Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Paweł Motyl

Nr albumu: 1065812

Implementacja wybranych algorytmów dla multigrafów w języku Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2015

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

Składam serdeczne podziękowania Opiekunowi mojej pracy, Panu dr. hab. Andrzejowi Kapanowskiemu, za okazaną nieocenioną pomoc, poświęcony czas oraz wielką cierpliwość, bez których ta praca nie mogłaby powstać.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów wykorzystujących multigrafy. Multigrafy są instancjami klasy MultiGraph, ale interfejs jest zgodny z klasą Graph, opisującą grafy proste. Dzięki temu klasy odpowiadające algorytmom mogą działać na obu typach grafów. Przygotowano dwie różne implementacje multigrafów, z wagami i bez wag.

Zaimplementowano trzy algorytmy do znajdowania cykli Eulera: algorytm Flory'ego, algorytm z wykorzystaniem stosu, oraz algorytm Hierholzera. Ponadto dołączono algorytmy rozwiązujące problem chińskiego listonosza.

Stworzono szereg heurystycznych algorytmów do znajdowania maksymalnego zbioru niezależnego: algorytm trywialny (zachłanny), algorytmy z dołączaniem wierzchołków, algorytmy z usuwaniem wierzchołków. Najwięcej algorytmów dotyczy kolorowania wierzchołków grafu. Jest to algorytm dokładny, sześć algorytmów sekwencyjnych i trzy algorytmy sekwencyjne z wymianą kolorów. Dołączono również algorytm kolorowania krawędzi, wykorzystujący graf krawędziowy.

Algorytmy zbiorów niezależnych i algorytmy kolorowania grafów zostały przetestowane. Sprawdzono ich jakość i realną złożoność obliczeniową.

Słowa kluczowe: grafy, multigrafy, graf Eulera, problem chińskiego listonosza, zbiór niezależny, kolorowanie grafów

English title: Python implementation of selected algorithms for multigraphs

Abstract

Python implementations of selected graph algorithms using multigraphs are presented. Multigraphs are instances of the MultiGraph class, but interface is compatible with the Graph class describing simple graphs. This is why algorithm classes can be used with both types of graphs. Two different implementations of the MultiGraph class are available, for weighted and unweighted graphs.

The algorithms for finding an Eulerian cycle are presented: the Fleury's algorithm, the algorithm using a stack, and the Hierholzer's algorithm. The algorithms solving the chineese postman problem are also included.

Several heuristic algorithms for finding a maximum independent set are shown: the trivial/greedy algorithm, the smallest first algorithm, the largest last algorithm, and their modified versions. Many algorithms for node coloring are prepared: the exact algorithm, six sequential algorithms, and three sequential algorithms with color interchange. Edge coloring algorithm using a line graphs is also included.

Independet set algorithms and graph coloring algoriths were tested and their quality and complexity was checked.

Keywords: graphs, multigraphs, Eulerian graph, Chinese postman problem, independent set, graph coloring

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Z pojęciem algorytmu spotykamy się po raz pierwszy już na wczesnych etapach nauczania, kiedy to pojęcie odnosi się do matematyki i rozwiązywania zadań za pomocą sprawdzonych metod. Wówczas, określone są poszczególne kroki, których przejście pozwala na poprawne rozwiązanie problemu.

Obecnie termin algorytm używany jest również w programowaniu i oznacza sposób na rozwiązanie jakiegoś zadania, a sztuką jest nie tylko opracowanie algorytmu, ale też opracowanie go w taki sposób, aby był on wydajny.

Niniejsza praca została poświęcona algorytmom multigrafowym. *Multigraf* jest pojęciem teorii grafów i w niniejszej pracy jest on rozumiany jako *graf*, w którym krawędzie łączące dwa wierzchołki mogą być *wielokrotne*, chociaż niektórzy autorzy takie rozszerzone pojęcie *grafu* również nazywają *grafem*.

Rozważania oparte zostały na języku Python, który jest językiem dbającym o czytelność i klarowność kodu dzięki m.in. dynamicznemu typowaniu i automatycznemu zarządzaniu pamięcią. Pozwoliło to na opracowanie przedstawionych algorytmów bez zawiłości składniowych, które wprowadzają inne języki programowania.

1.1. Organizacja pracy

Praca została podzielona na części, składające się w logiczną całość. Rozdział 1 jest wstępem do zagadnień poruszanych w niniejszej pracy. Rozdział 2 jest krótkim wstępem do technologii, która została wykorzystana w implementacji przedstawionych algorytmów. Rozdział 3 zawiera podstawowe definiecje z teorii grafów. Rozdział 4 w skrócie przedstawia interfejs klasy multigrafu wykorzystanej w przedstawionych algorytmach. Rozdział 5 przedstawia algorytmy wyznaczania cyklu Eulera, a także algorytm rozwiązania problemu chińskiego listonosza. Rozdział 6 zawiera omówienie zagadnienia zbiorów niezależnych, skojarzeń i pokrycia wierzchołkowego, oraz algorytmy służące do wyznaczenia zbiorów niezależnych. Rozdział 7 zawiera teorię związaną z kolorowaniem wierzchołków i krawędzi multigrafów, oraz implementacje algorytmów rozwiązujących problem kolorowania. Rozdział 8 zawiera podsumowanie pracy.

2. Wprowadzenie do Pythona

Python jest jednym z najłatwiejszych do nauczenia się językiem programowania, który jednocześnie jest bardzo szeroko wykorzystywany w edukacji i przemyśle komputerowym. Działa na wielu systemach, również systemach wbudowanych. Kod w języku Python jest łatwy do zrozumienia, zwięzły, wspierający dobre praktyki programowania. Jest to język skryptowy, interpretowany - co oznacza, że napisany skrypt wykonywany jest za pomocą interpretera.

Jedną z największych zalet Pythona jest bogata biblioteka standardowa, umożliwiająca wykonanie wielu zadań pojawiających się w codziennej pracy programisty. Dzięki niej, operacje takie jak pobranie pliku z Internetu lub utworzenie serwera WWW mogą być wykonane za pomocą kilku linii kodu. Istnieją również tysiące specjalizowanych bibliotek, takich jak NumPy (biblioteka funkcji liczbowych) czy Twisted (biblioteka funkcji sieciowych).

Warto wspomnieć, że oprócz Pythona napisanego w języku C, istnieje również Jython napisany w Javie, oraz Iron Python napisany w .NET. Różne implementacje stworzono w celu łatwiejszej integracji kodu Pythona z kodem napisanym w tych językach.

2.1. Całkowite typy danych

W Pythonie wbudowane są dwa całkowite typy danych: **int** oraz **bool**. W wyrażeniu logicznym 0 i False dają w wyniku False, a dowolna inna liczba całkowita i True w wyniku dają True. W wyrażeniach liczbowych True przyjmuje wartość 1, False wartość 0.

Wielkość liczby całkowitej w Pythonie ograniczona jest jedynie przez ilość dostępnej pamięci. Można utworzyć liczby całkowite składające się z dziesiątek cyfr i swobodnie z nimi pracować, jednak będzie to wolniejsze niż w przypadku liczb całkowitych, które mogą być przedstawione przez procesor.

Tabela 2.1 przedstawia spis funkcji i operatorów arytmetycznych dostępnych dla liczb całkowitych.

2.2. Zmiennoprzecinkowe typy danych

W Pythonie istnieją trzy wbudowane typy zmiennoprzecinkowe: **float**, **complex** i decimal.Decimal.

2.2.1. Liczby rzeczywiste

Typ **float** przechowuje liczby zmiennoprzecinkowe o podwójnej precyzji, których zakres zależy od kompilatora użytego do kompilacji Pythona.

Tabela 2.1. Funkcje i operatory działające na liczbach

Instrukcja	**Opis**
x + y	dodaje liczby x i y
x − y	odejmuje liczbę y od x
x ∗ y	mnoży liczbę x przez y
x / y	dzieli liczbę x przez y
x % y	oblicza wartość modulo dzielenia x przez y
x ∗∗ y	podnosi liczbę x do potęgi y
− x	zmienia znak niezerowej liczby x
abs(x)	zwraca wartość bezwzględną liczby x
divmod(x, y)	zwraca iloraz oraz resztę z dzielenia x przez y
pow(x, y)	działa tak samo jak x ∗∗ y
pow(x, y, z)	równoważność operacji (x ∗∗ y) % z
round(x, n)	zwraca liczbę x zaokrągloną do n cyfr po przecinku

Wszystkie operatory i funkcje przedstawione w tabeli 2.1 mogą być stosowane również z liczbami typu **float**.

2.2.2. Liczby dziesiętne

W odróżnieniu od liczb typu **float**, dokładność obliczeń ustaloną przez programistę, uzyskaną kosztem czasu wykonywania obliczeń, daje niezmienny typ Decimal z modułu decimal z biblioteki standardowej. W celu korzystania z liczb Decimal należy wcześniej zaimportować moduł decimal.

Liczby typu Decimal tworzone są przez funkcję decimal.Decimal(), która jako argument może przyjąć liczbę całkowitą, lub ciąg tekstowy. Do liczb typu Decimal można stosować wszystkie operatory i funkcje przedstawione w tabeli 2.1, przy pewnych ograniczeniach. Jeżeli lewy operand operatora ∗∗ jest typu Decimal, prawy operand musi być liczbą całkowitą. Również, jeżeli pierwszy argument funkcji **pow**() jest typu Decimal, drugi i ewentualnie trzeci argument muszą być typu całkowitego.

2.2.3. Liczby zespolone

Complex jest niezmiennym typem danych, przechowującym parę liczb **float**, z których jedna przedstawia część rzeczywistą (atrybut real), a druga część urojoną (atrybut imag). Również dla liczb zespolonych mogą być stosowane operatory i funkcje matematyczne przedstawione w tabeli 2.1, z wyłączeniem %, **divmod**() oraz trójargumentowego wywołania **pow**().

2.3. Ciągi tekstowe

Ciągi tekstowe są prezentowane przez niezmienny typ danych **str** przechowujący sekwencję znaków Unicode. Dosłowne ciągi tekstowe tworzone są poprzez ujęcie danego tekstu w cudzysłów lub apostrof (po obu stronach tekstu należy zastosować ten sam znak). Można użyć również potrójnie cy-

towanego ciągu tekstowego, który może zawierać w sobie pojedyncze znaki cudzysłowu lub apostrofu, bez dodatkowych zabiegów. Przykładowo:

```
text = """Potrojnie cytowany ciag tekstowy.
Moze zawierac "cytaty" i 'cytaty'."""
```

Aby w normalnym ciągu tekstowym użyć innego ciągu tekstowego, oba ciągi należy ograniczyć odmiennymi znakami, lub zmienić znaczenie znaków ograniczających. Przykładowo:

```
text = "Znakiem odmiennym dla cudzyslowu jest 'apostrof'."
```

Połączenie dwóch ciągów tekstowych można uzyskać poprzez zastosowanie operatora dodawania +.

2.4. Kolekcje

Python udostępnia kilka rodzajów kolekcji.

2.4.1. Listy

Listy w Pythonie są w zasadzie dynamicznymi tablicami. Do elementów listy odwołuje się przez indeks. Elementy listy nie muszą być tego samego typu, lista może zawierać jednocześnie typy danych oraz kolekcje typów danych.

Pusta lista może zostać utworzona przez funkcję **list**() lub przez puste nawiasy kwadratowe [].

Listing 2.1. Tworzenie nowej listy

```
L1 = list()
L2 = []
L3 = [1, "A", [2, "B"]]
```

W listingu 2.1 L1 jest pustą listą, tworzoną przez funkcję **list**(), L2 jest pustą listą tworzoną przez puste nawiasy klamrowe, natomiast L3 jest listą zawierającą trzy elementy różnych typów: liczbę całkowitą, ciąg tekstowy, oraz listę dwuelementową. Tabela 2.2 przedstawia zbiór funkcji obsługiwanych przez listy.

2.4.2. Krotki

Krotka jest niezmiennym odpowiednikiem listy - oznacza to, że raz utworzonej krotki nie można zmienić. Jeżeli istnieje potrzeba modyfikacji danych zawartych w krotce, można wykorzystać konwersję do listy przez funkcję **list**() lub wygenerować nową, zmienioną krotkę.

Pusta krotka może zostać utworzona przez funkcję **tuple**() lub przez puste nawiasy okrągłe (). Krotka udostępnia dwie metody: T.count(x) zwracającą ilość wystąpień elementu x w krotce T oraz T.index(x), zwracającą pozycję indeksu pierwszego wystąpienia elementu x w krotce T.

Tabela 2.2. Metody obsługiwane przez listy.

Instrukcja	Opis
L.append(x)	dołącza element x na koniec listy L
L.count(x)	zwraca liczbę wystąpień elementu x na liście L
L.extend(M)	dołącza wszystkie elementy M do listy L
L += M	działa tak samo jak L.extend(M)
L.index(x)	zwraca indeks pierwszego wystąpienia x na liście L
L.insert(i, x)	wstawia element x na listę L w pozycji indeksu i
L.pop()	zwraca i usuwa ostatni element listy L
L.pop(i)	zwraca i usuwa element listy L znajdujący się w pozycji i
L.remove(x)	usuwa pierwsze wystąpienie elementu x na liście L
L.reverse()	odwraca kolejność elementów na liście L
L.sort()	sortuje listę L w miejscu

2.4.3. Słowniki

Słownik to nieuporządkowana kolekcja zera lub więcej par *(klucz, wartość)*. Kluczem może być dowolny niezmienny typ (np. liczba, ciąg tekstowy). Kluczem może być również krotka, będąca typem niezmiennym, jeżeli każdy jej element jest typem niezmiennym. Natomiast wartością może być dowolny obiekt języka Python. Sam słownik jest typem modyfikowalnym, można więc dodawać, usuwać i modyfikować elementy. Ze względu na nieuporządkowaną strukturę, nie ma możliwości odwołania się do elementu poprzez indeks.

Pusty słownik może zostać utworzony przez funkcję **dict**() wywołaną bez argumentów lub przez zastosowanie pustych nawiasów klamrowych {}.

Listing 2.2. Tworzenie nowego słownika.

```
D1 = dict()
D2 = {}
D3 = {"nazwisko": "Motyl", "imie": "Pawel", "wzrost": 188}
D4 = dict({"nazwisko": "Motyl", "imie": "Pawel", "wzrost": 188})
D5 = dict(nazwisko="Motyl", imie="Pawel", wzrost=188)
```

W listingu 2.2 D1 oraz D2 są pustymi słownikami, D3 dosłownym słownikiem utworzonym przez nawiasy klamrowe, D4 jest słownikiem powstałym z dosłownego słownika w argumencie funkcji **dict**(), natomiast D5 powstaje z argumentów funkcji **dict**() w postaci *klucz=wartość*.

Tabela 2.3 przedstawia zbiór funkcji obsługiwanych przez słowniki.

2.4.4. Zbiory

Zbiór jest nieuporządkowaną kolekcją zera lub większej liczby odniesień do obiektów. Zbiór jest typem modyfikowalnym. Ze względu na nieuporządkowany charakter elementy zbioru nie mają oznaczenia pozycji indeksu. Elementy zbioru zawsze są unikalne, dodanie powtarzających się elementów jest bezpieczne, jednak nie zwiększa to ich krotności w zbiorze.

Pusty zbiór można utworzyć przez funkcję **set**(). Pomimo możliwości tworzenia zbiorów niepustych z wykorzystaniem nawiasów klamrowych {}, nie

Tabela 2.3. Metody obsługiwane przez słowniki.

Instrukcja	Opis
D.clear	usuwa wszystkie elementy ze słownika D
D.copy()	zwraca kopię słownika D
D.fromkeys(s, v)	zwraca słownik, w którym klucze są elementami sekwencji s, a wartości to None lub v (jeśli zostało podane)
D.get(k)	zwraca wartość przypisaną kluczowi k lub wartość None, jeśli klucz k nie istnieje w słowniku D
D.get(k, v)	zwraca wartość przypisaną kluczowi k lub wartość v, jeśli klucz k nie istnieje w słowniku D
D.items()	zwraca listę wszystkich par (klucz, wartość) w słowniku D
D.keys()	zwraca listę wszystkich kluczy w słowniku D
D.pop(k)	zwraca i usuwa element słownika D przypisany do klucza k jeśli k nie istnieje, funkcja zgłasza wyjątek KeyError
D.pop(k, v)	zwraca i usuwa element słownika D przypisany do klucza k jeśli k nie istnieje, funkcja zwraca wartość v
D.popitem()	zwraca i usuwa dowolną parę (klucz, wartość) ze słownika D jeśli słownik jest pusty, zgłasza wyjątek KeyError
D.setdefault(k, v)	działa jak funkcja get() z tą różnicą, że jeżeli klucz k nie znajduje się w słowniu, zostaje wstawiony nowy element o kluczu k i wartości None lub k (jeśli zostało podane)
D.update(A)	dodaje do słownika D każda parę (klucz, wartość) z A, która nie istnieje jeszcze w słowniku D, natomiast dla każdego klucza znajdującego się zarówno w D jak i w A zastępuje odpowiednią wartość z D wartością z A
D.values()	zwraca listę wszystkich wartości w słowniku D

dopuszcza się tworzenia w ten sposób zbiorów pustych (kolizja z notacją dla słowników).

Listing 2.3. Tworzenie nowego zbioru

```
S1 = set()
S2 = set("programowanie")
S3 = {"p", "r", "o", "g", "a", "m", "w", "n", "i", "e"}
S4 = set([1, 3, 1, 5, 1, 7])
```

W listingu 2.3 S1 jest pustym zbiorem, S2 oraz S3 są zbiorami zawierającymi te same elementy. Zbiór S3 został utworzony z unikatowych znaków ze słowa *programowanie*, natomiast w S2 w wyniku konwersji z ciągu znaków na zbiór pojedynczych znaków, zostały usunięte elementy (znaki) powtarzające się. Zbiór S4 został utworzony z sekwencji liczb, powtórzenia zostana usunięte. Tabela 2.4 przedstawia zbiór funkcji obsługiwanych przez zbiory.

2.5. Instrukcje sterujące

Instrukcje sterujące służą do zmiany typowego, sekwencyjnego sposobu przetwarzania instrukcji programu.

Tabela 2.4. Metody obsługiwane przez zbiory.

Instrukcja	**Opis**
S.add(x)	dodaje element x do zbioru S, jeżeli nie znajduje się już w S
S.clear()	usuwa wszystkie elementy ze zbioru S
S.copy()	zwraca kopię zbioru S
S.difference(T) S − T	zwraca nowy zbiór posiadający każdy element zbioru S, który nie jest w zbiorze T
S.difference_update(T) S -= T	usuwa ze zbioru S każdy element, który jest w zbiorze T
S.discard(x)	usuwa element x ze zbioru S
S.intersection(T) S & T	zwraca nowy zbiór, którego każdy element znajduje się zarówno w zbiorze S i T
S.intersection_update(T) S &= T	zbiór S zawiera iloczyn siebie i zbioru T
S.isdisjoint(T)	zwraca True jeśli zbiory S i T nie mają elementów wspólnych
S.issubset(T) S <= T	zwraca True jeśli zbiór S jest równy zbiorowi T lub jest jego podzbiorem
S.pop()	zwraca i usuwa losowy element ze zbioru S lub zwraca wyjątek KeyError jeśli zbiór S jest pusty
S.remove(x)	usuwa element x ze zbioru S lub zwraca wyjątek KeyError jeśli zbiór S nie zawiera elementu x
S.symetric_difference(T) S ^ T	zwraca zbiór składający się z elementów, które znajdują się tylko w zbiorze S lub tylko w zbiorze T
S.symetric_difference_ update(T) S ^= T	zbiór S zawiera różnicę symetryczną siebie i zbioru T
S.union(T) S | T	zwraca zbiór posiadający wszystkie elementy zbioru S i T
S.update(T) S |= T	do zbioru S dodaje każdy element zbioru T

2.5.1. Instrukcja warunkowa

Instrukcja warunkowa **if**−**else** służy do wykonania odpowiedniego fragmentu kodu w zależności od tego, jakie warunki logiczne zostały spełnione. Ogólna składnia polecenia wygląda następująco:

```
if wyrazenie_logiczne1:
    kod1
elif wyrazenie_logiczne2:
    kod2
...
elif wyrazenie_logicznеN:
    kodN
else:
    kod_else
```

Bloki **elif** oraz **else** nie są obowiązkowe. Jeżeli ich nie ma, blok kodu odpowiadający instrukcji **if** zostanie wykonany, gdy jedyne wyrażenie logiczne

przyjmie wartość True. Jeżeli więcej niż jedno wyrażenie logiczne jest spełnione, zostaje wykonany blok kodu rozpoczynający się po pierwszym wyrażeniu logicznym mającym wartość True i kończący przed najbliższym **elif** lub **else**. Jeżeli żadne wyrażenie logiczne nie jest prawdziwe i istnieje blok **else**, zostanie wykonany kod z bloku **else**.

W Pythonie istnieje również wyrażenie warunkowe odpowiadające operatorowi trójargumentowemu znanemu z innych języków programowania. Ogólna składnia wyrażenia:

```
wyrazenie1 if wyrazenie_logiczne else wyrazenie2
```

W tej konstrukcji, jeżeli wyrazenie_logiczne ma wartość True, wynikiem całego wyrażenia jest wyrazenie1. W przeciwnym wypadku wynikiem jest wyrazenie2. Przykład:

```
zmienna = 10
zmienna = 20 if zmienna > 10 else 5
print zmienna    # 5
```

2.5.2. Pętla while

Poniżej przedstawiono ogólną składnię wyrażenia **while**:

```
while wyrazenie_logiczne:
    kod_while
else:
    kod_else
```

Dopóki wyrazenie_logiczne jest prawdziwe, wykonywany jest blok kodu oznaczony kod_while. Jeżeli wartość wyrażenia wynosi lub przyjmie wartość False oraz pętla posiada opcjonalny blok **else**, przed kontynuacją wykonywania kodu programu, jednorazowo zostanie wykonany kod_else.

2.5.3. Pętla for

W Pythonie pętla **for** służy do wykonania tego samego kodu dla wszystkich elementów z sekwencji. Pełna składnia pętli **for**–**in**, podobnie jak pętla **while**, zawiera opcjonalny blok **else**. Poniżej przedstawiono ogólną składnię pętli:

```
for zmienna in sekwencja:
    kod_for
else:
    kod_else
```

Zmienna przyjmuje po kolei wartości wszystkich elementów z sekwencji i dla każdego elementu wykonywane są instrukcje oznaczone jako kod_for. Jeżeli działanie pętli zostanie przerwane po iteracji wszystkich elementów sekwencji, zostanie wykonany opcjonalny kod kod_else. Jeżeli działanie pętli zostanie przerwane w wyniku wywołania instrukcji **break**, **return** lub przez zgłoszenie wyjątku, polecenia z klauzuli **else** nie są wykonywane.

2.5.4. Polecenia continue i break

Poza instrukcjami sterującymi, do kontroli przebiegu wykonania pętli w Pythonie służą polecenia **continue** oraz **break**. Polecenia te mogą zostać wydane tylko w pętlach. Wywołanie **continue** powoduje przerwanie bieżącej iteracji pętli i przekazanie kontroli do początku kolejnej iteracji. Wywołanie w pętli **break** powoduje przerwanie wykonania pętli i przekazanie kontroli do pierwszej instrukcji po pętli.

2.6. Funkcje

Funkcję można rozumieć jako spakowaną i sparametryzowaną funkcjonalność. Funkcję definiuje się za pomocą instrukcji **def**. Przykład definicji własnej funkcji:

```
# Definiowanie funkcji.
def test_mul(x, y = 2):
    return x * y

# Uruchomienie funkcji.
print test_mul(4)
```

Na przykładnie powyższego kodu można przedstawić sposób definicji funkcji. Pierwsza linijka informuje, że tworzona jest funkcja o nazwie test_mul. Zmienne x, y znajdujące się w nawiasach po nazwie, to argumenty funkcji. W czasie wywołania funkcji, w miejsce x i y umieszcza się wartości (np. test_mul(10, 20)), które przypisywane są odpowiednio argumentom. Dodatkowo argument y ma przypisaną wartość domyślną 2, która zostanie wykorzystana, jeżeli funkcja test_mul zostanie wywołana z jednym argumentem. Instrukcja **return** przekazuje określoną wartość w miejsce wywołania funkcji. W przykładzie test_mul(4) widnieje po instrukcji **print**, zatem do instrukcji **print** przekazywana jest wartość zwracana przez funkcję, czyli iloczyn 4*2=8.

2.7. Klasy

Język Python jest językiem zorientowanym obiektowo, a klasy są podstawowym narzędziem do tworzenia nowych typów danych. W listingu 2.4 na przykładnie klasy Point przedstawiono sposób tworzenia nowej klasy oraz pracy z nią.

Listing 2.4. Przykład definicji nowej klasy.

```
class Point:
    """Klasa reprezentujaca punkt na plaszczyznie."""

    def __init__(self, x=0, y=0):
        """Utworzenie punktu o wspolrzednych x, y."""
        self.x = x
        self.y = y

    def length(self):
        """Odleglosc punktu od poczatku ukladu wspolrzednych."""
```

```
        import math
        return math.hypot(self.x, self.y)

# Utworzenie nowego obiektu klasy Point.
point = Point(3, 4)

# Wywolanie metody klasy Point.
print point.length()    # 5

# Odczyt atrybutu instancji klasy Point.
print point.x    # 3
```

Definicja klasy odbywa się za pomocą instrukcji **class**, zawierającej nazwę nowej klasy. W klasie Point znajdują się dwie metody: __init__() oraz length(). Każda metoda jako pierwszy argument przyjmuje **self**, który jest referencją do obiektu, na którym została wywołana. Metoda __init__() posiada dodatkowo dwa argumenty (w tym przykładzie domyślne), które służą do inicjalizacji wartości pól w klasie.

Metoda __init__() wywoływana jest w chwili tworzenia nowego obiektu i jest nazywana konstruktorem. W listingu 2.4 metoda __init__() wywoływana jest z argumentami x=3 i y=4. Metoda length() jest uruchamiana w momencie jej wywołania na rzecz instancji klasy Point (point.length()). Z instancji klasy Point można również uzyskać wartości atrybutów obiektu poprzez ich nazwę, np. point.x.

2.8. Wyjątki

Zgłoszenie wyjątku oznacza zwykle błąd programu i przerwanie jego normalnego działania. Czasem wyjątki służą do wykrywania sytuacji nietypowych, które nie są błędem. Wyjątek zgłoszony i nieobsłużony powoduje przerwanie działania programu. Przechwycenie wyjątku w celu obsłużenia odbywa się za pomocą bloków **try–except**. Poniżej przedstawiono ogólną składnię przechwytywania wyjątków:

```
try:
    kod_try
except wyjatek1:
    kod_wyjatek1
...
except wyjatekN:
    kod_wyjatekN
else:
    kod_else
finally:
    kod_finally
```

W konstrukcji musi się znajdować przynajmniej jeden blok **except**, bloki **else**, **finally** są opcjonalne. Kod kod_else zostanie wykonany, jeżeli kod_try zostanie wykonany w zwykły sposób, natomiast kod_finally zawsze będzie wykonany na końcu.

3. Teoria grafów

Teoria grafów to dział matematyki zajmujący się badaniem własności grafów. Jest ważnym narzędziem matematycznym używanym w wielu różnych dziedzinach, takich jak informatyka, badania operacyjne, genetyka, lingwistyka i socjologia. Rozwijanie algorytmów wyznaczających pewne właściwości grafów jest jednym z bardziej znaczących pól działania informatyki, szczególnie do przedstawiania struktury danych.

Opublikowany w 1741 roku przez Leonarda Eulera w pracy *Solutio problematis ad geometriam situs pertinentis w Commentarii academiae scientiarum Petropolitanae* opis *zagadnienia mostów królewieckich* jest uznawany za pierwszą pracę na temat teorii grafów.

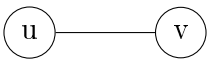
3.1. Multigrafy skierowane i nieskierowane

Graf definiowany jest jako uporządkowana para składająca się ze zbioru wierzchołków V i ze zbioru krawędzi E łączących wierzchołki.

Definicja: *Grafem nieskierowanym prostym* nazywamy parę uporządkowaną $G = (V, E)$, gdzie V jest to niepusty zbiór wierzchołków, natomiast E jest to zbiór *krawędzi nieskierowanych*,

$$E \subseteq \{\{u, v\} : u, v \in V\}. \tag{3.1}$$

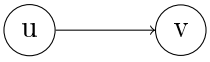
W grafie nieskierowanym krawędź $\{u, v\}$ jest *zbiorem* dwóch różnych wierzchołków, ponieważ kolejność wierzchołków nie ma znaczenia.



Definicja: *Grafem skierowanym prostym* nazywamy parę uporządkowaną $G = (V, E)$, gdzie V jest to niepusty zbiór wierzchołków, natomiast E jest zbiorem *krawędzi skierowanych*,

$$E \subseteq \{(u, v) : u, v \in V\}. \tag{3.2}$$

W grafie skierowanym krawędź (u, v) jest uporządkowaną parą różnych wierzchołków, z początkiem w pierwszym wierzchołku, a końcem w drugim wierzchołku.



Definicja: *Multigrafem nieskierowanym* nazywamy parę uporządkowaną $G = (V, E)$, gdzie V jest to niepusty zbiór wierzchołków, natomiast E jest *multizbiorem* zawierającym krawędzie nieskierowane. Elementy multizbioru mogą się powtarzać. Krawędź nieskierowana multigrafu jest to dwuelementowy multizbiór wierzchołków. *Pętla nieskierowana* jest to krawędź multigrafu postaci $\{v, v\}$. *Krawędzie równoległe nieskierowane* łączą dwa takie same wierzchołki.

Definicja: *Multigrafem skierowanym* nazywamy parę uporządkowaną $G = (V, E)$, gdzie V jest to niepusty zbiór wierzchołków, natomiast E jest *multizbiorem* zawierającym krawędzie skierowane. Krawędź skierowana multigrafu jest to uporządkowana para niekoniecznie różnych wierzchołków. *Pętla skierowana* jest to krawędź multigrafu postaci (v, v). *Krawędzie równoległe skierowane* mają takie same wierzchołki początkowe i końcowe.

3.2. Ścieżki i cykle

Ścieżką z v_0 do v_k w multigrafie $G = (V, E)$ nazywamy ciąg wierzchołków $(v_0, v_1, \ldots, v_k)$, przy czym muszą istnieć krawędzie od v_i do v_{i+1}. Długość takiej ścieżki wynosi k (liczba przeskoków). *Ścieżka prosta* nie ma powtarzających się wierzchołków.

Cykl jest to ścieżka, w której pierwszy i ostatni wierzchołek są takie same, $v_0 = v_k$. *Cykl prosty* nie ma powtarzających się wierzchołków, z wyjątkiem ostatniego. W literaturze za cykl prosty uważa się każdą pętlę, a także dwie krawędzie równoległe nieskierowane. Multigraf nie zawierający cykli prostych nazywamy *acyklicznym*.

3.3. Spójność

Multigraf nieskierowany jest *spójny* (ang. *connected*), jeżeli pomiędzy każdą parą wierzchołków istnieje łącząca je ścieżka (nieskierowana) [4]. Multigraf skierowany jest *silnie spójny* (ang. *strongly connected*), jeśli istnieje ścieżka (skierowana) od dowolnego wierzchołka do każdego innego jego wierzchołka [4].

4. Implementacja multigrafów

Kod źródłowy dla klas reprezentujących multigrafy i ich krawędzie przedstawiono w dodatku A. Tutaj opiszemy interfejs multigrafów.

4.1. Interfejs multigrafów

Algorytmy dla multigrafów przedstawione w niniejszej pracy zostały napisane z wykorzystaniem klasy MultiGraph. Przykład pracy z klasą MultiGraph przedstawia poniższa sesja interaktywna, natomiast kompletny interfejs klasy przedstawiono w tabeli 4.1.

```
>>> from edges import Edge
>>> from multigraphs import MultiGraph
>>> G = MultiGraph()
>>> G.is_directed()
False
>>> G.add_node("B")
>>> G.add_edge(Edge("A", "B", 2))
>>> G.add_edge(Edge("A", "C", 5))
>>> G.show()
A : C(1) B(1)
C : A(1)
B : A(1)
>>> G.degree("A")
>>> G.has_node("D")
False
>>> G.has_node("A")
True
>>> list(G.iternodes())
['A', 'C', 'B']
```

4.2. Analiza złożoności algorytmów

Dla dowolnego multigrafu $G = (V, E)$ mamy dwa podstawowe parametry opisujące jego strukturę: liczbę wierzchołków $n = |V|$ i liczbę krawędzi $m = |E|$. W przypadku grafów prostych liczba krawędzi jest ograniczona przez $n(n-1)/2$, co czasem pozwala na pewne uproszczenia wyrażeń opisujących złożoność algorytmów. Dla multigrafów takie ograniczenie nie występuje i należy traktować liczbę wierzchołków n i liczbę krawędzi m jako dwa niezależne parametry.

Tabela 4.1. Interfejs klasy MultiGraph

Instrukcja	**Opis**
G.v()	zwraca liczbę wierzchołków multigrafu G
G.e()	zwraca liczbę krawędzi multigrafu G
G.is_directed()	sprawdza czy multigraf G jest skierowany
G.add_node(node)	dodaje node do multigrafu G
G.has_node(node)	sprawdza czy w multigrafie G istnieje wierzchołek node
G.del_node(node)	z multigrafu G usuwa wierzchołek node
G.add_edge(edge)	dodaje krawędź edge do multigrafu G
G.del_edge(edge)	usuwa krawędź edge z multigrafu G
G.has_edge(edge)	sprawdza czy multigraf G zawiera krawędź edge
weight(edge)	zwraca liczbę krawędzi równoległych do edge
iternodes()	zwraca wierzchołki multigrafu
iteradjacent(source)	zwraca wierzchołki sąsiadujące z source
iteroutedges(source)	zwraca krawędzie wychodzące z source
iterinedges(source)	zwraca krawędzie wchodzące do source
iteredges()	zwraca krawędzie multigrafu
show()	zwraca tekstową reprezentację multigrafu
copy()	zwraca kopię multigrafu
transpose()	zwraca multigraf transponowany
degree(source)	zwraca stopień wierzchołka source
outdegree(source)	zwraca liczbę wierzchołków wychodzących z source
indegree(source)	zwraca liczbę wierzchołków wchodzących do source

5. Cykl Eulera

Jak wspomniano wcześniej, *zagadnienie mostów królewieckich* leży u podstaw *teorii grafów*. Próba znalezienia *cyklu Eulera* stanowi odpowiedź na pytanie, czy można przez każdy most w Królewcu przejść dokładnie jeden raz i wrócić do punktu startowego.

Drogę zamkniętą przechodzącą dokładnie jeden raz przez każdą krawędź grafu niezorientowanego nazywamy *niezorientowaną drogą Eulera*, a graf który ma ta drogę, jest *grafem Eulera* [3].

Drogę zamkniętą, przechodzącą dokładnie jeden raz przez każdy łuk grafu zorientowanego zgodnie z jego orientacją, nazywamy *zorientowaną drogą Eulera*, a graf który ma taką drogę, nazywamy *zorientowanym grafem Eulera* [3].

5.1. Wyznaczanie cyklu Eulera

Przedstawimy trzy najwazniejsze algorytmy wyznaczające cykl Eulera.

5.1.1. Algorytm Fleury'ego

Dane wejściowe: Multigraf spójny G.

Problem: Znalezienie cyklu Eulera w multigrafie G.

Opis algorytmu: Na początku algorytm wykonuje kopię grafu i ustala wierzchołek startowy. Następnie przechodzi wybraną krawędzią do następnego wierzchołka, a krawędź usuwa. Ważne jest, aby krawędź będącą mostem wybierać dopiero w ostateczności. W związku z tym krawędzie są sprawdzane, czy nie są mostami, przy użyciu algorymu BFS. Algorytm kończy się z chwilą wyczerpania krawędzi. Lista kolejno odwiedzanych wierzchołków daje cykl Eulera.

Złożoność obliczeniowa: Złożoność obliczeniowa jest szacowana na $O(V \cdot (V + E))$.

Złożoność pamięciowa: Kopia grafu zajmuje pamięć rzędu $O(V + E)$, a inne dane pomocnicze też łącznie dają wartość.

Listing 5.1. Algorytm Fleury'ego.

```
#!/usr/bin/python

from bfs import SimpleBFS
```

```
class Fleury:
    """Fleury's algorithm for finding an Eulerian cycle."""

    def __init__(self, graph):
        """The algorithm initialization."""
        self.graph = graph
        if not self._is_eulerian():
            raise ValueError("the graph is not eulerian")
        self.eulerian_cycle = list()
        self.graph_copy = self.graph.copy()

    def run(self, source=None):
        """Executable pseudocode."""
        if source is None:   # get first random node
            source = self.graph.iternodes().next()
        node = source
        self.eulerian_cycle.append(node)
        while self.graph_copy.outdegree(node) > 0:
            for edge in list(self.graph_copy.iteroutedges(node)):
                # graph_copy is changing!
                if not self._is_bridge(edge):
                    break
            self.graph_copy.del_edge(edge)
            self.eulerian_cycle.append(edge.target)
            node = edge.target
        del self.graph_copy

    def _is_bridge(self, edge):
        """Bridge test."""
        list1 = list()
        list2 = list()
        algorithm = SimpleBFS(self.graph_copy)
        algorithm.run(edge.source,
            pre_action=lambda node: list1.append(node))
        self.graph_copy.del_edge(edge)
        algorithm = SimpleBFS(self.graph_copy)
        algorithm.run(edge.source,
            pre_action=lambda node: list2.append(node))
        # Restore the edge.
        self.graph_copy.add_edge(edge)
        return len(list1) != len(list2)

    def _is_eulerian(self):
        """Test if the graph is eulerian."""
        if self.graph.is_directed():
            # We assume that the graph is strongly connected.
            for node in self.graph.iternodes():
                if self.graph.indegree(node) != self.graph.outdegree(node):
                    return False
        else:
            # We assume that the graph is connected.
            for node in self.graph.iternodes():
                if self.graph.degree(node) % 2 == 1:
                    return False
        return True
```

5.1.2. Algorytm z wykorzystaniem stosu

Dane wejściowe: Multigraf spójny G.

Problem: Znalezienie cyklu Eulera w multigrafie G.

Opis algorytmu: Na początku algorytm wykonuje kopię grafu i ustala wierzchołek startowy. Następnie rekurencyjnie odwiedzane są kolejne wierzchołki, a mijane krawędzie usuwane. Wierzchołki kładziemy na stos po wyczerpaniu krawędzi, kiedy powracamy do wierzchołka, z którego przyszliśmy. Po usunięciu wszystkich krawędzi grafu, wierzchołki zdejmowane ze stosu utworzą cykl Eulera.

Złożoność obliczeniowa: Złożoność wynosi $O(E)$.

Złożoność pamięciowa: Kopia grafu zajmuje pamięć rzędu $O(V + E)$, a oprócz tego trzeba obsłużyć rekurencyjne wywołania funkcji odwiedzającej wierzchołki.

Listing 5.2. Algorytm z wykorzystaniem stosu.

```
#!/usr/bin/python

from Queue import LifoQueue

class EulerianCycleDFS:
    """Finding an Eulerian cycle in a multigraph."""

    def __init__(self, graph):
        """The algorithm initialization."""
        self.graph = graph
        if not self._is_eulerian():
            raise ValueError("the graph is not eulerian")
        self.eulerian_cycle = list()
        self.graph_copy = self.graph.copy()
        self.stack = LifoQueue()
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is None:   # get first random node
            source = self.graph.iternodes().next()
        self._visit(source)
        while not self.stack.empty():
            self.eulerian_cycle.append(self.stack.get())
        del self.stack
        del self.graph_copy

    def _visit(self, source):
        """Visiting node."""
        while self.graph_copy.outdegree(source) > 0:
            edge = self.graph_copy.iteroutedges(source).next()
```

```
            self.graph_copy.del_edge(edge)
            self._visit(edge.target)
        self.stack.put(source)

    def _is_eulerian(self):
        """Test if the graph is eulerian."""
        if self.graph.is_directed():
            # we assume that the graph is strongly connected
            for node in self.graph.iternodes():
                if self.graph.indegree(node) != self.graph.outdegree(node):
                    return False
        else:
            # we assume that the graph is connected
            for node in self.graph.iternodes():
                if self.graph.degree(node) % 2 == 1:
                    return False
        return True
```

5.1.3. Algorytm Hierholzera

Dane wejściowe: Multigraf spójny G.

Problem: Znalezienie cyklu Eulera w multigrafie G.

Opis algorytmu: Na początku algorytm wykonuje kopię grafu i ustala wierzchołek startowy. Następnie buduje cykl Eulera składając go z mniejszych cykli, korzystając przy tym ze stosu.

Złożoność obliczeniowa: Złożoność wynosi $O(E)$.

Złożoność pamięciowa: Tu również potrzebna jest kopia grafu, oraz pamięć na stos rzędu $O(E)$.

Listing 5.3. Algorytm Hierholzera.

```
#!/usr/bin/python

from Queue import LifoQueue

class Hierholzer:
    """Finding an Eulerian cycle in a multigraph."""

    def __init__(self, graph):
        """The algorithm initialization."""
        self.graph = graph
        if not self._is_eulerian():
            raise ValueError("the graph is not eulerian")
        self.eulerian_cycle = list()    # list of nodes
        self.graph_copy = self.graph.copy()
        self.stack = LifoQueue()

    def run(self, source=None):
        """Executable pseudocode."""
        if source is None:    # get first random node
```

```
            source = self.graph.iternodes().next()
        self.eulerian_cycle.append(source)
        while True:
            if self.graph_copy.outdegree(source) > 0:
                edge = self.graph_copy.iteroutedges(source).next()
                self.stack.put(source)
                self.graph_copy.del_edge(edge)
                source = edge.target
            else:
                source = self.stack.get()
                self.eulerian_cycle.append(source)
            if self.stack.empty():
                break
        self.eulerian_cycle.reverse()
        del self.stack
        del self.graph_copy

    def _is_eulerian(self):
        """Test if the graph is eulerian."""
        if self.graph.is_directed():
            # we assume that the graph is strongly connected
            for node in self.graph.iternodes():
                if self.graph.indegree(node) != self.graph.outdegree(node):
                    return False
        else:
            # we assume that the graph is connected
            for node in self.graph.iternodes():
                if self.graph.degree(node) % 2 == 1:
                    return False
        return True
```

5.2. Problem chińskiego listonosza

Problem chińskiego listonosza polega na ustaleniu takiej drogi dla listonosza, aby mógł przejść po każdej ulicy w swoim rejonie przynajmniej jeden raz, przy minimalnym koszcie. Problem został pierwszy raz sformułowany w teorii grafów przez chińskiego matematyka, dlatego nazywa się go problemem *chińskiego* listonosza.

Aby rozwiązać problem, należy zbudować graf, którego krawędzie odpowiadają ulicom w rejonie, natomiast wierzchołki skrzyżowaniom ulic. Krawędziom należy nadać wagi odpowiadające kosztom przejścia ulicy. W tak skonstruowanym grafie należy znaleźć drogę o minimalnej sumie wag, która prowadzi przez każdą z ulic przynajmniej jeden raz.

Jeżeli graf posiada cykl Eulera,możliwe wyznaczenie drogi wymagającej przejścia każdej ulicy dokładnie jeden raz. Jeżeli w grafie nie istnieje cykl Eulera, należy skonstruować graf Eulerowski poprzez powielenie wybranych krawędzi zgodnie z algorytmem.

5.2.1. Algorytm dla grafów nieskierowanych

Dane wejściowe: Multigraf nieskierowany G.

Problem: Wyznaczenie w multigrafie drogi przechodzącej przez każdą krawędź przynajmniej jeden raz o minimalnej sumie wag.

Opis algorytmu: Dla grafu który nie jest Eulerowski algorytm konstruuje nowy graf, który składa się z wierzchołków grafu G o nieparzystych stopniach. W nowym grafie wszystkie wierzchołki są ze sobą połączone, a wagi krawędzi odpowiadają minimalnej sumie wag krawędzi łączących te wierzchołki w badanym grafie. Następnie w skonstruowanym grafie wyznaczane są maksymalne skojarzenia i w grafie G dublowane są ścieżki o najmniejszej wadze łączące skojarzone wierzchołki. Tak zmodyfikowany badany graf jest grafem Eulerowskim, a rozwiązaniem problemu jest znalezienie cyklu.

Listing 5.4. Problem chińskiego listonosza dla grafów nieskierowanych.

```
#!/usr/bin/python

class ChineseProblemUndirectedGraph:
    """Chinese Postman Problem for directed graphs."""

    def __init__(self, graph):
        """The algorithm initialization."""
        self.graph = graph

    def run(self):
        """Executable pseudocode."""
        distances = self.count_distances(self.odd_nodes())
        G = self.build_graph(self.odd_nodes(), distances)
        M = MinimumWeightMatching(G)
        M.run()
        pairs = list()
        tmp = set()
        for m in M.pair.values():
            if not set([m.source, m.target]).issubset(tmp):
                tmp.add(m.source)
                tmp.add(m.target)
                pairs.append(list([m.source, m.target]))

        for p in pairs:
            for edge in distances[p[0]].way_to(p[1])["way"]:
                self.graph.add_edge(
                    Edge(edge.source, edge.target, edge.weight))

        E = EulerianCycleDFS(self.graph)
        E.run(0)
        print E.eulerian_cycle

    def odd_nodes(self):
        odd = list()
        for v in self.graph.iternodes():
            if self.graph.degree(v) % 2 != 0:
                odd.append(v)
```

```
        return odd

    def count_distances(self, nodes):
        DJ = dict()
        for node in nodes:
            D = Dijkstra(self.graph, node)
            D.run()
            DJ.update({node : D})
        return DJ

    def build_graph(self, nodes, distances):
        G = MultiGraph(nodes.__len__(), directed=False)
        sources = set()
        for s in nodes:
            sources.add(s)
            for t in nodes:
                if set([t]).issubset(sources):
                    continue
                if s != t:
                    G.add_edge(
                        Edge(s,t,distances[s].d[t]))
        return G
```

5.2.2. Algorytm dla grafów skierowanych

Dane wejściowe: Multigraf nieskierowany G.

Problem: Wyznaczenie w multigrafie drogi przechodzącej przez każdą krawędź przynajmniej jeden raz o minimalnej sumie wag.

Opis algorytmu: Dla grafu który nie jest Eulerowski algorytm konstruuje nowy graf, który jest grafem dwudzielnym zbudowanym z wierzchołków badanego grafu G, takimi że $V(H) = V^-(H) \cup V^+(H)$. Do $V^+(H)$ zaliczamy wierzchołki $v \in V^+(H)$ takie, że $\delta d(v) = d^+(v) - d^-(v) > 0$, a do $V^-(H)$ zaliczamy wierzchołki $v \in V^-(H)$ takie że $\delta d(v) = d^+(v) - d^-(v) < 0$.

Każdy wierzchołek z $V^-(H)$ połączony jest z każdym wierzchołkiem z $V^+(H)$. Wagi krawędzi łączących wierzchołki odpowiadają minimalnej sumie wag krawędzi łączących te wierzchołki w grafie G. W tak skonstruowanym grafie znajdowany jest zbiór krawędzi o sumarycznie najmniejszym koszcie taki, że z wierzchołkiem $v \in V^-(H)$ połączonych jest $\delta d(v)$ krawędzi wychodzących, a z wierzchołkiem $v \in V^+(H)$ połączonych jest $\delta d(v)$ krawędzi wchodzących. Po zdublowaniu w grafie G ścieżek odpowiadających wyznaczonym krawędziom powstaje graf, w którym można wyznaczyć cykl Eulera.

Listing 5.5. Problem chińskiego listonosza dla grafów skierowanych.

```
#!/usr/bin/python

class ChineseProblemDirectedGraph:
    """Chinese Postman Problem for directed graphs."""

    def __init__(self, graph):
```

```
        """The algorithm initialization."""
        self.graph = graph
        self.result = None

    def run(self):
        """Executable pseudocode."""
        nodes = self.find_directed_vertexes()
        sumnodes = list()
        sumnodes.extend(nodes["vIn"])
        sumnodes.extend(nodes["vOut"])
        distances = self.count_distances(sumnodes)
        G = self.build_directed_graph(nodes, distances)
        deltadegrees = dict(
            (v, self.graph.outdegree(v) - self.graph.indegree(v))
            for v in G.iternodes())
        for v in G.iternodes():
            deltadegrees[v] = self.graph.outdegree(v) \
                              - self.graph.indegree(v)
            if deltadegrees[v] > 0:
                innodes = list()
                for tmp in G.iterinedges(v):
                    innodes.append(tmp)
                innodes.sort(key=lambda x: x.weight, reverse=False)

                tmpcounter = 0
                while tmpcounter < deltadegrees[v]:
                    tmpcounter += 1
                    gedge = innodes.pop(0)
                    for edge in \
                        distances[gedge.source].way_to(gedge.target)["way"]:
                        self.graph.add_edge(
                            Edge(edge.source, edge.target, edge.weight))

        E = EulerianCycleDFS(self.graph)
        E.run(0)
        self.result = E.eulerian_cycle

    def count_distances(self, nodes):
        DJ = dict()
        for node in nodes:
            D = Dijkstra(self.graph, node)
            D.run()
            DJ.update({node : D})
        return DJ

    def find_directed_vertexes(self):
        vOut = list()
        vIn = list()
        for v in self.graph.iternodes():
            if self.graph.outdegree(v) - self.graph.indegree(v) > 0:
                vOut.append(v)
            elif self.graph.outdegree(v) - self.graph.indegree(v) < 0:
                vIn.append(v)
        return {"vIn" : vIn, "vOut" : vOut}

    def build_directed_graph(self, nodes, distances):
        G = MultiGraph(
```

```
        nodes["vIn"].__len__() + nodes["vOut"].__len__(),
        directed=True)
    for vIn in nodes["vIn"]:
        for vOut in nodes["vOut"]:
            G.add_edge(Edge(vIn, vOut, distances[vIn].d[vOut]))
    return G
```

6. Zbiory niezależne, skojarzenia i pokrycia

Zbiór niezależny, skojarzenie i pokrycie to pojęcia charakteryzujące pewne podzbiory wierzchołków lub krawędzi, należących do grafu nieskierowanego. Takie podzbiory pojawiają się w różnych problemach decyzyjnych, mających praktyczne znaczenie.

6.1. Zbiory niezależne i kliki

Zbiorem niezależnym (ang. *independent/stable set*) grafu nieskierowanego $G = (V, E)$ nazywamy taki podzbiór zbioru wierzchołków V, że żadne dwa z nich nie są połączone krawędzią z E [4], [9]. Zbiór niezależny jest *maksymalny* (ang. *maximal independent set*), jeżeli nie jest podzbiorem żadnego innego zbioru niezależnego. Zbiór niezależny z największą liczbą wierzchołków nazywamy *największym zbiorem niezależnym* grafu (ang. *maximum independent set*), a jego liczność oznacza się symbolem $\alpha(G)$ (ang. *independence number*).

Przeciwieństwem zbioru niezależnego jest *klika* (ang. *clique*), czyli zbiór wierzchołków wzajemnie sąsiednich. Inaczej mówiąc, klika jest podgrafem pełnym. Klika jest *największa*, jeżeli w grafie nie ma kliki o większej liczbie wierzchołków. Rozmiar największej kliki grafu G oznaczamy $\omega(G)$ (ang. *clique number*). Wierzchołki każdego zbioru niezależnego grafu G są jednocześnie wierzchołkami pewnej kliki w grafie $\bar{G}$, będącym dopełnieniem grafu G.

Problem znalezienia największego zbioru niezależnego jest problemem NP-trudnym, czyli nie są znane algorytmy dokładne o wielomianowej złożoności obliczeniowej. Z tego powodu powstało wiele algorytmów heurystycznych, często o charakterze zachłannym. Problemy wyznaczania największego zbioru niezależnego i największej kliki są równoważne, czyli metoda rozwiązania jednego z nich może być zastosowana do rozwiązania drugiego, po utworzeniu dopełnienia grafu.

Przedstawimy kilka z algorytmów heurystycznych wyznaczania zbiorów niezależnych.

— Metoda *TIS* (ang. *Trivial Independent Set*), do pustego zbioru niezależnego dołączamy dozwolone wierzchołki w kolejności przypadkowej.
— Metoda *SFIS* (ang. *Smallest First Independent Set*), do pustego zbioru niezależnego dołączamy dozwolone wierzchołki o najmniejszym stopniu w pierwotnym grafie G.

— Metoda *MSFIS* (ang. *Modified Smallest First Independent Set*), do pustego zbioru niezależnego dołączamy dozwolone wierzchołki o najmniejszym stopniu *w malejącym grafie indukowanym.*
— Metoda *LLIS* (ang. *Largest Last Independent Set*), zbiór niezależny tworzymy przez usuwanie ze zbioru V wierzchołków o największym stopniu w pierwotnym grafie G, ale tylko tych z sąsiadami w zbiorze niezależnym. Rezultat zwykle nie jest zadowalający.
— Metoda *MLLIS* (ang. *Modified Largest Last Independent Set*), zbiór niezależny tworzymy przez usuwanie ze zbioru V wierzchołków o największym stopniu *w malejącym grafie indukowanym.*

6.1.1. Zastosowanie zbiorów niezależnych

Problem ośmiu hetmanów: Problem polega na rozstawieniu ośmiu hetmanów na tradycyjnej szachownicy 8×8 tak, aby się wzajemnie nie atakowały [10]. Graf dla tego problemu zawiera pola szachownicy jako wierzchołki. Krawędzie łączą takie pola, na których ustawieni hetmani mogą się atakować. Wyznaczone pozycje hetmanów tworzą zbiór niezależny. Istnieje dwanaście istotnie różnych rozwiązań tego problemu. Rozważa się także odmiany problemu dla innych rozmiarów szachownicy.

6.2. Algorytmy heurystyczne z dodawaniem wierzchołków

Algorytmy polegają na kolejnym dodawaniu dozwolonych wierzchołków do tworzonego zbioru niezależnego. Na początku zbiór niezależny jest pusty.

6.2.1. Algorytm TIS do zbiorów niezależnych

Dane wejściowe: Multigraf nieskierowany G.

Problem: Znalezienie zbioru niezależnego dla multigrafu G.

Opis algorytmu: Algorytm tworzy zbiór niezależny poprzez dodawanie do początkowo pustego zbioru wierzchołków grafu, które nie są połączone z żadnym już znajdującym się w zbiorze wierzchołkiem. Kolejność dodawania wierzchołków jest przypadkowa i wynika z działania iteratora po wierzchołkach w grafie. Można rozważyć jeszcze dodatkowe wymieszanie kolejności wierzchołków (random.shuffle).

Złożoność obliczeniowa: W metodzie run pierwsza pętla **for** jest wykonywana $O(V)$ razy. Druga pętla **for** w kolejnych uruchomieniach przebiega wszystkie listy sąsiedztwa, co pozwala wydedukować złożoność liniową algorytmu $O(V + E)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ zbiory independent_set oraz used mają najwyżej $O(V)$ elementów.

Listing 6.1. Algorytm TIS.

```
#!/usr/bin/python

class TrivialIndependentSet:
    """Find a maximal independent set."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        self.independent_set = set()
        self.used = set()

    def run(self):
        """Obliczenia."""
        for source in self.graph.iternodes():
            if source in self.used:
                continue
            self.independent_set.add(source)
            for edge in self.graph.iteroutedges(source):
                self.used.add(edge.target)
```

6.2.2. Algorytm SFIS do zbiorów niezależnych

Dane wejściowe: Multigraf nieskierowany G.

Problem: Znalezienie zbioru niezależnego dla multigrafu G.

Opis algorytmu: Algorytm tworzy zbiór niezależny poprzez dodawanie do początkowo pustego zbioru wierzchołków grafu, które nie są połączone z żadnym już znajdującym się w zbiorze wierzchołkiem. Kolejność dodawania wierzchołków jest ustalana według niemalejących stopni wierzchołków w grafie G.

Złożoność obliczeniowa: W metodzie run wywoływana jest funkcja biblioteczna **sorted** o złożoności $O(V \log V)$. Dalsze operacje są takie jak w algorytmie TIS. Zatem złożoność algorytmu szacujemy na $O(E + V \log V)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ zbiory independent_set oraz used mają wielkość $O(V)$.

Listing 6.2. Algorytm SFIS.

```
#!/usr/bin/python

class SmallestFirstIndependentSet:
    """Find a maximal independent set."""
```

```
    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        self.independent_set = set()
        self.used = set()

    def run(self):
        """Obliczenia."""
        for source in sorted(self.graph.iternodes(),
            key=self.graph.degree):
            if source in self.used:
                continue
            self.independent_set.add(source)
            for edge in self.graph.iteroutedges(source):
                self.used.add(edge.target)
```

6.2.3. Algorytm MSFIS do zbiorów niezależnych

Dane wejściowe: Multigraf nieskierowany G.

Problem: Znalezienie zbioru niezależnego dla multigrafu G.

Opis algorytmu: Algorytm tworzy zbiór niezależny poprzez dodawanie do początkowo pustego zbioru wierzchołków grafu, które nie są połączone z żadnym już znajdującym się w zbiorze wierzchołkiem. Do zbioru niezależnego dołączane są wierzchołki o najmniejszym stopniu w malejącym grafie indukowanym.

Złożoność obliczeniowa: W metodzie run stworzenie słownika degree_dict zajmuje czas $O(V)$. Pętla **while** wykonywana jest $O(V)$ razy. W pętli wywoływana jest funkcja biblioteczna **min** o złożoności $O(V)$. Dalej występuje podwójna pętla **for** po krawędziach wychodzących, co oszacujemy przez czas $O(\Delta^2)$. Zatem złożoność algorytmu szacujemy na $O(V^2 + V\Delta^2)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ zbiory independent_set, free, oraz słownik degree_dict mają wielkość $O(V)$.

Listing 6.3. Algorytm MSFIS.

```
#!/usr/bin/python

class ModifiedSmallestFirstIndependentSet:
    """Find a maximal independent set."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
```

```
            raise ValueError("the graph is directed")
        self.graph = graph
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        self.independent_set = set()
        self.free = set(self.graph.iternodes())

    def run(self):
        """Obliczenia."""
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())    # O(V) time
        while len(self.free) > 0:
            source = min((node for node in self.free),
                key=degree_dict.__getitem__)    # O(V) time
            self.independent_set.add(source)
            self.free.remove(source)
            for edge in self.graph.iteroutedges(source):
                self.free.discard(edge.target)
                for edge2 in self.graph.iteroutedges(edge.target):
                    degree_dict[edge2.target] -= 1
```

6.3. Algorytmy heurystyczne z usuwaniem wierzchołków

Algorytmy polegają na stopniowym usuwaniu wierzchołków z grafu do momentu uzyskania zbioru niezależnego.

6.3.1. Algorytm LLIS do zbiorów niezależnych

Dane wejściowe: Multigraf nieskierowany G.

Problem: Znalezienie zbioru niezależnego dla multigrafu G.

Opis algorytmu: Algorytm tworzy zbiór niezależny poprzez usuwanie ze zbioru independent_set wierzchołków o największym stopniu w pierwotnym grafie G, ale usuwane są tylko wierzchołki z sąsiadami w zbiorze niezależnym. Na początku w zbiorze independent_set znajdują się wszystkie wierzchołki grafu.

Złożoność obliczeniowa: W metodzie _is_independent wykonywana jest pętla **for** $O(E)$ razy, a wewnątrz niej mamy sprawdzenie należenia do zbioru, które jest robione w stałym czasie.

W metodzie run mamy jednorazowe sortowanie wierzchołków funkcją **sorted** w czasie $O(V \log V)$. Następnie pętla **for** przechodzi przez wszystkie wierzchołki grafu G, a przy tym uruchamia metodę _is_independent i zwykle wy-

konuje pętle po krawędziach wychodzących. Zatem złożoność algorytmu szacujemy na $O(VE + V \log V)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ zbiór independent_set ma wielkość $O(V)$.

Listing 6.4. Algorytm LLIS.

```
#!/usr/bin/python

class LargestLastIndependentSet:
    """Find a maximal independent set."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        self.independent_set = set(self.graph.iternodes())

    def run(self):
        """Obliczenia."""
        for source in sorted(self.graph.iternodes(),
            key=self.graph.degree, reverse=True):
                if self._is_independent():
                    break
                else:
                    for edge in self.graph.iteroutedges(source):
                        if edge.target in self.independent_set:
                            self.independent_set.discard(source)

    def _is_independent(self):
        """Test zbioru niezaleznego."""
        for edge in self.graph.iteredges():
            if (edge.source in self.independent_set and
                edge.target in self.independent_set):
                    return False
        return True
```

6.3.2. Algorytm MLLIS do zbiorów niezależnych

Dane wejściowe: Multigraf nieskierowany G.

Problem: Znalezienie zbioru niezależnego dla multigrafu G.

Opis algorytmu: Algorytm tworzy zbiór niezależny poprzez usuwanie ze zbioru independent_set wierzchołków o największym stopniu w malejącym grafie indukowanym. Na początku w zbiorze independent_set znajdują się wszystkie wierzchołki grafu.

Złożoność obliczeniowa: Metoda _is_independent zajmuje czas $O(E)$ i jest wykonywana w każdej pętli **while**. Wewnątrz pętli szukanie maksimum zaj-

muje czas $O(V)$. Wszystkie wykonania pętli **for** można ograniczyć przez $O(E)$. Zatem złożoność algorytmu szacujemy na $O(V^2 + VE)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ zbiór independent_set ma wielkość $O(V)$.

Listing 6.5. Algorytm MLLIS.

```
#!/usr/bin/python

class ModifiedLargestLastIndependentSet:
    """Find a maximal independent set."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        self.independent_set = set(self.graph.iternodes())

    def run(self):
        """Executable pseudocode."""
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())   # O(V) time
        while not self._is_independent():
            source = max((node for node in self.independent_set),
                key=degree_dict.__getitem__)
            self.independent_set.remove(source)
            for edge in self.graph.iteroutedges(source):
                degree_dict[edge.target] -= 1

    def _is_independent(self):
        """Test zbioru niezaleznego."""
        for edge in self.graph.iteredges():
            if (edge.source in self.independent_set and
                edge.target in self.independent_set):
                    return False
        return True
```

6.4. Pokrycie wierzchołkowe

Pokryciem wierzchołkowym (ang. *vertex/node cover*) grafu nieskierowanego $G = (V, E)$ nazywamy taki podzbiór C zbioru wierzchołków V, że każda krawędź z E jest incydentna do co najmniej jednego wierzchołka z C [12]. Minimalne pokrycie wierzchołkowe nazywamy *optymalnym pokryciem wierzchołkowym*. Trywialnym i największym pokryciem wierzchołkowym jest cały zbiór V.

Problem znalezienia najmniejszego pokrycia wierzchołkowego jest problemem optymalizacyjnym NP-trudnym. W wersji decyzyjnej problem polega

na stwierdzeniu, czy w danym grafie istnieje pokrycie wierzchołkowe o danej liczbie wierzchołków k. Jest to problem NP-zupełny.

Twierdzenie: Zbiór C jest pokryciem wierzchołkowym wtedy i tylko wtedy, gdy jego dopełnienie $V \backslash C$ jest zbiorem niezależnym.

Twierdzenie (Gallai, 1959): Liczba wierzchołków grafu jest równa jego minimalnemu pokryciu wierzchołkowemu dodać rozmiar największego zbioru niezależnego.

Twierdzenie (König, 1931): Dla dowolnego grafu dwudzielnego liczność najmniejszego pokrycia wierzchołkowego jest równa liczności największego skojarzenia. Dzięki temu twierdzeniu problem najmniejszego pokrycia wierzchołkowego dla grafów dwudzielnych może być rozwiązany w czasie wielomianowym.

7. Kolorowanie multigrafów

Kolorowanie multigrafu polega na przypisaniu określonym elementom składowym multigrafu (wierzchołkom, krawędziom lub ścianom) wybranych kolorów (etykiet) według ściśle określonych reguł [15]. W praktyce zamiast kolorów stosuje się kolejne liczby całkowite, zaczynając od zera lub jedynki.

7.1. Kolorowanie wierzchołków

Kolorowanie wierzchołków multigrafu (ang. *vertex coloring*) polega na przyporządkowaniu wierzchołkom kolorów w taki sposób, aby każda krawędź miała końce różnych kolorów (takie jest kolorowanie wierzchołkowe legalne, dozwolone). Z tego wynika, że zabronione jest występowanie pętli w multigrafach, a krawędzie wielokrotne w pojedyńczym pęku bez straty ogólności mogą być zastąpione przez pojedyńczą krawędź.

Multigraf jest *k-kolorowalny wierzchołkowo*, jeżeli istnieje legalne kolorowanie wierzchołków wykorzystujące k kolorów. Kolorowanie optymalne wierzchołków multigrafu zawiera najmniejszą możliwą liczbę kolorów, którą nazywamy *liczbą chromatyczną* χ lub χ_0. Problem kolorowania wierzchołków jest NP-trudny. Problem decyzyjny polegający na określeniu, czy graf jest k-kolorowalny wierzchołkowo jest NP-zupełny dla $k \geqslant 3$ [4]. Łatwo jest stwierdzić, czy graf jest 2-kolorowalny. Wystarczy przeszukać graf algorytmem BFS lub DFS, przydzielając na przemian dwa kolory. Problem kolorowania wierzchołkowego (i kilka innych) może być rozwiązany w czasie wielomianowym dla *grafu doskonałego* (ang. *perfect graph*) [13].

Twierdzenie: Jeżeli G jest grafem prostym, to jest $(\Delta + 1)$-kolorowalny wierzchołkowo [3]. Dowód przeprowadza się przez indukcję względem liczby wierzchołków grafu G. Najprostszy algorytm zachłanny nie przydzieli wierzchołkom więcej niż $\Delta + 1$ kolorów.

Twierdzenie (Brooks, 1941): Jeżeli G jest spójnym grafem prostym, nie będącym grafem pełnym K_n, oraz $\Delta \geqslant 3$ (wyłączamy grafy cykliczne C_n), to graf G jest Δ-kolorowalny wierzchołkowo [3].

Twierdzenie: Jeżeli G jest grafem prostym z m krawędziami, to zachodzi zależność [4]

$$\chi(G) \leqslant \frac{1}{2} + \sqrt{2m + \frac{1}{4}}. \tag{7.1}$$

Twierdzenie: Jeżeli G jest grafem prostym z m krawędziami, to zachodzi zależność [4]

$$\chi(G) \leqslant \sqrt{2m} + 1. \tag{7.2}$$

7.1.1. Kolorowanie wierzchołków grafu planarnego

W przypadku grafów planarnych istnieje szereg silnych twierdzeń dotyczących kolorowania wierzchołkowego.

Twierdzenie: Każdy planarny graf prosty jest 6-kolorowalny wierzchołkowo [3]. Dowód przeprowadza się przez indukcję względem liczby wierzchołków grafu. Korzysta się przy tym z twierdzenia, że każdy graf planarny prosty ma wierzchołek stopnia co najwyżej 5.

Twierdzenie o pięciu barwach (Heawood, 1890): Każdy planarny graf prosty jest 5-kolorowalny wierzchołkowo [3]. Dowód przeprowadza się przez indukcję względem liczby wierzchołków grafu.

Twierdzenie o czterech barwach (Appel, Haken, 1976): Każdy planarny graf prosty jest 4-kolorowalny wierzchołkowo [3]. Dowód był częściowo wykonany przy użyciu komputera, którym analizowano 1936 przypadków szczególnych.

7.1.2. Przykłady kolorowania wierzchołków

Warto podać kilka przykładów grafów prostych i ich kolorowania wierzchołkowego.

— Dla grafu pełnego K_n ($\Delta = n - 1$) mamy $\chi = \Delta + 1 = n$.
— Dla grafu dwudzielnego (np. drzewa lub gwiazdy $K_{1,s}$) mamy $\chi = 2$.
— Dla grafu cyklicznego C_n ($\Delta = 2$) mamy $\chi = 2$ (n parzyste) lub $\chi = 3$ (n nieparzyste).
— Dla grafu kołowego W_n ($\Delta = n - 1$) mamy $\chi = 4$ (n parzyste) lub $\chi = 3$ (n nieparzyste).

Widać, że liczba chromatyczna grafów prostych może wynosić od 1 (graf bez krawędzi) do n lub, inaczej mówiąc, od 1 do $\Delta + 1$. Jeżeli wyłączymy grafy pełne K_n i grafy cykliczne C_n, to liczba potrzebnych kolorów nie przekroczy Δ (twierdzenie Brooksa).

7.1.3. Zastosowanie kolorowania wierzchołków

Zastosowanie kolorowania wierzchołków polega na modelowaniu sytuacji konfliktu zasobów i poszukiwania rozwiązań, w których unika się tych konfliktów.

Kolejkowanie: Mamy zbiór zadań (wierzchołki), którym należy przydzielić odcinek czasu dla ich wykonania. Dwa zadania nie mogą być wykonane w tym samym odcinku czasu, jeżeli są w konflikcie (krawędź), np. potrzebują dostępu do tych samych zasobów. Szukamy najmniejszej liczby odcinków czasu potrzebnej do wykonania zadań.

Alokacja rejestrów: Kolory są rejestrami procesora, wierzchołki są wartościami, które chcemy przechowywać w rejestrach. Krawędzie modelują konflikty, kiedy dwie wartości są potrzebne w tym samym czasie. Kompilator

buduje tzw. graf interferencji, a zadaniem jest wyznaczenie liczby potrzebnych rejestrów.

7.1.4. Dokładny algorytm kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Najpierw kolorujemy wierzchołki wszystkimi możliwymi kombinacjami dwóch pierwszych kolorów. Przy kazdej kombinacji sprawdzamy warunek pokolorowania. Jeżeli jest spełniony, to kończymy. W kolejnych krokach zwiększamy liczbę kombinacji kolorów do 3, 4, ..., n i powtarzamy powyższy krok aż do znalezienia takiej kombinacji, która będzie spełniała warunek pokolorowania. Problem tworzenia kombinacji kolorów poszczególnych wierzchołków jest rozwiązany metodą licznika.

Złożoność: Algorytm jest powolny, ponieważ jego złożoność jest wykładnicza klasy $O(2^n)$.

Listing 7.1. Moduł nodecolorexact.

```
#!/usr/bin/python

class ExactNodeColoring:
    """Algorytm dokladny kolorowania wierzcholkow (powolny)."""
    # Idea taka jak w
    # http://edu.i-lo.tarnow.pl/inf/alg/001_search/0142.php
    # ale ma optymalizacje przez pomijanie sprawdzania kombinacji,
    # ktore byly sprawdzane przy mniejszej liczbie kolorow.

    def __init__(self, graph):
        """inicjalizacja algorytmu."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, 0) for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        base = 1
        is_colored = False
        while not is_colored:
            is_colored = True
            # Sprawdzenie poprawnosci kolorowania.
            for source in self.graph.iternodes():
                for edge in self.graph.iteroutedges(source):
                    if self.color[source] == self.color[edge.target]:
                        is_colored = False
                        break
            if not is_colored:
```

```
        roll = False
        for node in self.color:
            if self.color[node] == base - 1:
                roll = True
                self.color[node] = 0
            else:
                roll = False
                self.color[node] += 1
                break
        if roll:
            base += 1
            self.color = dict(((node, 0)
                for node in self.graph.iternodes()))
```

7.2. Algorytmy sekwencyjne kolorowania wierzchołków

Wyznaczanie optymalnego kolorowania wierzchołków jest czasochłonne, dlatego dla większych grafów stosuje się algorytmy heurystyczne, które są szybkie, ale nie gwarantują optymalnego rozwiązania. Algorytmy heurystyczne bada się pod względem złożoności obliczeniowej, ale także określa się pewne dodatkowe charakterystyki, które mówią o tym jak bardzo niedokładny może być dany algorytm, tzn. jak bardzo jego kolorowanie może odbiegać od optymalnego. Jedna z takich charakterystyk jest *najmniejszy graf trudny do pokolorowania*, dla którego algorytm zawsze znajduje nieoptymalne kolorowanie.

Do najprostszych heurystycznych schematów kolorowania należą *algorytmy sekwencyjne* [4]. W tego typu algorytmach wierzchołki są przeglądane według zadanej kolejności, specyficznej dla konkretnego wariantu algorytmu. Każdemu z przeglądanych wierzchołków przydziela się jeden z dopuszczalnych kolorów. Algorytmy sekwencyjne nigdy nie użyją więcej niż $\Delta + 1$ kolorów.

Algorytmy sekwencyjne są często uzupełniane *wymianą kolorów* (ang. *color interchange*). Polega ona na tym, że jeżeli w następnym kroku zachłannym potrzebny jest nowy kolor dla wierzchołka, to podejmuje się próbę przestawienia kolorów w taki sposób, aby odzyskać jeden z dotychczasowych kolorów do pomalowania danego wierzchołka. Wymianę kolorów można zaimplementować tak, aby pracowała w czasie $O(E)$.

Poniżej podamy najbardziej znane warianty przeglądania wierzchołków.

— Metoda *US* (ang. *Unordered Sequential*), kolejność wierzchołków wynika ze sposobu, w jaki implementacja odczytuje wierzchołki z grafu. Metoda może być zaimplementowana w czasie $O(V + E)$.

— Metoda *RS* (ang. *Random Sequential*), pseudolosowa kolejność wierzchołków. Metoda może być zaimplementowana w czasie $O(V + E)$.

— Metoda *LF* (ang. *Largest First*), wierzchołki są przeglądane i kolorowane według nierosnących stopni (Welsh, Powell). Metoda może być zaimplementowana w czasie $O(V + E)$.

— Metoda *SL* (ang. *Smallest Last*), jako ostatni jest kolorowany wierzchołek v_n o najmniejszym stopniu w grafie pierwotnym G, jako przedostatni jest

wybierany wierzchołek v_{n-1} o najmniejszym stopniu w grafie $G\backslash\{v_n\}$, itd. (Matula). Dla grafów planarnych metoda nie użyje więcej niż 6 kolorów. Jest to wniosek z twierdzenia, że w grafie planarnym wierzchołek o najniższym stopniu jest stopnia co najwyżej 5. Wystarczy w każdym kroku usuwać z grafu właśnie taki wierzchołek. Metoda może być zaimplementowana w czasie $O(V + E)$.

— Metoda *CS* (ang. *Connected Sequential*), bazuje na obserwacji, że wierzchołki sąsiednie do pokolorowanych powinny być kandydatami do pokolorowania. Metoda może być zaimplementowana w czasie $O(V + E)$, np. z wykorzystaniem algorytmu BFS lub DFS.

— Metoda *DSATUR* lub *SLF* (ang. *Saturation Largest First*), w każdym kroku do kolorowania wybierany jest wierzchołek o największym tzw. *stopniu nasycenia*, czyli liczbie różnych kolorów przydzielonych sąsiadom danego wierzchołka (to nie jest to samo co liczba sąsiadów z przydzielonym kolorem!). Jeżeli jest kilku kandydatów o tym samym stopniu nasycenia, to wybierany jest wierzchołek z największą liczbą krawędzi wychodzących. Algorytm *DSATUR* dla grafów dwudzielnych jest optymalny [16].

7.2.1. Algorytm US kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Wierzchołki kolorujemy w kolejności wyznaczonej przez implementację. Dla każdego wierzchołka kolor ustalamy jako najniższy kolor, który nie jest użyty przez żadnego z jego sąsiadów (metoda zachłanna).

Złożoność obliczeniowa: W metodzie run petla **for** wykonuje $|V|$ razy metodę _greedy_color. W metodzie _greedy_color mamy pętlę po krawędziach wychodzących, ograniczoną przez Δ, oraz pętlę po $|V|$ kolorach, która jednak przebiegnie najwyżej $\Delta + 1$ razy. Stąd dostajemy szacowanie złożoności $O(V\Delta)$. Można jeszcze zauważyć, że wszystkie uruchomienia pętli po krawedziach przebiegną wszystkie listy sąsiedztwa grafu w czasie $O(E)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ obok słownika z kolorami wierzchołków przechowywana jest lista used z kolorami zajętymi przez sąsiadów.

Listing 7.2. Algorytm US z modułu nodecolorus.

```
#!/usr/bin/python

class UnorderedSequentialNodeColoring:
    """Algorytm US (Unordered Sequential) kolorowania
     wierzcholkow w kolejnosci wyznaczonej przez implementacje."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
```

```
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, None)
                          for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        for source in self.graph.iternodes():
            self._greedy_color(source)

    def _greedy_color(self, source):
        """Przydziela i zwraca pierwszy dostepny kolor."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for i in xrange(n):
            if not used[i]:
                self.color[source] = i
                break
        return i
```

7.2.2. Algorytm RS kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Wierzchołki kolorujemy w kolejności pseudolosowej. Dla każdego wierzchołka kolor ustalamy jako najniższy kolor, który nie jest użyty przez żadnego z jego sąsiadów (metoda zachłanna).

Złożoność obliczeniowa: Złożoność jest taka jak dla algorytmu US, czyli $O(V\Delta)$. W metodzie run uzyskuje się pseudolosową kolejność wierzchołków za pomocą metody random.shuffle z biblioteki standardowej Pythona. Wykorzystany jest przy tym algorytm Fishera-Yatesa o złożoności liniowej $O(V)$ [17].

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ obok słownika z kolorami wierzchołków przechowywana jest lista used z kolorami zajętymi przez sąsiadów.

Listing 7.3. Algorytm RS z modułu nodecolorrs.

```
#!/usr/bin/python

import random
```

```
class RandomSequentialNodeColoring:
    """Algorytm RS (Random Sequential) kolorowania
     wierzcholkow w kolejnosci pseudolosowej."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, None)
                          for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        node_list = list(self.graph.iternodes())
        random.shuffle(node_list)
        for source in node_list:
            self._greedy_color(source)

    def _greedy_color(self, source):
        """Przydziela i zwraca pierwszy dostepny kolor."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for i in xrange(n):
            if not used[i]:
                self.color[source] = i
                break
        return i
```

7.2.3. Algorytm LF kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Wierzchołki kolorujemy w kolejności według nierosnących stopni wierzchołków. Dla każdego wierzchołka kolor ustalamy jako najniższy kolor, który nie jest użyty przez żadnego z jego sąsiadów.

Złożoność obliczeniowa: Złożoność metody `_greedy_color` jest $O(\Delta)$. Jest ona wykonywana w metodzie `run` $|V|$ razy. Jednorazowo wywoływana jest również funkcja biblioteczna **sorted** o złożoności $O(V \log V)$. Zatem złożo-

ność algorytmu jest $O(\Delta V + V \log V)$. Złożoność sortowania można poprawić stosując sortowanie kubełkowe.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$ ponieważ wykorzystany jest słownik color o złożoności $O(V)$, oraz lista used kolorów wykorzystanych przez sąsiadów o złożoności $O(V)$.

Listing 7.4. Algorytm LF z modułu nodecolorlf.

```
#!/usr/bin/python

class LargestFirstNodeColoring:
    """Algorytm LF (Largest First) kolorowania
    wierzcholkow wedlug nierosnacych stopni."""
    def __init__(self, graph):
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, None)
              for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        for source in sorted(self.graph.iternodes(),
            key=self.graph.degree, reverse=True):
                self._greedy_color(source)

    def _greedy_color(self, source):
        """Przydziela i zwraca pierwszy dostepny kolor."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for i in xrange(n):
            if not used[i]:
                self.color[source] = i
                break
        return i
```

7.2.4. Algorytm SL kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Działanie algorytmu podzielone jest na dwie fazy. W pierwszej fazie znajduje wierzchołek o najmniejszym stopniu i usuwa go z grafu. Operacja ta jest powtarzana na zmodyfikowanym grafie, dopóki graf nie jest pusty. Kolejność znajdowania wierzchołków jest zapisywana. W drugiej

fazie wierzchołki są kolorowane według kolejności ustalonej w fazie pierwszej, zaczynając od wierzchołków usuwanych później. Dla każdego wierzchołka kolor ustalamy jako najniższy kolor, który nie jest użyty przez żadnego z jego sąsiadów.

Złożoność obliczeniowa: Złożoność metody _greedy_color jest taka jak we wszystkich powyższych algorytmach, czyli $O(\Delta)$.

W pierwszej pętli **for** metody run, która ma $|V|$ iteracji, wywoływana jest biblioteczna funkcja **min** o złożoności $O(V)$ oraz pętla **for** po krawędziach wychodzących z wierzchołka o złożoności (Δ). Zatem cała pierwsza pętla **for** ma złożoność $O(V^2 + \Delta V)$.

Druga pętla **for** ma złożoność $O(\Delta V)$, więc złożoność algorytmu jest $O(V^2 + \Delta V)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$ ponieważ wykorzystano kolekcje, z których każda ma złożoność $O(V)$.

Listing 7.5. Algorytm SL z modułu nodecolorsl.

```
#!/usr/bin/python

class SmallestLastNodeColoring:
    """Algorytm SL (Smallest Last) kolorowania wierzcholkow."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, None)
                          for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        n = self.graph.v()
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())
        in_queue = dict((node, True)
            for node in self.graph.iternodes())
        ordering = list()
        for step in xrange(n):
            source = min((node for node in self.graph.iternodes()
                if in_queue[node]), key=lambda x: degree_dict[x])
            ordering.append(source)
            in_queue[source] = False
            for edge in self.graph.iteroutedges(source):
                degree_dict[edge.target] -= 1
        for source in reversed(ordering):
            self._greedy_color(source)

    def _greedy_color(self, source):
        """Przydziela i zwraca pierwszy dostepny kolor."""
```

```
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for i in xrange(n):
            if not used[i]:
                self.color[source] = i
                break
        return i
```

7.2.5. Algorytm CS kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: W algorytmie pierwsze w kolejce do pokolorowania są wierzchołki będące sąsiadami już pokolorowanych wierzchołków. Ustalenie kolejności kolorowania wierzchołków odbywa się za pomocą algorytmu BFS. Dla każdego wierzchołka kolor ustalamy jako najniższy kolor, który nie jest użyty przez żadnego z jego sąsiadów.

Złożoność obliczeniowa: Złożoność metody _greedy_color jest taka jak we wszystkich powyższych algorytmach, czyli $O(\Delta)$. Algorytm BFS wykorzystywany w metodzie run do ustalenia kolejności wierzchołków ma złożoność $O(V+E)$, a pętla **for** ma złożoność $O(\Delta V)$. Stąd cały algorytm ma złożoność $(\Delta V + E)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$ ponieważ wykorzystane kolekcje mają złożoność $O(V)$.

Listing 7.6. Algorytm CS z modułu nodecolorcs.

```
#!/usr/bin/python

class ConnectedSequentialNodeColoring:
    """Find a connected sequential (CS) node coloring."""

    def __init__(self, graph):
        """Inicjalizuje algorytm."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, None) for node in self.graph.internodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        ordering = list()
```

```
        algorithm = SimpleBFS(self.graph)
        algorithm.run(pre_action=lambda node: ordering.append(node))
        for source in ordering:
            self._greedy_color(source)

    def _greedy_color(self, source):
        """Przydziela i zwraca pierwszy dostepny kolor."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for i in xrange(n):
            if not used[i]:
                self.color[source] = i
                break
        return i
```

7.2.6. Algorytm DSATUR kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Algorytm w każdym kroku do kolorowania wybiera wierzchołek o największym tzw. stopniu nasycenia, czyli liczbie różnych kolorów przydzielonych sąsiadom danego wierzchołka. Jeżeli jest kilku kandydatów o tym samym stopniu nasycenia, to wybierany jest wierzchołek z największą liczbą krawędzi wychodzących. Dla każdego wierzchołka kolor ustalamy jako najniższy kolor, który nie jest użyty przez żadnego z jego sąsiadów.

Złożoność obliczeniowa: Metoda _greedy_color_with_saturation ma konstrukcję podobną _greedy_color, zatem złożoność ma $O(\Delta)$. W metodzie run pętla **for** funkcja biblioteczna **max** ma złożoność $O(V)$, metoda _greedy_color_with_saturation ma złożoność $O(\Delta)$, zatem złożoność algorytmu jest $O(V^2 + \Delta V)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V^2)$ ponieważ słownik saturation ma $O(V)$ elementów, z których każdy jest zbiorem rozmiaru $O(V)$.

Listing 7.7. Algorytm DSATUR z modułu nodecolords.

```
#!/usr/bin/python

class DSATURNodeColoring:
    """Wyznacza kolorowanie wierzcholkow metoda DSATUR."""

    def __init__(self, graph):
        """Inicjalizuje algorytm."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
```

```
        self.color = dict((node, None)
            for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        self.saturation = dict((node, set())
            for node in self.graph.iternodes())

    def run(self):
        """Obliczenia."""
        n = self.graph.v()
        for step in xrange(n):
            source = max((node for node in self.graph.iternodes()
                if self.color[node] is None), key=lambda x:
                n * len(self.saturation[x]) + self.graph.degree(x))
            self._greedy_color_with_saturation(source)

    def _greedy_color_with_saturation(self, source):
        """Zwraca pierwszy dostepny kolor."""
        for i in xrange(self.graph.v()):
            if i not in self.saturation[source]:
                self.color[source] = i
                break
        for edge in self.graph.iteroutedges(source):
            self.saturation[edge.target].add(self.color[source])
        return i
```

7.3. Algorytmy z wymianą kolorów

Wymiana kolorów zwykle prowadzi do poprawy kolorowania. Wymiana jest uruchamiana wtedy, gdy w następnym kroku zachłannym potrzebny jest nowy kolor dla wierzchołka v. Istnieje wiele sposobów zrealizowania wymiany kolorów.

Pierwszy sposób polega na tym, aby dla każdego kolorowego *najbliższego sąsiada* wierzchołka v spróbować zmienić kolor w ramach dotychczas używanej puli kolorów. Może to prowadzić do zwolnienia koloru dla wierzchołka v.

Inny sposób to coś w rodzaju błądzenia przypadkowego po kolorowych wierzchołkach, o początku w wierzchołku v, połączone z próbą ich przekolorowania (*annealing interchange*) [18]. Trzeba eksperymentalnie ustalić dozwoloną liczbę kroków błądzenia.

Poniżej podamy najbardziej znane wersje algorytmów z wymiana kolorów.

— Metoda *RSI* (ang. *Random Sequential with Interchange*). Metoda może być zaimplementowana w czasie $O(V \cdot E)$.
— Metoda *LFI* (ang. *Largest First with Interchange*). Metoda może być zaimplementowana w czasie $O(V \cdot E)$.
— Metoda *SLI* (ang. *Smallest Last with Interchange*). Metoda może być zaimplementowana w czasie $O(V \cdot E)$.
— Metoda *CSI* (ang. *Connected Sequential with Interchange*),
— Metoda *DSI* (ang. *DSATUR with Interchange*).

7.3.1. Algorytm USI kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Wierzchołki kolorujemy w kolejności wynikającej z implementacji. Dla każdego wierzchołka kolor ustalamy jako najniższy kolor, który nie jest użyty przez żadnego z jego sąsiadów (metoda zachłanna). Jednak przed przydzieleniem koloru wierzchołkowi, próbujemy sąsiadom tego wierzchołka pozamieniać kolory z ich sąsiadami tak, aby może zwolnić jakiś z kolorów wcześniej użytych dla sąsiadów.

Złożoność obliczeniowa: Algorytm jest modyfikacją US o metodę _recolor. Bazując na wcześniejszych rozważaniach, pętla po krawędziach wychodzących jest ograniczona przez Δ wykonań, zatem złożoność metody _recolor jest $O(\Delta^2 + \Delta V)$.

W metodzie _greedy_color_width_interchange pierwsza pętla **for** ma złożoność $O(\Delta)$, druga $O(V)$. Dodatkowo mamy co najwyżej jednokrotne wywołanie metody _recolor o złożoności $O(\Delta^2 + \Delta V)$ oraz wywołanie metody _greedy_color o złożoności Δ). Zatem w metodzie _greedy_color_width_interchange dominująca jest złożoność metody _recolor, czyli $O(\Delta^2 + \Delta V)$.

Całkowita złożoność algorytmu jest szacowana na $O(\Delta^2 V + \Delta V^2)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ obok słownika z kolorami wierzchołków przechowywane są listy used z kolorami zajętymi przez sąsiadów.

Listing 7.8. Algorytm USI z modułu nodecolorusi.

```
#!/usr/bin/python

class USINodeColoring:
    """Algorytm US z wymiana kolorow."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu"""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, None)
            for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        for source in self.graph.iternodes():
            self._greedy_color_with_interchange(source)

    def _greedy_color_with_interchange(self, source):
```

```
        """Przydziela pierwszy dostepny kolor, sprawdzajac
        czy nie mozna zwolnic ktoregos z kolorow sasiadow."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        i_max = 0
        i_free = None
        for i in xrange(n):
            if used[i]:
                i_max = i
            elif i_free is None:
                i_free = i
        if i_free > i_max:
            self._recolor(source, i_max)
        self._greedy_color(source)

    def _recolor(self, source, i_max):
        """Proba zamiany kolorow"""
        n = self.graph.v()
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is None:
                continue
            used = [False] * n
            used[self.color[edge.target]] = True
            for edge2 in self.graph.iteroutedges(edge.target):
                if self.color[edge2.target] is not None:
                    used[self.color[edge2.target]] = True
            for i in xrange(n):
                if not used[i]:
                    new_color = i
                    break
            if new_color <= i_max:
                old_color = self.color[edge.target]
                self.color[edge.target] = new_color
                return old_color
        return i_max + 1

    def _greedy_color(self, source):
        """Przydziela i zwraca pierwszy dostepny kolor."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for i in xrange(n):
            if not used[i]:
                self.color[source] = i
                break
        return i
```

7.3.2. Algorytm RSI kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Wierzchołki kolorujemy w kolejności pseudolosowej. Dla każdego wierzchołka kolor ustalamy jako najniższy kolor, który nie jest użyty przez żadnego z jego sąsiadów (metoda zachłanna). Jednak przed przydzieleniem koloru wierzchołkowi, próbujemy sąsiadom tego wierzchołka pozamieniać kolory z ich sąsiadami tak, aby może zwolnić jakiś z kolorów wcześniej użytych dla sąsiadów.

Złożoność obliczeniowa: Algorytm jest modyfikacją RS o metodę _recolor. Bazując na wcześniejszych rozważaniach, szacujemy całkowitą złożoność algorytmu na $O(\Delta^2 V + \Delta V^2)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$, ponieważ obok słownika z kolorami wierzchołków przechowywane są listy used z kolorami zajętymi przez sąsiadów.

Listing 7.9. Algorytm RSI z modułu nodecolorrsi.

```
#!/usr/bin/python

class RSINodeColoring:
    """Algorytm RS z wymiana kolorow"""

    def __init__(self, graph):
        """Inicjalizacja algorytmu"""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict(
            (node, None) for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia"""
        node_list = list(self.graph.iternodes())
        random.shuffle(node_list)
        for source in node_list:
            self._greedy_color_with_interchange(source)

    def _greedy_color_with_interchange(self, source):
        """Przydziela pierwszy dostepny kolor, sprawdzajac
        czy nie mozna zwolnic ktoregos z kolorow sasiadow"""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
```

```
        i_max = 0
        i_free = None
        for i in xrange(n):
            if used[i]:
                i_max = i
            elif i_free is None:
                i_free = i
        if i_free > i_max:
            self._recolor(source, i_max)
        self._greedy_color(source)

    def _recolor(self, source, i_max):
        """Proba zamiany kolorow"""
        n = self.graph.v()
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is None:
                continue
            used = [False] * n
            used[self.color[edge.target]] = True
            for edge2 in self.graph.iteroutedges(edge.target):
                if self.color[edge2.target] is not None:
                    used[self.color[edge2.target]] = True
            for i in xrange(n):
                if not used[i]:
                    new_color = i
                    break
            if new_color <= i_max:
                old_color = self.color[edge.target]
                self.color[edge.target] = new_color
                return old_color
        return i_max + 1

    def _greedy_color(self, source):
        """Przydziela i zwraca pierwszy dostepny kolor."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for i in xrange(n):
            if not used[i]:
                self.color[source] = i
                break
        return i
```

7.3.3. Algorytm CSI kolorowania wierzchołków

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie wierzchołków multigrafu G.

Opis algorytmu: Wierzchołki kolorujemy w kolejności wyznaczonej przez BFS, czyli kolorujemy sąsiadów wierzchołków z przydzielonymi kolorami.

Jeżeli potrzeby jest nowy kolor, to uruchamiana jest procedura wymiany kolorów, a następnie mamy powrót do kolorowania zachłannego.

Złożoność obliczeniowa: Algorytm jest modyfikacją CS o metodę _recolor. Algorytm BFS wykorzystywany w metodzie run do ustalenia kolejności wierzchołków ma złożoność $O(V + E)$. Bazując na wcześniejszych rozważaniach, szacujemy całkowitą złożoność algorytmu na $O(\Delta^2 V + \Delta V^2 + E)$.

Złożoność pamięciowa: Złożoność pamięciowa jest $O(V)$ ponieważ wykorzystane kolekcje mają złożoność $O(V)$.

Listing 7.10. Algorytm CSI z modułu nodecolorcsi.

```
#!/usr/bin/python

from bfs import SimpleBFS

class CSINodeColoring:
    """Find a connected sequential (CS) node coloring."""

    def __init__(self, graph):
        """Inicjalizuje algorytm."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, None) for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        ordering = list()
        algorithm = SimpleBFS(self.graph)
        algorithm.run(pre_action=lambda node: ordering.append(node))
        for source in ordering:
            self._greedy_color_with_interchange(source)


    def _greedy_color_with_interchange(self, source):
        """Przydziela pierwszy dostepny kolor, sprawdzajac
        czy nie mozna zwolnic ktoregos z kolorow sasiadow."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        i_max = 0
        i_free = None
        for i in xrange(n):
            if used[i]:
                i_max = i
            elif i_free is None:
                i_free = i
        if i_free > i_max:
            self._recolor(source, i_max)
```

```
        self._greedy_color(source)

    def _recolor(self, source, i_max):
        """Proba zamiany kolorow."""
        n = self.graph.v()
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is None:
                continue
            used = [False] * n
            used[self.color[edge.target]] = True
            for edge2 in self.graph.iteroutedges(edge.target):
                if self.color[edge2.target] is not None:
                    used[self.color[edge2.target]] = True
            for i in xrange(n):
                if not used[i]:
                    new_color = i
                    break
            if new_color <= i_max:
                old_color = self.color[edge.target]
                self.color[edge.target] = new_color
                return old_color
        return i_max + 1

    def _greedy_color(self, source):
        """Przydziela i zwraca pierwszy dostepny kolor."""
        n = self.graph.v()
        used = [False] * n
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for i in xrange(n):
            if not used[i]:
                self.color[source] = i
                break
        return i
```

7.4. Algorytmy zbiorów niezależnych

Kolorowanie wierzchołków grafu prowadzi do podziału grafu na zbiory niezależne. Każdemu kolorowi odpowiada osobny zbiór niezależny. Stąd pomysł algorytmów zbiorów niezależnych polega na sukcesywnym wyznaczaniu maksymalnych zbiorów niezależnych w grafie i przydzielaniu wierzchołkom poszczególnych zbiorów niezależnych odrębnych kolorów [4].

Algorytmy kolorowania wyznaczają zbiory niezależne na podstawie pewnych heurystycznych reguł.

— *GIS* (ang. *Greedy Independent Set*)
— *RLF* (ang. *Recursive Largest First*)

7.5. Kolorowanie krawędzi

Kolorowanie krawędzi multigrafu (ang. *edge coloring*) polega na przyporządkowaniu krawędziom kolorów w taki sposób, aby dla każdego wierzchołka

stykające się w nim krawędzie miały różne kolory (takie jest kolorowanie krawędziowe legalne/dozwolone) [19]. Stąd w multigrafie nie mogą występować pętle. Multigraf jest *k-kolorowalny krawędziowo*, jeżeli istnieje legalne kolorowanie krawędzi wykorzystujące k kolorów. Optymalne kolorowanie krawędzi multigrafu to legalne kolorowanie, przy którym wykorzystano najmniejszą możliwą liczbę kolorów, nazywaną *indeksem chromatycznym* χ' lub χ_1.

Problem kolorowania krawędzi jest w ogólności NP-trudny, ale np. dla grafów dwudzielnych istnieją algorytmy wielomianowe znajdujące optymalne kolorowanie.

Twierdzenie (Vizing, 1964): Jeżeli G jest grafem prostym, to

$$\Delta(G) \leqslant \chi'(G) \leqslant \Delta(G) + 1. \tag{7.3}$$

Twierdzenie (König, 1916 lub 1931): Jeżeli G jest multigrafem dwudzielnym, to

$$\chi'(G) = \Delta(G). \tag{7.4}$$

Twierdzenie (Shannon, 1949): Dla każdego multigrafu G zachodzi

$$\chi'(G) \leqslant (3/2)\Delta(G), \tag{7.5}$$
$$\chi'(G) \leqslant \Delta(G) + \mu(G), \tag{7.6}$$

gdzie $\mu(G)$ to wielokrotność (ang. *multiplicity*), czyli największa liczba krawędzi równoległych w jednej wiązce. Dla grafu prostego $\mu(G) = 1$.

7.5.1. Przykłady kolorowania krawędzi

Warto podać kilka przykładów grafów prostych i ich kolorowania krawędziowego.

— Dla grafu pełnego K_n, gdzie $\Delta = n - 1$, mamy $\chi = n - 1$ (n parzyste) lub $\chi' = n$ (n nieparzyste)..
— Dla grafu dwudzielnego $\chi' = \Delta$ (twierdzenie (Königa).
— Dla grafu cyklicznego C_n, gdzie $\Delta = 2$, mamy $\chi' = 2$ (n parzyste) lub $\chi' = 3$ (n nieparzyste).

Widać, że indeks chromatyczny grafów prostych może wynosić od Δ do $\Delta+1$.

7.5.2. Zastosowanie kolorowania krawędzi

Zastosowanie kolorowania krawędzi polega na opisie sytuacji kolejkowania zdarzeń wymagających uczestnictwa dwóch obiektów, przy czym dany obiekt nie może uczestniczyć w więcej niż jednym zdarzeniu jednocześnie.

Równoległe procesy montażu: Wierzchołek reprezentuje element składowy. Krawędź odpowiada sytuacji, w której jej dwa końce mają być połączone podczas pewnej tury montażu. Ograniczenie polega na tym, że jeden element składowy nie może uczestniczyć jednocześnie w dwóch operacjach łączenia, czyli w dwóch turach montażu. Celem jest minimalizacja liczby kolejnych tur całego montażu, czyli jak największa liczba faz montażu powinna odbyć

się równolegle. Krawędzie jednego koloru odpowiadają fazom montażu, które moga się odbywać równolegle.

Konstrukcja harmonogramu: Kolory reprezentują kolejne przedziały czasowe, w których może odbyć się operacja odpowiadająca krawędzi, np. spotkanie dwóch osób. Wierzchołki reprezentują osoby biorące udział w spotkaniach. Jedna osoba nie może w tym samym czasie uczestniczyć w kilku różnych spotkaniach. Liczba wykorzystanych kolorów krawędzi mówi o liczbie przedziałów czasu niezbędnych do przeprowadzenia spotkań.

7.5.3. Graf krawędziowy

Graf krawędziowy (ang. *line graph*) grafu nieskierowanego $G = (V_G, E_G)$ to graf $F = L(G) = (V_F, E_F)$, w którym $V_F = E_G$, natomiast E_F jest zbiorem par elementów z E_G, przy czym wierzchołki (s, t) i (v, u) grafu F są połączone wtedy i tylko wtedy, gdy $s = v$ lub $s = u$ lub $t = v$ lub $t = u$ [14].

Wiele własności związanych z krawędziami w grafie G przenosi się na własności zwiazane z wierzchołkami w grafie $L(G)$. Przykładowo skojarzeniu w grafie G odpowiada zbiór niezależny w grafie $L(G)$. A kolorowaniu krawędzi w grafie G odpowiada kolorowanie wierzchołków w grafie $L(G)$. Zostanie to wykorzystane w odpowiednim algorytmie kolorowania.

7.5.4. Algorytm kolorowania z grafem krawędziowym

Dane wejściowe: Multigraf nieskierowany G bez pętli.

Problem: Kolorowanie krawędzi multigrafu G.

Opis algorytmu: Algorytm polega na stworzeniu grafu krawędziowego $L(G)$, a następnie uruchomieniu algorytmu kolorowania wierzchołków dla grafu $L(G)$.

Złożoność: Złożoność czasowa algorytmu zależy od czasu budowy grafu krawędziowego $L(G)$ i czasu kolorowania wierzchołków grafu $L(G)$. Czas budowy grafu krawędziowego można oszacować przez $O(V\Delta^2)$, gdzie Δ jest największym stopniem wierzchołka w G. Inne oszacowanie to $O(E\Delta)$, które wynika z obserwacji, że dwie pierwsze pętle **for** przebiegają listy sąsiedztwa w grafie G.

Uwagi: W algorytmie wykorzystujemy własność naszej implementacji, w której krawędzie są hashowalne i uporządkowane. W innych implementacjach krawędziom grafu G przyporządkowuje się kolejne liczby całkowite od zera, które już mogą być np. indeksami w macierzy sąsiedztwa.

Listing 7.11. Algorytm kolorowania z grafem krawędziowym

```
#!/usr/bin/python
#
# Budujemy graf krawedziowy na bazie pomyslu z
```

```
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0126c.php

class EdgeColoringWithLineGraph:
    """Algorytm kolorowania krawedzi z grafem krawedziowym."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu"""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = None
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")

    def run(self):
        """Obliczenia."""
        line_graph = self.graph.__class__(
            self.graph.e(), directed=False)
        for node in self.graph.iternodes():
            for edge1 in self.graph.iteroutedges(node):
                for edge2 in self.graph.iteroutedges(edge1.target):
                    if edge1.source > edge1.target:
                        edge1 = ~edge1
                    if edge2.source > edge2.target:
                        edge2 = ~edge2
                    if edge1 < edge2:
                        line_graph.add_edge(Edge(edge1, edge2))
        algorithm = NodeColoring(line_graph)
        algorithm.run()
        self.color = algorithm.color
```

7.6. Kolorowanie ścian

Kolorowanie ścian można sprowadzić do problemu kolorowania wierzchołków grafu dualnego.

8. Podsumowanie

W pracy przedstawiono dwie implementacje klasy MultiGraph, które można wykorzystać przy pracy z multigrafami. Zaimplementowanp szereg algorytmów operujących na multigrafach. Są to algorytmy do wyznaczania cyklu Eulera, znajdowanie zbiorów niezależnych, oraz kolorowanie wierzchołków i krawędzi w multigrafach.

Zastosowanie języka Python znajduje uzasadnienie głównie w czytelności kodu i łatwości wprowadzania modyfikacji. Kosztem wydajności w stosunku do innych popularnych języków programowania, uzyskano możliwość koncentracji na zagadnieniach związanych z rozwiązywanym problemem, pomijając kwestie deklaracji zmiennych czy zarządzania pamięcią.

Zgodnie z początkowymi zamierzeniami w pracy mało zostać zaprezentowane również zastosowanie algorytmów zbiorów niezależnych do kolorowania wierzchołków. Algorytmy zbiorów niezależnych pozwalają zwykle uzyskać pokolorowanie mniejszą liczbą kolorów [3]. Ze względu na brak czasu, nie udało się tego zamierzenia zrealizować.

Inne pomysły na dalsze badania to algorytmy związane z pokryciem wierzchołkowym, pokryciem krawędziowym i skojarzeniami.

A. Kod źródłowy dla krawędzi i multigrafów

W tym dodatku przedstawimy kody źródłowe klas reprezentujących krawędzie i multigrafy.

A.1. Klasa Edge

Klasa Edge pochodzi z implementacji grafów rozwijanej w Instytucie Fizyki Uniwersytetu Jagiellońskiego w Krakowie [1].

Listing A.1. Klasa Edge z modułu edges.

```
#!/usr/bin/python

class Edge:
    """Definicja klasy dla krawedzi skierowanych."""

    def __init__(self, source, target, weight=1):
        """Load up an Edge instance."""
        self.source = source
        self.target = target
        self.weight = weight

    def __repr__(self):
        """Zwraca tekstowa reprezentacje krawedzi."""
        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:
            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 krawedzi."""
        return hash(repr(self))

    def __invert__(self):
        """Zwraca krawedz o odwrotnym kierunku."""
        return Edge(self.target, self.source, self.weight)

    inverted = __invert__
```

A.2. Klasa dla multigrafów bez wag

Klasa MiltiGraph dla multigrafów bez wag bazuje na idei macierzy sąsiedztwa, którą opisuje się grafy proste z wagami. W miejscu wag przechowywane są teraz liczby całkowite nieujemne, opisujące liczby krawędzi równoległych między wierzchołkami. Ponadto zamiast pełnej macierzy kwadratowej wykorzystana jest struktura zagnieżdżonych słowników, która zapewnia szybkie wyszukiwanie krawedzi, a przy tym oszczędza pamięć, bo zera nie są przechowywane.

Listing A.2. Klasa MultiGraph dla multigrafów bez wag z modułu multigraphs1.

```
#!/usr/bin/python
#
# Ta implementacja multigrafow korzysta z krawedzi z klasy
# Edge, ale nie korzysta z wag krawedzi (wagi sa ignorowane).
# Nie sa obslugiwane multigrafy z wagami.

from edges import Edge

class MultiGraph(dict):
    """Definicja klasy dla multigrafow bez wag."""

    def __init__(self, n=0, directed=False):
        """Inicjalizacja atrybutow klasy."""
        self.n = n
        self.directed = directed

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

    def e(self):
        """Zwraca liczbe krawedzi."""
        loops = sum(self[node].get(node, 0) for node in self)
        edges = 0
        for source in self:
            for target in self[source]:
                edges = edges + self[source][target]
        if self.is_directed():
            return edges
        else:
            return (edges + loops) / 2
```

```
    def is_directed(self):
        """Sprawdza czy multigraf jest skierowany."""
        return self.directed

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

    def has_node(self, node):
        """Sprawdza czy wierzcholek istnieje."""
        return node in self

    def del_node(self, node):
        """Usuwa wierzcholek z multigrafu."""
        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 krawedz do multigrafu."""
        self.add_node(edge.source)
        self.add_node(edge.target)
        self[edge.source][edge.target] = (
            self[edge.source].get(edge.target, 0) + 1)
        if not self.is_directed() and edge.source != edge.target:
            self[edge.target][edge.source] = (
                self[edge.target].get(edge.source, 0) + 1)

    def del_edge(self, edge):
        """Usuwa krawedz z multigrafu."""
        if self[edge.source][edge.target] > 1:
            self[edge.source][edge.target] -= 1
            if not self.is_directed() and edge.source != edge.target:
                self[edge.target][edge.source] -= 1
        else:
            del self[edge.source][edge.target]
            if not self.is_directed() and edge.source != edge.target:
                del self[edge.target][edge.source]

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

    def weight(self, edge):
        """Zwraca liczbe krawedzi rownoleglych."""
        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 generator wierzcholkow multigrafu."""
```

```
        return self.iterkeys()

    def iteradjacent(self, source):
        """Zwraca generator wierzcholkow sasiadujacych z source."""
        return self[source].iterkeys()

    def iteroutedges(self, source):
        """Zwraca generator krawedzie wychodzacych z source."""
        for target in self[source]:
            for step in xrange(self[source][target]):
                yield Edge(source, target)

    def iterinedges(self, source):
        """Zwraca generator krawedzi wchodzacych do source."""
        if self.is_directed():
            for target in self.iternodes():
                if source in self[target]:
                    for step in xrange(self[target][source]):
                        yield Edge(target, source)
        else:
            for target in self[source]:
                for step in xrange(self[source][target]):
                    yield Edge(target, source)

    def iteredges(self):
        """Zwraca generator krawedzi multigrafu."""
        for source in self.iternodes():
            for target in self[source]:
                if self.is_directed() or source <= target:
                    for step in xrange(self[source][target]):
                        yield Edge(source, target)

    def show(self):
        """Drukuje tekstowa reprezentacje multigrafu."""
        for source in self.iternodes():
            print source, ":",
            for target in self[source]:
                print "%s(%s)" % (target, self[source][target]),
            print

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

    def transpose(self):
        """Zwraca multigraf transponowany."""
        new_graph = MultiGraph(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 degree(self, source):
```

```
        """Zwraca stopien wierzcholka source."""
        if self.is_directed():
            raise ValueError("the graph is directed")
        loops = self[source].get(source, 0)
        edges = 0
        for target in self[source]:
            edges = edges + self[source][target]
        return edges + loops

    def outdegree(self, source):
        """Zwraca liczbe krawedzi wychodzacych z source."""
        loops = self[source].get(source, 0)
        edges = 0
        for target in self[source]:
            edges = edges + self[source][target]
        if self.is_directed():
            return edges
        else:
            return edges + loops

    def indegree(self, source):
        """Zwraca liczbe krawedzi wchodzacych do source."""
        if self.is_directed():
            edges = 0
            for target in self.iternodes():
                if source in self[target]:
                    edges = edges + self[target][source]
            return edges
        else:
            loops = self[source].get(source, 0)
            edges = 0
            for target in self[source]:
                edges = edges + self[source][target]
            return edges + loops

    def __eq__(self, other):
        """Sprawdza czy multigrafy sa takie same."""
        if self.is_directed() is not other.is_directed():
            return False
        if self.v() != other.v():
            return False
        for node in self.iternodes():
            if not other.has_node(node):
                return False
        if self.e() != other.e():
            return False
        for edge in self.iteredges():
            if not other.has_edge(edge):
                return False
        return True

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

    def add_multigraph(self, other):
        """Dodaje multigraf do multigrafu."""
```

```
        for node in other.iternodes():
            self.add_node(node)
        for edge in other.iteredges():
            self.add_edge(edge)
```

A.3. Klasa dla multigrafów z wagami

Klasa MiltiGraph dla multigrafów z wagami wykorzystuje strukturę zagnieżdżonych słowników, a do tego mamy listy przechowujące całe obiekty krawędzi równoległych. W ten sposób bez zmiany klasy można przechowywać krawędzie z innymi atrybutami.

Listing A.3. Klasa MultiGraph dla multigrafów z wagami z modułu multigraphs2.

```
#!/usr/bin/python

class MultiGraph(dict):
    """Definicja klasy dla multigrafow z wagami."""

    def __init__(self, n=0, directed=False):
        """Inicjalizacja atrybutow klasy."""
        self.n = n
        self.directed = directed

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

    def e(self):
        """Zwraca liczbe krawedzi."""
        loops = 0
        for node in self:
            if node in self[node]:
                loops = loops + len(self[node][node])
        edges = 0
        for source in self:
            for target in self[source]:
                edges = edges + len(self[source][target])
        if self.is_directed():
            return edges
        else:
            return (edges + loops) / 2

    def is_directed(self):
        """Sprawdza czy multigraf jest skierowany."""
        return self.directed

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

    def has_node(self, node):
        """Sprawdza czy wierzcholek istnieje."""
```

```
        return node in self

    def del_node(self, node):
        """Usuwa wierzcholek z multigrafu."""
        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 krawedz do multigrafu."""
        self.add_node(edge.source)
        self.add_node(edge.target)
        if edge.target not in self[edge.source]:
            self[edge.source][edge.target] = list()
        if not self.is_directed() and edge.source not in self[edge.target]:
            self[edge.target][edge.source] = list()
        self[edge.source][edge.target].append(edge)
        if not self.is_directed() and edge.source != edge.target:
            self[edge.target][edge.source].append(~edge)

    def del_edge(self, edge):
        """Usuwa krawedz z multigrafu."""
        self[edge.source][edge.target].remove(edge)
        if len(self[edge.source][edge.target]) == 0:
            del self[edge.source][edge.target]
        if not self.is_directed() and edge.source != edge.target:
            self[edge.target][edge.source].remove(~edge)
            if len(self[edge.target][edge.source]) == 0:
                del self[edge.target][edge.source]

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

    def weight(self, edge):
        """Zwraca liczbe krawedzi rownoleglych."""
        if edge.source in self and edge.target in self[edge.source]:
            return len(self[edge.source][edge.target])
        else:
            return 0

    def iternodes(self):
        """Zwraca generator wierzcholkow multigrafu."""
        return self.iterkeys()

    def iteradjacent(self, source):
        """Zwraca wierzcholki sasiadujace z source."""
        return self[source].iterkeys()

    def iteroutedges(self, source):
        """Zwraca generator krawedzi wychodzacych z source."""
        for target in self[source]:
            for edge in self[source][target]:
                yield edge
```

```
    def iterinedges(self, source):
        """Zwraca generator krawedzi wchodzacych do source."""
        if self.is_directed():
            for target in self.iternodes():
                if source in self[target]:
                    for edge in self[target][source]:
                        yield edge
        else:
            for target in self[source]:
                for edge in self[source][target]:
                    yield ~edge

    def iteredges(self):
        """Zwraca generator krawedzie multigrafu."""
        for source in self.iternodes():
            for target in self[source]:
                # source <= target, because loops are possible.
                if self.is_directed() or source <= target:
                    for edge in self[source][target]:
                        yield edge

    def show(self):
        """Drukuje tekstowa reprezentacje multigrafu."""
        for source in self.iternodes():
            print source, ":",
            for target in self[source]:
                print "%s(%s)" % (target, len(self[source][target])),
            print

    def copy(self):
        """Zwraca kopie multigrafu."""
        new_graph = MultiGraph(n=self.n, directed=self.directed)
        for source in self.iternodes():
            new_graph[source] = dict()
            for target in self[source]:
                new_graph[source][target] = list(self[source][target])
        return new_graph

    def transpose(self):
        """Zwraca multigraf transponowany."""
        new_graph = MultiGraph(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 degree(self, source):
        """Zwraca stopien wierzcholka source."""
        if self.is_directed():
            raise ValueError("the graph is directed")
        if source in self[source]:
            loops = len(self[source][source])
        else:
            loops = 0
        edges = 0
```

```
        for target in self[source]:
            edges = edges + len(self[source][target])
        return edges + loops

    def outdegree(self, source):
        """Zwraca liczbe krawedzi wychodzacych z source."""
        if source in self[source]:
            loops = len(self[source][source])
        else:
            loops = 0
        edges = 0
        for target in self[source]:
            edges = edges + len(self[source][target])
        if self.is_directed():
            return edges
        else:
            return edges + loops

    def indegree(self, source):
        """Zwraca liczbe krawedzi wchodzacych do source."""
        if self.is_directed():
            edges = 0
            for target in self.iternodes():
                if source in self[target]:
                    edges = edges + len(self[target][source])
            return edges
        else:   #
            if source in self[source]:
                loops = len(self[source][source])
            else:
                loops = 0
            edges = 0
            for target in self[source]:
                edges = edges + len(self[source][target])
            return edges + loops

    def __eq__(self, other):
        """Sprawdza czy multigrafy sa takie same."""
        if self.is_directed() is not other.is_directed():
            return False
        if self.v() != other.v():
            return False
        for node in self.iternodes():
            if not other.has_node(node):
                return False
        if self.e() != other.e():
            return False
        for edge in self.iteredges():
            if not other.has_edge(edge):
                return False
        return True

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

    def add_multigraph(self, other):
```

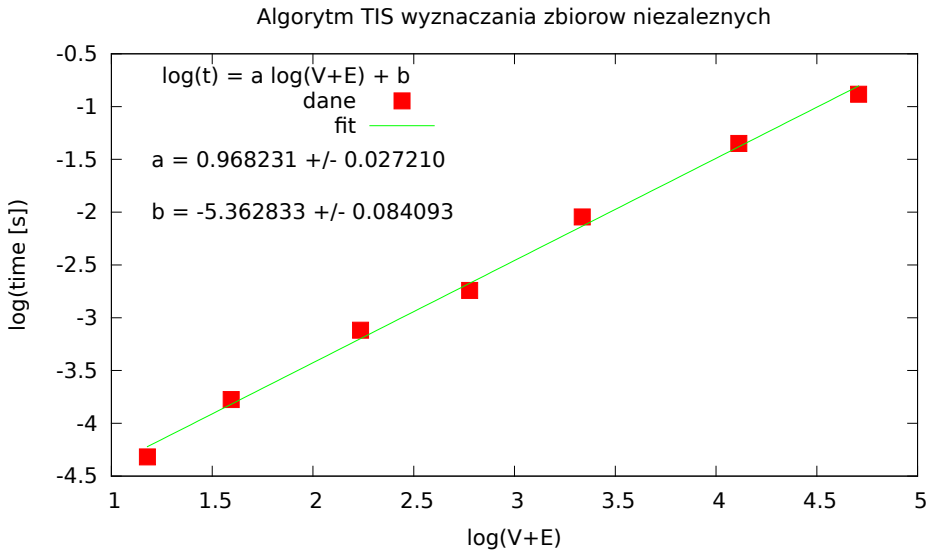
```
        """Dodaje multigraf do multigrafu."""
        for node in other.iternodes():
            self.add_node(node)
        for edge in other.iteredges():
            self.add_edge(edge)
```

B. Testy algorytmów zbiorów niezależnych

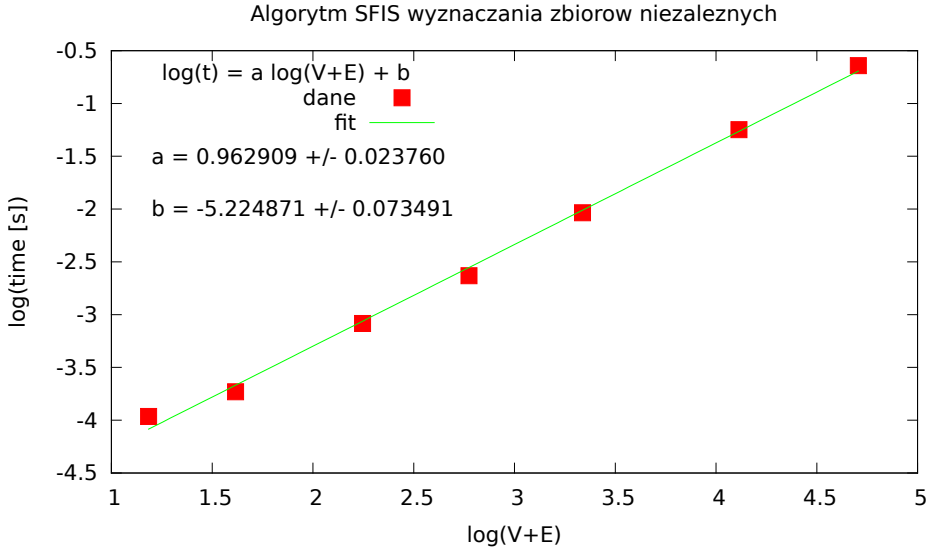
W ramach pracy przeprowadzono testy wszystkich algorytmów zbiorów niezależnych. Wyniki dla grafów przypadkowych przedstawiono w tabeli B.1, oraz na rysunkach od B.1 do B.5. Wyniki dla drzew pokazano na rysunkach od B.6 do B.10. Dla algorytmów TIS i SFIS eksperyment potwierdza teorię. Dla innych algorytmów wydaje się, że oszacowania teoretyczne są zawyżone, ale nie jest jasne, jak je poprawić.

Tabela B.1. Średnia liczba wierzchołków w zbiorze niezależnym dla grafu losowego o prawdopodobieństwie $p = 0.1$ istnienia krawędzi między wierzchołkami w zależności od liczby wierzchołków n.

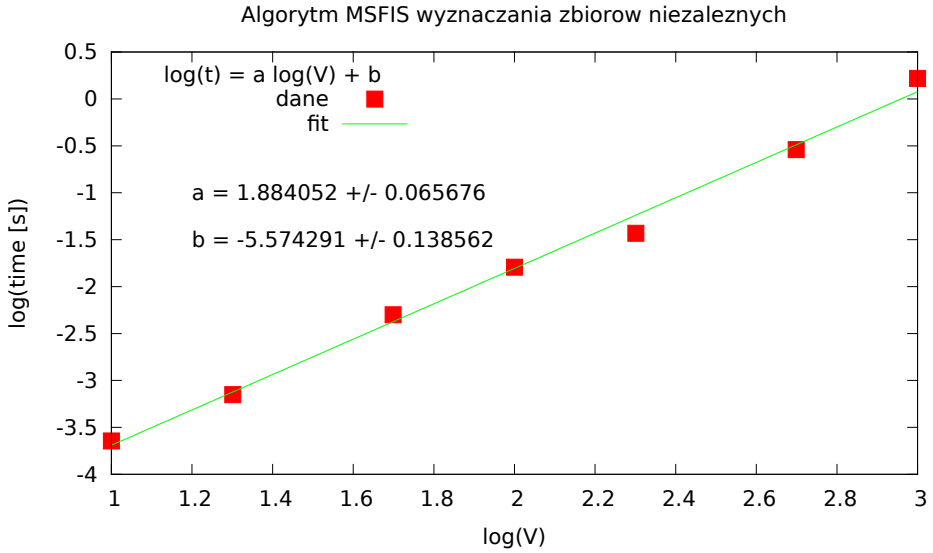
n	**TIS**	**SFIS**	**MSFIS**	**LLIS**	**MLLIS**
10	7.0	7.2	7.5	7.4	7.3
20	11.3	11.7	11.8	11.2	11.6
50	17.3	20.7	19.8	16.4	19.3
100	23.4	27.1	26.1	19.4	27.7
200	29.1	33.0	33.1	18.4	34.3
500	38.9	43.7	42.3	17.5	43.7
1000	43.3	48.9	48.2	15.7	50.1



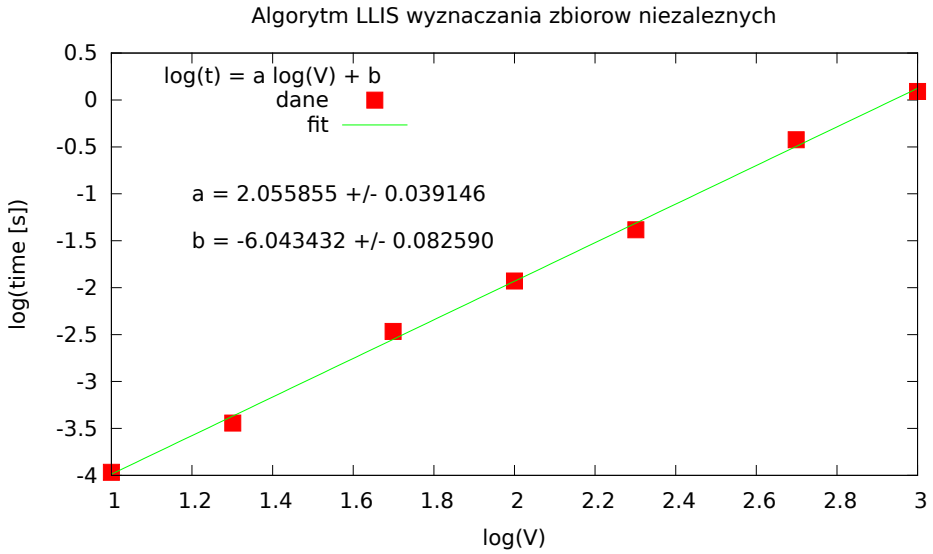
Rysunek B.1. Wykres wydajności algorytmu TIS wyznaczania zbiorów niezależnych dla grafów losowych. Współczynnik a bliski 1 potwierdza zależność liniową.



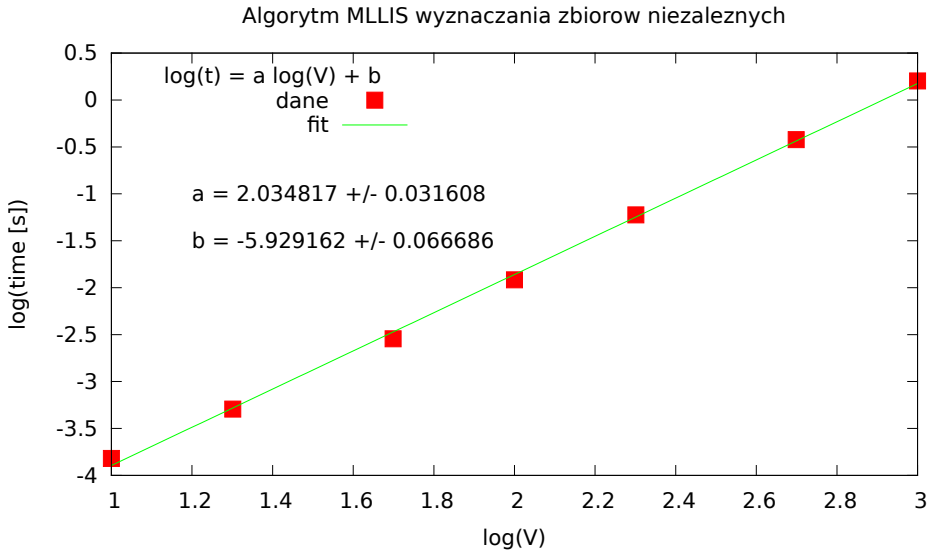
Rysunek B.2. Wykres wydajności algorytmu SFIS wyznaczania zbiorów niezależnych dla grafów losowych. Współczynnik a bliski 1 sugeruje zależność liniową.



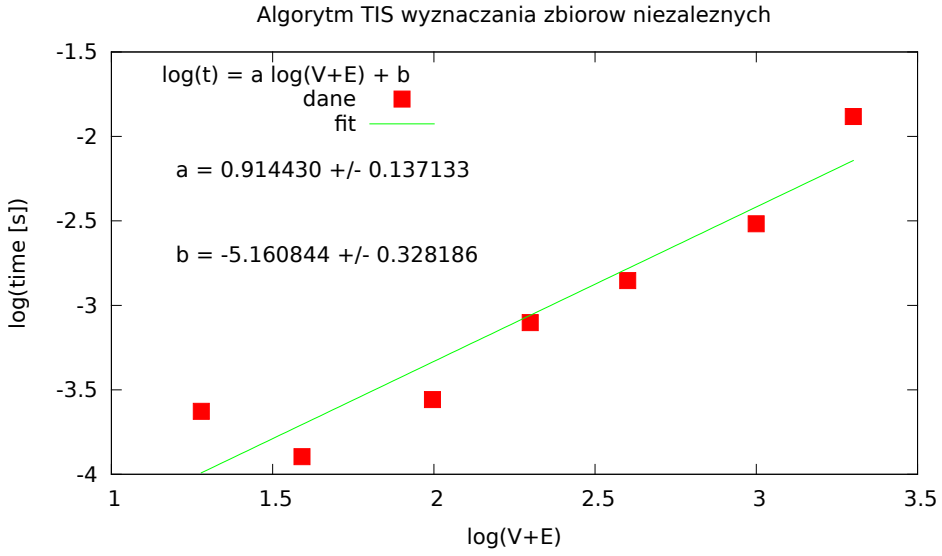
Rysunek B.3. Wykres wydajności algorytmu MSFIS wyznaczania zbiorów niezależnych dla grafów losowych. Współczynnik a bliski 2 sugeruje dominującą zależność $O(V^2)$.



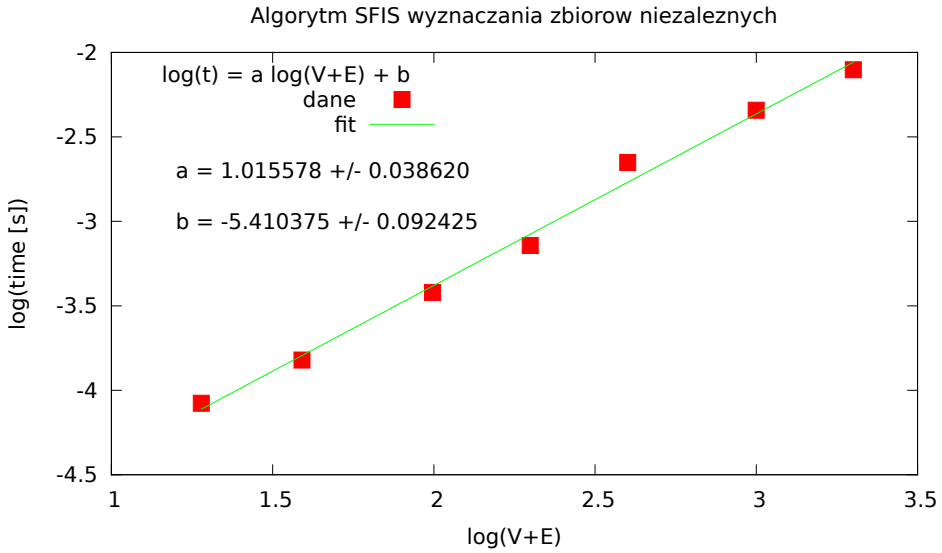
Rysunek B.4. Wykres wydajności algorytmu LLIS wyznaczania zbiorów niezależnych dla grafów losowych. Współczynnik a przekracza 2.



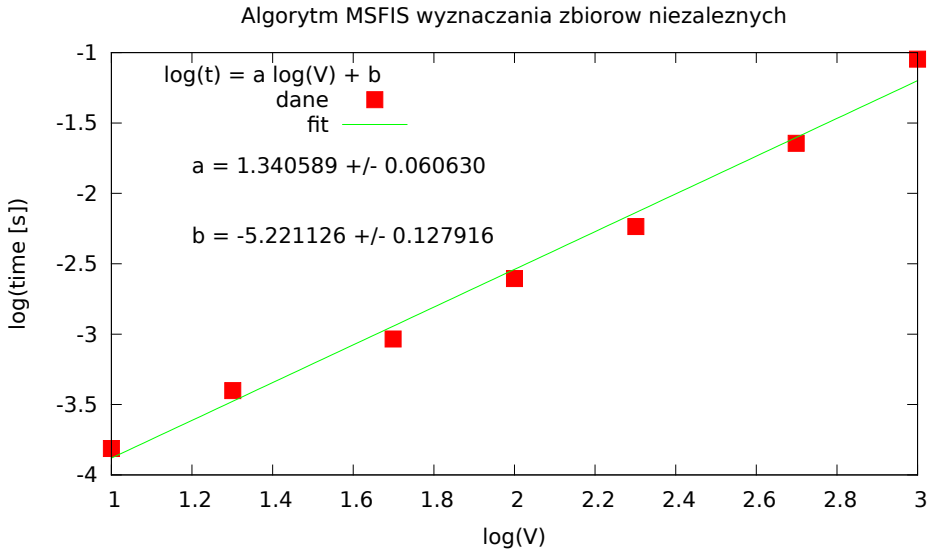
Rysunek B.5. Wykres wydajności algorytmu MLLIS wyznaczania zbiorów niezależnych dla grafów losowych. Interpretacja nie jest jasna.



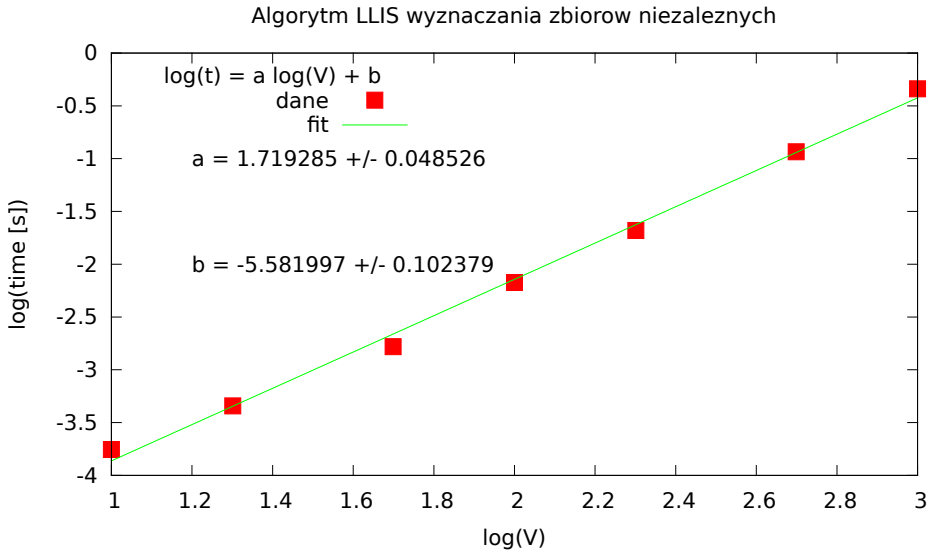
Rysunek B.6. Wykres wydajności algorytmu TIS wyznaczania zbiorów niezależnych dla drzew. Współczynnik a bliski 1 potwierdza zależność liniową.



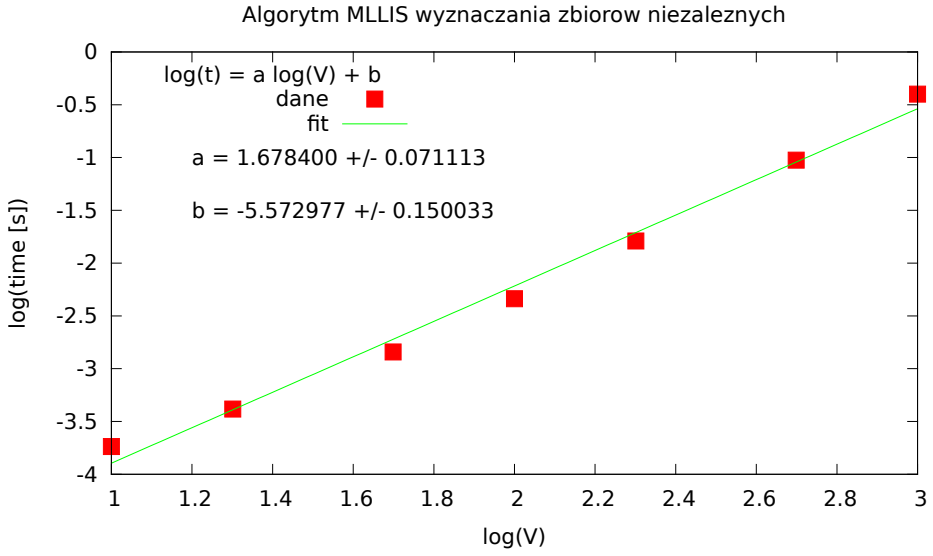
Rysunek B.7. Wykres wydajności algorytmu SFIS wyznaczania zbiorów niezależnych dla drzew. Współczynnik a lekko powyżej 1 sugeruje zależność bliską liniowej.



Rysunek B.8. Wykres wydajności algorytmu MSFIS wyznaczania zbiorów niezależnych dla drzew. Interpretacja nie jest jasna.



Rysunek B.9. Wykres wydajności algorytmu LLIS wyznaczania zbiorów niezależnych dla drzew. Interpretacja nie jest jasna.



Rysunek B.10. Wykres wydajności algorytmu MLLIS wyznaczania zbiorów niezależnych dla drzew. Interpretacja nie jest jasna.

C. Testy algorytmów kolorowania wierzchołków

W ramach pracy wykonano testy wszystkich algorytmów kolorowania wierzchołków grafu. Zbadano kolorowanie grafów przypadkowych o różnej liczbie wierzchołków i różnych prawdopodobieństwach wystąpienia krawędzi między wierzchołkami. Średnie liczby kolorów przydzielanych przez algorytmy pokazano w tabelach C.1, C.2, C.3 i C.4. Na rysunkach przedstawiono wyniki eksperymentów z grafami przypadkowymi z $p = 0.5$.

Tabela C.1. Średnia liczba kolorów potrzebnych do pokolorowania grafu losowego z liczbą wierzchołków $n = 1000$ algorytmami bez wymiany kolorów w zależności od prawdopodobieństwa p istnienia krawędzi między wierzchołkami. Algorytm DSATUR generuje najmniej kolorów.

p	**RS**	**US**	**CS**	**LF**	**SL**	**DSATUR**
0.1	31.2	31.3	31.5	29.3	30.2	26.5
0.2	53.6	53.7	53.7	51.2	52.7	47.0
0.3	76.1	76.3	76.7	72.9	74.3	67.6
0.4	100.5	100.4	100.4	96.8	98.4	90.3
0.5	126.8	126.1	126.7	122.9	124.3	115.5
0.6	156.7	157.5	157.3	152.3	154.3	133.6
0.7	194.1	194.3	196.0	189.8	190.8	179.4
0.8	245.3	243.0	243.7	237.0	238.3	226.9
0.9	321.0	320.8	322.8	313.3	317.2	303.8

Tabela C.2. Średnia liczba kolorów potrzebnych do pokolorowania grafu losowego z liczbą wierzchołków $n = 1000$ algorytmami z wymianą kolorów w zależności od prawdopodobieństwa p istnienia krawędzi między wierzchołkami. Algorytmy osiągają porównywalne wyniki.

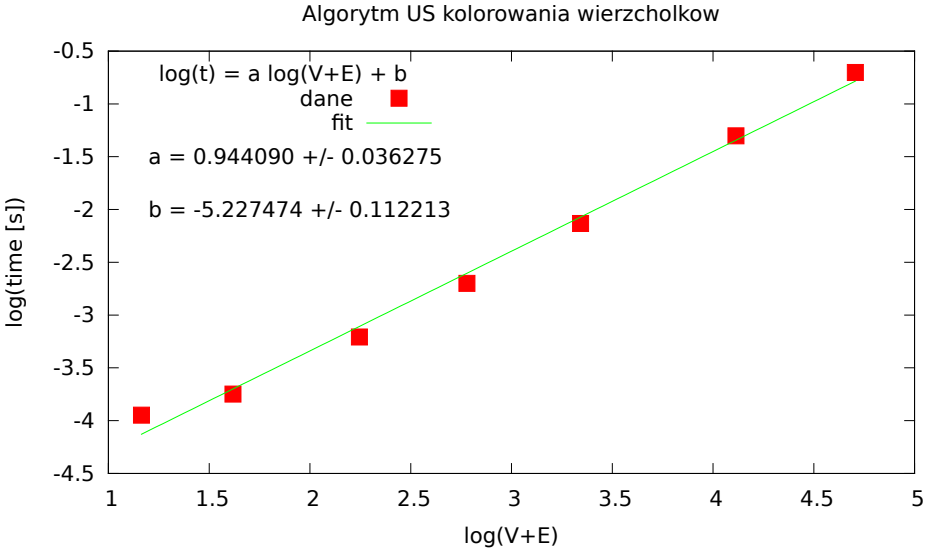
p	**RSI**	**USI**	**CSI**
0.1	31.7	31.1	31.7
0.2	53.5	53.9	53.8
0.3	75.6	76.5	76.9
0.4	100.2	100.9	99.6
0.5	128.1	126.5	127.0
0.6	157.5	158.0	157.9
0.7	193.9	195.2	193.6
0.8	244.7	243.1	243.1
0.9	321.4	322.3	321.7

Tabela C.3. Średnia liczba kolorów potrzebnych do pokolorowania algorytmami bez wymiany kolorów grafu losowego z prawdopodobieństwem istnienia krawędzi między każdą parą wierzchołków $p = 0.5$ w zależności od liczby wierzchołków n.

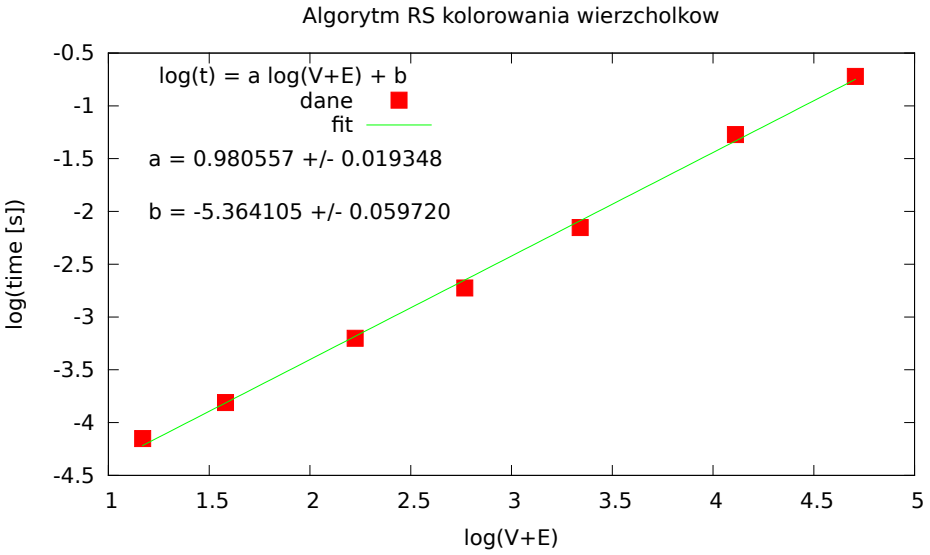
n	**RS**	**US**	**CS**	**LF**	**SL**	**DSATUR**
10	4.5	4.3	4.0	4.0	3.9	3.8
20	7.2	7.0	6.6	6.3	6.5	6.0
50	13.5	12.9	13.3	11.9	11.7	11.3
100	21.5	21.1	20.9	19.9	20.5	18.2
200	35.6	35.5	35.4	33.6	34.4	30.8
500	71.7	73.1	72.3	69.4	73.1	65.4
1000	126.5	126.1	126.9	122.8	123.5	114.7

Tabela C.4. Średnia liczba kolorów potrzebnych do pokolorowania algorytmami z wymianą kolorów grafu losowego z prawdopodobieństwem istnienia krawędzi między każdą parą wierzchołków $p = 0.5$ w zależności od liczby wierzchołków n.

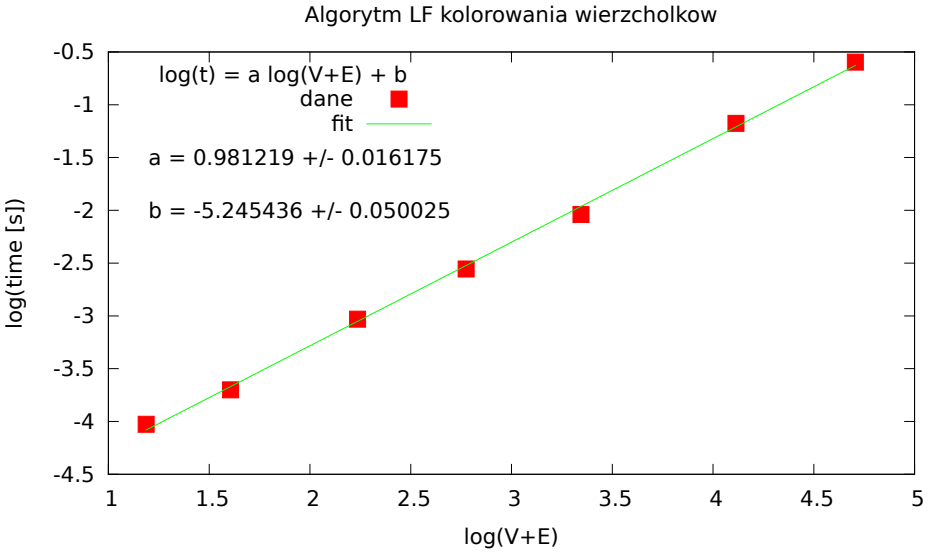
n	**RSI**	**USI**	**CSI**
10	4.1	4.3	4.3
20	6.9	6.6	6.6
50	13.1	12.6	12.3
100	21.3	21.5	21.2
200	35.4	35.9	35.3
500	72.3	72.7	72.2
1000	126.9	126.7	127.3



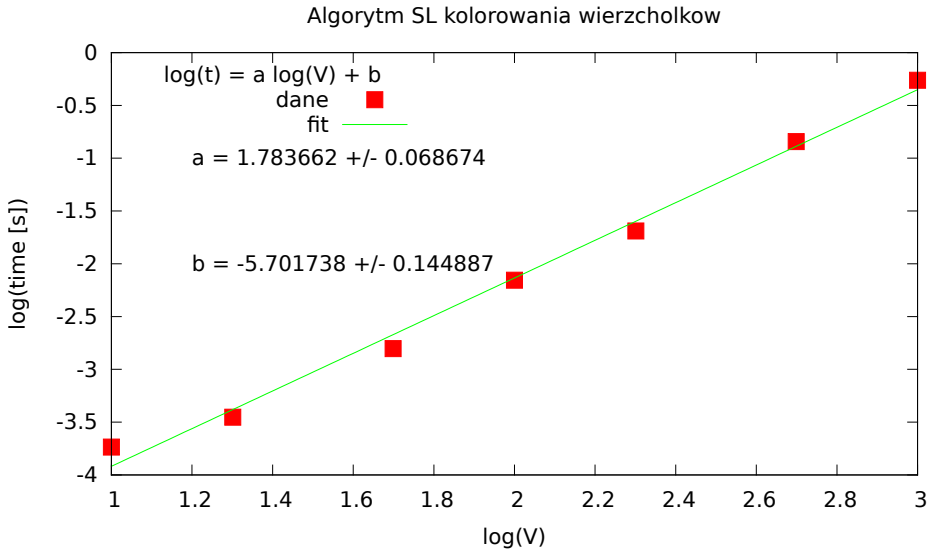
Rysunek C.1. Wykres wydajności algorytmu US kolorowania wierzchołków. Współczynnik a bliski 1 sugeruje zależność liniową.



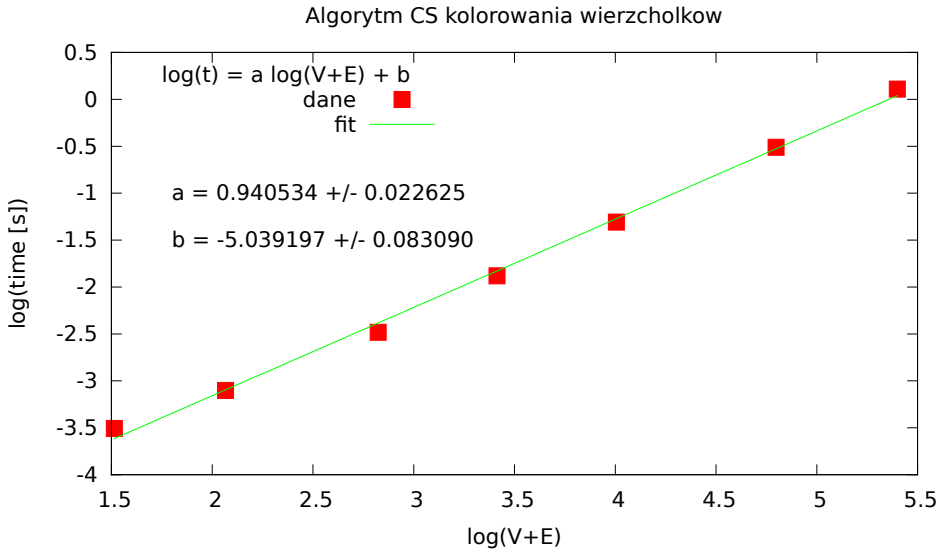
Rysunek C.2. Wykres wydajności algorytmu RS kolorowania wierzchołków. Współczynnik a bliski 1 sugeruje zależność liniową.



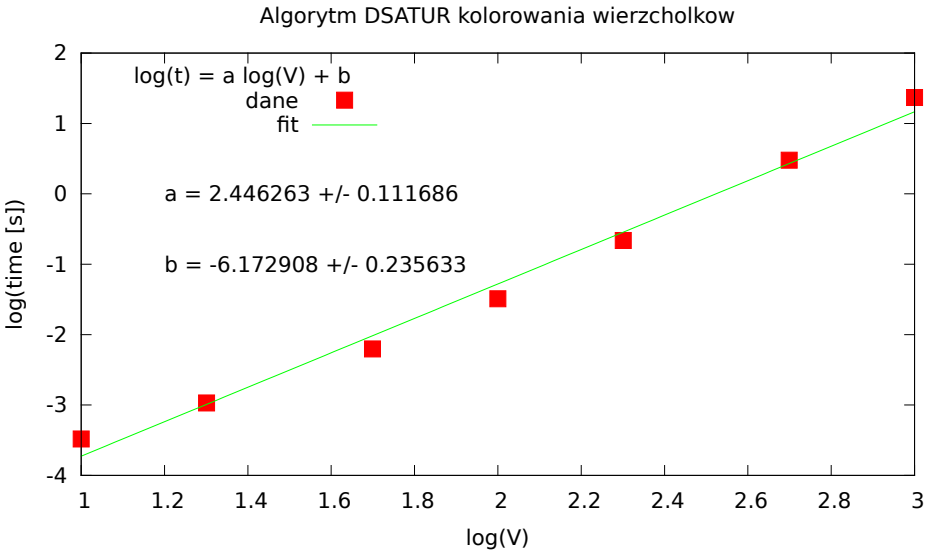
Rysunek C.3. Wykres wydajności algorytmu LF kolorowania wierzchołków. Współczynnik a bliski 1 sugeruje zależność liniową.



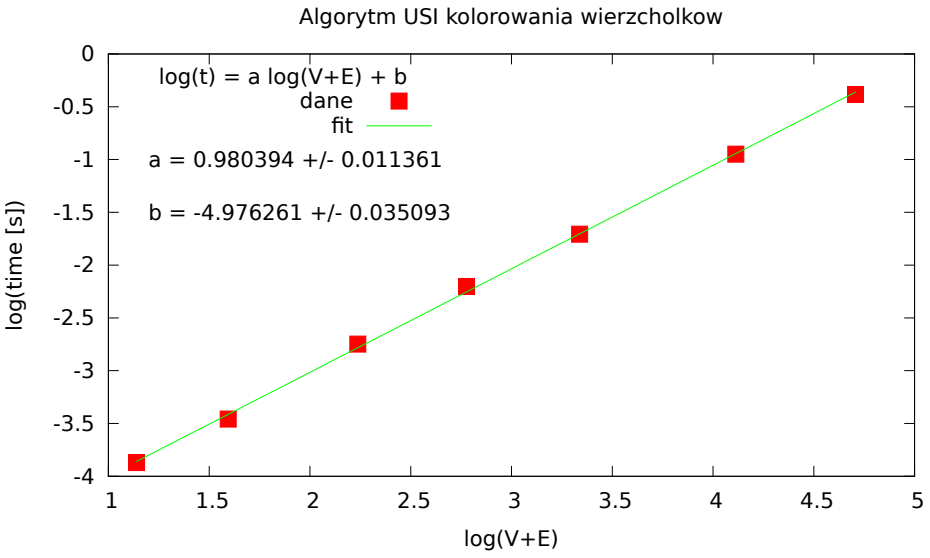
Rysunek C.4. Wykres wydajności algorytmu SL kolorowania wierzchołków. Współczynnik a sugeruje zależność poniżej $O(V^2)$, np. $O(V + E)$.



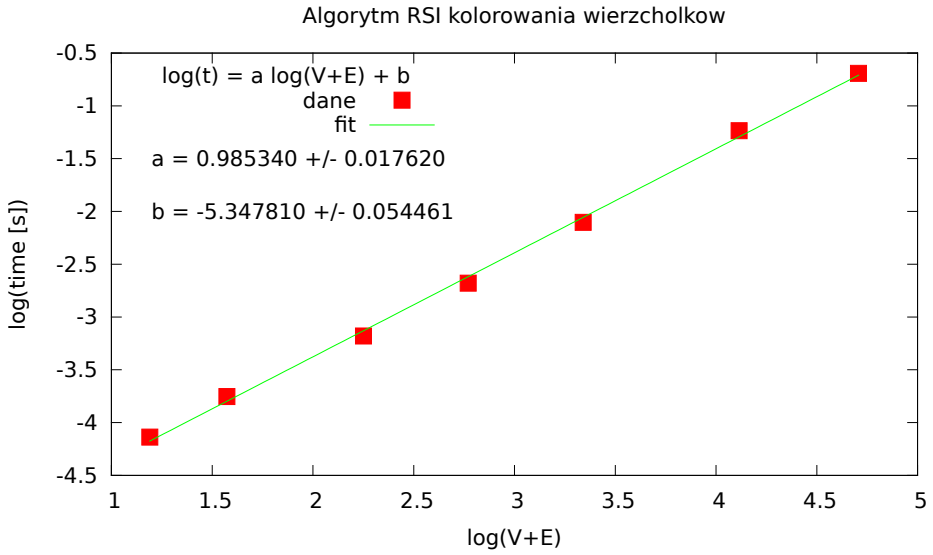
Rysunek C.5. Wykres wydajności algorytmu CS kolorowania wierzchołków. Współczynnik a bliski 1 sugeruje zależność liniową.



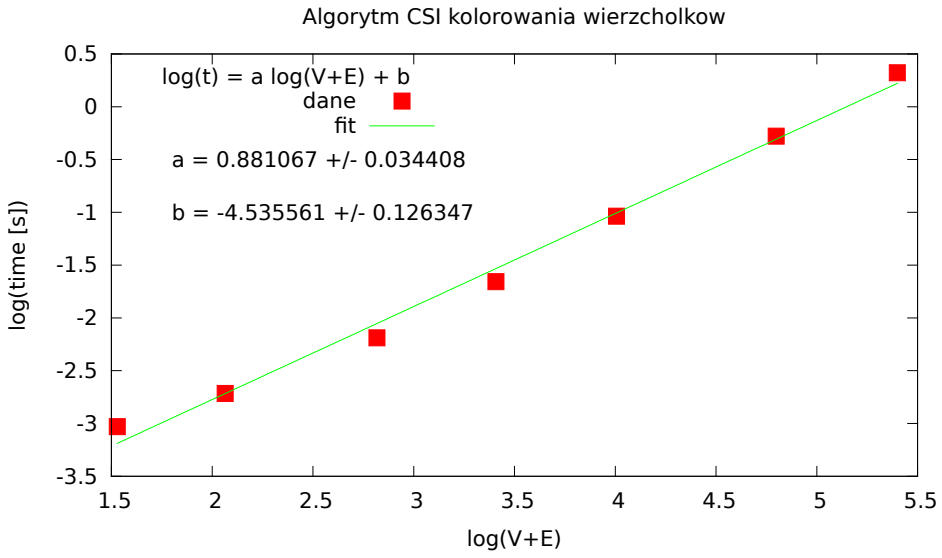
Rysunek C.6. Wykres wydajności algorytmu DSATUR kolorowania wierzchołków. Interpretacja nie jest jasna.



Rysunek C.7. Wykres wydajności algorytmu USI kolorowania wierzchołków. Współczynnik a bliski 1 sugeruje zależność liniową.



Rysunek C.8. Wykres wydajności algorytmu RSI kolorowania wierzchołków. Współczynnik a bliski 1 sugeruje zależność liniową.



Rysunek C.9. Wykres wydajności algorytmu CSI kolorowania wierzchołków. Współczynnik a bliski 1 sugeruje zależność liniową.

Bibliografia

[1] A. Kapanowski, graphs-dict, GitHub repository, 2015, https://github.com/ufkapano/graphs-dict/.

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

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

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

[5] Narsingh Deo, *Teoria grafów i jej zastosowania w technice i informatyce*, PWN, Warszawa 1980.

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

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

[8] Wikipedia, Graf dwudzielny, 2015, http://pl.wikipedia.org/wiki/Graf_dwudzielny.

[9] Wikipedia, Zbiór niezależny, 2015, http://pl.wikipedia.org/wiki/Zbiór_niezależny.

[10] Wikipedia, Problem ośmiu hetmanów, 2015, http://pl.wikipedia.org/wiki/Problem_ośmiu_hetmanów.

[11] R. Boppana, M. M. Halldórsson, *Approximating maximum independent sets by excluding subgraphs*, BIT Numerical Mathematics 32, 180–196 (1992).

[12] Wikipedia, Vertex cover, 2015, http://en.wikipedia.org/wiki/Vertex_cover.

[13] Wikipedia, Perfect graph, 2015, http://en.wikipedia.org/wiki/Perfect_graph.

[14] Wikipedia, Graf krawędziowy, 2015, http://pl.wikipedia.org/wiki/Graf_krawedziowy.

[15] Wikipedia, Kolorowanie grafu, 2015, http://pl.wikipedia.org/wiki/Kolorowanie_grafu.

[16] D. Brélaz, *New methods to color the vertices of a graph*, Comm. of the ACM 22 (4), 251-256 (1979).

[17] Wikipedia, Fisher-Yates shuffle, 2015, http://en.wikipedia.org/wiki/Fisher-Yates_shuffle.

[18] Andrew A. Radin, *Graph Coloring Heuristics from Investigation of Smallest Hard to Color Graphs*, MS Thesis, Rochester Institute of Technology, 2000.

[19] Wikipedia, Kolorowanie krawędzi, 2015, http://pl.wikipedia.org/wiki/Kolorowanie_krawędzi.