Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Kacper Dziubek

Nr albumu: 1054166

Badanie planarności grafów z językiem 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

Pragnę złożyć najserdeczniejsze podziękowania Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu, za nieocenioną pomoc, wsparcie, wyrozumiałość oraz poświecony czas i zaangażowanie, bez których ukończenie niniejszej pracy nie byłoby możliwe.

Streszczenie

W pracy przedstawiono podstawy teoretyczne i implementacje w języku Python wybranych algorytmów związanych z przeszukiwaniem grafów, badaniem spójności grafów, oraz z testowaniem planarności grafów. Do przechowywania grafów prostych stworzono klasę Graph. Implementacja powstała na bazie słownika słowników i pozwala przechowywać całe krawędzie jako obiekty klasy Edge.

W celu dydaktycznym omówiono algorytmy przeszukiwania grafów w głąb (DFS) i wszerz (BFS). Przedstawione zostały również ich przykładowe implementacje. Szczególna uwaga została poświęcona algorytmowi DFS, który jest podstawą kolejnych algorytmów zaprezentowanych w pracy. W ramach lepszego zrozumienia DFS zaimplementowane zostały algorytmy związane ze spójnością grafów: testowanie spójności, trywialny algorytm znajdowania mostów, algorytm Tarjana znajdowania mostów, trywialny algorytm znajdowania punktów artykulacji i algorytm Tarjana znajdowania punktów artykulacji.

W ramach badania planarności zaimplementowano algorytm lewy-prawy. Algorytm lewy-prawy sprawdza czy dany graf jest planarny i jeśli tak, to dla grafu abstrakcyjnego wyznacza graf topologiczny, czyli określa kolejność krawędzi wychodzących z wierzchołków oraz ściany grafu.

Każdy algorytm został zaimplementowany jako osobna klasa. Dla wszystkich algorytmów przygotowane zostały testy jednostkowe z wykorzystaniem modułu unittest. Ponadto wykonano eksperymenty komputerowe sprawdzające zgodność rzeczywistej wydajności algorytmów z założeniami teoretycznymi.

Słowa kluczowe: grafy, planarność, przeszukiwanie wszerz, przeszukiwanie w głąb, punkty artykulacji, mosty, algorytm lewy-prawy, graf topologiczny,

Abstract

In this work, theoretical foundations and Python implementation of selected algorithms for graph searching, connectivity, and for planarity testing of graphs are presented. The Graph class is created for simple graphs. It uses a dictionary of dictionaries data structure and it allows to store edges as instances of the Edge class.

Depth-first search (DFS) and breadth-first search (BFS) algorithms are described theoretically and their sample implementation is provided. Special attention is given to DFS which is essential for many algorithms. In order to better understand DFS properties, we present algorithms for finding bridges, cutnodes, and testing graph connectivity.

To test graph planarity, the left-right algorithm is implemented. It checks if an abstract graph is planar. In the case of a planar graph, the left-right algorithm calculates its topological graph (embedding). Then the ordering of the edges around each node is established.

Each algorithm was implemented as a separate class. All algorithms were tested with unit tests based on unittest framework. Additionally, computer experiments were conducted to test real performance against theoretical predictions.

Keywords: graphs, planarity testing, depth-first search, breadth-first search, cutnode, bridge, left-right algorithm, topologial graph

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Graf planarny można narysować na płaszczyźnie tak, że jego krawędzie nie będą się przecinać. Specyfika i właściwości grafów planarnych są znane od lat trzydziestych XX wieku, jednak pierwsze algorytmy testujące planarność grafów w czasie liniowym pojawiły się dopiero w latach siedemdziesiątych. Testowanie planarności grafów jest w literaturze uważane za trudne zagadnienie, a podstawowe podręczniki do teorii grafów i algorytmów nie poruszają wyczerpująco tego zagadnienia. Mimo to, grafy planarne odgrywają ważną rolę w obszarze teorii grafów i w rysowaniu grafów. Mają wiele interesujących właściwości: są rzadkie, 4-kolorowalne, pozwalają wykonać wiele operacji wydajniej niż dla ogólnych grafów [1]. Ze względu na brak przecięć, wizualizacje grafów planarnych są uważane za czytelniejsze.

W badaniach nad planarnością grafów pętle i krawędzie równoległe nie wnoszą nic nowego do problemu, dlatego bez straty ogólności można zawęzić badania do grafów prostych. Skierowanie grafu również nie jest istotne z punktu widzenia planarności, dlatego zawężimy się do grafów nieskierowanych. Można się również zawęzić do grafów spójnych, ponieważ każdą składową spójną bada się osobno. W pewnych algorytmach wyklucza się wierzchołki wiszące [$\deg(v) = 1$] i wierzchołki stopnia dwa [$\deg(v) = 2$].

1.1. Cele pracy

Niniejsza praca skupia się na badaniu planarności grafów, czyli przede wszystkim problemie testowania planarności grafów, oraz wybranych zagadnieniach związanych z grafami planarnymi. Pierwszym celem pracy jest jest implementacja w języku Python algorytmu testowania planarności lewy-prawy. Dla grafu planarnego oznacza to wyznaczenie grafu topologicznego. Algorytm ten został opublikowany przez de Fraysseix i Rosenstiehl [2], [3] i bazuje na właściwościach drzew przeszukiwań DFS. Na chwilę obecną jest on uznawany za najszybszy algorytm testujący planarność grafów [4]. Wiele algorytmów sprawdzających planarność grafów korzysta z rekurencyjnego algorymu przeszukiwania grafu w głąb (DFS). Z tego powodu w pracy dużo uwagi zostało poświęcone temu algorytmowi i jego specyficznym właściwościom. Drugim celem pracy jest więc pokazanie innych zastosowań DFS w algorytmach wyznaczających spójne i silnie spójne składowe grafu, oraz wyznaczających mosty i punkty artykulacji.

Do implementacji grafów oraz algorytmów wybrany został język Python. Python jest językiem wysokiego poziomu, który łączy w sobie przejrzystość, czytelność, oraz wydajność aplikacji. Biblioteka standardowa Pythona oferuje wiele przydatnych mechanizmów i struktur danych. Pozwala również

na łatwe testowanie napisanego kodu poprzez wsparcie dla pisania testów jednostkowych, którymi pokryte zostały prezentowane w pracy algorytmy.

Praca powstała przede wszsytkim w oparciu o znane książki na temat teorii grafów autorstwa Wilsona [5], Wojciechowskiego [6] oraz Deo [7]. Podstawy teoretyczne algorytmów zostały zaczerpnięte z książki *Wprowadzenie do algorytmów* [8], oraz *Algorytmy w C++. Część 5. Grafy* [9]. Informacji o algorytmach testujących planarność grafów dostarczyły przede wszystkim artykuły naukowe Hopcrofta i Tarjana [10], Boyera i Myrvold [11], de Fraysseix i Rosenstiehl [2], [3], oraz Ulrika Brandesa [12]. Szczególnie praca tego ostatniego okazała się pomocna, gdyż znajdują się w niej pseudokody opisujące poszczególne kroki algorytmu lewy-prawy.

1.2. Organizacja pracy

Praca została zorganizowana w następujący sposób: Rozdział 1 zawiera wprowadzenie do niniejszej pracy. Rozdział 2 to wprowadzenie do języka Python, opis podstawowych typów danych, składni, funkcji i programowania obiektowego. W rozdziale 3 znajdują się opisy wybranych zagadnień z teorii grafów, w szczególności związanych z badaniem planarności. Rozdział 4 przedstawia opis implementacji grafów używany w pracy. Pokazuje także jak wykorzystywać implementację w przykładowym skrypcie. Rozdział 5 zawiera opisy podstawowych algorytmów do przeszukiwania grafów, w szczególności algorytmu DFS, który jest podstawą do dalszych rozważań na temat spójności i planarności. W rozdziale 6 znajdują się algorytmy badające spójne składowe, punkty artykulacji i mosty w grafach. Rozdział 7 jest poświęcony badaniu planarności grafów i zawiera opis implementacji algorytmu lewy-prawy. Rozdział 8 zawiera podsumowanie pracy i wskazanie dalszych możliwych kierunków badań. Dodatkowo na końcu pracy załączone zostały dodatki prezentujące kodu źródłowe grafów, krawędzi oraz testy wydajnościowe algorytmów.

2. Wprowadzenie do Pythona

Python jest obiektowo-zorientowanym językiem programowania wysokiego poziomu [13]. Oznacza to, że jego składnia i słowa kluczowe mają maksymalnie ułatwić zrozumienie kodu dla programisty. Python jest językiem ogólnego przeznaczenia, czyli może być z powodzeniem wykorzystywany zarówno do budowania serwisów internetowych, jak i do tworzenia narzędzi administracyjnych w systemach operacyjnych. Python wspiera różne paradygmaty programowania, w tym: programowanie obiektowe, proceduralne i funkcyjne. Posiada dynamiczny system typowania, co oznacza, że obiekty mają swój typ, natomiast zmienne są referencjami do obiektów. W różnych chwilach działania programu jedna zmienna może być referencją do obiektów różnych typów. Zarządzanie pamięcią zostało zrealizowane z użyciem automatycznego odśmiecania pamięci (ang. *garbage collection*), czyli za jej zwalnianie odpowiedzialny jest programowy zarządca.

Charakterystyczną cechą Pythona jest jego składnia, w której bloki kodu są wydzielane poprzez wcięcia. Aby dodatkowo zwiększyć czytelność kodu używamy angielskich słów tam, gdzie inne języki korzystają ze znaków interpunkcyjnych. Ponadto, twórcy języka opracowali standard formatowania kodu (PEP8), który pomaga różnym programistom tworzyć kod o zbliżonym stylu, co jest szczególnie ważne w bibliotece standardowej. Standard określa takie rzeczy jak: szerokośc wcięć, maksymalny rozmiar linii kodu, sposób importowania modułów i inne.

Python rozwijany jest jako projekt *open source* zarządzany przez Python Software Foundation, która jest organizacją non-profit. Interetery języka dostępne są dla większości systemów operacyjnych, a rozbudowana biblioteka standardowa znacząco ułatwia wydajne tworzenie oprogramowania. Społeczność skupiona wokół języka Python może zgłaszać propozycje rozwoju języka m.in. za pomocą powiększającego się zbioru dokumentów PEP (ang. *Python Enhancement Proposal*). Istnieje również wiele grup dyskusyjnych, konferencji, nagród, które integrują społeczność i promują Pythona w świecie.

2.1. Typy danych

Python oferuje szeroki zakres podstawowych typów danych, zarówno typów prostych (typy liczbowe, logiczne) oraz kolekcji.

2.1.1. Typy proste

Typy proste w Pythonie dzielą się na na typy numeryczne i tekstowe. Python jest językiem silnie typowanym, co oznacza, że jakakolwiek konwersja typów, która mogłaby powodować zmianę precyzji, musi zostać jawnie

zlecona przez programistę. Tekst zawierający liczbę musi być jawnie konwertowany na typ liczbowy i odwrotnie, liczba musi być jawnie konwertowana na tekst. Pomiędzy niektórymi typami występuje niejawna konwersja, np. liczby całkowite są automatycznie konwertowane na liczby zmiennoprzecinkowe, jeśli zachodzi taka potrzeba. W tabeli 2.1 zaprezentowane zostały typy podstawowe.

Typ	Opis	Przykład
str	napis w ACSII	"To jest napis w Pythonie"
unicode	napis w Unicode	u"Unicode w Pythonie 2"
int	liczba całkowita	669
long	liczba całkowita długa	49542363454L
float	liczba zmiennoprzecinkowa	3.3345
complex	liczba zespolona	5+2.6j
bool	wartość logiczna (prawda, fałsz)	True, False
None	pusty obiekt	None

Tabela 2.1: Typy proste w języku Python

2.1.2. Kolekcje

Python oferuje dwa rodzaje uporządkowanych kolekcji, przypominających tablice z innych języków programowania - listy (typ **list**) i krotki (typ **tuple**). Obie kolekcje są iterowalne i mogą zawierać elementy różnych typów. Główna różnica pomiędzy nimi polega na tym, że listy są modyfikowalne, zaś krotki niezmienne.

Innymi typami kolekcji są kolekcje nieuporządkowane: słowniki (typ **dict**), w innych językach znane jako mapy lub tablice asocjacyjne, oraz zbiory (typy **set** i **frozenset**). Klucze słowników oraz elementy zbiorów muszą być obiektami haszowalnymi. W tabeli 2.2 zaprezentowane zostały kolekcje oraz przykłady ich użycia.

Typ	Opis	Przykład
tuple	krotka (typ niezmienny)	(1, 2, "string")
list	lista (zmienna dlugość i zawartość)	[1, 2, "string", {}]
set	zbiór (zmienny)	**set**([1, "a"])
frozenset	zbiór (niezmienny)	**frozenset**([1, "a"])
dict	słownik (zmienny)	{'liczba': 3.3345, 1: True}

Tabela 2.2: Kolekcje w języku Python

Listing 2.1. Przykłady użycia kolekcji w Pythonie.

```
# Utworzenie krotki.
empty_tuple = tuple()
new_tuple = (1, 2.3, "word", True)

# Utworzenie listy.
empty_list = list()
new_list = [4, 5.6, "abc", False, None]
```

```
# Utworzenie zbioru.
empty_set = set()
set_from_list = set([1, 2, 2, 3, 3, 3])    # set([1, 2, 3])

# Utworzenie slownika.
empty_dict = dict()
new_dict = {"key": "value", 3: 4.5}
```

2.2. Składnia języka

Python jest językiem, którego składnia ma być przede wszystkim czytelna. Częste używanie angielskich słów, oraz stosunkowo mała ilość konstrukcji składniowych (w porównianiu do takich języków jak C, Perl, czy Pascal), sprawiają, że układ graficzny kodu jest prosty i przejrzysty.

2.2.1. System wcięć

Do rozdzielania bloków kodu w Pythonie stosujemy system wcięć oparty o białe znaki zamiast interpunkcji (np. { } w języku C), czy słów kluczowych (np. begin, end w języku Pascal). Pojedyncza instrukcja kończy się wraz końcem linii, chyba że jawnie poinformujemy interpeter o złamianiu wiersza wykorzystując znak ukośnika wstecznego \(ang. *backslash*).

Listing 2.2. Przykłady systemu wcięć w Pythonie.

```
for x in xrange(10):
    if x % 2 == 1:
        print x
    else:
        print "liczba parzysta"
```

2.2.2. Operatory

Operatory w Pythonie możemy podzielić na trzy grupy: operatory arytmetyczne, logiczne i porównania. Do operatorów numerycznych zaliczamy +, -, *, /, %(dzielenie modulo) i ** (potęgowanie).

Operatory porównania to ==, !=, <, >, <=, >= oraz operator **is**. Operatory te jako wynik swojego działania zwracają wartość logiczną True albo False.

Operatory logiczne to **and** oraz **or**, odpowiadające kolejno koniunkcji i alternatywie. Rezultatem ich działania jest wartość ostatnio ewaluowanego operanda, a nie wartość logiczna True albo False, ale najbezpieczniej jest używać tych operatorów z wyrażeniami logicznymi.

Listing 2.3. Przykłady działania operatorów logicznych.

```
(4 or 5)  # zwroci wartosc 4
(4 and 5)  # zwroci wartosc 5
```

2.2.3. Komentarze

Python posiada dwa sposoby dodawania adnotacji do kodu. Pierwszy z nich polega na użyciu komentarza do wytłumaczenia konkretnej części kodu i znany jest jako *komentarz jednolinijkowy*. Rozpoczynamy go stosując znak *hash* (#), natomiast jego koniec następuje w momencie przejścia do kolejnej linii. Komentarze zajmujące więcej niż jedną linijkę tworzymy przez wstawienie wielolinijkowego stringa, który nie jest przypisany do zmiennej, ani ewaluowany w żaden inny sposób. Jego początek i koniec oznaczamy przez użycie trzech znaków cudzysłowu lub apostrofu obok siebie (""" lub ''').

Listing 2.4. Przykłady komentarzy w Pythonie.

```
for x in xrange(10):
    if x % 2 == 1:
        print x  # wypisz x
    else:
        """
        wypisz komunikat
        """
        print "liczba parzysta"
```

Komentarzy wielolinijkowych używamy również do tworzenia dokumentacji w kodzie. Wstawienie takiego komentarza jako pierwszego elementu funkcji, klasy lub modułu, automatycznie przypisuje jego treść do atrybutu `__doc__` danego obiektu, który ma w zamierzeniu przechowywać czytelny i zrozumiały opis działania lub celu tego obiektu.

Listing 2.5. Przykład dokumentacji w Pythonie.

```
def getline():
    """
    Zwraca jedna linie odczytana ze standardowego wejscia.
    """
    return sys.stdin.readline()
```

2.3. Instrukcje sterujące

Trudno doszukać się języka programowania, który nie korzystałby z instrukcji sterujących. Python nie jest pod tym względem językiem wyjątkowym, aczkolwiek ilość instrukcji do wyboru jest stosunkowo ograniczona względem innych języków programowania.

2.3.1. Instrukcja warunkowa

Instrukcja **if** służy do warunkowego wykonania znajdującego się pod nią bloku kodu. Opcjonalnie, obok instrukcji if możemy również używać słów kluczowych **elif** (sprawdzanie innego warunku) i **else** (żaden z powyższych warunków).

Listing 2.6. Przykład instrukcji warunkowych.

```
if x > 0:
    print "X is positive"
elif x < 0:
    print "X is negative"
else:
    print "x is zero"
```

2.3.2. Pętle

W Pythonie mamy dostęp do dwóch rodzajów pętli. Pętla **for** iteruje po kolejnych elementach kolekcji, która została do niej przesłana. Jest to pewna różnica w stosunku do innych języków programowania w których należy jawnie zwiększać wartość indeksu iteratora. Do wytworzenia podstawowych kolekcji liczb z zakresu od 0 do $n-1$ włącznie możemy używać funkcji **range**(n) lub **xrange**(n).

Listing 2.7. Przykład pętli **for**.

```
n = 10
for x in xrange(n):
    print n
```

Drugim dostępnym rodzajem pętli jest pętla **while**, która wykonuje znajdujący się pod nią blok kodu, dopóki wyrażenie warunkowe do niej przesłane jest prawdziwe. Jeżeli wyrażenie warunkowe jest fałszem, to pętla nie wykona się ani razu.

Listing 2.8. Przykład pętli **while**.

```
n = 0
while n < 10:
    print n
    n += 1
```

W powyższych pętlach możemy korzystać z instrukcji **break**, która powoduje natychmiastowe wyjście z pętli, oraz **continue**, która wyzwala następne przejście pętli.

2.4. Funkcje

Funkcje służą do grupowania spójnych i powtarzalnych fragmentów kodu w jedną całość. Mają one znaczący wpływ na czytelność kodu, oraz zapobiegają jego duplikowaniu. W Pythonie funkcje deklarowane są słowem kluczowym **def**, po którym następuje nazwa funkcji. W ten sposób tworzony jest obiekt funkcji o nadanej nazwie, który można wywołać bezpośrednio lub przekazać do innej funkcji lub obiektu. Python pozwala na stosowanie w funkcjach argumentów domyślnych, argumentów nazwanych, oraz zmiennej liczby argumentów. Domyślnie wszystkie funkcje zwracają obiekt None, chyba że wskazana wartość zostanie jawnie zwrócona z użyciem słowa kluczowego **return**.

Listing 2.9. Przykład funkcji w Pythonie.

```
def function_example(x, y=10):    # y ma wartosc domyslna
    """Mnozenie argumentow."""
    return x * y
```

2.4.1. Programowanie funkcyjne

Jedną z zalet języka Python jest wsparcie dla programowania funkcyjnego. Czyni to pracę z listami i innymi kolekcjami znacznie łatwiejszą. Szczególnie przydatne są *listy składane* (ang. *list comprehension*), które pozwalają znacząco skrócić ilość kodu. Poniższy przykład pokazuje w jaki sposób utworzyć listę kolejnych potęg liczby 10.

Listing 2.10. Listy składane w Pythonie.

```
powers = [10 ** n for n in xrange(6)]
```

2.5. Obiektowość

Python wspiera również techniki programowania obiektowego, takie jak dziedziczenie (w tym dziedziczenie wielokrotne), polimorfizm i abstrakcje. W Pythonie wszystko (funkcje, klasy, moduły) jest obiektem.

2.5.1. Klasy

Klasy służą do definiowania szablonu (nowego typu) według którego następnie są tworzone instancje danego typu. Do ich deklarowania wykorzystuje się słowo kluczowe **class**, po którym następuje nazwa klasy. Przy tworzeniu obiektu danej klasy wywoływany jest konstruktor, któremu w Pythonie odpowiada metoda __init__. Metoda ta może przyjmować zestaw argumentów, które zostaną przekazane do obiektu w momencie jego tworzenia.

Każda niestatyczna metoda wewnątrz klasy jako pierwszy argument przyjmuje zmienną **self**, która jest referencją do obiektu, na którym wywoływana jest dana metoda. Python nie wspiera metod prywatnych, ani chronionych na poziomie składni języka, więc nie ma w nim słów kluczowych odpowiadających modyfikatorom dostępu. Wszystkie pola i metody są dostępne publicznie, jednak zgodnie z konwencją prywatne zasoby klas przyjęło się oznaczać poprzez znak podkeślenia (ang. *underscore*) poprzedzający nazwę danego zasobu.

Listing 2.11. Definicja klasy w Pythonie

```
class SampleClass:

    def __init__(self, value):    # konstruktor
        self.value = value

    def __str__(self):    # postac napisowa
        return str(self.value)
```

```
    def get_value(self):
        return self.value
```

2.5.2. Dziedziczenie

Klasy w Pythonie mogą dziedziczyć z dowolnego typu, w tym z typów wbudowanych takich jak napisy, listy, czy słowniki. Możliwe jest również dziedziczenie wielokrotne, czyli dziedzenie metod i atrybutów więcej niż z jednej klasy. Aby określić, że tworzona klasa dziedziczy po innej klasie (lub klasach), po jej nazwie wstawia się w nawiasie nazwę klasy bazowej.

Listing 2.12. Dziedziczenie w Pythonie.

```
class NewClass(SampleClass):

    def __init__(self):
        SampleClass.__init__("data")    # akcje starego konstruktora
        # nowe akcje ...
```

2.5.3. Wyjątki

Wyjątki są nowoczesnym i powszechnie stosowanym mechanizmem do obsługi błędów i sytuacji nietypowych w programach. W Pythonie wyjątki to obiekty, których klasy dziedziczą po klasie Exception. Do rzucenia wyjątku używane jest słowo kluczowe **raise**. Rzucony wyjątek można przechwycić i odpowiednio obsłużyć, używając konstrukcji składającej się ze słów kluczowych **try**, **except** i **finally** (opcjonalnie).

Listing 2.13. Przykład obsługi wyjątków.

```
def exception_function():
    raise ValueError("this is an exception")   # rzucenie wyjatku

try:
    exception_function()
except ValueError as exception:   # przechwycenie wyjatku
    print "Exception: " + str(exception)
```

3. Teoria grafów

Teoria grafów to dział matematyki i informatyki zajmujący się badaniem grafów i ich właściwości[14]. W literaturze można spotkać różne definicje odnoszące się do teorii grafów. Z tego powodu poniżej podamy wybrane definicje pojęć używanych w pracy, głównie opierając się na podręcznikach Cormena [8], oraz Wilsona [5]. Zbiór pojęć ograniczymy do tych najbardziej niezbędnych podczas rozważania planarności grafów.

3.1. Grafy

Graf nieskierowany (ang. *undirected graph*) to para $G = (V, E)$, gdzie V to niepusty zbiór wierzchołków, a E to zbiór krawędzi nieskierowanych. *Krawędź nieskierowana* $\{s, t\}$ to dwuelementowy podzbiór zbioru V. Wierzchołek t jest *sąsiadem* wierzchołka s, jeżeli w grafie istnieje krawędź $\{s, t\}$. Jeśli krawędź $\{s, t\}$ łączy dwa wierzchołki to mówimy, że jest z nimi *incydentna*.

Graf skierowany (ang. *directed graph*) to para $G = (V, E)$, gdzie V to niepusty zbiór wierzchołków, a E to zbiór krawędzi skierowanych. *Krawędź skierowana* (s, t) to uporządkowana para dwóch różnych wierzchołków ze zbioru V. Krawędź wychodzi z wierzchołka s (*wierzchołek źródłowy*) i wchodzi do wierzchołka t (*wierzchołek docelowy*).

Pętla (pętla własna) to krawędź łącząca wierzchołek z nim samym. Dwa wierzchołki mogą być połączone ze sobą za pomocą więcej niż jednej krawędzi. Wtedy takie krawędzie nazywamy *krawędzią wielokrotną* (ang. *multi-edge*) lub krawędziami równoległymi. *Graf prosty* (ang. *simple graph*) to graf, który nie zawiera pętli, ani krawędzi wielokrotnych.

Graf pełny to graf prosty nieskierowany, w którym dla każdej pary wierzchołków istnieje krawędź je łącząca. Graf pełny o n wierzchołkach oznacza się K_n.

Graf dwudzielny to trójka $G = (U, V, E)$, gdzie U i V są rozłącznymi zbiorami wierzchołków, a $E \subseteq U \times V$ to zbiór krawędzi.

Graf pełny dwudzielny to graf dwudzielny, w którym wszystkie pary wierzchołków (s, t) takie, że $s \in U$, $t \in V$, są połączone krawędzią. Graf pełny dwudzielny oznacza się $K_{r,s}$, gdzie $r = |U|$, a $s = |V|$. Liczba krawędzi w takim grafie wynosi $|E| = rs$.

Pętle, krawędzie wielokrotne, oraz skierowanie grafu, nie mają wpływu na planarność grafu, w związku z czym w dalszej części pracy określenie *graf* będzie się odnosiło do grafów prostych nieskierowanych.

3.2. Ścieżki, cykle i spójność

Ścieżką (drogą) P z s do t w grafie $G = (V, E)$ nazywamy sekwencję wierzchołków $(v_0, v_1, ..., v_n)$, gdzie $v_0 = s$, $v_n = t$, oraz $(v_{i-1}, v_i)(i = 1, ..., n)$ są krawędziami z E. Długość ścieżki P wynosi n. *Ścieżka prosta* to ścieżka, w której wszystkie wierzchołki są różne.

Cykl to ścieżka zamknięta, czyli taka, której koniec (ostatni wierzchołek) jest identyczny z początkiem (pierwszym wierzchołkiem). Graf, który nie zawiera cykli, nazywamy *acyklicznym. Cykl prosty* to cykl w którym wszystkie wierzchołki są różne, z wyjątkiem ostatniego.

Graf spójny (ang. *connected graph*) jest to graf, w którym dla każdej pary wierzchołków (s, t) istnieje ścieżka, która je ze sobą łaczy. Graf niespójny to graf, który nie jest spójny. *Graf k-spójny* to graf, który po usunięciu dowolnie wybranych $k - 1$ wierzchołków i incydentnych z nimi krawędzi pozostaje spójny.

Spójna składowa grafu $G = (V, E)$ to spójny podgraf grafu G nie zawarty w większym podgrafie spójnym grafu G. Graf spójny ma tylko jedną spójną składową.

3.3. Drzewa i lasy

Drzewo jest to graf nieskierowany, spójny i acykliczny. *Las* jest to graf nieskierowany, niespójny i acykliczny.

Drzewo z wyróżnionym wierzchołkiem, korzeniem (ang. *root*), nazywamy *drzewem ukorzenionym.* W takim drzewie, dla dowolnej ścieżki prostej rozpoczynającej się w korzeniu i zawierającej wierzchołek v określamy:
— Przodków - wierzchołki znajdujące się na ścieżce przed wierzchołkiem v,
— Potomków - wierzchołki znajdujące się na ścieżce po wierzchołku v,
— Rodzica - wierzchołek znajdujący się na ścieżce bezpośrednio przed wierzchołkiem v,
— Dziecko - wierzchołek znajdujący się na ścieżce bezpośrednio po wierzchołku v,
— Braci - wierzchołki mające wspólnego ojca.

Drzewo rozpinające grafu jest drzewem, w którym zawierają się wszystkie wierzchołki grafu oraz niektóre z jego krawędzi.

3.4. Planarność

Grafem planarnym (ang. *planar graph*) nazywamy graf, który można narysować na płaszczyźnie bez przecięć krawędzi. Każdy taki rysunek nazywamy *grafem płaskim* (ang. *plane graph*). Należy zaznaczyć, że planarność jest wewnętrzną własnością danego grafu, niezależną od sposobu narysowania grafu. Można na przykład narysować graf planarny z przecinającymi się krawędziami, co nie przekreśla jego planarności.

W grafach płaskich, poza wierzchołkami i krawędziami, rozważa się również *ściany* (ang. *faces*), czyli obszary płaszczyzny otoczone krawędziami grafu. Ściana w grafie płaskim G to spójny obszar płaszczyzny po usunięciu

linii reprezentujących krawędzie. Innymi słowy ściana to zbiór punktów płaszczyzny, które da się połączyć krzywą nieprzecinającą żadnej krawędzi. Każdy graf płaski posiada jedną ścianę nieograniczoną, zewnętrzną (ang. *external/outer face*), oraz skończoną liczbę ścian wewnetrznych, zamkniętych, czyli ograniczonych krawędziami grafu (ang. *inner faces*). Ściana zewnętrzna nie jest specjalnie wyróżniona, ponieważ przez przekształcenie wykorzystujące rzutowanie na sferę każdą ścianę grafu planarnego można uczynić ścianą zewnętrzną.

Graf M jest *minorem* grafu G, gdy możemy otrzymać M poprzez usuwanie wierzchołków i krawędzi lub ściąganie krawędzi.

Twierdzenie (Wagner, 1936; Fáry, 1948): Każdy planarny graf prosty może być narysowany za pomocą odcinków [5].

Twierdzenie Eulera (1750): Rozważmy płaski rysunek grafu planarnego G, na którym znajduje się n wierzchołków, m krawędzi i f ścian. Wtedy

$$n - m + f = 2. \tag{3.1}$$

Dowód przez indukcję względem liczby krawędzi można znaleźć w książce Wilsona [5]. Jeżeli graf G ma k składowych spójnych, to wzór ma postać $n - m + f = k + 1$. Ciekawym wnioskiem z twierdzenia Eulera jest następne twierdzenie.

Twierdzenie: Każdy planarny graf prosty zawiera wierzchołek stopnia co najwyżej 5 [5]. W przeciwnym razie mielibyśmy nierówność $6n \leqslant 2m$, a więc $3n \leqslant m$. Z Kryterium I otrzymujemy sprzeczność $3n \leqslant 3n - 6$.

3.5. Kryteria planarności

W literaturze znanych jest kilka kryteriów planarności. Poniżej znajdują się najważniejsze z nich.

Twierdzenie Kuratowskiego (1930): Dany graf jest planarny wtedy i tylko wtedy, gdy nie zawiera podgrafu homeomorficznego z grafem K_5 lub z grafiem $K_{3,3}$ [5].

Twierdzenie Wagnera (1937): Dany graf jest planarny wtedy i tylko wtedy, gdy jego minory nie zawierają grafu K_5 ani grafu $K_{3,3}$ [1].

Dla grafu prostego spójnego $G = (V, E)$ oznaczamy $n = |V|$, $m = |E|$.

Kryterium I: Jeżeli $n \geqslant 3$, to zachodzi warunek $m \leqslant 3n - 6$.

Kryterium II: Jeżeli $n \geqslant 3$ i graf nie ma cykli o długości 3 (czyli trójkątów), to $m \leqslant 2n - 4$.

Kryteria te są wnioskami z twierdzenia Eulera [5]. Dzięki tym kryteriom można podać prosty dowód twierdzenia o dwóch ważnych grafach nieplanarnych.

Twierdzenie: Grafy K_5 i $K_{3,3}$ są nieplanarne. Dla grafu K_5 z Kryterium I otrzymujemy sprzeczność $10 \leqslant 9$. Dla grafu $K_{3,3}$ z Kryterium II otrzymujemy sprzeczność $9 \leqslant 8$.

3.6. Przykładowe grafy planarne

Podamy kilka ciekawych przykładów rodzin grafów planarnych.

3.6.1. Grafy planarne maksymalne

Graf planarny nazywamy *maksymalnym*, kiedy dodanie nowej krawędzi (przy ustalonym zbiorze wierzchołków) spowoduje powstanie grafu, który nie jest planarny. W grafie planarnym maksymalnym z $n \geqslant 3$ każda ściana jest trójkątem (brzeg ściany ma trzy krawędzie). Stąd dostajemy $3f = 2m$, a korzystając z twierdzenia Eulera wyznaczamy $f = 2n - 4$, $m = 3n - 6$.

Sieć Apoloniusza (ang. *Apollonian network*) [15] jest przykładem grafu planarnego maksymalnego. Sieć powstaje przez rekurencyjne dzielenie trójkątnych ścian na trzy trójkąty, rozpoczynając od pojedyńczego trójkąta. Apoloniusz z Pergi (ok. 260 p.n.e. - ok. 190 p.n.e.) znany jest także z problemu Apoloniusza polegającego na stworzeniu okręgu stycznego do trzech innych okręgów.

3.6.2. Grafy planarne ze ścianami kwadratowymi

Graf ma ściany będące kwadratami, czyli każda ściana jest ograniczona przez cztery krawędzie. Stąd dostajemy $4f = 2m$, a korzystając z twierdzenia Eulera wyznaczamy $f = n - 2$, $m = 2n - 4$. Najprostszy graf z tej kategorii to graf cykliczny C_4. Wszystkie grafy z tej grupy są dwudzielne, podobnie jak inne grafy planarne ze ścianami ograniczonymi przez parzystą liczbę krawędzi.

W teorii grafów mówi się o grafach kwadratowych (ang. *squaregraph*) [16], w których ściany wewnętrzne są kwadratami, a ściana zewnętrzna niekoniecznie. Ponadto każdy wierzchołek stopnia trzy lub dwa graniczy ze ścianą zewnętrzną. Do grafów kwadratowych należą drzewa, graf krata (ang. *lattice/mesh/grid graph*), oraz tzw. *gear graph* (powstaje z grafu koła W_n przez dodanie dodatkowych wierzchołków pomiędzy każdą parą sąsiednich wierzchołków, znajdujących się na brzegu koła).

3.6.3. Grafy zewnętrznoplanarne

Grafy zewnętrznoplanarne (ang. *outerplanar graphs*) [17] to rodzaj grafów planarnych, w których wszystkie wierzchołki należą do zewnętrznej ściany. Grafy zewnętrznoplanarne jako pierwsi opisali Chartrand i Harary [18] w nawiązaniu do problemu określania planarności grafów powstałych w skutek łączenia dwóch grafów bazowych z wykorzystaniem skojarzeń. Wszystkie grafy

zewnętrznoplanarne, które nie zawierają pętli, można pokolorować z użyciem trzech kolorów.

3.6.4. Grafy Halina

Graf Halina (ang. *Halin graph*) [19] jest konstruowany z drzewa mającego przynajmniej cztery wierzchołki. Żaden wierzchołek nie jest wierzchołkiem stopnia drugiego (z dwoma sąsiadami). Drzewo jest narysowane na płaszczyźnie tak, aby jego krawędzie się nie przecinały Do takiego drzewa dodajemy cykl łączący po kolei wszystkie liście drzewa. Utworzony w ten sposób graf jest planarny i 3-spójny. Przykładem grafu Halina jest graf koło W_n.

3.7. Grubość grafu

Rozszerzeniem pojęcia płaskości grafu jest *grubość grafu*. Jest to najmniejsza liczba rozłącznych podzbiorów zbioru krawędzi grafu, takich że krawędzie każdego z tych podzbiorów rozpięte na zbiorze wierzchołków grafu tworzą graf planarny. Obliczenie grubości grafu jest trudne, a dostępne algorytmy mało efektywne [6].

4. Implementacja grafów

W języku Python grafy można reprezentować na kilka sposobów. Możliwe jest stworzenie macierzy sąsiedztwa, która technicznie implementowana jest jako lista list. Złożoność pamięciowa takiego rozwiązania wynosi $O(n^2)$ względem liczby wierzchołków co sprawia, że nie jest to najbardziej optymalne rozwiązanie, szczególnie przy grafach planarnych, które często są grafami rzadkimi. Inną możliwością jest przedstawianie grafu za pomocą słowników i list. W tym rozwiązaniu graf jest słownikiem, w którym kluczami są wierzchołki, a każdemu kluczowi odpowiada lista wierzchołków połączych krawędziami z tym wierzchołkiem.

W rozważaniach nad planarnością przydatna okazuje się możliwość manipulowania samym obiektem krawędzi, dlatego jako implementację grafu wykorzystamy słownik słowników. Tak jak w poprzedniej implementacji kluczami są wierzchołki, jednak tym razem zawierają one słowniki, w których jako klucze przechowywane są wierzchołki połączone krawędzią z danym wierzchołkiem, a wartościami odpowiadające tym połączeniom obiekty krawędzi.

4.1. Struktury danych dla grafów abstrakcyjnych

Wierzchołek: W implementacji macierzowej grafu wierzchołki są liczbami całkowitymi od 0 do $n-1$, gdzie n jest liczbą wszystkich wierzchołków w grafie. W implementacji słownikowej wierzchołki mogą być dowolnymi obiektami hashowalnymi, a najczęściej są to obiekty typu `string` lub liczby.

Krawędź skierowana: Krawędź skierowana grafu to instancja klasy `Edge`, która inicjowana jest wierzchołkiem źródłowym i docelowym. Do wierzchołków mamy dostęp odpowiednio przez atrybyty `source` i `target`. Krawędzie są obiektami hashowalnymi i można je porównywać.

Krawędź nieskierowana: Krawędź nieskierowana to instancja klasy `UndirectedEdge`, która dziedziczy po klasie `Edge`. W krawędzi nieskierowanej nie ma rozróżnienia pomiędzy wierzchołkiem źródłowym a docelowym. Kostruktor sortuje podane dwa wierzchołki tak, że `source` $<$ `target`.

Graf: Graf jest instancją klasy `Graph` i zaimplementowany jest jako słownik słowników. Graf posiada atrybut `directed`, który określa czy graf jest skierowany czy nie. Skierowanie grafu nie ma wpływu na jego planarność, więc w dalszej części będziemy posługiwali grafami nieskierowanymi.

Drzewo rozpinające: Drzewo rozpinające grafu powstaje naturalnie w wielu algorytmach, między innymi podczas przeszukiwania grafu w głąb. Drzewa

można przechowywać jako słownik, gdzie kluczami są dzieci, a wartościami ich rodzic (lub tez krawędź prowadząca do rodzica). Innym sposobem przechowywania drzewa jest graf prosty, skierowany (drzewo z korzeniem) lub nieskierowany (drzewo bez korzenia).

4.2. Interfejs klasy Graph

Interfejs klasy Graph, pomimo innej wewnętrznej implementacji, pozostał niezmieniony w stosunku do poprzednio publikowanej wersji [20]. Aktualna implementacja wzbogaca go natomiast o dwie metody. Pierwsza z nich zwraca ilość ścian w grafie planarnym. Druga zwraca generator ścian grafu. Interfejs nowych metod został przedstawiony w tabeli 4.1.

Operacja	Opis	Metoda
G.f()	ilość ścian w grafie G	f()
G.iterfaces()	iterator ścian w grafie G	iterfaces()

Tabela 4.1: Metody dodane do interfejsu klasy Graph.

4.3. Struktury danych dla grafów planarnych

W badaniu planarności wykorzystamy dodatkowe struktury danych. Rozszerzona zostanie również klasa Graph, do której dodane zostaną struktury edge_next i edge_next. Są to słowniki, w których przechowywane będą listy cykliczne krawędzi wychodzących z wierzchołka, odpowiednio w kierunku przeciwnym i zgodnym z ruchem wskazówek zegara. Jest to pomysł oparty na mapach kombinatorycznych [21], które są używane do modelowana struktur topologicznych, np. sympleksów. Pełny kod źródłowy klasy Graph znajduje się w dodatku A.

Interwał: Interwał to zbiór dwóch krawędzi grafu. Dodatkowo definiujemy metodę pozwalającą łatwo stwierdzić czy interwał jest zbiorem pustym. Klasa Interval używana jest w algorytmie lewy-prawy zdefiniowanym w 7.1.

Listing 4.1. Klasa Interval z modułu intervals.

```
#!/usr/bin/python

class Interval:
    """The class defining an interval of return edges."""

    def __init__(self, low_edge=None, high_edge=None):
        """Load up an interval instance."""
        self.low = low_edge
        self.high = high_edge

    def __repr__(self):
        """Compute the string representation of the interval."""
        return "Interval({0}, {1})".format(str(self.low), str(self.high))

    def __nonzero__(self):
```

```
        """
        Method to test if interval is an empty set.
        Returns False if interval is empty.
        """
        return bool(self.low) or bool(self.high)
```

Para: Para służy do przechowywania dwóch interwałów. Klasa Pair jest używana w algorytmie lewy-prawy zdefiniowanym w 7.1,

Listing 4.2. Klasa Pair z modułu pairs.

```
#!/usr/bin/python

class Pair:
    """The class defining a pair of intervals."""

    def __init__(self, left_interval=None, right_interval=None):
        """Load up a pair instance."""
        self.left = left_interval
        self.right = right_interval

    def __repr__(self):
        """Compute the string representation of the pair."""
        return "Pair({0}, {1}".format(str(self.left), str(self.right))
```

4.4. Przykładowa sesja interaktywna

Poniższa sesja interaktywna pokazuje przykładowe wykorzystanie struktur danych użytych do implementacji grafów.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> graph = Graph()    # undirected graph
>>> graph.add_node(1)    # add node
>>> graph.add_node(2)
>>> graph.add_edge(Edge(1, 2))    # add edge
>>> list(graph.iternodes())
>>> [1, 2]
>>> print graph.v(), graph.e()    # number of nodes and edges
>>> 2, 1
>>> for edge in graph.iteroutedges():
>>>     print edge.source, edge.target
```

5. Przeszukiwanie grafów

Zbadanie struktury grafu jest często pierwszą czynnością wykonywaną przy badaniu nieznanego grafu. Przeszukiwanie wszerz (ang. *breadth-first search*, BFS) i przeszukiwanie w głąb (ang. *depth-first search*, DFS) to dwa podstawowe sposoby przeszukiwania grafu. Omówimy je korzystając z podręcznika Cormena [8], ponieważ autorzy bardzo dobrze przedstawili istotne własności tych dwóch algorytmów. Szczególnie ważne jest zrozumienie własności algorytmu DFS, ponieważ jest on podstawą wielu innych algorytmów, np. związanych z badaniem spójności i planarności grafów.

Podane implementacje algorytmów BFS i DFS odwiedzają sąsiadów danego wierzchołka przy pomocy metody iteroutedges. Jest to faktycznie iteracja po całych krawędziach wychodzących. Dzięki temu mamy jednoznaczność działania algorytmów dla multigrafów, a także mamy łatwy dostęp do atrybutów krawędzi prowadzących do sąsiadów.

5.1. Przeszukiwanie wszerz (BFS)

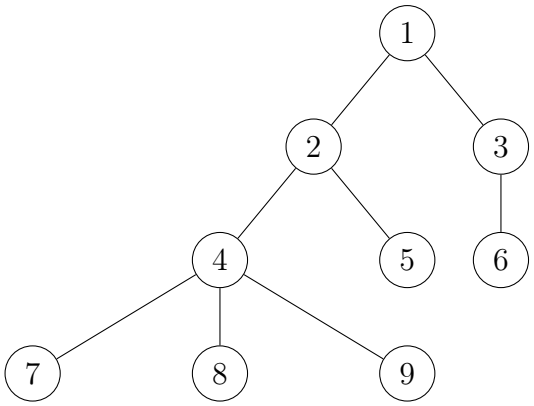
Algorytm przeszukiwania wszerz jest jednym z podstawowych algorytmów operujących na grafach. Wierzchołki odwiedzane są w kolejności definiowanej przez odległość krawędziową od wierzchołka początkowego, czyli najpierw odwiedzane są wierzchołki oddalone o jedną krawędź, następnie dwie itd. Wierzchołki do odwiedzenia przechowywane są w kolejce.

Dane wejściowe: Dowolny graf, opcjonalnie wierzchołek początkowy.

Problem: Przeszukiwanie grafu wszerz.

Opis algorytmu: Algorytm rozpoczynamy od dowolnego wierzchołka grafu. Nieodwiedzone wierzchołki osiągalne z wybranego wierzchołka wstawiamy do kolejki. Następnie wybrany wierzchołek zostaje oznaczony jako odwiedzony (w tym momencie można wykonanać dodatkowe czynności na wierzchołku). W kolejnym kroku, algorytm zdejmuje z kolejki pierwszy wierzchołek i powtarza na nim całą procedurę. Algorytm zostaje zakończony kiedy kolejka jest pusta, czyli nie zostały już żadne wierzchołki do odwiedzenia.

W czasie działania algorytmu tworzone jest *drzewo rozpinające grafu BFS* o korzeniu w wierzchołku od którego rozpoczęło się przeszukiwanie.



Rysunek 5.1. Kolejność odwiedzania wierzchołków w algorytmie BFS.

Złożoność: Złożoność czasowa algorytmu jest uzależniona od implementacji grafu. Jeśli graf reprezentowany jest jako macierz sąsiedztwa to złożoność czasowa wynosi $O(V^2)$. Przy reprezentacji grafu poprzez listy sąsiedztwa złożoność czasowa wynosi $O(V + E)$.

Uwagi: Przykładowy algorytm BFS został zaimplementowany jako klasa SimpleBFS. Uruchomienie algorytmu następuje przez wywołanie metody run.

Listing 5.1. Algorytm BFS z modułu bfs.

```
#!/usr/bin/python

from Queue import Queue

class SimpleBFS:
    """Breadth-First Search."""

    def __init__(self, graph):
        """The algorithm initialization."""
        self.graph = graph
        self.parent = dict()
        self.dag = self.graph.__class__(self.graph.v(), directed=True)

    def run(self, source=None, pre_action=None, post_action=None):
        """Executable pseudocode."""
        if source is not None:
            self._visit(source, pre_action, post_action)
        else:
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self._visit(node, pre_action, post_action)

    def _visit(self, node, pre_action=None, post_action=None):
        """Explore the connected component."""
        Q = Queue()
        self.parent[node] = None    # before Q.put
        Q.put(node)
        if pre_action:    # when Q.put
            pre_action(node)
        while not Q.empty():
```

```
            source = Q.get()
            for edge in self.graph.iteroutedges(source):
                if edge.target not in self.parent:
                    self.parent[edge.target] = source    # before Q.put
                    self.dag.add_edge(edge)
                    Q.put(edge.target)
                    if pre_action:    # when Q.put
                        pre_action(edge.target)
            if post_action:
                post_action(source)
```

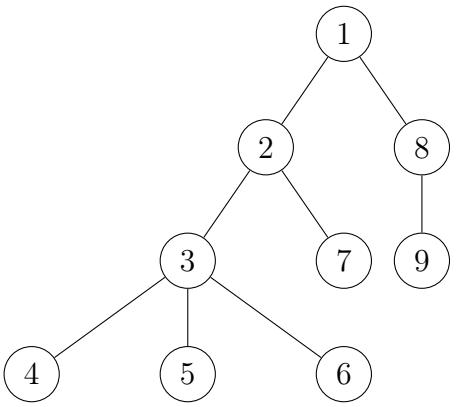
5.2. Przeszukiwanie w głąb (DFS)

Algorytm przeszukiwania w głąb polega na badaniu wszystkich krawędzi wychodzących z danego wierzchołka. Po odwiedzeniu wszystkich krawędzi, algorytm powraca do wierzchołka, z którego dany wierzchołek został odwiedzony.

Dane wejściowe: Dowolny graf, opcjonalnie wierzchołek początkowy.

Problem: Przeszukiwanie grafu wgłąb.

Opis algorytmu: Przeszukiwanie rozpoczynamy od dowolnie wybranego wierzchołka i oznaczamy go jako odwiedzonego. Następnie przechodzimy do pierwszego z jego nieodwiedzonych sąsiadów i ponawiamy procedurę. Jeśli odwiedzany wierzchołek nie ma już nieodwiedzonych sąsiadów to algorytm powraca do wierzchołka rodzica. Podobnie jak algorytm BFS, DFS również wyznacza drzewo rozpinające grafu. Wierzchołki są jednak na nim ułożone inaczej, co wynika z innej kolejności odwiedzania.



Rysunek 5.2. Kolejność odwiedzania wierzchołków w algorytmie DFS.

Złożoność: Złożoność czasowa algorytmu jest uzależniona od implementacji grafu. Jeśli graf reprezentowany jest jako macierz sąsiedztwa to złożoność

czasowa wynosi $O(V^2)$. Przy reprezentacji grafu poprzez listy sąsiedztwa złożoność czasowa wynosi $O(V + E)$.

Uwagi: Przykładowy algorytm został zaimplementowany jako klasa SimpleDFS. Uruchomienie algorytmu następuje przez wywołanie metody run.

Listing 5.2. Algorytm DFS z modułu dfs.

```
#!/usr/bin/python

class SimpleDFS:
    """Depth-First Search with a recursion."""

    def __init__(self, graph):
        """The algorithm initialization."""
        self.graph = graph
        self.parent = dict()
        self.dag = self.graph.__class__(self.graph.v(), directed=True)
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None, pre_action=None, post_action=None):
        """Executable pseudocode."""
        if source is not None:
            self.parent[source] = None    # before _visit
            self._visit(source, pre_action, post_action)
        else:
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self.parent[node] = None    # before _visit
                    self._visit(node, pre_action, post_action)

    def _visit(self, node, pre_action=None, post_action=None):
        """Explore recursively the connected component."""
        if pre_action:
            pre_action(node)
        for edge in self.graph.iteroutedges(node):
            if edge.target not in self.parent:
                self.parent[edge.target] = node    # before _visit
                self.dag.add_edge(edge)
                self._visit(edge.target, pre_action, post_action)
        if post_action:
            post_action(node)
```

6. Spójność grafów

W praktycznych zastosowaniach teorii grafów duże znaczenie ma badanie różnych aspektów spójności grafów. Przykładowo w sieci komputerowej czy energetycznej ważne jest zlokalizowanie węzłów, których awaria spowoduje brak połączenia pomiędzy różnymi częściami sieci (punkty artykulacji). Można również szukać połączeń, których przerwanie spowoduje przerwanie połączenia między częściami sieci. W tym rozdziale przedstawimy podstawowe algorytmy związane ze spójnością grafów. Warto zauważyć jak szerokie zastosowanie ma algorytm DFS.

6.1. Wyznaczanie składowych spójnych

Składowe spójne wyznacza się dla grafu nieskierowanego. Składowa spójna jest to maksymalny spójny podgraf indukowany. Algorytmy wykorzystujące DFS lub BFS do wyznaczania liczby składowych spójnych już były publikowane [20], dlatego pokażemy tylko prostą funkcję testującą spójność na bazie DFS. Jeżeli startując z pewnego wierzchołka grafu nie można dotrzeć do wszystkich innych wierzchołków, to graf nie jest spójny.

Listing 6.1. Funkcja testująca spójność grafu nieskierowanego.

```
def is_connected(graph):
    """Test if the undirected graph is connected."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    algorithm = SimpleDFS(graph)
    order = list()
    source = graph.iternodes().next()
    algorithm.run(source, lambda node: order.append(node))
    return len(order) == graph.v()
```

6.2. Wyznaczanie silnie spójnych składowych

Silnie spójne składowe wyznacza się dla grafu skierowanego. Algorytm opiera się na podwójnym zastosowaniu DFS [8]. Implementacja algorytmu w języku Python jest opublikowana [20], jest to klasa StronglyConnectedComponents z modułu connected.

W literaturze jest też znany algorytm Tarjana z roku 1972.

6.3. Spójność krawędziowa

Zbiorem rozspajającym grafu spójnego G nazywamy zbiór krawędzi, których usunięcie spowoduje, że graf G przestanie być spójny [5]. *Rozcięcie* jest to zbiór rozspajający, którego żaden podzbiór właściwy nie jest już zbiorem rozspajającym. Jeżeli rozcięcie składa się z jednej krawędzi, to tę krawędź nazywamy *mostem* (ang. *bridge*, *cut-edge*). Równoważnie można powiedzieć, że krawędź jest mostem wtedy i tylko wtedy, gdy nie jest zawarta w żadnym cyklu.

Spójnością krawędziową $\lambda(G)$ grafu spójnego G nazywamy liczbę krawędzi należących do najmniej licznego rozcięcia. Mówimy, że graf jest *k-spójny krawędziowo* (ang. *k-edge-connected*), jeżeli $\lambda(G) \geqslant k$.

6.3.1. Algorytm trywialny znajdowania mostów

Opiszemy naiwny sposób znajdowania mostów w grafie nieskierowanym. Najpierw znajdujemy liczbę spójnych składowych grafu. Następnie dla każdej krawędzi powtarzamy następujące czynności: usunięcie krawędzi, sprawdzenie liczby spójnych składowych, przywrócenie krawędzi. Jeżeli po tymczasowym usunięciu krawędzi liczba składowych spójnych powiększa się, to dana krawędź jest mostem. Złożoność czasowa wynosi $O(E(V + E))$, jeżeli liczbę spójnych składowych wyznaczamy przy pomocy BFS lub DFS.

Listing 6.2. Moduł cutedgetrivial.

```
#!/usr/bin/python
#
# Kod bazuje na opisie ze strony
# J. Walaszek, Algorytmy, struktury danych.
# Znajdowanie mostow w grafie,
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0130a.php

from main.algorithms.dfs import SimpleDFS

class TrivialCutEdge:
    """Trivial bridge-finding algorithm."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("graph is directed")
        self.graph = graph
        self.bridges = list()    # lub set()

    def run(self, source=None):
        """Executable pseudocode."""
        old_ncc = self._find_ncc()
        for edge in self.graph.iteredges():
            self.graph.del_edge(edge)
            new_ncc = self._find_ncc()
            self.graph.add_edge(edge)
            if new_ncc > old_ncc:
                self.bridges.append(edge)

    def _find_ncc(self):
```

```
        """Return the number of connected components."""
        # Tutaj nie wiemy i nie musimy wiedziec do ktorej
        # skladowej spojnej nalezy node.
        visited = set()
        ncc = 0
        algorithm = SimpleDFS(self.graph)
        for source in self.graph.iternodes():
            if source not in visited:
                algorithm.run(source, pre_action=lambda node:
                    visited.add(node))
                ncc = ncc + 1
        return ncc
```

6.3.2. Algorytm Tarjana znajdowania mostów

Algorytm Tarjana znajdowania mostów wykorzystuje przejście DFS przez graf nieskierowany [22]. Korzysta się z dwóch obserwacji: (1) most nie może być częścią cyklu, (2) most musi należeć do drzewa rozpinającego [22].Algorytm dzieli krawędzie grafu na krawędzie drzewa DFS, oraz krawędzie wtórne (ang. *back edges*). Ponadto każdemu wierzchołkowi nadawany jest numer DFS w kolejności *preorder*. Wreszcie dla każdego wierzchołka v wyznaczany jest parametr *low*. Jest to najmniejsza liczba: (1) z numeru DFS danego wierzchołka v, (2) z parametrów *low* wszystkich synów wierzchołka v w drzewie DFS, (3) z numerów DFS wierzchołków połączonych z v za pomocą krawędzi wtórnych.

Dane wejściowe: Dowolny graf nieskierowany, opcjonalnie wierzchołek początkowy w przypadku badania tylko jednej składowej spójnej grafu.

Problem: Wyznaczanie mostów w grafie.

Opis algorytmu: Algorytm rozpoczynamy od dowolnego wierzchołka s, któremu nadajemy numer DFS. Przetwarzamy wszystkich nieodwiedzonych sąsiadów wierzchołka s, czyli sąsiadów bez numeru DFS. Na końcu przetwarzamy wierzchołek s. Jeżeli napotkany wierzchołek v ma numer DFS równy parametrowi *low* i wierzchołek ten posiada rodzica w drzewie DFS, to krawędź od tego ojca do v jest mostem.

Złożoność: Złożoność czasowa algorytmu jest liniowa $O(V + E)$, ponieważ wykonywane jest pojedyńcze przejście DFS. Złożoność pamięciowa algorytmu jest rzędu $O(V)$, ponieważ liczba wyznaczanych parametrów jest proporcjonalna do liczby wierzchołków, a liczba mostów również nie może być większa niż liczba wierzchołków, bo każdy most należy do drzewa rozpinającego grafu.

Uwagi: W oryginalnym artykule Tarjana [22] dla każdego wierzchołka v rozważane są trzy funkcje $L(v)$ (odpowiednik *low*), $H(v)$, $ND(v)$, ale istota algorytmu pozostaje taka sama.

Listing 6.3. Moduł cutedgetarjan.

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

```
# Kod bazuje na opisie ze strony
# J. Walaszek, Algorytmy, struktury danych.
# Znajdowanie mostow w grafie,
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0130a.php

class TarjanCutEdge:
    """Tarjan's bridge-finding algorithm."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("graph is directed")
        self.graph = graph
        # Parametr dla wierzcholka wprowadzony przez Tarjana.
        self.low = dict(((node, None) for node in self.graph.iternodes()))
        self.parent = dict(((node, None) for node in self.graph.iternodes()))
        self.time = 0    # time stamp
        self.dd = dict(((node, 0) for node in self.graph.iternodes()))
        self.dag = self.graph.__class__(self.graph.v(), directed=True)
        self.bridges = list()   # lub set()
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:   # badanie jednej skladowej spojnej
            self._visit(source)
        else:
            for node in self.graph.iternodes():
                if self.dd[node] == 0:   # nieodwiedzony
                    self._visit(node)

    def _visit(self, node):
        """Explore recursively the connected component."""
        self.time = self.time + 1
        self.dd[node] = self.time
        self.low[node] = self.time   # wstepne ustawienie
        for edge in self.graph.iteroutedges(node):
            if edge.target == self.parent[node]:   # moze byc None!
                # Ojca wierzcholka node w drzewie DFS pomijamy.
                continue
            if self.dd[edge.target] == 0:   # nieodwiedzony
                self.parent[edge.target] = node
                self.dag.add_edge(edge)
                self._visit(edge.target)
                self.low[node] = min(self.low[node], self.low[edge.target])
            else:   # back edge
                self.low[node] = min(self.low[node], self.dd[edge.target])
        # Wszyscy sasiedzi odwiedzeni. Sprawdzamy warunek mostu.
        if self.parent[node] is not None and self.low[node] == self.dd[node]:
            # Most to jest krawedz prowadzaca do node od jego rodzica.
            # Tu jest klopotliwe wyciaganie calej krawedzi.
            for edge in self.dag.iteroutedges(self.parent[node]):
                if edge.target == node:
                    self.bridges.append(edge)
```

6.4. Spójność wierzchołkowa

Zbiorem rozdzielającym grafu spójnego G nazywamy zbiór wierzchołków, których usunięcie powoduje, że graf G przestaje być spójny. Usuwanie wierzchołka rozumiemy jako jego usuwanie razem z krawędziami z nim incydentnymi. Jeżeli zbiór rozdzielający składa się z jednego wierzchołka, to ten wierzchołek nazywamy *wierzchołkiem rozcinającym* (ang. *cut-node*), *punktem artykulacji* (ang. *articulation point*) lub *wierzchołkiem przegubowym*. Według innej definicji punkt artykulacji to taki wierzchołek, którego usunięcie zwiększa liczbę spójnych składowych grafu.

Blokiem grafu nazywamy jego maksymalny podgraf indukowany niezawierający punktów artykulacji. Dwa różne bloki mają co najwyżej jeden wierzchołek wspólny (punkt artykulacji) oraz są krawędziowo rozłączne [6].

Spójnością wierzchołkową $\kappa(G)$ grafu spójnego G nazywamy liczbę wierzchołków należących do najmniej licznego zbioru rozdzielającego. Mówimy, że graf jest *k-spójny wierzchołkowo* (ang. *k-vertex-connected*), jeżeli $\kappa(G) \geqslant k$ i nie jest to graf pełny K_n. Graf 1-spójny wierzchołkowo nazywamy spójnym, a graf 2-spójny wierzchołkowo nazywamy *dwuspójnym* (ang. *biconnected*). Grafy dwuspójne nie zawierają punktów artykulacji.

Twierdzenie: Jeżeli G jest dowolnym grafem spójnym, to $\kappa(G) \leqslant \lambda(G)$ [5].

6.4.1. Algorytm trywialny znajdowania punktów artykulacji

Trywialny sposób znajdowania punktów artykulacji polega na znalezieniu pierwotnej liczby składowych spójnych grafu, a następnie na powtarzaniu następujących czynności dla każdego wierzchołka v: usunięcie wierzchołka v razem z krawędziami incydentnymi, sprawdzenie liczby spójnych składowych, przywrócenie wierzchołka v razem usuniętymi wcześniej krawędziami. Jeżeli po tymczasowym usunięciu wierzchołka i krawędzi liczba składowych spójnych zwiększa się, to wierzchołek jest punktem artykulacji. Złożoność czasową algorytmu szacujemy na $O(V(V+E))$, jeżeli liczbę składowych spójnych wyznaczamy za pomocą BFS lub DFS. Czas $O(E)$ zajmuje usuwanie i przywracanie krawędzi incydentnych do usuwanych wierzchołków.

Listing 6.4. Moduł cutnodetrivial.

```
#!/usr/bin/python
#
# Kod bazuje na opisie ze strony
# J. Walaszek, Algorytmy, struktury danych.
# Znajdowanie punktow artykulacji w grafie,
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0130b.php

from dfs import SimpleDFS

class TrivialCutNode:
    """Trivial algorithm for finding cut nodes (articulation points)."""

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

```
            raise ValueError("graph is directed")
        self.graph = graph
        self.cut_nodes = list()    # lub set()

    def run(self, source=None):
        """Executable pseudocode."""
        old_ncc = self._find_ncc()
        for source in self.graph.iternodes():
            removed = list(self.graph.iteroutedges(source))
            for edge in removed:
                self.graph.del_edge(edge)
            new_ncc = self._find_ncc()
            for edge in removed:
                self.graph.add_edge(edge)
            if new_ncc > old_ncc + 1:    # source nie usuwalem
                self.cut_nodes.append(source)

    def _find_ncc(self):
        """Return the number of connected components."""
        # Tutaj nie wiemy i nie musimy wiedziec do ktorej
        # skladowej spojnej nalezy node.
        visited = set()
        ncc = 0
        algorithm = SimpleDFS(self.graph)
        for source in self.graph.iternodes():
            if source not in visited:
                algorithm.run(source, pre_action=lambda node:
                    visited.add(node))
                ncc = ncc + 1
        return ncc
```

6.4.2. Algorytm Tarjana znajdowania punktów artykulacji

Algorytm Tarjana znajdowania punktów artykulacji wykorzystuje przejście DFS, podobnie jak w przypadku wyznaczania mostów. Wykorzystuje się dwie własności punktów artykulacji [23].

Po pierwsze, korzeń drzewa DFS jest punktem artykulacji wtedy i tylko wtedy, gdy ma co najmniej dwóch synów. Jeżeli ma jednego syna, to albo korzeń jest wierzchołkiem wiszącym, albo dochodzą do niego jeszcze krawędzie wtórne, czyli korzeń należy do pewnego cyklu i nie jest punktem artykulacji. Sprawdzenie tej własności jest proste.

Po drugie, wierzchołek v nie będący korzeniem drzewa DFS jest punktem artykulacji, jeżeli przynajmniej dla jednego z jego synów nie istnieje krawędź wtórna, która łączy potomka wierzchołka v z przodkiem wierzchołka v. Inaczej mówiąc, do syna wierzchołka v można dojść jedynie krawędzią łączącą go z wierzchołkiem v. Sprawdzenie tej własności opiera się na parametrze *low*, który jest obliczany dla każdego wierzchołka przy przejściu DFS. Parametr ten to w istocie najmniejszy numer DFS wierzchołka, do którego istnieje ścieżka w dół drzewa DFS. Jeżeli parametr *low* jednego z synów wierzchołka v będzie większy lub równy numerowi DFS wierzchołka v, to będzie to oznaczało, że ścieżka zawierająca wierzchołek v i tego syna nie

posiada krawędzi wtórnej do przodka wierzchołka v. A wtedy wierzchołek v jest punktem artykulacji.

Dane wejściowe: Dowolny graf nieskierowany, opcjonalnie wierzchołek początkowy w przypadku badania tylko jednej składowej spójnej grafu.

Problem: Wyznaczanie punktów artykulacji w grafie.

Opis algorytmu: Algorytm rozpoczynamy od dowolnego wierzchołka s, który jest przetwarzany specjalnie jako korzeń drzewa DFS (należy sprawdzić liczbę synów). Dalsze wywołania rekurencyjne DFS dla potomków korzenia są podobne do przypadku wyznaczania mostów. Inny jest jednak warunek na punkt artykulacji.

Złożoność: Złożoność czasowa algorytmu jest liniowa $O(V + E)$, ponieważ wykonywane jest pojedyńcze przejście DFS. Złożoność pamięciowa algorytmu jest rzędu $O(V)$, ponieważ liczba wyznaczanych parametrów jest proporcjonalna do liczby wierzchołków, a liczba punktów artykulacji również nie może być większa niż liczba wierzchołków.

Listing 6.5. Moduł cutnodetarjan.

```
#!/usr/bin/python
#
# Kod bazuje na opisie ze strony
# J. Walaszek, Algorytmy, struktury danych.
# Znajdowanie punktow artykulacji w grafie,
# http://edu.i-lo.tarnow.pl/inf/alg/001_search/0130b.php

class TarjanCutNode:
    """Tarjan's algorithm for finding cut nodes."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("graph is directed")
        self.graph = graph
        # Parametr dla wierzcholka wprowadzony przez Tarjana.
        self.low = dict(((node, None) for node in self.graph.iternodes()))
        self.parent = dict(((node, None) for node in self.graph.iternodes()))
        self.time = 0    # time stamp
        self.dd = dict(((node, 0) for node in self.graph.iternodes()))
        self.dag = self.graph.__class__(self.graph.v(), directed=True)
        self.cut_nodes = list()   # lub set()
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:   # badanie jednej skladowej spojnej
            self._visit_root(source)
        else:
            for node in self.graph.iternodes():
```

```
            if self.dd[node] == 0:    # nieodwiedzony
                self._visit_root(node)

    def _visit_root(self, node):
        """Explore recursively the connected component from root."""
        # Korzen przetwarzamy specjalnie.
        self.time = self.time + 1
        self.dd[node] = self.time
        self.low[node] = self.time    # chyba niepotrzebne
        n_sons = 0    # licznik synow dla korzenia
        for edge in self.graph.iteroutedges(node):
            if self.dd[edge.target] == 0:    # nieodwiedzony
                n_sons = n_sons + 1
                self.parent[edge.target] = node
                self.dag.add_edge(edge)
                self._visit(edge.target)
                # Nie ma uaktualnienia low[node], bo nie trzeba.
        # Czy korzen jest punktem artykulacji?
        if n_sons > 1:
            self.cut_nodes.append(node)

    def _visit(self, node):
        """Explore recursively the connected component."""
        self.time = self.time + 1
        self.dd[node] = self.time
        self.low[node] = self.time    # wstepne ustawienie
        is_cut_node = False
        for edge in self.graph.iteroutedges(node):
            if edge.target == self.parent[node]:    # moze byc None!
                # Ojca wierzcholka node w drzewie DFS pomijamy.
                continue
            if self.dd[edge.target] == 0:    # nieodwiedzony
                self.parent[edge.target] = node
                self.dag.add_edge(edge)
                self._visit(edge.target)
                self.low[node] = min(self.low[node], self.low[edge.target])
                # Test na punkt artykulacji.
                if self.low[edge.target] >= self.dd[node]:
                    is_cut_node = True
            else:    # back edge
                self.low[node] = min(self.low[node], self.dd[edge.target])
        # Wszyscy sasiedzi odwiedzeni. Sprawdzamy wynik testu.
        if is_cut_node:
            self.cut_nodes.append(node)
```

7. Planarność grafów

Testowanie planarności grafów (ang. *planarity testing*) jest uważane w literaturze za trudne zagadnienie. W podstawowych podręcznikach do teorii grafów można znaleźć główne twierdzenia matematyczne, ale brak jest pseudokodów algorytmów testujących planarność o złożoności liniowej. W książce Deo [7] można znaleźć jedynie zarys algorytmu dekompozycji obwodowo-ścieżkowej Hopcrofta i Tarjana. Artykuły naukowe również często ograniczają się jedynie do teoretycznego opisu algorytmu, bez prezentowania pseudokodu.

Bezpośrednie poszukiwanie grafów Kuratowskiego lub zastosowanie twierdzenia Wagner zaowocowałyby algorytmami o złożoności $O(2^n)$ i $O(n!)$ [1], dlatego skupimy się innych kombinatorycznych sposobach sprawdzania planarności grafów. W literaturze można wyróżnić trzy podejścia, które zaowocowały algorytmami o złożoności liniowej [12]:

— Podejście z dodawaniem wierzchołków (ang. *vertex-addition approach*), którego pionierami byli Lempel, Even i Cederbaum (1967).
— Podejście z dodawaniem ścieżek (ang. *path-addition approach*), rozpoczęte przez Hopcrofta i Tarjana (1974).
— Podejście z cyklami lewymi i prawymi (ang. *left-right approach*), którego autorami byli Fraysseix i Rosenstiehl (1982).

Podejście z dodawaniem wierzchołków i podejście z dodawaniem ścieżek pierwotnie zaprezentowane przez ich twórców okazały się tak skomplikowane, że wymagały dodatkowych publikacji wyjaśniających sposób konstruowania grafu topologicznego dla grafu planarnego [1]. W 1996, ponad 20 lat od opublikowania pierwotnego algorytmu, Mehlorn i Mutzel [24] przedstawili pracę wyjaśniającą algorytm Hopcrofta i Tarjana (1974). W 1985 zaś, N. Chiba, T. Nishizeki, S. Abe, i T. Ozawa [25] zaprezentowali publikację opisującą konstrukcję grafu topologicznego z użyciem podejścia z dodawniam ścieżek Lempela, Evena i Cederbauma.

Dla danego grafu G można podać cztery problemy związane z planarnością [12]:

1. Określenie, czy graf G jest planarny.
2. Jeżeli graf G jest planarny, to określenie jego *grafu topolgicznego* (ang. *planar embedding*).
3. Jeżeli graf G nie jest planarny, to znalezienie podgrafu Kuratowskiego.
4. Dla danego *grafu topologicznego* grafu G znalezienie jego *grafu płaskiego*, czyli nadanie wierzchołkom określonych współrzędnych na płaszczyźnie iokreślenie, czy krawędzie to odcinki czy łuki.

W niniejszej pracy skupimy się na określeniu czy graf jest planarny i przejściu z poziomu grafu abstrakcyjnego na poziom grafu topologicznego.

7.1. Test planarności lewy-prawy

Test planarności lewy-prawy (ang. *left-right planarity test*) bazuje na właściwościach przeszukiwania grafów w głąb [26]. Kryterium podali de Fraysseix i Rosenstiehl [2], [3], a następnie razem z Mendezem użyli do stworzenia algorytmu testującego planarność grafu w czasie liniowym [27], [4]. Algorytm ten jest aktualnie uznawany za najszybszy, spośród wszystkch istniejących implementacji testujących planarność grafów [4].

Głównym źródłem problemów w testowaniu planarności grafów są cykle. Na płaszczyźnie cykle są reprezentowane przez krzywe zamknięte, które dzielą obszar płaszczyzny na dwie części. Należy określić które pozostałe składowe grafu powinny znaleźć się wewnątrz, a które na zewnątrz tych cykli [12].

Cykle proste można przedstawić planarnie na płaszczyźnie jedynie na dwa sposoby, czyli zgodnie z ruchem wskazówek zegara, lub przeciwnie do ruchu wskazówek zegara. Wybranie orientacji jednego cyklu może jednak nieść ze sobą konieczność nałożenia pewnych ograniczeń na orientacje innych cykli, które posiadają wspólne krawędzie. Badanie planarności grafu można w takim przypadku ograniczyć do stwierdzenia, czy istnieje spójna orientacja wszystkich cykli w grafie [12]. Okazuje się, że wystarczy rozważyć tylko mały zbiór cykli, który reprezentuje całą strukturę cykli w grafie. Ten mały zbiór reprezentantywnych cykli określa się za pomocą DFS.

7.1.1. Właściwości przeszukiwania grafu w głąb

Przeszukanie nieskierowanego grafu $G = (V, E)$ w głąb wyznacza na nim *orientację DFS*, czyli graf skierowany $\overrightarrow{G} = (V, \overrightarrow{E})$, w którym każda nieskierowana krawędź zostaje zorientowana zgodnie z kierunkiem przechodzenia. Dodatkowo wygenerowany zostaje podział zbioru krawędzi E na dwa rozłączne podzbiory T i B , gdzie T to krawędzie tworzące drzewo rozpinające grafu G, a B to pozostałe krawędzie. Krawędzie w T nazywamy *krawędziami drzewa* (ang. *tree edge*), a krawędzie w B *krawędziami wstecznymi* (ang. *back edge*). Przeszukiwanie w głąb określa również wysokość wierzchołków. Pierwszy wierzchołek (korzeń drzewa rozpinającego) ma wysokość równą zero. Wysokość kolejnych wierzchołów jest określana przez ich odległość od korzenia w drzewie rozpinającym. Wierzchołek v jest *niższy* od wierzchołka w, jeśli wysokość v jest mniejsza od wysokości w. Dla każdego wierzchołka $v \in V$ definiujemy również zbiór krawędzi z niego wychodzących $E^+(v) = \{(v, w) \in E : w \in V\}$ [12].

Charakterystyczną cechą grafu powstałego w wyniku orientacji DFS jest fakt, że wierzchołek docelowy w każdej krawędzi wstecznej jest przodkiem w drzewie rozpinającym dla jej wierzchołka źródłowego v. W związku z tym, dla każdej krawędzi wstecznej, zbiór $T + (v, w)$ zawiera cykl. Cykle te nazywane są *cyklami podstawowymi* (ang. *fundamental cycle*) [4].

Najniższy punkt krawędzi edge zwraca funkcja lowpt, która każdej krawędzi drzewa DFS przyporządkowuje wysokość najniższego wierzchołka, jaki można osiągnąć podążając dowolną scieżką zaczynającą się w edge i zawierającą przynajmniej jedną krawędź wsteczną [4].

7.1.2. Kryterium planarności

Podział LR (ang. *LR Partition*): Niech $G = (V, T \cup B)$ będzie grafem zorientowanym przez DFS. Podział $B = L \cup R$ krawędzi wstecznych na dwie klasy (lewa i prawa) jest nazywany podziałem lewy-prawy (w skrócie *podział LR*), jeśli dla każdego *rozwidlenia* składającego się z $(u, v) \in T$ i $e_1, e_2 \in E^+(v)$ zachodzą równocześnie warunki:

1. wszystkie krawędzie powrotne e_1, kończące się wyżej niż $lowpt(e_2)$, należą do jednej klasy,
2. wszystkie krawędzie powrotne e_2, kończące się wyżej niż $lowpt(e_1)$, należą do drugiej klasy.

Kryterium planarności lewe-prawe: Graf jest planarny wtedy i tylko wtedy, gdy istnieje dla niego podział LR [12].

7.1.3. Implementacja algorytmu

W naszej implementacji algorytm reprezentowany jest jako klasa LeftRightPlanarity. Obiekt algorytmu inicjowany jest obiektem klasy Graph. Podczas inicjacji określane są wartości początkowe zmiennych używanych w fazie właściwej pracy algorytmu.

Listing 7.1. Klasa LeftRightPlanarity.

```
class LeftRightPlanarity:
    """The left-right planarity test."""

    def __init__(self, graph):
        """The algorithm initialization."""
        self.height = dict()
        self.lowpt = dict()
        self.lowpt2 = dict()
        self.nesting_depth = dict()

        self.ref = dict()
        self.side = dict()
        self.stack = list()
        self.stack_bottom = dict()
        self.lowpt_edge = dict()
        self.roots = list()
        self.left_ref = dict()
        self.right_ref = dict()
        self.parent_edge = dict()
        self.first_edge = dict()

        self.graph = graph

        for node in graph.iternodes():
            self.height[node] = float("inf")
```

Algorytm testujący planarność grafu podzielony jest na trzy fazy. Orientację, testowanie i osadzanie. W każdej fazie wykonywane jest przeszukwanie grafu w głąb i wykowanie dodatkowych instrukcji. Za przebieg algorytmu

odpowiada metoda run, która tworzy graf topologiczny, jeśli graf abstrakcyjny jest planarny, lub wyzwala wyjątek GraphNotPlanarError w przeciwnym wypadku.

Listing 7.2. Algorytm testujący planarność lewy-prawy.

```
def run(self):
    """
    Perform left right planarity test.
    Raise GraphNotPlanarError if the graph is not planar.
    """
    # orientation
    for node in self.graph.iternodes():
        if self.height[node] == float("inf"):
            self.height[node] = 0
            self.roots.append(node)
            self._dfs1(node)

    # init sides
    for edge in self.lowpt:
        self.side[edge] = 1

    # testing
    for node in self.roots:
        self._dfs2(node)

    # embedding
    for edge in self.lowpt:
        self.nesting_depth[edge] *= self._sign(edge)

    # init topological graph
    for node in self.graph.iternodes():
        self._init_topological_graph(node)
    for node in self.roots:
        self._dfs3(node)

def _init_topological_graph(self, node):
    """
    Save outgoing edges from node ordered by nesting_depth
    to edge_next and edge_prev dicts representing
    topological graph.
    """
    sorted_edges = self._sort_by_nesting_order(node)
    if sorted_edges:
        self.first_edge[node] = sorted_edges[0]
        for index, edge in enumerate(sorted_edges):
            next_index = (index+1) % len(sorted_edges)  # next edge index
            prev_index = (index-1) % len(sorted_edges)  # prev edge index
            self.graph.edge_next[edge] = sorted_edges[next_index]
            self.graph.edge_prev[edge] = sorted_edges[prev_index]
            self.graph.edge_next[~edge] = ~sorted_edges[next_index]
            self.graph.edge_prev[~edge] = ~sorted_edges[prev_index]

def _sign(self, edge):
    if self.ref.get(edge):
        self.side[edge] *= self._sign(self.ref[edge])
        self.ref[edge] = None
    return self.side[edge]
```

Nazwa	Typ	Opis
height	słownik {edge: **int**}	odległości wierzchołków od korzenia
lowpt	słownik {edge: **int**}	wysokości najniższych punktów krawędzi
lowpt2	słownik {edge: **int**}	wysokości drugich w kolejności, najniższych punktów krawędzi (tylko krawędzie drzewa)
nesting_depth	słownik {edge: **int**}	reprezentacja częściowego porządku zawierania się krawędzi
side	słownik {edge: -1 lub $+1$}	strona krawędzi
stack	lista (stos)	stos skonflikowanych par
stack_bottom	słownik {edge: Pair}	pierwszy element stosu podczas przechodzenia krawędzi edge
lowpt_edge	słownik {edge: edge}	kolejna krawędź wsteczna
roots	lista	korzenie drzew spójnych składowych
parent_edge	słownik {node: edge}	krawędź prowadząca od rodzica do node
first_edge	słownik {node: edge}	wskaźnik na pierwszą krawędź w liście przystawania node

Tabela 7.1: Zmienne używane w algorytmie lewy-prawy.

7.1.4. Orientacja

Algorytm rozpoczyna się od przeszukania grafu w głąb i wyznaczenia jego drzewa rozpinającego. Dla każdej spójnej składowej, na liście roots zapamiętany zostaje korzeń drzewa rozpinającego tej składowej. Odległości poszczególnych wierzchołków od korzenia zapamiętywane są w słowniku height.

Podczas przeszukania grafu obliczane są najniższe punkty wszystkich krawędzi, które przechowywane są następnie w słowniku lowpoint. Wyznaczony zostaje również częściowy porządek zawierania się krawędzi. Reprezentowany jest on przez słownik nesting_depth.

Listing 7.3. DFS - faza orientacji.

```
def _dfs1(self, node):
    parent_edge = self.parent_edge.get(node, None)

    for edge in self.graph.iteroutedges(node):
        if edge not in self.lowpt and ~edge not in self.lowpt:
            self.lowpt[edge] = self.height[node]
            self.lowpt2[edge] = self.height[node]

            if self.height[edge.target] == float("inf"):
                # tree edge
                self.parent_edge[edge.target] = edge
```

```
                self.height[edge.target] = self.height[node] + 1
                self._dfs1(edge.target)
            else:
                # back edge
                self.lowpt[edge] = self.height[edge.target]

            # determine nesting depth
            self.nesting_depth[edge] = 2 * self.lowpt[edge]
            # if chordal
            if self.lowpt2[edge] < self.height[edge.source]:
                self.nesting_depth[edge] += 1

            # update lowpoints of parent edge e
            if parent_edge:
                if self.lowpt[edge] < self.lowpt[parent_edge]:
                    self.lowpt2[parent_edge] = \
                        min(self.lowpt[parent_edge], self.lowpt2[edge])
                    self.lowpt[parent_edge] = self.lowpt[edge]
                elif self.lowpt[edge] > self.lowpt[parent_edge]:
                    self.lowpt2[parent_edge] = \
                        min(self.lowpt2[parent_edge], self.lowpt[edge])
                else:
                    self.lowpt2[parent_edge] = \
                        min(self.lowpt2[parent_edge], self.lowpt2[edge])
```

7.1.5. Testowanie

Druga faza algorytmu odpowiada za ustalenie podziału LR grafu, jeśli taki istnieje. Podział zapisywany jest jako słownik side, w którym krawędziom przyporządkowane zostają wartości $+1$ i -1, odpowiadające odpowiednio *prawej* i *lewej* stronie.

Do przechowywania ograniczeń nakładanych na krawędzie używane są struktury danych Interval i Pair. W strukturze Pair znajdują się interwały krawędzi, których ograniczenia są ze sobą sprzeczne.

Drugie przejście grafu w głąb tworzy podział LR poprzez łączenie ze sobą skonflikowanych par. Sprzeczne pary przetrzymywane są na stosie S i reprezentują wszystkie ograniczenia związane z krawędzią drzewa, która została już odwiedzona. Krawędzie przemierzane są w takim samym kierunku jak podczas pierwszego przeszukania. Różna jest jednak kolejność ich przeglądania, ponieważ krawędzie zostają uporządkowane rosnąco względem ich wartości w nesting_depth.

Sortowanie krawędzi odbywa się w czasie liniowym z użyciem sortowania kubełkowego i odpowiada za nie metoda sort_by_nesting_order.

Listing 7.4. Sortowanie kubełkowe krawędzi wychodzących.

```
def _sort_by_nesting_order(self, node):
    """
    Return all outgoing edges from node ordered by nesting_depth.
    """
    # lets use bucket sort for linear sorting time
    buckets = dict()
    for depth in self.nesting_depth.itervalues():
        buckets[depth] = list()
```

```
    for edge in self.graph.iteroutedges(node):
        if edge not in self.lowpt:    # incoming edge
            continue
        buckets[self.nesting_depth[edge]].append(edge)
    sorted_edges = list()
    sorted_buckets = sorted(buckets)
    for depth in sorted_buckets:
        sorted_edges.extend(buckets[depth])
    return sorted_edges
```

Głównym zadaniem drugiego przejścia DFS jest rekurencyjne określenie ograniczeń krawędzi wychodzących $e_i \in E^+(v)$ z wierzchołka v i zintegrowanieich z ograniczeniami powiązanymi z krawędzią parent_edge[v]. Przed przejściem krawędzi $e_i \in E^+(v)$ w stack_bottom[edge_i] zapamiętana zostaje para znajdująca się na szczycie stosu S. Jeśli e_i jest krawędzią drzewa to wszystkie powiązane z nią ograniczenia są określane rekurencyjnie i dodawane do stosu S. Jeśli e_i jest krawędzią wsteczną to na stos odkładana jest para zawierająca krawędź e_i w konflikcie z samą sobą, gdyż może być ona uwikłana w późniejsze ograniczenia.

Kiedy wszystkie krawędzie wychodzące zostaną odwiedzone, stos S zostaje zredukowany o te krawędzie wsteczne, które są krawędziami wstecznymi dla wszystkich $e_i \in E^+(v)$, ale nie dla parent_edge[v].

Listing 7.5. DFS - faza testowania.

```
def _dfs2(self, node):
    parent_edge = self.parent_edge.get(node)
    sorted_edges = self._sort_by_nesting_order(node)
    for edge_i in sorted_edges:
        if len(self.stack) != 0:
            self.stack_bottom[edge_i] = self.stack[-1]
        else:
            self.stack_bottom[edge_i] = None
        pe = self.parent_edge.get(edge_i.target)
        if pe and edge_i == pe:
            # tree edge
            self._dfs2(edge_i.target)
        else:
            # back edge
            self.lowpt_edge[edge_i] = edge_i
            self.stack.append(Pair(None, Interval(edge_i, edge_i)))

        # integrate new return edges
        if self.lowpt[edge_i] < self.height[node]:
            # edge_i has return edge
            if edge_i == sorted_edges[0]:
                self.lowpt_edge[parent_edge] = \
                    self.lowpt_edge[sorted_edges[0]]
            else:
                self._add_constraints(parent_edge, edge_i)

    if parent_edge:  # node is not root
        source_node = parent_edge.source
        self._trim_back_edges_at_parent(source_node)
        # if has return edge
        if self.lowpt[parent_edge] < self.height[source_node]:
```

```
            try:
                highest_left = self.stack[-1].left.high
            except AttributeError:
                highest_left = None
            try:
                highest_right = self.stack[-1].right.high
            except AttributeError:
                highest_right = None
            if highest_left is not None and (highest_right is None or
                self.lowpt[highest_left] > self.lowpt[highest_right]):
                    self.ref[parent_edge] = highest_left
            else:
                self.ref[parent_edge] = highest_right
```

Za łączenie ograniczeń powiązanych z krawędzią e_i z ograniczeniami $e_1, ..., e_{i-1}$ odpowiedzialna jest metoda _add_constraints(parent_edge, edge_i).

Jeśli w czasie łączenia ograniczeń algorytm natrafi na parę z dwoma niepustymi interwałami to graf nie jest planarny i rzucany jest wyjątek GraphNotPlanarError.

Listing 7.6. Faza testowania - łączenie ograniczeń.

```
def _add_constraints(self, parent_edge, edge_i):
    pair = Pair()
    # merge return edges of edge_i to pair.right
    while True:
        top_pair = self.stack.pop()

        if top_pair.left:
            tmp = top_pair.left
            top_pair.left = top_pair.right
            top_pair.right = tmp
        if top_pair.left:
            raise GraphNotPlanarError()
        else:
            # merge intervals
            if self.lowpt[top_pair.right.low] > self.lowpt[parent_edge]:
                if not pair.right:
                    pair.right = Interval(None, top_pair.right.high)
                else:
                    self.ref[pair.right.low] = top_pair.right.high
                pair.right.low = top_pair.right.low
            else:  # align
                self.ref[top_pair.right.low] = self.lowpt_edge[parent_edge]

        if self.stack[-1] == self.stack_bottom[edge_i]:
            break

    # merge conflicting edges into p.left
    while self._conflicting(self.stack[-1].left, edge_i) \
            or self._conflicting(self.stack[-1].right, edge_i):
        top_pair = self.stack.pop()

        if self._conflicting(top_pair.right, edge_i):
            tmp = top_pair.left
            top_pair.left = top_pair.right
            top_pair.right = tmp
```

```
            if self._conflicting(top_pair.right, edge_i):
                raise GraphNotPlanarError()
            else:
                # merge interval below lowpt(edge_i) into pair.right
                self.ref[pair.right.low] = top_pair.right.high \
                    if top_pair.right else None
                if top_pair.right and top_pair.right.low:
                    pair.right.low = top_pair.right.low
            if not pair.left:
                pair.left = Interval(None, top_pair.left.high)
            else:
                self.ref[pair.left.low] = top_pair.left.high
            pair.left.low = top_pair.left.low

        if pair.left or pair.right:
            self.stack.append(pair)

    def _conflicting(self, interval, edge):
        return interval and self.lowpt[interval.high] > self.lowpt[edge]
```

Metoda trim_back_edges_at_parent(source_node) jest odpowiedzialna za usunięcie z par znajdujących się na stosie tych krawędzi wstecznych, których lowpt jest w wierzchołku źródłowym aktualnie przetwarzanej krawędzi.

Listing 7.7. Faza testowania - usuwanie krawędzi wstecznych.

```
    def _trim_back_edges_at_parent(self, node):
        while len(self.stack) != 0 \
                and self._lowest(self.stack[-1]) == self.height[node]:
            # drop entire conflict pairs
            pair = self.stack.pop()
            if pair.left and pair.left.low is not None:
                self.side[pair.left.low] = -1

        if len(self.stack) != 0:
            # one more conflict pair to consider
            pair = self.stack.pop()
            while pair.left and pair.left.high \
                    and pair.left.high.target == node:
                pair.left.high = self.ref.get(pair.left.high)
            if (pair.left is None or pair.left.high is None) \
                    and (pair.left and pair.left.low is not None):
                self.ref[pair.left.low] = pair.right.low
                self.side[pair.left.low] = -1
                pair.left.low = None
            self.stack.append(pair)

    def _lowest(self, pair):
        if not pair.left:
            return self.lowpt[pair.right.low]
        if not pair.right:
            return self.lowpt[pair.left.low]
        return min(self.lowpt[pair.left.low], self.lowpt[pair.right.low])
```

Wyczerpujący opisy teoretyczny i kombinatoryczny łączenia ograniczeń i usuwania krawędzi wstecznych jest podany w [12].

7.1.6. Osadzanie

W ostatniej fazie algorytmu utworzony zostaje graf topologiczny. Krawędzie wychodzące w każdym z wierzchołków są już uporządkowane według odpowiadających im wartości nesting_depth. Należy jeszcze wstawić w odpowiednie miejsca na listach sąsiedztwa krawędzie wsteczne incydentne z wierzchołkami do których wracają.

Listing 7.8. DFS - faza osadzania.

```
def _dfs3(self, node):
    sorted_edges = self._sort_by_nesting_order(node)
    for edge_i in sorted_edges:
        w = edge_i.target
        pe = self.parent_edge.get(w)
        if pe and edge_i == pe:
            # make edge_i first edge in adjacency list of w
            if w in self.first_edge:
                first_edge = self.first_edge[w]
            else:
                first_edge = edge_i
            self._insert_into_embedding_before(first_edge, edge_i, w)
            self.left_ref[node] = edge_i
            self.right_ref[node] = edge_i
            self._dfs3(w)
        else:
            if self.side[edge_i] == 1:
                self._insert_into_embedding_after(
                    self.right_ref[w], edge_i, w)
                pass
            else:
                self._insert_into_embedding_before(
                    self.left_ref[w], edge_i, w)
                self.left_ref[w] = edge_i

def _insert_into_embedding_before(self, edge, edge_to_insert, node):
    """
    Inserts edge_to_insert before edge
    in topological adjacency list of node
    """
    if node != edge.source:
        edge = ~edge

    if node != edge_to_insert.source:
        edge_to_insert = ~edge_to_insert
    prev = self.graph.edge_prev[edge]

    self.graph.edge_prev[edge] = edge_to_insert
    self.graph.edge_next[edge_to_insert] = edge

    self.graph.edge_next[prev] = edge_to_insert
    self.graph.edge_prev[edge_to_insert] = prev

def _insert_into_embedding_after(self, edge, edge_to_insert, node):
    """
    Inserts edge_to_insert after edge
    in topological adjacency list of node
    """
```

```
        if node != edge.source:
            edge = ~edge

        if node != edge_to_insert.source:
            edge_to_insert = ~edge_to_insert
        next = self.graph.edge_next[edge]

        self.graph.edge_prev[next] = edge_to_insert
        self.graph.edge_next[edge_to_insert] = next

        self.graph.edge_next[edge] = edge_to_insert
        self.graph.edge_prev[edge_to_insert] = edge
```

7.1.7. Przykładowy skrypt wykorzystujący planarność

Poniższy skrypt prezentuje użycie algorytmu lewy-prawy. Najpierw tworzymy graf planarny. Kolejne kroki prezentują wywołanie algorytmu lewy-prawy, wypisanie uporządkowanych krawędzi wychodzących z wierzchołków, oraz wypisanie ścian grafu jako list krawędzi obiegających ściany.

Listing 7.9. Skrypt prezentujący użycie algorytmu lewy-prawy.

```
#!/usr/bin/python
"""
Interactive session showing how to print on screen embedded graph.
Source code first prints all outgoing edges from vertices, and then
prints all faces in planar graph embedding.
"""
from main.algorithms.left_right import LeftRightPlanarity
from main.data_structures.edges import Edge
from main.data_structures.graphs import Graph

# Utworzenie grafu
graph = Graph()
graph.add_node(1)
graph.add_node(2)
graph.add_node(3)
graph.add_node(4)
graph.add_node(5)
graph.add_edge(Edge(1, 2))
graph.add_edge(Edge(1, 3))
graph.add_edge(Edge(1, 4))
graph.add_edge(Edge(1, 5))
graph.add_edge(Edge(2, 3))
graph.add_edge(Edge(2, 4))
graph.add_edge(Edge(2, 5))
graph.add_edge(Edge(3, 4))
graph.add_edge(Edge(3, 5))

# Wykonanie algorytmu
algorithm = LeftRightPlanarity(graph)
algorithm.run()

# Wypisanie krawedzie wychodzacych z wierzcholkow
print "Edges:"
for node in graph.iternodes():
```

```
    print "Node", node, ":",
    first_edge = graph.iteroutedges(node).next()
    edge = first_edge
    print edge,
    edge = graph.edge_next[edge]
    while edge != first_edge:
        print edge,
        edge = graph.edge_next[edge]
    print
print "Faces:"
# Wypisanie krawedzi wychodzacych z wierzcholkow
for face in graph.iterfaces():
    print face
```

W ramach pracy stworzono także generator maksymalnych grafów planarnych, zawarty w klasie PlanarMap. Generator może także tworzyć przypadkowe grafy planarne spójne.

7.1.8. Złożoność czasowa

Algorytm wykonuje trzy przejścia DFS, podczas których dwukrotnie zmieniana jest kolejność krawędzi. Złożoność algorytmu DFS to $O(V + E)$ [8].

Krawędzie sortowane są w czasie liniowym z wykorzystaniem sortowania kubełkowego według wartości w nesting_depth. Maksymalny zakres wartości to $2n$, gdzie n jest liczbą wierzchołków w grafie [12]. Teoretycznie, w związku z powyższym złożoność obliczeniowa algorytmu lewy-prawy wynosi $O(V + E)$, co potwierdzają [12], [3].

Rezultaty testów czasu działania implementacji w Pythonie znajdują się w rozdziale B. Przeprowadzone eksperymenty sugerują, że nasza implementacja ma złożoność wyższą niż liniowa.

8. Podsumowanie

Głównym celem pracy było zaimplementowanie algorytmu badającego planarność grafów w języku Python. Algorytm lewy-prawy, uznawany obecnie za najszybszy algorytm sprawdzający planarność grafu, został opisany w rozdziale 7.1. Jego implementacja znajduje się w klasie LeftRightPlanarity. W związku z dużą zależnością pomiędzy planarnością a przeszukiwaniem grafu w głąb, w pracy został również dokładnie omówiony algorytm DFS, oraz inne algorytmy badające grafy w oparciu o DFS. Są to algorytmy znajdowania mostów i punktów artykulacji, oraz testowania spójności grafu znajdujące się w rozdziale 6. W ramach pracy stworzona została nowa implementacja klasy Graph, oparta o słownik słowników i umożliwiająca przechowywanie obiektów krawędzi grafu. Klasa jest zgodna z interfejsem z [20]. Dodatkowo zostały w niej zaimplementowane metody, które dla grafów planarnych zwracają ilość ścian grafu, oraz iterator ścian.

Kod źródłowy w języku Python został napisany zgodnie ze standardem PEP8. Przejrzystość składni Pythona, oraz sugestywne nazwy zmiennych i funkcji ułatwiają zrozumienie algorytmów. Kod może być uruchomiony na dowolnym systemie operacyjnym wyposażonym w interpeter języka Python. Poprawność kodu źródłowego została przetestowana za pomocą testów jednostkowych. Algorytmy został również przetestowany pod kątem złożoności obliczeniowej za pomocą testów wydajnościowych. Rezultaty tych testów znajdują się w dodatku B.

W związku z brakiem książek w języku polskim poruszających dogłębnie temat planarności grafów od strony algorytmicznej, implementacja algorytmu lewy-prawy powstała na bazie artykułów naukowych. W głównej mierze były to prace de Fraysseix i Rosenstiehl [2], [3] oraz Brandesa [12]. Warto wspomnieć, że de Fraysseix i Mendez stworzyli w języku C++ bibliotekę *PIGALE*, w której również znajduje się implementacja algorytmu lewy-prawy. Zważywszy na użytą technologię, jest to z pewnością wydajniejsza implementacja. Jednak ze względu na składnię języka, oraz użyty przez autorów interfejs i struktury danych, zrozumienie algorytmu na podstawie kodu źródłowego jest zdecydowanie trudniejsze.

Rezultatem działania algorytmu lewy-prawy dla grafów planarnych jest *graf topologiczny*, czyli struktura definiującą ściany grafu, oraz kolejność krawędzi incydentnych w wierzchołkach. Może to stanowić pukt wyjścia i podstawę do dalszych badań i rozwoju algorytmu, który dla grafów nieplanarnych wyznacza podgraf Kuratowskiego lub minimalnu zbiór krawędzi, których usunięcie daje graf planarny. Graf topologiczny może być także bazą do wyznaczania grafów geometrycznych, czyli algorytmów rysujących grafy na płaszczyźnie.

A. Kod źródłowy dla krawędzi i grafów

Algorytmy grafowe prezentowane w niniejszej pracy bazują na dwóch podstawowych klasach: klasie Edge dla krawędzi skierowanych i klasie Graph dla grafów prostych. W tym dodatku przedstawimy kody źródłowe tych klas.

A.1. Klasa Edge

Klasa Edge pochodzi z implementacji grafów rozwijanej w Instytucie Fizyki Uniwersytetu Jagiellońskiego w Krakowie [20].

Listing A.1. Klasa Edge z modułu edges.

```
class Edge:
    """The class defining a directed edge."""

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

    def __repr__(self):
        """Compute the string representation of the edge."""
        if self.weight == 1:
            return "%s(%s, %s)" % (
                self.__class__.__name__,
                repr(self.source),
                repr(self.target))
        else:
            return "%s(%s, %s, %s)" % (
                self.__class__.__name__,
                repr(self.source),
                repr(self.target),
                repr(self.weight))

    def __cmp__(self, other):
        """Comparing of edges (the weight first)."""
        # Check weights.
        if self.weight > other.weight:
            return 1
        if self.weight < other.weight:
            return -1
        # Check the first node.
        if self.source > other.source:
            return 1
        if self.source < other.source:
            return -1
        # Check the second node.
```

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

    def __hash__(self):
        """Hashable edges."""
        return hash(repr(self))

    def __invert__(self):
        """Return the edge with the opposite direction."""
        return self.__class__(self.target, self.source, self.weight)

    inverted = __invert__
```

A.2. Klasa Graph

Klasa Graph opiera się na słownikach, które umożliwiają w Pythonie wyszukiwanie kluczy w stałym czasie. Pierwszy słownik jako klucze zawiera wszystkie wierzchołki grafu, a wartościami są zagnieżdżone słowniki drugiego rzędu. Słowniki drugiego rzędu jako klucze zawierają wierzchołki, do których prowadzą krawędzie o początku w kluczach słownika pierwszego rzędu. Wartościami w słownikach drugiego rzędu są całe obiekty krawędzi, instancje klasy Edge. Poprzednio publikowane implementacje klasy Graph miały w tym miejscu wagę krawędzi (liczbę).

Obecna implementacja ma interfejs zgodny z dawnym podejściem, co zapewnia stosowalność istniejących implementacji algorytmów. Zaletą tej implementacji jest możliwość przechowywania w grafie krawędzi z większą liczbą atrybutów, o ile w klasie Edge zostanie odpowiednio rozszerzona operacja ~edge, zwracająca krawędź przeciwnie skierowaną do edge.

Do przechowywania grafu topologicznego w klasie Graph zastosowane zostały słowniki edge_next i edge_prev. Słowniki te implementują listy przechowujące uporządkowane krawędzie wychodzące z wierzchołka.

Korzystając z właśności edge_next i edge_prev zaimplementowana została metoda get_faces(), która zwraca listę wszystkich ścian grafu. Pojedyncza ściana jest reprezentowana jako lista kolejnych krawędzi tworzących daną ścianę.

Listing A.2. Klasa Graph z modułu graphs.

```
#!/usr/bin/python

import random
from edges import Edge

class Graph(dict):
    """The class defining a graph."""

    def __init__(self, n=0, directed=False):
        """Load up a Graph instance."""
        self.n = n                  # compatibility
```

```
        self.directed = directed  # bool

        # Structures defining topological representation of the graph.
        self.edge_next = None
        self.edge_prev = None

    def iterfaces(self):
        """Generate all faces from the graph, based on its embedding
        calculated during planarity testing."""
        if not self.edge_next or not self.edge_prev:
            raise ValueError("Embedding not calculated.")
        used = set()
        for edge in self.edge_next:
            if edge in used:
                continue
            used.add(edge)
            face = [edge]
            edge = self.edge_next[~edge]
            while edge not in used:
                used.add(edge)
                face.append(edge)
                edge = self.edge_next[~edge]
            yield face

    def is_directed(self):
        """Test if the graph is directed."""
        return self.directed

    def f(self):
        """Return the number of faces (for planar graphs)."""
        if not self.edge_next:
            raise ValueError("run planarity test first")
        return self.e() + 2 - self.n   # twierdzenie Eulera

    def v(self):
        """Return the number of nodes (the graph order)."""
        return len(self)

    def e(self):
        """Return the number of edges in O(V) time."""
        edges = sum(len(self[node]) for node in self)
        return (edges if self.is_directed() else edges / 2)

    def add_node(self, node):
        """Add a node to the graph."""
        if node not in self:
            self[node] = dict()

    def has_node(self, node):
        """Test if a node exists."""
        return node in self

    def del_node(self, node):
        """Remove a node from the graph (with edges)."""
        # dictionary changes size during iteration.
        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):
        """Add an edge to the graph (missing nodes are created)."""
        if edge.source == edge.target:
            raise ValueError("loops are forbidden")
        self.add_node(edge.source)
        self.add_node(edge.target)
        if edge.target not in self[edge.source]:
            self[edge.source][edge.target] = edge
        else:
            raise ValueError("parallel edges are forbidden")
        if not self.is_directed():
            if edge.source not in self[edge.target]:
                self[edge.target][edge.source] = ~edge
            else:
                raise ValueError("parallel edges are forbidden")

    def del_edge(self, edge):
        """Remove an edge from the graph."""
        del self[edge.source][edge.target]
        if not self.is_directed():
            del self[edge.target][edge.source]

    def has_edge(self, edge):
        """Test if an edge exists (the weight is not checked)."""
        return edge.source in self and edge.target in self[edge.source]

    def weight(self, edge):
        """Return the edge weight or zero."""
        if edge.source in self and edge.target in self[edge.source]:
            return self[edge.source][edge.target].weight
        else:
            return 0

    def iternodes(self):
        """Generate the nodes from the graph on demand."""
        return self.iterkeys()

    def iteradjacent(self, source):
        """Generate the adjacent nodes from the graph on demand."""
        return self[source].iterkeys()

    def iteroutedges(self, source):
        """Generate the outedges from the graph on demand."""
        for target in self[source]:
            yield self[source][target]

    def iterinedges(self, source):
        """Generate the inedges from the graph on demand."""
        if self.is_directed():   # O(V) time
            for target in self.iternodes():
                if source in self[target]:
                    yield self[target][source]
```

```
        else:
            for target in self[source]:
                yield self[target][source]

    def iteredges(self):
        """Generate the edges from the graph on demand."""
        for source in self.iternodes():
            for target in self[source]:
                if self.is_directed() or source < target:
                    yield self[source][target]

    def show(self):
        """The graph presentation."""
        for source in self.iternodes():
            print source, ":",
            for edge in self.iteroutedges(source):
                if edge.weight == 1:
                    print edge.target,
                else:
                    print "%s(%s)" % (edge.target, edge.weight),
            print

    def copy(self):
        """Return the graph copy."""
        new_graph = Graph(n=self.n, directed=self.directed)
        for node in self.iternodes():
            new_graph[node] = dict(self[node])
        return new_graph

    def transpose(self):
        """Return the transpose of the graph."""
        new_graph = Graph(n=self.n, directed=self.directed)
        for node in self.iternodes():
            new_graph.add_node(node)
        for edge in self.iteredges():
            new_graph.add_edge(~edge)
        return new_graph

    def degree(self, source):
        """Return the degree of the node in the undirected graph."""
        if self.is_directed():
            raise ValueError("the graph is directed")
        return len(self[source])

    def outdegree(self, source):
        """Return the outdegree of the node."""
        return len(self[source])

    def indegree(self, source):
        """Return the indegree of the node."""
        if self.is_directed():   # O(V) time
            counter = 0
            for target in self.iternodes():
                if source in self[target]:
                    counter = counter + 1
            return counter
        else:                    # O(1) time
```

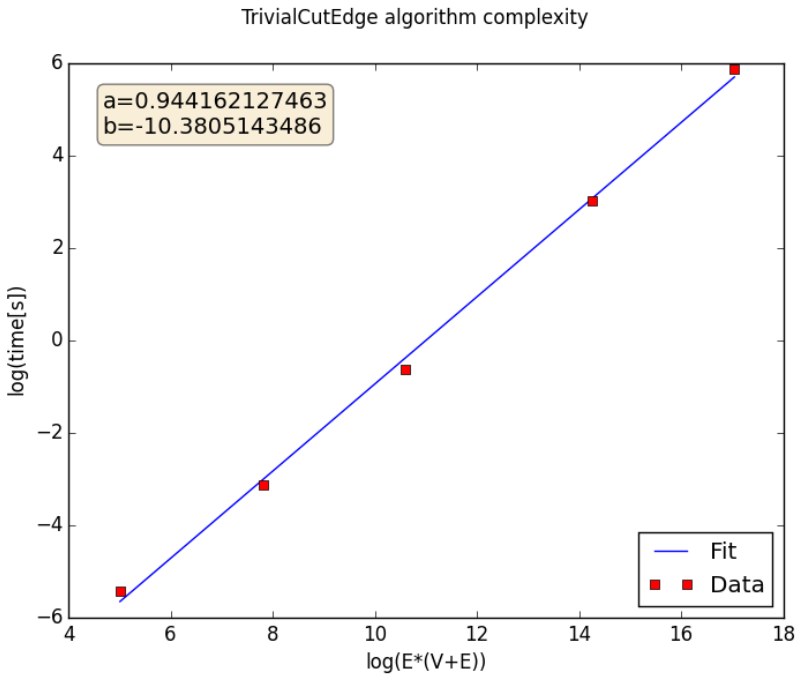
```
            return len(self[source])

    def __eq__(self, other):
        """Test if the graphs are equal."""
        if self.is_directed() is not other.is_directed():
            #print "directed and undirected graphs"
            return False
        if self.v() != other.v():
            #print "|V1| != |V2|"
            return False
        for node in self.iternodes():   # O(V) time
            if not other.has_node(node):
                #print "V1 != V2"
                return False
        if self.e() != other.e():   # inefficient, O(E) time
            #print "|E1| != |E2|"
            return False
        for edge in self.iteredges():   # O(E) time
            if not other.has_edge(edge):
                #print "E1 != E2"
                return False
            if edge.weight != other.weight(edge):
                return False
        return True

    def __ne__(self, other):
        """Test if the graphs are not equal."""
        return not self == other

    def add_graph(self, other):
        """Add a graph to this graph (the current graph is modified)."""
        if self.is_directed() is not other.is_directed():
            raise ValueError("directed vs undirected")
        for node in other.iternodes():
            self.add_node(node)
        for edge in other.iteredges():
            self.add_edge(edge)
```

B. Testy wydajnościowe algorytmów

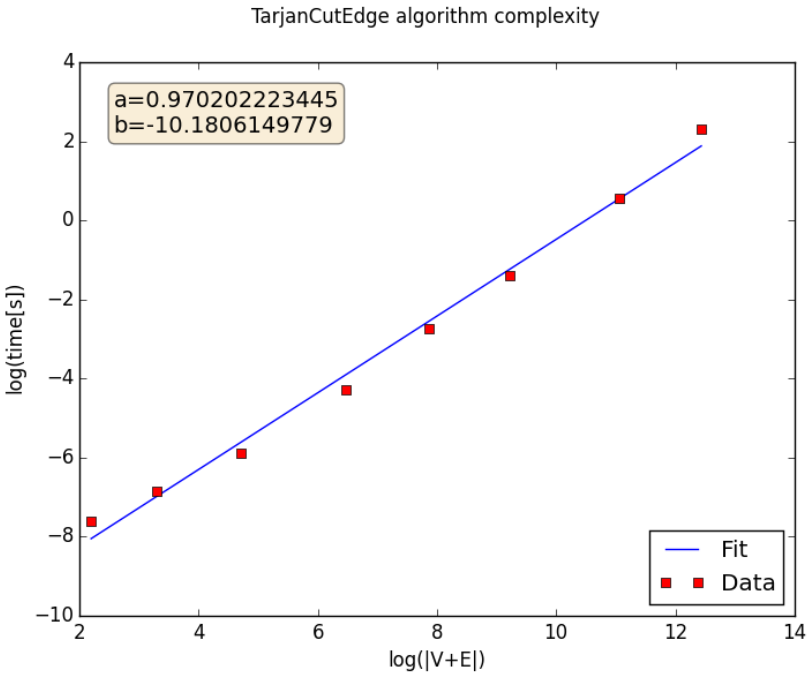
W ramach pracy przeprowadzono testy algorytmów związanych ze spójnością i planarnością grafów.

B.1. Testy algorytmów spójności

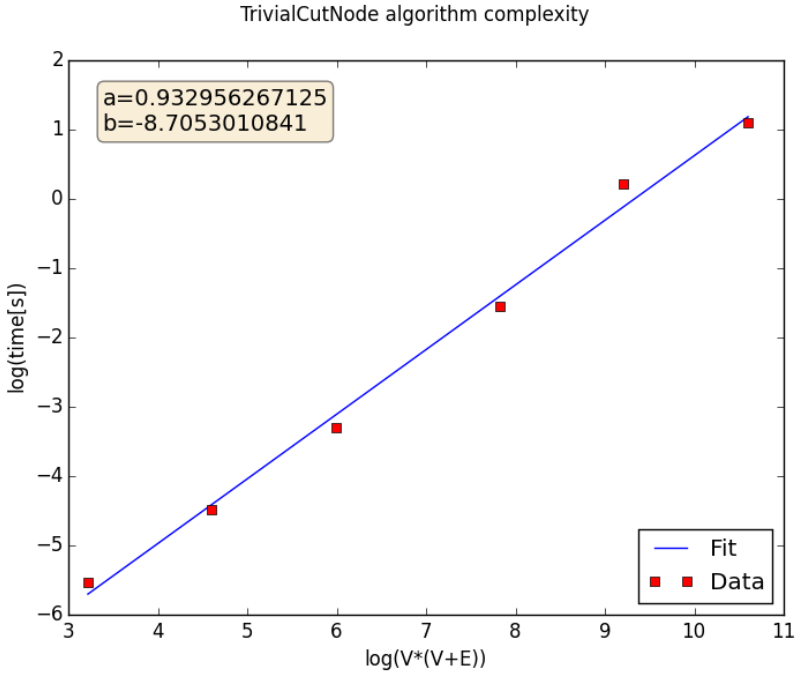
Testy wykonano dla grafów przypadkowych z prawdopodobieństwem istnienia krawędzi równym $p = 0.2$.



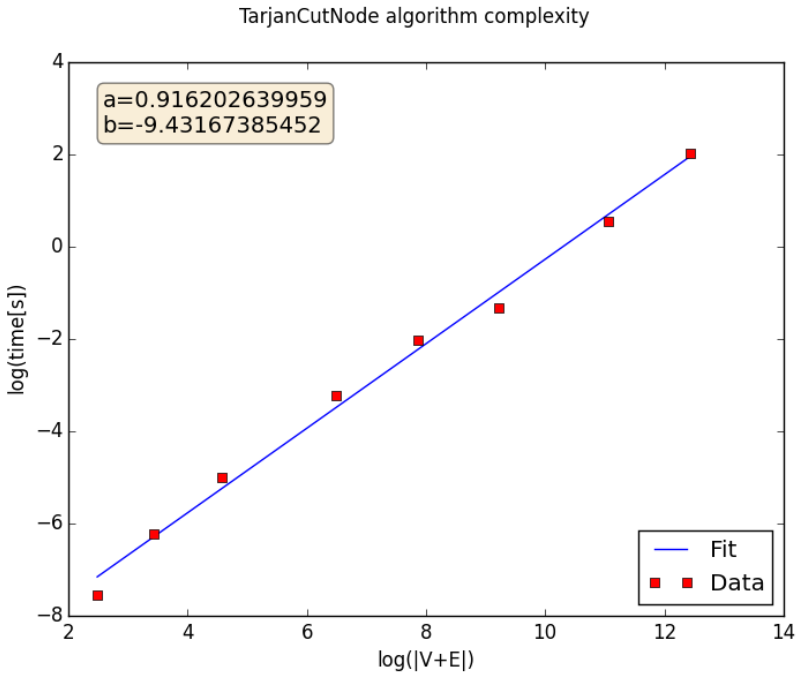
Rysunek B.1. Wykres wydajności algorytmu trywialnego znajdowania mostów. Współczynnik a bliski 1 potwierdza zależność liniową.



Rysunek B.2. Wykres wydajności algorytmu Tarjana znajdowania mostów. Współczynnik a bliski 1 potwierdza zależność liniową.



Rysunek B.3. Wykres wydajności algorytmu naiwnego znajdowania punktów artykulacji. Współczynnik a bliski 1 potwierdza zależność liniową.

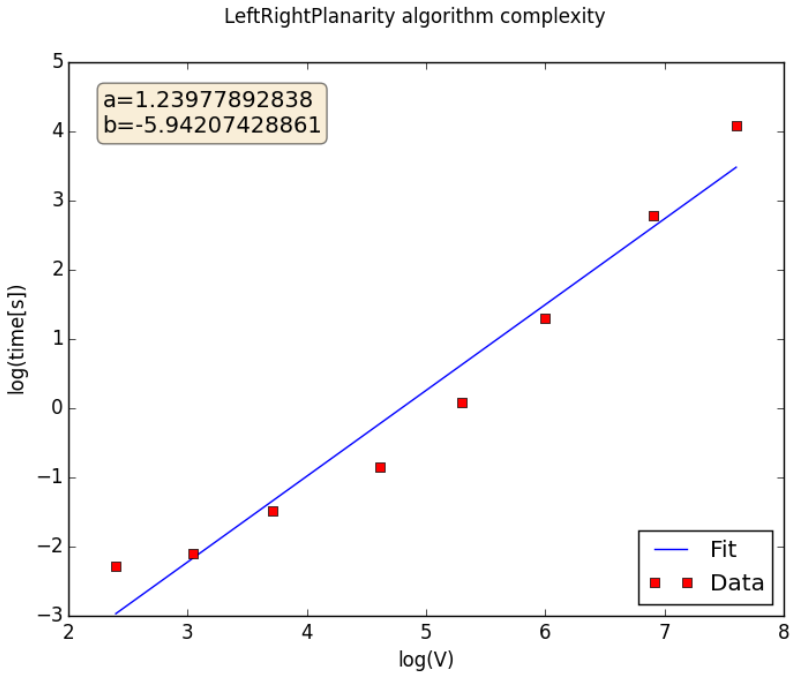


Rysunek B.4. Wykres wydajności algorytmu Tarjana znajdowania punktów artykulacji. Współczynnik a bliski 1 potwierdza zależność liniową.

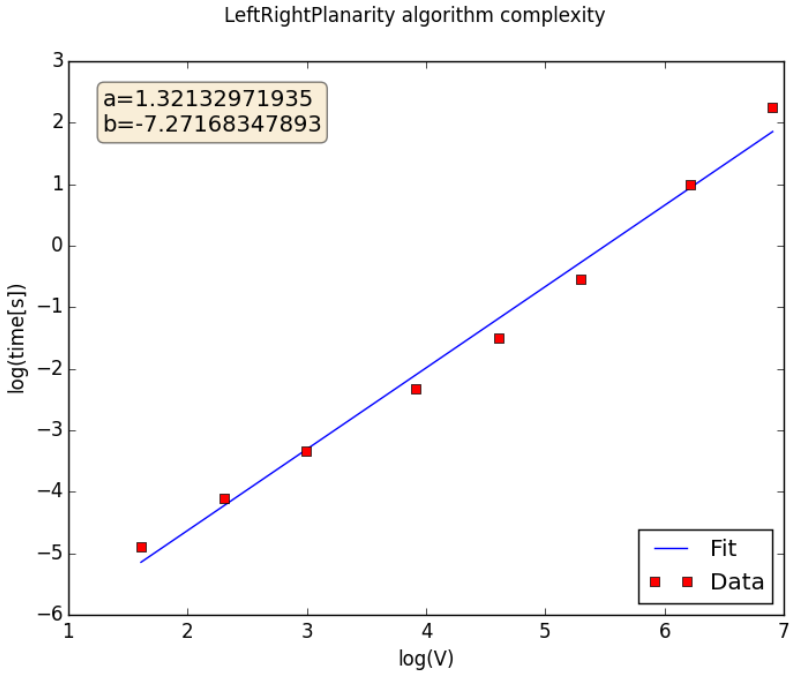
B.2. Testy algorytmu lewy-prawy

Testy zostały wykonane dla drzew, drzew z poprzeczkami, oraz grafów cyklicznych.

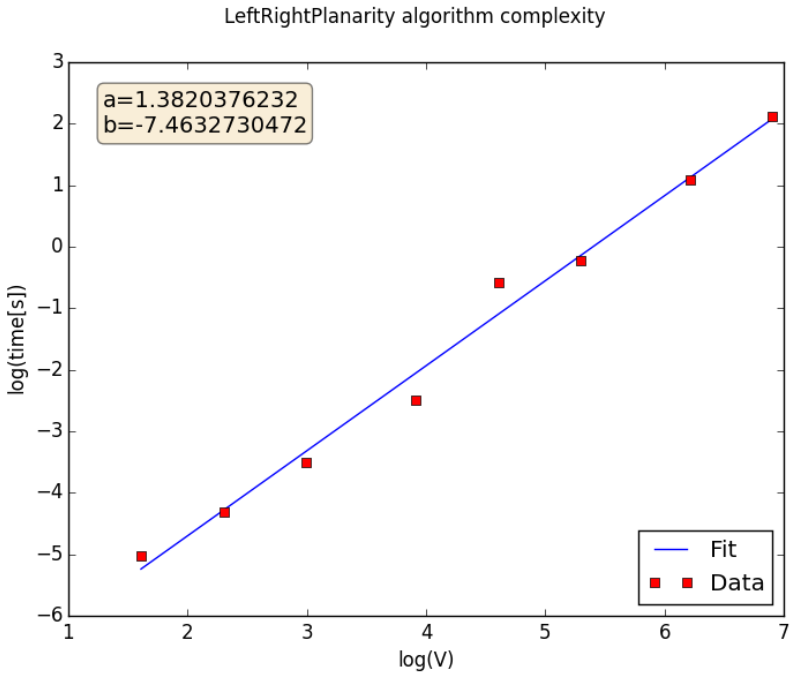
Wykresy sugerują, że implementacja zawiera czynnik o złożoności powyżej liniowej. Pierwszym kandydatem jest sortowanie krawędzi w metodzie _sort_by_nesting_order, ponieważ nie jest to całkowicie sortowanie kubełkowe, postulowane przez Brandesa.



Rysunek B.5. Wykres wydajności algorytmu lewy-prawy dla drzew z poprzeczkami. Współczynnik a większy od 1 sugeruje istnienie w implementacji czynnika o wyższej złożoności niż liniowa.



Rysunek B.6. Wykres wydajności algorytmu lewy-prawy dla grafów cyklicznych. Współczynnik a większy od 1 sugeruje istnienie w implementacji czynnika o wyższej złożoności niż liniowa.



Rysunek B.7. Wykres wydajności algorytmu lewy-prawy dla drzew. Współczynnik a większy od 1 sugeruje istnienie w implementacji czynnika o wyższej złożoności niż liniowa.

Bibliografia

[1] *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*, Roberto Tamassia (Editor), Chapman and Hall/CRC Press, 2013.

[2] H. de Fraysseix, P. Rosenstiehl, *A depth-first-search characterization of planarity*, Annals of Discrete Mathematics 13, 75–80 (1982).

[3] H. de Fraysseix, P. Rosenstiehl, *A characterization of planar graphs by Trémaux orders*, Combinatorica 5, 127–135 (1985).

[4] H. de Fraysseix, P. O. de Mendez, *Trémaux trees and planarity*, European Journal of Combinatorics 33, 279–293 (2012).

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

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

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

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

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

[10] J. Hopcroft, R. Tarjan, *Efficient planarity testing*, Journal of the Association for Computing Machinery 21, 549–568 (1974)

[11] John M. Boyer, Wendy J. Myrvold *On the Cutting Edge: Simplified O(n) Planarity by Edge Addition*, Journal of Graph Algorithms and Applications 8, 241–273 (2004).

[12] U. Brandes, *The Left-Right Planarity Test*, manuskrypt przedłożony do publikacji (2009).

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

[14] Wikipedia, Graph Teory, 2015, `https://en.wikipedia.org/wiki/Graph_theory`.

[15] Wikipedia, Apollonian network, 2015, `https://en.wikipedia.org/wiki/Apollonian_network`.

[16] Wikipedia, Squaregraph, 2015, `http://en.wikipedia.org/wiki/Squaregraph`.

[17] Wikipedia, Outerplanar graph, 2015, `http://en.wikipedia.org/wiki/Outerplanar_graph`.

[18] G. Chartrand, F. Harary, *Planar permutation graphs*, Annales de l'institut Henri Poincaré (B) Probabilités et Statistiques 3 (4), 433–438 (1967).

[19] Wikipedia, Halin graph, 2015, `http://en.wikipedia.org/wiki/Halin_graph`.

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

[21] Wikipedia, Combinatorial map, 2015,
https://en.wikipedia.org/wiki/Combinatorial_map.

[22] R. E. Tarjan, *A note on finding the bridges of a graph*, Information Processing Letters 2, 160-161 (1974).

[23] J. Wałaszek, *Algorytmy, struktury danych. Znajdowanie punktów artykulacji w grafie*, 2015,
http://edu.i-lo.tarnow.pl/inf/alg/001_search/0130b.php.

[24] K. Mehlhorn, P. Mutzel, *On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm*, Algorithmica 16, 233–242, (1996).

[25] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, *A linear algorithm for embedding planar graphs using PQ-trees* Journal of Computer and System Sciences, 30, 54–76 1985.

[26] Wikipedia, Left-right planarity test, 2015,
http://en.wikipedia.org/wiki/Left-right_planarity_test.

[27] H. de Fraysseix, P. O. de Mendez, P. Rosenstiehl, *Trémaux trees and planarity*, International Journal of Foundations of Computer Science 17, 1017–1029 (2006); arXiv:math.CO/0610935.

[28] Wikipedia, Planarity testing, 2015,
http://en.wikipedia.org/wiki/Planarity_testing.

[29] J. Wałaszek, *Algorytmy, struktury danych. Znajdowanie mostów w grafie*, 2015,
http://edu.i-lo.tarnow.pl/inf/alg/001_search/0130a.php.