Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Igor Samson

Nr albumu: 1063308

Kolorowanie grafów z językiem Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2016

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 magisterskiej, za nieocenioną życzliwość, cenne uwagi merytoryczne, wszechstronną pomoc oraz powięcony czas, dzięki którym niniejsza praca powstała w tym kształcie i formie.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów kolorowania grafów i wyznaczania pokrycia wierzchołkowego. Założono klasyczny model kolorowania, ale wspomniano o innych modelach. Zebrano wyniki teoretyczne dla grafów prostych, grafów planarnych i grafów dwudzielnych. Przygotowano testy poprawności i testy złożoności obliczeniowej, korzystające ze standardowych modułów Pythona (unittest, timeit). Stworzono także generator grafów dwudzielnych przypadkowych.

Zaimplementaowano szereg algorytmów kolorowania wierzchołków grafu: algorytm dokładny dla problemu m-kolorowania (algorytm z powrotami), algorytm bazujący na dowodzie twierdzenia Brooksa, dwa algorytmy zbiorów niezależnych (GIS, RLF). Powstały nowe implementacje algorytmów sekwencyjnych z wykorzystaniem saturacji (US, RS, LF, SL, CS).

Pokazano kilka algorytmów kolorowania krawędzi: trzy algorytmy sekwencyjne (US, RS, algorytm bazujący na BFS) i algorytm NTL. Dzięki mechanizmom zabezpieczającym języka Python wykryto braki w typowych opisach procedury przekolorowania krawędzi z algorytmu NTL.

W końcu zaimplementowano dwa algorytmy znajdujące najmniejsze pokrycie wierzchołkowe: algorytm 2-aproksymacyjny i algorytm zachłanny wykorzystujący stopnie wierzchołków.

Słowa kluczowe: grafy, multigrafy, kolorowanie wierzchołków, kolorowanie krawędzi, pokrycie wierzchołkowe, zbiory niezależne

English title: Graph coloring with Python

Abstract

Python implementation of selected graph coloring algorithms and vertex cover algorithms is presented. The classical coloring model is assumed, but some other models are mentioned. Known theoretical results for simple graphs, planar graphs, and bipartite graphs are collected. Tests for correctness and computational complexity are provided, were standard Python modules are used (unittest, timeit). A generator of random bipartite graphs is added.

The algorithms for a proper vertex coloring are presented: the exact algorithm for an m-coloring using backtracking, the algorithm based on the Brooks' theorem, two independent sets algorithms (GIS, RLF). New implementation (using saturation) for several sequential algorithms is added (US, RS, LF, SL, CS).

The algorithms for a proper edge coloring are shown: three sequential algorithms (US, RS, the algorithm using BFS) and the NTL algorithm. It was found that the recolor procedure from the NTL algorithm is often not properly described in the literature.

Finally, two algorithms for finding a minimum vertex cover are implemented: a 2-approximation algorithm and a greedy algorithm (using degrees of vertices).

Keywords: graphs, multigraphs, vertex coloring, edge coloring, vertex cover, independent sets

Spis treści

Spis tabel . . . 4
Spis rysunków . . . 5
Listings . . . 6
1. Wstęp . . . 7
1.1. Cel pracy . . . 7
1.2. Organizacja pracy . . . 7
2. Teoria grafów . . . 9
2.1. Grafy skierowane i nieskierowane . . . 9
2.2. Ścieżki i cykle . . . 10
2.3. Spójność . . . 10
2.4. Stopień grafu . . . 10
2.5. Wybrane rodziny grafów . . . 11
3. Implementacja grafów . . . 12
3.1. Struktury danych z biblioteki grafów . . . 13
4. Kolorowanie wierzchołków . . . 14
4.1. Ograniczenia na liczbę chromatyczną . . . 14
4.2. Kolorowanie wierzchołków grafu planarnego . . . 16
4.3. Zbiory niezależne . . . 16
4.3.1. Poprawność wyznaczenia zbioru niezależnego . . . 16
4.4. Pokrycie wierzchołkowe . . . 17
4.4.1. Poprawność wyznaczenia pokrycia wierzchołkowego . . . 17
4.4.2. Algorytm 2-aproksymacyjny dla pokrycia wierzchołkowego 17
4.4.3. Algorytm zachłanny dla pokrycia wierzchołkowego . . . 18
4.5. Poprawność kolorowania wierzchołków . . . 19
4.6. Algorytm dokładny kolorowania wierzchołków . . . 19
4.7. Algorytm z powrotami dla kolorowania wierzchołków . . . 20
4.8. Kolorowanie zachłanne wierzchołków . . . 21
4.9. Algorytm kolorowania z twierdzenia Brooksa . . . 22
4.10. Algorytmy sekwencyjne kolorowania wierzchołków . . . 24
4.11. Algorytmy z wymianą kolorów . . . 26
4.12. Algorytmy zbiorów niezależnych . . . 26
4.12.1. Algorytm GIS kolorowania wierzchołków . . . 26
4.12.2. Algorytm RLF kolorowania wierzchołków . . . 27
5. Kolorowanie krawędzi . . . 29
5.1. Kolorowanie krawędzi grafu dwudzielnego . . . 30
5.2. Kolorowanie krawędzi grafu planarnego . . . 30
5.3. Skojarzenia . . . 30
5.4. Pokrycie krawędziowe . . . 31
5.5. Poprawność kolorowania krawędzi . . . 32
5.6. Kolorowanie zachłanne jednej krawędzi . . . 32

5.7. Kolorowanie krawędzi z grafem krawędziowym 34
5.8. Algorytm US kolorowania krawędzi 35
5.9. Algorytm RS (NC) kolorowania krawędzi 36
5.10. Kolorowanie krawędzi z BFS 37
5.11. Algorytm NTL kolorowania krawędzi 39
5.12. Algorytm Misry i Griesa kolorowania krawędzi 45
6. Inne modele kolorowania grafów 46
6.1. Sprawiedliwe kolorowanie grafów 46
6.2. Totalne kolorowanie grafów 46
6.3. Kolorowanie grafów w trybie on-line 46
7. Podsumowanie 47
A. Algorytmy i struktury danych z biblioteki grafów 48
B. Testy pokrycia wierzchołkowego 50
C. Testy kolorowania wierzchołków 52
D. Testy kolorowania krawędzi 56
Bibliografia 59

Spis tabel

A.1 Algorytmy i struktury danych z biblioteki grafów IF UJ. 49
A.2 Nowe moduły dodane do biblioteki grafów. 49

C.1 Wyniki kolorowania grafów przypadkowych z $n = 1000$. 53
C.2 Wyniki kolorowania grafów przypadkowych z $p = 0.5$. 53

Spis rysunków

5.1 Graf planarny klasy 2 z $\Delta = 3$. 31
5.2 Graf planarny klasy 2 z $\Delta = 4$. 32
5.3 Graf planarny kubiczny klasy 2. 33
5.4 Graf Petersena z zaznaczonym skojarzeniem doskonałym. 34

B.1 Wydajność algorytmu aproksymacyjnego pokrycia wierzchołkowego. . 51
B.2 Wydajność algorytmu zachłanego pokrycia wierzchołkowego. 51

C.1 Wydajność algorytmu z twierdzenia Brooksa (grafy prawie pełne). . . 54
C.2 Wydajność algorytmu z twierdzenia Brooksa (grafy Halina). 54
C.3 Wydajność algorytmu GIS kolorowania wierzchołków. 55
C.4 Wydajność algorytmu RLF kolorowania wierzchołków. 55

D.1 Wydajność algorytmu US kolorowania krawędzi. 57
D.2 Wydajność algorytmu RS kolorowania krawędzi. 57
D.3 Wydajność algorytmu BFS kolorowania krawędzi. 58
D.4 Wydajność algorytmu NTL kolorowania krawędzi. 58

Listings

3.1 Korzystanie z kolorowania wierzchołków GIS. 12
3.2 Korzystanie z kolorowania krawędzi RS. 12
4.1 Test poprawności zbioru niezależnego. 16
4.2 Test poprawności pokrycia wierzchołkowego. 17
4.3 Moduł nodecoverapp. 18
4.4 Moduł nodecoverdeg. 18
4.5 Test poprawności kolorowania wierzchołków. 19
4.6 Moduł nodecolorbt. 20
4.7 Moduł nodecolorbrooks. 22
4.8 Moduł nodecolorsl. 25
4.9 Moduł nodecolorgis. 27
4.10 Moduł nodecolorrlf. 28
5.1 Test poprawności kolorowania krawędzi. 32
5.2 Kolorowanie zachłanne jednej krawędzi. 32
5.3 Kolorowanie jednej krawędzi z saturacją. 33
5.4 Moduł edgecolorus. 35
5.5 Moduł edgecolorrs. 36
5.6 Moduł edgecolorbfs. 38
5.7 Moduł edgecolorntl. 40

1. Wstęp

Kolorowanie grafów jest działem teorii grafów, która z kolei zaliczana jest do dziedziny pod nazwą optymalizacja dyskretna [1]. Istnieje wiele modeli kolorowania grafów, czyli przypisywania kolorów wierzchołkom, krawędziom, ścianom grafu planarnego. W niniejszej pracy zajmujemy się klasycznym kolorowaniem wierzchołków i klasycznym kolorowaniem krawędzi grafu. Chodzi o przyporządkowanie elementom grafu kolorów w taki sposób, aby sąsiadujące elementy otrzymały różne kolory. Modele nieklasyczne wprowadzają zwykle dodatkowe ograniczenia na używane kolory.

Kolorowanie grafów ma duże znaczenie praktyczne. Tak modeluje się zadania podziału zbioru, zwierającego wewnętrzne konflikty między elementami, na podzbiory bezkonfliktowe [1]. W ogólności są to zadania trudne, czyli nie są znane efektywne algorytmy rozwiązujące je w czasie wielomianowym. Stąd rozważane są różne szybkie algorytmy przybliżone lub też szuka się szczególnych rodzin grafów, dla których można znaleźć algorytmy dokładne, działające w czasie wielomianowym.

1.1. Cel pracy

Celem pracy jest przygotowanie implementacji w języku Python klasycznych algorytmów kolorowania grafów. Mimo istnienia wielu modeli kolorowania grafów, dla wielu zastosowań podstawowe znaczenie ma model klasyczny. W nauczaniu teorii grafów również naturalne jest rozpoczęcie tematu od modelu klasycznego. Język Python umożliwia przygotowanie działającego kodu, który z drugiej strony może być analizowany tak jak pseudokod w podręcznikach algorytmiki.

Kolorowanie grafów pojawia się w wielu książkach z teorii grafów: [1], [2], [3], [4], [5]. Często są to jednak dowody matematyczne, czasem z pseudokodami, natomiast trudno znaleźć wskazówki implementacyjne. Z pewnością sposób implementacji częściowo zależy od użytego języka programowania, ale pewne pomysły mogą być bardziej uniwersalne. W każdym razie możliwość sprawdzenia i uruchomienia poprawnej implementacji na pewno pomaga w lepszym zrozumieniu danego algorytmu.

1.2. Organizacja pracy

Praca została podzielona na rozdziały, stopniowo rozwijające tematykę kolorowania grafów. Rozdział 1 jest wprowadzeniem do niniejszej pracy. Rozdział 2 w usystematyzowany sposób przedstawia podstawowe pojęcia z teorii grafów potrzebne do opisu kolorowania grafów. Rodział 3 prezentuje imple-

mentację grafów ze strukturami danych używanymi w bibliotece grafowej i w prezentowanych algorytmach. Rodział 4 przedstawia algorytmy kolorowania wierzchołków, wyznaczania pokrycia wierzchołkowego, oraz problem zbiorów niezależnych. Rozdział 5 zawiera algorytmy kolorowania krawędzi oraz zagadnienia dotyczące skojarzenia i pokrycia krawędziowego. Rozdział 6 omawia inne modele kolorowania grafów. Rodział 7 zawiera podsumowanie pracy. W dodatku A zestawiono stare i nowe moduły używane do kolorowania grafów. W dodatkach B, C, D zebrano wyniki testów odpowiednio dla pokrycia wierzchołkowego, kolorowania wierzchołków i kolorowania krawędzi.

2. Teoria grafów

Teoria grafów jest działem matematyki zajmującym się badaniem własności grafów, które są strukturami matematycznymi używanymi do modelowania relacji pomiędzy obiektami. Teoria grafów jest istotnym narzędziem matematycznym wykorzystywanym w wielu innych dziedzinach nauki, takich jak informatyka, genetyka, socjologia, lingwistyka, czy badania operacyjne. Jedną z największych zadań informatyki teoretycznej jest rozwój algorytmów wyznaczających pewne właściwości grafów.

Opublikowana w 1741 roku praca autorstwa Leonarda Eulera pod tytułem *Solutio problematis ad geometriam situs pertinentis* w czasopiśmie *Commentarii academiae scientiarum Petropolitanae*, poruszająca problem mostów królewieckich, uznawana jest za pierwszą pracą na temat teorii grafów.

2.1. Grafy skierowane i nieskierowane

Graf $G = (V, E)$ definiowany jest jako uporządkowana para składająca się ze zbioru wierzchołków V oraz ze zbioru krawędzi E łączących wierzchołki. Wierzchołki grafu mogą być etykietowane i czasem stanowią reprezentację jakichś obiektów, natomiast krawędzie mogą wówczas obrazować relacje między takimi obiektami. Wierzchołki należące do krawędzi nazywane są jej końcami. Krawędzie mogą mieć wyznaczony kierunek, a graf zawierający takie krawędzie nazywany jest grafem skierowanym. Krawędź grafu może posiadać wagę, to znaczy przypisaną liczbę, która może określać odległość między wierzchołkami, koszt, czas przejazdu, itp. W grafie skierowanym wagi mogą być zależne od kierunku przechodzenia pomiędzy wierzchołkami.

Definicja: *Graf skierowany (prosty)* to taka uporządkowana para $G = (V, E)$, w której V to niepusty zbiór wierzchołków, a E to zbiór krawędzi skierowanych

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

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

Definicja: *Graf nieskierowany (prosty)* to taka uporządkowana para $G = (V, E)$, w której V to niepusty zbiór wierzchołków, a E to zbiór krawędzi nieskierowanych

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

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

Definicja: *Multigraf skierowany* to taka uporządkowana para $G = (V, E)$, w której V to niepusty zbiór wierzchołków, a E to wielozbiór krawędzi skierowanych. W E elementy mogą się powtarzać (krawędzie równoległe), oraz mogą występować pętle typu (v, v).

Definicja: *Multigraf nieskierowany* to taka uporządkowana para $G = (V, E)$, w której V to niepusty zbiór wierzchołków, a E to wielozbiór krawędzi nieskierowanych. Krawędzie nieskierowane to są wielozbiory dwuelementowe, a pętle mają postać $\{v, v\}$.

2.2. Ścieżki i cykle

Definicja: *Ścieżka* pomiędzy wierzchołkami v_0 oraz v_k w grafie $G = (V, E)$ to ciąg wierzchołków $(v_0, v_1, \ldots, v_k)$ taki, że dla każdego $j \in \{0, 1, \ldots, k-1\}$ istnieje krawędź z v_j do v_{j+1}, a wierzchołki w ścieżce mogą się powtarzać. Długość k takiej ścieżki to liczba przeskoków pomiędzy wierzchołkami. *Ścieżka prosta* jest to ścieżka, w której wierzchołki nie powtarzają się.

Definicja: *Cykl* to taka ścieżka, w której początkowy i końcowy wierzchołek ścieżki są takie same, $v_0 = v_k$. *Cykl prosty* jest to cykl, w którym wierzchołki nie mogą się powtarzać, za wyjątkiem pierwszego i ostatniego wierzchołka. W literaturze za cykl prosty uznaje się każdą pętlę oraz dwie nieskierowane krawędzie równoległe. Graf niezawierający cykli prostych nazywany jest grafem acyklicznym.

2.3. Spójność

Definicja: Graf nieskierowany nazywamy *spójnym* (ang. *connected*), jeśli pomiędzy każdą parą wierzchołków tego grafu istnieje łącząca je nieskierowana ścieżka.

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

2.4. Stopień grafu

Definicja: *Stopień wierzchołka* to liczba krawędzi incydentnych do wierzchołka. Stopień wierzchołka równy jest sumie wszystkich krawędzi wchodzą-

cych, wychodzących oraz pętli, które liczone są jak dwie krawędzie. Stopień wierzchołka v oznacza się poprzez $\deg(v)$.

Definicja: *Stopień grafu* $\Delta(G)$ to maksymalny stopień wierzchołka w grafie. *Graf regularny stopnia* r to graf, w którym wszystkie wierzchołki mają stopień r.

$$\Delta(G) = \max\{\deg(v) : v \in V(G)\}. \tag{2.3}$$

2.5. Wybrane rodziny grafów

Definicja: *Graf pełny* to graf nieskierowany prosty, w którym dla każdej pary wierzchołków istnieje krawędź łącząca te dwa wierzchołki. Graf pełny o n wierzchołkach oznacza się jako K_n.

Definicja: *Graf cykliczny* to graf nieskierowany spójny, w którym każdy wierzchołek jest stopnia drugiego, to znaczy że liczba krawędzi wychodzących od tego wierzchołka równa się 2. Graf cykliczny o n wierzchołkach oznacza się jako C_n.

Definicja: *Graf regularny stopnia* n taki graf, w którym wszystkie wierzchołki grafu są stopnia n, czyli z każdego danego grafu wychodzi dokładnie n krawędzi. Graf regularny stopnia n określa się także *grafem n-regularnym*. Szczególnym przypadkiem grafów regularnych są *grafy kubiczne*, inaczej nazywane grafami 3-regularnymi.

Definicja: *Graf dwudzielny* to graf, którego zbiór wierzchołków można podzielić na dwa rozłączne zbiory w ten sposób, że krawędzie nie łączą wierzchołków należących do tego samego zbioru. Czasem graf dwudzielny definiuje się jako trójkę $G = (U, V, E)$, gdzie U i V to dwa niepuste i rozłączne zbiory wierzchołków, a zbiór krawędzi $E \subseteq U \times V$.

Definicja: *Graf planarny* to graf, który może zostać narysowany na płaszczyźnie tak, by krzywe obrazujące krawędzie grafu nie przecinały się ze sobą. Odwzorowanie grafu planarnego na płaszczyźnie nazywa się *rysunkiem płaskim grafu*. Graf planarny o zbiorze wierzchołków i krawędzi zdefiniowanym poprzez rysunek płaski nazywamy *grafem płaskim*.

3. Implementacja grafów

W celu skorzystania z zaimplementowanych algorytmów kolorowania wierzchołków lub krawędzi grafów, należy uruchomić sesję interaktywną interpretera Python, a następnie przejść do lokalizacji w której znajdują się pliki z kodem źródłowym. Przed uruchomieniem algorytmu należy zaimportować z modułu edges klasę Edge, z modułu graphs klasę Graph, oraz z modułu factory obiekt GraphFactory, a następnie z modułu algorytmu zaimportować obiekt algorytmu. Kolejnym krokiem będzie stworzenie fabryki grafów, za pomocą której w kolejnym kroku wygenerować można wybrany typ grafu, a następnie uruchomić na stworzonym grafie algorytm. Wynikowe kolorowanie można sprawdzić odczytując słownik algorithm.color.

Oto sesja interaktywna, zawierająca przykładowe użycie algorytmu kolorowania wierzchołków GIS.

Listing 3.1. Korzystanie z kolorowania wierzchołków GIS.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from nodecolorgis import GISNodeColoring
>>> graph_factory = GraphFactory(Graph)
>>> V = 10
>>> G = graph_factory.make_random(V, False, 0.5)
>>> algorithm = GISNodeColoring(G)
>>> algorithm.run()
>>> algorithm.color
{0: 0, 1: 2, 2: 1, 3: 3, 4: 0, 5: 0, 6: 4, 7: 1, 8: 1, 9: 0}
```

Następna sesja interaktywna zawiera przykładowe użycie algorytmu kolorowania krawędzi RS.

Listing 3.2. Korzystanie z kolorowania krawędzi RS.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from edgecolorrs import RandomSequentialEdgeColoring
>>> graph_factory = GraphFactory(Graph)
>>> V = 5
>>> G = graph_factory.make_random(V, False, 0.5)
>>> algorithm = RandomSequentialEdgeColoring(G)
>>> algorithm.run()
>>> algorithm.color
{Edge(3, 4, 7): 3, Edge(1, 3, 9): 4, Edge(0, 4, 5): 2,
Edge(0, 1, 10): 0, Edge(1, 4, 2): 1, Edge(0, 3, 4): 1,
Edge(2, 4): 0, Edge(1, 2, 3): 2}
```

3.1. Struktury danych z biblioteki grafów

Dla wygody użytkownika przedstawimy zestawienie struktur danych wykorzystywanych w naszej bibliotece grafowej. W kilku przypadkach występuje kilka możliwych struktur danych dla danego zagadnienia, ponieważ testowano różne implementacje.

Wierzchołek: Obiekt hashowalny, najczęściej liczba lub string.

Krawędź: Instancja klasy Edge, która odpowiada krawędzi skierowanej. Graf nieskierowany przechowuje wewnętrznie krawędź nieskierowaną jako dwie krawędzie skierowane z przeciwnymi kierunkami.

Graf: Instancja klasy Graph.

Multigraf: Instancja klasy MultiGraph.

Algorytm: Klasa z typowymi metodami __init__ do inicjalizacji danych i run do właściwej pracy. Wyniki działania algorytmu są zapisane w odpowiednich atrybutach klasy.

Drzewo rozpinające: Instancja klasy Graph lub słownik parent z parami (node, node) lub (node, None) dla korzenia.

Klika: Zbiór wierzchołków clique lub słownik z parami (node, bool).

Zbiór niezależny: Zbiór wierzchołków independent_set lub słownik z parami (node, bool).

Pokrycie wierzchołkowe: Zbiór wierzchołków node_cover lub słownik z parami (node, bool).

Pokrycie krawędziowe: Zbiór krawędzi edge_cover, ale edge.source $<$ edge.target.

Skojarzenie: Zbiór krawędzi lub słownik mate z parami (node, node) lub (node, None) dla wierzchołka bez pary. W przypadku grafów ważonych słownik mate może zawierać pary (node, edge) lub (node, None) dla wierzchołka bez pary, gdzie krawędź z wagą jest skierowana do drugiego wierzchołka tworzącego parę.

Kolorowanie wierzchołków: Słownik color z parami (node, int) lub (node, None) przy braku koloru. Kolory są numerowane od 0 w górę.

Kolorowanie krawędzi: Słownik color z parami (edge, int) lub (edge, None) przy braku koloru, ale edge.source $<$ edge.target. Kolory są numerowane od 0 w górę. Dla multigrafów krawędzie muszą być unikalne, tzn. przy krawędziach równoległych muszą być różnice w atrybucie edge.weight.

4. Kolorowanie wierzchołków

Kolorowanie wierzchołków multigrafu (ang. *vertex coloring*) polega na przyporządkowaniu wierzchołkom kolorów tak, że każda krawędź łączy wierzchołki o różnych kolorach [8]. Jest to tzw. kolorowanie właściwe lub dozwolone (ang. *proper coloring*), ale na etapach pośrednich algorytmów kolorowania może pojawić się też kolorowanie niewłaściwe. Kolorowanie niewłaściwe pojawia się także w dowodach twierdzeń o kolorowaniach.

W problemie kolorowania wierzchołków rozpatruje się multigrafy spójne, nieskierowane i bez pętli. Krawędzie wielokrotne mogą być pominięte bez straty ogólności. Jeżeli dla danego multigrafu istnieje dozwolone kolorowanie wierzchołków zawierające k kolorów, to mówimy że multigraf jest *k-kolorowalny wierzchołkowo*. Kolorowanie wierzchołków jest *optymalne*, jeżeli zawiera najmniejszą możliwą liczbę kolorów. Ta najmniejsza liczba kolorów to *liczba chromatyczna* (ang. *chromatic number*), a oznaczana jest przez $\chi(G)$ lub $\chi_0(G)$. Problem znalezienia kolorowania optymalnego jest NP-trudny. Problem decyzyjny polegający na określeniu, czy dany graf jest k-kolorowalny wierzchołkowo, jest NP-zupełny dla $k \geqslant 3$ [9]. Grafy 1-kolorowalne to grafy bez krawędzi. Grafy 2-kolorowalne to grafy dwudzielne, które można wykryć w czasie liniowym $O(V+E)$. Optymalne kolorowanie wierzchołków grafu jest podziałem jego wierzchołków na minimalną liczbę zbiorów niezależnych.

Stwierdzenie: Jeżeli H jest podgrafem grafu G, to $\chi(H) \leqslant \chi(G)$.

Stwierdzenie: Jeżeli graf G jest niespójny, to

$$\chi(G) = \max\{\chi(C),\ C \text{ składowa spójna } G\}. \tag{4.1}$$

4.1. Ograniczenia na liczbę chromatyczną

— Jeżeli każdemu wierzchołkowi zostanie przydzielony inny kolor, to otrzymamy poprawne, ale na ogół nieoptymalne, kolorowanie wierzchołków. Stąd najprostsze ograniczenie ma postać

$$1 \leqslant \chi(G) \leqslant n, \tag{4.2}$$

gdzie $G = (V, E)$, $n = |V|$, $m = |E|$. Dla grafu pełnego $\chi(K_n) = n$.

— Jeżeli graf G został optymalnie pokolorowany, to musi istnieć co najmniej jedna krawędź pomiędzy zbiorami wierzchołków pomalowanych na dwa różne kolory [2], czyli

$$\chi(G)(\chi(G) - 1) \leqslant 2m. \tag{4.3}$$

— Jeżeli graf zawiera klikę, czyli podgraf będący grafem pełnym, to rozmiar tej kliki jest dolnym ograniczeniem na liczbę chromatyczną. Dokładniej, liczba chromatyczna nie może być mniejsza od rozmiaru największej kliki w grafie. To ograniczenie jest ścisłe dla grafów doskonałych, do których należą np. grafy przedziałowe. Z drugiej strony, istnieją grafy Mycielskiego, dla których największe kliki mają rozmiar 2, a liczba chromatyczna może być dowolnie duża [10].
— Dla grafów dwudzielnych $\chi(G) = 2$.
— Przy wykorzystaniu algorytmu zachłannego można pokać, że

$$\chi(G) \leqslant \Delta(G) + 1, \tag{4.4}$$

gdzie $\Delta(G)$ jest największym stopniem wierzchołka w grafie G. Ograniczenie $\Delta(G) + 1$ jest najgorszą liczbą kolorów, jaką można otrzymać z kolorowania zachłannego. Co ciekawe, istnieje takie uporządkowanie wierzchołków, że kolorowanie zachłanne przydzieli optymalną liczbę kolorów. Ale znalezienie takiego uporządkowania wierzchołków pośród $n!$ permutacji jest problemem NP-trudnym.
— Graf ma *degenerację* d, jeżeli każdy jego podgraf ma wierzchołek stopnia co najwyżej d. Uporządkowanie degeneracji wierzchołków jest to takie uporządkowanie, w którym każdy wierzchołek ma co najwyżej d sąsiadów w zbiorze wcześniejszych wierzchołków. Przy wykorzystaniu algorytmu zachłannego z uporządkowaniem degeneracji otrzymujemy ograniczenie

$$\chi(G) \leqslant d + 1. \tag{4.5}$$

W ten sposób działa algorytm SL (ang. *smallest last*).

Twierdzenie (Brooks, 1941): Dla grafu prostego nieskierowanego spójnego G, który nie jest grafem pełnym i nie jest cyklem nieparzystym, zachodzi

$$\chi(G) \leqslant \Delta(G). \tag{4.6}$$

Jeżeli graf G jest grafem pełnym lub jest cyklem nieparzystym, to liczba chromatyczna wynosi $\chi(G) = \Delta(G)+1$ [11], [12]. Uproszczony dowód twierdzenia Brooksa podał Lovasz [13].

Jeżeli graf G zawiera wierzchołek v stopnia mniejszego niż $\Delta(G)$, to algorytm zachłanny powinien kolorować wierzchołki dalsze, a następnie bliższe v, a wtedy zostanie użyte najwyżej $\Delta(G)$ kolorów.

Jeżeli graf nie jest dwuspójny (ang. *biconnected*), to każda składowa dwuspójna może być pokolorowana osobno, a następnie należy złączyć pokolorowania.

Najtrudniejszy jest przypadek grafu dwuspójnego Δ-regularnego z $\Delta \geqslant 3$. Należy znaleźć drzewo rozpinające o korzeniu v, gdzie wierzchołki u i w są sąsiadami v, ale nie ma krawędzi między u i w. Takie wierzchołki u i w zawsze można znaleźć, ponieważ w przeciwnym wypadku graf byłby grafem pełnym. Wierzchołki u i w otrzymują najmniejszy kolor. Pozostałe wierzchołki kolorujemy zachłannie od najbardziej oddalonych od v, przez co zawsze będzie wolny jeden z Δ kolorów. Na końcu dojdziemy do wierzchołka v, ale tu też będzie wolny jeden z Δ kolorów, ponieważ u i w mają wspólny kolor.

Warto zauważyć, że w literaturze jest wiele niezależnych dowodów twierdzenia Brooksa [14], [15]. Jednak nie wszystkie są konstruktywne, np. dowody nie wprost, a wtedy nie są zbyt pomocne w implementacji.

4.2. Kolorowanie wierzchołków grafu planarnego

Dla grafów planarnych udowodniono szereg mocnych twierdzeń.

Twierdzenie: Jeżeli G jest grafem prostym planarnym, to $\chi(G) \leqslant 6$. W dowodzie indukcyjnym korzysta się z faktu, że każdy graf planarny prosty ma wierzchołek stopnia co najwyżej 5.

Twierdzenie (Heawood, 1890): Jeżeli G jest grafem prostym planarnym, to $\chi(G) \leqslant 5$.

Twierdzenie (Appel, Haken, 1976): Jeżeli G jest grafem prostym planarnym, to $\chi(G) \leqslant 4$. Uproszczony dowód tego twierdzenia można znaleźć w pracy [16].

Problem stwierdzenia, czy graf planarny G jest 3-kolorowalny wierzchołkowo, jest NP-zupełny dla $\Delta(G) \geqslant 4$. Jeżeli $\Delta(G) = 3$ i graf G jest różny od grafu pełnego K_4, to G jest 3-kolorowalny wierzchołkowo z twierdzenia Brooksa.

4.3. Zbiory niezależne

Zbiór niezależny (ang. *independent set*) grafu nieskierowanego $G = (V, E)$ jest to podzbiór S zbioru wierzchołków grafu V, taki że żadne dwa wierzchołki z S nie są połączone krawędzią z E [17]. Maksymalny zbiór niezależny (ang. *maximal independent set*) nie jest podzbiorem większego zbioru niezależnego. Największy zbiór niezależny (ang. *maximum independent set*) jest zbiorem niezależnym o największej liczności w grafie G. Problem znalezienia największego zbioru niezależnego jest NP-trudny.

Twierdzenie: Zbiór S jest zbiorem niezależnym wtedy i tylko wtedy, gdy jego dopełnienie $V \backslash S$ jest pokryciem wierzchołkowym. Każda krawędź może być styczna do najwyżej jednego wierzchołka ze zbioru S. Stąd każda krawędź jest styczna do co najmniej jednego wierzchołka ze zbioru $V \backslash S$.

4.3.1. Poprawność wyznaczenia zbioru niezależnego

Listing przedstawia metodę klasy wywiedzionej z innej klasy unittest.TestCase, która testuje algorytm zawarty w fikcyjnej klasie IndependentSet.

Listing 4.1. Test poprawności zbioru niezależnego.

```
def test_independent_set(self):
    algorithm = IndependentSet(self.G)
    algorithm.run()
    for edge in self.G.iteredges():
```

```
        self.assertFalse(edge.source in algorithm.independent_set
                         and edge.target in algorithm.independent_set)
```

4.4. Pokrycie wierzchołkowe

Pokrycie wierzchołkowe (ang. *vertex cover*) grafu nieskierowanego $G = (V, E)$ jest to podzbiór C zbioru wierzchołków grafu V, taki że każda krawędź z E ma jako koniec jakiś wierzchołek z C [18]. Przykładem trywialnego pokrycia wierzchołkowego jest cały zbiór V.

Problem znalezienia najmniejszego pokrycia wierzchołkowego jest klasycznym problemem optymalizacyjnym NP-trudnym. W wersji decyzyjnej problem jest NP-zupełny i polega na stwierdzeniu, czy w danym grafie istnieje pokrycie wierzchołkowe o danej liczbie wierzchołków k.

4.4.1. Poprawność wyznaczenia pokrycia wierzchołkowego

Listing przedstawia metodę klasy wywiedzionej z innej klasy unittest.TestCase, która testuje algorytm zawarty w fikcyjnej klasie NodeCover.

Listing 4.2. Test poprawności pokrycia wierzchołkowego.

```
def test_node_cover(self):
    algorithm = NodeCover(self.G)
    algorithm.run()
    for edge in self.G.iteredges():
        self.assertTrue(edge.source in algorithm.node_cover
                        or edge.target in algorithm.node_cover)
```

4.4.2. Algorytm 2-aproksymacyjny dla pokrycia wierzchołkowego

Istnieje elegancki algorytm 2-aproksymacyjny dla problemu pokrycia wierzchołkowego [5].

Dane wejściowe: Graf prosty nieskierowany $G = (V, E)$.

Problem: Wyznaczenie pokrycia wierzchołkowego, którego rozmiar nie przekracza rozmiaru pokrycia optymalnego więcej niż dwukrotnie.

Opis algorytmu: Algorytm rozpoczynamy od pustego pokrycia wierzchołkowego C. Następnie rozważamy po kolei wszystkie krawędzie z E. Jeżeli oba końce krawędzi e nie należą do C, to te oba końce dodajemy do C. W przeciwnym razie krawędź e odrzucamy.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V+E)$ (dla reprezentacji list sąsiedztwa), ponieważ mamy pętlę po krawędziach, a w nich dodawanie $O(V)$ wierzchołków do pokrycia C. Złożoność pamięciowa wynosi $O(V)$, ponieważ zapamiętujemy wierzchołki należące do pokrycia.

Uwagi: Podamy uzasadnienie, że jest to algorytm 2-aproksymacyjny [5]. Zbiór C jest pokryciem wierzchołkowym, ponieważ każda krawędź z E zostanie pokryta przez jakiś wierzchołek z C. Niech A oznacza zbiór krawędzi,

których wierzchołki weszły do C. Krawędzie w A nie mają wspólnych końców, więc $|C| = 2|A|$. Pokrycie optymalne C^* musi pokrywać wszystkie krawędzie z A, zatem $|C^*| \geqslant |A|$. Łącznie dostajemy oszacowanie $|C^*| \leqslant |C| \leqslant 2|C^*|$.

Listing 4.3. Moduł nodecoverapp.

```
#!/usr/bin/python

class NodeCoverSet:
    """Find a minimum node cover (approximation)."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.node_cover = set()    # pokrycie wierzcholkowe
        self.cardinality = 0

    def run(self):
        """Executable pseudocode."""
        for edge in self.graph.iteredges():
            if (edge.source in self.node_cover or
                edge.target in self.node_cover):
                continue
            else:
                self.node_cover.add(edge.source)
                self.node_cover.add(edge.target)
                self.cardinality += 2
```

4.4.3. Algorytm zachłanny dla pokrycia wierzchołkowego

Dane wejściowe: Graf prosty nieskierowany $G = (V, E)$.

Problem: Wyznaczenie pokrycia wierzchołkowego, którego może nie być optymalne.

Opis algorytmu: Algorytm rozpoczynamy od pustego pokrycia wierzchołkowego C. Następnie rozważamy po kolei wszystkie krawędzie z E. Jeżeli oba końce krawędzi e nie należą do C, to do C dodajemy koniec o większym stopniu, ponieważ w ten sposób możemy pokryć większą liczbę krawędzi. W przeciwnym razie krawędź e odrzucamy.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V+E)$ (dla reprezentacji list sąsiedztwa), ponieważ mamy pętlę po krawędziach, a w nich dodawanie $O(V)$ wierzchołków do pokrycia C. Złożoność pamięciowa wynosi $O(V)$, ponieważ zapamiętujemy wierzchołki należące do pokrycia.

Listing 4.4. Moduł nodecoverdeg.

```
#!/usr/bin/python

class DegreeNodeCoverSet:
```

```
    """Find a minimum node cover."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.node_cover = set()    # pokrycie wierzcholkowe
        self.cardinality = 0

    def run(self):
        """Executable pseudocode."""
        for edge in self.graph.iteredges():
            if (edge.source in self.node_cover or
                edge.target in self.node_cover):
                    continue
            else:
                if (self.graph.degree(edge.source) >
                    self.graph.degree(edge.target)):
                        self.node_cover.add(edge.source)
                else:
                    self.node_cover.add(edge.target)
                self.cardinality += 1
```

4.5. Poprawność kolorowania wierzchołków

Istnieje technika tworzenia programowania o nazwie *test-driven development (TDD)*, w ramach której najpierw programista tworzy automatyczny test sprawdzający dodawaną funkcjonalność, a potem implementuje funkcjonalność. Postępując w tym duchu podamy test, jaki powinna spelniać każda metoda kolorowania wierzchołków. Listing przedstawia metodę klasy wywiedzionej z innej klasy unittest.TestCase, która testuje algorytm zawarty w fikcyjnej klasie NodeColoring. W teście porównywane są kolory wierzchołków na końcach każdej krawędzi.

Listing 4.5. Test poprawności kolorowania wierzchołków.

```
def test_node_coloring(self):
    algorithm = NodeColoring(self.G)
    algorithm.run()
    for node in self.G.iternodes():
        self.assertNotEqual(algorithm.color[node], None)
    for edge in self.G.iteredges():
        self.assertNotEqual(algorithm.color[edge.source],
                            algorithm.color[edge.target])
```

4.6. Algorytm dokładny kolorowania wierzchołków

W naszej bibliotece jest już obecny dokłady algorytm kolorowania wierzchołków (moduł nodecolorexact). Polega on na sukcesywnym sprawdzaniu wszystkich kombinacji dwóch, trzech i większej liczby kolorów, aż do uzyskania po-

prawnego kolorowania. Algorytm ma złożoność $O(2^n)$ i jest stosowany tylko do kolorowania małych grafów (kilkanaście wierzchołków).

4.7. Algorytm z powrotami dla kolorowania wierzchołków

Algorytmy z powrotami to algorytmy rekurencyjne, które systematycznie przeszukują drzewo potencjalnych rozwiązań. W porównaniu z algorytmami siłowymi są bardziej inteligentne, ponieważ gałęzie nie rokujące nadziei na znalezienie rozwiązania są porzucane.

W odniesieniu do kolorowania wierzchołków określa się problem m-kolorowania (ang. *m-coloring problem*), czyli problem znalezienia poprawnego kolorowania wierzchołków grafu przy użyciu co najwyżej m kolorów. Czasem należy znaleźć wszystkie możliwe m-kolorowania, a czasem wystarczy nam jedno przykładowe kolorowanie.

Dane wejściowe: Graf prosty nieskierowany G, liczba m dostępnych kolorów.

Problem: Kolorowanie wierzchołków grafu G przy użyciu co najwyżej m kolorów. Znaleźć przykładowe rozwiązanie lub zasygnalizować brak rozwiązań.

Opis algorytmu: Na początku tworzona jest lista wierzchołków grafu G o nazwie node_list, przy czym kolejność wierzchołkow na liście jest dowolna. Sercem algorytmu jest procedura rekurencyjna _graph_color(k), której argumentem jest indeks kolejnego wierzchołka na liście node_list. Pierwsze uruchomienie procedury _graph_color(k) jest dla $k = 0$. W każdym kroku sprawdzane jest m kolorów dla wierzchołka k. Maksymalna głębokość rekurencji to $k = n - 1$. Rozwiązania częściowe są zapisywane, a w przypadku dojścia do ślepej uliczki są usuwane. Do sprawdzenia poprawności częściowego rozwiązania używana jest funkcja _is_safe().

Złożoność: Pesymistyczna złożoność czasowa algorytmu wynosi $O(nm^n)$, gdzie $n = |V|$, m jest liczbą dostępnych kolorów. Czynnik $O(n)$ pochodzi od funkcji _is_safe(). Algorytm ma praktyczne znaczenie dla małych grafów lub małej liczby dostępnych kolorów.

Uwagi: Jeżeli istnieje rozwiązanie problemu, to na pewno zostanie znalezione. Przedstawiona implementacja znajduje jedno przykładowe m-kolorowanie lub wyzwala wyjątek, kiedy rozwiązanie nie istnieje.

Listing 4.6. Moduł nodecolorbt.

```
#!/usr/bin/python
#
# Based on the description from
# http://www.geeksforgeeks.org/backttracking-set-5-m-coloring-problem/
```

```
class BacktrackingNodeColoring:
    """m-coloring problem with backtracking."""

    def __init__(self, graph, m_colors):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.node_list = list(self.graph.iternodes())
        # Colors from 0 to m_colors-1.
        self.m_colors = m_colors
        self.color = dict((node, None) for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v()*2, recursionlimit))

    def run(self):
        """Executable pseudocode."""
        if not self._graph_color(0):
            raise ValueError("solution does not exist")

    def _is_safe(self, node, c):
        """Check if the current color is safe for the node."""
        # Iterate trough adjacent vertices and check if the vertex
        # color is different from c.
        return all(self.color[target] != c
            for target in self.graph.iteradjacent(node))

    def _graph_color(self, k):
        """Solve m-coloring problem."""
        # Check if all vertices are assigned a color.
        if k == self.graph.v():
            return True
        node = self.node_list[k]
        # Trying differents c color for the vertex node.
        for c in xrange(self.m_colors):
            # Check if assignment of color c to node is possible.
            if self._is_safe(node, c):
                # Assign color c to node.
                self.color[node] = c
                # Recursively assign colors to the rest of the vertices.
                if self._graph_color(k+1):
                    return True
                # If there is no solution, remove color (BACKTRACK).
                self.color[node] = None
        return False
```

4.8. Kolorowanie zachłanne wierzchołków

Kolorowanie zachłanne (ang. *greedy coloring*) jest to kolorowanie wierzchołków przy pomocy algorytmu zachłannego, który przegląda wierzchołki grafu w pewnej kolejności, oraz przydziela im pierwszy dostępny kolor [19].

W ogólnym przypadku kolorowanie zachłanne nie przydziela najmniejszej możliwej liczby kolorów. Z drugiej strony, kolorowanie zachłanne jest podstawą wielu algorytmów przybliżonych. Kolorowanie zachłanne jednego wierzchołka realizuje metoda _greedy_color, która będzie pokazana przy algorytmie SL.

Graf korona (ang. *crown graph*) [20] jest przykładem bardzo złego kolorowania metodą zachłanną. Graf korona powstaje z grafu pełnego dwudzielnego $K_{r,r}$ przez usunięcie skojarzenia doskonałego. Graf korona jest grafem dwudzielnym z $V = V_1 \cup V_2$, $|V_1| = |V_2| = r$ i przy podawaniu do algorytmu zachłannego wierzchołków najpierw z $|V_1|$, a potem z $|V_2|$, wykorzystane zostaną dwa kolory. Jeżeli jednak do algorytmu będą podawane wierzchołki naprzemiennie z $|V_1|$ i $|V_2|$, stanowiące parę z usuniętego skojarzenia, to liczba wykorzystanych kolorów wyniesie r.

Wierzchołki każdego grafu można uporządkować tak, aby algorytm zachłanny stworzył kolorowanie optymalne. Wystarczy dla danego kolorowania optymalnego wybrać wierzchołki jednego koloru, tworzącego zbiór największy. Jako drugi zbiór wierzchołków dla drugiego koloru, wybieramy zbiór największy ze względu na pierwszy kolor, itd.

Niestety znalezienie uporządkowania wierzchołków, przy którym algorytm zachłanny stworzy kolorowanie optymalne, jest w ogólności problemem NP-trudnym. Dlatego algorytmy heurystyczne często postulują pewne uporządkowanie, w nadziei na uzyskanie kolorowania bliskiego optymalnemu.

4.9. Algorytm kolorowania z twierdzenia Brooksa

Dane wejściowe: Graf prosty nieskierowany spójny G.

Problem: Kolorowanie wierzchołków grafu G przy użyciu $\Delta(G)$ kolorów.

Opis algorytmu: Podstawą jest dowód twierdzenia Brooksa. Algorytm działa dla grafu spójnego, który ma wierzchołek stopnia mniejszego niż $\Delta(G)$. Jeżeli graf ma wszystkie wierzchołki stopnia Δ (graf Δ-regularny), to algorytm działa dla grafów 3-spójnych. Dla innych grafów Δ-regularnych algorytm zadziała, jeżeli uda się znaleźć wierzchołek v, którego dwaj sąsiedzi u i w nie są połączeni krawędzią, a przy tym usunięcie z grafu wierzchołków u i w nie rozdzieli grafu na osobne składowe. W naszej implementacji nie ma gwarancji znalezienia wierzchołków o takich własnościach.

Złożoność: Złożoność czasowa algorytmu wynosi co najwyżej $O(V^2)$, dla grafów bliskich grafowi pełnemu.

Uwagi: Ze względu na użycie algorytmu BFS grafy dwudzielne będą optymalnie pokolorowane dwoma kolorami.

Listing 4.7. Moduł nodecolorbrooks.

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

```
import itertools
from Queue import Queue
from edges import Edge
from connected import is_connected


class BrooksNodeColoring:
    """Find a node coloring based on Brooks' theorem."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        if not is_connected(graph):
            raise ValueError("the graph is not connected")
        self.graph = graph
        self.color = dict((node, None) for node in self.graph.iternodes())
        self.order = list()   # kolejnosc kolorowania wierzcholkow
        self.parent = dict()   # for BFS
        # Sprawdzam czy nie ma petli, a przy okazji licze krawedzie.
        self.n = self.graph.v()
        self.m = 0
        for edge in self.graph.iteredges():
            self.m += 1
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        if 2 * self.m == self.n * (self.n - 1):
            raise ValueError("complete graph detected")

    def run(self):
        """Executable pseudocode."""
        Delta = max(self.graph.degree(node) for node in self.graph.iternodes())
        if Delta < 3:
            raise ValueError("trivial case with Delta less then three")
        source = min(self.graph.iternodes(), key=self.graph.degree)
        if self.graph.degree(source) < Delta:
            # Wystarczy zrobic BFS zaczynajac od source.
            # Tutaj wystarczy, gdy graf jest spojny.
            # Ustalam kolejnosc kolorowania wierzcholkow za pomoca BFS.
            self._visit(source)   # BFS in O(V+E) time
        else:
            # Delta-regular 3-connected graph.
            # Wybieram wierzcholek o najwiekszym stopniu, O(V) time.
            source = max(self.graph.iternodes(), key=self.graph.degree)
            for nodes in itertools.combinations(
                self.graph.iteradjacent(source), 2):   # O(Delta**2) time
                    neighbor1, neighbor2 = nodes
                    if not self.graph.has_edge(Edge(neighbor1, neighbor2)):
                        break
            # Mark two neighbors of source as visited.
            self.parent[neighbor1] = source
            self.parent[neighbor2] = source
            self._visit(source)   # modified BFS in O(V+E) time
            self.order.append(neighbor1)   # second colored, color 0
            self.order.append(neighbor2)   # first colored, color 0
        # Mozemy kolorowac zachlannie wg znalezionej kolejnosci.
```

```
        self.order.reverse()    # kolejnosc odwrotna! O(V) time
        for source in self.order:
            self._greedy_color(source)

    def _visit(self, node):
        """Explore the connected component with BFS."""
        Q = Queue()
        self.parent[node] = None    # before Q.put
        Q.put(node)
        self.order.append(node)    # pre_action
        while not Q.empty():
            source = Q.get()
            for target in self.graph.iteradjacent(source):
                if target not in self.parent:
                    self.parent[target] = source    # before Q.put
                    Q.put(target)
                    self.order.append(target)    # pre_action

    def _greedy_color(self, source):
        """Give node the smallest possible color."""
        n = self.graph.v()    # memory O(V)
        used = [False] * n    # is color used?
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for c in xrange(n):    # check colors
            if not used[c]:
                self.color[source] = c
                break
        return c
```

4.10. Algorytmy sekwencyjne kolorowania wierzchołków

Algorytmy sekwencyjne kolorowania wierzchołków bazują na kolorowaniu zachłannym, a wierzchołki są przeglądane w kolejności specyficznej dla konkretnego wariantu algorytmu. Rozważaliśmy następujące warianty:

— Metoda US (ang. *Unordered Sequential*), gdzie kolejność wierzchołków wynika z implementacji grafu. Złożoność obliczeniowa wynosi $O(V + E)$.
— Metoda RS (ang. *Random Sequential*), gdzie kolejność wierzchołków jest pseudolosowa. Złożoność obliczeniowa wynosi $O(V + E)$.
— Metoda LF (ang. *Largest First*), gdzie wierzchołki są kolorowane według nierosnących stopni. Złożoność obliczeniowa wynosi $O(V + E)$.
— Metoda SL (ang. *Smallest Last*), gdzie lista wierzchołków jest przygotowywana od końca, przez wybieranie (i usuwanie) wierzchołków o najmniejszym stopniu na danym etapie. Złożoność obliczeniowa wynosi $O(V + E)$. W naszej implementacji wyznaczanie uporządkowania wierzchołków SL zajmuje czas $O(V^2)$. W pracy [21] opisano sposób uzyskania złożoności $O(V + E)$ przy użyciu list powiązanych podwójnie (rodzaj sortowania bukietowego wierzchołków ze względu na stopnie).

— Metoda CS (ang. *Connected Sequential*), gdzie kolejność wierzchołków jest wyznaczona przez BFS lub DFS. Złożoność obliczeniowa wynosi $O(V+E)$.
— Metoda DSATUR lub SLF (ang. *Saturation Largest First*), gdzie w każdym kroku wybierany jest wierzchołek o największym *stopniu nasycenia* (liczba kolorów występująca u sąsiadów).

Podane algorytmy były już obecne w naszej bibliotece, ale dodaliśmy nowe wersje, w których dla każdego wierzchołka przechowywane są zbiory kolorów używanych przez sąsiadów. Niestety nie prowadzi to do zwiększenia szybkości działania. Analogiczny pomysł dla problemu kolorowania krawędzi przynosi ogromną poprawę wydajności.

Listing 4.8 przedstawia nową wersję algorytmu SL, która wykorzystuje kopię grafu do wyznaczenia odpowiedniej kolejności wierzchołków. Metoda _greedy_color realizuje kolorowanie zachłanne.

Listing 4.8. Moduł nodecolorsl.

```
#!/usr/bin/python

class SmallestLastNodeColoring:
    """Find a smallest last (SL) node coloring."""

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

    def run(self):
        """Executable pseudocode."""
        graph_copy = self.graph.copy()
        order = list()
        while graph_copy.v() > 0:    # nie dla kazdej implementacji
            source = min(graph_copy.iternodes(), key=graph_copy.degree)
            order.append(source)
            graph_copy.del_node(source)    # usuwa wierzcholek z krawedziami
        for source in reversed(order):    # iterator odwrotny
            self._greedy_color(source)

    def _greedy_color(self, source):
        """Give node the smallest possible color."""
        n = self.graph.v()    # memory O(V)
        used = [False] * n    # is color used?
        for edge in self.graph.iteroutedges(source):
            if self.color[edge.target] is not None:
                used[self.color[edge.target]] = True
        for c in xrange(n):    # check colors
            if not used[c]:
                self.color[source] = c
                break
        return c
```

4.11. Algorytmy z wymianą kolorów

Algorytmy sekwencyjne mogą być uzupełnione mechanizmem wymiany kolorów (ang. *color interchange*), który zwykle prowadzi do poprawy kolorowania kosztem pewnego nakładu pracy. Wymiana kolorów jest uruchamiana w sytuacji, kiedy w danym kroku potrzebny jest nowy kolor dla przetwarzanego wierzchołka. Wymianę kolorów można zrobić na różne sposoby. Jednym ze sposobów jest próba przekolorowania najbliższego sąsiada przetwarzanego wierzchołka, w celu zwolnienia jednego koloru. W naszej bibliotece są obecne następujące algorytmy z wymianą kolorów:

— Metoda USI, czyli US z wymianą kolorów.
— Metoda RSI, czyli RS z wymianą kolorów.
— Metoda CSI, czyli CS z wymianą kolorów.

4.12. Algorytmy zbiorów niezależnych

Kolorowanie wierzchołków grafu jest w zasadzie podziałem zbioru wierzchołków na pewną liczbę zbiorów niezależnych. Wierzchołki jednego koloru tworzą jeden zbiór niezależny. Stąd pomysł na osobną rodzinę algorytmów, do której należy algorytm GIS (ang. *Greedy Independent Sets*), oraz algorytm RLF (ang. *Recursive Largest First*) [22]. W tych algorytmach sukcesywnie wyznacza się maksymalne zbiory niezależne w grafie i przydziela się wierzchołkom poszczególnych zbiorów niezależnych odrębne kolory. Algorytmy różnią się stosowaną regułą wyboru wierzchołków dołączanych do tworzonych zbiorów niezależnych [2].

4.12.1. Algorytm GIS kolorowania wierzchołków

Dane wejściowe: Dowolny graf prosty nieskierowany G.

Problem: Kolorowanie wierzchołków grafu G.

Opis algorytmu: Algorytm przyporządkowuje wierzchołkom kolor c tak, aby dany wierzchołek nie sąsiadował z innym wierzchołkiem z kolorem c. Wierzchołki są kolorowane w kolejności *najmniejszych* stopni w malejącym podgrafie indukowanym. Podgraf indukowany zawiera tylko wierzchołki bez koloru, które nie sąsiadują z wierzchołkami z kolorem c. Po przyporządkowaniu koloru c wszystkim możliwym wierzchołkom, wierzchołki te są usuwane. Dalej procedura powtarza się z użyciem koloru $c+1$, aż do momentu pokolorowania wszystkich wierzchołków grafu.

Złożoność: Złożoność czasowa algorytmu podawana w literaturze wynosi $O(V^2+VE)$, co dla grafów spójnych prowadzi do $O(VE)$ [2]. Złożoność pamięciowa algorytmu zależy od implementacji. Jeżeli wykonujemy kopię grafu, to złożoność pamięciowa jest liniowa $O(V+E)$. Jeżeli wykorzystujemy tylko

listę stopni wierzchołków, to złożoność pamięciowa wyniesie $O(V)$. W niniejszej pracy sprawdziliśmy oba podejścia.

Uwagi: Doświadczenie pokazuje, że optymalne kolorowania nie zawsze odpowiadają największym zbiorom niezależnym, do których zmierza reguła GIS w każdej iteracji.

Listing 4.9. Moduł nodecolorgis.

```
#!/usr/bin/python

class GISNodeColoring:
    """Greedy Independent Sets (GIS) algorithm for node coloring."""

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

    def run(self):
        """Executable pseudocode."""
        uncolored_graph = self.graph.copy()
        c = 0
        while uncolored_graph.v() > 0:
            available_graph = uncolored_graph.copy()
            while available_graph.v() > 0:
                source = min(available_graph.iternodes(),
                    key=available_graph.degree)
                self.color[source] = c
                uncolored_graph.del_node(source)
                to_delete = [source]
                to_delete.extend(available_graph.iteradjacent(source))
                for target in to_delete:
                    available_graph.del_node(target)
            c += 1
```

4.12.2. Algorytm RLF kolorowania wierzchołków

Dane wejściowe: Dowolny graf prosty nieskierowany G.

Problem: Kolorowanie wierzchołków grafu G.

Opis algorytmu: Algorytm przyporządkowuje wierzchołkom kolor c tak, aby dany wierzchołek nie sąsiadował z innym wierzchołkiem z kolorem c. Jako pierwszy wierzchołek do kolorowania wybierany jest ten o największym stopniu w podgrafie generowanym przez wierzchołki niepokolorowane. Następne wierzchołki są kolorowane w kolejności *największych* stopni w podgrafie indukowanym. Podgraf indukowany zawiera tylko wierzchołki bez koloru,

które sąsiadują z wierzchołkami z kolorem c. Po przyporządkowaniu koloru c wszystkim możliwym wierzchołkom, wierzchołki te są usuwane. Dalej procedura powtarza się z użyciem koloru $c+1$, aż do momentu pokolorowania wszystkich wierzchołków grafu.

Złożoność: Złożoność czasowa algorytmu podawana w literaturze wynosi $O(V^2 + VE)$, co dla grafów spójnych prowadzi do $O(VE)$ [2].

Uwagi: Reguła RLF zmierza do wyznaczania takich zbiorów niezależnych, aby część niepokolorowana grafu pozostała z jak najmniejszą liczbą krawędzi. Zwykle jest to lepsza strategia niż reguła GIS.

Listing 4.10. Moduł nodecolorrlf.

```
#!/usr/bin/python

class RLFNodeColoring:
    """Recursive Largest First (RLF) algorithm for node coloring."""

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

    def run(self):
        """Executable pseudocode."""
        uncolored_graph = self.graph.copy()    # O(V+E) memory
        c = 0
        while uncolored_graph.v() > 0:
            available_graph = uncolored_graph.copy()
            source = max(available_graph.iternodes(),
                key=available_graph.degree)
            self.color[source] = c
            uncolored_graph.del_node(source)
            neighbors = set(available_graph.iteradjacent(source))
            to_delete = [source]
            to_delete.extend(available_graph.iteradjacent(source))
            for target in to_delete:
                available_graph.del_node(target)
            while available_graph.v() > 0:
                source = max(available_graph.iternodes(), key=lambda node:
                    len(set(uncolored_graph.iteradjacent(node)) & neighbors))
                self.color[source] = c
                uncolored_graph.del_node(source)
                neighbors.union(available_graph.iteradjacent(source))
                to_delete = [source]
                to_delete.extend(available_graph.iteradjacent(source))
                for target in to_delete:
                    available_graph.del_node(target)
            c += 1
```

5. Kolorowanie krawędzi

Kolorowanie krawędzi multigrafu (ang. *edge coloring*) polega na przyporządkowaniu krawędziom kolorów tak, że dowolne dwie krawędzie mające wspólny wierzchołek dostają różne kolory [23]. Rozpatruje się tylko multigrafy spójne, nieskierowane, nie zawierające pętli. Jeżeli dla danego multigrafu istnieje dozwolone kolorowanie krawędzi zawierające k kolorów, to mówimy że multigraf jest *k-kolorowalny krawędziowo*. Kolorowanie krawędzi jest *optymalne*, jeżeli zawiera najmniejszą możliwą liczbę kolorów. Ta najmniejsza liczba kolorów nazywana jest *indeksem chromatycznym* (ang. *chromatic index*), a oznaczana jest symbolem $\chi'(G)$ lub $\chi_1(G)$. Problem znajdowania optymalnego kolorowania krawędzi jest NP-zupełny. Optymalne kolorowanie krawędzi grafu jest podziałem jego krawędzi na minimalną liczbę skojarzeń [2].

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

$$\Delta(G) \leqslant \chi'(G) \leqslant \Delta(G) + 1. \tag{5.1}$$

Stąd istnieje podział grafów na grafy klasy 1 $[\chi'(G) = \Delta(G)]$ i grafy klasy 2 $[\chi'(G) = \Delta(G) + 1]$. Badania pokazują, że liczba grafów z n wierzchołkami w klasie 2 jest znacznie mniejsza od liczby grafów w klasie 1.

Do klasy 1 należą grafy dwudzielne, grafy pełne K_n (n parzyste), grafy planarne o $\Delta \geqslant 8$, grafy koła W_n, większość grafów przypadkowych. Do klasy 2 należą grafy cykliczne C_n (n nieparzyste), grafy pełne K_n (n nieparzyste), żmirłacze (ang. *snarks*, grafy kubiczne 2-spójne krawędziowo, np. graf Petersena) [24].

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

$$\chi'(G) \leqslant (3/2)\Delta(G), \tag{5.2}$$

$$\chi'(G) \leqslant \Delta(G) + \mu(G), \tag{5.3}$$

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

Stwierdzenie: Wszystkie grafy kubiczne hamiltonowskie są klasy 1. W grafach kubicznych liczba wierzchołków jest parzysta, a cykl Hamiltona zawiera parzystą liczbę krawędzi. Wystarczy krawędzie należące do cyklu Hamiltona pokolorować na przemian dwoma kolorami, a trzeci kolor wykorzystać dla pozostałych krawędzi.

5.1. Kolorowanie krawędzi grafu dwudzielnego

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

$$\chi'(G) = \Delta(G). \tag{5.4}$$

Istnieją wydajne sekwencyjne algorytmy optymalnego kolorowania krawędzi grafów dwudzielnych (Δ kolorów) [25]. Uzyskane złożoności obliczeniowe to $O(E\Delta)$, a nawet $O(E \log \Delta)$.

5.2. Kolorowanie krawędzi grafu planarnego

Twierdzenie (Vising, 1965): Grafy planarne z $\Delta \geqslant 8$ są klasy 1.

Vising podał również hipotezę (Planar Graph Conjecture), że również grafy planarne z $\Delta = 6$ i $\Delta = 7$ są klasy 1 (czyli łącznie grafy planarne z $\Delta \geqslant 6$). W roku 2001 Sanders i Zhao pokazali, że grafy planarne z $\Delta = 7$ są klasy 1 [26]. Przypadek $\Delta = 6$ pozostaje otwarty.

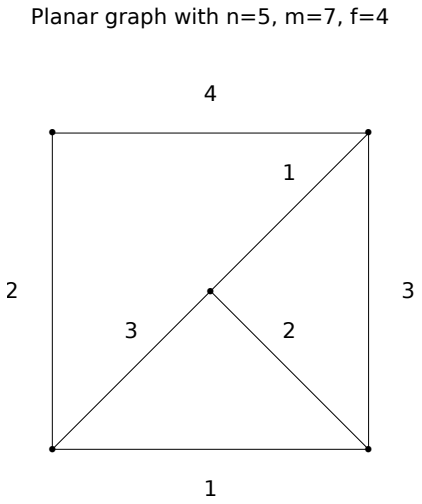
W pracy [27] podano algorytm sekwencyjny przydzielający Δ kolorów przy $\Delta \geqslant 9$, o złożoności $O(V \log V)$. Autorzy bazują na dowodzie twierdzenia Visinga, a metoda nie działa dla grafów planarnych z $\Delta = 8$. Ciekawe, że metoda nie korzysta z płaskiej reprezentacji grafu i działa dla pewnych grafów toroidalnych. Algorytm z pracy [28] przydziela Δ kolorów grafom planarnym z $\Delta \geqslant 8$ i ma złożoność $O(V^2)$.

Podamy przykłady grafów planarnych z małym Δ, które są klasy 2. Vising pokazał, że dla $\Delta = 3, 4, 5$ można skonstruować grafy planarne klasy 2 na bazie brył platońskich (wielościanów foremnych), w których na jednej krawędzi umieszcza się dodatkowy wierzchołek. Dla $\Delta = 2$ mamy cykle nieparzyste, np. $C_3 = K_3$, C_5. Dla $\Delta = 3$ przykładem jest graf z $|V| = 5$ z rysunku 5.1, zbudowany na bazie grafu tetraedru (czworościanu). Dla $\Delta = 4$ przykładem jest graf z $|V| = 5$ z rysunku 5.2. Innym przykładem jest graf z $|V| = 7$, zbudowany na bazie grafu oktaedru (ośmiościanu). Dla $\Delta = 5$ przykładem jest graf z $|V| = 13$, zbudowany na bazie grafu ikosaedru (dwudziestościanu).

5.3. Skojarzenia

Skojarzenie (ang. *matching*) w grafie nieskierowanym $G = (V, E)$ jest to taki podzbiór krawędzi M, że każdy wierzchołek z V jest końcem co najwyżej jednej krawędzi z M [29]. Czasem używana jest nazwa *zbiór niezależny krawędzi* (ang. *independent edge set*). Skojarzenie jest *maksymalne* (ang. *maximal matching*), jeżeli nie jest podzbiorem żadnego innego skojarzenia. Skojarzenie jest *największe (najliczniejsze)* (ang. *maximum matching*), jeżeli w grafie nie istnieje skojarzenie o większej liczbie krawędzi.

Skojarzenie jest *doskonałe* (ang. *perfect matching*), kiedy każdy wierzchołek grafu jest końcem pewnej krawędzi należącej do tego skojarzenia. Skojarzenie doskonałe może istnieć tylko dla grafu o parzystej liczbie wierzchołków.



Rysunek 5.1. Graf planarny klasy 2 z $\Delta = 3$. Jest to graf K_4, w którym na środku jednej krawędzi dodano piąty wierzchołek.

Skojarzenie doskonałe jest maksymalne i największe. Skojarzenie doskonałe jest jednocześnie pokryciem krawędziowym grafu o najmniejszej liczności.

W przypadku grafów dwudzielnych [30] istnieje kilka algorytmów wielomianowych wyznaczających największe skojarzenie:

— algorytm z metodą Forda-Fulkersona,
— algorytm z metodą ścieżki powiększającej,
— algorytm Hopcrofta-Karpa.

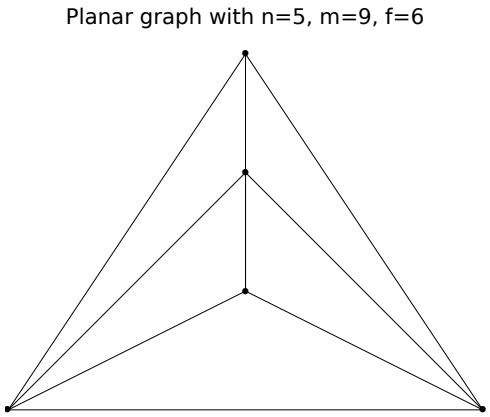
Związek kolorowania krawędzi ze skojarzeniami czasem pomaga w ustaleniu indeksu chromatycznego, ale czasem nie daje żadnych wskazówek [23].

Przykład: Graf kubiczny na rysunku 5.3 ma $n = 16$ wierzchołków i $m = 24$ krawędzie, ale dowolne największe skojarzenie ma 7 krawędzi. Stąd $24/7 > 3$ i indeks chromatyczny wynosi 4.

Przykład: Graf Petersena jest kubiczny, ma $n = 10$ wierzchołków i $m = 15$ krawędzi (rysunek 5.4). Graf posiada skojarzenia doskonałe zawierające 5 krawędzi. Mimo że $15/5 = 3$, to indeks chromatyczny wynosi 4.

5.4. Pokrycie krawędziowe

Pokrycie krawędziowe (ang. *edge cover*) grafu nieskierowanego $G = (V, E)$ jest to podzbiór C zbioru krawędzi E, taki że każdy wierzchołek z V jest końcem co najmniej jednej krawędzi z C [31]. Trywialnym i największym pokryciem krawędziowym jest cały zbiór E.



Rysunek 5.2. Graf planarny klasy 2 z $\Delta = 4$. Jest to graf K_4, w którym trzy wierzchołki połączono z piątym wierzchołkiem.

5.5. Poprawność kolorowania krawędzi

Kierując się techniką *test-driven development* podamy test, jaki powinna spełniać każda metoda poprawnego kolorowania krawędzi. Listing przedstawia metodę, która testuje algorytm zawarty w fikcyjnej klasie EdgeColoring. W teście porównywany jest stopień każdego wierzchołka z liczbą kolorów krawędzi wychodzących z wierzchołka.

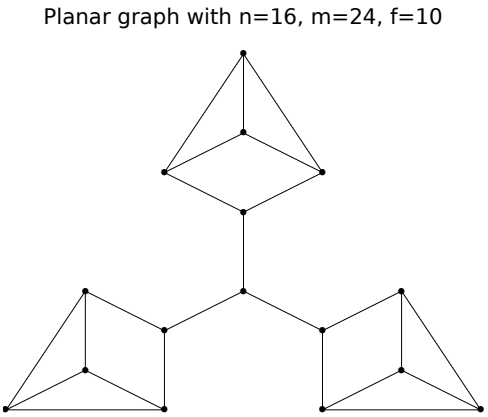
Listing 5.1. Test poprawności kolorowania krawędzi.

```
def test_edge_coloring(self):
    algorithm = EdgeColoring(self.G)
    algorithm.run()
    for edge in self.G.iteredges():
        self.assertNotEqual(algorithm.color[edge], None)
    for node in self.G.iternodes():
        color_set = set()
        for edge in self.G.iteroutedges(node):
            if edge.source > edge.target:
                color_set.add(algorithm.color[~edge])
            else:
                color_set.add(algorithm.color[edge])
        self.assertEqual(len(color_set), self.G.degree(node))
```

5.6. Kolorowanie zachłanne jednej krawędzi

Listing przedstawia metodę _greedy_color która realizuje kolorowanie zachłanne jednej krawędzi.

Listing 5.2. Kolorowanie zachłanne jednej krawędzi.



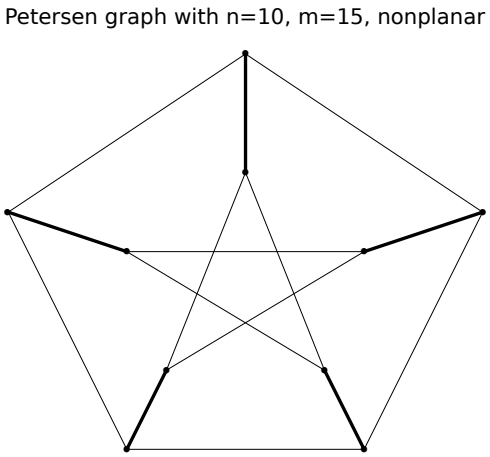
Rysunek 5.3. Graf planarny kubiczny klasy 2. Jego dowolne największe skojarzenie zawiera 7 krawędzi.

```
def _greedy_color(self, edge):
    """Give edge the smallest possible color."""
    k = 2 * self.graph.v()    # tyle miejsc na kolory wystarczy
    used = [False] * k
    for edge2 in self.graph.iteroutedges(edge.source):
        if edge2.source > edge2.target:
            edge2 = ~edge2
        if self.color[edge2] is not None:
            used[self.color[edge2]] = True
    for edge2 in self.graph.iteroutedges(edge.target):
        if edge2.source > edge2.target:
            edge2 = ~edge2
        if self.color[edge2] is not None:
            used[self.color[edge2]] = True
    for c in xrange(k):    # check colors
        if not used[c]:
            self.color[edge] = c
            break
    return c
```

Testy pokazują, że obliczanie za każdym razem kolorów zajętych przez sąsiadujące krawędzie jest powolne. Lepiej jest dla każdego wierzchołka przechowywać zbiór kolorów wykorzystanych przez krawędzie dochodzące. Takie podejście realizuje funkcja _greedy_color_with_saturation(). Zajęta pamięć jest rzędu $O(V^2)$.

Listing 5.3. Kolorowanie jednej krawędzi z saturacją.

```
def _greedy_color_with_saturation(self, edge):
    """Give edge the smallest possible color."""
    for c in xrange(self.m):
        if (c in self.saturation[edge.source] or
            c in self.saturation[edge.target]):
            continue    # kolor juz uzyty
```



Rysunek 5.4. Graf Petersena z zaznaczonym skojarzeniem doskonałym.

```
        else:    # kolor jest wolny
            self.color[edge] = c
            self.saturation[edge.source].add(c)
            self.saturation[edge.target].add(c)
            break    # kolor przydzielony
    return c
```

5.7. Kolorowanie krawędzi z grafem krawędziowym

Dla grafu G można utworzyć graf krawędziowy $L(G)$ w następujący sposób. Wierzchołki w $L(G)$ reprezentują krawędzie w G. Krawędzie w $L(G)$ reprezentują sąsiedztwo krawędzi w G (czy mają wspólny wierzchołek w G). W tej sytuacji kolorowanie krawędzi grafu G sprowadza się do kolorowania wierzchołków grafu krawędziowego. Stąd można wysnuć wniosek, że model kolorowania wierzchołków jest ogólniejszy [2].

Algorytm kolorowania krawędzi z grafem krawędziowym jest dostępny w bibliotece algorytmów grafowych rozwijanej w Instytucie Fizyki [7]. Algorytm jest zawarty w klasie EdgeColoringWithLineGraph.

5.8. Algorytm US kolorowania krawędzi

Dane wejściowe: Dowolny graf prosty nieskierowany G.

Problem: Kolorowanie krawędzi grafu G.

Opis algorytmu: Algorytm koloruje kolejne krawędzie metodą zachłanną, w kolejności wyznaczonej przez implementację (iterator krawędzi).

Złożoność: Dla każdej krawędzi wykonywana jest metoda zachłanna, której czas jest rzędu $O(\Delta)$, co sumaryczne daje czas $O(E\Delta)$. Złożoność czasowa algorytmu jest więc szacowana na $O(V + E\Delta)$ [sprawdzamy $O(VE)$]. Złożoność pamięciowa algorytmu jest liniowa $O(V + E)$, ponieważ słownik color ma rozmiar $O(V)$, natomiast pomocniczy słownik saturation jest ograniczony przez $O(V + E)$.

Uwagi: Algorytm jest bardzo prosty, ale uzyskane kolorowanie może nie być optymalne nawet dla prostych grafów.

Listing 5.4. Moduł edgecolorus.

```
#!/usr/bin/python

class UnorderedSequentialEdgeColoring:
    """Find an unordered sequential edge coloring."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict()   # {edge: int}
        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
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")
        self.saturation = dict((node, set())
            for node in self.graph.internodes())

    def run(self):
        """Executable pseudocode."""
        for edge in self.graph.iteredges():
            self._greedy_color_with_saturation(edge)

    def _greedy_color_with_saturation(self, edge):
        """Give edge the smallest possible color."""
        for c in xrange(self.m):
            if (c in self.saturation[edge.source] or
                c in self.saturation[edge.target]):
```

```
                continue    # kolor juz uzyty
            else:    # kolor jest wolny
                self.color[edge] = c
                self.saturation[edge.source].add(c)
                self.saturation[edge.target].add(c)
                break    # kolor przydzielony
        return c
```

5.9. Algorytm RS (NC) kolorowania krawędzi

Dane wejściowe: Dowolny graf prosty nieskierowany G.

Problem: Kolorowanie krawędzi grafu G.

Opis algorytmu: Algorytm koloruje kolejne krawędzie metodą zachłanną, w kolejności pseudolosowej.

Złożoność: Złożoność jest taka, jak dla algorytmu US, czyli w przybliżeniu $O(V + E\Delta)$ [sprawdzamy $O(VE)$]. W metodzie run() uzyskuje się pseudolosową kolejność krawędzi za pomocą metody random.shuffle z biblioteki standardowej Pythona, o złożoności $O(E)$. Złożoność pamięciowa algorytmu jest liniowa $O(V + E)$, ponieważ słownik color ma rozmiar $O(V)$, natomiast pomocniczy słownik saturation jest ograniczony przez $O(V + E)$.

Uwagi: Algorytm jest bardzo prosty, ale uzyskane kolorowanie może nie być optymalne nawet dla prostych grafów. Algorytm RS w literaturze nosi również nazwę algorytmu NC (ang. *naive coloring*).

Łatwo pokazać, że algorytmy US i RS są algorytmami 2-aproksymacyjnymi, czyli liczba kolorów przydzielonych krawędziom nie przekroczy $2\chi'(G)$. Wiemy, że $\Delta(G) \leqslant \chi'(G)$. Z drugiej strony, kolorowanie zachłanne krawędzi może wymagać liczby kolorów równej $2(\Delta(G) - 1) + 1 = 2\Delta(G) - 1$. Stąd liczba kolorów k wykorzystanych do kolorowania krawędzi grafu ma ograniczenie

$$\chi'(G) \leqslant k \leqslant 2\Delta(G) - 1 < 2\Delta(G) \leqslant 2\chi'(G). \tag{5.5}$$

Listing 5.5. Moduł edgecolorrs.

```
#!/usr/bin/python

import random

class RandomSequentialEdgeColoring:
    """Find a random sequential edge coloring."""

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

```
        self.color = dict()    # {edge: int}
        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
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")
        self.saturation = dict((node, set())
            for node in self.graph.iternodes())

    def run(self):
        """Executable pseudocode."""
        edge_list = list(self.graph.iteredges())
        random.shuffle(edge_list)    # O(E) time
        for edge in edge_list:
            self._greedy_color_with_saturation(edge)

    def _greedy_color_with_saturation(self, edge):
        """Give edge the smallest possible color."""
        for c in xrange(self.m):
            if (c in self.saturation[edge.source] or
                c in self.saturation[edge.target]):
                continue   # kolor juz uzyty
            else:   # kolor jest wolny
                self.color[edge] = c
                self.saturation[edge.source].add(c)
                self.saturation[edge.target].add(c)
                break   # kolor przydzielony
        return c
```

5.10. Kolorowanie krawędzi z BFS

Dane wejściowe: Dowolny graf prosty nieskierowany G.

Problem: Kolorowanie krawędzi grafu G.

Opis algorytmu: Algorytm realizuje przechodzenie przez graf metodą BFS. Kolorowane są wszystkie krawędzie wychodzące z przetwarzanego wierzchołka (kolorowanie zachłanne). Po zakończeniu pracy algorytmu, oprócz słownika color z kolorami krawędzi, otrzymujemy słownik parent, który jest lasem przeszukiwania BFS.

Złożoność: Przechodzenie przez graf metodą BFS zajmuje czas $O(V + E)$. Dla każdej krawędzi wykonywana jest metoda _greedy_color, której czas jest rzędu $O(\Delta)$, co sumaryczne daje czas $O(E\Delta)$. Złożoność czasowa algorytmu jest więc szacowana na $O(V + E\Delta)$ [sprawdzimy $O(VE)$]. Złożoność pamięciowa algorytmu jest liniowa $O(V + E)$, ponieważ słowniki color i parent mają

rozmiar $O(V)$, natomiast pomocniczy słownik _used jest ograniczony przez $O(V + E)$.

Uwagi: Algorytm wyznacza optymalne kolorowanie krawędzi dla drzew. Algorytm z klasy BFSEdgeColoring można by nazwać również *Connected Sequential Edge Coloring*, przez analogię do podobnego kolorowania wierzchołków.

Listing 5.6. Moduł edgecolorbfs.

```
#!/usr/bin/python

from Queue import Queue

class BFSEdgeColoring:
    """Find a greedy BFS edge coloring."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.parent = dict()    # BFS tree
        self.color = dict()    # {edge: int}
        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
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")
        self.saturation = dict((node, set())
            for node in self.graph.iternodes())

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:    # tylko jedna skladowa spojna
            self._visit(source)
        else:
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self._visit(node)

    def _visit(self, node):
        """Explore a connected component."""
        Q = Queue()
        self.parent[node] = None    # before Q.put
        Q.put(node)
        while not Q.empty():
            source = Q.get()
            for edge in self.graph.iteroutedges(source):
                if edge.target not in self.parent:    # tu nie bylismy
                    self.parent[edge.target] = source    # before Q.put
                    Q.put(edge.target)
                if edge.source > edge.target:
                    edge = ~edge
```

```
            if self.color[edge] is None:
                self._greedy_color_with_saturation(edge)

    def _greedy_color_with_saturation(self, edge):
        """Give edge the smallest possible color."""
        for c in xrange(self.m):
            if (c in self.saturation[edge.source] or
                c in self.saturation[edge.target]):
                continue   # color is used
            else:   # color is free
                self.color[edge] = c
                self.saturation[edge.source].add(c)
                self.saturation[edge.target].add(c)
                break   # color is set
        return c
```

5.11. Algorytm NTL kolorowania krawędzi

Nazwa algorytmu pochodzi od pierwszych liter nazwisk twórców (Nishizeki, Terada, Leven) [32], [28]. W algorytmie wykorzystywane jest przekolorowywanie krawędzi (zamiana kolorów). Algorytm NTL przydzieli krawędziom co najwyżej $\Delta+1$ kolorów. Do opisu algorytmu potrzebne są pewne definicje [1].

Definicja: *Kolor brakujący* (ang. *missing color*) dla wierzchołka v grafu G to kolor, który nie został przydzielony żadnej krawędzi dochodzącej do v. Symbol $M(v)$ oznacza zbiór wszystkich kolorów brakujących dla v.

Definicja: Dla każdego wierzchołka v ustalamy pewien jego kolor brakujący $m(v)$. *Wachlarz* F (ang. *fan*) przy wierzchołku v jest to ciąg krawędzi (v, w_0), (v, w_1), ..., (v, w_s), taki że krawędź (v, w_i) ma przydzielony kolor $m(w_{i-1})$, $i > 0$. Wachlarz rozpoczyna się krawędzią bez koloru (v, w_0), a liczba s to rozpiętość wachlarza (ang. *span of the fan*).

Stwierdzenie: Jeżeli wybrana krawędź (u, v) nie jest pokolorowana, to każdy z wierzchołków u, v ma przynajmniej dwa kolory brakujące.

Procedura przekolorowania: Celem procedury przekolorowania dla krawędzi (u, v) jest uzyskanie *wspólnego* brakującego koloru dla wierzchołków u i v, aby ten kolor mógł być użyty do pokolorowania krawędzi (u, v). Wyznaczamy maksymalny wachlarz F przy wierzchołku v, przy czym $w_0 = u$. Dalej możliwe są dwa przypadki.
(1) $m(w_s)$ należy do $M(v)$. Wtedy kolorujemy krawędzie (v, w_i) kolorami $m(w_i)$, $i = 0, \ldots, s$ [zwalniamy kolor $m(w_0) = m(u)$].
(2) $m(w_s)$ nie należy do $M(v)$. Rozważamy ścieżkę P w grafie G zaczynającą się w w_s, złożoną z krawędzi pokolorowanych na przemian kolorami $m(v)$ i $m(w_s)$. Dalej jest kilka możliwości.
(2a) Ścieżka P jest zerowa. Wtedy krawędzie (v, w_i) otrzymują kolor $m(w_i)$, $i = 0, \ldots, s-1$, krawędź (v, w_s) otrzymuje kolor $m(v)$.

(2b) Ścieżka P osiąga wierzchołek v. Istnieje wtedy krawędź (v, w_j) $(0 < j < s-1)$ [ważne!], która ma kolor $m(w_{j-1}) = m(w_s)$. Zmieniamy kolejność kolorów na ścieżce P, krawędź (v, w_j) otrzymuje kolor $m(v)$, krawędzie (v, w_i) otrzymują kolor $m(w_i)$, $i = 0, \ldots, j-1$.
(2c) Ścieżka P nie osiąga wierzchołka v, ale osiąga wierzchołek w_j będący końcem pewnej krawędzi z wachlarza, a do tego ścieżka ma długość nieparzystą i kolor przy w_j to $m(v)$. Wtedy zmieniamy kolejność kolorów na ścieżce P, krawędź (v, w_j) otrzymuje kolor $m(v)$, krawędzie (v, w_i) otrzymują kolor $m(w_i)$, $i = 0, \ldots, j-1$ [jeżeli $w_j = w_0$, to jedynę operacją jest kolorowanie krawędzi (v, w_0) kolorem $m(v)$].
(2d) Ścieżka P nie osiąga wierzchołka v i nie dochodzi do brzegu wachlarza kolorem $m(v)$. Wtedy zmieniamy kolejność kolorów na ścieżce P, krawędź (v, w_s) otrzymuje kolor $m(v)$, krawędzie (v, w_i) otrzymują kolor $m(w_i)$, $i = 0, \ldots, s-1$.

Dane wejściowe: Dowolny graf prosty nieskierowany G.

Problem: Kolorowanie krawędzi grafu G.

Opis algorytmu: Algorytm rozpoczyna się od wyznaczenia stopnia grafu $\Delta(G)$. Jeżeli $\Delta(G) \leqslant 2$, to uruchamiany jest prosty algorytm kolorowania krawędzi, w naszej implementacji jest to kolorowanie krawędzi z BFS. Dla większych $\Delta(G)$ algorytm wyznacza liczbę dostępnych kolorów jako $k = \Delta(G)+1$, następnie próbuje zachłannie przydzielić każdej krawędzi najmniejszy wspólny kolor brakujący jej końców. Jeżeli nie istnieje wspólny kolor brakujący, to dla krawędzi uruchamiana jest kluczowa metoda _recolor, omawiana wcześniej.

Złożoność: Procedura przekolorowania może działać w czasie liniowym, stąd całkowity czas pracy algorytmu NTL wynosi $O(VE)$ [1]. Złożoność pamięciową algorytmu szacujemy na $O(V^2)$ [słownik missing].

Uwagi: Algorytm wyznacza optymalne kolorowanie krawędzi dla szerokiej rodziny grafów, m.in. dla grafów cyklicznych C_n, grafów kołowych W_n, gwiazd $K_{1,s}$, grafów planarnych $(\Delta \geqslant 9)$ i prawie wszystkich grafów przypadkowych [1]. Testy naszej implementacji pokazują, że przekolorowanie krawędzi pojawia się dopiero po pewnym czasie pracy algorytmu, ponieważ na początku są dostępne wspólne kolory brakujące. Po drugie, warto podkreślić znaczenie warunku $k = \Delta + 1$, nawet gdy wiemy, że powinno wystarczyć Δ kolorów (grafy dwudzielne). Dzięki temu warunkowi istnieje kolor brakujący dla wierzchołka z wszystkimi pokolorowanymi krawędziami, a to jest potrzebne przy przekolorowywaniu wachlarza. Po trzecie, w znanej nam literaturze opis procedury przekolorowania krawędzi jest niepełny, brakuje istotnego przypadku 2c, kiedy ścieżka dochodzi do brzegu wachlarza, ale nie osiąga jego centrum.

Listing 5.7. Moduł edgecolorntl.

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

```
#
# Based on Java code from
# https://github.com/martakuzak/GIS

from edgecolorbfs import BFSEdgeColoring

class NTLEdgeColoring:
    """Find the NTL edge coloring."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict()   # {edge: int}
        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
                self.m += 1
        if len(self.color) < self.m:
            raise ValueError("edges are not unique")
        self.missing = None   # kolory brakujace wierzcholkow

    def run(self, source=None):
        """Executable pseudocode."""
        Delta = max(self.graph.degree(node)
            for node in self.graph.iternodes())
        if Delta <= 2:
            algorithm = BFSEdgeColoring(self.graph)
            algorithm.run()
            self.color = algorithm.color
        else:
            # Ustal liczbe wykorzystywanych kolorow.
            k = Delta + 1   # prawie optymalnie (grafy proste!)
            self.missing = dict((node, set(xrange(k)))
                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)   # najmniejszy kolor dostepny
                    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."""
        # Przygotowanie kolorow brakujacych m(*).
        mis = dict((node, min(self.missing[node]))
            for node in self.graph.iternodes())
        # Tworzymy wachlarz dla krawedzi edge.
        # Wachlarz rozpoczyna sie od wierzcholka w_0 (krawedz edge).
        fan = [edge]   # cale krawedzie wychodzace z edge.source
        # Zbior do szybkiego sprawdzania, czy wierzcholek nalezy do wachlarza.
        fan_set = set([edge.target])   # zbior koncow krawedzi
        # alpha to kolor brakujacy dla edge.source
        alpha = mis[edge.source]
        tmp_v = edge.target   # do chodzenia po koncach krawedzi wachlarza
        finished = False
        # W petli szukamy kolejnych krawedzi wachlarza.
        while not finished:
            finished = True
            for edge1 in self.graph.iteroutedges(edge.source):
                # Kolor krawedzi ma byc kolorem brakujacym
                # poprzedniego wierzcholka.
                if edge1.source > edge1.target:
                    c = self.color[~edge1]
                else:
                    c = self.color[edge1]
                if c == mis[tmp_v] and edge1.target not in fan_set:
                    # Dodajemy krawedz do wachlarza.
                    tmp_v = edge1.target
                    fan.append(edge1)
                    fan_set.add(edge1.target)
                    finished = False
                    break
        # Wachlarz zostal skonstruowany.
        # tmp_v oznacza teraz ostatni wierzcholek wachlarza w_s.
        # Definiujemy kolor beta jako kolor brakujacy wierzcholka w_s.
        beta = mis[tmp_v]
        # Jezeli kolor brakujacy w_s jest rowniez kolorem brakujacym
        # edge.source, to mozemy przesunac wachlarz, a krawedz fan[-1]
        # pokolorowac kolorem beta.
        if beta in self.missing[edge.source]:   # PRZYPADEK 1
            # Przesuwamy kolory w wachlarzu.
            for i in xrange(len(fan)-1):
                edge1 = fan[i]
                edge2 = fan[i+1]
                c = mis[edge1.target]   # to chcemy dac edge1
                self._del_color(edge2, c)
                self._add_color(edge1, c)
            # Kolor beta dajemy ostatniej krawedzi wachlarza.
            edge1 = fan[-1]
            self._add_color(edge1, beta)
        else:   # PRZYPADEK 2, beta not in self.missing[edge.source]
```

```
        # Tworzymy sciezke o poczatku w w_s i skladajaca sie
        # z krawedzi na przemian kolorow alpha i beta.
        path = []   # tu beda cale krawedzie
        path_set = set([tmp_v])   # w_s, aby przyspieszyc wyszukiwanie
        tmp2_v = tmp_v   # chodzi po wierzcholkach sciezki
        finished = False
        # Zmienna parity pozwala kontrolowac, czy nastepna krawedz
        # powinna byc pokolorowana kolorem alpha czy beta.
        parity = 0
        # W petli szukamy kolejnych krawedzi sciezki path.
        while not finished:
            finished = True
            if parity % 2 == 0:   # kolor alpha
                for edge1 in self.graph.iteroutedges(tmp2_v):
                    # Kolor krawedzi ma byc alpha.
                    if edge1.source > edge1.target:
                        c = self.color[~edge1]
                    else:
                        c = self.color[edge1]
                    if c == alpha and edge1.target not in path_set:
                        tmp2_v = edge1.target
                        path.append(edge1)
                        path_set.add(edge1.target)
                        finished = False
                        break
            else:   # parity % 2 == 1, kolor beta
                for edge1 in self.graph.iteroutedges(tmp2_v):
                    # Kolor krawedzi ma byc beta.
                    if edge1.source > edge1.target:
                        c = self.color[~edge1]
                    else:
                        c = self.color[edge1]
                    if c == beta and edge1.target not in path_set:
                        tmp2_v = edge1.target
                        path.append(edge1)
                        path_set.add(edge1.target)
                        finished = False
                        break
            # Przed przejsciem do szukania kolejnego wierzcholka
            # sciezki zmieniamy parity.
            parity += 1
        # Sciezka path zostala skonstrowana. Sciezka moze nie istniec.
        if len(path) == 0:
            # Przesuwamy kolory w wachlarzu.
            for i in xrange(len(fan)-1):
                edge1 = fan[i]
                edge2 = fan[i+1]
                c = mis[edge1.target]   # to chcemy dac edge1
                self._del_color(edge2, c)
                self._add_color(edge1, c)
            # Kolor alpha dajemy ostatniej krawedzi wachlarza.
            edge1 = fan[-1]
            self._add_color(edge1, alpha)
        elif path[-1].target == edge.source:
            # path dochodzi do edge.source kolorem beta.
            # Odwracamy kolory na sciezce.
            # Najpierw usuwam kolory (pierwszy to alpha), bez ostatniego.
```

```
            for i in xrange(len(path)-1):   # bez ostatniej krawedzi
                c = alpha if (i % 2 == 0) else beta
                self._del_color(path[i], c)
            # Krawedz path[-1] nalezy do wachlarza i ma jeszcze kolor beta.
            # Przesuwamy kolory w wachlarzu, ale nie do konca.
            for i in xrange(len(fan)-1):
                edge1 = fan[i]
                edge2 = fan[i+1]
                c = mis[edge1.target]   # to chcemy dac edge1
                self._del_color(edge2, c)
                self._add_color(edge1, c)
                if c == beta:
                    break
            # Teraz jedna krawedz wachlarza, wspolna ze sciezka,
            # nie ma koloru.
            # Dodaje odwrocone kolory w path (pierwszy to beta).
            for i, edge1 in enumerate(path):   # cala sciezka
                c = beta if (i % 2 == 0) else alpha
                self._add_color(edge1, c)
        # Dalej path[-1].target != edge.source
        elif path[-1].target in fan_set and (len(path) % 2 == 1):
            # path ma dlugosc nieparzysta i osiaga wierzcholek
            # nalezacy do wachlarza klawedzia koloru alpha.
            # Najpierw usuwam kolory (pierwszy to alpha).
            for i, edge1 in enumerate(path):   # cala sciezka
                c = alpha if (i % 2 == 0) else beta
                self._del_color(edge1, c)
            # Mozemy przypadkiem trafic w pierwsza krawedz wachlarza.
            if path[-1].target == edge.target:
                # Nie przesuwamy wachlarza.
                # Dodaje krawedz wachlarza do sciezki dla wygody.
                path.append(~edge)   # odwrotny kierunek!
                # Teraz sciezka ma parzysta liczbe krawedzi.
            else:
                # Przesuwamy kolory w wachlarzu, ale nie do konca.
                for i in xrange(len(fan)-1):
                    edge1 = fan[i]
                    edge2 = fan[i+1]
                    c = mis[edge1.target]   # to chcemy dac edge1
                    self._del_color(edge2, c)
                    self._add_color(edge1, c)
                    if edge2.target == path[-1].target:
                        # Dodaje krawedzi wachlarza do sciezki, aby
                        # latwiej nadac jej kolor.
                        path.append(~edge2)   # odwrotny kierunek!
                        # Teraz sciezka ma parzysta liczbe krawedzi.
                        break
                # Dodaje odwrocone kolory w path
                # (pierwszy to beta, ostatni to alpha).
                for i, edge1 in enumerate(path):
                    c = beta if (i % 2 == 0) else alpha
                    self._add_color(edge1, c)
        else:
            # path moze sie konczyc kolorem alpha lub beta.
            # Przesuwamy kolory w wachlarzu.
            for i in xrange(len(fan)-1):
                edge1 = fan[i]
```

```
            edge2 = fan[i+1]
            c = mis[edge1.target]   # to chcemy dac edge1
            self._del_color(edge2, c)
            self._add_color(edge1, c)
        # Ostatnia krawedz wachlarza jest teraz bez koloru.
        # Odwracamy kolory na sciezce.
        # Najpierw usuwam kolory (pierwszy to alpha).
        for i, edge1 in enumerate(path):   # cala sciezka
            c = alpha if (i % 2 == 0) else beta
            self._del_color(edge1, c)
        # Teraz dodaje odwrocone kolory (pierwszy to beta).
        for i, edge1 in enumerate(path):
            c = beta if (i % 2 == 0) else alpha
            self._add_color(edge1, c)
        # Obrocilismy sciezke.
        # Kolor alpha dajemy ostatniej krawedzi wachlarza.
        edge1 = fan[-1]
        self._add_color(edge1, alpha)
```

5.12. Algorytm Misry i Griesa kolorowania krawędzi

Inny algorytm gwarantujący przydzielenie co najwyżej $\Delta + 1$ kolorów został opublikowany przez Misrę i Griesa [33]. Jego złożoność obliczeniowa wynosi $O(VE)$. W algorytmie korzysta się z wachlarzy, wykonuje się operacje rotowania wachlarzy i odwracania ścieżek [34].

6. Inne modele kolorowania grafów

Klasyczne modele kolorowania grafów są wzbogacane dodatkowymi elementami, które są dodatkowymi więzami, albo uogólnieniami. W tym rozdziale przedstawimy wybrane nowe warianty kolorowania grafów.

6.1. Sprawiedliwe kolorowanie grafów

Sprawiedliwe kolorowanie grafów (ang. *equitable coloring*) polega na kolorowaniu wierzchołków tak, aby (1) dwa sąsiednie wierzchołki miały różne kolory, oraz (2) liczby wierzchołków w dowolnych dwóch klasach kolorów różniły się co najwyżej o jeden [35]. Podobnie definiuje się sprawiedliwe kolorowanie krawędzi.

6.2. Totalne kolorowanie grafów

Totalne kolorowanie grafów (ang. *total coloring*) polega na jednoczesnym kolorowaniu wierzchołków i krawędzi, przy czym żadne dwa sąsiednie obiekty nie mogą mieć tego samego koloru [36].

6.3. Kolorowanie grafów w trybie on-line

Klasyczne algorytmy działają w trybie *off-line*, czyli cały zbiór danych reprezentujących instancję (graf) jest znany przed przystąpieniem do działania. Jednak w pewnych problemach zbiór danych może być poznawany etapami i wtedy rozwiązanie musi być budowane na bieżąco. Odpowiedni algorytm musi działać w trybie *on-line*. Raz wygenerowane rozwiązanie częściowe nie może być modyfikowane. Algorytm nie ma żadnej znajomości przyszłych porcji danych. Problemy w wersji on-line są trudniejsze niż off-line, a otrzymane rozwiązania mogą być niższej jakości.

Kolorowanie on-line definiuje się jako sekwencję żadań, z których każde zawiera nowy wierzchołek v i podzbiór krawędzi łączących nowy wierzchołek v z niektórymi ujawnionymi wcześniej wierzchołkami. Algorytm on-line ma przyporządkować wierzchołkowi v dopuszczalny kolor.

7. Podsumowanie

Najważniejszym zagadnieniem omawianym w niniejszej pracy było kolorowanie wierzchołków i krawędzi grafów. Wykonano przegląd algorytmów kolorowania wierzchołków, w kilku przypadkach stworzono nowe implementacje. Zestawienie nowych i istniejących modułów z biblioteki grafów znajduje się w dodatku A. Poniżej krótko omówimy wyniki pracy.

Zaimplementowano algorytm z powrotami rozwiązujący problem m-kolorowania grafu. Zaimplementowano algorytm korzystający z twierdzenia Brooksa. Zaimplementowano nowe wersje algorytmów sekwencyjnych (US, RS, LF, SL, CS) z wykorzystaniem saturacji. Zaimplementowano dwa algorytmy kolorowania wierzchołków metodą zbiorów niezależnych (GIS, RLF).

W pracy zaimplementowano szereg algorytmów kolorowania krawędzi, poza istniejącym w bibliotece algorytmem wykorzystującym graf krawędziowy. Zaimplementowano trzy algorytmy sekwencyjne (US, RS, BFS), oraz algorytm NTL gwarantujący rozwiązanie gorsze najwyżej o jeden kolor od optymalnego. Algorytm NTL jest najbardziej złożonym algorytmem w niniejszej pracy. Warto podkreślić, że w literaturze często można znaleźć niepełny opis procedury przekolorowania krawędzi grafu, co zostało wykryte dzięki mechanizmom zabezpieczającym języka Python. Błędy wynikające z niepełnego opisu ujawniają się dopiero dla odpowiednio dużych grafów (około 100 wierzchołków)

Nie udało się znaleźć wystarczających informacji, aby zaimplementować algorytm wielomianowy realizujący twierdzenie Königa, czyli algorytm wykorzystujący Δ kolorów do pokolorowania krawędzi grafu dwudzielnego. Innym ciekawym zagadnieniem na przyszłość jest implementacja algorytmów kolorowania krawędzi grafu planarnego.

W pracy zaimplementowano dwa algorytmy wyznaczające pokrycie wierzchołkowe grafu. Pierwszy to algorytm 2-aproksymacyjny, a drugi to algorytm zachłanny. Sprawdzono eksperymentalnie liniową złożoność czasową tych algorytmów.

Przy komputerowym szukaniu rozwiązań jakiegoś problemu kluczową sprawą jest posiadanie pełnych testów sprawdzających poprawność uzyskanego rozwiązania. Dlatego w pracy przygotowano testy sprawdzające poprawność rozwiązań dla problemu wyznaczania zbioru niezależnego, pokrycia wierzchołkowego, kolorowania wierzchołków i krawędzi grafu. Ponadto sprawdzono praktyczną złożoność obliczeniową zaimplementowanych algorytmów. Wykorzystano istniejące generatory grafów, jak również stworzono generator grafów dwudzielnych przypadkowych.

A. Algorytmy i struktury danych z biblioteki grafów

Dla wygody czytelnika w tabeli A.1 zebrano moduły istniejące wcześniej w bibliotece grafów, a w tabeli A.2 zestawiono nowo powstałe moduły i nowe implementacje istniejących modułów.

Tabela A.1. Algorytmy i struktury danych z biblioteki grafów rozwijanej w Instytucie Fizyki UJ, które wykorzystano w pracy.

Klasa lub funkcja	**Moduł**
Edge	edges
Graph	graphs
MultiGraph	multigraphs
GraphFactory	factory
SimpleBFS	bfs
SimpleDFS	dfs
BipartiteGraphBFS	bipartite
is_connected	connected
ExactNodeColoring	nodecolorexact
USINodeColoring	nodecolorusi
RSINodeColoring	nodecolorrsi
CSINodeColoring	nodecolorcsi
DSATURNodeColoring	nodecolordsatur
EdgeColoringWithLineGraph	edgecolorlg

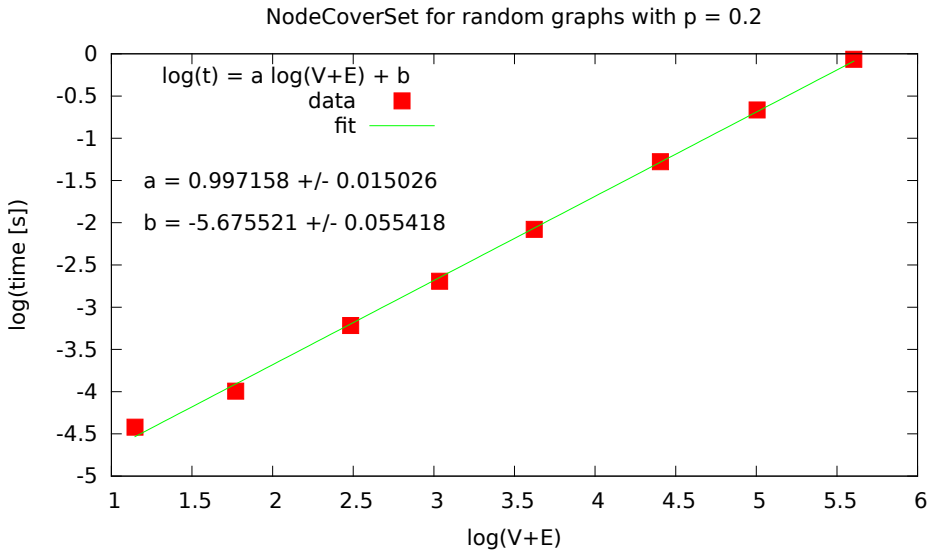
Tabela A.2. Nowe moduły dodane do biblioteki grafów. Znak † oznacza nową implementację danego modułu.

Klasa lub funkcja	**Moduł**
NodeCoverSet	nodecoverapp
DegreeNodeCoverSet	nodecoverdeg
GISNodeColoring	nodecolorgis
RLFNodeColoring	nodecolorrlf
UnorderedSequentialNodeColoring	nodecolorus †
RandomSequentialNodeColoring	nodecolorrs †
ConnectedSequentialNodeColoring	nodecolorcs †
SmallestLastNodeColoring	nodecolorsl †
LargestFirstNodeColoring	nodecolorlf †
BrooksNodeColoring	nodecolorbrooks
BacktrackingNodeColoring	nodecolorbt
UnorderedSequentialEdgeColoring	edgecolorus
RandomSequentialEdgeColoring	edgecolorrs
BFSEdgeColoring	edgecolorbfs
NTLEdgeColoring	edgecolorntl

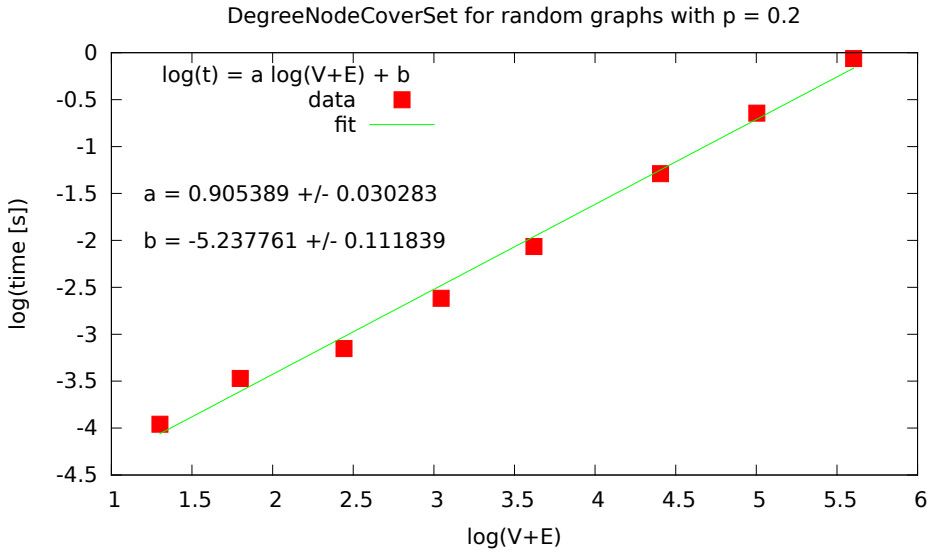
B. Testy pokrycia wierzchołkowego

Testowanie algorytmu aproksymacyjnego. Potwierdzona zależność liniowa $O(V + E)$ (graf przypadkowy z $p = 0.2$). Wyniki testu są przedstawione na rysunku B.1.

Testowanie algorytmu zachłannego. Potwierdzona zależność liniowa $O(V + E)$ (graf przypadkowy z $p = 0.2$). Wyniki testu są przedstawione na rysunku B.2.



Rysunek B.1. Wykres wydajności algorytmu aproksymacyjnego pokrycia wierzchołkowego dla grafów przypadkowych. Współczynnik a bliski 1 potwierdza zależność liniową.



Rysunek B.2. Wykres wydajności algorytmu zachłannego pokrycia wierzchołkowego dla grafów przypadkowych. Współczynnik a bliski 1 potwierdza zależność liniową.

C. Testy kolorowania wierzchołków

W ramach pracy wykonano testy algorytmów kolorowania wierzchołków. Dla algorytmów sekwencyjnych i algorytmów zbiorów niezależnych wyznaczono średnie liczby kolorów przydzielonych przez algorytmy grafom przypadkowym. Wyniki zestawiono w tabelach C.1 i C.2.

Testowanie algorytmu z powrotami. Dla sieci trójkątnej i 3 kolorów [optymalne kolorowanie] zależność nie jest wielomianowa. Dla sieci trójkątnej i 2 kolorów [brak rozwiązania] zależność jest praktycznie $O(V)$. Dla sieci kwadratowej i 2 kolorów [optymalne kolorowanie] zależność jest praktycznie $O(V)$. Dla grafu koło i 4 kolorów [optymalne kolorowanie przy parzystej liczbie wierzchołków] zależność jest praktycznie $O(V)$. Ogólny wniosek jest taki, że dla małej liczby kolorów algorytm nie jest aż tak powolny, jak sugeruje pesymistyczne oszacowanie.

Testowanie algorytmu z twierdzenia Brooksa. Dla grafu pełnego bez jednej krawędzi potwierdzamy złożoność $O(V^2)$ [optymalne $\Delta = |V| - 1$ kolorów]. Dla planarnego kubicznego grafu Halina (graf 3-spójny) dostajemy zależność bliską $O(V)$ [optymalne $\Delta = 3$ kolory], co wynika z małej liczby krawędzi $|E| = 3|V|/2$ i małej liczby potrzebnych kolorów. Wyniki testów są przedstawione na rysunkach C.1 i C.2. Bardziej szczegółowa analiza złożoności algorytmu sugeruje wyraz $O(V + E)$ pochodzący z BFS i wyraz $O(V\Delta)$ z zastosowania metody _greedy_color. Łączna złożoność $O(V + E + V\Delta)$ rzeczywiście redukuje się do $O(V)$ w przypadku kubicznych grafów Halina.

Testowanie algorytmu GIS. Potwierdza się złożoność $O(VE)$. Z analizy kodu nie bardzo widać tej zależności. W testach sprawdzono dwie implementacje, z kopią grafu i słownikiem zawierającym stopnie wierzchołków podgrafu. Szybsza jest wersja z kopią grafu. Testowano grafy przypadkowe i sieć trójkątną. Wyniki testów dla grafów przypadkowych są przedstawione na rysunku C.3.

Testowanie algorytmu RLF. Potwierdza się złożoność $O(VE)$. Mamy dwie implementacje (jak dla GIS), z kopią grafu i słownikiem zawierającym stopnie wierzchołków podgrafu. Szybsza jest wersja ze słownikiem zawierającym stopnie wierzchołków. Testowano grafy przypadkowe i sieć trójkątną. Wyniki testów dla grafów przypadkowych są przedstawione na rysunku C.4.

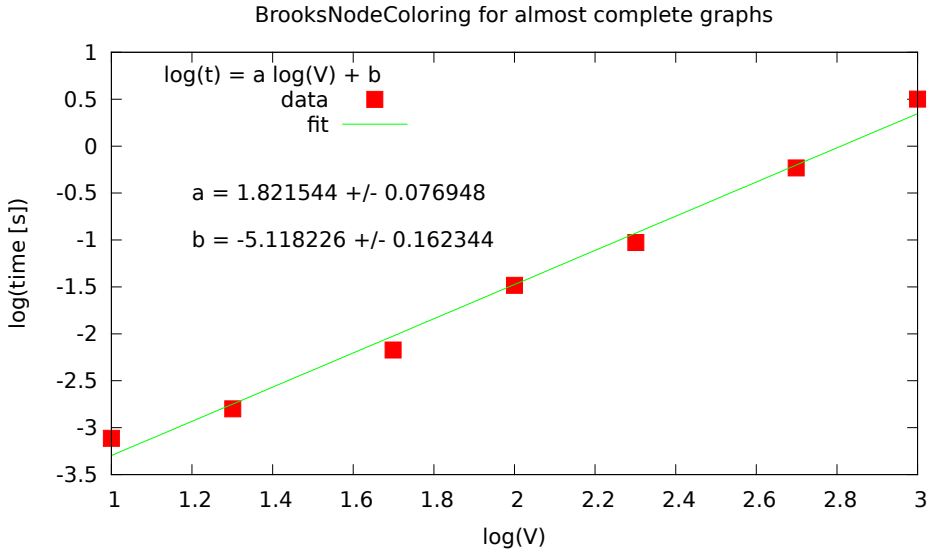
Testowanie algorytmu SL. W bibliotece była dostępna implementacja ze słownikiem zawierającym stopnie wierzchołków podgrafu. Dodano implementację z kopią grafu. Szybsza jest wersja ze słownikiem zawierającym stopnie wierzchołków. Testowano grafy przypadkowe i sieć trójkątną. Testy sugerują złożoność typu $O(V^2)$.

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

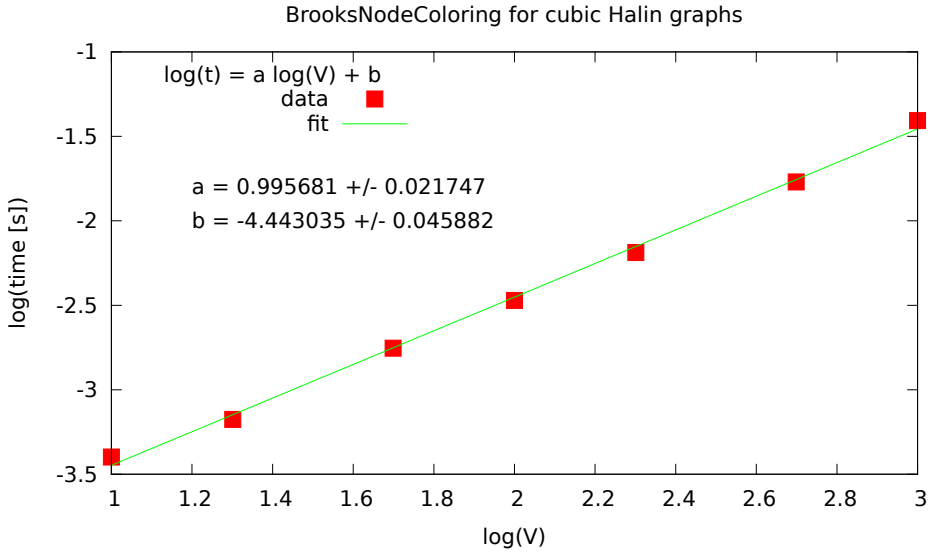
p	**LF**	**SL**	**SLF**	**GIS**	**RLF**
0.1	28.4	29.2	25.4	26.7	26.9
0.2	50.1	51.6	45.9	45.5	46.7
0.3	71.8	73.6	67.0	64.3	66.7
0.4	95.0	96.9	89.4	84.4	88.0
0.5	122.1	124.2	114.3	106.6	111.6
0.6	152.5	153.5	144.1	132.5	138.5
0.7	187.9	190.3	180.4	164.3	172.7
0.8	235.0	239.5	224.6	208.0	214.2
0.9	312.1	314.5	301.0	281.2	283.0

Tabela C.2. Średnia liczba kolorów potrzebnych do pokolorowania grafu losowego z prawdopodobieństwem istnienia krawędzi między każdą parą wierzchołków $p = 0.5$ w zależności od liczby wierzchołków n Dla małych grafów najlepszy jest algorytm SLF, a dla większych grafów (od 500 wierzchołków) najlepszy jest algorytm GIS.

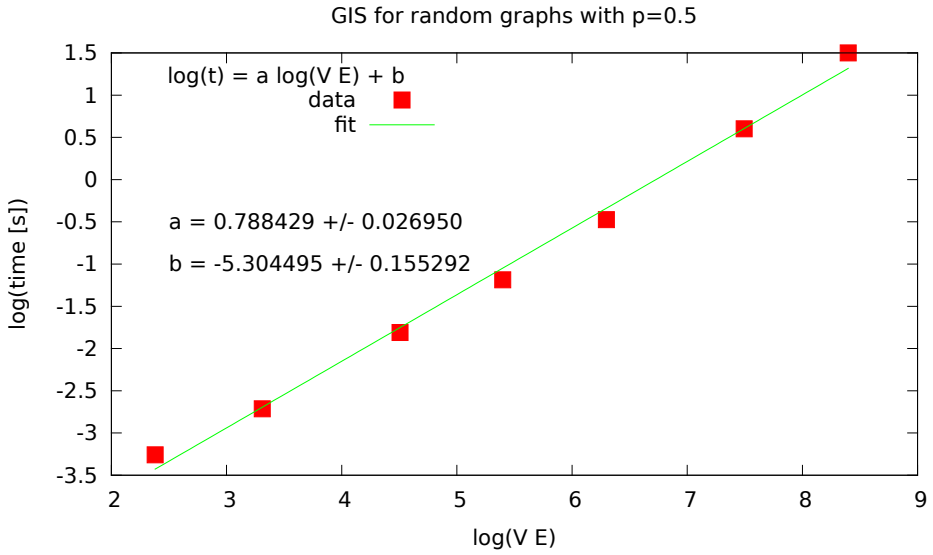
n	**LF**	**SL**	**SLF**	**GIS**	**RLF**
10	3.3	3.1	3.2	3.6	3.1
20	5.4	5.0	4.7	5.7	4.9
50	10.9	11.2	10.0	11.8	10.1
100	19.0	19.6	17.2	18.9	16.8
200	32.7	33.3	30.0	30.5	29.0
500	68.2	69.8	64.6	61.6	62.2
1000	122.3	123.5	114.6	107.1	110.6



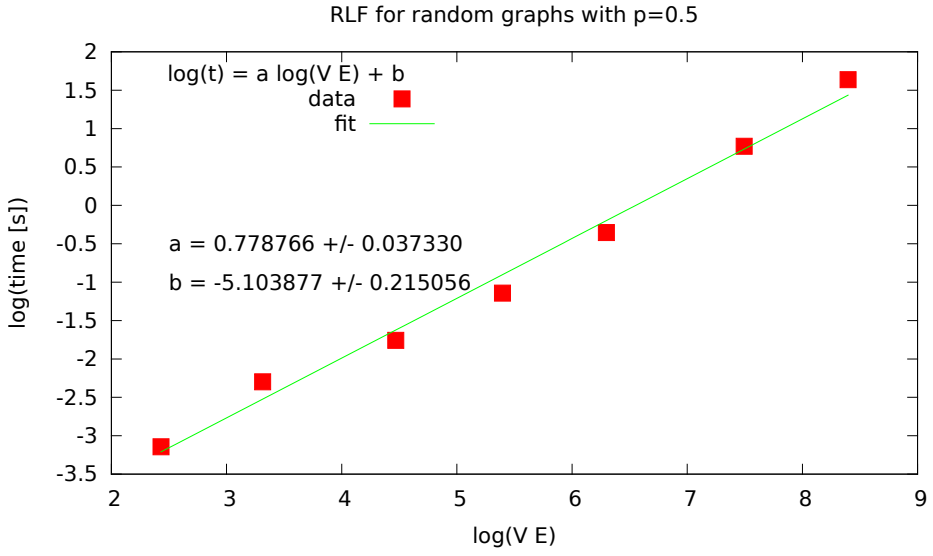
Rysunek C.1. Wykres wydajności algorytmu kolorowania wierzchołków z twierdzenia Brooksa dla grafów pełnych bez jednej krawędzi. Współczynnik a bliski 2 potwierdza złożoność $O(V^2)$.



Rysunek C.2. Wykres wydajności algorytmu kolorowania wierzchołków z twierdzenia Brooksa dla kubicznych grafów Halina (grafy planarne 3-spójne). Współczynnik a bliski 1 pokazuje szybszą pracę niż $O(V^2)$, nawet liniową $O(V)$.



Rysunek C.3. Wykres wydajności algorytmu GIS kolorowania wierzchołków dla grafów przypadkowych z $p = 0.5$. Współczynnik a bliski 0.8 potwierdza złożoność $O(VE)$.

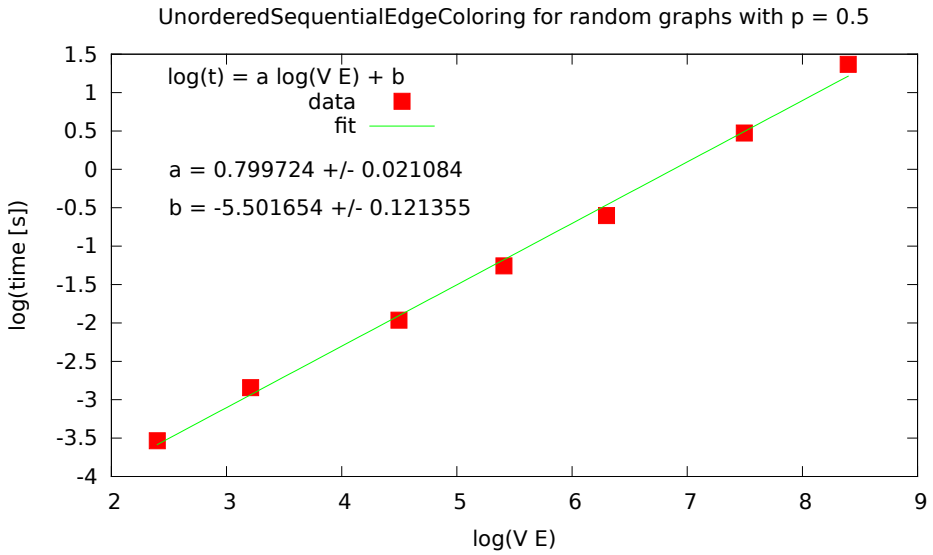


Rysunek C.4. Wykres wydajności algorytmu RLF kolorowania wierzchołków dla grafów przypadkowych z $p = 0.5$. Współczynnik a bliski 0.8 potwierdza złożoność $O(VE)$.

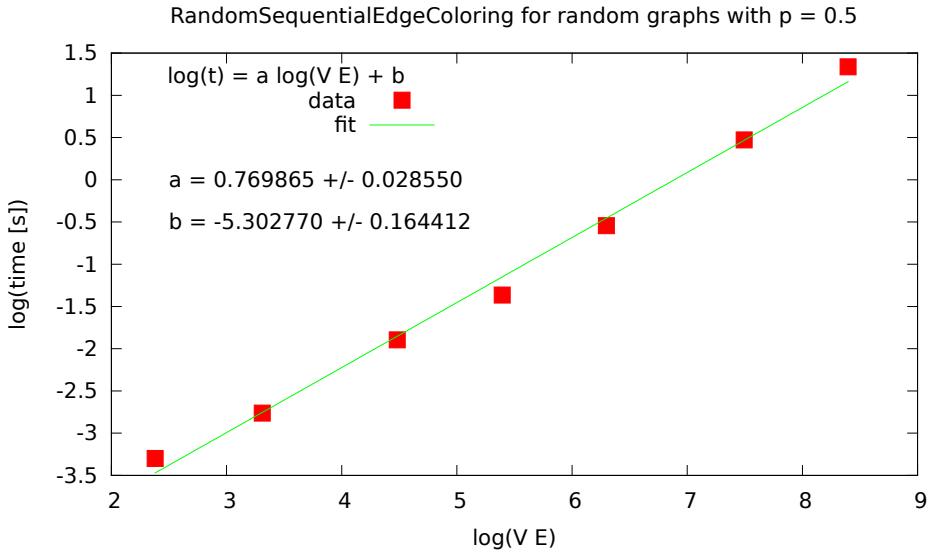
D. Testy kolorowania krawędzi

Testowanie algorytmów US, RS, BFS kolorowania krawędzi. Testy potwierdzają, że złożoność tych algorytmów nie przekracza $O(VE)$. Testy wykonano dla grafów przypadkowych z $p = 0.5$. Liczba przydzielonych kolorów czasem przekracza $\Delta + 1$. Wyniki testów są przedstawione na rysunkach D.1, D.2, D.3.

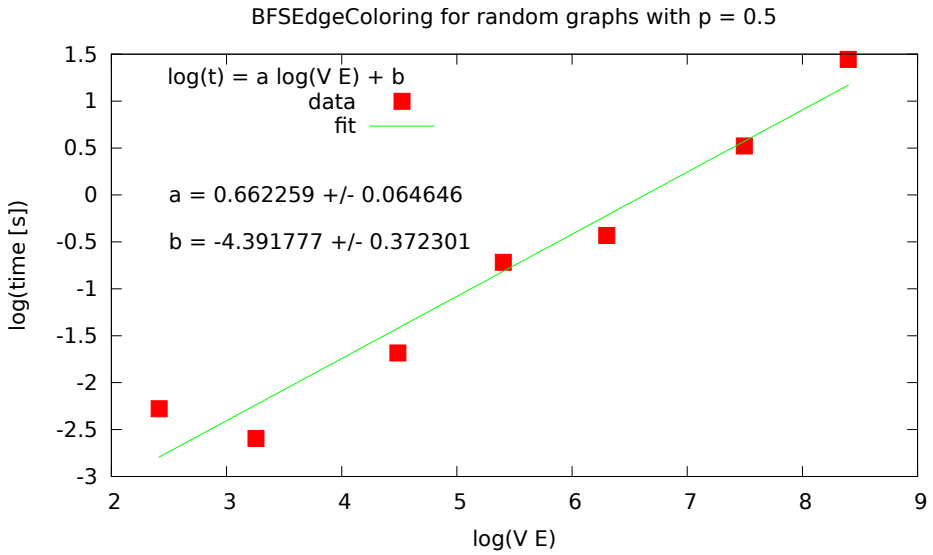
Testowanie algorytmu NTL. Liczba przydzielonych kolorów to Δ lub $\Delta + 1$. Dla grafów gęstych potwierdza się złożoność $O(VE)$ (grafy pełne, grafy przypadkowe). Dla grafów rzadkich, np. planarnych, algorytm pracuje znacznie szybciej. Przykładowo dla sieci trójkątnej testy pokazują złożoność rzędu $O(V)$. Wyniki dla grafów przypadkowych z $p = 0.5$ przedstawia rysunek D.4.



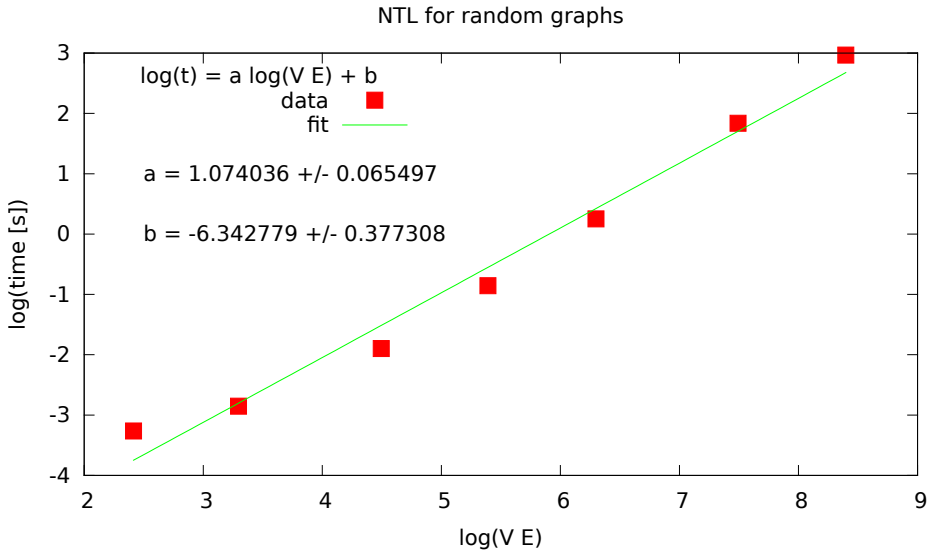
Rysunek D.1. Wykres wydajności algorytmu US kolorowania krawędzi. Współczynnik a poniżej 1 sugeruje zależność $O(VE)$.



Rysunek D.2. Wykres wydajności algorytmu RS kolorowania krawędzi. Współczynnik a poniżej 1 sugeruje zależność $O(VE)$.



Rysunek D.3. Wykres wydajności algorytmu BFS kolorowania krawędzi. Współczynnik a poniżej 1 sugeruje zależność $O(VE)$.



Rysunek D.4. Wykres wydajności algorytmu NTL kolorowania krawędzi. Współczynnik a bliski 1 potwierdza zależność $O(VE)$.

Bibliografia

[1] Marek Kubale (red.), *Optymalizacja dyskretna: Modele i metody kolorowania grafów*, Wydawnictwa Naukowo-Techniczne, Warszawa, 2002.

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

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

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

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

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

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

[8] Wikipedia, Graph coloring, 2016, `https://en.wikipedia.org/wiki/Graph_coloring`.

[9] M. R. Garey, D. S. Johnson, L. Stockmeyer, *Some simplified NP-complete graph problems*, Theoretical Computer Science 1, 237-267 (1976).

[10] Wikipedia, Mycielskian, 2016, `https://en.wikipedia.org/wiki/Mycielskian`.

[11] Rowland Leonard Brooks, *On Coloring the Nodes of a Network*, Proc. Cambridge Philos. Soc. 37, 194-197 (1941).

[12] Wikipedia, Brooks' theorem, 2016, `https://en.wikipedia.org/wiki/Brooks'_theorem`.

[13] Laszlo Lovasz, *Three short proofs in graph theory*, Journal of Combinatorial Theory (B) 19, 269-271 (1975).

[14] L. S. Melnikov, V. G. Vising, *New Proof of Brooks' Theorem*, Journal of Combinatorial Theory 7, 289-290 (1989).

[15] Daniel W. Cranston, Landon Rabern, *Brooks' Theorem and Beyond*, arXiv:1403.0479 [math.CO].

[16] Neal Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas, *The four-colour theorem*, Journal of Combinatorial Theory, Series B 70, 2-44 (1997).

[17] Wikipedia, Independent set (graph theory), 2016, `https://en.wikipedia.org/wiki/Independent_set_(graph_theory)`.

[18] Wikipedia, Vertex cover, 2016, `https://en.wikipedia.org/wiki/Vertex_cover`.

[19] Wikipedia, Greedy coloring, 2016, `https://en.wikipedia.org/wiki/Greedy_coloring`.

[20] Wikipedia, Crown graph, 2016, `https://en.wikipedia.org/wiki/Crown_graph`.

[21] David W. Matula, Leland L. Beck, *Smallest-Last Ordering and Clustering*

and Graph Coloring Algorithms, Journal of the Associacion for Computing Machinery 30(3), 417-427 (1983).

[22] Frank Thomson Leighton, *A Graph Coloring Algorithm for Large Scheduling Problems*, Journal of Research of the National Bureau of Standards 84, 489-505 (1979).

[23] Wikipedia, Edge coloring, 2016, `https://en.wikipedia.org/wiki/Edge_coloring`.

[24] Wikipedia, Snark (graph theory), 2016, `https://en.wikipedia.org/wiki/Snark_(graph_theory)`.

[25] Richard Cole, John E. Hopcroft, *On edge coloring bipartite graphs*, SIAM Journal on Computing 11, 540-546 (1982).

[26] Daniel P. Sanders, Yue Zhao, *Planar Graphs of Maximum Degree Seven are Class I*, Journal of Combinatorial Theory, Series B 83, 201-212 (2001).

[27] Marek Chrobak, Takao Nishizeki, *Improved Edge-Coloring Algorithms for Planar Graphs*, Journal of Algorithms 11, 102-116 (1990).

[28] H. N. Gabow, T. Nishizeki, O. Kariv, D. Leven, O. Terada, *Algorithms for edge-coloring graphs*, TR-41/85, Department of Computer Science, Tel Aviv University (1985).

[29] Wikipedia, Matching (graph theory), 2016, `https://en.wikipedia.org/wiki/Matching_(graph_theory)`.

[30] Wikipedia, Bipartite graph, 2016, `https://en.wikipedia.org/wiki/Bipartite_graph`.

[31] Wikipedia, Edge cover, 2016, `https://en.wikipedia.org/wiki/Edge_cover`.

[32] T. Nishizeki, O. Terada, D. Leven, *Algorithms for edge-coloring graphs*, Dept. Elec. Comm. Tohoku University, Technical Report TRECIS 83001 (1983).

[33] Jayadev Misra, David Gries, *A Constructive Proof of Vizing's Theorem*, Information Processing Letters 41 (3), 131-133 (1992).

[34] Wikipedia, Misra & Gries edge coloring algorithm, 2016, `https://en.wikipedia.org/wiki/Misra_%26_Gries_edge_coloring_algorithm`.

[35] Wikipedia, Equitable coloring, 2016, `https://en.wikipedia.org/wiki/Equitable_coloring`.

[36] Wikipedia, Total coloring, 2016, `https://en.wikipedia.org/wiki/Total_coloring`.