Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Piotr Szestało

Nr albumu: 1064484

Badanie grafów Hamiltona z językiem Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2015

1:6255631627

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

2:1069620033

Pragnę serdecznie podziękować Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za ogromne zaangażowanie, rady, poświęcony cenny czas oraz udzieloną pomoc w napisaniu tej pracy magisterskiej.

3:7557771896

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów związanych z grafami Hamiltona. Wykorzystano interfejs dla grafów oparty na dwóch podstawowych klasach Edge i Graph. Klasa Edge reprezentuje krawędzie skierowane z wagą. Klasa Graph reprezentuje grafy proste ważone, skierowane i nieskierowane.

W pracy zaimplementowano dwa algorytmy przeszukiwania grafów: algorytm przeszukiwania wszerz (BFS), oraz algorytm przeszukiwania w głąb (DFS). Zaimplementowano algorytm rekurencyjny na bazie DFS, który znajduje wszystkie ścieżki i cykle Hamiltona w grafach skierowanych i nieskierowanych.

Dla grafów nieskierowanych ważonych zbadano problem komiwojażera. Zaimplementowano algorytm dokładny na bazie DFS, oraz szereg algorytmów przybliżonych: algorytm najbliższych sąsiadów, algorytm najbliższych sąsiadów z powtórzeniami, algorytm sortowania krawędzi. Dla metrycznego problemu komiwojażera przedstawiono algorytm 2-aproksymacyjny, bazujący na minimalnym drzewie rozpinającym. Omówiono również trzy metaheurystyki, które są stosowane w kontekście problemu komiwojażera.

Dla grafów skierowanych acyklicznych zaimplementowany został algorytm sortowania topologicznego, na bazie DFS. Stworzono również funkcje do badania tranzytywności turnieju (grafu pełnego skierowanego), oraz do znajdywania ścieżek Hamiltona w turniejach.

Ważną częścią pracy są zestawienia twierdzeń matematycznych, które dotyczą cykli Hamiltona, z wieloma przykładami edukacyjnymi. Dla algorytmów rozwiązujących problem komiwojażera wykonane zostały testy wydajnościowe, oraz testy dokładności.

Słowa kluczowe: teoria grafów, grafy hamiltonowskie, przeszukiwanie wszerz, przeszukiwanie w głąb, sortowanie topologiczne, problem komiwojażera, metaheurystyki, turniej

4:7823180585

English title: Study of Hamiltonian graphs with Python

Abstract

Python implementation of selected graph algorithms connected with Hamiltonian graphs are presented. Graphs interface based on two clases is used. The `Edge` class represents directed weighted edges. The `Graph` class is for simple weighted graphs, directed and undirected.

In this work, two graphs traversing algorithms are implemented: breadth-first search (BFS) and depth-first search (DFS). The recursive algorithm based on DFS is implemented, for finding all Hamiltonian paths and cycles in directed or undirected graphs.

In the case of weighted undirected graphs, the travelling salesman problem is considered. The exact (brute force) algorithm based on DFS is given. Several heuristic algorithms are presented: the nearest neighbor algorithm, the repeated nearest neighbor algorithm, and the sorted edge algorithm. For the metric travelling salesman problem, 2-approximation algorithm is shown, which is based on the minimum spanning tree.

In the case of directed graphs, the topological sorting algorithm is implemented. Two additional functions are given: for transitivity testing of tournaments and for finding a Hamiltonian path in tournaments.

The important part of the work is a range of theorems on Hamiltonian cycles, with many educational examples. The algorithms solving the travelling salesman problem were tested for the complexity and the accuracy.

Keywords: graph theory, Hamiltonian graphs, breadth-first search, depth-first search, topological sorting, travelling salesman problem, metaheuristics, tournament

Spis treści

Spis tabel 4
Spis rysunków 5
Listings 6
1. Wstęp 7
1.1. Cele pracy 7
1.2. Plan pracy 7
2. Wprowadzenie do Pythona 9
2.1. Składnia języka 9
2.1.1. Wcięcia 9
2.1.2. Komentarze 10
2.2. Typy danych 10
2.2.1. Liczby 10
2.2.2. Typ logiczny 10
2.2.3. Łańcuchy znaków 11
2.2.4. Krotki 11
2.2.5. Listy 11
2.2.6. Słowniki 12
2.2.7. Zbiory 12
2.3. Instrukcje sterujące 12
2.3.1. Instrukcja if 12
2.3.2. Pętla for 13
2.3.3. Pętla while 14
2.3.4. Instrukcja break, continue, pass 15
2.4. Funkcje 16
2.4.1. Widoczność zmiennych 16
2.5. Moduły 16
2.6. Klasy 17
2.7. Wyjątki 17
2.8. Dziedziczenie 17
3. Teoria grafów 19
3.1. Grafy skierowane 19
3.2. Grafy nieskierowane 20
3.3. Ścieżki 20
3.4. Cykle 20
3.5. Drzewa 20
3.6. Grafy ważone 20
3.7. Spójność 21
3.8. Grafy hamiltonowskie 21
3.9. Przykłady grafów hamiltonowskich 21
3.9.1. Grafy cykliczne 21
3.9.2. Grafy pełne 21

6:3325636680

3.9.3. Grafy dwudzielne 22
3.9.4. Graf skoczka szachowego 22

4. Implementacja grafów 24
4.1. Obiekty związane z grafami 25
4.2. Interfejs grafów 25

5. Przeszukiwanie grafów 27
5.1. Przeszukiwanie wszerz (BFS) 27
5.2. Przeszukiwanie w głąb (DFS) 29

6. Cykle Hamiltona w grafach nieskierowanych 31
6.1. Algorytmy znajdywania cykli i ścieżek 33
6.1.1. Algorytm rekurencyjny na bazie DFS 33
6.1.2. Algorytm Dharwadkera 34

7. Problem komiwojażera 35
7.1. Algorytmy dokładne 35
7.1.1. Algorytm siłowy 35
7.2. Algorytmy heurystyczne 36
7.2.1. Algorytm najbliższych sąsiadów 37
7.2.2. Algorytm najbliższych sąsiadów z powtórzeniami 38
7.2.3. Algorytm sortowania krawędzi 39
7.3. Algorytmy z metaheurystykami 41
7.3.1. Symulowane wyżarzanie 42
7.3.2. Przeszukiwanie z zakazami 42
7.3.3. Algorytm mrówkowy 43
7.4. Odmiany problemu komiwojażera 44
7.4.1. Algorytm 2-aproksymacyjny dla metrycznego problemu komiwojażera 44

8. Cykle Hamiltona w grafach skierowanych 47
8.1. Sortowanie topologiczne 47
8.1.1. Sortowanie topologiczne z wykorzystaniem DFS 48
8.2. Turnieje 48
8.2.1. Sprawdzanie tranzytywności turnieju 52
8.2.2. Wyznaczanie ścieżki Hamiltona w turnieju 52

9. Podsumowanie 53

A. Kod źródłowy dla krawędzi i grafów 54
A.1. Klasa Edge 54
A.2. Klasa Graph 55

B. Testy dla problemu komiwojażera 59
B.1. Porównanie wydajności algorytmów dla problemu komiwojażera . 59
B.2. Porównanie wydajności algorytmów dla metrycznego problemu komiwojażera 59
B.3. Porównanie cykli algorytmów dla metrycznego problemu komiwojażera 64

C. Testy dla grafów skierowanych 68
C.1. Testy wydajności znajdywania ścieżki Hamiltona w turnieju 68
C.2. Testy wydajności sprawdzania tranzytywności w turnieju 68

Bibliografia 70

7:6034317414

Spis tabel

2.1 Podział typów liczbowych. 10
2.2 Operacje arytmetyczne. 11
2.3 Operacje logiczne. 11
2.4 Operacje na łańcuchach. 12
2.5 Operacje na krotkach. 13
2.6 Operacje na listach. 14
2.7 Operacje na słownikach. 15
2.8 Operacje na zbiorach. 15
2.9 Podstawowe instrukcje do obsługi wyjątków. 17

4.1 Interfejs grafów. 26

B.1 Test cyklu Hamiltona dla grafów pełnych z unikalnymi wagami. . . . 62
B.2 Test cyklu Hamiltona dla grafów pełnych metrycznych. 62

8:3611745410

Spis rysunków

5.1 Przeglądanie grafu algorytmem BFS. 28
5.2 Przeglądanie grafu algorytmem DFS. 29

8.1 Turniej tranzytywny dla czterech wierzchołków. 49
8.2 Turniej z jednym źródłem dla czterech wierzchołków. 50
8.3 Turniej z jednym ujściem dla czterech wierzchołków. 50
8.4 Turniej z cyklem Hamiltona dla czterech wierzchołków. 50
8.5 Turniej tranzytywny z pięcioma wierzchołkami. 51
8.6 Turniej eulerowski z pięcioma wierzchołkami. 51

B.1 Wykres wydajności algorytmu najbliższych sąsiadów. 60
B.2 Wykres wydajności algorytmu najbliższych sąsiadów z powtórzeniami. 60
B.3 Wykres wydajności algorytmu sortowania krawędzi. 61
B.4 Wykres wydajności algorytmu siłowego. 61
B.5 Porównanie algorytmów TSP dla grafów pełnych z unikalnymi wagami. 63
B.6 Wykres wydajności algorytmu minimalnego drzewa rozpinającego. . . 63
B.7 Porównanie algorytmów TSP dla grafów metrycznych. 64
B.8 Cykl według algorytmu najbliższego sąsiada dla grafu metrycznego. . 65
B.9 Cykl według algorytmu najbliższego sąsiada z powtórzeniami dla grafu metrycznego. 65
B.10 Cykl według algorytmu sortowania krawędzi dla grafu metrycznego. . 66
B.11 Cykl według algorytmu z wykorzystaniem minimalnego drzewa rozpinającego dla grafu metrycznego. 66
B.12 Minimalne drzewo rozpinające dla grafu metrycznego. 67

C.1 Test wydajności metody find_hamiltonian_path. 69
C.2 Test wydajności metody is_transitive 69

9:5067823081

Listings

2.1 Przykład systemu wcięć w Pythonie. 9
2.2 Instrukcja warunkowa **if**. 13
2.3 Przykłady użycia pętli **for**. 13
2.4 Przykłady użycia pętli **while**. 14
2.5 Przykłady użycia **break**, **continue**, **pass**. 15
2.6 Przykłady funkcji. 16
2.7 Przykład widoczności zmiennej. 16
2.8 Przykład deklaracji klasy w Pythonie. 17
2.9 Przykład dziedziczenia wielokrotnego. 17
5.1 Moduł bfs. 27
5.2 Moduł dfs. 30
6.1 Moduł hamilton2. 33
7.1 Moduł tspbf. 35
7.2 Moduł tspnn. 37
7.3 Moduł tsprnn. 38
7.4 Moduł tspse. 40
7.5 Moduł tspmst. 45
8.1 Moduł sorttop. 48
8.2 Funkcja testująca tranzytywność turnieju. 52
8.3 Znajdowanie ścieżki Hamiltona w turnieju. 52
A.1 Moduł edges. 54
A.2 Moduł graphs. 55

10:7439995455

1. Wstęp

Tematem niniejszej pracy jest badanie grafów hamiltonowskich, czyli grafów zawierających cykl Hamiltona. Jest to zamknięta ścieżka przechodząca przez każdy wierzchołek grafu dokładnie jeden raz. Będą rozważane również grafy półhamiltonowskie, które zawierają (otwartą) ścieżkę Hamiltona. Dokładne definicje tych pojęć zostaną podane w rozdziale 3.

1.1. Cele pracy

Praca magisterska powstała w celu dydaktycznym, aby przedstawić algorytmy dotyczące grafów hamiltonowskich w języku Python. Tematyka była podejmowana w wielu artykułach oraz książkach, ale nie wszystkie algorytmy posiadały pseudokod lub implementację w innych językach programowania, która byłaby łatwo dostępna. Dzięki tej pracy, algorytmy zostały zebrane i napisane w jednolity sposób w języku Python. Algorytmy zaimplementowano tak, aby ich wydajność odpowiadała przewidywaniom teorii.

Istotną częścią pracy są wyniki teoretyczne, czyli twierdzenia matematyczne dające wgląd w strukturę rodziny grafów hamiltonowskich. Zadziwiające jest bogactwo sposobów, na jakie próbowano rozwiązać trudne problemy z tego obszaru badań.

Innym powodem napisania tej pracy było stworzenie implementacji szeregu algorytmów przybliżonych, rozwiązujących problem komiwojażera. Jest to problem o szerokim praktycznym zastosowaniu. Na co dzień ludzie korzystają z różnych środków transportu, aby przemieścić się z jednego miasta do drugiego w sposób najbardziej ekonomiczny, tzn. najszybszy lub najtańszy. Algorytmy dla problemu komiwojażera pomagają rozwiązać ten problem wyznaczając cykl Hamiltona o minimalnej wadze. Spotykamy tutaj różne techniki informatyczne ogólnego zastosowania.

Zaimplementowane algorytmy rozszerzają bibliotekę grafową rozwijaną w Instytucie Fizyki UJ [1]. Sama praca magisterska może być jakby kompendium wiedzy o grafach hamiltonowskich. Przy jej tworzeniu korzystano z wielu książek i artykułów z teorii grafów. Podstawowe podręczniki w języku polskim poświęcone częściowo lub w całości grafom to prace Deo [2], Wilsona [3], Wojciechowskiego i Pienkosza [4], oraz klasyczna "biblia algorytmów" Cormena, Leisersona, Rivesta i Steina [5].

1.2. Plan pracy

Rozdział 1 wyjaśnia cele i motywację niniejszej pracy. W rozdziale 2 omówiono podstawowe składniki języka Python, wykorzystywanego do implemen-

11:4834598373

tacji algorytmów. Rozdział 3 zawiera najważniejsze definicje z teorii grafów potrzebne do zrozumienia algorytmów. Implementację grafów omówiono w rozdziale 4. Implementację dwóch algorytmów przeszukiwania grafów zaprezentowano w rozdziale 5. Cykle Hamiltona w grafach nieskierowanych zbadano w rozdziałach 6 (twierdzenia matematyczne i algorytmy dokładne) i 7 (problem komiwojażera). Dla grafów skierowanych, w rozdziale 8 zebrano twierdzenia matematyczne, oraz omówiono turnieje. Rozdział 9 zawiera podsumowanie pracy.

12:1280901808

2. Wprowadzenie do Pythona

Python jest obiektowo-zorientowanym językiem programowania wysokiego poziomu, zawierającym bogatą bibliotekę standardową [6]. Składnia języka jest przejrzysta, a kod napisany w tym języku, w porównaniu z innymi językami programowania, jest o wiele krótszy i bardzej czytelny. Python umożliwia pisanie programów w metodologii programowania obiektowego, proceduralnego, a także funkcyjnego. Dodatkowym atutem Pythona jest automatyczne zarządzanie pamięcią, co eliminuje wiele potencjalnych błędów i ułatwia pracę programisty. Będąc językiem z typami dynamicznymi, Python jest często używany jako język skryptowy. Interpretery Pythona dostępne są dla większości systemów operacyjnych. Kod programów napisanych w Pythonie jest w dużej mierze niezależny od platformy sprzętowej.

2.1. Składnia języka

Proste programy napisane w języku Python nie wymagają klas czy funkcji. Do budowy większych programów wykorzystuje się funkcje, moduły, klasy, oraz techniki programowania obiektowego. Przy tworzeniu zaawansowanych algorytmów, często mogą się przydać wspierane przez Python elementy programowania funkcyjnego. W przypadku nadużywania przez programistę tych elementów, czytelność kodu może ulec pogorszeniu.

2.1.1. Wcięcia

Podstawowy model programowania w języku Python jest zbliżony do modelu innych języków programowania takich jak np. Java lub C++. Znaczącą różnicą jest system wcięć, który służy do rozdzielania bloków kodu, a jego stosowanie jest obowiązkowe. Na końcu każdej linii kodu nie trzeba używać średników. Przykład wcieć pokazuje listing 2.1

Listing 2.1. Przykład systemu wcięć w Pythonie.

```
n = 3
if n < 0:
    print "Liczba ujemna"
elif n == 0:
    print "Liczba zero"
else:
    print "Liczba dodatnia"
    if n % 2 == 0:
        print "Liczba parzysta"
    else:
        print "Liczba nieparzysta"
```


13:1140823173

2.1.2. Komentarze

Komentarze służą do oddzielenia kodu interpretowanego od uwag programisty. W języku Python są dwa typy komentarzy: jednoliniowe i wieloliniowe. Komentarze jednoliniowe zaczynają się od znaku hash (#) i kończą wraz z końcem wiersza, natomiast komentarze wieloliniowe można wstawić pomiędzy potrójnymi znakami apostrofu (''') lub cudzysłowiu (""""). Wieloliniowy komentarz użyty na początku modułu, klasy lub funkcji, traktowany jest jako łańcuch dokumentacyjny.

2.2. Typy danych

W języku Python nie zmienne, ale wartości posiadają typ, tak więc nie ma potrzeby definiowania typu zmiennej. Python jest językiem z typami dynamicznymi, co oznacza, że dana zmienna może w trakcie działania programu przechowywać wartości różnych typów. Poniższy dział prezentuje opis standardowych typów danych zawartych w języku Python.

2.2.1. Liczby

Liczby mogą być reprezentowane w różny sposób. W zależności od zastosowania oraz potrzeb, liczby można przechowywać w zmiennych o różnych typach. W niektórych przypadkach np. przy iterowaniu pętli **for** lub przy numerowaniu stron, wykorzystuje się tylko liczby całkowite. W innych przypadkach, jak np. przy podawaniu prędkości lub ceny za dany produkt, potrzebna jest liczba zmiennoprzecinkowa. Python posiada kilka wbudowanych typów liczbowych. Zestaw typów liczbowych zawarty jest w tabeli 2.1, natomiast operacje arytmetyczne przedstawia tabela 2.2.

Tabela 2.1. Podział typów liczbowych.

Typ	Opis	Przykład
int	liczba całkowita	5, -100
long	liczba całkowita długa	100200300400L
float	liczba zmiennoprzecinkowa	1.123, -10.43
complex	liczba zespolona	1.02+10j

2.2.2. Typ logiczny

Typ logiczny został wprowadzony do języka Python w wersji 2.3. Wtedy to dodano stałe True oraz False. Stałe te wprowadzono już wcześniej, lecz reprezentowały one zwykłe wartości 1 i 0. Obiekt nowego typu nazwano **bool**. Dzięki tym zmianom poprawiła się czytelność kodu. Przykładowo, jeśli funkcja kończy się przez **return** True, to oczywiste jest, że mamy do czynienia z wartością logiczną. Inaczej jest, gdy napotkamy instrukcję **return** 1 - wtedy można zastanawiać się, co oznacza 1. Przykład na operacjach logicznych przedstawia tabela 2.3.

14:2830219009

Tabela 2.2. Operacje arytmetyczne.

Operacja	**Wynik**
a + b	suma a i b
a − b	różnica a i b
a ∗ b	iloczyn a i b
a / b	iloraz a i b
a % b	reszta z dzielenia a / b (modulo)
a ∗∗ b	a do potęgi b
pow(a, b)	a do potęgi b
−a	negacja a
+a	a niezmienione
abs(a)	wartość bezwzględna a
int(a)	konwersja na liczbę całkowitą
long(a)	konwersja na długą liczbę całkowitą
float(a)	konwersja na liczbę zmiennoprzecinkową
complex(real, imag)	liczba rzeczywista z częścią rzeczywistą real i częścią urojoną imag

Tabela 2.3. Operacje logiczne.

Operacja	**Opis**	**Wynik**
a **or** b	alternatywa	jeśli a jest fałszem, to b, w przeciwnym razie a
a **and** b	koniunkcja	jeśli a jest fałszem, to a, w przeciwnym razie b
not a	negacja	jeśli a jest fałszem, to True, w przeciwnym razie False

2.2.3. Łańcuchy znaków

Oprócz liczb, w Pythonie można manipulować ciągami znaków (stringi). Do ich oznaczenia służy pojedyńczy lub podwójny apostrof. W przypadku gdy chcemy, żeby nasz tekst zawierał specjalny znak końca linii, należy użyć potrójnych apostrofów. Przykład operacji na stringach pokazuje tabela 2.4.

2.2.4. Krotki

Krotki są sekwencją obiektów dostępnych niezależnie. W Pythonie są one podstawowym typem danych. Raz stworzonej krotki nie można zmienić. Do poszczególnego elementu w krotce można się odwołać poprzez jego indeks. Przykład operacji na krotkach w tabeli 2.5.

2.2.5. Listy

Listy, tak jak krotki, są sekwencjami danych zawierającymi indeksy rozpoczynające się od zera. Jeśli lista `L` jest niepusta, to `L[0]` jest jej pierwszym elementem. Listy są bardzo elastyczną strukturą danych, mogą się rozrastać i kurczyć, oraz mogą one być zagnieżdżane na dowolną głębokość. Przykład operacji na listach przedstawia tabela 2.6.

15:1098776940

Tabela 2.4. Operacje na łańcuchach.

Operacja	Znaczenie
S = "" ; S = ''	pusty napis
S = "abc"	stworzenie napisu
S = **str**(word)	stworzenie napisu
len(S)	długość napisu
S1 + S2	łączenie napisów
3 * S, S * 4	powtarzanie
S[i]	indeksowanie
S[i:j]	wycięcie
S2 = S1[:]	kopiowanie
S2 = **str**(S1)	kopiowanie
for i **in** S:	iterowanie
S1 **in** S2	zawieranie
S1 **not in** S2	
S.join(iterable)	sklejanie
del S	usuwanie

2.2.6. Słowniki

Słowniki są bardzo podobne do krotek i list. W przypadku krotek i list do danych należy odwołać się po numerze indeksu, co jest proste i wydajne. Jeśli zajdzie potrzeba np. sprawdzenia, czy dany element znajduje się w kolekcji, należy taką kolekcję przeszukać co już nie jest wydajne. Dlatego rozwiązaniem w słownikach są tablice asocjacyjne, które pozwalają przechowywać pary (klucz, wartość). Odwołanie się do elementu słownika następuje więc poprzez klucz, którym mogą być liczby, ciągi znaków lub inne obiekty niezmienne. Próba pobrania nieistniejącego klucza jest błędem. Słowniki mogą się rozrastać i kurczyć, oraz mogą one być zagnieżdżane na dowolną głębokość. Przykład operacji na słownikach ilustruje tabela 2.7.

2.2.7. Zbiory

Zbiory dają możliwość eliminacji duplikatów i przechowywania nieuporządkowanych elementów z unikalnymi wartościami. Wspierają one również operacje arytmetyczne na zbiorach takie jak np. część wspólna lub różnica. Przykład operacji na zbiorach w tabeli 2.8.

2.3. Instrukcje sterujące

Python zawiera instrukcje sterujące, podobnie jak inne języki programowania. Do instrukcji sterujących należą pętle i instrukcje warunkowe.

2.3.1. Instrukcja if

Instrukcja warunkowa **if** służy do odpowiedniego wykonania algorytmu, zgodnie z określonym w niej warunkiem. Dzięki wyrazom **if**, **elif**, **else** można

16:7387396348

Tabela 2.5. Operacje na krotkach.

Operacja	Znaczenie
T = () ; T = **tuple**()	pusta krotka
T = (0,)	krotka jednopozycyjna z obowiązkowym nawiasem
T = (2, 4, 6, 8)	krotka czteropozycyjna ze zbędnymi nawiasami
T = **tuple**(sequence)	krotka z sekwencji
T[i]	indeks
T[i:j]	nowa krotka (wycinek)
len(T)	długość
T1 + T2	łączenie
T * 3, 4 * T	powtórzenie
for item **in** T:	iteracja
item **in** T	zawieranie
item **not in** T	
del T	usuwanie
(a, b) = (1, 3)	podstawianie

w łatwy sposób zastąpić instrukcje switch i case, których w Pythonie nie ma. Przykład użycia instrukcji warunkowej **if** pokazuje listing 2.2.

Listing 2.2. Instrukcja warunkowa **if**.

```
if condition1:
    # lista instrukcji
elif condition2:    # opcjonalnie
    # lista instrukcji
elif condition3:    # opcjonalnie
    # lista instrukcji
...
else:    # opcjonalnie
    # lista instrukcji
```

W instrukcji warunkowej dwukropki na końcu wierszy są obowiązkowe, tak samo jak wcięcia. Bloki **elif** oraz **else** nie są obowiązkowe i instrukcja **if** może istnieć bez nich. Warunki nie muszą się wykluczać, szukany jest pierwszy spełniony warunek, po którym blok jest wykonywany do następnego **elif** lub **else**. Instrukcja warunkowa nie może być pusta, a w przypadku braku jej instrukcji można użyć **pass**.

2.3.2. Pętla for

W języku Python, pętla **for** różni się od tej używanej w innych językach programowania. Iteracji nie prowadzi się od wartości do wartości, lecz po elementach sekwencji np. listy lub krotki. Przykłady użycia pętli przedstawia listing 2.3.

Listing 2.3. Przykłady użycia pętli **for**.

```
a = [10, 20, 30, 40]
for i in a:
    print i
for i in xrange(10):    # wypisanie liczb od 0 do 9
```


17:1047991779

Tabela 2.6. Operacje na listach.

Operacja	Znaczenie
L = [] ; L = **list**()	pusta lista
L = [3, 2.4]	lista dwupozycyjna
L = **list**(sequence)	lista z sekwencji
len(L)	długość listy
L1 + L2	łączenie
L * 3 ; 4 * L	powtarzanie
L[i]	indeks
L[i:j]	wycinek
L1[i:j] = L2	podstawienie pod wycinek
L[i] = item	nowy element
L1 = **list**(L2)	kopiowanie
L1 = L2[:]	kopiowanie
for item **in** L:	iterowanie
item **in** L	zawieranie
item **not in** L	
del L	usuwanie całej listy
del L[i]	usuwanie elementu z listy
del L[i:j]	usuwanie wycinka

```
    print i
```

2.3.3. Pętla while

Pętla **while** działa podobnie jak w innych językach programowania. Wewnątrz jej instrukcji można stosować instrukcję natychmiastowego wyjścia z pętli **break** oraz instrukcję kontynuacji pętli **continue**. Przykłady użycia są zamieszczone w listingu 2.4.

Listing 2.4. Przykłady użycia pętli **while**.

```
# pierwszy przyklad
x = 5
while x > 0:
    print x
    x = x - 1

# petla nieskonczona
while True:
    print "Tekst bedzie wypisywany w nieskonczonosc"

# kolejny przyklad
word = "abcdefgh"
while word:
    print word
    word = word[1:]
```


18:7886253933

Tabela 2.7. Operacje na słownikach.

Operacja	Znaczenie
D = {} ; D = **dict**()	pusty słownik
D = {2: "r", 3: "p"}	słownik z dwoma kluczami
len(D)	liczba elementów słownika
D[key]	dostęp do wartości
D[key] = value	dodanie wartości
D = **dict**(T)	tworzenie z krotki zawierającej pary
D1 = **dict**(D2)	kopiowanie
D1 = D2.copy()	kopiowanie
key **in** D	zawieranie
key **not in** D	
for key **in** D:	iterowanie po kluczach
del D	usuwanie słownika
del D[key]	usuwanie klucza

Tabela 2.8. Operacje na zbiorach.

Operacja	Znaczenie
Z = **set**() ; Z = **set**([])	pusty zbiór
for item **in** Z:	iterowanie
len(Z)	liczba elementów zbioru
item **in** Z	należenie elementu do zbioru
item **not in** Z	
Z1.issubset(Z2)	czy zbiór Z1 należy do zbioru Z2
Z1.issuperset(Z2)	czy zbiór Z2 należy do zbioru Z1
Z1.union(Z2)	suma zbiorów
Z1.difference(Z2)	różnica zbiorów
Z1.intersection(Z2)	iloczyn zbiorów
Z1.symmetric_difference(Z2)	różnica symetryczna zbiorów
Z2 = Z1.copy()	kopiowanie zbiorów

2.3.4. Instrukcja break, continue, pass

Instrukcja **break** powoduje wyjście z najbliżej zagnieżdzonej pętli **for** lub **while**. Instrukcja **continue** pooduje przejście do kolejnej iteracji pętli. Obie instrukcje zostały zapożyczone z języka C. Instrukcja **pass** nie robi nic i służy wszędzie tam, gdzie musi zostać zachowana składnia. Przykład użycia przedstawia listing 2.5.

Listing 2.5. Przykłady użycia **break**, **continue**, **pass**.

```
while loop1:
    instrukcje        # cialo petli
    if condition1:
        break         # wyjscie z petli, opcjonalne
    if condition2:
        continue      # przejscie do loop1, opcjonalne
    if condition3:
        pass    # nie robi nic
else:    # opcjonalne
```


19:1033865311

```
instrukcje    # wykonane, jesli petli nie zakonczylo break
```

2.4. Funkcje

W Pythonie można stosować funkcje do grupowania fragmentów kodu po to, żeby można było je później wywołać z dowolnego miejsca w programie. Takie rozwiązanie zapobiega duplikowaniu fragmentów kodu i sprawia, że kod staje się czytelniejszy. Funkcje deklarowane są za pomocą słowa **def**. Aby funkcja zwracała jakąś wartość, należy użyć instrukcji **return**. W Pythonie mamy możliwość przypisania do funkcji argumentów domyślnych, które zostaną zastosowane w przypadku, gdy przy wywołaniu funkcji nie zostanie podana wartość. Przykład funkcji pokazuje listing 2.6.

Listing 2.6. Przykłady funkcji.

```
def adder(x = 2, y = 2):
    """Dodawanie argumentow."""
    return x + y
```

2.4.1. Widoczność zmiennych

Język Python wykorzystuje tzw. przestrzeń nazw (ang. *namespace*). Każda zmienna żyje w określonej przestrzeni nazw. Funkcje definiują zakres lokalny zmiennych, a moduły definiują zakres globalny. Należy unikać stosowania zmiennych globalnych, ponieważ zmniejsza to czytelnośc, a także program jest trudniejszy w debugowaniu. Przykład widoczności zmiennej przedstawia listing 2.7.

Listing 2.7. Przykład widoczności zmiennej.

```
a = 23                         # zmienna globalna (automatycznie)

def action():
    #print a                # blad, poniewaz a bedzie lokalne
    a = 10                  # zmienna lokalna
    print a                 # 10, odwolanie do lokalnej a

action()
print a                        # 23, globalna zmienna a
```

2.5. Moduły

Moduły to osobne pliki, w których są zawarte definicje obiektów oraz instrukcje. Definicje danego modułu można importować, a taki import tworzy przestrzeń nazw.

20:1154170650

2.6. Klasy

Klasa umożliwia zdefiniowanie swojego własnego typu danych, dostosowanego do potrzeb programisty. Użycie tej techniki w odpowiedni sposób może zwiększyć czytelność programu oraz jego modularyzację. Klasy obsługują mechanizm dziedziczenia, czyli ponownego wykorzystaniu kodu i jego dostosowania do potrzeb. Klasę tworzy się używając słowa **class**, a po nim nazwę klasy. Konstruktorem w klasie jest metoda __init__ , która tak jak każda inna metoda czy funkcja, może posiadać argumenty domyślne. Każda metoda niestatyczna w klasie posiada pierwszy argument **self**, będący referencją do obiektu. Pola w klasie nie mają modyfikatorów dostępu i są publiczne. Przykład deklaracji klasy pokazuje listing 2.8

Listing 2.8. Przykład deklaracji klasy w Pythonie.

```
class FirstClass:

    def __init__(self, a = 0):
        self.data = set()
        self.data.add(a)

    def show(self):
        print self.data
```

2.7. Wyjątki

Wyjątek jest zdarzeniem, które może spowodować zmianę przebiegu działania programu. Nawet poprawne składniowo wyrażenie może spowodować błąd podczas jego wykonania. Takie zdarzenie niekoniecznie musi zakończyć działanie programu. Język Python posiada mechanizm obsługi takich zdarzeń (wyjątków). Przykład instrukcji do obsługi wyjątków zawiera tabela 2.9.

Tabela 2.9. Podstawowe instrukcje do obsługi wyjątków.

Operacja	Znaczenie
raise	ręczne wywołanie wyjątku
try/except/else/finally	przechwytywanie wyjątków
assert	warunkowe wywołanie wyjątku
with/as	implementacja menedżerów kontekstu

2.8. Dziedziczenie

Dziedziczenie jest techniką umożliwiającą stworzenie nowego typu na bazie już istniejącego. Python pozwala na dziedziczenie po więcej niż jednej klasie tzw. dziedziczenie wielokrotne. Przykład dziedziczenia zawiera listing 2.9

Listing 2.9. Przykład dziedziczenia wielokrotnego.

21:1021402442

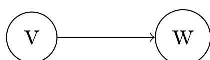
```python
class Class1:

    def __init__(self):
        pass

class Class2:

    def __init__(self):
        pass

class Class3(Class1, Class2):     # dziedziczenie po Class1 i Class2

    def __init__(self):
        pass
```

3. Teoria grafów

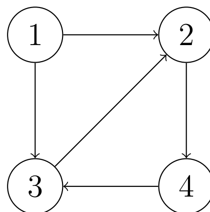
Teoria grafów jest gałęzią informatyki i matematyki, do której powstania przyczyniło się słynne zagadnienie mostów królewieckich, rozwiązane przez Leonharda Eulera w roku 1736. Wiele praktycznych problemów można sformułowac w języku teorii grafów, dlatego nie dziwi popularność i stały rozwój tej dziedziny wiedzy. Z nastaniem komputerów pojawiły się nowe zadania do rozwiązania, ale też możliwe są nowe metody szukania rozwiązań. Sięgając do różnych źródeł można natknąć się na różne definicje, dlatego w tym rozdziale opiszemy definicje używane w niniejszej pracy. Będziemy się opierać głównie na podręczniku Cormena [5], ale z pewnymi zmianami.

3.1. Grafy skierowane

Graf skierowany lub *digraf* (ang. *directed graph*) jest to struktura składająca się ze zbioru wierzchołków V, połączonymi ze sobą krawędziami ze zbioru E, gdzie krawędzie pokazują kierunek ruchu po grafie. Ruch możliwy jest tylko w wyznaczonym kierunku. Formalnie, graf skierowany jest uporządkowaną parą $G = (V, E)$. Krawędź jest to uporządkowana para dwóch wierzchołków (v, w), gdzie wierzchołek v jest wierzchołkiem początkowym, a wierzchołek w jest wierzchołkiem końcowym. Wierzchołek w jest wierzchołkiem sąsiednim względem wierzchołka v, jeśli w grafie G istnieje krawędź (v, w). Relację taką nazywamy relacją sąsiedztwa.



Strzałka, która reprezentuje krawędź, skierowana jest od wierzchołka początkowego do wierzchołka końcowego. Najczęściej wierzchołki stosowane są do przedstawiania obiektów, a krawędzie między nimi reprezentują relację między tymi obiektami. Przykładowo obiektami mogą być miasta, a krawędziami połączenia autobusowe między tymi miastami.



Stopniem wyjściowym $\text{outdeg}(v)$ nazywamy liczbę krawędzi wychodzących z wierzchołka v. *Stopniem wejściowym* $\text{indeg}(v)$ nazywamy liczbę krawędzi wchodzących do wierzchołka v.

23:3616474694

3.2. Grafy nieskierowane

Graf nieskierowany $G = (V, E)$ jest to struktura składająca się ze skończonego zbioru wierzchołków V i zbioru krawędzi E, gdzie kierunek połączeń między wierzchołkami nie jest istotny, ważne jest tylko istnienie połączenia. Krawędź grafu nieskierowanego zawiera parę dwóch wierzchołków $\{v, w\}$. W grafie nieskierowanym relacja między sąsiadami jest symetryczna. Każda krawędź może zawierać dokładnie dwa różne wierzchołki, ponieważ pętle są niedozwolone.

Stopień wierzchołka $\deg(v)$ w grafie nieskierowanym jest to liczba krawędzi połączonych z wierzchołkiem v.

3.3. Ścieżki

Ścieżką 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 wynosi n. Ścieżka składająca się z jednego wierzchołka ma długość zero. Jeżeli istnieje ścieżka z s do t, to mówimy, że t jest *osiągalny* z s. *Ścieżka prosta* to ścieżka, w której wszystkie wierzchołki są różne.

3.4. Cykle

Cykl jest to ścieżka, w której pierwszy i ostatni wierzchołek są takie same, $v_0 = v_n$. *Cykl prosty* jest to cykl, w którym wszystkie wierzchołki są różne, z wyjątkiem ostatniego. Graf niezawierający cykli prostych nazywamy *acyklicznym*. Graf skierowany acykliczny nazywamy *dagiem* (ang. *directed acyclic graph*).

3.5. Drzewa

Drzewo (ang. *tree*) jest to graf nieskierowany, spójny i acykliczny. *Drzewo rozpinające* (ang. *spanning tree*) grafu nieskierowanego G jest to drzewo, które zawiera wszystkie wierzchołki grafu G i jest podgrafem grafu G. *Drzewo ukorzenione* jest takim drzewem, w którym wyróżniono wierzchołek główny zwany *korzeniem* (ang. *root*). Dla dowolnej ścieżki prostej wychodzącej od korzenia używa się takich pojęć jak: przodek, potomek, rodzic, dziecko.

3.6. Grafy ważone

Graf ważony (ang. *weighted graph*) to graf zawierający krawędzie z wagami. Wagi to wartości liczbowe, które mogą reprezentować odległość, przepustowość kanału, koszt, itd. Wagę krawędzi (s, t) lub $\{s, t\}$ oznaczamy przez $w(s, t)$. *Wagą ścieżki/cyklu/drzewa* w grafie ważonym nazywamy sumę wag odpowiednich krawędzi.

24:6084030895

3.7. 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.

3.8. Grafy hamiltonowskie

Ścieżka Hamiltona to ścieżka przechodząca przez każdy wierzchołek grafu dokładnie raz. *Cykl Hamiltona* jest to cykl prosty przebiegający przez wszystkie wierzchołki grafu dokładnie jeden raz, oprócz ostatniego wierzchołka. *Graf hamiltonowski* (ang. *Hamiltonian graph*) to graf zawierający cykl Hamiltona [8]. *Graf półhamiltonowski* to graf zawierający ścieżkę Hamiltona. Problem znajdowania ścieżki i cyklu Hamiltona jest NP-zupełny.

Jeżeli dla każdej pary wierzchołków grafu G istnieje ścieżka Hamiltona łącząca te wierzchołki, to graf G nazywamy *hamiltonowsko spójnym*. Przykładem są grafy pełne K_3, K_4, K_5.

Stwierdzenie: Jeżeli graf ma cykl Hamiltona, to ma również ścieżkę Hamiltona [4]. Wystarczy usunąć jedną krawędź z cyklu.

3.9. Przykłady grafów hamiltonowskich

Pewne klasy grafów zawierają tylko grafy hamiltonowskie. Pokażemy wybrane przykłady takich klas.

3.9.1. Grafy cykliczne

Graf cykliczny C_n (ang. *cycle graph*) jest to graf prosty nieskierowany $G = (V, E)$, który składa się z pojedyńczego cyklu prostego, przy czym $|V| = |E| = n$ dla $n > 2$ [9]. Graf cykliczny C_n jest hamiltonowski dla $n > 2$. Dla każdego wierzchołka $v \in V$ zachodzi $\deg(v) = 2$.

Graf cykliczny skierowany (ang. *directed cycle graph*) jest skierowaną wersją grafu cyklicznego, gdzie wszystkie krawędzie są skierowane w tym samym kierunku. Powstaje w ten sposób skierowany cykl Hamiltona.

3.9.2. Grafy pełne

Graf pełny K_n (ang. *complete graph*) jest to graf prosty nieskierowany $G = (V, E)$, w którym każda para wierzchołków jest połączona krawędzią, przy czym $|V| = n$, $|E| = n(n-1)/2$ [10]. Graf pełny K_n jest hamiltonowski dla $n > 2$.

Liczba różnych cykli: Liczba różnych cykli Hamiltona w grafie pełnym K_n wynosi $(n-1)!/2$ ([4], s. 159). Liczba różnych permutacji wierzchołków wynosi $n!$, ale pewne cykle należy utożsamić. Po pierwsze, cykle różniące się przesunięciem cyklicznym są traktowane jako identyczne (dzielenie przez n).

25:6011841305

Po drugie, cykle obiegane w przeciwnych kierunkach są identyczne (dzielenie przez 2).

Cykle rozłączne krawędziowo: W grafie pełnym K_n o $n \geqslant 3$ wierzchołkach i nieparzystej wartości n jest $(n-1)/2$ krawędziowo rozłącznych cykli Hamiltona ([4], s. 159). Jeden cykl zawiera n krawędzi, a więc $(n-1)/2$ cykli rozłącznych krawędziowo wyczerpuje liczbę wszystkich krawędzi w grafie pełnym. Kolejne cykle budujemy następująco,

$$C_k = [k, k-1, k+1, k-2, \ldots, k+(n-1)/2, k], \tag{3.1}$$

gdzie k zmienia się od 1 do $(n-1)/2$. Operacje w podanym wzorze wykonywane są mod(n).

3.9.3. Grafy dwudzielne

Graf dwudzielny (ang. *bipartite graph*) jest to graf prosty nieskierowany $G = (V, E)$, którego zbiór wierzchołków V można podzielić na dwa rozłączne zbiory V_1 i V_2 tak, że krawędzie nie łączą wierzchołków tego samego zbioru [11].

Twierdzenie: Jeżeli G jest grafem dwudzielnym, to każdy cykl w G ma długość parzystą. Dowód twierdzenia można znaleźć w książce Wilsona [3]. Z twierdzenia wynika, że graf dwudzielny o nieparzystej liczbie wierzchołków nie może być grafem hamiltonowskim.

Graf dwudzielny pełny $K_{r,s}$ jest to graf dwudzielny, w którym istnieją krawędzie pomiędzy wszystkimi parami wierzchołków należących do różnych zbiorów. Jeżeli $|V_1| = r$ i $|V_2| = s$, to $|V| = r + s$, $|E| = rs$.

Twierdzenie: Niech $G = (V, E)$ będzie grafem dwudzielnym i niech $V = V_1 \cup V_2$ będzie podziałem wierchołków G. Jeżeli G ma cykl Hamiltona, to $|V_1| = |V_2|$. Jeżeli G ma ścieżkę Hamiltona, to $||V_1| - |V_2|| \leqslant 1$. Dla grafów dwudzielnych pełnych zachodzą też implikacje w lewo [11].

Liczba cykli Hamiltona: Liczba różnych cykli Hamiltona w grafie $K_{m,m}$ wynosi $(m-1)!m!/2$ ([4], s. 177). Przyjmijmy, że V_1 zawiera wierzchołki parzyste, a V_2 nieparzyste. W cyklu Hamiltona wierzchołki o numerach parzystych i nieparzystych muszą występować naprzemiennie. Można skonstruować $(m!)^2$ takich ciągów, ale pewne cykle należy utożsamić. Po pierwsze, utożsamiamy cykle różniące się przesunięciem cyklicznym co 2 (dzielenie przez m). Po drugie, cykle obiegane w przeciwnych kierunkach są identyczne (dzielenie przez 2).

3.9.4. Graf skoczka szachowego

Problem skoczka szachowego (ang. *knight's tour problem*) polega na obejściu skoczkiem wszystkich pól planszy tak, aby na każdym polu stanąć dokładnie jeden raz [12]. Odpowiada to szukaniu ścieżki Hamiltona w nieskierowanym *grafie skoczka* (ang. *knight's graph*), w którym wierzchołki odpowiadają polom planszy, a krawędzie dozwolonym ruchom skoczka. Jeżeli skoczek

26:5209520922

może po ostatnim ruchu wrócić na pole, z którego zaczynał, to dostajemy cykl Hamiltona.

Problem skoczka szachowego może być rozwiązany w czasie liniowym [13]. Dla planszy $n \times m$ graf skoczka ma parametry $|V| = nm$, $|E| = 4mn - 6(m + n) + 8$. Graf skoczka jest grafem dwudzielnym (białe i czarne pola szachownicy).

27:1054935008

4. Implementacja grafów

W pracy korzystamy z implementacji grafów w języku Python rozwijanej w Instytucie Fizyki UJ [1]. Implementacja bazuje na klasie Edge dostarczającej krawędzie skierowane, oraz klasie Graph, dostarczającej grafy proste z wagami. Interfejs grafów przedstawimy w ramach sesji interaktywnej.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> N = 3                            # liczba wierzcholkow
>>> G = Graph(N)                     # utworzenie grafu
# Sprawdzenie czy graf jest skierowany.
>>> G.is_directed()
False
# Dodanie wierzcholkow do grafu.
>>> for node in [0, 1, 2]:
...     G.add_node(node)
# Dodanie krawedzi do grafu.
>>> for edge in [Edge(0, 1, 5), Edge(0, 2, 10), Edge(1, 2, 4)]:
...     G.add_edge(edge)
>>> print G.v(), G.e()
(3, 3)                  # liczba wierzcholkow i krawedzi
>>> G.show()
0 : 1(5) 2(10)
1 : 0(5) 2(4)
2 : 0(10) 1(4)
>>> list(G.iternodes())
[0, 2, 1]               # lista wierzcholkow
>>> list(G.degree(node) for node in G.iternodes())
[2, 2, 2]               # stopnie wierzcholkow
>>> list(G.iteredges())         # lista krawedzi
[Edge(0, 1, 5), Edge(0, 2, 10), Edge(1, 2, 4)]
>>> sorted(edge.weight for edge in G.iteredges())
[4, 5, 10]              # posortowana lista wag krawedzi
>>> sorted(G.iteredges())       # sortowanie wg wag
[Edge(1, 2, 4), Edge(0, 1, 5), Edge(0, 2, 10)]
>>> min(G.iteredges())
Edge(1, 2, 4)           # krawedz o najmniejszej wadze
# Wykorzystanie algorytmu BruteForceTSP z modulu tspbf.
>>> from tspbf import BruteForceTSP
>>> algorithm = BruteForceTSP(G)
>>> algorithm.run()
>>> algorithm.hamilton_cycle
[Edge(0, 1, 5), Edge(1, 2, 4), Edge(2, 0, 10)]
>>> sum(edge.weight for edge in algorithm.hamilton_cycle)
19                      # waga cyklu Hamiltona
```


28:1076115862

4.1. Obiekty związane z grafami

W teorii grafów definiuje się wiele pojęć, które powinny mieć swój odpowiednik w kodzie źródłowym. Przedstawimy pythonowe realizacje wybranych obiektów.

Wierzchołek: Wierzchołki mogą być dowolnymi obiektami pythonowymi, które są hashowalne i porównywalne. Na ogół są to jednak liczby całkowite lub stringi. Liczbami całkowitymi od 0 do $n - 1$ są wierzchołki zapisane w implementacji macierzowej, natomiast w implementacji słownikowej nie ma tego ograniczenia.

Krawędź: Krawędzie skierowane są instancjami klasy Edge(source, target, weight), gdzie source jest wierzchołkiem początkowym, target jest wierzchołkiem końcowym, a weight jest wagą krawędzi. Parametr weight jest opcjonalny i bez jego podania waga wynosi 1.

Graf prosty: Grafy proste są instancjami klasy Graph.

Ścieżka: Z definicji ścieżka jest ciągiem wierzchołków, reprezentowanym przez listę Pythona. Dla grafów ważonych stosujemy również drugą reprezentację, mianowicie listę krawędzi. Dzięki temu łatwo można obliczyć długość ścieżki i jej wagę. Zamiast korzystać z listy krawędzi możemy krawędzie wstawić do osobnego grafu, co daje trzecią możliwą reprezentację.

Cykl: Cykl jest specyficzną ścieżką, więc stosujemy reprezentacje takie jak dla ścieżek. Dla przykładu zapiszemy cykl Hamiltona dla grafu pełnego K_3 z wagami przy użyciu podanych wcześniej konwencji.

```
cycle1 = [0, 1, 2, 0]
cycle2 = [Edge(0, 1, 2), Edge(1, 2, 1), Edge(2, 0, 3)]
cycle_length = len(cycle2)
cycle_weight = sum(edge.weight for edge in cycle2)
```

Drzewo: Drzewo jest grafem, więc jest przechowywane jako instancja klasy Graph. Innym sposobem przechowywania drzewa z korzeniem jest wykorzystanie słownika, gdzie kluczami są wierzchołki, a wartościami ich rodzice (lub krawędzie skierowane prowadzące do rodziców). Wartością przyporządkowaną do korzenia jest None.

4.2. Interfejs grafów

Dla operacji na grafach został zaproponowany interfejs klasy Graph przedstawiony w poniższej tabeli 4.1.

Tabela 4.1. Interfejs grafów w klasie Graph. G oznacza graf, n jest liczbą całkowitą dodatnią.

Operacja	**Znaczenie**
G = Graph(n)	stworzenie grafu nieskierowanego
G = Graph(n, directed=True)	stworzenie grafu skierowanego
G.is_directed()	sprawdzenie czy graf jest skierowany
G.v()	liczba wierzchołków grafu
G.e()	liczba krawędzi grafu
G.add_node(node)	dodawanie wierzchołka do grafu
G.has_node(node)	czy wierzchołek istnieje
G.del_node(node)	usuwanie wierzchołka z krawędziami incydentnymi
G.add_edge(edge)	dodawanie krawędzi do grafu
G.del_edge(edge)	usuwanie krawędzi
G.has_edge(edge)	czy krawędź istnieje
G.weight(edge)	waga krawędzi
G.iternodes()	iteracja wierzchołków
G.iteradjacent(node)	iteracja wierzchołków sąsiednich
G.iteroutedges(node)	iteracja po krawędziach wychodzących
G.iterinedges(node)	iteracja po krawędziach przychodzących
G.iteredges()	iteracja krawędzi
G.show()	wyświetlenie grafu
G.copy()	utworzenie kopii grafu
G.transpose()	transponowanie grafu
G.degree(node)	stopień wierzchołka w grafie nieskierowanym
G.outdegree(node)	stopień wyjściowy wierzchołka
G.indegree(node)	stopień wejściowy wierzchołka

30:7158328141

5. Przeszukiwanie grafów

W tym rozdziale zostaną przedstawione algorytmy przeszukiwania grafów. Przeszukiwaniem grafu jest przejście przez jego krawędzie w taki sposób, aby przejść przez jego wszystkie wierzchołki. Dzięki algorytmom do przeszukiwania grafu można zebrać ważne informacje o jego strukturze. Dwoma podstawowymi algorytmami do przeszukiwania grafu są: przeszukiwanie wszerz (ang. *breadth-first search*, BFS) i przeszukiwanie w głąb (ang. *depth-first search*, DFS). Wiele algorytmów grafowych podczas pracy wykorzystuje BFS lub DFS.

Przedstawimy implementację algorytmów BFS i DFS wg książki Cormena [5], gdzie stan wierzchołków oznaczany jest trzema koloram: WHITE wierzchołek nieodwiedzony, GREY wierzchołek napotkany nieprzetworzony, BLACK wierzchołek napotkany przetworzony.

5.1. Przeszukiwanie wszerz (BFS)

Przeszukiwanie wszerz jest jednym z najprostszych algorytmów przeszukiwania grafu i pierwowzorem wielu innych algorytmów grafowych.

Dane wejściowe: Dowolny graf, opcjonalnie wierzchołek początkowy.

Problem: Przeszukiwanie grafu wszerz.

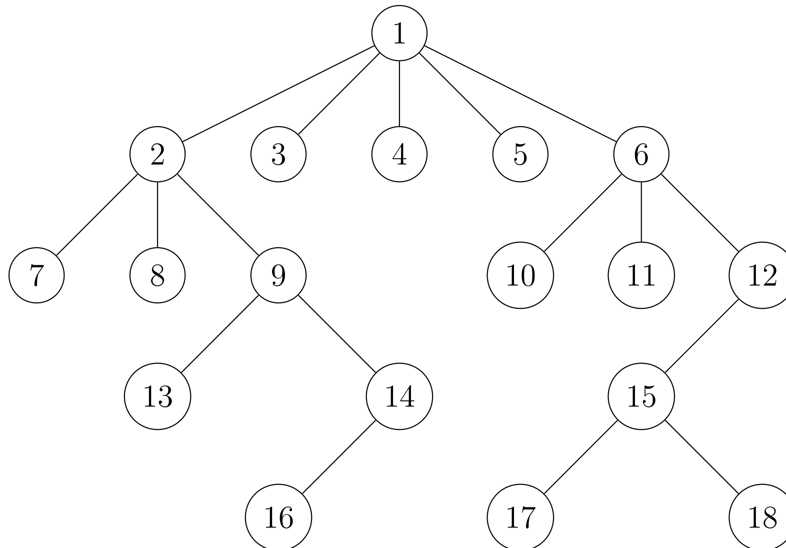
Opis algorytmu: Algorytm rozpoczynamy od dowolnego wierzchołka grafu i oznaczamy wszystkie wierzchołki jako nieodwiedzone. Wybrany wierzchołek wstawiamy do kolejki zawierającej wierzchołki do odwiedzenia. Następnie wyciągamy wierzchołki z kolejki tak długo, aż kolejka nie będzie pusta. Pobrane wierzchołki oznaczamy jako odwiedzone, a ich nieodwiedzonych sąsiadów wstawiamy do kolejki.

Złożoność: Dla macierzy sąsiedztwa złożoność czasowa wynosi $O(V^2)$, natomiast dla list sąsiedztwa złożoność wynosi $O(E + V)$.

Listing 5.1. Moduł bfs.

```
#!/usr/bin/python

from Queue import Queue


class BreadthFirstSearch:
    """Algorytm przeszukiwania wszerz (BFS)."""
```

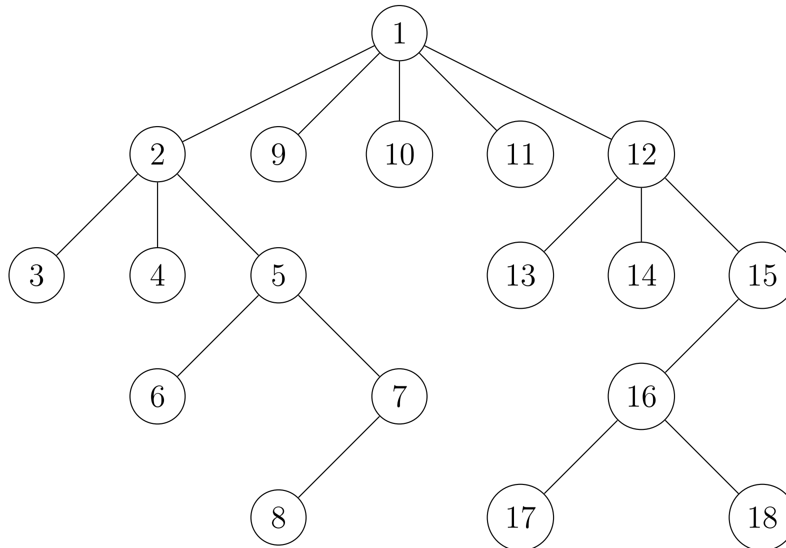

31:4945055946



Rysunek 5.1. Przykład przeglądania grafu algorytmem BFS. Etykiety wierzchołków są zgodne z kolejnością odwiedzania.

```
def __init__(self, graph):
    """Initializacja algorytmu."""
    self.graph = graph
    self.color = dict(((node, "WHITE")
        for node in self.graph.iternodes()))
    self.distance = dict(((node, float("inf"))
        for node in self.graph.iternodes()))
    self.parent = dict(((node, None)
        for node in self.graph.iternodes()))
    self.dag = self.graph.__class__(self.graph.v(), directed=True)

def run(self, source=None, pre_action=None, post_action=None):
    """Uruchomienie algorytmu BFS."""
    if source is not None:
        self._visit(source, pre_action, post_action)
    else:
        for node in self.graph.iternodes():
            if self.color[node] == "WHITE":
                self._visit(node, pre_action, post_action)

def _visit(self, node, pre_action=None, post_action=None):
    """Zwiedzanie spojnej skladowej grafu.."""
    self.color[node] = "GREY"
    self.distance[node] = 0
    self.parent[node] = None
    Q = Queue()
    Q.put(node)  # node jest GREY
    if pre_action:   # po Q.put
        pre_action(node)
    while not Q.empty():
        source = Q.get()
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] == "WHITE":
                self.color[edge.target] = "GREY"
                self.distance[edge.target] = self.distance[source] + 1
```


32:4514184575



Rysunek 5.2. Przykład przeglądania grafu algorytmem DFS. Etykiety wierzchołków są zgodne z kolejnością odwiedzania.

```
                self.parent[edge.target] = source
                self.dag.add_edge(edge)
                Q.put(edge.target)  # target jest GREY
                if pre_action:    # po Q.put
                    pre_action(edge.target)
        self.color[source] = "BLACK"
        if post_action:   # source became BLACK
            post_action(source)
```

5.2. Przeszukiwanie w głąb (DFS)

Algorytm przeszukiwania w głąb jest drugim podstawowym algorytmem do przeszukiwania grafu. Polega na odwiedzaniu krawędzi wychodzących z ostatnio odwiedzonego wierzchołka, z którego jeszcze nie zostały odwiedzone krawędzie.

Dane wejściowe: Dowolny graf, opcjonalnie wierzchołek początkowy.

Problem: Przeszukiwanie grafu w głąb.

Opis algorytmu: Algorytm rozpoczynamy od dowolnego wierzchołka grafu i oznaczamy wszystkie wierzchołki jako nieodwiedzone. Z wybranego przez nas wierzchołka sprawdzamy jego sąsiadów, czy któryś jest jeszcze nieodwiedzony. Jeśli jest, to przechodzimy do niego i ponownie szukamy nieodwiedzonych sąsiadów. Jeśli dany wierzchołek ma już wszystkich odwiedzonych sąsiadów lub nie ma on żadnego sąsiada, to należy przejść do wierzchołka, z którego został on odwiedzony.

Złożoność: Dla macierzy sąsiedztwa złożoność czasowa wynosi $O(V^2)$, natomiast dla list sąsiedztwa złożoność wynosi $O(V + E)$.

33:2195807370

Listing 5.2. Moduł dfs.

```
#!/usr/bin/python

class DepthFirstSearch:
    """Algorytm przeszukiwania w glab z rekurencja."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
        self.graph = graph
        self.color = dict(((node, "WHITE")
            for node in self.graph.iternodes()))
        self.parent = dict(((node, None)
            for node in self.graph.iternodes()))
        self.time = 0    # znacznik czasu
        self.dd = dict()    # czas napotkania wierzcholka
        self.ff = dict()    # czas przetworzenia wierzcholka
        self.dag = self.graph.__class__(self.graph.v(), directed=True)
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v()*2, recursionlimit))

    def run(self, source=None, pre_action=None, post_action=None):
        """Eksploracja grafu."""
        if source is not None:
            self._visit(source, pre_action, post_action)
        else:
            for node in self.graph.iternodes():
                if self.color[node] == "WHITE":
                    self._visit(node, pre_action, post_action)

    def _visit(self, node, pre_action=None, post_action=None):
        """Badanie polczonych ze soba wierzcholkow."""
        self.time = self.time + 1
        self.dd[node] = self.time
        self.color[node] = "GREY"
        if pre_action:    # _visit start
            pre_action(node)
        for edge in self.graph.iteroutedges(node):
            if self.color[edge.target] == "WHITE":
                self.parent[edge.target] = node
                self.dag.add_edge(edge)
                self._visit(edge.target, pre_action, post_action)
        self.time = self.time + 1
        self.ff[node] = self.time
        self.color[node] = "BLACK"
        if post_action:    # node ustawiony na BLACK
            post_action(node)
```

34:9979005414

6. Cykle Hamiltona w grafach nieskierowanych

Wyniki uzyskane dla grafów hamiltonowskich są o wiele słabsze od wyników dla grafów eulerowskich, mimo dużego podobieństwa definicji tych grafów. Jeden z najstarszych wyników Diraca opiera się na intuicji, że graf jest hamiltonowski, kiedy zawiera "wystarczająco dużo" krawędzi. Późniejsze i bardziej wyszukane twierdzenia również są ograniczone do dość gęstych grafów. W twierdzeniach przyjmujemy założenie, że grafy są nieskierowane i spójne.

Twierdzenie (Dirac, 1952): Jeżeli G jest grafem prostym o $n \geqslant 3$ wierzchołkach oraz dla każdego wierzchołka v zachodzi nierówność

$$\deg(v) \geqslant n/2, \tag{6.1}$$

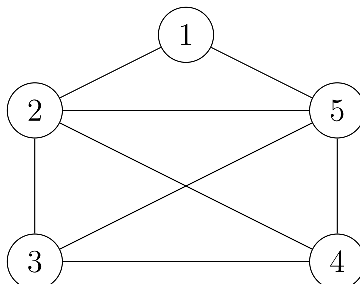
to graf G jest hamiltonowski [14]. Dowód korzystający z twierdzenia Orego podany jest w [4].

Twierdzenie (Ore, 1960): Jeżeli G jest grafem prostym o $n \geqslant 3$ wierzchołkach oraz

$$\deg(s) + \deg(t) \geqslant n \tag{6.2}$$

dla każdej pary niesąsiadujących wierzchołków s, t, to graf G jest hamiltonowski [15]. Dowód można znaleźć w [4].

Twierdzenie Ore jest silniejsze od twierdzenia Diraca, o czym świadczy graf hamiltonowski z rysunku. Nie są spełnione założenia twierdzenia Diraca, bo $\deg(1) = 2 < 5/2 = n/2$. Są spełnione założenia twierdzenia Ore, ponieważ $\deg(1) + \deg(3) = 2 + 3 = 5 = n$, $\deg(1) + \deg(4) = 2 + 3 = 5 = n$.



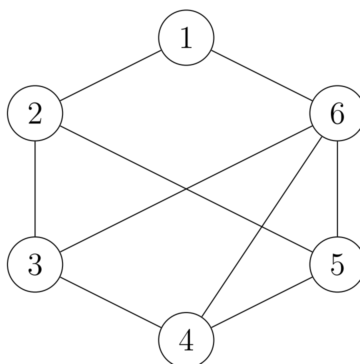
Twierdzenie (Pósa, 1962): Jeżeli wierzchołki grafu prostego G o $n \geqslant 3$ wierzchołkach można ponumerować kolejnymi liczbami naturalnymi w taki sposób, że ich stopnie tworzą ciąg niemalejący oraz

$$\deg(k) > k \text{ dla } 1 \leqslant k < n/2, \tag{6.3}$$

35:3855012820

to graf G jest hamiltonowski [16].

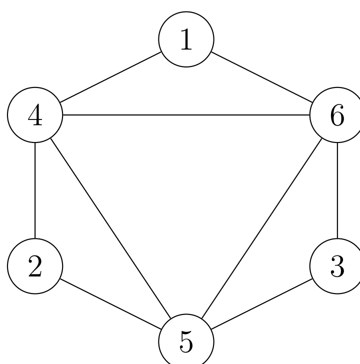
Twierdzenie Pósy jest silniejsze od twierdzenia Ore, o czym świadczy graf hamiltonowski z rysunku. Nie są spełnione założenia twierdzenia Ore, ponieważ $\deg(1) + \deg(3) = 5 < 6 = n$, $\deg(1) + \deg(4) = 5 < n$, $\deg(1) + \deg(5) = 5 < n$. Są spełnione założenia twierdzenia Pósy, ponieważ $n/2 = 3$, $\deg(1) = 2 > 1$, $\deg(2) = 3 > 2$.



Twierdzenie (Chvátal, 1972): Niemalejący ciąg liczb naturalnych $(d_1, d_2, \ldots, d_n)$, $d_n < n$, $n > 3$, jest ciągiem stopni grafu hamiltonowskiego wtedy i tylko wtedy, gdy dla każdego $k < n/2$ zachodzi [17]

$$\deg(k) \leqslant k \Rightarrow \deg(n-k) \geqslant n-k. \tag{6.4}$$

Twierdzenie Chvátala jest silniejsze od twierdzenia Pósy, o czym świadczy graf hamiltonowski z rysunku. Nie są spełnione założenia twierdzenia Pósy, ponieważ $\deg(1) = 2 > 1$ (w porządku), $\deg(2) = 2$ (nie jest większe od 2). Są spełnione założenia twierdzenia Chvátala, ponieważ $\deg(2) = 2 \leqslant 2$ i $\deg(4) = 4 \geqslant 4$.



Twierdzenie Bondy'ego-Chvátala (1976): Niech G będzie grafem prostym o $n > 3$ wierzchołkach. Niech $C(G)$ oznacza jego *nadgraf* (ang. *closure*) zbudowany przez dołączenie do G krawędzi (s,t) pomiędzy niesąsiednimi wierzchołkami s,t takimi, że

$$\deg_G(s) + \deg_G(t) \geqslant n. \tag{6.5}$$

Graf G jest hamiltonowski wtedy i tylko wtedy, gdy $C(G)$ jest hamiltonowski [18]. Dowód można znaleźć w [4].

36:9397391959

6.1. Algorytmy znajdywania cykli i ścieżek

Przedstawimy kilka wariantów algorytmów znajdowania ścieżek i cykli Hamiltona w grafach nieskierowanych.

6.1.1. Algorytm rekurencyjny na bazie DFS

Poniżej przedstawiamy algorytm rekurencyjny do znalezienia wszystkich cykli oraz ścieżek Hamiltona wykorzystujący zmodyfikowaną procedurę DFS.

Dane wejściowe: Graf nieskierowany lub skierowany.

Problem: Znalezienie wszystkich cykli i ścieżek Hamiltona.

Opis algorytmu: Algorytm polega na wyszukaniu w grafie wszystkich możliwych cykli oraz ścieżek Hamiltona za pomocą zmodyfikowanej procedury DFS. Wybór wierzchołka startowego nie ma znaczenia. Przykładowo zaczynamy od wierzchołka 0, który umieszczamy na stosie. Sprawdzamy czy stos zawiera n wierzchołków - jeśli tak, to mamy ścieżkę Hamiltona. Jeśli ostatni wierzchołek ścieżki ma połączenie z wierzchołkiem 0, to ścieżka tworzy cykl Hamiltona. Jeśli stos nie zawiera n wierzchołków, to ponownie w sposób rekurencyjny wywołujemy procedurę DFS dla nieodwiedzonych sąsiadów.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n!)$.

Listing 6.1. Moduł hamilton2.

```
#!/usr/bin/python

class HamiltonCycleDFS:
    """Algorytm znajdujacy wszystkie sciezki i cykle Hamiltona.
    Kod bazuje na opisie ze strony:
    http://edu.i-lo.tarnow.pl/inf/alg/001_search/0136.php
    """

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
        self.graph = graph
        self.stack = list()
        self.used = dict((node, False) for node in self.graph.iternodes())
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v()*2, recursionlimit))

    def run(self, source=None):
        """Obliczenia."""
        if source is None:   # pierwszy wylosowany wierzcholek
            source = self.graph.iternodes().next()
        self.source = source
        self._hamilton_dfs(self.source)

    def _hamilton_dfs(self, node):
        """Zmodyfikowane przejscie DFS z danego wierzcholka."""
        self.stack.append(node)    # dodaje node na poczatku metody
```


37:3701154790

```
if len(self.stack) == self.graph.v():
    # Wszystkie wierzcholki zostaly wykorzystane.
    # Mozemy miec sciezke lub cykl Hamiltona.
    is_cycle = False    # zakladamy brak cyklu
    for edge in self.graph.iteroutedges(node):
        if edge.target == self.source:
            is_cycle = True
            break
    if is_cycle:
        print "Hamiltonian Cycle :", self.stack + [self.source]
    else:
        print "Hamiltonian Path  :", self.stack
else:
    self.used[node] = True
    for edge in self.graph.iteroutedges(node):
        if not self.used[edge.target]:
            self._hamilton_dfs(edge.target)
    self.used[node] = False
self.stack.pop()    # usuwamy node na koncu metody
```

6.1.2. Algorytm Dharwadkera

Przetestowaliśmy również algorytm podany przez Dharwadkera w artykule z roku 2004 [19]. Implementacja algorytmu w języku C++ jest dostępna na stronie WWW artykułu. Algorytm został uruchomiony, a jego wyniki zostały porównane z naszymi rozwiązaniami i dały taki sam rezultat. Według autora algorytm znajduje cykle Hamiltona w czasie wielomianowym dla grafów spełniających twierdzenie Diraca. Algorytm jest więc konstruktywnym dowodem tego twierdzenia. Wielomianowy czas działania jest zadziwiający i dlatego autor stawia pytanie: Czy istnieje graf ze ścieżką lub cyklem Hamiltona, których algorytm nie potrafi znaleźć?

38:6722193456

7. Problem komiwojażera

Problem komiwojażera (ang. *travelling salesman problem*, TSP) polega na znalezieniu cyklu Hamiltona o najmniejszej wadze. Zwykle problem komiwojażera określa się dla grafu nieskierowanego ważonego pełnego. Jeżeli graf ważony nie jest pełny, to w zasadzie można go rozszerzyć do grafu pełnego dodając brakujące krawędzie z bardzo dużymi (nieskończonymi) wagami. Tak określony problem jest NP-trudny.

W wersji decyzyjnej problemu komiwojażera, danymi są graf i pewna liczba L. Należy odpowiedzieć na pytanie, czy istnieje trasa komiwojażera krótsza od L. Tak sformułowany problem jest NP-zupełny [20].

W tym rozdziale przedstawimy wybrane algorytmy dokładne i przybliżone służące do rozwiązywania problemu komiwojażera. Testy algorytmów znajdują się w dodatku B.

7.1. Algorytmy dokładne

Algorytmy dokładne dla problemu komiwojażera mogą być stosowane jedynie dla małych grafów, ze względu na szybko rosnący czas działania. Dla liczby wierzchołków $|V| = n$ i symetrycznego problemu komiwojażera stajemy przed koniecznością sprawdzenia $(n-1)!/2$ różnych cykli Hamiltona (wierzchołek startowy nie jest istotny).

7.1.1. Algorytm siłowy

Dane wejściowe: Graf pełny ważony.

Problem: Problem komiwojażera.

Opis algorytmu: Algorytm polega na wyszukaniu w grafie wszystkich możliwych cykli za pomocą permutacji wierzchołków, a następnie wybraniu cyklu o najmniejszej wadze. Algorytm daje nam najlepsze rozwiązanie, lecz niestety wraz ze zwiększaniem liczby wierzchołków staje się on coraz mniej wydajny.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n!)$.

Listing 7.1. Moduł tspbf.

```
#!/usr/bin/python

class BruteForceTSP:
    """Algorytm silowy dla problemu komiwojazera."""

    def __init__(self, graph):
```


39:1155731265

```python
        """Inicjalizacja algorytmu."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        # Cykl Hamiltona jako graf nieskierowany.
        self.tsp = self.graph.__class__(self.graph.v())
        # Cykl Hamiltona jako lista krawedzi.
        self.hamilton_cycle = list()
        self.used = set()
        self.stack = list()
        self.best_weight = float("inf")
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v()*2, recursionlimit))

    def run(self, source=None):
        """Obliczenia."""
        if source is None:
            source = self.graph.iternodes().next()
        self.source = source
        self._hamilton_dfs(self.source)
        # Budowa grafu na bazie cyklu.
        for edge in self.hamilton_cycle:
            self.tsp.add_edge(edge)

    def _hamilton_dfs(self, node):
        """Zmodyfikowane przejscie rekurencyjne DFS."""
        if len(self.stack) == self.graph.v()-1:
            for edge in self.graph.iteroutedges(node):
                if edge.target == self.source:
                    self.stack.append(edge)
                    weight = sum(edge.weight for edge in self.stack)
                    if weight < self.best_weight:
                        self.best_weight = weight
                        self.hamilton_cycle = list(self.stack)
                    self.stack.pop()
        else:
            self.used.add(node)
            for edge in self.graph.iteroutedges(node):
                if edge.target not in self.used:
                    self.stack.append(edge)
                    self._hamilton_dfs(edge.target)
                    self.stack.pop()
            self.used.remove(node)
```

7.2. Algorytmy heurystyczne

Algorytmy heurystyczne dla problemu komiwojażera nie dają gwarancji znalezienia rozwiązania optymalnego. Uzyskane rozwiązanie jest przybliżone, ale czas jego znalezienia jest dość krótki. Algorytmy heurystyczne stosuje się zazwyczaj wtedy, gdy nie są znane algorytmy dokładne działające wystarczająco szybko.

Oto wybrane algorytmy heurystyczne:

40:3608060961

— Algorytm najbliższych sąsiadów (ang. *nearest neighbor algorithm* lub *greedy algorithm*).
— Algorytm najbliższych sąsiadów z powtórzeniami (ang. *repetitive nearest neighbor algorithm*), w którym powtarzamy poszukiwania startując z kolejnych wierzchołków.
— Algorytm sortowania krawędzi (ang. *sorted edge algorithm* lub *cheapest link algorithm*).
— Algorytm aproksymacyjny bazujący na minimalnym drzewie rozpinającym.
— Pairwise exchange, inaczej technika 2-opt.
— Heurystyka k-opt lub Lin–Kernighana.
— Heurystyka V-opt.

7.2.1. Algorytm najbliższych sąsiadów

Dane wejściowe: Graf pełny ważony, opcjonalnie wierzchołek początkowy.

Problem: Problem komiwojażera.

Opis algorytmu: Algorytm rozpoczynamy od dowolnego wierzchołka grafu. Znajdujemy krawędź o najmniejszej wadze, która prowadzi do wierzchołka jeszcze nie odwiedzonego. Krawędź dodajemy do cyklu i przechodzimy do drugiego końca krawędzi. Czynności powtarzamy $n-1$ razy, gdzie n to liczba wierzchołków. Na koniec zamykamy cykl dołączając krawędź prowadzącą do startowego wierzchołka.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V^2)$, ponieważ w każdym z $n-1$ kroków musimy przejrzeć $n-1$ krawędzi w poszukiwaniu tej o najmniejszej wadze.

Uwagi: Obecna implementacja algorytmu jako pierwszy wierzchołek wybiera ten zwracany przez metodę iternodes. Można wprowadzić pełną losowość wyboru pierwszego wierzchołka za pomocą random.choice.

Listing 7.2. Moduł tspnn.

```
#!/usr/bin/python

class NearestNeighborTSP:
    """Algorytm najblizszych sasiadow dla problemu komiwojazera."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        # Cykl Hamiltona jako graf nieskierowany.
        self.tsp = self.graph.__class__(self.graph.v())
        # Cykl Hamiltona jako lista krawedzi.
        self.hamilton_cycle = list()
        self.used = set()
```


41:2839658068

```
    def run(self, source=None):
        """Obliczenia."""
        if source is None:
            source = self.graph.iternodes().next()
        start_node = source
        self.used.add(source)
        # Szukamy n-1 kolejnych krawedzi. Unikamy tworzenia cyklu.
        for step in xrange(self.graph.v()-1):
            edge = min(edge for edge in self.graph.iteroutedges(source)
                if edge.target not in self.used)
            self.tsp.add_edge(edge)
            self.hamilton_cycle.append(edge)
            self.used.add(edge.target)
            source = edge.target
        # Jeszcze trzeba zamknac cykl.
        for edge in self.graph.iteroutedges(source):
            if edge.target == start_node:
                self.tsp.add_edge(edge)
                self.hamilton_cycle.append(edge)
                break
        del self.used
```

7.2.2. Algorytm najbliższych sąsiadów z powtórzeniami

Dane wejściowe: Graf pełny ważony.

Problem: Problem komiwojażera.

Opis algorytmu: Algorytm bazuje na algorytmie najbliższych sąsiadów. Algorytm rozpoczynamy od dowolnego wierzchołka grafu. Znajdujemy krawędź o najmniejszej wadze, która prowadzi do wierzchołka jeszcze nie odwiedzonego. Krawędź dodajemy do cyklu i przechodzimy do drugiego końca krawędzi. Czynności powtarzamy $n-1$ razy, gdzie n to liczba wierzchołków. Na koniec zamykamy cykl dołączając krawędź prowadzącą do startowego wierzchołka. Algorytm powtarzamy dla każdego wierzchołka w grafie, a następnie wybieramy cykl o najmniejszej wadze.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V^3)$, ponieważ n razy powtarzamy czynności z algorytmu najbliższych sasiadów.

Listing 7.3. Moduł tsprnn.

```
#!/usr/bin/python

class RepeatedNearestNeighborTSP:
    """Algorytm najblizszych sasiadow z powtorzeniami
    dla problemu komiwojazera sprawdzajacy kazdy
    wierzcholek w grafie jako wierzcholek poczatkowy."""

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


42:8086044010

```
            raise ValueError("the graph is directed")
        self.graph = graph
        # Cykl Hamiltona jako graf nieskierowany.
        self.tsp = self.graph.__class__(self.graph.v())
        # Cykl Hamiltona jako lista krawedzi.
        self.hamilton_cycle = list()

    def run(self):
        """Obliczenia."""
        best_weight = float("inf")
        for source in self.graph.iternodes():
            start_node = source
            used = set([source])
            # Cykl Hamiltona jako graf nieskierowany.
            tmp_tsp = self.graph.__class__(self.graph.v())
            # Cykl Hamiltona jako lista krawedzi.
            tmp_hamilton_cycle = list()
            # Szukamy n-1 kolejnych krawedzi. Unikamy tworzenia cyklu.
            for step in xrange(self.graph.v()-1):
                edge = min(edge for edge in self.graph.iteroutedges(source)
                    if edge.target not in used)
                tmp_tsp.add_edge(edge)
                tmp_hamilton_cycle.append(edge)
                used.add(edge.target)
                source = edge.target
                # Jeszcze trzeba zamknac cykl.
            for edge in self.graph.iteroutedges(source):
                if edge.target == start_node:
                    tmp_tsp.add_edge(edge)
                    tmp_hamilton_cycle.append(edge)
                    break
            weight = sum(edge.weight for edge in tmp_hamilton_cycle)
            weight2 = sum(edge.weight for edge in tmp_tsp.iteredges())
            if weight != weight2:
                raise ValueError("rozne wagi cyklu Hamiltona")
            if weight < best_weight:
                best_weight = weight
                self.hamilton_cycle = tmp_hamilton_cycle
                self.tsp = tmp_tsp
```

7.2.3. Algorytm sortowania krawędzi

Dane wejściowe: Graf pełny ważony.

Problem: Problem komiwojażera.

Opis algorytmu: Algorytm rozpoczyna się od posortowania krawędzi względem wag, a następnie pobierane są kolejne krawędzie z najmniejszą wagą. Alternatywnie może zostać wykorzystana kolejka priorytetowa, która zwraca elementy o najmniejszym priorytecie. Kolejno pobierane krawędzie używane są do budowy cyklu Hamiltona o najmniejszej wadze. Należy pamiętać, aby

43:9139953805

z danego wierzchołka wychodziły maksymalnie dwie krawędzie oraz aby nie zamknąć cyklu przed wykorzystaniem wszystkich wierzchołków.

Złożoność: Złożoność czasowa algorytmu znaleziona eksperymentalnie wynosi $O(V^2)$.

Uwagi: Pomysł sortowania wszystkich krawędzi według wag jest taki sam jak w algorytmie Kruskala, który służy do wyznaczania minimalnego drzewa rozpinającego grafu.

Listing 7.4. Moduł tspse.

```
#!/usr/bin/python

from Queue import PriorityQueue
from unionfind import UnionFind

class SortedEdgeTSP:
    """Algorytm sortowania krawedzi dla problemu komiwojazera."""

    def __init__(self, graph):
        """Inicjalizacja."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        # Cykl Hamiltona jako graf nieskierowany.
        self.tsp = self.graph.__class__(self.graph.v())
        # Musimy wstawic wierzcholki, bo potem szukamy stopni.
        for node in self.graph.iternodes():
            self.tsp.add_node(node)
        # Cykl Hamiltona jako lista krawedzi.
        self.hamilton_cycle = list()
        self.uf = UnionFind()
        self.pq = PriorityQueue()

    def run(self, source=None):
        """Obliczenia."""
        for node in self.graph.iternodes():
            self.uf.create(node)
        for edge in self.graph.iteredges():
            self.pq.put((edge.weight, edge))
        while not self.pq.empty():
            _, edge = self.pq.get()
            degree1 = self.tsp.degree(edge.source)
            degree2 = self.tsp.degree(edge.target)
            if degree1 == 2 or degree2 == 2:
                continue
            if degree1 == 0 or degree2 == 0:  # tu nie zamykamy cyklu
                self.uf.union(edge.source, edge.target)
                self.tsp.add_edge(edge)
                continue
            # Tutaj degree1 = degree2 = 1.
            if self.uf.find(edge.source) != self.uf.find(edge.target):
                # Laczymy rozne kawalki.
                self.uf.union(edge.source, edge.target)
                self.tsp.add_edge(edge)
```


44:8447477715

```
        elif self.tsp.e() == self.tsp.v()-1:
            # Zamykamy cykl Hamiltona.
            self.tsp.add_edge(edge)
    # Przekształcam cykl Hamiltona do postaci listy krawedzi.
    # Pobieram dowolna krawedz.
    edge = self.tsp.iteredges().next()
    self.hamilton_cycle.append(edge)
    source = edge.source
    target = edge.target
    for step in xrange(self.tsp.v()-1):
        for edge in self.tsp.iteroutedges(target):
            if edge.target != source:
                self.hamilton_cycle.append(edge)
                source = edge.source
                target = edge.target
                break
    del self.uf
    del self.pq
```

7.3. Algorytmy z metaheurystykami

Metaheurystyka jest to ogólny algorytm do rozwiązywania problemów obliczeniowych [22]. Algorytm metaheurystyczny można używać do rozwiązywania dowolnego problemu, o ile ten problem można opisać w języku metaheurystyki. Zazwyczaj metaheurystyki służą do rozwiązania problemu optymalizacyjnego i podają sposób przechodzenia między możliwymi rozwiązaniami. Nie ma gwarancji znalezienia optymalnego rozwiązania, ani ustalonego czasu działania programu.

Metaheurystyki mogą jednak dostarczyć wystarczająco dobre rozwiązanie w sytuacji ograniczonych zasobów obliczeniowych. Pomysły na metaheurystyki czasem są czerpane z natury, czasem korzysta się z losowości. Zwykle metaheurystyki mają zestaw parametrów, które trzeba doświadczalnie dobrać do danego problemu.

Wybrane algorytmy metaheurystyczne:

— Lokalne przeszukiwanie (ang. *local search*), w każdym kroku badane są *sąsiednie (całe) rozwiązania* i wybierane jest takie, które maksymalizuje pewne kryterium. Dla problemu komiwojażera jest to algorytm 2-opt [23].
— Algorytm zachłanny (ang. *greedy algorithm*), w każdym kroku dokonuje decyzji lokalnie optymalnej. Dla problemu komiwojażera jest to algorytm najbliższych sąsiadów.
— Symulowane wyżarzanie (ang. *simulated annealing*).
— Przeszukiwanie z zakazami (ang. *tabu search*).
— Algorytm genetyczny (ang. *genetic algorithm*).
— Algorytm mrówkowy (ang. *ant colony optimization*).

W podrozdziałach zostaną opisane wybrane algorytmy z metaheurystykami, zastosowane do rozwiązywania problemu komiwojażera.

45:8596931654

7.3.1. Symulowane wyżarzanie

Symulowane wyżarzanie jest to heurystyka probabilistyczna do przeszukiwania przestrzeni rozwiązań, w celu znalezienia dobrego przybliżenia rozwiązania optymalnego [24]. Metoda jest często stosowana, kiedy przestrzeń poszukiwań jest dyskretna. Nazwa algorytmu pochodzi od podobieństwa do procesu wyżarzania metalu, w którym powolne jego studzenie sprawia, że metal przechodzi z płynnego w strukturę krystaliczną o najniższej energii, przy zredukowanej liczbie defektów.

Kluczowe w tej metodzie jest akceptowanie z pewnym prawdopodobieństwem rozwiązań gorszych niż aktualne po to, aby szerzej badać przestrzeń rozwiązań. Wraz z obniżaniem temperatury w układzie, prawdopodobieństwo przyjęcia gorszego rozwiązania maleje.

Istnieje analogia z układem fizycznym, który znajduje się w stanie s, w temperaturze T, oraz ma energię wewnętrzną $e = E(s)$. Celem jest doprowadzenie układu do stanu s_{min} o najmniejszej energii $E(s_{min})$. Przejście ze stanu s do stanu s' o energii $e' = E(s')$ zachodzi z prawdopodobieństwem $P(e, e', T)$. Najczęściej (ale nie zawsze) funkcję $P(e, e', T)$ określa się następująco: dla $e' < e$ wynosi 1, a dla $e' > e$ wynosi $\exp[-(e' - e)/T]$ (rozkład Boltzmanna). W problemie komiwojażera odpowiednikiem energii jest długość cyklu Hamiltona.

Kod algorytmu symulowanego wyżarzania w języku C, wraz z przykładem zastosowania do problemu komiwojażera, jest dostępny jako rozszerzenie do biblioteki GSL [25].

Oto strategia algorytmu symulowanego wyżarzania, zastosowana do problemu komiwojażera, na podstawie strony internetowej związanej z aplikacją *Shiny* [26]:

1. Stwórz losową listę miast. Przyjmij maksymalną temperaturę.
2. W każdej iteracji zamień miejscami dwa miasta na liście (lub wybierz inne *sąsiednie* rozwiązanie). Całkowity dystans to droga przebyta pomiędzy wszystkimi miastami.
3. Jeżeli po zmianie nowy dystans jest mniejszy od poprzedniego, to należy go zachować.
4. Jeżeli nowy dystans jest większy od poprzedniego, to należy go zachować z pewnym prawdopodobieństwem, które zależy od temperatury i różnicy dystansów.
5. Należy stopniowo obniżać temperaturę przy każdej iteracji (kroki od 2 do 5) do momentu osiągnięcia temperatury minimalnej.

Możliwe są różne scenariusze obniżania dodatniej i bezwymiarowej temperatury T aż do zera. Jednym ze sposobów jest w każdym kroku mnożenie temperatury przez stały czynnik z przedziału $(0, 1)$.

7.3.2. Przeszukiwanie z zakazami

Przeszukiwanie z zakazami [27] jest jedną z metod metaheurystycznych służących do znalezienia przybliżonego optymalnego rozwiązania. Algorytm rozpoczynamy zapamiętaniem pierwszego losowego rozwiązania startowego, a następnie w iteracyjnym przeszukiwaniu spośród wszystkich rozwiązań, zapamiętujemy już wykorzystane rozwiązania, które zapisujemy na liście tabu.

46:6109001532

Oznacza to, że danego rozwiązania nie można już wykorzystać przy następnych iteracjach. Zadaniem listy tabu jest wykluczenie prawdopodobieństwa zapętlenia podczas przeszukiwania oraz wyszukanie przez algorytm nowych, jeszcze nie wykorzystanych rowiązań. Nowe znalezione rozwiązanie porównujemy z aktualnym najlepszym rozwiązaniem i zapamiętujemy je jeśli jest lepsze. Zazwyczaj przeszukiwania kończą się, gdy osiągnięty zostanie górny limit liczby iteracji w pętli lub szukany wynik.

Autorem metody przeszukiwania z zakazami jest Fred Glover [28]. W skrócie można powiedzieć, że metoda wprowadza pamięć, oraz strategiczne ograniczenia w eksploracji przestrzeni rozwiązań. Nie ma elementów stochastycznych. Oto strategia algorytmu na podstawie prezentacji Becerra i Amado [29].

1. Zainicjalizuj wstępne rozwiązanie oraz liczbę iteracji.
2. Znajdź nowe rozwiązanie (lub zbiór rozwiazań) i porównaj je z poprzednim najlepszym rozwiązaniem.
3. Jeżeli po zmianie nowy dystans jest mniejszy od poprzedniego, to należy go zachować, w przeciwnym wypadku odrzucić.
4. Dopisz rozwiązanie do listy tabu.
5. Gdy osiągnięty zostanie limit iteracji, to otrzymamy przybliżone rozwiązanie, a wykonywanie algorytmu dobiegnie końca. W przeciwnym razie należy wrócić do punktu 2.

7.3.3. Algorytm mrówkowy

Algorytm mrówkowy [30] służy do znalezienia najlepszej drogi wykorzystując przy tym zachowanie mrówek. Mrówki poruszają się w sposób losowy szukając pożywienia. Gdy znajdą pożywienie, wracają do swoich kolonii pozostawiając po sobie ślad zawierający feromony. Feromony po pewnym czasie parują. Im dłuższa jest droga od znalezienia pokarmu do kolonii, tym więcej czasu na odparowanie śladu. Inne mrówki wyczuwając ślad przestają poruszać się w sposób losowy i idą tym śladem, dzięki czemu zapach właściwej drogi staje się silniejszy. Na koniec wszystkie mrówki będą poruszać się tym śladem, przez co inne losowe ślady wyparują. W ten sposób korzystając z zachowania mrówek, możemy stworzyć algorytm mrówkowy do otrzymania przybliżonego rozwiązania najkrótszej trasy.

Oto strategia algorytmu na podstawie artykułu Yanga i in. [31]:

1. Zainicjalizuj wstępne rozwiązanie oraz liczbę iteracji.
2. Rozmieść wszystkie mrówki w różnych miastach.
3. Dla każdej mrówki znajdź losową trasę, przy wybieraniu następnego miasta skorzystaj z feromonów innych mrówek.
4. Po ustaleniu trasy dla każdej z mrówek, zaktualizuj zapach feromonów.
5. Jeśli trasa przebyta przez mrówkę jest krótsza niż aktualnie najlepsza trasa, to ją zachowaj.
6. Gdy osiągnięty zostanie limit iteracji, to otrzymamy przybliżone rozwiązanie, a wykonywanie algorytmu dobiegnie końca.

47:1432562622

7.4. Odmiany problemu komiwojażera

Ze względu na szerokie zastosowania badano różne odmiany problemu komiwojażera. Przedstawimy kilka najciekawszych odmian.

Asymetryczny problem komiwojażera: Rozważa się graf skierowany ważony, w którym waga krawędzi (s, t) może być różna od wagi krawędzi przeciwnie skierowanej (t, s) (ang. *asymmetric TSP*). W ten sposób można opisać ulice jednokierunkowe, czy różne opłaty lotnicze dla przylotów i odlotów.

Metryczny problem komiwojażera: W tym problemie (ang. *metric TSP*) narzuca się na wagi krawędzi (odległości) między wierzchołkami warunek trójkąta, tzn.

$$w(s,t) \leqslant w(s,u) + w(u,t), \tag{7.1}$$

gdzie s, t, u są wierzchołkami grafu nieskierowanego ważonego [20]. W ten sposób zbiór wierzchołków staje się przestrzenią metryczną z metryką $d(s,t) = w(s,t)$. Przykładowe metryki to *metryka euklidesowa* (euklidesowy problem komiwojażera) i *metryka Manhatanu (taksówkowa)*. Okazało się, że dla euklidesowego problemu komiwojażera stworzono szybsze algorytmy, niż w ogólnym przypadku.

Problem komiwojażera z powtarzającymi się wierzchołkami: Jeżeli odrzucimy założenie o braku powtarzających się wierzchołków w cyklu Hamiltona, to możemy zdefiniować metrykę $d(s,t)$ jako najkrótszą ścieżkę między wierzchołkami. Wtedy oryginalny graf niemetryczny zastępujemy grafem pełnym metrycznym z krawędziami odpowiadającymi najkrótszym ścieżkom.

7.4.1. Algorytm 2-aproksymacyjny dla metrycznego problemu komiwojażera

Rozważmy graf pełny ważony G z wagami krawędzi spełniającymi nierówność trójkąta. Algorytm bazuje na strukturze minimalnego drzewa rozpinającego T, którego waga stanowi dolne ograniczenie na długość optymalnego cyklu Hamiltona H^* [5]. Cykl Hamiltona z usuniętą dowolną krawędzią staje się drzewem rozpinającym, a więc mamy oszacowanie dolne $w(T) \leqslant w(H^*)$.

Z drugiej strony rozważmy cykl C będącym pełnym przejściem drzewa T. Jest to jakby obchodzenie ściany zewnętrznej na płaskim rysunku drzewa T. Każdą krawędź przechodzimy dwukrotnie, co daje równanie na wagi $w(C) = 2w(T)$. Cykl C zawiera powtarzające się wierzchołki. Korzystając z nierówności trójkąta możemy jednak wyeliminować wizyty w powtarzających się wierzchołkach, nie powiększając kosztu. Otrzymujemy cykl Hamiltona H z następującym oszacowaniem kosztu

$$w(H) \leqslant w(C) = 2w(T) \leqslant 2w(H^*). \tag{7.2}$$

48:9171341404

Pomimo dobrego współczynnika aproksymacji, opisany algorytm zwykle nie jest najpraktyczniejszym narzędziem dla tego problemu [5]. Christofedes poprawił ten algorytm uzyskując współczynnik aproksymacji 3/2 [21].

Dane wejściowe: Graf pełny ważony z wagami krawędzi spełniającymi nierówność trójkata; opcjonalnie wierzchołek będący korzeniem minimalnego drzewa rozpinającego.

Problem: Metryczny problem komiwojażera.

Opis algorytmu: Algorytm oblicza minimalne drzewo rozpinające T za pomocą algorytmu Prima. Następnie wyznaczana jest lista wierzchołków drzewa T w kolejności *preorder*, do czego wykorzystany jest algorytm DFS. Ta lista jest szukanym cyklem Hamiltona H.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V^2)$, ponieważ wynika to z zastosowania algorytmu Prima (implementacja z macierzą sąsiedztwa dla grafu gęstego).

Listing 7.5. Moduł tspmst.

```
#!/usr/bin/python
# Implementacja na podstawie:
# http://www.mimuw.edu.pl/delta/artykuly/delta0208/komiwojazer.pdf

from prim import PrimMatrixMST
from dfs import SimpleDFS

class MSTreeTSP:
    """Algorytm dla problemu komiwojazera z wykorzystaniem
    minimalnego drzewa rozpinajacego."""

    def __init__(self, graph):
        """Inicjalizacja."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        # Cykl Hamiltona jako graf.
        self.tsp = self.graph.__class__(self.graph.v())
        # Cykl Hamiltona jako lista krawedzi.
        self.hamilton_cycle = list()

    def run(self, source=None):
        """Obliczenia."""
        if source is None:
            source = self.graph.iternodes().next()
        self.source = source
        algorithm = PrimMatrixMST(self.graph)    # O(V**2) time
        algorithm.run(self.source)
        self.mst = algorithm.to_tree()
        # Obchodzenie MST w kolejnosci preorder z uzyciem DFS.
        # Cykl Hamiltona jako lista wierzcholkow.
        order = list()
        algorithm = SimpleDFS(self.mst)    # O(V) time
        algorithm.run(self.source, pre_action=lambda node: order.append(node))
```


49:9618493725

```
order.append(self.source)    # zamykam cykl, teraz dlugosc V+1
# Szukam krawedzi realizujacych cykl Hamiltona, O(V**2) time.
for i in xrange(self.graph.v()):
    source = order[i]
    target = order[i+1]
    for edge in self.graph.iteroutedges(source):
        if edge.target == target:
            self.hamilton_cycle.append(edge)
            self.tsp.add_edge(edge)
            break
```

8. Cykle Hamiltona w grafach skierowanych

Twierdzenia podające warunki istnienia skierowanego cyklu Hamiltona są często bardziej złożonymi wersjami analogicznych twierdzeń dla grafów nieskierowanych.

Twierdzenie (Meyniel, 1973): Jeżeli G jest silnie spójnym grafem skierowanym prostym, oraz dla każdej pary (s, t) niesąsiadujących wierzchołków zachodzi

$$\deg(s) + \deg(t) \geqslant 2n - 1, \tag{8.1}$$

gdzie $\deg(v) = \text{outdeg}(v) + \text{indeg}(v)$ jest całkowitym stopniem wierzchołka v, to graf G jest hamiltonowski [4].

Twierdzenie (Ghouila, 1960): Jeżeli G jest silnie spójnym grafem skierowanym prostym, oraz dla każdego wierzchołka v zachodzi

$$\deg(v) \geqslant n, \tag{8.2}$$

to graf G jest hamitonowski [4]. W dowodzie wybieramy dowolną parę (s, t) wierzchołków niesąsiednich. Z założenia mamy $\deg(s) \geqslant n$, $\deg(t) \geqslant n$, $\deg(s) + \deg(t) \geqslant 2n$. Spełnione są założenia twierdzenia Meyniela, a więc graf G jest hamiltonowski.

Twierdzenie (Woodall, 1972): Jeżeli G jest grafem skierowanym prostym, oraz dla każdej pary (s, t) różnych wierzchołków zachodzi

$$\text{outdeg}(s) + \text{indeg}(t) \geqslant n, \tag{8.3}$$

to graf G jest hamiltonowski [32]. W dowodzie najpierw pokazuje się, ze dla każdej pary różnych wierzchołków istnieje ścieżka skierowana która je łączy, a więc graf G jest silnie spójny. Następnie korzysta się z twierdzenia Meyniela [4].

8.1. Sortowanie topologiczne

Istnieje ciekawy związek między sortowaniem topologicznym daga, a ścieżką Hamiltona. Jeżeli po posortowaniu topologicznym każda para kolejnych wierzchołków jest połączona krawędzią, to powstaje ścieżka Hamiltona, a do tego jest to jedyne możliwe uporządkowanie topologiczne daga [33]. Przeciwnie, jeżeli sortowanie topologiczne nie tworzy ścieżki Hamiltona, to znaczy że istnieje wiele możliwych rozwiązań problemu sortowania topologicznego.

51:6515703761

Wnioskiem z tych rozważań jest możliwość testowania w czasie liniowym istnienia ścieżki Hamiltona i jednoznaczności rozwiązania problemu sortowania topologicznego.

8.1.1. Sortowanie topologiczne z wykorzystaniem DFS

Dane wejściowe: Graf acykliczny skierowany.

Problem: Sortowanie topologiczne wierzchołków.

Opis algorytmu: Algorytm sortowania topologicznego grafu wykorzystuje algorytm przeszukiwania w głąb. Wykorzystujemy własność grafów skierowanych acyklicznych, polegającą na istnieniu w grafie wierzchołka stopnia zero, bez krawędzi dochodzących. Po usunięciu wierzchołka, graf nie staje się cykliczny, więc po każdej takiej operacji w grafie będzie wierzchołek o stopniu wychodzącym równym zero. Na końcu otrzymujemy listę wierzchołków, która po odwróceniu daje nam poprawną kolejność wierzchołków w sortowaniu topologicznym.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V+E)$, czyli tyle samo co w algorytmie sortowania w głąb.

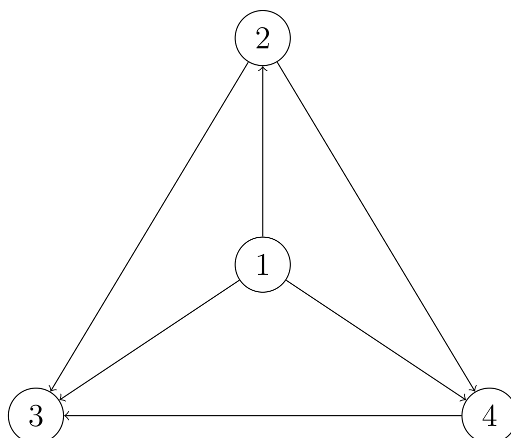
Listing 8.1. Moduł sorttop.

```
#!/usr/bin/python
# Implementacja algorytmu na podstawie:
# https://en.wikipedia.org/wiki/Topological_sorting

from dfs import DepthFirstSearch

class TopologicalSort:
    """Algorytm sortowania topologicznego
    wykorzystujacy rekurencyjny algorytm dfs."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
        if not graph.is_directed():
            raise ValueError("the graph is not directed")
        self.graph = graph
        self.order = list()

    def run(self):
        """Obliczenia."""
        DepthFirstSearch(self.graph).run(post_action=lambda node:
            self.order.append(node))
        self.order.reverse()
```

8.2. Turnieje

Turniejem (grafem pełnym asymetrycznym) (ang. *tournament*) nazywamy graf skierowany, w którym każda para wierzchołków jest połączona dokładnie jedną krawędzią [34]. Turniej jest *tranzytywny* (ang. *transitive*), jeżeli

52:1815528288



Rysunek 8.1. Turniej tranzytywny dla czterech wierzchołków. Stopnie wejściowe wierzchołków to $(0, 1, 2, 3)$.

dla każdej trójki wierzchołków (s, t, u) istnienie krawędzi (s, t) i (t, u) pociąga za sobą istnienie krawędzi (s, u).

Twierdzenie: Turniej jest tranzytywnym wtedy i tylko wtedy, gdy ciągiem wychodzących stopni wierzchołków jest $(0, 1, \ldots, n-1)$ ([4], s. 163).

Twierdzenie: Turniej jest tranzytywny wtedy i tylko wtedy, gdy jest acykliczny [34].

Twierdzenie: Turniej jest tranzytywny wtedy i tylko wtedy, gdy nie zawiera cyklu o długości 3 [34].

Twierdzenie (Rédei, 1934): Każdy turniej ma skierowaną ścieżkę Hamiltona ([4], s. 162). Dowód jest przez indukcję względem liczby wierzchołków turnieju. Twierdzenie Rédei jest nawet mocniejsze, mówi że każdy skończony turniej ma *nieparzystą* liczbę ścieżek Hamiltona.

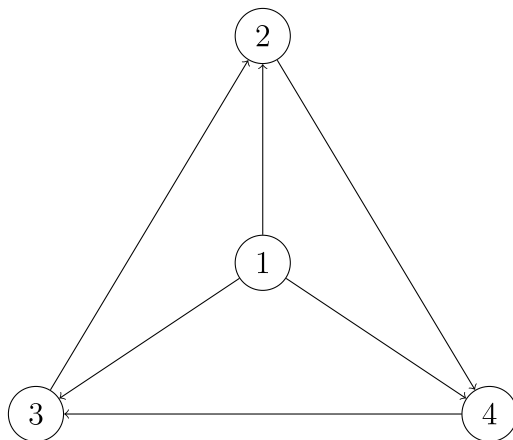
Twierdzenie: Turniej ma dokładnie jedną ścieżkę Hamiltona wtedy i tylko wtedy, gdy jest tranzytywny ([4], s. 164). W takim turnieju ścieżka Hamiltona zaczyna się w wierzchołku o najwyższym stopniu wyjściowym, a kończy w wierzchołku o najwyższym stopniu wejściowym.

Twierdzenie (Camion, 1959): Każdy turniej silnie spójny jest hamiltonowski ([3], s. 143).

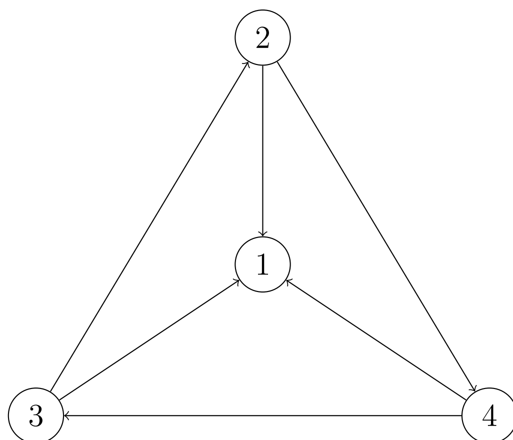
Na rysunkach od 8.1 do 8.4 przedstawione zostały wszystkie możliwe turnieje z czterema wierzchołkami. Wygodnie jest je rozróżniać za pomocą ciągu stopni wejściowych wierzchołków.

Rysunki 8.5 i 8.6 przedstawiają dwa wybrane turnieje z pięcioma wierzchołkami. Pierwszy turniej jest tranzytywny, natomiast drugi jest jednocześnie grafem Eulera i grafem Hamiltona. Co ciekawe, drugi graf można przedstawić jako sumę dwóch rozłącznych cykli Hamiltona (obwód i środkowa gwiazda).

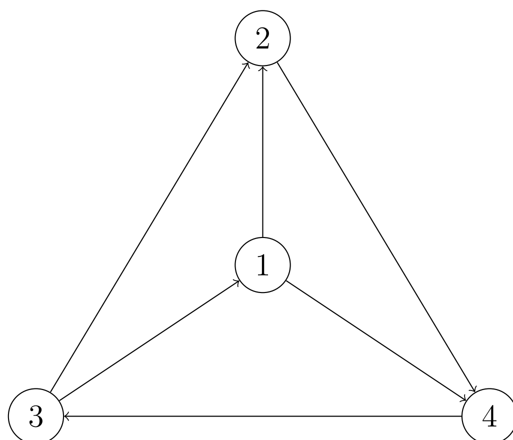
53:7686035804



Rysunek 8.2. Turniej z jednym źródłem dla czterech wierzchołków. Stopnie wejściowe wierzchołków to $(0, 2, 2, 2)$.

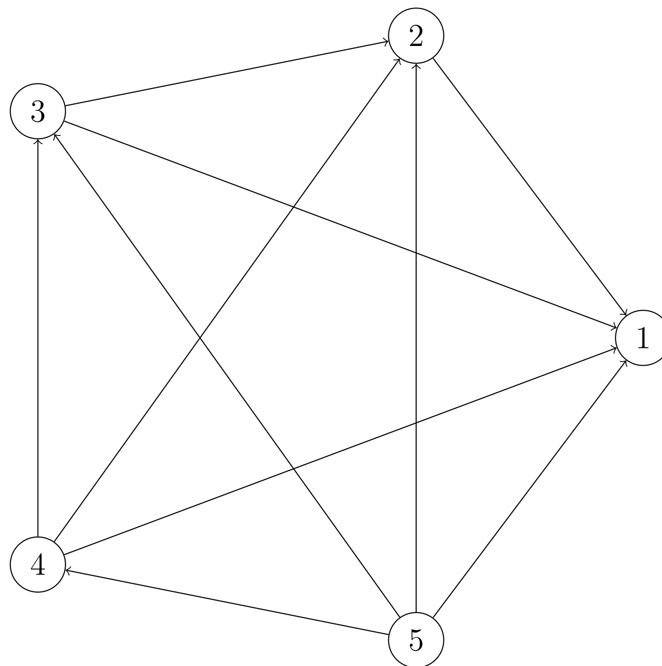


Rysunek 8.3. Turniej z jednym ujściem dla czterech wierzchołków. Stopnie wejściowe wierzchołków to $(1, 1, 1, 3)$.

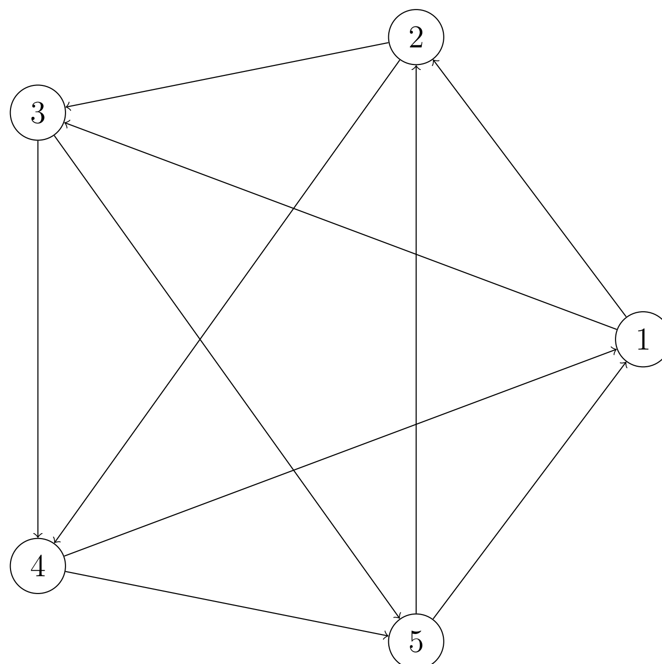


Rysunek 8.4. Turniej z cyklem Hamiltona dla czterech wierzchołków. Stopnie wejściowe wierzchołków to $(1, 1, 2, 2)$.

54:9919358300



Rysunek 8.5. Turniej tranzytywny z pięcioma wierzchołkami. Stopnie wejściowe wierzchołków to $(0, 1, 2, 3, 4)$.



Rysunek 8.6. Turniej eulerowski z piecioma wierzchołkami, który jest zbudowany -z dwóch rozłącznych cykli Hamiltona. Stopnie wejściowe wierzchołków to $(2, 2, 2, 2, 2)$.

55:8330749967

8.2.1. Sprawdzanie tranzytywności turnieju

Test tranzytywności turnieju można wykonać w czasie $O(V)$. Wystarczy sprawdzić, czy stopnie wejściowe (lub wyjściowe) wierzchołków są różnymi liczbami od 0 do $n-1$, jeżeli turniej ma n wierzchołków. Dla każdego wierzchołka v w turnieju zachodzi związek

$$\mathrm{outdeg}(v) + \mathrm{indeg}(v) = n - 1. \tag{8.4}$$

Listing 8.2. Funkcja testująca tranzytywność turnieju.

```
def is_transitive(graph):
    """Sprawdzenie tranzytywnosci turnieju w czasie O(V)."""
    if not graph.is_directed():
        raise ValueError("the graph is not directed")
    node_list = [None] * graph.v()
    # Obliczam stopnie wyjsciowe, bo to jest szybsze.
    # Maja byc rozne liczby od 0 do V-1.
    for node in graph.iternodes():
        node_list[graph.outdegree(node)] = node
    return all(item is not None for item in node_list)
```

8.2.2. Wyznaczanie ścieżki Hamiltona w turnieju

Skierowaną ścieżkę Hamiltona w turnieju można wyznaczyć metodą rekurencyjną, bazującą na dowodzie twierdzenia Rédei. W najgorszym razie trzeba wtedy przejrzeć wszystkie krawędzie, co daje złożoność $O(V^2)$. Istnieje jednak efektywniejszy algorytm, który wymaga zbadania tylko $O(V \log V)$ krawędzi (Bar-Noy, Naor, 1990) [35].

Główna idea algorytmu to odpowiedniość między zbiorem krawędzi skierowanych turnieju, a zbiorem porównań wykonywanych przy sortowaniu obiektów (wierzchołków grafu). Z teorii wiadomo, że najlepsze algorytmy sortujące mają złożoność $O(n \log n)$, a więc nie trzeba przeglądnąć wszystkich krawędzi, aby otrzymać ścieżkę Hamiltona. Listing 8.3 pokazuje funkcję bazującą na standardowym algorytmie sortowania w Pythonie (timsort), która znajduje ścieżkę Hamiltona w dowolnym turnieju.

Listing 8.3. Znajdowanie ścieżki Hamiltona w turnieju.

```
def find_hamiltonian_path(graph):
    """Znajdz sciezke Hamiltona w turnieju."""
    if not graph.is_directed():
        raise ValueError("the graph is not directed")
    return sorted(graph.iternodes(), cmp=lambda x, y:
        -1 if graph.has_edge(Edge(x, y)) else 1)
```

Można podać jeszcze inną procedurę znajdowania ścieżki Hamiltona w turnieju, bardzo bliską poprzednio opisanej. Turniej tranzytywny jest acykliczny, a wtedy wystarczy wykonać sortowanie topologiczne wierzchołków. Jeżeli turniej nie jest tranzytywny, to zawiera cykle, a wtedy tymczasowo "naprawia" się pewne krawędzie, przywracając acykliczność turnieju. Okazuje się, że te "naprawione" krawędzie nie wchodzą do znalezionej ścieżki Hamiltona. Wydaje się, że to podejście będzie miało złożoność $O(V^2)$.

56:3551630067

9. Podsumowanie

W przedstawionej pracy magisterskiej zaimplementowano wiele algorytmów grafowych związanych z grafami Hamiltona. Są algorytmy znajdujące cykle i ścieżki Hamiltona, czy rozwiązujące problem komiwojażera. Kod źródłowy algorytmów został napisany w języku Python, w sposób jak najbardziej wydajny i czytelny, a także nieodbiegający znacząco od ich pierwowzorów w pseudokodzie. Dzięki temu można go stosować w artykułach i prezentacjach. Implementacja algorytmów z wielu dziedzin skupiona w jednym języku programowaniu sprawia, że użytkownik dostaje cenne narzędzia rozszerzające bibliotekę standardową Pythona.

Zaprezentowana implementacja kodu źródłowego jest zgodna ze standardami języka Python (PEP8). Spełnia również dobre praktyki programowania takie jak: podział na funkcje, klasy, moduły, a także odpowiednio zastosowane komentarze w miejscach, w których ułatwiają zrozumienie kodu. Algorytmy zostały również odpowiednio przetestowane pod względem poprawności i wydajności, oraz porównane z wynikami teoretycznymi.

Jesteśmy przekonani, że niniejsza praca ma dużą wartość dydaktyczną, a algorytmy w niej zawarte mają szereg perspektyw wykorzystania np. do nauki programowania w języku Python, a także do nauki teorii grafów. Możliwy jest ich dalszy rozwój oraz prace badawcze. Przykładowo można zaimplementować wybrane metaheurystyki na bazie opisów zawartych w niniejszej pracy, a następnie zastosować je do problemu komiwojażera.

57:4700538403

A. Kod źródłowy dla krawędzi i grafów

Do modelowania krawędzi i grafów przyjęto klasy przygotowane na Wydziale Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Jagiellońskiego w Krakowie. Kod jest dobrze przetestowany i dostępny z publicznego repozytorium w serwisie GitHub [1]. Dla kompletności pracy zamieszczamy kod źródłowy tych klas.

A.1. Klasa Edge

Listing A.1. Moduł edges.

```
#!/usr/bin/python
#
# Hashable edges - the idea for __hash__ from
# http://stackoverflow.com/questions/793761/built-in-python-hash-function

class Edge:
    """The class defining an edge."""

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

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

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


58:9963118264

```
            return -1
        # Check the second node.
        if self.target > other.target:
            return 1
        if self.target < other.target:
            return -1
        return 0

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

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

    inverted = __invert__
```

A.2. Klasa Graph

Listing A.2. Moduł graphs.

```
#!/usr/bin/python

import random
from edges import Edge

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

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

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

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

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

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

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


59:5312699404

```
        return node in self

    def del_node(self, node):
        """Remove a node from the graph (with edges)."""
        # dictionary changes size during iteration.
        for edge in list(self.iterinedges(node)):
            self.del_edge(edge)
        if self.is_directed():
            for edge in list(self.iteroutedges(node)):
                self.del_edge(edge)
        del self[node]

    def add_edge(self, edge):
        """Add an edge to the graph (missing nodes are created)."""
        if edge.source == edge.target:
            raise ValueError("loops are forbidden")
        self.add_node(edge.source)
        self.add_node(edge.target)
        if edge.target not in self[edge.source]:
            self[edge.source][edge.target] = edge.weight
        else:
            raise ValueError("parallel edges are forbidden")
        if not self.is_directed():
            if edge.source not in self[edge.target]:
                self[edge.target][edge.source] = edge.weight
            else:
                raise ValueError("parallel edges are forbidden")

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

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

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

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

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

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


60:1065817254

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

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

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

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

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

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

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

    def indegree(self, node):
        """Return the indegree of the node."""
        if self.is_directed():   # O(V) time
            counter = 0
            for sources_dict in self.itervalues():
```


61:8523531084

```
                if node in sources_dict:
                    counter = counter + 1
            return counter
        else:                       # O(1) time
            return len(self[node])

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

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

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

B. Testy dla problemu komiwojażera

W celu potwierdzenia wydajności omawianych algorytmów dla problemu komiwojażera wykonano szereg testów. W pierwszej części wykorzystywano grafy pełne z unikalnymi wagami, będącymi kolejnymi liczbami naturalnymi. W drugiej części wykorzystano grafy pełne z wagami odpowiadającymi odległościom między losowymi punktami na płaszczyźnie.

B.1. Porównanie wydajności algorytmów dla problemu komiwojażera

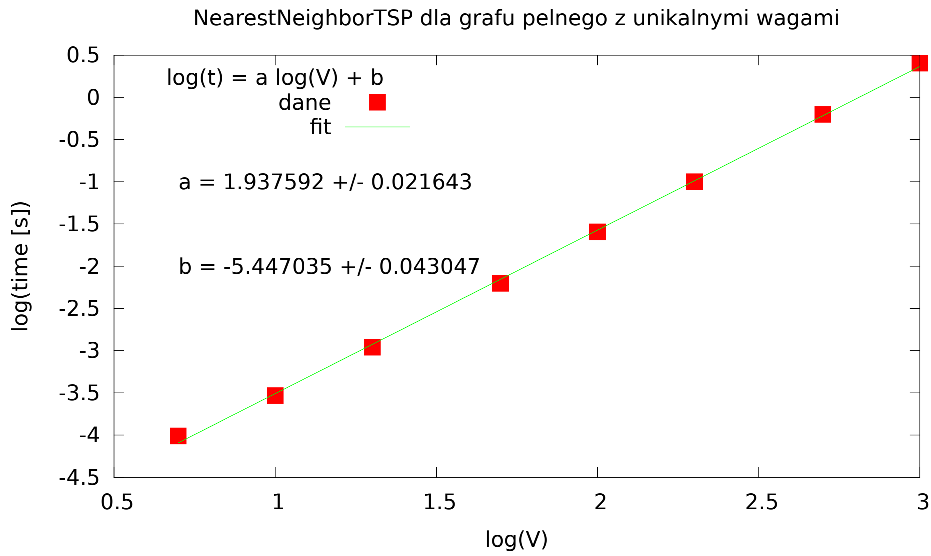
Dla ustalonej liczby wierzchołków n wygenerowano dziesięć grafów pełnych z unikalnymi wagami (liczby od 1 do $n(n-1)/2$) rozmieszczonymi losowo w krawędziach grafu. Waga cyklu Hamiltona mieści się więc w przedziale od $n(n+1)/2$ do $n(n-1)^2/2$. Cykl Hamiltona o najmniejszej wadze zwykle nie osiąga dolnej granicy, ponieważ krawędzie o niskiej wadze mogą się skupiać przy pewnych wierzchołkach.

Wyniki testów znajdują się w tabeli B.1. Dokładny algorytm siłowy jest nieprzydatny już dla grafów o kilkunastu wierzchołkach. Z algorytmów przybliżonych najlepsze oszacowania otrzymano dla algorytmu najbliższych sąsiadów z powtórzeniami, z wyjątkiem przypadku $|V| = 1000$, gdzie minimalnie lepszy był algorytm sortowania krawędzi. Dla grafów o 10, 20, 50, 100, 500 wierzchołkach bezkonkurencyjny był algorytm najbliższych sąsiadów z powtórzeniami. Dla 1000 wierzchołków 6 razy lepszy okazał się algorytm najbliższych sąsiadów, a 4 razy wygrał algorytm sortowania krawędzi. Przy 2000 wierzchołkach 7 razy lepszy był algorytm najbliższych sąsiadów z powtórzeniami, a 3 razy lepszy okazał się algorytm sortowania krawędzi. Dla większych grafów potrzeba coraz więcej czasu do znalezienia wyników, ale na podstawie otrzymanych przez nas wyników można wnioskować, że algorytmy najbliższych sąsiadów z powtórzeniami oraz sortowania krawędzi są porównywalne.

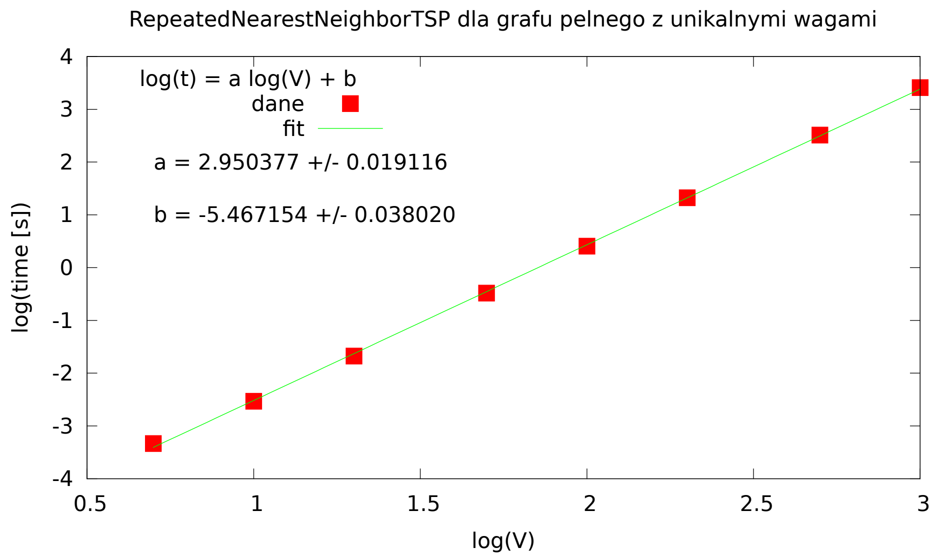
B.2. Porównanie wydajności algorytmów dla metrycznego problemu komiwojażera

Dla ustalonej liczby wierzchołków n wygenerowano dziesięć grafów pełnych ważonych. Wierzchołki grafu odpowiadają losowym punktom wewnątrz kwadratu jednostkowego. Wagi krawędzi są równe odległości euklidesowej pomiędzy punktami, czyli mamy do czynienia z instancją metrycznego problemu komiwojażera. Wyniki testów znajdują się w tabeli B.2. Dokładny algorytm siłowy jest nieprzydatny już dla grafów o kilkunastu wierzchołkach,

63:3732453833

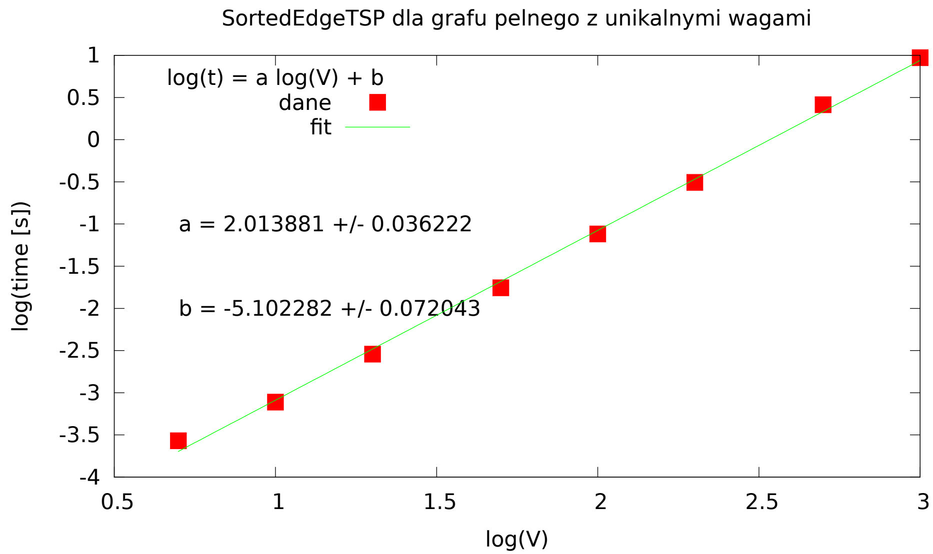


Rysunek B.1. Wykres wydajności algorytmu najbliższych sąsiadów dla grafu pełnego z unikalnymi wagami krawędzi. Współczynnik a bliski 2 potwierdza złożoność czasową $O(V^2)$.

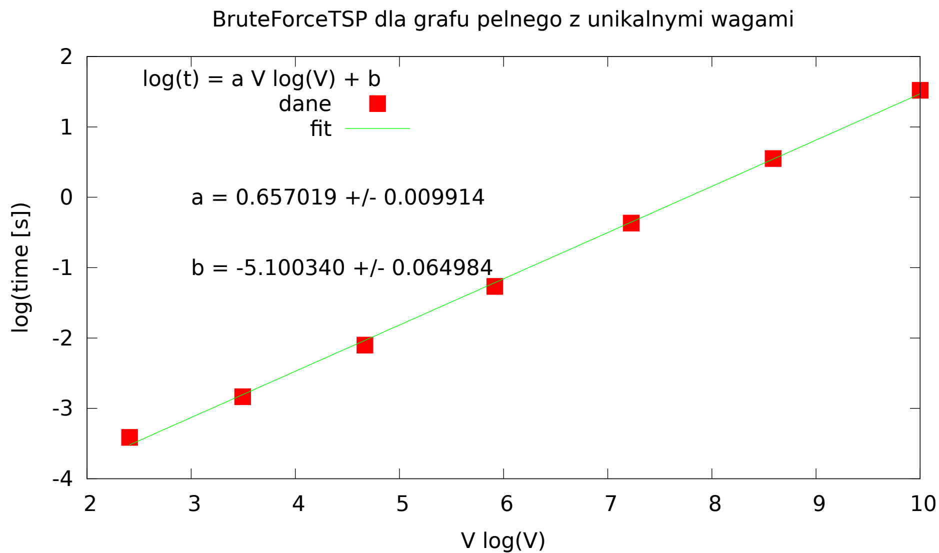


Rysunek B.2. Wykres wydajności algorytmu najbliższych sąsiadów z powtórzeniami dla grafu pełnego z unikalnymi wagami krawędzi. Współczynnik a bliski 3 potwierdza złożoność czasową $O(V^3)$.

64:9653999737



Rysunek B.3. Wykres wydajności algorytmu sortowania krawędzi dla grafu pełnego z unikalnymi wagami krawędzi. Współczynnik a bliski 2 potwierdza złożoność czasową $O(V^2)$.



Rysunek B.4. Wykres wydajności algorytmu siłowego dla grafu pełnego z unikalnymi wagami krawędzi. Liniowa zależność potwierdza złożoność czasową $O(V!)$.

65:4953110952

Tabela B.1. Test średniej wagi cyklu Hamiltona dla dziesięciu wygenerowanych grafów pełnych z unikalnymi wagami krawędzi. Oznaczenia algorytmów: BF algorytm siłowy, NN algorytm najbliższych sąsiadów, RNN algorytm najbliższych sąsiadów z powtórzeniami, SE algorytm sortowania krawędzi.

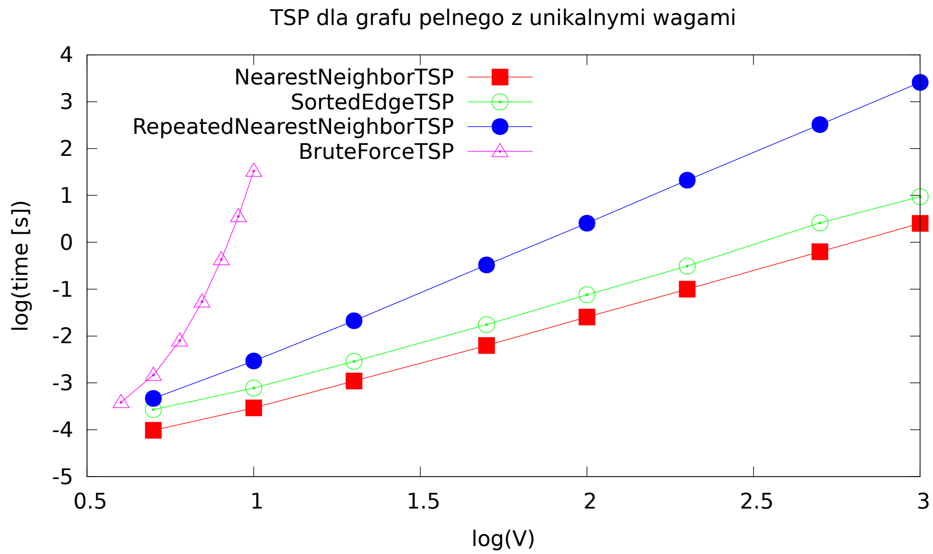
n wierzchołków	**Granica**	**BF**	**NN**	**RNN**	**SE**
5	15	19.5	21.0	19.6	19.8
10	55	80.3	95.5	86.7	92.0
20	210	-	540.5	451.3	538.7
50	1275	-	5305.1	4084.4	4897.3
100	5050	-	23315.5	17090.4	19218.8
200	20100	-	110260.9	78806.9	88283.4
500	125250	-	787811.1	574479.6	627460.4
1000	500500	-	3368143.8	2601063.6	2594669.3
2000	2001000	-	15427415.0	11596224.1	12096106.0

dlatego przetestowano go do 10 wierzchołków. Z wykorzystanych algorytmów do 200 wierzchołków najlepsze oszacowania otrzymano dla algorytmu najbliższych sąsiadów z powtórzeniami, od $|V| > 200$ najlepszy okazał się algorytm sortowania krawędzi, dzięki któremu otrzymano najmniejszą średnią wagę. Algorytm z wykorzystaniem minimalnego drzewa rozpinającego zastosowany przy tych testach okazał się najgorszym algorytmem i dla wszystkich przetestowanych wierzchołków jego średnia waga była największa. Dla większych grafów potrzeba coraz więcej czasu do znalezienia wyników, ale na podstawie otrzymanych przez nas wyników można wnioskować, że algorytmy najbliższych sąsiadów z powtórzeniami oraz sortowania krawędzi są porównywalne.

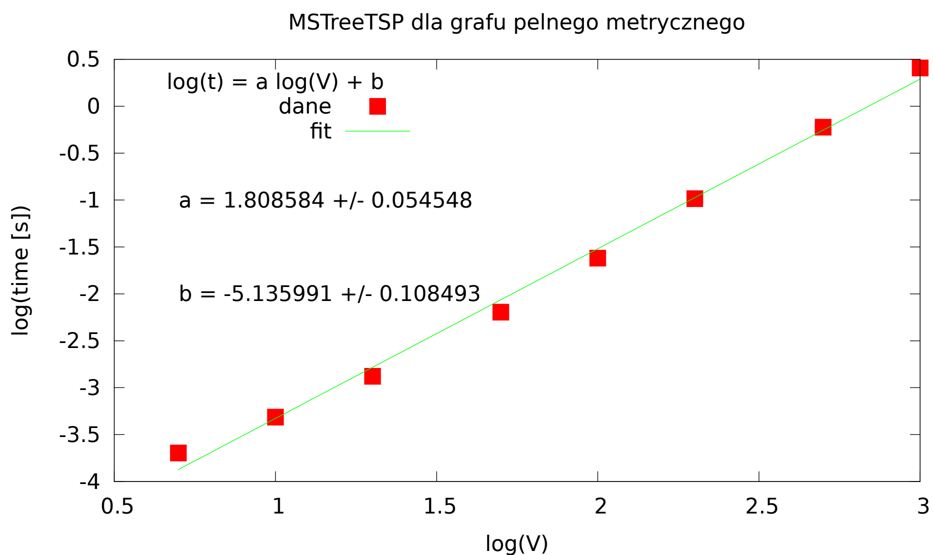
Tabela B.2. Test średniej wagi cyklu Hamiltona dla dziesięciu wygenerowanych grafów pełnych z wagami odpowiadającymi odległościom między losowymi punktami na płaszczyźnie. Oznaczenia algorytmów: BF algorytm siłowy, NN algorytm najbliższych sąsiadów, RNN algorytm najbliższych sąsiadów z powtórzeniami, SE algorytm sortowania krawędzi, MST algorytm z wykorzystaniem minimalnego drzewa rozpinającego.

n wierzchołków	**BF**	**NN**	**RNN**	**SE**	**MST**
5	2.130	2.230	2.134	2.163	2.243
10	2.880	3.024	2.894	3.071	3.508
20	-	4.209	3.797	3.930	4.622
50	-	7.077	6.429	6.670	7.372
100	-	9.935	8.978	9.338	10.743
200	-	13.384	12.331	11.412	14.780
500	-	20.694	19.657	19.330	23.088
1000	-	28.966	27.821	26.987	32.478

66:1011084495

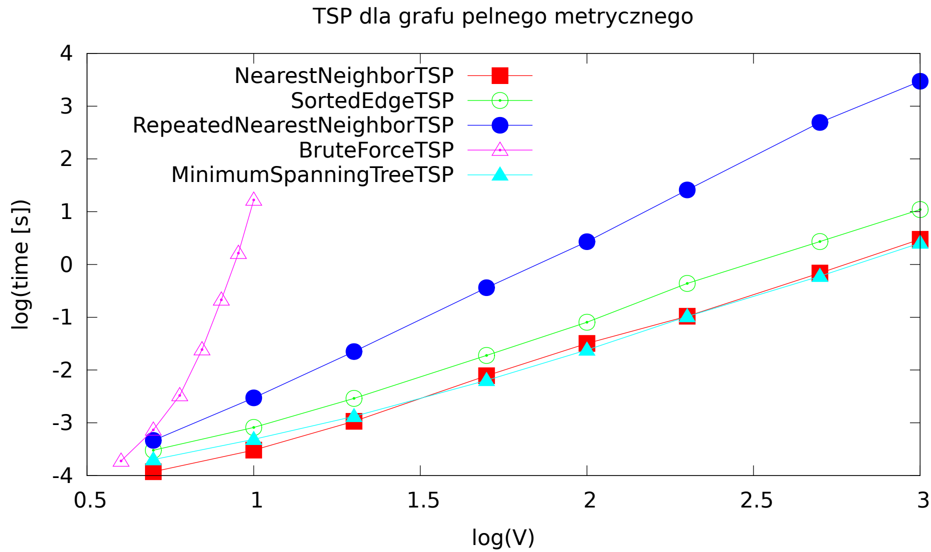


Rysunek B.5. Porównanie czasów obliczeń dla poszczególnych algorytmów rozwiązujących problem komiwojażera. Do testów wykorzystano grafy pełne z unikalnymi wagami krawędzi.



Rysunek B.6. Wykres wydajności algorytmu wykorzystujący minimalne drzewo rozpinające dla grafu pełnego metrycznego.

67:2964026629

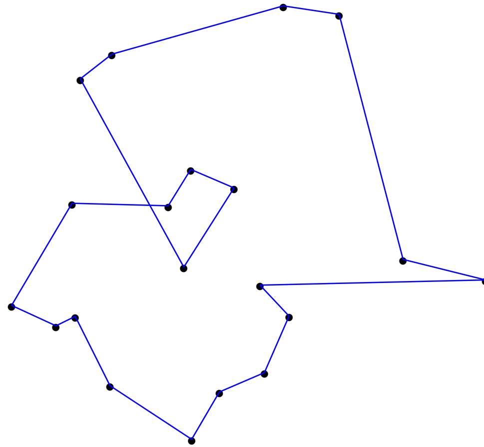


Rysunek B.7. Porównanie czasów obliczeń dla poszczególnych algorytmów rozwiązujących problem komiwojażera. Do testów wykorzystano grafy pełne metryczne.

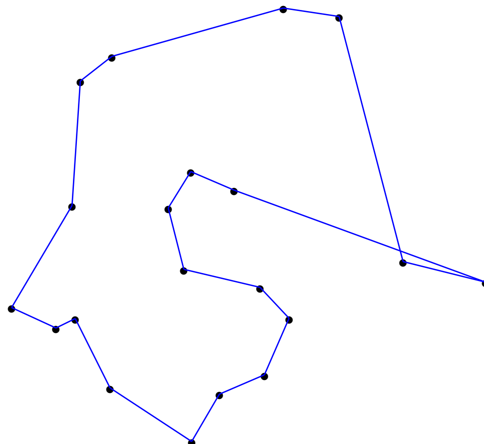
B.3. Porównanie cykli algorytmów dla metrycznego problemu komiwojażera

Dla ustalonej liczby wierzchołków $n = 20$ przetestowano przykładowy graf metryczny. Wierzchołki grafu odpowiadają losowym punktom wewnątrz kwadratu jednostkowego. Wagi krawędzi są równe odległości euklidesowej pomiędzy punktami, czyli mamy do czynienia z instancją metrycznego problemu komiwojażera.

Wyniki testów znajdywania cyklu Hamiltona w grafie metrycznym przedstawiają poniższe rysunki. Każdy rysunek przedstawia cykl Hamiltona znaleziony przez dany algorytm. W testach wykorzystano: algorytm siłowy, algorytm najbliższego sąsiada, algorytm najbliższego sąsiada z powtórzeniami, algorytm sortowania krawędzi oraz algorytm z wykorzystaniem minimalnego drzewa rozpinającego.

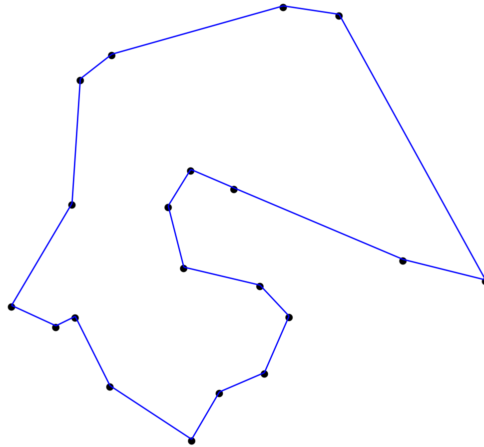


Rysunek B.8. Cykl według algorytmu najbliższego sąsiada rozwiązującego problem komiwojażera dla grafu metrycznego. Do testu wykorzystano przykładowy graf pełny metryczny.

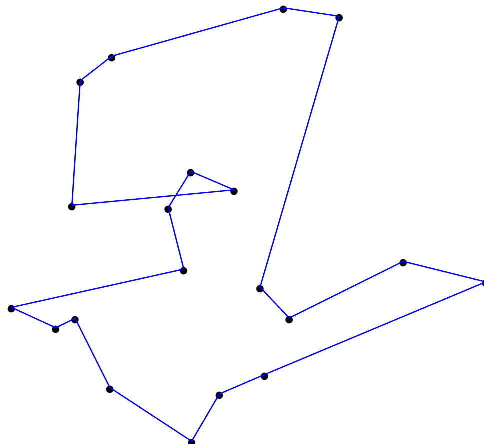


Rysunek B.9. Cykl według algorytmu najbliższego sąsiada z powtórzeniami rozwiązującego problem komiwojażera dla grafu metrycznego. Do testu wykorzystano przykładowy graf pełny metryczny.

69:6165575819



Rysunek B.10. Cykl według algorytmu sortowania krawędzi rozwiązującego problem komiwojażera dla grafu metrycznego. Do testu wykorzystano przykładowy graf pełny metryczny.



Rysunek B.11. Cykl według algorytmu z minimalnym drzewem rozpinającym rozwiązującego problem komiwojażera dla grafu metrycznego. Do testu wykorzystano przykładowy graf pełny metryczny.

70:9777920343

Minimalne drzewo rozpinajace dla grafu metrycznego n=20

Rysunek B.12. Minimalne drzewo rozpinające dla grafu metrycznego. Do testu wykorzystano przykładowy graf pełny metryczny.

71:1140749511

C. Testy dla grafów skierowanych

W tym dodatku zamieszczone zostały wyniki testów dla grafów skierowanych.

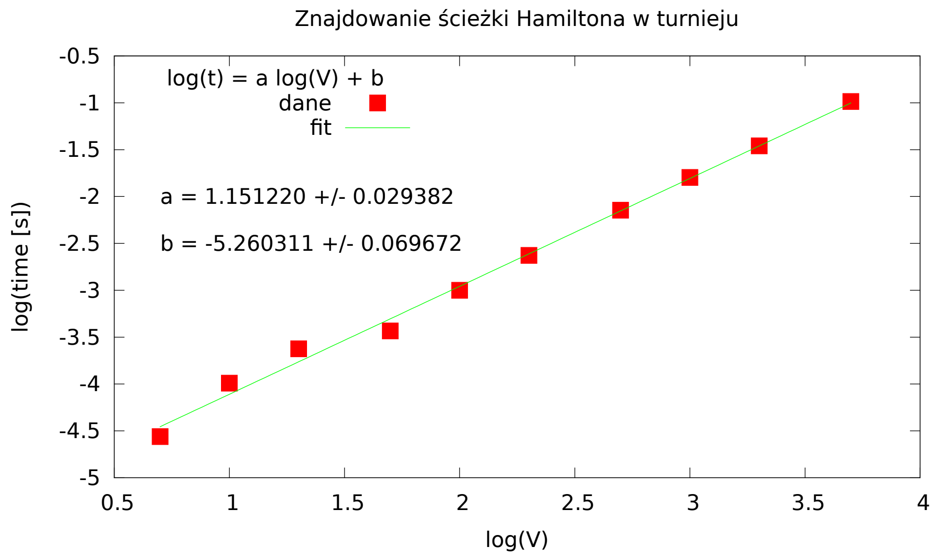
C.1. Testy wydajności znajdywania ścieżki Hamiltona w turnieju

Poniżej wykonano testy wydajności dla funkcji find_hamiltonian_path. Testy zostały zrobione dla 5, 10, 20, 50, 100, 200, 500, 1000, 2000 wierzchołków. Wyniki przedstawia rysunek C.1.

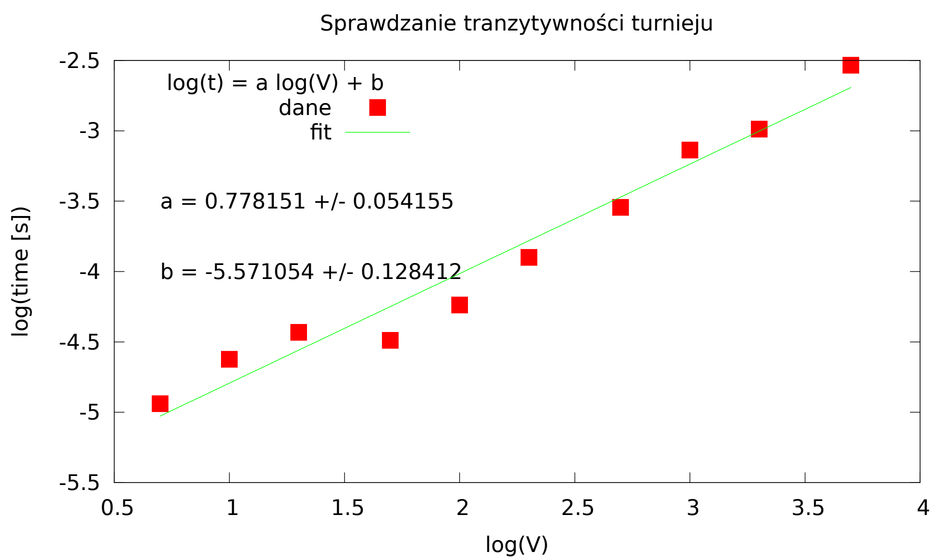
C.2. Testy wydajności sprawdzania tranzytywności w turnieju

Poniżej wykonano testy wydajności dla funkcji is_transitive . Testy zostały zrobione dla 5, 10, 20, 50, 100, 200, 500, 1000, 2000 wierzchołków. Wyniki przedstawia rysunek C.2.

72:1336945802



Rysunek C.1. Test wydajności metody find_hamiltonian_path. Współczynnik a o wartości około 1.2 sugeruje zależność powyżej liniowej, co jest zgodne z teoretyczną granicą $O(V \log V)$.



Rysunek C.2. Test wydajności metody is_transitive . Współczynnik a bliski 1 potwierdza zależność $O(V)$.

73:1117036796

Bibliografia

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

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

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

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

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

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

[7] Peter C. Norton, Alex Samuel, Dave Aitel, *Beginning Python*, 2005.

[8] Wikipedia, Graf hamiltonowski, 2015,
http://pl.wikipedia.org/wiki/Graf_hamiltonowski.

[9] Wikipedia, Cycle graph, 2015,
http://en.wikipedia.org/wiki/Cycle_graph.

[10] Wikipedia, Complete graph, 2015,
http://en.wikipedia.org/wiki/Complete_graph.

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

[12] Wikipedia, Knight's tour, 2015,
http://en.wikipedia.org/wiki/Knight's_tour.

[13] A. Conrad, T. Hindrichs, H. Morsy, and I. Wegener, *Solution of the Knight's Hamiltonian Path Problem on Chessboards*, Discrete Applied Mathematics 50, 125–134 (1994).

[14] G. A. Dirac, *Some problems on abstract graphs*, Proceedings of the London Mathematical Society 3rd Ser. 2, 69-81 (1952).

[15] O. Ore, *Note on Hamilton circuits*, The American Mathematical Monthly 67, 55 (1960).

[16] L. Pósa, *A theorem concerning Hamilton lines*, Magyar Tud. Akad. Mat. Kutató Int. Közl. 7, 225-226 (1962).

[17] V. Chvátal, *On Hamilton's ideals*, Journal of Combinatorial Theory, Series B 12, 163–168 (1972).

[18] J. A. Bondy, V. Chvátal, *A method in graph theory*, Discrete Mathematics 15, 111-135 (1976).

[19] A. Dharwadker, *A new algorithm for finding hamiltonian circuits*, 2004,
http://www.dharwadker.org/hamilton/.

[20] Wikipedia, Travelling salesman problem, 2015,
http://en.wikipedia.org/wiki/Travelling_salesman_problem.

[21] N. Christofides, *Worst-case analysis of a new heuristic for the travelling sa-*

74:9395391522

lesman problem, Report 388, Graduate School of Industrial Administration, CMU, 1976.

[22] Wikipedia, Metaheurystyka, 2015, https://pl.wikipedia.org/wiki/Metaheurystyka.

[23] Wikipedia, Local search (optimization), 2015, https://en.wikipedia.org/wiki/Local_search_(optimization).

[24] Wikipedia, Simulated annealing, 2015, https://en.wikipedia.org/wiki/Simulated_annealing.

[25] GSL - GNU Scientific Library, version 1.16, 2013, http://www.gnu.org/software/gsl/.

[26] Todd W. Schneider, *The Traveling Salesman with Simulated Annealing, R, and Shiny*, 2014, http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-

[27] Wikipedia, Tabu search, 2015, https://en.wikipedia.org/wiki/Tabu_search.

[28] F. Glover, *Future Paths for Integer Programming and Links to Artificial Intelligence*, Computers and Operations Research 5, 533-549 (1986).

[29] J. G. G. Becerra, R. J. A. Amado, *A tabu search approach for the travelling salesman problem*, Workshop, 2006, http://www.redheur.org/sites/default/files/metodos/TS02.pdf.

[30] Wikipedia, Ant colony optimization algorithms, 2015, https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms.

[31] Jinhui Yang, Xiaohu Shi, Maurizio Marchese, Yanchun Liang, *An ant colony optimization method for generalized TSP problem*, Progress in Natural Science 18, 1417–1422 (2008). http://www.sciencedirect.com/science/article/pii/S1002007108002736.

[32] D. R. Woodall, *Sufficient conditions for circuits in graphs*, Proc. London Math. Soc. 24, 739-755 (1972).

[33] Wikipedia, Topological sorting, 2015, http://en.wikipedia.org/wiki/Topological_sorting.

[34] Wikipedia, Tournament (graph theory), 2015, https://en.wikipedia.org/wiki/Tournament_(graph_theory).

[35] A. Bar-Noy, J. Naor, *Sorting, Minimal Feedback Sets and Hamilton Paths in Tournaments*, SIAM Journal on Discrete Mathematics 3, 7–20 (1990).

75:1058233915