Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Łukasz Gałuszka

Nr albumu: 1051530

Implementacja wybranych algorytmów dla grafów z wagami w języku Python

Praca magisterska na kierunku Informatyka

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

Kraków 2014

Oświadczenie autora pracy

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

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

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

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

Kraków, dnia Podpis kierującego pracą

Składam serdeczne podziękowania Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu, promotorowi mojej pracy, za okazaną pomoc i cenne uwagi, bez których niemożliwe byłoby ukończenie tej pracy.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów i struktur danych dla grafów z wagami. Zdefiniowano interfejs grafów ważonych i stworzono dwie klasy o odmiennych właściwościach, realizujące ten interfejs. Klasa MatrixGraph bazuje na reprezentacji macierzy sąsiedztwa, natomiast klasa DictGraph jest zbliżona do reprezentacji list sąsiedztwa. Dla krawędzi stworzono osobne klasy, mianowicie Edge oraz UdirectedEdge. Dodano także generatory dla kilku typów grafów.

W pracy zaimplementowano algorytmy grafowe związane z przeszukiwaniem grafów, wyznaczaniem minimalnego drzewa rozpinającego, wyznaczaniem najkrótszych ścieżek z jednego i wielu źródeł, znajdowania maksymalnego przepływu w sieci przepływowej. Każdemu algorytmowi odpowiada osobna klasa, a w danej klasie może być zawartych kilka różnych implementacji. W pracy pojawiły się też pewne pomocnicze algorytmy i struktury danych, np. przeszukiwanie grafów wszerz i przeszukiwanie w głąb. Przeszukiwanie w głąb wykorzystano do sortowania topologicznego wierzchołków dagu. Korzystano również ze struktury danych zbiorów rozłącznych.

Zaimplementowano trzy algorytmy rozwiązujące problem minimalnego drzewa rozpinającego: algorytm Borůvki, algorytm Prima (trzy implementacje), algorytm Kruskala. Zaimplementowano trzy algorytmy rozwiązujące problem najkrótszych ścieżek z jednym źródłem: najkrótsze ścieżki w dagu, algorytm Bellmana-Forda, algorytm Dijkstry (dwie implementacje). Zaimplementowano dwa algorytmy rozwiązujące problem najkrótszych ścieżek między wszystkimi parami wierzchołków: algorytm Floyda-Warshalla, algorytm Johnsona. W końcu zaimplementowano dwa algorytmy rozwiązujące problem maksymalnego przepływu w sieci przepływowej: algorytm Forda-Fulkersona, algorytm Edmondsa-Karpa.

Dla wszystkich algorytmów przygotowano testy poprawności z wykorzystaniem modułu unittest, który jest standardowym narzędziem wspomagającym sprawdzanie kodu w Pythonie. Ponadto wykonano eksperymenty komputerowe sprawdzające zgodność rzeczywistej wydajności kodu z przewidywaniami teoretycznymi.

Słowa kluczowe: grafy ważone, przeszukiwanie wszerz, przeszukiwanie w głąb, sortowanie topologiczne, minimalne drzewo rozpinające, najkrótsze ścieżki, sieci przepływowe, maksymalny przepływ

Abstract

Python implementations of weighted graphs and selected graph algorithms are presented. Graphs interface is defined and two different classes are created which implement this interface. The MatrixGraph class is based on the adjacency-matrix representation of graphs. The DictGraph class is based on the adjacency-list representation, but with fast lookup of nodes and neighbours (dict-of-dict structure). There are also classes for graph edges: Edge and UdirectedEdge. Some graph generators are also included.

In this work, many graph algorithms are implemented using a unified approach. There are separate classes devoted to different algorithms, but sometimes several algorithm versions are included in the same class. Some auxilliary algorithms and data structures are used. Two graphs searching or traversing algorithms are implemented: breadth-first search (BFS) and depth-first search (DFS). Topological sorting algorithm is based on DFS. A disjoint-set data structure is also included.

Three algorithms for finding a minimum spanning tree are implemented: the Borůvka's algorithm, the Prim's algorithm (three implementations), and the Kruskal's algorithm. Three algorithms for solving the single-source shortest path problem are implemented: the dag shortest path algorithm, the Bellman-Ford algorithm, and the Dijkstra's algorithm (two implementations). Two algorithms for solving all-pairs shortest path problem are implemented: the Floyd-Warshall algorithm and the Johnson's algorithm. Two algorithms for computing the maximum flow in a flow network are implemented: the Ford-Fulkerson algorithm and the Edmonds-Karp algorithm.

All algorithms were tested by means of the unittest module, the Python unit testing framework. Additional computer experiments were done in order to compare real and theoretical computational complexity.

Keywords: weighted graphs, breadth-first search, depth-first search, topological sorting, minimum spanning tree, shortest paths, flow networks, maximum flow

Spis treści

1. Wstęp . . . 3
- 1.1. Grafy . . . 3
- 1.2. Organizacja pracy . . . 4

2. Język Python . . . 5
- 2.1. Typy danych . . . 5
  - 2.1.1. Obiekt null . . . 5
  - 2.1.2. Typ logiczny . . . 5
  - 2.1.3. Typy liczbowe . . . 6
  - 2.1.4. Stringi . . . 7
  - 2.1.5. Listy . . . 7
  - 2.1.6. Krotki . . . 7
  - 2.1.7. Zbiory . . . 8
  - 2.1.8. Słowniki . . . 8
  - 2.1.9. Wyjątki . . . 9
- 2.2. Instrukcje sterujące . . . 9
  - 2.2.1. Instrukcja warunkowa if . . . 9
  - 2.2.2. Pętla for . . . 10
  - 2.2.3. Pętla while . . . 10
  - 2.2.4. Instrukcje break, continue, pass . . . 10
- 2.3. Funkcje . . . 10
- 2.4. Moduły . . . 10
- 2.5. Klasy . . . 11

3. Teoria grafów . . . 12
- 3.1. Grafy skierowane . . . 12
- 3.2. Grafy nieskierowane . . . 12
- 3.3. Grafy z wagami . . . 12
- 3.4. Gęstość grafu . . . 13
- 3.5. Ścieżki i cykle . . . 13
- 3.6. Spójność . . . 13
- 3.7. Drzewa i las . . . 14

4. Implementacja grafów . . . 15
- 4.1. Implementacja obiektów z teorii grafów . . . 15
- 4.2. Sposoby reprezentacji grafów . . . 16
  - 4.2.1. Macierz sąsiedztwa . . . 16
  - 4.2.2. Lista sąsiedztwa . . . 17
  - 4.2.3. Reprezentacja słownikowa . . . 17
- 4.3. Interfejs grafów . . . 17

5. Algorytmy i ich implementacje . . . 19
- 5.1. Przeszukiwanie grafów . . . 19
  - 5.1.1. Przeszukiwanie wszerz . . . 19

5.1.2. Przeszukiwanie wgłąb . . . 22
5.1.3. Sortowanie topologiczne . . . 23
5.2. Minimalne drzewo rozpinające . . . 24
5.2.1. Algorytm Borůvki . . . 24
5.2.2. Algorytm Prima . . . 28
5.2.3. Algorytm Kruskala . . . 33
5.3. Najkrótsza ścieżka pomiędzy parą wierzchołków . . . 37
5.3.1. Najkrótsza ścieżka w dagu . . . 40
5.3.2. Algorytm Bellmana-Forda . . . 41
5.3.3. Algorytm Dijkstry . . . 43
5.4. Najkrótsze ścieżki pomiędzy wszystkimi parami wierzchołków . . . 45
5.4.1. Algorytm "mnożenia macierzy" . . . 47
5.4.2. Algorytm Floyda-Warshalla . . . 50
5.4.3. Algorytm Johnsona . . . 53
5.5. Maksymalny przepływ . . . 55
5.5.1. Sieć przepływowa . . . 57
5.5.2. Algorytm Forda-Fulkersona . . . 58
5.5.3. Algorytm Edmondsa-Karpa . . . 60
6. Podsumowanie . . . 65
A. Kod źródłowy dla klas krawędziowych . . . 66
A.1. Klasa Edge . . . 66
A.2. Klasa UndirectedEdge . . . 67
B. Kod źródłowy dla klas grafowych . . . 68
B.1. Klasa BaseGraph . . . 68
B.2. Klasa DictGraph . . . 70
B.3. Klasa MatrixGraph . . . 73
C. Testy dla klas grafowych . . . 77
C.1. Wymagania pamięciowe grafów . . . 77
C.1.1. Wymagania pamięciowe implementacji słownikowej . . . 77
C.1.2. Wymagania pamięciowe implementacji macierzowej . . . 77
C.1.3. Porównanie pamięciowe implementacji macierzowej i słownikowej . . . 78
C.1.4. Porównanie wydajnościowe wybranych operacji na implementacji macierzowej i słownikowej . . . 78
Bibliografia . . . 86

1. Wstęp

Algorytmy towarzyszyły nam od zawsze, były nawet długo przed pojawieniem się komputerów, a wraz z ich pojawieniem zaczęły się bardzo dynamicznie rozwijać. Od wyboru algorytmu zależy, czy przetwarzanie danych przez komputer będzie szybkie (wydajność czasowa), albo czy w ogóle wykonalne (wydajność pamięciowa).

Praca magisterska poświęcona jest algorytmom grafowym związanym z grafami ważonymi. Powstała po pierwsze w celu dydaktycznym, aby przedstawić w intuicyjny i zwięzły sposób często skomplikowane algorytmy. Drugim celem było stworzenie działającego i przetestowanego kodu, który pozwoli na rozwiązywanie średniej wielkości problemów obliczeniowych związanych z grafami. Nie będziemy się zajmować problemem wizualizacji grafów.

Do prezentacji algorytmów wybrano język Python [1], który w swoich założeniach miał na uwadze czytelność kodu, przejrzystą składnię i łatwość nauki (rozdział 2). Dla algorytmów powstały testy jednostkowe (ang. *unit tests*), które testują poprawność implementacji.

1.1. Grafy

Graf jest to zbiór obiektów (wierzchołków), gdzie pewne pary obiektów są ze sobą powiązane za pomocą łączy (krawędzi) [2], [3]. Jest to struktura matematyczna, która pozwala na modelowanie różnego rodzaju relacji i procesów w układach fizycznych, biologicznych, społecznych i informatycznych. W zależności od potrzeb strukturę grafu można wzbogacać o dodatkowe elementy, np. krawędzie mogą mieć kierunek (grafy skierowane) lub przypisaną pewną liczbę, nazywaną *wagą* (grafy ważone). Za pierwszego badacza grafów uważa się Leonarda Eulera, który rozstrzygnął zagadnienie mostów królewieckich w XVIII wieku.

Z nastaniem komputerów grafy możemy reprezentować za pomocą struktur danych, które ułatwiają i przyspieszają obliczenia. Ze strukturami danych nierozerwalnie łączą się algorytmy, które je przetwarzają. W pracy przebadano wydajność algorytmów grafowych bazujących na dwóch implementacjach grafów - słownikowej (rozdział 4.2.3) oraz macierzowej (rozdział 4.2.1). Obie implementacje wykorzystują wspólny interfejs grafów, czyli zestaw operacji potrzebnych do działań na grafach.

1.2. Organizacja pracy

Rozdział 1 zawiera wprowadzenie do niniejszej pracy. Rozdział 2 zawiera krótki opis języka Python. Rozdział 3 zawiera wprowadzenie do teorii grafów. Rozdział 4 zawiera opis interfejsu grafów oraz sposobów jego realizacji.

Rozdział 5 zawiera opis wybranych algorytmów grafowych dla grafów z wagami. Są to algorytmy przeszukiwania grafów (rozdział 5.1), wyznaczania minimalnego drzewa rozpinającego (rozdział 5.2) wyznaczania najkrótszej ścieżki pomiędzy parą wierzchołków (rozdział 5.3), pomiędzy wszystkimi parami wierzchołków (rozdział 5.4), oraz maksymalnego przepływu w sieci przepływowej (rozdział 5.5).

Rozdział 6 zawiera podsumowanie wyników pracy. W dodatkach został umieszczony kod źródłowy klas dla krawędzi (dodatek A), klas dla grafów (dodatek B), oraz opis testów komputerowych (dodatek C).

2. Język Python

Python jest językiem skryptowym wysokiego poziomu. Ogólnie programy napisane w Pythonie są zwykle wolniejsze od napisanych w Javie czy C++, ale pisanie w tym języku oszczędza czas i wymaga mniej linii kodu. Pythonowe programy mogą być nawet wielokrotnie krótsze. Python jest językiem z typami dynamicznymi, co sprawia że programista nie musi tracić czasu na deklarowanie typów. Python wspiera wiele metodologii programowania, np. programowanie zorientowane obiektowo czy programowanie funkcyjne. W Pythonie są dostępne struktury danych wysokiego poziomu, takie jak słowniki czy listy powiązane. Cechą charakterystyczną Pythona jest elegancka składnia i prawie jednolity styl kodowaniania, co sprawia że programy napisane przez jednych programistów są czytelne dla innych. Cechy te sprawiają, że Python jest łatwy do nauczenia w krótkim czasie, a kod napisany w nim jest łatwy do czytania i zrozumienia.

Obecnie są dostępne dwie główne gałęzie Pythona, które nie są ze sobą w pełni kompatybilne: Python 2 i Python 3. Niniejsza praca powstała na bazie Pythona 2, który ciągle jest domyślnym ustawieniem w dystrybucjach Linuksa, ze względu na jego stabilność i dużo dodatkowych modułów. Kod testowano z użyciem wersji 2.6 i 2.7.

2.1. Typy danych

Sekcja zawiera opis standardowych typów danych dostępnych w Pythonie.

2.1.1. Obiekt null

Jest tylko jedna instacja obiektu null, która nazwana została None. Stała ta używana jest do reprezentacji pustej wartości, np. kiedy nazwa zmiennej jest znana, ale jej wartość jeszcze nie została określona. Jeżeli funkcja w Pythonie jawnie nie zwraca wartości przez **return**, to zwraca właśnie None.

2.1.2. Typ logiczny

Każdy obiekt w Pythonie może być testowany jako wartość logiczna. Ponaddto zdefiniowano dwie stałe logiczne True oraz False, oznaczające odpowiednio prawdę i fałsz.

Poniższe wartości zawsze oznaczają fałsz:

— None,
— False,
— zero dla typów numerycznych,
— puste kolekcje,
— puste stringi,

— instancje klas użytkownika, jeśli definiują metody __nonzero__ lub __len__, a zwracają odpowiednio False lub 0.

Operacje logiczne:
— alternatywa x **or** y,
— koniunkcja x **and** y,
— negacja **not** x.

2.1.3. Typy liczbowe

int - typ liczb całkowitych, po przekroczeniu zakresu następuje automatyczna konwersja do typu long,
long - typ liczb całkowitych,
float - typ liczb zmiennoprzecinkowy,
complex - typ liczb zespolonych,
decimal - typ liczb dziesiętne o ustalonej precyzji (moduł decimal, Python 2.4+),
fractions - typ reprezentujący ułamki (moduł fractions, Python 2.6+).

Operacje arytmetyczne:
— dodawanie x + y,
— odejmowanie x − y,
— mnożenie x ∗ y,
— potęgowanie x ∗∗ y lub pow(x, y),
— dzielenie x / y,
— dzielenie całkowite x // y,
— reszta z dzielenia x % y,
— negacja −x,
— wartość bezwzględna abs(x),
— konwersja do danego typu: int (x), long(x), float (x),
— para (x // y, x % y) divmod(x,y),
— obcięcie cześci ułamkowej math.trunc(x),
— zaokrąglenie do częsci dziesiętnej round(x[, n]), gdzie n jest opcjonalne, domyślnie 0,
— floor największa liczba całkowita <= x math.floor(x),
— ceil najmniejsza liczba całkowita >= x math.ceil(x),
— minimum min(iterable) dla jednego argumentu z iteratorem lub wielu argumentów min(a, b, c, ...) ,
— maksimum max(iterable) dla jednego argumentu z iteratorem lub wielu argumentów max(a, b, c, ...) ,
— suma sekwencji liczb sum(sequence).

Operacje bitowe:
— alternatywa x | y,
— koniunkcja x & y,
— alternatywa wykluczająca x ^ y,
— przesunięcie w lewo x << y,
— przesunięcie w prawo x >> y,
— negacja ~x.

2.1.4. Stringi

String jest to uporządkowany ciąg znaków, służący do przechowywania informacji tekstowej. Należy do sekwencji niezmiennych (ang. *immutable*), tzn. że nie można ich zmieniać bo utworzeniu. Stringi pozwalają m.in. na indeksowany dostęp do poszczególnych znaków, czy konkatenację (sklejanie) napisów.

Podstawowe operacje na stringach:
— tworzenie napisu pustego "" lub '',
— tworzenie napisu abc "abc" lub 'abc',
— długość napisu len(text),
— konkatenacja text1 + text2,
— powtarzanie text * n lub n * text,
— indeksowany dostęp do poszczególnych znaków text[i],
— wycinanie text[i:j],
— kopiowanie copied=text[:] lub copied=str(text),
— iteracja **for** char **in** text,
— zawieranie text1 **in** text2 lub text1 **not in** text2,
— formatowanie: "python %f i %s"% (2.7, 3.4) generuje "python 2.7 i 3.4",
— usuwanie **del** var_with_text,
— sklejanie fragmentów text.join(iterable_with_texts).

2.1.5. Listy

Listy są uporządkowanymi sekwencjami obiektów, a do poszczególnych elementów z listy można się odwoływać poprzez indeks. Listy są kolekcjami zmiennymi (ang. *mutable*), czyli mogą się zmieniać.

Podstawowe operacje na listach:
— tworzenie pustej listy empty_list= [ ] lub empty_list = list(),
— tworzenie listy heterogenicznej some_list= [1, 'ab', 0.5],
— długość listy len(some_list),
— konkatenacja list1 + list2,
— powtarzanie some_list * n lub n * some_list,
— indeksowany dostęp do poszczególnych elementów some_list[i],
— wycinanie podlisty some_list[i:j],
— kopiowanie copied=some_list[:] lub copied=list(some_list),
— iteracja **for** item **in** some_list,
— zawieranie item **in** some_list lub item **not in** some_list,
— usuwanie elementu **del** some_list[i],
— usuwanie wycinka **del** some_list[i:j],
— usuwanie całej listy **del** some_list.

2.1.6. Krotki

Krotki są uporządkowanymi sekwencjami obiektów, a do poszczególnych elementów z krotki można się odwoływać poprzez indeks. W odróżnieniu od list, krotki są kolekcjami niezmiennymi (ang. *immutable*), czyli raz stworzone nie mogą się zmieniać.

Podstawowe operacje na krotkach:

— tworzenie pustej krotki empty_tuple = () lub empty_tuple = tuple(),
— tworzenie krotki z jednym elementem ('a') some_tuple = ('a',),
— tworzenie krotki ('a', 1, 2.3) some_tuple = ('a', 1, 2.3),
— długość krotki len(some_tuple),
— konkatenacja tuple1 + tuple2,
— powtarzanie some_tuple * n lub n * some_tuple,
— indeksowany dostęp do poszczególnych elementów some_tuple[i],
— wycinanie some_tuple[i:j],
— iteracja **for** item **in** some_tuple,
— zawieranie item **in** some_tuple lub item **not in** some_tuple,
— podstawianie (name, height) = ('luke', 1.75),
— usuwanie krotki **del** some_tuple.

2.1.7. Zbiory

Zbiory to nieuporządkowane kolekcje unikalnych i niezmiennych obiektów. Obsługują one zwykłe operacje matematyczne na zbiorach. W pythonie zostały wbudowane dwa typy zbiorów: set jako typ mutable oraz frozenset jak imutable.

Podstawowe operacje na zbiorach:
— tworzenie pustego zbioru empty_set = set(),
— tworzenie zbioru np. z listy some_set = set ([1,2,3,3,4,5,1]) ,
— ilość elementów w zbiorze len(some_set),
— iteracja **for** key **in** some_set,
— zawieranie key **in** some_set lub key **not in** some_set,
— sprawdzenie zawierania się zbiorów set_a <= set_b lub set_b >= set_a
— iloczyn set_a & set_b,
— suma set_a | set_b,
— różnica set_a − set_b,
— różnica symetryczna set_a ^ set_b.
Dodatkowe operacje dla zbiorów mutable:
— dodanie elementu some_set.add(item),
— usuwanie elementu some_set.remove(item) lub some_set.discard(item),
— usuwanie wszystkich elementów some_set.clear(),
— iloczyn (w miejscu) set_a &= set_b,
— suma (w miejscu) set_a |= set_b,
— różnica (w miejscu) set_a −= set_b,
— różnica symetryczna (w miejscu) set_a ^= set_b.

2.1.8. Słowniki

Słownik to nieuporządkowany zbiór par klucz i wartość. Wartości w słownikach są dostępne za pomocą kluczy, a nie ich pozycji względnej.

Podstawowe operacje na słownikach:
— tworzenie pustego słownika empty_dict = {} lub empty_dict = dict(),
— tworzenie słownika z kluczami some_dict = {'name':'luke', 'weight':70} lub some_dict = dict([('name', 'luke'), ('weight', 70)]),
— ilość elementów w słowniku len(some_dict),
— dostęp do wartości some_dict[key],

— wstawienie nowego elementu some_dict[key]=value,
— kopiowanie new_dict = dict(some_dict),
— iteracja po kluczach **for** key **in** some_dict,
— zawieranie key **in** some_dict lub key **not in** some_dict,
— usuwanie klucza **del** some_dict[key],
— usuwanie słownika **del** some_dict.

2.1.9. Wyjątki

Nawet jeśli dane wyrażenie jest syntaktycznie poprawne, to podczas jego wykonywania może pojawić sie błąd. Wtedy tworzony jest wyjątek (ang. *exception*). Wyjątki mogą być stosowane nie tylko do obsługi błędów, ale też do powiadomiania o zdarzeniach. Rzucony wyjątek można złapać i obsłużyć, lub zostawić, ale spowoduje to wypisanie na standardowe wyjście opisu błędu i zakończenie programu. W Pythonie istnieje standardowy zestaw wyjątków, ponaddto można definiować własne wyjątki.

Podstawowe instrukcje do obsługi wyjątków
— ręczne wywołanie wyjątku **raise**,
— warunkowe wywołanie wyjątku assert,
— przechwytywanie wyjątków **try**/**except**/**else**/**finally**.
Przykładowy kod wykorzystujący wyjątki:

```
def divide(x, y):
    """Funkcja do dzielenia."""
    try:
        result = x / y
    except ZeroDivisionError:
        print "division by zero!"
    else:
        print "result is", result
    finally:
        print "executing finally clause"
```

2.2. Instrukcje sterujące

Przez instrukcje sterujące rozumiemy instrukcje, które mogą zmienić liniowy sposób przetwarzania instrukcji. Jest to instrukcja warunkowa i pętle.

2.2.1. Instrukcja warunkowa if

Instrukcja służy do warunkowego wykonywania jakiegoś bloku instrukcji. Opcjonalnie można sprawdzać więcej przypadków przez **elif**. Można też w końcowej sekcji **else** umieścić kod do wykonania, jeżeli żaden warunek nie będzie spełniony. Przykładowy kod z instrukcją warunkową:

```
if x > 0:
    print 'Positive number'
elif: x < 0:
    print 'Negative number'
else:
    print 'Zero'
```

2.2.2. Pętla for

W wielu językach programowania pętla **for** wykonuje iterację od pewnej wartości początkowej, aż do wartości końcowej. W Pythonie pętla **for** wykonuje iterację po obiektach z pewnej sekwencji, np. listy, krotki, czy stringu. Przykład pętli **for**:

```
for number in sequence_of_numbers:
    print number
```

2.2.3. Pętla while

Pętlę **while** służy do powtarzania wykonywania bloku instrukcji, dopóki warunek logiczny jest prawdziwy. Można tworzyć pętle nieskończone **while** True, które można zakończyć za pomocą instrukcji **break**, znajdującą się w bloku instrukcji **while**. Przykład pętli **while**:

```
number = 5
while number > 0:
    print number
    number = number - 1
```

2.2.4. Instrukcje break, continue, pass

Instrukcja **break** służy do natychmiastowego wyjścia z pętli. Instrukcja **continue** to przejście do następnej iteracji pętli, podobnie jak w Java czy C++. Instrukcja **pass** nic nie robi, jest to pusty pojemnik instrukcji.

2.3. Funkcje

Funkcje służą do grupowania spójnych fragmentów kodu w jedną całość, co zwiększa czytelność i zapobiega duplikowaniu kodu. Funkcje są tworzone głównie za pomocą instrukcji wykonywalnej **def**, która tworzy obiekt funkcji i przypisuje go do nazwy. Można stosować argumenty domyślne, zmienną listę argumentów, rozpakowywanie argumentów z listy, funkcje anonimowe **lambda**. W Pythonie nie ma deklaracji typu argumentów, dlatego cechą funkcji jest polimorfizm: zachowanie funkcji zależy od danych do niej przekazanych. Nie ma też deklaracji zwracanego typu, domyślnie wszystkie funkcje zwracają None, chyba że wskażemy wartość zwracaną poprzez **return**. Przykład funkcji:

```
def some_function(x, y):
    return 2 * x + y
```

2.4. Moduły

Moduły to pliki pythonowe, gdzie przechowywane są definicje obiektów oraz poszczególne instrukcje. Moduły wspomagają ponowne wykorzystywanie kodu poprzez importy. Ponadto każdy moduł tworzy wydzieloną przestrzeń nazw, co zapobiega konfliktom.

2.5. Klasy

Klasy służą do definiowania nowych typów. Klasy w Pythonie udostępniają wszystkie standardowe mechanizmy charakterystyczne dla Programowania Zorientowanego Obiektowo, m.in. dziedziczenie po wielu klasach bazowych, czy przeciążanie metod z klas bazowych (ang. *override*). Wszystkie składowe klasy są publiczne, a metody są wirtualne. Używanie klas pozwala poprzez kompozycję grupować wiele komponentów w jedną całość.

Przykładowa definicja klasy:

```
class MyClass:

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

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

    def get_number(self):
        return number
```

3. Teoria grafów

Teoria grafów to dział matematyki i informatyki zajmujący się badaniem własności grafów [7]. Terminologia teorii grafów nie zawsze jest jednoznacznie ustalona i często znaczenie danego pojęcia zależy od autora. Z tego powodu podamy definicje pojęć używanych w pracy, w większości korzystając z podręcznika Cormena [8].

3.1. Grafy skierowane

Graf (prosty) jest to uporządkowana para $G = (V, E)$, gdzie V to zbiór wierzchołków (skończony i niepusty), a E to zbiór krawędzi. Krawędzią nazywamy uporządkowanę parę (s, t) dwóch *różnych* wierzchołków ze zbioru V. Tak określona krawędź jest skierowana z s do t, a graf G nazywamy *skierowanym*. Krawędź (s, t) jest *wychodząca* z wierzchołka s i jest *wchodząca* do wierzchołka t. Wierzchołek t jest *sąsiedni* względem wierzchołka s, jeżeli w grafie istnieje krawędź (s, t) (relacja sąsiedztwa).

Graf $G' = (V', E')$ jest *podgrafem* grafu $G = (V, E)$, jeżeli V' jest podzbiorem V, oraz E' jest podzbiorem E. Symbole $|V|$ i $|E|$ oznaczają odpowiednio liczbę wierzchołków i liczbę krawędzi grafu. W zapisie z notacją O będziemy dla prostoty pisać V i E.

3.2. Grafy nieskierowane

Często definiuje się krawędzie *nieskierowane* jako dwuelementowe podzbiory zbioru V, np. $\{s, t\}$. Można powiedzieć, że w zbiorze E istnieją jednocześnie dwie krawędzie skierowane (s, t) i (t, s), które są liczone jako jedna krawędź nieskierowana. Graf zawierający tylko krawędzie nieskierowane nazywamy *grafem nieskierowanym*. Krawędź $\{s, t\}$ jest *incydentna* z wierzchołkami s i t. W grafie nieskierowanym relacja sąsiedztwa jest symetryczna.

W wielu zastosowaniach podane definicje grafów są niewystarczające i wprowadza się różne rozszerzenia (pętle, krawędzie wielokrotne). Jednak na nasze potrzeby te definicje są wystarczające.

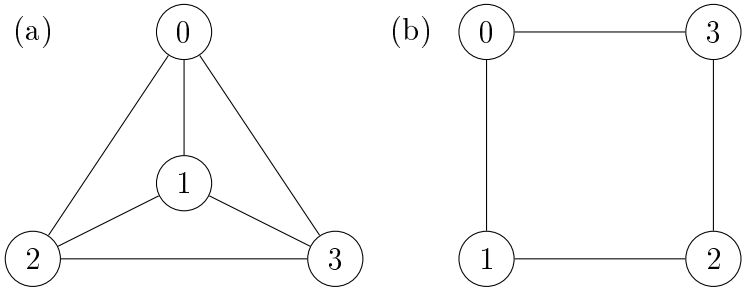
3.3. Grafy z wagami

W pewnych zastosowaniach krawędzie grafu mają przypisane wartości liczbowe, nazywane *wagami*. Wagi mogą reprezentować odległość, przepustowość łącza, koszt, itd. *Graf ważony* to graf zawierający krawędzie z wagami. Wagę krawędzi (s, t) lub $\{s, t\}$ oznaczamy przez $w(s, t)$.

3.4. Gęstość grafu

Gęstość grafu określa nam stosunek liczby krawędzi w danym grafie do maksymalnej możliwej liczby krawędzi w grafie o tej samej liczbie wierzchołków. Jeżeli graf posiada $|V|$ wierzchołków, to może mieć maksymalnie $|V|(|V|-1)/2$ krawędzi. Używa się określeń *graf gęsty* albo *graf rzadki*, chociaż znaczenie tych słów zależy od kontekstu. Zwykle w grafie rzadkim $|E|$ jest $O(V)$, a w grafie gęstym $|E|$ jest $O(V^2)$.

Graf pełny K_n ($|V| = n$, $|E| = n(n-1)/2$) jest to graf o maksymalnej liczbie krawędzi i jest gęsty. *Graf cykliczny* C_n ($|V| = n$, $|E| = n$), *graf liniowy* P_n ($|V| = n$, $|E| = n-1$), czy *graf koło* W_n ($|V| = n$, $|E| = 2n-2$) są grafami rzadkimi.



Rysunek 3.1. (a) Graf pełny $K_4 = W_4$. (b) Graf cykliczny C_4.

3.5. Ścieżki i cykle

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

Ścieżka prosta to ścieżka, w której wszystkie wierzchołki są różne. *Wagą ścieżki* w grafie ważonym nazywamy sumę wag odpowiednich krawędzi.

Cykl jest to ścieżka, w której pierwszy i ostatni wierzchołek są takie same, $v_0 = v_n$. *Cykl prosty* jest to cykl, w którym wszystkie wierzchołki są różne, z wyjątkiem ostatniego. Wydaje się, że w literaturze ścieżki typu (s) i (s, t, s) (grafy nieskierowane) nie są uważane za cykle proste.

Graf niezawierający cykli prostych nazywamy *acyklicznym*. Graf skierowany acykliczny nazywamy *dagiem* (ang. *directed acyclic graph*).

3.6. Spójność

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

3.7. Drzewa i las

Drzewo (ang. *tree*) jest to graf nieskierowany, spójny i acykliczny. *Wagą drzewa* w grafie ważonym nazywamy sumę wag odpowiednich krawędzi. *Drzewo rozpinające* (ang. *spanning tree*) grafu G jest to drzewo, które zawiera wszystkie wierzchołki grafu G i jest podgrafem grafu G. *Las* jest to niespójny graf nieskierowany i acykliczny, czy suma rozłącznych drzew.

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

4. Implementacja grafów

Przy implementowaniu grafów należy zdecydować, jak abstrakcyjne obiekty matematyczne z teorii grafów mają być realizowane w maszynie cyfrowej. Ponadto trzeba ustalić zestaw elementarnych operacji, które będą potrzebne do manipulowania grafami.

4.1. Implementacja obiektów z teorii grafów

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

Krawędź skierowana: Jest to instancja klasy Edge(s, t, w), przy czym należy podać wierzchołek początkowy s, wierzchołek końcowy t i opcjonalnie wagę w (domyślnie równa 1). Dla krawędzi edge mamy dostęp do wierzchołków i wagi za pomocą atrybutów: edge.source, edge.target, edge.weight. Krawędzie zorientowane są hashowalne i można je porównywać.

Krawędź nieskierowana: Dla wygody została także określona krawędź nieskierowana jako instancja klasy UndirectedEdge(s, t, w). Wewnętrznie wierzchołki są zawsze uporządkowane, czyli edge.source <= edge.target.
Klasa UndirectedEdge jest klasą podrzędną klasy Edge. Krawędzie niezorientowane są hashowalne i można je porównywać.

Graf: W implementacji macierzowej graf jest instancją klasy MatrixGraph(n), gdzie n jest liczbą wierzchołków. W implementacji słownikowej graf jest instancją klasy DictGraph(n), gdzie parametr n jest ignorowany i służy do zapewnienia kompatybilności z implementacją macierzową. Opcjonalnym parametrem klas jest directed=True dla grafów skierowanych (zorientowanych), oraz directed=False dla grafów nieskierowanych (niezorientowanych). Domyślnie tworzone są grafy nieskierowane (brak parametru directed).

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

Drzewo swobodne (niezorientowane): Drzewo swobodne jest po prostu grafem nieskierowanym. Wagę drzewa T można łatwo obliczyć jako
sum(edge.weight **for** edge **in** T.iteredges()).

Drugim sposobem reprezentacji drzewa jest zbiór krawędzi nieskierowanych. Ta reprezentacja naturalnie pojawia się w algorytmie Kruskala, a dla spójności jest także stosowana w innych algorytmach.

Drzewo ukorzenione (zorientowane): Drzewo ukorzenione powstaje naturalnie w wielu algorytmach, np. w algorytmie Prima lub Dijkstry. Wygodnym sposobem przedstawienia takiego drzewa jest słownik, gdzie kluczami są dzieci, a wartościami ich rodzice. Rodzicem korzenia jest wartość None. Kierunek (od korzenia lub do korzenia) oraz wagi krawędzi muszą być przechowywane osobno.

W implementacji macierzowej drzewo ukorzenione można także przedstawić za pomocą listy L, gdzie L[node] oznacza rodzica węzła node. Rodzicem korzenia może być -1.

Innym sposobem przechowywania drzewa ukorzenionego jest graf skierowany, który od razu zawiera wagi i kierunki krawędzi.

Las: Las drzew swobodnych będzie grafem nieskierowanym lub zbiorem krawędzi nieskierowanych, a las drzew ukorzenionych będzie słownikiem z wieloma korzeniami.

4.2. Sposoby reprezentacji grafów

Istnieją różne sposoby reprezentacji grafów: macierz sąsiedztwa 4.2.1, lista sąsiedztwa 4.2.2, reprezentacja przy pomocy słowników 4.2.3. Każda z tych reprezentacji ma wady i zalety. Dla grafów gęstych w większości przypadków najlepsze jest podejście z macierzą sąsiedztwa, dla grafów rzadkich można użyć słowników lub list.

4.2.1. Macierz sąsiedztwa

W reprezentacji macierzy sąsiedztwa graf jest zapamiętywany w tablicy dwuwymiarowej,

$$A_{ij} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1j} \\ w_{21} & w_{22} & \cdots & w_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \cdots & w_{ij} \end{bmatrix}, \tag{4.1}$$

gdzie $w_{ij} \neq 0$ oznacza istnienie krawędzi od i do j. W przypadku grafów z wagami w macierzy przechowywane są wagi krawędzi. Pojawia się przy tym niejednoznaczność dla krawędzi o wadze zero.

Dla grafów nieskierowanych można w macierzy przechowywać wartość logiczną True/False, która informuje o istnieniu krawędzi. Sprawdzenie istnienia krawędzi (odczytanie wagi) zajmuje czas $O(1)$, natomiast przejrzenie wszystkich krawędzi (niezależnie od tego, czy graf jest rzadki, czy gęsty, zajmuje czas $O(V^2)$. Złożoność pamięciowa tej reprezentacji to $O(V^2)$. Zatem tej reprezentacji rozsądnie jest używać tylko dla grafów gęstych lub w sytuacji, gdy spodziewamy się dużej liczby zapytań o istnienie krawędzi. Przy implementacji przy pomocy zwykłej tablicy tablic ujawnia się wada, że liczba

wierzchołków jest z góry ograniczona, tzn. zmiana liczby wierzchołków jest bardzo kosztowna.

4.2.2. Lista sąsiedztwa

Jest to sposób reprezentacji grafu polegający na tym, że z każdym wierzchołkiem skojarzona jest lista z wierzchołkami sąsiednimi. Złożoność operacji dostępu do dowolnej krawędzi to $O(V)$, bo wierzchołek może mieć $|V| - 1$ sąsiadów. Złożoność pamięciowa to $O(V + E)$. Jeden ze sposobów implementacji to tablica przechowująca wskaźniki do list sąsiedztwa. Wierzchołki są tutaj liczbami z zakresu od 0 do $|V|-1$. Przy takiej implementacji za pomocą zwykłej tablicy liczba wierzchołków jest z góry ograniczona. Wagi krawędzi mogą być przechowywane z wierzchołkami końcowymi krawędzi.

4.2.3. Reprezentacja słownikowa

Jest to sposób reprezentacji grafu będący połączeniem listy sąsiedztwa i macierzy sąsiedztwa. Graf jest przechowywany jako słownik słowników, co w większości przypadków jest tablicą tablic, ponieważ pythonowy słownik zaimplementowany jest jako tablica hashująca [5]. Zaletą użycia słownika jest to, że kluczami mogą być dowolne obiekty hashowalne, gdzie hash musi być immutable, tzn. za każdym razem musi zwracać tą samą wartość [6], zatem nie ma ograniczenia do używania tylko indeksów liczbowych. Z drugiej strony jednak podejście to jest również podobne do list sąsiedztwa, ponieważ nie przechowujemy wszystkich możliwych wierzchołków końcowych krawędzi, tylko te z istniejących krawędzi, co jest istotne dla grafów rzadkich. Zaletą użycia słownika zamiast listy jest krótszy czas dostępu do krawędzi, bo średni czas wynosi $O(1)$ [4]. Warto zaznaczyć, że słowniki są standardowym typem danych w Pythonie. Dla grafów gęstych tradycyjna macierz sąsiedztwa będzie szybsza niż reprezentacja słownikowa, ponieważ przy dostępie do elementów słownika wyliczana jest funkcja skrótu, tzw. hash, a jest to wolniejsze od prostego pobrania elementu z macierzy.

4.3. Interfejs grafów

Dla operacji na grafach zaproponowano interfejs przestawiony w tabeli 4.1. Implementacja słownikowa z klasą DictGraph oraz macierzowa z klasą MatrixGraph realizują ten interfejs. Przy implementacji wykorzystano abstrakcyją klasę bazową BaseGraph, w której interfejs został zapisany za pomocą metod abstrakcyjnych (dekorator @abstractmethod). Jest to zaawansowane narzędzie języka Python, które automatycznie weryfikuje interfejs w klasach klienta.

Na koniec podamy kilka uwag o zachowaniu najważniejszych metod. Ponowne dodanie wierzchołka jest ignorowane. Usunięcie wierzchołka powoduje automatycznie usunięcie wszystkich krawędzi wchodzących do wierzchołka i wychodzących z wierzchołka. W reprezentacji macierzowej wierzchołki faktycznie nie są usuwane.

Tabela 4.1. Interfejs grafów. Graph oznacza klasę DictGraph lub MatrixGraph, G i T są grafami, n jest liczbą całkowitą dodatnią.

Operacja	**Znaczenie**	**Metoda**
G = Graph(n)	tworzenie grafu nieskierowanego	__init__
G = Graph(n, directed=True)	tworzenie grafu skierowanego	__init__
G.is_directed()	czy graf skierowany	is_directed
G.v()	liczba wierzchołków	v
G.e()	liczba krawędzi	e
G.add_node(node)	dodanie wierzchołka	add_node
G.del_node(node)	usunięcie wierzchołka	del_node
G.has_node(node)	czy istnieje wierzchołek	has_node
G.add_edge(edge)	dodanie krawędzi	add_edge
G.del_edge(edge)	usunięcie krawędzi	del_edge
G.has_edge(edge)	czy istnieje krawędź	has_edge
G.weight(edge)	zwraca wagę krawędzi	weight
G.iternodes()	iterator wierzchołków	iternodes
G.iteredges()	iterator krawędzi	iteredges
G.iteradjacent(node)	iterator wierzchołków sąsiednich	iteradjacent
G.iteroutedges(node)	iterator krawędzi wychodzących	iteroutedges
G.iterinedges(node)	iterator krawędzi przychodzących	iterinedges
G.show()	wyświetlanie małych grafów	show
G == T	porównywanie grafów	__eq__
G != T	porównywanie grafów	__ne__

Dodanie krawędzi powoduje automatyczne utworzenie brakujących wierzchołków. Próba dodania pętli generuje wyjątek ValueError. Próba dodania krawędzi równoległej generuje wyjątek ValueError. Zmianę wagi krawędzi należy przeprowadzić przez usunięcie starej krawędzi i dodanie nowej, z nową wagą. Sprawdzenie istnienia krawędzi przez has_edge(edge) nie sprawdza wagi krawędzi w grafie. Przy usuwaniu krawędzi przez del_edge(edge) nie jest sprawdzana waga krawędzi w grafie.

5. Algorytmy i ich implementacje

Algorytmy przedstawione w tym rozdziale na ogół mają jednolity sposób uruchamiania. Przykładowa sesja interaktywna przedstawia sposób korzystania z algorytmów.

```
>>> from edges import Edge
>>> from dictgraph import DictGraph
>>> G = DictGraph()
>>> G.is_directed()
False
>>> G.add_edge(Edge('A', 'B', 5))
>>> G.add_edge(Edge('A', 'C', 7))
>>> print G.v(), G.e()    # liczby wierzcholkow i krawedzi
3 2
>>> list(G.iternodes())
['A', 'C', 'B']
# Typowe wykorzystanie algorytmu Foo z modulu foo.
>>> from foo import Foo
>>> result = Foo.execute(G)
```

5.1. Przeszukiwanie grafów

Przez przeszukiwanie grafów rozumie się systematyczne przechodzenie wzdłuż jego krawędzi w celu odwiedzenia jego wierzchołków [8]. Przeszukiwanie wykonujemy w celu zbierania informacji o strukturze grafu, w celu wykonania pewnych operacji na wierzchołkach w określonym porządku, itp. Pewne algorytmy są modyfikacjami dwóch podstawowych algorytmów przeszukiwania: przeszukiwania wszerz (ang. *breadth-first search*, BFS) i przeszukiwania w głąb (ang. *depth-first search*, DFS).

5.1.1. Przeszukiwanie wszerz

Dane wejściowe: Dowolny graf.

Problem: Przeszukiwanie grafu wszerz.

Opis algorytmu: Algorytm BFS może służyć do przeglądnięcia wszystkich wierzchołków grafu lub tylko tych osiągalnych z zadanego źródła *root*. Działanie algorytmu polega na tym, że startuje z wybranego wierzchołka, wstawia sąsiadujące z nim nieodwiedzone jeszcze wierzchołki do kolejki (tu może wykonywać jakieś działanie na wierzchołku), oznacza je jako odwiedzone, po przetworzeniu wyciąga następny wierzchołek itd., aż wszystkie wierzchołki

zostaną odwiedzone tj. kolejka będzie pusta. Wierzchołki przetwarzane są *warstwowo*, tzn. najpierw odległe o jedną krawędź, później o dwie, trzy, itd. Podczas działania algorytmu powstaje tzw. *drzewo przeszukiwań wszerz* o korzeniu w *root*, odpowiadające kolejności odwiedzania wierzchołków. Drzewo to ma taką właściwość, że wierzchołki *root* i jakiś inny osiągalny z niego wierzchołek połączone są ścieżką o najmniejszej liczbie krawędzi. Przy założeniu, że wszystkie wagi krawędzi są jednakowe, wyznaczona została najkrótsza scieżka w źródle *root* pomiędzy pozostałymi wierzchołkami (rozdział 5.3).

Złożoność czasowa: Złożoność czasowa algorytmu wynosi $O(V+E)$. Czas $O(V)$ zajmuje początkowa inicjalizacja. W czasie $O(E)$ algorytm przechodzi po wszystkich sąsiadach dla każdego wierzchołka (reprezentacja słownikowa lub list sąsiedztwa). Algorytm używa kolejki, ale wkładanie i wyciąganie kosztuje $O(1)$, zatem nie wpływa to na złożoność całkowitą.

Uwagi: Algorytm BFS został zaimplementowany w klasie BFS. W analizie algorytmu BFS przydatne jest kolorowanie wierzchołków: wierzchołki białe są nieodwiedzone; wierzchołki szare są napotkane, ale nie przetworzone; wierzchołki czarne są napotkane i lista ich sąsiadów jest przetworzona. Wierzchołki przebywające w kolejce są szare. Drzewo przeszukiwań wszerz przechowywane jest w słowniku parent. Listę L kolejno odwiedzanych wierzchołków grafu G możemy uzyskać za pomocą polecenia
L=[] ; BFS.execute(G, root, on_visit=**lambda** v: L.append(v)).

Listing 5.1. Moduł bfs.py

```
#! /usr/bin/python
# Kod implementujacy pseudokod ze str. 595 CLRS.

from utils.Enum import Enum
from Queue import Queue

class BFS(object):

    Colors = Enum(["WHITE", "GRAY", "BLACK"])

    @staticmethod
    def execute(graph, source=None, on_visit=None):
        visited = set()
        if source is None:
            for node in graph.iternodes():
                if not node in visited:
                    BFS._visit(graph, node, visited, on_visit)
        else:
            BFS._visit(graph, source, visited, on_visit)

    @staticmethod
    def _visit(graph, node, visited, on_visit=None):
        visited.add(node)
```

```
        queue = Queue()
        queue.put(node)
        if on_visit:  # when queue.put
            on_visit(node)
        while not queue.empty():
            u = queue.get()
            for v in graph.iteradjacent(u):
                if not v in visited:
                    visited.add(v)
                    queue.put(v)
                    if on_visit:  # when queue.put
                        on_visit(v)

    @staticmethod
    def _init(graph, color, distance, parent, source):
        for node in graph.iternodes():
            color[node] = BFS.Colors.WHITE
            distance[node] = float("inf")
            parent[node] = None
        color[source] = BFS.Colors.GRAY
        distance[source] = 0

    @staticmethod
    def execute_extended(graph, source, on_visit=None):
        color = {}
        distance = {}  # distance between each node and source
        parent = {}
        BFS._init(graph, color, distance, parent, source)
        queue = Queue()
        queue.put(source)
        if on_visit:  # when queue.put
            on_visit(source)
        while not queue.empty():
            u = queue.get()
            for v in graph.iteradjacent(u):
                if color[v] == BFS.Colors.WHITE:
                    color[v] = BFS.Colors.GRAY
                    distance[v] = distance[u] + 1
                    parent[v] = u
                    queue.put(v)
                    if on_visit:  # when queue.put
                        on_visit(v)
            color[u] = BFS.Colors.BLACK
        return distance, parent   # similar to Dijkstra
```

5.1.2. Przeszukiwanie wgłąb

Dane wejściowe: Dowolny graf.

Problem: Przeszukiwanie grafu wgłąb.

Opis algorytmu: Algorytm DFS może służyć do przeglądnięcia wszystkich wierzchołków grafu lub tylko tych osiągalnych z zadanego źródła *root*. Jak nazwa wskazuje, algorytm zagłębia się w hierarchii wierzchołków tak głęboko jak to możliwe, czyli przechodzi po krawędzi wychodzącej z ostatnio odwiedzonego wierzchołka, pozostawiając pozostałe wychodzące krawędzie dotąd, aż bardziej wgłąb dojść się już nie da, po czym dopiero wraca do poprzednich krawędzi i je przetwarza w analogiczny sposób dotąd, aż wszystkie wierzchołki osiągalne ze źródła *root* zostaną odwiedzone.

Jeśli jakieś wierzchołki zostały nieodwiedzone, a algorytm się skończył, to oznacza, że graf jest niespójny. Jeżeli chcemy przetworzyć wszystkie wierzchołki grafu, to należy wybrać nowe źródło (z pozostałych nieodwiedzonych wierzchołków) i kontynuować powyższy algorytm.

Złożoność czasowa: Złożoność czasowa wynosi $O(V + E)$. Czas $O(V)$ zajmuje początkowa inicjalizacja. W czasie $O(E)$ algorytm przechodzi po wszystkich sąsiadach dla każdego wierzchołka (reprezentacja słownikowa lub list sąsiedztwa).

Uwagi: Algorytm DFS można zrealizować rekurencyjnie, albo za pomocą stosu. Oba te podejścia różnią się kolejnością, w jakiej kończymy przetwarzanie danego wierzchołka. Wydaje się, że większe znaczenie ma realizacja rekurencyjna, przykładowo przy użyciu rekurencyjnego algorytmu DFS można posortować dag topologicznie.

Listing 5.2. Moduł dfs.py

```
#! /usr/bin/python
# Implementacja na bazie pseudokodu z str. 604 CRLS.


class DFS(object):

    @staticmethod
    def execute(graph, source=None, preorder_visit=None,
                postorder_visit=None):
        visited = set()
        if source is None:
            for node in graph.iternodes():
                if not node in visited:
                    DFS._visit(graph, node, visited,
                               preorder_visit, postorder_visit)
        else:
```

```
            DFS._visit(graph, source, visited,
                       preorder_visit, postorder_visit)

    @staticmethod
    def _visit(graph, node, visited, preorder_visit=None,
               postorder_visit=None):
        visited.add(node)
        if preorder_visit:
            preorder_visit(node)
        for v in graph.iteradjacent(node):
            if not v in visited:
                DFS._visit(graph, v, visited,
                           preorder_visit, postorder_visit)
        if postorder_visit:
            postorder_visit(node)
```

5.1.3. Sortowanie topologiczne

Sortowanie topologiczne polega na zanurzeniu porządku częściowego wierzchołków w porządku liniowym [9]. Graf skierowany jest posortowany topologicznie wtedy, gdy dla każdej krawędzi Edge(s, t) wierzchołek s występuje przed wierzchołkiem t. Graf musi być acykliczny, ponieważ w przeciwnym wypadku takie uporządkowanie nie jest możliwe. Dla danego grafu może wystąpić kilka wersji poprawnego uporządkowania.

Dane wejściowe: Graf skierowany acykliczny (dag).

Problem: Sortowanie topologiczne.

Opis algorytmu: Wykorzystujemy algorytm DFS do znalezienia listy wierzchołków odwiedzonych w kolejności *postorder*, czyli w momencie powrotu z wywołania wgłąb. Po odwróceniu listy otrzymujemy poprawną kolejność wierzchołków w sortowaniu topologicznym.

Złożoność czasowa: Złożoność czasowa wynosi $O(V + E)$, ponieważ jest to złożoność algorytmu DFS.

Uwagi: Aby uzyskać poprawną kolejność wierzchołków należy uruchomić rekurencyjną wersję algorymu DFS.

Listing 5.3. Moduł topologicalsort.py

```
#! /usr/bin/python
# Kod inspirowany pseudokodem algorytmu ze str. 613 CLRS.

from algorithms.dfs import DFS

class TopologicalSort(object):
```

```
    @staticmethod
    def execute(graph):
        sorted_nodes = []
        DFS.execute(graph, postorder_visit=lambda node:
                    sorted_nodes.append(node))
        sorted_nodes.reverse()
        return sorted_nodes
```

5.2. Minimalne drzewo rozpinające

Minimalne drzewo rozpinające (ang. *Minimum Spanning Tree*, MST) jest to takie drzewo rozpinające, że nie istnieje inne drzewo rozpięte na tym grafie o mniejszej wadze. Dla danego grafu może istnieć wiele różnych minimalnych drzew rozpinających. Inaczej mówiąc, MST jest to acykliczny podzbiór źródłowego grafu łączący wszystkie wierzchołki krawędziami o minimalnej sumie wag. MST można znaleźć za pomocą następujących algorytmów zachłannych: Boruvki (rozdział 5.2.1), Prima (rozdział 5.2.2), Kruskala (rozdział 5.2.3). Strategia zachłanna polega na tym, iż w każdym kroku wykonujemy najlepszy wybór w danej chwili, w tym przypadku powiększamy drzewo o dodatkową krawędź. W ogólności algorytmy zachłanne nie gwarantują znalezienia globalnie optymalnego rozwiązania. W przypadku MST odpowiednie twierdzenia gwarantują poprawność rozwiązania.

Zanim przejdziemy do opisu algorytmów, przedstawimy ogólne porównanie ich wydajności, tzn. czasy znajdowania MST. Wykresy 5.1 i 5.2 dotyczą implementacji grafów z klasą DictGraph natomiast wykresy 5.3 i 5.4 dotyczą implementacji grafów z klasą MatrixGraph. Pierwszy wniosek jest taki, że zgodnie z oczekiwaniami implementacja Prima $O(VE)$ jest bardzo niewydajna w każdym przypadku. Drugi wniosek to dobra wydajność algorytmu Kruskala dla grafów rzadkich. Trzeci wniosek to dobra wydajność algorytmu Prima $O(V^2)$ dla grafów gęstych.

5.2.1. Algorytm Borůvki

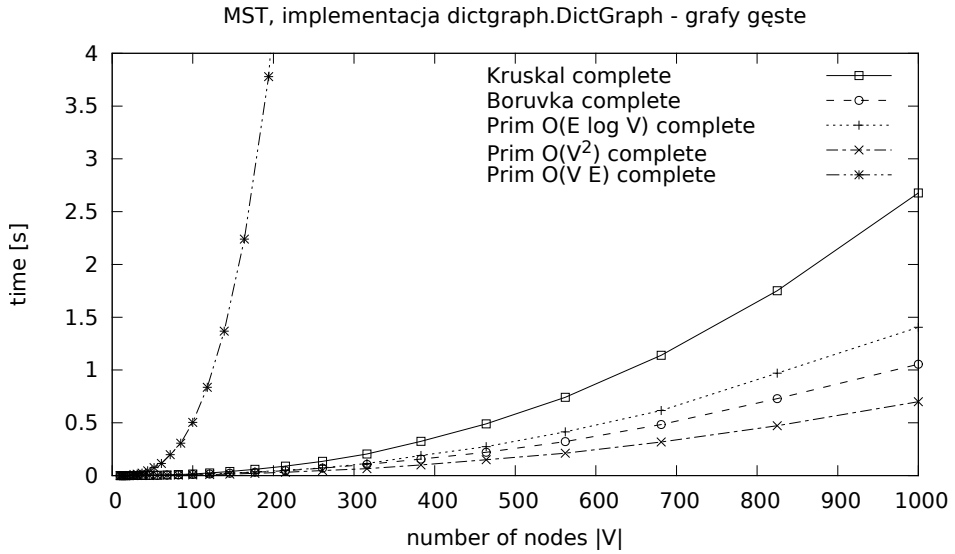
Algorytm został po raz pierwszy opublikowany w roku 1926 przez Otokara Borůvkę. Następnie algorytm był kilka razy odkrywany na nowo, m. in. przez Sollina w 1965 roku, stąd inna nazwa to algorytm Sollina.

Dane wejściowe: Graf ważony nieskierowany i spójny; wszystkie wagi krawędzi są różne.

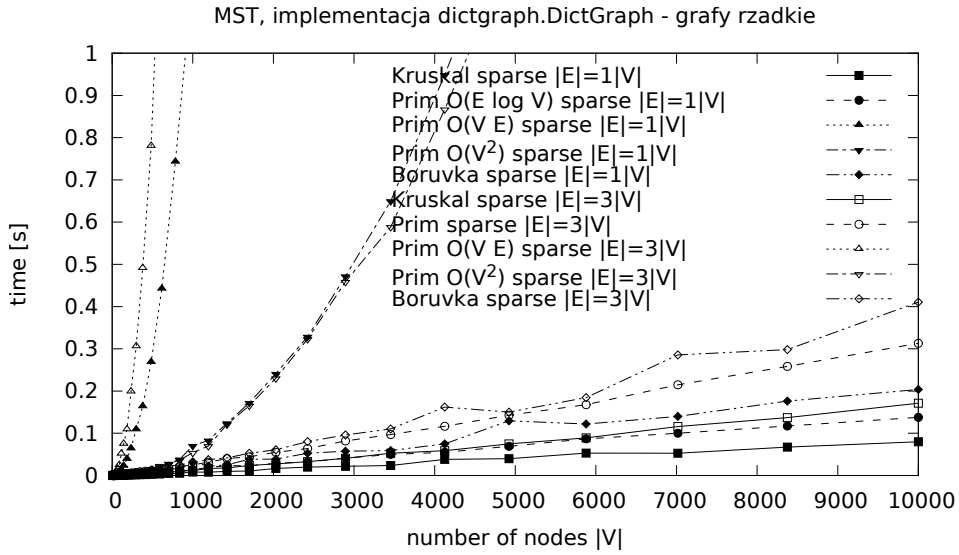
Problem: Wyznaczenie minimalnego drzewa rozpinającego.

Opis algorytmu: Algorytm Borůvki wyznacza MST poprzez zastosowanie "ściągania" krawędzi. Inicjalizacja polega na stworzeniu $|V|$ poddrzew, po jednym dla każdego wierzchołka. "Ściąganie" krawędzi polega na tym, że z danego poddrzewa wybierana jest minimalna krawędź prowadząca do najbliższego sąsiada z grafu, a wierzchołki, które łączyła ta krawędź scalamy

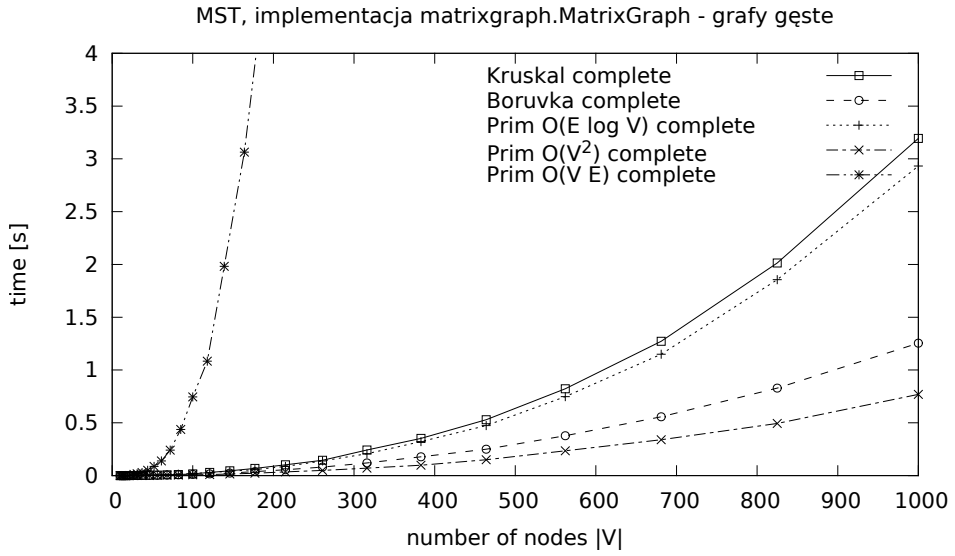
Rysunek 5.1. Porównanie wydajności algorytmów obliczających MST dla grafu dictgraph - grafy gęste.



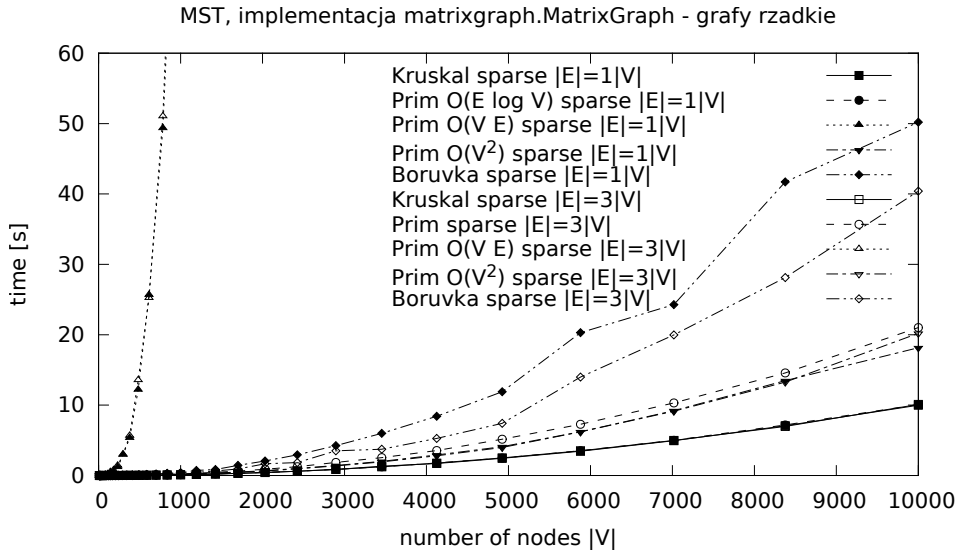
Rysunek 5.2. Porównanie wydajności algorytmów obliczających MST dla grafu dictgraph - grafy rzadkie.



Rysunek 5.3. Porównanie wydajności algorytmów obliczających MST dla grafu matrixgraph - grafy gęste.



Rysunek 5.4. Porównanie wydajności algorytmów obliczających MST dla grafu matrixgraph - grafy rzadkie.



w jeden nowy wierzchołek. Jeśli przed scaleniem z obydwóch wierzchołków istniała jakaś krawędź prowadząca do trzeciego wspólnego wierzchołka, to przy scalaniu pojawia się duplikat krawędzi. W takim przypadku wybiera się krawędź o korzystniejszej wadze. Proces scalania powtarzany jest dotąd, aż drzewo rozpinające pokryje wszystkie wierzchołki, tzn. po scalaniu zostanie tylko jedno podrzewo. W zasadzie "ściąganie" może być osiągnięte przez modyfikację grafu, ale jest to trochę skomplikowane w implementacji.

Innym podejściem może być zastosowanie struktury zbiorów rozłącznych, zamiast wspomnianego wcześniej "ściągania" krawędzi. Algorytm Union-Find na początku tworzy zbiory (komponenty) składające się z pojedyńczych wierzchołków, a reprezentantem komponentu jest taki wierzchołek. Operacja Find, dla podanego wierzchołka, zwraca reprezentanta komponentu. Jeśli najkrótsza krawędź łączy różne komponenty, to można dodać ją do MST, bo nie utworzy cyklu. Dodatkowo następuje wtedy scalenie tych dwóch komponentów w jeden, przy pomocy operacji Union. Algorytm kończy się z chwilą, gdy wszystkie wierzchołki należą do jednego komponentu.

Złożoność czasowa: Po każdym kroku scalania komponentów liczba komponentów zmaleje co najmniej dwukrotnie, zatem liczba tych kroków wynosi $O(\log V)$. W każdym kroku przeglądamy krawędzie w czasie $O(E)$, aby wybrać krawędzie minimalne (reprezentacja słownikowa lub list sąsiedztwa). Złożoność algorytmu wynosi więc $O(E \log V)$. Wyniki eksperymentów komputerowych przedstawione są na wykresach 5.5, 5.6, 5.7.

Uwagi: Nasza implementacja działa także dla grafu niespójnego, wtedy wyznaczany jest las minimalnych drzew rozpinających poszczególnych komponentów. Jeżeli graf ma powtarzające się wagi krawędzi, to można każdej krawędzi przypisać unikalny indeks i w przypadku równych wag można wybrać krawędź z mniejszym indeksem.

Listing 5.4. Moduł boruvka.py

```
#! /usr/bin/python
# Kod implementujacy pseudokod ze strony
# http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/mst

from utils.disjointset import DisjointSet
from model.edge import Edge
from model.undirectededge import UndirectedEdge

class Boruvka(object):

    @staticmethod
    def execute(graph):
        mst = set()
        disjointset = DisjointSet()
        forest = set()
        for node in graph.iternodes():
            disjointset.create(node)  # create set for each node
```

```
            forest.add(node)  # initialize forest with one-node-trees
        dummy_edge = Edge(None, None, float("inf"))
        new_len = len(forest)
        old_len = new_len + 1
        while old_len > new_len:
            old_len = new_len
            minimal_edges = dict((node, dummy_edge) for node in forest)
            # finding the cheapest edges
            for edge in graph.iteredges():
                if edge is dummy_edge: # a disconnected graph
                    continue
                source = disjointset.find(edge.source)
                target = disjointset.find(edge.target)
                if source != target:  # different components
                    if edge.weight < minimal_edges[source].weight:
                        minimal_edges[source] = edge
                    if edge.weight < minimal_edges[target].weight:
                        minimal_edges[target] = edge
            # connecting components
            forest = set()
            for edge in minimal_edges.itervalues():
                source = disjointset.find(edge.source)
                target = disjointset.find(edge.target)
                if source != target:
                    disjointset.union(source, target)
                    forest.add(source)
                    mst.add(UndirectedEdge(edge.source, edge.target, edge.weight))
            forest = set(disjointset.find(node) for node in forest)
            new_len = len(forest)
            if new_len == 1:   # a connected graph
                break
        return mst
```

5.2.2. Algorytm Prima

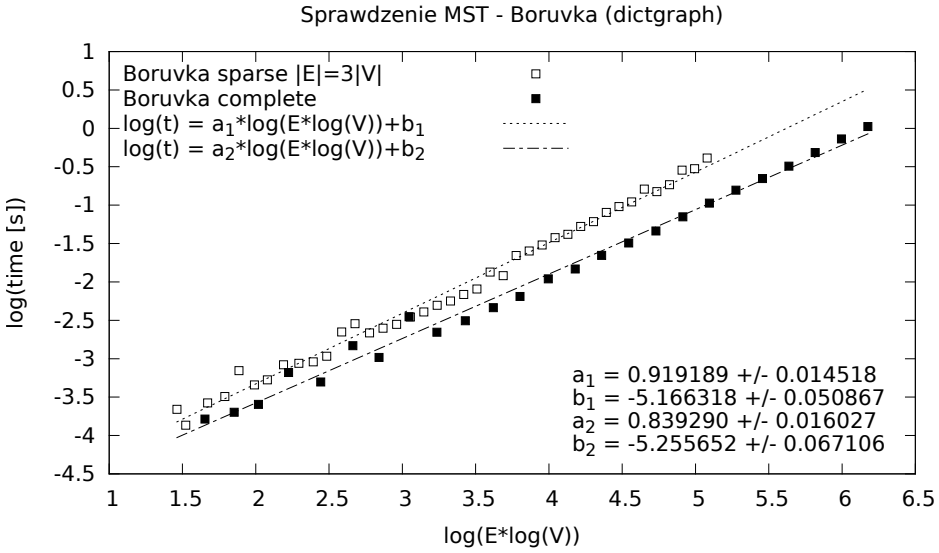
Algorytm Prima działa podobnie jak algorytm Dijkstry (rozdział 5.3.3) przy szukaniu najkrótszej ścieżki w grafie pomiędzy dwoma wierzchołkami. Cechą charakterystyczną algorytmu Prima jest budowanie jednego, rosnącego drzewa, które staje się MST.

Dane wejściowe: Graf ważony nieskierowany spójny.

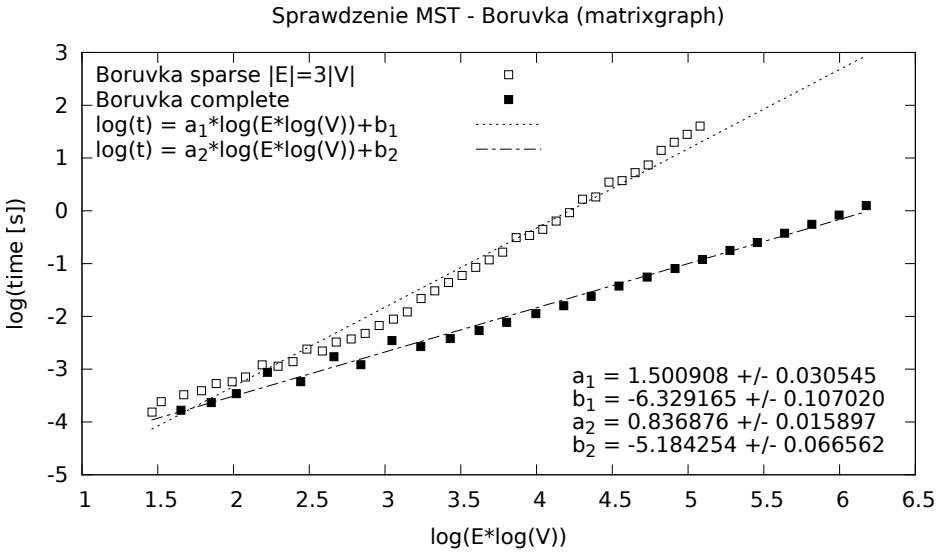
Problem: Wyznaczenie minimalnego drzewa rozpinającego.

Opis algorytmu: Algorytm Prima startuje z jakiegoś losowego wierzchołka w grafie, przechodzi po grafie, dołączając kolejne wierzchołki do wyliczanego drzewa MST, dopóki nie utworzy całego drzewa. Każdy krok dołącza najlżejszą krawędź, jaka łączy wyizolowany wierzchołek, czyli taki, który nie

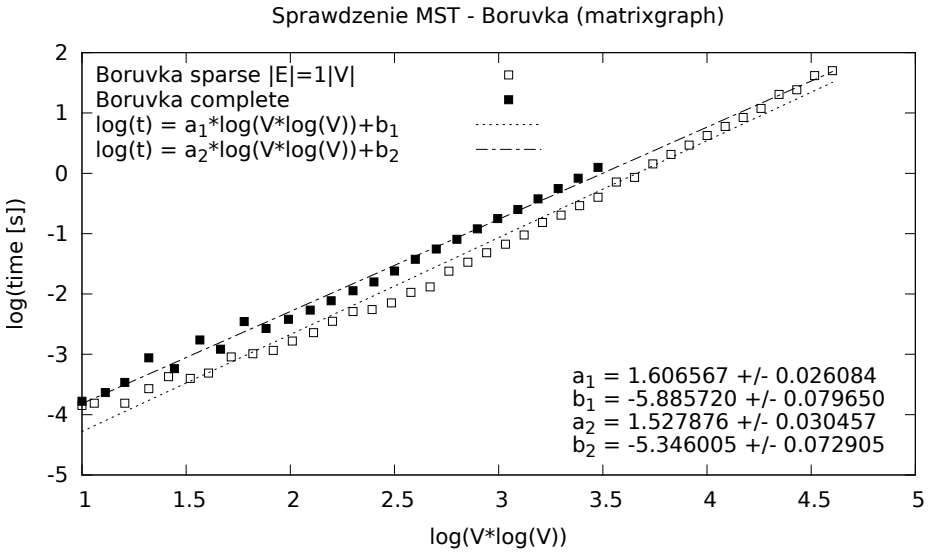
Rysunek 5.5. Test złożoności obliczeniowej algorytmu Boruvki w wersji $O(E \log V)$ dla grafu dictgraph. Współczynniki a bliskie jedynki potwierdzają teoretyczną wydajność tej implementacji.



Rysunek 5.6. Test złożoności obliczeniowej algorytmu Boruvki w wersji $O(E \log V)$ dla grafu matrixgraph. Zależności są różne dla różnej gęstości grafów, co wymaga wyjaśnienia.



Rysunek 5.7. Test złożoności obliczeniowej algorytmu Boruvki w wersji $O(E \log V)$ dla grafu matrixgraph $|E| = |V|$. Widać, że w reprezentacji macierzowej kluczowa jest liczba wierzchołków grafu, a tylko w niewielkim stopniu korzystamy z małej liczby krawędzi.



został jeszcze dodany do drzewa, w przeciwnym wypadku powstał by cykl. Ta strategia zachłanna wyznacza poprawny wynik. Do efektywnego wyznaczania kolejnych krawędzi można użyć kolejki priorytetowej. Podczas pracy w kolejce są tylko takie wierzchołki, które jeszcze nie zostały dodane do drzewa. Przechowywana jest też najmniejsza, znana w danym momencie, waga krawędzi wychodzącej z danego wierzchołka. Dla grafów gęstych, zamiast kolejki priorytetowej, korzystniejsze jest użycie zwykłej kolejki opartej na tablicy (w pracy została wybrana tablica hashująca).

Złożoność czasowa: Złożoność dla implementacji z kolejką priorytetową wynosi $O(E \log V)$, ponieważ algorytm przechodzi po wszystkich krawędziach, a wkładanie do kolejki priorytetowej nowej wartości kosztuje $O(\log V)$. Eksperymenty komputerowe zobrazowano na wykresach 5.8 i 5.9.

Złożoność dla implementacji ze zwykłą kolejką to $O(V^2)$, ponieważ algorytm przechodzi po wszystkich wierzchołkach, a dla każdego z nich wyszukuje najmniejszą wagę w czasie $O(V)$. Wyniki eksperymentów przedstawiają wykresy 5.12 i 5.11.

Dla pełnego opisu algorytmu Prima została zaimplementowana wersja trywialna, działająca w czasie $O(VE)$. Tutaj algorytm wykonuje $|V|-1$ kroków, a w każdym kroku przebiega wszystkie krawędzie, aby znaleźć najlżejszą, łączącą drzewo z pewnym osobnym wierzchołkiem. Rezultaty eksperymentów przedstawiają wykresy 5.13 i 5.14.

Uwagi: Nasze implementacje pozwalają na wskazanie wierzchołka, od którego zacznie się budowa MST. Dla grafu niespójnego algorytm wygeneruje MST dla pewnej spójnej składowej grafu. Asymptotyczny czas działania algorytmu Prima można teoretycznie poprawić do $O(E + V \log V)$, stosując

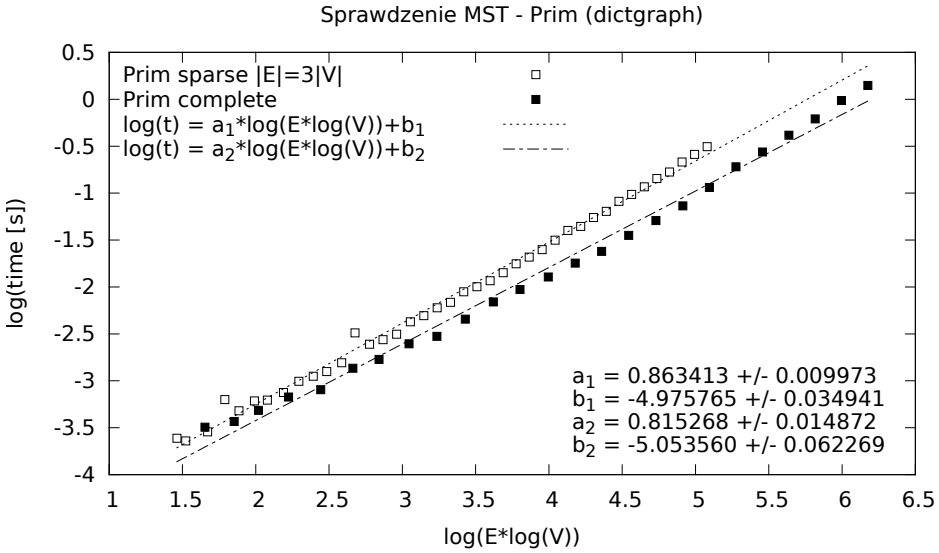
kopce Fibonacciego. Jednak w praktyce raczej się tego nie robi, ze względu na skomplikowaną implementację takich kopców.

Listing 5.5. Moduł prim.py

```
#! /usr/bin/python
# Kod implementujacy pseudokod ze str. 634 CLRS.

from model.undirectededge import UndirectedEdge
from model.edge import Edge
from Queue import PriorityQueue


class Prim(object):
    """ Algorytm inspirowany implementacja z ksiazki
    Python Algorithms: Mastering Basic Algorithms in the Python
    Language, 2010 by Magnus Lie Hetland."""

    @staticmethod
    def execute(graph, root=None):
        if root is None:
            root = graph.iternodes().next()  # get first random element
        parents = {}
        pq = PriorityQueue()
        pq.put((0, None, root))
        mst = set()
        while not pq.empty():
            weight, parent, node = pq.get()
            if node in parents:
                continue
            parents[node] = parent
            if not parent is None:
                mst.add(UndirectedEdge(parent, node, weight))
            for edge in graph.iteroutedges(node):
                pq.put((edge.weight, edge.source, edge.target))
        return mst

    @staticmethod
    def execute2(graph, root=None):
        """Prim's algorithm for finding MST in O(E log V) time."""
        if root is None:
            root = graph.iternodes().next()  # get first random element
        minimum_weights = dict(((node, float("inf"))
                                for node in graph.iternodes()))
        parents = {}
        minimum_weights[root] = 0
        pq = PriorityQueue()
        pq.put((0, root))
        mst = set()
```

```
        visited = set()
        while not pq.empty():
            weight, source = pq.get()
            visited.add(source)
            for edge in graph.iteroutedges(source):
                if (not edge.target in visited and
                        edge.weight < minimum_weights[edge.target]):
                    minimum_weights[edge.target] = edge.weight
                    pq.put((edge.weight, edge.target))
                    parents[edge.target] = edge.source
        for source, target in parents.iteritems():
            mst.add(UndirectedEdge(source, target,
                    minimum_weights[source]))
        return mst

    @staticmethod
    def execute_best_with_complete_graph(graph, root=None):
        """Prim's algorithm for finding MST in O(V**2) time."""
        if root is None:
            root = graph.iternodes().next()  # get first random element
        mst = set()
        queue = {}           # set can be used here
        minimum_weights = {}
        parents = {}
        for node in graph.iternodes():
            queue[node] = None
            minimum_weights[node] = float("inf")
        minimum_weights[root] = 0
        while queue:
            source = min(queue, key=minimum_weights.get)
            del queue[source]
            for edge in graph.iteroutedges(source):
                if (edge.target in queue and
                        edge.weight < minimum_weights[edge.target]):
                    minimum_weights[edge.target] = edge.weight
                    parents[edge.target] = edge.source
        for source, target in parents.iteritems():
            mst.add(UndirectedEdge(source, target,
                    minimum_weights[source]))
        return mst

    @staticmethod
    def execute_trivial(graph, source=None):
        """Prim's algorithm for finding MST in O(V*E) time."""
        in_mst = dict((node, False) for node in graph.iternodes())
        if source is None:   # get first random node
            source = graph.iternodes().next()
        in_mst[source] = True
```

Rysunek 5.8. Test złożoności obliczeniowej algorytmu Prima w wersji $O(E \log V)$ dla grafu dictgraph. Wydajność potwierdzona.




```
mst = set()
for step in xrange(graph.v()-1):      # |V|-1 times
    # finding min edge, O(E) time
    min_edge = min(edge for edge in graph.iteredges()
                   if in_mst[edge.source] != in_mst[edge.target])
    mst.add(UndirectedEdge(min_edge.source,
            min_edge.target, min_edge.weight))
    in_mst[min_edge.source] = True
    in_mst[min_edge.target] = True
return mst
```

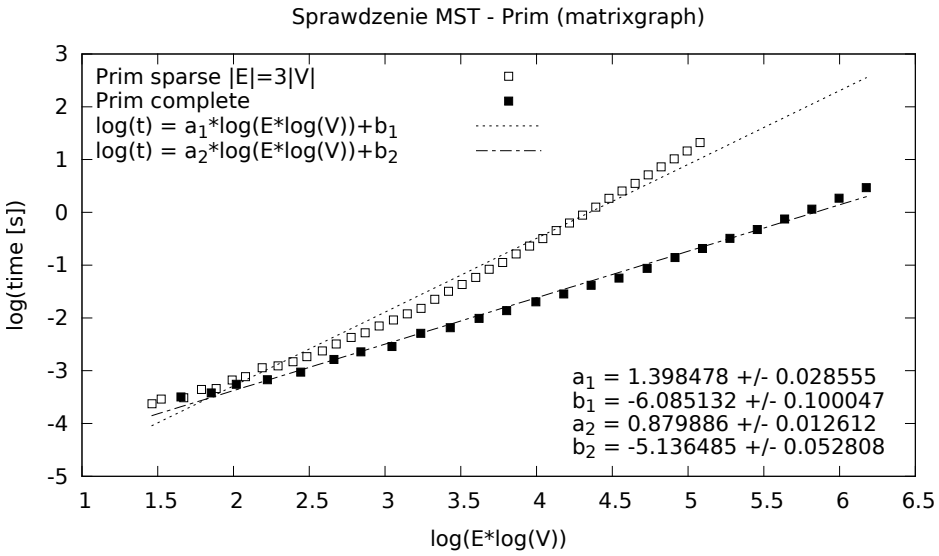
5.2.3. Algorytm Kruskala

Dane wejściowe: Graf ważony nieskierowany.

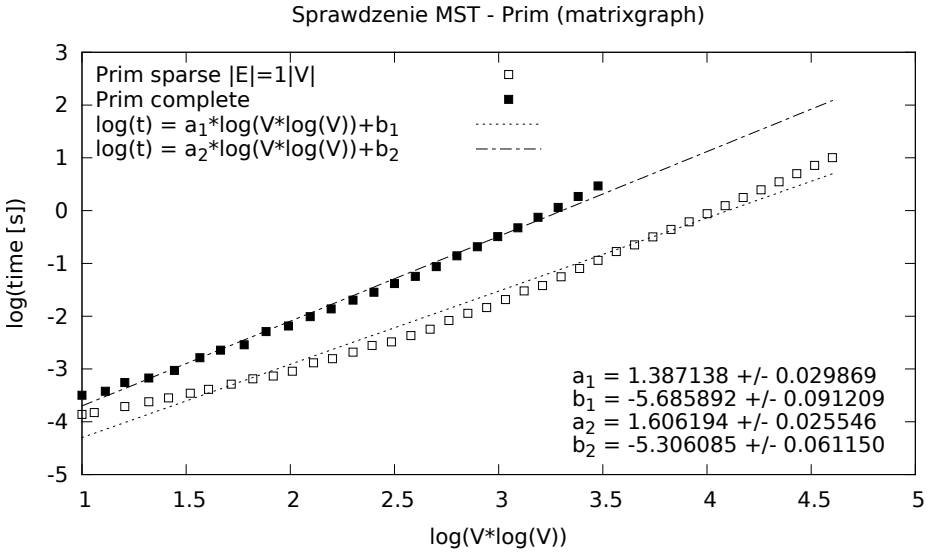
Problem: Wyznaczenie minimalnego drzewa rozpinającego.

Opis algorytmu: Algorytm Kruskala (1956) przechodzi po krawędziach grafu w kolejności rosnącej (po wagach), dlatego wcześniej potrzebne jest sortowanie krawędzi. W każdym kroku mając minimalną krawędź, próbuje dodać do MST, jeśli nie spowoduje to stworzenia cyklu. Inicjalizacja polega na stworzeniu $|V|$ podrzew, po jednym dla każdego wierzchołka. Przy pomocy struktury zbiorów rozłącznych sprawdzana jest przynależność krawędzi Edge(s, t, w) do dwóch podrzew - jednego z wierzchołkiem s, drugiego t. Jeśli oba wierzchołki krawędzi należą do tego samego podrzewa, to odrzucamy taką krawędź. Jeśli wierzchołki należą do różnych poddrzew, to scalamy oba

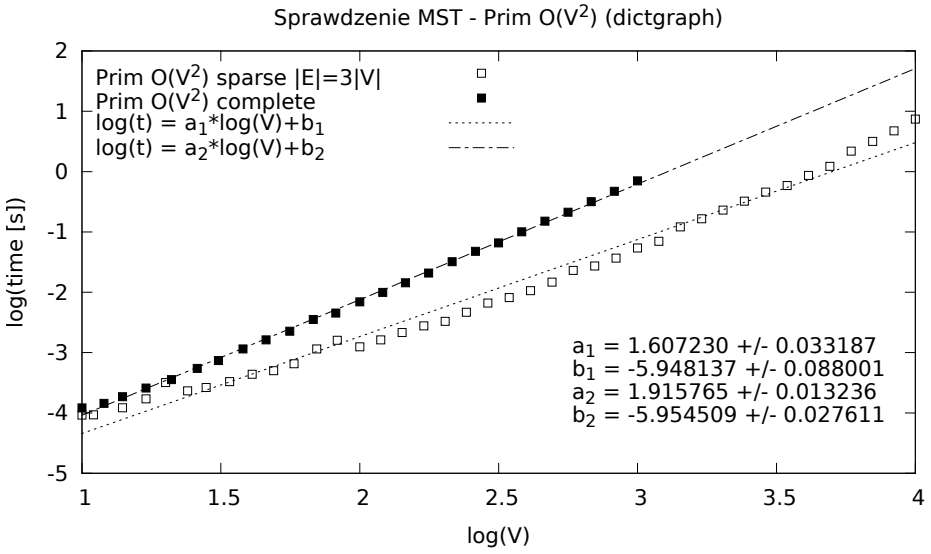
Rysunek 5.9. Test złożoności obliczeniowej algorytmu Prima w wersji $O(E \log V)$ dla grafu matrixgraph. Zależności zależą od gęstości grafu.



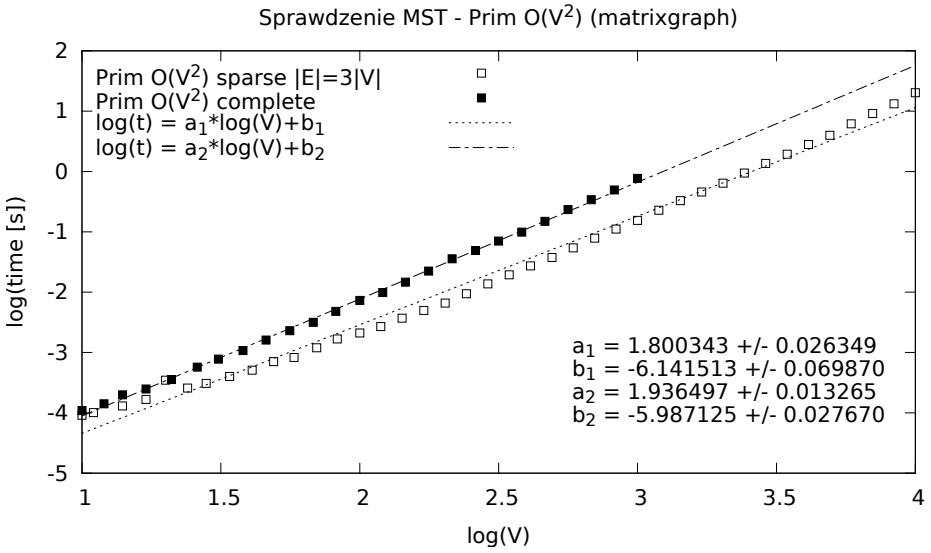
Rysunek 5.10. Test złożoności obliczeniowej algorytmu Prima w wersji $O(E \log V)$ dla grafu matrixgraph $|E| = |V|$. Istotna jest liczba wierzchołków, a nie krawędzi.



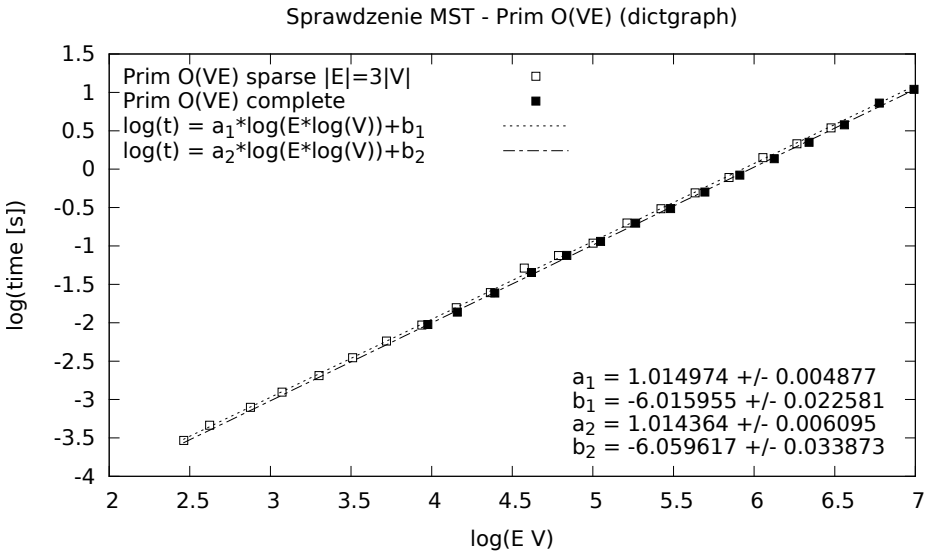
Rysunek 5.11. Test złożoności obliczeniowej algorytmu Prima w wersji $O(V^2)$ dla grafu dictgraph. Współczynniki a bliskie 2 potwierdzają teorię.



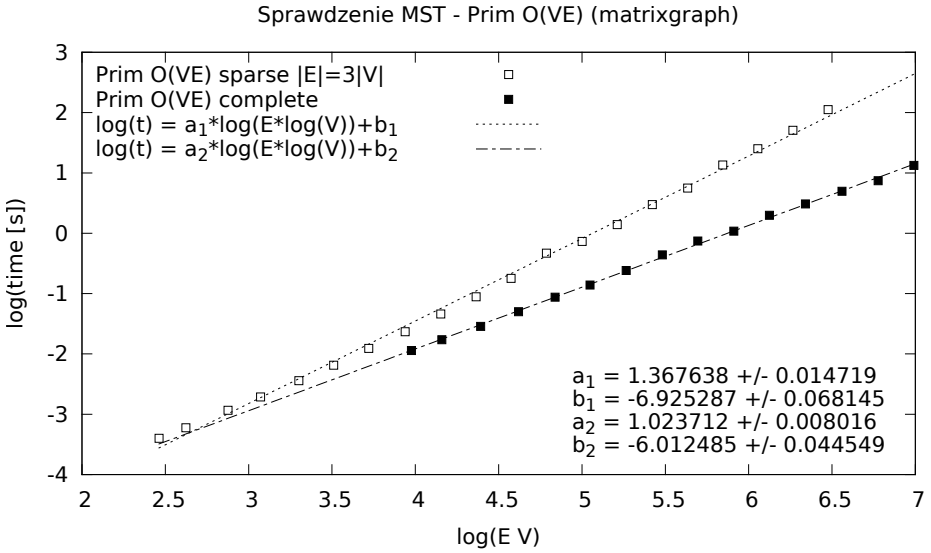
Rysunek 5.12. Test złożoności obliczeniowej algorytmu Prima w wersji $O(V^2)$ dla grafu matrixgraph. Współczynniki a bliskie 2 potwierdzają teorię.



Rysunek 5.13. Test złożoności obliczeniowej algorytmu Prima w wersji $O(VE)$ dla grafu dictgraph. Piękna zgodność przewidywań z praktyką.



Rysunek 5.14. Test złożoności obliczeniowej algorytmu Prima w wersji $O(VE)$ dla grafu matrixgraph. Widać, że reprezentacja macierzowa gorzej pracuje dla grafów rzadkich.



podrzewa. Proces kontynuujemy, aż przejdziemy przez wszystkie krawędzie grafu.

Złożoność czasowa: Złożoność pierwszej fazy algorytmu (sortowanie krawędzi) wynosi $O(E \log E) = O(E \log V)$, ponieważ $|E| < |V|^2$. Przechodzenie po krawędziach w drugiej częsci algorytmu kosztuje $O(E\alpha(V))$, gdzie $\alpha(V)$ jest funkcją bardzo wolno rosnącą [8], wynikającą z operacji na zbiorach rozłącznych. Zatem całkowita złożoność algorytmu to $O(E \log V)$.

Jeżeli krawędzie są już na wejściu posortowane wg wag lub jeśli wagi krawędzi pozwalają na użycie szybszych algorytmów sortowania (np. sortowanie przez zliczanie lub sortowanie pozycyjne), to algorytm może działać w czasie bliskim $O(E\alpha(V))$.

Uwagi: Zamiast od razu sortować wszystkie krawędzie, można skorzystać z kolejki priorytetowej do częściowego uporządkowania krawędzi. Dla grafu niespójnego algorytm wyznacza las minimalnych drzew rozpinających poszczególnych składowych spójnych. Wyniki eksperymentów przedstawiają wykresy 5.15, 5.16 i 5.17.

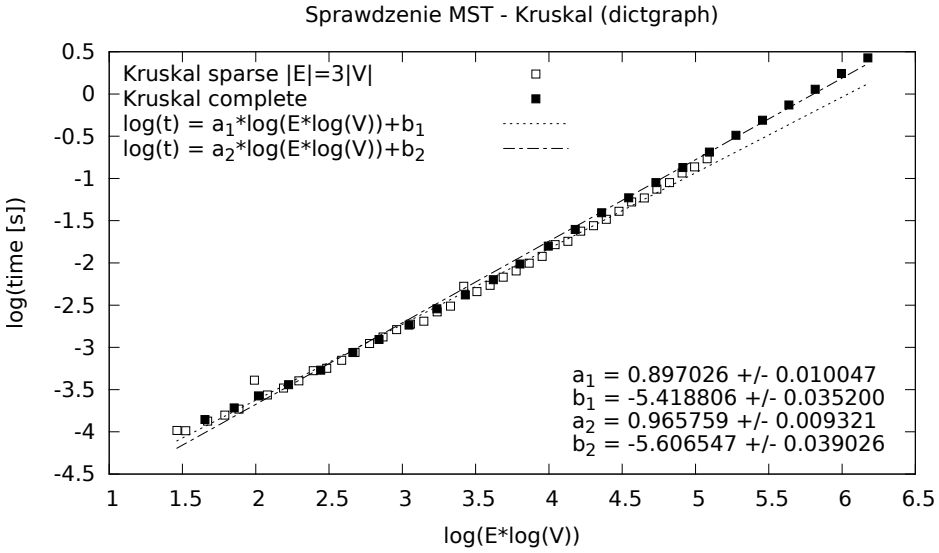
Listing 5.6. Moduł kruskal.py

```
#! /usr/bin/python
# Kod implementujacy pseudokod ze str. 631 CLRS.

from utils.disjointset import DisjointSet
from model.undirectededge import UndirectedEdge

class Kruskal(object):

    @staticmethod
    def execute(graph):
        mst = set()
        disjointset = DisjointSet()
        for node in graph.iternodes():
            disjointset.create(node)
        # sort edges by weight
        sorted_edges = sorted(graph.iteredges(), key=lambda edge: edge.weight)
        for edge in sorted_edges:
            if disjointset.find(edge.source) != disjointset.find(edge.target):
                mst.add(UndirectedEdge(edge.source, edge.target, edge.weight))
                disjointset.union(edge.source, edge.target)
        return mst
```

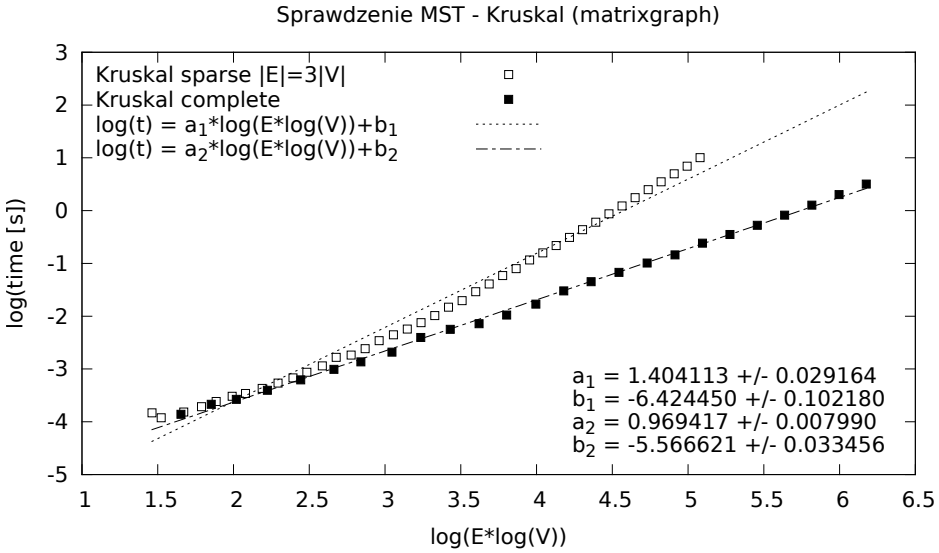
5.3. Najkrótsza ścieżka pomiędzy parą wierzchołków

Problem polega na wyznaczeniu najkrótszej ścieżki pomiędzy dwoma wierzchołkami w grafie ważonym skierowanym. Wagi krawędzi w grafie mogą

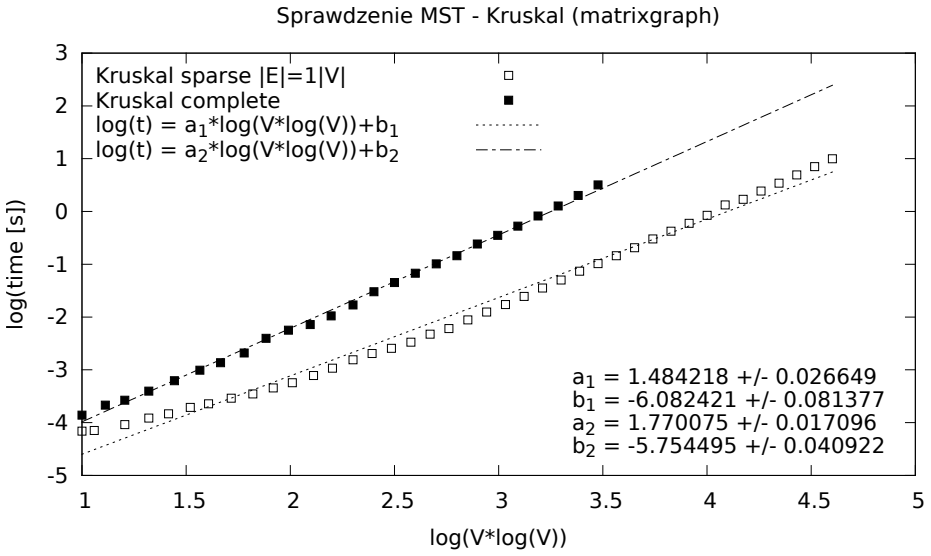
Rysunek 5.15. Test złożoności obliczeniowej algorytmu Kruskala w wersji $O(E \log V)$ dla grafu dictgraph. Współczynniki a bliskie 1 potwierdzają teorię.



Rysunek 5.16. Test złożoności obliczeniowej algorytmu Kruskala w wersji $O(E \log V)$ dla grafu matrixgraph. Zależności zależą od gęstości grafu.



Rysunek 5.17. Test złożoności obliczeniowej algorytmu Kruskala w wersji $O(E \log V)$ dla grafu matrixgraph $|E| = |V|$. W tej reprezentacji liczba wierzchołków jest kluczowa.



reprezentować odległości, czasy, koszty, kary, itd. W przypadku grafu bez wag (wszystkie wagi jednakowe), może być użyty algorytm BFS o złożoności $O(V + E)$ (rozdział 5.1.1).

Należy podkreślić, że dla problemu najkrótszej ścieżki między parą wierzchołków nie jest znany żaden algorytm działający w najgorszym przypadku asymptotycznie szybciej niż najlepszy znany algorytm dla *problemu najkrótszych ścieżek z jednym źródłem* [8]. W algorytmach z tego rozdziału stosowana jest metoda *relaksacji*, a ponadto spotykamy się z problemem *ujemnych wag krawędzi*.

Relaksacja: Proces relaksacji krawędzi służy do skrócenia estymowanej *najkrótszej ścieżki*. Polega to na prostym teście, czy najkrótsza ścieżka znaleziona do tej pory może być skrócona, kiedy będzie przechodzić przez dodatkowy wierzchołek. Jeśli tak, to należy uaktualnić ścieżkę o ten wierzchołek.

Ujemne wagi: Grafy mogą zawierać ujemne wagi, ale najkrótsza scieżka jest zdefiniowana tylko wtedy, kiedy nie istnieje ujemny cykl. Niektóre algorytmy zakładają, że wszystkie wagi muszą być nieujemne (algorytm Dijktry, rozdział 5.3.3). Inne algorytmy dopuszczają ujemne wagi, o ile nie osiągniemy ujemnego cyklu (algorytm Bellmana-Forda, rozdział 5.3.2).

Bez straty ogólności można przyjąć, że najkrótsza ścieżka nie zawiera cykli. Oznacza to, że długości najkrótszych ścieżek nie będą przekraczały $|V| - 1$.

Reprezentowanie najkrótszych ścieżek: Najkrótsze ścieżki będą reprezentowane za pomocą drzewa o korzeniu w źródle s, zwanym *drzewem najkrótszych ścieżek* [8]. W Pythonie będzie to słownik parent, któremu towarzyszy drugi słownik distance, przechowujący odległości wierzchołków od źródła. Algorytmy zwracają oba te słowniki.

5.3.1. Najkrótsza ścieżka w dagu

Najkrótsze ścieżki są zawsze dobrze zdefiniowane w dagu, nawet jeśli występują ujemne wagi, ponieważ nie ma ujemnych cykli.

Dane wejściowe: Graf ważony skierowany acykliczny (dag); wierzchołek początkowy s (źródło).

Problem: Wyznaczenie najkrótszych ścieżek z wierzchołka s do pozostałych wierzchołków grafu.

Opis algorytmu: Algorytm rozpoczyna się sortowaniem topologicznym wierzchołków oraz wstępną inicjalizacją odległości. Jeśli dag zawiera ścieżkę z wierzchołka u do v, wtedy u jest przed v w porządku topologicznym. Aby wyznaczyć odległości, wystarczy przejść przez wszystkie wierzchołki w porządku topologicznym, oraz poddać reklasacji wszystkie krawędzie wychodzące z danego wierzchołka.

Złożoność czasowa: Złożoność algorytmu wynosi $O(E+V)$, ponieważ proces inicjalizacji zajmuje czas $O(V)$, a proces relaksacji wywoływany jest na każdej krawędzi, czyli trwa $O(E)$. Sortowanie topologiczne wierzchołków to czas $O(E+V)$.

Listing 5.7. Moduł dagshortestpath.py

```
#! /usr/bin/python
# Kod implementujacy pseudokod ze str. 655 CLRS.

from algorithms.topologicalsort import TopologicalSort

class DAGShortestPath(object):

    @staticmethod
    def execute(graph, start_node):
        if not graph.is_directed():
            raise ValueError("graph is not directed")
        distance = {}
        parent = {}
        DAGShortestPath._init(graph, distance, parent, start_node)
        sorted_nodes = TopologicalSort.execute(graph)
        for source in sorted_nodes:
            for edge in graph.iteroutedges(source):
                DAGShortestPath._relax(edge, distance, parent)
        return distance, parent   # similar to Dijkstra

    @staticmethod
    def _init(graph, distance, parent, start_node):
        for node in graph.iternodes():
            distance[node] = float("inf")
```

```
            parent[node] = None
        distance[start_node] = 0

    @staticmethod
    def _relax(edge, distance, parent):
        new_distance = distance[edge.source] + edge.weight
        if distance[edge.target] > new_distance:
            distance[edge.target] = new_distance
            parent[edge.target] = edge.source
            return True
        return False
```

5.3.2. Algorytm Bellmana-Forda

Dane wejściowe: Graf ważony skierowany, możliwe są wagi ujemne; wierzchołek początkowy s (źródło).

Problem: Wyznaczenie najkrótszych ścieżek z wierzchołka s do pozostałych wierzchołków grafu. Jeżeli istnieje ujemny cykl, to ma zostać wykryty.

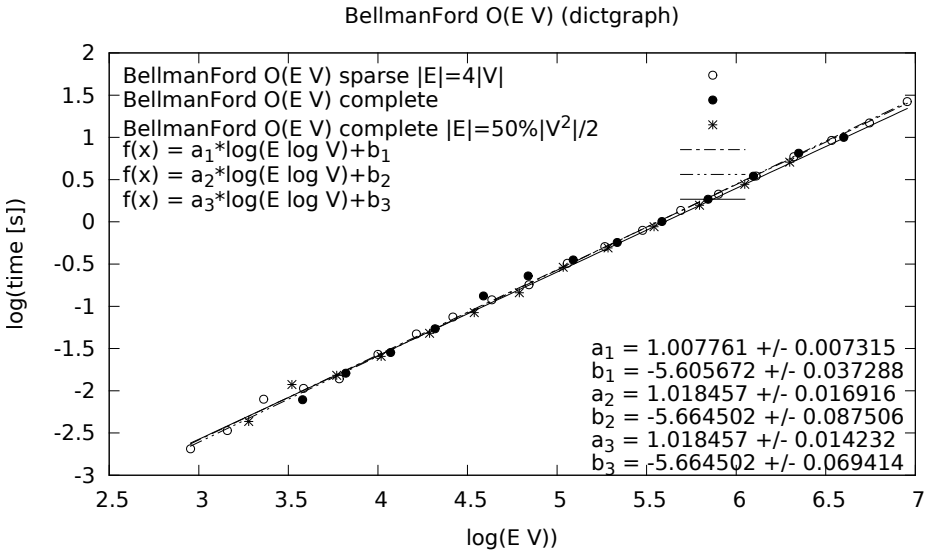
Opis algorytmu: Algorytm rozpoczyna się od inicjalizacji struktur danych przechowujących odległości i rodziców wierzchołków. Dalej algorytm przechodzi $|V| - 1$ razy po wszystkich krawędziach, wykonując na każdej z nich proces relaksacji. Po wyznaczeniu najkrótszej ścieżki wykonywany jest krok sprawdzający, czy istnieje ujmeny cykl. Taki cykl istnieje tylko wtedy, kiedy dodatkowa iteracja po krawędziach poprawi oszacowanie najkrótszej ścieżki. Warto zauważyć, że ta dodatkowa czynność nie powiększa złożoności algorytmu.

Złożoność czasowa: Złożoność algorytmu wynosi $O(VE)$, ponieważ $|V|-1$ razy wykonywana jest relaksacja po krawędziach w czasie $O(E)$. Inicjalizacja zajmuje czas $O(V)$. Wyniki eksperymentów przedstawiają wykresy 5.18 i 5.19.

Listing 5.8. Moduł bellmanford.py

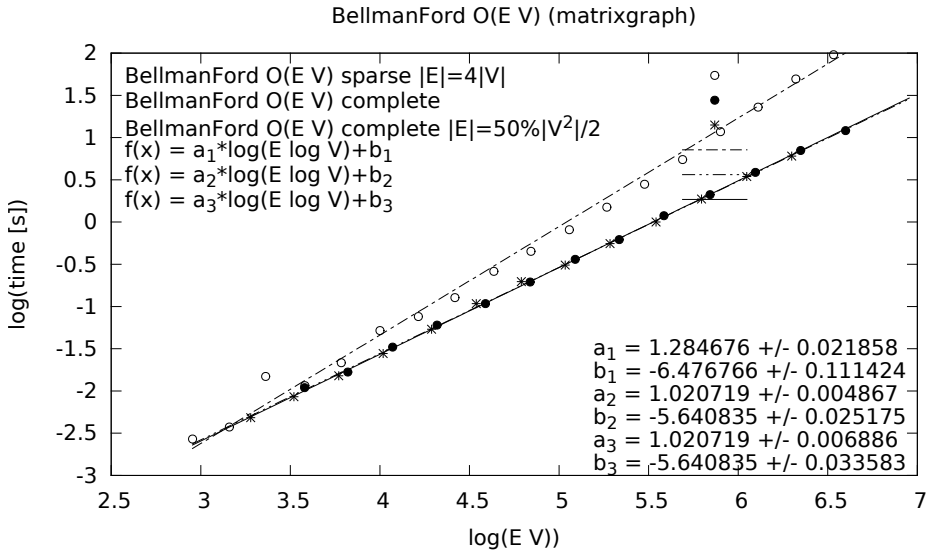
```
#! /usr/bin/python
# Kod implementujacy pseudokod ze str. 651 CLRS.


class BellmanFord(object):

    @staticmethod
    def execute(graph, start_node):
        if not graph.is_directed():
            raise ValueError("graph is not directed")
        distance = {}
        parent = {}
```

Rysunek 5.18. Test złożoności obliczeniowej algorytmu Bellmana-Forda w wersji $O(EV)$ dla grafu dictgraph. Zależność teoretyczna potwierdzona, a bliskie 1.




```
        BellmanFord._init(graph, distance, parent, start_node)
        for i in range(graph.v() - 1):
            for edge in graph.iteredges():
                BellmanFord._relax(edge, distance, parent)
        for edge in graph.iteredges():
            if distance[edge.target] > distance[edge.source] + edge.weight:
                raise ValueError("negative cycle detected")
        return distance, parent   # similar to Dijkstra

    @staticmethod
    def _init(graph, distance, parent, start_node):
        for node in graph.iternodes():
            distance[node] = float("inf")
            parent[node] = None
        distance[start_node] = 0

    @staticmethod
    def _relax(edge, distance, parent):
        new_distance = distance[edge.source] + edge.weight
        if distance[edge.target] > new_distance:
            distance[edge.target] = new_distance
            parent[edge.target] = edge.source
            return True
        return False
```

Rysunek 5.19. Test złożoności obliczeniowej algorytmu Bellmana-Forda w wersji Bellmana-Forda w wersji $O(EV)$ dla grafu matrixgraph. Grafy rzadkie są gorzej obsługiwane.



5.3.3. Algorytm Dijkstry

Dane wejściowe: Graf ważony skierowany o krawędziach z wagami nieujemnymi; wierzchołek początkowy s.

Problem: Wyznaczenie najkrótszych ścieżek z wierzchołka s do pozostałych wierzchołków grafu.

Opis algorytmu: Rozpoczynamy od wstępnej inicjalizacji kolejki priorytetowej oraz odległości pomiędzy źródłem a resztą wierzchołków (początkowe odległości to $+\infty$, a dla źródła 0). Następnie z kolejki wyciągane są kolejno wierzchołki o najmniejszych odległościach. W każdym kroku, mając najbliższy wierzchołek u od źródła, algorytm poprawia odległości pozostałym wierzchołkom (sąsiadom) względem u. Gdy wierzchołek został już odwiedzony, wywołanie relaksacji nic nie zmienia, zatem można je pominąć. Algorytm Dijkstry jest algorytmem zachłannym, ponieważ zawsze wybiera najbliższy wierzchołek.

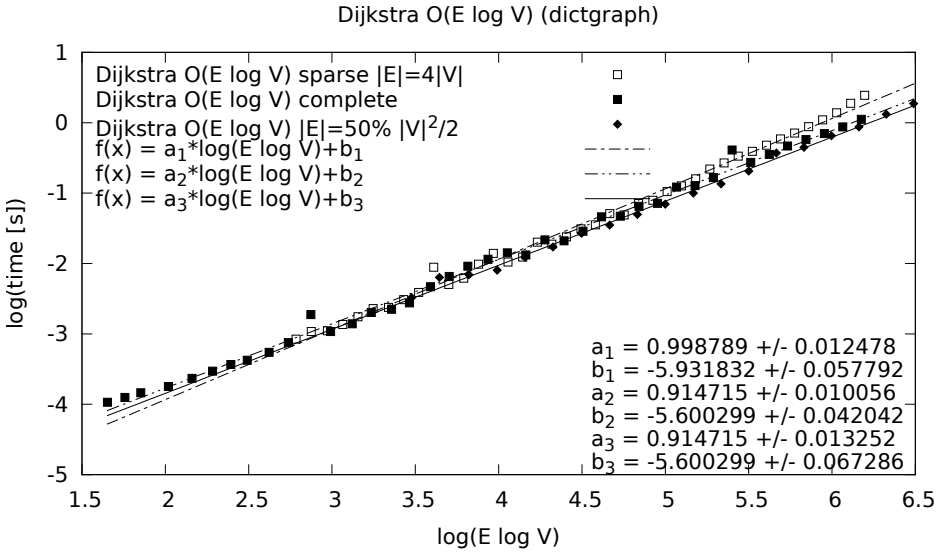
Złożoność czasowa: Złożoność wynosi $O(E \log V)$, ponieważ algorytm przechodzi po wszystkich krawędziach, a przy czym wyciągana jest jeszcze wartość z kolejki, co zajmuje $O(\log V)$. Wyniki eksperymentów przedstawiają wykresy 5.20 i 5.21.

Uwagi: Wersja bez kolejki priorytetowej (lepsza dla grafów gęstych) ma złożoność $O(V^2)$. Wyniki eksperymentów dla tej wersji przedstawiają wykresy 5.22 i 5.23.

Listing 5.9. Moduł dijkstra.py

```
#! /usr/bin/python
# Kod implementujacy pseudokod ze str. 658 CLRS.

from Queue import PriorityQueue
from model.edge import Edge

class Dijkstra(object):

    @staticmethod
    def execute(graph, start_node):
        """Algorytm Dijkstry w czasie O(E log V)."""
        if not graph.is_directed():
            raise ValueError("graph is not directed")
        distance = {}
        parent = {}
        visited = set()
        Dijkstra._init(graph, distance, parent, start_node)
        queue = PriorityQueue()
        queue.put((distance[start_node], start_node))
        while not queue.empty():
            pri, node = queue.get()
            if not node in visited:
                visited.add(node)
                for edge in graph.iteroutedges(node):
                    if (edge.target not in visited and
                    Dijkstra._relax(edge, distance, parent)):
                        queue.put((distance[edge.target], edge.target))
        return distance, parent

    @staticmethod
    def execute2(graph, start_node):
        """Algorytm Dijkstry w czasie O(V**2)."""
        if not graph.is_directed():
            raise ValueError("graph is not directed")
        distance = {}
        parent = {}
        queue = dict((node, None) for node in graph.iternodes())
        Dijkstra._init(graph, distance, parent, start_node)
        while queue:
            node = min(queue, key=distance.get)
            del queue[node]
            for edge in graph.iteroutedges(node):
                if edge.target in queue:
                    Dijkstra._relax(edge, distance, parent)
        return distance, parent
```

Rysunek 5.20. Test złożoności obliczeniowej algorytmu Dijkstry w wersji $O(E \log V)$ dla grafu dictgraph. Zależność teoretyczna potwierdzona.

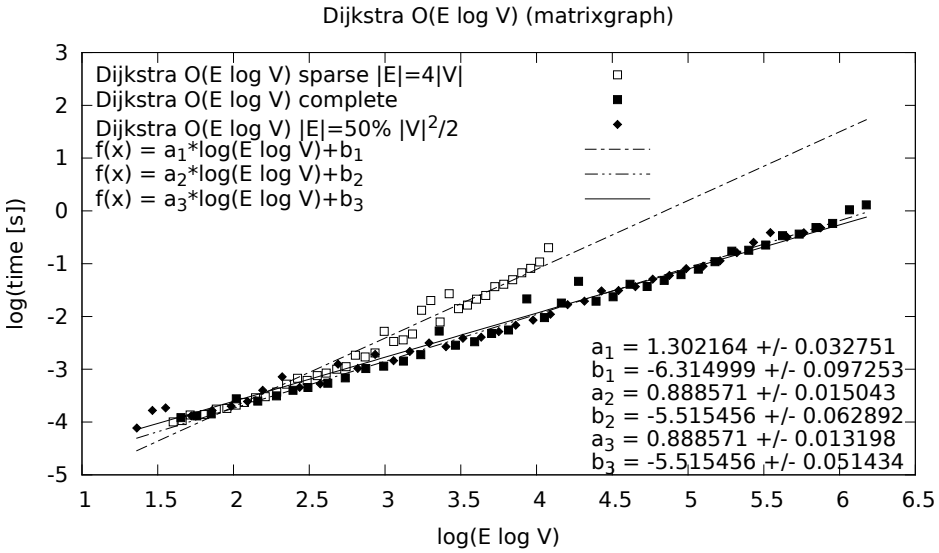



```
@staticmethod
def _init(graph, distance, parent, start_node):
    for node in graph.iternodes():
        distance[node] = float("inf")
        parent[node] = None
    distance[start_node] = 0

@staticmethod
def _relax(edge, distance, parent):
    new_distance = distance[edge.source] + edge.weight
    if distance[edge.target] > new_distance:
        distance[edge.target] = new_distance
        parent[edge.target] = edge.source
        return True
    return False
```

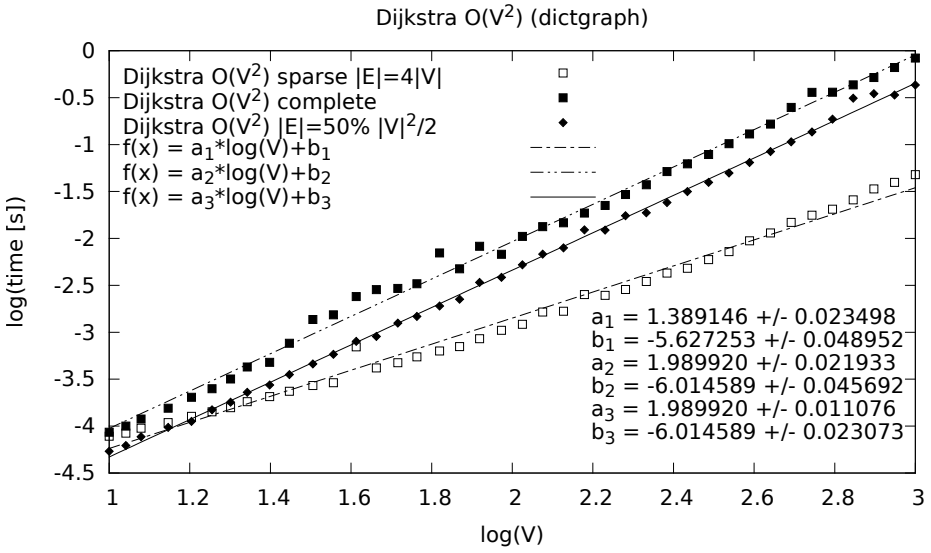
5.4. Najkrótsze ścieżki pomiędzy wszystkimi parami wierzchołków

Problem polega na wyznaczeniu najkrótszej ścieżki pomiędzy wszystkimi parami wierzchołków. Do tego celu można zastosować algorytmy wyszukiwania najkrótszej ścieżki pomiędzy dwoma wierzchołkami (rozdział 5.3). Dla grafów z nieujemnymi wagami można zastosować $|V|$ razy algorytm Dijkstry (rozdział 5.3.3). Dla grafów rzadkich wersja z kolejką priorytetową zajmie $O(VE \log V)$, a dla grafu gęstego wersja oparta na tablicy zajmie czas $O(V^3)$. Dla grafu z dowolnymi wagami, nie można zastosować wspomnianego wyżej algorytmu, ale można użyć wolniejszego algorytmu Bellmana-Forda (rozdział

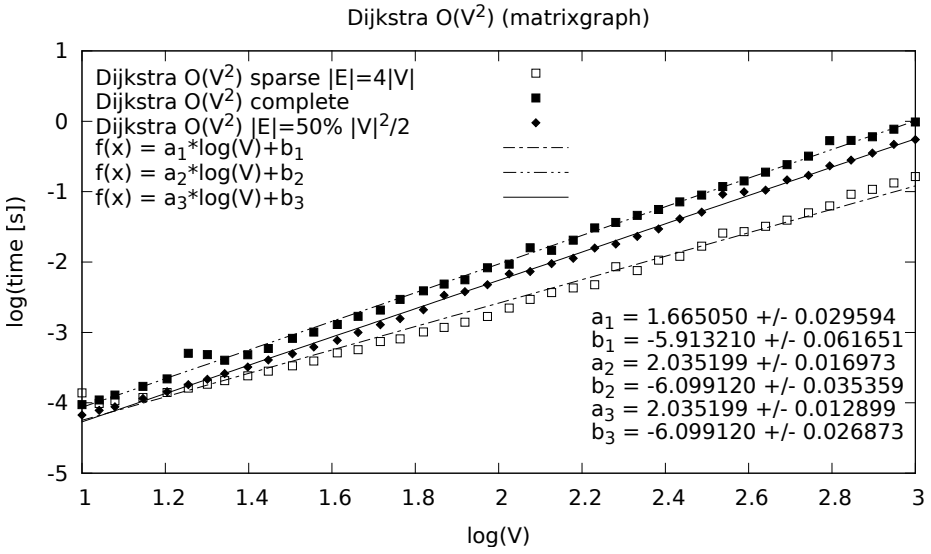
Rysunek 5.21. Test złożoności obliczeniowej algorytmu Dijkstry w wersji $O(E \log V)$ dla grafu matrixgraph. Widać pogorszenie dla grafów rzadkich.



Rysunek 5.22. Test złożoności obliczeniowej algorytmu Dijkstry w wersji $O(V^2)$ dla grafu dictgraph. Dla grafów rzadkich zależność nie jest kwadratowa.



Rysunek 5.23. Test złożoności obliczeniowej algorytmu Dijkstry w wersji $O(V^2)$ dla grafu matrixgraph. Zależność kwadratowa praktycznie potwierdzona (a bliskie 2).



5.3.2), co daje czas $O(V^2E)$ dla grafów rzadkich, a dla grafów gęstych $O(V^4)$. Istnieją jednak algorytmy poświęcone znajdowaniu ścieżek pomiędzy wszystkimi parami wierzchołków, takie jak algorytm Floyda-Warshalla (rozdział 5.4.2), czy algorytm Johnsona (rozdział 5.4.3), które działają szybciej.

5.4.1. Algorytm "mnożenia macierzy"

Jest to algorytm programowania dynamicznego, w którym w pętli głównej programu wykonywane są operacje podobne do mnożenia macierzy, stąd nazwa algorytmu. Wykorzystuje się lemat [8], że wszystkie podścieżki każdej najkrótszej ścieżki są najkrótszymi ścieżkami. Niech $\delta(s,t)$ oznacza najkrótszą ścieżkę o długości $m > 0$. Z lematu wynika, że $\delta(s,t) = \delta(s,u) + w(u,t)$, gdzie $\delta(s,u)$ jest najkrótszą ścieżką o długości $m-1$. Tak wygląda struktura optymalnego rozwiązania, którą wykorzystujemy w algorytmie.

Dane wejściowe: Graf ważony skierowany, bez ujemnych cykli.

Problem: Wyznaczenie najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków grafu.

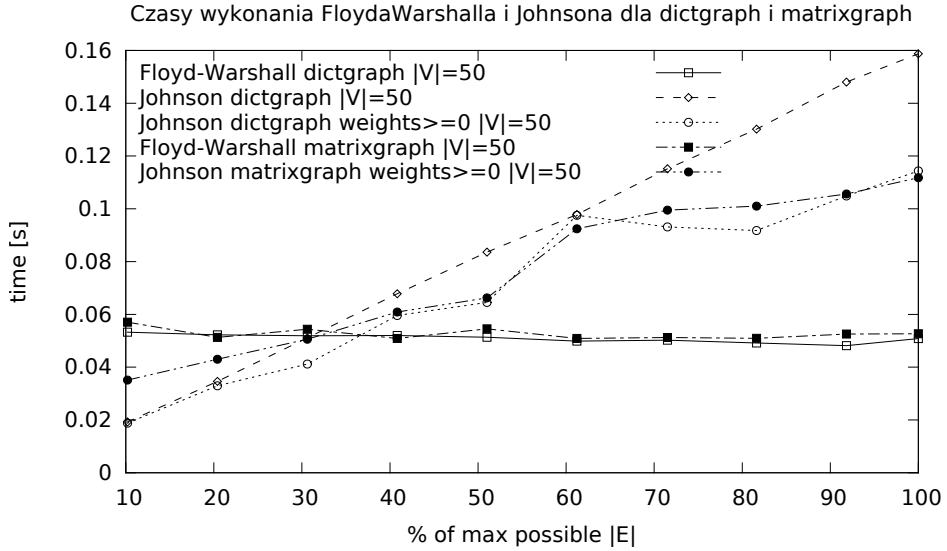
Opis algorytmu: Oznaczamy przez $l_{ij}^{(m)}$ najmniejszą wagę ścieżki z wierzchołka i do wierzchołka j spośród tych, które zawierają co najwyżej m krawędzi. Jeśli $m = 0$, to istnieje ścieżka długości zero dla $i = j$, czyli

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{dla } i = j, \\ +\infty & \text{dla } i \neq j. \end{cases} \tag{5.1}$$

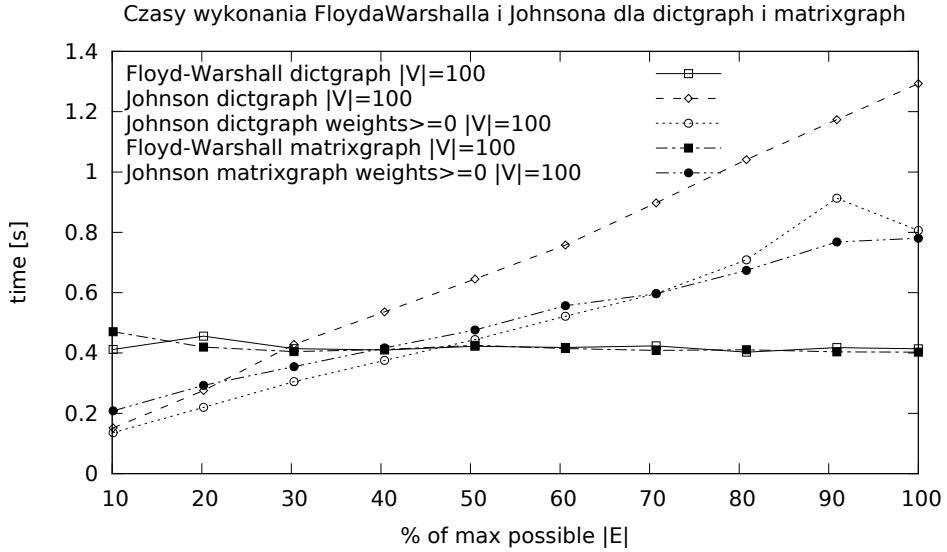
Dla $m > 0$ obliczamy $l_{ij}^{(m)}$ jako następujące minimum:

$$l_{ij}^{(m)} = \min_k \{l_{ik}^{(m-1)} + w(k,j)\}, \tag{5.2}$$

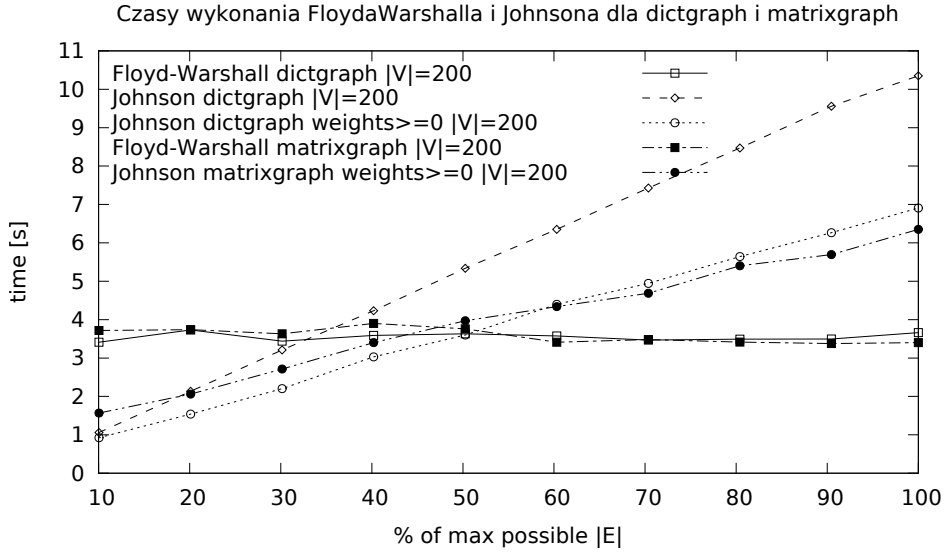
Rysunek 5.24. Porównanie wydajności algorytmów obliczających najkrótsze ścieżki pomiędzy wszystkimi wierzchołkami dla grafu dictgraph oraz matrixgraph - grafy gęste $|V| = 50$.



Rysunek 5.25. Porównanie wydajności algorytmów obliczających najkrótsze ścieżki pomiędzy wszystkimi wierzchołkami dla grafu dictgraph oraz matrixgraph - grafy gęste $|V| = 100$.



Rysunek 5.26. Porównanie wydajności algorytmów obliczających najkrótsze ścieżki pomiędzy wszystkimi wierzchołkami dla grafu dictgraph oraz matrixgraph - grafy gęste $|V| = 200$.



gdzie z założenia $w(j, j) = 0$ dla wszystkich j. Wagi najkrótszych ścieżek obliczamy metodą wstępującą.

Złożoność czasowa: Złożoność wynosi $O(V^4)$.

Złożoność pamięciowa: Złożoność pamięciowa algorytmu wynosi $O(V^2)$, ponieważ zapamiętujemy 5 macierzy (słowniki).

Uwagi: Algorytm wykrywa ujemne cykle poprzez wykrycie ujemnych wartości na diagonali macierzy odległości. Złożoność czasową można poprawić do $O(V^3 \log V)$ wykorzystując metodę wielokrotnego podnoszenia do kwadratu [8]. Wierzchołki najkrótszych ścieżek trzeba wtedy wyznaczyć osobno w czasie $O(V^3)$. Wyniki eksperymentów przedstawiają wykresy 5.27 i 5.28.

Listing 5.10. Moduł allpairs.py

```
#! /usr/bin/python
# Kod implementujacy pseudokod ze str. CLRS.

class AllPairs(object):

    @staticmethod
    def execute(graph):
        if not graph.is_directed():
            raise ValueError("graph is not directed")
        distance = {}
        weights = {}
        parent = {}
        for nodeA in graph.iternodes():
```

```
            distance[nodeA] = {}
            parent[nodeA] = {}
            weights[nodeA] = {}
            for nodeB in graph.iternodes():
                distance[nodeA][nodeB] = float("inf")
                weights[nodeA][nodeB] = float("inf")
                parent[nodeA][nodeB] = None
            distance[nodeA][nodeA] = 0
            weights[nodeA][nodeA] = 0
        for edge in graph.iteredges():
            distance[edge.source][edge.target] = edge.weight
            weights[edge.source][edge.target] = edge.weight
            parent[edge.source][edge.target] = edge.source
        # now L^{(1)} = W
        for m in xrange(2, graph.v()):    # |V|-2 times
            distance, parent = AllPairs._extended_shortest_paths(
                graph, distance, parent, weights)
        if any(distance[node][node] < 0 for node in graph.iternodes()):
            raise ValueError("negative cycle")
        return distance, parent

    @staticmethod
    def _extended_shortest_paths(graph, old_distance, old_parent, weights):
        distance = {}
        parent = {}
        for nodeI in graph.iternodes():
            distance[nodeI] = {}
            parent[nodeI] = {}
            for nodeJ in graph.iternodes():
                distance[nodeI][nodeJ] = old_distance[nodeI][nodeJ] # IMPORTANT
                parent[nodeI][nodeJ] = old_parent[nodeI][nodeJ] # IMPORTANT
                for nodeK in graph.iternodes():
                    alt = old_distance[nodeI][nodeK] + weights[nodeK][nodeJ]
                    if alt < distance[nodeI][nodeJ]:
                        distance[nodeI][nodeJ] = alt
                        parent[nodeI][nodeJ] = nodeK
        return distance, parent
```

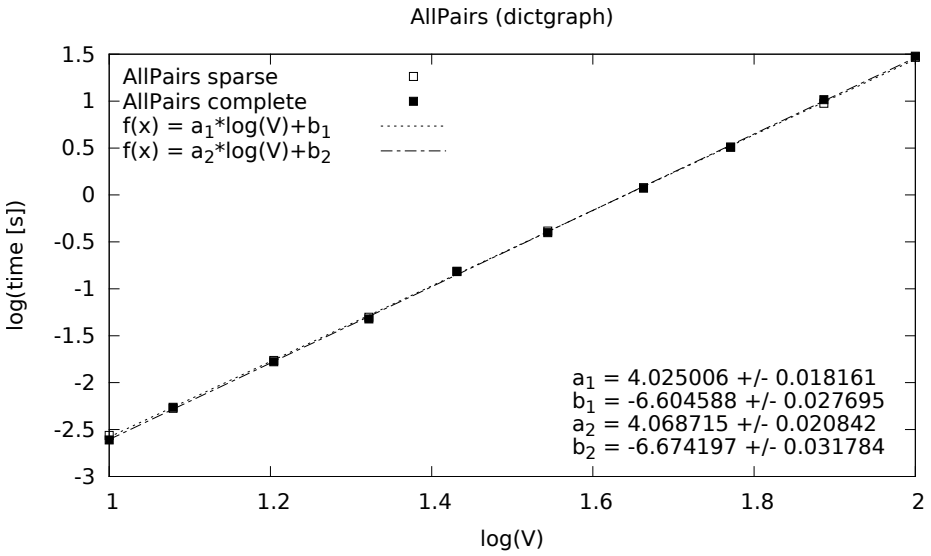
5.4.2. Algorytm Floyda-Warshalla

Dane wejściowe: Graf ważony skierowany, bez ujemnych cykli.

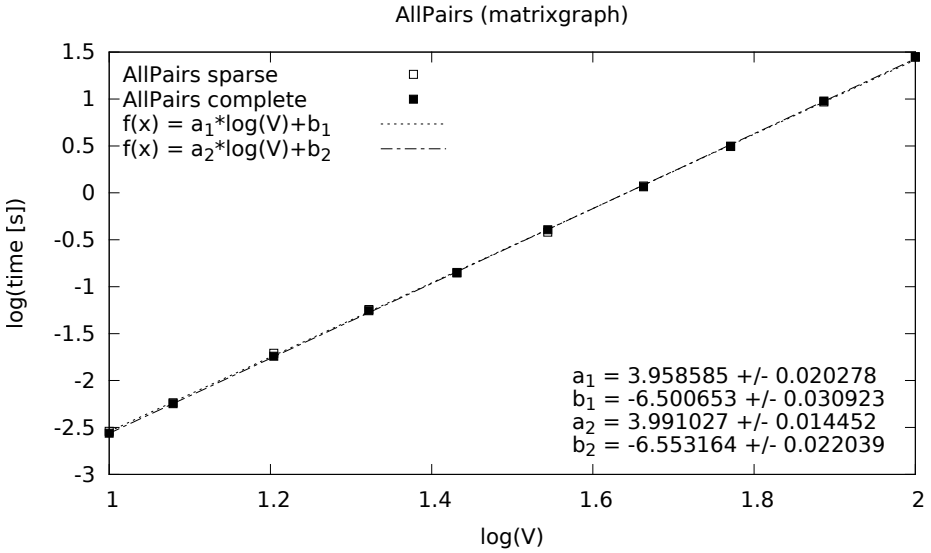
Problem: Wyznaczenie najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków grafu.

Opis algorytmu: Początkowo inicjalizujemy tablicę $|V| \times |V|$ odległości pomiędzy wszystkimi wierzchołkami w taki sposób, że droga do tego samego

Rysunek 5.27. Test złożoności obliczeniowej algorytmu mnożenia macierzy w wersji $O(V^4)$ dla grafu dictgraph. Teoria potwierdzona, a bliskie 4.



Rysunek 5.28. Test złożoności obliczeniowej algorytmu mnożenia macierzy w wersji $O(V^4)$ dla grafu matrixgraph. Teoria potwierdzona, a bliskie 4.



wierzchołka wynosi 0, dla krawędzi Edge(s, t, w) wynosi w, a dla reszty $+\infty$. Algorytm polega na tym, że próbuje zmniejszyć drogę z wierzchołka s do t poprzez wierzchołki pośrednie, tzn. takie przez które można dojść z s do t. Owo wyliczanie jest zdefiniowane rekurencyjnie następująco:

$$d_{ij}^{(k)} = \begin{cases} w(i,j) & \text{dla } k = 0, \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{dla } k \geqslant 1. \end{cases} \tag{5.3}$$

Jest to przykład metody programowania dynamicznego, ponieważ obliczając krok k korzysta z wyliczonej wartości dla kroku $k-1$. Dla każdej pary wierzchołków sprawdza, czy da się zmniejszyć odległość pomiędzy tymi wierzchołkami, przeprowadzając ją przez dodatkowy wierzchołek k. Wystarczy teraz próbować tak skrócić odległości przez wszystkie wierzchołki.

Złożoność czasowa: Złożoność wynosi $O(V^3)$, bo przechodzi po wszystkich parach wierzchołków $|V|$ razy.

Złożoność pamięciowa: Złożoność algorytmu wynosi $O(V^2)$.

Uwagi: Pierwsza implementacja wylicza tylko odległości. Druga implementacja buduje również macierz poprzedników, która umożliwia konstruowanie najkrótszych ścieżek. W obu implementacjach wykrywany jest ujemny cykl poprzez wykrycie ujemnych wartości na diagonali macierzy odległości. Wyniki eksperymentów dla tej wersji przedstawiają wykresy 5.29 i 5.30.

Listing 5.11. Moduł floydwarshall.py

```
#! /usr/bin/python
# Kod implementujacy pseudokod ze str. 695 CLRS.

class FloydWarshall(object):

    @staticmethod
    def execute(graph):
        if not graph.is_directed():
            raise ValueError("graph is not directed")
        distance = {}
        for nodeA in graph.iternodes():
            distance[nodeA] = {}
            for nodeB in graph.iternodes():
                distance[nodeA][nodeB] = float("inf")
            distance[nodeA][nodeA] = 0
        for edge in graph.iteredges():
            distance[edge.source][edge.target] = edge.weight
        for nodeK in graph.iternodes():
            for nodeI in graph.iternodes():
                for nodeJ in graph.iternodes():
                    distance[nodeI][nodeJ] = min(distance[nodeI][nodeJ],
                        distance[nodeI][nodeK] + distance[nodeK][nodeJ])
```

```
        if any(distance[node][node] < 0 for node in graph.iternodes()):
            raise ValueError("negative cycle")
        return distance

    @staticmethod
    def execute_with_paths(graph):
        if not graph.is_directed():
            raise ValueError("graph is not directed")
        distance = {}
        parent = {}
        for nodeA in graph.iternodes():
            distance[nodeA] = {}
            parent[nodeA] = {}
            for nodeB in graph.iternodes():
                distance[nodeA][nodeB] = float("inf")
                parent[nodeA][nodeB] = None
            distance[nodeA][nodeA] = 0
        for edge in graph.iteredges():
            distance[edge.source][edge.target] = edge.weight
            parent[edge.source][edge.target] = edge.source
        for nodeK in graph.iternodes():
            for nodeI in graph.iternodes():
                for nodeJ in graph.iternodes():
                    alt = distance[nodeI][nodeK] + distance[nodeK][nodeJ]
                    if alt < distance[nodeI][nodeJ]:
                        distance[nodeI][nodeJ] = alt
                        parent[nodeI][nodeJ] = parent[nodeK][nodeJ]
        if any(distance[node][node] < 0 for node in graph.iternodes()):
            raise ValueError("negative cycle")
        return distance, parent
```

5.4.3. Algorytm Johnsona

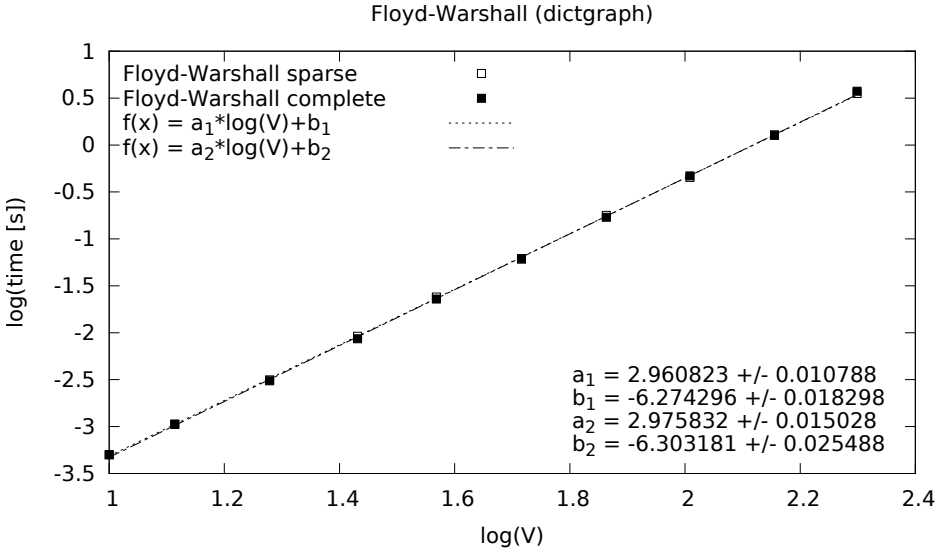
Dane wejściowe: Graf ważony skierowany rzadki, bez ujemnych cykli.

Problem: Wyznaczenie najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków grafu.

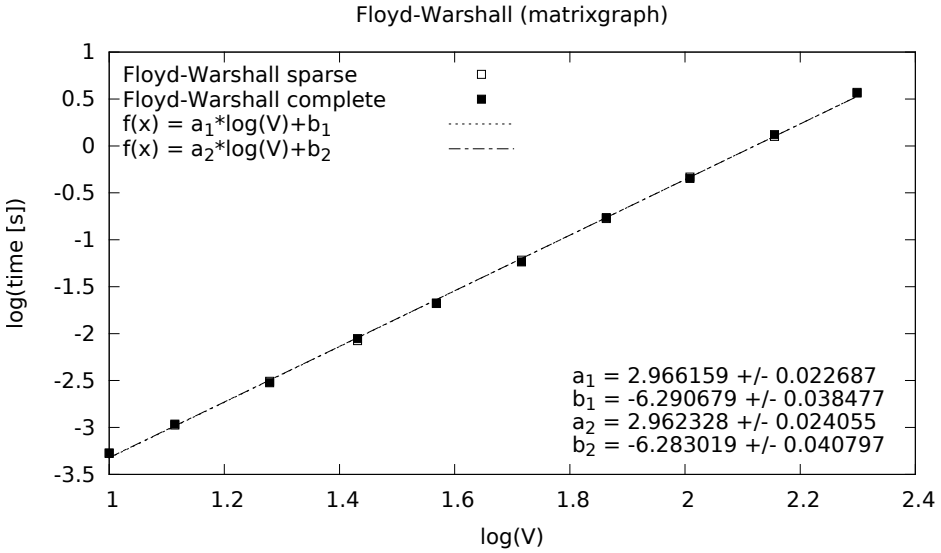
Opis algorytmu: Dla grafów rzadkich algorytm jest asymptotycznie szybszy niż algorytm Floyda-Warshalla (rozdział 5.4.2). Kiedy algorytm nie może wyliczyć najkrótszych ścieżek z powodu istnienia ujemnego cyklu, algorytm o tym raportuje. Wykorzystywane są dwa inne algorytmy: Bellmana-Forda (rozdział5.3.2) oraz Dijkstry (rozdział 5.3.3).

Algorytm składa się z 6 kroków, w tym 5 opcjonalnych, jeśli graf nie ma ujemnych wag:

Rysunek 5.29. Test złożoności obliczeniowej algorytmu Floyda-Warshalla w wersji $O(V^3)$ dla grafu dictgraph. Teoria potwierdzona, a bliskie 3.



Rysunek 5.30. Test złożoności obliczeniowej algorytmu Floyda-Warshalla w wersji $O(V^3)$ dla grafu matrixgraph. Teoria potwierdzona, a bliskie 3.



1. (opcjonalny) Dodanie dodatkowego wierzchołka. Aby algorytm dało się zastosować do grafów z ujemnymi wagami potrzebne jest przemapowanie wag na wagi nieujemne. Polega to na dołożeniu do grafu źródłowego nowego wierzchołka s i połączeniu go krawędziami o wadze 0 z wszystkimi pozostałymi wierzchołkami. Należy to zrobić tak, by nie dało dojść do wierzchołka s z żadnego innego, co za tym idzie żadna najkrótsza ścieżka nie wychodząca z s nie będzie przechodziła prez s, tzn. nie wpłynie to w żaden sposób na oryginalne najkrótsze ścieżki. Nowy graf nie będzie zawierał ujemnego cyklu tylko wtedy, kiedy oryginalny graf takiego cyklu nie posiadał.
2. (opcjonalny) Użycie algorytmu Bellmana-Forda do wyliczenia minimalnych wag krawędzi jakie wchodzą do danego wierzchołka.
3. (opcjonalny) Przemapowanie wag krawędzi na wagi dodatnie, $\hat{w}(u,v) = w(u,v) + h(u) - h(v)$, gdzie $h(u)$ jest minimalną odległością od s do u.
4. (opcjonalny) Usunięcie dołożonego wcześniej wierzchołka s
5. Wykonanie $|V|$ razy algorytmu Dijkstry, dla obliczenia najkrótszej ścieżki dla każdego wierzchołka.
6. (opcjonalny) Przeliczenie obliczonych odległości z powrotem do początkowych wag.

Złożoność czasowa: Złożoność wynosi dla kroku:
1. (opcjonalny) Dodanie dodatkowego wierzchołka. $O(V)$
2. (opcjonalny) Użycie algorytmu Bellmana-Forda do wyliczenia minimalnych wag jakie wchodzą do danego wierzchołka. $O(EV)$
3. (opcjonalny) Przemapowanie wag krawędzi na wagi dodatnie. $O(E)$
4. (opcjonalny) Usunięcie dołożonego wcześniej wierzchołka s
5. Wykonanie $|V|$ razy algorytmu Dijkstry - w zależności od wersji z listą $O(V^3)$, z kolejką priorytetową $O(VE \log V)$
6. (opcjonalny) Przeliczenie obliczonych odległości z powrotem do początkowych wag. $O(V^2)$

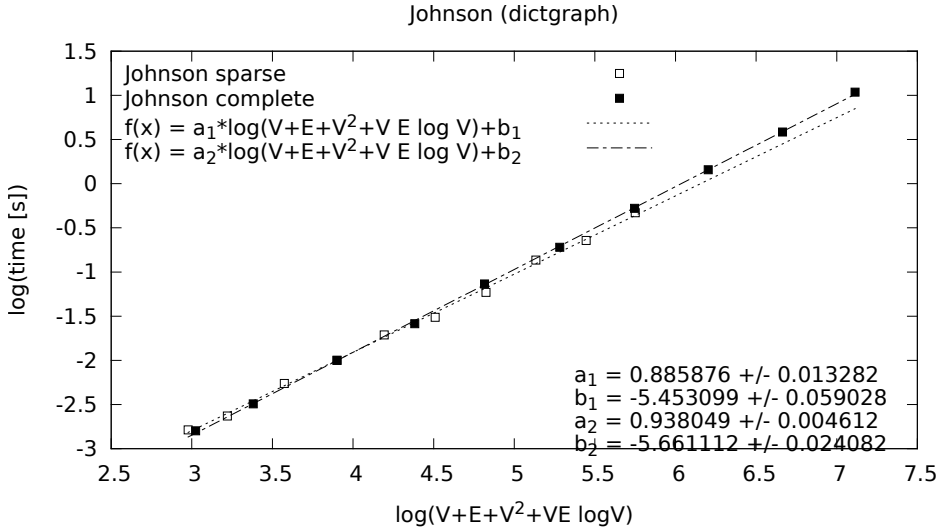
Najdroższy jest krok z użyciem algorytmu Dijkstry, zatem złożoność asymptotyczna algorytmu wynosi właśnie tyle.

Uwagi: Odpowiednie lematy gwarantują, że przewagowanie grafu nie powoduje zmiany najkrótszych ścieżek. Podobnie ujemny cykl również będzie istniał po przewagowaniu grafu. Wyniki eksperymentów przedstawiają wykresy 5.31, 5.32 i 5.33.

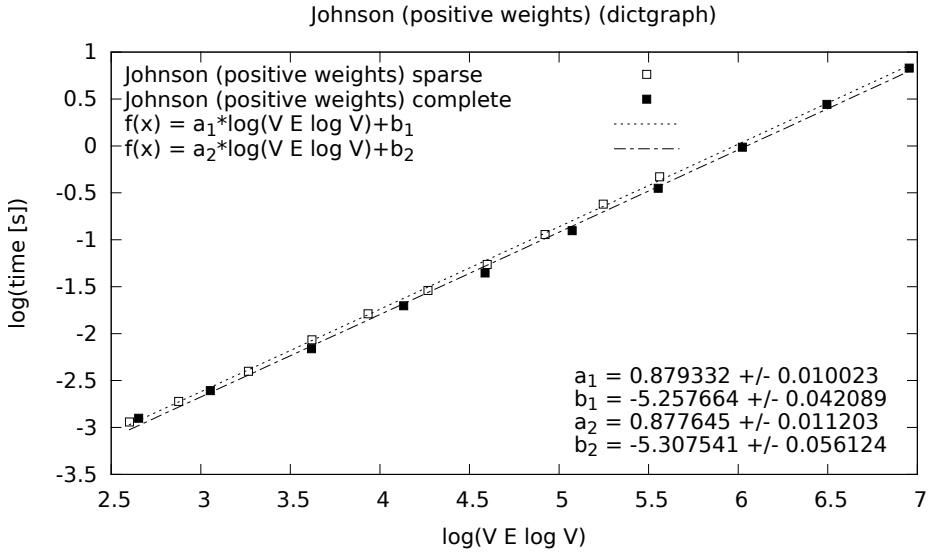
5.5. Maksymalny przepływ

Tak samo, jak można interpretować mapę drogową jako graf i szukać tam najkrótszych ścieżek, to można rozważyć graf jako *sieć przepływową* (rozdział 5.5.1) i wykorzystać ją do badania przepływu pewnej substancji w tej sieci. W ten sposób można modelować wiele problemów, np. obliczanie maksymalnego przepływu cieczy w rurach, części na liniach produkcyjnych lub pojazdów na drogach.

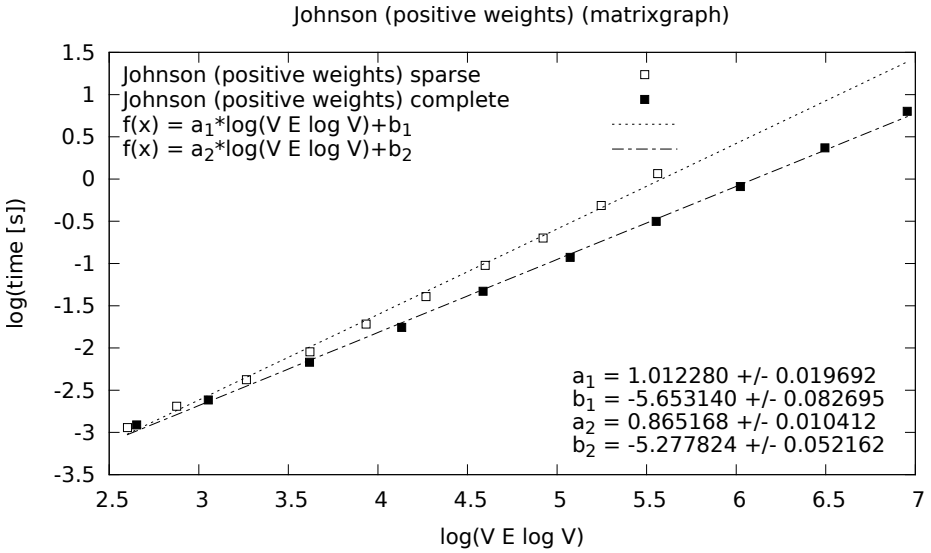
Rysunek 5.31. Test złożoności obliczeniowej algorytmu Johnsona w wersji $O(VE \log V)$ dla grafu dictgraph - dowolne wagi.



Rysunek 5.32. Test złożoności obliczeniowej algorytmu Johnsona w wersji $O(VE \log V)$ dla grafu dictgraph - dodatnie wagi.



Rysunek 5.33. Test złożoności obliczeniowej algorytmu Johnsona w wersji $O(VE \log V)$ dla grafu matrixgraph - dodatnie wagi.



5.5.1. Sieć przepływowa

Sieć przepływowa to graf skierowany $G = (V, E)$ o nieujemnej *przepustowości* krawędzi, $c(u, v) \geqslant 0$ dla $(u, v) \in E$. Wyróżnia się dwa wierzchołki: źródło *source* oraz ujście *sink*. Zakłada się, że w sieci przepływowej nie występują krawędzie przeciwne. Jeżeli dany problem zawiera krawędzie przeciwne, to można zbudować sieć równoważną bez krawędzi przeciwnych. Na jednej z krawędzi przeciwnych, np. (u, v) o przepustowości $c(u, v)$, wprowadza się dodatkowy wierzchołek h. Zamiast jednej krawędzi (u, v) mamy wtedy dwie krawędzie (u, h) i (h, v), obie o przepustowości równej $c(u, v)$.

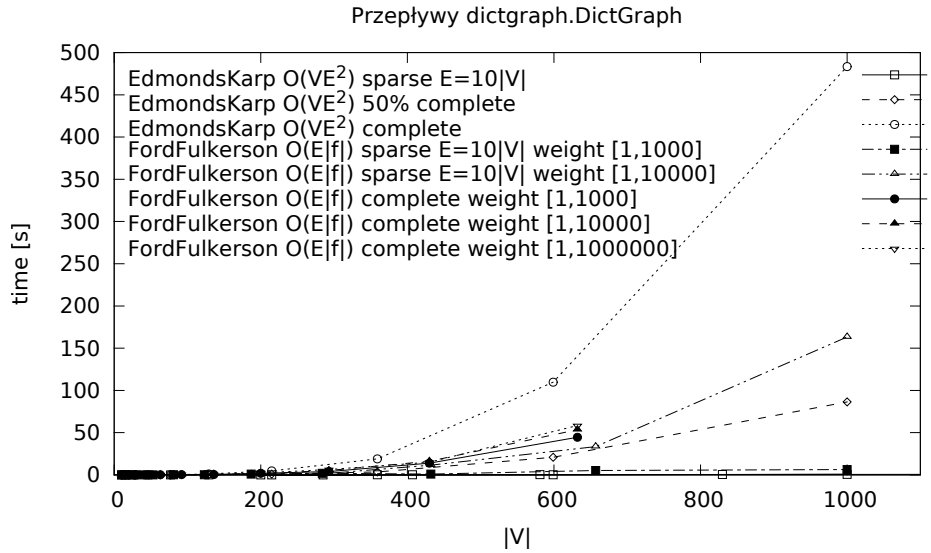
Można również badać sieć, która ma wiele źródeł i wiele ujść. W tym celu wystarczy dodać dodatkowy wierzchołek źródłowy *supersource* i połączyć go krawędziami skierowanymi do reszty wierzchołków źródłowych z nieskończoną przepustowością. Dla ujść postępujemy analogicznie, a dla takiej prostej sieci mogą być zastosowane algorytmy z tego rozdziału.

Przepływem w sieci G nazywamy każdą funkcję f o wartościach rzeczywistych, spełniającą *warunek przepustowości*, $0 \leqslant f(u, v) \leqslant c(u, v)$, oraz *warunek zachowania przepływu*. *Wartością przepływu* $|f|$ jest łączny przepływ opuszczający źródło minus przepływ wchodzący do źródła.

Dla danej sieci przepływowej $G = (V, E)$ i przepływu f definiuje się *sieć residualną* $G_f = (V, E_f)$ o przepustowości krawędzi c_f. Krawędziami w E_f są albo krawędzie z E $[c_f(u, v) = c(u, v) - f(u, v)]$, albo krawędzie do nich przeciwne $[c_f(u, v) = f(v, u)]$. Przepływ w sieci residualnej może być dozwolony, choć w sieci oryginalnej jest zabroniony.

Ścieżką powiększającą nazywamy ścieżkę prostą ze źródła do ujścia w sieci residualnej. *Przepustowość residualna* ścieżki powiększającej jest to najmniejsza przepustowość jej krawędzi.

Rysunek 5.34. Porównanie wydajności algorytmu Edmondsa-Karpa oraz Forda-Fulkersona dla grafu dictgraph.



5.5.2. Algorytm Forda-Fulkersona

Dane wejściowe: Sieć przepływowa; źródło *source* i ujście *sink*.

Problem: Wyznaczenie maksymalnego przepływu.

Opis algorytmu: Początkowo wykonywana jest inicjalizacja sieci residualnej i zerowego przepływu. W każdej iteracji poprzez znajdowanie *ścieżki powiększającej* zwiększa się wyliczany przepływ sieci. Do znajdowania ścieżki powiększającej można zastosować DFS (rozdział 5.1.2) lub BFS (rozdział 5.1.1). Chociaż każda iteracja zwiększa przepływ, trzeba pamiętać, że poszczególne krawędzie grafu mogą zwiększać lub zmniejszać przepływ (zmniejszać, ponieważ niektóre krawędzie mogą być bardziej potrzebne na innej ścieżce). Należy zwiększać przepływ od źródła *source* do ujścia *sink* dopóki jest to możliwe, czyli dopóki istnieje ścieżka powiększająca. W każdym kroku dana krawędź może przyjąć dodatkowy przepływ równy różnicy jej maksymalnej pojemności a aktualnej wartości przepływu w niej, jednak do przepływu w każdym kroku dodawana jest maksymalna wartość, jaka może być przepchnięta przez wszystkie krawędzie z aktualnej ścieżki powiększającej, czyli minimum z wszystkich wcześniej wspomnianych dodatkowych możliwych przepływów. Sieć residualna może zawierać również krawędzie, które nie występują w oryginalnym grafie. Jako że algorytm manipuluje przepływem w celu zwiększenia jego ogólnej wartości, czasem może potrzebować zmniejszyć (wycofać) przepływ na jednej z krawędzi. W celu reprezentowania przypuszczalnych zmniejszeń (wycofania) przepływu na krawędzach, należy dodać do sieci krawędzie odwrotne, których przepustowość wynosi 0, ponieważ nie

znajdowały się one w grafie, czyli nie można przez nie puścić dodatkowego przepływu, a jedynie wycofać już istniejący.

Złożoność czasowa: Złożoność jest ograniczona poprzez $O(E|f|)$, ponieważ ścieżka powiększająca może być znaleziona w czasie $O(E)$, a ta z kolei może zwiększać przepływ co najmniej o 1, aż do maksymalnej wartości przepływu $|f|$.

Uwagi: Przedstawiona implementacja algorytmu Forda-Fulkersona do znajdowania ścieżki powiększającej wykorzystuje algorytm DFS w wersji ze stosem. Przepływy są umieszczone w tablicy dwuwymiarowej (słownik słowników), przy czym przepływ ujemny oznacza, że prawdziwy przepływ jest w kierunku przeciwnym.

Listing 5.12. Moduł fordfulkerson.py

```
#! /usr/bin/python
# Kod inspirowany implementacja ze strony
# http://en.wikipedia.org/wiki/Ford–Fulkerson_algorithm

from model.edge import Edge
from Queue import LifoQueue
import copy

class FordFulkerson(object):

    @staticmethod
    def execute(graph, source, sink):
        copied_graph = copy.deepcopy(graph)  # residual network
        for edge in graph.iteredges():
            copied_graph.add_edge(Edge(edge.target, edge.source, 0))
        flows = {}
        for node in copied_graph.iternodes():
            flows[node] = {}
        while True:
            min_capacity, parent = FordFulkerson._find_path(
                copied_graph, source, sink, flows)
            if min_capacity == 0:
                break
            target = sink
            while target != source:
                node = parent[target]
                flows[node][target] = (
                    flows[node].get(target, 0) + min_capacity)
                flows[target][node] = (
                    flows[target].get(node, 0) - min_capacity)
                target = node
        max_flow = sum(flows[source].get(node, 0)
                       for node in copied_graph.iternodes())
```

```
        return max_flow, flows

    @staticmethod
    def _find_path(residual, source, sink, flows):
        parent = {}
        for node in residual.iternodes():
            parent[node] = None
        capacity = {source: float("inf")}
        stack = LifoQueue()
        stack.put(source)
        while not stack.empty():
            node = stack.get()
            for edge in residual.iteroutedges(node):
                cap = edge.weight - flows[edge.source].get(edge.target, 0)
                if cap > 0 and parent[edge.target] is None:
                    parent[edge.target] = edge.source
                    capacity[edge.target] = min(capacity[edge.source], cap)
                    if edge.target != sink:
                        stack.put(edge.target)
                    else:
                        return capacity[sink], parent
        return 0, parent
```

5.5.3. Algorytm Edmondsa-Karpa

Dane wejściowe: Sieć przepływowa; źródło *source* i ujście *sink*.

Problem: Wyznaczenie maksymalnego przepływu.

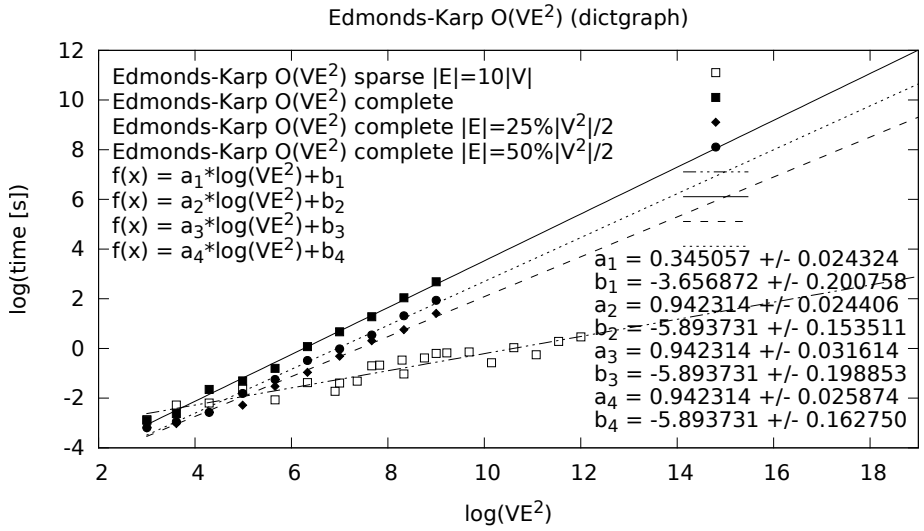
Opis algorytmu: Algorytm jest specjalizacją algorytmu Forda-Fulkersona (rozdział 5.5.2), która poprawia jego złożoność. Ulepszenie polega na innym wyborze ścieżki powiększającej, tj. używając BFS (rozdział 5.1.1) wybiera ścieżkę (od źródła *sink* do ujścia *sink*) o najmniejszej liczbie krawędzi. Z tego wynika, że w trakcie wykonywania algorytmu długości ścieżek powiększających nie maleją. Reszta odbywa się jak w algorytmie Forda-Fulkersona.

Złożoność czasowa: Złożoność wynosi $O(VE^2)$, ponieważ liczba iteracji zwiększających przepływ to $O(VE)$. Dlatego, że na każdej ścieżce powiększającej występuje co najmniej jedna krawędź, która jest krytyczna, co znaczy, że ona decyduje jaki maksymalny przepływ ma cała ścieżka, a każda krawędź $|E|$ może być krytyczna maksymalnie $|V|/2$ razy. Kiedy krawędź osiągnie krytyczne zapełnienie, znika z sieci przepływowej. Ponadto nie może się powtórnie pojawiać dopóki inna ścieżka powiększająca nie wycofa przepływu na tej krawędzi. Krawędź może stać się ponownie krytyczna, jeśli na ścieżce przybyło co najmniej 2 krawędzie, dlatego może stać się $|V|/2$ razy krytyczną. Wyszukanie najkrótszej ścieżki to z kolei czas $O(E)$, jaki zajmuje BFS (rozdział 5.1.1).

Listing 5.13. Moduł edmondskarp.py

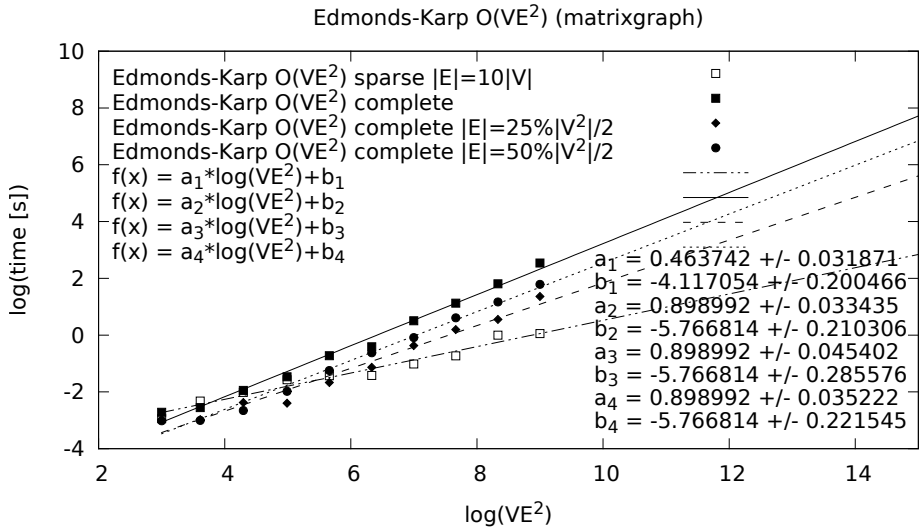
```
#! /usr/bin/python
# Algorytm implementujacy pseudokod ze strony
# http://en.wikipedia.org/wiki/Edmonds-Karp_algorithm

from model.edge import Edge
from Queue import Queue
import copy

class EdmondsKarp(object):

    @staticmethod
    def execute(graph, source, sink):
        copied_graph = copy.deepcopy(graph)  # residual network
        for edge in graph.iteredges():
            copied_graph.add_edge(Edge(edge.target, edge.source, 0))
        flows = {}
        for node in copied_graph.iternodes():
            flows[node] = {}
        while True:
            min_capacity, parent = EdmondsKarp._find_path(
                copied_graph, source, sink, flows)
            if min_capacity == 0:
                break
            target = sink
            while target != source:
                node = parent[target]
                flows[node][target] = (
                    flows[node].get(target, 0) + min_capacity)
                flows[target][node] = (
                    flows[target].get(node, 0) - min_capacity)
                target = node
        max_flow = sum(flows[source].get(node, 0)
                       for node in copied_graph.iternodes())
        return max_flow, flows

    @staticmethod
    def _find_path(residual, source, sink, flows):
        parent = {}
        for node in residual.iternodes():
            parent[node] = None
        capacity = {source: float("inf")}
        queue = Queue()
        queue.put(source)
        while not queue.empty():
            node = queue.get()
            for edge in residual.iteroutedges(node):
```

Rysunek 5.35. Test złożoności obliczeniowej algorytmu Edmondsa-Karpa w wersji $O(VE^2)$ dla grafu dictgraph.

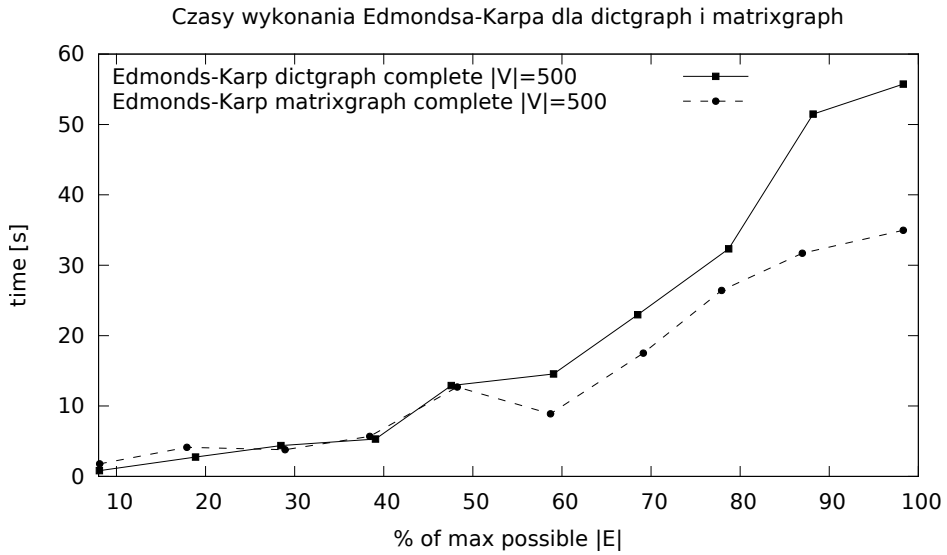



```
                cap = edge.weight - flows[edge.source].get(edge.target, 0)
                if cap > 0 and parent[edge.target] is None:
                    parent[edge.target] = edge.source
                    capacity[edge.target] = min(capacity[edge.source], cap)
                    if edge.target != sink:
                        queue.put(edge.target)
                    else:
                        return capacity[sink], parent
        return 0, parent
```

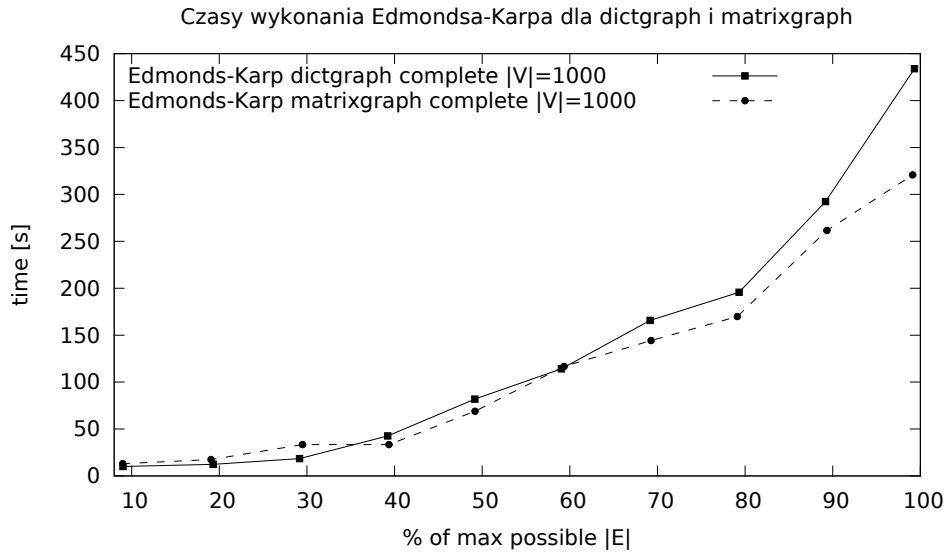
Rysunek 5.36. Test złożoności obliczeniowej algorytmu Edmondsa-Karpa w wersji $O(VE^2)$ dla grafu matrixgraph.



Rysunek 5.37. Porównanie czasu wykonania algorytmu Edmondsa-Karpa dla grafu dictgraph oraz matrixgraph $|V| = 500$.



Rysunek 5.38. Porównanie czasu wykonania algorytmu Edmondsa-Karpa dla grafu dictgraph oraz matrixgraph $|V| = 1000$.



6. Podsumowanie

W niniejszej pracy przedstawiono pythonową implementację grafów ważonych i wielu algorytmów działających na takich grafach. Kod źródłowy algorytmów w wielu wypadkach jest bliski pseudokodowi używanemu w książkach i artykułach informatycznych, ale poza tym można go uruchomić na komputerze z zainstalowanym Pythonem. Naszym zdaniem ma to wielką wartość dydaktyczną, ponieważ można łatwo eksperymentować, sprawdzać różne rozszerzenia i warianty algorytmów, wreszcie można rozwiązywać konkretne problemy obliczeniowe.

Prezentowany kod spełnia standardy ze świata Pythona (zgodność z PEP8), ale także propaguje dobre praktyki programowania: dobrze dobrane nazwy zmiennych, modularność, komentarze w kluczowych punktach programu. Kod jest dobrze przetestowany pod względem poprawności (testy jednostkowe), ale także pod względem wydajności. Aby korzystać z przygotowanych w pracy modułów, należy umieścić katalog src projektu na ścieżce przeszukiwania modułów Pythona, np. można dołączyć nazwę katalogu do zmiennej PYTHONPATH.

W pracy zdefiniowano interfejs grafów i podano dwie różne jego realizacje, słownikową i macierzową. Z testów wynika, że implementacja słownikowa w większości przypadków jest tak szybka, jak implementacja macierzowa, a przy tym jest bardziej oszczędna pamięciowo dla grafów rzadkich. Zakładane cele pracy zostały osiągnięte. Jest jasne, że kod napisany w Pythonie zwykle nie będzie szybszy niż ten napisany w języku C/C++, czy Java. Jednak ogromną zaletą Pythona jest czytelność kodu, możliwość szybkiego tworzenia działających prototypów programów, aby sprawdzić różne koncepcje programistyczne. Czasem wystarczy przepisać do innego języka tylko pewien krytyczny fragment programu, aby mieć w pełni funkcjonalną aplikację.

W świecie wolnego oprogramowania można znaleźć wiele bibliotek grafowych, również napisanych w Pythonie. Najbardziej rozbudowaną wydaje się biblioteka *NetworkX* [10], a inne można znaleźć na stronach Python Package Index [11]. Funkcjonalność tych bibliotek jest zwykle większa niż naszego kodu, ponieważ mają za sobą wiele lat rozwoju i sztab programistów. Jednak w żadnym zbadanym przypadku kod źródłowy nie był tak przejrzysty, aby mógł służyć do nauki algorytmów. Z reguły stosowane są zaawansowane konstrukcje języka Python, jest dużo udogodnień dla użytkownika, które jednak zaciemniają sedno użytych algorytmów.

A. Kod źródłowy dla klas krawędziowych

W tym dodatku przedstawimy kod źródłowy klas reprezentujących krawędzie skierowane (klasa Edge) i nieskierowane (klasa UndirectedEdge).

A.1. Klasa Edge

Listing A.1. Moduł edge.py

```
#! /usr/bin/python

class Edge(object):

    def __init__(self, source, target, weight=1):
        self.source = source
        self.target = target
        self.weight = weight

    def inverted(self):
        return Edge(self.target, self.source, self.weight)

    __invert__ = inverted

    def __repr__(self):
        return "Edge(%s, %s, %s)" % (
        repr(self.source), repr(self.target), repr(self.weight))

    def __hash__(self):
        h = 7 + hash(self.source)
        h = h * 17 + hash(self.target)
        return h

    def __eq__(self, other):
        return self.source == other.source and self.target == other.target

    def __ne__(self, other):
        return not self == other

    def __lt__(self, other):
        return self.weight < other.weight
```

```
    def __gt__(self, other):
        return self.weight > other.weight

    def __le__(self, other):
        return not self > other

    def __ge__(self, other):
        return not self < other
```

A.2. Klasa UndirectedEdge

Listing A.2. Moduł undirectededge.py

```
#! /usr/bin/python

import model.edge

class UndirectedEdge(model.edge.Edge):

    def __init__(self, source, target, weight=1):
        if source > target:
            self.source, self.target = target, source
        else:
            self.source, self.target = source, target
        self.weight = weight

    def __repr__(self):
        return "UndirectedEdge(%s, %s, %s)" % (
        repr(self.source), repr(self.target), repr(self.weight))

    def __hash__(self):
        h = 7 + hash(self.source) + hash(self.target)
        return h
```

B. Kod źródłowy dla klas grafowych

Interfejs klas grafowych został zawarty w abstrakcyjnej klasie bazowej BaseGraph. Następnie ten interfejs został zrealizowany na dwa różne sposoby, w implementacji słownikowej (klasa DictGraph) i implementacji macierzowej (klasa MatrixGraph). Poniżej prezentujemy kod źródłowy tych klas.

B.1. Klasa BaseGraph

Listing B.1. Moduł basegraph.py

```
#! /usr/bin/python

from abc import ABCMeta, abstractmethod

class BaseGraph(object):
    # For tests: isinstance(x, BaseGraph).
    __metaclass__ = ABCMeta

    @abstractmethod
    def is_directed(self):
        pass

    @abstractmethod
    def v(self):
        pass

    @abstractmethod
    def e(self):
        pass

    @abstractmethod
    def has_node(self, node):
        pass

    @abstractmethod
    def add_node(self, node, value=None):
        pass

    @abstractmethod
    def get_value(self, node):
        pass
```

```
    @abstractmethod
    def set_value(self, node):
        pass

    @abstractmethod
    def del_node(self, node):
        pass

    remove_node = del_node

    @abstractmethod
    def add_edge(self, edge):
        pass

    @abstractmethod
    def del_edge(self, edge):
        pass

    remove_edge = del_edge

    @abstractmethod
    def has_edge(self, edge):
        pass

    @abstractmethod
    def iteradjacent(self, node):
        pass

    @abstractmethod
    def iteroutedges(self, source):
        pass

    @abstractmethod
    def iterinedges(self, source):
        pass

    @abstractmethod
    def iternodes(self):
        pass

    @abstractmethod
    def iteredges(self):
        pass

    @abstractmethod
    def weight(self, edge):
        pass
```

```
    @abstractmethod
    def __eq__(self, other):
        pass

    @abstractmethod
    def __ne__(self, other):
        pass

    @abstractmethod
    def add_graph(self, other):
        pass

    @abstractmethod
    def show(self):
        pass
```

B.2. Klasa DictGraph

Listing B.2. Moduł dictgraph.py

```
#! /usr/bin/python

from model.basegraph import BaseGraph
from model.edge import Edge

class DictGraph(BaseGraph):

    def __init__(self, directed=False):
        self.edges = {}
        self.nodes = {}
        self.directed = directed

    def is_directed(self):
        return self.directed

    def v(self):
        return len(self.nodes)

    def e(self):    # could be faster
        return sum(1 for edge in self.iteredges())

    def has_node(self, node):
        return node in self.nodes

    def add_node(self, node, value=None):
```

```
        self.nodes[node] = value

    def get_value(self, node):
        return self.nodes[node]

    def set_value(self, node, value):
        self.nodes[node] = value

    def remove_node(self, node):
        # remove node with inedges and outedges
        for edge in list(self.iterinedges(node)):
            self.del_edge(edge)
        if self.directed:
            for edge in list(self.iteroutedges(node)):
                self.del_edge(edge)
        del self.nodes[node]

    del_node = remove_node

    def add_edge(self, edge):
        if edge.source == edge.target:
            raise ValueError("loops are forbidden")
        if not edge.source in self.nodes:
            self.nodes[edge.source] = None
        if not edge.target in self.nodes:
            self.nodes[edge.target] = None

        if not self.directed:
            if not edge.target in self.edges:
                self.edges[edge.target] = {}
            if edge.source not in self.edges[edge.target]:
                self.edges[edge.target][edge.source] = edge.weight
            else:
                raise ValueError("parallel edges are forbidden")

        if not edge.source in self.edges:
            self.edges[edge.source] = {}
        if edge.target not in self.edges[edge.source]:
            self.edges[edge.source][edge.target] = edge.weight
        else:
            raise ValueError("parallel edges are forbidden")

    def remove_edge(self, edge):
        if not self.directed:
            del self.edges[edge.target][edge.source]
        del self.edges[edge.source][edge.target]

    del_edge = remove_edge
```

```
def has_edge(self, edge):
    return (edge.source in self.edges) and (
    edge.target in self.edges[edge.source])

def iteradjacent(self, node):
    if not node in self.edges:
        return iter([])
    return self.edges[node].iterkeys()

def iteroutedges(self, source):
    if source in self.edges:
        for (target, weight) in self.edges[source].iteritems():
            yield Edge(source, target, weight)

def iterinedges(self, source):
    if self.is_directed():
        for (target, sources) in self.edges.iteritems():  # time O(V)
            if source in sources:
                yield Edge(target, source, sources[source])
    else:
        for target in self.edges[source]:
            yield Edge(target, source, self.edges[target][source])

def iternodes(self):
    return self.nodes.iterkeys()

def iteredges(self):
    for (source, adj) in self.edges.iteritems():
        for (target, weight) in adj.iteritems():
            if self.directed or source < target:
                yield Edge(source, target, weight)

def weight(self, edge):
    if edge.source in self.edges and edge.target in self.edges[edge.source]:
        return self.edges[edge.source][edge.target]
    else:
        return 0    # MatrixGraph compatibility

def __eq__(self, other):
    if self.is_directed() is not other.is_directed():
        return False
    if self.v() != other.v():
        return False
    for node in self.iternodes():  # time O(V)
        if not other.has_node(node):
            return False
    for edge in self.iteredges():  # time O(E)
```

```
            if not other.has_edge(edge):
                return False
        return True

    def __ne__(self, other):
        return not self == other

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

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

B.3. Klasa MatrixGraph

Listing B.3. Moduł matrixgraph.py

```
#! /usr/bin/python

from model.basegraph import BaseGraph
from model.edge import Edge

class MatrixGraph(BaseGraph):

    def __init__(self, n, directed=False):
        self.n = n
        self.directed = directed
        self.data = [[0] * self.n for node in xrange(self.n)]

    def is_directed(self):
        return self.directed

    def v(self):
        return self.n

    def e(self):
        counter = self.n**2
        for source in xrange(self.n):
            non_edges_number = self.data[source].count(0)
```

```
            counter -= non_edges_number
        return counter if self.directed else counter / 2

    def has_node(self, node):
        return True if 0 <= node < self.n else False

    def add_node(self, node, value=None):
        pass

    def get_value(self, node):
        pass

    def set_value(self, node):
        pass

    def del_node(self, node):
        for source in xrange(self.n):
            self.data[source][node] = 0
            self.data[node][source] = 0

    remove_node = del_node

    def add_edge(self, edge):
        if edge.source == edge.target:
            raise ValueError("loops are forbidden")
        if self.data[edge.source][edge.target] == 0:
            self.data[edge.source][edge.target] = edge.weight
        else:
            raise ValueError("parallel edges are forbidden")
        if not self.directed:
            if self.data[edge.target][edge.source] == 0:
                self.data[edge.target][edge.source] = edge.weight
            else:
                raise ValueError("parallel edges are forbidden")

    def del_edge(self, edge):
        self.data[edge.source][edge.target] = 0
        if not self.directed:
            self.data[edge.target][edge.source] = 0

    remove_edge = del_edge

    def has_edge(self, edge):
        return self.data[edge.source][edge.target] != 0

    def iteradjacent(self, node):
        for target in xrange(self.n):
            if self.data[node][target] != 0:
```

```
            yield target

    def iteroutedges(self, source):
        for target in xrange(self.n):
            if self.data[source][target] != 0:
                yield Edge(source, target, self.data[source][target])

    def iterinedges(self, source):
        for target in xrange(self.n):
            if self.data[target][source] != 0:
                yield Edge(target, source, self.data[target][source])

    def iternodes(self):
        return iter(xrange(self.n))

    def iteredges(self):    # time O(V**2)
        for source in xrange(self.n):
            for target in xrange(self.n):
                if self.data[source][target] != 0 and (self.directed or source < target):
                    yield Edge(source, target, self.data[source][target])

    def weight(self, edge):
        return self.data[edge.source][edge.target]

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

    def __eq__(self, other):
        if self.is_directed() is not other.is_directed():
            return False
        if self.v() != other.v():
            return False
        for node in self.iternodes():  # time O(V)
            if not other.has_node(node):
                return False
        if self.e() != other.e():  # time O(V**2)
            return False
        for edge in self.iteredges():  # time O(V**2)
            if not other.has_edge(edge):
                return False
        return True

    def __ne__(self, other):
        return not self == other
```

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

C. Testy dla klas grafowych

W tym dodatku przedstawimy wyniki testów, w których sprawdzono zapotrzebowanie na pamięć przy przetwarzaniu grafów, a także zbadano wydajność czasową wybranych operacji na grafach.

C.1. Wymagania pamięciowe grafów

Przy przetwarzaniu grafów ważna jest znajomość zapotrzebowania na pamięć operacyjną oraz dyskową, kiedy chcemy trwale zapisać obiekty grafów. Zbadamy przede wszystkim wielkości grafów, które są zachowywane na dysku za pomocą standardowego modułu pickle do serializacji obiektów. W praktyce do zapisu lub odczytu grafu z pliku dyskowego potrzeba około dwa razy większej ilości pamięci operacyjnej, niż wielkość pliku dyskowego.

C.1.1. Wymagania pamięciowe implementacji słownikowej

Graf oparty na implementacji słownikowej zajmuje pamięć proporcjonalnie do liczby wierzchołków i krawędzi $O(V + E)$. Rozmiar grafu w pamięci z samymi wierzchołkami bez krawędzi będzie się skalował jak $O(V)$. Zależności te są takie same dla grafów skierowanch (rys. C.4) i nieskierowanych (rys. C.5 oraz rys. C.1). Implementacja zawarta w tej pracy tworzy graf nieskierowany tak, że jednej krawędzi nieskierowanej odpowiadają wewnętrznie dwie krawędzie skierowane. W związku z tym graf nieskierowany będzie zajmował w pamięci około 2 razy tyle miejca co graf skierowany, kiedy $|E|$ jest znacznie większe od $|V|$ (rys. C.2 oraz rys. C.3). Na rysunku C.3 można zauważyć tę zależność, bo wykres grafu nieskierowanego $|E| = |V|$ nałożył się z wykresem dla grafu skierowanego $|E| = 2|V|$. Na rysunku C.3 widać także, że wielkość grafu jest taka sama dla skierowanego bądź nieskierowanego grafu, kiedy rozważa się same wierzchołki bez krawędzi (wykresy nałożyły się na siebie).

C.1.2. Wymagania pamięciowe implementacji macierzowej

Graf oparty na implementacji macierzowej zajmuje pamięć proporcjonalnie do kwadratu liczby wierzchołków $O(V^2)$ i nie ma znaczenia, czy jest to graf gęsty czy rzadki, skierowany (rys.C.7) czy nieskierowany (rys.C.8), bo zaalokowana pamięć i tak zawsze wynosi w przybliżenu tyle samo - tworzona jest kwadratowa macierz sąsiedztwa. Rys. C.6 pokazuje rozmiar pamięci potrzebny na alokacje grafów skierowanych i nieskierowanych. Widać, że pamięć potrzebna na alokację grafów rzadkich i bez krawędzi wynosi w przybliżeniu tyle samo, ale grafy gęste zajmują więcej niż pozostałe, ponadto graf pełny nieskierowany więcej niż pełny skierowany. Jest to efekt uboczny serializacji,

zero to jeden znak, a inne liczby mogą mieć więcej cyfr. W pamięci operacyjnej grafy będą zajmować tyle samo miejsca.

C.1.3. Porównanie pamięciowe implementacji macierzowej i słownikowej

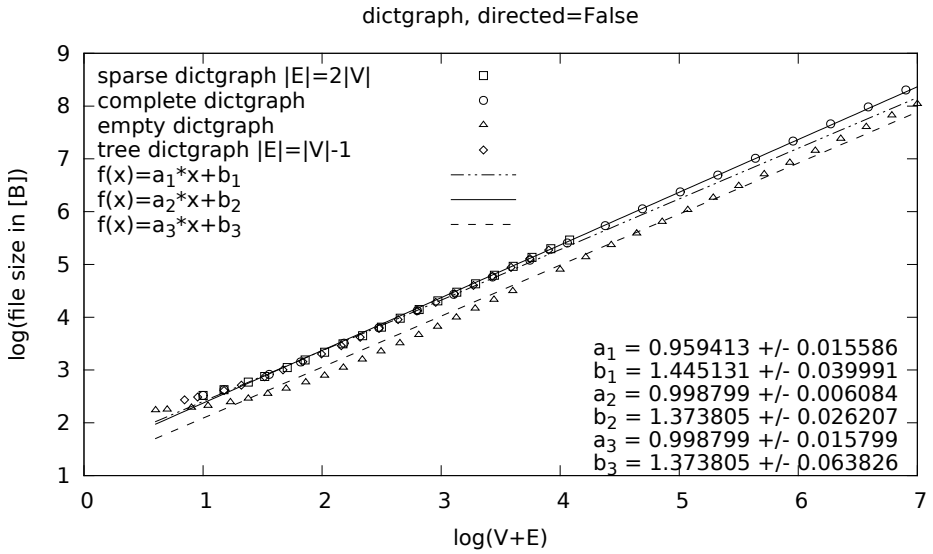
Rozmiar grafu w implementacji macierzowej jest mniej więcej stały w zależności od liczby krawędzi. Rozmiar grafu w implementacji słownikowej jest zmienny. Rysunek C.9 zawiera zestawienie rozmiarów grafów skierowanych i nieskierowanych dla implementacji słownikowej i macierzowej w zależności od liczby wierzchołków. Rysunek C.11 przedstawia porównanie rozmiarów grafów pełnych nieskierowanych dla $|V| = 1000$. Rysunek C.10 przedstawia porównanie rozmiarów grafów pełnych skierowanych dla $|V| = 1000$. Widać tutaj, że mniej więcej przy połowie zapełnienia maksymalnej liczby możliwych krawędzi, implementacja słownikowa zaczyna zajmować więcej pamięci. Dzieje się tak dlatego, że stuktura danych słownika zajmuje więcej miejsca niż prosta tablica, ponadto przy przekroczeniu pewnego współczynnika zapełnienia następuje realokacja pamięci. Trzeba również mieć na uwadze, że słownik alokuje trochę więcej pamięci niż aktualnie potrzebuje.

C.1.4. Porównanie wydajnościowe wybranych operacji na implementacji macierzowej i słownikowej

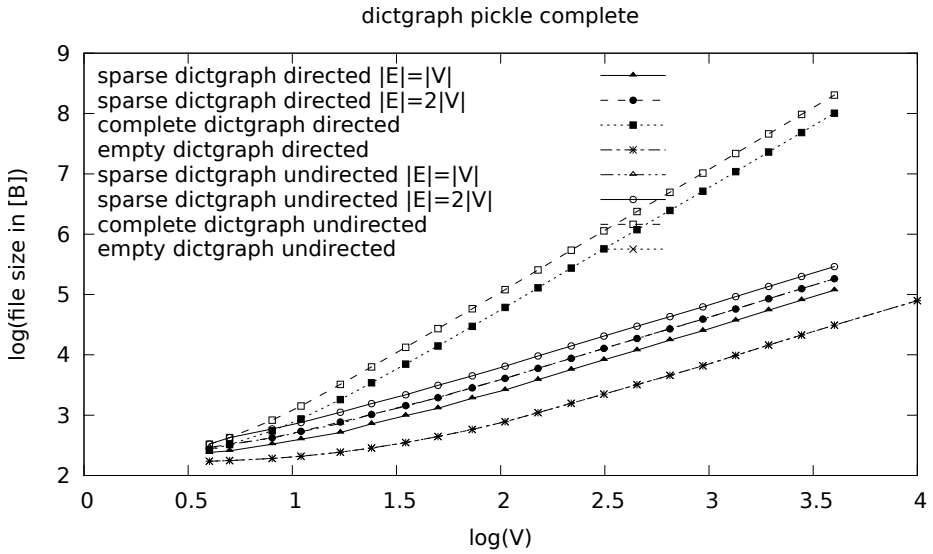
Rysunek C.12 zawiera wykres średniego czasu dostępu do wagi krawędzi. Widać, że czas jest stały, ale implementacja macierzowa jest szybsza. Wynika to z tego, że słowniki wewnątrz swojej implementacji obliczają funkcje hashujące, co stanowi własnie ten narzut.

Rysuek C.13 zawiera wykres średniego czasu iteracji przypadającego na jedną krawędź. Iteracja po krawędziach w implementacji słownikowej ma złożoność $O(E)$, czyli czas przypadający na jedną krawędź jest stały. Iteracja po krawędziach w implementacji macierzowej ma złożoność $O(V^2)$. Dla grafu gęstego widać stałą zależność, ponieważ krawędzi jest właśnie $O(V^2)$, Natomiast dla grafu rzadkiego widać liniową zależność, co oznacza że na każdą krawędź przy iteracji przypada średnio $O(V)$ operacji.

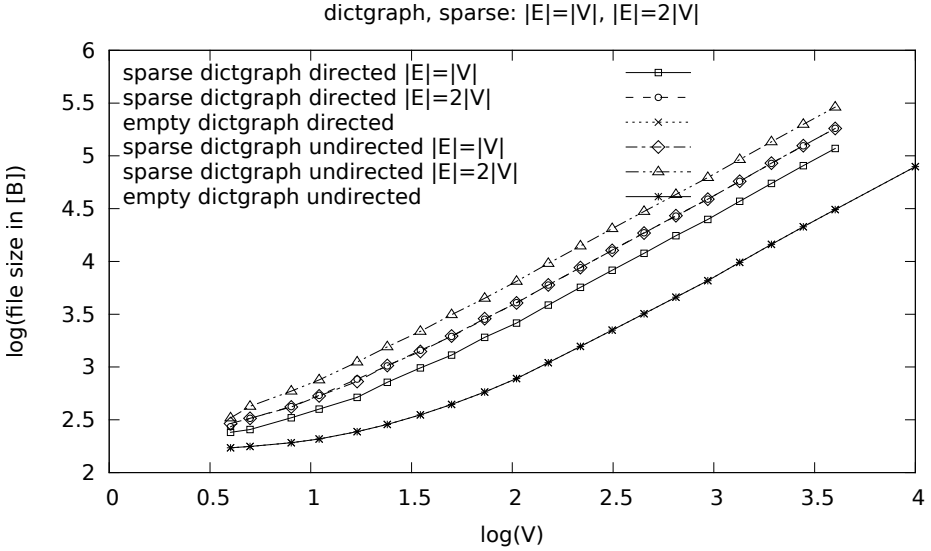
Rysunek C.1. Porówanie wielkości grafów nieskierowancych w pamięci dla dictgraph, w zależności od $V + E$.



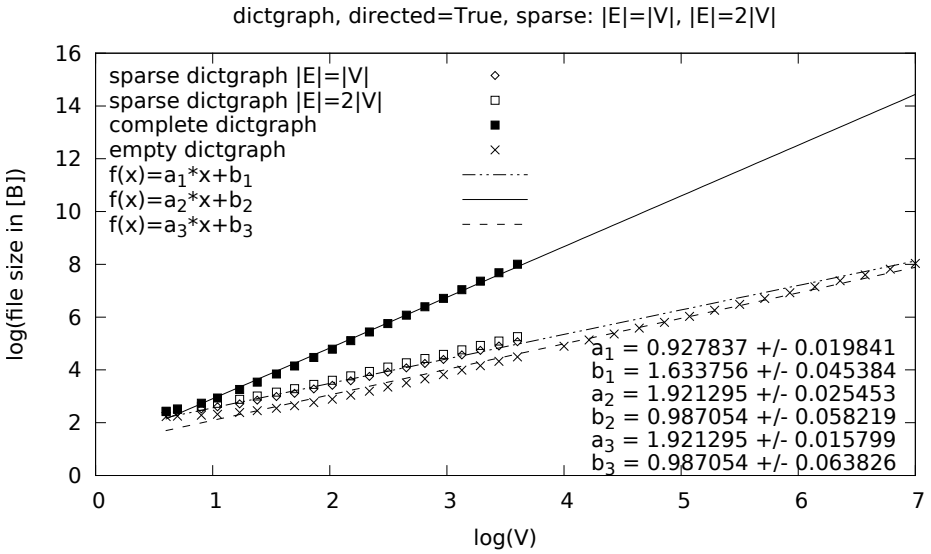
Rysunek C.2. Porówanie wielkości grafów skierowanych i nieskierowancych dla grafów gęstych w pamięci dla dictgraph, w zależności od V.



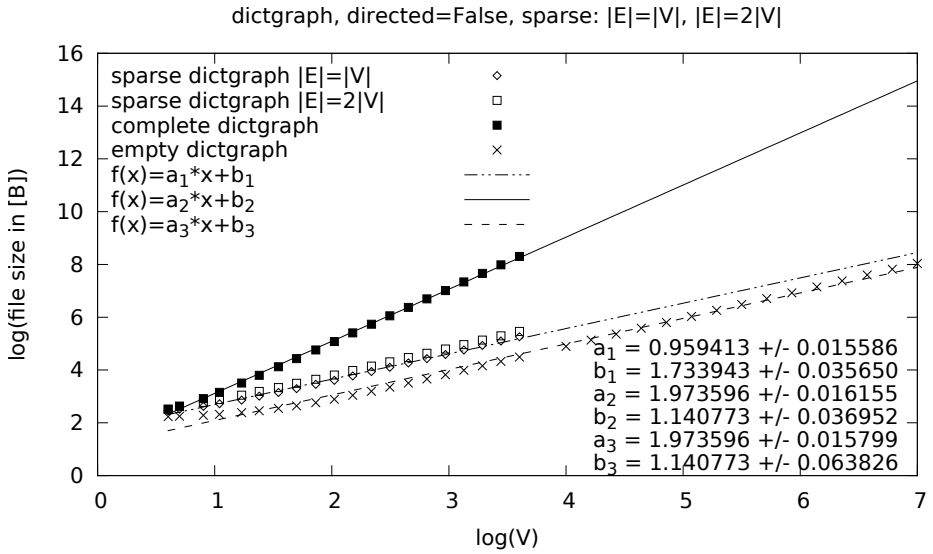
Rysunek C.3. Porówanie wielkości grafów skierowanych i nieskierowancych dla grafów rzadkich w pamięci dla dictgraph, w zależności od V.



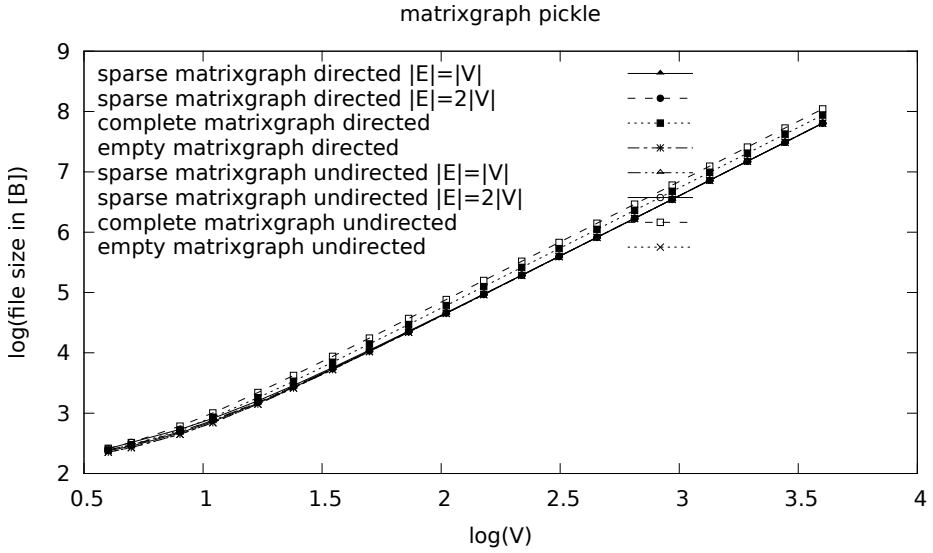
Rysunek C.4. Sprawdzanie wielkości grafów skierowanych w pamięci dla dictgraph, w zależności od V.



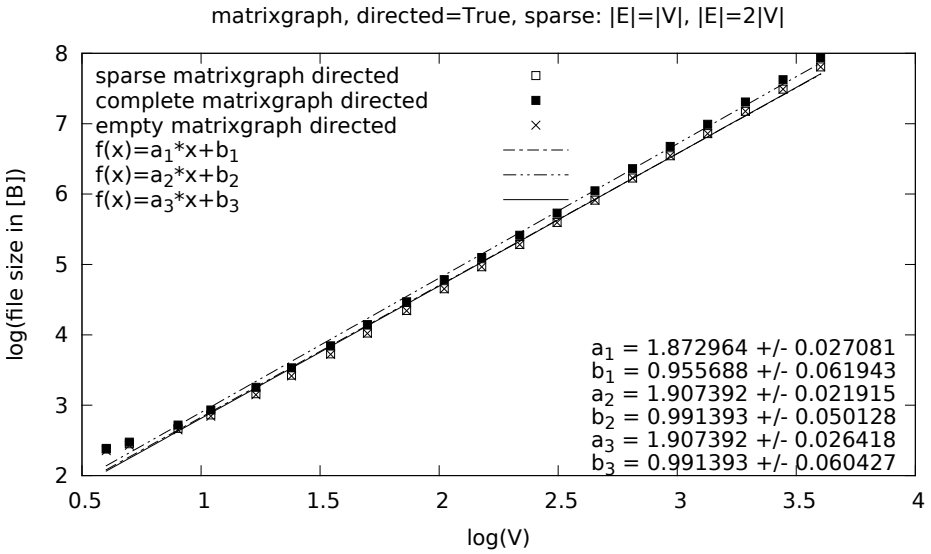
Rysunek C.5. Sprawdzanie wielkości grafów nieskierowanych w pamięci dla dictgraph.



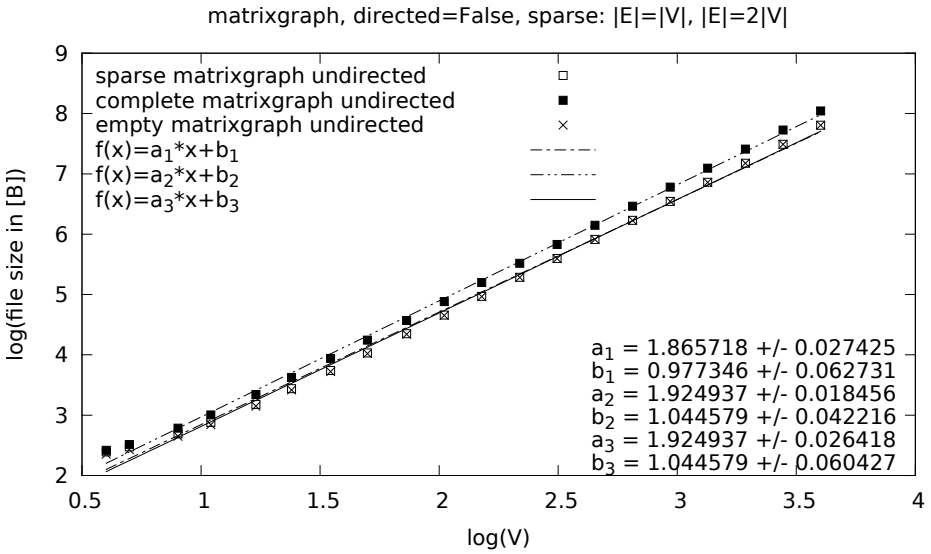
Rysunek C.6. Porównanie wielkości grafów skierowanych i nieskierowancych w pamięci dla matrixgraph, w zależności od V.



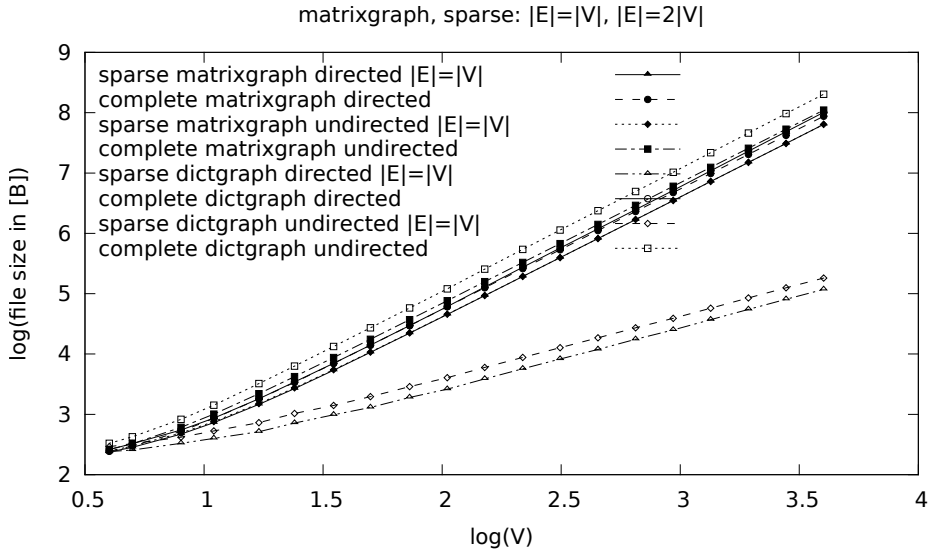
Rysunek C.7. Sprawdzanie wielkości grafów skierowanych w pamięci dla matrixgraph, w zależności od V.



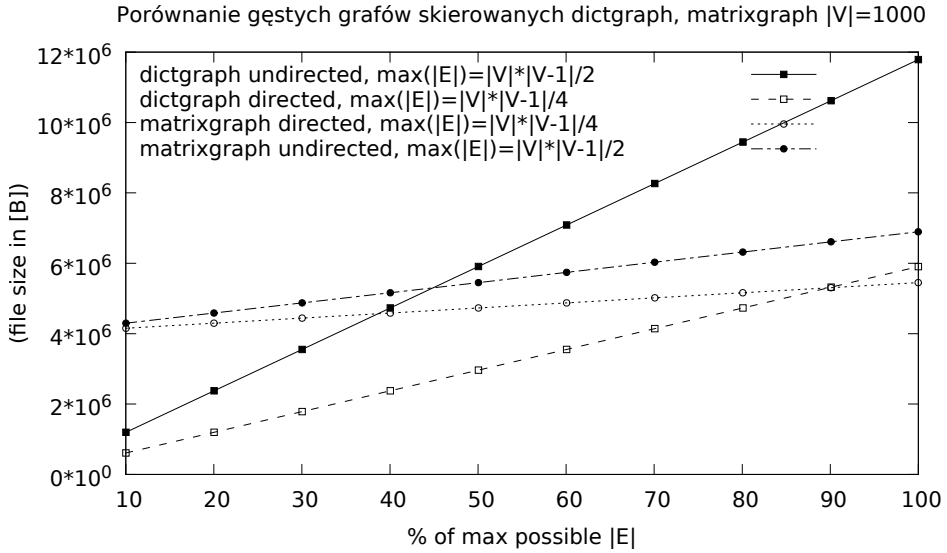
Rysunek C.8. Sprawdzanie wielkości grafów nieskierowanych w pamięci dla matrixgraph, w zależności od V.



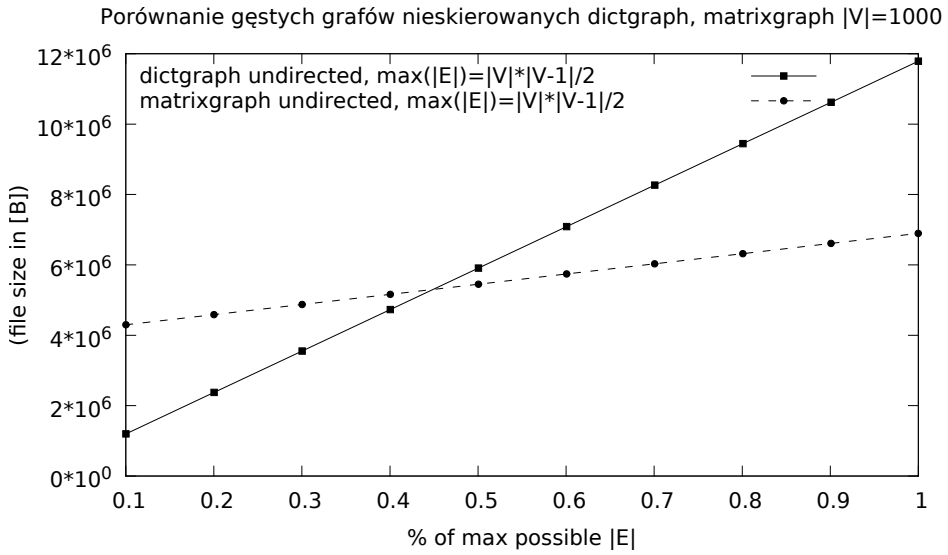
Rysunek C.9. Porównanie rozmiaru grafów dla implementacji słownikowej i macierzowej.



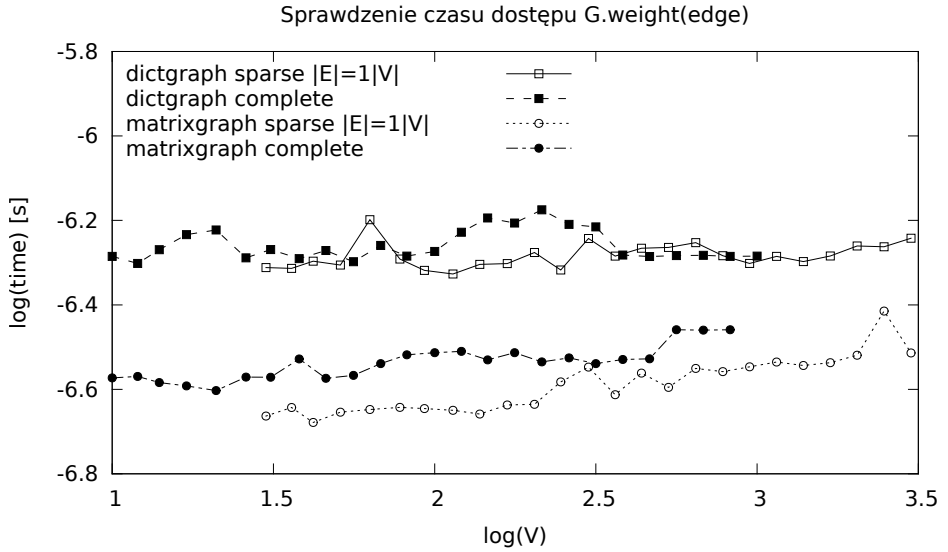
Rysunek C.10. Porównanie rozmiaru grafu (macierzowego i słownikowego) skierowanego dla $|V| = 1000$, w zależności od liczby krawędzi.



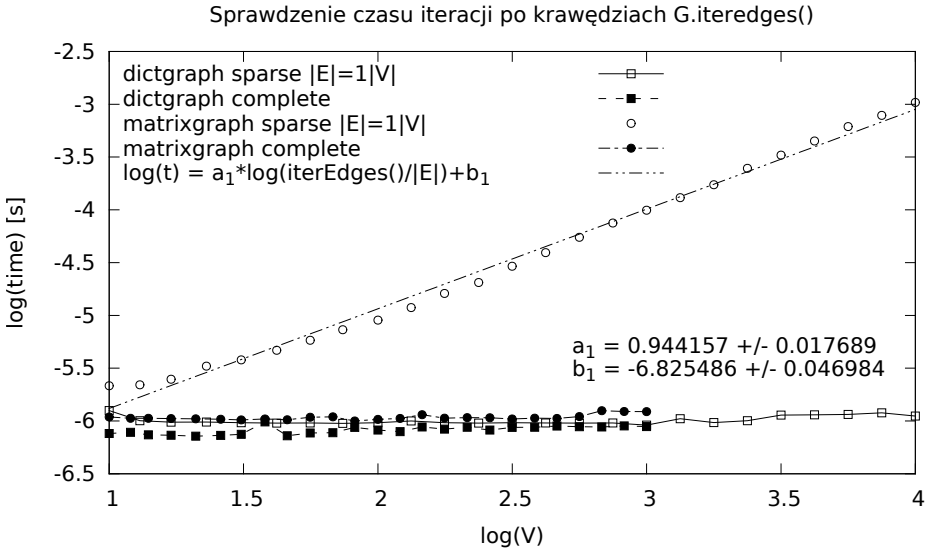
Rysunek C.11. Porównanie rozmiaru grafu (macierzowego i słownikowego) nieskierowanego dla $|V| = 1000$, w zależności od liczby krawędzi.



Rysunek C.12. Sprawdzenie czasu dostępu do wagi krawędzi.



Rysunek C.13. Sprawdzenie czasu iteracji po krawędziach.



Bibliografia

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

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

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

[4] Python Programming Language - Data Structure's Time Complexity (CPython), `https://wiki.python.org/moin/TimeComplexity/`.

[5] Python Programming Language - How are dictionaries implemented?, `https://docs.python.org/2/faq/design.html#how-are-lists-implemented/`.

[6] Python Programming Language - Why must dictionary keys be immutable?, `https://docs.python.org/2/faq/design.html#why-must-dictionary-keys-be-immutable/`.

[7] Wikipedia, Graph theory, 2014, `https://en.wikipedia.org/wiki/Graph_theory`.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, 2009.

[9] D. E. Knuth, *Sztuka programowania, tom 1*, Wydawnictwa Naukowo-Techniczne, Warszawa 2002.

[10] NetworkX, wersja 1.9, 2014, `https://networkx.github.io/`.

[11] PyPI - the Python Package Index, `https://pypi.python.org/pypi/`