został

2400072736(61)

jeszcze użyty. Algorytm kończy się, kiedy wszystkim wierzchołkom zostanie przydzielony kolor.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(E+V)$.

Uwagi: Algorytm nie zawsze znajduje optymalne rozwiązanie. Najmniejszym dość trudnym grafem jest graf P_4. Nie istnieją grafy trudne dla tego algorytmu.

Listing 5.18. Moduł coloring_seq.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu sprawdzania dwudzielnosci grafu na
# podstawie artykulow ze stron:
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0142.php
# http://pl.wikipedia.org/wiki/Kolorowanie_grafu
# kaims.pl/~deren/ag/wyklady/14_kolorowanie.pdf

class VerticesColoringSequential(object):
    """Algorytm kolorowania grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktorego wierzcholkom
        zostana nadane kolory."""
        self.graph = graph

    def run(self):
        """Metoda uruchamiajaca algorytm kolorowania wierzcholkow
        grafu. Kolory to liczby calkowite od zera. Wynik
        zapisywany jest w zmiennej self.color."""
        self.color = dict(
            ((node, None) for node in self.graph.iternodes()))
        for node in self.graph.iternodes():
            self.color[node] = self._get_color(node)

    def _get_color(self, source):
        """Metoda przydzielajaca kolor danemu wierzcholkowi."""
        occupied_colors = set()
        for target in self.graph.iteradjacent(source):
            if target in self.color:
                occupied_colors.add(self.color[target])
        for i in xrange(len(occupied_colors)):
            if i not in occupied_colors:
                return i
        return len(occupied_colors)
```


1100212442(62)

5.8.3. Przybliżony algorytm kolorowania wierzchołków grafu LF (ang. *largest first*)

Dane wejściowe: Dowolny multigraf G.

Problem: Kolorowanie wierzchołków G.

Opis algorytmu: Na początku należy uporządkować wierzchołki nierosnąco według ich stopni. Następnie tak uporządkowane wierzchołki grafu odwiedzamy kolejno nadając im kolory w sposób zachłanny, czyli wybieramy optymalne rozwiązanie lokalne dla danego wierzchołka. Sprawdzamy kolory sąsiadów wierzchołka, a następnie wybieramy pierwszy kolor, który nie został jeszcze użyty. Algorytm kończy się, kiedy wszystkim wierzchołką zostanie przydzielony kolor.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(E+V)$. Sortowanie wierzchołków użyte na początku algorytmu można wykonać w czasie liniowym za pomocą sortowania przez zliczanie (ang. *counting sort*).

Uwagi: Algorytm nie zawsze znajduje optymalne rozwiązanie. Najmniejszym dość trudnym grafem jest graf P_6. Najmniejszym grafem trudnym jest koperta. Do optymalnego kolorowania koperty potrzeba trzech kolorów, natomiast algorytm ten używa czterech.

Listing 5.19. Moduł coloring_lf.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu sprawdzania dwudzielnosci grafu na
# podstawie artykulow ze stron:
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0142.php
# http://pl.wikipedia.org/wiki/Kolorowanie_grafu
# kaims.pl/~deren/ag/wyklady/14_kolorowanie.pdf

from heapq import heappush, heappop

class VerticesColoringLF(object):
    """Algorytm kolorowania grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktorego wierzcholkom
        zostana nadane kolory."""
        self.graph = graph

    def run(self):
        """Metoda uruchamiajaca algorytm kolorowania wierzcholkow
        grafu. Kolory to liczby calkowite od zera. Wynik
        zapisywany jest w zmiennej self.color."""
        self.color = dict(
            ((node, None) for node in self.graph.iternodes()))
        sorted_vertices = self._sort_vertices()
        while sorted_vertices:
```


1297391968(63)

```
            node = heappop(sorted_vertices)[1]
            self.color[node] = self._get_color(node)

    def _sort_vertices(self):
        """Metoda sortujaca wierzcholki grafu malejaco wedlug ich
        stopni (liczby krawedzi z nich wychodzacych)."""
        priority_queue = []
        for node in self.graph.iternodes():
            heappush(priority_queue, (
            -len(list(self.graph.iteroutedges(node))), node))
        return priority_queue

    def _get_color(self, source):
        """Metoda przydzielajaca kolor danemu wierzcholkowi."""
        occupied_colors = set()
        for target in self.graph.iteradjacent(source):
            if target in self.color:
                occupied_colors.add(self.color[target])
        for i in xrange(len(occupied_colors)):
            if i not in occupied_colors:
                return i
        return len(occupied_colors)
```

5.8.4. Przybliżony algorytm kolorowania wierzchołków grafu SL (ang. *smallest last*)

Dane wejściowe: Dowolny multigraf G.

Problem: Kolorowanie wierzchołków G.

Opis algorytmu: Zaczynamy od znalezienia wierzchołka o minimalnym stopniu, a następnie usuwamy go z grafu. Czynność tą powtarzamy aż graf będzie pusty, a kolejność usuwania zapamiętujemy. Uporządkowane w ten sposób wierzchołki kolorujemy w sposób zachłanny, czyli wybieramy optymalne rozwiązanie lokalne dla danego wierzchołka. Sprawdzamy kolory sąsiadów wierzchołka, a następnie wybieramy pierwszy kolor, który nie został jeszcze użyty. Algorytm kończy się kiedy wszystkim wierzchołką zostanie przydzielony kolor.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(E+V)$.

Uwagi: Algorytm nie zawsze znajduje optymalne rozwiązanie. Najmniejszym dość trudnym rozwiązaniem jest "pryzma". Najmniejszym trudnym grafem jest "pryzmatoid". Algorytm stosunkowo dobrze sobie radzi, gdy kolorowanym grafem jest drzewo, koło, grafy jednocykliczne, grafy Mycielskiego lub grafy Johnsona.

Listing 5.20. Moduł coloring_lf.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
```


4915036831(64)

```
# Implementacja algorytmu sprawdzania dwudzielnosci grafu na
# podstawie artykulow ze stron:
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0142.php
# http://pl.wikipedia.org/wiki/Kolorowanie_grafu
# kaims.pl/~deren/ag/wyklady/14_kolorowanie.pdf

from heapq import heappush, heappop

class VerticesColoringLF(object):
    """Algorytm kolorowania grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktorego wierzcholkom
        zostana nadane kolory."""
        self.graph = graph

    def run(self):
        """Metoda uruchamiajaca algorytm kolorowania wierzcholkow
        grafu. Kolory to liczby calkowite od zera. Wynik
        zapisywany jest w zmiennej self.color."""
        self.color = dict(
            ((node, None) for node in self.graph.iternodes()))
        sorted_vertices = self._sort_vertices()
        while sorted_vertices:
            node = heappop(sorted_vertices)[1]
            self.color[node] = self._get_color(node)

    def _sort_vertices(self):
        """Metoda sortujaca wierzcholki grafu malejaco wedlug ich
        stopni (liczby krawedzi z nich wychodzacych)."""
        priority_queue = []
        for node in self.graph.iternodes():
            heappush(priority_queue, (
            -len(list(self.graph.iteroutedges(node))), node))
        return priority_queue

    def _get_color(self, source):
        """Metoda przydzielajaca kolor danemu wierzcholkowi."""
        occupied_colors = set()
        for target in self.graph.iteradjacent(source):
            if target in self.color:
                occupied_colors.add(self.color[target])
        for i in xrange(len(occupied_colors)):
            if i not in occupied_colors:
                return i
        return len(occupied_colors)
```

5.8.5. Przybliżony algorytm kolorowania wierzchołków grafu SLF (ang. *saturated largest first*)

Dane wejściowe: Dowolny multigraf G.

Problem: Kolorowanie wierzchołków G.

Opis algorytmu: Dopóki istnieją niepokolorowane wierzchołki w grafie, należy wybrać wierzchołek o maksymalnym stopniu z wierzchołków o maksymalnym stopniu nasycenia, gdzie stopień nasycenia wierzchołka to ilość różnych kolorów wierzchołków incydentalnych z nim. Wybrany wierzchołek kolorujemy w sposób zachłanny, czyli wybieramy optymalne rozwiązanie lokalne dla danego wierzchołka. Sprawdzamy kolory sąsiadów wierzchołka, a następnie wybieramy pierwszy kolor, który nie został jeszcze użyty. Algorytm kończy się kiedy wszystkim wierzchołkom zostanie przydzielony kolor.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(E \log V)$.

Uwagi: Algorytm nie zawsze znajduje optymalne rozwiązanie.

Listing 5.21. Moduł coloring_slf.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Implementacja algorytmu sprawdzania dwudzielnosci grafu na
# podstawie artykulow ze stron:
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0142.php
# http://pl.wikipedia.org/wiki/Kolorowanie_grafu
# kaims.pl/~deren/ag/wyklady/14_kolorowanie.pdf

from heapq import heappush, heappop

class VerticesColoringLF(object):
    """Algorytm kolorowania grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktorego wierzcholkom
        zostana nadane kolory."""
        self.graph = graph

    def run(self):
        """Metoda uruchamiajaca algorytm kolorowania wierzcholkow
        grafu. Kolory to liczby calkowite od zera. Wynik
        zapisywany jest w zmiennej self.color."""
        self.color = dict(
            ((node, None) for node in self.graph.iternodes()))
        sorted_vertices = self._sort_vertices()
        while sorted_vertices:
            node = heappop(sorted_vertices)[1]
            self.color[node] = self._get_color(node)

    def _sort_vertices(self):
        """Metoda sortujaca wierzcholki grafu malejaco wedlug ich
```


8575618095(66)

```
        stopni (liczby krawedzi z nich wychodzacych)."""
        priority_queue = []
        for node in self.graph.iternodes():
            heappush(priority_queue, (
            -len(list(self.graph.iteroutedges(node))), node))
        return priority_queue

    def _get_color(self, source):
        """Metoda przydzielajaca kolor danemu wierzcholkowi."""
        occupied_colors = set()
        for target in self.graph.iteradjacent(source):
            if target in self.color:
                occupied_colors.add(self.color[target])
        for i in xrange(len(occupied_colors)):
            if i not in occupied_colors:
                return i
        return len(occupied_colors)
```

5.9. Kolorowanie krawędzi

Kolorowaniem krawędzi grafu nazywamy takie nadanie kolorów krawędziom grafu, żeby żadne stykające się krawędzie nie miały tego samego koloru. Minimalną liczbą kolorów użytą do kolorowania krawędzi grafu nazywamy **indeksem chromatycznym** grafu i oznaczamy $\chi'(G)$.

Twierdzenie (Vizig, 1964): Jeżeli G jest grafem prostym, w którym największy stopień wierzchołka wynosi Δ, to $\Delta \leqslant \chi'(G) \leqslant \Delta + 1$.

5.9.1. Algorytm kolorowania krawędzi grafu

Dane wejściowe: Dowolny multigraf G.

Problem: Kolorowanie krawędzi G.

Opis algorytmu: Algorytm rozpoczynamy od stworzenia grafu krawędziowego dla wejściowego grafu. Graf krawędziowy tworzymy w taki sposób, że krawędzie z grafu wejściowego stają się wierzchołkami w grafie krawędziowym. Następnie na tak przygotowanym grafie wykonujemy algorytm kolorowania wierzchołków 5.8.1

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(2^V)$, ponieważ opiera się na sprawdzaniu wszystkich kombinacji kolorowania krawędzi.

Uwagi: Algorytm ma złożoność wykładniczą co sprawia, że nawet dla małej liczby krawędzi jego wykonanie zajmuje dużo czasu.

Listing 5.22. Moduł coloring_edges.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```


3837059460(67)

```
#
# Implementacja algorytmu sprawdzania dwudzielnosci grafu na
# podstawie artykulow ze stron:
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0142.php
# http://pl.wikipedia.org/wiki/Kolorowanie_grafu
# kaims.pl/~deren/ag/wyklady/14_kolorowanie.pdf

from coloring import VerticesColoring
from main.structures.edges import Edge

class EdgesColoring(object):
    """Algorytm kolorowania krawedzi grafu."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu grafem, ktorego krawedziom
        zostana nadane kolory."""
        self.graph = graph
        self.color = dict(
            ((node, 0) for node in self.graph.iternodes()))
        self.number_of_color = 1

    def run(self):
        """Metoda uruchamiajaca algorytm kolorowania krawedzi
        grafu. Kolory to liczby calkowite od zera. Wynik
        zapisywany jest w zmiennej self.color."""
        graph_e = self.graph.__class__(self.graph.e(),
                                       self.graph.is_directed())
        i = 0
        edges = dict()
        for source in self.graph.iternodes():
            for target1 in self.graph.iteradjacent(source):
                for target2 in self.graph.iteradjacent(target1):
                    if target2 != source:
                        if (source, target1) not in edges:
                            edges[(source, target1)] = i
                            if not graph_e.is_directed():
                                edges[(target1, source)] = i
                            i += 1
                        if (target1, target2) not in edges:
                            edges[(target1, target2)] = i
                            if not graph_e.is_directed():
                                edges[(target2, target1)] = i
                            i += 1
                        graph_e.add_edge(
                            Edge(edges[(source, target1)],
                                 edges[(target1, target2)]))
        algorithm = VerticesColoring(graph_e)
        algorithm.run()
        self.color = algorithm.color
        self.number_of_color = algorithm.number_of_color
```

6. Podsumowanie

W pracy przedstawiono implementację grafów i wielu algorytmów grafowych, związanych z grafami bez wag. Implementacja została wykonana w języku Python, ponieważ łączy on wydajność, czytelność kodu i dostępność struktur danych wysokiego poziomu. Dzięki temu kod źródłowy może pełnić rolę pseudokodu używanego w literaturze do prezentacji algorytmów, a z drugiej strony kod można uruchomić w wybranym systemie operacyjnym w celu rozwiązania średniej wielkości problemów grafowych. Przykłady korzystania z algorytmów są zawarte w dostarczonych testach jednostkowych.

Prezentowany kod jest zgodny z formalnymi standardami języka Python (PEP8), oraz z praktykami dobrego programowania: adekwatne nazwy zmiennych, odpowiednie komentarze, logiczny podział kodu na moduły, klasy, funkcje. Poprawność kodu została potwierdzona poprzez testy jednostkowe.

W pracy zebrano podstawowe algorytmy przeszukiwania grafów, sortowania topologicznego, znajdowania składowych spójnych i silnie spójnych składowych. Pokazano również algorytmy znajdowania maksymalnego skojarzenia w grafach dwudzielnych, obliczania domknięcia przechodniego grafu, kolorowania wierzchołków i krawędzi grafu. W końcu zaimplementowano także algorytmy związane z grafami eulerowskimi i hamiltonowskimi.

Część teoretyczna pracy używa głównie multigrafów, które są uogólnieniem grafów prostych. Działanie algorytmów przedstawionych w pracy sprawdzono tylko dla grafów prostych. Dostosowanie kodu do pracy z multigrafami wymaga jeszcze dodatkowych prac, z których zrezygnowano na korzyść powiększenia bazy algorytmów. Warto zwrócić uwagę na kilka elementów, które różnią implementacje grafów prostych i multigrafów.

— `G.weight(edge)` dla grafów prostych służy do odczytu wagi krawędzi, a dla multigrafów chcemy odczytać liczbę krawędzi równoległych. W reprezentacji macierzy sąsiedztwa właśnie ta liczba znajduje się w miejscu wagi. Stąd w tej implementacji nie można przedstawić multigrafów ważonych. Można ewentualnie wykorzystać atrybut krawędzi `edge.weight` do reprezentowanie pęku krawędzi równoległych pomiędzy dwoma wierzchołkami.
— `G.iteradjacent(source)` zwraca iterator kolejnych sąsiadów danego wierzchołka (bez powtórzeń), niezależnie od liczby krawędzi równoległych prowadzących do sąsiada.
— `G.iteredges()`, `G.iterinedges(source)`, `G.iteroutedges(source)` zwracają iteratory krawędzi, przy czym dla krawędzi równoległych pojawią się powtórzenia krawędzi.
— W multigrafach zezwalamy na dodawanie pętli i kolejnych krawędzi równoległych, a dla grafów prostych wyzwalane są wyjątki.

W świecie oprogramowania można znaleźć wiele bibliotek grafowych, ale

2852604295(69)

nie jest nam znany przykład takiego połączenia czytelności, elastyczności i dostępności kodu, który mógłby spełniać nasze oczekiwania. Z drugiej strony warto zapoznać się z możliwościami innych bibliotek, aby skorzystać z dodatkowych algorytmów, czy też narzędzi do graficznej prezentacji grafów.

A. Kod źródłowy dla krawędzi i grafów

Dwie podstawowe klasy wykorzystywane we wszystkich algorytmach to klasa Edge, reprezentująca krawędzie skierowane, oraz klasa Graph, reprezentująca grafy skierowane i nieskierowane. Sprawdzono sześć implementacji klasy Graph, cztery implementacje nie korzystające z wag, oraz dwie implementacje przygotowane do obsługi grafów ważonych, ale w naszych rozważaniach wszystkie wagi były domyślnie równe 1.

A.1. Klasa Edge

Listing A.1. Moduł edges.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Uniwersalna klasa dla krawedzi.
# Hashable edges - pomysl na __hash__ na podstawie
# http://stackoverflow.com/questions/793761/
# built-in-python-hash-function

class Edge(object):
    """Klasa reprezentujaca krawedz skierowana w grafie."""

    def __init__(self, source, target, weight=1):
        """Inicjalizuje obiekt krawedzi.
        Jesli graf ma wagi, to mozna przekazac dodatkowo
        wage krawedzi, ktora dla grafu bez wag przyjmuje
        wartosc domyslna rowna 1."""
        self.source = source
        self.target = target
        self.weight = weight

    def __repr__(self):
        """Zwraca krawedz w reprezentacji tekstowej."""
        if self.weight == 1:
            return "Edge(%s, %s)" % (
                repr(self.source),
                repr(self.target))
        else:
            return "Edge(%s, %s, %s)" % (
                repr(self.source),
                repr(self.target),
                repr(self.weight))

    def __cmp__(self, other):
        """Porownywanie krawedzi."""
        if self.weight > other.weight:
```


4198421909(71)

```
            return 1
        if self.weight < other.weight:
            return -1
        if self.source > other.source:
            return 1
        if self.source < other.source:
            return -1
        if self.target > other.target:
            return 1
        if self.target < other.target:
            return -1
        return 0

    def __hash__(self):
        """Zwraca hash dla krawedzi."""
        return hash(repr(self))

    def __invert__(self):
        """Zwraca nowa krawedz z przeciwnym kierunkiem."""
        return Edge(self.target, self.source, self.weight)

    inverted = __invert__
```

A.2. Klasa Graph (słownik słowników)

Listing A.2. Moduł graphs5.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from edges import Edge

class Graph(dict):
    """
    Podejscie z listami sasiedztwa. Sasiednie wierzcholki
    trzymane sa w slownikach. Wierzcholki moga byc zarowno
    liczbami jaki i dowolnymi obiektami np. stringami, itp.
    {"A":{"B":1,"C":2}, "B":{"C":3,"D":4}, "C":{"D":5},
    "D":{"C":6}, "E":{"C":7}, "F":{}}
    """

    def __init__(self, n=0, directed=False):
        """Inicjalizacja obiektu grafu."""
        self.n = n  # kompatybilnosc z Sedgewickiem
        self.directed = directed

    def is_directed(self):
        """Sprawdza, czy graf jest skierowany, czy nieskierowany."""
        return self.directed

    def v(self):
        """Zwraca liczbe wierzcholkow grafu."""
        return len(self)

    def e(self):
```


3256299143(72)

```
        """Zwraca liczbe krawedzi grafu."""
        edges = sum(len(self[node]) for node in self)
        return (edges if self.is_directed() else edges / 2)

    def add_node(self, node):
        """Dodaje nowy wierzcholek do grafu."""
        if node not in self:
            self[node] = dict()

    def has_node(self, node):
        """Sprawdzenie, czy wierzcholek nalezy do grafu."""
        return node in self

    def del_node(self, node):
        """Usuwa wierzcholek z grafu."""
        for edge in list(self.iterinedges(node)):
            self.del_edge(edge)
        if self.is_directed():
            for edge in list(self.iteroutedges(node)):
                self.del_edge(edge)
        del self[node]

    def add_edge(self, edge):
        """Dodaje nowa krawedz do grafu."""
        if edge.source == edge.target:
            raise ValueError("petle sa zabronione")
        self.add_node(edge.source)
        self.add_node(edge.target)
        if edge.target not in self[edge.source]:
            self[edge.source][edge.target] = edge.weight
        else:
            raise ValueError("krawedzie rownolegle sa zabronione")
        if not self.is_directed():
            if edge.source not in self[edge.target]:
                self[edge.target][edge.source] = edge.weight
            else:
                raise ValueError("krawedzie rownolegle sa zabronione")

    def del_edge(self, edge):
        """Usuwa istniejaca krawedz grafu."""
        del self[edge.source][edge.target]
        if not self.is_directed():
            del self[edge.target][edge.source]

    def has_edge(self, edge):
        """Sprawdza czy krawedz istnieje w grafie."""
        return edge.source in self and edge.target in self[edge.source]

    def weight(self, edge):
        """Zwraca wage krawedzi lub zero."""
        if edge.source in self and edge.target in self[edge.source]:
            return self[edge.source][edge.target]
        else:
            return 0

    def iternodes(self):
        """Zwraca iterator do listy wierzcholkow grafu."""
```


6086993598(73)

```
        return self.iterkeys()

    def iteradjacent(self, source):
        """Zwraca iterator wierzcholkow sasiednich."""
        return self[source].iterkeys()

    def iteroutedges(self, source):
        """Zwraca iterator krawedzi wychodzacych grafu."""
        for target in self[source]:
            yield Edge(source, target, self[source][target])

    def iterinedges(self, source):
        """Zwraca iterator krawedzi wchodzacych grafu."""
        if self.is_directed():  # O(V) time
            for (target, sources_dict) in self.iteritems():
                if source in sources_dict:
                    yield Edge(target, source, sources_dict[source])
        else:
            for target in self[source]:
                yield Edge(target, source, self[target][source])

    def iteredges(self):
        """Zwraca iterator krawedzi grafu."""
        for source in self.iternodes():
            for target in self[source]:
                if self.is_directed() or source < target:
                    yield Edge(source, target, self[source][target])

    def show(self):
        """Prezentacja grafu."""
        for source in self.iternodes():
            print source, ":",
        for edge in self.iteroutedges(source):
            print "%s(%s)" % (edge.target, edge.weight),
        print

    def copy(self):
        """Zwraca kopie grafu."""
        new_graph = Graph(n=self.n, directed=self.directed)
        for node in self.iternodes():
            new_graph[node] = dict(self[node])
        return new_graph

    def transpose(self):
        """Zwraca graf transponowany."""
        new_graph = Graph(n=self.n, directed=self.directed)
        for node in self.iternodes():
            new_graph.add_node(node)
        for edge in self.iteredges():
            new_graph.add_edge(~edge)
        return new_graph

    def __eq__(self, other):
        """Sprawdzenie, czy grafy sa rowne."""
        raise NotImplementedError()

    def __ne__(self, other):
```


1984891598(74)

```
        """Sprawdzenie, czy grafy sa rozne."""
        return not self == other

    def add_graph(self, other):
        """Dolacza inny graf do istniejacego."""
        for node in other.iternodes():
            self.add_node(node)
        for edge in other.iteredges():
            self.add_edge(edge)
```

A.3. Klasa Graph (lista list)

Listing A.3. Moduł graphs6.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from edges import Edge

class Graph(object):
    """Klasa reprezentujaca graf skierowany lub nieskierowany."""

    def __init__(self, n, directed=False):
        """Inicjalizacja obiektu grafu."""
        if n < 0:
            raise ValueError("n musi byc liczba dodatnia")
        self.n = n
        self.directed = directed
        self.data = [[0] * self.n for node in xrange(self.n)]

    def is_directed(self):
        """Sprawdza, czy graf jest skierowany, czy nieskierowany."""
        return self.directed

    def v(self):
        """Zwraca liczbe wierzcholkow grafu."""
        return self.n

    def e(self):
        """Zwraca liczbe krawedzi grafu."""
        counter = 0
        for source in xrange(self.n):
            for target in xrange(self.n):
                if self.data[source][target] != 0:
                    counter = counter + 1
        return (counter if self.directed else counter / 2)

    def add_node(self, node):
        """
        Metoda dodaje wierzcholek. W przypadku macierzy sasiedztwa
        funkcja nie jest zaimplementowana, poniewaz wszystkie
        wierzcholki sa dodane od razu.
        """
        pass
```


3466459359(75)

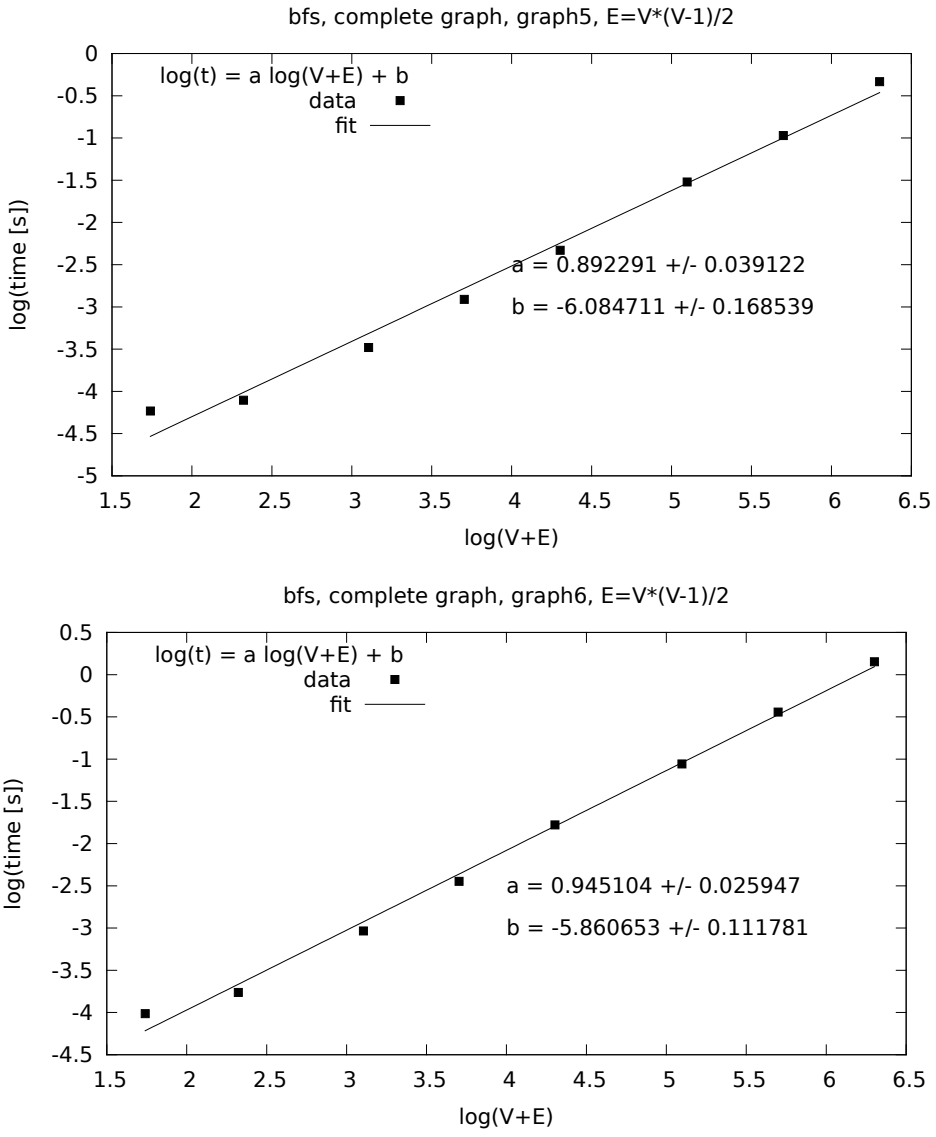
```
    def has_node(self, node):
        """Sprawdzenie, czy wierzcholek nalezy do grafu."""
        return 0 <= node < self.n

    def del_node(self, source):
        """
        Metoda usuwa wierzcholek. W przypadku macierzy
        sasiedztwa funkcja nie jest zaimplementowana.
        """
        pass

    def add_edge(self, edge):
        """Dodaje nowa krawedz do grafu."""
        if edge.source == edge.target:
            raise ValueError("petle sa zabronione")
        self.add_node(edge.source)
        self.add_node(edge.target)
        if self.data[edge.source][edge.target] == 0:
            self.data[edge.source][edge.target] = edge.weight
        else:
            raise ValueError("krawedzie rownolegle sa zabronione")
        if not self.directed:
            if self.data[edge.target][edge.source] == 0:
                self.data[edge.target][edge.source] = edge.weight
        else:
            raise ValueError("krawedzie rownolegle sa zabronione")

    def del_edge(self, edge):
        """Usuwa istniejaca krawedz grafu."""
        self.data[edge.source][edge.target] = 0
        if not self.directed:
            self.data[edge.target][edge.source] = 0

    def has_edge(self, edge):
        """Sprawdzenie, czy krawedz istnieje w grafie."""
        return self.data[edge.source][edge.target] != 0

    def weight(self, edge):
        """Zwraca wage krawedzi lub zero."""
        return self.data[edge.source][edge.target]

    def iternodes(self):
        """Zwraca iterator do listy wierzcholkow grafu."""
        return iter(xrange(self.n))

    def iteradjacent(self, source):
        """Zwraca iterator do listy wierzcholkow sasiednich."""
        for target in xrange(self.n):
            if self.data[source][target] != 0:
                yield target

    def iteroutedges(self, source):
        """Zwraca iterator krawedzi wychodzacych."""
        for target in xrange(self.n):
            if self.data[source][target] != 0:
                yield Edge(source, target, self.data[source][target])
```


2963879837(76)

```
    def iterinedges(self, source):
        """Zwraca iterator krawedzi wchodzacych."""
        for target in xrange(self.n):
            if self.data[target][source] != 0:
                yield Edge(target, source, self.data[target][source])

    def iteredges(self):
        """Zwraca iterator krawedzi grafu."""
        for source in xrange(self.n):
            for target in xrange(self.n):
                if self.data[source][target] != 0 and \
                        (self.directed or source < target):
                    yield Edge(source, target, self.data[source][target])

    def show(self):
        """Prezentacja grafu."""
        for source in xrange(self.n):
            print source, ":",
            for target in self.iteradjacent(source):
                print "%s(%s)" % (target, self.data[source][target]),
            print

    def copy(self):
        """Zwraca kopie grafu."""
        new_graph = Graph(n=self.n, directed=self.directed)
        for source in xrange(self.n):
            for target in xrange(self.n):
                new_graph.data[source][target] = self.data[source][target]
        return new_graph

    def transpose(self):
        """Zwraca graf transponowany."""
        new_graph = Graph(n=self.n, directed=self.directed)
        for source in xrange(self.n):
            for target in xrange(self.n):
                new_graph.data[source][target] = self.data[target][source]
        return new_graph

    def __eq__(self, other):
        """Sprawdzenie, czy grafy sa rowne."""
        raise NotImplementedError()

    def __ne__(self, other):
        """Sprawdzenie, czy grafy sa rozne."""
        return not self == other

    def add_graph(self, other):
        """Dolacza inny graf do istniejacego"""
        for node in other.iternodes():
            self.add_node(node)
        for edge in other.iteredges():
            self.add_edge(edge)
```

B. Testy wydajnościowe wybranych algorytmów

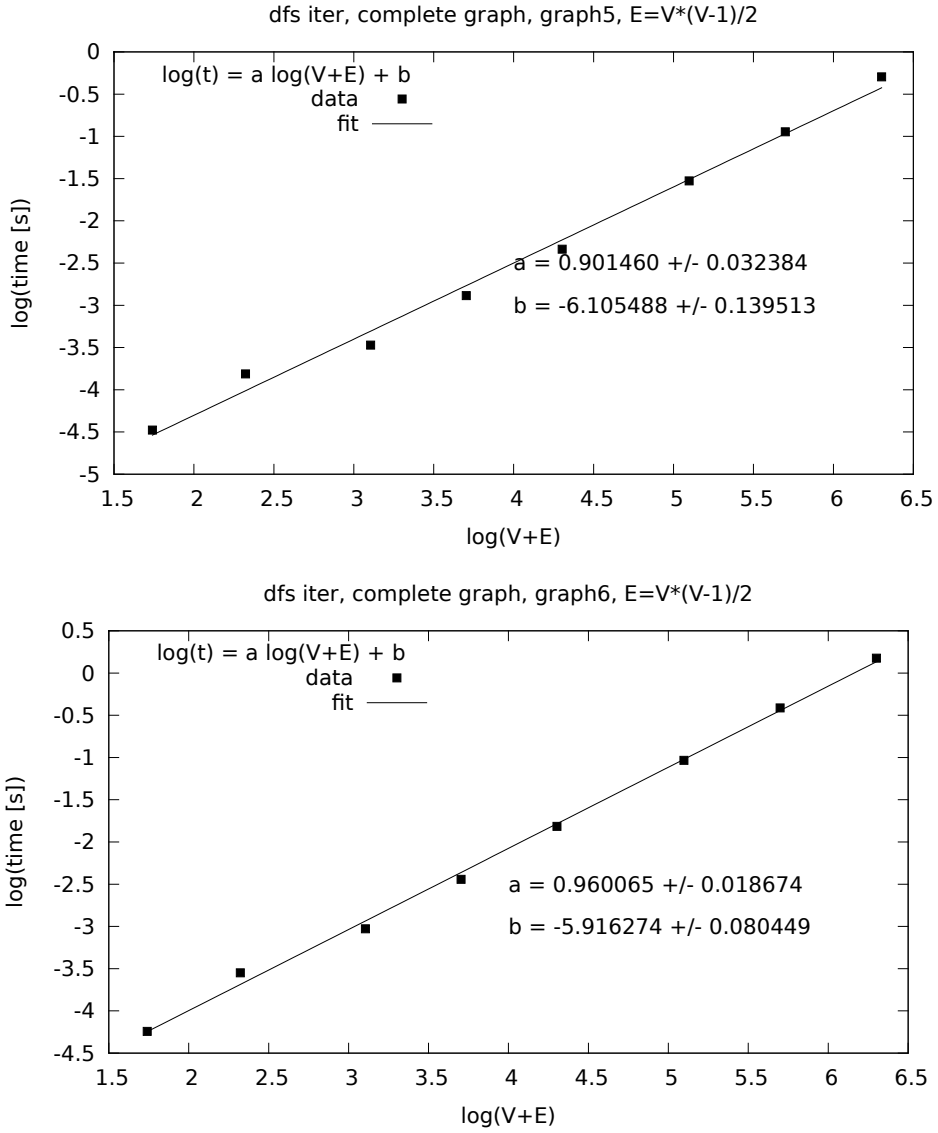
B.1. Test wydajności algorytmu przeszukiwania wszerz



Rysunek B.1. Wykresy wydajności algorytmu przeszukiwania wszerz dla dwóch różnych implementacji grafów pełnych. Dla grafu pełnego dla dużych grafów szybsza jest implementacja graph5 oparta na słowniku słownika. Współczynniki a bliskie 1 potwierdzają liniową zależność czasu od rozmiaru grafu.

3279713690(79)

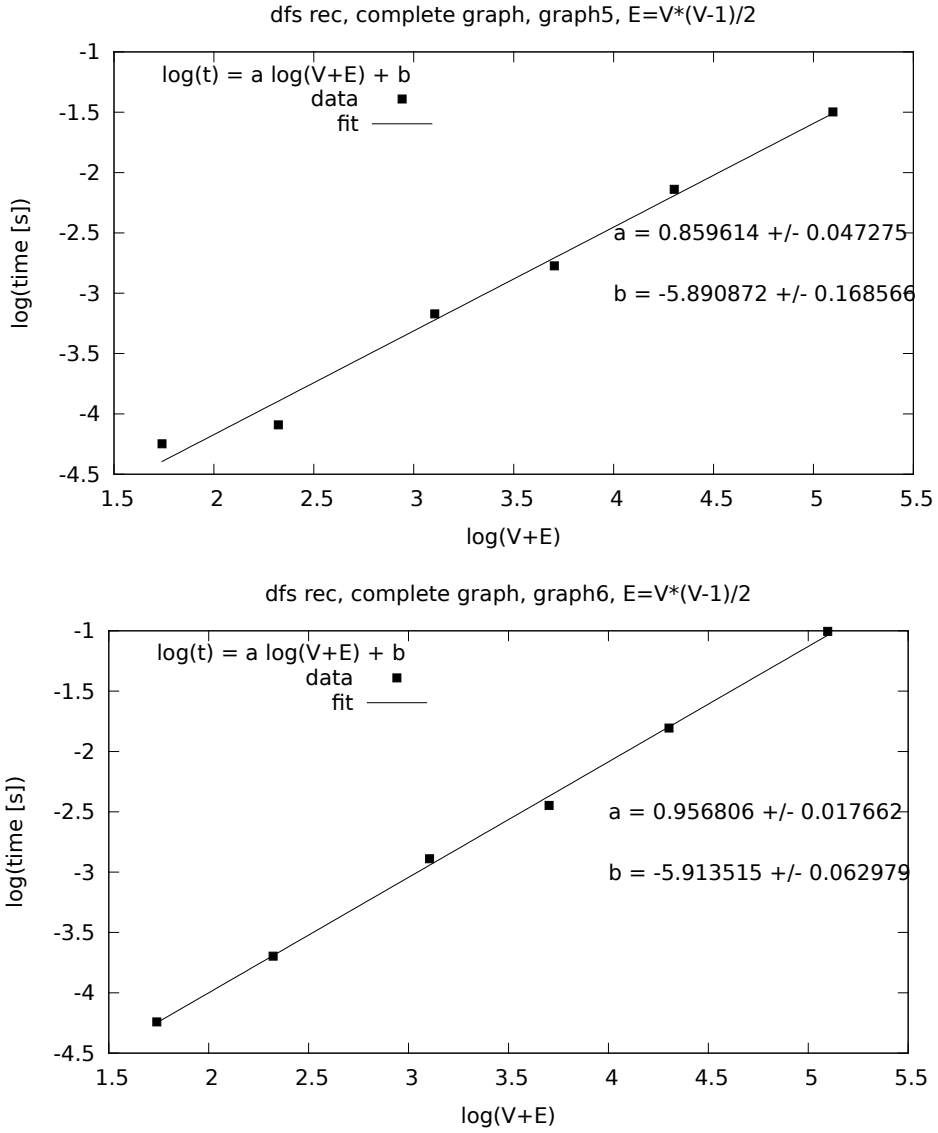
B.2. Test wydajności algorytmu przeszukiwania w głąb iteracyjnego



Rysunek B.2. Wykresy wydajności algorytmu przeszukiwania w głąb iteracyjnego dla dwóch różnych implementacji grafów pełnych. Implementacja graph5 jest szybsza od graph6. Współczynniki a bliskie 1 potwierdzają liniową zależność czasu od rozmiaru grafu.

2134504046(80)

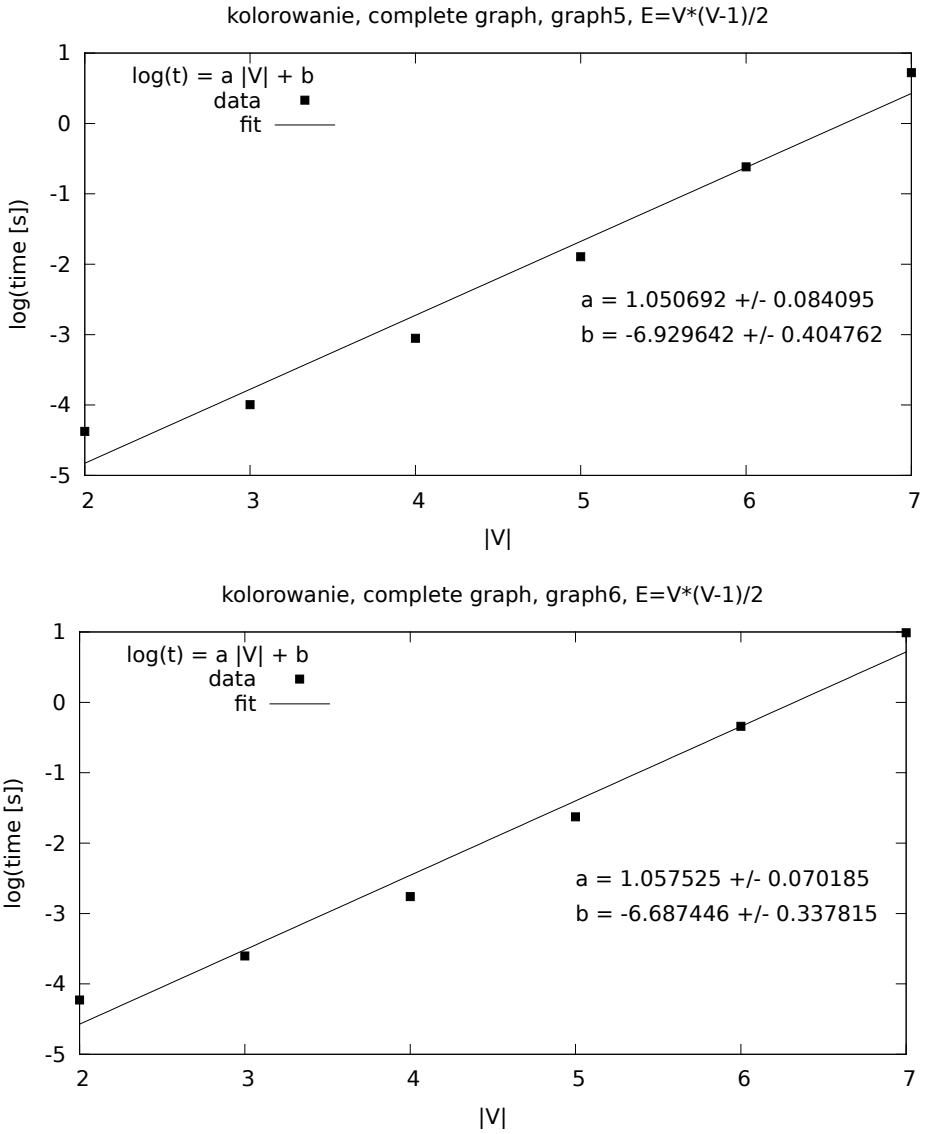
B.3. Test wydajności algorytmu przeszukiwania w głąb rekurencyjnego



Rysunek B.3. Wykresy wydajności algorytmu przeszukiwania w głąb rekurencyjnego dla dwóch różnych implementacji grafów pełnych. Implementacja graph5 jest szybsza od graph6. Współczynniki a bliskie 1 potwierdzają liniową zależność czasu od rozmiaru grafu.

3122141315(81)

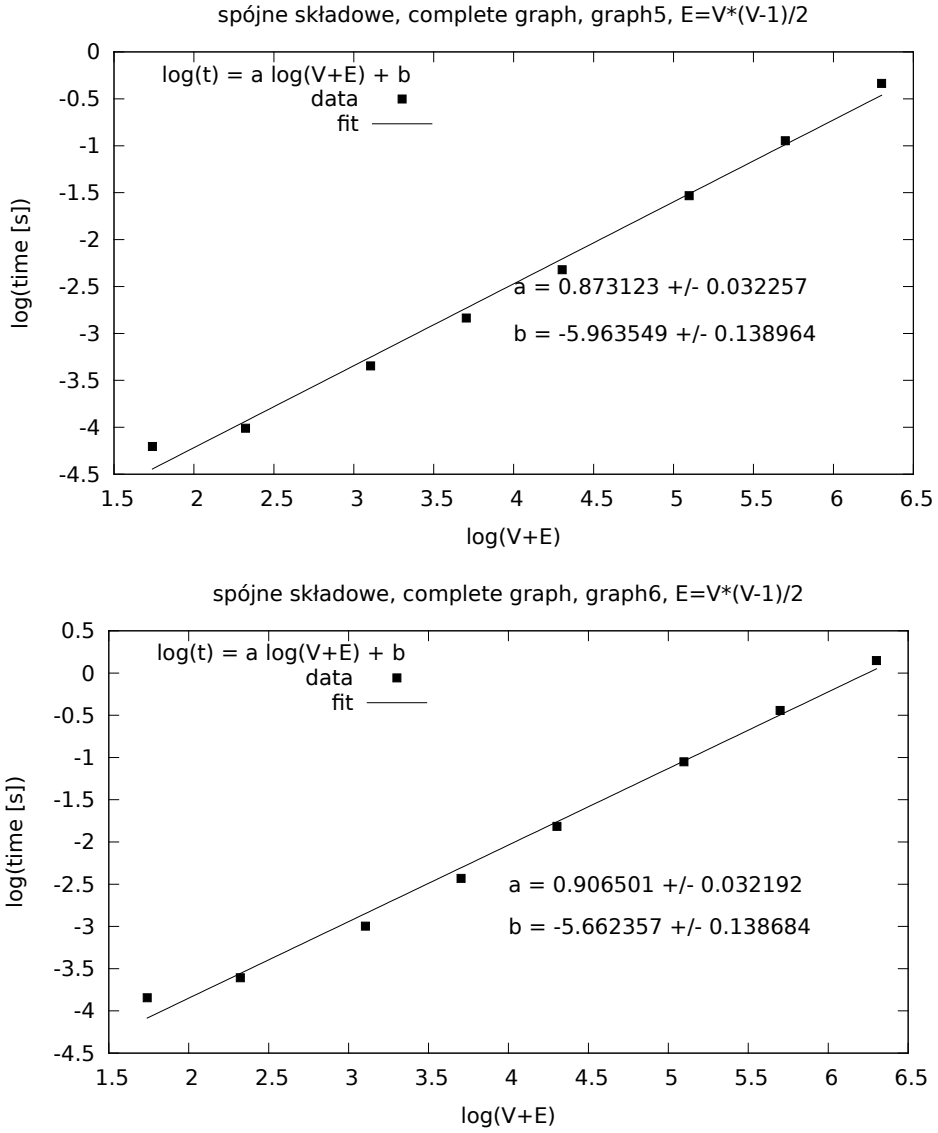
B.4. Test wydajności algorytmu kolorowania grafu



Rysunek B.4. Wykresy wydajności algorytmu kolorowania grafu dla dwóch różnych implementacji grafów pełnych. Wykresy wydają się potwierdzać zależność wykładniczą, choć ilość danych nie jest duża.

5157147205(82)

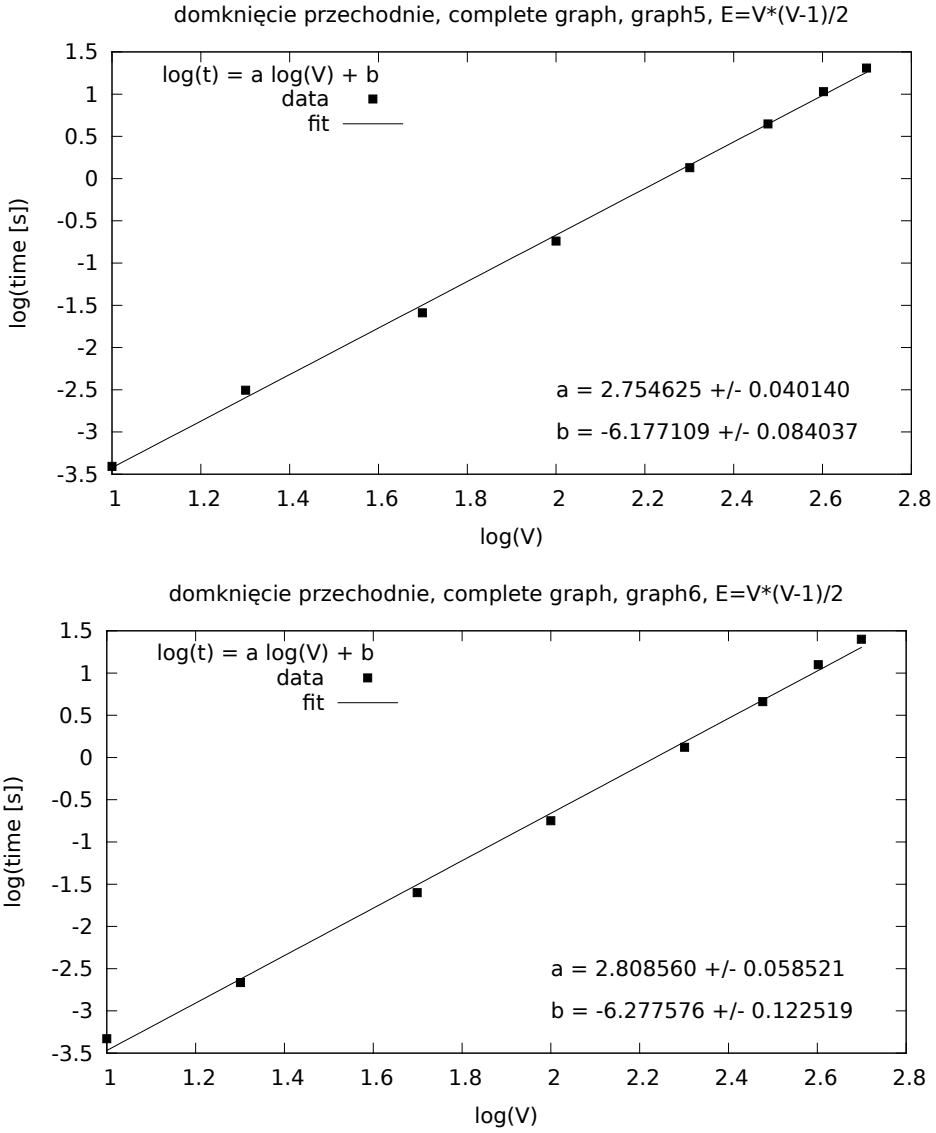
B.5. Test wydajności algorytmu wyszukiwania spójnych składowych



Rysunek B.5. Wykresy wydajności algorytmu wyszukiwania spójnych składowych grafu dla dwóch różnych implementacji grafów pełnych. Implementacja graph5 jest szybsza od graph6. Współczynniki a bliskie 1 potwierdzają liniową zależność czasu od rozmiaru grafu.

2429120057(83)

B.6. Test wydajności algorytmu wyszukiwania domknięcia przechodniego



Rysunek B.6. Wykresy wydajności algorytmu wyszukiwania domknięcia przechodniego grafu dla dwóch różnych implementacji grafów pełnych. Implementacja graph5 jest szybsza od graph6. Współczynniki a bliskie 3 potwierdzają zależność $O(V^3)$.

Bibliografia

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

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

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

[4] Robert Sedgewick, *Algorytmy w C++. Część 5. Grafy*, Wydawnictwo RM, Warszawa 2003.

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

[6] Wikipedia, Graf dwudzielny, 2014, `http://pl.wikipedia.org/wiki/Graf_dwudzielny`.

[7] Ashay Dharwadker, *A new algorithm for finding hamiltonian circuits* `http://edu.i-lo.tarnow.pl/inf/utils/002_roz/hamilton.pdf`

2799930991(85)