Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Konrad Gałuszka

Nr albumu: 1077347

Badanie grafów szeregowo-równoległych z językiem Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2018

Oświadczenie autora pracy

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

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

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

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

Kraków, dnia Podpis kierującego pracą

Składam serdeczne podziękowania Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za olbrzymie zaangażowanie, cierpliwość, poświęcony czas i wszelkie rady merytoryczne, bez których ta praca z pewnością nie miałaby prawa powstać.

Streszczenie

Praca prezentuje implementację w języku Python wybranych algorytmów dla grafów szeregowo-równoległych. Grafy szeregowo-równoległe (sp-grafy) są to grafy planarne z dwoma wyróżnionymi wierzchołkami, przy czym grafy są tworzone rekurencyjnie za pomocą trzech operacji: szeregowej, równoległej i *jackknife*. Najprostszym sp-grafem jest pojedyńcza krawędź. W pracy rozważane są jedynie sp-grafy proste nieskierowane.

W pracy opisano wiele właściwości sp-grafów i wykorzystano je do implementacji wydajnych algorytmów, na ogół działających z liniową złożonością obliczeniową. Zbudowano generatory dla przypadkowych sp-grafów, sp-drzew i k-drzew. Przygotowano algorytm rozpoznawania sp-grafów, który buduje sp-drzewo dla danego sp-grafu. Znalezione sp-drzewo jest wykorzystywane w algorytmach stosujących metodę programowania dynamicznego do znajdowania największego zbioru niezależnego, najmniejszego zbioru dominującego, największego skojarzenia, najmniejszego pokrycia wierzchołkowego. Podano algorytmy do kolorowania wierzchołków sp-grafu i do znajdowania doskonałego uporządkowania wierzchołków (PEO) dla uzupełnienia cięciwowego danego sp-grafu.

Dodatkowo przygotowano algorytm do znajdowania PEO dla uzupełnienia cięciwowego grafu Halina. Graf Halina jest grafem planarnym, zbudowanym z płaskiego rysunku drzewa, posiadającego co najmniej cztery wierzchołki, bez wierzchołków stopnia dwa. Wszystkie liście drzewa, w kolejności wyznaczonej przez rysunek drzewa, są połączone krawędziami tworzącymi cykl zewnętrzny. Grafy Halina i sp-grafy należą do grafów o ograniczonej szerokości drzewowej.

Słowa kluczowe: grafy, multigrafy, grafy szeregowo-równoległe, dekompozycja drzewowa, szerokość drzewowa, kolorowanie grafu, zbiory niezależne, zbiory dominujące, pokrycie wierzchołkowe, skojarzenia, grafy Halina

English title: Study of series-parallel graphs with Python

Abstract

Python implementation of selected algorithms for series-parallel graphs is presented. Series-parallel graphs (sp-graphs) are planar graphs with two distinguished vertices, formed recursively by three composition operations: series, parallel, and jackknife. The simplest sp-graph is a single edge. In this work, simple undirected sp-graphs are considered only.

Many properties of sp-graphs are described and used to implement efficient algorithms working, as a rule, with linear complexity. Generators for random sp-graphs, sp-trees, and k-trees are built. The algorithm for recognizing sp-graphs and building a corresponding sp-tree is prepared. An sp-tree is used in dynamic programming algorithms finding a maximum independent set, a minimum dominating set, a maximum matching, a minimum vertex cover. Algorithms for sp-graph vertex coloring and for a perfect elimination ordering (PEO) of a chordal completion are given.

Additionaly, the algorithm for a PEO of a chordal completion for a Halin graph is given. A Halin graph is a planar graph constructed from a planar drawing of a tree having four or more vertices, no vertices of degree two, by adding an outer cycle of edges connecting all leafs of the tree in the cyclic order. Halin graphs and sp-graphs are graphs of bounded treewidth.

Keywords: graphs, multigraphs, series-parallel graphs, tree decomposition, treewidth, graph coloring, independent sets, dominating sets, vertex cover, matching, Halin graphs

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy jest badanie grafów szeregowo-równoległych z wykorzystaniem języka Python do implementacji wybranych algorytmów. Grafy szeregowo-równoległe pojawiają się w wielu zastosowaniach. Klasyczny sposób obliczania oporu zastępczego dla sieci oporników zakłada, że ta sieć jest grafem szeregowo-równoległym (należy jeszcze dodać transformację trójkąta w gwiazdę i na odwrót). Grafy szeregowo-równoległe mogą zostać użyte w skomplikowanych systemach komputerowych, w których pojawia się silna potrzeba optymalnego zorganizowania wykonywanych zadań, tak aby maksymalnie wykorzystać dostępne zasoby. Sprowadza się to do odpowiedniego zamodelowania przepływu operacji, gdzie część zostanie wykonana pojedynczo (szeregowo), a część równolegle.

Celem pracy jest implementacja w języku Python wybranych algorytmów związanych z grafami szeregowo-równoległymi. Algorytmy te w przybliżeniu można zaliczyć do jednej z trzech kategorii.

1. Rozpoznawanie grafów szeregowo-równoległych i przedstawienie ich struktury w wygodniejszej postaci sp-drzewa (definicja w rozdziale 2).
2. Generowanie grafów szeregowo-równoległych przypadkowych. Jest to potrzebne np. do testowania algorytmów pracujących na grafach szeregowo-równoległych.
3. Rozwiązywanie problemów trudnych w przypadku ogólnych grafów (problemy nie posiadające rozwiązań działających w czasie wielomianowym), po zawężeniu się do grafów szeregowo-równoległych. Wiele problemów trudnych ma praktyczne znaczenie, dlatego ważne jest szukanie takich rodzin grafów, dla których istnieją algorytmy wielomianowe dające optymalne rozwiązania. Grafy szeregowo-równoległe należą do takich szczególnych rodzin. Inną szczególną rodzinę tworzą grafy cięciwowe i czasem potrzebne jest znalezienie nadgrafu cięciwowego dla badanego grafu. Odpowiednie algorytmy będą pokazane w pracy.

Trzeba pamiętać, że konstrukcja grafów szeregowo-równoległych może prowadzić do powstania krawędzi równoległych, a wtedy trzeba mówić o multigrafach, a nie grafach prostych. Niniejsza praca dotyczy grafów szeregowo-równoległych nieskierowanych. W zastosowaniach pojawiają się również grafy szeregowo-równoległe skierowane. Służą do wizualizacji diagramów przepływu, sieci zależności, sieci PERT.

Grafy szeregowo-równoległe mają szerokość drzewową równą dwa (definicja w rozdziale 2), przez co są podobne do drzew. Ogólnie wyróżnia się grafy o małej szerokości drzewowej, ponieważ dla nich istnieją wydajne algorytmy oparte na technice programowania dynamicznego, rozwiązujące trudne problemy obliczeniowe. Grafy Halina (definicja w rozdziale 2) mają małą

szerokość drzewową równą trzy, dlatego część niniejszej pracy nawiązuje do tych grafów.

Podstawowe informacje o grafach pojawiają się w każdym poważnym podręczniku do informatyki [1], istnieją również pozycje w języku polskim w całości poświęcone grafom [2], [3], [4], [5]. Informacje o grafach szeregowo-równoległych są trudniej dostępne, korzystaliśmy głównie z artykułów w języku angielskim.

Do przygotowania nowych implementacji algorytmów dla grafów szeregowo-równoległych wybrano język Python [6] ze względu na czytelną składnię, bogatą bibliotekę standardową, oraz istniejący pakiet graphtheory rozwijany w Instytucie Fizyki UJ [7].

Treść niniejszej pracy jest zorganizowana następująco. Rozdział 1 podaje cele pracy. Rozdział 2 zawiera wprowadzenie do teorii grafów wraz z potrzebnymi definicjami i faktami. Rozdział 3 prezentuje interfejs biblioteki grafowej wykorzystanej do przygotowania implementacji algorytmów dla grafów szeregowo-równoległych. Rozdział 4 opisuje nowe implementacje algorytmów. Rozdział 5 stanowi podsumowanie pracy. W dodatku A zebrano wyniki testów wydajnościowych.

2. Teoria grafów

W tym rozdziale poświęcimy trochę uwagi podstawowym definicjom i twierdzeniom. Stworzymy tym samym bazę teoretyczną, która będzie stanowić punkt wyjścia do tworzenia algorytmów opartych na analizowanych rodzinach grafowych.

2.1. Podstawowe definicje

Ogólnie graf $G = (V, E)$ jest to para uporządkowana, gdzie $V(G)$ to zbiór wierzchołków, a $E(G)$ to zbiór krawędzi. Oznaczamy liczbę wierzchołków i liczbę krawędzi odpowiednio przez $n = |V(G)|$, $m = |E(G)|$. $N_G(v)$ to sąsiedztwo wierzchołka $v \in V(G)$, czyli zbiór wierzchołków osiągalnych z v.

Graf skierowany jest grafem, w którym zbiór krawędzi $E(G)$ składa się z uporządkowanych par wierzchołków. Innymi słowy orientacja poszczególnych krawędzi zdefiniowana jest poprzez kolejność wierzchołków w odpowiadającej parze. *Graf nieskierowany* jest natomiast grafem, w którym dla zbioru $E(G)$ nie rozróżnia się kolejności wierzchołków w parach, w związku z czym orientacja krawędzi nie jest zdefiniowana.

Podgraf H jest grafem, który zawiera się w całości w innym grafie G. Ściślej rzecz ujmując, zbiór wierzchołków $V(H)$ zawiera się w $V(G)$, a zbiór krawędzi $E(H)$ zawiera się w $E(G)$. *Graf indukowany* H jest szczególnym przykładem podgrafu grafu G. Mianowicie wymaga się, aby wszystkie krawędzie ze zbioru $E(G)$, których oba końce należą do $V(H)$, należały do zbioru $E(H)$.

Klika jest to zbiór wierzchołków indukujący *graf pełny*, w którym każde dwa wierzchołki są ze sobą połączone krawędzią. *Klika maksymalna* jest kliką, której nie da się powiększyć o kolejny wierzchołek. Innymi słowy, nie istnieje w grafie żadna inna klika zawierająca daną klikę w całości. *Klika największa* jest kliką o największej liczności w kontekście danego grafu. Jednocześnie ma własność kliki maksymalnej.

Ścieżka jest ciągiem wierzchołków wybranych w taki sposób, że każde dwa sąsiadujące ze sobą w tym ciągu wierzchołki są połączone krawędzią. *Ścieżka prosta* jest ścieżką, w której żaden wierzchołek nie występuje więcej niż jeden raz. *Cykl* jest ścieżką, w której pierwszy i ostatni wierzchołek są równe. *Cykl prosty* jest natomiast cyklem, w którym tylko pierwszy i ostatni wierzchołek są sobie równe. *Graf spójny* jest grafem, w którym pomiędzy każdą parą wierzchołów istnieje jakakolwiek ścieżka.

Stopień wierzchołka $\deg(v)$ to liczba krawędzi wychodzących z wierzchołka (incydentnych), przy czym pętle liczone są jako dwie krawędzie. *Najmniejszy stopień grafu* $\delta(G)$ to wielkość najmniejszego stopnia wierzchołka w grafie.

Największy stopień grafu $\Delta(G)$ to wielkość największego stopnia wierzchołka w grafie.

2.2. Wybrane rodziny grafów

Graf pełny K_n jest nieskierowanym grafem o n wierzchołkach, w którym każde dwa różne wierzchołki są ze sobą połączone krawędzią. Każdy graf pełny stanowi swoją największą klikę.

Graf cykliczny C_n jest to spójny graf nieskierowany, w którym wszystkie wierzchołki są stopnia drugiego. Własność ta sprawia, że powstaje jeden cykl biegnący przez wszystkie wierzchołki grafu. *Graf liniowy* P_n powstaje z grafu cyklicznego C_n przez usunięcie jednej dowolnej krawędzi.

Graf regularny stopnia k (k-regularny) jest grafem spójnym nieskierowanym, w którym wszystkie wierzchołki mają stopień równy k. Przykładami takich grafów są na przykład grafy kubiczne (3-regularne), czyli takie o wierzchołkach stopnia 3. Również wspomniany wcześniej graf pełny K_n można potraktować jako graf $(n-1)$-regularny.

Grafami planarnymi nazywamy wszystkie grafy, które da się narysować na płaszczyźnie w taki sposób, aby żadne dwie krawędzie nie przecinały się ze sobą. Po przeniesieniu takiego grafu na płaszczyznę, poprzez odwzorowanie wierzchołków w punkty o konkretnych współrzędnych, powstaje graf płaski.

Graf kołowy W_n powstaje poprzez połączenie nowego pojedynczego wierzchołka bezpośrednio z każdym wierzchołkiem grafu cyklicznego C_{n-1}. Każdy graf kołowy jest jednocześnie grafem planarnym i grafem Hamiltona.

Graf dwudzielny to graf nieskierowany, którego wierzchołki można podzielić na dwa rozłączne zbiory w taki sposób, że każda krawędź łączyć będzie dwa wierzchołki jednocześnie z obu tych zbiorów. Wynikającą z tego właściwością grafu dwudzielnego jest to, że nie zawiera on nieparzystych cykli. Graf pełny dwudzielny $K_{p,q}$ jest grafem dwudzielnym, w którym każda para wierzchołków z różnych zbiorów jest połączona krawędzią. Innymi słowy jest on maksymalnym pod względem liczby krawędzi grafem dwudzielnym prostym.

Graf cięciwowy to graf nieskierowany spójny charakteryzujący się tym, że każdy jego cykl składający się z co najmniej czterech wierzchołków posiada cięciwę. Cięciwa jest to krawędź nie należąca do cyklu, która łączy dwa niesąsiednie wierzchołki z cyklu. Powstają w ten sposób mniejsze podcykle, z których najmniejsze zawierają dokładnie trzy wierzchołki.

2.3. Kolorowanie grafu

Kolorowaniem grafu w ogólności nazywamy przyporządkowanie poszczególnym elementom grafu kolorów (etykiet) w taki sposób, aby żadne dwa sąsiadujące ze sobą elementy nie otrzymały takiego samego koloru. W praktyce kolory oznacza się liczbami naturalnymi. Zwykle poszukuje się najmniejszej liczby kolorów, która daje poprawne kolorowanie. Niestety jest to problem NP-trudny i jest mała szansa na znalezienie algorytmu działającego w czasie

wielomianowym. Oto najbardziej popularne rodzaje kolorowań, podzielone ze względu na typ elementów jakich dotyczą:

1. *Kolorowanie wierzchołków*, nazywane też klasycznym, odnosi się do nadawania wierzchołkom grafu kolorów w taki sposób, że żadne dwa połączone tą samą krawędzią wierzchołki nie mają tego samego koloru. *Liczba chromatyczna* grafu G to liczba $\chi(G)$ określająca najmniejszą liczbę kolorów potrzebnych do wierzchołkowego pokolorowania grafu G.
2. *Kolorowanie krawędzi* polega na przydzieleniu krawędziom kolorów zgodnie z zasadą, że żadne dwie współdzielące ze sobą wierzchołek krawędzie nie otrzymają tego samego koloru. *Indeks chromatyczny* grafu G to liczba $\chi'(G)$ określająca najmniejszą liczbę kolorów potrzebnych do krawędziowego pokolorowania grafu.
3. *Kolorowanie ścian*, które odnosi się do grafów planarnych, przypisuje kolory ścianom tak, aby każde dwie współdzielące krawędź ściany różniły się kolorem.

2.4. Zbiory niezależne

Zbiorem niezależnym nazywamy taki podzbiór wierzchołków grafu, spośród których żadne dwa nie są połączone krawędzią. Znalezienie *największego zbioru niezależnego* ze względu na liczność okazuje się być dość dużym wyzwaniem i podobnie jak w przypadku optymalnego kolorowania, należy do klasy problemów NP-trudnych. Czasami rozważa się również *maksymalne zbiory niezależne* pod względem inkluzji, czyli takie dla których nie istnieje żaden większy zbiór niezależny zawierający je w całości. Znalezienie tych takich zbiorów jest prostsze i można je wykonać w czasie wielomianowym.

2.5. Zbiory dominujące

Zbiorem dominującym grafu nazywamy podzbiór wierzchołkow wybrany w taki sposób, że każdy wierzchołek spoza tego zbioru sąsiaduje z pewnym wierzchołkiem ze zbioru dominującego. W wielu sytuacjach pojawia się potrzeba znalezienia *najmniejszego zbioru dominującego* dla danego grafu G. Jego liczność opisuje liczba $\gamma(G)$, która nazywana jest *liczbą dominowania*.

Na uwagę zasługują również *zbiory totalnie dominujące*, czyli takie w których każdy wierzchołek ma co najmniej jednego sąsiada z tego zbioru. Liczebność natomiast opisuje liczba $\gamma_t(G)$, którą nazywamy *liczbą totalnego zdominowania*.

2.6. Pokrycia wierzchołkowe

Pokryciem wierzchołkowym nazywamy taki podzbiór wierzchołków, że każda krawędź w danym grafie ma przynajmniej jeden koniec należący do tego zbioru. *Najmniejszym pokryciem wierzchołkowym* grafu G nazywamy pokrycie o najmniejszej liczności i oznaczamy przez $\tau(G)$. Problem znalezienia

najmniejszego pokrycia wierzchołkowego jest NP-trudny. Można zauważyć, że pokrycie wierzchołkowe jest jednocześnie pewnym zbiorem dominującym grafu.

2.7. Skojarzenia

Skojarzeniem w grafie nazywamy podzbiór jego krawędzi, w którym żadne dwie krawędzie nie współdzielą ze sobą wierzchołka. Wyróżnia się kilka rodzajów skojarzeń:

1. *Skojarzenie maksymalne* - jest to skojarzenie maksymalne pod względem inkluzji. Wynika z tego, że dodanie do niego jakiejkolwiek nowej krawędzi sprawi, że zbiór przestanie być skojarzeniem.
2. *Skojarzenie największe* - jest to skojarzenie, które dla danego grafu ma największą liczność.
3. *Skojarzenie doskonałe* - jest to takie skojarzenie, że każdy wierzchołek grafu jest końcem pewnej krawędzi należącej do skojarzenia.
4. *Skojarzenie prawie doskonałe* - to takie skojarzenie, w którym dokładnie jeden wierzchołek pozostaje nie skojarzony.

2.8. Dekompozycja drzewowa

Dekompozycja drzewowa (ang. *tree decomposition*) grafu $G = (V, E)$ jest to para (X, T), gdzie $X = \{X_i : i \in I\}$ jest rodziną podzbiorów $V(G)$ (są to *worki*, ang. *bags*), oraz $T = (I, F)$ jest drzewem, która spełnia następujące warunki [8]:

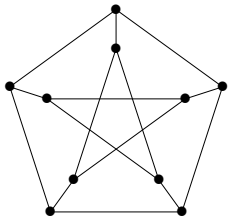
— Suma worków X_i jest równa $V(G)$.
— Dla każdej krawędzi $st \in E(G)$ istnieje indeks $i \in I$, taki że wierzchołki $s, t \in X_i$.
— Dla każdego wierzchołka $v \in V(G)$, worki zawierające v tworzą spójne poddrzewo w drzewie T (dokładniej mówiąc to indeksy worków tworzą spójne poddrzewo, bo indeksy są wierzchołkami w T).

Szerokość dekompozycji drzewowej (X, T) to rozmiar największego worka X_i w X minus 1. *Szerokość drzewowa* $\mathrm{tw}(G)$ grafu G to minimalna szerokość po wszystkich dekompozycjach drzewowych grafu. Szerokość drzewowa jest miarą tego, jak bardzo dany graf przypomina drzewo. Duże znaczenie mają grafy z ograniczoną szerokością drzewową (ang. *bounded treewidth*), ponieważ wiele problemów trudnych w ogólnym przypadku można rozwiązać w czasie wielomianowym dla tych grafów.

Szerokość drzewowa wybranych klas grafów.

— Dla drzewa $\mathrm{tw}(T) = 1$ (to jest przyczyna odejmowania jedynki od rozmiaru największego worka).
— $\mathrm{tw}(G) = 2$, jeżeli G jest 2-drzewem, częściowym 2-drzewem, cyklem C_n, grafem szeregowo-równoległym, grafem zewnętrznoplanarnym. Wszystkie te grafy są planarne.
— $\mathrm{tw}(G) = 3$, jeżeli G jest 3-drzewem, częściowym 3-drzewem, grafem kołowym W_n, grafem Halina, siecią Apoloniusza. Nie wszystkie te grafy są pla-

narne. Przykładowo dla grafu pełnego dwudzielnego mamy $\mathrm{tw}(K_{3,3}) = 3$, a nie jest to graf planarny (twierdzenie Kuratowskiego).
— Jeżeli G jest planarnym grafem kratą $n \times n$ (ang. *grid graph*), to wtedy $\mathrm{tw}(G) = n$. Jest to zaskakujące, bo największa klika dla tego grafu to K_2.
— Dla grafu Petersena $\mathrm{tw}(G) = 4$ (rysunek 2.1).
— Dla grafu pełnego $\mathrm{tw}(K_n) = n - 1$.
— Dla grafu pełnego dwudzielnego $\mathrm{tw}(K_{p,q}) = \min(p, q)$.



Rysunek 2.1. Graf Petersena.

2.9. k-drzewa i częściowe k-drzewa

k-drzewo (ang. *k-tree*) jest to graf nieskierowany $G = (V, E)$ określony rekurencyjnie w następujący sposób ($n > k$) [9].
(1) Graf pełny K_{k+1} jest k-drzewem (mamy $k + 1$ wierzchołków).
(2) Niech będzie dane k-drzewo H z n wierzchołkami. Budujemy k-drzewo G z $n + 1$ wierzchołkami przez wybranie k wierzchołków z $V(H)$ tworzących k-klikę, a następnie przez połączenie tych wierzchołków z nowym wierzchołkiem [powstaje $(k + 1)$-klika].
Zauważmy, że jeżeli $|V(G)| = n$, to $|E(G)| = nk - k(k + 1)/2$.

k-drzewa są maksymalnymi grafami z szerokością drzewową równą k, czyli dodanie do nich nowej krawędzi zwiększa szerokość drzewową. k-drzewa są grafami cięciwowymi. Planarne 3-drzewa to inaczej sieci Apoloniusza.

Częściowe k-drzewa (ang. *partial k-trees*) są to podgrafy k-drzew i mają szerokość drzewową co najwyżej k. Do częściowych 2-drzew należą grafy zewnętrznoplanarne i grafy szeregowo-równoległe. Do częściowych 3-drzew należą grafy Halina.

Podana definicja k-drzew wyznacza pewne uporządkowanie wierzchołków dodawanych do mniejszego k-drzewa. Kolejność pierwszych k wierzchołków jest dowolna. Po odwróceniu tej kolejności otrzymujemy *uporządkowanie eliminacji* (ang. *elimination ordering*), w którym każdy wierzchołek ma co najwyżej k sąsiadów o wyższych numerach. Wykorzystuje się to do rozpoznawania k-drzew i częściowych k-drzew. Uporządkowanie eliminacji jest *doskonałe* (ang. *perfect*), jeżeli każdy wierzchołek razem z sąsiadami o wyższych nume-

rach tworzy klikę. Takie uporządkowanie występuje dla pełnych k-drzew, a także występuje w kontekście grafów cięciwowych (ang. *chordal graphs*).

2.10. Grafy szeregowo-równoległe

Grafy szeregowo-równoległe (ang. *series-parallel graphs*) [10] są definiowane na kilka sposobów. W ogólności grafy szeregowo-równoległe mogą zawierać krawędzie równoległe, a więc mogą być multigrafami. Dla prostoty będziemy używać krótszej nazwy występującej w literaturze, mianowicie *sp-grafy* (ang. *sp-graphs*). Podamy definicję wprowadzoną przez Davida Eppsteina [11], który używa nazwy *grafy z dwoma końcówkami* (ang. *two-terminal graphs*).

Definicja I: W grafie nieskierowanym G wyróżnia się dwa różne wierzchołki: źródło s (ang. *source*) i zlew t (ang. *sink*), $s \neq t$. Dalsza definicja jest rekurencyjna.
(1) Pojedyńcza krawędź st jest sp-grafem (graf pełny K_2).
(2) *Połączenie równoległe* dwóch sp-grafów G_1, G_2 daje sp-graf G,
$s = s_1 = s_2$, $t = t_1 = t_2$.
(3) *Połączenie szeregowe* dwóch sp-grafów G_1, G_2 daje sp-graf G,
$s = s_1$, $t = t_2$, $t_1 = s_2$.
W nowszej literaturze dodaje się jeszcze jeden typ operacji, ale jedynie w przypadku sp-grafów nieskierowanych.
(3) *Połączenie jackknife* dwóch sp-grafów G_1, G_2 daje sp-graf G,
$s = s_1$, $t = t_1$, $t_1 = s_2$.

Podana definicja jest *globalna*, to znaczy mówi o łączeniu dużych struktur, przez co nie jest wygodna do użycia. Przykładowo rozpoznawanie sp-grafów wymagałoby rekurencyjnego badania coraz mniejszych grafów użytych przy połączeniach, co komplikowałoby projektowanie wydajnego algorytmu [12].

Definicja II: Graf nieskierowany jest sp-grafem, jeżeli może być sprowadzony do grafu K_2 przez sekwencję następujących operacji [13]:
(1) Zamiana pary krawędzi równoległych przez jedną krawędź, która łączy ich wspólne końce (połączenie równoległe).
(2) Zamiana pary krawędzi ux, xv stykających się w wierzchołku x stopnia 2, innego niż s lub t, przez jedną krawędź uv (połączenie szeregowe).
(3) Zamiana pary krawędzi ux, xv stykających się w wierzchołku x, przez jedną krawędź ux, czyli usunięcie krawędzi xv (połączenie *jackknife*). Stopień wierzchołka v musi być równy jeden.

Po wykonaniu pierwszej operacji *jackknife* musimy zaznaczyć, że odtąd $t = x$. Wydaje się, że operację *jackknife* wykonuje się w ostateczności, kiedy nie ma wierzchołków stopnia 2 w grafie i nie można wykonać pozostałych dwóch operacji.

Ta definicja jest *lokalna*, ponieważ operuje na sąsiednich krawędziach. Jest to podejście wygodniejsze do użycia przy implementacji algorytmów związanych z sp-grafami. Pewna niewygoda w stosowaniu tej definicji polega

na użyciu krawędzi równoległych i multigrafów, nawet jeżeli wejściowy graf jest grafem prostym. Dlatego stosuje się jeszcze inną definicję.

Definicja III: Graf nieskierowany jest sp-grafem bez krawędzi równoległych (graf prosty), jeżeli może być zbudowany na bazie następujących reguł, w których x oznacza nowy wierzchołek [12]:
(1) Pojedyńcza krawędź st jest sp-grafem.
(2) Do krawędzi uv mogą być dodane dwie krawędzie ux i xv (połączenie równoległe).
(3) Krawędź uv może być zastąpiona przez dwie krawędzie ux i xv (połączenie szeregowe).
(4) Do krawędzi ut (t to jest zlew) może być dodana krawędź tx (połączenie *jackknife*).

Tak określone sp-grafy są rzadkie, czyli że liczba krawędzi jest rzędu $O(V)$. Można indukcyjnie udowodnić, że $m \leqslant 2n - 3$.

Grafy szeregowo-równoległe skierowane: W literaturze badane są również sp-grafy skierowane, gdzie krawędzie mają kierunek od źródła do zlewu. Definicja tych grafów jest prostym przekształceniem podanych wcześniej definicji sp-grafów nieskierowanych [12]. Grafy te są planarne i acykliczne. Jednym z problemów badanych dla sp-grafów skierowanych jako sieci przepływowych jest problem maksymalnego przepływu. Twierdzenie mówi, że dowolny przepływ blokujący w sp-grafie skierowanym jest przepływem maksymalnym.

Twierdzenie (Duffin, 1965): Grafy szeregowo-równoległe nie mają podgrafu homeomorficznego do grafu K_4 [13]. Poniżej podamy kilka twierdzeń charakteryzujących pewne rodziny grafów poprzez minory.

Twierdzenie: Graf G jest grafem zewnętrznoplanarnym wtedy i tylko wtedy, gdy G nie zawiera grafów K_4 i $K_{2,3}$ jako minoru [14].

Twierdzenie (Bodlaender, 1998): Graf G ma $\mathrm{tw}(G) = 2$ wtedy i tylko wtedy, gdy G nie zawiera grafu K_4 jako minoru. Jest to twierdzenie 17 z pracy [15]. Uogólnienie tego twierdzenia mówi, że dla każdego ustalonego k, zbiór grafów mających szerokość drzewową co najwyżej równą k może być określony przez skończony zbiór minorów zabronionych.

Twierdzenie (Arnborg, Proskurowski, Corneil, 1990): Graf G ma $\mathrm{tw}(G) = 3$ wtedy i tylko wtedy, gdy G nie zawiera jako minoru następujących

czterech grafów: K_5, 3-antypryzma, 5-pryzma, graf Wagnera [16]. Wszystkie te cztery grafy mają szerokość drzewową równą cztery.

Dekompozycja drzewowa: sp-grafy mają szerokość drzewową co najwyżej 2. Maksymalne sp-grafy to 2-drzewa. sp-grafy mogą być scharakteryzowane poprzez *ear decompositions* [11].

Rozpoznawanie grafów szeregowo-równoległych: Algorytm liniowy rozpoznawania grafów został podany przez Valdesa, Tariana i Lawlera [17], [18]. Pokazano także jak uzyskać drzewo dekompozycji w czasie liniowym.

Zbiory niezależne: Problem znalezienia największego zbioru niezależnego w sp-grafie może być rozwiązany w czasie liniowym (metoda Borie).

Zbiory dominujące: Problem znalezienia najmniejszego zbioru dominującego w sp-grafie może być rozwiązany w czasie liniowym (metoda Borie).

Pokrycia wierzchołkowe: Problem znalezienia najmniejszego pokrycia wierzchołkowego w sp-grafie może być rozwiązany w czasie liniowym [19] (metoda Borie).

Skojarzenia: Problem znalezienia największego skojarzenia może być rozwiązany w czasie liniowym [19] (metoda Borie).

Kolorowanie wierzchołków: sp-grafy dwudzielne można rozpoznać w czasie liniowym, wtedy potrzebne są dwa kolory dla wierzchołków. Inne sp-grafy (te zawierające cykle nieparzyste) potrzebują dokładnie trzech kolorów dla wierzchołków. Wynika to z faktu, że sp-grafy są częściowymi 2-drzewami, a 2-drzewa można pokolorować zachłannie trzema kolorami podczas ich budowania wg definicji k-drzew. Algorytm kolorowania wierzchołków sp-grafu w czasie liniowym podamy w rozdziale 4.

Kolorowanie krawędzi: Problem kolorowania krawędzi nie jest trywialny. Pokazano, że indeks chromatyczny sp-grafów wynosi Δ, z wyjątkiem cykli nieparzystych [20].

Z konstrukcji sp-grafów wynika, że połączenie szeregowe pozwala na użycie $\Delta(G)$ kolorów, jeżeli łączone grafy wykorzystują $\Delta(G_1)$ i $\Delta(G_2)$ kolorów. Przy połączeniu równoległym nie musi tak być, o czym świadczy przykład łączenia równoległego dwóch grafów liniowych P_3 i P_4. Mamy $\Delta(P_3) = \Delta(P_4) = 2$, dla obu grafów są potrzebne dwa kolory krawędzi. Jako wynik dostajemy cykl nieparzysty z $\Delta(C_5) = 2$ i liczbą potrzebnych kolorów równą 3.

Algorytm liniowy kolorowania krawędzi multigrafów szeregowo-równoległych podali Zhou, Suzuki i Nishizeki [21]. Inny prosty algorytm tych autorów dzia-

Tabela 2.1. Liczba sp-grafów spójnych z n wierzchołkami. Osobno zaznaczono sp-grafy zbudowane z użyciem operacji *jackknife*.

n	1	2	3	4	5
spójne	1	1	2	6	21
sp-grafy	0	1	2	4	12
jackknife	0	0	0	1	3

ła w czasie $O(n\Delta^3)$. Algorytm liniowy sekwencyjny (i optymalny równoległy) kolorowania krawędzi k-drzew podali Zhou, Nakano i Nishizeki [22].

Kliki: Największa możliwa klika to K_3, ponieważ sp-grafy to częściowe 2-drzewa.

Ścieżka i cykl Hamiltona: Istnienie ścieżki lub cyklu Hamiltona w sp-grafach można wykryć w czasie liniowym [23].

Problem komiwojażera: sp-grafy mają najwyżej jeden cykl Hamiltona, więc problem komiwojażera sprowadza się do stwierdzenia, czy dany sp-graf posiada cykl Hamiltona (Sysło, 1983).

Problem ścieżek rozłącznych krawędziowo: Problem polega na stwierdzeniu, czy istnieje p parami rozłącznych krawędziowo ścieżek łączących p par wierzchołków. Okazuje się, że ten problem ma wydajne rozwiązanie dla grafów zewnętrznoplanarnych, natomiast jest NP-zupełny dla sp-grafów i częściowych 2-drzew [24].

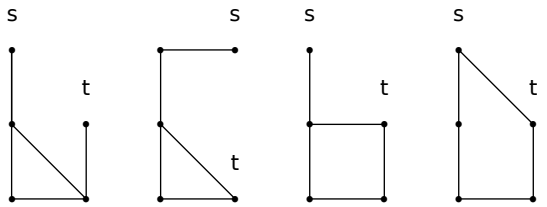
2.11. Galeria grafów szeregowo-równoległych

W tym rozdziale pokażemy rysunki najprostszych sp-grafów wykonane przy wykorzystaniu modułu Gnuplot. Liczby sp-grafów spójnych o danej liczbie wierzchołków zostały zebrane w tabeli 2.1.

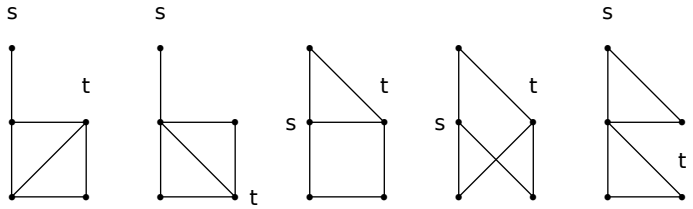
Dla $n = 2$ mamy jedną krawędź, czyli graf P_2. Dla $n = 3$ mamy ścieżkę P_3 i cykl C_3. Dla $n = 4$ mam cztery grafy: ścieżka P_4, cykl C_4, cykl C_4 z cięciwą, cykl C_3 z czwartym wierzchołkiem połączonym krawędzią z dowolnym wierzchołkiem cyklu.

Dla $n = 5$ mamy łącznie 12 sp-grafów, które przedstawimy wraz z rosnącą liczbą krawędzi. Jedyne drzewo z $m = 4$ to ścieżka P_5. Następnie mamy grafy z liczbą krawędzi równą $m = 5$ (rysunek 2.2), $m = 6$ (rysunek 2.3), $m = 7$ (rysunek 2.4). Na rysunkach zaznaczono wyróżnione wierzchołki s, t, przy czym czasem wybór nie jest jednoznaczny. Przykładowo cykl C_5 może być zbudowany jako połączenie równoległe P_2 i P_5 lub P_3 i P_4.

Osobno należy rozważyć sp-grafy zbudowane z użyciem operacji *jackknife*. Dla $n = 4$ jest to graf dwudzielny pełny $K_{1,3}$. Dla $n = 5$ są to dodatkowe trzy sp-grafy (rysunek 2.5).



Rysunek 2.2. Cztery sp-grafy z $n = 5$, $m = 5$.

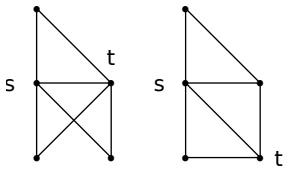


Rysunek 2.3. Pięć sp-grafów z $n = 5$, $m = 6$.

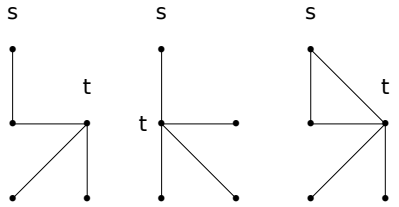
2.12. Grafy Halina

Szczegółowe informacje o grafach Halina można znaleźć w pracy magisterskiej Krzysztofa Krawczyka [25]. W tym rozdziale przypomnimy tylko najważniejsze definicje i fakty.

Graf Halina zapisuje się symbolicznie jako $H = T \cup C$, gdzie T oznacza drzewo z co najmniej czterema wierzchołkami i bez wierzchołków stopnia dwa, natomiast C jest cyklem łączącym kolejne liście drzewa na płaskim rysunku drzewa. Najmniejszym grafem Halina jest graf kołowy $W_4 = K_4$. Wszystkie wierzchołki grafu Halina mają stopień co najmniej trzy.



Rysunek 2.4. Dwa sp-grafy z $n = 5$, $m = 7$.



Rysunek 2.5. Trzy sp-grafy z $n = 5$ zbudowane z operacją *jackknife*.

3. Implementacja grafów

Aby możliwe było wydajne i jednocześnie czytelne ujęcie prezentowanych algorytmów oraz struktur w formę kodu języka Python potrzebna jest elegancka i praktyczna implementacja biblioteki bazowej. Przede wszystkim powinna ona ukrywać przed użytkownikiem niepotrzebne szczegóły techniczne, wystawiając na pierwszy plan zrozumiały i intuicyjny interfejs. Pozwala to zwiększyć zwięzłość kodu i co najważniejsze lepiej skoncentrować się na specyfice analizowanego problemu. Takie założenia spełnia pakiet graphtheory wykorzystany w pracy [7].

3.1. Interfejs biblioteki

Struktura biblioteki została rozbita na klasy odpowiadające zarówno poszczególnym elementom grafów, jak również samym algorytmom. Poniżej przedstawiono podstawowe klasy, które posłużą do budowania bardziej złożonych implementacji algorytmów.

— Klasy Edge i UndirectedEdge reprezentują krawędzie - odpowiednio skierowane i nieskierowane. Pola source oraz target oznaczają końce krawędzi, a ze zrozumiałych powodów w przypadku klasy UndirectedEdge zaniedbujemy ukierunkowanie. Krawędzie posiadają również wagi opisane przez pole weight. Domyślna waga krawędzi wynosi 1. W praktyce wygodniejsze jest używanie klasy Edge nawet w kontekście grafów nieskierowanych.
— Klasa Graph opiera się na słowniku i w dość ogólny sposób określa grafy proste skierowane i nieskierowane. Klasa zawiera dużą liczbę użytecznych metod ułatwiających budowanie, analizowanie jak i wyświetlanie grafów. Wybór słownikowego podejścia do przechowywania informacji o budowie grafu łączy zalety list sąsiedztwa (oszczędność pamięci) i macierzy sąsiedztwa (szybki dostęp do krawędzi).

Jeśli chodzi o sposób nazywania i późniejszego odwoływania się do wierzchołków, to implementacja zostawia swobodę użytkownikowi biblioteki. Dozwolone są liczby całkowite, stringi i inne obiekty hashowalne.

Do budowania grafów używać będziemy głównie metod: graph.add_node(node), graph.del_node(node), graph.add_edge(edge), graph.del_edge(edge), graph.copy().

Do przeglądania struktury grafu przydatne będą metody iterujące po wierzchołkach, krawędziach, sąsiadach: graph.iternodes(), graph.iteredges(), graph.iterinedges(source), graph.iteroutedges(source), graph.iteradjacent(source).

Na potrzeby algorytmów dostępne są również metody dynamicznie wyliczające parametry grafu, np. graph.v() (liczba wierzchołków), graph.e() (liczba krawędzi), graph.f() (liczba ścian grafu planarnego), graph.degree(node) (stopień wierzchołka).

3.2. Przykładowe obliczenia

Pierwszy przykład pokazuje sesję interaktywną z obliczeniami dla pewnego grafu nieskierowanego.

Listing 3.1. Obliczenia dla grafu nieskierowanego.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from connected import is_connected
>>> G = Graph(4)
>>> G.add_edge(Edge(1, 2))
>>> G.add_edge(Edge(2, 3))
>>> G.add_edge(Edge(2, 4))
>>> G.v()
4
>>> list(G.iternodes())
[1, 2, 3, 4]
>>> G.show()
1 : 2
2 : 1 3 4
3 : 2
4 : 2
>>> is_connected(G)
True
```

Następny przykład pokazuje obliczenia dla pewnego sp-grafu.

Listing 3.2. Obliczenia dla sp-grafu.

```
# Utworzenie przypadkowego sp-grafu.
>>> from spgen2 import make_random_spgraph
>>> G = make_random_spgraph(10)
# Wyznaczanie sp-tree.
>>> from sptree3 import find_sptree
>>> from spnodes import btree_print
>>> T = find_sptree(G)
>>> btree_print(T)
# Obliczanie najwiekszego zbioru niezaleznego.
>>> from spiset1 import SPIndependentSet
>>> algorithm = SPIndependentSet(G, T)
>>> algorithm.run()
>>> algorithm.independent_set
set(...)
# Obliczanie najmniejszego zbioru dominujacego.
>>> from spdset1 import SPDominatingSet
>>> algorithm = SPDominatingSet(G, T)
>>> algorithm.run()
>>> algorithm.dominating_set
set(...)
# Obliczanie najmniejszego pokrycia wierzcholkowego.
>>> from spcover1 import SPNodeCover
>>> algorithm = SPNodeCover(G, T)
>>> algorithm.run()
>>> algorithm.node_cover
set(...)
# Obliczanie najwiekszego skojarzenia.
>>> from spmate1 import SPMatching
```

```
>>> algorithm = SPMatching(G, T)
>>> algorithm.run()
>>> algorithm.mate_set          # zbior krawedzi
set(...)
# Wyznaczanie kolorowania wierzcholkow.
>>> from spcolor1 import SPNodeColoring
>>> algorithm = SPNodeColoring(G)
>>> algorithm.run()
>>> algorithm.color             # dict
# Wyznaczanie PEO.
>>> from spgen2 import find_peo_spgraph
>>> peo = find_peo_spgraph(G) # list
>>> print peo
```

4. Algorytmy

Grafy szeregowo-równoległe i grafy Halina są przykładami grafów o ograniczonej szerokości drzewowej. Można to wykorzystać do znalezienia wydajnych algorytmów rozwiązujących problemy, które są trudne w przypadku ogólnych grafów. Zwykle stosowana jest technika programowania dynamicznego.

Wśród problemów trudnych występują problemy o *naturze lokalnej*, dla których poprawność rozwiązania można sprawdzić przeglądając otoczenie każdego wierzchołka. Przykładami są problem największego zbioru niezależnego, najmniejszego zbioru dominującego, najmniejszego pokrycie wierzchołkowego, kolorowania wierzchołków. Te problemy dają się efektywnie rozwiązać dla grafów o ograniczonej szerokości drzewowej. Wykorzystuje się dekompozycję drzewową, albo specyficzne algorytmy dopasowane do danej rodziny grafów. W tym rozdziale pokażemy przykłady takich algorytmów.

Istnieją problemy trudne o *naturze globalnej*, które również można wydajnie rozwiązać dla grafów o ograniczonej szerokości drzewowej. Przykładem jest problem komiwojażera, który ma wydajne rozwiązanie dla grafów Halina. Tutaj stosuje się technikę programowania dynamicznego i trzeba śledzić wszystkie sposoby, na które rozwiązania mogą przechodzić przez worek w drzewie dekompozycji. Istotna jest ograniczona liczba stanów, które trzeba zapamiętać przy każdym worku.

Wreszcie istnieją problemy o naturze globalnej, które pozostają trudne dla grafów o ograniczonej szerokości drzewowej. Jako cechy charakterystyczne takich problemów wymienia się (1) dużą liczbę możliwych stanów każdego wierzchołka, (2) zdefiniowanie problemu dla krawędzi zamiast dla wierzchołków, czy (3) istnienie oddziaływań w problemie nie uchwyconych przez model grafu [26]. Jest to obszar aktywnych badań.

4.1. Generowanie k-drzew przypadkowych

Do testowania algorytmów obliczających m.in. szerokość drzewową grafu potrzebne są grafy o znanych szerokościach. Dlatego stworzono generator k-drzew przypadkowych. Aby otrzymać częściowe k-drzewo przypadko-

we należy usuwać krawędzie k-drzewa z ustalonym prawdopodobieństwem $0 \leqslant p \leqslant 1$ [27].

Dane wejściowe: Dwie liczby naturalne, liczba wierzchołków grafu n, szerokość drzewowa $k < n$.

Problem: Generowanie k-drzew przypadkowych.

Opis algorytmu: Algorytm rozpoczynamy od stworzenia $(k+1)$-kliki z wierzchołków od $n-k-1$ do $n-1$. Następnie wybieramy przypadkową k-klikę i łączymy ją z nowym wierzchołkiem. Ten etap powtarzamy dla wierzchołków o numerach malejących od $n-k-2$ do 0.

Złożoność: Złożoność czasową algorytmu szacujemy na $O(nk)$. Testy wydajnościowe zostały opisane w dodatku A.4.

Uwagi: Końcowe k-drzewo ma uporządkowanie eliminacji w postaci rosnącego ciągu wierzchołków od 0 do $n-1$. Warto jeszcze zwrócić uwagę na sposób losowania k-kliki. Najpierw losowana jest duża $(k+1)$-klika przy wykorzystaniu częściowego uporządkowania eliminacji. Następnie z dużej kliki jest losowana mała k-klika przez usunięcie przypadkowego wierzchołka.

Listing 4.1. Generator k-drzew przypadkowych.

```
def make_random_ktree(n, k):
    """Make a random k-tree with n vertices."""
    if k >= n:
        raise ValueError("bad k")
    graph = Graph(n)
    if n < 1:
        raise ValueError("bad n")
    elif n == 1:
        graph.add_node(0)
    else:
        for node in xrange(n):
            graph.add_node(node)
        # Make {n-k-1, ..., n-1} into (k+1)-clique in graph.
        for source in xrange(n-k-1, n):
            for target in xrange(n-k-1, n):
                if source < target:
                    graph.add_edge(Edge(source, target))
        node = n-k-2
        while node >= 0:
            # Wybor source z przedzialu od node+1 do n-k-1.
            # To jest jakby wybor duzej (k+1)-kliki,
            source = random.choice(xrange(node+1, n-k))
            # Teraz zbieram wierzcholki tej kliki, ale one maja byc
            # wieksze od source, wieksze numery!
            neighbors = list(target for target in graph.iteradjacent(source)
                             if source < target)
            neighbors.append(source)    # closed neighborhood
            # Z duzej (k+1)-kliki wybieram mala k-klike.
            idx = random.randrange(0, len(neighbors))
            swap(neighbors, idx, -1)
```

```
            neighbors.pop()
            # Connect node to all nodes from neighbors.
            for target in neighbors:
                graph.add_edge(Edge(node, target))
            node -= 1
    return graph
```

4.2. Generowanie grafów szeregowo-równoległych

Generator przypadkowych sp-grafów nieskierowanych stosuje w praktyce definicję III z rozdziału 2. Generator został przygotowany w dwóch wersjach. Pierwsza wersja [funkcja make_random_spgraph(n)] zwraca instancję klasy Graph. Druga wersja [funkcja make_random_sptree(n)] zwraca binarne sp-drzewo, dokładniej łącze do korzenia. Druga wersja generatora jest wygodniejsza przy testowaniu algorytmów rozwiązujących problemy trudne, jeżeli algorytmy korzystają z rekurencyjnej budowy sp-grafów. Nie musimy wykonywać etapu przekształcania sp-grafu na sp-drzewo.

Dane wejściowe: Liczba wierzchołków grafu $n > 1$.

Problem: Generowanie przypadkowych sp-grafów nieskierowanych.

Opis algorytmu: Algorytm w pierwszej fazie wykonuje operacje z definicji III na rosnącej liście krawędzi. Ustalone jest źródło $s = 0$ i zlew $t = n - 1$. W każdym przebiegu pętli **while** przypadkowa krawędź jest poddawana losowej operacji z podanego zestawu, a przy tym kolejny wierzchołek jest dołączany do grafu. Operacja *jackknife* może być zastosowana jedynie wtedy, gdy koniec wybranej krawędzi jest zlewem.

Złożoność: Złożoność czasową algorytmu szacujemy na $O(n)$, ponieważ pętla **while** wykona się $n-2$ razy, a operacje w jednym przebiegu pętli zajmują stały czas $O(1)$. Liczba krawędzi dodawanych do grafu jest rzędu $O(n)$. Testy wydajnościowe zostały opisane w dodatku A.5.

Uwagi: Podany algorytm może być użyty do generowania sp-grafów skierowanych, należy jedynie usunąć operację *jackknife* z zestawu losowanych operacji.

Listing 4.2. Generator sp-grafów przypadkowych.

```
def make_random_spgraph(n):
    """Make a random sp-graph with n vertices."""
    if n < 2:
        raise ValueError("bad n")
    graph = Graph(n)
    for node in xrange(n):
        graph.add_node(node)
    source = 0
    sink = n-1
    edge_list = [Edge(source, sink)]
```

```
    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.
        if edge.target == sink:
            action = random.choice(["series", "parallel", "jackknife"])
        else:
            action = random.choice(["series", "parallel"])
        if action == "series":
            edge_list.pop()
            edge_list.append(Edge(edge.source, node))
            edge_list.append(Edge(node, edge.target))
        elif action == "parallel":
            edge_list.append(Edge(edge.source, node))
            edge_list.append(Edge(node, edge.target))
        elif action == "jackknife":
            edge_list.append(Edge(edge.target, node))
        node -= 1
    for edge in edge_list:
        graph.add_edge(edge)
    return graph
```

4.3. Znajdowanie PEO dla grafu szeregowo-równoległego

Graf szeregowo-równoległy jest częściowym 2-drzewem i czasem potrzebne jest wyznaczenie pełnego 2-drzewa zawierającego dany sp-graf. Takie pełne 2-drzewo jest grafem cięciwowym i jest *uzupełnieniem cięciwowym* (ang. *chordal completion*) danego sp-grafu. Przedstawimy algorytm [funkcja find_peo_spgraph()] wyznaczający PEO tego pełnego 2-drzewa, które jest grafem cięciwowym [28]. PEO będzie wykorzystane przy kolorowaniu wierzchołków sp-grafu.

Algorytm polega na sukcesywnym usuwaniu z grafu wierzchołków stopnia dwa (2-liście) i dodawaniu ich do listy wierzchołków tworzącej PEO. Jeżeli dwaj sąsiedzi usuwanego wierzchołka nie są połączeni krawędzią, to dodajemy taką krawędź. Po usunięciu wszystkich 2-liści możliwe są trzy przypadki.

W pierwszym przypadku pozostanie jedna krawędź, a jej dwa końce stopnia jeden wyznaczają źródło s i zlew t sp-grafu. Dany sp-graf nie zawiera operacji *jackknife*. Wierzchołki s i t będą na końcu PEO.

W drugim przypadku pozostanie graf gwiazda z $k > 3$ wierzchołkami. Wtedy środek gwiazdy stopnia $k-1$ jest zlewem t, a jeden z jej końców stopnia jeden będzie źródłem s. Dany sp-graf zawiera $k-2$ operacje *jackknife*. Wierzchołki s i t będą na końcu PEO.

Wreszcie trzeci przypadek to pozostanie innego zestawu wierzchołków, co oznacza, że wejściowy graf nie jest sp-grafem. Warto zauważyć, że funkcja find_peo_spgraph() może być użyta do rozpoznawania sp-grafów.

Przygotowaliśmy dwie implementacje algorytmu wyznaczającego PEO, które różnią się sposobem zarządzania zbiorem wierzchołków stopnia 2 podczas usuwania wierzchołków z grafu. W pierwszej wersji wierzchołki przemieszczane są między bukietami odpowiadającymi stopniom wierzchołków. W drugiej wersji monitoruje się tylko zbiór wierzchołków stopnia 2, ale trzeba sprawdzać, czy stopień 2 jest dalej aktualny. Listing 4.3 prezentuje pierwszą wersję. Złożoność algorytmu szacujemy na $O(n)$, ponieważ przetwarzanie każdego wierzchołka odbywa się w stałym czasie. Testy wydajnościowe zostały opisane w dodatku A.6.

Listing 4.3. Wyznaczanie PEO dla sp-grafów.

```
def find_peo_spgraph(graph):    # graph has to be connected
    """Find PEO for a supergraph (2-tree) of an sp-graph."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    order = list()      # PEO of 2-tree
    graph_copy = graph.copy()
    degree_dict = dict((node, graph.degree(node))
        for node in graph.iternodes())                # O(V) time
    bucket = list(set() for deg in xrange(graph.v()))    # O(V) time
    for node in graph.iternodes():    # wstawiam do kubelkow, O(V) time
        bucket[graph.degree(node)].add(node)
    # Dopoki sa wierzcholki stopnia 2 wykonuj odrywanie.
    deg = 2
    while bucket[deg]:
        source = bucket[deg].pop()
        order.append(source)
        node1, node2 = list(graph_copy.iteradjacent(source))
        edge = Edge(node1, node2)
        if graph_copy.has_edge(edge):
            # Jezeli ma krawedz, to trzeba poprawic stopnie wierzcholkow,
            # bo przy usuwaniu krawedzi przy source zmniejsza sie stopnie.
            deg1 = degree_dict[node1]    # stary stopien
            bucket[deg1].remove(node1)
            bucket[deg1-1].add(node1)
            degree_dict[node1] = deg1-1    # nowy stopien
            deg2 = degree_dict[node2]    # stary stopien
            bucket[deg2].remove(node2)
            bucket[deg2-1].add(node2)
            degree_dict[node2] = deg2-1    # nowy stopien
        else:    # tu nie trzeba poprawiac stopni
            graph_copy.add_edge(edge)
        # Usuwamy krawedzie z source.
        graph_copy.del_edge(Edge(source, node1))
        graph_copy.del_edge(Edge(source, node2))
    # Sprawdzamy co zostalo.
    len1 = len(bucket[1])
    if len1 == 2 and len(order) + len1 == graph.v():
        # Zostala jedna krawedz, dodajemy konce do PEO.
        order.append(bucket[1].pop())
        order.append(bucket[1].pop())
    elif len(bucket[len1]) == 1 and len(order) + len1 + 1 == graph.v():
        # Zostala gwiazda, jest jackknife.
        while bucket[1]:
            order.append(bucket[1].pop())
```

```
            order.append(bucket[len1].pop())
        else:
            raise ValueError("not an sp-graph")
        return order
```

4.4. Rozpoznawanie grafów szeregowo-równoległych

Algorytm rozpoznawania sp-grafu dzieli się na kilka etapów. W pierwszym etapie następuje redukcja danego grafu przez usuwanie 2-liści, analogicznie jak w algorytmie wyznaczania PEO z funkcji find_peo_spgraph(). W drugim etapie następuje analiza pozostałych wierzchołków i dokończenie redukcji do postaci jednej krawędzi. Jeżeli badany graf nie jest sp-grafem, to algorytm rzuca wyjątek.

W obu tych etapach na stosie zapisywane są wykonywane operacje, usuwane wierzchołki, oraz ich sąsiedzi. Dla operacji *jackknife* zapisujemy usuwany wierzchołek, źródło i zlew (są już rozpoznane na tym etapie).

W trzecim etapie algorytmu następuje budowa sp-drzewa na bazie zapisanych wcześniej operacji. Tutaj kod przypomina generator przypadkowego sp-drzewa z funkcji make_random_sptree(). Korzeń sp-drzewa powstaje z krawędzi łączącej źródło i zlew. Dalej ze stosu pobierane są kolejne zapisy operacji. Pomocniczy słownik tnode_dict zawiera tylko węzły odpowiadające krawędziom budowanego sp-grafu i pomaga znaleźć węzły podlegające przekształceniom. Na końcu zwracany jest korzeń wyznaczonego sp-drzewa.

Złożoność czasową algorytmu szacujemy na $O(n)$. Testy wydajnościowe zostały opisane w dodatku A.7.

Listing 4.4. Rozpoznawanie sp-grafów.

```
def find_sptree(graph):   # graph has to be connected
    """Find an sp-tree for an sp-graph."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    graph_copy = graph.copy()
    degree2 = set(node for node in graph.iternodes()
        if graph.degree(node) == 2)   # active nodes with degree 2
    call_stack = []
    tnode_dict = dict()
    # Etap I. Redukcja grafu do krawedzi lub gwiazdy.
    # Dopoki sa wierzcholki stopnia 2 wykonuj odrywanie.
    while degree2:
        source = degree2.pop()
        if graph_copy.degree(source) != 2:
            # Czasem stopien wierzcholka moze sie zmniejszyc!
            continue
        node1, node2 = tuple(graph_copy.iteradjacent(source))
        edge = Edge(node1, node2)
        if graph_copy.has_edge(edge):
            # Jezeli ma krawedz, to trzeba poprawic stopnie wierzcholkow,
            # bo przy usuwaniu krawedzi przy source zmniejsza sie stopnie.
            if graph_copy.degree(node1) == 3:
                degree2.add(node1)
            if graph_copy.degree(node2) == 3:
```

```
                degree2.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(source, node1))
        graph_copy.del_edge(Edge(source, node2))
    # Etap II. Sprawdzamy co zostalo.
    degree1 = set(node for node in graph_copy.iternodes()
        if graph_copy.degree(node) == 1)
    if len(degree1) == 2 and len(call_stack) + 2 == graph.v():
        # Zostala jedna krawedz, dodajemy konce do PEO.
        node1 = degree1.pop()
        node2 = degree1.pop()
        root = Node(node1, node2, "edge")
        tnode_dict[(node1, node2)] = root
    elif len(call_stack) + len(degree1) + 1 == graph.v():
        # Zostala gwiazda, jest jackknife.
        # Szukam centrum gwiazdy.
        for node in graph_copy.iternodes():
            deg = graph_copy.degree(node)
            if deg > 1:
                if deg == len(degree1):
                    sink = node   #  centum gwiazdy
                    break
                else:
                    raise ValueError("not an sp-graph")
        # Trzeba ustalic jedno ramie jako koniec, w parze do centrum.
        source = degree1.pop()
        while degree1:
            call_stack.append(("jackknife", degree1.pop(), source, sink))
        root = Node(source, sink, "edge")
        tnode_dict[(source, sink)] = root
    else:
        raise ValueError("not an sp-graph")
    # 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)]
        elif (node2, node1) in tnode_dict:
            tnode = tnode_dict[(node2, node1)]
        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":
            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
        elif action == "jackknife":
            tnode.type = "jackknife"
            tnode.left = Node(tnode.source, tnode.target, "edge")
            tnode.right = Node(tnode.target, node, "edge")
            tnode_dict[(tnode.source, tnode.target)] = tnode.left
            tnode_dict[(tnode.target, node)] = tnode.right
        else:
            raise ValueError("bad action")
    return root
```

4.5. Kolorowanie wierzchołków grafu szeregowo-równoległego

Algorytm kolorowania wierzchołków sp-grafu w pierwszym etapie sprawdza, czy sp-graf nie jest dwudzielny, co w przypadku grafu dwudzielnego od razu daje 2-kolorowanie. Jeżeli sp-graf nie jest dwudzielny, to wyznaczane jest PEO 2-drzewa, którego podgrafem jest dany sp-graf. Przy okazji następuje rozpoznawanie sp-grafu. Następnie wierzchołki sp-grafu są 3-kolorowane zachłannie w kolejności odwrotnej do PEO. Złożoność obliczeniowa algorytmu jest liniowa $O(n)$.

Listing 4.5. Kolorowanie wierzchołków sp-grafu.

```
class SPNodeColoring:
    """Find sp-graph node coloring."""

    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._color_list = [False] * self.graph.v()

    def run(self):
        """Executable pseudocode."""
        try:
            algorithm = BipartiteGraphBFS(self.graph)
            algorithm.run()
            self.color = algorithm.color
        except ValueError:
            order = find_peo_spgraph(self.graph)
            for source in reversed(order):
                self._greedy_color(source)

    def _greedy_color(self, source):
        """Give node the smallest possible color."""
        for target in self.graph.iteradjacent(source):
```

```
            if self.color[target] is not None:
                self._color_list[self.color[target]] = True
        for c in xrange(self.graph.v()):    # check colors
            if not self._color_list[c]:
                self.color[source] = c
                break
        for target in self.graph.iteradjacent(source):
            if self.color[target] is not None:
                self._color_list[self.color[target]] = False
        return c
```

4.6. Zbiór niezależny dla grafu szeregowo-równoległego

Wyznaczanie największego zbioru niezależnego dla sp-grafu zadanego w postaci sp-drzewa. Łatwo zauważyć, że dla sp-grafu rozwiązanie optymalne może zawierać lub może nie zawierać każdy z dwóch wyróżnionych wierzchołków s, t. W naturalny sposób powstają cztery klasy rozwiązań, które należy monitorować przy łączeniu sp-grafów. Przyjmujemy następujące oznaczenia dla rozwiązania zawierającego największy zbiór niezależny z danej kategorii.

— Rozwiązanie $r00$ - zbiór niezależny **nie** zawiera s, t.
— Rozwiązanie $r01$ - zbiór niezależny **nie** zawiera s i zawiera t.
— Rozwiązanie $r10$ - zbiór niezależny zawiera s i **nie** zawiera t.
— Rozwiązanie $r11$ - zbiór niezależny zawiera s, t.

Należy przygotować trzy tabliczki składania rozwiązań, odpowiadające trzem rodzajom połączeń: równoległemu (tabela 4.1), szeregowemu (tabela 4.2), *jackknife* (tabela 4.3). Końcowe rozwiązanie można zapisać jako $\max\{r00, r01, r10, r11\}$.

Tabela 4.1. Tabliczka składania rozwiązań dla zbioru niezależnego (połączenie równoległe). Liczby x, y oznaczają liczność zbioru niezależnego.

równoległe	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$			
$r01, x$		$r01, x+y-1$		
$r10, x$			$r10, x+y-1$	
$r11, x$				$r11, x+y-2$

Tabela 4.2. Tabliczka składania rozwiązań dla zbioru niezależnego (połączenie szeregowe). Liczby x, y oznaczają liczność zbioru niezależnego.

szeregowe	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$	$r01, x+y$		
$r01, x$			$r00, x+y-1$	$r01, x+y-1$
$r10, x$	$r10, x+y$	$r11, x+y$		
$r11, x$			$r10, x+y-1$	$r11, x+y-1$

Tabela 4.3. Tabliczka składania rozwiązań dla zbioru niezależnego (połączenie *jackknife*). Liczby x, y oznaczają liczność zbioru niezależnego.

jackknife	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$	$r00, x+y$		
$r01, x$			$r01, x+y-1$	$r01, x+y-1$
$r10, x$	$r10, x+y$	$r10, x+y$		
$r11, x$			$r11, x+y-1$	$r11, x+y-1$

Listing 4.6. Wyznaczanie największego zbioru niezależnego.

```
#!/usr/bin/python

class SPIndependentSet:
    """Find a maximum independent set for sp-graphs."""

    def __init__(self, graph, root):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph          # series-parallel graph
        self.root = root            # binary sp-tree
        self.independent_set = set()
        self.cardinality = 0
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self):
        """Executable pseudocode."""
        arg2 = self._visit(self.root)
        self.independent_set.update(max(arg2, key=len))
        self.cardinality = len(self.independent_set)

    def _compose_series(self, arg1, arg2):
        """Compose results for series operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = max(r00a|r00b, r01a|r10b, key=len)
        r01 = max(r00a|r01b, r01a|r11b, key=len)
        r10 = max(r10a|r00b, r11a|r10b, key=len)
        r11 = max(r10a|r01b, r11a|r11b, key=len)
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
        return (r00, r01, r10, r11)

    def _compose_parallel(self, arg1, arg2):
        """Compose results for parallel operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = r00a|r00b
        r01 = r01a|r01b
        r10 = r10a|r10b
        r11 = r11a|r11b
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
        return (r00, r01, r10, r11)
```

```
    def _compose_jackknife(self, arg1, arg2):
        """Compose results for jackknife operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = max(r00a|r00b, r00a|r01b, key=len)
        r01 = max(r01a|r10b, r01a|r11b, key=len)
        r10 = max(r10a|r00b, r10a|r01b, key=len)
        r11 = max(r11a|r10b, r11a|r11b, key=len)
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
        return (r00, r01, r10, r11)

    def _visit(self, node):
        """Traversing postorder."""
        if node.type == "edge":
            # r11 faktycznie jest niepoprawne (to jest r00),
            # ale takie przyjecie upraszcza przetwarzanie..
            # Kolejnosc (r00, r01, r10, r11).
            return (set(),
                    set([node.target]),
                    set([node.source]),
                    set())
        arg1 = self._visit(node.left)
        arg2 = self._visit(node.right)
        if node.type == "series":
            return self._compose_series(arg1, arg2)
        elif node.type == "parallel":
            return self._compose_parallel(arg1, arg2)
        elif node.type == "jackknife":
            return self._compose_jackknife(arg1, arg2)
        else:
            raise ValueError("bad node type")
```

4.7. Zbiór dominujący dla grafu szeregowo-równoległego

Wyznaczanie najmniejszego zbioru dominującego dla sp-grafu zadanego w postaci sp-drzewa. Można zauważyć, że dla sp-grafu rozwiązanie optymalne może zawierać wierzchołki s, t, albo rozwiązanie może nie zawierać tych wierzchołków, ale będą one zdominowane przez pewnego sąsiada. Okazuje się, że przy rozwiązaniach podproblemów potrzebne są jeszcze formalnie nieprawidłowe rozwiązania, w których wierzchołki s, t nie są zdominowane i nie należą do zbioru dominującego. W ten sposób powstaje dziewięć klas rozwiązań, które należy monitorować przy łączeniu sp-grafów. Przyjmujemy następujące oznaczenia dla rozwiązania zawierającego najmniejszy zbiór dominujący z danej kategorii.

— Rozwiązanie $r00$ - zbiór dominujący **nie** zawiera s, t i nie są one zdominowane.
— Rozwiązanie $r01$ - zbiór dominujący **nie** zawiera s, t, s **nie** jest zdominowany, t jest zdominowany.
— Rozwiązanie $r02$ - zbiór dominujący zawiera t, s **nie** jest zdominowany.
— Rozwiązanie $r10$ - zbiór dominujący **nie** zawiera s, t, s jest zdominowany, t **nie** jest zdominowany.

— Rozwiązanie $r11$ - zbiór dominujący **nie** zawiera s, t, oba są zdominowane.
— Rozwiązanie $r12$ - zbiór dominujący zawiera t, s jest zdominowany.
— Rozwiązanie $r20$ - zbiór dominujący zawiera s, t **nie** jest zdominowany.
— Rozwiązanie $r21$ - zbiór dominujący zawiera s, t jest zdominowany.
— Rozwiązanie $r22$ - zbiór dominujący zawiera s, t.

Należy przygotować trzy tabliczki składania rozwiązań, odpowiadające trzem rodzajom połączeń: równoległemu (tabela 4.4), szeregowemu (tabela 4.5), *jackknife* (tabela 4.6). Końcowe rozwiązanie wybieramy jako najlepsze z poprawnych rozwiązań, więc można je zapisać jako $\min\{r11, r12, r21, r22\}$.

Tabela 4.4. Tabliczka składania rozwiązań dla zbioru dominującego (połączenie równoległe). Liczby x, y oznaczają liczność zbioru dominującego.

równo- ległe	$r00$ y	$r01$ y	$r02$ y	$r10$ y	$r11$ y	$r12$ y	$r20$ y	$r21$ y	$r22$ y
$r00$ x	$r00$ $x+y$	$r01$ $x+y$		$r10$ $x+y$	$r11$ $x+y$				
$r01$ x	$r01$ $x+y$	$r01$ $x+y$		$r11$ $x+y$	$r11$ $x+y$				
$r02$ x			$r02$ $x+y-1$			$r12$ $x+y-1$			
$r10$ x	$r10$ $x+y$	$r11$ $x+y$		$r10$ $x+y$	$r11$ $x+y$				
$r11$ x	$r11$ $x+y$	$r11$ $x+y$		$r11$ $x+y$	$r11$ $x+y$				
$r12$ x			$r12$ $x+y-1$			$r12$ $x+y-1$			
$r20$ x							$r20$ $x+y-1$	$r21$ $x+y-1$	
$r21$ x							$r21$ $x+y-1$	$r21$ $x+y-1$	
$r22$ x									$r22$ $x+y-2$

Tabela 4.5. Tabliczka składania rozwiązań dla zbioru dominującego (połączenie szeregowe). Liczby x, y oznaczają liczność zbioru dominującego. Wierzchołek na łączeniu musi być zdominowany lub w zbiorze dominującym.

szeregowo	$r00$ y	$r01$ y	$r02$ y	$r10$ y	$r11$ y	$r12$ y	$r20$ y	$r21$ y	$r22$ y
$r00$ x				$r00$ $x+y$	$r01$ $x+y$	$r02$ $x+y$			
$r01$ x	$r00$ $x+y$	$r01$ $x+y$	$r02$ $x+y$	$r00$ $x+y$	$r01$ $x+y$	$r02$ $x+y$			
$r02$ x							$r00$ $x+y-1$	$r01$ $x+y-1$	$r02$ $x+y-1$
$r10$ x				$r10$ $x+y$	$r11$ $x+y$	$r12$ $x+y$			
$r11$ x	$r10$ $x+y$	$r11$ $x+y$	$r12$ $x+y$	$r10$ $x+y$	$r11$ $x+y$	$r12$ $x+y$			
$r12$ x							$r10$ $x+y-1$	$r11$ $x+y-1$	$r12$ $x+y-1$
$r20$ x				$r20$ $x+y$	$r21$ $x+y$	$r22$ $x+y$			
$r21$ x	$r20$ $x+y$	$r21$ $x+y$	$r22$ $x+y$	$r20$ $x+y$	$r21$ $x+y$	$r22$ $x+y$			
$r22$ x							$r20$ $x+y-1$	$r21$ $x+y-1$	$r22$ $x+y-1$

Tabela 4.6. Tabliczka składania rozwiązań dla zbioru dominującego (połączenie *jackknife*). Liczby x, y oznaczają liczność zbioru dominującego.

jackknife	$r00$ y	$r01$ y	$r02$ y	$r10$ y	$r11$ y	$r12$ y	$r20$ y	$r21$ y	$r22$ y
$r00$ x		$r00$ $x+y$	$r00$ $x+y$		$r01$ $x+y$	$r01$ $x+y$			
$r01$ x		$r01$ $x+y$	$r01$ $x+y$		$r01$ $x+y$	$r01$ $x+y$			
$r02$ x								$r02$ $x+y-1$	$r02$ $x+y-1$
$r10$ x		$r10$ $x+y$	$r10$ $x+y$		$r11$ $x+y$	$r11$ $x+y$			
$r11$ x		$r11$ $x+y$	$r11$ $x+y$		$r11$ $x+y$	$r11$ $x+y$			
$r12$ x								$r12$ $x+y-1$	$r12$ $x+y-1$
$r20$ x		$r20$ $x+y$	$r20$ $x+y$		$r21$ $x+y$	$r21$ $x+y$			
$r21$ x		$r21$ $x+y$	$r21$ $x+y$		$r21$ $x+y$	$r21$ $x+y$			
$r22$ x								$r22$ $x+y-1$	$r22$ $x+y-1$

Listing 4.7. Wyznaczanie najmniejszego zbioru dominującego.

```
#!/usr/bin/python

class SPDominatingSet:
    """Find a minimum dominating set for sp-graphs."""

    def __init__(self, graph, root):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph          # series-parallel graph
        self.root = root            # binary sp-tree
        self.dominating_set = set()
        self.cardinality = 0
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self):
        """Executable pseudocode."""
        arg2 = self._visit(self.root)
        r00, r01, r02, r10, r11, r12, r20, r21, r22 = arg2
        self.dominating_set.update(min(r11, r12, r21, r22, key=len))
        self.cardinality = len(self.dominating_set)

    def _compose_series(self, arg1, arg2):
        """Compose results for series operation."""
        r00a, r01a, r02a, r10a, r11a, r12a, r20a, r21a, r22a = arg1
        r00b, r01b, r02b, r10b, r11b, r12b, r20b, r21b, r22b = arg2
        r00 = min(r00a|r10b, r01a|r00b, r01a|r10b, r02a|r20b, key=len)
        r01 = min(r00a|r11b, r01a|r01b, r01a|r11b, r02a|r21b, key=len)
        r02 = min(r00a|r12b, r01a|r02b, r01a|r12b, r02a|r22b, key=len)
        r10 = min(r10a|r10b, r11a|r00b, r11a|r10b, r12a|r20b, key=len)
        r11 = min(r10a|r11b, r11a|r01b, r11a|r11b, r12a|r21b, key=len)
        r12 = min(r10a|r12b, r11a|r02b, r11a|r12b, r12a|r22b, key=len)
        r20 = min(r20a|r10b, r21a|r00b, r21a|r10b, r22a|r20b, key=len)
        r21 = min(r20a|r11b, r21a|r01b, r21a|r11b, r22a|r21b, key=len)
        r22 = min(r20a|r12b, r21a|r02b, r21a|r12b, r22a|r22b, key=len)
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
        return (r00, r01, r02, r10, r11, r12, r20, r21, r22)

    def _compose_parallel(self, arg1, arg2):
        """Compose results for parallel operation."""
        r00a, r01a, r02a, r10a, r11a, r12a, r20a, r21a, r22a = arg1
        r00b, r01b, r02b, r10b, r11b, r12b, r20b, r21b, r22b = arg2
        r00 = r00a|r00b
        r01 = min(r01a|r01b, r00a|r01b, r01a|r00b, key=len)
        r02 = r02a|r02b
        r10 = min(r10a|r10b, r00a|r10b, r10a|r00b, key=len)
        r11 = min(r11a|r11b, r11a|r10b, r11a|r01b,
                  r01a|r11b, r10a|r11b, r11a|r00b,
                  r01a|r10b, r10a|r01b, r00a|r11b, key=len)
        r12 = min(r12a|r12b, r12a|r02b, r02a|r12b, key=len)
        r20 = r20a|r20b
        r21 = min(r21a|r21b, r21a|r20b, r20a|r21b, key=len)
        r22 = r22a|r22b
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
```

```
        return (r00, r01, r02, r10, r11, r12, r20, r21, r22)

    def _compose_jackknife(self, arg1, arg2):
        """Compose results for jackknife operation."""
        r00a, r01a, r02a, r10a, r11a, r12a, r20a, r21a, r22a = arg1
        r00b, r01b, r02b, r10b, r11b, r12b, r20b, r21b, r22b = arg2
        r00 = min(r00a|r01b, r00a|r02b, key=len)
        r01 = min(r00a|r11b, r00a|r12b,
                  r01a|r01b, r01a|r02b,
                  r01a|r11b, r01a|r12b, key=len)
        r02 = min(r02a|r21b, r02a|r22b, key=len)
        r10 = min(r10a|r01b, r10a|r02b, key=len)
        r11 = min(r10a|r11b, r10a|r12b,
                  r11a|r11b, r11a|r12b,
                  r11a|r01b, r11a|r02b, key=len)
        r12 = min(r12a|r21b, r12a|r22b, key=len)
        r20 = min(r20a|r01b, r20a|r02b, key=len)
        r21 = min(r20a|r11b, r20a|r12b,
                  r21a|r11b, r21a|r12b,
                  r21a|r01b, r21a|r02b, key=len)
        r22 = min(r22a|r21b, r22a|r22b, key=len)
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
        return (r00, r01, r02, r10, r11, r12, r20, r21, r22)

    def _visit(self, node):
        """Traversing postorder."""
        if node.type == "edge":
            # r01 faktycznie jest niepoprawne (to jest r22),
            # r02 faktycznie jest niepoprawne (to jest r22),
            # r10 faktycznie jest niepoprawne (to jest r22),
            # r11 faktycznie jest niepoprawne (to jest r22),
            # r20 faktycznie jest niepoprawne (to jest r22),
            # ale takie przyjecie upraszcza przetwarzanie..
            # Kolejnosc (r00, r01, r02, r10, r11, r12, r20, r21, r22).
            return (set(),
                    set([node.source, node.target]),
                    set([node.source, node.target]),
                    set([node.source, node.target]),
                    set([node.source, node.target]),
                    set([node.target]),
                    set([node.source, node.target]),
                    set([node.source]),
                    set([node.source, node.target]))
        arg1 = self._visit(node.left)
        arg2 = self._visit(node.right)
        if node.type == "series":
            return self._compose_series(arg1, arg2)
        elif node.type == "parallel":
            return self._compose_parallel(arg1, arg2)
        elif node.type == "jackknife":
            return self._compose_jackknife(arg1, arg2)
        else:
            raise ValueError("bad node type")
```

4.8. Pokrycie wierzchołkowe dla grafu szeregowo-równoległego

Wyznaczanie najmniejszego pokrycia wierzchołkowego dla sp-grafu zadanego w postaci sp-drzewa. W tym przypadku podczas budowania optymalnych rozwiązań będziemy rozważać dwie sytuacje. Wierzchołki s, t będą należeć do szukanego pokrycia, albo nie będą - pod warunkiem, że wszystkie ich sąsiednie wierzchołki się w nim znajdą. Podczas składania rozwiązań nie pojawią się rozwiązania lokalnie błędne jak w innych przypadkach. Rozważać będziemy zatem 4 kategorie optymalnych rozwiązań.

Przyjmujemy następujące oznaczenia dla rozwiązania zawierającego najmniejsze pokrycie wierzchołkowe z danej kategorii.

— Rozwiązanie $r00$ - pokrycie wierzchołkowe **nie** zawiera s, t, ale zawiera ich sąsiadów.
— Rozwiązanie $r01$ - pokrycie wierzchołkowe zawiera t, ale **nie** zawiera s, tylko jego sąsiadów.
— Rozwiązanie $r10$ - pokrycie wierzchołkowe zawiera s i **nie** zawiera t, tylko jego sąsiadów.
— Rozwiązanie $r11$ - pokrycie wierzchołkowe zawiera s, t.

Należy przygotować trzy tabliczki składania rozwiązań, odpowiadające trzem rodzajom połączeń: równoległemu (tabela 4.7), szeregowemu (tabela 4.8), *jackknife* (tabela 4.9). Końcowe rozwiązanie można zapisać jako $\min\{r00, r01, r10, r11\}$.

Tabela 4.7. Tabliczka składania rozwiązań dla pokrycia wierzchołkowego (połączenie równoległe). Liczby x, y oznaczają liczność zbioru będącego szukanym pokryciem.

równoległe	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$			
$r01, x$		$r01, x+y-1$		
$r10, x$			$r10, x+y-1$	
$r11, x$				$r11, x+y-2$

Tabela 4.8. Tabliczka składania rozwiązań dla pokrycia wierzchołkowego (połączenie szeregowe). Liczby x, y oznaczają liczność zbioru będącego szukanym pokryciem.

szeregowe	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$	$r01, x+y$		
$r01, x$			$r00, x+y-1$	$r01, x+y-1$
$r10, x$	$r10, x+y$	$r11, x+y$		
$r11, x$			$r10, x+y-1$	$r11, x+y-1$

Tabela 4.9. Tabliczka składania rozwiązań dla pokrycia wierzchołkowego (połączenie *jackknife*). Liczby x, y oznaczają liczność zbioru będącego szukanym pokryciem.

jackknife	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$	$r00, x+y$		
$r01, x$			$r01, x+y-1$	$r01, x+y-1$
$r10, x$	$r10, x+y$	$r10, x+y$		
$r11, x$			$r11, x+y-1$	$r11, x+y-1$

Listing 4.8. Wyznaczanie najmniejszego pokrycia wierzchołkowego.

```
#!/usr/bin/python

class SPNodeCover:
    """Find a minimum cardinality node cover for sp-graphs."""

    def __init__(self, graph, root):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph          # series-parallel graph
        self.root = root            # binary sp-tree
        self.node_cover = set()
        self.cardinality = 0
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self):
        """Executable pseudocode."""
        arg2 = self._visit(self.root)
        self.node_cover.update(min(arg2, key=len))
        self.cardinality = len(self.node_cover)

    def _compose_series(self, arg1, arg2):
        """Compose results for series operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = min(r00a|r00b, r01a|r10b, key=len)
        r01 = min(r00a|r01b, r01a|r11b, key=len)
        r10 = min(r10a|r00b, r11a|r10b, key=len)
        r11 = min(r10a|r01b, r11a|r11b, key=len)
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
        return (r00, r01, r10, r11)

    def _compose_parallel(self, arg1, arg2):
        """Compose results for parallel operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = r00a|r00b
        r01 = r01a|r01b
        r10 = r10a|r10b
        r11 = r11a|r11b
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
        return (r00, r01, r10, r11)
```

```
    def _compose_jackknife(self, arg1, arg2):
        """Compose results for jackknife operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = min(r00a|r00b, r00a|r01b, key=len)
        r01 = min(r01a|r10b, r01a|r11b, key=len)
        r10 = min(r10a|r00b, r10a|r01b, key=len)
        r11 = min(r11a|r10b, r11a|r11b, key=len)
        # Powtarzajace sie wierzcholki same wypadna w zbiorach.
        return (r00, r01, r10, r11)

    def _visit(self, node):
        """Traversing postorder."""
        if node.type == "edge":
            # r00 faktycznie jest niepoprawne (to jest r11),
            # ale takie przyjecie upraszcza przetwarzanie..
            # Kolejnosc (r00, r01, r10, r11).
            return (set([node.source, node.target]),
                    set([node.target]),
                    set([node.source]),
                    set([node.source, node.target]))
        arg1 = self._visit(node.left)
        arg2 = self._visit(node.right)
        if node.type == "series":
            return self._compose_series(arg1, arg2)
        elif node.type == "parallel":
            return self._compose_parallel(arg1, arg2)
        elif node.type == "jackknife":
            return self._compose_jackknife(arg1, arg2)
        else:
            raise ValueError("bad node type")
```

4.9. Największe skojarzenie dla grafu szeregowo-równoległego

Wyznaczanie największego skojarzenia dla sp-grafu. Podobnie jak poprzednio, przy łączeniu rozwiązań częściowych będziemy rozważać dwa przypadki. Każdy z wierzchołków s, t będzie skojarzony, albo nie będzie skojarzony. W drugim przypadku dany nieskojarzony wierzchołek będzie mógł jeszcze znaleźć się w skojarzeniu w wyniku dalszego składania rozwiązań. Również tym razem nie pojawią się częściowe rozwiązania, które byłyby formalnie niepoprawne.

Przyjmujemy następujące oznaczenia dla rozwiązania będącego maksymalnym skojarzeniem.

— Rozwiązanie $r00$ - wierzchołki s, t **nie** są skojarzone.
— Rozwiązanie $r01$ - wierzchołek s **nie** jest skojarzony, wierzchołek t jest skojarzony.
— Rozwiązanie $r10$ - wierzchołek s jest skojarzony, wierzchołek t **nie** jest skojarzony.
— Rozwiązanie $r11$ - oba wierzchołki s, t są skojarzone.

Należy przygotować trzy tabliczki składania rozwiązań, odpowiadające trzem rodzajom połączeń: równoległemu (tabela 4.10), szeregowemu (tabela 4.11), *jackknife* (tabela 4.12). Końcowe rozwiązanie można zapisać jako $\max\{r00, r01, r10, r11\}$.

Tabela 4.10. Tabliczka składania rozwiązań dla maksymalnego skojarzenia (połączenie równoległe). Liczby x, y oznaczają liczność skojarzenia.

równoległe	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$	$r01, x+y$	$r10, x+y$	$r11, x+y$
$r01, x$	$r01, x+y$		$r11, x+y$	
$r10, x$	$r10, x+y$	$r11, x+y$		
$r11, x$	$r11, x+y$			

Tabela 4.11. Tabliczka składania rozwiązań dla maksymalnego skojarzenia (połączenie szeregowe). Liczby x, y oznaczają liczność skojarzenia.

szeregowe	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$	$r01, x+y$	$r00, x+y$	$r01, x+y$
$r01, x$	$r00, x+y$	$r01, x+y$		
$r10, x$	$r10, x+y$	$r11, x+y$	$r10, x+y$	$r11, x+y$
$r11, x$	$r10, x+y$	$r11, x+y$		

Tabela 4.12. Tabliczka składania rozwiązań dla maksymalnego skojarzenia (połączenie *jackknife*). Liczby x, y oznaczają liczność skojarzenia.

jackknife	$r00, y$	$r01, y$	$r10, y$	$r11, y$
$r00, x$	$r00, x+y$	$r00, x+y$	$r01, x+y$	$r01, x+y$
$r01, x$	$r01, x+y$	$r01, x+y$		
$r10, x$	$r10, x+y$	$r10, x+y$	$r11, x+y$	$r11, x+y$
$r11, x$	$r11, x+y$	$r11, x+y$		

Listing 4.9. Wyznaczanie największego skojarzenia.

```
#!/usr/bin/python

from edges import Edge

class SPMatching:
    """Find a maximum matching for sp-graphs."""

    def __init__(self, graph, root):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph          # series-parallel graph
        self.root = root            # binary sp-tree
        self.mate_set = set()       # set with mate edges
        self.mate = dict((node, None) for node in self.graph.iternodes())
        self.cardinality = 0        # size of mate set
        import sys
        recursionlimit = sys.getrecursionlimit()
```

```
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self):
        """Executable pseudocode."""
        arg2 = self._visit(self.root)
        self.mate_set.update(max(arg2, key=len))
        for edge in self.mate_set:
            self.mate[edge.source] = edge.target
            self.mate[edge.target] = edge.source
        self.cardinality = len(self.mate_set)

    def _compose_series(self, arg1, arg2):
        """Compose results for series operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = max(r00a|r00b, r01a|r00b, r00a|r10b, key=len)
        r01 = max(r00a|r01b, r01a|r01b, r00a|r11b, key=len)
        r10 = max(r10a|r00b, r11a|r00b, r10a|r10b, key=len)
        r11 = max(r10a|r01b, r10a|r11b, r11a|r01b, key=len)
        return (r00, r01, r10, r11)

    def _compose_parallel(self, arg1, arg2):
        """Compose results for parallel operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = r00a|r00b
        r01 = max(r00a|r01b, r01a|r00b, key=len)
        r10 = max(r10a|r00b, r00a|r10b, key=len)
        r11 = max(r00a|r11b, r01a|r10b, r10a|r01b, r11a|r00b, key=len)
        return (r00, r01, r10, r11)

    def _compose_jackknife(self, arg1, arg2):
        """Compose results for jackknife operation."""
        r00a, r01a, r10a, r11a = arg1
        r00b, r01b, r10b, r11b = arg2
        r00 = max(r00a|r00b, r00a|r01b, key=len)
        r01 = max(r00a|r10b, r00a|r11b, r01a|r00b, r01a|r01b, key=len)
        r10 = max(r10a|r00b, r10a|r01b, key=len)
        r11 = max(r11a|r00b, r11a|r01b, r10a|r10b, r10a|r11b, key=len)
        return (r00, r01, r10, r11)

    def _visit(self, node):
        """Traversing postorder."""
        if node.type == "edge":
            # r01, r10 faktycznie sa niepoprawne (to jest r00),
            # ale takie przyjecie upraszcza przetwarzanie..
            # Kolejnosc (r00, r01, r10, r11).
            if node.source <= node.target:
                edge = Edge(node.source, node.target)
            else:
                edge = Edge(node.target, node.source)
            return (set(), set(), set(), set([edge]))
        arg1 = self._visit(node.left)
        arg2 = self._visit(node.right)
        if node.type == "series":
            return self._compose_series(arg1, arg2)
        elif node.type == "parallel":
```

```
            return self._compose_parallel(arg1, arg2)
        elif node.type == "jackknife":
            return self._compose_jackknife(arg1, arg2)
        else:
            raise ValueError("bad node type")
```

4.10. Algorytmy równoległe dla grafów szeregowo-równoległych

W niniejszej pracy rozważamy różne algorytmy sekwencyjne. Warto zauważyć, że w literaturze badane są również algorytmy równoległe, przeznaczone dla komputerów z wieloma jednostkami obliczeniowymi. Wykorzystywane są różne modele pamięci.

W roku 1996 Bodlaender i de Fluiter [29] przedstawili algorytmy równoległe do tworzenia drzewa dekompozycji dla grafu szeregowo-równoległego.

4.11. Znajdowanie PEO dla grafu Halina

Graf Halina jest częściowym 3-drzewem i czasem potrzebne jest wyznaczenie pełnego 3-drzewa (uzupełnienia cięciwowego) zawierającego dany graf. Przedstawimy algorytm [funkcja find_peo_halin()] wyznaczający PEO tego pełnego 3-drzewa. Warto zwrócić uwagę, że problem jest trudniejszy niż dla częściowych 2-drzew, ponieważ nie można usuwać z grafu dowolnych 3-liści. Dozwolone jest usuwanie dwóch rodzajów 3-liści, tak aby zmniejszony graf dalej był grafem Halina.

Kolejno usuwane 3-liście są dodawane do listy wierzchołków tworzącej PEO. Jeżeli trzej sąsiedzi usuwanego wierzchołka nie tworzą kliki (trójkąta), to uzupełniamy odpowiednie krawędzie. Usuwanie 3-liści kończymy na etapie grafu K_4, którego cztery wierzchołki będą na końcu PEO w dowolnej kolejności.

Pierwszy dozwolony rodzaj 3-liści do usunięcia to te leżące na brzegu grafu koła lub wewnątrz brzegu wachlarza. Te 3-liście należą do cyklu zewnętrznego grafu Halina. Niech v_1 będzie takim 3-liściem, jego sąsiedzi to v_2, v_3, u, przy czym u jest centrum wachlarza, a v_2, v_3 leżą na cyklu zewnętrznym. Dodajemy krawędź v_2v_3, oraz usuwamy krawędzie v_1v_2, v_1v_3, v_1u [potrzebny rysunek].

Drugi dozwolony rodzaj 3-liści do usunięcia to centra małych wachlarzy, które mają tylko dwa wierzchołki na cyklu zewnętrznym. Niech u_1 będzie centrum małego wachlarza, jego sąsiedzi to v_1, v_2, u_2, przy czym u_2 nie leży na cyklu zewnętrznym. Dodajemy krawędzie v_1u_2, v_2u_2, oraz usuwamy krawędzie u_1v_1, u_1v_2, u_1u_2 [potrzebny rysunek].

Listing 4.10. Wyznaczanie PEO dla grafów Halina.

```
def find_peo_halin(graph):
    """Find PEO for a supergraph (3-tree) of a Halin graph."""
    pass
```

5. Podsumowanie

Grafy szeregowo-równoległe są ciekawą klasą grafów planarnych o szerokości drzewowej co najwyżej 2, czyli są o jeden stopień "grubsze" od drzew. W pracy zebrano podstawowe informacje o tych grafach i przygotowano narzędzia informatyczne do pracy z nimi.

W ramach pracy przygotowano generator k-drzew przypadkowych, oraz generator sp-grafów nieskierowanych (w dwóch wersjach). Generator sp-grafów w pierwszej wersji zwraca instancję klasy Graph, a w drugiej wersji zwraca sp-drzewo.

Zaimplementowano algorytm rozpoznawania sp-grafów nieskierowanych i budowania sp-drzewa binarnego, algorytm wyznaczania PEO dla 2-drzewa (grafu cięciwowego) zawierającego dany sp-graf.

Zaimplementowano wydajne algorytmy rozwiązujące problemy z teorii grafów, które są trudne dla ogólnych grafów. W szczególności rozważono problem największego zbioru niezależnego, problem najmniejszego zbioru dominującego, problem najmniejszego pokrycia wierzchołkowego, problem największego skojarzenia, problem kolorowania wierzchołków. Struktura sp-grafów pozwoliła w wielu przypadkach na zastosowanie techniki programowania dynamicznego. W przypadku kolorowania wierzchołków wykorzystano fakt, że sp-grafy są 2-drzewami i można było zastosować kolorowanie zachłanne bazujące na wyznaczonym PEO nadgrafu cięciwowego. Dostajemy w ten sposób 3-kolorowanie wierzchołkowe dla sp-grafów, co do których wiemy, że nie są dwudzielne.

Grafy Halina należą do grafów planarnych o szerokości drzewowej równej 3, przez co są "grubsze" od sp-grafów. W dalszym ciągu jednak pozwalają na szybkie znajdowanie optymalnych rozwiązań dla wielu problemów trudnych w ogólnym przypadku. Dla grafów Halina podano algorytm wyznaczania PEO dla 3-drzewa (grafu cięciwowego) zawierającego dany graf. Implementacje innych algorytmów można znaleźć w osobnej pracy [25]. Ciekawe jest, że grafy Halina są 3-kolorowalne wierzchołkowo, z wyjątkiem grafów kołowych o parzystej liczbie wierzchołków. Stąd do kolorowania grafów Halina nie wykorzystujemy PEO (bo daje 4 kolory), tylko specjalizowany algorytm.

Zaimplementowanym algorytmom towarzyszą testy jednostkowe wykonane przy pomocy modułu unittest. Rzeczywistą wydajność algorytmów sprawdzono z użyciem modułu timeit.

A. Testy algorytmów

Pełne sprawdzenie implementacji każdego algorytmu powinno zawierać testy poprawności kodu (testy jednostkowe), a także testy wydajnościowe, które w głównej mierze sprawdzają zależność rzeczywistego czasu pracy algorytmu od wielkości danych wejściowych. W naszych testach nie pominęlismy również innego istotnego aspektu, jakim jest złożoność pamięciowa.

Ten dodatek mimo wszystko najwięcej uwagi poświęca złożoności czasowej, ponieważ ze względu na specyfikę problemów grafowych jest ona najbardziej interesująca.

A.1. Struktura testów jednostkowych

W tym przypadku skupiliśmy się na weryfikacji rezultatów wykonania poszczególnych algorytmów dla standardowych, jak również bardzo skrajnych danych wejściowych. Dla jasności - pod tym terminem rozumiemy grafy, w których kluczowe parametry (określające liczbę krawędzi, wierzchołków, połączeń określonego typu) osiągają swoje maksymalne lub minimalne wartości. Przykładem może być graf szeregowo-równoległy składający się tylko ze źródła i ujścia lub graf zbudowany tylko z połączeń szeregowych.

Każdy z testów posiada podobną strukturę. Składa się z kilku (lub więcej) tzw. przypadków testowych przygotowujących grafy o ściśle zadanej budowie. Później uruchomiany jest aktualnie sprawdzany algorytm, po czym za pomocą asercji kontrolowany jest jego wynik.

Przykładowy test jednostkowy pokazany jest na listingu A.1.

Listing A.1. Przykładowy test jednostkowy.

```
#!/usr/bin/python

import sys
sys.path.append('../main')
import unittest
from spdominatingset import SPDominatingSet
from test_spgraph_factory import *

class TestDominatingSet(unittest.TestCase):

    def initFromGraphInfo(self, graphInfo):
        self.G = graphInfo['graph']
        self.N = graphInfo['size']
        self.root = graphInfo['sptree']
        self.nodes = graphInfo['nodes']
        self.edges = graphInfo['edges']

    def test_sp_dominating_set_small(self):
```

```
        self.initFromGraphInfo(prepareSmallGraph())
        algorithm = SPDominatingSet(self.G, self.root)
        algorithm.run()
        expected1 = set([1])
        self.assertEqual(algorithm.cardinality, len(expected1))
        self.assertTrue(algorithm.dominating_set == expected1)

    def test_sp_dominating_set_small3(self):
        self.initFromGraphInfo(prepareSmallGraph3())
        algorithm = SPDominatingSet(self.G, self.root)
        algorithm.run()
        expected1 = set([1, 2])
        expected2 = set([1, 3])
        expected3 = set([1, 4])
        self.assertEqual(algorithm.cardinality, len(expected1))
        self.assertTrue(algorithm.dominating_set == expected1
                        or algorithm.dominating_set == expected2
                        or algorithm.dominating_set == expected3)

    def test_sp_dominating_set_medium(self):
        self.initFromGraphInfo(prepareMediumGraph())
        algorithm = SPDominatingSet(self.G, self.root)
        algorithm.run()
        expected = set([8, 5, 6])
        self.assertEqual(algorithm.cardinality, len(expected))
        self.assertEqual(algorithm.dominating_set, expected)

    def test_sp_dominating_set_medium2(self):
        self.initFromGraphInfo(prepareMediumGraph2())
        algorithm = SPDominatingSet(self.G, self.root)
        algorithm.run()
        expected = set([2, 6])
        self.assertEqual(algorithm.cardinality, len(expected))
        self.assertEqual(algorithm.dominating_set, expected)

if __name__ == "__main__":
    unittest.main()
```

Fragment kodu generującego grafy wejściowe dla powyższego testu przedstawia listing A.2.

Listing A.2. Generator grafów dla testu jednostkowego.

```
#!/usr/bin/python
import sys
sys.path.append('../main')
from edges import Edge
from graphs import Graph
from spnodes import Node

...

def prepareSmallGraph2():
    #       0         s=0, t=1
    #       |
    #   3--1--2
    #       |
    #       4
```

```
    N = 5
    G = Graph(N)
    nodes = range(N)
    edges = [Edge(0, 1), Edge(1, 2), Edge(1, 3), Edge(1, 4)]
    for node in nodes:
        G.add_node(node)
    for edge in edges:
        G.add_edge(edge)
    node1 = Node(0, 1, "edge")
    node2 = Node(1, 2, "edge")
    node3 = Node(1, 3, "edge")
    node4 = Node(1, 4, "edge")
    node5 = Node(0, 1, "jackknife", node1, node2)
    node6 = Node(0, 1, "jackknife", node5, node3)
    node7 = Node(0, 1, "jackknife", node6, node4)
    root = node7

    return {
        'size': N,
        'graph': G,
        'sptree': root,
        'nodes': nodes,
        'edges': edges
    }


def prepareSmallGraph3():
    #  0        s=0, t=4
    #  |
    #  1
    #  | \
    #  2  3
    #  | /
    #  4
    N = 5
    G = Graph(N)
    nodes = range(N)
    edges = [Edge(0, 1), Edge(1, 2), Edge(1, 3), Edge(2, 4), Edge(3, 4)]
    for node in nodes:
        G.add_node(node)
    for edge in edges:
        G.add_edge(edge)
    node1 = Node(0, 1, "edge")
    node2 = Node(1, 2, "edge")
    node3 = Node(1, 3, "edge")
    node4 = Node(2, 4, "edge")
    node5 = Node(3, 4, "edge")
    node6 = Node(1, 4, "series", node2, node4)
    node7 = Node(1, 4, "series", node3, node5)
    node8 = Node(1, 4, "parallel", node6, node7)
    node9 = Node(0, 4, "series", node1, node8)

    root = node9

    return {
        'size': N,
        'graph': G,
```

```
        'sptree': root,
        'nodes': nodes,
        'edges': edges
    }

    ...
```

A.2. Struktura testów złożoności pamięciowej

Z uwagi na specyfikę języka Python w zakresie zarządzania pamięcią zbadanie dokładnego je wykorzystania nie jest zadaniem trywialnym. Wyniki, które generuje test, prezentują jedynie szacunkowy obraz rzeczywistego wykorzystania zasobów podczas działania algorytmu.

Dodatkowo jeśli chodzi o algorytmy uruchamiane na zbudowanych sp-drzewach (takie jak np. SPNodeCover), to zużywają one minimalną i tym samym trudno mierzalną ilość pamięci. Wynika to z faktu, że sp-drzewa budowane są w sposób zrównoważony, a zatem rekurencyjny algorytm odkłada na stosie zbliżoną do logarytmicznej względem wierzchołków liczbę wywołań. Wyniki wykorzystania pamięci części algorytmów zostały dołączone do konkretnych sekcji w dalszej części tego dodatku.

Testy składają się z klasy bazowej oraz poszczególnych konkretnych klas, które implementują metodę przygotowującą testowany algorytm. Schemat wykonania pełnego testu zakładał wielokrotne uruchamianie kodu i stopniowe zwiększanie liczby wierzchołków. Użyta została biblioteka do mierzenia wydajności kodu memory_profiler [30].

Listing A.3. Klasa bazowa testów złożoności pamięciowej.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
sys.path.append('../main')
import abc
import gc
from memory_profiler import profile

fp = open('memory_profiler.log','a')

class MemoryComplexityTest:

    def __init__(self, n):
        self.n = self.arg_or_default(1, n)

    @abc.abstractmethod
    def get_algorithm(self, nth):
        return

    def run(self):
        gc.disable()
        self.run_test()

    @profile(stream=fp)
    def run_test(self):
```

```
        algorithm = self.get_algorithm(self.n)
        algorithm()

    def arg_or_default(arg_num, default_value):
        if (len(sys.argv) > arg_num):
            return int(sys.argv[arg_num])
        else:
            return default_value
```

Listing A.4. Przykładowy test wykorzystania pamięci.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
sys.path.append('../main')
from memory_complexity_test import MemoryComplexityTest
from spgen2 import make_random_ktree

class KtreeStaticKMemoryTest(MemoryComplexityTest):

    def __init__(self):
        MemoryComplexityTest.__init__(self, 100)
        self.title = u"Znajdowanie k-drzew dla k=5"
        self.k = 5

    def get_algorithm(self, nth):
        if (nth <= self.k):
            return lambda: None
        else:
            return lambda: make_random_ktree(nth, self.k)

test = KtreeStaticKMemoryTest()
test.run()
```

A.3. Struktura testów złożoności czasowej

Do pełnego potwierdzenia i zaprezentowania zakładanej złożoności czasowej zaimplementowanych algorytmów analiza statyczna jest niewystarczająca. W związku z tym zostały stworzone testy obliczające faktyczny czas wykonania kodu dla dużych zadań.

Test uruchamia dany algorytm wielokrotnie dla rosnących danych wejściowych i zapisuje wyniki tworząc funkcje wartości czasu wykonania algorytmu od liczby wierzchołków. Tym samym wyniki dają się naturalnie przedstawić na wykresach. W celu zminimalizowania błędów pomiaru test algorytmu został powtórzony odpowiednią liczbę razy dla tych samych danych, a czasy wykonania uśrednione.

Listing A.5. Klasy pomocnicze do konwertowania argumentów i wartości wyników.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import abc
```

```
import math

class CalculationMethod:

    @abc.abstractmethod
    def convert_arg(self, nth):
        return

    @abc.abstractmethod
    def convert_value(self, value):
        return

class LinearMethod(CalculationMethod):

    def __init__(self, jump=1):
        self.jump = jump

    def calculate_xt(self, nth):
        return nth * self.jump

    def result_from_calculation(self, value):
        return value

class ExponentialMethod(CalculationMethod):

    def convert_arg(self, nth):
        return int(10**nth)

    def convert_value(self, value):
        return math.log10(value)
```

Test umożliwia użycie dwóch metod wyliczania liczby wierzchołków (listing A.5): w sposób liniowy lub wykładniczy. W szczególności druga metoda pozwala dobrze zaprezentować zachowanie algorytmu dla coraz większych danych. Odpowiednie użycie parametrów klasy pozwala uzyskać oczekiwaną dokładność testu.

Listing A.6. Klasa bazowa testu badania czasu wykonania.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
sys.path.append('../main')
import abc
import timeit
import PyGnuplot as gp
import numpy as np
import sys

class TimeComplexityTest:

    def __init__(self, method, start_n, stop_n, avg_runs=10, steps=10):
        self.calculation_method = method
        self.start_n = self.arg_or_default(1, start_n)
        self.stop_n = self.arg_or_default(2, stop_n)
        self.runs_for_avg = avg_runs
        self.steps = steps
```

```
    @abc.abstractmethod
    def get_algorithm(self, nth):
        return

    def run(self):
        results_x, results_y = self.calculate_results()
        gp.s([np.array(results_x), np.array(results_y)])

    def calculate_results(self):
        results_x = []
        results_y = []
        for i in range(self.start_n*self.steps, self.stop_n*self.steps):
            x_t = self.calculation_method.convert_arg(float(i)/self.steps)
            algorithm = self.get_algorithm(x_t)
            times = timeit.Timer(algorithm).repeat(self.runs_for_avg, 1)
            y_tavg = sum(times)/self.runs_for_avg
            results_x.append(self.calculation_method.convert_value(x_t))
            results_y.append(self.calculation_method.convert_value(y_tavg))
        return results_x, results_y

    def arg_or_default(self, arg_num, default_value):
        if (len(sys.argv) > arg_num):
            return int(sys.argv[arg_num])
        else:
            return default_value
```

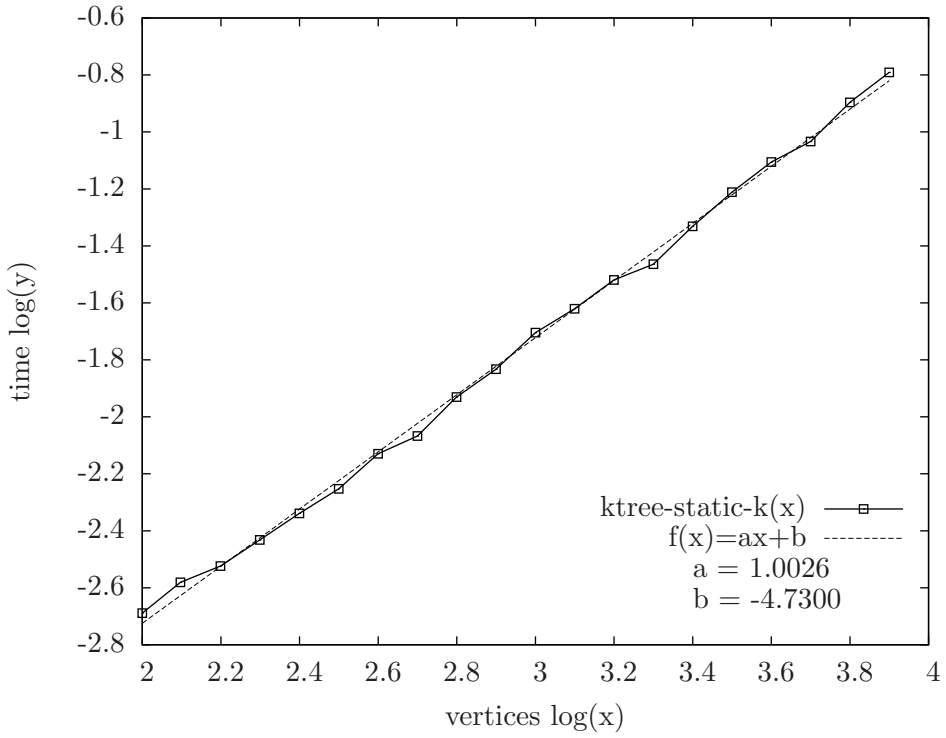
Testy rozszerzają klasę bazową TimeComplexityTest (listing A.6) i implementują metodę odpowiedzialną za przygotowanie danych wejściowych i danego algorytmu do testu. Jako przykład zamieściliśmy klasę NodeCoverTimeTest (listing A.7), która dotyczy algorytmu do wyznaczania pokrycia wierzchołkowego z klasy SPNodeCover.

Listing A.7. Klasa testowa przygotowująca do testu konkretny algorytm.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
sys.path.append('../main')
from time_complexity_test import TimeComplexityTest
from calculation_methods import ExponentialMethod
from sptree2 import find_sptree
from spgen2 import make_random_spgraph
from spcover import SPNodeCover

class NodeCoverTimeTest(TimeComplexityTest):

    def __init__(self):
        TimeComplexityTest.__init__(self, ExponentialMethod(), 2, 3)

    def get_algorithm(self, nth):
        graph = make_random_spgraph(nth)
        sptree = find_sptree(graph)
        algorithm = SPNodeCover(graph, sptree)
        return lambda: algorithm.run()

test = NodeCoverTimeTest()
```

A.4. Testy generatora k-drzew

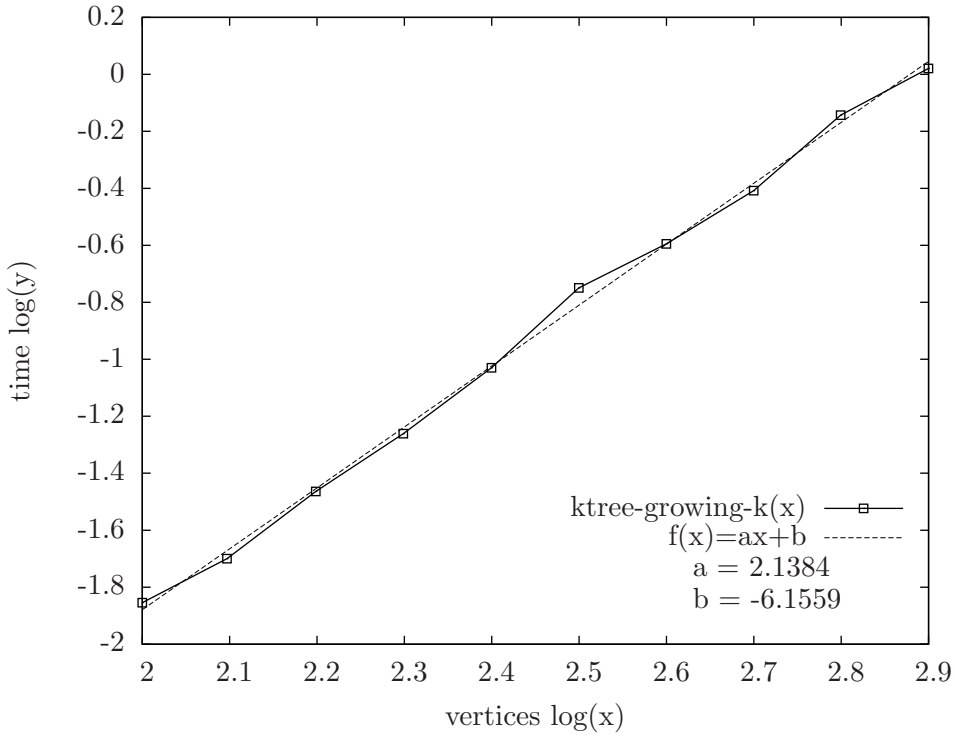
Testowanie generatora k-drzew, czyli funkcji make_random_ktree(). Sprawdziliśmy czas generacji k-drzew o stałej szerokości drzewowej i k-drzew o szerokości rosnącej z liczbą wierzchołków grafu. Dla $k = 5$ mamy rzadkie grafy z $|E| = 5n - 15 = O(n)$. Dla $k = n/2$ mamy gęste grafy z $|E| = n(3n - 2)/8 = O(n^2)$. Rysunki A.1 i A.2 potwierdzają liniową złożoność algorytmu $O(V+E)$. Rysunek A.3 potwierdza liniową złożoność pamięciową.



Rysunek A.1. Generowanie k-drzewa przy $k = 5$.

A.5. Testy generatora grafów szeregowo-równoległych

Testowanie generatora sp-grafów przypadkowych nieskierowanych. Funkcja make_random_spgraph() zwraca sp-graf w postaci instancji klasy Graph. Funkcja make_random_sptree() zwraca sp-graf w postaci binarnego sp-drzewa. Rysunek A.4 potwierdza liniową złożoność algorytmu $O(V)$ dla wersji zwracającej instancję klasy Graph.



Rysunek A.2. Generowanie k-drzewa przy $k = n/2$.

A.6. Testy wyznaczania PEO dla grafów szeregowo-równoległych

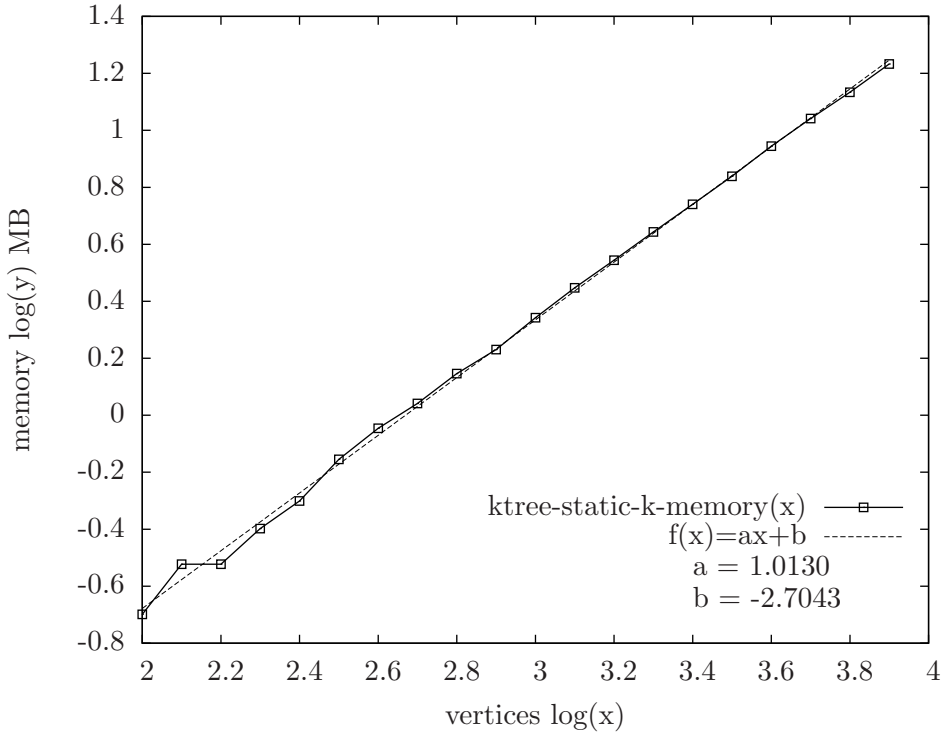
Testowanie wyznaczania PEO dla 2-drzew zawierających sp-grafy, czyli testowanie funkcji find_peo_spgraph(). Rysunek A.5 potwierdza złożoność algorytmu $O(V)$. Sprawdzane były sp-grafy przypadkowe i 2-drzewa przypadkowe.

A.7. Testy rozpoznawania grafów szeregowo-równoległych

Testowanie rozpoznawania sp-grafów i budowy sp-drzewa, czyli testowanie funkcji find_sptree(). Rysunki A.6 i A.7 potwierdzają liniową złożoność algorytmu $O(V)$. Sprawdzane były sp-grafy przypadkowe i 2-drzewa przypadkowe. Rysunek A.8 potwierdza liniową złożoność pamięciową.

A.8. Testy kolorowania wierzchołków grafów szeregowo-równoległych

Testowanie kolorowania wierzchołków sp-grafów. Algorytm kolorowania zawarty jest w klasie SPNodeColoring. Kolorowanie (rozpoznawanie) grafów dwudzielnych było sprawdzane w przeszłości i wiadomo, że działa w czasie liniowym $O(V+E)$. Z tego powodu w testach używaliśmy jedynie sp-grafów,



Rysunek A.3. Generowanie k-drzewa przy $k = 5$ (pamięć).

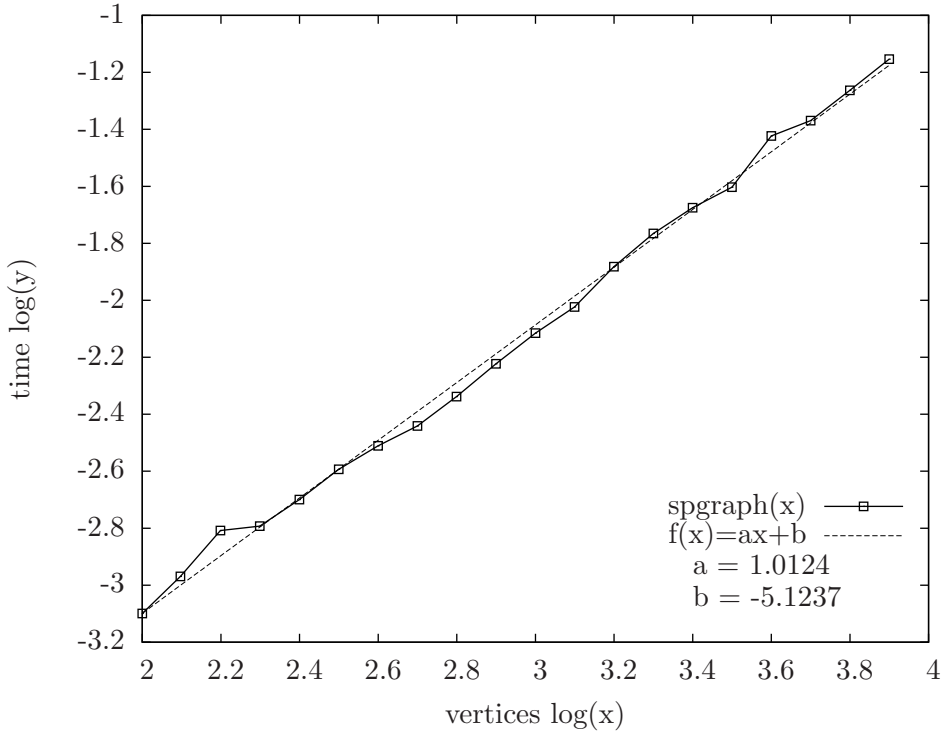
które nie są dwudzielne. W praktyce ogromna większość sp-grafów to nie są grafy dwudzielne. Rysunek A.9 potwierdza złożoność algorytmu $O(V)$. Sprawdzane były sp-grafy przypadkowe i 2-drzewa przypadkowe.

A.9. Testy wyznaczania pokrycia wierzchołkowego dla grafów szeregowo-równoległych

Testowanie wyznaczania najmniejszego pokrycia wierzchołkowego dla sp-grafów. Algorytm jest zawarty w klasie SPNodeCover. Na wejście algorytmu przekazywano sp-graf w postaci sp-drzewa. Rysunek A.10 potwierdza złożoność algorytmu $O(V)$.

A.10. Testy wyznaczania najmniejszego zbioru dominującego dla grafów szeregowo-równoległych

Testowanie wyznaczania najmniejszego zbioru dominującego dla sp-grafów. Algorytm jest zawarty w klasie SPDominatingSet. Na wejście algorytmu przekazywano sp-graf w postaci sp-drzewa. Rysunek A.11 potwierdza złożoność algorytmu $O(V)$.



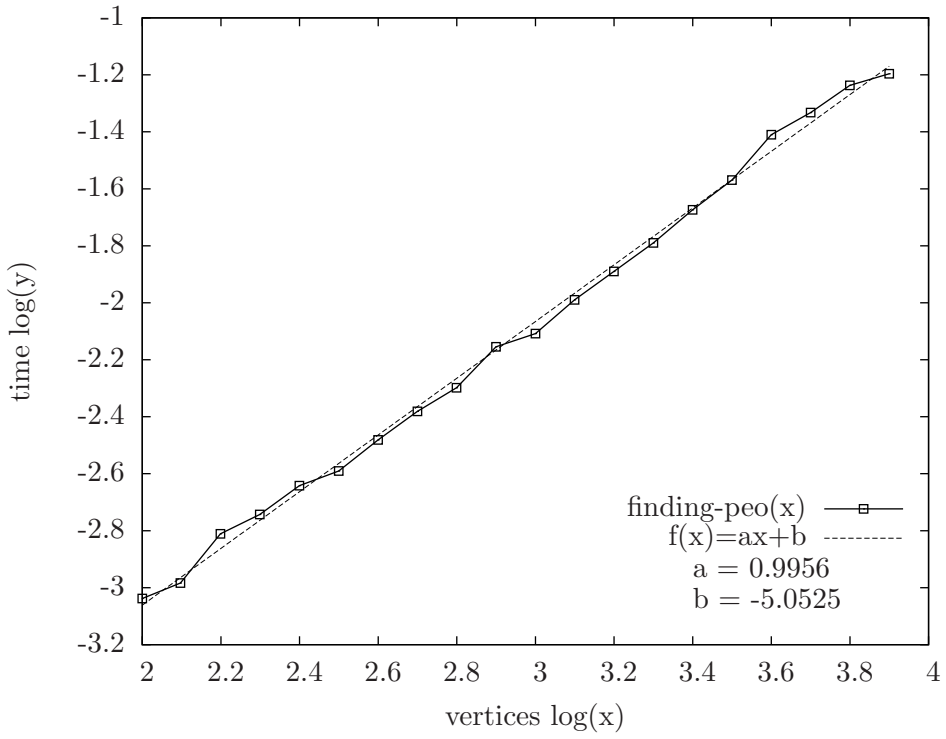
Rysunek A.4. Generowanie sp-grafu w postaci drzewa.

A.11. Testy wyznaczania największego zbioru niezależnego dla grafów szeregowo-równoległych

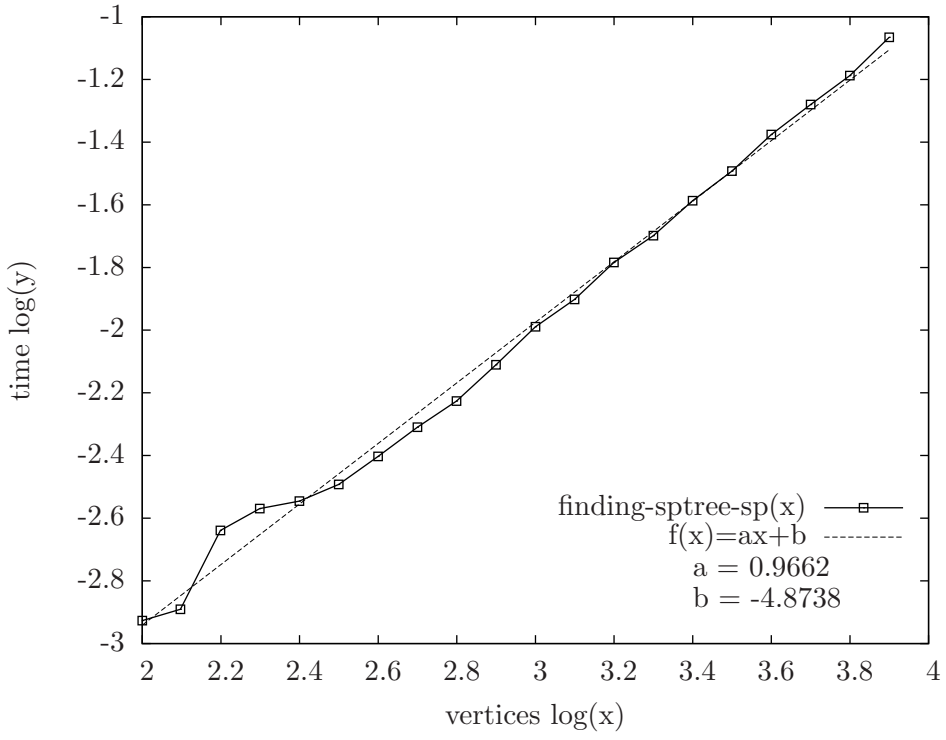
Testowanie wyznaczania największego zbioru niezależnego dla sp-grafów. Algorytm jest zawarty w klasie SPIndependentSet. Na wejście algorytmu przekazywano sp-graf w postaci sp-drzewa. Rysunek A.12 potwierdza złożoność algorytmu $O(V)$.

A.12. Testy wyznaczania największego skojarzenia dla grafów szeregowo-równoległych

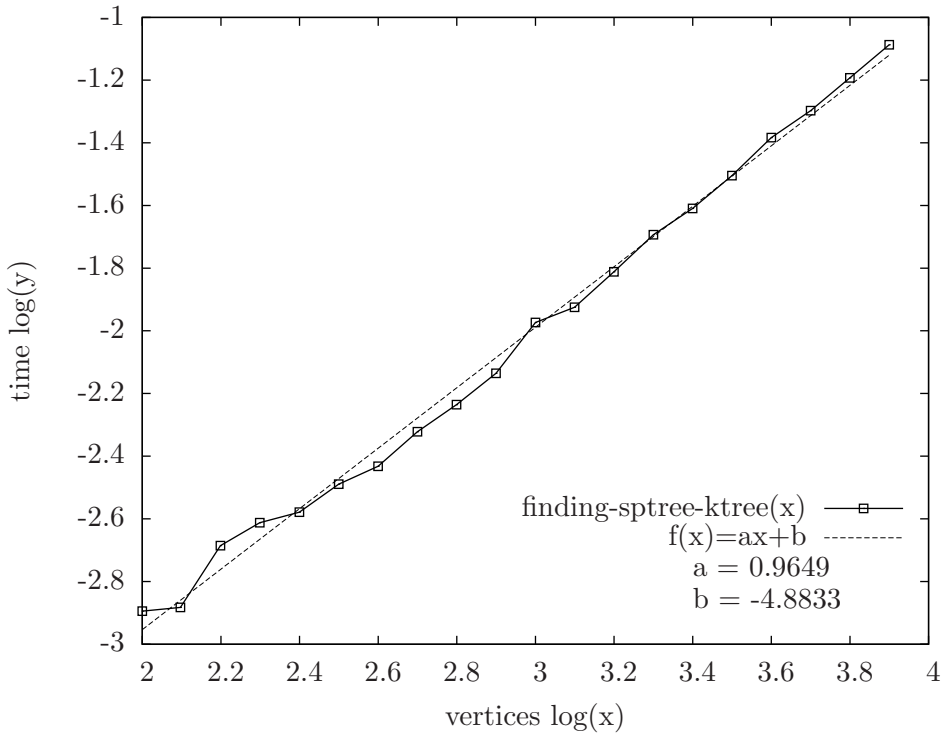
Testowanie wyznaczania największego skojarzenia dla sp-grafów. Algorytm jest zawarty w klasie SPMatchingSet. Na wejście algorytmu przekazywano sp-graf w postaci sp-drzewa. Rysunek A.13 potwierdza złożoność algorytmu $O(V)$.



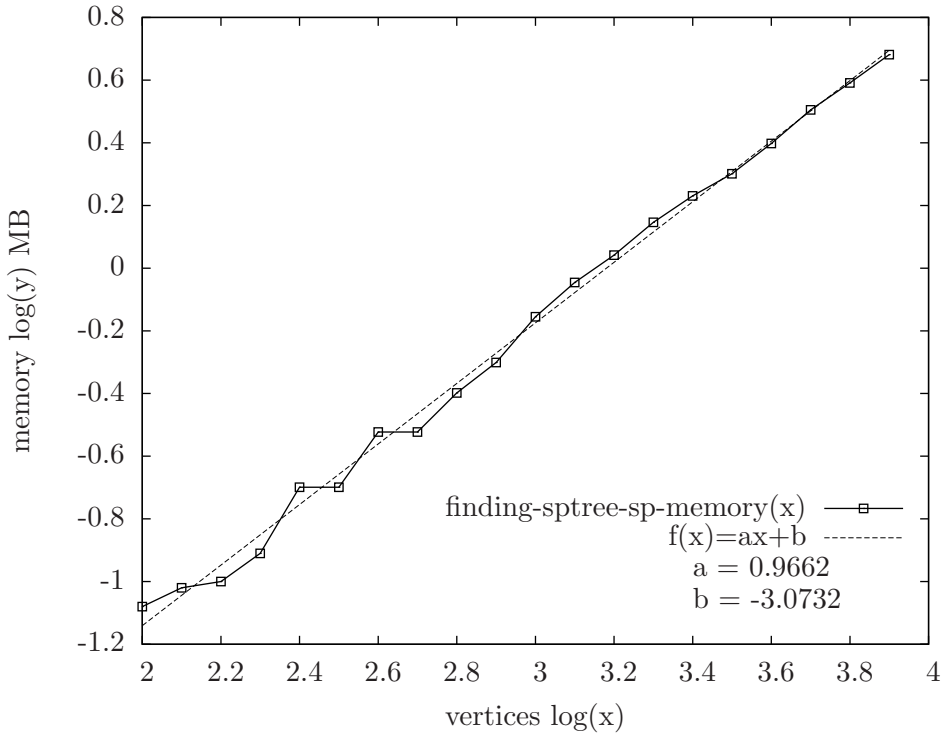
Rysunek A.5. Wyznaczanie PEO dla sp-grafu.



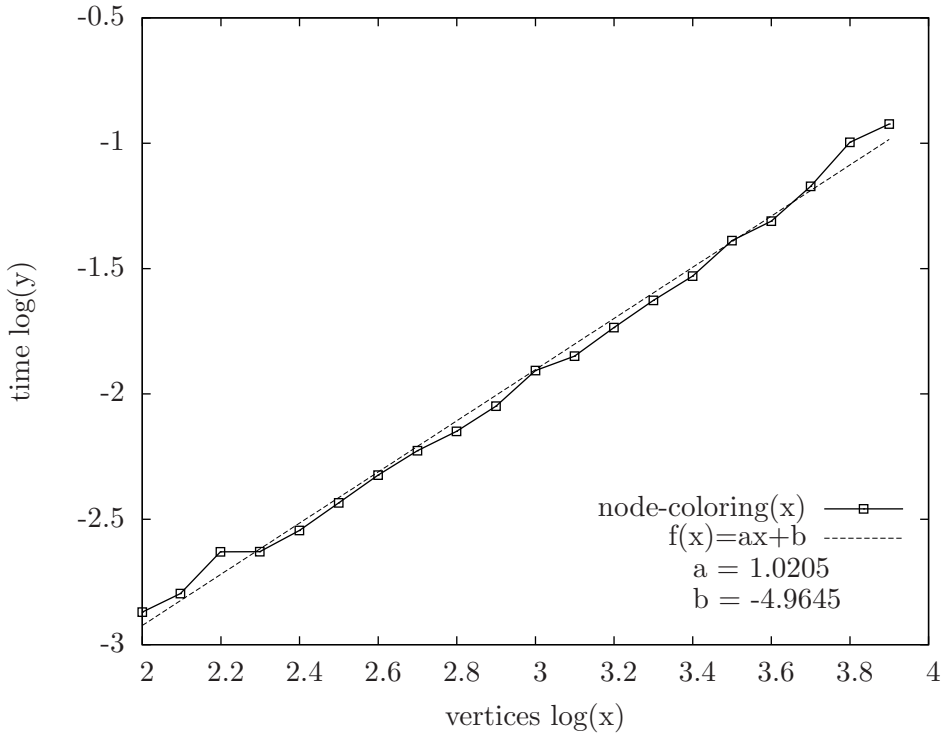
Rysunek A.6. Wyznaczanie sp-drzewa dla sp-grafu.



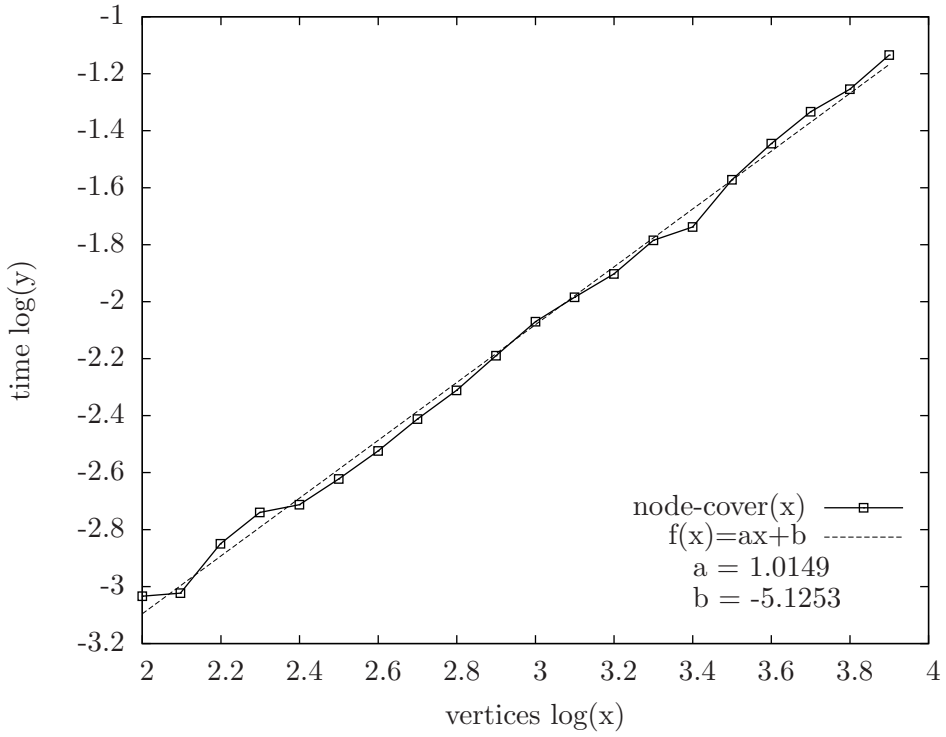
Rysunek A.7. Wyznaczanie sp-drzewa dla 2-drzewa.



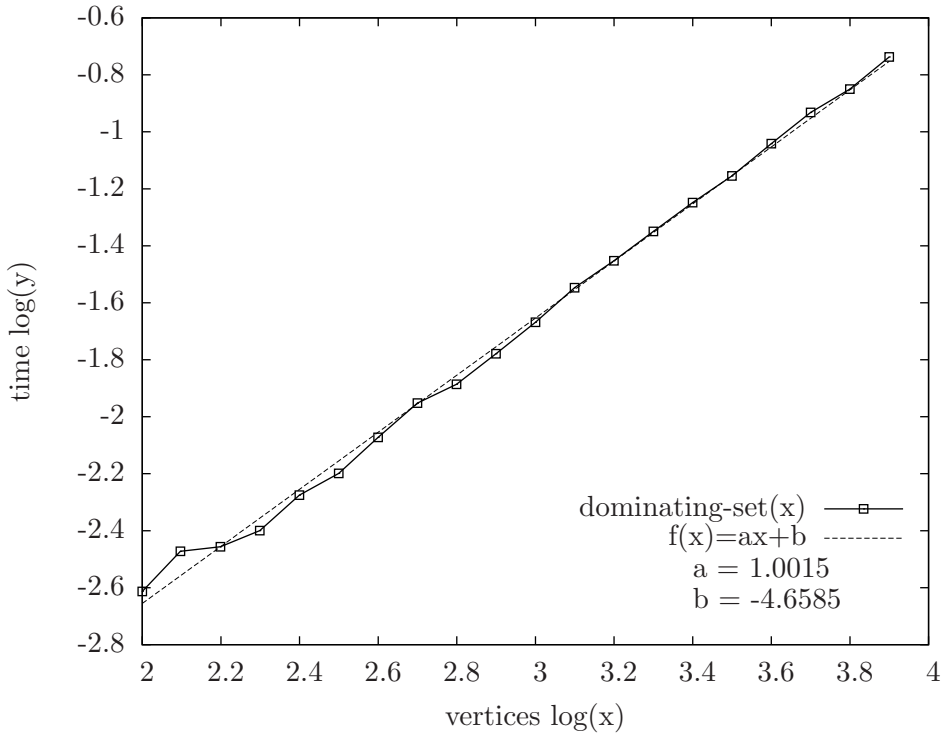
Rysunek A.8. Wyznaczanie sp-drzewa dla sp-grafu (pamięć).



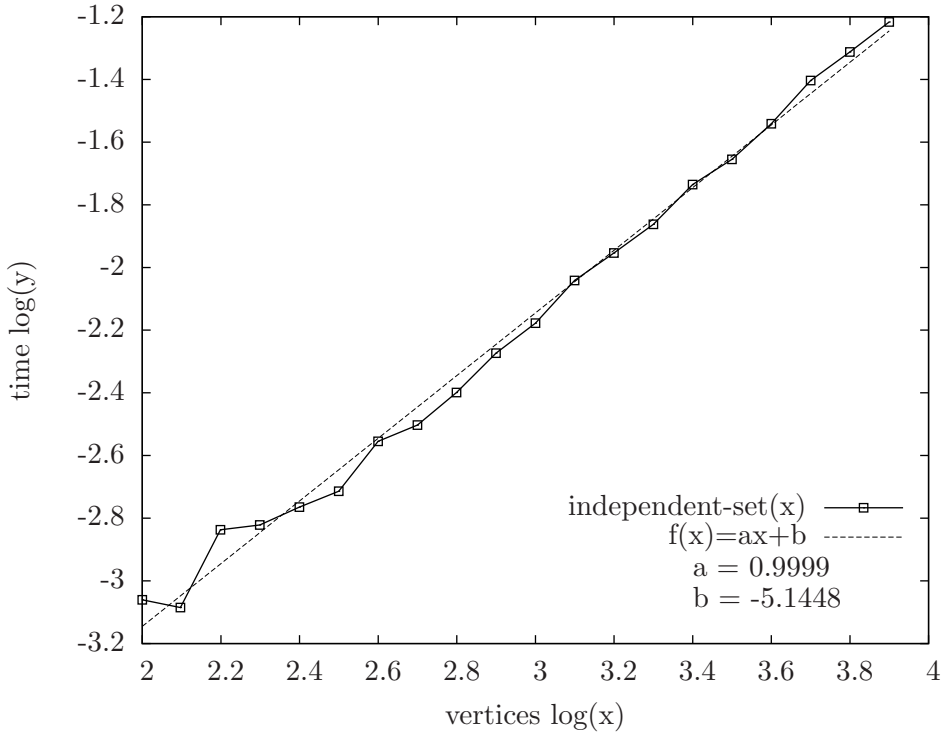
Rysunek A.9. Kolorowanie wierzchołków sp-grafu.



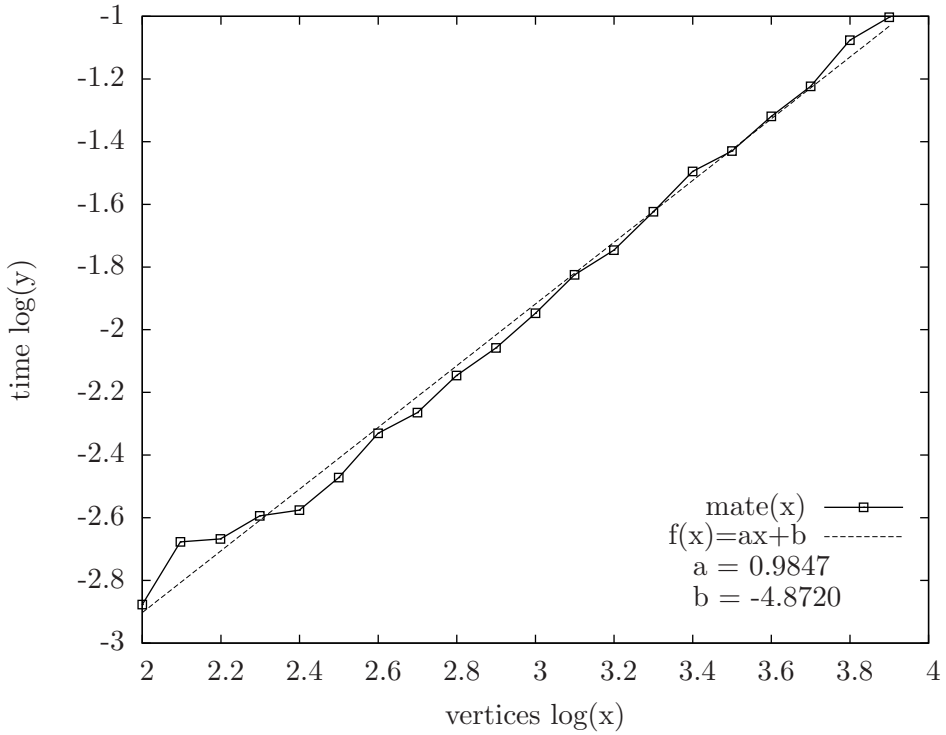
Rysunek A.10. Wyznaczanie pokrycia wierzchołkowego.



Rysunek A.11. Znajdowanie najmniejszego zbioru dominującego.



Rysunek A.12. Znajdowanie największego zbioru niezależnego.



Rysunek A.13. Znajdowanie największego skojarzenia.

Bibliografia

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

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

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

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

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

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

[7] Andrzej Kapanowski, graphs-dict, GitHub repository, 2018, `https://github.com/ufkapano/graphs-dict/`.

[8] Wikipedia, Tree decomposition, 2017, `https://en.wikipedia.org/wiki/Tree_decomposition`.

[9] Wikipedia, K-tree, 2017, `https://en.wikipedia.org/wiki/K-tree`.

[10] Wikipedia, Series-parallel graph, 2017, `https://en.wikipedia.org/wiki/Series-parallel_graph`.

[11] David Eppstein, *Parallel recognition of series-parallel graphs*, Information and Computation 98 (1), 41-55 (1992).

[12] Berry Schoenmakers, *A new algorithm for the recognition of series parallel graphs*, Technical Report CS-R9504, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, 1995.

[13] R. J. Duffin, *Topology of Series-Parallel Networks*, Journal of Mathematical Analysis and Applications 10, 303-313 (1965).

[14] Reinhard Diestel, *Graph Theory*, Graduate Texts in Mathematics, tom 173, Springer-Verlag, 2000.

[15] Hans L. Bodlaender, *Tutorial. A partial k-arboretum of graphs with bounded treewidth*, Theoretical Computer Science 209, 1-45 (1998).

[16] S. Arnborg, A. Proskurowski, D. G. Corneil, *Forbidden minors characterization of partial 3-trees*, Discrete Math. 80, 1-19 (1990).

[17] Jacobo Valdes, *Parsing Flowcharts and Series-Parallel Graphs*, Technical Report STAN-CS-78-682, Computer Science Department. Stanford University, CA, 1978.

[18] Jacobo Valdes, Robert E. Tarjan, Eugene L. Lawler, *The recognition of series parallel digraphs*, SIAM J. Comput. 11 (2), 298-313 (1982).

[19] K. Takamizawa, T. Nishizeki, N. Saito, *Linear-time computability of combinatorial problems on series-parallel graphs*. Journal of the Association for Computing Machinery 29 (3), 623-641 (1982).

[20] Yuval Caspi, Eliezer Dekel, *Edge Coloring Series Parallel Graphs*, Journal of Algorithms 18, 296-321 (1995).

[21] Xiao Zhou, Hitoshi Suzuki, Takao Nishizeki, *A Linear Algorithm for Edge-Coloring Series-Parallel Multigraphs*, Journal of Algorithms 20, 174-201 (1996).

[22] Xiao Zhou, Shin-ichi Nakano, Takao Nishizeki, *Edge-Coloring Partial k-Trees*, Journal of Algorithms 21, 598-617 (1996).

[23] Richard B. Borie, R. Gary Parker, Craig A. Tovey, *Solving Problems on Recursively Constructed Graphs*, ACM Computing Surveys 41, 4 (2008).

[24] Takao Nishizeki, Jens Vygen, Xiao Zhou, *The edge-disjoint paths problem is NP-complete for series–parallel graphs*, Discrete Applied Mathematics 115, 177-186 (2001).

[25] Aleksander Krawczyk, *Badanie grafów Halina z językiem Python*, Praca magisterska, Uniwersytet Jagielloński, Kraków, 2016.

[26] StackExchange, Theoretical Computer Science, *What separates easy global problems from hard global problems on graphs of bounded treewidth?*, 2014, `http://cstheory.stackexchange.com`.

[27] Moses Ganardi, *Matching-based algorithms for computing treewidth*, Bachelor Thesis, Aachen 2012.

[28] Andrzej Proskurowski, Maciej M. Sysło, *Efficient Computations in Tree-Like Graphs*, Computing Suppl. 7, 1-15 (1990).

[29] Hans L. Bodlaender, Babette de Fluiter, *Parallel algorithms for series parallel graphs*. In: J. Diaz, M. Serna (eds) Algorithms — ESA '96. ESA 1996. Lecture Notes in Computer Science, vol 1136. Springer, Berlin, Heidelberg 1996.

[30] Fabian Pedregosa, Memory Profiler, Python Package Index, *memory_profiler 0.54.0*, `https://pypi.org/project/memory_profiler/`.