Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Magdalena Stępień

Nr albumu: 1127094

Wybrane algorytmy dla grafów planarnych

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2019

1:6025178395

Oświadczenie autora pracy

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

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

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

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

Kraków, dnia Podpis kierującego pracą

2:1032787274

Pragnę złożyć najserdeczniejsze wyrazy wdzięczności Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za ogromne wsparcie, zaangażowane i wyrozumiałość, poświęcony czas, a także nieocenioną pomoc, bez których nie byłoby możliwe napisanie tej pracy.

3:6003594292

Streszczenie

W pracy przedstawiono implementację wybranych algorytmów dla grafów planarnych w języku Python. Poruszono trzy zagadnienia, dla których zawężenie się do grafów planarnych pozwoliło na stworzenie szybszych algorytmów niż dla ogólnych grafów.

Zaimplementowano i omówiono dwa algorytmy znajdujące minimalne drzewo rozpinające: algorytm Cheritona-Tarjana oraz zmodyfikowany algorytm Boruvki. Algorytmy te uzyskują w praktyce liniową złożoność czasową mimo pewnych uproszczeń w implementacji. W pracy przedstawiono również algorytm 5-kolorowania wierzchołków grafu planarnego działający w czasie liniowym. Algorytm ten wykorzystuje lemat dotyczący wierzchołków stopnia 5. Ostatnie zagadnienie dotyczy grafów szeregowo-równoległych skierowanych (dsp-grafów). Podano algorytm rozpoznawania dsp-grafów, oraz stworzono generatory przypadkowych dsp-grafów i dsp-drzew.

Przygotowany kod źródłowy został sprawdzony z użyciem testów jednostkowych języka Python. Sprawdzono również praktyczną złożoność obliczeniową i skalowalność algorytmów.

Słowa kluczowe: grafy planarne, minimalne drzewo rozpinające, kolorowanie grafu, grafy szeregowo-równoległe

4:2637920024

English title: Selected algorithms for planar graphs

Abstract

Python implementation of selected graph algorithms for planar graphs is presented in this document. It contains three topics where a restriction to planar graphs allowed the creation of faster algorithms then in the general case.

Two algorithms finding the minimum spanning tree have been implemented and discussed: the Cheriton-Tarjan algorithm and the modified Boruvka algorithm. These algorithms achieve the approximate linear time complexity despite some simplications. The next one is the algorithm for 5-vertex-coloring of planar graphs. This algorithm achieves linear time complexity thanks to the properties of 5-degree vertices. The last topic deals with directed series-parallel graphs (dsp-graphs). The algorithm for recognizing dsp-graphs is prepared. Generators for random dsp-graphs and random dsp-trees are also built.

The Python unit testing tools were used to check the source code. The real computational complexity and scalability were also tested.

Keywords: planar graphs, minimum spanning tree, graph coloring, series--parallel graphs

5:2855920412

Spis treści

Spis rysunków . . . 3
Listings . . . 4
1. Wstęp . . . 5
2. Teoria grafów . . . 7
2.1. Grafy . . . 7
2.2. Spójność . . . 8
2.3. Planarność . . . 8
2.4. Kolorowanie wierzchołków grafu planarnego . . . 10
2.5. Grafy szeregowo-równoległe skierowane . . . 11
3. Implementacja grafów . . . 15
3.1. Struktury danych . . . 15
3.2. Obliczenia . . . 16
4. Algorytmy . . . 18
4.1. Minimalne drzewo rozpinające dla grafu planarnego . . . 18
4.1.1. Algorytm Cheritona-Tarjana . . . 18
4.1.2. Zmodyfikowany algorytm Borůvki . . . 20
4.2. Kolorowanie wierzchołków grafu planarnego . . . 21
4.3. Grafy szeregowo-równoległe skierowane . . . 24
4.3.1. Generator dsp-grafu . . . 24
4.3.2. Znajdowanie drzewa szeregowo-równoległego . . . 26
4.3.3. Generator dsp-drzewa . . . 28
5. Podsumowanie . . . 30
A. Testy algorytmów . . . 31
A.1. Testy minimalnego drzewa rozpinającego . . . 31
A.2. Testy kolorowania wierzchołków grafu planarnego . . . 31
A.3. Testy grafów szeregowo-równoległych skierowanych . . . 34
Bibliografia . . . 38

6:1115721273

Spis rysunków

2.1. Kompozycja równoległa. 12
2.2. Kompozycja szeregowa. 12
2.3. Reguły z definicji III. 13
2.4. Podstawowe grafy szeregowo-równoległe dla n wierzchołków. 14
2.5. Graf acykliczny skierowany (dag), ale nie szeregowo-równoległy. . . . 14

A.1. Wydajność algorytmu Cheritona-Tarjana dla sieci kwadratowej. . . . 32
A.2. Wydajność algorytmu Cheritona-Tarjana dla sieci trójkątnej. 32
A.3. Wydajność zmodyfikowanego algorytmu Borůvki dla sieci kwadratowej. 33
A.4. Wydajność zmodyfikowanego algorytmu Borůvki dla sieci trójkątnej. . 33
A.5. Wydajność algorytmu kolorowania dla antiprism. 35
A.6. Wydajność algorytmu kolorowania dla sieci trójkątnej. 35
A.7. Wydajność algorytmu kolorowania dla grafu dzielonego. 36
A.8. Wydajność algorytmu kolorowania dla grafu 3-antypryzma. 36
A.9. Wydajność algorytmu rozpoznawania dsp-grafów. 37

7:3818576119

Listings

4.1 Moduł cheritontarjan. 19
4.2 Moduł boruvkaplanar. 20
4.3 Moduł nodecolorplanar2. 22
4.4 Moduł make_dspgraph. 25
4.5 Moduł find_dsptree. 26
4.6 Moduł make_dsptree. 28

8:7949000515

1. Wstęp

Tematem niniejszej pracy są grafy planarne, czyli takie grafy, które można narysować na płaszczyźnie bez wzajemnego przecinania się krawędzi [1]. Planarność jest wewnętrzną cechą danego grafu, niezależną od sposobu narysowania grafu. Można narysować graf planarny z przecinającymi się krawędziami, ale to nie przekreśla jego planarności. W kontekście planarności na ogół rozważa się grafy nieskierowane proste, bez pętli i krawędzi równoległych (wielokrotnych).

Grafy planarne w naturalny sposób pojawiają się w zastosowaniach, tak opisuje się mapy z granicami państw, województw, połączenia drogowe, kolejowe, rzeki, kanały wodne, układy elektroniczne, itp. Grafy planarne można rozpoznać w czasie liniowym, choć znane algorytmy rozpoznawania są skomplikowane. Warto przypomnieć kilka interesujących właściwości grafów planarnych. Oznaczmy graf $G = (V, E)$, gdzie V to zbiór wierzchołków grafu, a E to zbiór krawędzi grafu, $n = |V|$, $m = |E|$.

- Grafy planarne są rzadkie, $m \leqslant 3n - 6$.
- Grafy planarne są 4-kolorowalne wierzchołkowo (twierdzenie o czterech barwach). Algorytm kolorowania jest skomplikowany.
- Graf planarny narysowany na płaszczyźnie to graf płaski, który dzieli płaszczyznę na spójne obszary nazywane *ścianami*. Do zbioru ścian F należy też dokładnie jeden obszar nieograniczony, nazywany *ścianą zewnętrzną*. *Wzór Eulera* mówi, że

$$n - m + f = k + 1, \tag{1.1}$$

gdzie k to liczba składowych spójnych grafu, $f = |F|$ to liczba ścian.

Pewne problemy z teorii grafów można rozwiązać szybciej dla grafów planarnych, niż w przypadku ogólnych grafów. Jednym z powodów jest mała liczba krawędzi rzędu $O(n)$. Innym powodem jest możliwość szybkiego podziału grafu planarnego na trzy części A, B i S (separator), gdzie A i B zawierają pewien procent wierzchołków oryginalnego grafu, a separator jest rzędu $O(\sqrt{n})$ (twierdzenie o planarnym separatorze). Czasem w literaturze dodaje się nieformalnie, że w grafach planarnych lokalne zmiany nie mogą mieć zbyt dużego efektu globalnego. Celem niniejszej pracy jest przedstawienie wybranych algorytmów wykorzystujących cechy grafów planarnych.

Algorytmy zostaną zaimplementowane w języku Python [2], ponieważ jego czytelna składnia i bogata biblioteka standardowa pozwalają uzyskać zwięzły, przejrzysty i wydajny kod źródłowy. Napisane programy wzbogacą pakiet graphtheory rozwijany w Instytucie Fizyki UJ [3].

Jednym ze znanych problemów jest znajdowanie największej kliki w grafie, czyli podgrafu pełnego K_n danego grafu. Z twierdzenia Kuratowskiego wiemy,

9:2032838374

że grafy planarne nie zawierają podgrafu K_5. Z tego wynika, że największą kliką może być K_4 lub K_3, a to można sprawdzić w czasie wielomianowym.

Inne problemy rozważane dla grafów planarnych to znajdowanie minimalnego drzewa rozpinającego, znajdowanie najkrótszych ścieżek, problem przepływu maksymalnego w planarnych sieciach przepływowych.

Znaleziono ogólną technikę, która może być użyta do otrzymania schematów aproksymacyjnych dla różnych problemów NP-zupełnych w grafach planarnych [4]. Technika opiera się na dekompozycji grafu planarnego na podgrafy k-zewnętrznoplanarne. Dla takich grafów można znaleźć optymalne rozwiązania trudnych problemów w czasie liniowym przy pomocy programowania dynamicznego. Podana technika stosuje się m.in. do problemu największego zbioru niezależnego, najmniejszego zbioru dominującego, najmniejszego pokrycia wierzchołkowego.

Organizacja niniejszej pracy jest następująca. Rozdział 1 zawiera wprowadzenie do tematyki pracy. Rozdział 2 zawiera definicje pojęć i twierdzenia z teorii grafów, potrzebne do zrozumienia przedstawianych algorytmów. Rozdział 3 prezentuje implementację grafów w języku Python, wykorzystywaną przy budowie algorytmów. Rozdział 4 omawia implementacje algorytmów, które są głównym wynikiem niniejszej pracy. Rozdział 5 stanowi podsumowanie pracy. W dodatku A pokazane są wyniki testów wydajnościowych dla algorytmów zaimplementowanych w pracy.

10:9386916561

2. Teoria grafów

Rozdział zawiera podstawowe definicje i twierdzenia z teorii grafów wykorzystywane w niniejszej pracy [5], [6].

2.1. Grafy

Graf: Graf jest definiowany jako para uporządkowana $G = (V, E)$, czyli jest to struktura składająca się z niepustego zbioru wierzchołków V oraz zbioru krawędzi E.

Wierzchołki i krawędzie: Wierzchołki są elementami zbioru wierzchołków V, często opisywane jako cyfry lub litery i przedstawiane na rysunku jako punkty lub kółka z etykietami lub bez. Krawędzie są elementami zbioru krawędzi E, najczęściej zapisywane są jako pary wierzchołków. Krawędź określa relację jaka zachodzi pomiędzy poszczególnymi wierzchołkami i przedstawiana jest jako linia łącząca te wierzchołki (graf nieskierowany), albo linia zakończona strzałką (graf skierowany). Liczbę wierzchołków i krawędzi zapisujemy odpowiednio jako $n = |V|$ i $m = |E|$.

Graf pusty i pełny: Graf pusty to taki graf, który zawiera wierzchołki, ale którego zbiór krawędzi E jest pusty, czyli nie posiada żadnych krawędzi. Natomiast graf pełny z n wierzchołkami jest grafem nieskierowanym, w którym wszystkie wierzchołki są ze sobą parami połączone, a oznaczany jest jako K_n.

Graf prosty i multigraf: Petla jest to krawędź łącząca wierzchołek z nim samym. Krawędzie wielokrotne łączą te same dwa wierzchołki. Graf prosty nie zawiera żadnych pętli ani krawędzi wielokrotnych. Natomiast multigraf może zawierać pętle lub też krawędzie wielokrotne. W literaturze można spotkać różne definicje, samo słowo *graf* może oznaczać zarówno graf prosty jaki i multigraf u różnych autorów.

Graf skierowany i nieskierowany: Graf skierowany (digraf) to graf, który zawiera krawędzie skierowane, czyli uporządkowane pary wierzchołków. Dla danej krawędzi jest określony wierzchołek początkowy i wierzchołek końcowy. W grafie nieskierowanym kolejność wierzchołków nie ma znaczenia.

Graf ważony: Jest to graf, którego krawędzie mają przyporządkowane wartości liczbowe, czyli wagi. Czasem taka krawędź jest opisana jako uporządkowana trójka, w której kolejno występuje wierzchołek początkowy, wierzchołek końcowy, a trzeci element to waga krawędzi.

11:9411515199

Graf dwudzielny i dwudzielny pełny: Graf nieskierowany jest dwudzielny wtedy, gdy jego zbiór wierzchołków da się podzielić na dwa rozłączne zbiory, dla których nie istnieje krawędź łącząca wierzchołki z tego samego zbioru. W przypadku grafu dwudzielnego pełnego istnieje krawędź łącząca każdą parę wierzchołków z różnych zbiorów. Graf dwudzielny pełny oznaczany jest jako $K_{p,q}$, gdzie p i q oznaczają liczności wierzchołków w obu zbiorach.

2.2. Spójność

Stopień wierzchołka: Jest to atrybut danego wierzchołka v w grafie nieskierowanym oznaczany jako $\deg(v)$. Jest to liczba wszystkich krawędzi incydentnych z danym wierzchołkiem. W grafie skierowanym dla danego wierzchołka określa się stopień wejściowy i stopień wyjściowy, czyli odpowiednio liczbę krawędzi wchodzących do wierzchołka i wychodzących z niego.

Ścieżka: Jest to ciąg wierzchołków $(v_0, v_1, ..., v_k)$ od wierzchołka początkowego v_0 do wierzchołka końcowego v_k taki, że istnieje krawędź pomiędzy kolejnymi wierzchołkami. Czasem przez ścieżkę rozumie się zbiór krawędzi, które łączą kolejne wierzchołki tego ciągu $(v_0v_1, v_1v_2, ..., v_{k-1}v_k)$. W grafie może istnieć kilka różnych ścieżek między wybranymi, krańcowymi wierzchołkami. Ścieżka jest *prosta*, jeśli każdy wierzchołek lub krawędź występuje na ścieżce tylko raz.

Cykl: To ścieżka zamknięta, której wierzchołek początkowy i końcowy są równe $v_0 = v_k$, czyli zaczyna się i kończy w tym samym wierzchołku oraz przechodzi przez co najmniej jeszcze jeden inny wierzchołek.

Graf cykliczny i acykliczny: Graf cykliczny jest to graf, w którym występuje co najmniej jeden cykl. W grafie acyklicznym cykle nie występują.

Spójność: Graf nieskierowany jest spójny, jeśli każde dwa wierzchołki w grafie są połączone pewną ścieżką.

Drzewo: Jest to graf prosty, nieskierowany, spójny i acykliczny.

Drzewo rozpinające: Drzewem rozpinającym (ang. *spanning tree*) jest drzewo zawierające wszystkie wierzchołki grafu oraz krawędzie, które zawierają się w zbiorze krawędzi grafu. Dla grafu ważonego można uzyskać minimalne drzewo rozpinające (ang. *minimum spanning tree*), którego suma wag krawędzi jest najmniejsza.

2.3. Planarność

Minory: Graf nieskierowany H jest *minorem* grafu G, jeżeli możemy otrzymać H z G przez usuwanie krawędzi, usuwanie wierzchołków izolowanych, oraz przez ściąganie krawędzi (ang. *contracting edges*) [7].

12:2246656683

Kryteria planarności: Graf planarny można narysować na płaszczyźnie tak, że jego krawędzie nie będą się przecinać ze sobą. W literaturze znanych jest kilka kryteriów planarności grafów. Najbardziej znane to twierdzenie Kuratowskiego (podgrafy homeomorficzne) i twierdzenie Wagnera (minory).

Minory zabronione: W teorii grafów rozważa się rodziny grafów, które nie zawierają pewnej skończonej liczby *minorów zabronionych* (ang. *forbidden minors*), co zapisuje się jako $\mathrm{Ex}(H_1, \ldots, H_k)$. Wiele ważnych rodzin grafów posiada taką charakteryzację. Warto podać kilka przykładów: drzewa $\mathrm{Ex}(K_3)$, grafy zewnętrznoplanarne $\mathrm{Ex}(K_4, K_{2,3})$, grafy szeregowo-równoległe $\mathrm{Ex}(K_4)$, **grafy planarne** $\mathrm{Ex}(K_5, K_{3,3})$.

Uogólnieniem charakteryzacji przez minory zabronione jest charakteryzacja przez *zabronione podstruktury*, np. podgrafy, podgrafy indukowane, podgrafy homeomorficzne. Przykładowo w grafach dwudzielnych nie mogą występować cykle o nieparzystej długości. W grafach cięciwowych nie mogą występować cykle indukowane o długości większej niż trzy. W wielu wypadkach można w czasie wielomianowym wykryć istnienie zabronionej podstruktury w danym grafie, co jest wykorzystywane w algorytmach rozpoznających grafy z danej rodziny.

Rodziny grafów zamknięte ze względu na minory: Wiele rodzin grafów ma właściwość, że każdy minor grafu z tej rodziny też należy do tej rodziny. Mówimy, że rodzina jest zamknięta ze względu na minory (ang. *minor-closed graph family*). Przykładem są grafy planarne, przy tworzeniu minorów musimy jedynie unikać multigrafów przez odrzucanie pętli i krawędzi wielokrotnych.

Z drugiej strony, rodzina grafów zamknięta ze względu na minory może być opisana przez skończoną liczbę minorów zabronionych, co wynika z twierdzenia Robertsona-Seymoura [8].

Izomorfizm grafów planarnych: Dla dwóch grafów planarnych o n wierzchołkach można w czasie $O(n)$ stwierdzić, czy te grafy są izomorficzne, czy nie [9].

Najkrótsze ścieżki w planarnym dagu: W dagu, czyli grafie skierowanym acyklicznym, można w czasie $O(n + m)$ znaleźć najkrótsze ścieżki z wierzchołka s do innych wierzchołków grafu [13]. Wykorzystuje się przy tym sortowanie topologiczne wierzchołków. W przypadku planarnego dagu otrzymamy algorytm działający w czasie $O(n)$.

Najkrótsze ścieżki w grafach planarnych: W przypadku nieujemnych wag krawędzi dla ogólnych grafów skierowanych mamy algorytm Dijkstry, którego typowe implementacje mają złożoność $O(n^2)$ lub $O(m \log n)$. Algorytm znajduje najkrótsze ścieżki z wierzchołka s do innych wierzchołków grafu. Dla grafów planarnych w roku 1997 podano algorytm działający w czasie $O(n)$, który rekurencyjnie dzieli graf na mniejsze obszary [10].

Jeżeli dozwolone są ujemne wagi krawiędzi, to dla ogólnych grafów mamy algorytm Bellmana-Forda działający w czasie $O(nm)$ [dla grafu planarnego otrzymamy złożoność $O(n^2)$]. Ujemne cykle są zabronione. Istnieją inne

13:2459347145

algorytmy, których złożoność zależy jawnie od ujemnej wagi krawędzi. Dla grafów planarnych podano m.in. algorytm działający w czasie $O(n^{3/2})$, wykorzystujący planarne separatory [11]. Algorytm Fredericksona działa w czasie $O(n\sqrt{\log n})$ [12].

Przepływy w grafach planarnych: *Sieci przepływowe* to grafy skierowane ważone posiadające pewne dodatkowe właściwości [13]. W sieciach przepływowych szuka się *maksymalnego przepływu* od źródła do ujścia. W zastosowaniach często sieć przepływowa odpowiada grafowi planarnemu skierowanemu, dlatego badano problem maksymalnego przepływu w kontekście grafów planarnych (ang. *maximum st-flow problem*).

Dla ogólnych grafów szybkie algorytmy mają wydajność $O(n^3)$, a dla grafów planarnych uzyskano złożoność $O(n \log n)$ [14]. Jeżeli źródło s i ujście t łączy pojedyncza krawędź sąsiadująca ze ścianą zewnętrzną (ang. *st-planar graphs*), to istnieje algorytm $O(n)$ znajdujący maksymalny przepływ. Do planarnych sieci przepływowych należą grafy szeregowo-równoległe skierowane [15].

2.4. Kolorowanie wierzchołków grafu planarnego

Kolorowanie wierzchołków grafu polega na przyporządkowaniu wierzchołkom kolorów (liczb lub etykiet tekstowych) w taki sposób, aby każda krawędź w grafie miała końce w różnych kolorach. Zależy nam na wykorzystaniu jak najmniejszej liczby kolorów.

Problem kolorowania wierzchołków grafu planarnego ma długą historię. Istnieje lemat mówiący o tym, że każdy graf planarny ma wierzchołek o stopniu co najwyżej 5 [16]. Dzięki temu algorym SL (ang. *Smallest Last*) może w czasie $O(n)$ pokolorować wierzchołki grafu planarnego sześcioma kolorami. Implementacja algorytmu SL jest już dostępna w pakiecie graphtheory.

Z drugiej strony słynne twierdzenie o czterech barwach mówi, że wystarczą cztery kolory. Istnieje wiele dowodów tego twierdzenia, ale zwykle znajduje się dużą liczbę konfiguracji, które należy po kolei przeanalizować, najczęściej robi się to komputerowo. Jest to skomplikowane, więc skupiliśmy się na kolorowaniu grafu planarnego pięcioma kolorami [16].

Lemat: Niech G będzie grafem planarnym, w którym każdy wierzchołek ma stopień co najwyżej 5. Wtedy istnieje wierzchołek stopnia 5 mający dwóch niepołączonych sąsiadów o stopniach nie większych niż 11 [16]. Lemat zostanie wykorzystany w algorytmie 5-kolorowania wierzchołków grafu planarnego.

W ogólnym zarysie działanie algorytmu 5-kolorowania wierzchołków można podzielić na dwie fazy. Pierwsza faza polega na zredukowaniu grafu do postaci 5-wierzchołkowej, jeśli graf zawiera więcej niż 5 wierzchołków, a następnie pokolorowaniu każdego z wierzchołków na inny kolor. W drugiej fazie natomiast graf ten zostaje odtworzony do postaci pierwotnej wraz z jednoczesnym pokolorowaniem pozostałych wierzchołków podczas tego procesu.

14:1135928758

W bardziej szczegółowym ujęciu wygląda to tak, że w pierwszej fazie powtarzany jest proces redukcji wierzchołków w grafie dotąd, aż zostanie ich maksymalnie 5. W pierwszej kolejności do zredukowania wybierane są wierzchołki o stopniu mniejszym lub równym 4. Natomiast jeśli takich wierzchołków nie ma, dopiero wtedy do redukcji wybierany jest wierzchołek o stopniu 5 po spełnieniu warunku o jego dwóch niepołączonych ze sobą sąsiadach. Po każdej redukcji graf jest aktualizowany, zatem istnieje możliwość, że w danym grafie wierzchołek o stopniu 5 nigdy nie zostanie wybrany do redukcji, nawet jeśli taki w grafie istnieje, ponieważ zredukowanie wierzchołków o mniejszym stopniu będzie wystarczające.

W fazie drugiej następuje kolorowanie pozostałych wierzchołków podczas ich odtwarzania. Do grafu zwracane są wierzchołki wcześniej usunięte podczas fazy redukowania. W momencie przywracania do grafu są kolorowane na inny kolor niż wierzchołki, do których są przyłączane. Procedura ta jest powtarzana dotąd, aż cały graf zostanie odbudowany do pierwotnej postaci, a wszystkie wierzchołki pokolorowane.

2.5. Grafy szeregowo-równoległe skierowane

W pracy [17] przedstawiono właściwości grafów szeregowo-równoległych nieskierowanych (sp-grafów). Są to grafy planarne tworzone rekurencyjnie za pomocą trzech operacji: szeregowej, równoległej i *jackknife*. Zbudowano generatory przypadkowych sp-grafów i sp-drzew, przygotowano algorytm rozpoznawania sp-grafów.

Okazuje się, że po pewnych modyfikacjach podanego kodu źródłowego można opisać grafy szeregowo-równoległe skierowane (dsp-grafy). Należy przede wszystkim usunąć operację *jackknife*, aby mieć tylko jeden wierzchołek bez krawędzi wychodzących. Krawędzie dsp-grafu są skierowane i dla wierzchołków trzeba operować stopniem wejściowym i stopniem wyjściowym, krawędziami wchodzącymi i wychodzącymi.

Warto zwrócić uwagę, że każdy dsp-graf jest dagiem, ale nie na odwrót (przykład na rysunku 2.5). Można wykorzystać generatory dsp-grafów do tworzenia przykładowych planarnych sieci przepływowych, trzeba jeszcze ustalić sposób losowania wag (pojemności) krawędzi.

Graf szeregowo-równoległy skierowany (ang. *directed series-parallel graph*) można opisać za pomocą kilku równoważnych definicji.

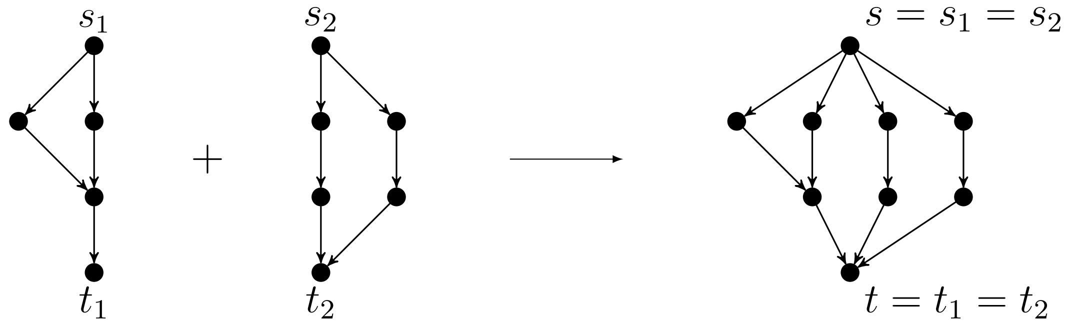
Definicja I: Graf szeregowo-równoległy jest grafem o dwóch końcówkach (ang. *two-terminal series-parallel graph*), który posiada dwa krańcowe wierzchołki: źródło s (ang. *source*) oraz ujście t (ang. *sink*). Źródło ma tylko krawędzie wychodzące, a ujście tylko krawędzie wchodzące. Taki graf jest zdefiniowany rekurencyjnie przez następujące połączenia:

— **połączenie równoległe** – powstaje poprzez połączenie dwóch dsp-grafów o źródłach s_1, s_2 i ujściach t_1, t_2 w taki sposób, że źródła są połączone w jedno źródło $s = s_1 = s_2$ i ujścia są połączone w jedno ujście $t = t_1 = t_2$.

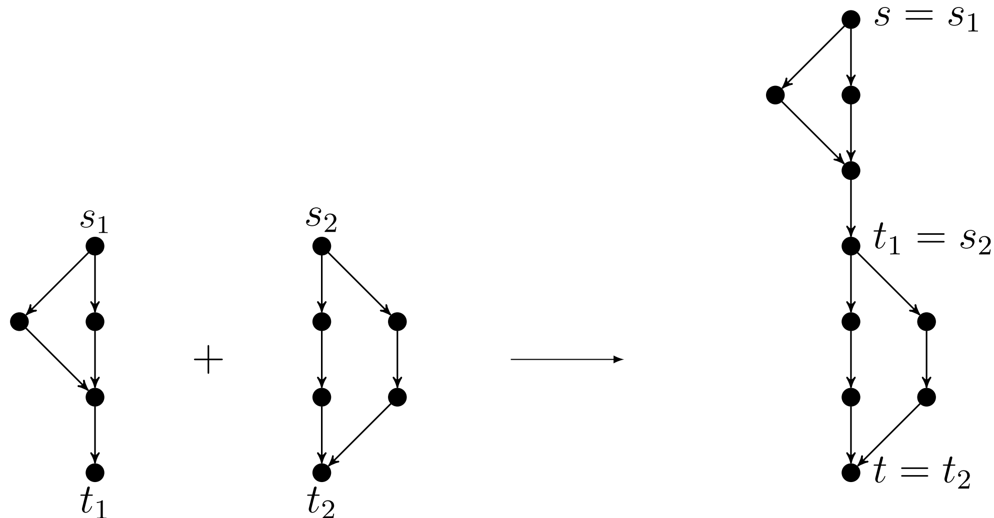
15:6986269482

— **połączenie szeregowe** – powstaje poprzez połączenie dwóch dsp-grafów o źródłach s_1, s_2 i ujściach t_1, t_2 w taki sposób, że źródłem jest $s = s_1$, ujściem jest $t = t_2$, a ujście t_1 i źródło s_2 są połączone $t_1 = s_2$.

Graf szeregowo-równoległy skierowany może być konstruowany przez sekwencję kompozycji szeregowych i równoległych startującej od pojedynczej krawędzi skierowanej st, która jest najprostszym dsp-grafem.



Rysunek 2.1. Kompozycja równoległa.



Rysunek 2.2. Kompozycja szeregowa.

Definicja II: Graf jest szeregowo-równoległy, jeśli może zostać sprowadzony do postaci pojedynczej krawędzi skierowanej st przy pomocy sekwencji dwóch operacji:

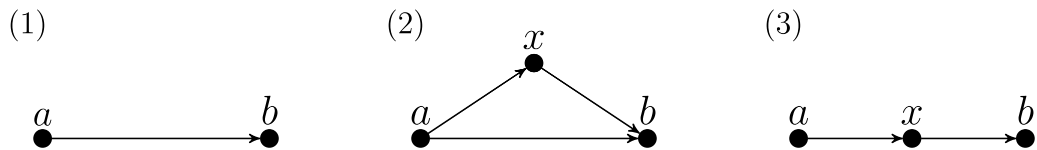
— **operacja równoległa** - para równoległych krawędzi zostaje zastąpiona jedną krawędzią, która łączy ich wierzchołki końcowe.
— **operacja szeregowa** - para krawędzi połączonych z wierzchołkiem posiadającym jedną krawędź wychodzącą i jedną wchodzącą, który zatem nie jest źródłem ani ujściem, zostaje zastąpiona jedną krawędzią.

16:5007043016

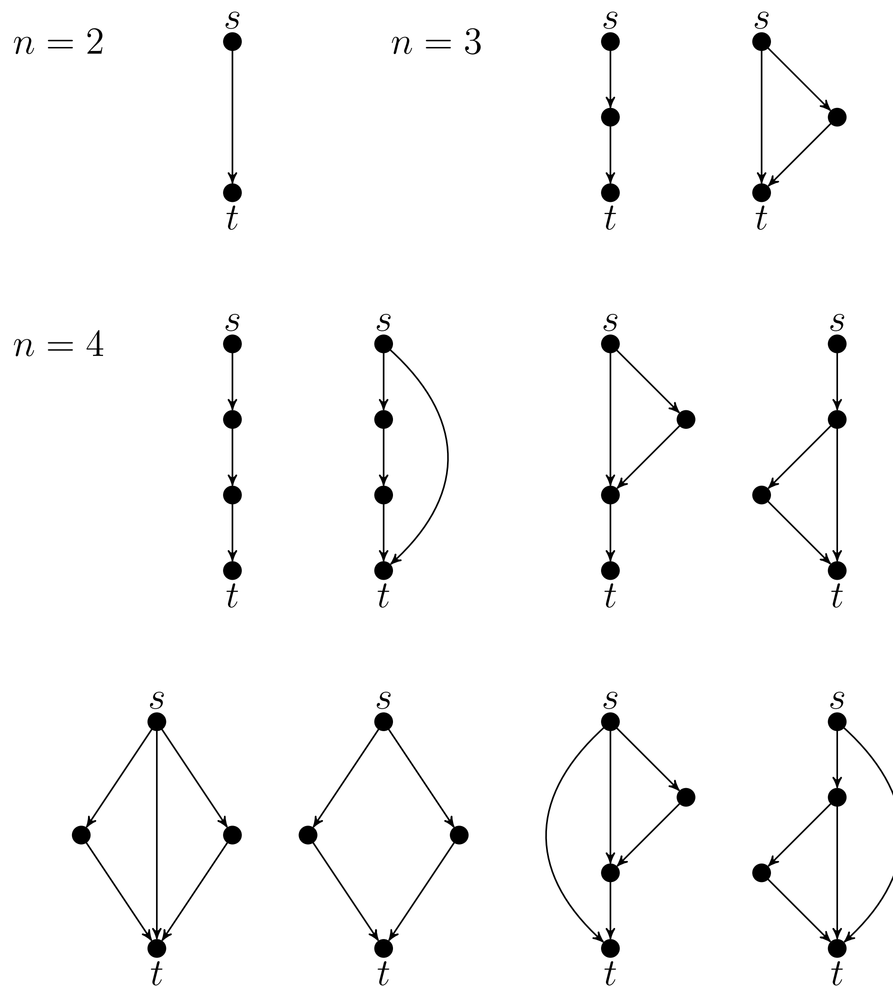
Dwie powyższe definicje mogą na pewnych etapach prowadzić do powstania multigrafów. Z tego względu do operacji na grafach prostych bardziej przydatna będzie trzecia definicja.

Definicja III: Graf jest szeregowo-równoległy, jeśli może być utworzony na podstawie następujących reguł:

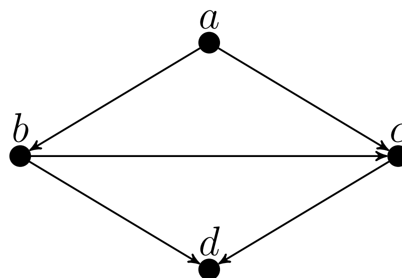
— (1) pojedyncza krawędź skierowana od źródła s do ujścia t tworzy podstawowy graf szeregowo-równoległy skierowany.
— (2) dodanie do pojedynczej krawędzi skierowanej ab krawędzi skierowanych ax i xb tworzy połączenie równoległe.
— (3) zastąpienie pojedynczej krawędzi skierowanej ab dwiema krawędziami skierowanymi ax i xb tworzy połączenie szeregowe.



Rysunek 2.3. Reguły z definicji III.



Rysunek 2.4. Podstawowe grafy szeregowo-równoległe dla n wierzchołków.



Rysunek 2.5. Graf acykliczny skierowany (dag), ale nie szeregowo-równoległy.

18:6462587765

3. Implementacja grafów

W rozdziale tym przedstawiono zastosowane struktury danych oraz przykładowe obliczenia realizowane przez algorytmy zawarte w niniejszej pracy.

3.1. Struktury danych

Algorytmy grafowe w niniejszej pracy korzystają głównie z klas Graph oraz Edge z pakietu graphtheory, które umożliwiają przejrzystą oraz optymalną reprezentację grafów. W tym celu klasa Graph jest rozbudowaną strukturą słownika, w którym klucze, czyli wierzchołki grafu, odwołują się do kolejnego słownika przechowującego sąsiadujące z nimi wierzchołki wraz z łączącymi je krawędziami.

Graph: Podstawowa klasa wykorzystywana na potrzeby każdego algorytmu. Każdy graf jest obiektem klasy Graph i służy do reprezentacji grafu za pomocą słownika wierzchołków oraz krawędzi będącymi obiektami klasy Edge. Zawiera informacje o tym czy graf jest skierowany oraz metody, które pozwalają na modyfikowanie grafu, w tym dodawanie, usuwanie wierzchołków lub krawędzi czy też przeszukiwanie wybranych elementów. Może być również wykorzystana do reprezentowania drzewa dla danego grafu.

Edge: Podstawowa klasa opisująca krawędzie grafu. Krawędzie są obiektami klasy Edge, które posiadają atrybuty source, target oraz weight. Oznaczają one odpowiednio wierzchołek początkowy, wierzchołek końcowy oraz wagę danej krawędzi.

UnionFind: Klasa realizująca strukturę zbiorów rozłącznych. Funkcja find() pozwala na określenie podzbioru, do którego przynależy dany wierzchołek i jeśli dwa różne wierzchołki należą do innych podzbiorów funkcja union() łączy je w jeden nowy podzbiór.

Node: Klasa stworzona na potrzeby przetwarzania grafów szeregowo-równoległych. Pomaga w rozpoznawaniu czy graf jest szeregowo-równoległy oraz umożliwia budowanie drzewa szeregowo-równoległego (dsptree).

Klasy algorytmów oraz funkcje: Algorytmy w kodzie reprezentowane są jako klasy. Każda z nich posiada swoje własne atrybuty oraz metodę run() służącą do uruchomienia algorytmu.

— CheritonTartjan: Algorytm Cheritona-Tarjana do znajdowania minimalnego drzewa rozpinającego.

19:6312912749

— BoruvkaPlanarMST: Zmodyfikowany algorytm Boruvki do znajdowania minimalnego drzewa rozpinającego.
— FiveColoringPlanarGraphs: Algorytm 5-kolorowania grafów planarnych.
— make_random_dspgraph(): funkcja tworząca graf szeregowo-równoległy.
— make_random_dsptree(): funkcja tworząca drzewo szeregowo-równoległe.
— find_dsptree(): funkcja znajdująca drzewo szeregowo-równoległe w danym grafie szeregowo-równoległym.

3.2. Obliczenia

Przykład 1: Obliczanie minimalnego drzewa rozpinającego dla grafu planarnego.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
# Tworzymy graf planarny wazony (graf krata).
>>> gf = GraphFactory(Graph)
>>> G = gf.make_grid(size=5)
# Algorytm Cheritona-Tarjana.
>>> from cheritontarjan import CheritonTarjan
>>> algorithm = CheritonTarjan(G)
>>> algorithm.run()
>>> algorithm.mst.show()   # MST jako graf
# Suma wag krawedzi nalezacych do MST.
>>> mst_weight = sum(edge.weight for edge in algorithm.mst.iteredges())
# Algorytm Boruvki dla grafu planarnego.
>>> from boruvkaplanar import BoruvkaPlanarMST
>>> algorithm = BoruvkaPlanarMST(G)
>>> algorithm.run()
>>> algorithm.mst.show()   # MST jako graf
```

Przykład 2: Kolorowanie wierzchołków grafu planarnego pięcioma kolorami.

```
# Wygenerowanie grafu planarnego o danym poziomie.
>>> from generate_planar_graph import GeneratePlanarGraph
>>> graph = GeneratePlanarGraph(level=10)
>>> G = graph.run()
# Kolorowanie wygenerowanego grafu planarnego
>>> from nodecolorplanar2 import FiveColoringPlanarGraph
>>> algorithm = FiveColoringPlanarGraph(G)
>>> algorithm.run()
>>> algorithm.color
```

Przykład 3: Obliczenia z grafami szeregowo-równoległymi skierowanymi.

```
# Utworzenie losowego grafu szeregowo-rownoleglego.
>>> from make_dspgraph import make_random_dspgraph
>>> G = make_random_dspgraph(10)
# Znalezienie drzewa szeregowo-rownoleglego w danym grafie.
>>> from find_dsptree import find_dsptree
```


20:1247021934

```
>>> T = find_dsptree(G)
# Utworzenie losowego drzewa szeregowo-rownoleglego.
>>> from make_dsptree import make_random_dsptree
>>> T = make_random_dsptree(10)
```

4. Algorytmy

Rozdział zawiera opis implementacji algorytmów dla grafów planarnych: minimalne drzewo rozpinające, 5-kolorowanie wierzchołków, grafy szeregowo-równoległe skierowane.

4.1. Minimalne drzewo rozpinające dla grafu planarnego

Pierwszy algorytm o wydajności $O(m \log \log n)$ podali Cheriton i Tarjan [18]. Na początku w każdym wierzchołku utworzona jest kolejka priorytetowa z krawędziami incydentnymi do wierzchołka. Na tym etapie MST składa się z izolowanych wierzchołków/komponentów. Następnie w wybranym komponencie znajduje się krawędź o najmniejszej wadze, która łączy różne komponenty. Krawędź zalicza się do MST, komponenty są łączone, ich kolejki priorytetowe z krawędziami również. Ten etap powtarza się $n-1$ razy. Kluczowe znaczenie dla szybkości algorytmu ma implementacja kolejek priorytetowych, z szybkim wybieraniem elementów i łączeniem dwóch kolejek.

Podejście wykorzystujące graf dualny [5] podał Matsui [19]. Algorytm znajduje jednocześnie dwa MST, jedno dla danego grafu planarnego, a drugie dla jego grafu dualnego.

MST dla grafu planarnego można znaleźć w czasie $O(n)$ za pomocą zmodyfikowanego algorytmu Borůvki [20]. Podobne podejście działa dla innych klas grafów zamkniętych ze względu na minory [21].

Testy implementacji algorytmów wyznaczających minimalne drzewo rozpinające znajdują się w dodatku A.1.

4.1.1. Algorytm Cheritona-Tarjana

Dane wejściowe: Graf planarny spójny ważony G.

Problem: Znajdowanie minimalnego drzewa rozpinającego dla G.

Dane wyjściowe: Minimalne drzewo rozpinające T.

Opis algorytmu: Nasza implementacja algorytmu Cheritona-Tarjana wykorzystuje strukturę zbiorów rozłącznych i dla prostoty standardową kolejkę priorytetową, zamiast specjalnej wersji z szybkim łączeniem kolejek. Okazało się, że w praktyce w dalszym ciągu uzyskuje się wydajność nieznacznie gorszą niż $O(n)$.

22:6888152756

Listing 4.1. Moduł cheritontarjan.

```
#!/usr/bin/python

from Queue import PriorityQueue
from unionfind import UnionFind

class CheritonTarjan:
    """Finding a minimum spanning tree for planar graphs."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.mst = self.graph.__class__(self.graph.v())
        for node in self.graph.iternodes():   # isolated nodes are possible
            self.mst.add_node(node)
        self._uf = UnionFind()   # komponenty jak dla Kruskala

    def run(self):
        """Finding MST."""
        pq = dict()   # kolejki priorytetowe w kazdym wierzcholku
        for node in self.graph.iternodes():
            self._uf.create(node)   # component T_u
            pq[node] = PriorityQueue()   # create PQ_u
            for edge in self.graph.iteroutedges(node):
                pq[node].put((edge.weight, edge))
        # Wszystkie 2E krawedzi bedzie roznych, bo sa skierowane.
        S = set(self.graph.iternodes())   # zbior komponentow
        for step in xrange(self.graph.v()-1):   # MST ma V-1 krawedzi
            # Select a component T_u.
            node = S.pop()
            S.add(node)   # chyba inny komponent moze zniknac
            while not pq[node].empty():
                _, edge = pq[node].get()
                source = self._uf.find(edge.source)
                target = self._uf.find(edge.target)
                if source != target:   # znalezione rozne komponenty
                    break
            # Merge T_u and T_v.
            self._uf.union(edge.source, edge.target)
            # Merge PQ_u and PQ_v.
            if source == self._uf.find(source): # zniknal target
                S.remove(target)
                while not pq[target].empty():
                    pq[source].put(pq[target].get())
            else: # zniknal source
                S.remove(source)
                while not pq[source].empty():
                    pq[target].put(pq[source].get())
            # Add edge to MST.
            self.mst.add_edge(edge)

    def to_tree(self):
        """Return MST as a graph."""
        return self.mst
```


23:1134336209

4.1.2. Zmodyfikowany algorytm Borůvki

Dane wejściowe: Graf planarny spójny ważony G.

Problem: Znajdowanie minimalnego drzewa rozpinającego dla G.

Dane wyjściowe: Minimalne drzewo rozpinające T.

Opis algorytmu: W oryginalnym algorytmie Borůvki w pierwszym kroku dla każdego wierzchołka wybiera się krawędź wychodzącą z niego o najmniejszej wadze. Wszystkie te krawędzie tworzą początkowy las. W następnych krokach wybiera się krawędzie o najmniejszej wadze łączące różne drzewa. W każdym kroku analizuje się wszystkie krawędzie grafu, maleje jedynie liczba rozłącznych drzew.

W zmodyfikowanym algorytmie Borůvki w każdym kroku redukujemy co najmniej o połowę liczbę krawędzi, przez co dla grafów planarnych liczba operacji na krawędziach jest ograniczona przez $O(n)$.

Listing 4.2. Moduł boruvkaplanar.

```
#!/usr/bin/python

from unionfind import UnionFind
from edges import Edge

class BoruvkaPlanarMST:
    """Finding a minimum spanning tree for planar graphs."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        # Graf ma byc spojny.
        self.graph = graph
        self.mst = self.graph.__class__(self.graph.v())
        for node in self.graph.iternodes():   # isolated nodes are possible
            self.mst.add_node(node)
        self._uf = UnionFind()

    def run(self):
        """Finding MST."""
        for node in self.graph.iternodes():
            self._uf.create(node)
        vforest = set(self.graph.iternodes())
        eforest = set(self.graph.iteredges())
        dummy_edge = Edge(None, None, float("inf"))
        while len(vforest) > 1:
            min_edges = dict(((node, dummy_edge) for node in vforest))
            # Finding the cheapest edges.
            for edge in eforest:
                source = self._uf.find(edge.source)
                target = self._uf.find(edge.target)
                if source != target:   # different components
                    if edge < min_edges[source]:
                        min_edges[source] = edge
```


24:8746869774

```
                if edge < min_edges[target]:
                    min_edges[target] = edge
        # Connecting components, total time is O(V).
        for edge in min_edges.itervalues():
            if edge is dummy_edge: # a disconnected graph
                #continue
                raise ValueError("disconnected graph")
            source = self._uf.find(edge.source)
            target = self._uf.find(edge.target)
            if source != target:   # different components
                self._uf.union(source, target)
                self.mst.add_edge(edge)
        # Remove duplicates, total time is O(V).
        # Redukcja liczby wierzcholkow co najmniej o polowe.
        vforest = set(self._uf.find(node) for node in vforest)
        # Selecting edges.
        # Redukcja liczby krawedzi co najmniej o polowe.
        min_edges = dict()
        for edge in eforest:
            source = self._uf.find(edge.source)
            target = self._uf.find(edge.target)
            # Petle wlasne pomijamy.
            if source != target:   # different components
                if source > target:
                    # Nie chce miec powtorzen.
                    source, target = target, source
                if edge < min_edges.get((source, target), dummy_edge):
                    min_edges[source, target] = edge
        eforest = set(min_edges.itervalues())

    def to_tree(self):
        """Return MST as a graph."""
        return self.mst
```

4.2. Kolorowanie wierzchołków grafu planarnego

Dane wejściowe: Graf planarny G.

Problem: Kolorowanie wierzchołków grafu planarnego G pięcioma kolorami.

Dane wyjściowe: Słownik color zawierający kolorowanie wierzchołków.

Opis algorytmu: Ogólne działanie algorytmu można podzielić na dwie fazy. Pierwsza faza polega na wykonywaniu poszczególnych procedur na wierzchołkach w celu ich zredukowania, w zależności od tego jakiego jest on stopnia. W drugiej fazie zaś następuje odwrócenie całego procesu i odbudowanie grafu. W algorytmie zastosowano dwie kolejki priorytetowe, w których znajdują się wierzchołki. W jednej znajdują się wierzchołki o stopniu 4 lub mniejszym, a w drugiej wierzchołki stopnia 5. Rozgraniczenie to jest potrzebne do procedury redukcji poszczególnych wierzchołków, które są pobierane z kolejek

25:6857385006

w kolejności najpierw z pierwszej, a potem z drugiej, jeśli pierwsza kolejka jest pusta.

Przypisanie wierzchołka do odpowiedniej kolejki wykonuje procedura consider. W pierwszej fazie pobierany jest wierzchołek z kolejki i w zależności od jego stopnia przeprowadzana jest procedura **reduce** i dodatkowo procedura identify, jeśli wierzchołek jest stopnia 5, a jego sąsiedzi zostają odpowiednio przypisani do kolejek w procedurze consider. W procedurze **reduce** wierzchołek zostaje usunięty z grafu i zapisany na specjalnym stosie potrzebnym do odtworzenia grafu. Jeśli procedura ta wykonywana jest dla wierzchołka stopnia 5 i posiada on dwóch niepołączonych ze sobą sąsiadów, to dodatkowo wykonana zostaje procedura identify, w której oba wierzchołki zostają scalone w jeden superwierzchołek i zostaje on odłożony na stos jako całość oraz usunięty z grafu.

Procedury wykonywane są dotąd, aż w grafie pozostanie maksymalnie 5 wierzchołków. Wierzchołki te kolorowane są na 5 różnych kolorów. W fazie drugiej pobierane są wierzchołki ze stosu, na który wcześniej zostały odłożone i po kolei wstawiane z powrotem do grafu wraz z ich sąsiadami. Podczas wstawiania wierzchołek jest kolorowany na inny kolor niż wierzchołek, do którego jest przyłączany. Tak samo zostaje odtworzony superwierzchołek. Jest on ponownie rozdzielany na niepołączone ze sobą wierzchołki, które otrzymują ten sam kolor.

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

Uwagi: Istotne jest, że w algorytmie procedury **reduce**() i identify() są wykonywane dla wierzchołków stopnia co najwyżej 11, przez co czas ich pracy jest $O(1)$. Testy implementacji algorytmu kolorowania wierzchołków grafu planarnego znajdują się w dodatku A.2.

Listing 4.3. Moduł nodecolorplanar2.

```
#!/usr/bin/python

import graphs
import edges
import random
import itertools

class FiveColoringPlanarGraph():

    def __init__(self, planar_graph):
        self.planar_graph = planar_graph.copy()
        self.stack_of_nodes = list()
        self.queueQ, self.queueR = set(), set()
        self.colors = dict()

    def run (self):
        for node in self.planar_graph.iternodes():
            self.procedure_consider(node)
        self.phase_1()
        self.phase_2()
        return self.colors
```


26:1061749283

```
def phase_1 (self):
    while self.planar_graph.v() > 5:
        if self.queueQ:
            v = self.queueQ.pop()
            if self.planar_graph.has_node(v):
                self.procedure_reduce(v)
        elif self.queueR:
            v = self.queueR.pop()
            if self.planar_graph.has_node(v):
                node_pair = self.is_reducible(v)
                self.procedure_reduce(v)
                self.procedure_identify(node_pair)

def phase_2 (self):
    for (i, vertice) in enumerate(self.planar_graph.iternodes()):
        self.colors[vertice] = i+1

    while self.stack_of_nodes:
        node_from_stack = self.stack_of_nodes.pop()
        if len(node_from_stack) is 2:
            colors_number = {1,2,3,4,5}
            for edge in node_from_stack[1]:
                colors_number.discard(self.colors[edge.target])
                self.planar_graph.add_edge(edge)
            if colors_number:
                self.colors[node_from_stack[0]] = colors_number.pop()
        elif len(node_from_stack) is 5:
            if node_from_stack[0] in self.colors:
                color_of_node = self.colors[node_from_stack[0]]
                del self.colors[node_from_stack[0]]
                self.colors[node_from_stack[1]] = color_of_node
                self.colors[node_from_stack[3]] = color_of_node
                self.planar_graph.del_node(node_from_stack[0])
                for edge in node_from_stack[2]:
                    self.planar_graph.add_edge(edge)
                for edge in node_from_stack[4]:
                    self.planar_graph.add_edge(edge)

def is_reducible (self, node):
    exist_edge = False
    if self.planar_graph.degree(node) == 5:
        neighbors = [v for v in self.planar_graph.iteradjacent(node)
            if self.planar_graph.degree(v) <= 11]
        for x,y in itertools.combinations(neighbors, 2):
            new_edge = edges.Edge(x,y)
            exist_edge = self.planar_graph.has_edge(new_edge)
            if not exist_edge:
                return x,y
    return exist_edge

def procedure_consider (self, node):
    if self.planar_graph.degree(node) <= 4:
        self.queueR.discard(node)
        self.queueQ.add(node)
    elif self.planar_graph.degree(node) == 5:
        if self.is_reducible(node):
            self.queueR.add(node)
```


27:3928192107

```
            else:
                self.queueR.discard(node)
        elif self.planar_graph.degree(node) == 11:
            for neighbor in self.planar_graph.iteradjacent(node):
                if self.is_reducible(neighbor):
                    self.queueR.add(neighbor)

    def procedure_reduce (self, node):
        neighbors = list(self.planar_graph.iteroutedges(node))
        self.stack_of_nodes.append((node, neighbors))
        self.planar_graph.del_node(node)
        for edge in neighbors:
            self.procedure_consider(edge.target)

    def procedure_identify (self, node_pair):
        x, y = node_pair
        while True:
            new_node = random.random()
            if self.planar_graph.has_node(new_node):
                continue
            else:
                self.planar_graph.add_node(new_node)
                break
        neighbors_x = list(self.planar_graph.iteroutedges(x))
        neighbors_y = list(self.planar_graph.iteroutedges(y))
        new_neighbors = set(edge.target for edge in neighbors_x)
        new_neighbors.update(edge.target for edge in neighbors_y)
        self.planar_graph.del_node(x)
        self.planar_graph.del_node(y)
        self.planar_graph.add_node(new_node)
        for neighbor in new_neighbors:
            new_edge = edges.Edge(new_node, neighbor)
            self.planar_graph.add_edge(new_edge)
        self.stack_of_nodes.append((new_node, x, neighbors_x, y, neighbors_y))
        self.procedure_consider(new_node)
        for neighbor in new_neighbors:
            self.procedure_consider(neighbor)
```

4.3. Grafy szeregowo-równoległe skierowane

W tym rozdziale przedstawimy najpierw generator dsp-grafu przypadkowego, a następnie algorytm wykrywania dsp-grafu, który dla poprawnego dsp-grafu zwraca wyznaczone dsp-drzewo. Na końcu pokażemy generator przypadkowego dsp-drzewa, który można wykorzystać do testowania algorytmów przetwarzających dsp-grafy.

4.3.1. Generator dsp-grafu

Dane wejściowe: Liczba wierzchołków grafu G.

Dane wyjściowe: Graf szeregowo-równoległy skierowany G.

28:8931866839

Opis algorytmu: Funkcja ta tworzy losowy skierowany graf szeregowo-równoległy. Jest to implementacja reguł budowania grafu na podstawie definicji III o dsp-grafach (rozdział 2.5). Na początku zostaje utworzona pojedyncza krawędź skierowana z wierzchołkami początkowym i końcowym, a następnie graf jest rozbudowywany poprzez przypadkowy wybór krawędzi, na której następnie zostaje wykonana operacja szeregowa lub równoległa, podczas których powstają nowe wierzchołki. Graf zostaje ukończony w momencie osiągnięcia wyznaczonej ilości wierzchołków.

Listing 4.4. Moduł make_dspgraph.

```
#!/usr/bin/python

import random
from edges import Edge
from graphs import Graph

def swap(L, i, j):
    """Swap items on the list."""
    L[i], L[j] = L[j], L[i]

def make_random_dspgraph(n):
    """Make a directed series-parallel graph with n vertices."""
    if n < 2:
        raise ValueError("bad n")
    graph = Graph(n, True)    # graf SKIEROWANY
    for node in xrange(n):
        graph.add_node(node)
    source = 0
    sink = n-1
    idx = 1
    edge_list = [Edge(source, sink, idx)]
    idx += 1
    node = n-2
    while node > 0:
        # Losowanie krawedzi na ktorej bedzie operacja.
        i = random.randrange(0, len(edge_list))
        swap(edge_list, i, -1)
        edge = edge_list[-1]
        # Losowanie operacji.
        action = random.choice(["series", "parallel"])
        #print "action", action
        if action == "series":
            edge_list.pop()
            edge_list.append(Edge(edge.source, node, idx))
            idx += 1
            edge_list.append(Edge(node, edge.target, idx))
            idx += 1
        elif action == "parallel":
            edge_list.append(Edge(edge.source, node, idx))
            idx += 1
            edge_list.append(Edge(node, edge.target, idx))
            idx += 1
        node -= 1
    for edge in edge_list:
        graph.add_edge(edge)
    return graph
```


29:1006339339

4.3.2. Znajdowanie drzewa szeregowo-równoległego

Rozpoznawanie dsp-grafu oznacza faktycznie wyznaczanie dsp-drzewa, czyli drzewa binarnego, którego liście odpowiadają krawędziom dsp-grafu, a węzły wewnętrzne odpowiadają operacjom szeregowym i równoległym, które doprowadziły do powstania dsp-grafu. Takie dsp-drzewo zawiera pełną informację o sposobie konstruowania dsp-grafu. Z definicji III wynika, że operacja szeregowa wiąże się z powiększeniem liczby krawędzi o 1, a operacja równoległa wiąże się z powiększeniem liczby krawędzi o 2. Łącznie liczba operacji szeregowych i równoległych jest rzędu $O(m)$, a więc liczba węzłów dsp-drzewa też jest rzędu $O(m)$.

Dane wejściowe: Graf szeregowo-równoległy skierowany G.

Problem: Algorytm znajdowania dsp-drzewa dla danego dsp-grafu G.

Dane wyjściowe: Drzewo szeregowo-równoległe T.

Opis algorytmu: Algorytm otrzymuje graf do przetworzenia. W pierwszym etapie redukuje graf, aż do uzyskania jednej krawędzi skierowanej z wierzchołkami początkowym i końcowym. Proces redukcji polega na znajdowaniu połączeń szeregowych lub równoległych i zastępowanie ich pojedynczymi krawędziami tak, jak to opisano w definicji II o dsp-grafach. Jest to zatem proces odwrotny do operacji tworzenia takich połączeń. Jeśli jednak graf nie zostanie doprowadzony do postaci jednej krawędzi oznacza to, że graf ten nie jest grafem szeregowo-równoległym. W drugim etapie tworzone jest dsp-drzewo na podstawie znalezionej sekwencji połączeń szeregowych i równoległych.

Złożoność: W pierwszym etapie redukcji grafu przetwarzane są wierzchołki ze stopniem wejściowym równym 1 i stopniem wyjściowym równym 1. Dla każdego wierzchołka rozpoznawana jest operacja szeregowa lub równoległa, a ich liczba jest ograniczona przez $O(m)$. Wydaje się, że liczba operacji wykonywanych przy przetwarzaniu każdego wierzchołka jest stała, jednak pojawia się tam operacja wyznaczania stopnia wejściowego pewnych wierzchołków grafu, graph.indegree(node). Dla grafów skierowanych w naszej implementacji ta operacja zajmuje czas $O(n)$, stąd całkowity czas algorytmu rozpoznawania dsp-grafu szacujemy na $O(n^2)$.

Listing 4.5. Moduł find_dsptree.

```
#!/usr/bin/python

from spnodes import Node
from edges import Edge
from graphs import Graph

def find_dsptree(graph):
    """Find an sp-tree for an sp-graph."""
```


30:9098521572

```
    if not graph.is_directed():
        raise ValueError("the graph is undirected")
    graph_copy = graph.copy()
    degree11 = set(node for node in graph.iternodes()
        if graph.indegree(node) == 1 and graph.outdegree(node) == 1)
        # active nodes with indegree=1 and outdegree=1
    call_stack = []
    tnode_dict = dict()   # tnode dla krawedzi
    # Etap I. Redukcja grafu do krawedzi.
    # Dopoki sa wierzcholki wykonuj odrywanie.
    while degree11:
        source = degree11.pop()
        if graph_copy.indegree(source) != 1:
            # Czasem stopien wierzcholka moze sie zmniejszyc!
            continue
        if graph_copy.outdegree(source) != 1:
            # Czasem stopien wierzcholka moze sie zmniejszyc!
            continue
        for edge in graph_copy.iterinedges(source):
            node1 = edge.source   # tylko jedna
        for edge in graph_copy.iteroutedges(source):
            node2 = edge.target   # tylko jedna
        edge = Edge(node1, node2)
        if graph_copy.has_edge(edge):
            # Jezeli ma krawedz, to trzeba poprawic stopnie wierzcholkow,
            # bo przy usuwaniu krawedzi przy node1 zmniejsza sie outdegree,
            # a przy node2 zmniejsza sie indegree.
            if (graph_copy.outdegree(node1) == 2 and
                graph_copy.indegree(node1) == 1):
                degree11.add(node1)
            if (graph_copy.outdegree(node2) == 1 and
                graph_copy.indegree(node2) == 2):
                degree11.add(node2)
            call_stack.append(("parallel", source, node1, node2))
        else:   # tu nie trzeba poprawiac stopni
            graph_copy.add_edge(edge)
            call_stack.append(("series", source, node1, node2))
        # Usuwamy krawedzie z source.
        graph_copy.del_edge(Edge(node1, source))
        graph_copy.del_edge(Edge(source, node2))
    # Etap II. Sprawdzamy co zostalo. Ma zostac jedna krawedz.
    if graph_copy.e() == 1:
        # Zostala jedna krawedz.
        edge = next(graph_copy.iteredges())
        root = Node(edge.source, edge.target, "edge")
        tnode_dict[(edge.source, edge.target)] = root
    else:
        raise ValueError("not an sp-graph, e() != 1")
    # Etap III. Budowa sp-tree na bazie call_stack.
    while call_stack:
        action, node, node1, node2 = call_stack.pop()
        if (node1, node2) in tnode_dict:
            tnode = tnode_dict[(node1, node2)]
        else:
            raise ValueError("key not in tnode_dict")
        if action == "series":
            del tnode_dict[(tnode.source, tnode.target)]
```

```
            tnode.type = "series"
            tnode.left = Node(tnode.source, node, "edge")
            tnode.right = Node(node, tnode.target, "edge")
            tnode_dict[(tnode.source, node)] = tnode.left
            tnode_dict[(node, tnode.target)] = tnode.right
        elif action == "parallel":
            #del tnode_dict[(tnode.source, tnode.target)]   # nadpisuje
            tnode.type = "parallel"
            tnode.left = Node(tnode.source, tnode.target, "edge")
            tnode.right = Node(tnode.source, tnode.target, "series")
            tnode.right.left = Node(tnode.source, node, "edge")
            tnode.right.right = Node(node, tnode.target, "edge")
            tnode_dict[(tnode.source, tnode.target)] = tnode.left
            tnode_dict[(tnode.source, node)] = tnode.right.left
            tnode_dict[(node, tnode.target)] = tnode.right.right
        else:
            raise ValueError("bad action")
        assert all(tnode_dict[key].type == "edge" for key in tnode_dict)
    return root
```

4.3.3. Generator dsp-drzewa

Dane wejściowe: Liczba wierzchołków drzewa T.

Dane wyjściowe: Drzewo szeregowo-równoległe T.

Opis algorytmu: Jest to funkcja tworząca losowe drzewo szeregowo-równoległe. Działa podobnie jak funkcja tworząca dsp-graf, ale na innej strukturze danych. Na początku zostaje utworzony zostaje korzeń, czyli najprostsze drzewo szeregowo-równoległe składające się z pojedynczej krawędzi skierowanej. Następnie przetwarzana jest losowo kolejna krawędź poprzez operację szeregową lub równoległą. Krawędzie przetwarzane są dotąd, aż drzewo uzyska wymaganą liczbę wierzchołków. Tak skonstruowane drzewo jest drzewem szeregowo-równoległym. Algorytm zwraca łącze do korzenia.

Listing 4.6. Moduł make_dsptree.

```
#!/usr/bin/python

import random
from spnodes import Node

def swap(L, i, j):
    """Swap items on the list."""
    L[i], L[j] = L[j], L[i]

def make_random_dsptree(n):
    """Make a random dsp-tree with n vertices."""
    if n < 2:
        raise ValueError("bad n")
    source = 0
    sink = n-1
    root = Node(source, sink, "edge")
    tnode_list = [root]   # na tej liscie maja byc type=edge
```


32:4051613140

```
    node = n-2
    while node > 0:
        # Losowanie krawedzi na ktorej bedzie operacja.
        i = random.randrange(0, len(tnode_list))
        swap(tnode_list, i, -1)
        tnode = tnode_list.pop()
        # Losowanie operacji.
        action = random.choice(("series", "parallel"))
        if action == "series":
            tnode.type = "series"
            tnode.left = Node(tnode.source, node, "edge")
            tnode.right = Node(node, tnode.target, "edge")
            tnode_list.append(tnode.left)
            tnode_list.append(tnode.right)
        elif action == "parallel":
            tnode.type = "parallel"
            tnode.left = Node(tnode.source, tnode.target, "edge")
            tnode.right = Node(tnode.source, tnode.target, "series")
            tnode.right.left = Node(tnode.source, node, "edge")
            tnode.right.right = Node(node, tnode.target, "edge")
            tnode_list.append(tnode.left)
            tnode_list.append(tnode.right.left)
            tnode_list.append(tnode.right.right)
        else:
            raise ValueError("bad action")
        node -= 1
        assert all(tnode.type == "edge" for tnode in tnode_list)
    return root
```

5. Podsumowanie

W ramach pracy zaimplementowano dwa algorytmy znajdujące minimalne drzewo rozpinające dla grafu planarnego. W algorytmie Cheritona-Tarjana dla prostoty użyto standardowej kolejki priorytetowej, co w praktyce niewiele pogorszyło jego wydajność. W zmodyfikowanym algorytmie Borůvki istotna jest redukcja liczby krawędzi o połowę w każdym kroku.

Problem kolorowania wierzchołków grafu planarnego ma długą historię, której zwieńczeniem jest trudny algorytm 4-kolorowania wierzchołków. Przedstawiliśmy w pracy prostszy algorytm 5-kolorowania wierzchołków, który korzysta m.in. z procedury identyfikacji dwóch sąsiadów wierzchołka stopnia 5 niepołączonych krawędzią. W pierwszej fazie algorytmu graf jest redukowany przez usuwanie wybranych wierzchołków. W drugiej fazie wierzchołki są przywracane i kolorowane.

Trzecie zagadnienie rozważane pracy to grafy szeregowo-równoległe skierowane ważone (dsp-grafy), które należą do planarnych sieci przepływowych. Przygotowano generator przypadkowych dsp-grafów oraz generator przypadkowych dsp-drzew. Na bazie kodu dla grafów szeregowo-równoległych nieskierowanych przygotowano algorytm rozpoznawania dsp-grafów. W kontekście sieci przepływowych można będzie szybko sprawdzić, czy dana sieć jest planarnym dsp-grafem, a wtedy będzie można stosować specjalizowane algorytmy na maksymalny przepływ.

Zaimplementowanym algorytmom towarzyszy kod sprawdzający poprawność i praktyczną wydajność programów.

34:1044510901

A. Testy algorytmów

Dodatek zawiera testy sprawdzające rzeczywistą wydajność algorytmów zaimplementowanych w pracy.

A.1. Testy minimalnego drzewa rozpinającego

Testowanie algorytmów wyznaczających minimalne drzewo rozpinające dla grafów planarnych. Do testów wykorzystano generatory grafów planarnych ważonych z pakietu graphtheory: sieć kwadratową $[n = s^2, m = 2s(s-1)]$ i sieć trójkątną $[n = s^2, m = (3s-1)(s-1)]$. Zbadano algorytm Cheritona-Tarjana i zmodyfikowany algorytm Borůvki.

Rysunki A.1 i A.2 pokazują, że nawet implementacja wykorzystująca zwykłą kolejkę priorytetową pozwala osiągnąć wydajność bliską liniowej $O(n)$. Rysunki A.3 i A.4 potwierdzają liniową złożoność zmodyfikowanego algorytmu Borůvki.

A.2. Testy kolorowania wierzchołków grafu planarnego

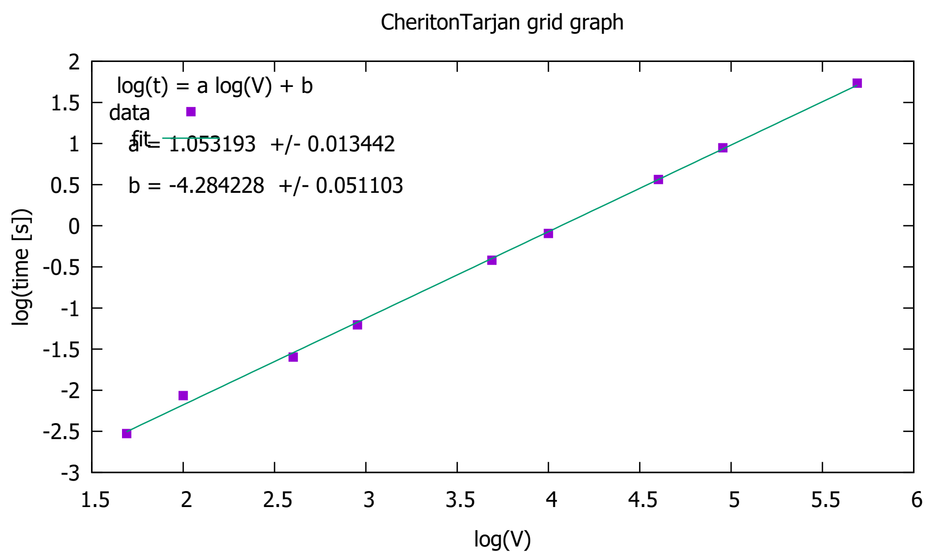
Testowanie kolorowania wierzchołków grafu planarnego pięcioma kolorami. Do testów wykorzystano cztery generatory grafów planarnych przygotowanych specjalnie dla tej pracy:

— Generator na bazie sieci trójkątnej z brzegiem, gdzie do generatora z pakietu graphtheory dodano nowe krawędzie na brzegu, aby najmniejszy stopień wierzchołka wynosił 4, a nie 2. Parametrem generatora jest $s \geqslant 3$, $n = s^2$, $m = 3n - 7$.

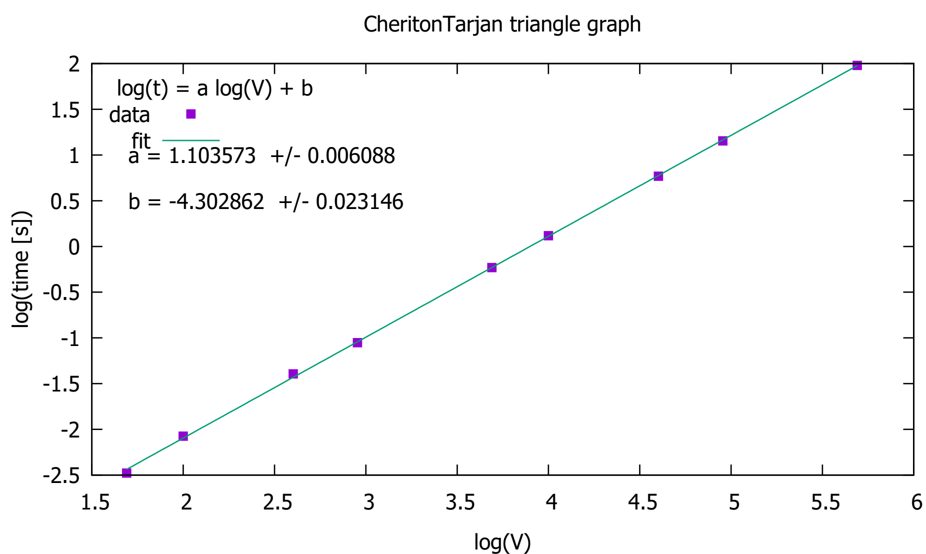
— Generator na bazie grafu antypryzmy, gdzie dodano dwa dodatkowe wierzchołki połączone z pozostałymi tak, aby najmniejszy stopień wierzchołka wynosił 5, a nie 4. Parametrem generatora jest $s > 4$, $n = 2s+2$, $m = 6s$.

— Generator bazujący na grafie dla dwudziestościanu foremnego, który to graf jest 5-regularny. Generator w zadanej liczbie kroków dodaje nowe wierzchołki i krawędzie w taki sposób, że wszystkie nowe wierzchołki mają stopień 6, a wierzchołki pierwotnego dwudziestościanu mają stopień 5. Pojedynczy krok generatora ma dwie fazy: (1) umieszczenie nowych wierzchołków na środkach wszystkich krawędzi (czyli podział starych krawędzi), (2) dla każdej ściany, połączenie nowych trzech wierzchołków leżących na brzegu ściany trzema krawędziami (czyli podział starej trójkątnej ściany na cztery nowe ściany trójkątne).
  Liczby wierzchołków n_k, krawędzi m_k i ścian f_k w kolejnych krokach można powiązać związkami rekurencyjnymi: $n_{k+1} = n_k + m_k$, $f_{k+1} = 4f_k$, $m_{k+1} = 2m_k + 3f_k = 4m_k$, gdzie kolejne wartości (n_k, m_k, f_k) dla

35:3020032952

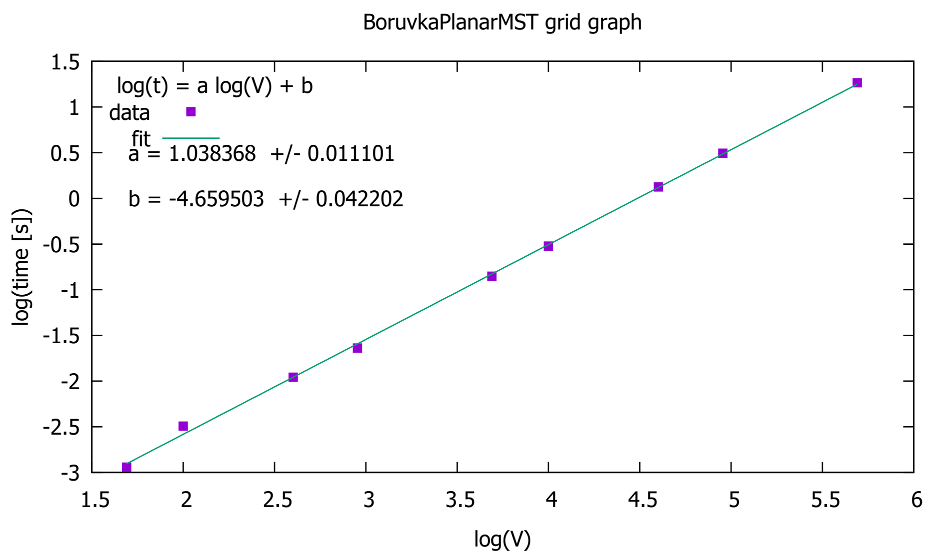


Rysunek A.1. Wydajność algorytmu Cheritona-Tarjana dla sieci kwadratowej. Współczynnik a bliski 1 potwierdza złożoność niewiele wyższą od liniowej.

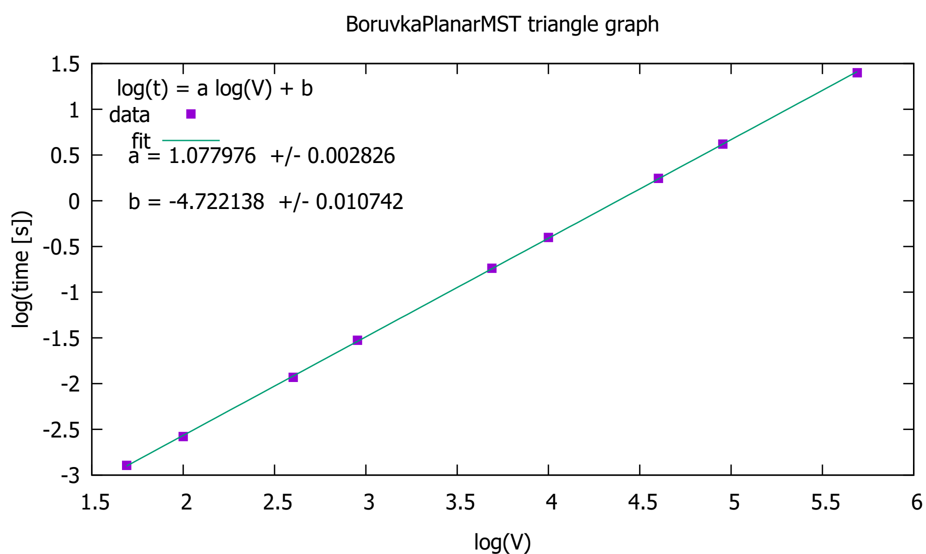


Rysunek A.2. Wydajność algorytmu Cheritona-Tarjana dla sieci trójkątnej. Współczynnik a bliski 1 potwierdza złożoność niewiele wyższą od liniowej.

36:4296242916



Rysunek A.3. Wydajność zmodyfikowanego algorytmu Borůvki dla sieci kwadratowej. Współczynnik a bliski 1 potwierdza złożoność liniową.



Rysunek A.4. Wydajność zmodyfikowanego algorytmu Borůvki dla sieci trójkątnej. Współczynnik a bliski 1 potwierdza złożoność liniową.

37:1100471316

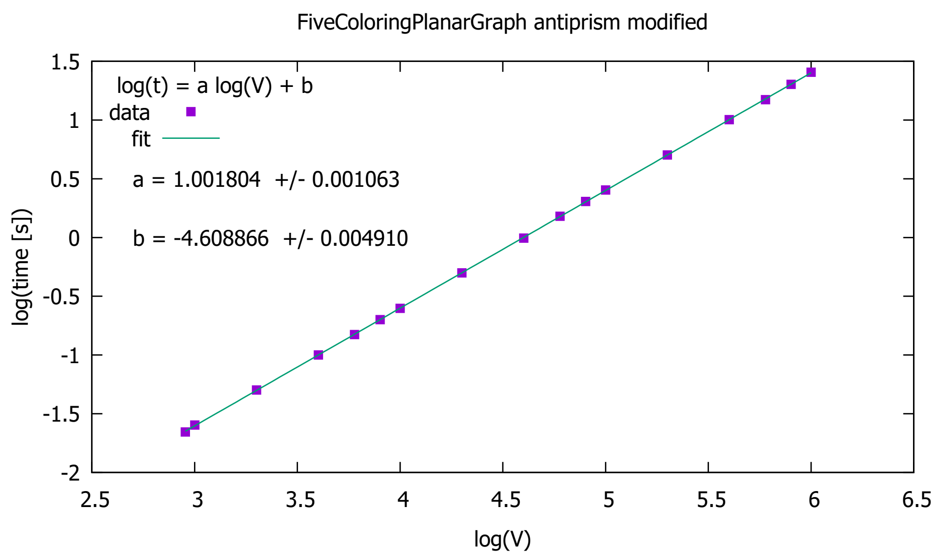
$k = 1, 2, 3$ wynoszą $(12, 30, 20)$, $(42, 80, 120)$, $(162, 320, 480)$. Okazuje się, że można podać zwarte wzory na parametry grafu: $n_k = (5/2) \cdot 4^k + 2$, $m_k = (15/2) \cdot 4^k$, $f_k = 5 \cdot 4^k$.

— Generator startujący od grafu K_3 ($k = 1$), który dodaje w każdym kroku k trzy wierzchołki. Ściany grafu są trójkątami. Dla $k = 2$ powstaje graf 3-antypryzma z parametrami $n_2 = 6$, $m_2 = 12$, $f_2 = 8$. Można podać zwarte wzory na parametry grafu: $n_k = 3k$, $m_k = 9k - 6$, $f_k = 6k - 4$ $[m = 3n - 6,\ f = 2n - 4]$. Dla $k = 4$ otrzymujemy graf dwudziestościanu foremnego. Dla $k \geqslant 4$ graf posiada tylko wierzchołki stopni 5 i 6.

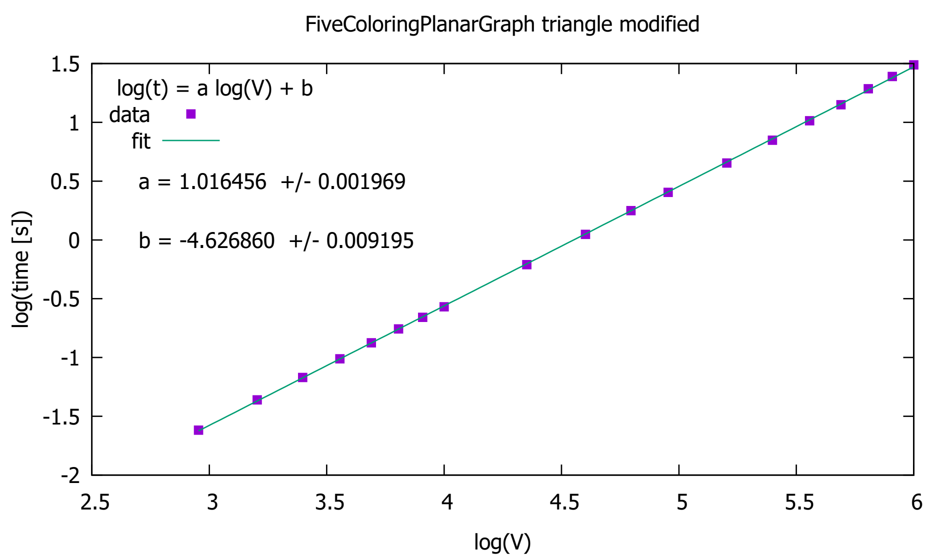
Rysunki od A.5 do A.8 pokazują czasy 5-kolorowania wierzchołków dla różnych rodzajów grafów planarnych. Czas działania algorytmu kolorowania jest liniowy $O(n)$.

A.3. Testy grafów szeregowo-równoległych skierowanych

Testowanie rozpoznawania dsp-grafów, czyli funkcji find_dsptree(). Do testów wykorzystano generator dsp-grafów przypadkowych. Rysunek A.9 pokazuje prawie kwadratowy czas pracy algorytmu wynikający z użycia czasochłonnej funkcji znajdowania stopnia wejściowego wierzchołków w grafie skierowanym.

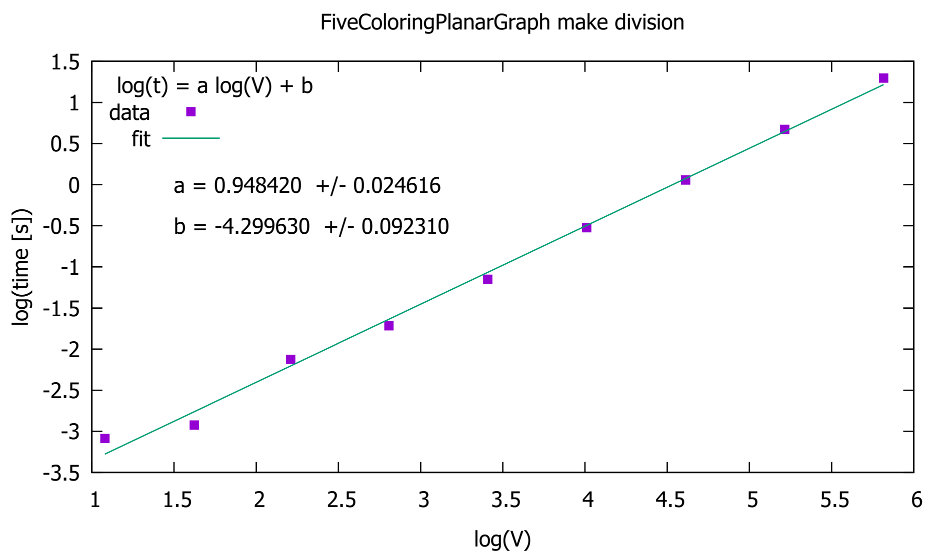


Rysunek A.5. Wydajność algorytmu kolorowania dla antiprism. Współczynnik a bliski 1 potwierdza złożoność liniową.

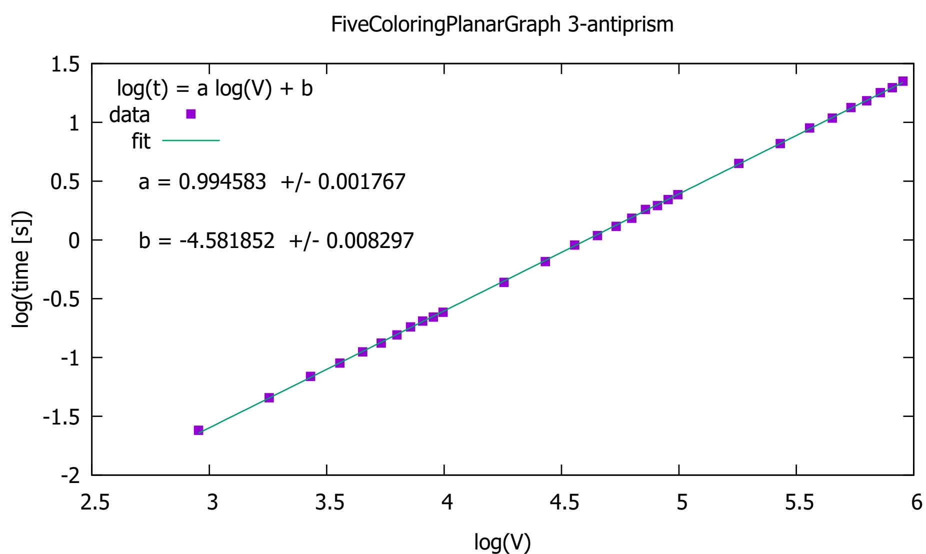


Rysunek A.6. Wydajność algorytmu kolorowania dla sieci trójkątnej. Współczynnik a bliski 1 potwierdza złożoność liniową.

39:3906630034

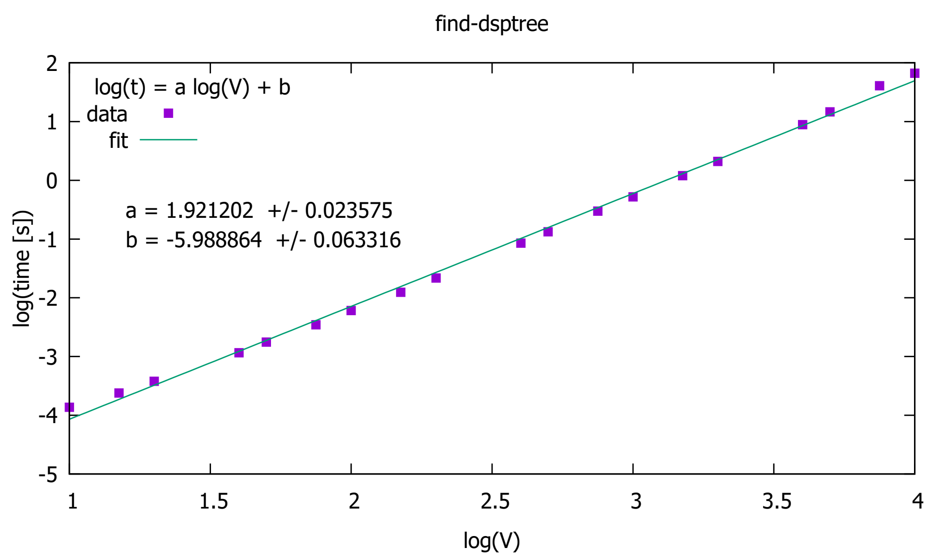


Rysunek A.7. Wydajność algorytmu kolorowania dla grafu dzielonego. Współczynnik a bliski 1 potwierdza złożoność liniową.



Rysunek A.8. Wydajność algorytmu kolorowania dla grafu 3-antypryzmy. Współczynnik a bliski 1 potwierdza złożoność liniową.

40:8328146260



Rysunek A.9. Wydajność algorytmu rozpoznawania dsp-grafów. Współczynnik a bliski 2 potwierdza złożoność $O(n^2)$.

41:8329445038

Bibliografia

[1] Wikipedia, Planar graph, 2018,
https://en.wikipedia.org/wiki/Planar_graph.

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

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

[4] Brenda S. Baker, *Approximation Algorithms for NP-Complete Problems on Planar Graphs*, Journal of the Association for Computing Machinery 41(1), 153-180 (1994).

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

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

[7] Wikipedia, Graph minor, 2018,
https://en.wikipedia.org/wiki/Graph_minor.

[8] Wikipedia, Robertson-Seymour theorem, 2018,
https://en.wikipedia.org/wiki/Robertson%E2%80%93Seymour_theorem.

[9] I. S. Filotti, Jack N. Mayer, *A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus*, Proceedings of the 12th Annual ACM Symposium on Theory of Computing, p. 236-243 (1980).

[10] Monika R. Henzinger, Philip Klein, Satish Rao, Sairam Subramanian, *Faster Shortest-Path Algorithms for Planar Graphs*, Journal of Computer and System Sciences 55, 3-23 (1997).

[11] Richard J. Lipton, Donald J. Rose, Robert Endre Tarjan, *Generalized nested dissection*, SIAM J. Numer. Anal. 16, 346-358 (1979).

[12] Greg N. Frederickson, *Fast algorithms for shortest paths in planar graphs*, SIAM Journal on Computing 16(6), 1004-1022 (1987).

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

[14] Glencora Borradaile, Philip Klein, *An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph*, Journal of the ACM 56(2), Article 9 (2009).

[15] Wikipedia, Series-parallel graph, 2018,
https://en.wikipedia.org/wiki/Series-parallel_graph.

[16] Lech Banachowski, Krzysztof Diks, Wojciech Rytter, *Algorytmy i struktury danych*, Wydawnictwo Naukowe PWN, Warszawa 2018.

[17] Konrad Gałuszka, *Badanie grafów szeregowo-równoległych z językiem Python*, Uniwersytet Jagielloński, Kraków, 2018.

[18] D. Cheriton, R. E. Tarjan, *Finding minimum spanning trees*, SIAM Journal on Computing 5, 724-742 (1976).

[19] Tomomi Matsui, *The minimum spanning tree problem on a planar graph*, Discrete Applied Mathematics 58, 91-94 (1995).

42:1016396396

[20] Wikipedia, Borůvka's algorithm, 2018,
https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm.

[21] Martin Mareš, *Two linear time algorithms for MST on minor closed graph classes*, Archivum Mathematicum 40(3), 315-320 (2004).

43:2468272000