Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Piotr Wlazło

Nr albumu: 1137140

Wybrane algorytmy grafowe kolorowania krawędzi

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2019

Oświadczenie autora pracy

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

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

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

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

Kraków, dnia Podpis kierującego pracą

Składam serdeczne podziękowania Panu dr. hab. Andrzejowi Kapanowskiemu, promotorowi mojej pracy licencjackiej za jego pomoc, uwagi oraz poświęcony czas, dzięki któremu niniejsza praca powstała w tym kształcie i formie.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów kolorowania krawędzi grafów. Zebrano wyniki teoretyczne dla grafów pełnych oraz dwudzielnych. Przygotowane zostały również testy poprawności i testy złożoności obliczeniowej korzystające ze standardowych modułów Pythona (unittest, timeit).

Zaimplementowano pięć algorytmów kolorowania krawędzi grafów: dla grafu pełnego, dla grafu dwudzielnego pełnego, dla grafu dwudzielnego prostego, dla grafu dwudzielnego regularnego, oraz dla grafu planarnego. Stworzono również generator grafów planarnych z ograniczonym największym stopniem wierzchołka.

Podczas implementacji korzystano z pakietu graphtheory rozwijanego w Instytucie Fizyki UJ. Przeprowadzono również migrację pakietu z Pythona 2 do Pythona 3.

Słowa kluczowe: grafy dwudzielne, grafy planarne, kolorowanie krawędzi, twierdzenie Vizinga

English title: Selected graph algorithms for edge coloring

Abstract

Python implementation of selected graph algorithms for edge coloring is presented. Known theoretical results for complete graphs, and bipartite graphs are collected. Tests for correctness and computational complexity are provided, were standard Python modules are used (unittest, timeit).

The algorithms for a proper edge coloring of selected graphs are presented: complete graphs, complete bipartite graphs, simple bipartite graphs, regular bipartite graphs, and planar graphs. Planar graph generators are provided where the maximum vertex degree is limited.

The graphtheory package is used during implementation. Package migration from Python 2 to Python 3 is done.

Keywords: bipartite graphs, planar graphs, edge coloring, Vising's theorem

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy jest kolorowanie krawędzi grafów [1]. Jest to dział teorii grafów zaliczany do dziedziny optymalizacji dyskretnej. Zaprojektowano już wiele modeli przypisywania kolorów do krawędzi grafu. W tejże pracy zająłem się klasycznym kolorowaniem krawędzi grafu. Chodzi tu o przyporządkowanie krawędziom grafu kolorów w taki sposób, aby sąsiadujące krawędzie otrzymały różne kolory. Problem kolorowania krawędzi grafu polega na znalezieniu optymalnego sposobu rozwiązania tego zadania, co nie jest prostym zadaniem, gdyż kolorowanie krawędzi grafu zaliczamy do problemów NP-trudnych, czyli takich, w których nie są znane efektywne rozwiązania działające w czasie wielomianowym. Kolorowanie krawędzi grafu znalazło szerokie zastosowanie w praktyce. Dla przykładu kolorowanie krawędzi grafu pełnego znajduje zastosowanie w turniejach typu round-robin, gdzie liczbę rund chcemy rozbić na liczbę jak najmniej możliwą.

Innym klasycznym przykładem jest układanie rozkładu lekcji w szkole, gdzie nauczyciele muszą przeprowadzić pewną liczbę godzin zajęć z różnymi klasami. Każda godzina zajęć musi być przeprowadzona w innym przedziale czasu, przy czym wszystkie zajęcia w szkole powinny być zaplanowne w jak najmniejszej liczbie przedziałów czasu. Krawędzie grafu reprezentują godziny zajęć do przeprowadzenia, a kolory krawędzi odpowiadają przedziałom czasu.

Celem pracy jest implementacja algorytmów kolorowania krawędzi, które jeszcze nie są obecne w pythonowej bibliotece algorytmów grafowych rozwijanej w Instytucie Fizyki UJ [2]. Użycie języka Python [3] pomaga w czytelnym zapisie algorytmów bez pogorszenia oczekiwanej złożoności obliczeniowej. Kody algorytmów z biblioteki mogą służyć do nauki konkretnych algorytmów, ale także mogą pomóc w przygotowywaniu implementacji w innych językach programowania. Warto podkreślić jednolity interfejs grafów i algorytmów, a także działanie biblioteki w obu wersjach Pythona 2 i 3. W ramach niniejszej pracy nastąpiło dostosowanie kodu biblioteki do działania z Pythonem 3.

Zagadnienie kolorowania krawędzi grafów pojawiło się już w pracach Motyla [4] i Samsona [5], gdzie przedstawiono najbardziej popularne algorytmy. W niniejszej pracy chcemy przedstawić algorytmy albo rzadziej cytowane, albo algorytmy często opisywane słownie, ale bez działających implementacji pozwalających wykonać konkretne obliczenia. Podstawy teorii grafów są opisane w szeregu książek w języku polskim, np. [6], [7], [8], [9], [10], ale opisy mniej popularnych algorytmów należy szukać w artykułach źródłowych.

Praca została podzielona na części stopniowo rozwijające tematykę kolorowania krawędzi grafów. Rozdział 1 jest krótkim wprowadzeniem do tematu bieżącej pracy. Rozdział 2 w sposób przystępny prezentuje podstawowe definicje z zakresu teorii grafów oraz kolorowania ich krawędzi. Rozdział 3 prezentuje implementację grafów ze strukturami danych użytymi w przedstawionych algorytmach. Rozdział 4 prezentuje po kolei implementacje kolorowania krawędzi różnych grafów. Rozdział 5 zawiera podsumowanie całej pracy. W dodatku A znajdują się wyniki testów złożoności obliczeniowej dla wszystkich algorytmów zaimplementownych w tej pracy. W dodatku B opisano najważniejsze różnice między Pythonem 2 a Pythonem 3, a także podano praktyczne sposoby na tworzenie kodu niezależnego od wersji Pythona.

2. Teoria grafów

Teoria grafów jest działem matematyki zajmująca się badaniem własności grafów. Dziedzina ta jest istotną częścią wielu gałęzi nauki takich jak informatyka, genetyka, socjologia, lingwistyka, czy badania operacyjne. Za pierwszą pracę na temat teorii grafów uznaje się opublikowaną w 1741 roku *Solutio problematis ad geometriam situs pertinentis w Commentarii academiae scientiarum Petropolitanae* pióra Leonarda Eulera, w której to opisane zostało *zagadnienie mostów królewieckich*.

2.1. Podstawowe definicje

Graf $G = (V, E)$ jest to uporządkowana para składająca się ze zbioru wierzchołków V oraz ze zbioru krawędzi E. Krawędzie grafu mogą posiadać wyznaczony kierunek dzięki czemu istnieje podział na grafy skierowane i nieskierowane. Krawędzie grafu mogą mieć przypisane pewne atrybuty liczbowe (wagi) lub atrybuty tekstowe (kolory). Wagi krawędzi mogą obrazować na przykład koszt lub czas przejazdu między wierzchołkami połączonymi krawędziami.

2.1.1. Grafy skierowane i nieskierowane

Definicja: *Graf nieskierowany (prosty)* jest to para uporządkowana $G = (V, E)$, gdzie V jest niepustym zbiorem wierzchołków, a E to zbiór krawędzi nieskierowanych,

$$E \subseteq \{\{u, v\} : u, v \in V\}. \tag{2.1}$$

W takim grafie krawędź $\{u, v\}$ jest zbiorem składającym się z dwóch różnych wierzchołków, których kolejność nie ma znaczenia.

Definicja: *Graf skierowany (prosty)* jest to para uporządkowana $G = (V, E)$, gdzie V jest niepustym zbiorem wierzchołków, a E to zbiór krawędzi skierowanych,

$$E \subseteq \{(u, v) : u, v \in V\}. \tag{2.2}$$

W takim grafie krawędź (u, v) jest uporządkowaną parą składającą się z dwóch różnych wierzchołków, z początkiem w pierwszym wierzchołku, a końcu w drugim. Dla wygody często dla krawędzi stosuje się oznaczenie uv dla grafu skierowanego i nieskierowanego.

Definicja: *Stopień wierzchołka* jest to liczba krawędzi przylegająca do danego wierzchołka. Jest on równy sumie wszystkich krawędzi wchodzących, wychodzących i pętli, które liczymy podwójnie. Stopień wierzchołka v oznacza się poprzez $\deg(v)$.

2.1.2. Ścieżki i cykle

Definicja: *Sciezka* od wierzchołka v_0 do wierzchołka v_k w grafie $G = (V, E)$ to taki ciąg wierzchołków $(v_0, v_1, \ldots, v_k)$, że dla każdego $i \in \{0, 1, ..., k-1\}$ istnieje krawędź $v_i v_{i+1}$, a wierzchołki mogą się powtarzać. Liczba przeskoków k stanowi długość takiej ścieżki. *Ścieżka prosta* zaś to taka ścieżka, w której wierzchołki się nie powtarzają.

Definicja: *Cykl* to ścieżka w której wierzchołek początkowy i końcowy są takie same, $v_0 = v_k$. *Cykl prosty* to cykl, w którym wierzchołki się nie powtarzają, za wyjątkiem wierzchołka początkowego i końcowego. W literaturze za cykl prosty uważa się każdą pętlę, a także dwie krawędzie równoległe nieskierowane. Graf który nie zawiera cykli nazywamy grafem *acyklicznym*.

2.1.3. Spójność

Definicja: Graf nieskierowany jest *spójny* (ang. *connected*), jeśli każdą parę wierzchołków tego grafu łączy ścieżka nieskierowana.

Definicja: Graf skierowany jest *silnie spójny* (ang. *strongly connected*), jeśli pomiędzy każdą parą wierzchołków tego grafu istnieje ścieżka skierowana.

2.2. Wybrane rodziny grafów

Grafy pełne: *Graf pełny* (ang. *complete graph*) [11] to graf nieskierowany prosty, w którym każda para różnych wierzchołków jest połączona krawędzią nieskierowaną. Taki graf o n wierzchołkach oznacza się jako K_n.

Grafy dwudzielne: *Graf dwudzielny* (ang. *bipartite graph*) [12] to graf, którego zbiór wierzchołków można podzielić na dwa rozłączne, niepuste zbiory $V = A \cup B$ tak, aby krawędzie nie łączyły wierzchołków należących do tego samego zbioru.

Graf pełny dwudzielny (ang. *complete bipartite graph*) jest to graf dwudzielny $K_{p,q}$ $(p, q \geqslant 1)$, gdzie zbiór wierzchołków podzielony jest na dwa niepuste rozłączne podzbiory o liczności p i q i istnieje krawędź pomiędzy każdą parą wierzchołków pochodzących z różnych podzbiorów.

Grafy regularne: *Graf regularny stopnia k* (ang. *regular graph*) [13] to graf w którym z każdego z jego wierzchołków wychodzi dokładnie k krawędzi. Taki graf określa się również grafem k-regularnym. Graf pełny K_n jest $(n-1)$-regularny. Grafy 3-regularne nazywane są *grafami kubicznymi* (graf Petersena).

Grafy eulerowskie: *Graf Eulera* (ang. *Eulerian graph*) [14] to taki graf, w którym istnieje cykl Eulera przechodzący przez wszystkie krawędzie. W przypadku grafów nieskierowanych grafy Eulera mają wszystkie wierzchołki stopnia parzystego. W przypadku grafów skierowanych grafy Eulera mają w każdym wierzchołku równą liczbę krawędzi wchodzących i wychodzących.

Grafy planarne: *Graf planarny* (ang. *planar graph*) [15] to taki graf, który może zostać narysowany na płaszczyźnie tak, aby jego krawędzie nie przecinały się ze sobą. Odwzorowanie grafu planarnego na płaszczyźnie nazywamy *rysunkiem płaskim grafu.*

2.3. Kolorowanie krawędzi grafów

Kolorowanie krawędzi grafu to przyporządkowanie kolorów (np. etykiet napisowych, liczb) do krawędzi w taki sposób, aby żadna z krawędzi mających jeden koniec w danym wierzchołka grafu nie była pokolorowana na ten sam kolor [1]. Problem kolorowania krawędzi grafu polega na znalezieniu sposobu, aby możliwe było pokolorowanie krawędzi używając co najwyżej k kolorów lub jak najmniejszej liczby kolorów. Mówimy wtedy, że graf jest *k-kolorowalny wierzchołkowo*. Minimalną liczbę kolorów potrzebną do pokolorowania wszystkich krawędzi grafu nazywamy *indeksem chromatycznym* (ang. *chromatic index*), a oznaczamy ją jako $\chi^{'}(G)$ lub $\chi_1(G)$. Problem znalezienia optymalnego rozwiązania kolorowania krawędzi grafu jest NP-zupełny dla $k \geqslant 3$ [8].

Twierdzenie Vizinga: Twierdzenie mówi, że dla każdego grafu nieskierowanego liczba kolorów potrzebna do pokolorowania krawędzi jest równa co najmniej Δ kolorów i co najwyżej $\Delta + 1$ kolorów, gdzie Δ to maksymalny stopień wierzchołka grafu. Na bazie tego twierdzenia grafy można podzielić na dwie klasy: dla grafów z klasy pierwszej wystarczy Δ kolorów, dla grafów z klasy drugiej potrzeba $\Delta + 1$ kolorów. Problem decyzyjny polegający na zdecydowaniu do której klasy należy dany graf jest NP-zupełny, nawet przy zawężeniu do grafów kubicznych.

2.4. Kolorowanie krawędzi grafów ogólnych

Wykorzystanie kolorowania krawędzi grafów ogólnych nasuwa się samoistnie. Praktycznym przykładem jest np. układanie terminarzu rozgrywek sportowych, w których każda drużyna musi zagrać z każdą. Niech zbiór wierzchołków grafu reprezentują drużyny biorące udział w turnieju, zaś zbiór krawędzi - zbiór meczy do rozegrania. Załóżmy, że kolorujemy krawędzie przypisując im kolory od 1 do Δ. Wtedy to każdy kolor będzie reprezentował kolejkę w której będą rozgrywane odpowiednie mecze i tak w pierwszej kolejce zmierzą się ze sobą zespoły, które są połączone krawędzią pokolorowaną na numer 1, itd.

W tym przypadku nie ma znaczenia czy liczba zespołów jest parzysta czy nie, gdyż w przypadku gdy dowolny wierzchołek nie będzie miał przypisanej do siebie krawędzie z kolorem przykładowo 5, to zespół będzie po prostu "pauzował" w kolejce numer 5. Nie robi różnicy również dodanie lub usunięcie jaiegoś zespołu po utworzeniu terminarza, gdyż możemy wtedy odpowiednio pokolorować krawędzie jednego wierzchołka lub usunąć również jego krawędzie.

2.5. Kolorowanie krawędzi grafów dwudzielnych

W praktycznych zastosowaniach często w naturalny sposób pojawiają się grafy dwudzielne. Rozważmy problem układania planu lekcji w pewnej szkole. Mamy zbiór A nauczycieli i zbiór B klas. W jednym tygodniu nauczyciel ma poprowadzić pewną liczbę godzin zajęć z każda klasą, co zaznaczamy jako odpowiednią liczbę krawędzi łączących nauczyciela z klasą. W ten sposób naturalnie pojawiają się multigrafy. Jest jasne, że nauczyciel każdą godzinę swoich zajęć musi przeprowadzić w innym przedziale czasowym, co odpowiada różnym kolorom krawędzi. Podobnie klasa ma odbyć każdą godzinę zajęć w innych przedziale czasowym. Dążenie do znalezienia najmniejszej liczby kolorów krawędzi odpowiada szukaniu najmniejszej liczby godzin, w których nauczyciele i klasy muszą przebywać w szkole.

W tym kontekście warto zauważyć, że dla rosnącej liczby uczniów i klas w szkole zasadniczo nie zmienia się liczba godzin zajęć prowadzonych w tygodniu przez danego nauczyciela, jak również liczba godzin zajęć danej klasy w tygodniu. W języku grafów nie zmienia się największy stopień wierzchołka grafu Δ. Stąd czasem złożoność obliczeniową algorytmów podaje się używając parametru Δ, obok typowych parametrów takich jak liczba wierzchołków n, czy liczba krawędzi m. Zgodnie z twierdzeniem, do pokolorowania krawędzi grafu dwudzielnego wystarczy Δ kolorów, czyli grafy dwudzielne należą do klasy pierwszej.

2.6. Kolorowanie krawędzi grafów regularnych

Rozłożenie krawędzi grafu k-regularnego na skojarzenia doskonałe, nazywane inaczej *1-faktoryzacją* (ang. *1-factorization*), jest tym samym to kolorowanie krawędzi grafu z $\Delta = k$. Oznacza to, że graf posiada *1-faktoryzację* wtedy i tylko wtedy, gdy graf jest klasy pierwszej.

Warto zaznaczyć, że nie każdy graf regularny posiada 1-faktoryzację. Przykładem takiego grafu jest graf Petersena, który jest grafem klasy drugiej. W nawiązaniu do twierdzenia Königa, każdy regularny graf dwudzielny posiada *1-faktoryzację* [18].

2.7. Kolorowanie krawędzi grafów planarnych

Grafy planarne z $\Delta \in \{2, 3, 4, 5\}$ mogą należeć do klasy pierwszej lub drugiej. Dla $\Delta = 2$ do klasy drugiej należą tylko takie grafy planarne, które posiadają cykl nieparzysty. Dla $\Delta = 3$ do klasy pierwszej należą przykładowo grafy kubiczne nie zawierające mostów, czyli krawędzi, których usunięcie spowoduje zwiększenie liczby spójnych składowych. Dla $\Delta \geqslant 7$ grafy planarne należą do klasy pierwszej. Przypadek $\Delta = 6$ nie jest rozstrzygnięty, ale przypuszcza się, że te grafy planarne również należą do klasy pierwszej.

3. Implementacja grafów

Algorytmy kolorowania krawędzi grafów zostały wykonane za pomocą biblioteki graphtheory rozwijanej w Instytucie Fizyki UJ [2]. Przedstawimy podstawowe struktury danych biblioteki i przykładowe obliczenia.

3.1. Grafowe struktury danych

Wierzchołek: Dowolny obiekt hashowalny służący jako klucz w tablicy hashowalnej.

Krawędź: Instancja klasy Edge. Posiada ona trzy atrybuty: source i target wskazujące odpowiednio na wierzchołek początkowy i końcowy krawędzi, oraz atrybut weight oznaczający wagę krawędzi (domyślnie równa jeden). Krawędź edge jest więc krawędzią skierowaną, a krawędź przeciwną otrzymujemy jako ~edge.

Graf: Instancja klasy Graph. Jego struktura przechowywana jest w postaci słownika języka Python, inaczej tablicy z hashowaniem. Atrybut directed określa czy graf jest skierowany. Jeśli graf nie jest skierowany, to w grafie przechowywane są jednocześnie dwie krawędzie skierowane przeciwne edge i ~edge.

Algorytm: Klasa posiadająca co najmniej dwie metody. Metoda __init__ (konstruktor) służy do inicjalizacji algorytmu, stworzenia potrzebnych struktur danych, itp. Metoda run() służy do uruchomienia właściwych obliczeń algorytmu. Wyniki działania algorytmu odczytujemy poprzez odwołanie się do jego atrybutów. Proste algorytmy są czasem zaimplementowane jako zwykłe funkcje, a nie klasy.

Kolorowanie wierzchołków Kolorowanie przechowywane jest w słowniku języka Python color z parami (node, **int**) lub (node, None) (brak przydzielonego koloru). Numeracja kolorów od 0 w górę.

Kolorowanie krawędzi: Kolorowanie przechowywane jest w słowniku języka Python color z parami (edge, **int**) lub (edge, None) (brak przydzielonego koloru). Numeracja kolorów od 0 w górę.

3.2. Przykładowe obliczenia

Przykład wykorzystania tej biblioteki pokazany jest poniżej. Przed uruchomieniem algorytmu należy zaimportować kilka modułów z tej biblioteki. Najważniejsze z nich to moduł graphs wraz z klasą Graph, moduł edges z klasą Edge, moduł factory z klasą GraphFactory do generowania wybranych grafów, oraz moduł z wybranym algorytmem kolorowania krawędzi.

Przykład 1: Kolorowanie krawędzi grafu pełnego z pięcioma wierzchołkami przy pomocy modułu edgecolorcomplete w sesji interaktywnej.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from edgecolorcomplete import CompleteGraphEdgeColoring
>>> gf = GraphFactory(Graph)
>>> G = gf.make_complete(5)
>>> algorithm = CompleteGraphEdgeColoring(G)
>>> algorithm.run()
>>> algorithm.color
{Edge(0, 1): 1, Edge(0, 2, 2): 2, Edge(0, 3, 3): 3, Edge(0, 4, 4): 4,
Edge(1, 2, 6): 3, Edge(1, 3, 8): 4, Edge(1, 4, 7): 0, Edge(2, 3, 5): 0,
Edge(2, 4, 10): 1, Edge(3, 4, 9): 2}
```

Przykład 2: Kolorowanie krawędzi grafu pełnego dwudzielnego z ośmioma wierzchołkami przy pomocy modułu edgecolorbipartitefull w sesji interaktywnej.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from edgecolorbiprtitefull import CompleteBipartiteGraphEdgeColoring
>>> gf = GraphFactory(Graph)
>>> G = gf.make_bipartite(4, 4, False, edge_propability=1.0)
>>> algorithm = CompleteBipartiteGraphEdgeColoring(G)
>>> algorithm.run()
>>> algorithm.color
{Edge(0, 4, 12): 0, Edge(0, 5, 14): 1, Edge(0, 6, 3): 2, Edge(0, 7, 6): 3,
Edge(1, 4, 10): 1, Edge(1, 5, 9): 2, Edge(1, 6): 3, Edge(1, 7, 11): 0,
Edge(2, 4, 15): 2, Edge(2, 5, 16): 3, Edge(2, 6, 7): 0, Edge(2, 7, 2): 1,
Edge(3, 4, 8): 3, Edge(3, 5, 4): 0, Edge(3, 6, 13): 1, Edge(3, 7, 5): 2}
```

Przykład 3: Kolorowanie krawędzi grafu dwudzielnego ogólnego (przypadkowego) z ośmioma wierzchołkami przy pomocy modułu edgecolorbipartite w sesji interaktywnej.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from edgecolorbipartite import BipartiteGraphEdgeColoring
>>> gf = GraphFactory(Graph)
>>> G = gf.make_bipartite(4, 4, False, edge_propability=0.5)
>>> algorithm = BipartiteGraphEdgeColoring(G)
>>> algorithm.run()
```

```
>>> algorithm.color
{Edge(0, 4, 10): 0, Edge(0, 5, 3): 1, Edge(0, 7, 5): 2, Edge(1, 6, 15): 0,
Edge(2, 5, 13): 0, Edge(2, 6, 12): 1, Edge(3, 4, 11): 1, Edge(3, 5, 7): 2,
Edge(3, 7, 6): 0}
```

Przykład 4: Kolorowanie krawędzi grafu dwudzielnego regularnego z dziesięcioma wierzchołkami przy pomocy modułu edgecoloreuler w sesji interaktywnej.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from edgecoloreuler import EulerianEdgeColoring
>>> gf = GraphFactory(Graph)
>>> G = gf.make_cyclic(10)    # graf regularny stopnia 2
>>> algorithm = EulerianEdgeColoring(G)
>>> algorithm.run()
>>> algorithm.color
{Edge(0, 1, 5): 0, Edge(0, 9, 8): 1, Edge(1, 2, 10): 1, Edge(2, 3, 4): 0,
Edge(3, 4): 1, Edge(4, 5, 3): 0, Edge(5, 6, 2): 1, Edge(6, 7, 6): 0,
Edge(7, 8, 7): 1, Edge(8, 9, 9): 0}
```

Przykład 5: Kolorowanie krawędzi grafu planarnego. W ramach pracy przygotowane zostały generatory grafów planarnych przyjmujące jako argumenty liczbę wierzchołków oraz maksymalny stopień grafu. Poniżej przykład generowania grafu planarnego przy pomocy modułu planartools w sesji interaktywnej.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from planartools import make_planar_delta
>>> G = make_planar_delta(n=8, Delta=6)
>>> G.show()
0 : 1(10) 2(11) 3(14) 4(5) 5(6) 7
1 : 0(10) 2(13) 3(18) 5(4) 6(8)
2 : 0(11) 1(13) 3(15) 4(9) 5(3) 6(16)
3 : 0(14) 1(18) 2(15) 4(7) 6(2) 7(17)
4 : 2(9) 0(5) 3(7) 7(12)
5 : 0(6) 2(3) 1(4)
6 : 1(8) 2(16) 3(2)
7 : 0 3(17) 4(12)
```

Dalej pokażemy przykład kolorowania krawędzi grafu planarnego z dziesięcioma wierzchołkami i maksymalnym stopniu równym 12 przy pomocy modułu edgecolorplanar w sesji interaktywnej.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from edgecolorplanar import PlanarGraphEdgeColoring
>>> from planartools import make_planar_delta
>>> G = make_planar_delta(n=10, Delta=12)
>>> algorithm = EulerianEdgeColoring(G)
>>> algorithm.run()
```

```
>>> algorithm.color
{Edge(0, 1, 24): 3, Edge(0, 2, 16): 2, Edge(0, 3, 3): 4, Edge(0, 4, 5): 7,
Edge(0, 7, 14): 5, Edge(0, 8, 9): 1, Edge(0, 9, 7): 0, Edge(1, 2, 12): 4,
Edge(1, 3, 22): 0, Edge(1, 4, 17): 6, Edge(1, 5, 23): 1, Edge(1, 6, 6): 2,
Edge(2, 3, 20): 1, Edge(2, 7, 19): 0, Edge(3, 4, 15): 5, Edge(3, 5, 10): 7,
Edge(3, 7, 2): 3, Edge(3, 8, 13): 6, Edge(3, 9, 4): 2, Edge(4, 5, 8): 0,
Edge(4, 6): 1, Edge(5, 6, 11): 3, Edge(7, 8, 21): 2, Edge(8, 9, 18): 3}
```

4. Algorytmy

Kod stworzony w ramach niniejszej pracy zawiera implementację kilku wybranych algorytmów kolorowania krawędzi grafów. Algorytmy zaimplementowano jako klasy z jednolitym interfejsem użytkownika.

4.1. Kolorowanie krawędzi grafu pełnego

Kolorowanie krawędzi grafu pełnego K_n należy podzielić na dwa przypadki w zależności od parzystości liczby wierzchołków n. Dla n nieparzystego wykorzystuje się n kolorów krawędzi. Algorytm można opisać następująco. Narysuj wierzchołki grafu na okręgu tak, aby powstał wielokąt foremny. Pokoloruj n krawędzi na obwodzie używając n kolorów. Pokoloruj krawędzie wewnątrz wielokąta używając koloru krawędzi równoległej leżącej na obwodzie. W naszej implementacji przyporządkowujemy wierzchołkom indeksy od 0 do $n-1$. Wtedy krawędzie równoległe można łatwo rozpoznać przez obliczenie sumy indeksów wierzchołków końcowych (modulo n), a ta suma jest jednocześnie użyta jako kolor tych krawędzi równoległych.

Dla n parzystego algorytm przydziela krawędziom $n-1$ kolorów. Najpierw tymczasowo usuwamy dowolny wierzchołek v wraz z krawędziami incydentnymi, aby dostać graf pełny z nieparzystą liczbą wierzchołków. Kolorujemy krawędzie tego grafu opisanym wcześniej algorytmem. Przywracamy usunięty tymczasowo wierzchołek v i krawędzie incydentne. Każdy sąsiad wierzchołka v ma niewykorzystany jeden z $n-1$ kolorów krawędzi, więc możemy wykorzystać ten kolor do pokolorowania krawędzi łączącej sąsiada z wierzchołkiem v.

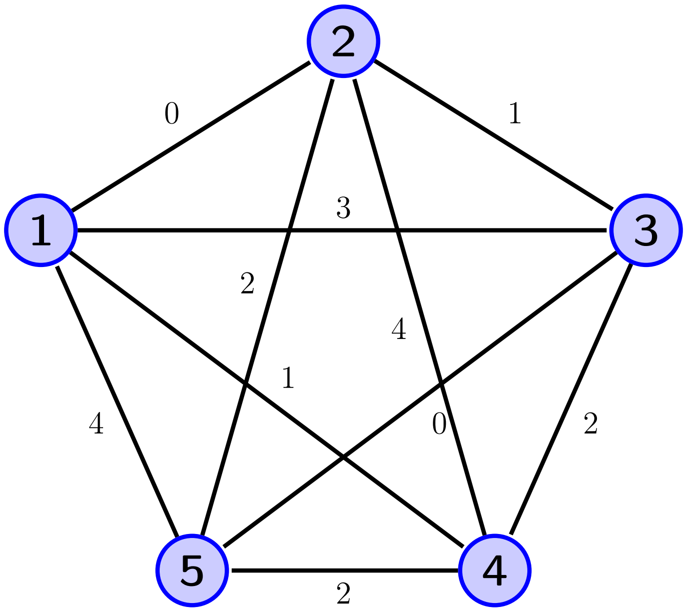
Listing 4.1. Moduł edgecolorcomplete.

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

try:
    integer_types = (int, long)
except NameError:   # Python 3
    integer_types = (int,)
    xrange = range

class CompleteGraphEdgeColoring:
    """Find an edge coloring for a complete graph."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict()
```



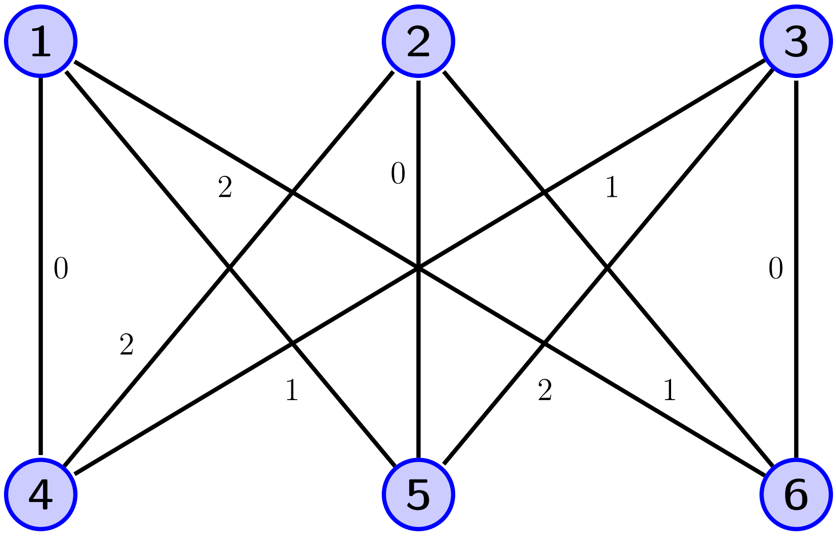
Rysunek 4.1. Graf pełny K_5 z $n = 5$ i $\Delta = 4$ po operacji kolorowania krawędzi. Potrzeba $n = \Delta + 1 = 5$ kolorów.

```
        self.m = 0    # graph.e() is slow
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
            else:
                self.color[edge] = None    # edge.source < edge.target
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")

    def run(self):
        """Executable pseudocode."""
        if self.graph.v() % 2 == 1:
            self.run_odd()
        else:
            self.run_even()

    def run_odd(self):
        """Edge coloring for n odd (n colors)."""
        n = self.graph.v()
        # Numerowanie wierzcholkow.
        D = dict((node, i) for (i, node) in enumerate(self.graph.iternodes()))
        # Kolorowanie krawedzi.
        for edge in self.graph.iteredges():
            c = (D[edge.source] + D[edge.target]) % n
            self.color[edge] = c

    def run_even(self):
        """Edge coloring for n even (n-1 colors)."""
        n = self.graph.v()
        removed_node = next(self.graph.iternodes())
        removed_edges = list(self.graph.iteroutedges(removed_node))
        self.graph.del_node(removed_node)
```



Rysunek 4.2. Graf dwudzielny pełny $K_{3,3}$ z $n = 6$ i $\Delta = 3$ po operacji kolorowania krawędzi. Potrzeba Δ kolorów.

```
self.run_odd()
# Teraz trzeba znalezc brakujace kolory przy wierzcholkach.
free = dict((node, set(xrange(n-1)))
    for node in self.graph.iternodes())
for edge in self.graph.iteredges():
    c = self.color[edge]
    free[edge.source].remove(c)
    free[edge.target].remove(c)
# Przywracanie wierzcholka z krawedziami.
self.graph.add_node(removed_node)
for edge in removed_edges:
    c = free[edge.target].pop()
    self.graph.add_edge(edge)
    if edge.source > edge.target:
        edge = ~edge
    self.color[edge] = c
```

4.2. Kolorowanie krawędzi grafu dwudzielnego pełnego

Kolorowanie krawędzi grafu pełnego dwudzielnego K_{pq}, gdzie zbiór wierzchołków ma podział $V = A \cup B$, $|A| = p$, $|B| = q$. Wykorzystuje się $\Delta = \max(p, q)$ kolorów. Algorytm można opisać następująco. Wierzchołkom ze zbioru A przyporządkowujemy indeksy od 0 do $p - 1$. Wierzchołkom ze zbioru B przyporządkowujemy indeksy od 0 do $q-1$. Każdej krawędzi przyporządkowujemy kolor równy sumie indeksów wierzchołków (modulo Δ). Łatwo widać, że kolory krawędzi spotykających się przy każdym wierzchołku będą różne, bo indeksy wierzchołków osobno w zbiorach A i B są różne.

Listing 4.2. Moduł edgecolorbipartitefull .

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

```python
from bipartite import BipartiteGraphBFS as Bipartite

class CompleteBipartiteGraphEdgeColoring:
    """Find an edge coloring for a complete bipartite graph."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict()
        self.m = 0   # graph.e() is slow
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
            else:
                self.color[edge] = None   # edge.source < edge.target
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")
        algorithm = Bipartite(self.graph)   # O(V+E) time
        algorithm.run()
        # Slowniki na indeksy wierzcholkow.
        self.D1 = dict()
        self.D2 = dict()
        idx1 = 0
        idx2 = 0
        for node in self.graph.iternodes():   # O(V) time
            if algorithm.color[node] == 1:
                self.D1[node] = idx1
                idx1 += 1
            else:
                self.D2[node] = idx2
                idx2 += 1
        if self.m != len(self.D1) * len(self.D2):
            raise ValueError("the graph is not complete bipartite")

    def run(self):
        """Executable pseudocode."""
        # Liczba dostepnych kolorow krawedzi.
        Delta = max(len(self.D1), len(self.D2))
        for node in self.D1:   # lacznie czas O(E)
            for edge in self.graph.iteroutedges(node):
                # Konce krawedzi sa w D2.
                c = (self.D1[edge.source] + self.D2[edge.target]) % Delta
                if edge.source > edge.target:   # moga byc odwrocone krawedzie
                    edge = ~edge
                self.color[edge] = c
```

4.3. Kolorowanie krawędzi grafu dwudzielnego ogólnego

Dane wejściowe: Multigraf dwudzielny G.

Problem: Kolorowanie krawędzi multigrafu G.

Dane wyjściowe: Słownik color zawierający kolorowanie krawędzi.

Opis algorytmu: Najpierw wyznaczany jest największy stopień wierzchołka Δ. Jeżeli $\Delta = 2$, to do kolorowania krawędzi wykorzystywany jest algorytm *CS* (ang. *Connected Sequential*), który daje optymalne kolorowanie w czasie liniowym. Jeżeli $\Delta > 2$, to krawędzie są kolorowane wg dowolnej kolejności, np. kolejności wyznaczonej przez implementację (iterator krawędzi), przy czym wykorzystuje się jeden z Δ kolorów.

Jeżeli oba końce rozważanej krawędzi uv mają wspólny kolor brakujący, to ten kolor jest przydzielany krawędzi. Jeżeli nie ma wspólnego wolnego koloru dla końców rozważanej krawędzi uv, wtedy uruchamiana jest procedura przekolorowania krawędzi. Niech α oznacza wolny kolor dla wierzchołka u, β oznacza wolny kolor dla wierzchołka v.

Wyznaczamy najdłuższą ścieżkę P o początku w wierzchołku u, do której będą należeć krawędzie o kolorach na przemian β i α. Ścieżka na pewno istnieje i nie przechodzi przez wierzchołek v. Na ścieżce P zamieniamy ze sobą kolory β i α, co nie psuje poprawności kolorowania krawędzi grafu. W tym momencie kolor β staje się kolorem wolnym dla obu końców krawędzi uv i może być wykorzystany do kolorowania tej krawędzi.

Złożoność: Dla każdej krawędzi może zachodzić potrzeba wyznaczenia i przekolorowania ścieżki P w czasie $O(n)$. Wyznaczenie wspólnego koloru brakującego dla końców krawędzi również zajmuje czas $O(n)$, ze względu na użycie zbiorów. Złożoność czasowa algorytmu wynosi więc $O(nm)$. Złożoność pamięciową szacujemy na $O(n\Delta)$, ze względu na potrzebę przechowywania kolorów brakujących dla wierzchołków.

Uwagi: Przekolorowywanie krawędzi pojawia się także w algorytmie NTL, służącym do przybliżonego kolorowania krawędzi dowolnego grafu z użyciem $\Delta + 1$ kolorów.

Listing 4.3. Moduł edgecolorbipartite.

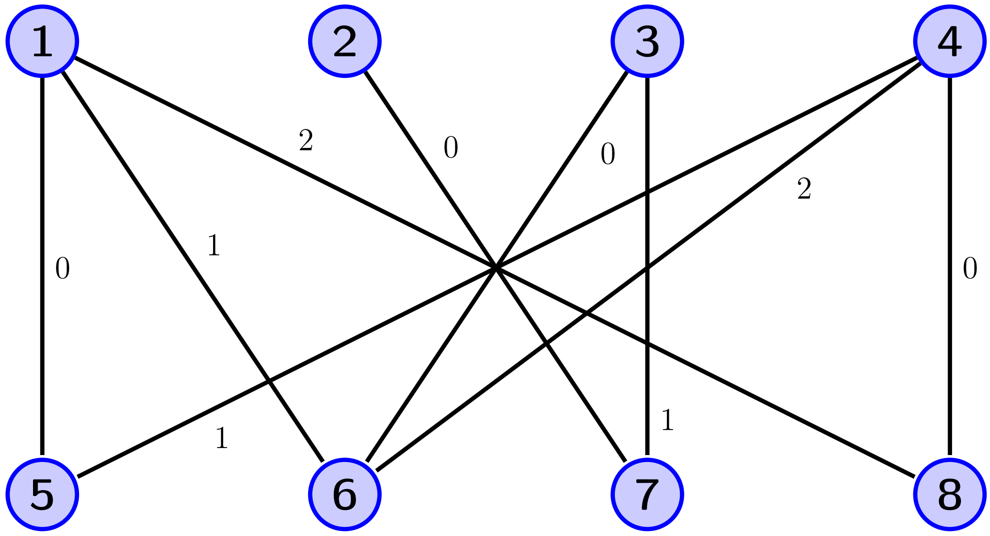
```
#!/usr/bin/python

try:
    integer_types = (int, long)
except NameError:   # Python 3
    integer_types = (int,)
    xrange = range

from bipartite import BipartiteGraphBFS as Bipartite
#from bipartite import BipartiteGraphDFS as Bipartite
from edgecolorcs import ConnectedSequentialEdgeColoring

class BipartiteGraphEdgeColoring:
    """Find an edge coloring for a bipartite graph."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
```



Rysunek 4.3. Graf dwudzielny ogólny z $n = 8$ i $\Delta = 3$ po operacji kolorowania krawędzi. Potrzeba Δ kolorów.

```
        self.graph = graph
        self.color = dict()
        self.m = 0    # graph.e() is slow
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
            else:
                self.color[edge] = None    # edge.source < edge.target
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")
        # Test czy graf jest dwudzielny.
        algorithm = Bipartite(self.graph)    # O(V+E) time
        algorithm.run()
        # dict with missing colors for nodes.
        self.missing = None

    def run(self):
        """Executable pseudocode."""
        # Ustal liczbe wykorzystywanych kolorow.
        Delta = max(self.graph.degree(node) for node in self.graph.iternodes())
        if Delta <= 2:
            # Greedy coloring suffies.
            algorithm = ConnectedSequentialEdgeColoring(self.graph)
            algorithm.run()
            self.color = algorithm.color
        else:
            self.missing = dict((node, set(xrange(Delta)))
                for node in self.graph.iternodes())
            for edge in self.graph.iteredges():
                # Sprawdz wspolny kolor brakujacy.
                both = self.missing[edge.source] & self.missing[edge.target]
                if len(both) == 0:
                    self._recolor(edge)
                else:
```

```
                c = min(both)
                self._add_color(edge, c)

    def _add_color(self, edge, c):
        """Add color."""
        if edge.source > edge.target:
            edge = ~edge
        self.color[edge] = c
        self.missing[edge.source].remove(c)
        self.missing[edge.target].remove(c)

    def _del_color(self, edge, c):
        """Delete color."""
        if edge.source > edge.target:
            edge = ~edge
        self.color[edge] = None
        self.missing[edge.source].add(c)
        self.missing[edge.target].add(c)

    def _recolor(self, edge):
        """Swap edge colors and add color."""
        # edge.source i edge.target maja rozne kolory brakujace.
        alpha = min(self.missing[edge.source])
        beta = min(self.missing[edge.target])
        # Tworze sciezke o poczatku w edge.source i kolorach
        # na przemian beta i alpha.
        # Sciezka sie urwie, jak nie znajdziemy danego koloru.
        # Na sciezce na pewno nie spotkamy edge.target, bo tam
        # nie ma koloru beta.
        path = []
        node = edge.source   # chodzi po wierzcholkach sciezki
        finished = False
        # Zmienna parity pozwala kontrolowac jaki kolor szukamy.
        parity = 0
        while not finished:
            finished = True
            if parity % 2 == 0:   # szukamy kolor beta
                for edge1 in self.graph.iteroutedges(node):
                    # Kolor krawedzi ma byc beta.
                    if edge1.source > edge1.target:
                        c = self.color[~edge1]
                    else:
                        c = self.color[edge1]
                    if c == beta:   # c moze byc None!
                        node = edge1.target
                        path.append(edge1)
                        finished = False   # bedziemy szukac drugiego koloru
                        break
            else:   # parity % 2 == 1, szukamy kolor alpha
                for edge1 in self.graph.iteroutedges(node):
                    # Kolor krawedzi ma byc alpha.
                    if edge1.source > edge1.target:
                        c = self.color[~edge1]
                    else:
                        c = self.color[edge1]
                    if c == alpha:   # c moze byc None!
                        node = edge1.target
```

```
                        path.append(edge1)
                        finished = False   # bedziemy szukac drugiego koloru
                        break
                parity += 1
            # Sciezka zostala znaleziona i na pewno istnieje.
            # Zamieniamy kolory alpha i beta na sciezce.
            # Najpierw usuwam kolory, pierwszy to beta.
            for i, edge1 in enumerate(path):
                c = beta if (i % 2 == 0) else alpha
                self._del_color(edge1, c)
            # Teraz dodaje kolory, pierwszy to alpha.
            for i, edge1 in enumerate(path):
                c = alpha if (i % 2 == 0) else beta
                self._add_color(edge1, c)
            # Teraz mamy wolny kolor beta dla krawedzi edge.
            self._add_color(edge, beta)
```

4.4. Kolorowanie krawędzi grafu dwudzielnego regularnego

W pewnej grupie algorytmów kolorowania krawędzi grafu dwudzielnego korzysta się z grafów dwudzielnych regularnych. Kolorowanie krawędzi ma wtedy trzy etapy: (1) uzupełnianie grafu dwudzielnego do grafu k-regularnego, (2) kolorowanie krawędzi grafu dwudzielnego k-regularnego, (3) usuwanie elementów dodanych w pierwszym etapie.

Wierzchołki grafu dwudzielnego k-regularnego można podzielić na dwa zbiory o równej liczności V_1 i V_2, przy czym każda krawędź ma jeden koniec w zbiorze V_1, a drugi koniec w zbiorze V_2. Stąd w etapie pierwszym czasem zachodzi konieczność dodania nowych wierzchołków, a także dodania nowych krawędzi. Jeżeli pierwotny graf był grafem prostym, to w pierwszym etapie można uzyskać uzupełniony graf prosty. Zwykle prościej jest jednak uzyskać po pierwszym etapie multigraf bez pętli.

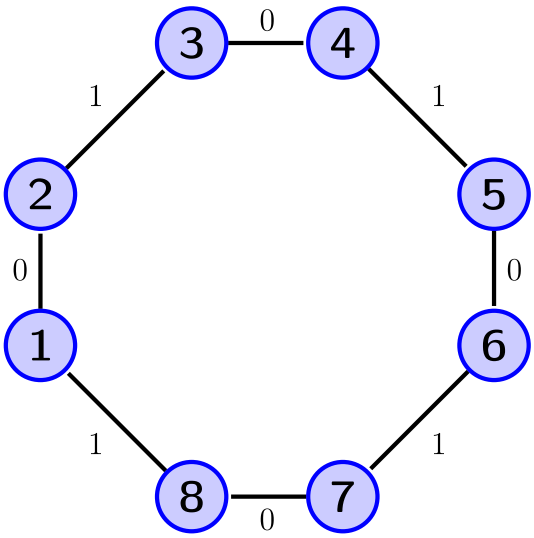
W naszej pracy zajmiemy się kluczowym etapem drugim, czyli kolorowaniem krawędzi multigrafu dwudzielnego k-regularnego. Schrijver podał algorytm działający w czasie $O(km)$ [16], który rozwija idee Gabowa dotyczące znajdowania skojarzenia doskonałego w grafach dwudzielnych k-regularnych [17].

Dane wejściowe: Multigraf dwudzielny k-regularny G.

Problem: Kolorowanie krawędzi multigrafu G.

Dane wyjściowe: Słownik color zawierający kolorowanie krawędzi.

Opis algorytmu: Najpierw sprawdzana jest poprawność grafu wejściowego. Dalsza część algorytmu ma strukturę rekurencyjną, opartą na metodzie find_colors(). Rekurencja stopniowo obniża największy stopień k przetwarzanych grafów, a dla $k = 1$ wszystkie krawędzie mogą być kolorowane tym samym kolorem. Jeżeli k jest nieparzyste, to znajdowane jest skojarzenie



Rysunek 4.4. Graf cykliczny 2-regularny C_n z $n = 8$ i $\Delta = 2$ po pokolorowaniu krawędzi. Potrzeba Δ kolorów.

doskonałe, które istnieje na mocy twierdzenia. Krawędzie należące do skojarzenia są usuwane z grafu (k obniża się o jeden) i kolorowane jednym kolorem. Jeżeli k jest parzyste, to znajdowany jest cykl Eulera. Krawędzie cyklu są na przemian przydzielane do dwóch mniejszych grafów dwudzielnych $(k/2)$-regularnych, które są dalej przetwarzane rekurencyjnie.

Uwaga 1: W naszej implementacji skojarzenie doskonałe znajdowane jest algorytmem opartym na szukaniu ścieżek rozszerzających, którego implementacja była dostępna w bibliotece. Jego złożoność jest szacowana na $O(nm)$. Schrijver proponuje w tym punkcie nowy algorytm działający w czasie $O(km)$, który polega na stopniowym selekcjonowaniu krawędzi należących do skojarzenia przez znajdowanie cykli i manipulowanie wagami krawędzi.

Uwaga 2: W naszej implementacji cykl Eulera znajdujemy metodą rekurencyjną opartą na przeszukiwaniu w głąb. Metoda ma optymalną złożoność $O(m)$, ale głębokość rekurencji jest rzędu liczby krawędzi. W praktyce dla dużych grafów prowadzi to do przekroczenia systemowego limitu rekurencji. Rozwiązaniem byłoby podanie nierekurencyjnej wersji algorytmu znajdowania cyklu Eulera.

Złożoność: Teoretyczna złożoność obliczeniowa czasowa algorytmu wynosi $O(km)$. Nasza implementacja ma trochę gorszą złożoność z powodu użycia innego algorytmu znajdowania skojarzenia doskonałego. Zauważmy, że dla $k = 2^t$ algorytm znajdujący skojarzenie doskonałe nigdy nie będzie wykonywany, a kolorowanie krawędzi może być znalezione w czasie $O(tm) = O(m \log k)$.

Listing 4.4. Moduł edgecoloreuler.

```
#!/usr/bin/python

from bipartite import BipartiteGraphBFS as Bipartite
```

```python
from MatchingUsingAugmentingPath import *
from eulerian_cycle import EulerianCycleDFSWithEdges

class EulerianEdgeColoring:
    """Find an edge coloring for a regular bipartite graph."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict()
        self.m = 0    # graph.e() is slow
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
            else:
                self.color[edge] = None    # edge.source < edge.target
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")
        # Test czy graf jest dwudzielny.
        algorithm = Bipartite(graph)   # O(V+E) time
        algorithm.run()
        self.v1 = set()
        self.v2 = set()
        for node in self.graph.iternodes():
            if algorithm.color[node] == 1:
                self.v1.add(node)
            else:
                self.v2.add(node)
        assert len(self.v1) == len(self.v2)
        k = self.graph.degree(node)
        # Test czy graf jest k-regularny. O(V) time.
        assert self.graph.v() * k == 2 * self.m
        if any(self.graph.degree(node) != k for node in self.graph.iternodes()):
            raise ValueError("the graph is not regular")
        # Powiekszenie limitu rekurencji.
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self):
        """Executable pseudocode."""
        node = next(self.graph.iternodes())
        k = self.graph.degree(node)
        self.find_colors(self.graph.copy(), k, 0)

    def find_colors(self, graph, k, free_color):
        """Edge coloring of a k-regular bipartite graph."""
        # Pracujemy na kopii grafu, wiec mozna ja modyfikowac.
        if k == 1:
            # Kolorujemy wprost krawedzie
            for edge in graph.iteredges():
                self.color[edge] = free_color
        elif k % 2 == 1:
            # Trzeba usunac jedno skojarzenie doskonale.
```

```
        algorithm = MatchingUsingAugmenticPath(graph)
        algorithm.run()    # powolne O(V*E)
        # Ustalam krawedzie nalezace do skojarzenia.
        matching = []
        for edge in graph.iteredges():
            if edge.target == algorithm.mate[edge.source]:
                matching.append(edge)
        for edge in matching:
            self.color[edge] = free_color
            graph.del_edge(edge)
        self.find_colors(graph, k-1, free_color+1)
        # Przeskoczymy do k parzystego, aby tam zrobic podzial.
    else:    # k % 2 == 0
        # Znajdujemy cykl Eulera.
        algorithm = EulerianCycleDFSWithEdges(graph)
        algorithm.run()
        # Podzial na dwa grafy (k/2)-regularne.
        graph2 = graph.__class__(graph.v())
        for node in graph.iternodes():    # te same wierzcholki
            graph2.add_node(node)
        for edge in algorithm.eulerian_cycle:
            if edge.source in self.v1:
                graph2.add_edge(edge)
                graph.del_edge(edge)
        k = k // 2
        if k == 1:    # ograniczam rekurencje
            # Kolorujemy wprost krawedzie
            for edge in graph.iteredges():
                self.color[edge] = free_color
            for edge in graph2.iteredges():
                self.color[edge] = free_color + k
        else:
            self.find_colors(graph, k, free_color)
            self.find_colors(graph2, k, free_color + k)
```

4.5. Generatory grafów planarnych

Przed stworzeniem implementacji algorytmu kolorowania krawędzi grafu planarnego należy przygotować generatory grafów planarnych, które będą wykorzystywane do testów wydajnościowych. Potrzebne będą grafy planarne z ograniczeniem na największy stopień wierzchołka Δ. Generatory zostały umieszczone w module planartools, a tworzone grafy planarne są ważone. Pierwszym argumentem generatorów jest liczba wierzchołków n, a drugim założony największy stopień wierzchołka Δ. Domyślnie nie ma ograniczenia na Δ. Zauważmy, że dla zbyt małych wartości Δ mogą być trudności z wygenerowaniem odpowiedniego grafu.

Pierwszy generator to funkcja make_planar_delta(), która tworzy sieć Apoloniusza. Generator startuje od grafu K_3, który ma dwie ściany trójkątne. Dalej w każdym kroku losowana jest jedna ze ścian, dodawany jest nowy wierzchołek na środku ściany, połączony z trzema wierzchołkami trójkątnej ściany. W ten sposób stara ściana jest dzielona na trzy nowe ściany trójkątne. Jest jasne, że po tej operacji stopnie wierzchołków starej ściany rosną o jeden, dlatego wolno dzielić ścianę jedynie wtedy, gdy stopnie wierzchołków są mniejsze od założonego stopnia Δ. Końcowa liczba krawędzi wynosi $m = 3n - 6$, liczba ścian $f = 2n - 4$.

Drugi generator to funkcja make_planar_square(), która tworzy grafy planarne ważone o ścianach kwadratowych (brzeg ściany to cykl C_4). Generator startuje od cyklu C_4, który ma dwie ściany kwadratowe. Dalej w każdym kroku losowana jest jedna ze ścian, dodawany jest nowy wierzchołek na środku ściany, połączony z dwoma przeciwległymi wierzchołkami kwadratowej ściany. W ten sposób stara ściana jest dzielona na dwie nowe kwadratowe ściany. Wolno dzielić ścianę jedynie wtedy, gdy dwa przeciwległe wierzchołki mają stopnie mniejsze od założonego stopnia Δ. Końcowa liczba krawędzi wynosi $m = 2n - 4$, liczba ścian $f = n - 2$.

4.6. Kolorowanie krawędzi grafu planarnego

Prezentowany algorytm pochodzi z pracy Cole i Kowalika z roku 2008 [19] i pozwala na kolorowanie krawędzi grafu planarnego liczbą kolorów nie większą niż $\max(\Delta, 12)$ w czasie liniowym. Praca zawiera także drugi algorytm działający w czasie liniowym, który wykorzystuje $\max(\Delta, 9)$ kolorów. Autorzy tych dwóch algorytmów bazują na dwóch koncepcjach. Pierwsza koncepcja to redukcja grafu i identyfikacja zestawu pewnych szczególnych konfiguracji. Druga koncepcja (występuje tylko w drugim algrytmie) to *discharging technique*, która oryginalnie powstała przy dowodzie twierdzenia o czterech kolorach. Druga koncepcja jest bardzo zaawansowana, dlatego zdecydowaliśny się na implementację tylko pierwszego algorytmu, który również niesie ze sobą wyzwania implementacyjne.

Dane wejściowe: Multigraf planarny G.

Problem: Kolorowanie krawędzi multigrafu G.

Dane wyjściowe: Słownik color zawierający kolorowanie krawędzi z liczbą kolorów ograniczoną przez $\max(\Delta, 12)$.

Opis algorytmu: Głównym celem algorytmu jest utworzenie z grafu wejściowego małego podgrafu poprzez usunięcie krawędzi bądź utworzenie określonych konfiguracji, pokolorowanie tych podgrafów, a następnie rozszerzenie kolorowania na cały graf wejściowy.

Na początku ustalany jest parametr Δ danego grafu i inicjalizowane są kolejki: dwie dla konfiguracji A i B, oraz jedna dla krawędzi które będą usuwane; są to *krawędzie redukowalne* (ang. *reducible edges*). Następnie znajdujemy krawędzie redukowalne oraz konfiguracje A i B, które są rozdzielne ze sobą tj. nie posiadają wspólnych krawędzi. Odbywa się to wszystko w czasie liniowym. Wyznaczamy również zbiór *wierzchołków extremalnych*, czyli wierzchołków stopnia 2 połączonych z dwoma wierzchołkami stopnia Δ. Wierzchołki ekstremalne pomagają w rozpoznaniu konfiguracji A i B. Rozpoznawanie konfiguracji A i B odbywa się w metodzie _find_A_B(), gdzie startując od wierzchołka ekstremalnego x znajduje się odpowiednio cykl $uxvy$ (stopnie $\Delta, 2, \Delta, 2$) lub ścieżkę $uxvyw$ (stopnie $\Delta, 2, \Delta, 2, \Delta$).

Algorytm przeprowadza dwie operacje modyfikujące graf: jedną jest zamiana 2-ścieżki na krawędź, a następnie usuwanie tej krawędzi. Wtedy też kolejki są uaktualniane. Drugą operacją jest usunięcie krawędzi redukowalnej. Wtedy zmniejszają się stopnie końców krawędzi i mogą pojawić się nowe krawędzie redukowalne lub nowe konfiguracje A i B.

Algorytm rekurencyjnie przetwarza wszystkie krawędzie redukowalne usuwając je, a przy braku krawędzi redukowalnych przetwarza konfiguracje A i B. Po wyczerpaniu wszystkich krawędzi grafu wstawiamy je ponownie do grafu, kolorujemy je zachłannie, przekolorowując krawędzie z konfiguracji A i B w czasie $O(1)$, jeśli to konieczne.

Złożoność: Teoretyczna złożoność obliczeniowa badanego algorytmu wynosi jest liniowa $O(n)$. Testy praktyczne dały również taką złożoność.

Uwaga 1: Kolejki dla konfiguracji A i B realizowane są jako stos (lista Pythona), ale może to być też inny kontener. Natomiast dla krawędzi redukowalnych wykorzystaliśmy zbiory, aby zapobiec powtarzaniu się krawędzi w kolejce. W naszej implementacji istotne jest, aby w kolejkach znajdowały się krawędzie edge, dla których edge.source < edge.target. Taką konwencję stosuje też iterator krawędzi iteredges() przy wypisywaniu jednej z dwóch krawędzi skierowanych edge i ~edge reprezentujących wewnętrznie jedną krawędź nieskierowaną.

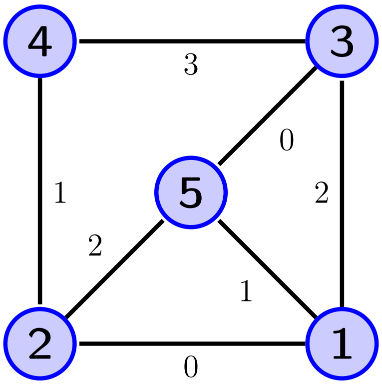
Uwaga 2: Algorytm ma strukturę rekurencyjną względem krawędzi grafu, przez co dla dużych grafów (n rzędu 10^4) występuje przekroczenie systemowego limitu rekurencji. Rozwiązaniem byłoby podanie nierekurencyjnej wersji algorytmu.

Uwaga 3: Autorzy algorytmu w swojej pracy [19] podali szereg wskazówek implementacyjnych, które pomagają otrzymać algorytm działający w czasie liniowym. Nie wszystkie wskazówki wykorzystaliśmy w naszej implementacji. Przykładowo autorzy sugerują, aby każda krawędź z konfiguracji A lub B przechowywała łącze do pozycji w kolejce Q_A lub Q_B. Wtedy przy obniżeniu stopnia jednego z końców krawędzi można szybko usunąć daną konfiguracje z kolejki. W naszym podejściu sprawdzamy poprawność konfiguracji A czy B dopiero przy pobieraniu jej z kolejki.

Inna sprawa to lista kolorów brakujących dla każdego wierzchołka. W naszej implementacji każdy wierzchołek startuje z listą $\max(\Delta, 12)$ kolorów brakujących, co daje czas $O(n\Delta)$ przy inicjalizacji. Autorzy sugerują dla wierzchołka v stworzenie listy $\min(\Delta, \deg(v)+1)$ kolorów brakujących, co dawałoby czas $O(n)$. Wtedy jednak może nie być wspólnego koloru brakującego dla obu końców danej krawędzi i wymagane będzie dodatkowe przekolorowanie. Nasze podejście jest wystarczająco szybkie w praktyce.

Listing 4.5. Moduł edgecolorplanar.

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



Rysunek 4.5. Graf planarny z $n = 5$, $m = 7$, $f = 4$, $\Delta = 3$ po pokolorowaniu krawędzi. Potrzeba $\Delta + 1$ kolorów (graf klasy 2).

```
#
# Algorytm z [2008 Cole Kowalik].
# Liczba przydzielonych kolorow to max(Delta, 12).

from edges import Edge
from graphs import Graph
from planartools import make_planar_delta
from planartools import make_planar_square
import sys

class PlanarGraphEdgeColoring:
    """Find an edge coloring for a planar graph."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        # Nie robimy testu planarnosci.
        self.graph = graph
        self.color = dict()
        self.m = 0    # graph.e() is slow
        for edge in self.graph.iteredges():    # O(E) time
            if edge.source == edge.target:
                raise ValueError("a loop detected")
            else:
                self.color[edge] = None    # edge.source < edge.target
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")
        # Ustal liczbe wykorzystywanych kolorow. O(n) time
        self.Delta = max(self.graph.degree(node) for node in self.graph.iternodes()
        # dict with missing colors for nodes.
        self.missing = dict((node, set(range(max(self.Delta, 12))))
            for node in self.graph.iternodes())    # O(Delta*n) time
        # Rekurencja idzie po krawedziach, co jest glebokie.
        # Ustawiam glebokosc rekurencji 2m=6n.
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(6 * self.graph.v(), recursionlimit))

    def run(self):
```


32:2823434365

```
        """Executable pseudocode."""
        self._preprocessing()
        self._coloring()

    def _preprocessing(self):
        """Finding reducible edges and configurations A and B."""
        self.Q_e = set()   # queue for reducible edges
        self.Q_A = []    # queue for configurations A
        self.Q_B = []    # queue for configurations B
        # Finding reducible edges.
        # Przechodzimy po wszystkich krawedziach, waga krawedzi to suma
        # stopni wchodzacych i wychodzacych node'a.
        # Warunki dolaczenia do reducible_edges.
        for edge in self.graph.iteredges():   # O(E) time
            weight = self.graph.degree(edge.source) + self.graph.degree(edge.target
            if weight <= 13:
                self.Q_e.add(edge)
            elif (self.graph.degree(edge.source) == 1 or
                  self.graph.degree(edge.target) == 1):
                self.Q_e.add(edge)
            elif (self.graph.degree(edge.source) == 2 and
                  self.graph.degree(edge.target) == self.Delta-1):
                self.Q_e.add(edge)
            elif (self.graph.degree(edge.target) == 2 and
                  self.graph.degree(edge.source) == self.Delta-1):
                self.Q_e.add(edge)
        # Finding configurations A and B.
        # Kazda krawedz moze byc zaliczona tylko do jednej konfiguracji.
        # D_2 krawedzie z deg 2 do deg Delta.
        # D_d krawedzie z deg Delta do deg 2.
        # D_de krawedzie z deg Delta do deg 2 extremal.
        # Klucze to wszystkie wierzcholki, bo moga sie zmieniac.
        self.D_2 = dict((node, set()) for node in self.graph.iternodes())
        self.D_d = dict((node, set()) for node in self.graph.iternodes())
        self.D_de = dict((node, set()) for node in self.graph.iternodes())
        # Jezeli Delta < 12, to do D_2 i D_d moga trafic krawedzie bedace
        # juz w Q_e. Nie jest to problem, bo najpierw oprozniamy Q_e,
        # a potem sprawdzamy poprawnosc konfiguracji wyjmowanych z Q_A i Q_B.
        for edge in self.graph.iteredges():   # O(m) time
            if (self.graph.degree(edge.source) == 2 and
                self.graph.degree(edge.target) == self.Delta):
                # Dodaje wszystkie krawedzie o okreslonej konfiguracji
                # do okreslonego slownika.
                # Jezeli spelniaja warunki.
                    self.D_2[edge.source].add(edge)
                    self.D_d[edge.target].add(~edge)
            elif (self.graph.degree(edge.source) == self.Delta and
                self.graph.degree(edge.target) == 2):
                    self.D_2[edge.target].add(~edge)
                    self.D_d[edge.source].add(edge)
        # A 2-vertex adjacent to two degree Delta vertices is extremal.
        self.extremal = set()
        for node in self.graph.iternodes():   # O(n) time
            if len(self.D_2[node]) == 2:
                self.extremal.add(node)
        for x in self.extremal:
            edge1, edge2 = self.D_2[x]
```

```
            # Bierzemy krawedzie z node'a extremal do dwoch delta wierzcholkow.
            self.D_de[edge1.target].add(~edge1)
            self.D_de[edge2.target].add(~edge2)
            # Dodajemy je do kolejki w odwroconej kolejnosci, poczatek w delta.
        for x in self.extremal:
            if len(self.D_2[x]) == 0:
                continue  # krawedzie wykorzystane
            self._find_A_B(x)

    def _find_A_B(self, node):
        """Finding configurations A and B starting from an extremal node."""
        # Bierzemy ze slownika krawedzie z extremal do delta.
        edge1 = self.D_2[node].pop()
        edge2 = self.D_2[node].pop()
        #dwie krawedzie
        assert len(self.D_2[node]) == 0
        v1 = edge1.target
        v2 = edge2.target
        # v1 i v2 to delta, usuwanie krawedzi ze slownikow.
        self.D_d[v1].remove(~edge1)
        self.D_d[v2].remove(~edge2)
        self.D_de[v1].remove(~edge1)
        self.D_de[v2].remove(~edge2)
        if len(self.D_de[v1]) > 0:
            # Szukamy edge3 prowadzacej do wierzcholka deg 2 z delta.
            edge3 = self.D_de[v1].pop()
            self.D_d[v1].remove(edge3)
            self.D_2[edge3.target].remove(~edge3)
            edge4 = self.D_2[edge3.target].pop()
            self.D_d[edge4.target].remove(~edge4)
            self.D_de[edge4.target].remove(~edge4)
            assert len(self.D_2[edge3.target]) == 0
            # Znalezione (~edge2, edge1, edge3, edge4).
            if edge4.target == v2:
                self.Q_A.append((~edge2, edge1, edge3, edge4))
                # Jesli zostal zatoczony cykl do drugiej delty.
            else:
                self.Q_B.append((~edge2, edge1, edge3, edge4))
                #print ("new B")
                # Dodajemy te krawedzie do konfiguracji.
        elif len(self.D_de[v2]) > 0:
            # Przypadek gdy w druga delta ma sasiadow extremal nodes.
            edge5 = self.D_de[v2].pop()
            self.D_d[v2].remove(edge5)
            self.D_2[edge5.target].remove(~edge5)
            edge6 = self.D_2[edge5.target].pop()
            self.D_d[edge6.target].remove(~edge6)
            self.D_de[edge6.target].remove(~edge6)
            # Wyrzucamy krawedzie i tworzymy z nich konfiguracje.
            assert len(self.D_2[edge5.target]) == 0
            # Znalezione (~edge1, edge2, edge5, edge6).
            if edge6.target == v1:
                self.Q_A.append((~edge1, edge2, edge5, edge6))
            else:
                self.Q_B.append((~edge1, edge2, edge5, edge6))
        else:   # przywracam niewykorzystane edge1 i edge2
            self.D_2[node].add(edge1)
```

```
            self.D_2[node].add(edge2)
            self.D_d[v1].add(~edge1)
            self.D_d[v2].add(~edge2)
            self.D_de[v1].add(~edge1)
            self.D_de[v2].add(~edge2)

    def _coloring(self):
        """Edge coloring."""
        # Metoda rekurencyjna do przetwarzania kolejnych krawedzi.
        if len(self.Q_e) > 0:
            # Rekurencyjne kolorowanie reducible edges.
            # Do Q_e musza trafiac krawedzie z edge.source < edge.target.
            # Step 1.
            edge = self.Q_e.pop()
            # Czyszczenie D_d, D_2, D_de, extremal przed usunieciem krawedzi.
            if self.graph.degree(edge.source) == self.Delta: # bedzie potem deg De
                while len(self.D_d[edge.source]) > 0:
                    e = self.D_d[edge.source].pop()
                    self.D_2[e.target].remove(~e)
                    if e in self.D_de[edge.source]:
                        # e.target przestanie byc extremal.
                        self.extremal.remove(e.target)
                        self.D_de[edge.source].remove(e)
                        # get corresponding delta -> extremal through second deg2 -
                        e2 = self.D_2[e.target].pop()
                        assert len(self.D_2[e.target]) == 0  # przez moment
                        # remove corrseponding delta -> extremal
                        self.D_de[e2.target].remove(~e2)
                        # reinsert deg2 -> delta edge to queue
                        self.D_2[e.target].add(e2)
            elif self.graph.degree(edge.source) == 2: # bedzie potem deg 1
                while len(self.D_2[edge.source]) > 0 :
                    e = self.D_2[edge.source].pop()
                    self.D_d[e.target].remove(~e)
                    if ~e in self.D_de[e.target]:
                        self.D_de[e.target].remove(~e)
            if self.graph.degree(edge.target) == self.Delta: # bedzie potem deg De
                while len(self.D_d[edge.target]) > 0:
                    e = self.D_d[edge.target].pop()
                    self.D_2[e.target].remove(~e)
                    if e in self.D_de[edge.target]:
                        # e.target przestanie byc extremal.
                        self.extremal.remove(e.target)
                        self.D_de[edge.target].remove(e)
                        # get corresponding delta -> extremal through second deg2 -
                        e2 = self.D_2[e.target].pop()
                        assert len(self.D_2[e.target]) == 0  # przez moment
                        # remove corrseponding delta -> extremal
                        self.D_de[e2.target].remove(~e2)
                        # reinsert deg2 -> delta edge to queue
                        self.D_2[e.target].add(e2)
            elif self.graph.degree(edge.target) == 2: # bedzie potem deg 1
                while len(self.D_2[edge.target]) > 0:
                    e = self.D_2[edge.target].pop()
                    self.D_d[e.target].remove(~e)
                    if ~e in self.D_de[e.target]:
                        self.D_de[e.target].remove(~e)
```

```
        self.graph.del_edge(edge)
        # Wyrzucamy reducible edges
        # Step 2. Update Q_e, Q_A, and Q_B.
        for e in self.graph.iteroutedges(edge.source):
            if e.source > e.target:    # w Q_e sa tylko takie krawedzie
                e = ~e
            weight = self.graph.degree(e.source) + self.graph.degree(e.target)
            if weight == 13:    # krawedz nie mogla byc wczesniej w Q_e
                self.Q_e.add(e)
            elif (self.graph.degree(e.source) == 1 or
                    self.graph.degree(e.target) == 1):
                self.Q_e.add(e)
            elif (self.graph.degree(e.source) == 2 and
                    self.graph.degree(e.target) == self.Delta-1):
                self.Q_e.add(e)
            elif (self.graph.degree(e.target) == 2 and
                    self.graph.degree(e.source) == self.Delta-1):
                self.Q_e.add(e)
        for e in self.graph.iteroutedges(edge.target):
            if e.source > e.target:    # w Q_e sa tylko takie krawedzie
                e = ~e
            weight = self.graph.degree(e.source) + self.graph.degree(e.target)
            if weight == 13:    # krawedz nie mogla byc wczesniej w Q_e
                self.Q_e.add(e)
            elif (self.graph.degree(e.source) == 1 or
                    self.graph.degree(e.target) == 1):
                self.Q_e.add(e)
            elif (self.graph.degree(e.source) == 2 and
                    self.graph.degree(e.target) == self.Delta-1):
                self.Q_e.add(e)
            elif (self.graph.degree(e.target) == 2 and
                    self.graph.degree(e.source) == self.Delta-1):
                self.Q_e.add(e)
        #update Q_A and Q_B
        #Jesli w wyniku usuniecia krawedzi wierzcholek stal sie deg2
        # rozwazamy go
        if self.graph.degree(edge.source) == 2:
            # Sprawdzamy czy wierzcholek jest extremal,
            for e in self.graph.iteroutedges(edge.source): # tylko 2 obiegi pe
                if self.graph.degree(e.target) == self.Delta:
                    self.D_2[edge.source].add(e)
                    self.D_d[e.target].add(~e)
            if len(self.D_2[edge.source]) == 2:
                self.extremal.add(edge.source)
                for e in self.D_2[edge.source]:
                    self.D_de[e.target].add(~e)
                self._find_A_B(edge.source)
        if self.graph.degree(edge.target) == 2:
            # Sprawdzamy czy wierzcholek jest extremal,
            for e in self.graph.iteroutedges(edge.target): # tylko 2 obiegi pe
                if self.graph.degree(e.target) == self.Delta:
                    self.D_2[edge.target].add(e)
                    self.D_d[e.target].add(~e)
            if len(self.D_2[edge.target]) == 2:
                self.extremal.add(edge.target)
                for e in self.D_2[edge.target]:
                    self.D_de[e.target].add(~e)
```

```
                self._find_A_B(edge.target)
            # Step 3. The recursive call.
            self._coloring()
            # Step 4. The edge is reinserted to the graph and colored.
            self.graph.add_edge(edge)
            self._greedy_color(edge)
            return
        # Nastepnie powtarzamy to dla nastepnego elementu z kolejki reducible
        while len(self.Q_A) > 0:
            # Step 1. (ux, xv, vy, yu)
            edge1, edge2, edge3, edge4 = self.Q_A.pop()
            t = (self.graph.degree(edge1.source),
                 self.graph.degree(edge1.target),
                 self.graph.degree(edge2.target),
                 self.graph.degree(edge3.target))
            if t != (self.Delta, 2, self.Delta, 2):
                continue
            # Dalej konfiguracja A jest poprawna.
            # Mamy cykl uxvy ze stopniami Delta, 2, Delta, 2.
            # Usuniecie krawedzi edge2=xv, dalej rekurencyjnie.
            self.graph.del_edge(edge2)
            # Step 2. Update Q_e, Q_A, and Q_B.
            # x ma stopien 1
            # Tu nie trzeba modyfikowac D_2, D_d, D_de, extremal,
            # bo krawedzi z konfiguracji A tam nie ma.
            if edge1.source < edge1.target:
                self.Q_e.add(edge1)
            else:
                self.Q_e.add(~edge1)
            # v ma stopien Delta-1, y ma stopien 2
            if edge3.source < edge3.target:
                self.Q_e.add(edge3)
            else:
                self.Q_e.add(~edge3)
            # Step 3. The recursive call.
            self._coloring()
            # Step 4. The edge is reinserted to the graph and colored.
            self.graph.add_edge(edge2)
            # alpha to kolor edge1=ux.
            alpha = self._get_color(edge1)
            if alpha in self.missing[edge2.target]:
                # Swap colors edge1=ux and edge4=yu.
                beta = self._get_color(edge4)
                self._del_color(edge1, alpha)
                self._del_color(edge4, beta)
                self._add_color(edge1, beta)
                self._add_color(edge4, alpha)
                self._add_color(edge2, alpha)
            else:   # alpha jest zajety przy v
                beta = min(self.missing[edge2.target])
                self._add_color(edge2, beta)
            return
        # Konfiguracja B moze stac sie niepoprawna, trzeba to sprawdzic.
        while len(self.Q_B) > 0:
            # Step 1. (ux, xv, vy, yw)
            edge1, edge2, edge3, edge4 = self.Q_B.pop()
            t = (self.graph.degree(edge1.source),
```

```
            self.graph.degree(edge1.target),
            self.graph.degree(edge2.target),
            self.graph.degree(edge3.target),
            self.graph.degree(edge4.target))
        if t != (self.Delta, 2, self.Delta, 2, self.Delta):
            continue
        # Dalej konfiguracja B jest poprawna.
        uv = Edge(edge1.source, edge2.target)
        vw = Edge(edge3.source, edge4.target)
        if self.graph.has_edge(uv):
            # Case 2. Jest uv, vw nie wiadomo.
            # Usuniecie krawedzi edge3=vy.
            self.graph.del_edge(edge3)
            # Step 2. Update Q_e, Q_A, and Q_B.
            # v ma stopien Delta-1, x ma stopien 2
            if edge2.source < edge2.target:
                self.Q_e.add(edge2)
            else:
                self.Q_e.add(~edge2)
            # y ma stopien 1, w ma stopien Delta.
            # y juz nie jest extremal.
            self.extremal.remove(edge3.target)
            if edge4.source < edge4.target:
                self.Q_e.add(edge4)
            else:
                self.Q_e.add(~edge4)
            # Step 3. The recursive call.
            self._coloring()
            # Step 4.
            self.graph.add_edge(edge3)
            # alpha to kolor edge4=yw.
            # beta to kolor edge1=ux.
            # gamma to kolor edge2=xv.
            alpha = self._get_color(edge4)
            beta = self._get_color(edge1)
            gamma = self._get_color(edge2)
            if alpha in self.missing[edge2.target]:
                if beta != alpha:
                    self._del_color(edge2, gamma)
                    self._add_color(edge2, alpha)
                    self._add_color(edge3, gamma)
                else:   # beta == alpha
                    # Sukam krawedzi uv. O(Delta) time
                    for edge5 in self.graph.iteroutedges(edge1.source):
                        if edge5.target == edge2.target:
                            break
                    delta = self._get_color(edge5)
                    self._del_color(edge1, alpha)
                    self._del_color(edge5, delta)
                    self._add_color(edge1, delta)
                    self._add_color(edge5, alpha)
                    self._add_color(edge3, delta)
            else:   # alpha jest zajety przy v
                c = min(self.missing[edge2.target])
                self._add_color(edge3, c)
        elif self.graph.has_edge(vw):
            # Case 2. Nie ma uv, jest vw.
```

```
            self.graph.del_edge(edge2)
            # Step 2. Update Q_e, Q_A, and Q_B.
            # u ma stopien Delta, x ma stopien 1.
            # x juz nie jest extremal.
            self.extremal.remove(edge1.target)
            if edge1.source < edge1.target:
                self.Q_e.add(edge1)
            else:
                self.Q_e.add(~edge1)
            # y ma stopien 2, v ma stopien Delta-1
            if edge3.source < edge3.target:
                self.Q_e.add(edge3)
            else:
                self.Q_e.add(~edge3)
            # Step 3. The recursive call.
            self._coloring()
            # Step 4.
            self.graph.add_edge(edge2)
            # alpha to kolor edge1=ux.
            # beta to kolor edge4=yw.
            # gamma to kolor edge3=vy.
            alpha = self._get_color(edge1)
            beta = self._get_color(edge4)
            gamma = self._get_color(edge3)
            if alpha in self.missing[edge2.target]:
                if beta != alpha:
                    self._del_color(edge3, gamma)
                    self._add_color(edge3, alpha)
                    self._add_color(edge2, gamma)
                else:   # beta == alpha
                    # Szukam krawedzi vw. O(Delta) time
                    for edge6 in self.graph.iteroutedges(edge3.source):
                        if edge6.target == edge4.target:
                            break
                    delta = self._get_color(edge6)
                    self._del_color(edge4, alpha)
                    self._del_color(edge6, delta)
                    self._add_color(edge4, delta)
                    self._add_color(edge6, alpha)
                    self._add_color(edge2, delta)
            else:   # alpha jest zajety przy v
                c = min(self.missing[edge2.target])
                self._add_color(edge2, c)
        else:   # Case 1. Nie ma uv, nie ma vw.
            # x i y powinny zniknac z extremal.
            self.extremal.remove(edge1.target)
            self.extremal.remove(edge3.target)
            self.graph.del_edge(edge1)
            self.graph.del_edge(edge2)
            self.graph.del_edge(edge3)
            self.graph.del_edge(edge4)
            self.graph.add_edge(uv)
            self.graph.add_edge(vw)
            # Step 3. The recursive call.
            self._coloring()
            # Step 4.
            self.graph.del_edge(uv)
```

```python
                self.graph.del_edge(vw)
                self.graph.add_edge(edge1)
                self.graph.add_edge(edge2)
                self.graph.add_edge(edge3)
                self.graph.add_edge(edge4)
                # Szukam kolorow przydzielonych uv i vw.
                alpha = self._get_color(uv)
                beta = self._get_color(vw)
                self._del_color(uv, alpha)
                self._del_color(vw, beta)
                self._add_color(edge1, alpha)
                self._add_color(edge2, beta)
                self._add_color(edge3, alpha)
                self._add_color(edge4, beta)
                # Usuwanie wpisow ze sztucznymi krawedziami.
                if uv.source > uv.target:
                    uv = ~uv
                if vw.source > vw.target:
                    vw = ~vw
                del self.color[uv]
                del self.color[vw]
            return

    def _add_color(self, edge, c):
        """Add color."""
        if edge.source > edge.target:
            edge = ~edge
        self.color[edge] = c
        self.missing[edge.source].remove(c)
        self.missing[edge.target].remove(c)

    def _del_color(self, edge, c):
        """Delete color."""
        if edge.source > edge.target:
            edge = ~edge
        self.color[edge] = None
        self.missing[edge.source].add(c)
        self.missing[edge.target].add(c)

    def _get_color(self, edge):
        """Get color."""
        if edge.source > edge.target:
            edge = ~edge
        return self.color[edge]

    def _greedy_color(self, edge):
        """Greedy edge coloring."""
        both = self.missing[edge.source] & self.missing[edge.target]
        if len(both) == 0:
            raise ValueError("no color available")
        else:
            c = min(both)   # choose min color available
            self._add_color(edge, c)

    def show_colors(self):
        """Show edge coloring (undirected graphs)."""
        L = []
```

```
        for source in self.graph.iternodes():
            L.append("{} : ".format(source))
            for edge in self.graph.iteroutedges(source):
                # It should work for multigraphs.
                c = self._get_color(edge)
                L.append("{}({}) ".format(edge.target, c))
            L.append("\n")
        print("".join(L))

# EOF
```

5. Podsumowanie

Zagadnieniem omawianym w tej pracy było kolorowanie krawędzi grafów. Stworzono kilka nowych implementacji algorytmów kolorowania, oraz kilka generatorów grafów o specyficznych właściwościach. Poniżej krótko omówimy wyniki pracy.

Zaimplementowano pięć różnych algorytmów kolorowania krawędzi dla następujących grafów: grafu pełnego, grafu dwudzielnego pełnego i zwykłego, grafu regularnego dwudzielnego, grafu planarnego. Stworzono rownież generatory grafów planarnych z ograniczeniem na największy stopień wierzchołka. Pierwszy generator tworzy sieć Apoloniusza ze ścianami trójkątnymi, drugi tworzy graf dwudzielny ze ścianami kwadratowymi. Zagadnieniem na przyszłość może być szybsza implementacja algorytmu kolorowania krawędzi grafu regularnego dwudzielnego. Najtrudniejszym do implementacji okazał się algorytm kolorowania krawędzi dla grafu planarnego. Jego struktura jest rekurencyjna, a uzyskanie w praktyce liniowego czasu pracy zależy od dopracowania wielu szczegółów.

W pracy korzystano z pakietu graphtheory rozwijanego w Instytucie Fizyki UJ [2]. Pakiet potwierdził swoją użyteczność w implementacji algorytmów grafowych. Pakiet specjalnie na potrzeby tej pracy został dostosowany do trzeciej wersji języka Python i obecnie działa dla Pythona 2.7 i 3.2+.

W pracy również przygotowano testy sprawdzające poprawność zaimplementowanych algorytmów za pomocą modułu unittest. Ponadto sprawdzono praktyczną złożoność obliczeniową zaimplementowanych algorytmów przy pomocy modułu timeit.

A. Testy algorytmów

Dodatek zawiera wyniki testów wydajnościowych algorytmów zaimplementowanych w pracy.

A.1. Testy kolorowania krawędzi grafu pełnego

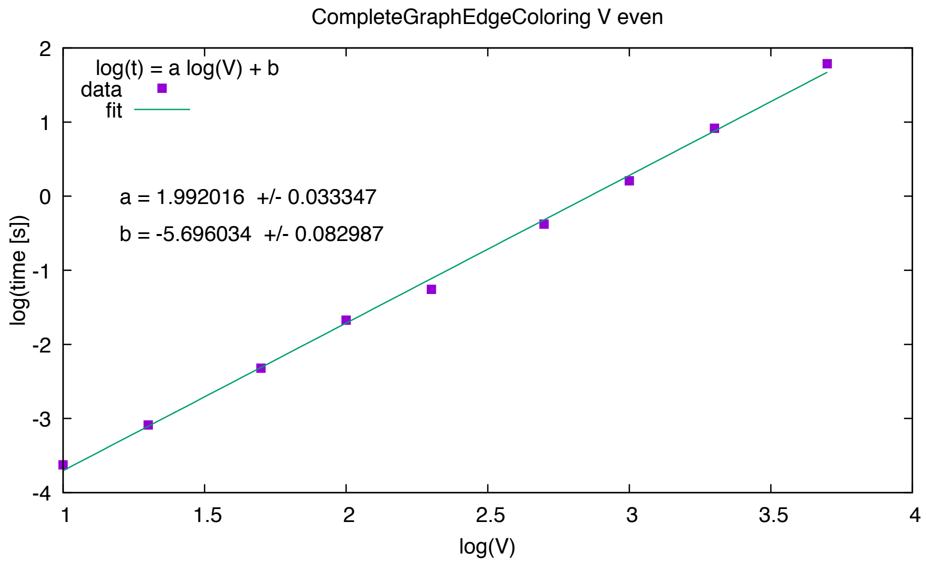
Testowanie algorytmu kolorowania krawędzi grafu pełnego, czyli klasy CompleteGraphEdgeColoring. Grafy pełne były generowane przy pomocy generatora make_complete() z modułu factory. Wyniki są przedstawione na rysunkach A.1 i A.2. Wykresy pokazują praktyczną złożoność implementacji rzędu $O(n^2)$.

A.2. Testy kolorowania krawędzi grafu dwudzielnego pełnego

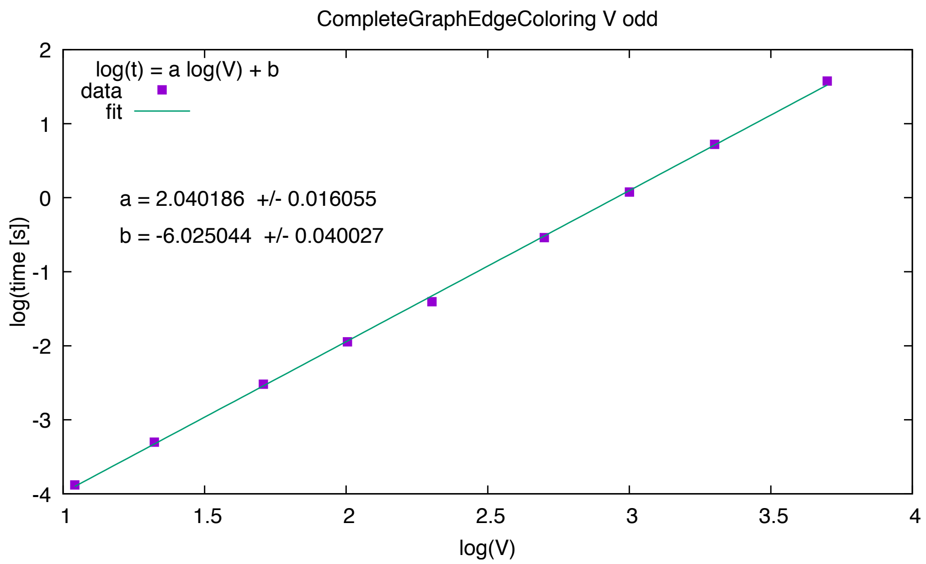
Testowanie algorytmu kolorowania krawędzi grafu dwudzielnego pełnego, czyli klasy CompleteBipartiteGraphEdgeColoring. Grafy te były generowane za pomocą generatora make_bipartite() z modułu factory ze zmienną edge_propability o wartości 1. Wyniki są przedstawione na rysunku A.3 Wykresy pokazują złożoność implementacji rzędu $O(n^2)$.

A.3. Testy kolorowania krawędzi grafu dwudzielnego ogólnego

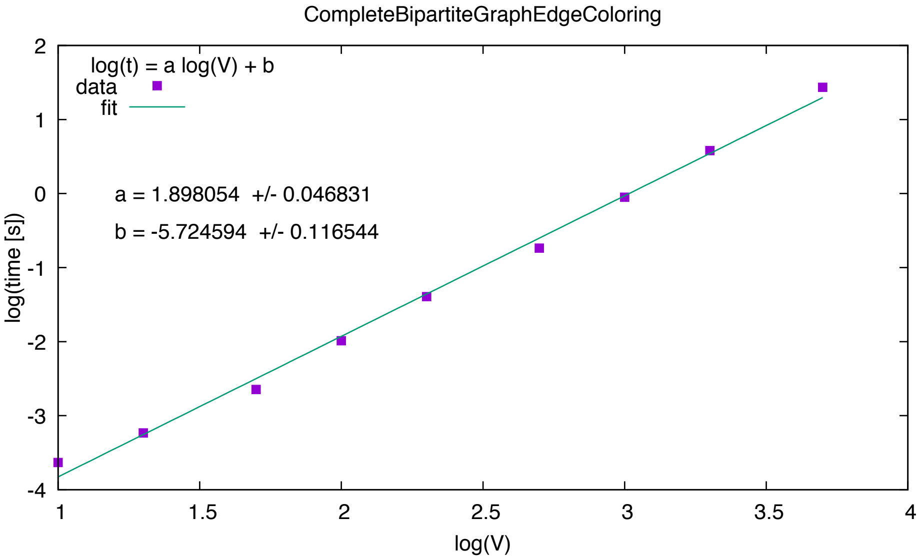
Testowanie algorytmu kolorowania krawedzi grafu dwudzielnego ogólnego, czyli klasy BipartiteGraphEdgeColoring. Grafy te były generowane za pomocą generatora make_bipartite() z modułu factory ze zmienną edge_propability o wartości 0.5 (grafy gęste), oraz za pomocą generatora make_grid() (grafy rzadkie). Wyniki są przedstawione na rysunkach A.4 (grafy gęste) i A.5 (grafy rzadkie) Dla grafów gęstych złożoność implementacji jest rzędu $O(mn) = O(n^3)$. Dla grafów rzadkich (grid) praktyczna złożoność implementacji okazała się być liniowa, ponieważ dla planarnego grafu kraty $m = O(n)$ oraz $\Delta = 4$ (największy stopień wierzchołka).



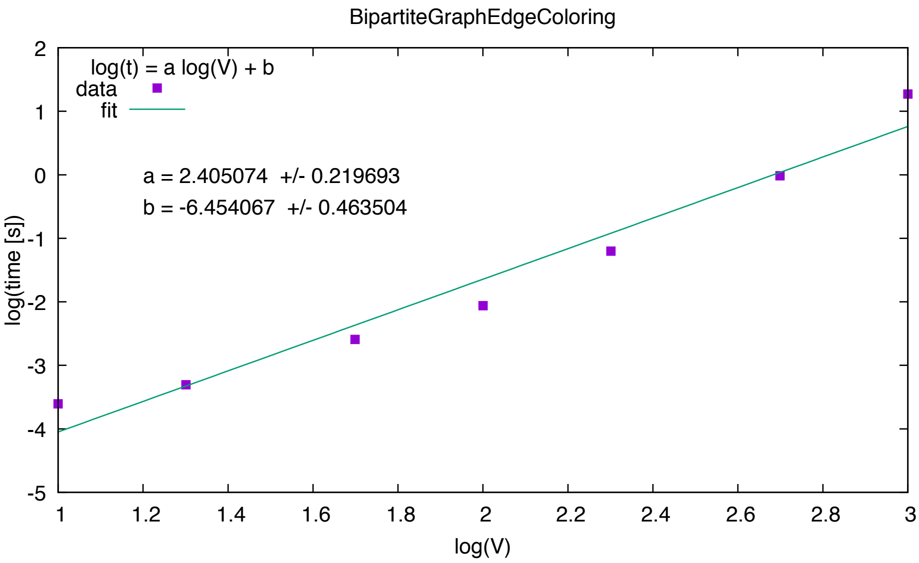
Rysunek A.1. Wykres wydajności algorytmu kolorowania krawędzi grafu pełnego dla n parzystego. Współczynnik a w przybliżeniu równy 2 potwierdza złożoność $O(n^2)$.



Rysunek A.2. Wykres wydajności algorytmu kolorowania krawędzi grafu pełnego dla n nieparzystego. Współczynnik a w przybliżeniu równy 2 potwierdza złożoność $O(n^2)$.



Rysunek A.3. Wykres wydajności algorytmu kolorowania krawędzi grafu dwudzielnego pełnego. Współczynnik a w przybliżeniu równy 2 potwierdza złożoność $O(n^2)$.



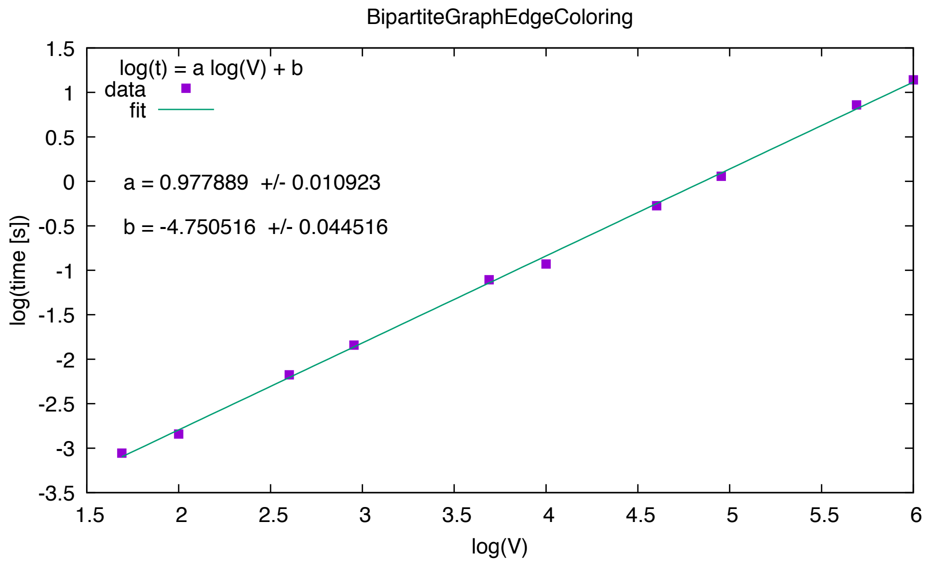
Rysunek A.4. Wykres wydajności algorytmu kolorowania krawędzi grafu dwudzielnego ogólnego. Współczynnik a pomiędzy 2 a 3 potwierdza złożoność $O(mn)$.

A.4. Testy kolorowania krawędzi grafu dwudzielnego regularnego

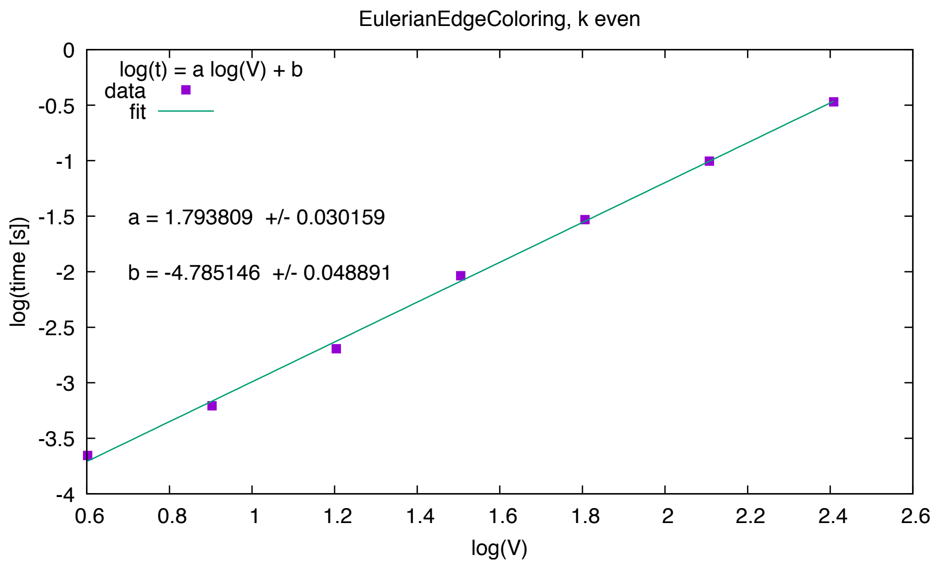
Testowanie algorytmu kolorowania krawędzi grafu dwudzielnego k-regularnego, czyli klasy EulerianEdgeColoring. Wykorzystano grafy dwudzielne pełne K_{pp}, które były generowane przy pomocy generatora make_bipartite() z modułu factory. Wyniki są przedstawione na rysunkach A.6 i A.7 Dla grafu K_{pp} mamy $n = 2p$, $k = p$, $m = p^2$, czyli $O(km) = O(n^3)$. Praktyczna złożoność obliczeniowa implementacji zależy od wartości k. Dla k nieparzystego dostajemy złożoność $O(n^3)$. Dla $k = 2^r$ parzystego unikamy wywoływania powolnej procedury znajdowania skojarzenia doskonałego i dostajemy złożoność liniową $O(n + m) = O(n^2)$.

A.5. Testy kolorowania krawędzi grafu planarnego

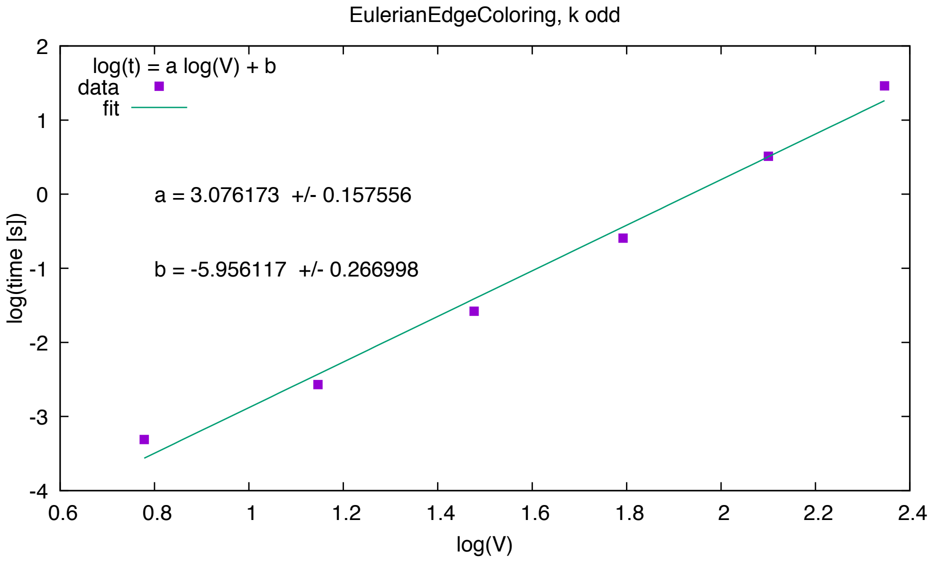
Testowanie algorytmu kolorowania krawędzi grafu planarnego o $\Delta = 12$ czyli klasy PlanarGraphEdgeColoring. Wykorzystano grafy planarne utworzone przy pomocy generatorów make_planar_square() i make_planar_delta() z modułu planartools. Wyniki są przedstawione na rysunkach A.8 i A.9. Dla grafu $\Delta = 12$ mamy złożoność $O(n)$.



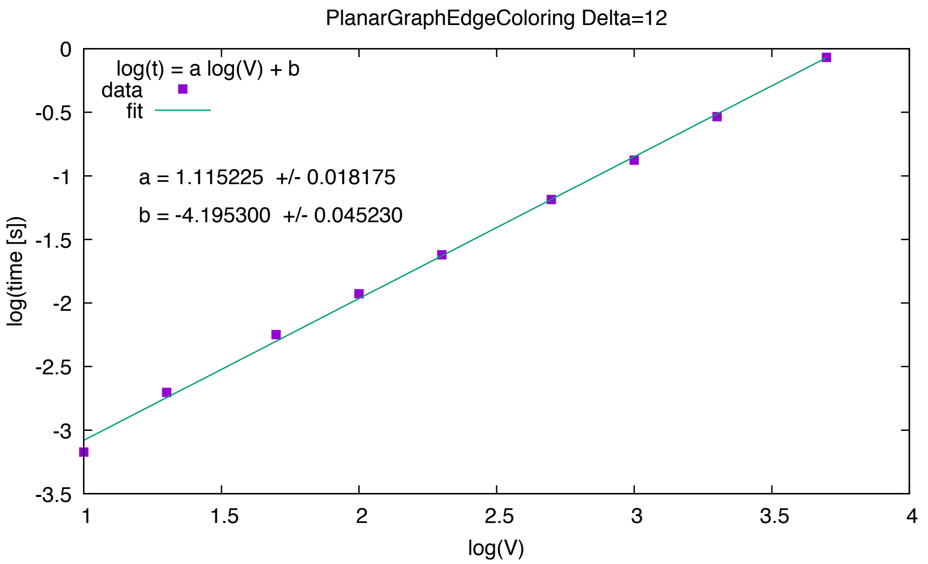
Rysunek A.5. Wykres wydajności algorytmu kolorowania krawędzi grafu dwudzielnego ogólnego rzadkiego (grid). Współczynnik a w przybliżeniu równy 1 potwierdza złożoność $O(n)$.



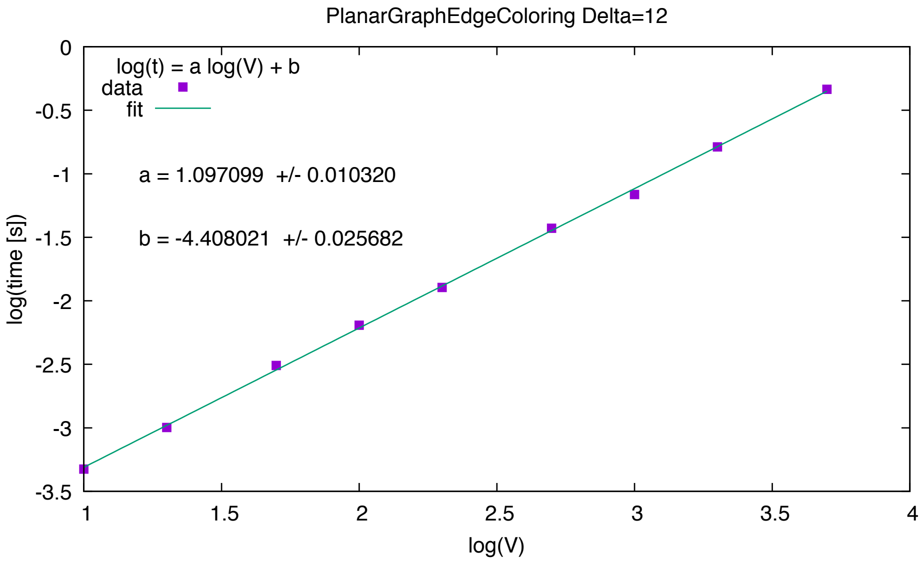
Rysunek A.6. Wykres wydajności algorytmu kolorowania krawędzi grafu k-regularnego o parzystej wartości k. Współczynnik a w przybliżeniu równy 2 daje złożoność $O(n^2)$.



Rysunek A.7. Wykres wydajności algorytmu kolorowania krawędzi grafu k-regularnego o nieparzystej wartości k. Współczynnik a w przybliżeniu równy 3 potwierdza złożoność $O(n^3)$.



Rysunek A.8. Wykres wydajności algorytmu kolorowania krawędzi grafu planarnego z $\Delta = 12$ wygenerowanego za pomocą funkcji make_planar_delta() (sieć Apoloniusza). Współczynnik a w przybliżeniu równy 1 potwierdza złożoność $O(n)$.



Rysunek A.9. Wykres wydajności algorytmu kolorowania krawędzi grafu planarnego z $\Delta = 12$ wygenerowanego za pomocą funkcji make_planar_square() (graf ze ścianami kwadratowymi). Współczynnik a w przybliżeniu równy 1 potwierdza złożoność $O(n)$.

B. Różnice między Pythonem 2 a Pythonem 3

Na potrzeby tej pracy biblioteka grafowa rozwijana w Instytucie Fizyki UJ została przystosowana do pracy z drugą i trzecią wersję języka Python. Między tymi wersjami występują pewne subtelne różnice w strukturze pisanych programów. Dla wygody czytelnika zostaną one przedstawione w tym dodatku.

Najważniejsze informacje konieczne do tworzenia kodu w taki sposób, aby był kompatybilny z obydwoma wersjami Python znajdziemy w oficjalnej dokumentacji języka Python [20]. Najważniejszą informacją jest to, najlepiej jest skorzystać z Pythona 2.7, gdyż jest on dalej wspierany i jest możliwość przeportowania kodu na Python 3. Starsze wersje Pythona 2 nie są już wspierane. Zauważmy, że na ogół nie wykrywamy jawnie wersji Pythona (ang. *version detection*), tylko sprawdzamy obecność pewnych obiektów lub atrybutów (ang. *feature detection*). Niezbędne jest także pokrycie testami jednostkowymi jak największej części kodu źródłowego (powyżej 80%).

Przydatnym modułem dla uzyskania kompatybilności obydwu wersji jest moduł `__future__`. Jest on warstwą kompatybliną między Pythonem 2, a Pythonem 3. Pozwala on na użycie czystej podstawy kodu z Pythona 3 wspieranej zarówno w Pythonie 2 jak i Pythonie 3. Przy jego użyciu należy jednak uważać by nie przeoczyć jakiegoś importu.

W kwestii kompatybilności należy pamiętać o głównych zmianach pomiędzy dwoma wersjami Pythona. Jedną z istotniejszych jest użycie `print`. W Pythonie 2 `print` jest instrukcją, zaś w Pythonie 3 jest funkcją. Chcąc zachować kompatybilność nie możemy napisać `print(item1, item2)`, gdyż wypisana zostanie krotka. Musimy użyć zapisu `print( "one {} two {}".format(item1, item2) )`. Zalecane jest użycie importu `from __future__ import print_function`.

Kolejną istotną rzeczą jest kwestia dzielenia liczb całkowitych. Dla takich najlepiej jest używać instrukcji `a // b`. Jeżeli chcemy uzyskać wynik `float`, to aby nie zmylić użytkownika dobrze jest napisać np. `3 / 2.0` lub `float(x) / 2`. Przy dzieleniu należy pamiętać o imporcie `from __future__ import division`. We własnych klasach należy również pamiętać o metodach `__div__`, `__floordiv__`, `__truediv__`.

Kolejną różnicą są zmiany w typach liczbowych. W Pythonie 2 występują typy `int` oraz `long`, zaś w Python 3 zostały one ograniczone jedynie do `int`. Sprawdzanie typów liczbowych można zrobić przez komendę `try`/`except` widoczną w kodach algorytmów:

```
try:
    integer_types = (int, long)
except NameError:    # Python 3
    integer_types = (int,)
```

```
# Zastosowanie: isinstance(variable, integer_types)
```

W Pythonie 2 mamy funkcje **str**(), **unicode**() i **bytearray**(). W Pythonie 3 mamy funkcję **str**() dla Unicode, zaś dla bajtów mamy **bytearray**() oraz byte(). W naszej bibliotece stosujemy czyste ASCII i str() lub repr() do napisów.

W Pythonie 2 jest iterator **xrange**(), a funkcja **range**() tworzy listę. W Pythonie 3 nie ma **xrange**(), a **range**() jest iteratorem. Listę można tworzyć uniwersalnie przez **list**(**range**(n)), jednak z iteratorem jest problem. Istnieje moduł six, który w Pythonie 3 rzekomo posiada **xrange**(), ale nie ma go w bibliotece standardowej. Nasze podejście obchodzące ten problem polega na podstawieniu **xrange** = **range** w Pythonie 3.

W Pythonie 3 nie ma wbudowanej funkcji **cmp**(). Obejście proponowane w oficjalnej dokumentacji Pythona ma postać **cmp** = **lambda** x, y: (x > y) − (x < y). W Pythonie 3 istnieje metoda __contains__ dla obiektów **range**. W Pythonie 2 nie ma takiej funkcji. W pracy przyjmujemy, że wyszukiwanie w liście jest wolne rzędu $O(n)$ i zwykle korzystamy ze zbioru lub słownika.

Istnieje różnica w zgłaszaniu wyjątków. Python 2 dopuszcza starą składnię z przecinkiem: **raise** IOError, "file error". W naszej pracy zawsze stosujemy nową składnię z nawiasem (jak konstruktor): **raise** IOError("file error").

Taka sama sytuacja zachodzi w przechwytywaniu wyjątków. Python poniżej 2.6 dopuszcza starą składnię z przecinkiem. W naszej pracy stosujemy nową składnię ze słowem kluczowym as.

Funkcja **next**() jest w Python 3, a w Pythonie 2 odpowiada jej metoda .**next**(). Jednakże w Pythonie 2.6 pojawiła się wbudowana funkcja **next**(), więc kod może już być uniwersalny.

Istnieje problem ze zmiennymi globalnymi w pętli **for**, tzn. zmienna użyta w pętli żyje poza nią. Należy używać lepszych nazw zmiennych, aby nazwy globalne nie mieszały się z lokalnymi.

W Pythonie 3 została naprawiona możliwość porównywania różnych typów np. [1,2] > "a" zwracające False. W Pythonie 3 rzucany jest wyjątek TypeError.

Pojawiły się różnice w funkcji **input**(). W Pythonie 3 **input**() odpowiada **raw_input**() z Pythona 2 i zwraca **str**(). W Pythonie 2 **input**() było niebezpieczne (i powolne), bo próbowało wykonywać kod. Najlepiej jest zawsze używać **input**() po skorzystaniu z **try**/**except**:

```
try:
    input = raw_input
except NameError:    # jestesmy w Pythonie 3
    pass
```

W Pythonie 3 wiele funkcji zwraca obiekty iterowalne zamiast list. Zmiany zaszły w funkcjach: **zip**(), **map**(), **filter**(), D.keys(), D.values(), D.items(). W Pythonie 3 słowniki nie mają metody D.has_key(k), więc lepiej zawsze używać operatora **in**, na przykład k **in** D. Obiekty iterowalne możemy łatwo zamienić na listę, np. kod **list**(**map**(...)) zawsze stworzy listę. Możliwe jest obejście problemu przez **try**/**except**:

```
try:
    values = D.itervalues()
```

```
except AttributeError:    # jestesmy w Pythonie 3
    values = D.values()
```

Moduł Queue z Pythona 2 nazywa się queue w Pythonie 3. Używamy do tego detekcji importu zamiast detekcji wersji Pythona:

```
try:
    from Queue import Queue    # kolejka
except ImportError:    # Python 3
    from queue import Queue
```

W Pythonie2 **exec** jest instrukcją, zaś w Pythonie 3 jest funkcją.

Istnieje różnica w definiowaniu metaklas. W Pythonie 2 jest atrybut __metaclass__ w ciele klasy, zaś w Pythonie 3 jest parametr metaclass przekazywany w definicji klasy.

Sortowanie w Pythonie pojawia się w metodzie list L.sort() i w funkcji **sorted**(). W Pythonie 2 są dwa parametry sortowania **cmp** i key, zaś w Pythonie 3 jest tylko key. Powinniśmy więc używać tylko parametru key.

Funkcja wbudowana **reduce**() z Pythona 2 znajduje się w module functools w Pythonie 3. Jednak w Pythonie 2 funkcja też jest w tym module, co sugeruje rozwiązanie uniwersalne: **from** functools **import reduce**.

Bibliografia

[1] Wikipedia, Edge coloring, 2019,
https://en.wikipedia.org/wiki/Edge_coloring.

[2] Andrzej Kapanowski, graphs-dict, GitHub repository, 2019,
https://github.com/ufkapano/graphs-dict/.

[3] Python Programming Language - Official Website,
https://www.python.org/.

[4] Paweł Motyl, *Implementacja wybranych algorytmów dla multigrafów w języku Python*, Praca magisterska, Uniwersytet Jagielloński, Kraków, 2015.

[5] Igor Samson, *Kolorowanie grafów z językiem Python*, Praca magisterska, Uniwersytet Jagielloński, Kraków, 2016.

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

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

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

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

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

[11] Wikipedia, Complete graph, 2019,
https://en.wikipedia.org/wiki/Complete_graph.

[12] Wikipedia, Bipartite graph, 2019,
https://en.wikipedia.org/wiki/Bipartite_graph.

[13] Wikipedia, Regular graph, 2019,
https://en.wikipedia.org/wiki/Regular_graph.

[14] Eric W. Weisstein, Eulerian Graph, From MathWorld–A Wolfram Web Resource, 2019,
http://mathworld.wolfram.com/EulerianGraph.html.

[15] Wikipedia, Planar graph, 2019,
https://en.wikipedia.org/wiki/Planar_graph.

[16] Alexander Schrijver, *Bipartite Edge Coloring in $O(\Delta m)$ Time*, SIAM Journal on Computing 28, 841-846 (1998).

[17] H. N. Gabow, *Using Euler partitions to edge color bipartite multigraphs*, International Journal of Computer and Information Sciences 5, 345-355 (1976).

[18] Denes Konig, *1-factorization of regular bipartite graphs*, Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre, Mathematische Annalen 77 (4), 453-465 (1916).

[19] Richard Cole, Łukasz Kowalik, *New Linear-Time Algorithms for Edge-Coloring Planar Graphs*, Algorithmica 50, 351-368 (2008).

[20] Brett Cannon, Porting Python 2 code to Python 3, 2019,
https://docs.python.org/3/howto/pyporting.html.