Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Aleksander Krawczyk

Nr albumu: 1054611

Badanie grafów Halina z językiem Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2016

Oświadczenie autora pracy

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

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

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

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

Kraków, dnia Podpis kierującego pracą

Składam najserdeczniejsze podziękowania oraz wyrazy uznania Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za olbrzymie zaangażowanie, poświęcony czas, rady, okazaną cierpliwość i nieocenioną pomoc, bez której powstanie tej pracy nie byłoby możliwe.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów związanych z grafami Halina i drzewami. Grafy Halina powstają z płaskiego rysunku drzewa, które nie ma wierzchołków stopnia dwa, przez dodanie cyklu łączącego liście drzewa w kolejności wyznaczonej przez rysunek drzewa. Dla grafów Halina wiele trudnych problemów może być rozwiązanych w czasie wielomianowym, np. największy zbiór niezależny, najmniejsze pokrycie wierzchołkowe, najmniejszy zbiór dominujący, kolorowanie wierzchołków, kolorowanie krawędzi, problem komiwojażera.

Zebrano najważniejsze własności grafów Halina i grafów kołowych. Zbudowano kilka generatorów grafów i zaimplementowano algorytmy rozpoznające grafy Halina i grafy kołowe. Opisano klasę grafów konstruowanych rekurencyjnie razem z implementacjami pewnych szybkich algorytmów dla trudnych problemów (drzewa, grafy Halina). Zaprezentowano galerię małych grafów Halina.

Słowa kluczowe: grafy Halina, grafy koła, drzewa, kolorowanie grafów, zbiór niezależny, pokrycie wierzchołkowe, zbiór dominujący, problem komiwojażera

English title: Study of Halin graphs with Python

Abstract

Python implementation of selected graph algorithms connected with Halin graphs and trees are presented. Halin graphs can be formed from a tree with no degree-two vertices, embedded in the plane, by adding a cycle connecting the leaf vertices of the tree in the cyclic order given by the embedding. Halin graphs allow many hard problems to be solved in polynomial time, e.g. a maximum independent set, a minimum vertex cover, a minimum dominating set, vertex coloring, edge coloring, the travelling salesman problem.

The most important properties of Halin graphs and wheel graphs are collected. Several graph generators are built and algorithms for recognizing Halin graphs and wheel graphs are implemented. Recursively constructed graphs with some fast algorithms for hard problems are described and implemented (trees, Halin graphs). A gallery of small Halin graphs is presented.

Keywords: Halin graphs, wheel graphs, trees, graph coloring, independent set, vertex cover, dominating set, travelling salesman problem

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

W informatyce i w teorii grafów wiele problemów o praktycznym znaczeniu jest NP-zupełnych, czyli są małe szanse na odkrycie algorytmu o złożoności wielomianowej, dostarczającego rozwiązania dokładnego. W tej sytuacji są co najmniej trzy sposoby radzenia sobie z danym problemem [1].

1. Dla danych wejściowych o małym rozmiarze może wystarczyć algorytm o złożoności wykładniczej. Można też szukać algorytmów "mniej wykładniczych", które trochę powiększą rozmiar problemu nadający się do bezpośredniego rozwiązywania. Wśród algorytmów o czasowej złożoności wykładniczej można szukać algorytmów o wielomianowej (a nie wykładniczej) złożoności pamięciowej.
2. Można wyodrębnić ważne przypadki szczególne, które daje się rozwiązać w czasie wielomianowym. W praktyce też zwykle występują pewne ograniczenia na domenę problemu.
3. Czasem istnieje możliwość znalezienia rozwiązań prawie optymalnych w czasie wielomianowym. Istnieją algorytmy aproksymacyjne, dające gwarancję bycia blisko rozwiązania optymalnego.

W teorii grafów przypadki szczególne z punktu drugiego to wybrane klasy grafów, do których zawęża się dany problem. Istotne jest, aby łatwo (najlepiej w czasie liniowym w rozmiarze grafu) umieć rozpoznać grafy należące do wybranej klasy. W literaturze rozważane były między innymi następujące klasy grafów:

1. Grafy dwudzielne (ang. *bipartite graphs*).
2. Grafy regularne, np. kubiczne.
3. Grafy acykliczne (drzewa i lasy).
4. Grafy zewnętrznoplanarne (ang. *outerplanar graphs*).
5. Grafy Halina.
6. Grafy planarne (ang. *planar graphs*).
7. Grafy przedziałowe (ang. *interval graphs*).
8. Grafy szeregowo-równoległe (ang. *series-parallel graphs.*)
9. Grafy cięciwowe (ang. *chordal graphs*).
10. Grafy doskonałe (ang. *perfect graphs*).

Tematem niniejszej pracy jest badanie własności grafów Halina. Grafy Halina powstają z płaskiego rysunku drzewa T, które nie zawiera wierzchołków stopnia 2. Minimalna liczba wierzchołków drzewa T wynosi 4. Następnie należy dodać krawędzie cyklu C, który łączy kolejne liście drzewa T w taki sposób, że powstaje graf planarny. Galeria przykładowych grafów Halina znajduje się w rozdziale 7.

Grafy Halina posiadają szereg ciekawych własności, które szczegółowo omówimy w rozdziale 4. Najciekawsze wydaje się to, że wiele problemów trudnych z teorii grafów ma algorytmy wyznaczające optymalne rozwiązania w czasie wielomianowym. Można powiedzieć, że grafy Halina są uogólnieniem drzew, dla których również istnieją algorytmy znajdujące optymalne rozwiązania dla wielu trudnych problemów. Celem pracy jest po pierwsze implementacja algorytmów rozpoznawania grafów Halina i ich szczególnego przypadku, jakim są grafy kołowe. Drugim celem jest zrozumienie i implementacja algorytmów dla wielu trudnych problemów.

Do przygotowania implementacji algorytmów wykorzystamy język Python [2], ponieważ sprawdził się on w przeszłości jako doskonałe narzędzie do przejrzystej prezentacji algorytmów, a przy tym daje możliwość uruchomienia kodu z wydajnością przewidywaną przez teorię [3]. W pracy korzystaliśmy z wielu oryginalnych artykułów oraz z kilku książek po polsku poświęconym częściowo lub w całości grafom [1], [4], [5], [6].

Treść niniejszej pracy jest zorganizowana następująco. Rozdział 1 omawia cele pracy. W rozdziale 2 zebrano podstawowe definicje z teorii grafów, oraz przedstawiono szereg rodzin grafów i status wybranych problemów trudnych w odniesieniu do tych rodzin. Rozdział 3 omawia implementację grafów wykorzystaną w pracy. W rozdziale 4 zebrano najważniejsze własności grafów kołowych i ogólnych grafów Halina. Rozdział 5 dotyczy grafów konstruowanych rekurencyjnie, do których należą drzewa i grafy Halina. Omówiono algorytmy wyznaczające rozwiązania optymalne dla wielu problemów grafowych. W rozdziale 6 przedstawiono algorytmy rozpoznawania grafów kołowych i grafów Halina, które działają w czasie liniowym $O(V)$. Rozdział 7 zawiera galerię małych grafów Halina. Rozdział 8 jest podsumowaniem pracy. Dodatek A zawiera wyniki testów wybranych algorytmów.

2. Teoria grafów

Teoria grafów zajmuje się badaniem własności grafów, a także rozwija algorytmy służące wyznaczaniu pewnych wielkości dla grafów. Elementy teorii grafów wykorzystywane są w wielu dziedzinach nauki i techniki. W tym rozdziale przypomnimy podstawowe definicje i twierdzenia z teorii grafów, które będą wykorzystywane w dalszej części pracy. Podamy także, jak wyglądają rozwiązania problemów trudnych dla wybranych klas grafów.

2.1. Podstawowe definicje

Graf jest to para uporządkowana $G = (V, E)$, gdzie V to niepusty zbiór wierzchołków, a E jest zbiorem krawędzi łączących pewne dwa wierzchołki grafu. W *grafie zerowym* zbiór krawędzi jest pusty. *Wierzchołek izolowany* nie jest połączony krawędziami z innymi wierzchołkami.

Graf nieskierowany jest to graf, którego krawędzie (nieskierowane) nie mają określonej orientacji, czyli kierunku przechodzenia pomiędzy wierzchołkami. *Graf skierowany* jest to graf, w którym krawędzie (skierowane) mają określoną orientację. W takim grafie krawędź łącząca wierzchołki ma wierzchołek początkowy i wierzchołek końcowy.

Graf prosty jest to graf, w którym nie występują pętle (krawędź łącząca wierzchołek z nim samym) oraz krawędzie wielokrotne (wierzchołki połączone ze sobą więcej niż jedną krawędzią). Graf, który nie spełnia któregoś z powyższych warunków, nazywamy go *multigrafem*.

Stopień wierzchołka w grafie nieskierowanym jest liczbą krawędzi, które się z nim łączą. Pętla liczona jest za 2. W grafie skierowanym wyróżniamy stopień wierzchołka wychodzący (liczba krawędzi wychodzących z wierzchołka), oraz na stopień wierzchołka wchodzący (liczba krawędzi wchodzących do wierzchołka).

Ścieżka w grafie jest to ciąg krawędzi, który prowadzi z wierzchołka startowego do wierzchołka końcowego. W grafie może istnieć wiele różnych ścieżek pomiędzy wybranymi wierzchołkami. *Ścieżka prosta* nie ma powtarzających się wierzchołków. Długość ścieżki to liczba krawędzi w niej zawartych. *Graf spójny* jest to graf nieskierowany, w którym dla każdej pary wierzchołków istnieje ścieżka pomiędzy nimi. *Graf silnie spójny* jest to graf skierowany, w którym dla każdej pary wierzchołków istnieje ścieżka pomiędzy nimi.

Cykl jest to ścieżka kończąca się w wierzchołku, w którym się rozpoczęła. *Cykl prosty* nie ma powtarzających się wierzchołków, z wyjątkiem ostatniego. Graf acykliczny nie zawiera żadnych cykli. Cykl Hamlitona jest to cykl prosty zawierający wszystkie wierzchołki grafu.

2.2. Grafy dwudzielne

Definicja: Graf dwudzielny (ang. *bipartite graph*) jest to graf prosty nieskierowany, którego zbiór wierzchołków V można podzielić na dwa rozłączne niepuste podzbiory V_1 i V_2, $V = V_1 \cup V_2$. Wtedy każda krawędź grafu ma jeden koniec w V_1, a drugi w V_2. Każdy graf dwudzielny jest grafem doskonałym [7].

Rozpoznawanie: Wystarczy skorzystać z BFS lub DFS, czas jest liniowy.

Kolorowanie wierzchołków: Liczba chromatyczna wynosi $\chi(G) = 2$, ponieważ jeden kolor wykorzystujemy dla wierzchołków z V_1, a drugi kolor dla wierzchołków z V_2. Czas kolorowania jest liniowy, bo korzysta się z BFS.

Kolorowanie krawędzi: Z twierdzenia Königa dla grafów dwudzielnych wynika, że indeks chromatyczny wynosi $\chi'(G) = \Delta(G)$. [jaka wydajność algorytmu?]

Problem komiwojażera: Istnieje twierdzenie, że każdy cykl w grafie dwudzielnym ma długość parzystą. Stąd wynika, że cykl Hamiltona może istnieć tylko w grafie dwudzielnym o parzystej liczbie wierzchołków. Inne twierdzenie mówi, że jeżeli cykl Hamiltona istnieje, to dwa podzbiory wierzchołków muszą być równoliczne.

Największe skojarzenie: W grafach dwudzielnych największe skojarzenie można znaleźć w czasie wielomianowym. Przykładowe algorytmy: metoda Forda-Fulkersona, metoda ścieżki powiększającej, algorytm Hopcrofta-Karpa. W dowolnych grafach największe skojarzenie można znaleźć za pomocą słynnego algorytmu kwiatowego (ang. *blossom algorithm*) Edmondsa, którego złożoność jest szacowana na co najwyżej $O(V^4)$ [8], [9].

Najmniejsze pokrycie wierzchołkowe: Twierdzenie Königa mówi, że w grafach dwudzielnych najmniejsze pokrycie wierzchołkowe jest liczbowo równe największemu skojarzeniu. Stąd problem najmniejszego pokrycia wierzchołkowego ma rozwiązanie w czasie wielomianowym.

Największy zbiór niezależny: W grafach dwudzielnych spójnych, dopełnienie zbioru niezależnego stanowi pokrycie wierzchołkowe, czyli istnieje rozwiązanie w czasie wielomianowym. Twierdzenie Königa mówi, że w grafach dwudzielnych liczba wierzchołków w największym zbiorze niezależnym równa się liczbie krawędzi w minimalnym pokryciu krawędziowym.

Największa klika: Największa możliwa klika odpowiada K_2, bo K_3 jest cyklem nieparzystym, a cykle nieparzyste nie istnieją w grafach dwudzielnych. Problem ma rozwiązanie wielomianowe $O(V^2)$, ponieważ wystarczy rozważyć wszystkie podzbiory mające dwa wierzchołki. A właściwie końce każdej krawędzi dają klikę, co w reprezentacji list sąsiedztwa pozwala wypisać kliki K_2 w czasie $O(E)$.

2.3. Grafy regularne

Definicja: Graf regularny (ang. *regular graph*) jest to graf prosty nieskierowany, w którym każdy wierzchołek ma taki sam stopień r. Mówimy, że jest to graf r-regularny lub graf regularny stopnia r [10]. Grafy kubiczne (ang. *cubic graphs*) są to grafy 3-regularne.

Rozpoznawanie: Wystarczy sprawdzić stopień każdego wierzchołka grafu. W zależności od implementacji może to zająć czas od $O(V)$ do $O(V^2)$.

Kolorowanie wierzchołków: Zgodnie z twierdzeniem Brooksa każdy spójny graf r-regularny, różny od K_{r+1} i cyklu nieparzystego, może być pokolorowany z użyciem r kolorów. Stąd grafy kubiczne różne od K_4 są 3-kolorowalne wierzchołkowo.

Kolorowanie krawędzi: Z twierdzenia Vizinga wynika, że potrzeba r lub $r+1$ kolorów do pokolorowania krawędzi. Spójny graf kubiczny bez mostów, o indeksie chromatycznym $\chi'(G) = 4$, to jest żmirłacz (ang. *snark*) [11]. Najmniejszym żmirłaczem jest graf Petersena.

Największy zbiór niezależny: Korzystając z kolorowania wierzchołków można stwierdzić, że każdy spójny graf r-regularny, różny od K_{r+1}, ma zbiór niezależny o liczności co najmniej n/r. Wystarczy wybrać najliczniejszą klasę koloru w r-kolorowaniu.

Najmniejsze pokrycie wierzchołkowe: Problem jest NP-zupełny nawet dla grafów kubicznych [12].

Największa klika: W grafie r-regularnym największa możliwa klika odpowiada K_{r+1} i wtedy jest to osobna składowa spójna. Liczba możliwych klik jest rzędu $O(V^{r+1})$, więc rozwiązanie jest wielomianowe. Trzeba rozważyć wszystkie podzbiory mające $r+1$ lub mniej wierzchołków [12].

2.4. Drzewa

Definicja: Drzewo jest to graf nieskierowany, spójny, acykliczny. Drzewo ma $m = n - 1$ krawędzi. Każde dwa wierzchołki drzewa są połączone dokładnie jedną ścieżką prostą [13]. Liście to wierzchołki stopnia jeden w drzewie. Każde drzewo jest grafem dwudzielnym. Szczególne drzewa to graf liniowy i graf gwiazda.

Rozpoznawanie: Wystarczy skorzystać z BFS lub DFS, czas jest liniowy.

Kolorowanie wierzchołków: Drzewa są dwudzielne, więc liczba chromatyczna wynosi $\chi(G) = 2$.

Kolorowanie krawędzi: Drzewa są dwudzielne, więc z twierdzenia Königa indeks chromatyczny wynosi $\chi'(G) = \Delta(G)$.

Problem komiwojażera: Drzewa są acykliczne, więc nie istnieje w nich cykl Hamiltona.

Największy zbiór niezależny: W grafach dwudzielnych spójnych największy zbiór niezależny jest dopełnieniem najmniejszego pokrycia wierzchołkowego.

Najmniejsze pokrycie wierzchołkowe: Istnieje algorytm zachłanny działający w czasie liniowym.

Największe skojarzenie: Drzewa są dwudzielne, więc problem ma rozwiązanie wielomianowe.

Największa klika: Największa możliwa klika w drzewie odpowiada K_2, bo już K_3 tworzy cykl, a drzewa są acykliczne. Stąd wszystkie największe kliki można wypisać w czasie $O(V)$, bo końce każdej krawędzi tworzą taką klikę.

2.5. Grafy zewnętrznoplanarne

Definicja: Graf zewnętrznoplanarny (ang. *outerplanar graph*) jest to graf planarny, który narysowany na płaszczyźnie bez przecięć krawędzi ma wszystkie wierzchołki należące do ściany zewnętrznej (ang. *outer face*). Graf maksymalny zewnętrznoplanarny (ang. *maximal outerplanar graph*) ma maksymalną liczbę krawędzi równą $m = 2n - 3$, a wszystkie ściany wewnętrzne są trójkątami [14]. Grafy zewnętrznoplanarne nie mogą zawierać podgrafu homeomorficznego z K_4 lub z $K_{2,3}$.

Rozpoznawanie: W czasie liniowym.

Kolorowanie wierzchołków: Jeżeli graf zewnętrznoplanarny ma co najmniej jedną krawędź, to liczba chromatyczna wynosi $\chi(G) = 2$ (graf dwudzielny) lub $\chi(G) = 3$ [15]. 3-kolorowanie wierzchołkowe może być znalezione metodą zachłanną. Należy tymczasowo usunąć wierzchołki stopnia co najwyżej dwa, pokolorować graf, następnie przywrócić usunięte wierzchołki, dając im kolor różny od ich sąsiadów.

Kolorowanie krawędzi: Jeżeli graf zewnętrznoplanarny nie jest cyklem nieparzystym, to indeks chromatyczny wynosi $\chi'(G) = \Delta(G)$ [16].

Problem komiwojażera: Grafy zewnętrznoplanarne są co najwyżej 2-spójne, a każdy 2-spójny graf zewnętrznoplanarny ma dokładnie jeden cykl Hamiltona (ściana zewnętrzna) [25].

Problem minimalnego cyklu dominującego: Problem polega na znalezieniu najmniejszego cyklu w grafie, takiego że każdy wierzchołek grafu albo należy do cyklu, albo sąsiaduje z wierzchołkiem należącym do cyklu. Dla grafów zewnętrznoplanarnych 2-spójnych cykl można znaleźć w czasie liniowym [17].

2.6. Grafy planarne

Definicja: Graf planarny (ang. *planar graph*) jest to graf, który może być narysowany na płaszczyźnie bez przecięć krawędzi. Twierdzenie Kuratowskiego mówi, że graf jest planarny wtedy i tylko wtedy, gdy nie zawiera podgrafu homeomorficznego z K_5 lub z $K_{3,3}$ [18]. Każdy graf planarny, którego ściany mają długość parzystą, jest grafem dwudzielnym.

Rozpoznawanie: Istnieją co najmniej trzy odmiany algorytmów do testowania planarności grafu w czasie liniowym. Implementacje tych algorytmów są skomplikowane.

Kolorowanie wierzchołków: Twierdzenie o czterech barwach mówi, że dla grafów planarnych liczba chromatyczna $\chi(G) \leqslant 4$ (Appel, Haken, 1976). Problem, czy dany graf planarny jest 3-kolorowalny wierzchołkowo, jest NP-zupełny, nawet przy założeniu $\Delta(G) \geqslant 4$ [12]. Jeżeli $\Delta(G) = 3$, to graf planarny różny od K_4 jest 3-kolorowalny z twierdzenia Brooksa. Istnieją algorytmy działające w czasie liniowym, które wykorzystują 5 kolorów dla grafu planarnego [19]. Algorytm sekwencyjny SL (ang. *Smallest Last*) pokoloruje graf planarny najwyżej sześcioma kolorami.

Kolorowanie krawędzi: Z twierdzenia Vizinga wynika, że dla dowolnego grafu prostego indeks chromatyczny wynosi $\chi'(G) = \Delta(G)$ (grafy klasy 1) lub

$\chi'(G) = \Delta(G) + 1$ (grafy klasy 2). Dla grafów planarnych o wysokim $\Delta(G)$ pokazano, że $\chi'(G) = \Delta(G)$.

Problem komiwojażera: Problem jest NP-zupełny dla grafów planarnych, nawet przy założeniu $\Delta(G) \leqslant 3$ [12]. Dla metrycznego problemu komiwojażera istnieje algorytm 2-aproksymacyjny o złożoności $O(V^2)$.

Problem minimalnego cyklu dominującego: Problem jest NP-zupełny dla grafów planarnych.

Największy zbiór niezależny: Problem jest NP-trudny dla grafów planarnych. Z drugiej strony, istnieją wydajne algorytmy aproksymacyjne, które znajdują duże zbiory niezależne. Dopełnienie zbioru niezależnego stanowi pokrycie wierzchołkowe. Z twierdzenia o czterech barwach wynika, że każdy graf planarny posiada zbiór niezależny o rozmiarze nie mniejszym niż $n/4$.

Najmniejsze pokrycie wierzchołkowe: Problem jest NP-zupełny dla grafów planarnych z $\Delta(G) \geqslant 3$.

Największa klika: Z twierdzenia Kuratowskiego wynika, że największa możliwa klika odpowiada K_4. Problem ma rozwiązanie wielomianowe $O(V^4)$, ponieważ wystarczy rozważyć wszystkie podzbiory mające cztery lub mniej wierzchołków [12].

Minimalne drzewo rozpinające: Problem znalezienia minimalnego drzewa rozpinającego (MST) dla grafów planarnych może być rozwiązany w czasie liniowym $O(V)$ [20], [21]. Dla ogólnych grafów jest to czas $O(V^2)$ lub $O(E \log V)$.

3. Implementacja grafów

W pracy bazowaliśmy na podstawowych klasach istniejących w bibliotece grafowej, rozwijanej w Instytucie Fizyki UJ. Moduł edges zawiera klasę Edge, która reprezentuje krawędzie skierowane ważone (domyślna waga 1). Moduł graphs zawiera klasę Graph, reprezentującą grafy skierowane i nieskierowane. Algorytmy grafowe są implementowane jako klasy i zwykle mają dedykowane moduły.

Przykładowa sesja interaktywna przedstawia korzystanie z grafów Halina (listing 3.1).

Listing 3.1. Sesja interaktywna z grafami Halina.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> graph_factory = GraphFactory(Graph)
# Rozpoznawanie grafow kolowych.
>>> from wheels import is_wheel, WheelGraph
>>> G = graph_factory.make_complete(4, False)
>>> algorithm = WheelGraph(G)
>>> algorithm.run()
>>> algorithm.hub
0
>>> is_wheel(G)
True
# Rozpoznawanie grafow Halina.
>>> from halin import HalinGraph
>>> G = graph_factory.make_halin(10, False)
# G = graph_factory.make_halin_cubic(10, False)
>>> algorithm = HalinGraph(G)
>>> algorithm.run()
>>> algorithm.outer
set(...)
```

Druga sesja interaktywna pokazuje zastosowanie różnych algorytmów dla drzew (listing 3.2).

Listing 3.2. Sesja interaktywna z drzewami.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from bipartite import is_bipartite
>>> from connected import is_connected
>>> from factory import GraphFactory
>>> graph_factory = GraphFactory(Graph)
>>> V = 10
>>> T = graph_factory.make_tree(V, False)    # random tree
>>> is_bipartite(T)
True
```

```
>>> is_connected(T)
True
# Obliczanie najmniejszego pokrycia wierzcholkowego.
>>> from treecover import TreeNodeCoverSet
>>> algorithm = TreeNodeCoverSet(T)
>>> algorithm.run()
>>> algorithm.node_cover
set(...)
# Obliczanie najwiekszego zbioru niezaleznego.
>>> from treeiset import TreeIndependentSet
>>> algorithm = TreeIndependentSet(T)
>>> algorithm.run()
>>> algorithm.independent_set
set(...)
# Obliczanie najmniejszego zbioru dominujacego.
>>> from boriedset import BorieDominatingSet
>>> algorithm = BorieDominatingSet(T)
>>> algorithm.run()
>>> algorithm.dominating_set
set(...)
```

4. Własności grafów Halina

W tym rozdziale zostaną podane najważniejsze własności grafów Halina. Szczególnym przypadkiem grafu Halina jest graf koło, który będzie opisany osobno.

4.1. Ogólne grafy Halina

Graf Halina jest szczególnym grafem planarnym, zapisywanym symbolicznie jako $H = T \cup C$. Powstaje z drzewa T narysowanego na płaszczyźnie bez przecinania się krawędzi. Drzewo T musi mieć co najmniej cztery wierzchołki, przy czym nie mogą występować wierzchołki stopnia dwa. Do drzewa T dodaje się krawędzie, które łączą liście drzewa (wierzchołki stopnia jeden) w jeden cykl C [22]. Z konstrukcji wynika, że grafy Halina mają wszystkie wierzchołki stopnia co najmniej trzy. Nazwa grafów pochodzi od niemieckiego matematyka Rudolfa Halina, który badał te grafy w roku 1971 [23]. Kubiczne grafy Halina były badane przez Kirkmana ponad sto lat wcześniej [24].

Wachlarze: Wachlarz (ang. *fan*) lub wiatrak w grafie Halina jest indukowany przez wierzchołek v drzewa T, który nie jest liściem. Wierzchołek v jest połączony z dokładnie jednym innym wierzchołkiem u drzewa T, który też nie jest liściem. Inni sąsiedzi wierzchołka v leżą na cyklu C i tworzą zbiór, który oznaczymy przez $C(v)$. Jeżeli graf Halina nie jest grafem kołowym, to ma co najmniej dwa wachlarze. Ściągnięcie wachlarza do punktu na cyklu C tworzy nowy (mniejszy) graf Halina. Taka procedura redukcji rekurencyjnie definiuje grafy Halina, a kończy się uzyskaniem grafu kołowego. Każdy wachlarz jest połączony z pozostałą częścią grafu Halina poprzez trzy krawędzie, jedną należącą do drzewa T i dwiema z cyklu C (*3-edge cutset*).

Twierdzenie: Grafy Halina nie są dwudzielne. Wynika to ze stwierdzenia, że każde drzewo bez wierzchołków stopnia dwa zawiera dwa liście mające tego samego rodzica. Na płaskim rysunku te dwa liście w grafie Halina są połączone krawędzią, a więc powstaje trójkąt.

Twierdzenie: *Halin graphs are edge-minimal 3-connected planar graphs* [23]. Wszystkie wierzchołki grafu $H = T \cup C$ mają stopień co najmniej 3. Pokażemy, że dla dowolnych wierzchołków u i v istnieją trzy ścieżki je łączące, które są rozłączne wierzchołkowo [25]. Pierwsza ścieżka p (wyznaczona jednoznacznie) łączy u i v w drzewie T. Jeżeli u i v należą do cyklu C, to dzielą one cykl na dwie ścieżki, obie rozłączne ze ścieżką p.

Niech u będzie wewnętrznym wierzchołkiem drzewa T. Ma stopień co najmniej 3 i jest jedynym wspólnym elementem trzech lub więcej poddrzew drzewa T. Stąd istnieją dwie rozłączne ścieżki łączące u z dwoma liśćmi T, rozłączne ze ścieżką p. Oznaczmy te liście przez u_1 i u_2. Podobne rozumowanie można przeprowadzić dla wierzchołka v, gdzie będą dwie ścieżki do liści $v \to v_1$ i $v \to v_2$. Liście v_1 i v_2 leżą na zewnątrz ścieżki $u_1 \to u_2$. Załóżmy, że ścieżki $u_1 \to v_1$ i $u_2 \to v_2$, leżące wzdłuż cyklu C, są rozłączne. Wtedy ścieżki p, $u \to u_1 \to v_1 \to v$ i $u \to u_2 \to v_2 \to v$ są wzajemnie rozłączne.

Twierdzenie: Zewnętrzna ściana grafu Halina ma $m - n + 1$ krawędzi [25]. Z definicji grafu Halina z n wierzchołkami zachodzi równość $m = m_t + m_c$, gdzie $m_t = n - 1$ to liczba krawędzi drzewa T, a m_c to liczba krawędzi cyklu C łączącego liście drzewa. Stąd $m_c = m - m_t = m - n + 1$. Dla grafów planarnych 3-spójnych istnienie ściany z $m - n + 1$ krawędziami jest warunkiem wystarczającym, aby to był graf Halina [25].

Twierdzenie: Ograniczenia na liczbę krawędzi (i wierzchołków) ściany zewnętrznej są następujące:

$$\frac{n}{2} + 1 \leqslant m_c \leqslant n - 1. \tag{4.1}$$

Górne ograniczenie ma miejsce dla grafu koła. Dolne ograniczenie pojawia się dla grafu kubicznego (n parzyste), który jest podobny do naszyjnika (ang. *necklace*) [26]. Warto zauważyć, że każda krawędź należąca do ściany zewnętrznej należy innej ściany wewnętrznej grafu Halina.

Twierdzenie: Każdy graf Halina jest hamiltonowski [27]. W grafach Halina ściana zewnętrzna ma wspólną krawędź z każdą inną ścianą grafu, co w połączeniu z 3-spójnością powoduje, że graf jest hamiltonowski [28]. Co więcej, usunięcie jednego dowolnego wierzchołka z grafu Halina daje w wyniku graf hamiltonowski. Można również sprawdzić, że dla każdej krawędzi e, istnieje cykl Hamiltona zawierający e, oraz istnieje inny cykl Hamiltona, który nie zawiera e. Barefoot pokazał, że grafy Halina są hamiltonowsko spójne, czyli istnieje ścieżka Hamiltona pomiędzy każdą parą wierzchołków [29].

Twierdzenie: Każdy graf Halina jest prawie *pancykliczny* (ang. *pancyclic*), czyli zawiera cykle o długości od 3 do n, ewentualnie z wyjątkiem jednego cyklu o długości parzystej [30], [31]. Jeżeli w grafie Halina wszystkie wierzchołki stopnia 3 leżą na cyklu zewnętrznym C, to graf jest w pełni pancykliczny.

Problem komiwojażera: Problem komiwojażera dla grafów Halina ma rozwiązanie wielomianowe [32]. Jest to związane z istnieniem pewnej dekompozycji w grafach Halina i w grafach pryzmatycznych (ang. *prismatic graphs*) [33].

Rozpinający podgraf Halina: Ze względu na interesujące własności grafów Halina pojawiła się w literaturze propozycja szukania rozpinającego podgrafu Halina (ang. *spanning Halin subgraph*) dla danego grafu. Niestety problem okazał się w ogólności NP-zupełny [26], więc zaczęto poszukiwania pewnych specjalnych klas grafów, w których łatwo można znaleźć ten podgraf.

Rozpoznawanie grafów Halina: W literaturze można znaleźć kilka różnych sposobów rozpoznawania grafów Halina.

[Opis 1] Należy sprawdzić, czy graf jest 3-spójny. Takie grafy mają jednoznaczne zanurzenie w płaszczyznę. Następnie należy znaleźć graf topologiczny (ang. *plane embedding*). W końcu trzeba znaleźć taką ścianę, że po usunięciu jej krawędzi (cykl C) otrzymamy drzewo T bez wierzchołków stopnia 2 [32].

[Opis 2] Należy znaleźć graf topologiczny, jeżeli istnieje. Następnie należy znaleźć ścianę zewnętrzną liczącą co najmniej $n/2+1$ wierzchołków, wszystkie stopnia 3. Może być co najwyżej cztery takie ściany. Dla każdej takiej ściany należy sprawdzić, czy reszta grafu tworzy drzewo, którego liście to wierzchołki należące do ściany zewnętrznej [25].

[Opis 3] Należy wykonać dekompozycję Eppsteina, która redukuje graf Halina do grafu K_4. Dekompozycja polega na znalezieniu wierzchołków stopnia trzy, a następnie rozpoznaje się dwie sytuacje: (1) wierzchołek leży na obwodzie wachlarza, a wtedy można go usunąć z obwodu, (2) wierzchołek z dwoma sąsiadami tworzy trójkąt, który można ściągnąć do punktu. Obie operacje są kolejno zapisywane w dzienniku, aż do osiągnięcia grafu K_4. Następnie wykonuje się analizę dziennika operacji, podczas której zostaje wyznaczony zbiór wierzchołków tworzący cykl zewnętrzny C grafu Halina. Warto zauważyć, że na początku analizy dziennika należy ustalić który wierzchołek grafu K_4 jest wewnętrzny (z drzewa T). Czasem informacja zgromadzona podczas dekompozycji jest niewystarczająca i należy metodą prób i błędów sprawdzić jedną z najwyżej trzech możliwości. Z drugiej strony, dla pewnych grafów istnieje więcej niż jedna możliwość płaskiego narysowania grafu, co właśnie koresponduje z opisaną wcześniej niejednoznacznością. Jeżeli analizowany graf nie jest grafem Halina, to zostanie to rozpoznane na etapie dekompozycji.

Kolorowanie wierzchołków: Grafy Halina są planarne, ale nie są dwudzielne, stąd liczba chromatyczne wynosi $\chi(H) = 3$ (np. W_5, W_7, 3-pryzma) lub $\chi(H) = 4$ (np. K_4, W_6). Kubiczne grafy Halina różne od K_4 można pokolorować optymalnie trzema kolorami przy wykorzystaniu twierdzenia Brooksa.

Kolorowanie krawędzi: Indeks chromatyczny wszystkich grafów Halina wynosi $\chi'(H) = \Delta(H)$ (grafy klasy 1) [26].

Największy zbiór niezależny: Problem może być rozwiązany w czasie liniowym za pomocą programowania dynamicznego [34].

Skojarzenie o największej wadze: Problem może być rozwiązany w czasie liniowym $O(V)$ metodą ściągania wachlarzy [35].

Największa klika: Grafy Halina są planarne, czyli największa możliwa klika to cały graf $K_4 = W_4$. W innych grafach Halina największa klika to K_3, która jest częścią wachlarza lub elementem grafu kołowego.

Uogólnione grafy Halina: W literaturze badane są również uogólnione grafy Halina, czyli *k-Halin graphs*. Jest to graf $G = (V, E)$ planarny, bez wierzchołków stopnia 2. Zbiór krawędzi E może być podzielony na zbiory $E = T \cup C_1 \cup \ldots \cup C_k$, gdzie T jest drzewem bez wierzchołków stopnia 2, C_i są to parami rozłączne cykle, których wierzchołki dają zbiór wszystkich liści drzewa T. Zwykłe grafy Halina to są *1-Halin graphs*. Każdy 2-spójny *2-Halin graph* jest hamiltonowski [36].

4.2. Grafy koła

Graf koło W_n powstaje przez dodanie do cyklu C_{n-1} nowego wierzchołka i połączenie tego wierzchołka ze wszystkimi wierzchołkami cyklu C_{n-1}. Poniżej podamy najważniejsze własności grafu koła $W_n = (V, E)$.

— Liczba wierzchołków $|V| = n > 3$, liczba krawędzi $|E| = m = 2n - 2$.
— Stopnie wierzchołków $\delta(W_n) = 3$ (najmniejszy stopień), $\Delta(W_n) = n - 1$ (największy stopień centralnego wierzchołka).
— Średnica grafu (ang. *diameter*) wynosi 1 ($n = 4, W_4 = K_4$) lub 2 ($n > 4$) [odległość na jaką są oddalone dwa najodleglejsze wierzchołki grafu].
— Obwód grafu (ang. *girth*) wynosi 3 [długość najkrótszego cyklu zawartego w grafie]. Liczba wszystkich cykli w W_n wynosi $1 + (n - 2)(n - 1) = n^2 - 3n + 3$ [37].
— Liczba chromatyczna $\chi(W_n) = 3$ (n nieparzyste) lub $\chi(W_n) = 4$ (n parzyste) [kolorowanie wierzchołków].
— Indeks chromatyczny $\chi'(W_n) = \Delta(W_n) = n - 1$ [kolorowanie krawędzi].
— Wielomian chromatyczny grafu W_n ma postać $P(x) = x[(x - 2)^{n-1} - (-1)^n(x - 2)]$.
— Spójność wierzchołkowa $\kappa(W_n) = 3$.
— Spójność krawędziowa $\lambda(W_n) = \kappa'(W_n) = 3$.
— Grafy koła są planarne. Z twierdzenia Eulera wynika, że liczba ścian wynosi $f = n$. Zewnętrzna ściana ma $n - 1$ krawędzi.
— Każdy graf koło jest grafem Halina.

— Graf koło jest identyczny ze swoim grafem dualnym (ang. *self-dual graph*).
— Grafy koła nie posiadają ani cyklu, ani ścieżki Eulera. Dla n nieparzystego tylko centralny wierzchołek (ang. *hub*) ma stopień parzysty, pozostałe wierzchołki (ang. *rim vertices*) mają stopień nieparzysty dla każdego n.
— Grafy koła są hamiltonowskie. Liczba różnych nieskierowanych cykli Hamiltona wynosi $n - 1$. Liczba różnych nieskierowanych ścieżek Hamiltona wynosi $2(n - 1)(n - 2)$ [38].
— Problem komiwojażera ma rozwiązanie wielomianowe w czasie $O(V)$. Wystarczy sprawdzić wszystkie cykle Hamiltona i wybrać ten o najmniejszej wadze, przy czym do sprawdzania cyklu wystarczą trzy krawędzie tworzące trójkąt zawierający huba.

5. Grafy konstruowane rekurencyjnie

W teorii grafów istnieją rodziny grafów konstruowanych rekurencyjnie (ang. *recursively constructed graphs*) [39], [40]. Autorzy podają: *Specific classes include trees, series-parallel graphs, k-terminal graphs, treewidth-k graphs, k-trees, partial k-trees, k-jackknife graphs, pathwidth-k graphs, bandwidth-k graphs, cutwidth-k graphs, branchwidth-k graphs, Halin graphs, cographs, cliquewidth-k graphs, k-NLC graphs, k-HB graphs, and rankwidth-k graphs.* Okazuje się, że wiele trudnych problemów posiada wydajne algorytmy (nawet działające w czasie liniowym), jeżeli zawęzimy instancje problemu do grafów konstruowanych rekurencyjnie. Algorytmy typowo bazują na programowaniu dynamicznym, czyli rozwiązanie większego problemu jest tworzone za pomocą rozwiązań dla mniejszych problemów. Pokażemy przykłady takich algorytmów dla drzew i grafów Halina.

Każda klasa grafów konstruowanych rekurencyjnie jest zdefiniowana za pomocą zbioru grafów bazowych (ang. *base graphs*), oraz zbioru reguł budujących (ang. *composition rules*). Każdy graf ma co najmniej jedno drzewo dekompozycji (ang. *decomposition tree*), które opisuje jak budować dany graf na bazie grafów bazowych.

W przypadku drzewa grafem bazowym jest pojedyńczy wierzchołek. Reguła budowy większego drzewa będzie opisana dalej.

W przypadku grafów Halina grafami bazowymi są grafy kołowe W_n. Reguła redukcji grafu Halina do grafu koła polega na ściąganiu do punktu wachlarzy, które łączą się z resztą grafu poprzez trzy krawędzie.

5.1. Drzewa z korzeniem konstruowane rekurencyjnie

Definicja rekurencyjna drzewa z korzeniem jest następująca.
(1) Pojedyńczy wierzchołek r jest drzewem (T, r) z korzeniem r, $V(T) = \{r\}$, $E(T) = \emptyset$.
(2) Dla dwóch drzew (T_1, r_1) i (T_2, r_2) tworzymy drzewo (T, r) w następujący sposób: $r = r_1$, $V(T) = V(T_1) \cup V(T_2)$, $E(T) = E(T_1) \cup E(T_2) \cup \{(r_1, r_2)\}$.

Algorytmy dla drzew wykorzystują własność optymalnej podstruktury, czyli optymalne rozwiązanie dla podproblemu jest używane do stworzenia optymalnego rozwiązania dla większych problemów. Stosowana jest technika programowania dynamicznego.

5.1.1. Kolorowanie wierzchołków drzewa

Drzewa są dwudzielne, więc wystarczy zastosować zwykły algorytm wykrywania grafów dwudzielnych, który koloruje wierzchołki grafu dwoma kolorami (wersja z BFS lub DFS). Złożoność czasowa algorytmu to $O(V)$.

Tabela 5.1. Tabliczka składania rozwiązań dla pokrycia wierzchołkowego drzewa [40].

$\oplus$	$T_2.a$	$T_2.b$
$T_1.a$	$T.a$	$T.a$
$T_1.b$	$T.b$	

5.1.2. Kolorowanie krawędzi drzewa

Optymalne kolorowanie krawędzi drzewa można wykonać algorytmem na bazie BFS (*connected sequential edge coloring*). Złożoność czasowa algorytmu to $O(V)$.

5.1.3. Pokrycie wierzchołkowe dla drzewa - składanie drzew

Naszym celem jest znalezienie najmniejszego pokrycia wierzchołkowego grafu T, czyli takiego podzbioru C zbioru wierzchołków $V(T)$, że każda krawędź z $E(T)$ ma przynajmniej jeden koniec w C. Pokrycie wierzchołkowe drzewa może, ale nie musi zawierać korzenia tego drzewa. Przyjmujemy oznaczenia [40]:
— $T.a$ to liczebność najmniejszego pokrycia wierzchołkowego zawierającego korzeń drzewa T,
— $T.b$ to liczebność najmniejszego pokrycia wierzchołkowego nie zawierającego korzenia drzewa T,
— $T.c$ to liczebność najmniejszego pokrycia wierzchołkowego drzewa T.
Składanie rozwiązań wygodnie jest przedstawić w tabeli 5.1.

Algorytm rekurencyjny wyznaczania pokrycia wierzchołkowego możemy zapisać następująco.
(1) Jeżeli $|V(T)| = 1$, to $T.a = 1$, $T.b = 0$.
(2) Jeżeli $T = T_1 \oplus T_2$, to
$T.a = \min\{T_1.a + T_2.a, T_1.a + T_2.b\}$,
$T.b = T_1.b + T_2.a$.
(3) $T.c = \min\{T.a, T.b\}$.

Implementacja algorytmu jest zawarta w module boriecover. Wykorzystano algorytm DFS do przechodzenia przez graf.

Listing 5.1. Moduł boriecover.

```
#!/usr/bin/python

class BorieNodeCover:
    """Find a minimum node cover cardinality for trees."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.parent = dict()   # dla DFS
        self.node_cover = set()
        self.cardinality = 0
        import sys
```

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

    def run(self, source=None):
        """Executable pseudocode."""
        # Struktura DFS wzbogacona obliczaniem liczebnosci pokrycia.
        if source is not None:
            # Badanie tylko jedne skladowej spojnej, jedno drzewo.
            self.parent[source] = None   # before _visit
            arg2 = self._visit(source)
            self.node_cover.update(min(arg2, key=len))
            self.cardinality = len(self.node_cover)
        else:
            # Tu jest las, moze byc wiele drzew rozlacznych.
            # Pokrycie bedzie suma pokryc rozlacznych drzew.
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self.parent[node] = None   # before _visit
                    arg2 = self._visit(node)
                    self.node_cover.update(min(arg2, key=len))
            self.cardinality = len(self.node_cover)

    def compose(self, arg1, arg2):
        """Compose results."""
        # a_set : minimalne pokrycie zawierajace roota
        # b_set : minimalne pokrycie nie zawierajace roota
        a1_set, b1_set = arg1
        a2_set, b2_set = arg2
        a_set = a1_set | min(arg2, key=len)   # wybieram mniejszy
        b_set = b1_set | a2_set
        return (a_set, b_set)

    def _visit(self, root):
        """Explore recursively the connected component."""
        # Zaczynamy od drzewa z jednym wierzcholkiem.
        arg1 = (set([root]), set())
        for target in self.graph.iteradjacent(root):
            if target not in self.parent:
                self.parent[target] = root   # before _visit
                arg2 = self._visit(target)
                arg1 = self.compose(arg1, arg2)
        return arg1
```

5.1.4. Pokrycie wierzchołkowe dla drzewa - odrywanie liści

Dla porównania podamy algorytm zachłanny, który znajduje optymalne pokrycie wierzchołkowe dla drzew w czasie liniowym $O(V)$. Jest to problem 35.1-4 w książce Cormena [1], a uzasadnienie podano w serwisie *CS Stack Exchange* [41]. W tym algorytmie również można dopatrzeć się struktury rekurencyjnej (drzewo po oderwaniu liści z rodzicami będzie drzewem lub lasem).

Rozważamy graf drzewo $T = (V, E)$, $|E| = |V| - 1$. Istnieje pokrycie optymalne C_1, które nie zawiera liści. Jeżeli znamy pokrycie optymalne C_2

z liśćmi, to można w C_2 zamienić liście z ich rodzicami, przez co dostaniemy pokrycie nie większe niż C_2 i możemy je przyjąć za C_1.

Przy wyznaczaniu pokrycia wybieramy zachłannie rodziców liści. Następnie usuwamy wszystkie pokryte krawędzie. Ponownie do pokrycia dodajemy wszystkich rodziców liści. Kontynuując procedurę otrzymujemy pokrycie optymalne dla drzewa. Podczas działania algorytmu drzewo może stać się niespójne. Aby zapewnić prawidłową kolejność przetwarzania liści wykorzystywana jest kolejka FIFO. Oryginalny graf nie jest modyfikowany, a efekt odrywania liści z rodzicami jest monitorowany przez słownik ze stopniami wierzchołków. Złożoność czasowa i pamięciowa algorytmu wynosi $O(V)$. Sprawdzono też inną wersję algorytmu, gdzie zamiast słownika degree_dict liście z rodzicami były faktycznie usuwane z kopii grafu. Testy pokazały, że wersja ze słownikiem jest prawie dwa razy szybsza. Implementacja algorytmu jest zawarta w module treecover. Algorytm działa poprawnie również dla lasu.

Listing 5.2. Moduł treecover.

```
#!/usr/bin/python

from Queue import Queue

class TreeNodeCoverSet:
    """Find a minimum node cover for trees."""

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

    def run(self):
        """Executable pseudocode."""
        # Slownik ze stopniami wierzcholkow, O(V) time.
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())
        Q = Queue()
        # Wstawiam liscie do kolejki, O(V) time.
        for node in self.graph.iternodes():
            if degree_dict[node] == 1:
                Q.put(node)
        while not Q.empty():    # kolejka na liscie
            source = Q.get()
            if degree_dict[source] == 0:    # wierzcholek izolowany
                continue
            for target in self.graph.iteradjacent(source):
                if degree_dict[target] > 0:    # to jest parent
                    self.node_cover.add(target)    # dodajemy rodzica
                    self.cardinality += 1
                    # Usuwam wszystkie krawedzie idace z target.
                    # Pomijam krawedzie usuniete wczesniej.
                    for node in self.graph.iteradjacent(target):
                        if degree_dict[node] > 0:
                            degree_dict[node] -= 1
```

Tabela 5.2. Tabliczka składania rozwiązań dla zbioru niezależnego drzewa [40].

$\oplus$	$T_2.a$	$T_2.b$
$T_1.a$		$T.a$
$T_1.b$	$T.b$	$T.b$

```
                degree_dict[target] -= 1
                if degree_dict[node] == 1:    # nowy lisc
                    Q.put(node)
            break    # parent przetworzony
```

5.1.5. Zbiór niezależny dla drzewa - składanie drzew

Naszym celem jest znalezienie największego zbioru niezależnego grafu T, czyli takiego podzbioru S zbioru wierzchołków $V(T)$, że każda krawędź z $E(T)$ ma najwyżej jeden koniec w S. Zbiór niezależny może, ale nie musi zawierać korzenia tego drzewa. Przyjmujemy oznaczenia [40]:

— $T.a$ to liczebność największego zbioru niezależnego zawierającego korzeń drzewa T,

— $T.b$ to liczebność największego zbioru niezależnego nie zawierającego korzenia drzewa T,

— $T.c$ to liczebność największego zbioru niezależnego drzewa T.

Składanie rozwiązań wygodnie jest przedstawić w tabeli 5.2.

Algorytm rekurencyjny wyznaczania zbioru niezależnego możemy zapisać następująco.

(1) Jeżeli $|V(T)| = 1$, to $T.a = 1$, $T.b = 0$.

(2) Jeżeli $T = T_1 \oplus T_2$, to

$T.a = T_1.a + T_2.b$,

$T.b = \max\{T_1.b + T_2.a, T_1.b + T_2.b\}$.

(3) $T.c = \max\{T.a, T.b\}$.

Implementacja algorytmu jest zawarta w module borieiset. Wykorzystano algorytm DFS do przechodzenia przez graf.

Listing 5.3. Moduł borieiset.

```
#!/usr/bin/python

class BorieIndependentSet:
    """Find a maximum independent set for trees."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.parent = dict()    # dla DFS
        self.independent_set = set()
        self.cardinality = 0
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))
```

```python
    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:
            # Badanie tylko jedne skladowej spojnej, jedno drzewo.
            self.parent[source] = None   # before _visit
            arg2 = self._visit(source)
            self.independent_set.update(max(arg2, key=len))
            self.cardinality = len(self.independent_set)
        else:
            # Tu jest las, moze byc wiele drzew rozlacznych.
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self.parent[node] = None   # before _visit
                    arg2 = self._visit(node)
                    self.independent_set.update(max(arg2, key=len))
            self.cardinality = len(self.independent_set)

    def compose(self, arg1, arg2):
        """Compose results."""
        # a_set : max iset zawierajacy roota
        # b_set : max iset nie zawierajacy roota
        a1_set, b1_set = arg1
        a2_set, b2_set = arg2
        a_set = a1_set | b2_set
        b_set = b1_set | max(arg2, key=len)   # wybieram wiekszy
        return (a_set, b_set)

    def _visit(self, root):   # tak jak dla node cover
        """Explore recursively the connected component."""
        # Zaczynamy od drzewa z jednym wierzcholkiem.
        arg1 = (set([root]), set())
        for target in self.graph.iteradjacent(root):
            if target not in self.parent:
                self.parent[target] = root   # before _visit
                arg2 = self._visit(target)
                arg1 = self.compose(arg1, arg2)
        return arg1
```

5.1.6. Zbiór niezależny dla drzewa - odrywanie liści

Podamy algorytm zachłanny, który znajduje największy zbiór niezależny dla drzew w czasie liniowym $O(V)$. Rozumowanie jest podobne do przypadku pokrycia wierzchołkowego.

Rozważamy graf drzewo $T = (V, E)$, $|E| = |V| - 1$. Istnieje optymalny zbiór niezależny S_1, który zawiera wszystkie liście (wyjątkiem jest pojedyńcza krawędź). Jeżeli znamy optymalny zbiór niezależny S_2 bez pewnych liści, to można w S_2 zamienić rodziców na liście, przez co dostaniemy zbiór niezależny nie mniejszy niż S_2 i możemy go przyjąć za S_1.

Przy wyznaczaniu zbioru niezależnego wybieramy zachłannie wszystkie liście. Następnie usuwamy z grafu wszystkie liście z rodzicami. Ponownie do zbioru niezależnego dodajemy wszystkie nowe liście. Kontynuując procedurę otrzymujemy optymalny zbiór niezależny dla drzewa. Podczas działania algorytmu drzewo może stać się niespójne. Aby zapewnić prawidłową kolejność przetwarzania liści wykorzystywana jest kolejka FIFO. W algorytmie korzystamy z pomocniczego zbioru _used, do którego wstawiamy elementy powstającego zbioru niezależnego i ich sąsiadów. Dzięki temu prawidłowo obsługiwany jest przypadek pojedyńczej krawędzi, której oba końce to liście, ale tylko jeden z nich może znaleźć się w zbiorze niezależnym. Implementacja algorytmu jest zawarta w module treeiset. Algorytm działa poprawnie dla lasu, przy czym wierzchołki izolowane należy wstawić od razu do zbioru niezależnego.

Listing 5.4. Moduł treeiset.

```
#!/usr/bin/python

from Queue import Queue

class TreeIndependentSet:
    """Find a maximum independent set for trees."""

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

    def run(self):
        """Executable pseudocode."""
        # Slownik ze stopniami wierzcholkow, O(V) time.
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())
        Q = Queue()
        # Wstawiam liscie do kolejki, O(V) time.
        for node in self.graph.iternodes():
            if degree_dict[node] == 0:
                # Wierzcholek izolowany od samego poczatku.
                self.independent_set.add(node)
                self._used.add(node)
                self.cardinality += 1
            elif degree_dict[node] == 1:   # lisc do kolejki
                Q.put(node)
        while not Q.empty():   # kolejka na liscie
            source = Q.get()
            # Lisc moze stac sie wierzcholkiem izolowanym.
            if degree_dict[source] == 0 and source not in self._used:
                self.independent_set.add(source)
                self._used.add(source)
                self.cardinality += 1
                continue
            for target in self.graph.iteradjacent(source):
                if degree_dict[target] > 0:   # to jest parent
                    self.independent_set.add(source)   # dodaje liscia
                    self._used.add(source)
                    self._used.add(target)
                    self.cardinality += 1
                    # Usuwam wszystkie krawedzie idace z target.
```

Tabela 5.3. Tabliczka składania rozwiązań dla zbioru dominującego drzewa [40].

$\oplus$	$T_2.a$	$T_2.b$	$T_2.c$
$T_1.a$	$T.a$	$T.a$	$T.a$
$T_1.b$	$T.b$	$T.b$	
$T_1.c$	$T.b$	$T.c$	

```
        # Pomijam krawedzie usuniete wczesniej.
        for node in self.graph.iteradjacent(target):
            if degree_dict[node] > 0:
                degree_dict[node] -= 1
                degree_dict[target] -= 1
                if degree_dict[node] == 1:     # nowy lisc
                    Q.put(node)
        break    # parent przetworzony
```

5.1.7. Zbiór dominujący dla drzewa - składanie drzew

Mamy znaleźć najmniejszy zbiór dominujący grafu T, czyli taki podzbiór D zbioru wierzchołków $V(T)$, że każdy wierzchołek nie należący do D ma co najmniej jednego sąsiada w D [42]. Zbiór dominujący może, ale nie musi zawierać korzenia tego drzewa. Przyjmujemy oznaczenia [40]:

— $T.a$ to liczebność najmniejszego zbioru dominującego zawierającego korzeń drzewa T,

— $T.b$ to liczebność najmniejszego zbioru dominującego nie zawierającego korzenia drzewa T,

— $T.c$ to liczebność najmniejszego zbioru prawie dominującego, gdzie tylko korzeń drzewa T nie jest zdominowany.

— $T.d$ to liczebność najmniejszego zbioru dominującego drzewa T.

Składanie rozwiązań wygodnie jest przedstawić w tabeli 5.3.

Algorytm rekurencyjny wyznaczania zbioru dominującego możemy zapisać następująco.

(1) Jeżeli $|V(T)| = 1$, to $T.a = 1$, $T.b = \infty$, $T.c = 0$.

(2) Jeżeli $T = T_1 \oplus T_2$, to

$T.a = T_1.a + \min\{T_2.a, T_2.b, T_2.c\}$,

$T.b = \min\{T_1.b + T_2.a, T_1.b + T_2.b, T_1.c + T_2.a\}$,

$T.a = T_1.c + T_2.b$.

(3) $T.d = \min\{T.a, T.b\}$.

Implementacja algorytmu jest zawarta w module boriedset. Wykorzystano algorytm DFS do przechodzenia przez graf.

Listing 5.5. Moduł boriedset.

```
#!/usr/bin/python

class BorieDominatingSet:
    """Find a minimum cardinality dominating set for trees."""

    def __init__(self, graph):
        """The algorithm initialization."""
```

```
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.parent = dict()   # dla DFS
        self.dominating_set = set()
        self.cardinality = 0
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:
            # Badanie tylko jednej skladowej spojnej, jedno drzewo.
            self.parent[source] = None   # before _visit
            a2_set, b2_set, c2_set = self._visit(source)
            self.dominating_set.update(min([a2_set, b2_set], key=len))
            self.cardinality = len(self.dominating_set)
        else:
            # Tu jest las, moze byc wiele drzew rozlacznych.
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self.parent[node] = None   # before _visit
                    a2_set, b2_set, c2_set = self._visit(node)
                    self.dominating_set.update(
                        min([a2_set, b2_set], key=len))
            self.cardinality = len(self.dominating_set)

    def compose(self, arg1, arg2):
        """Compose results."""
        # a_set : dominating_set zawierajacy roota
        # b_set : dominating_set nie zawierajacy roota
        # c_set : root undominated (niepelne rozwiazanie!)
        a1_set, b1_set, c1_set = arg1
        a2_set, b2_set, c2_set = arg2
        a_set = a1_set | min(arg2, key=len)
        b_set = min([b1_set | a2_set, b1_set | b2_set,
            c1_set | a2_set], key=len)
        c_set = c1_set | b2_set
        return (a_set, b_set, c_set)

    def _visit(self, root):
        """Explore recursively the connected component."""
        # Zaczynamy od drzewa z jednym wierzcholkiem.
        arg1 = (set([root]), set([root]), set())
        for target in self.graph.iteradjacent(root):
            if target not in self.parent:
                self.parent[target] = root   # before _visit
                arg2 = self._visit(target)
                arg1 = self.compose(arg1, arg2)
        return arg1
```

5.1.8. Zbiór dominujący dla drzewa - odrywanie liści

Podamy algorytm zachłanny, który znajduje najmniejszy zbiór dominujący dla drzew w czasie liniowym $O(V)$. Rozumowanie jest podobne do przypadku pokrycia wierzchołkowego.

Rozważamy graf drzewo $T = (V, E)$, $|E| = |V| - 1$. Istnieje optymalny zbiór dominujący S_1, który nie zawiera liści (wyjątkiem jest pojedyńcza krawędź). Jeżeli znamy optymalny zbiór dominujący S_2 z pewnymi liśćmi, to można w S_2 zamienić liście na rodziców, przez co dostaniemy zbiór dominujący nie mniejszy niż S_2 i możemy go przyjąć za S_1.

Przy wyznaczaniu zbioru dominującego wybieramy zachłannie rodziców liści. Następnie usuwamy z grafu wszystkich rodziców liści i inne wierzchołki sąsiadujące z rodzicami. Dalej ponownie do zbioru dominującego dodajemy rodziców nowych liści, itd. Podczas działania algorytmu drzewo może stać się niespójne. Aby zapewnić prawidłową kolejność przetwarzania liści wykorzystywana jest kolejka FIFO.

Przy wstawianiu każdego wierzchołka do zbioru dominującego należy zaznaczyć wszystkie wierzchołki sąsiednie jako pokryte. Dzięki temu prawidłowo obsłużymy wierzchołki izolowane pokryte (pominięcie) i niepokryte (zaliczenie do zbioru dominującego). Ponadto prawidłowo obsłużymy przypadek pojedyńczej krawędzi, oraz liści pokrytych (pominiecie) i niepokrytych (zaliczenie rodzica do zbioru dominującego). Algorytm działa prawidłowo również dla lasu. Implementacja algorytmu jest zawarta w module treedset.

Listing 5.6. Moduł treedset.

```
#!/usr/bin/python

from Queue import Queue

class TreeDominatingSet:
    """Find a maximum dominating set for trees."""

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

    def run(self):
        """Executable pseudocode."""
        # Slownik ze stopniami wierzcholkow, O(V) time.
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())
        Q = Queue()
        # Wstawiam liscie do kolejki, O(V) time.
        for node in self.graph.iternodes():
            if degree_dict[node] == 0:
                # Wierzcholek izolowany od samego poczatku.
                self.dominating_set.add(node)
                self._used.add(node)
```

```
                self.cardinality += 1
            elif degree_dict[node] == 1:    # lisc do kolejki
                Q.put(node)
        while not Q.empty():    # kolejka na liscie
            source = Q.get()
            # Lisc moze stac sie wierzcholkiem izolowanym.
            if degree_dict[source] == 0:
                if source in self._used:    # parent w dset
                    pass
                else:   # parent nie jest w dset
                    self.dominating_set.add(source)
                    self._used.add(source)
                    self.cardinality += 1
            elif degree_dict[source] == 1:
                if source in self._used:
                    # Usuwam takiego liscia.
                    for node in self.graph.iteradjacent(source):
                        if degree_dict[node] > 0:    # to jest parent
                            degree_dict[node] -= 1
                            degree_dict[source] -= 1
                            if degree_dict[node] == 1:    # nowy lisc
                                Q.put(node)
                            break    # parent przetworzony
                else:   # source not in used. parent idzie do dset.
                    for target in self.graph.iteradjacent(source):
                        if degree_dict[target] > 0:    # to jest parent
                            self.dominating_set.add(target)
                            # Dodawanie dzieci do used bedzie potem.
                            self._used.add(target)
                            self.cardinality += 1
                            # Usuwam wszystkie krawedzie idace z target.
                            # Pomijam krawedzie usuniete wczesniej.
                            for node in self.graph.iteradjacent(target):
                                if degree_dict[node] > 0:
                                    degree_dict[node] -= 1
                                    degree_dict[target] -= 1
                                    self._used.add(node) # dzieci do used
                                    if degree_dict[node] == 1: # nowy lisc
                                        Q.put(node)
                            break    # parent przetworzony
            else:
                raise ValueError("degree_dict[source] > 1")
```

5.2. Grafy Halina konstruowane rekurencyjnie

Dzięki rekurencyjnej strukturze grafów Halina można znaleźć wielomianowe rozwiązania wielu trudnych problemów z teorii grafów.

5.2.1. Kolorowanie wierzchołków grafu Halina

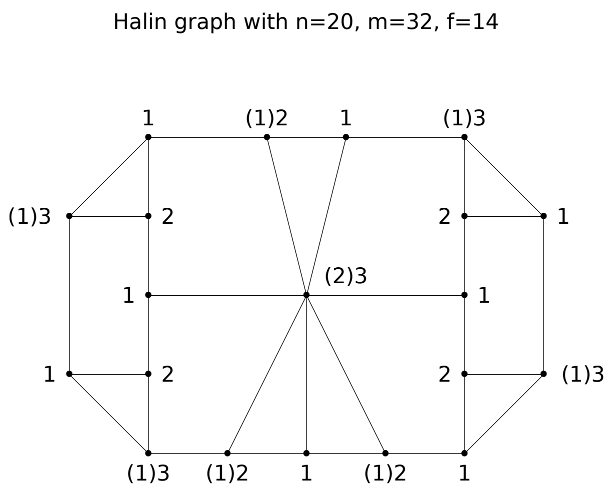
Wszystkie grafy Halina są planarne, więc są 4-kolorowalne wierzchołkowo. Cztery kolory są niezbędne do pokolorowania grafów kołowych W_n z parzystą liczbą wierzchołków n. Używając procedury redukcji (zwykłej lub Eppsteina) możemy pokolorować każdy graf Halina czterema kolorami, ale nie zawsze będzie to optymalna liczba kolorów.

Okazuje się, że jest wiele grafów Halina, dla których liczba chromatyczna wynosi $\chi(H) = 3$. Podamy przykłady takich grafów.

— Grafy kołowe W_n z nieparzystą liczbą wierzchołków n. Hub ma kolor c_1, a wierzchołki cyklu C mają na przemian kolory c_2 i c_3.
— Kubiczne grafy Halina można pokolorować trzema kolorami algorytmem z twierdzenia Brooksa.
— Unicentryczne grafy Halina $H_1(k, D)$ są 3-kolorowalne wierzchołkowo dla $k > 1$ [43]. Dla $k = 1$ dostajemy grafy kołowe $H_1(1, D) = W_{D+1}$, a wtedy D musi być parzyste. Parametr D jest stopniem wierzchołków drzewa T, które nie są liśćmi, a parametr k oznacza liczbę poziomów rekurencyjnego rozwinięcia grafu. Dla $k \geqslant 1$ liczba wierzchołków grafu unicentrycznego wynosi
$1 + D + D(D-1) + D(D-1)^2 + \ldots + D(D-1)^{k-1}$.
— Bicentryczne grafy Halina $H_2(k, D)$ są 3-kolorowalne wierzchołkowo [43]. Dla $k \geqslant 1$ liczba wierzchołków grafu bicentrycznego wynosi
$2[1 + (D-1) + (D-1)^2 + \ldots + (D-1)^k]$.
— Procedura redukcji bazuje na ściąganiu do punktu wachlarzy, które są połączone z resztą grafu trzema krawędziami. Jeżeli końce tych trzech krawędzi od strony reszty grafu mają dwa lub trzy różne kolory, to wachlarz można pokolorować trzema dostępnymi kolorami. Jeżeli jednak końce tych krawędzi są jednego koloru, to wachlarz o środku w wierzchołku v i parzystą liczbą wierzchołków na brzegu $C(v)$ musi być pokolorowany z użyciem czwartego koloru. Nie jest jasne, czy można uniknąć takiej sytuacji.

W pracy [44] opisano algorytm równoległy kolorowania wierzchołków grafu Halina. Wydaje się, że opis algorytmu nie jest kompletny. Na tej bazie stworzyliśmy algorytm szeregowy, który koloruje wierzchołki wszystkich grafów Halina (oprócz W_n z n parzystym) trzema kolorami. Takiego opisu nie udało się nam znaleźć w literaturze.
(1) Dla danego grafu Halina $H = T \cup C$ kolorujemy jego drzewo T dwoma kolorami c_1 i c_2. Konflikty kolorów mogą teraz pojawić się na cyklu C.
(2) Jeżeli cykl C jest parzysty, to co drugi wierzchołek cyklu kolorujemy kolorem c_3, co daje sekwencję kolorów na cyklu $[(c_1|c_2)c_3]$. Nawiasy kwadratowe oznaczają powtarzanie sekwencji, a w nawiasach okrągłych mamy dwa możliwe kolory występujące na cyklu.
(2) Jeżeli cykl C jest nieparzysty, ale występują w nim kolory c_1 i c_2, to prawidłowe kolorowanie daje sekwencja $c_1c_2c_3[(c_1|c_2)c_3]$.
(3) Jeżeli cykl C jest nieparzysty, występuje w nim tylko kolor c_1, ale istnieje wachlarz o środku w v i nieparzystej liczbie wierzchołków na brzegu $C(v)$, to istniejącą sekwencję można zapisać jako $c_1[c_1c_1][c_1c_1]$. Wierzchołek v ma kolor c_2, pierwsze trzy znaki sekwencji $c_1[c_1c_1]$ odpowiadają brzegowi wachlarza $C(v)$. W tej sytuacji wierzchołek v dostaje kolor c_3, a prawidłowa sekwencja kolorów na cyklu C ma postać $c_2[c_1c_2][c_1c_3]$.
(4) Jeżeli cykl C jest nieparzysty, występuje w nim tylko kolor c_1, nie ma wacharza nieparzystego, to istnieje nieparzysta liczba kolejnych wierzchołków cyklu C, które mają wspólnego sąsiada v (kolor c_2) z drzewa T. Nie jest

to wachlarz, bo ten zbiór wierzchołków jest połączony z resztą grafu większą liczbą krawędzi niż trzy. Istniejąca sekwencja kolorów to $c_1[c_1c_1][c_1c_1]$. W tej sytuacji wierzchołek v dostaje kolor c_3, a prawidłowa sekwencja kolorów na cyklu C ma postać $c_2[c_1c_2][c_1(c_3|c_2)]c_1c_3$. Fragment sekwencji postaci $[c_1(c_3|c_2)]$ opisuje możliwość, że wierzchołek v z nowym kolorem c_3 może mieć sąsiadów leżących w kilku miejscach na cyklu C. Taką sytuację przedstawia rysunek 5.2.1.



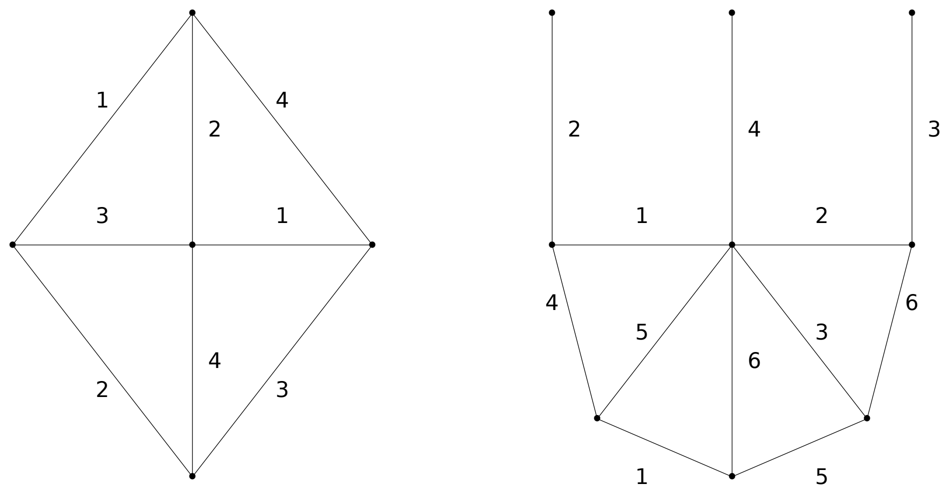
Rysunek 5.1. Kolorowanie wierzchołków dla grafów Halina. Rysunek przedstawia graf z nieparzystym cyklem C, który ma wierzchołki jednego koloru na etapie pokolorowania drzewa T. Graf nie ma wachlarza nieparzystego. W nawiasach znajdują się kolory wynikające z pokolorowania samego drzewa T, które musiały zostać zmienione, aby uniknąć konfliktów.

5.2.2. Kolorowanie krawędzi grafu Halina

Grafy Halina są klasy 1, czyli wystarczy $\Delta(H)$ kolorów do poprawnego pokolorowania krawędzi [26]. Rysunek 5.2.2 pokazuje zasadę kolorowania krawędzi dla grafu koła, w kierunku przeciwnym do ruchu wskazówek zegara. Obok pokazano kolorowanie krawędzi wachlarza, przy wykorzystaniu kolorów już zdeterminowanych.

5.2.3. Cykle Hamiltona dla grafu Halina

Dla kubicznych grafów Halina istnieją dokładnie trzy cykle Hamiltona, które można znaleźć przy pomocy redukcji. Zauważmy, że dla kubicznych grafów Halina wszystkie wachlarze mają na brzegu tylko dwa punkty. Proces redukcji kończy się wraz z uzyskaniem grafu koła $W_4 = K_4$, dla którego mamy trzy różne cykle Hamiltona. Odwracając proces redukcji stwierdzamy, że możemy rozszerzyć dany cykl Hamiltona na nowe wierzchołki z wachlarza tylko w jeden sposób. Stąd na końcu otrzymamy tylko trzy różne cykle Hamiltona.



Rysunek 5.2. Kolorowanie krawędzi dla grafów Halina. Trzy krawędzie dochodzące do wachlarza mają już ustalone kolory, pochodzące z dolnego wierzchołka lewego grafu.

Dla dowolnych grafów Halina zwykle jest więcej możliwych cykli Hamiltona, ale za pomocą procedury redukcji można wyznaczyć jeden z nich.

5.2.4. Problem komiwojażera dla grafu Halina

Rozwiązanie problemu podał Cornuejols ze współpracownikami [32], [33]. Warto zwrócić uwagę na fakt, że problem komiwojażera nie staje się łatwy z powodu małej liczby możliwych cykli Hamiltona, tylko z powodu istnienia dekompozycji. Cornuejols podaje przykład grafów Halina H^k, w których liczba możliwych cykli Hamiltona rośnie wykładniczo z liczbą wierzchołków i wynosi $k \cdot 2^{k-2}$, przy czym liczba wierzchołków wynosi $4k+1$ [32].

Opiszemy pokrótce rozwiązanie problemu komiwojażera dla grafów Halina. Jeżeli graf jest kołem W_n, to wystarczy sprawdzić wszystkie $n-1$ cykli Hamiltona. Jeżeli graf H nie jest kołem, to ma wachlarze. Wykonujemy sukcesywne ściąganie wachlarzy do punktu, przy czym w odpowiedni sposób ustalamy wagi nowych krawędzi.

Rozważmy wachlarz o środku w x i brzegu $C(x) = \{u_1, \ldots, u_r\}$. Cięcie zawiera krawędzie (v_1, u_1), (v_2, x), (v_3, u_r) (rysunek 5.2.4). Po ściągnięciu wachlarza do punktu y powstaną nowe krawędzie (v_1, y), (v_2, y), (v_3, y).

Niech K będzie sumą wag krawędzi tworzących brzeg wachlarza, a $K(u, v)$ wkładem wag z wachlarza do cyklu Hamiltona wchodzącego do wachlarza z wierzchołka u i wychodzącego z wachlarza do wierzchołka v.

$$K = w(u_1, u_2) + w(u_2, u_3) + \ldots + w(u_{r-1}, u_r), \tag{5.1}$$

$$K(v_1, v_2) = K + w(x, u_r), \tag{5.2}$$

$$K(v_2, v_3) = K + w(x, u_1), \tag{5.3}$$

$$K(v_1, v_3) = K + \min\{w(u_i, x) + w(x, u_{i+1}) - w(u_i, u_{i+1})\}, \tag{5.4}$$

$$w(v_1, y) = w(v_1, u_1) + \frac{1}{2}[K(v_1, v_2) + K(v_1, v_3) - K(v_2, v_3)], \tag{5.5}$$

$$w(v_2, y) = w(v_2, x) + \frac{1}{2}[K(v_1, v_2) + K(v_2, v_3) - K(v_1, v_3)], \tag{5.6}$$

$$w(v_3, y) = w(v_3, u_r) + \frac{1}{2}[K(v_1, v_3) + K(v_2, v_3) - K(v_1, v_2)]. \tag{5.7}$$

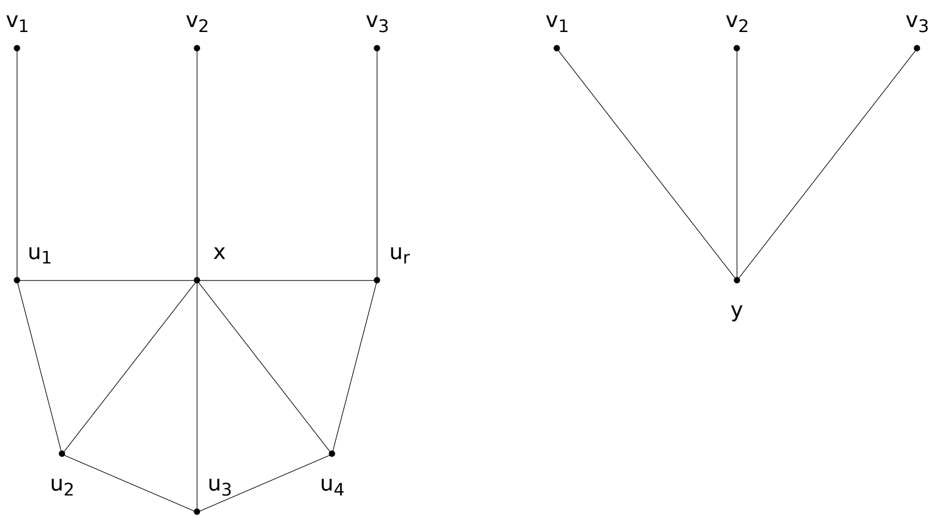
Wagi nowych krawędzi zostały tak dobrane, aby

$$w(v_1, y) + w(v_2, y) = w(v_1, u_1) + w(v_2, x) + K(v_1, v_2), \tag{5.8}$$

$$w(v_1, y) + w(v_3, y) = w(v_1, u_1) + w(v_3, u_r) + K(v_1, v_3), \tag{5.9}$$

$$w(v_2, y) + w(v_3, y) = w(v_2, x) + w(v_3, u_r) + K(v_2, v_3). \tag{5.10}$$

Sukcesywna redukcja doprowadzi do grafu koła, dla którego znamy rozwiązanie. Następnie należy odwrócić operacje redukcji, aby zrekonstruować optymalny cykl Hamiltona w pierwotnym grafie H. Odpowiednie struktury danych pozwalają wykonać całą pracę w czasie $O(V)$.



Rysunek 5.3. Problem komiwojażera dla grafów Halina. Wachlarz o środku w x zostaje ściągnięty do punktu y ($u_r = u_5$).

5.2.5. Pokrycie wierzchołkowe dla grafu Halina

Istnieje algorytm wielomianowy, prawdopodobnie związany z dekompozycją drzewową grafu Halina.

5.2.6. Zbiór niezależny dla grafu Halina

Problem może być rozwiązany w czasie liniowym za pomocą programowania dynamicznego [34].

5.2.7. Zbiór dominujący dla grafu Halina

Istnieje algorytm wielomianowy, prawdopodobnie związany z dekompozycją drzewową grafu Halina.

6. Algorytmy rozpoznawania grafów

W tym rozdziale zostaną przedstawione algorytmy rozpoznawania grafów kołowych i ogólnych grafów Halina.

6.1. Algorytm rozpoznawania grafów kołowych

Dane wejściowe: Dowolny graf prosty.

Problem: Rozpoznawanie grafu kołowego.

Opis algorytmu: Algorytm najpierw sprawdza możliwość wystąpienia grafu pełnego K_4. Następnie sprawdzany jest warunek liczby krawędzi $|E| = 2|V| - 2$ i dozwolone stopnie wierzchołków (3 i $|V| - 1$). W końcu usuwany jest wierzchołek centralny stopnia $|V| - 1$, co powinno prowadzić do cyklu z $|V| - 1$ wierzchołkami (test z BFS). W czasie pracy algorytmu oryginalny graf jest modyfikowany, ale wszystkie zmiany są na końcu wycofywane.

Złożoność: Złożoność czasowa algorytmu dla dowolnego grafu szacowana jest na $O(V+E)$ [wyznaczanie liczby krawędzi może zająć czas $O(E)$], a dla grafu kołowego lub rzadkiego tylko $O(V)$.

Uwagi: Dla wygody użytkowników dodano funkcję is_wheel, która wykonuje test i nie udostępnia innych danych o grafie.

Listing 6.1. Moduł wheels.

```
#!/usr/bin/python

from bfs import SimpleBFS

class WheelGraph:
    """Wheel graphs detection."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.hub = None

    def run(self):
        """Executable pseudocode."""
        n = self.graph.v()
        e = self.graph.e()
```

```
        if n < 4:
            raise ValueError("the number of nodes is less then 4")
        elif n == 4 and e == 6:    # graf K4, wszystkie stopnie 3
            self.hub = self.graph.iternodes().next()
            return
        if e != 2 * n - 2:
            raise ValueError("bad number of edges")
        dset = set(self.graph.degree(node)
            for node in self.graph.iternodes())
        if dset != set([3, n-1]):
            raise ValueError("bad set of node degrees")
        self.hub = max(self.graph.iternodes(), key=self.graph.degree)
        # Po usunieciu huba z krawedziami powinien zostac cykl.
        removed = list(self.graph.iteroutedges(self.hub))
        for edge in removed:
            self.graph.del_edge(edge)
        # Wybieram wierzcholek rozny od huba.
        for node in self.graph.iternodes():
            if node != self.hub:
                source = node
                break
        order = []
        algorithm = SimpleBFS(self.graph)
        algorithm.run(source, pre_action=lambda node: order.append(node))
        for edge in removed:    # naprawiamy graf
            self.graph.add_edge(edge)
        if len(order) != n-1:
            raise ValueError("not a wheel graph")


def is_wheel(graph):
    """Test if a graph is a wheel graph."""
    try:
        algorithm = WheelGraph(graph)
        algorithm.run()
        return True
    except ValueError:
        return False
```

6.2. Algorytm rozpoznawania grafów Halina

Dane wejściowe: Dowolny graf prosty.

Problem: Rozpoznawanie grafu Halina $H = T \cup C$.

Opis algorytmu: Algorytm opiera się na dekompozycji Eppsteina i wyznacza zbiór wierzchołków należący do cyklu C.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V)$, choć analiza kodu źródłowego nie jest łatwa.

Uwagi: Nasza implementacja bazuje na kodzie biblioteki PADS Dawida Eppsteina [45], [46]. Nasze testy wykryły, że oryginalny kod Eppsteina nie uwzględnia niejednoznaczności w wyborze wierzchołka wewnętrznego, na etapie grafu K_4. W wyniku korespondencji z Eppsteinem, do biblioteki PADS zostały wprowadzone pewne zmiany. Wydaje się, że po tych poprawkach niejednoznaczność występuje tylko wtedy, kiedy w danym grafie Halina więcej niż jedna ściana może być ścianą zewnętrzną. Wobec tego wybór wierzchołka wewnętrznego z kilku możliwości oznacza wybór konkretnej ściany zewnętrznej i nie jest to oznaka błędu w samej procedurze dekompozycji.

Listing 6.2. Moduł halin.

```
#!/usr/bin/python

from edges import Edge

class HalinGraph:
    """Halin graphs detection."""
    def __init__(self, graph):
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self._graph_copy = self.graph.copy()
        self.outer = set()  # nodes from the outer face
        self.degree3 = set(node for node in self.graph.iternodes()
            if self.graph.degree(node) == 3)  # active nodes with degree 3
        self._calls = []

    def run(self):
        """Executable pseudocode: check if a graph is halin."""
        while self.degree3:
            node = self.degree3.pop()
            if (self._graph_copy.has_node(node) and
                self._graph_copy.v() > 4 and
                self._graph_copy.degree(node) == 3):
                    # Decode set of neighbors.
                    a, b, c = tuple(self._graph_copy.iteradjacent(node))
                    self._reduce(a, node, b)
                    self._reduce(a, node, c)
                    self._reduce(b, node, c)
        if not self.is_outer_k4():
            raise ValueError("not a Halin graph")
        self._reconstruct_cycle()

    def other_neighbor(self, a, b, c):
        """Return a neighbor of b_vertex which is not included in the triangle."""
        neighbors = list(node for node in self._graph_copy.iteradjacent(b)
            if node != a and node != c)
        return neighbors.pop()

    def _reduce(self, a, b, c):
        """Try reduce for vertex y in the middle."""
        for node in (a, b, c):
            if (not self._graph_copy.has_node(node) or
                self._graph_copy.degree(node) != 3):
                    return
        if self._graph_copy.has_edge(Edge(a, c)):
            self._reduce_triangle(a, b, c)
        else:
```

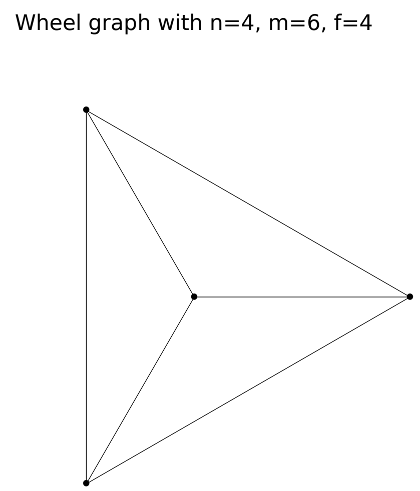
```
            self._reduce_path(a, b, c)

    def _reduce_triangle(self, a, b, c):
        """Collapse degree-three vertcies from a triangle
           with three distinct neighbors to a single vertex."""
        Na = self.other_neighbor(b, a, c)
        Nb = self.other_neighbor(a, b, c)
        Nc = self.other_neighbor(a, c, b)
        if Na == Nb or Nb == Nc or Na == Nc:
            return   # need to have three distinct neighbors
        # BEGIN HALIN
        if a in self.outer and b in self.outer and c in self.outer:
            return   # can't collapse when all three are outer
        # If node belongs to the outer face, mark its neighbor as outer.
        for node, neighbor in [(a, Na), (b, Nb), (c, Nc)]:
            if node in self.outer:
                self.outer.add(neighbor)
        # Add also the collapsed node to the outer set.
        new_node = "{0}_{1}_{2}".format(a, b, c)
        self.outer.add(new_node)   # Eppstein ma tu supervertex
        self._calls.append(("triangle", a, b, c, Na, Nb, Nc, new_node))
        # END HALIN
        # Make the change
        self._graph_copy.del_node(a)
        self._graph_copy.del_node(b)
        self._graph_copy.del_node(c)
        self._graph_copy.add_edge(Edge(Na, new_node))
        self._graph_copy.add_edge(Edge(Nb, new_node))
        self._graph_copy.add_edge(Edge(Nc, new_node))
        # Update the active nodes.
        for node in (new_node, Na, Nb, Nc):
            if self._graph_copy.degree(node) == 3:
                self.degree3.add(node)

    def _reduce_path(self, a, b, c):
        """Degree-three vertices from a path with one shared neighbor
           contracted to a two-vertex path."""
        neighbors = (set(self._graph_copy.iteradjacent(a)) &
                     set(self._graph_copy.iteradjacent(b)) &
                     set(self._graph_copy.iteradjacent(c)))
        if len(neighbors) != 1:
            return
        neighbor = neighbors.pop()
        # BEGIN HALIN
        if neighbor in self.outer:
            return  # as we can't shorten path with outer apex
        if self._graph_copy.v() == 5:
            self.outer.update(node for node in self._graph_copy.iternodes()
                if node != neighbor)
        else:
            self.outer.update((a, c))  # mark remaining nodes as outer
        self._calls.append(("path", a, b, c, neighbor))
        # END HALIN
        # Make the change!
        self._graph_copy.del_node(b)  # remove b with edges
        self._graph_copy.add_edge(Edge(a, c))
        # Update the active nodes.
```

```python
        if self._graph_copy.degree(neighbor) == 3:
            self.degree3.add(neighbor)

    def _reconstruct_cycle(self):
        """Recursively reconstruct the leaf cycle of Halin graph."""
        self._calls.reverse()
        for node in self._graph_copy.iternodes():
            if node not in self.outer:
                for source in self._graph_copy.iternodes():
                    if node != source:
                        self.outer.add(source)
                break
        for items in self._calls:
            name, arguments = items[0], items[1:]
            if name == "triangle":
                self._undo_triangle(*arguments)
            elif name == "path":
                self._undo_path(*arguments)
        # Remove nodes from reduction.
        self.outer = self.outer & set(self.graph.iternodes())

    def _undo_triangle(self, a, b, c, Na, Nb, Nc, x):
        """Undo a triangle reduction."""
        nout = 0
        for node, neighbor in [(a, Na), (b, Nb), (c, Nc)]:
            if neighbor in self.outer:
                self.outer.add(node)
                nout += 1
        if nout != 2:
            raise ValueError("problem with triangle")

    def _undo_path(self, a, b, c, neighbor):
        """Undo a path reduction."""
        if neighbor in self.outer:
            raise ValueError("neighbor in outer")
        self.outer.update((a, b, c))

    def is_k4(self):
        """Check if the graph is K4."""
        if self._graph_copy.v() != 4:
            return False
        return all(self._graph_copy.degree(node) == 3
            for node in self._graph_copy.iternodes())

    def is_outer_k4(self):
        """Have we reduced to a K4 with a non-outer node?"""
        if not self.is_k4():
            return False
        return any(node not in self.outer
            for node in self._graph_copy.iternodes())
```

7. Galeria grafów Halina

W tym rozdziale pokażemy rysunki najprostszych grafów Halina. Wykorzystamy moduł Gnuplot i szkielet skryptów do rysowania grafów z pracy Sandry Pażyniowskiej [47]. Liczby grafów Halina o danej liczbie wierzchołków zostały zebrane w tabeli 7.1. Dwa grafy są uważane za równoważne, jeżeli można jeden graf przekształcić w drugi przy pomocy obrotu lub odbicia, z dokładnością do długości krawędzi, czy kątów między krawędziami.

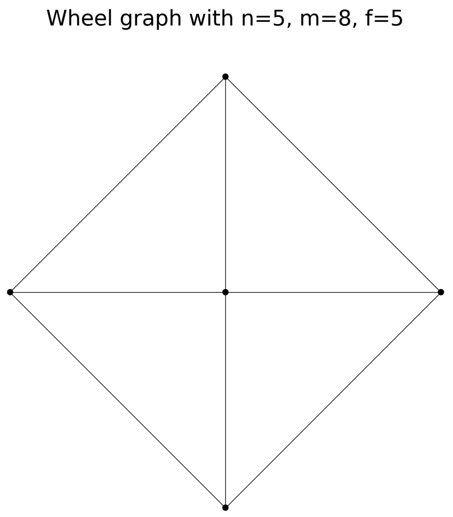
7.1. Galeria grafów kołowych



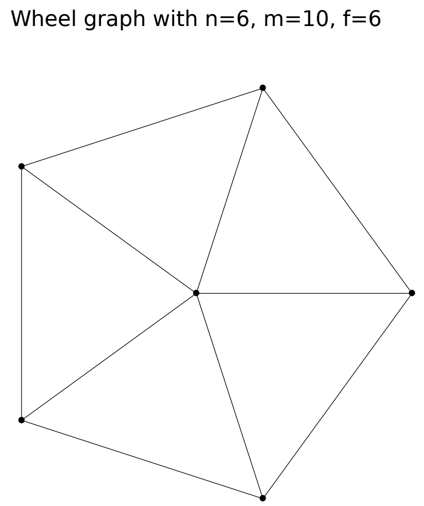
Rysunek 7.1. Graf koło $W_4 = K_4$, kubiczny. Ściana zewnętrzna grafu może być wybrana na cztery sposoby.

Tabela 7.1. Liczba wszystkich i kubicznych grafów Halina z n wierzchołkami. Dla $n = 4$ mamy tylko graf pełny tożsamy z grafem koło $K_4 = W_4$. Dla $n = 5$ mamy tylko graf koło W_5. Dla $n = 6$ mamy graf koło W_6 i 3-pryzmę.

n	1	2	3	4	5	6	7	8	9	10
wszystkie	0	0	0	1	1	2	2	4	6	13
kubiczne	0	0	0	1	0	1	0	1	0	3

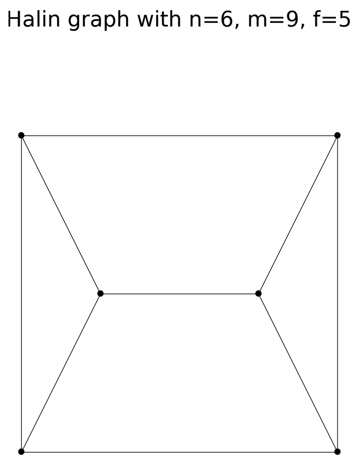


Rysunek 7.2. Graf koło W_5.

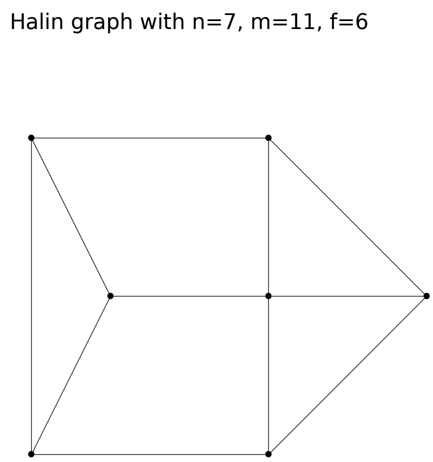


Rysunek 7.3. Graf koło W_6.

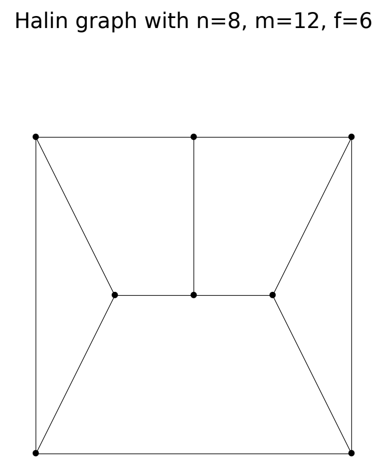
7.2. Małe grafy Halina



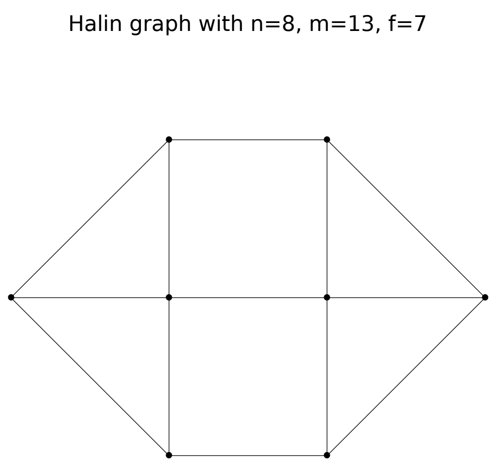
Rysunek 7.4. Graf Halina z $n = 6$, 3-pryzma, kubiczny. Ściana zewnętrzna grafu może być wybrana na trzy sposoby.



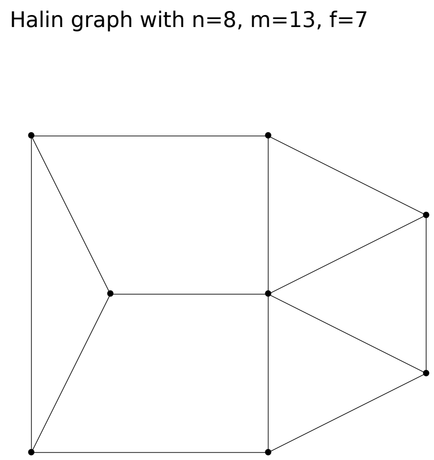
Rysunek 7.5. Graf Halina z $n = 7$.



Rysunek 7.6. Graf Halina z $n = 8$ (#1), kubiczny. Ściana zewnętrzna grafu może być wybrana na dwa sposoby. Jest to przykład grafu-naszyjnika.

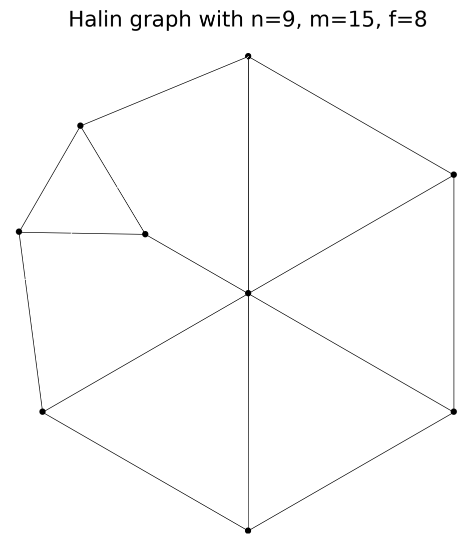


Rysunek 7.7. Graf Halina z $n = 8$ (#2). Jest to przykład grafu bicentrycznego $H_2(k, D) = H_2(1, 4)$ z pracy [43].

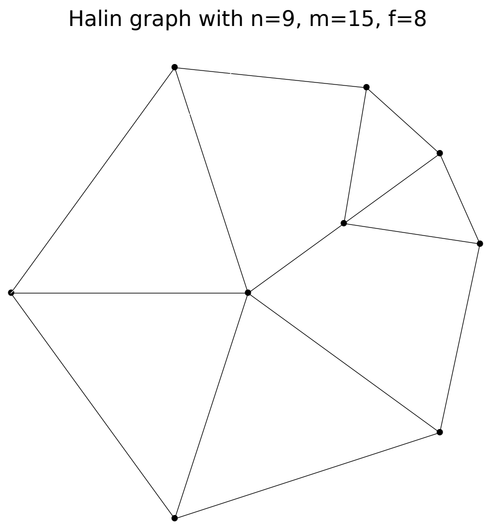


Rysunek 7.8. Graf Halina z $n = 8$ (#3).

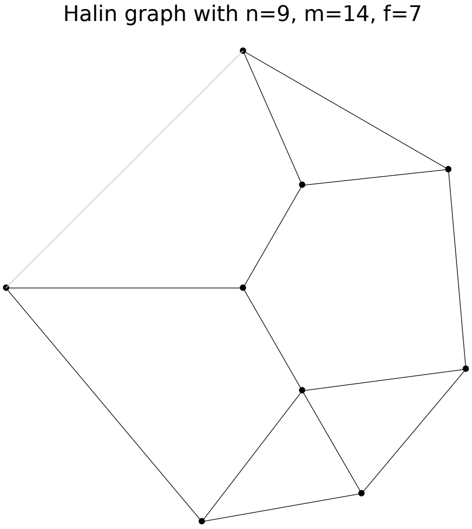
7.3. Grafy Halina z dziewięcioma wierzchołkami



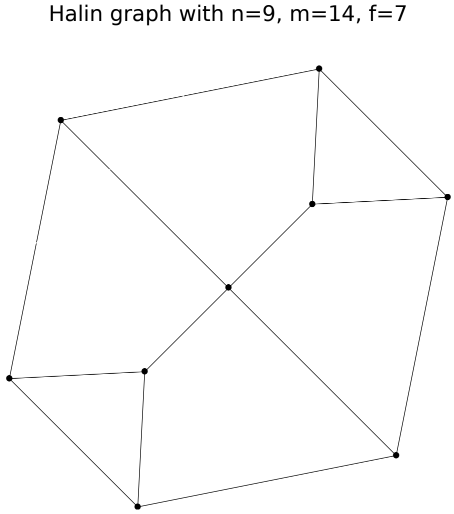
Rysunek 7.9. Graf Halina z $n = 9$ (#1).



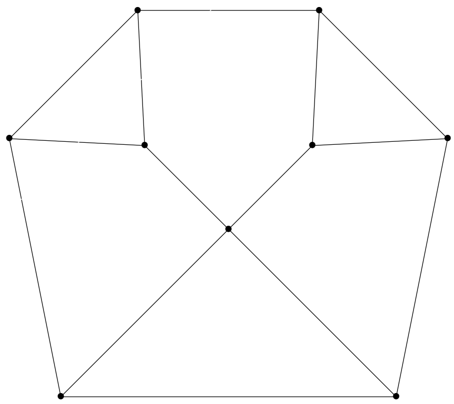
Rysunek 7.10. Graf Halina z $n = 9$ (#2).



Rysunek 7.11. Graf Halina z $n = 9$ (#3).



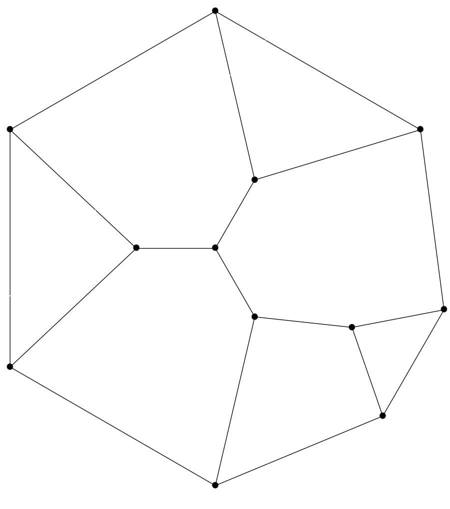
Rysunek 7.12. Graf Halina z $n = 9$ (#4).



Rysunek 7.13. Graf Halina z $n = 9$ (#5).

7.4. Większe grafy Halina

Graf Fruchta, $n = 12$, $m = 18$, kubiczny, liczba chromatyczna 3 (drzewo T ma 2 kolory, cykl C jest nieparzysty z dwoma kolorami, więc łatwo rozwiązać konflikty przez trzeci kolor), indeks chromatyczny 3 (cykl Hamiltona ma 2 kolory, reszta trzeci).



Rysunek 7.14. Graf Fruchta z $n = 12$.

8. Podsumowanie

W pracy zebrano najważniejsze informacje o grafach Halina. Pokazano, jak problemy trudne obliczeniowo mogą mieć rozwiązanie wielomianowe dla pewnej mniejszej klasy grafów. W szczególności omówiono grafy dwudzielne, grafy regularne, grafy planarne i drzewa. W ramach pracy powstały implementacje algorytmów do rozpoznawania grafów kołowych i ogólnych grafów Halina (zastosowanie redukcji).

W ramach pracy powstał moduł planarfactory z generatorami kilku grafów topologicznych ważonych: grafu cyklicznego C_n, grafu koła W_n. Moduł factory został wzbogacony o generatory kilku grafów ważonych: grafu koła W_n, grafu przypominającego wiatrak (do testów rozpoznawania grafu koła), ogólnego grafu Halina, kubicznego grafu Halina.

W ramach pracy powstało kilka implementacji algorytmów dla drzew, ponieważ tak jak grafy Halina drzewa mają strukturę rekurencyjną. Są to algorytmy wyznaczania najmniejszego pokrycia wierzchołkowego (dwie wersje), algorytmy wyznaczania największego zbioru niezależnego (dwie wersje), algorytmy wyznaczania najmniejszego zbioru dominującego (dwie wersje). Omówiliśmy również problemy kolorowania wierzchołków i kolorowania krawędzi dla drzew i grafów Halina. Chcieliśmy zaobserwować, jak algorytm dla drzewa uogólnia się przy przejściu do grafu Halina. Jest to dobrze widoczne w problemie kolorowania wierzchołków i kolorowania krawędzi grafu. W innych przypadkach albo nie udało się dotrzeć do artykułów źródłowych z opisem algorytmu, albo nie starczyło czasu na przygotowanie implementacji. Dla grafów Halina można wykonać dekompozycję drzewową o szerokości drzewiastej (ang. *treewidth*) równej 3. Prawdopodobnie na tym opierają się algorytmy dla pokrycia wierzchołkowego i zbioru dominującego.

A. Testy algorytmów

Rozdział zawiera wyniki testów algorytmów z niniejszej pracy.

A.1. Testy rozpoznawania grafów Halina

Testowanie algorytmu do rozpoznawania grafu kołowego. Dostajemy zależność $O(V)$ zarówno dla prawdziwych grafów kołowych (wykres A.1), jak i dla grafów rzadkich z liczbą krawędzi $|E| = 2|V| - 2$, jak dla grafów kołowych (wykres A.2).

Testowanie algorytmu do rozpoznawania grafu Halina. Dostajemy zależność $O(V)$ zarówno dla dowolnych grafów Halina (wykres **??**), jak i kubicznych grafów Halina (wykres **??**).

A.2. Testy pokrycia wierzchołkowego dla drzew

Testowanie algorytmu do wyznaczania najmniejszego pokrycia wierzchołkowego dla drzew metodą składania drzew. Wyniki przedstawia wykres A.3. Dostajemy zależność $O(V)$ dla drzew przypadkowych.

Testowanie algorytmu do wyznaczania najmniejszego pokrycia wierzchołkowego dla drzew metodą odrywania liści. Wyniki przedstawia wykres A.4. Dostajemy zależność $O(V)$ dla drzew przypadkowych.

A.3. Testy zbiorów niezależnych dla drzew

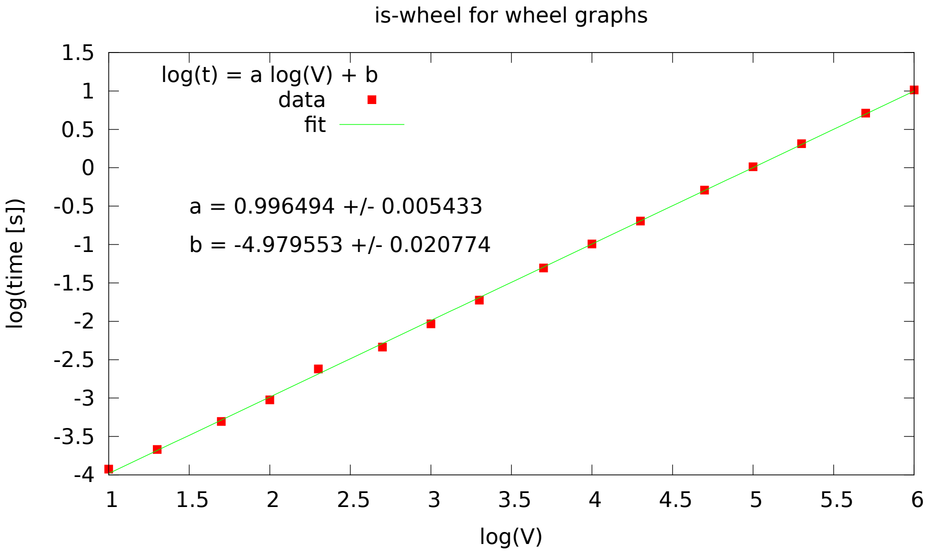
Testowanie algorytmu do wyznaczania największego zbioru niezależnego metodą składania drzew. Wyniki przedstawia wykres A.5. Dostajemy zależność $O(V)$ dla drzew przypadkowych.

Testowanie algorytmu do wyznaczania największego zbioru niezależnego metodą odrywania liści. Wyniki przedstawia wykres A.6. Dostajemy zależność $O(V)$ dla drzew przypadkowych.

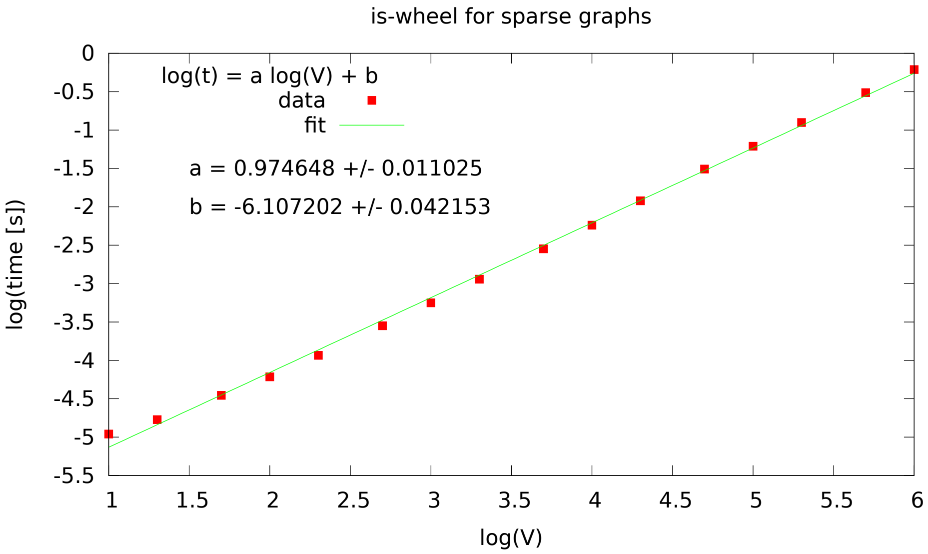
A.4. Testy zbiorów dominujących dla drzew

Testowanie algorytmu do wyznaczania najmniejszego zbioru dominującego metodą składania drzew. Wyniki przedstawia wykres A.7. Dostajemy zależność $O(V)$ dla drzew przypadkowych.

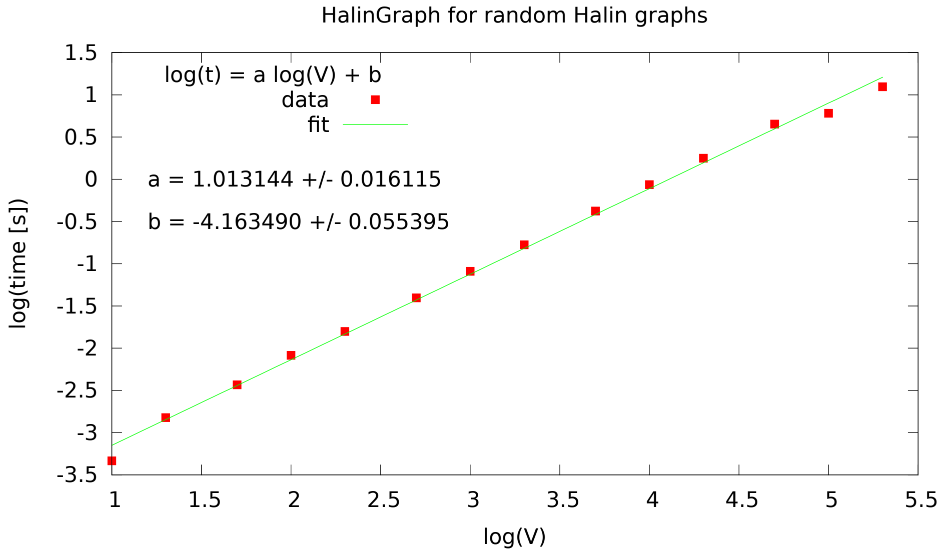
Testowanie algorytmu do wyznaczania najmniejszego zbioru dominującego metodą odrywania liści. Wyniki przedstawia wykres A.8. Dostajemy zależność $O(V)$ dla drzew przypadkowych.



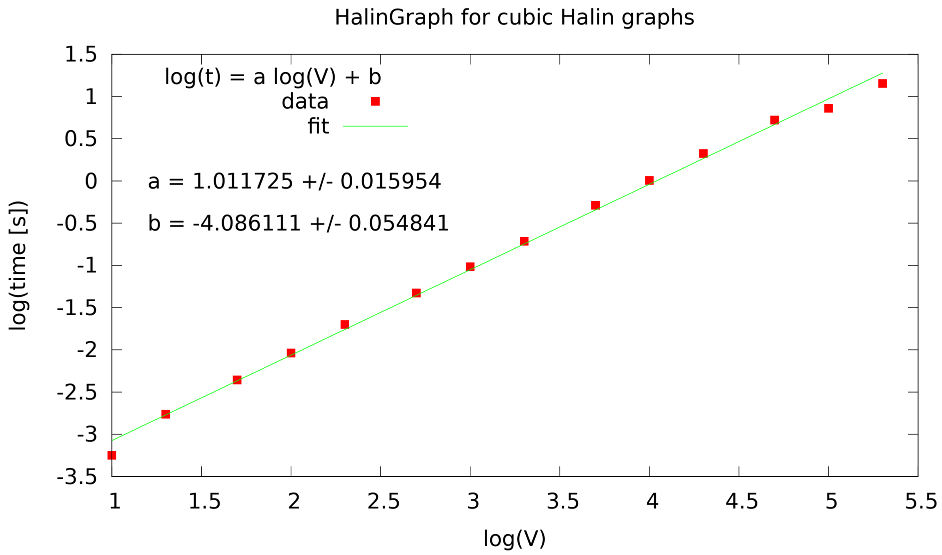
Rysunek A.1. Wykres wydajności rozpoznawania grafów kołowych (grafy kołowe). Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



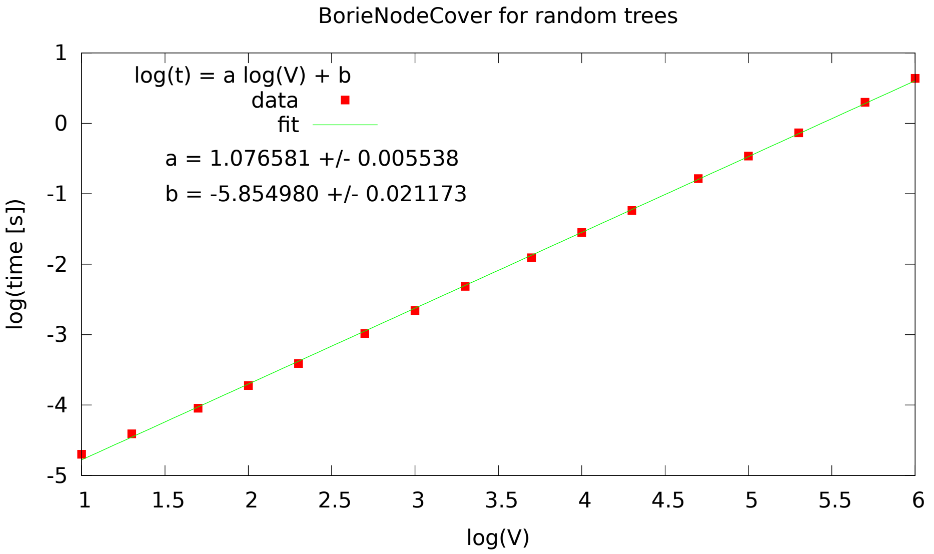
Rysunek A.2. Wykres wydajności rozpoznawania grafów kołowych (grafy rzadkie). Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



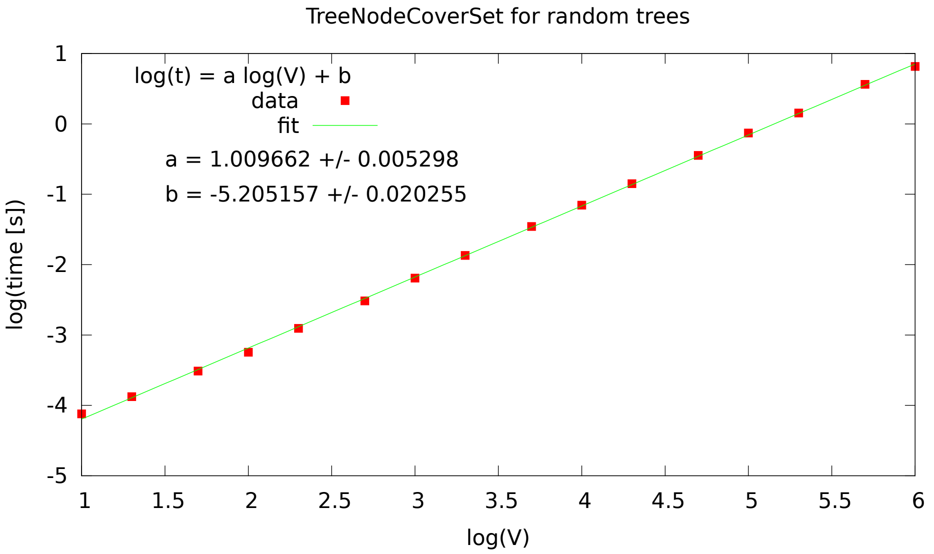
Rysunek A.3. Wykres wydajności rozpoznawania dowolnych grafów Halina. Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



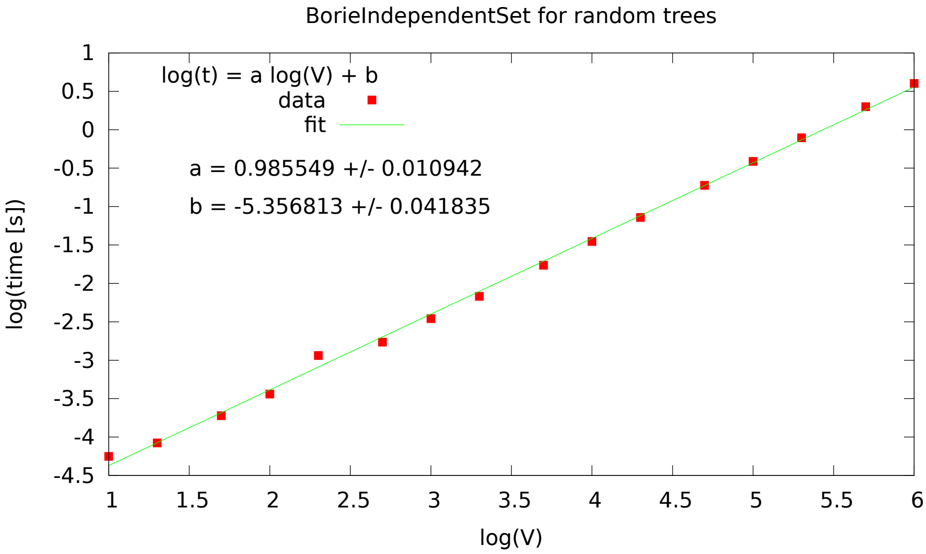
Rysunek A.4. Wykres wydajności rozpoznawania kubicznych grafów Halina. Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



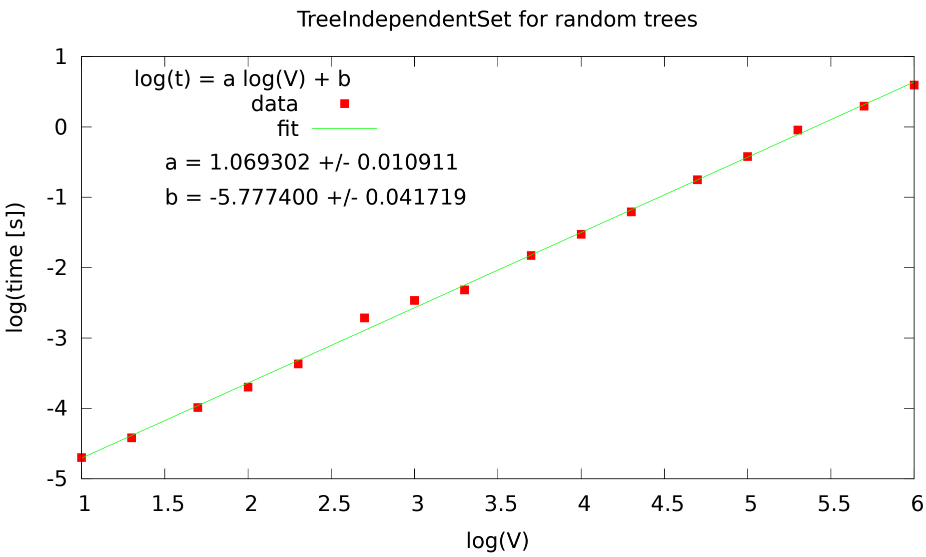
Rysunek A.5. Wykres wydajności algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla drzew metodą składania drzew. Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



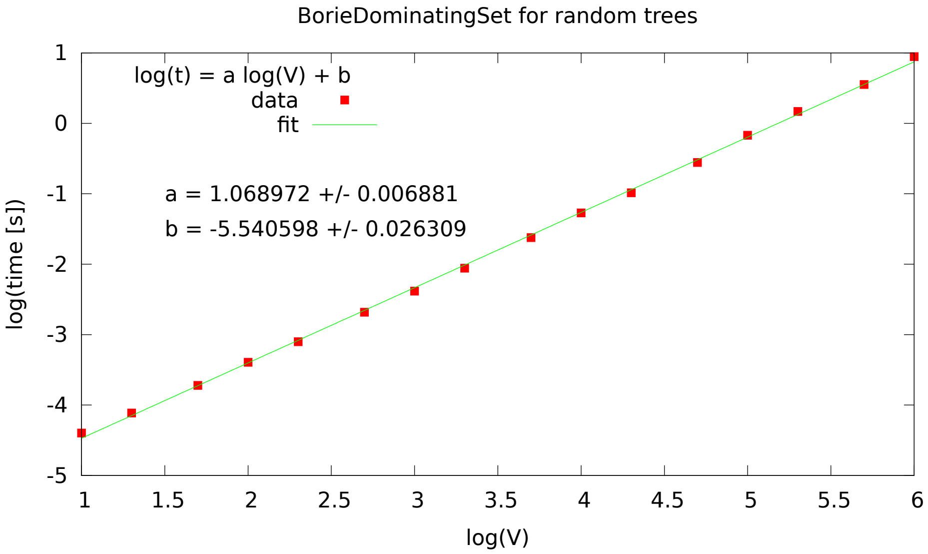
Rysunek A.6. Wykres wydajności algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla drzew metodą odrywania liści. Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



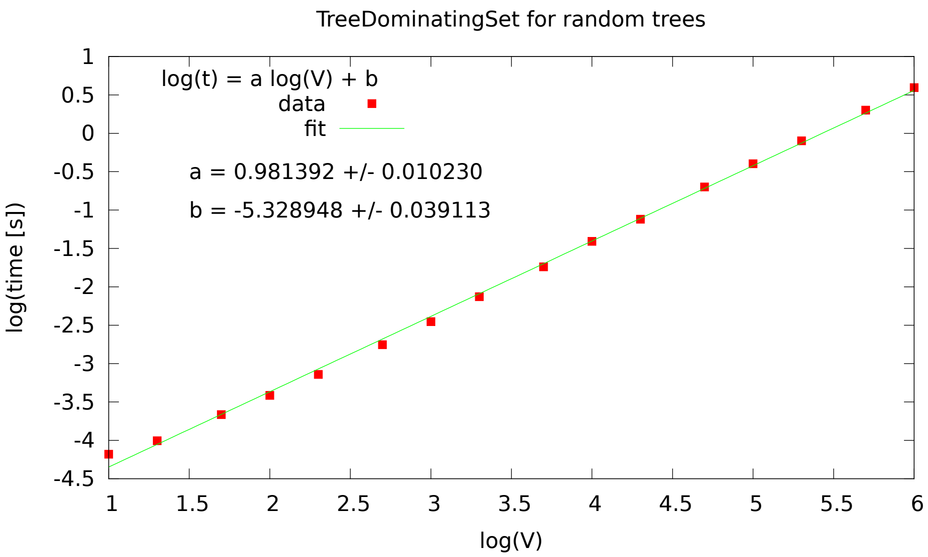
Rysunek A.7. Wykres wydajności algorytmu wyznaczania największego zbioru niezależnego dla drzew metodą składania drzew. Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



Rysunek A.8. Wykres wydajności algorytmu wyznaczania największego zbioru niezależnego dla drzew metodą odrywania liści. Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



Rysunek A.9. Wykres wydajności algorytmu wyznaczania najmniejszego zbioru dominującego dla drzew metodą składania drzew. Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.



Rysunek A.10. Wykres wydajności algorytmu wyznaczania najmniejszego zbioru dominującego dla drzew metodą odrywania liści. Współczynnik a bliski 1 potwierdza zależność liniową $O(V)$.

Bibliografia

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

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

[3] Andrzej Kapanowski, graphs-dict, GitHub repository, 2016, `https://github.com/ufkapano/graphs-dict/`.

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

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

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

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

[8] Wikipedia, Blossom algorithm, 2016, `https://en.wikipedia.org/wiki/Blossom_algorithm`.

[9] Jack Edmonds, *Paths, trees, and flowers*, Canadian Journal of Mathematics 17, 449-467 (1965).

[10] Wikipedia, Regular graph, 2016, `https://en.wikipedia.org/wiki/Regular_graph`.

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

[12] Michael R. Garey, David S. Johnson, Larry Stockmeyer, *Some simplified NP-complete problems*, Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, 47-63 (1974).

[13] Wikipedia, Tree (graph theory), 2016, `https://en.wikipedia.org/wiki/Tree_(graph_theory)`.

[14] Wikipedia, Outerplanar graph, 2016, `https://en.wikipedia.org/wiki/Outerplanar_graph`.

[15] Andrzej Proskurowski, Maciej M. Sysło, *Efficient vertex and edge coloring of outerplanar graphs*, SIAM. Journal on Algebraic and Discrete Methods 7, 131-136 (1986).

[16] Stanley Fiorini, *On the chromatic index of outerplanar graphs*, Journal of Combinatorial Theory, Series B 18, 35-38 (1975).

[17] Andrzej Proskurowski, Maciej M. Sysło, *Minimum dominating cycles in outerplanar graphs*, International Journal of Computer and Information Sciences 10, 127-139 (1981).

[18] Wikipedia, Planar graph, 2016, `https://en.wikipedia.org/wiki/Planar_graph`.

[19] Takao Nishizeki, *Planar Graph Problems*, Computing Supplementum 7, 53-68 (1990).

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

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

[22] Wikipedia, Halin graph, 2016,
`https://en.wikipedia.org/wiki/Halin_graph`.

[23] Rudolf Halin, *Studies on minimally n-connected graphs*, Combinatorial Mathematics and its Applications (Proc. Conf., Oxford, 1969), Academic Press, London, pp. 129-136 (1971).

[24] Thomas P. Kirkman, *On the enumeration of x-edra having triedral summits and an (x-1)-gonal base*, Philosophical Transactions of the Royal Society of London, 146, 399-411 (1856),
`http://www.jstor.org/stable/108592`.

[25] Maciej M. Sysło, Andrzej Proskurowski, *On Halin graphs*, Graph Theory: Proceedings of a Conference held in Łagów, Poland, February 10-13, 1981, eds. M. Borowiecki, John W. Kennedy, Maciej M. Sysło, Lecture Notes in Mathematics, Volume 1018, Springer-Verlag, Berlin-Heidelberg, pp. 248-256 (1983).

[26] S. B. Horton, R. G. Parker, R. B. Borie, *On some results pertaining to Halin graphs*, Congressus Numerantium 89, 65-87 (1992).

[27] J. A. Bondy, *Pancyclic graphs: recent results*, Infinite and Finite Sets, Colloquia Mathematica Societatis Janos Bolyai, Vol. 10, eds. A. Hajnal, R. Rado, V. T. Sos, Keszthely, Hungary, 181-187 (1973).

[28] L. Lovasz, M. Plummer, *On a family of planar bicritical graphs*, Proc. London Math. Soc. 30, 160-176 (1975).

[29] C. A. Barefoot, *Hamiltonian connectivity of the Halin graphs*, Congr. Numer. 58, 93-102 (1987); Eighteenth Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, Fla., 1987).

[30] J. A. Bondy, L. Lovasz, *Lengths of cycles in Halin Graphs*, Journal of Graph Theory 8, 397-410 (1985).

[31] Mirosława Skowrońska, *The Pancyclicity of Halin Graphs and their Exterior Contractions*, North-Holland Mathematics Studies, Volume 115, Pages 179-194, Cycles in Graphs, Annals of Discrete Mathematics 27, Elsevier Science Publishers (1985).

[32] G. Cornuejols, D. Naddef, W. R. Pulleyblank, *Halin Graphs and the Travelling Salesman Problem*, Mathematical Programming 26, 287-294 (1983).

[33] G. Cornuejols, D. Naddef, W. R. Pulleyblank, *The Traveling Salesman Problem in Graphs with 3-Edge Cutsets*, Journal of the Association for Computing Machinery 32, 383-410 (1985).

[34] Hans Bodlaender, *Dynamic programming on graphs with bounded treewidth*, Proceedings of the 15th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 317, Springer-Verlag, pp. 105-118 (1988).

[35] Y. Lu, Y. Li, D. Lou, *An algorithm to find the optimal matching an algorithm to find the optimal matching in Halin graphs*, IAENG International Journal of Computer Science 34(2) (2007).

[36] Magdalena Bojarska, *A note on Hamiltonian cycles in generalized Halin Graphs*, Discussiones Mathematicae, Graph Theory 30, 701-704 (2010).

[37] Wikipedia, Wheel graph, 2016,
`https://en.wikipedia.org/wiki/Wheel_graph`.

[38] Eric W. Weisstein, Hamiltonian Path. From MathWorld – A Wolfram Web Resource, 2016,
`http://mathworld.wolfram.com/HamiltonianPath.html`.

[39] Richard B. Borie, R. Gary Parker, Craig A. Tovey, *Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families*, Algorithmica 7, 555-581 (1992).

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

[41] A. Schulz, *Correctness-Proof of a greedy-algorithm for minimum vertex cover of a tree*, CS Stack Exchange, 2016, `http://cs.stackexchange.com/a/12198/1342`.

[42] Wikipedia, Dominating set, 2016, `https://en.wikipedia.org/wiki/Dominating_set`.

[43] T. Nicholas, G. R. Sanma, *Chromatic Number of in-Regular Types of Halin Graphs*, International Journal of Mathematics and Physical Sciences Research 3(2), 82-87 (October 2015-March 2016), `http://www.researchpublish.com/journal/IJMPSR/`.

[44] Alan Gibbons, Wojciech Rytter, *A fast parallel algorithms for optimally edge and vertex colouring Halin graphs*, Research Report 83, Departament of Computer Science, University of Warwick, October 1986.

[45] David Eppstein, *Simple Recognition of Halin Graphs and Their Generalizations*, Journal of Graph Algorithms and Applications 20(2), 323-346 (2016).

[46] David Eppstein, *Python Algorithms and Data Structures*, 2016, `http://www.ics.uci.edu/~eppstein/PADS/`.

[47] Sandra Pażyniowska, *Wizualizacja grafów w języku Python*, praca licencjacka, Uniwersytet Jagielloński, Kraków 2015.