Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Sandra Pażyniowska

Nr albumu: 1087593

Wizualizacja grafów w języku Python

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2015

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

Pragnę serdecznie podziękować Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za Jego życzliwość, nieocenioną pomoc i ogromne wsparcie, dzięki którym możliwe było powstanie tej pracy.

Streszczenie

W pracy zbadano problem rysowania grafów w języku Python. Zebrano szereg bibliotek i pakietów, które współpracują z Pythonem. Stworzono kilka implementacji nowych algorytmów, oraz pokazano struktury danych dla prostych grafów geometrycznych.

Zaimplementowano algorytm rekurencyjny do rysowania drzew radialnych, gdzie centrum drzewa jest w środku rysunku, a inne wierzchołki leżą na koncentrycznych kręgach. Stworzono też pomocniczy algorytm do wyznaczania centrum i promienia drzewa. Dla grafów skierowanych acyklicznych (dagów) przedstawiono zmodyfikowany algorytm sortowania topologicznego. Umożliwia on rysowanie dagów z krawędziami skierowanym z lewa na prawo.

Stworzono galerię grafów z wieloma grafami nazwanymi, zawierającą grafy pełne, grafy 3-regularne i 4-regularne, oraz grafy ze ścianami kwadratowymi. Zaimplementowano szereg skryptów do rysowania grafów. Jeden ze skryptów rysuje wierzchołki grafu na okręgu, co jest wygodne dla grafów Hamiltona, skierowanych i nieskierowanych. Wszystkie skrypty do rysowania używają Gnuplota 4, ze względu na jego elastyczność i interfejs wiersza poleceń. Główne algorytmy przetestowano pod kątem wydajności.

Słowa kluczowe: rysowanie grafów, grafy planarne, drzewa, grafy skierowane acykliczne

English title: Visualization of graphs with Python

Abstract

The problem of graph drawing with Python is considered. A range of libraries and packages is collected. Some new algorithms are created and implemented. Data structures for simple geometric graphs are shown.

The recursive algorithm for drawing radial trees is created, where the tree center is in the middle of the picture and other nodes are placed on concetric circles. The auxiliary algorithm for finding the tree center and the tree radius is used. For the case of directed acyclic graphs (dags), the modified topological sorting algorithm is created. It is used to draw dags with edges directed from left to right.

Gallery of graphs is created with several named graphs, including complete graphs, cubic graphs, quartic graphs, and graphs with square faces. A family of scripts for graph drawing is provided. There is a script for drawing graph nodes on the circle, which is very suitable for Hamiltonian graphs (directed or undirected). All scripts use Gnuplot 4 as a plotting engine, because it is a very fexible program with a command line interface. Main algorithms were tested in order to confirm the complexity.

Keywords: graph drawing, planar graphs, trees, directed acyclic graphs

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy jest przedstawienie narzędzi do wizualizacji grafów w języku Python. Wizualizacja grafów lub inaczej rysowanie grafów (ang. *graph drawing*) jest ważnym problemem praktycznym. Dobry rysunek może ujawnić relacje ukryte w plątaninie połączeń. Wydaje się, że nie ma jednej uniwersalnej metody rysowania grafów, tylko stosuje się wiele rozwiązań dobieranych do typów grafów i aktualnych zastosowań. Jednym z celów napisania pracy o takiej tematyce było więc zgromadzenie i prezentacja wybranych narzędzi do rysowania grafów. Drugim celem było stworzenie nowych algorytmów wspomagających rysowanie grafów i napisanie przykładowych skryptów, które wykorzystują te algorytmy. Skrypty miały posłużyć do utworzenia galerii przykładowych grafów.

1.1. Rysowanie grafów

Struktura matematyczna grafu znajduje szerokie zastosowanie, służy bowiem do przedstawienia i badania relacji między obiektami. Istnieje wiele algorytmów przeznaczonych do analizy grafów, np. znajdowanie najkrótszych ścieżek pomiędzy parą wierzchołków lub między wszystkimi wierzchołkami, czy znajdowanie minimalnego drzewa rozpinającego. Niniejsza praca poświęcona jest jednak nie samej analizie grafów, a przedstawieniu tych struktur graficznie, w sposób jak najbardziej przejrzysty - właśnie za pomocą narzędzi ich wizualizacji.

Bardzo istotną cechą grafów w kontekście ich wizualizacji jest kwestia planarności. Testowanie planarności grafu i możliwość narysowania go bez przecinających się krawędzi jest jednym z najbardziej fascynujących i intrygujących problemów algorytmicznych w dziedzinie rysowania grafów. Grafy planarne posiadają kilka interesujących właściwości, np. są rzadkie i dają się zawsze pokolorować przy użyciu co najwyżej czterech kolorów. Na grafach planarnych wiele operacji można przeprowadzić dużo bardziej efektywnie, niż w przypadku grafów nieplanarnych. Z punktu widzenia wizualizacji grafy planarne są bardziej czytelne i przejrzyste.

Istnieje wiele różnych podejść do problemu rysowania grafów - w zależności od tego, na jaką cechę chcemy położyć nacisk. Jednym z nich jest minimalizacja przecięć krawędzi w grafach generowanych automatycznie. Na ogół mniejsza liczba przecinających się krawędzi zwiększa czytelność grafu.

Innym podejściem jest dążenie do narysowania grafu z zachowaniem symetrii, co prócz estetycznego wyglądu uwidacznia istotne cechy grafu. W niektórych przypadkach symetryczna wersja grafu może być bardziej preferowana, niż jego wersja planarna. Przykładem grafu symetrycznego jest graf Petersena.

Kolejnym zagadnieniem jest rysowanie grafów będących drzewami. Istnieje wiele algorytmów służących do ich wizualizacji. Skupiają się one na automatycznym generowaniu układów relacyjnych. Drzewo jest właśnie typową strukturą danych przedstawiającą hierarchiczny model danych - wierzchołki drzewa odpowiadają jednostkom informacji, natomiast krawędzie symbolizują relacje między nimi.

Ciekawym przykładem reprezentacji graficznej grafu jest umieszczenie wierzchołków na okręgu. Krawędzie są wtedy liniami prostymi. Wiele przykładów grafów w tej reprezentacji znajduje się w galeriach w niniejszej pracy.

Istnieje również podejście zajmujące się rysowaniem prostokątnym - wierzchołki są punktami lub prostokątami, krawędzie są rysowane jako nieprzecinające się pionowe lub poziome linie, a każda ściana jest prostokątem [23].

Poza wyżej wymienionymi jest też mnóstwo innych sposobów na graficzne przedstawienie grafu. Wybór odpowiedniego podejścia zależy od wielkości grafu, od jego rodzaju oraz od cech, które go charakteryzują. Ważne jest również kryterium, na które kładziemy nacisk przy rysowaniu, możemy bowiem uzyskać różne rysunku tego samego grafu, w zależności od tego czy zależy nam np. na jak najmniejszej liczbie przecinających się krawędzi, czy na minimalnej odległości między wierzchołkami, czy też na symetrii.

W galeriach niniejszej pracy prezentujemy kilka wybranych sposobów reprezentacji grafów wraz z opisem najważniejszych własności.

1.2. Organizacja pracy

Rozdział 1 zawiera wprowadzenie do rysowania grafów. W rozdziale 2 omówiono narzędzia do wizualizacji grafów, które dobrze współpracują z Pythonem. W rozdziale 3 podano podstawowe definicje z teorii grafów. Rozdział 5 poświecony jest drzewom i algorytmom dla nich przygotowanym. Rozdział 6 poświecony jest grafom skierowanym acyklicznym i zastosowaniu sortowania topologicznego w rysowaniu tego typu grafów. Dalsze cztery rozdziały zawierają galerie grafów pełnych, regularnych stopnia 3 i 4, oraz grafów planarnych ze ścianami kwadratowymi. W rozdziale 11 opisano kilka popularnych sposobów wizualizacji grafów. Rozdział 12 zawiera podsumowanie pracy. W dodatku A omówiono strukturę skryptów rysujących grafy. W dodatku B przedstawiono wyniki testów wydajnościowych dla wybranych algorytmów.

2. Opis narzędzi do wizualizacji grafów

Do rysowania grafów można wykorzystać wiele istniejących bibliotek lub pakietów, a można też stworzyć nowe wyspecjalizowane programy. W tym rozdziale przedstawimy wybrane narzędzia powiązane z językiem Python.

2.1. Język Python

Język Python został wybrany do stworzenia implementacji ze względu na swą prostotę i intuicyjność. Jest to zorientowany obiektowo język programowania wysokiego poziomu, stworzony na początku lat dziewięćdziesiątych przez holenderskiego programistę Guido van Rossuma.

Python ceniony jest między innymi ze względu na łatwość pisania kodu. Jego niewątpliwą zaletą jest posiadanie dynamicznych typów danych, dzięki czemu nie trzeba deklarować za każdym razem typu zmiennej. Wspiera on zarówno programowanie obiektowe jak i funkcyjne - programista sam może wybrać, w którym stylu chce pisać swe programy. Poprawie czytelności tworzonego kodu służą wcięcia, które determinują zakres bloku instrukcji.

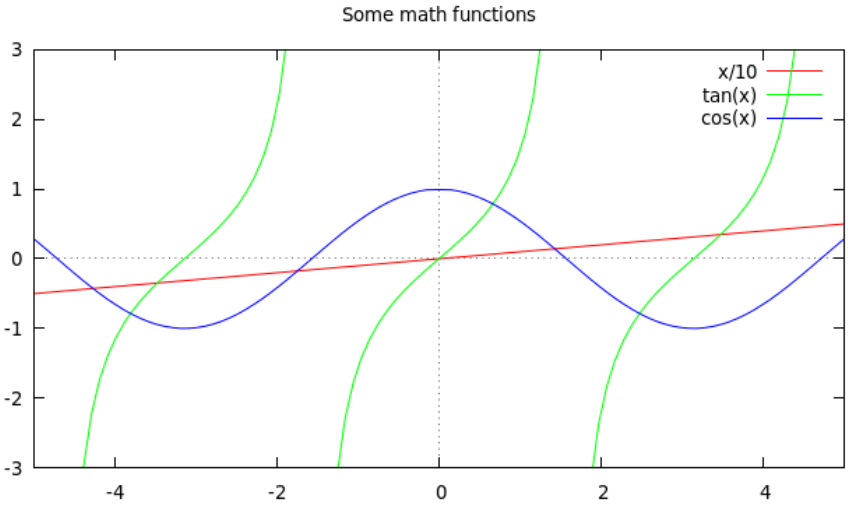
W ogólnym przypadku programy napisane w Pythonie działają zwykle wolniej niż na przykład te stworzone w C/C++ czy Javie, jednak samo pisanie ich zajmuje mniej czasu, ponieważ instrukcje są dużo krótsze i bardziej zwięzłe. Dzięki temu programy te wymagają o wiele mniej linii kodu (czasem różnica długości jest kilkukrotna!).

Python posiada bogatą bibliotekę standardową, obsługuje wyjątki i automatycznie zarządza pamięcią, co pozwala na tworzenie wydajnego oprogramowania w różnych dziedzinach nauki. Interpeter języka jest dostępny dla wielu platform sprzętowych.

Obecnie istnieją dwie gałęzie Pythona: Python 2 i Python 3. Do napisania skryptów w niniejszej pracy użyto Pythona 2, ze względu na uniwersalność i stabilność. Kod został napisany i przetestowany w wersji 2.7. [5].

Listing 2.1. Przykład deklaracji klasy w Pythonie.

```
class MyClass:
    """Opis klasy."""

    def __init__(self):
        """Inicjalizacja obiektu."""
        pass
```



Rysunek 2.1. Wykresy wygenerowane przy użyciu Gnuplota.

2.2. Pakiet Gnuplot.py

Do graficznego przedstawienia struktur grafowych użyto pakietu Gnuplot.py, który korzysta z interfejsu programu Gnuplot, służącego do tworzenia wykresów. Wybrano go ze względu na obecność wielu przydatnych funkcji oraz prostotę użycia.

Pakiet ten pozwala na używanie Gnuplota z poziomu Pythona. Można więc rysować tablice danych z pamięci, pliki danych czy funkcje matematyczne. Połączenie z Pythonem umożliwia automatyzację wielu funkcji.

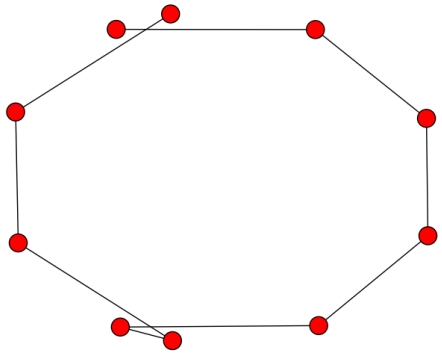
Pakiet ma konstrukcję obiektową, co pozwala na elastyczne ustawienie opcji wykresu, a także na uruchomienie wielu sesji interaktywnych jednocześnie. Do poprawnego działania pakietu Gnuplot.py wymagana jest obecność Pythona w wersji 2.2 lub późniejszej oraz numerycznego pakietu NumPy, potrzebnego do wykonania obliczeń związanych z prezentacją wykresu [7].

Listing 2.2. Przykład wykresu generowanego za pomocą Gnuplota.

```
gnuplot> set title "Some math functions"
gnuplot> set xrange [-5:5]
gnuplot> set yrange [-3:3]
gnuplot> set zeroaxis
gnuplot> plot x/10, tan(x), cos(x)
```

2.3. Pakiet NetworkX

NetworkX jest wieloplatformowym pakietem języka Python, służącym do tworzenia, manipulacji i badania struktur, dynamiki i funkcji złożonych sieci. Za jego pomocą można w prosty sposób stworzyć strukturę grafu, digrafu czy multigrafu. Dysponuje on wieloma algorytmami grafowymi (umożliwia m.in. znalezienie określonych podgrafów czy klik), zawiera generatory dla grafów



Rysunek 2.2. Graf wygenerowany przy użyciu biblioteki NetworkX.

klasycznych i losowych. Pozwala przetwarzać grafy z wierzchołkami prawie dowolnego typu, mogą to być np. teksty, obrazy, obiekty funkcji.

Pakiet ten został bardzo dobrze przetestowany. Do jego dodatkowych atutów należą szybkie prototypowanie i prostota użycia. Za jego pomocą można rysować grafy 2D i 3D [8].

Listing 2.3. Skrypt generujący przykładowy graf za pomocą NetworkX.

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> G = nx.path_graph(10)
>>> nx.draw(G)
>>> plt.savefig("simple_path.png")  # save as png
>>> plt.show()  # display
```

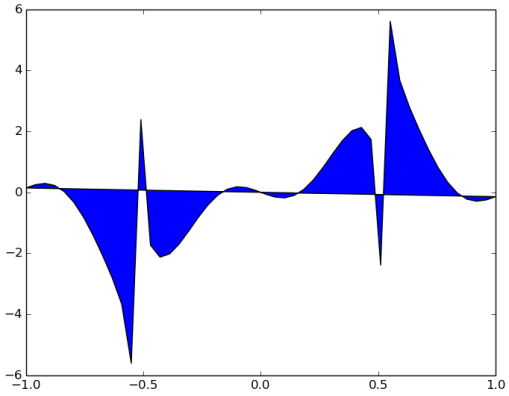
2.4. Biblioteka matplotlib

matplotlib jest biblioteką Pythona, przeznaczoną do konstrukcji wykresów 2D. Korzysta ona z pakietu numerycznego NumPy. Zawiera ona interfejs *pylab*, który z założenia ma być jak najbardziej podobny do interfejsu programu MATLAB, aby ułatwić korzystanie z biblioteki jego użytkownikom.

Biblioteka matplotlib pozwala na tworzenie wykresów, histogramów czy diagramów słupkowych i kołowych, za pomocą zaledwie kilku linii kodu. Zawiera wiele wbudowanych funkcji i stylów, dzięki którym można generować ogromną liczbę wykresów różnych typów. Jest silnie zintegrowana z Pythonem - użycie jej w innym języku jest zwykle niemożliwe [9].

Listing 2.4. Skrypt generujący przykładowy wykres za pomocą matplotlib.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1)
>>> y = np.cos(3 * np.pi * x) * np.tan(-3 * x)
>>> plt.fill(x, y, 'b')
>>> plt.show()
```



Rysunek 2.3. Wykres wygenerowany przy użyciu biblioteki matplotlib.

2.5. Pakiet igraph

igraph jest pakietem narzędzi do analizy sieci, kładącym nacisk na wydajność, przenośność i łatwość obsługi. Można go stosować w językach R, Python i C/C++.

Celem igraph jest dostarczenie zestawu typów danych i funkcji do szybkiej implementacji algorytmów grafowych oraz do łatwego radzenia sobie z grafami o dużych rozmiarach, z liczbą wierzchołków i krawędzi liczoną w milionach. Pakiet umożliwia trzy typy wizualizacji - za pomocą wbudowanej funkcji plot, poprzez użycie funkcji tkplot, używającej Tk GUI do podstawowych manipulacji grafami, oraz wykorzystując pakiet rgl, korzystający z OpenGL. [10].

Listing 2.5. Skrypt generujący przykładowy graf za pomocą igraph.

```
>>> from igraph import *
>>> g = Graph()
>>> g.add_vertices(6)
>>> g.add_edges([(0, 1), (0, 2), (1, 2), (1, 5), (2, 3), (2, 4)])
>>> layout = g.layout("fr")
>>> plot (g, layout=layout)
```

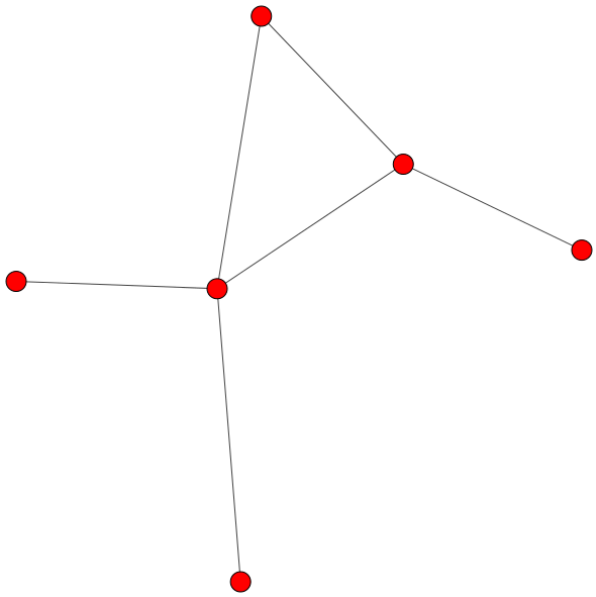
2.6. Biblioteka pySVG

Bibiloteka pySVG jest napisana w czystym Pythonie i służy do tworzenia dokumentów SVG [11]. Obecnie dostępna jest wersja 0.2.1 z roku 2010.

2.7. SVGFig

SVGFig to dość stary projekt z roku 2008 [12]. Napisany w czystym Pythonie, służy do generowania plików SVG.

Listing 2.6. Zastosowanie svgfig.



Rysunek 2.4. Graf wygenerowany przy użyciu biblioteki igraph.

```
>>> from svgfig import *
```

2.8. Graphviz

Graphviz jest to oprogramowanie *open source* do wizualizacji grafów [13]. Można tworzyć diagramy w wielu formatach, np. SVG, PDF, PS. Linux Debian zawiera pakiet *python-pygraphviz* z pythonowym interfejsem do programu Graphviz.

3. Teoria grafów

Teoria grafów to dział matematyki i informatyki zajmujący się badaniem własności grafów. Informatyka rozwija także algorytmy wyznaczające pewne właściwości grafów. Algorytmy te stosuje się do rozwiązywania wielu zadań praktycznych, często w dziedzinach na pozór nie związanych z grafami [14].

Definicje pojęć z teorii grafów mogą różnić się nieznacznie między sobą w zależności od źródeł, poniżej przedstawiono terminy istotne dla niniejszej pracy, sformuowane na podstawie podręczników Cormena [1], Wilsona [2], oraz Wojciechowskiego i Pieńkosza [3]

3.1. Grafy skierowane i nieskierowane

Graf skierowany lub digraf (ang. *directed graph*) jest parą uporządkowaną $G = (V, E)$, gdzie V jest skończonym i niepustym zbiorem *wierzchołków*, a E jest zbiorem *krawędzi*. Krawędź jest to uporządkowana para (u, v), gdzie u i v są wierzchołkami ze zbioru V. Krawędź jest *incydentna* do wierzchołka u, jeśli ma ona w tym wierzchołku swój koniec lub początek. W grafie możliwe jest istnienie *pętli* od danego wierzchołka do niego samego, czyli krawędzi typu (u, u). Grafy bez krawędzi wielokrotnych i pętli nazywamy *grafami prostymi*.

W grafie nieskierowanym $G = (V, E)$, zbiór krawędzi E to zbiór nieuporządkowanych par wierzchołków. Można powiedzieć, że dwie krawędzie skierowane (u, v) i (v, u) reprezentują tą samą krawędź nieskierowaną (jak właśnie jest w wielu implementacjach grafów). Graf prosty nieskierowany, w którym każda para różnych wierzchołków połączona jest krawędzią, nazywamy *grafem pełnym*.

Stopień wierzchołka w grafie nieskierowanym to liczba incydentnych z nim krawędzi. W grafie skierowanym *stopniem wejściowym* wierzchołka jest liczba krawędzi wchodzących do niego, natomiast *stopniem wyjściowym* jest liczba krawędzi z niego wychodzących. *Stopień grafu* to maksymalny stopień wierzchołka w grafie.

3.2. Ścieżki i cykle

Ścieżka o długości k z wierzchołka v_0 do wierzchołka v_k w grafie $G = (V, E)$ to ciąg wierzchołków $(v_0, v_1, v_2, ..., v_k)$ taki, że dla każdego $s \in [0, 1, ..., k-1]$ istnieje krawędź (v_s, v_{s+1}). *Ścieżka prosta* to taka ścieżka, w której żaden wierzchołek nie powtarza się.

Ścieżka $(v_0, v_1, v_2, ..., v_k)$ tworzy *cykl*, jeśli $v_0 = v_k$ oraz zawiera co najmniej jedną krawędź. Cykl nazywany jest *prostym*, jeśli dodatkowo $v_1, v_2, ..., v_k$ są różne, z wyjątkiem v_k. *Pętla* jest cyklem o długości 1. W grafie skierowa-

nym cyklami prostymi są przykładowo pętle i ścieżki typu (u, v, u). W grafie nieskierowanym prostym najprostszy cykl prosty o długości 3 tworzy ścieżka (u, v, t, u), gdzie u, v, t są różne. W multigrafie nieskierowanym krawędź podwójna tworzy cykl prosty o długości 2. Graf, który nie zawiera cykli, nazywany jest *acyklicznym*.

3.3. Spójność

Graf nieskierowany jest *spójny*, jeśli każda para wierzchołków jest połączona ścieżką. *Spójna składowa* grafu G to taki jego podgraf, który można wydzielić z całego grafu bez usuwania krawędzi. Graf spójny ma więc jedną spójną składową.

Graf skierowany jest *silnie spójny*, jeśli pomiędzy każdymi dwoma wierzchołkami istnieje ścieżka.

3.4. Drzewa

Drzewo jest to graf prosty nieskierowany, spójny i acykliczny. Więcej informacji o drzewach zostanie podanych w rozdziale 5.

3.5. Grafy dwudzielne

Graf dwudzielny G jest grafem, którego zbiór wierzchołków może być podzielony na dwa rozłączne zbiory A i B w taki sposób, by każda krawędź G łączyła wierzchołek zbioru A z wierzchołkiem zbioru B.

Pełny graf dwudzielny to graf dwudzielny, w którym każdy wierzchołek zbioru A jest połączony dokładnie jedną krawędzią z każdym wierzchołkiem zbioru B.

Sformuowane zostało twierdzenie, które pozwala określić, czy graf dwudzielny jest grafem hamiltonowskim (więcej informacji o grafie hamiltonowskim w podrozdziale 3.8).

Twierdzenie: Niech G będzie grafem dwudzielnym i niech $V(G) = V_1 \cup V_2$ będzie podziałem wierzchołków G. Jeśli G ma *cykl Hamiltona*, to $\mid V_1 \mid = \mid V_2 \mid$. Jeśli G ma *ścieżkę Hamiltona*, to wartości $\mid V_1 \mid$ i $\mid V_2 \mid$ różnią się co najwyżej o 1. Dla pełnych grafów dwudzielnych zachodzi implikacja w lewo, tj. jeśli $\mid V_1 \mid = \mid V_2 \mid$, to G ma cykl Hamiltona. Jeśli $\mid V_1 \mid$ i $\mid V_2 \mid$ różnią się co najwyżej o 1, to G ma ścieżkę Hamiltona.[6]

3.6. Grafy regularne

Graf regularny jest grafem, w którym każdy wierzchołek ma ten sam stopień [15]. Jeśli każdy wierzchołek ma stopień r, to graf nazywany jest *grafem regularnym stopnia r* lub krócej *grafem r-regularnym*.

Szczególne znaczenie mają *grafy kubiczne*, tzn. grafy regularne stopnia 3, przykładem jest graf Petersena. Można zauważyć, że graf pusty N_n jest grafem regularnym stopnia 0, graf cykliczny C_n jest grafem regularnym stopnia 2, a graf pełny K_n jest grafem regularnym stopnia $n-1$, gdzie n oznacza liczbę wierzchołków grafu.

3.7. Grafy cykliczne

Graf C_n spójny, regularny stopnia 2 nazywamy *grafem cyklicznym*. Graf otrzymany z grafu poprzez usunięcie jednej krawędzi nazywamy *grafem liniowym* P_n o n wierzchołkach. Graf powstały przez dodanie do cyklu C_{n-1} jeszcze jednego wierzchołka i połączenie go ze wszystkimi wierzchołkami cyklu nazywamy *kołem* i oznaczamy W_n.

3.8. Grafy hamiltonowskie

Graf hamiltonowski jest grafem zawierającym ścieżkę przechodzącą przez każdy wierzchołek dokładnie raz. Ścieżka taka nazywana jest *ścieżką Hamiltona*. Zamknięta ścieżka Hamiltona (przechodząca dokładnie raz przez wszystkie wierzchołki prócz pierwszego) zwana jest *cyklem Hamiltona*. Czasem graf, który posiada tylko ścieżkę Hamiltona, zwany jest *półhamiltonowskim*.

4. Implementacja grafów

W pracy została wykorzystana implementacja grafów rozwijana w Instytucie Fizyki Uniwersytetu Jagiellońskiego w Krakowie [16]. Przedstawimy krótko sposób korzystania z podstawowych klas odpowiadających strukturom lub algorytmom grafowym. Pierwsza sesja interaktywna z listingu 4.1 prezentuje utworzenie grafu nieskierowanego oraz wywołanie kilku przykładowych metod klasy Graph. Interfejs klasy może być zrealizowany na wiele sposobów, z różnymi wewnętrznymi strukturami danych, co świadczy o jego elastyczności.

Listing 4.1. Sesja interaktywna z grafem nieskierowanym.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> G = Graph(6)          # tworzy obiekt klasy Graph
>>> for node in [0, 1, 2, 3, 4, 5]:
...     G.add_node(node)       # dodaje wierzcholek
>>> G.add_edge(Edge(1, 2))     # dodaje krawedz
>>> G.add_edge(Edge(1, 0))
>>> G.add_edge(Edge(1, 4))
>>> G.add_edge(Edge(3, 4))
>>> G.add_edge(Edge(4, 5))
>>> G.is_directed()            # sprawdza, czy graf jest skierowany
False
>>> G.v()                      # zwraca liczbe wierzcholkow grafu
6
>>> G.e()                      # zwraca liczbe krawedzi grafu
5
>>> G.has_node(10)             # sprawdza istnienie wierzcholka
False
>>> G.has_node(5)
True
>>> G.show()                   # wyswietla graf
0 : 1
1 : 0 2 4
2 : 1
3 : 4
4 : 1 3 5
5 : 4
```

Kolejnym przykładem (listing 4.2) będzie zastosowanie algorytmu Floyda-Warshalla, służącego do znajdowania najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków w grafie ważonym. Algorytm ten opiera się na spostrzeżeniu, że jeśli koszt dojścia z wierzchołka v do u jest większy od sumy kosztów dojść z wierzchołka v do k i z k do u, to za lepszy koszt należy przyjąć tę nową, mniejszą wartość. Przetwarzany graf może posiadać krawędzie o

wagach ujemnych, ale nie mogą w nim występować cykle ujemne - algorytm ten może takie cykle wykrywać.

Listing 4.2. Prezentacja algorytmu Floyda-Warshalla.

```
>>> from floydwarshall import *
>>> G = Graph(5, True)          # graf skierowany
>>> nodes = [0, 1, 2, 3, 4]
>>> edges = [Edge(0, 2, 6), Edge(0, 3, 3), Edge(1, 0, 3),
        Edge(2, 3, 2), Edge(3, 1, 1), Edge(3, 2, 1),
        Edge(4, 1, 4), Edge(4, 3, 2)]
>>> for node in nodes:
...     G.add_node(node)
>>> for edge in edges:
...     G.add_edge(edge)
>>> G.show()
0 : 2(6) 3(3)
1 : 0(3)
2 : 3(2)
3 : 1 2
4 : 1(4) 3(2)
>>> algorithm = FloydWarshall(G)
>>> algorithm.run()
>>> algorithm.distance
{0: {0: 0, 1: 4, 2: 4, 3: 3, 4: inf},
1: {0: 3, 1: 0, 2: 7, 3: 6, 4: inf},
2: {0: 6, 1: 3, 2: 0, 3: 2, 4: inf},
3: {0: 4, 1: 1, 2: 1, 3: 0, 4: inf},
4: {0: 6, 1: 3, 2: 3, 3: 2, 4: 0}}
```

Algorytmy stworzone w niniejszej pracy wspierają konwencje i styl charakterystyczny dla innych, już istniejących w bibliotece algorytmów. Dzięki temu w spójny sposób rozszerzono bazę algorytmów na zagadnienia związane z rysowaniem grafów.

5. Rysowanie drzew

W tym rozdziale przedstawimy wybrane algorytmy wykorzystywane przy rysowaniu drzew. Implementacje algorytmów w języku Python powstały w ramach niniejszej pracy.

Drzewa od początku ogrywały ważną rolę w teorii grafów. A. Cayley (1821-1895) odkrył drzewa próbując obliczyć liczbę izomerów węglowodorów nasyconych C_kH_{2k-2} [4]. Dla takich cząsteczek przedstawionych jako grafy spójne mamy $|V| = n = 3k+2$, $|E| = 3k+1$. Liczba drzew zaetykietowanych wynosi n^{n-2}, jest to *wzór Cayleya*.

Dla każdego grafu spójnego nieskierowanego G można wyznaczyć (jedno lub kilka) *drzewo rozpinające* (ang. *spanning tree*). Jest to podgraf grafu G zawierający wszystkie wierzchołki G. Drzewo rozpinające niesie wiele pożytecznych informacji o strukturze pierwotnego grafu. W przypadku grafów ważonych często wyznacza się *minimalne drzewo rozpinające*, czyli drzewo rozpinające o minimalnej sumie wag krawędzi.

5.1. Własności drzew

W grafie $G = (V, E)$ następujące warunki są równoważne:

— G jest drzewem.
— Dla każdych dwóch wierzchołków s, t należących do V, w grafie G istnieje dokładnie jedna ścieżka z s do t.
— G jest spójny i $|E| = |V| - 1$.
— G jest acykliczny i $|E| = |V| - 1$.

Wszystkie drzewa są grafami dwudzielnymi planarnymi.

5.2. Przykłady drzew

Pokażemy wybrane klasy grafów, które są drzewami.

5.2.1. Graf liniowy

Graf liniowy P_n (ang. *linear graph* lub *path graph*) można otrzymać z grafu cyklicznego C_n przez usunięcie jednej krawędzi. Graf liniowy ma dwa liście.

5.2.2. Graf gwiazda

Graf gwiazda S_n (ang. *star graph*) jest to graf dwudzielny pełny $K_{1,n-1}$. Jest to drzewo z jednym wierzchołkiem wewnętrznym i $n-1$ liśćmi (ale dla $n < 3$ wszystkie wierzchołki to liście).

5.3. Algorytmy związane z drzewami

W grafie spójnym nieskierowanym $G = (V, E)$ *odległością* $d(s, t)$ między dwoma jego wierzchołkami s i t nazywamy długość najkrótszej ścieżki (tzn. liczbę krawędzi w najkrótszej ścieżce) między nimi [4]. Można udowodnić twierdzenie [4], że tak zdefiniowana odległość jest *metryką* w zbiorze V.

Ekscentryczność $E(s)$ wierzchołka s w grafie $G = (V, E)$ jest to odległość od s do najdalej od niego położonego wierzchołka w G [4], tzn.

$$E(s) = \max_{t \in V} d(s, t). \tag{5.1}$$

Wierzchołek o najmniejszej ekscentryczności w grafie $G = (V, E)$ nazywamy *centrum* grafu G. Udowodniono twierdzenie (König) [4], że każde drzewo ma albo jedno, albo dwa centra. Jeżeli drzewo ma dwa centra, to muszą one być przyległe.

Promień drzewa jest to ekscentryczność jego centrum. *Średnicę* drzewa definiuje się jako długość najdłuższej ścieżki w drzewie, co niekoniecznie jest równe dwóm promieniom [4].

5.3.1. Wyznaczanie centrum drzewa i promienia drzewa

Algorytm wyznaczania centrum drzewa jest pierwszym krokiem do problemu narysowania drzewa. Przedstawiony algorytm jest realizacją dowodu twierdzenia Königa, przedstawionego w książce Deo [4].

Dane wejściowe: Graf nieskierowany będący drzewem bez korzenia.

Problem: Wyznaczenie centrum drzewa i promienia drzewa.

Opis algorytmu: Algorytm rozpoczyna się od wyznaczenia stopni wierzchołków drzewa. Wierzchołki wiszące, mające stopień równy jeden, wstawiane są do kolejki. Następnie w kolejnych rundach usuwane są krawędzie ze znanymi wierzchołkami wiszącymi, przez co pojawiają się nowe wierzchołki wiszące, usuwane w następnej rundzie. Algorytm kończy się, kiedy ostatnie wierzchołki trafią do kolejki, a są to właśnie wierzchołki (jeden lub dwa) tworzące centrum. Algorytm wyznacza przy okazji promień drzewa, który jest równy liczbie wykonanych rund usuwania wierzchołków wiszących (przy dwóch centrach promień jest o jeden większy niż liczba rund).

Złożoność: Złożoność czasowa algorytmu wynosi $O(V)$, ponieważ dla drzewa $|E| = |V| - 1$, a w pętli while przetwarzamy listy sąsiedztwa wierzchołków drzewa. W reprezentacji słownikowej grafu stopień wierzchołka odczytujemy w czasie $O(1)$, co dla wszystkich wierzchołków zajmuje czas $O(V)$.

Uwagi: Algorytm przypomina sortowanie topologiczne wierzchołków dagu metodą usuwania wierzchołków niezależnych. Algorytm wykrywa sytuację, w której graf zawiera cykl, ponieważ w pewnym momencie nie ma już wierzchołków wiszących, a jeszcze nie wszystkie wierzchołki trafiły do kolejki.

Listing 5.1. Moduł treecenter.

```
#!/usr/bin/python

class TreeCenter:
    """Znajdowanie centrum drzewa i jego promienia."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
        self.graph = graph
        if self.graph.is_directed():
            raise ValueError("the graph is directed")
        self.tree_center = list()   # jeden lub dwa wierzcholki
        self.tree_radius = 0

    def run(self):
        """Przetwarzanie grafu."""
        # Wyznaczamy stopnie wierzcholkow drzewa.
        degree = dict((node, 0) for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            degree[edge.source] += 1
            degree[edge.target] += 1
        # Identyfikujemy wierzcholki stopnia 1.
        # Wstawiamy je do kolejki.
        Q = [None] * self.graph.v()
        qstart = 0  # pierwszy do pobrania
        qend = 0   # pierwszy wolny
        for node in self.graph.iternodes():
            if degree[node] == 1:
                Q[qend] = node
                qend += 1
        # Teraz etapami odrywamy wierzcholki od drzewa.
        while True:
            if qend == self.graph.v():  # zostaly dwa lub jeden
                for i in range(qstart, qend):
                    self.tree_center.append(Q[i])
                if len(self.tree_center) == 2:
                    self.tree_radius += 1
                break
            elif qstart == qend:
                raise ValueError("cycle detected")
            for step in range(qend-qstart):
                source = Q[qstart]
                qstart += 1
                degree[source] -= 1
                for target in self.graph.iteradjacent(source):
                    if degree[target] > 0:  # czy jest w drzewie
                        degree[target] -= 1
                    if degree[target] == 1:
                        Q[qend] = target
                        qend += 1
            self.tree_radius += 1
```

5.3.2. Wyznaczanie położeń wierzchołków drzewa

Zastosowanie algorytmu w skrypcie rysującym drzewa jest pokazane w dodatku A.2.

Dane wejściowe: Graf nieskierowany będący drzewem bez korzenia; opcjonalnie wierzchołek drzewa do umieszczenia w środku rysunku.

Problem: Wyznaczenie położenia wierzchołków drzewa na płaszczyźnie.

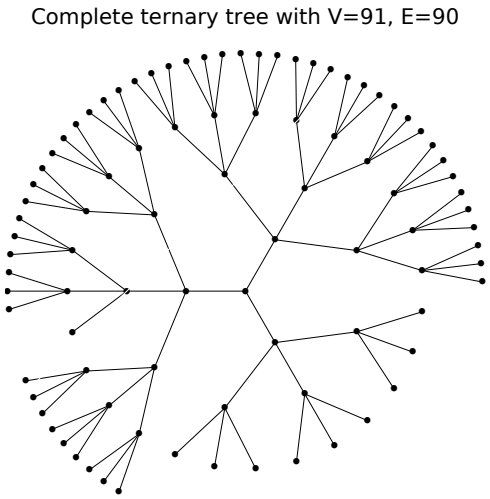
Opis algorytmu: Algorytm rozpoczyna się od wyznaczenia centrum drzewa, o ile nie podano wierzchołka, który ma się znaleźć w centrum rysunku. Następnie rekurencyjnie uruchamiana jest funkcja plot(), która dla każdego wierzchołka ustala jego współrzędne na płaszczyźnie. Startujemy od wierzchołka leżącego w środku rysunku. Potomkowie danego wierzchołka zostaną umieszczeni dalej od środka rysunku i w środku przedziału kątowego dla nich przeznaczonego. Przedział kątowy rodzica jest dzielony po równo dla jego dzieci. Każde pokolenie potomków znajduje się na jednym okręgu o promieniu proporcjonalnym do numeru pokolenia.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V)$, ponieważ przetwarzamy wszystkie wierzchołki drzewa.

Uwagi: Algorytm korzysta z klasy Point do przechowywania współrzędnych na płaszczyźnie. Najbardziej czytelne rysunki powstają wtedy, gdy wierzchołki o wysokim stopniu nie są zbyt oddalone od centrum drzewa. Inny korzystny przypadek to drzewa z wierzchołkami o niskim stopniu, np. drzewa binarne lub trójkowe. Rysunki od 5.1 do 5.3 przedstawiają kolejno: pełne drzewo trójkowe, pełne drzewo dwójkowe i drzewo przypadkowe.

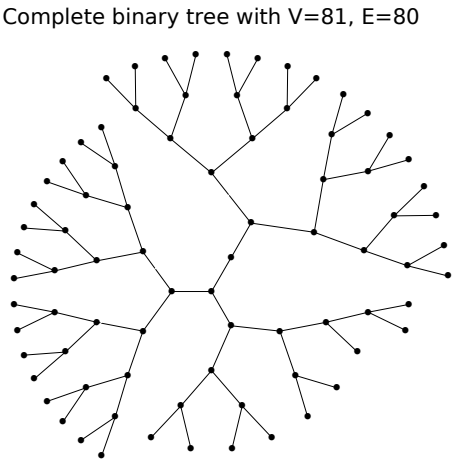
Listing 5.2. Moduł treeplot.

```
#!/usr/bin/python

import math
from treecenter import TreeCenter
from points import Point


class TreePlot:
    """Wyznaczanie pozycji wierzcholkow na plaszczyznie."""

    def __init__(self, graph):
        """Inicjalizacja algorytmu."""
        self.graph = graph
        self.points = dict()
        self.radius = 1.0

    def run(self, root=None):
        """Uruchomienie przetwarzania."""
        if root is None:
            algorithm = TreeCenter(self.graph)
            algorithm.run()
```

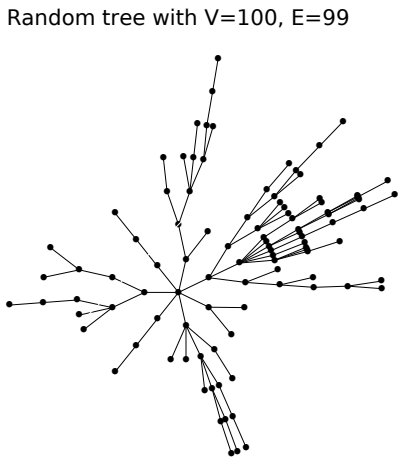


Rysunek 5.1. Pełne drzewo trójkowe.

```
            root = algorithm.tree_center[0]
        self.plot(root, 0.0, 2 * math.pi, level=0)

    def plot(self, source, left, right, level):
        """Rekurencyjne wyznaczanie wspolrzednych wierzcholka source
        w dostepnym przedziale katow (left, right), w kolejnej strefie
        oddalenia od srodka rysunku (level)."""
        alpha = 0.5 * (left + right)
        x = self.radius * level * math.cos(alpha)
        y = self.radius * level * math.sin(alpha)
        self.points[source] = Point(x, y)
        deg = self.graph.degree(source)
        if level == 0:    # analizujemy korzen drzewa
            if deg == 0:    # G.v() == 1, samotny wierzcholek
                return
            else:    # wszyscy sasiedzi do narysowania
                delta = (right - left) / deg
        else:
            if deg == 1:    # wiszacy wierzcholek
                return
            else:    # odpada wierzcholek, z ktorego przyszlismy
                delta = (right - left) / (deg - 1)
        for target in self.graph.iteradjacent(source):
            if target not in self.points:
                self.plot(target, left, left + delta, level + 1)
                left = left + delta
```



Rysunek 5.2. Pełne drzewo binarne.



Rysunek 5.3. Drzewo przypadkowe.

6. Rysowanie dagów

Dag jest to graf skierowany acykliczny (ang. *directed acyclic graph*) [17]. Dagi mają szerokie zastosowania w obliczeniach numerycznych, przetwarzaniu potokowym, kompresji, diagramach przepływu, itd. Dagi mogą opisywać porządek częściowy, który można przetworzyć do uporządkowania liniowego przy pomocy *sortowania topologicznego* [18]. Dla danego daga zwykle istnieje kilka sposobów posortowania topologicznego wierzchołków. Są dwa najpopularniejsze algorytmy sortowania topologicznego:
— algorytm usuwania wierzchołków niezależnych,
— wykorzystanie przeszukiwania grafu w głąb.

6.1. Algorytmy związane z dagami

Przedstawione w pracy algorytmy dla dagów koncentrują się na uwypukleniu uporządkowania topologicznego wierzchołków. Odpowiada to typowym zastosowaniom dagów, gdzie krawędzie skierowane pokazują kierunek przepływu, kolejność zadań do wykonania, itp.

6.1.1. Algorytm usuwania wierzchołków niezależnych

Przedstawimy specjalną wersję algorytmu usuwania wierzchołków niezależnych, która obok listy wierzchołków posortowanych topologicznie wyznacza numer cyklu, w którym dany wierzchołek mógł zostać usuniety jako niezależny. Ta informacja będzie pomocna przy rysowaniu dagów.

Dane wejściowe: Graf nieskierowany acykliczny.

Problem: Sortowanie topologiczne wierzchołków.

Opis algorytmu: Algorytm rozpoczyna się od wyznaczenia wierzchołków niezależnych, których stopień wejściowy jest zero. Następnie wierzchołki niezależne są usuwane w jednym cyklu, co generuje nowe wierzchołki niezależne do usunięcia w cyklu następnym. Operacje są powtarzane do wyczerpania wierzchołków. Dla kazdego wierzchołka zapisujemy informację w którym cyklu został usunięty.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V + E)$, ponieważ dla każdego wierzchołka przetwarzamy jego listę sąsiedztwa.

Uwagi: Algorytm wykrywa obecność cyklu w grafie, ponieważ wtedy nie ma już wierzchołków niezależnych, a jeszcze nie wszystkie wierzchołki zostały przetworzone.

Listing 6.1. Moduł topsort.

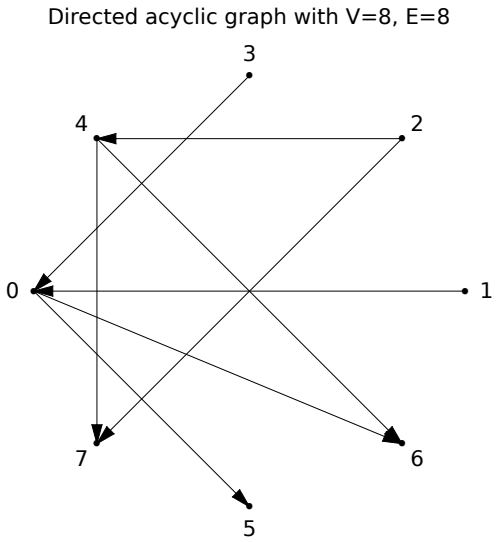
```
#!/usr/bin/python

class TopologicalSortList:
    """Sortowanie topologiczne wierzcholkow daga."""

    def __init__(self, graph):
        """Initializacja."""
        self.graph = graph
        self.sorted_nodes = [None] * self.graph.v()
        self.cycle = dict((node, 0) for node in self.graph.iternodes())

    def run(self):
        """Przetwarzanie."""
        inedges = dict((node, self.graph.indegree(node))
            for node in self.graph.iternodes())
        # Tworzymy kolejke wierzcholkow nie majacych poprzednikow.
        qstart = 0  # pierwszy do pobrania
        qend = 0    # pierwszy wolny
        for node in self.graph.iternodes():
            if inedges[node] == 0:
                self.sorted_nodes[qend] = node
                qend += 1
        # Teraz etapami odrywamy krawedzie.
        cycle_no = 0
        while True:
            if qstart == self.graph.v():   # wszystkie przetworzone
                    break
            elif qstart == qend:
                raise ValueError("cycle detected")
            for step in range(qend-qstart):
                source = self.sorted_nodes[qstart]
                self.cycle[source] = cycle_no
                qstart += 1
                # Usuwanie krawedzi wychodzacych.
                for edge in self.graph.iteroutedges(source):
                    inedges[edge.target] = inedges[edge.target]-1
                    if inedges[edge.target] == 0:
                        self.sorted_nodes[qend] = edge.target
                        qend += 1
            cycle_no += 1
```

6.1.2. Wyznaczanie położeń wierzchołków daga

Opisane zostaną dwa rozwiązania na wyznaczenie współrzędnych wierzchołków acyklicznego grafu skierowanego. W celu pokazania różnicy w ich działaniu, dla każdego zostanie wygenerowany ten sam graf.

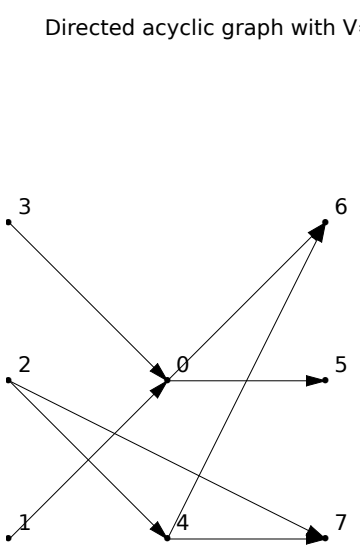
Pierwszym rozwiązaniem (rysunek 6.1) jest utworzenie listy wierzchołków grafu posortowanych topologicznie. Następnie wierzchołki są rozkładane na okręgu według kolejności występowania na liście. Lista jego posortowanych topologicznie wierzchołków to [1, 2, 3, 4, 0, 7, 5, 6].

Drugie rozwiązanie (rysunek 6.2) polega na ułożeniu wierzchołków w kolumnach wyznaczanych w czasie sortowania topologicznego. Numer kolumny dla danego wierzchołka jest równy numerowi cyklu sortowania topologiczne-



Rysunek 6.1. Acykliczny graf skierowany z ośmioma wierzchołkami na okręgu. Kolejność wierzchołków na okręgu odpowiada sortowaniu topologicznemu (kierunek przeciwny do ruchu wskazówek zegara).

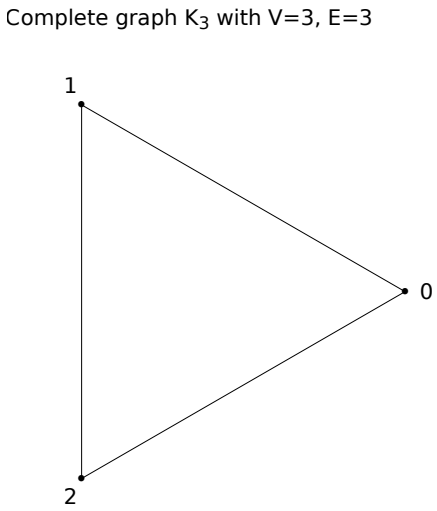
go, w którym wierzchołek był usuwany z grafu, jako wierzchołek bez krawędzi przychodzących. Wydaje się, że drugie rozwiązanie lepiej pokazuje wzajemne zależności pomiędzy wierzchołkami.



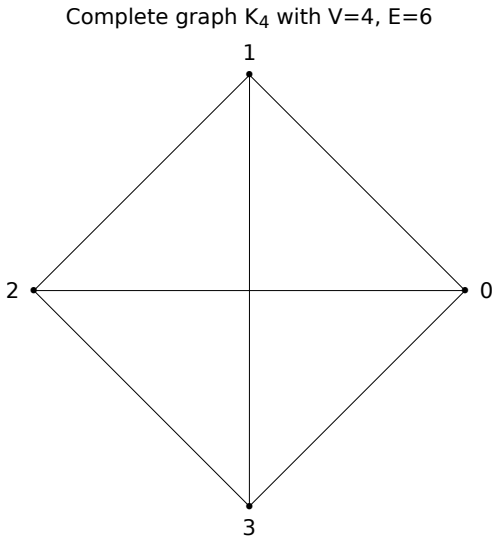
Rysunek 6.2. Acykliczny graf skierowany z ośmioma wierzchołkami w kolumnach. Krawędzie są skierowane z lewa na prawo.

7. Galeria grafów pełnych

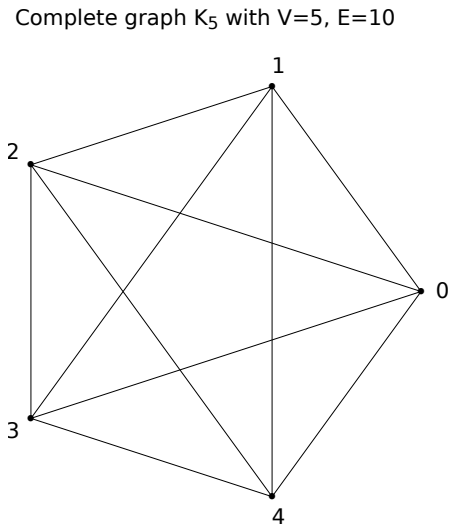
Przedstawimy rysunki grafów pełnych od K_3 do K_8.



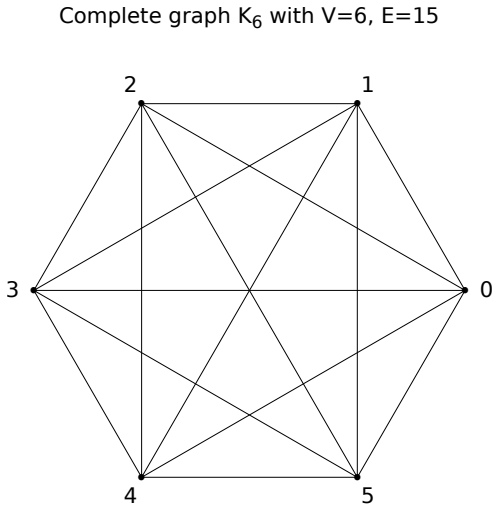
Rysunek 7.1. Graf pełny z trzema wierzchołkami.



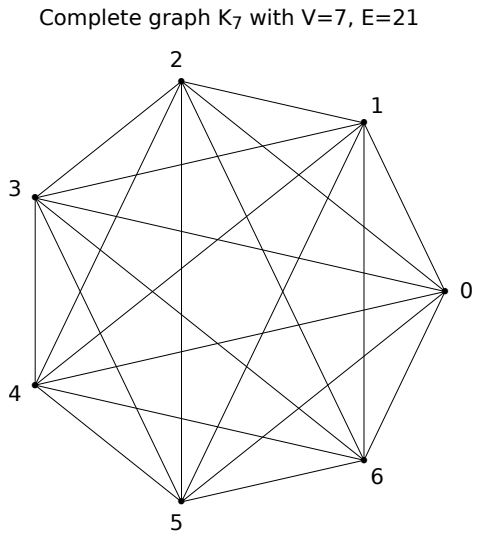
Rysunek 7.2. Graf pełny z czterema wierzchołkami.



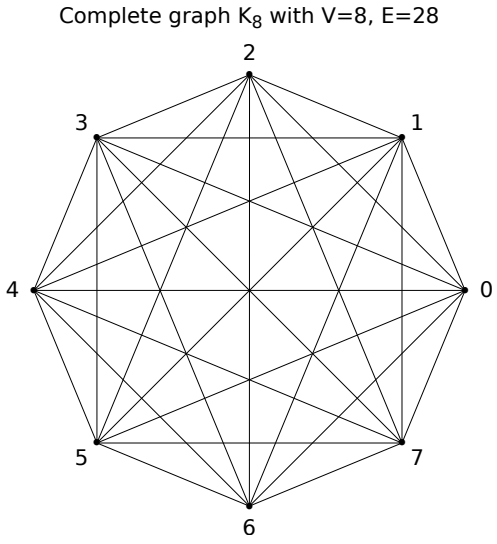
Rysunek 7.3. Graf pełny z pięcioma wierzchołkami.



Rysunek 7.4. Graf pełny z sześcioma wierzchołkami.



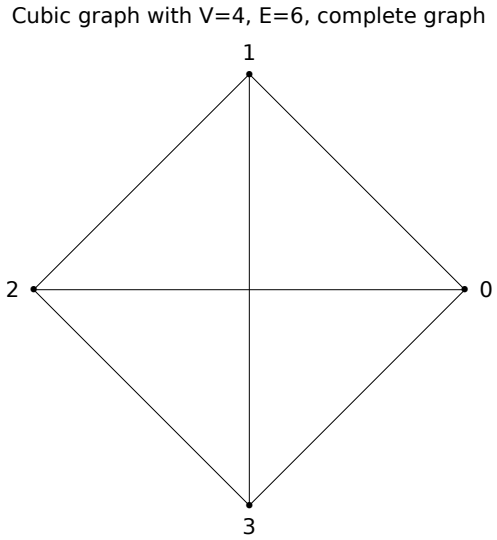
Rysunek 7.5. Graf pełny z siedmioma wierzchołkami.



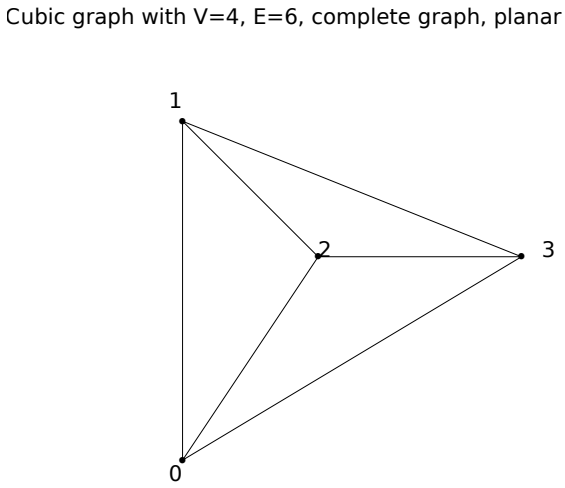
Rysunek 7.6. Graf pełny z ośmioma wierzchołkami.

8. Galeria grafów kubicznych

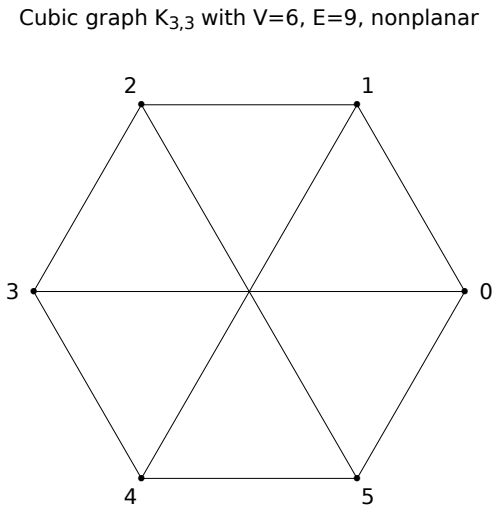
Graf kubiczny (ang. *cubic/trivalent graph*) to graf regularny stopnia 3 (graf 3-regularny) [19]. Wiele grafów kubicznych nosi nazwy pochodzące od nazwisk uczonych, co świadczy o ich dużej użyteczności. Z lematu o uściskach rąk wynika zależność $3|V| = 2|E|$, czyli grafy kubiczne muszą mieć parzystą liczbę wierzchołków. Przedstawimy galerię wszystkich grafów kubicznych spójnych z $|V| = 4(1), 6(2), 8(5)$, oraz wybrane większe grafy. Liczba różnych grafów dla danej liczby wierzchołków dana jest przez serię OEIS numer A002851 [20]. Meringer stworzył program GENREG do szybkiej generacji grafów regularnych [21].



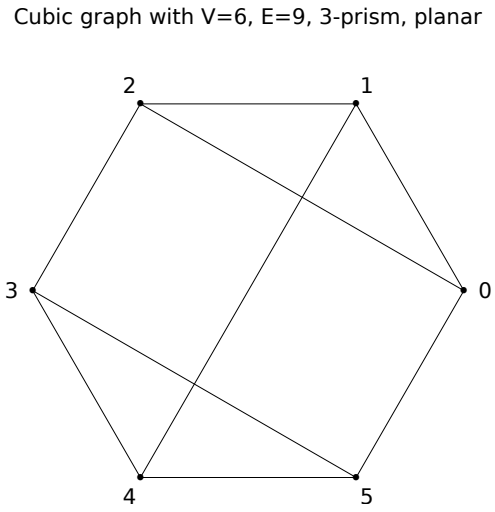
Rysunek 8.1. Graf pełny kubiczny z czterema wierzchołkami (#1).



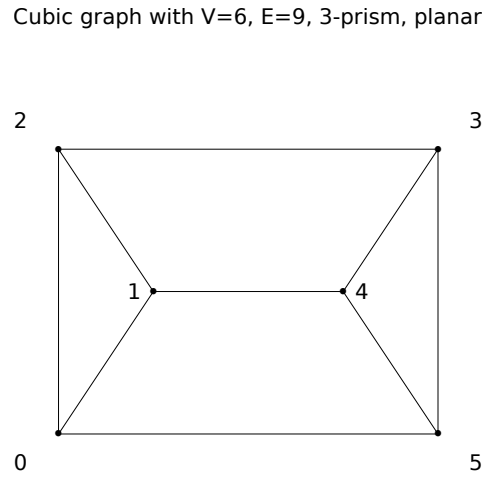
Rysunek 8.2. Graf pełny kubiczny z czterema wierzchołkami (#1 cd.).



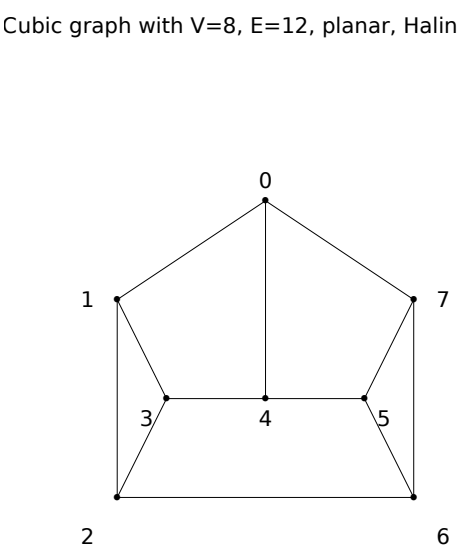
Rysunek 8.3. Graf kubiczny z sześcioma wierzchołkami (#1).



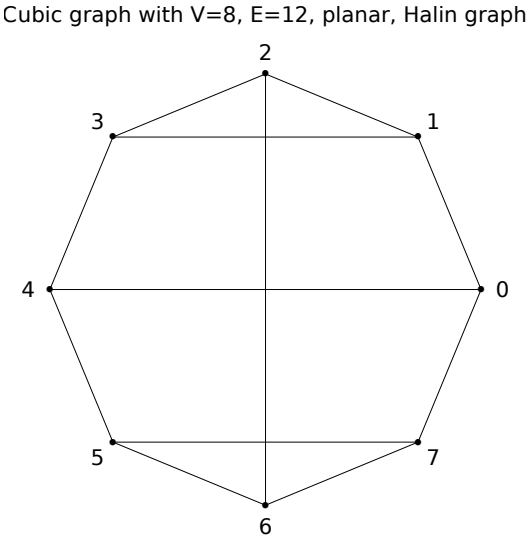
Rysunek 8.4. Graf kubiczny z sześcioma wierzchołkami (#2).



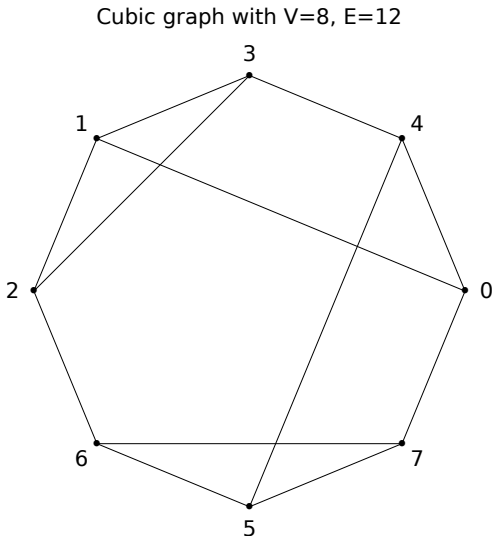
Rysunek 8.5. Graf kubiczny z sześcioma wierzchołkami (#2 cd.).



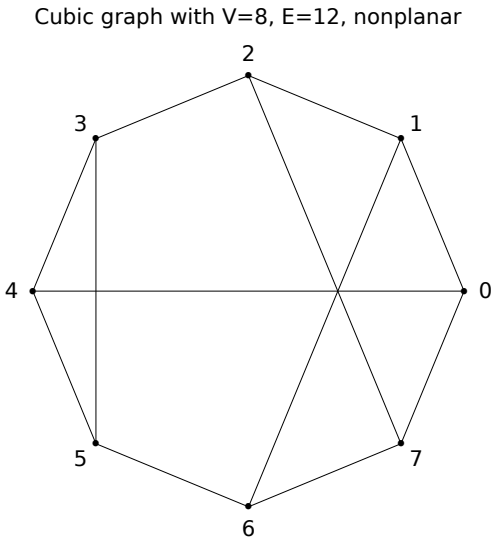
Rysunek 8.6. Graf kubiczny z ośmioma wierzchołkami (#1).



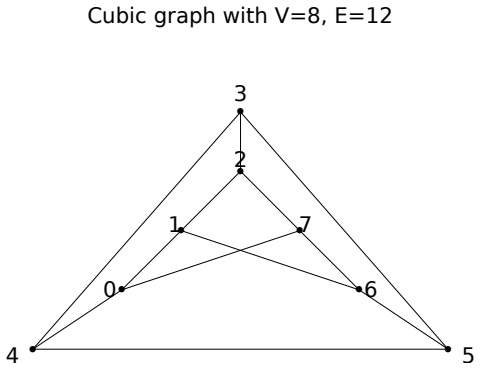
Rysunek 8.7. Graf kubiczny z ośmioma wierzchołkami (#1 cd.).



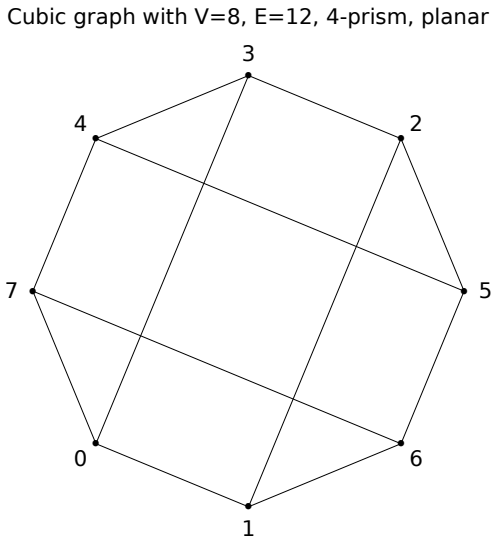
Rysunek 8.8. Graf kubiczny z ośmioma wierzchołkami (#1 cd.).



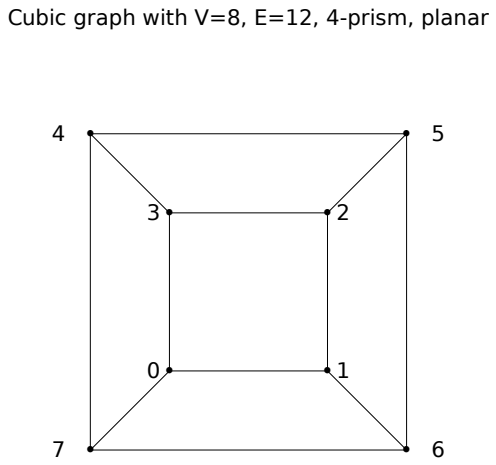
Rysunek 8.9. Graf kubiczny z ośmioma wierzchołkami (#2).



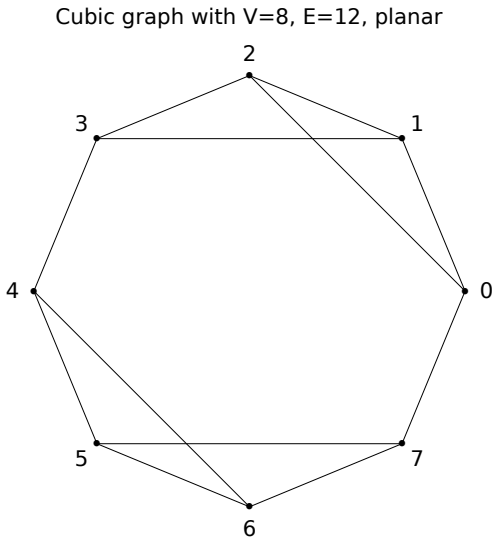
Rysunek 8.10. Graf kubiczny z ośmioma wierzchołkami (#2 cd.).



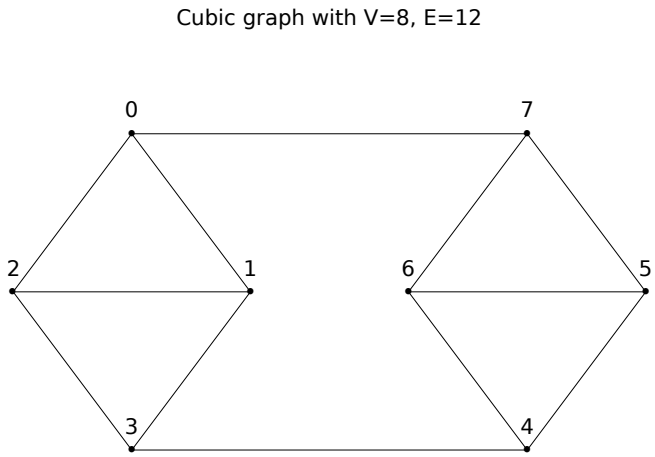
Rysunek 8.11. Graf kubiczny z ośmioma wierzchołkami (#3).



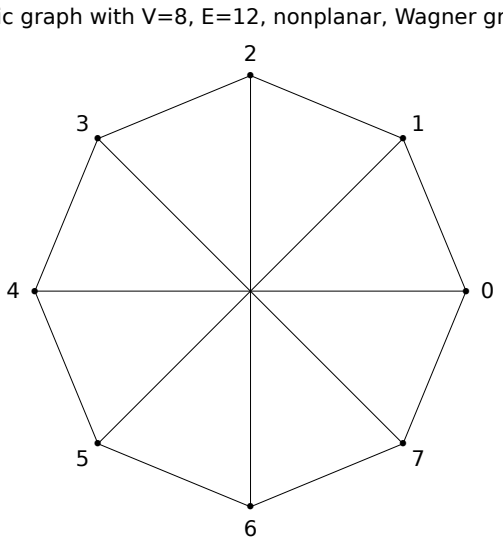
Rysunek 8.12. Graf kubiczny z ośmioma wierzchołkami (#3 cd.).



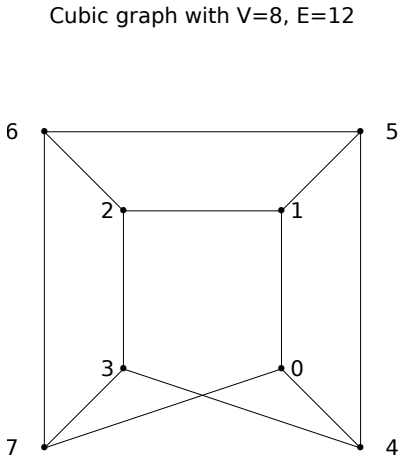
Rysunek 8.13. Graf kubiczny z ośmioma wierzchołkami (#4).



Rysunek 8.14. Graf kubiczny z ośmioma wierzchołkami (#4 cd.).



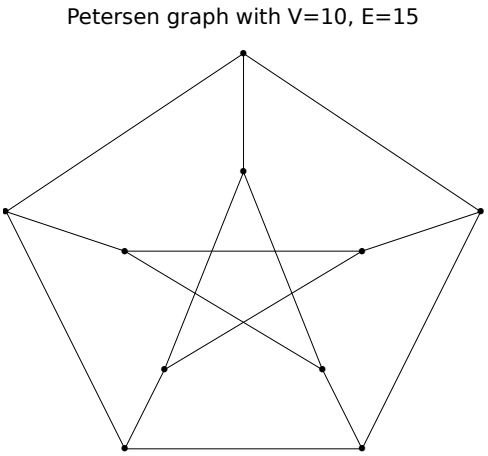
Rysunek 8.15. Graf kubiczny z ośmioma wierzchołkami (#5).



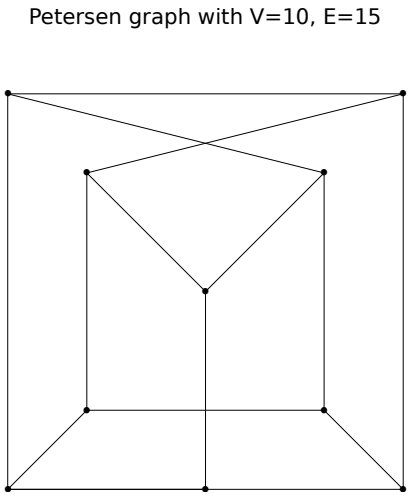
Rysunek 8.16. Graf kubiczny z ośmioma wierzchołkami (#5 cd.).

Graf Petersena jest najmniejszym grafem kubicznym bez mostów i cykli Hamiltona (posiada jednak ścieżkę Hamiltona). Nie jest grafem planarnym. Posiada 10 wierzchołków oraz 15 krawędzi.

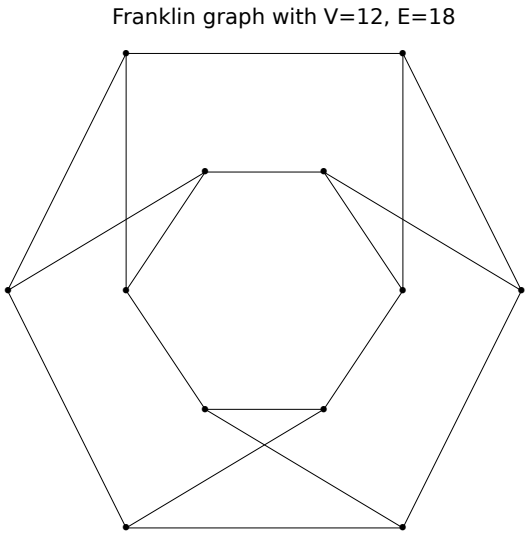
Graf Franklina jest kubicznym grafem hamiltonowskim o 12 wierzchołkach i 18 krawędziach.



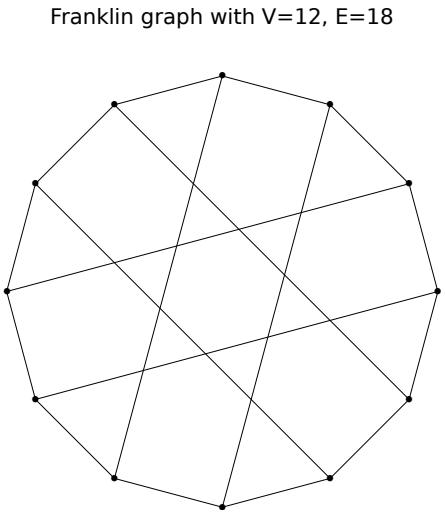
Rysunek 8.17. Graf Petersena (#1).



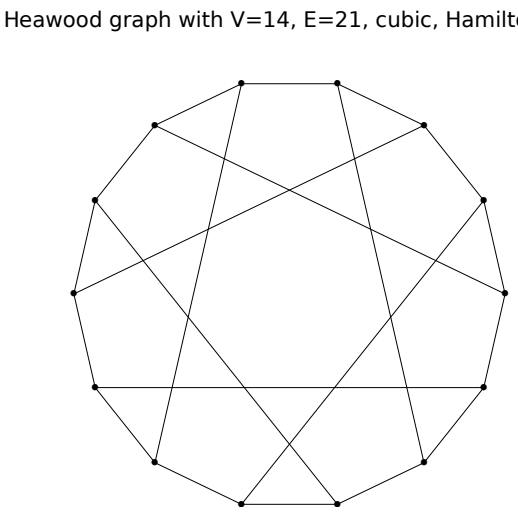
Rysunek 8.18. Graf Petersena (#2).



Rysunek 8.19. Graf Franklina (#1).



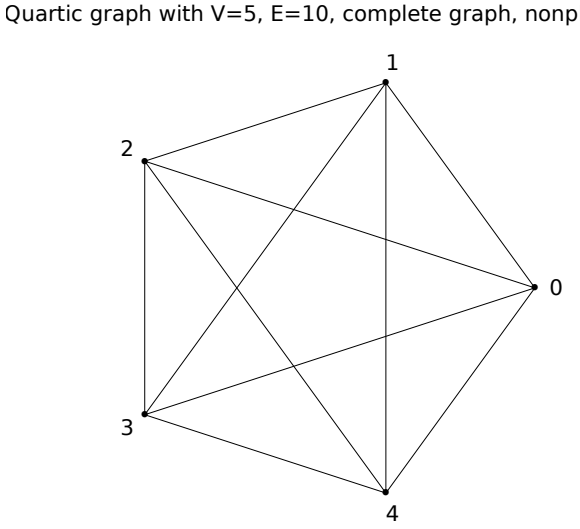
Rysunek 8.20. Graf Franklina (#2).



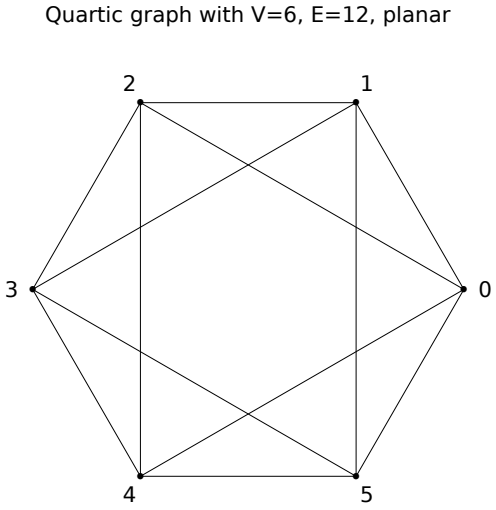
Rysunek 8.21. Graf Heawooda.

9. Galeria grafów regularnych stopnia 4

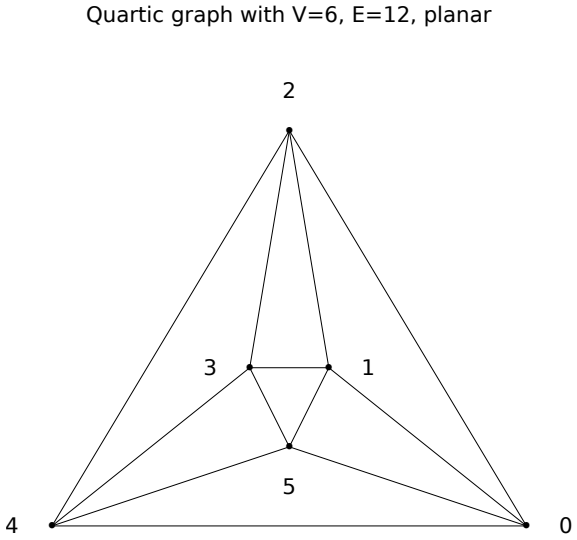
Graf 4-regularny (ang. *quartic graph*) ma wszystkie wierzchołki stopnia 4 [22]. Z lematu o uściskach rąk wynika zależność $|E| = 2|V|$. Przedstawimy galerię wszystkich grafów 4-regularnych spójnych z $|V| = 5(1), 6(1), 7(2), 8(6)$.



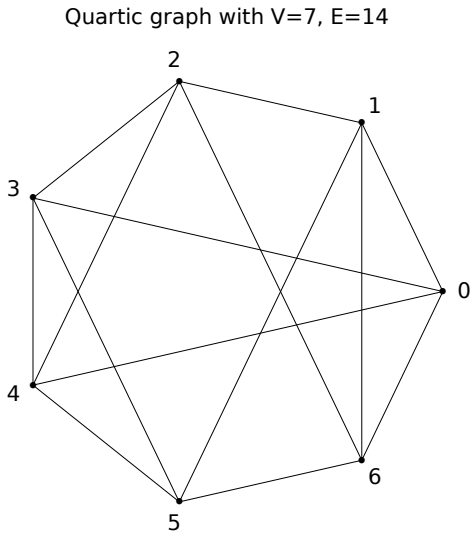
Rysunek 9.1. Graf 4-regularny z pięcioma wierzchołkami (#1).



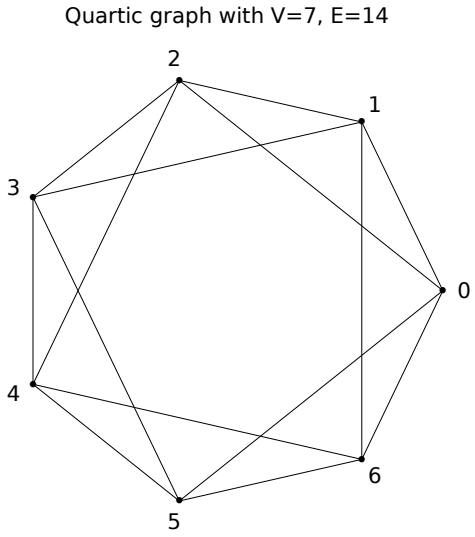
Rysunek 9.2. Graf 4-regularny z sześcioma wierzchołkami (#1).



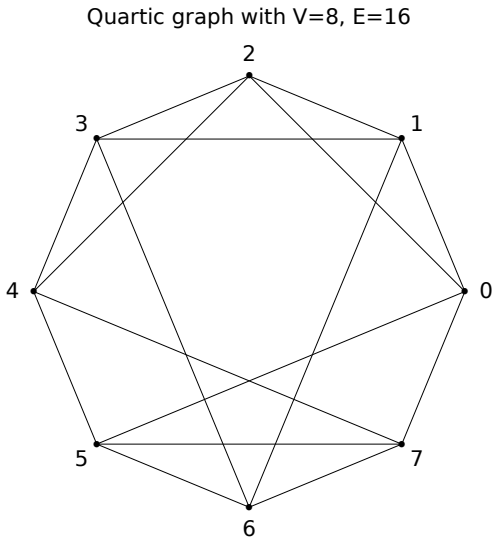
Rysunek 9.3. Graf 4-regularny z sześcioma wierzchołkami (#1 cd.).



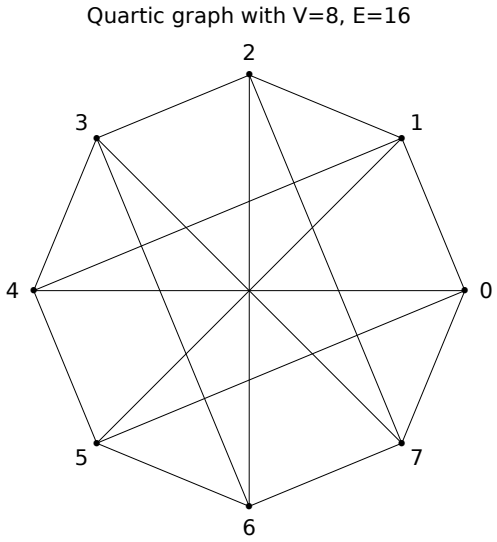
Rysunek 9.4. Graf 4-regularny z siedmioma wierzchołkami (#1).



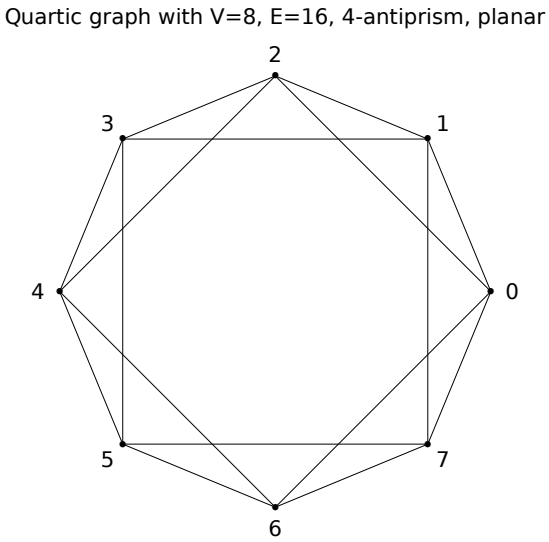
Rysunek 9.5. Graf 4-regularny z siedmioma wierzchołkami (#2).



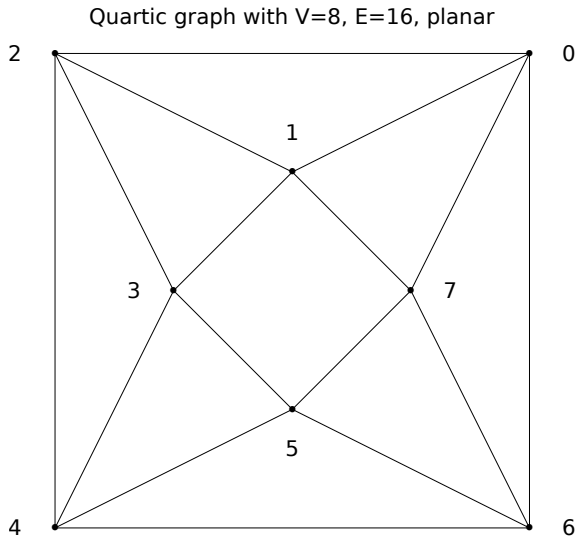
Rysunek 9.6. Graf 4-regularny z ośmioma wierzchołkami (#1).



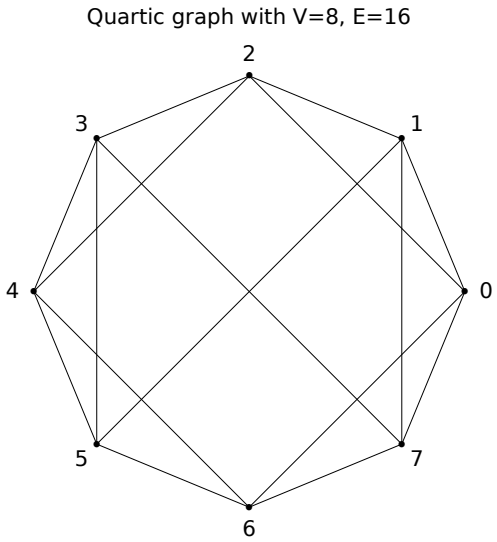
Rysunek 9.7. Graf 4-regularny z ośmioma wierzchołkami (#2).



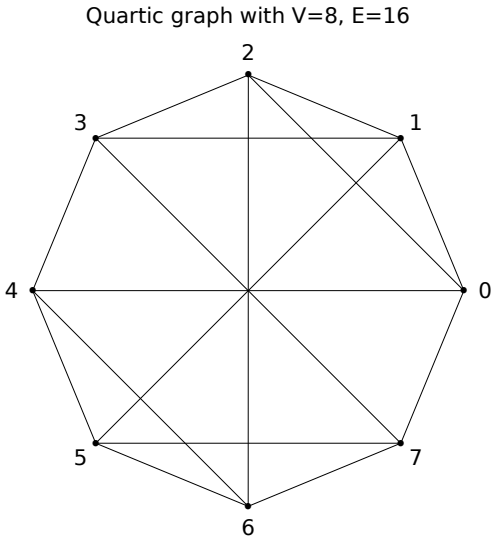
Rysunek 9.8. Graf 4-regularny z ośmioma wierzchołkami (#3).



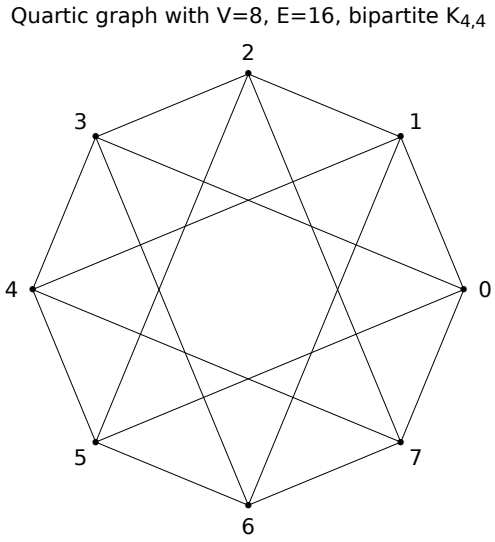
Rysunek 9.9. Graf 4-regularny z ośmioma wierzchołkami (#3 cd.).



Rysunek 9.10. Graf 4-regularny z ośmioma wierzchołkami (#4).



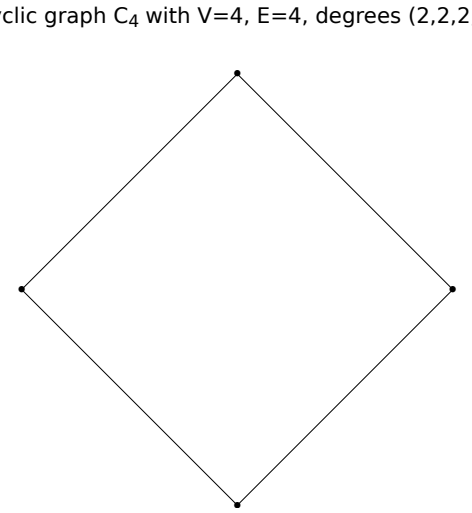
Rysunek 9.11. Graf 4-regularny z ośmioma wierzchołkami (#5).



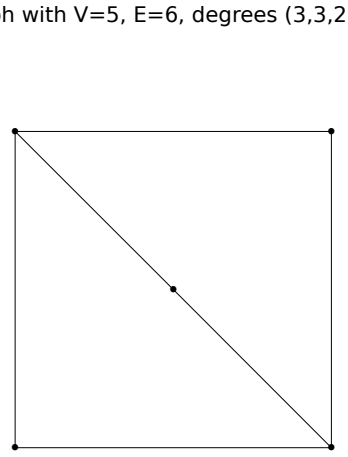
Rysunek 9.12. Graf 4-regularny z ośmioma wierzchołkami (#6).

10. Galeria grafów z kwadratowymi ścianami

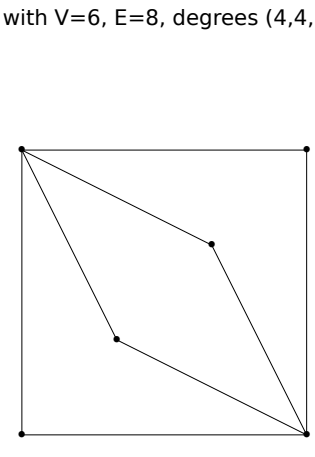
Jeśli G jest grafem planarnym, to każdy rysunek płaski grafu G dzieli zbiór punktów płaszczyzny, które nie leżą na G, na obszary zwane *ścianami* (ang. *faces*). W tym rozdziale pokażemy przykłady grafów o ścianach kwadratowych, czyli ścianach ograniczonych dokładnie czterema krawędziami. Z twierdzenia Eulera wynika, że liczba krawędzi wynosi $|E| = 2|V| - 4$, a liczba ścian $f = |V| - 2$.



Rysunek 10.1. Graf cykliczny o 4 wierzchołkach, $f = 2$.

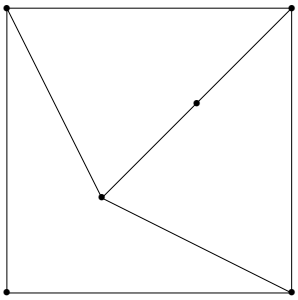


Rysunek 10.2. Graf z kwadratowymi ścianami o 5 wierzchołkach, $f = 3$.



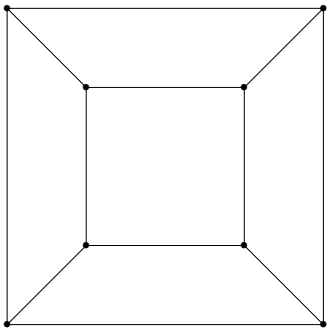
Rysunek 10.3. Graf z kwadratowymi ścianami o 6 wierzchołkach, $f = 4$.

Graph with V=6, E=8, degrees (3,3,3,3,2,2)

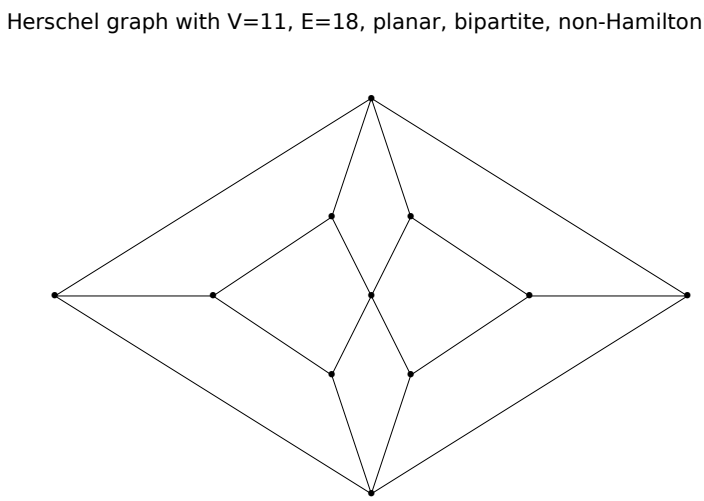


Rysunek 10.4. Graf z kwadratowymi ścianami o 6 wierzchołkach, $f = 4$.

Cubic graph with V=8, E=12, 4-prism, planar



Rysunek 10.5. Graf z kwadratowymi ścianami o 8 wierzchołkach.



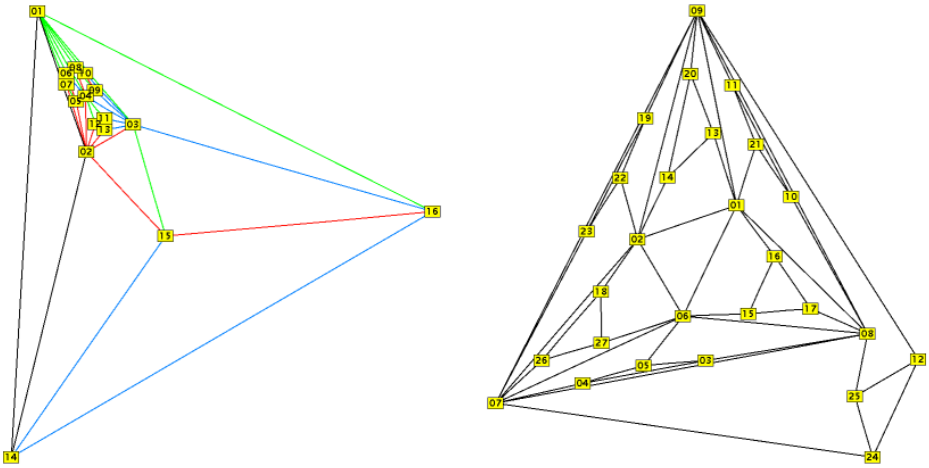
Rysunek 10.6. Graf Herschela, dwudzielny, planarny, półhamiltonowski.

11. Inne sposoby reprezentacji grafów

Istnieją również inne sposoby reprezentacji grafów, niż przedstawiony dotychczas rozkład wierzchołków na okręgu, czy też rysowanie grafu w postaci drzewa. Zaletą przedstawionych poniżej przykładowych podejść jest ich prostota oraz czytelność, krawędzie bowiem nie przecinają się, a rysunki są przejrzyste. Ponadto można je stosować zarówno dla grafów skierowanych, jak i nieskierowanych.

Oprócz narzędzi przedstawionych w rozdziale 2 istnieją biblioteki nie związane z językiem Python, które oferują możliwość rysowania grafów.

Interesującym narzędziem do konstrukcji grafów jest biblioteka PIGALE [24], skoncentrowana głównie wokół grafów planarnych. Poniżej znajdują się przykładowe grafy wygenerowany przy jej pomocy.



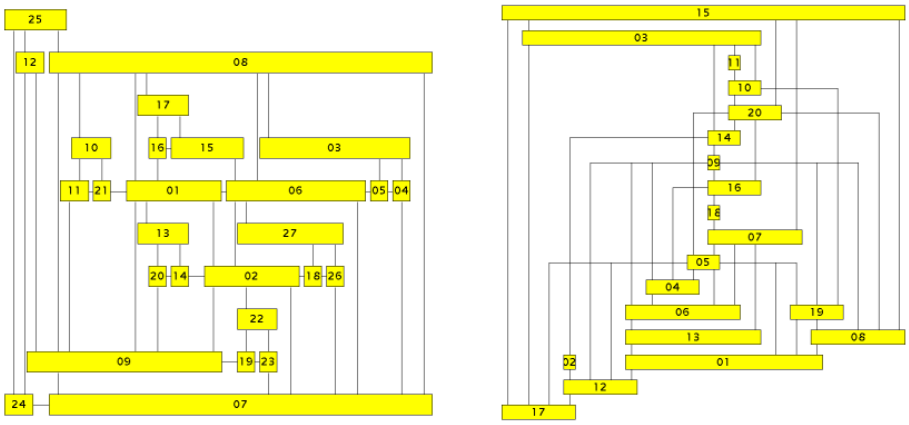
(a) Tutte drawing (b) Spring embedder

Rysunek 11.1. Różne reprezentacje tego samego grafu.

Innym znanym narzędziem jest komercyjna biblioteka LEDA [25], która oprócz algorytmów grafowych oferuje struftury danych i algorytmy związane z geometrią 2D i 3D.

11.1. Visibility representation

Visibility representation (w skrócie VR). VR grafu planarnego G to narysowanie tegoż grafu tak, że jego wierzchołki są reprezentowane przez nienakładające się poziome segmenty, natomiast krawędzie G to linie (pionowe, poziome lub łamane) łączące te segmenty.

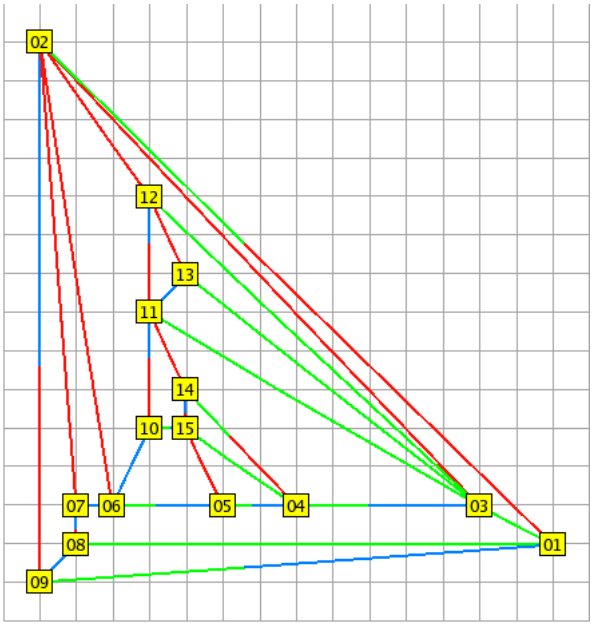


(a) Planar graph (b) Non-planar graph

Rysunek 11.2. Rysunek na bazie visibility representations.

11.2. Convex drawing

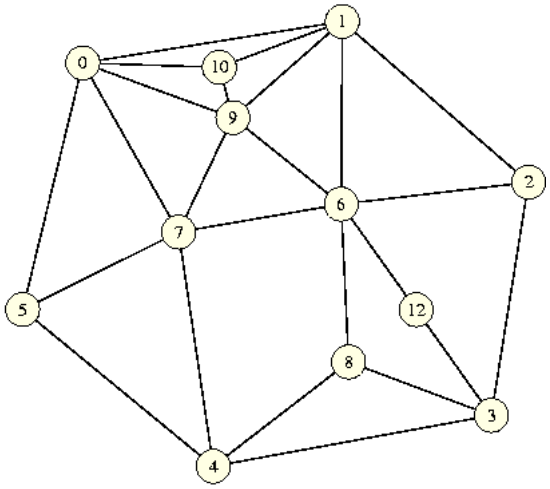
Wielokąt jest skończonym zbiorem odcinków, takim że punkt końcowy każdego z nich jest wspólny dla dokładnie dwóch odcinków. Wielokąt jest prosty, jeśli dwa jego boki mają punkt wspólny tylko gdy są sąsiadami (czyli nie występują przecięcia). Wielokąt prosty jest *wielokątem wypukłym*, gdy wszystkie kąty tego wielokąta są wypukłe (czyli mniejsze lub równe 180 stopniom). To, że wielokąt jest ściśle wypukły, oznacza że kąt 180 stopni nie jest dozwolony. *Convex drawing* grafu planarnego G to narysowanie go za pomocą linii prostych w taki sposób, że wszystkie ściany są wielokątami wypukłymi.



Rysunek 11.3. Convex drawing.

11.3. Tutte drawing

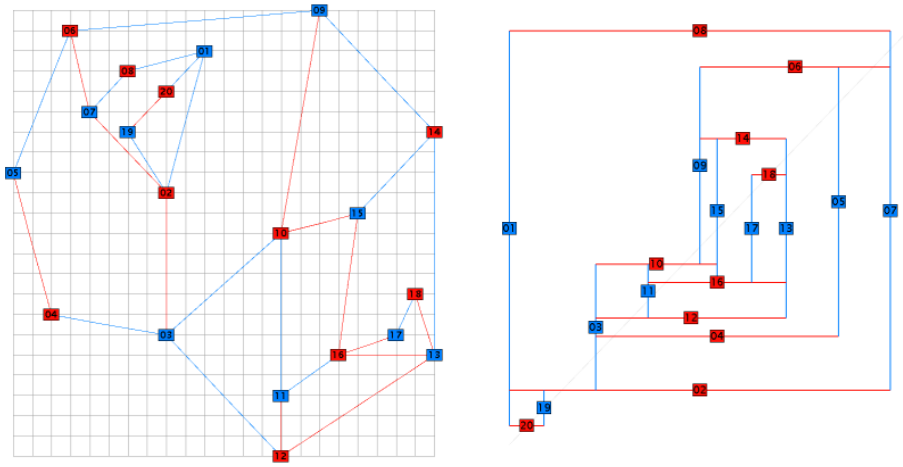
Tutte drawing lub *barycentric embedding* prostego 3-spójnego grafu planarnego (k-spójność grafu spójnego oznacza, że usunięcie mniej niż k dowolnych jego wierzchołków nie spowoduje jego rozspojenia) to metoda, w której krawędzie są nieprzecinającymi się liniami prostymi. Dodatkowo reprezentacja ta posiada taką właściwość, że ściana zewnętrzna jest wielokątem wypukłym, a każdy wewnętrzny wierzchołek znajduje się w punkcie środka ciężkości pozycji jego sąsiadów.



Rysunek 11.4. Tutte drawing.

11.4. Contacts of segments

Jest to reprezentacja planarnego grafu dwudzielnego, w której wierzchołki są przedstawiane jako pionowe i poziome segmenty, a krawędzie odpowiadają za połączenie między segmentami.



(a) (b)

Rysunek 11.5. Representation of a bipartite planar graph by contacts of segments.

12. Podsumowanie

W niniejszej pracy przedstawiono różne koncepcje graficznego przedstawienia grafów. Do ich implementacji użyto języka Python, ze względu na jego prostotę, czytelność kodu oraz wszechstronne zastosowanie, a także dostępność niezbędnych struktur danych. Całą pracę można podzielić na część teoretyczną i praktyczną.

W części teoretycznej przedstawiono pokrótce dostępne obecnie rozwiązania, stosowane przy rysowaniu grafów, wraz z listingami generującymi przykładowe grafy. Następnie opisano najważniejsze pojęcia z teorii grafów, związane z tematem ich wizualizacji. Dołączono również przykładowe sesje interaktywne tworzące przykładowe grafy i algorytmy, korzystające w implementacji istniejącej w Instytucie Fizyki Uniwersytetu Jagiellońskiego w Krakowie. W pracy znalazł się także rozdział poświęcony drzewom, kładący nacisk na różne ich reprezentacje graficzne oraz zawierający algorytmy pozwalające w czytelny sposób je narysować.

Na część praktyczną składają się galerie rozmaitych typów grafów posiadających ciekawe własności oraz reprezentacje graficzne. Są to więc grafy pełne, kubiczne, regularne stopnia 4 oraz grafy ze ścianami kwadratowymi. Wszystkie rysunki zostały wygenerowane przez skrypty napisane na potrzeby tej pracy. Na końcu znajduje się rozdział prezentujący przykładowe inne sposoby reprezentacji grafów, tworzone za pomocą specjalnych algorytmów.

Z pewnością istniejące już biblioteki, służące do generowania i wizualizacji grafów dają dużo szersze możliwości, niż przykładowe skrypty stworzone na potrzeby tej pracy. Nasz kod ma jednak tę przewagę, iż napisany jest w sposób prosty i czytelny - zrozumiały nawet dla początkujących programistów, którzy dopiero zapoznają się z tematyką algorytmów i struktur danych.

A. Kod źródłowy

W dziedzinie rysowania grafów można spotkać trzy pojęcia, które warto w tym miejscu zdefiniować, ponieważ będą dalej stosowane.

Graf abstrakcyjny to graf dany przez zbiór wierzchołków V i zbiór krawedzi E. W naszej implementacji odpowiada mu instancja klasy Graph.

Graf topologiczny to graf płaski, dla którego już określono ściany i dla każdego wierzchołka ustalona jest kolejność krawędzi wychodzących. Wyznaczenie grafu topologicznego dla danego grafu abstrakcyjnego to trudny problem nazywany testowaniem planarności. Obecnie znanych jest kilka algorytmów testujących planarność o złożoności liniowej.

Graf geometryczny to graf rozumiany jako rysunek, gdzie wierzchołkom przyporządkowane są punkty na płaszczyźnie, a krawędzie to odcinki, łamane lub łuki. Elementom grafu mogą być przyporządkowane różne atrybuty, takie jak kolor, etykiety, itp.

W naszej pracy zajmujemy sie wyznaczeniem grafu geometrycznego wprost z grafu abstrakcyjnego, oczywiście dla wybranych rodzin grafów.

A.1. Struktura skryptu rysującego grafy

Opiszemy ogólną budowę skryptu rysującego grafy, który wykorzystuje jedynie jeden słownik D do opisu grafu geometrycznego. Skrypt składa się z trzech części, każda ma inne zadania do wykonania.

A.1.1. Przygotowanie grafu abstrakcyjnego

Pierwsza część skryptu przygotowuje graf abstrakcyjny, instancję klasy Graph. Można ręcznie podać wierzchołki lub krawędzie, można skorzystać z wybranego generatora grafów, można wreszcze wczytać z pliku pickle wcześniej utworzony graf. Listing A.1 przedstawia tworzenie grafu przypadkowego z generatora.

Listing A.1. Tworzenie grafu abstrakcyjnego z generatora.

```
from graphs import Graph
from edges import Edge
from factory import GraphFactory

v = 10                  # liczba wierzcholkow
graph_factory = GraphFactory(Graph)
graph = graph_factory.make_random(v)
#graph = graph_factory.make_tree(v)    # drzewo przypadkowe
#graph = graph_factory.make_complete(v)    # graf pelny
#graph = graph_factory.make_cyclic(v)    # graf cykliczny
e = graph.e()           # liczba krawedzi
```

```
# Zmienne graph, v, e beda wykorzystywane.
```

A.1.2. Przygotowanie grafu geometrycznego

W drugiej części skryptu tworzony jest graf geometryczny, a w szczególności powstaje słownik D, którego kluczami są wierzchołki grafu, a wartościami punkty na płaszczyźnie, reprezentowane przez instancje klasy Point. Listing A.2 prezentuje umieszczanie wierzchołków grafu na okręgu. Kolejność wierzchołków może być przypadkowa lub odpowiadać cyklowi Hamiltona, czy kolejności sortowania topologicznego dla dagu.

Listing A.2. Utworzenie grafu geometrycznego.

```
import math
import random
from points import Point

node_list = list(graph.iternodes())
#random.shuffle(node_list)   # do kolejnosci przypadkowej
D = dict()
r = 0.5 * v   # przykladowy promien okregu
for (i, node) in enumerate(node_list):
    # i okresla polozenie na okregu,
    # node to jest wierzcholek w polozeniu i.
    D[node] = Point(r * math.cos(i * 2.0 * math.pi / v),
        r * math.sin(i * 2.0 * math.pi / v))
```

A.1.3. Eksport rysunku do pliku

W trzeciej części skryptu tworzony jest rysunek w programie Gnuplot. Krawędzie są odczytywane z grafu abstrakcyjnego, a następnie rysowane w postaci odcinków. Etykiety wierzchołków są napisową reprezentacją wierzchołków. Rysunek jest zapisywany w pliku PDF. Listing A.3 pokazuje korzystanie z modułu Gnuplot.

Listing A.3. Eksport rysunku grafu do pliku PDF.

```
import Gnuplot

gnu = Gnuplot.Gnuplot(persist = 1)
a = 1.1              # skala dla etykiet
begin = -a * r       # zakres osi
end = a * r          # zakres osi

# Rysowanie krawedzi.
for edge in graph.iteredges():
    # graf nieskierowany: nohead
    # graf skierowany: head filled
    gnu('set arrow from %s,%s to %s,%s nohead' % (
        D[edge.source].x, D[edge.source].y,
        D[edge.target].x, D[edge.target].y))

# Rysowanie wierzcholkow z etykietami.
for node in D:
```

```
    x = a * D[node].x
    y = a * D[node].y
    gnu('set label "%s" at %s,%s center' % (node, x, y))
    gnu('set label "{/Symbol \267}" at %s,%s center' % (
        D[node].x, D[node].y))

gnu('set terminal pdf enhanced')
gnu('set output "figure.pdf"')   # nazwa pliku z rysunkiem
gnu('unset key')
gnu('set size square')
gnu('unset border')
gnu('unset tics')
gnu('set xlabel ""')
gnu('set ylabel ""')
gnu('set title "Graph with V=%s, E=%s"' % (v, e))
gnu('set xrange [%f:%f]' % (begin, end))
gnu('set yrange [%f:%f]' % (begin, end))
gnu.plot('x+20 lc rgb "white" title ""')
gnu('unset output')
```

A.2. Skrypt do rysowania drzew

Skrypt do rysowania drzew ma budowę taką jak skrypt ogólny opisany w dodatku A.1, ale część druga jest inna. Wykorzystywane są dwa algorytmy związane z drzewami. Pierwszy algorytm oblicza centrum i promień drzewa. Drugi algorytm wyznacza położenia wierzchołków drzewa na płaszczyźnie, które są zapisywane w słowniku D grafu geometrycznego. Etykiety wierzchołków nie są rysowane.

Listing A.4. Graf geometryczny dla drzew.

```
from points import Point
from treecenter import TreeCenter
from treeplot import TreePlot

# Wyznaczam srodek drzewa.
algorithm = TreeCenter(graph)
algorithm.run()
# Jezeli centrum ma dwa wierzcholki, to wybieram bardziej rozgaleziony.
root = max(algorithm.tree_center, key=graph.degree)
radius = algorithm.tree_radius
begin = -radius      # zakres osi
end = radius         # zakres osi

# Wyznaczam slownik z punktami.
algorithm = TreePlot(graph)
algorithm.run(root)
D = algorithm.points
```

A.3. Skrypt do rysowania dagów

Skrypty do rysowania dagów mają również budowę taką jak skrypt ogólny, ale część druga jest zmieniona. Mamy dwa sposoby rysowania dagów.

Pierwszy sposób polega na umieszczaniu wierzchołków na okręgu w kolejności sortowania topologicznego. Tutaj postępujemy prawie tak jak w ogólnym skrypcie, jedynie krawędzie są skierowane. Drugi sposób polega na umieszczaniu wierzchołków w kolumnach odpowiadających cyklom sortowania topologicznego, w których wierzchołki były usuwane z dagu. Ten sposób został przedstawiony na listingu A.5.

Listing A.5. Graf geometryczny dla dagów.

```
from points import Point
from topsort import TopologicalSortList

algorithm = TopologicalSortList(graph)
algorithm.run()

# Ile wierzcholkow bylo w danym cyklu?
adict = dict()
for node in algorithm.cycle:
    c = algorithm.cycle[node]
    adict[c] = adict.get(c, 0) + 1
max_x = len(adict)   # liczba cykli
max_y = max(adict[c] for c in adict)  # najwiecej wierzcholkow
begin = 0                       # zakres osi
end = max(max_x, max_y)         # zakres osi

D = dict()
x = -1
y = None
cycle = -1
for node in algorithm.sorted_nodes:
    if algorithm.cycle[node] > cycle:    # zaczynamy nowy cykl
        cycle = algorithm.cycle[node]
        x += 1    # nowa kolumna
        y = 0
    else:
        y += 1    # wyzej w tej samej kolumnie
    D[node] = Point(x, y)
```

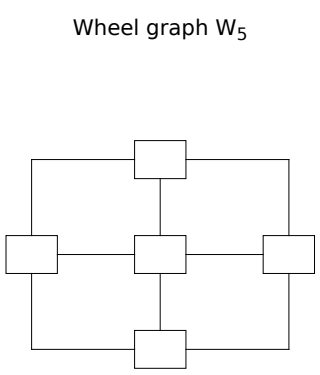
A.4. Skrypt do rysowania prostokątów i łamanych

W pracy podjęto próbę ustalenia struktury danych, która umożliwiałaby rysowanie grafu w taki sposób, aby wierzchołki były prostokątami, a krawędzie łamanymi ze strzałką lub bez (fragmenty pionowe i poziome). Strukturę danych składającą się z dwóch słowników przedstawia listing A.6. Przykładowe grafy znajdują się na rysunkach od A.4 do A.4. Rozmiar wierzchołków rośnie z jego stopniem.

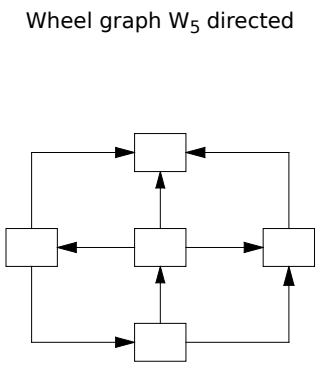
Listing A.6. Rysowanie prostokątów i łamanych.

```
# Slownik dla wierzcholkow (prostokaty).
D = dict()
D[node1] = [point1, point2]    # lewy dolny i prawy gorny rog
D[node2] = [point3, point4]
# Slownik dla krawedzi (odcinki i lamane).
```

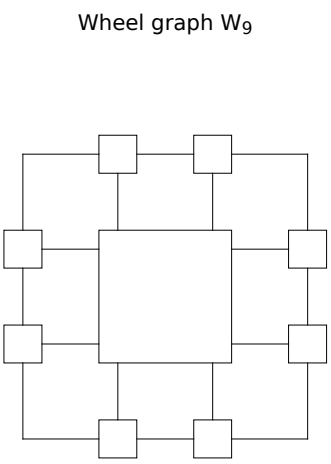
```
M = dict()
M[edge1] = [point5, point6]    # odcinek
M[edge2] = [point7, point8, point9]    # lamana
```



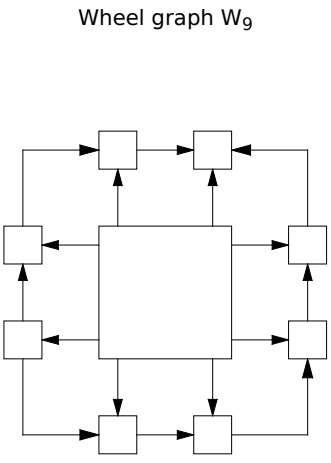
Rysunek A.1. Graf koło W_5.



Rysunek A.2. Graf skierowany na bazie W_5.



Rysunek A.3. Graf koło W_9.



Rysunek A.4. Graf skierowany na bazie W_9.

B. Testy wybranych algorytmów

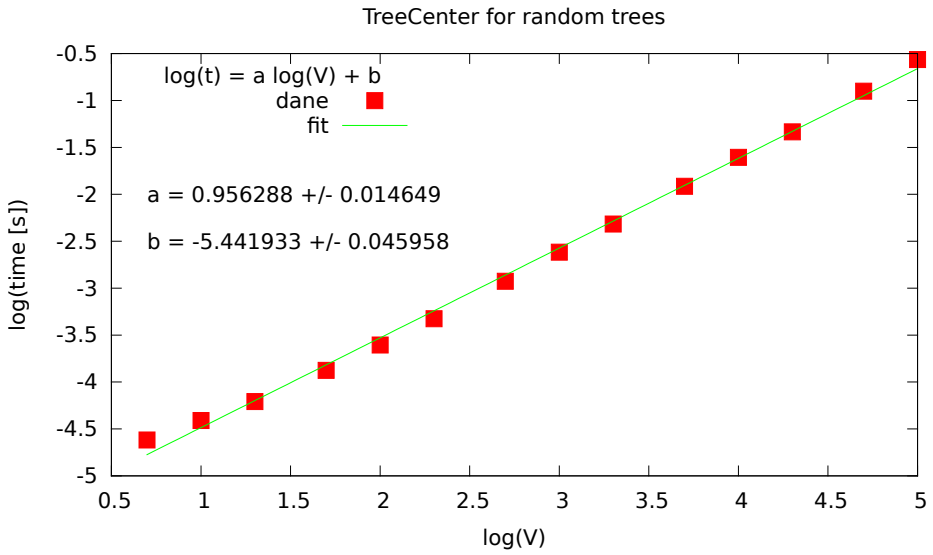
W tym dodatku przestawimy wyniki testów wydajnościowych dla dwóch podstawowych algorytmów stworzonych w pracy.

B.1. Testy wyznaczania centrum drzewa

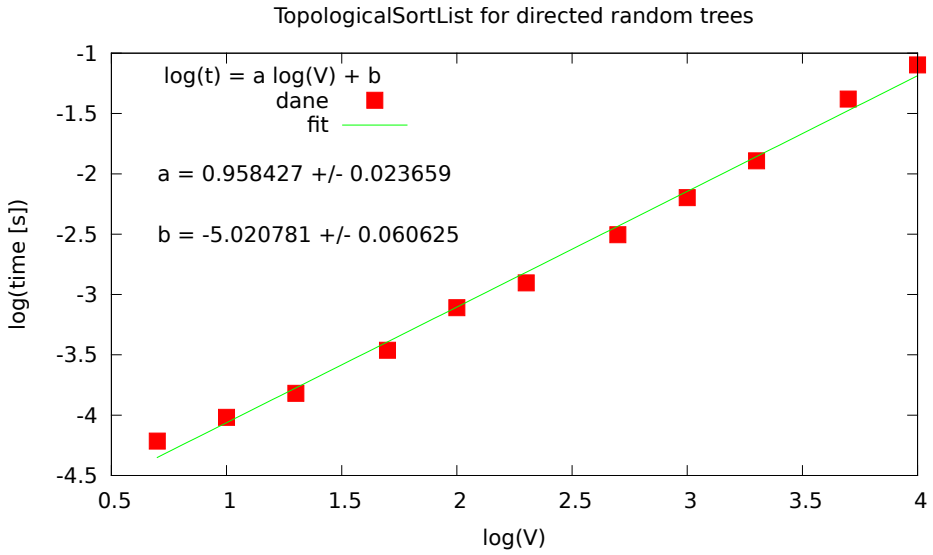
Przeprowadzono testy sprawdzające złożoność obliczeniową algorytmu wyznaczania centrum drzewa, zawartego w klasie TreeCenter. Wykorzystano generator drzew przypadkowych. Wyniki przedstawia rysunek B.1.

B.2. Testy sortowania topologicznego

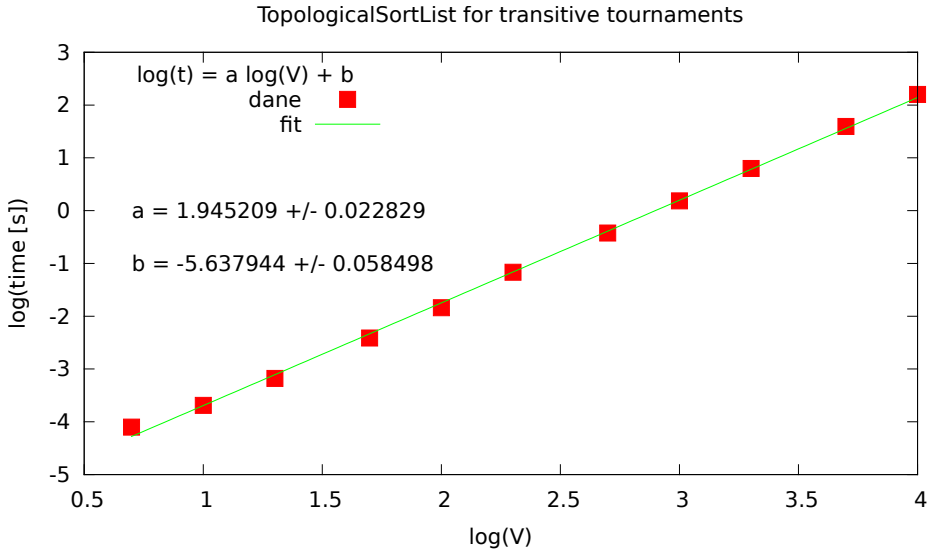
Przeprowadzono testy sprawdzające złożoność obliczeniową zmodyfikowanego algorytmu sortowania topologicznego, zawartego w klasie TopologicalSortList. Sprawdzono działanie implementacji na dwóch rodzinach grafów: drzewach przypadkowych skierowanych i turniejach tranzytywnych (grafy pełne skierowane acykliczne). Wyniki przedstawiają rysunki od B.2 do B.4.



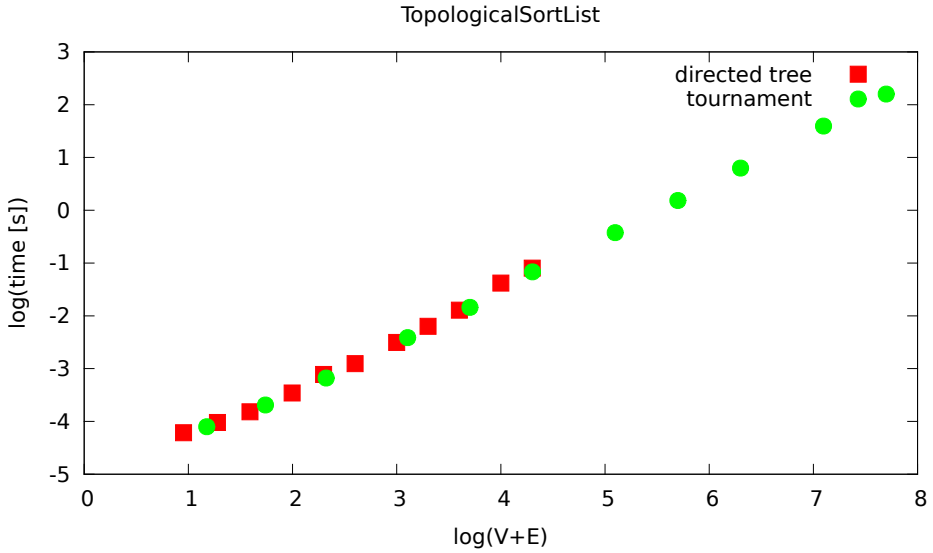
Rysunek B.1. Wykres wydajności algorytmu wyznaczania centrum drzewa. Współczynnik a bliski 1 potwierdza zależność $O(V)$.



Rysunek B.2. Wykres wydajności algorytmu sortowania topologicznego dla drzew przypadkowych skierowanych (drzewa zstępujące). Współczynnik a bliski 1 potwierdza zależność $O(V)$.



Rysunek B.3. Wykres wydajności algorytmu sortowania topologicznego dla turniejów tranzytywnych. Współczynnik a bliski 2 potwierdza zależność $O(V^2)$.



Rysunek B.4. Wykres wydajności algorytmu sortowania topologicznego dla dwóch rodzin grafów skierowanych acyklicznych. Ułożenie punktów wzdłuż jednej prostej, łącznie z poprzednimi dopasowaniami, potwierdza zależność liniową $O(V + E)$.

Bibliografia

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

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

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

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

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

[6] Wikipedia, Graf dwudzielny, 2014, `http://pl.wikipedia.org/wiki/Graf_dwudzielny`.

[7] Sourceforge.net, Gnuplot.py, 2015, `http://gnuplot-py.sourceforge.net`.

[8] A. Hagberg, D. A. Schult, and P. J. Swart, NetworkX, 2014, `http://networkx.github.io/`.

[9] J. D. Hunter, matplotlib, 2015, `http://matplotlib.org/`.

[10] igraph - The network analysis package, 2015, `http://igraph.org/python/`.

[11] pySVG - creating svg with python, 2015, `http://codeboje.de/pysvg/`.

[12] SVGFig, GitHub repository, 2015, `https://github.com/jpivarski/svgfig`.

[13] Graphviz - Graph Visualization Software, 2015, `http://www.graphviz.org/`.

[14] Wikipedia, Teoria grafów, 2015, `http://pl.wikipedia.org/wiki/Teoria_grafów`

[15] Wikipedia, Graf regularny, 2015, `http://pl.wikipedia.org/wiki/Graf_regularny`

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

[17] Wikipedia, Skierowany graf acykliczny, 2015, `https://pl.wikipedia.org/wiki/Skierowany_graf_acykliczny`.

[18] Wikipedia, Sortowanie topologiczne, 2015, `https://pl.wikipedia.org/wiki/Sortowanie_topologiczne`.

[19] Wikipedia, Cubic graph, 2015, `http://en.wikipedia.org/wiki/Cubic_graph`.

[20] N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences, A002851, Number of unlabeled trivalent (or cubic) connected graphs with 2n nodes, 2015, `http://oeis.org/A002851`.

[21] M. Meringer, *Fast Generation of Regular Graphs and Construction of Cages*,

Journal of Graph Theory 30, 137-146 (1999),
http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html.
[22] Wikipedia, Quartic graph, 2015,
http://en.wikipedia.org/wiki/Quartic_graph.
[23] Handbook of Graph Drawing and Visualization, 2013,
http://cs.brown.edu/~rt/gdhandbook/.
[24] PIGALE Library, 2012,
http://pigale.sourceforge.net/.
[25] Algorithmic Solutions Software GMBH, LEDA C++ library, 2015,
http://www.algorithmic-solutions.com/leda/index.htm.