Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Krzysztof Niedzielski

Nr albumu: 1089346

Skojarzenia w teorii grafów

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2017

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

Serdecznie dziękuję Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za poświęcony czas, zaangażowanie, cierpliwość, uwagi oraz ogromne wsparcie merytoryczne, które umożliwiły powstanie tej pracy.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów związanych ze skojarzeniami. Skojarzenie to podzbiór krawędzi w grafie nieskierowanym, dla którego co najwyżej jedna krawędź łączy się z którymkolwiek wierzchołkiem grafu. Odnalezienie największego skojarzenia w grafie stanowi ważny krok w wielu algorytmach aproksymacyjnych, np. dla problemu chińskiego listonosza, czy problemu komiwojażera. Dla pewnych typów grafów znajomość największego skojarzenia pozwala rozwiązywać inne trudne problemy w czasie wielomianowym, np. problem minimalnego pokrycia wierzchołkowego.

Zaimplementowano algorytmy pozwalające na odnajdywanie skojarzenia największego w różnych rodzajach grafów. Zastosowano technikę programowania dynamicznego do stworzenia algorytmu znajdującego największe skojarzenie dla drzew. Wyszukiwanie skojarzenia największego w grafach dwudzielnych można sprowadzić do problemu wyznaczania maksymalnego przepływu w sieci przepływowej. W związku z tym zaimplementowano kilka algorytmów związanych z tą tematyką. Zaprezentowano algorytm Edmondsa-Karpa, algorytmu Dynica i rekurencyjną implementację metody Forda-Fulkersona.

Kluczowym algorytmem pozwalającym na znalezienie skojarzenia największego w dowolnym grafie nieskierowanym jest algorytm kwiatowy Edmondsa. Jego wynalezienie udowodniło, że największe skojarzenie można znaleźć w czasie wielomianowym, co dało początek nowym metodom w teorii grafów. W pracy przedstawiono implementację algorytmu Edmondsa wraz ze szczegółowym opisem.

Skojarzenia w grafach dwudzielnych ważonych są związane z problemem przyporządkowania, stanowiącym ważne zagadnienie optymalizacji kombinatorycznej. Przedstawiono implementację algorytmu węgierskiego znajdującego skojarzenie o najmniejszej wadze. Dla przypadku ogólnych grafów ważonych pokazano algorytm zachłanny znajdujący przybliżone rozwiązanie tego problemu.

Słowa kluczowe: grafy, grafy dwudzielne, skojarzenia, algorytm kwiatowy, algorytm węgierski, sieci przepływowe, maksymalny przepływ, skojarzenie maksymalne, skojarzenie największe, skojarzenie o najmniejszej wadze

English title: Matching in graph theory

Abstract

Python implementation of selected graph algorithms for matchings is presented. Matching in an undirected graph is a subset of edges in which at most one edge is incident to any graph vertex. Finding a maximum matching in graph is an important step in many approximation algorithms (Chinese postman problem and travelling salesman problem among others) and for some types of graphs it can help in solving other difficult problems in polynomial time, for example in finding minimal vertex cover.

Algorithms for finding a maximum matching in various graphs are implemented. Dynamic programming is used to find a maximum matching in trees. Finding a maximum matching in the bipartite graphs leads to the maximum flow problem in flow networks. For this reason, a few algorithms concerning this problem are shown: Ford-Fulkerson algorithm (a recursive version), Edmonds-Karp algorithm, and Dinic algorithm.

The most important algorithm that allows to find a maximum matching in general graphs is the Edmonds blossom algorithm. Its implemetation is described in detail, because various methods are based on it. Its invention proved for the first time that a maximum matching could be found in polynomial time.

Matching in weighted bipartite graphs is related to the assignment problem, which is an important subject in the realm of combinatorial optimization. Hungarian algorithm used to solve this problem is shown. In the case of general weighted graphs, a greedy algorithm for finding a minimum weight matching is discussed.

Keywords: graphs, bipartite graphs, matching, blossom algorithm, hungarian algorithm, flow networks, maximum flow, maximum matching, maximal matching, minimum weighted matching

Spis treści

Spis rysunków

Spis tabel

Listings

1. Wstęp

Tematem niniejszej pracy są skojarzenia w teorii grafów. *Skojarzenie* M jest to podzbiór zbioru krawędzi grafu nieskierowanego taki, że żadne dwie krawędzie w M nie mają wspólnego wierzchołka [1]. Zwykle interesuje nas znalezienie skojarzenia o największej liczności (skojarzenie największe). Jeżeli krawędzie grafu mają określone wagi (graf ważony), to z wielu możliwych skojarzeń o największej liczności chcemy wybrać skojarzenie o najmniejszej (lub największej) wadze.

Skojarzenia mają wiele praktycznych zastosowań, które zostaną podane w niniejszej pracy. Dwa problemy opisane powyżej zwykle rozbija się na cztery problemy o rosnącym poziomie trudności, gdzie wyróżnia się grafy dwudzielne. W grafach dwudzielnych zbiór wierzchołków można podzielić na dwa niepuste i rozłączne zbiory A i B, przy czym wszystkie krawędzie grafu dwudzielnego mają jeden koniec w A, a drugi w B. Lista problemów jest następująca:

1. Największe skojarzenie w grafie dwudzielnym.
2. Największe skojarzenie w grafie ogólnym.
3. Skojarzenie o najmniejszej wadze w grafie dwudzielnym ważonym.
4. Skojarzenie o najmniejszej wadze w grafie ogólnym ważonym.

Pierwsze algorytmy działające w czasie wielomianowym w przypadku grafów ogólnych podał Jack Edmonds [2]. Wprowadził on koncepcję kwiatów, które grają kluczową rolę we wszystkich algorytmach dla ogólnych grafów. Algorytm Edmondsa i jego ulepszenia to najważniejszy motyw naszej pracy. Algorytm Edmondsa był przełomowy w teorii grafów z wielu względów. Po pierwsze był to pierwszy algorytm znajdujący największe skojarzenie w czasie wielomianowym. Po drugie, w dziedzinie programowania liniowego prowadził do pojęcia wielokomórki (wielotopu) skojarzenia (ang. *matching polytope*) i algorytmu na skojarzenie o najmniejszej wadze. Po trzecie, opis tej wielokomórki był przełomem w kombinatoryce wielokomórkowej (ang. *polyhedral combinatorics*) [2].

Naszym celem jest przedstawienie implementacji w języku Python [3] algorytmów dotyczących skojarzeń, w tym słynnego algorytmu Edmondsa.

Zagadnienia opisane w pracy należą do *optymalizacji kombinatorycznej* (ang. *combinatorial optimalization*) lub inaczej optymalizacji dyskretnej. Dziedzina ta omawia problemy optymalizacyjne, które można opisać używając grafów, sieci przepływowych, matroidów. Dobrymi źródłami informacji na temat problemów związanych ze skojarzeniami i sieciami przepływowymi są książki Lovásza i Plummera [4], Lawlera [5], Papadimitriou i Steiglitza [6], Ahuji, Magnantiego i Orlina [7]. Z literatury polskiej możemy podać jedynie pozycje poświęcone w ogólności problemom grafowym: [9], [10], [11], [8], [12].

Treść niniejszej pracy jest zorganizowana następująco. Rozdział 1 omawia cele pracy i podział problemów związanych ze skojarzeniami. W rozdziale 2 opisano najważniejsze pojęcia związane z teorią grafów i skojarzeniami. Rozdział 3 prezentuje bibliotekę grafową wykorzystaną i rozwijaną w pracy. W rozdziale 4 zawarto opisy oraz implementacje algorytmów służących do wyszukiwania skojarzeń w grafach. Rozdział 5 stanowi podsumowanie pracy. W dodatku A załączono wyniki testów dla wybranych implementacji algorytmów.

Implementacje algorytmów zostały zrealizowane i przetestowane przy użyciu języka Python w wersji 2.7.

2. Teoria grafów

W tym rozdziale zostaną przedstawione podstawowe definicje i twierdzenia z teorii grafów. W literaturze występują pewne różnice w definicjach, dlatego zawsze trzeba sprawdzić jak dany autor definiuje używane pojęcia. W naszej pracy korzystamy z definicji podanych w pracy Sampaio [13].

2.1. Podstawowe definicje

Graf nieskierowany jest to para uporządkowana $G = (V, E)$, gdzie $V(G)$ to zbiór wierzchołków, a $E(G) \subset [V]^2$ to zbiór krawędzi nieskierowanych, przy czym $[V]^2$ oznacza wszystkie dwuelementowe podzbiory $V(G)$. Oznaczamy liczbę wierzchołków przez $n = |V(G)|$, a liczbę krawędzi przez $m = |E(G)|$. Jeżeli $\{u, v\}$, w skrócie uv, jest krawędzią z $E(G)$, to wierzchołki u, v są sąsiednie i są końcami krawędzi uv.

Graf skierowany jest to para uporządkowana $G = (V, E)$, gdzie $V(G)$ to zbiór wierzchołków, a $E(G) \subset V \times V$ to zbiór krawędzi skierowanych. Jeżeli (u, v), w skrócie uv, jest krawędzią z $E(G)$, to wierzchołek u jest początkiem, a wierzchołek v (sąsiad u) końcem krawędzi skierowanej uv.

Zbiór sąsiadów wierzchołka v w grafie G oznaczamy przez $N(v)$. Notację rozszerzamy na zbiór $S \subseteq V(G)$, czyli $N(S)$ jest to suma wszystkich sąsiedztw wierzchołków ze zbioru S, bez wierzchołków ze zbioru S. Stopień wierzchołka v w grafie G to $\deg(v) = |N(v)|$. Najmniejszy stopień (ang. *minimum degree*) grafu G, oznaczany przez $\delta(G)$, jest to najmniejszy stopień wierzchołka w G. Największy stopień (ang. *maximum degree*) grafu G, oznaczany przez $\Delta(G)$, jest to największy stopień wierzchołka w G.

Dopełnienie (ang. *complement, inverse*) grafu G, oznaczane przez $\bar{G}$, jest to graf ze zbiorem wierzchołków $V(\bar{G}) = V(G)$, oraz zbiorem krawędzi $E(\bar{G}) = \{uv \in [V]^2 : uv \notin E(G)\}$.

Graf H jest *podgrafem* (ang. *subgraph*) grafu G, jeżeli zbiór wierzchołków $V(H) \subseteq V(G)$, oraz zbiór krawędzi $E(H) \subseteq \{uv \in E(G) : u, v \in V(H)\}$. Jeżeli $E(H) = \{uv \in E(G) : u, v \in V(H)\}$, to H nazywamy *podgrafem indukowanym* (ang. *induced subgraph*) grafu G. Podgraf grafu G indukowany przez zbiór $S \subset V(G)$ oznaczamy przez $G[S]$.

2.1.1. Spójność

Ścieżka (prosta) (ang. *path*) jest to graf P_n ze zbiorem wierzchołków $V(P_n) = \{v_1, \ldots, v_n\}$, oraz zbiorem krawędzi $E(P_n) = \{v_i v_{i+1} : 1 \leqslant i \leqslant n-1\}$. Długość ścieżki P_n to liczność zbioru krawędzi, czyli $m = n - 1$. Graf nieskierowany G jest *spójny* (ang. *connected*), jeżeli istnieje ścieżka pomiędzy

każdą parą różnych wierzchołków. *Składowa spójna* (ang. *connected component*) grafu nieskierowanego G jest to maksymalny spójny podgraf grafu G.

2.1.2. Cykle i drzewa

Cykl (prosty) (ang. *cycle*) jest to graf C_n ze zbiorem wierzchołków $V(C_n) = \{v_1, \ldots, v_n\}$, oraz zbiorem krawędzi $E(C_n) = \{v_i v_{i+1} : 1 \leqslant i \leqslant n-1\} \cup \{v_n v_1\}$. Długość cyklu C_n to liczność zbioru krawędzi, czyli $m = n$. Graf G jest *acykliczny* (ang. *acyclic*), jeżeli nie zawiera żadnego cyklu jako podgrafu. Graf acykliczny jest czasem nazywany *lasem* (ang. *forest*). *Drzewo* (ang. *tree*) jest to spójny graf acykliczny T. Dla drzewa zachodzi związek $m = n - 1$.

2.1.3. Kliki

Graf pełny (ang. *complete graph*) jest to graf nieskierowany K_n, w którym każda para wierzchołków jest połączona krawędzią. Dla grafu pełnego mamy związek $m = n(n-1)/2$. *Klika* (ang. *clique*) o rozmiarze k (k-klika) w grafie nieskierowanym G jest to zbiór k wierzchołków z $V(G)$, który indukuje podgraf pełny. Klika jest *maksymalna* (ang. *maximal*), jeżeli nie da się dodać do niej wierzchołka tak, aby powstała większa klika. Klika jest *największa* (ang. *maximum*), jeżeli nie ma w grafie większej kliki. Klika największa jest też maksymalna, ale implikacja w drugą stronę nie musi być prawdziwa. Rozmiar największej kliki w grafie G oznacza się przez $\omega(G)$ (ang. *clique number*).

2.1.4. Zbiory niezależne

Zbiór niezależny (ang. *stable set, independent set*) o rozmiarze k w grafie nieskierowanym G jest to zbiór k wierzchołków $S \subset V(G)$ taki, że każda krawędź z $E(G)$ może mieć najwyżej jeden koniec w zbiorze S [14]. Zbiór niezależny jest *maksymalny*, kiedy nie da się dodać do niego wierzchołka tak, aby powstał większy zbiór niezależny. Zbiór niezależny jest *największy*, kiedy nie ma w grafie większego zbioru niezależnego. Zbiór niezależny największy jest też maksymalny, ale implikacja w drugą stronę nie musi być prawdziwa. Rozmiar największego zbioru niezależnego w grafie G oznacza się przez $\alpha(G)$ (ang. *independence number*).

2.1.5. Dwudzielność

Graf dwudzielny (ang. *bipartite graph*) jest to graf nieskierowany G, w którym zbiór wierzchołków $V(G)$ może być podzielony na dwa niepuste zbiory niezależne, $V(G) = A \cup B$. Graf jest dwudzielny wtedy i tylko wtedy, gdy nie zawiera cykli nieparzystych [15]. *Graf pełny dwudzielny* (ang. *complete bipartite graph*) jest to graf dwudzielny $K_{p,q}$ ($p, q \geqslant 1$), w którym zbiór wierzchołków jest podzielony na dwie części o wielkościach p i q, oraz istnieje krawędź pomiędzy każdą parą wierzchołków z różnych części.

2.1.6. Skojarzenia

Skojarzenie (ang. *matching*) w grafie nieskierowanym G jest to zbiór krawędzi $M \subseteq E(G)$ taki, że każdy wierzchołek w $V(G)$ jest końcem co najwyżej

jednej krawędzi z M [1]. Skojarzenie jest *maksymalne*, kiedy nie da się dodać do niego krawędzi tak, aby powstało większe skojarzenie. Skojarzenie jest *największe*, kiedy nie ma w grafie większego skojarzenia. Skojarzenie największe jest też maksymalne, ale implikacja w drugą stronę nie musi być prawdziwa. Rozmiar największego skojarzenia w grafie G oznacza się przez $\mu(G)$ lub $\upsilon(G)$ (ang. *matching number*).

Wierzchołki z $V(G)$ będące końcami krawędzi należących do skojarzenia M są *nasycone/skojarzone* (ang. *saturated/matched*), natomiast pozostałe wierzchołki są *nienasycone/wolne* (ang. *unmatched/free*). Podobnie krawędź jest *skojarzona*, jeżeli należy do skojarzenia M, a w przeciwnym przypadku krawędź jest *wolna*.

Skojarzenie jest *doskonałe* (ang. *perfect matching*), kiedy każdy wierzchołek z $V(G)$ jest nasycony. Skojarzenie doskonałe może istnieć tylko w grafie z parzystą liczbą wierzchołków. Skojarzenie doskonałe jest najmniejszym pokryciem krawędziowym.

Ścieżka naprzemienna/alternująca ze względu na M (ang. *alternating path with respect to M*) jest to ścieżka prosta składająca się na przemian z krawędzi należących do zbioru $E(G) - M$ oraz M. Ścieżka naprzemienna jest *powiększająca ze względu na M* (ang. *augmenting path with respect to M*), jeżeli rozpoczyna się i kończy na wierzchołku wolnym. Ścieżka powiększająca ma długość nieparzystą, a w przypadku grafów dwudzielnych końce ścieżki powiększającej są w różnych zbiorach. Jeżeli w grafie mamy skojarzenie M i ścieżkę powiększającą P, to możemy skonstruować większe skojarzenie przez operację $M' = M \oplus P = (M - P) \cup (P - M)$, czyli zamieniając na ścieżce P krawędzie wolne na skojarzone i na odwrót. Nowe skojarzenie ma liczność $|M'| = |M| + 1$.

Twierdzenie (Berge, 1957): Skojarzenie M w grafie G jest największe wtedy i tylko wtedy, gdy w grafie G nie ma ścieżki powiększającej [10].

Stwierdzenie: Problem znajdowania skojarzenia doskonałego można wyrazić jako problem dokładnego pokrycia. Przestrzeń X odpowiada zbiorowi wierzchołków grafu, a rodzina podzbiorów tej przestrzeni to zbiory dwuelementowe, odpowiadające końcom każdej krawędzi. Niestety to spostrzeżenie niewiele pomaga w szukaniu wydajnego rozwiązania, ponieważ problem dokładnego pokrycia w ogólności jest NP-zupełny. Dla małych grafów można ewentualnie skorzystać z algorytmu X Knutha wykorzystującego *backtracking*.

2.1.7. Pokrycie wierzchołkowe

Pokrycie wierzchołkowe (ang. *vertex cover*) grafu nieskierowanego G jest to zbiór wierzchołków $C \subseteq V(G)$ taki, że każda krawędź z $E(G)$ ma co najmniej jeden koniec w zbiorze C [16]. Trywialne pokrycie wierzchołkowe to cały zbiór $V(G)$. Pokrycie wierzchołkowe jest *najmniejsze* (ang. *minimum*), jeżeli w grafie nie ma pokrycia wierzchołkowego o mniejszej liczności. Rozmiar najmniejszego pokrycia wierzchołkowego w grafie G oznacza się przez $\tau(G)$ (ang. *vertex cover number*).

2.1.8. Pokrycie krawędziowe

Pokrycie krawędziowe (ang. *edge cover*) grafu nieskierowanego G jest to zbiór krawędzi $L \subseteq E(G)$ taki, że każdy wierzchołek z $V(G)$ jest końcem co najmniej jednej krawędzi w zbiorze L [17]. Trywialne pokrycie krawędziowe to cały zbiór $E(G)$. Pokrycie krawędziowe jest *najmniejsze* (ang. *minimum*), jeżeli w grafie nie ma pokrycia krawędziowego o mniejszej liczności. Rozmiar najmniejszego pokrycia krawędziowego w grafie G oznacza się przez $\rho(G)$ (ang. *edge cover number*).

2.2. Kilka ważnych twierdzeń

Identyczności Gallai: Dla dowolnego grafu G
(1) $\tau(G) + \alpha(G) = |V(G)|$,
(2) $\mu(G) + \rho(G) = |V(G)|$, jeżeli G nie ma wierzchołków stopnia 0.
Ad. (1). Z definicji zbioru niezależnego S wynika, że $V(G) - S$ jest pokryciem wierzchołkowym. Ad. (2). Dowód w pracy [13].

Twierdzenie Minimax Königa (1931): Jeżeli graf G jest dwudzielny, to wtedy $\tau(G) = \mu(G)$. Twierdzenie Königa jest centralnym twierdzeniem w teorii grafów. Jest szczególnym przypadkiem kilku innych twierdzeń [13].

Twierdzenie Halla (1935): W grafie dwudzielnym $G = (A \cup B, E)$ istnieje skojarzenie wysycające zbiór A wtedy i tylko wtedy, gdy dla każdego zbioru $X \subseteq A$ zachodzi $|X| \leqslant |N(X)|$.

Twierdzenie Frobeniusa (1917): W grafie dwudzielnym $G = (A \cup B, E)$ istnieje skojarzenie doskonałe wtedy i tylko wtedy, gdy $|A| = |B|$, oraz dla każdego zbioru $X \subseteq A$ zachodzi $|X| \leqslant |N(X)|$.

2.3. Sieci przepływowe i maksymalny przepływ

Teoria skojarzeń jest silnie powiązana z teorią sieci przepływowych, dlatego podamy najważniejsze fakty z tego ostatniego działu [8], [18].

Sieci przepływowe: *Sieć przepływowa* (ang. *flow network*) jest to graf skierowany $G = (V, E)$, w którym wszystkie krawędzie (u, v) należące do zbioru E posiadają nieujemną przepustowość $c(u, v) \geqslant 0$. W grafie nie występują krawędzie przeciwne [8]. Jeżeli para wierzchołków (u, v) nie należy do zbioru krawędzi to przyjmujemy, że $c(u, v) = 0$. W sieci przepływowej wyróżniamy dwa wierzchołki: źródło s i ujście t. Przyjmujemy, że każdy wierzchołek leży na pewnej ścieżce od źródła do ujścia.

Przepływ: Przepływem jest każda funkcja rzeczywista $f : V \times V \to R$ w sieci przepływowej G, spełniająca następujące warunki:

— Warunek przepustowości: Dla wszystkich $u, v \in V$ zachodzi $f(u,v) \leqslant c(u,v)$.
— Warunek skośnej symetryczności: Dla wszystkich $u, v \in V$ zachodzi $f(u,v) = -f(v,u)$.
— Warunek zachowania przepływu: Dla każdego $u \in V - \{s,t\}$ zachodzi $\sum_{v \in V} f(u,v) = 0$.

Wielkość $f(u,v)$, która może mieć dodatnią lub ujemną wartość, jest nazywana przepływem netto z wierzchołka u do wierzchołka v i definiujemy ją jako $|f| = \sum_{v \in V} f(s,v)$, co oznacza, że wartością przepływu jest łączny przepływ który opuszcza źródło. W problemie *maksymalnego przepływu* mamy daną sieć przepływową G z źródłem s i ujściem t. Należy odnaleźć przepływ od źródła do ujścia posiadający maksymalną wartość.

Sieć residualna: *Przepustowością residualną* krawędzi w odniesieniu do przepływu f nazywamy różnicę pomiędzy przepustowością krawędzi oraz wartością przepływu: $c_f(u,v) = c(u,v) - f(u,v)$ [dla krawędzi przeciwnej $c_f(v,u) = 0 - f(v,u) = f(u,v)$]. Przy użyciu przepustowości residualnych krawędzi, dla grafu G można stworzyć *sieć residualną* indukowaną przez przepływ$f$: $E_f = \{(u,v) \in V \times V : c_f(u,v) \geqslant 0)\}$, która reprezentuje dostępną przepustowość na krawędziach w sieci przepływowej $G = (V,E)$. Sieć residualna może wykorzystywać krawędzie nie występujące w oryginalnej sieci (krawędzie przeciwne).

Ścieżka powiększająca: Jest to każda ścieżka ze źródła s do ujścia t w sieci residualnej G_f. Każda krawędź (u,v) na ścieżce powiększającej umożliwia dodatni przepływ netto pomiędzy wierzchołkami u i v bez naruszania przepustowości dla tej krawędzi, co wynika z definicji sieci residualnej. Przez sieć przepływa maksymalny przepływ wtedy i tylko wtedy, gdy w G_f nie istnieje żadna ścieżka powiększająca.

Sieć warstwowa: W sieci residualnej można zdefiniować funkcję $\mathrm{dist}(v)$ jako długość najkrótszej ścieżki od źródła s do wierzchołka v. Sieć warstwowa na bazie sieci residualnej to graf z krawędziami $(u,v) \in E_f$, dla których $\mathrm{dist}(v) = \mathrm{dist}(u) + 1$.

Przepływ blokujący: Przepływ w sieci warstwowej dla którego nie ma ścieżki powiększającej w tej sieci, nazywa się przepływem blokującym.

Metoda Forda-Fulkersona (1956): Jest to algorytm zachłanny, który wyznacza maksymalny przepływ poprzez znajdowanie ścieżek powiększających w sieci residualnej. Jest to właściwie metoda, a nie algorytm, ponieważ rozwiązanie Forda i Fulkersona jest podstawą kilku algorytmów o różnych czasach działania [19]. Jeżeli przepustowości krawędzi są całkowitoliczbowe, to czas działania metody wynosi $O(|f|E)$, gdzie $|f|$ jest wartością optymalnego przepływu. Jeżeli przepustowości krawędzi są liczbami niewymiernymi, to algorytm może działać w nieskończoność. Kilka implementacji algorytmu jest zawartych w module graphtheory.flow.fordfulkerson, gdzie do znajdowania

ścieżek powiększających wykorzystano DFS. W rozdziale 4 pokażemy nową implementację rekurencyjną, która pozwala lepiej zobaczyć podobieństwa i różnice w stosunku do szybszych metod.

Algorytm Edmondsa-Karpa (1972): Algorytm poprawia czas działania metody Forda-Fulkersona przez wybieranie ścieżek powiększających, które są najkrótszymi ścieżkami od źródła do ujścia w sieci residualnej (długość ścieżki jest mierzona liczbą krawędzi). Odpowiada to zastosowaniu BFS do znajdowania ścieżek powiększających. Algorytm działa w czasie $O(VE^2)$ [20]. Ciekawą własnością algorytmu jest fakt, że długości kolejnych znajdowanych ścieżek rosną monotonicznie. Implementacja algorytmu jest zawarta w module graphtheory.flow.edmondskarp. W rozdziale 4 pokażemy nową implementację algorytmu.

Algorytm Dynica (1970): Ten algorytm również korzysta z najkrótszych ścieżek powiększających do wyznaczenia maksymalnego przepływu, ale stosuje dodatkowo nowe techniki, takie jak sieć warstwowa (ang. *level graph*) i przepływ blokujący (ang. *blocking flow*). Algorytm działa w czasie $O(V^2E)$ [21]. Implementacja algorytmu Dinica jest przedstawiona w rozdziale 4. Warto wspomnieć, że algorytm został wynaleziony przez rosyjskiego (później izraelskiego) naukowca Chaima Dynica (ang. Yefim Dinic) w roku 1970, ale został spopularyzowany dopiero przez Shimona Evena i Alona Itai w wersji wykorzystującej BFS i DFS [21].

Algorytm trzech Hindusów (Malhotra, Kumar, Maheshwari, 1978): Algorytm MPM przypomina algorytm Dynica, ponieważ też korzysta z sieci warstwowej i przepływów blokujących. Pojawia się jeszcze pojęcie *przepustowości/potencjału wierzchołka* zdefiniowane następująco
$c(v) = \min\{\sum_{u \in V} c(u, v), \sum_{u \in V} c(v, u)\}$.
Potencjały wierzchołków są wykorzystywane do znajdowania przepływów blokujących. Dla ustalonej sieci warstwowej algorytm znajduje kolejne wierzchołki o najmniejszym potencjale, a następnie zwiększa przepływ od danego wierzchołka w kierunku ujścia, a także w drugą stronę w kierunku źródła. Wydajność algorytmu to $O(V^3)$ [22].

Algorytmy typu push-relabel: Metoda typu *prześlij-przemianuj* (ang. *push-relabel*) dostarcza wielu szybkich algorytmów obliczania maksymalnego przepływu [23]. Ogólny algorytm działa w czasie $O(V^2E)$ (Goldberg, 1986), jego modyfikacja *przemianuj i przesuń na początek* (ang. *relabel-to-front*) działa w czasie $O(V^3)$, rozwiązanie z wyborem najbardziej aktywnego wierzchołka $O(V^2\sqrt{E})$. Jeszcze szybsze są implementacje z dynamicznymi drzewami Sleatora-Tarjana. Algorytmy tego typu w czasie działania łamią warunek zachowania przepływu. Używa się pojęcia *przedprzepływu* (ang. *preflow*).

2.4. Skojarzenia w grafach dwudzielnych

Istnieje kilka metod znajdowania największego skojarzenia w grafach dwudzielnych. Niektóre z nich zostały już zaimplementowane w pakiecie graphtheory.

Na początek jednak podamy przykłady, w których problemy z życia wzięte można opisać używając skojarzeń z teorii grafów.

Przykład: W pewnym zakładzie pracy zatrudnieni są pracownicy ze zbioru A, oraz są przygotowane stanowiska pracy ze zbioru B. Nie każdy pracownik może pracować przy każdym stanowisku. Problem polega na optymalnym wykorzystaniu pracowników i obsadzeniu stanowisk pracy. W języku teorii grafów powiemy, że należy znaleźć skojarzenie największe w grafie dwudzielnym $G = (A \cup B, E)$, przy czym każda krawędź ze zbioru E łączy pracownika ze stanowiskiem pracy, na którym jest w stanie pracować [24].

Przykład: W sejmie istnieją różne komisje zajmujące się różnymi problemami. W skład komisji wchodzą posłowie. Każda z komisji ma swojego przewodniczącego i wygodnie jest gdy przewodniczącymi różnych komisji są różne osoby. Jak rozwiązać ten problem? Niech zbiór A zawiera wierzchołki odpowiadające różnym komisjom, a wierzchołki zbioru B odpowiadają wszystkim posłom. Krawędzie ze zbioru E łączą posłów ze zbioru B z komisjami ze zbioru A, do których należą. Powstaje graf dwudzielny $G = (A \cup B, E)$. Szukamy skojarzenia M wysycającego zbiór A, o którym mówi twierdzenie Halla. Wierzchołki nasycone ze zbioru B będą odpowiadać przewodniczącym komisji [24].

Opisany problem to inaczej wyznaczanie *zbioru różnych reprezentantów*. Chodzi o to, że dla zbioru posłów B mamy rodzinę jego podzbiorów (komisje sejmowe) i chcemy z tych podzbiorów wybrać różniących się reprezentantów.

Metoda Forda-Fulkersona: Metoda polega na stworzeniu sieci przepływowej z jednostkowymi wagami krawędzi. Odpowiednie twierdzenie gwarantuje znalezienie największego skojarzenia, które odpowiada maksymalnemu przepływowi w sieci [8]. Algorytm działa w czasie $O(VE)$ i jest zawarty w module graphtheory.bipartiteness.matching.

Metoda ścieżek powiększających: Metoda polega na stopniowym powiększaniu skojarzenia w grafie dwudzielnym $G = (A \cup B, E)$ przez wyszukiwanie ścieżek powiększających [18]. Dla usprawnienia wyszukiwania tworzy się pomocniczy graf skierowany, w którym krawędzie wolne są skierowane od A do B, a krawędzie skojarzone od B do A. Scieżka powiększająca to będzie ścieżka skierowana od wierzchołka wolnego w A do wierzchołka wolnego w B. Algorytm działa w czasie $O(VE)$, a jego implementacja jest podana w rozdziale 4.

Algorytm Hopcrofta-Karpa (1973): Algorytm również wykorzystuje ścieżki powiększające, ale zamiast jednej ścieżki na iterację znajduje maksymalny zbiór najkrótszych ścieżek powiększających [25]. Algorytm działa w czasie $O(\sqrt{V}E)$ i jest zawarty w module graphtheory.bipartiteness.hopcroftkarp.

2.5. Skojarzenia w grafach ogólnych

Możliwość odnajdywania skojarzenia największego w dowolnym grafie posiada praktyczne zastosowania. Do rozwiązywania tego problemu wykorzystywany jest przede wszystkim algorytm kwiatowy Edmondsa.

Przykład: Dawcy organów z powodu braku zgodności grup krwi bądź tkanek często nie są w stanie przekazać swoich narządów bliskim. Wprowadza się systemy umożliwiające parowanie rodzin dawców przy użyciu baz danych medycznych, tak by dawca z pierwszej rodziny mógł przekazać narząd osobie z drugiej rodziny i odwrotnie. Nie zawsze jest to jednak zadanie trywialne, gdyż wśród dawców i biorców mogą występować złożone zależności; przykładowo dawca z jednego małżeństwa może być w stanie przekazać narząd osobie z drugiej pary, partner tej osoby biorcy z pary trzeciej, zaś krewny tej osoby biorcy z pierwszego małżeństwa. [26]. Problem ten można opisać przy użyciu grafu, gdzie zbiór wierzchołków V reprezentuje różne pary, zaś zbiór krawędzi E możliwość wymiany narządów pomiędzy nimi. Odnalezienie skojarzenia największego pozwala znaleźć kombinację umożliwiającą dobranie do siebie największej liczby par.

Skojarzenia maksymalne: Maksymalne skojarzenie w dowolnym grafie można znaleźć za pomocą algorytmu zachłannego. Algorytm działa w czasie $O(V + E)$ i jest zawarty w module graphtheory.algorithms.matching.

Algorytm kwiatowy (Edmonds, 1961): Problem znajdowania największego skojarzenia w dowolnych grafach rozwiązuje algorytm Edmondsa, zwany algorytmem kwiatowym (ang. *blossom algorithm*) [2]. Algorytm został opublikowany w roku 1965 [27]. Naiwna implementacja algorytmu działa w czasie $O(V^4)$, bardziej dopracowane w czasie $O(V^2E)$, $O(V^3)$ lub szybciej.

2.6. Skojarzenia w grafach dwudzielnych ważonych

W przypadku grafów ważonych, z wielu możliwych skojarzeń największych chcemy wybrać skojarzenie o najmniejszej wadze, gdzie waga skojarzenia oznacza sumę wag krawędzi wchodzących w skład skojarzenia.

Algorytm węgierski (Kuhn, 1955; Munkres, 1957): Algorytm węgierski, zwany algorytmem Kuhna-Munkresa, dotyczy problemu przypisania (ang. *assignment problem*), ale rozwiązuje również problem skojarzenia o najmniejszej (lub największej) wadze w grafach dwudzielnych [28]. Wygodnie jest tutaj myśleć o grafie w reprezentacji macierzy sąsiedztwa. Złożoność oryginalnego algorytmu wynosi $O(V^4)$, a ulepszona wersja działa w czasie $O(V^3)$ (Tomizawa).

Problem przypisania można zdefiniować następująco. Mamy n zadań do wykonania i n maszyn, które mogą wykonywać zadania. Dane są koszty wykonania każdego zadania przez każdą maszynę (macierz $n \times n$). Należy przyporządkować maszyny do zadań tak, aby sumaryczny koszt był najmniejszy.

Warto zauważyć, że na problem przypisania można spojrzeć jak na problem znalezienia zbioru niezależnego o najmniejszej wadze w danej macierzy. Z macierzą kojarzymy graf, w którym wierzchołkami są pola macierzy. Dwa pola są połączone krawędzią, jeżeli znajdują się w tym samym wierszu lub tej samej kolumnie. Wartości liczbowe w polach macierzy są wagami wierzchołków.

2.7. Skojarzenia w grafach ogólnych ważonych

W problemie chińskiego listonosza pojawia się m.in. problem znalezienia skojarzenia doskonałego o najmniejszej wadze w grafie pełnym ważonym. Zauważmy, że liczba różnych skojarzeń doskonałych wynosi $(2n)!/n!2^n = (2n-1)!!$, gdzie n jest parzystą liczbą wierzchołków.

Do zastosowań odnajdywania minimalnego skojarzenia doskonałego należy także krok algorytmu aproksymacyjnego służącego do rozwiązywania problemu komiwojażera, tzw. algorytm Christofidesa[29]. Posiada on gwarancję na to, że znalezione rozwiązanie będzie nie gorsze niż 3/2 rozwiązania optymalnego, którego odnalezienie jest problemem NP-trudnym.

Algorytm zachłanny: Problem skojarzenia o najmniejszej wadze można rozwiązać w sposób przybliżony za pomocą algorytmu zachłannego. Do skojarzenia zaliczane są kolejne krawędzie o najmniejszej wadze, których oba końce są wolne. Algorytm działa w czasie $O(E \log V)$ i jest zawarty w module graphtheory.algorithms.matching. Krawędzie są porządkowane względem wag przy pomocy kolejki priorytetowej. Warto zauważyć, że jeżeli wagi krawędzi byłyby liczbami całkowitymi pochodzącymi z pewnego ograniczonego przedziału, to zastosowanie sortowania bukietowego pozwala uzyskać algorytm przybliżony działający w czasie liniowym $O(V + E)$.

Algorytm dokładny: Algorytm dokładny opisany jest w pracy Zvi Galil [30], a przykładowa implementacja znajduje się w bibliotece NetworkX [31], funkcja max_weight_matching(). Implementacja jest bardzo skomplikowana, bazuje na metodzie kwiatów Edmondsa przy wyszukiwaniu ścieżek powiększających i *primal-dual method* przy znajdowaniu skojarzenia o największej wadze.

3. Implementacja grafów

Algorytmy w tej pracy zaimplementowano przy użyciu biblioteki rozwijanej w Instytucie Fizyki UJ [32]. Dostarczane przez nią moduły pozwalają na reprezentację różnego rodzaju grafów. Zostaną tu omówione niektóre pojęcia i struktury danych wykorzystywane w algorytmach związanych ze skojarzeniami.

3.1. Grafowe struktury danych

Graf: Instancja klasy Graph. Struktura grafu przechowywana jest w postaci tablicy z haszowaniem - słownika języka Python. Atrybut directed określa czy graf jest skierowany. Jeżeli graf nie jest skierowany, to wewnątrz grafu dla każdej nowej krawędzi edge dodawana jest krawędź ~edge o przeciwnym zwrocie.

Algorytm: Klasa posiadająca co najmniej dwie metody run() oraz __init__, służące odpowiednio do uruchomienia algorytmu i przygotowania go do pracy poprzez określenie danych wejściowych. Zakończenie działania algorytmu wiąże się ze zmianą wewnętrznego stanu instancji tej klasy, wynik wykonanych operacji można sprawdzić odwołując się do jej atrybutów.

Wierzchołek (węzeł): Dowolny obiekt, który może służyć za klucz w tablicy z haszowaniem. Wymagana jest możliwość sortowania wierzchołków.

Krawędź: Instancja klasy Edge. Atrybuty source i target wskazują na wierzchołek początkowy i wierzchołek końcowy w krawędzi. Atrybut weight określa wagę krawędzi o domyślnej wartości 1. W przypadku wartości jednostkowej, przy wyświetlaniu reprezentacji obiektu jako łańcucha znaków waga jest pomijana.

Skojarzenie: Atrybut mate w klasach reprezentujących algorytmy do obliczania skojarzeń. Jest to słownik Pythona, w którym kluczami są wierzchołki grafu, a wartościami są inne węzły reprezentujące parę w skojarzeniu, bądź wartość None, jeśli wierzchołek nie ma pary. W algorytmach dla grafów ważonych słownik mate może przechowywać całą krawędź skierowaną do drugiego wierzchołka z pary. Dzięki temu łatwo można obliczyć wagę skojarzenia.

Przepływ: Słownik flow wykorzystywany w algorytmach operujących na sieciach przepływowych. Jego kluczami są wierzchołki stanowiące źródła krawędzi w sieci, zaś jego wartościami kolejne słowniki, których klucze reprezentują

węzły docelowe tych krawędzi. Dostęp do aktualnego przepływu przez krawędź można uzyskać poprzez notację flow[source][target]. Wartość maksymalnego przepływu przez sieć przepływową znajduje się w atrybucie max_flow.

3.2. Przykładowe sesje interaktywne

Przykład 1: Szukanie skojarzenia największego w grafie nieskierowanym przy użyciu algorytmu Edmondsa.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> graph_factory = GraphFactory(Graph)
>>> G = graph_factory.make_connected(200, False, 300)
>>> from blossom import MaximumMatchingEdmonds
>>> algorithm = MaximumMatchingEdmonds(G)
>>> algorithm.run()
>>> algorithm.mate   # wyznaczone skojarzenie
(dict)
```

Przykład 2: Wyznaczanie maksymalnego przepływu w sieci przepływowej (graf skierowany) z użyciem algorytmów Forda Fulkersona, Edmondsa-Karpa i Dinica.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> graph_factory = GraphFactory(Graph)
>>> V = 200
>>> G = graph_factory.make_flow_network(V)
>>> from flow1 import FordFulkersonRecursive
>>> from flow2 import EdmondsKarp
>>> from dinic1 import Dinic
>>> algorithm1 = FordFulkersonRecursive(G)
>>> algorithm1.run(0, V-1)
>>> algorithm1.max_flow   # maksymalny przeplyw przez siec
(number)
>>> algorithm1.flow   # przeplywy przez krawedzie
(dict)
>>> algorithm2 = EdmondsKarp(G)
>>> algorithm2.run(0, V-1)
>>> algorithm2.max_flow
(number)
>>> algorithm2.flow
(dict)
>>> algorithm3 = Dinic(G)
>>> algorithm3.run(0, V-1)
>>> algorithm3.max_flow
(number)
>>> algorithm3.flow
(dict)
```

Przykład 3: Wyznaczanie skojarzenia maksymalnego o najmniejszej wadze w grafie ważonym nieskierowanym (metoda zachłanna, rozwiązanie przybliżone).

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> graph_factory = GraphFactory(Graph)
>>> G = graph_factory.make_random(200, False)
>>> from matching import MinimumWeightMatchingWithEdges
>>> algorithm = MinimumWeightMatchingWithEdges(G)
>>> algorithm.run()
>>> algorithm.cardinality    # liczba krawedzi w skojarzeniu
(number)
>>> algorithm.mate    # wyznaczone skojarzenie
(dict)
# Obliczanie wagi skojarzenia.
>>> mate_weight = sum(algorithm.mate[node].weight
    for node in algorithm.mate if algorithm.mate[node]) / 2
```

4. Algorytmy

Przedstawione algorytmy służą do rozwiązywania problemu znajdowania skojarzenia największego w różnych rodzajach grafów. W przypadku grafów dwudzielnych zagadnienie to sprowadza się do znalezienia maksymalnego przepływu w sieci przepływowej, gdyż krawędzie w których odbywa się maksymalny przepływ odpowiadają krawędziom należącym do skojarzenia największego.

4.1. Metoda Forda-Fulkersona

W tym rozdziale pokażemy rekurencyjną implementację metody Forda-Fulkersona, która nie była wcześniej prezentowana. W nieskończonej pętli **while** znajdowane są ścieżki powiększające za pomocą DFS. Przeszukiwanie realizuje metoda _find_path_dfs(). Po dotarciu przeszukiwania do ujścia, jeżeli możliwy jest dodatkowy niezerowy przepływ, następuje powrót z wywołań rekurencyjnych przy jednoczesnym zapisywaniu tego dodatkowego przepływu w tablicy flow.

Złożoność: Szacowana złożoność obliczeniowa metody Forda-Fulkersona odpowiada $O(E|f|)$, gdzie f reprezentuje maksymalny przepływ w grafie. Odnalezienie nowej ścieżki powiększającej w czasie $O(E)$ powoduje poprawę przepływu o co najmniej jedną jednostkę, $|f|$ stanowi górne ograniczenie na liczbę koniecznych do wykonania iteracji[19].

Listing 4.1. Metoda Forda-Fulkersona.

```
#!/usr/bin/python

from edges import Edge

class FordFulkersonRecursive:
    """The Ford-Fulkerson algorithm for computing the maximum flow."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if not graph.is_directed():
            raise ValueError("the graph is not directed")
        self.graph = graph
        self.residual = self.graph.__class__(self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            self.residual.add_node(node)
        for edge in self.graph.iteredges():
            self.residual.add_edge(edge)
            self.residual.add_edge(Edge(edge.target, edge.source, 0))
        self.flow = dict()
```

```
        for source in self.graph.iternodes():
            self.flow[source] = dict()
            for target in self.graph.iternodes():
                self.flow[source][target] = 0
        self.max_flow = 0
        self.parent = None

    def run(self, source, sink):
        """Executable pseudocode."""
        if source == sink:
            raise ValueError("source and sink are the same")
        self.source = source
        self.sink = sink
        while True:
            self.parent = dict((node, None)
                for node in self.residual.iternodes())
            min_capacity = self._find_path_dfs(self.source, float("inf"))
            if min_capacity > 0:
                self.max_flow += min_capacity
            else:
                break

    def _find_path_dfs(self, source, start_capacity):
        """Finding augmenting paths in the residual network."""
        if source == self.sink:
            return start_capacity
        for edge in self.residual.iteroutedges(source):
            cap = edge.weight - self.flow[edge.source][edge.target]
            if cap > 0 and self.parent[edge.target] is None:
                self.parent[edge.target] = edge.source
                min_capacity = min(start_capacity, cap)
                min_capacity = self._find_path_dfs(edge.target, min_capacity)
                if min_capacity > 0:
                    self.flow[edge.source][edge.target] += min_capacity
                    self.flow[edge.target][edge.source] -= min_capacity
                    return min_capacity
        return 0
```

4.2. Algorytm Edmondsa-Karpa

Dane wejściowe: Sieć przepływowa, źródło i ujście.

Problem: Wyznaczenie maksymalnego przepływu.

Opis algorytmu: W nieskończonej pętli **while** znajdowane są ścieżki powiększające za pomocą BFS, czyli będą to ścieżki najkrótsze. Przeszukiwanie realizuje metoda _find_path_bfs(). Po dotarciu przeszukiwania do ujścia, przepływ jest zapisywamy w tablicy flow.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(VE^2)$. Odnalezienie ścieżki powiększającej realizowane jest w czasie $O(E)$. Co najmniej jedna

z krawędzi musi zostać przy tym nasycona zwiększonym przepływem, zaś długość nowej ścieżki jest nie większa niż liczba wierzchołków [20].

Listing 4.2. Algorytm Edmondsa-Karpa.

```
#!/usr/bin/python

from Queue import Queue
from edges import Edge

class EdmondsKarp:
    """The Edmonds–Karp algorithm for computing the maximum flow."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if not graph.is_directed():
            raise ValueError("the graph is not directed")
        self.graph = graph
        self.residual = self.graph.__class__(self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            self.residual.add_node(node)
        for edge in self.graph.iteredges():
            self.residual.add_edge(edge)
            self.residual.add_edge(Edge(edge.target, edge.source, 0))
        self.flow = dict()
        for source in self.graph.iternodes():
            self.flow[source] = dict()
            for target in self.graph.iternodes():
                self.flow[source][target] = 0
        self.max_flow = 0

    def run(self, source, sink):
        """Executable pseudocode."""
        if source == sink:
            raise ValueError("source and sink are the same")
        self.source = source
        self.sink = sink
        while True:
            min_capacity = self._find_path_bfs()
            if min_capacity > 0:
                self.max_flow += min_capacity
            else:
                break

    def _find_path_bfs(self):
        """Finding augmenting paths in the residual network."""
        parent = dict((node, None) for node in self.residual.iternodes())
        capacity = {self.source: float("inf")}
        Q = Queue()
        Q.put(self.source)
        while not Q.empty():
            node = Q.get()
            for edge in self.residual.iteroutedges(node):
                cap = edge.weight - self.flow[edge.source][edge.target]
                if cap > 0 and parent[edge.target] is None:
                    parent[edge.target] = edge.source
                    capacity[edge.target] = min(capacity[edge.source], cap)
                    if edge.target != self.sink:
```

```
                    Q.put(edge.target)
                else:
                    target = self.sink
                    while target != self.source:
                        node = parent[target]
                        self.flow[node][target] += capacity[self.sink]
                        self.flow[target][node] -= capacity[self.sink]
                        target = node
                    return capacity[self.sink]
        return 0
```

4.3. Algorytm Dynica na maksymalny przepływ

Dane wejściowe: Sieć przepływowa, źródło i ujście.

Problem: Wyznaczenie maksymalnego przepływu.

Opis algorytmu: Algorytm dzieli graf na warstwy wykorzystując BFS. Następnie znajdowany jest przepływ blokujący przy wykorzystaniu DFS. Przepływ w grafie jest powiększany, a następnie powracamy do kroku z podziałem grafu na warstwy. Jeżeli po podziale grafu na warstwy ujście nie jest osiągalne, to algorytm kończy się.

Złożoność: Liczba warstw w każdym przepływie blokującym rośnie za każdym razem co najmniej o jeden, więc w algorytmie wystąpi co najwyżej $|V|-1$ przepływów blokujących. Graf warstwowy może być zbudowany przy pomocy BFS w czasie $O(E)$, a przepływ blokujący może być przy tym znaleziony w czasie $O(VE)$. Stąd złożoność czasowa algorytmu Dynica wynosi $O(V^2E)$ [21].

Uwagi: Oryginalny algorytm Dynica różni się od wersji popularyzowanej przez Evena i Alona, która została tu pokazana.

Listing 4.3. Algorytm Dynica.

```
#!/usr/bin/python

from Queue import Queue
from edges import Edge

class Dinic:
    """The Dinic's algorithm for computing the maximum flow."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if not graph.is_directed():
            raise ValueError("the graph is not directed")
        self.graph = graph
        self.residual = self.graph.__class__(self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            self.residual.add_node(node)
```

```
        for edge in self.graph.iteredges():
            self.residual.add_edge(edge)
            self.residual.add_edge(Edge(edge.target, edge.source, 0))
        self.flow = dict()
        for source in self.graph.iternodes():
            self.flow[source] = dict()
            for target in self.graph.iternodes():
                self.flow[source][target] = 0
        self.max_flow = 0
        self.level = None

    def run(self, source, sink):
        """Executable pseudocode."""
        if source == sink:
            raise ValueError("source and sink are the same")
        self.source = source
        self.sink = sink
        while self._bfs():
            while True:
                min_capacity = self._dfs(self.source, float("inf"))
                if min_capacity > 0:
                    self.max_flow += min_capacity
                else:
                    break

    def _bfs(self):
        """Find if more flow can be sent from source to sink.
        Assign levels to nodes."""
        self.level = dict((node, None) for node in self.residual.iternodes())
        self.level[self.source] = 0
        Q = Queue()
        Q.put(self.source)
        while not Q.empty():
            node = Q.get()
            for edge in self.residual.iteroutedges(node):
                cap = edge.weight - self.flow[edge.source][edge.target]
                if self.level[edge.target] is None and cap > 0:
                    self.level[edge.target] = self.level[edge.source] + 1
                    Q.put(edge.target)
        return self.level[self.sink] is not None

    def _dfs(self, node, start_capacity):
        """A DFS based function to send flow after BFS has figured out
        that there is a possible flow. Levels were constructed during BFS."""
        if node == self.sink:
            return start_capacity
        for edge in self.residual.iteroutedges(node):
            cap = edge.weight - self.flow[edge.source][edge.target]
            if ((self.level[edge.target] == self.level[edge.source] + 1)
                and cap > 0):
                min_capacity = min(start_capacity, cap)
                min_capacity = self._dfs(edge.target, min_capacity)
                if min_capacity > 0:
                    self.flow[edge.source][edge.target] += min_capacity
                    self.flow[edge.target][edge.source] -= min_capacity
                    return min_capacity
        return 0
```

Tabela 4.1. Tabliczka składania rozwiązań dla skojarzeń w przypadku drzew. Liczby x, y oznaczają liczność skojarzeń.

drzewa	$r0, y$	$r1, y$
$r0, x$	$r1, x+y+1$	$r0, x+y$
$r1, x$	$r1, x+y$	$r1, x+y$

4.4. Największe skojarzenie dla drzew

Drzewa są grafami dwudzielnymi, więc dostępne są algorytmy znajdujące największe skojarzenie w grafach dwudzielnych. Ale można wykorzystać szczególną cechę drzew jaką jest struktura rekurencyjna, aby stworzyć szybszy algorytm. Wykorzystamy podejście podane przez Borie, oparte na metodzie programowania dynamicznego [33].

Przypomnijmy rekurencyjną definicję drzewa z korzeniem. Najmniejszym drzewem jest izolowany wierzchołek r, co zapisujemy jako drzewo (T, r), zbiór wierzchołków $V(T) = \{r\}$, zbiór krawędzi $E(T) = \emptyset$. Z dwóch drzew (T_1, r_1) i (T_2, r_2) tworzymy większe drzewo (T, r) przez połączenie korzeni krawędzią, $r = r_1$, $V(T) = V(T_1) \cup V(T_2)$, $E(T) = E(T_1) \cup E(T_2) \cup \{(r_1, r_2)\}$.

Największe skojarzenie może nasycać korzeń lub nie, dlatego przy łączeniu drzew należy monitorować dwie klasy rozwiązań. Przyjmujemy następujące oznaczenia.

— Rozwiązanie $r0$ - największe skojarzenie **nie** nasycające korzenia.
— Rozwiązanie $r1$ - największe skojarzenie nasycające korzeń.

Należy przygotować tabliczkę składania rozwiązań (tabela 4.1), a końcowe rozwiązanie będzie można zapisać jako $\max\{r0, r1\}$. Listing 4.4 przedstawia implementację algorytmu. Ma on strukturę DFS wzbogaconą obliczaniem największego skojarzenia. Na wejściu może być podane drzewo lub las, a wynik w postaci skojarzenia zapisany jest jako zbiór krawędzi mate_set i jako słownik mate. Złożoność algorytmu wynosi $O(V)$. Wyniki testów algorytmu znajdują się w dodatku A.1.

Listing 4.4. Moduł treemate.

```
#!/usr/bin/python

class BorieMatching:
    """Find a maximum matching 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
        # Zbior zawierajacy krawedzie nalezace do skojarzenia.
        self.mate_set = set()
        # Slownik przechowujacy skojarzenie.
        self.mate = dict((node, None) for node in self.graph.iternodes())
        self.cardinality = 0
        # Ustawianie rekurencji.
```

```
        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 jedne skladowej spojnej, jedno drzewo.
            self.parent[source] = None   # before _visit
            arg2 = self._visit(source)
            self.mate_set.update(max(arg2, key=len))
        else:
            # Tu jest las, moze byc wiele drzew rozlacznych.
            # Skojarzenie bedzie suma skojarzen 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.mate_set.update(max(arg2, key=len))
        self.cardinality = len(self.mate_set)
        for edge in self.mate_set:   # O(V) time
            self.mate[edge.source] = edge.target
            self.mate[edge.target] = edge.source

    def _compose(self, arg1, arg2, edge):
        """Compose results with an edge connecting roots."""
        if edge.source > edge.target:
            edge = ~edge
        r0a, r1a = arg1
        r0b, r1b = arg2
        r0 = r0a|r1b
        r1 = max(r1a|r0b, r1a|r1b, r0a|r0b|set([edge]), key=len)
        return (r0, r1)

    def _visit(self, root):
        """Explore recursively the connected component."""
        # Zaczynamy od drzewa z jednym wierzcholkiem.
        # Nie ma rozwiazania z rootem nalezacym do skojarzenia.
        arg1 = (set(), set())   # (a1_set, b1_set)
        for edge in self.graph.iteroutedges(root):
            if edge.target not in self.parent:
                self.parent[edge.target] = root   # before _visit
                # Dostaje parametry dla poddrzewa.
                arg2 = self._visit(edge.target)
                # Laczenie wynikow. Przekazuje krawedz laczaca korzenie.
                arg1 = self._compose(arg1, arg2, edge)
        return arg1
```

4.5. Największe skojarzenie dla grafów dwudzielnych

Przedstawimy nową implementację algorytmu znajdującego największe skojarzenie w grafie dwudzielnym, przy wykorzystaniu metody ścieżek powiększających [18].

Dane wejściowe: Graf dwudzielny.

Problem: Wyznaczenie największego skojarzenia.

Opis algorytmu: Algorytm rozpoczynamy od rozpoznania grafu dwudzielnego i podziału wierzchołków na dwa zbiory. Następnie w pętli wyszukujemy kolejne ścieżki powiększające i powiększamy skojarzenie. Wyszukiwanie ścieżek powiększających koncepcyjnie odbywa się w pomocniczym grafie skierowanym, ale faktycznie algorytm działa na oryginalnym grafie, tylko odpowiednio sprawdzane są kierunki przechodzenia przez krawędzie grafu. Znalezione ścieżki mogą zawierać cykle, dlatego potrzebna jest jakaś metoda usuwania cykli. W naszej implementacji do usunięcia cykli wykorzystujemy słownik poprzedników na ścieżce. Przechodząc ścieżkę od końca przy użyciu słownika, pomijamy wszystkie cykle. Taka sama metoda stosowana jest w algorytmie Wilsona generowania labiryntu.

Złożoność: W każdym kroku pętli **while** rozmiar skojarzenia rośnie o jeden. Górne ograniczenie na rozmiar skojarzenia to $|V|/2$, stąd pętla wykona się co najwyżej $O(V)$ razy. Wyszukanie jednej ścieżki zajmuje czas $O(E)$, więc złożoność czasowa algorytmu wynosi $O(VE)$.

Uwagi: Testy pokazują, że nasza implementacja ma trochę większą złożoność.

Listing 4.5. Moduł matchingap.

```
#!/usr/bin/python

from bipartite import BipartiteGraphBFS

class MatchingUsingAugmentingPath:
    """Algorytm znajdujacy maksymalne skojarzenie w grafie
    dwudzielnym. Algorytm wykorzystuje sciezke powiekszajaca."""

    def __init__(self, graph):
        """The algorithm initialization."""
        self.graph = graph
        self.mate = dict((node, None) for node in self.graph.iternodes())
        self.cardinality = 0
        algorithm = BipartiteGraphBFS(self.graph)
        algorithm.run()
        self.v1 = set()
        self.v2 = set()
        for node in self.graph.iternodes():
            if algorithm.color[node] == 1:
                self.v1.add(node)
            else:
                self.v2.add(node)

    def run(self):
        """Executable pseudocode."""
        while True:
            path_dict = self._find_augmenting_path()
            if path_dict:
                self.mate.update(path_dict)
                self.cardinality += 1
```

```
            else:
                break

    def _find_augmenting_path(self):
        """Metoda szukajaca sciezki powiekszajacej."""
        self.v1free = set(node for node in self.v1 if self.mate[node] is None)
        self.v2free = set(node for node in self.v2 if self.mate[node] is None)
        for node in self.v1free:
            self.visited_e = set()    # zapisujemy krawedzie w obie strony
            self.visited_v = list()   # lista wierzcholkow na sciezce (sa cykle!)
            last_node = self._visit(node)   # jak sie nie uda to wyjdzie node
            if last_node in self.v2free:
                path = self._remove_cycles()
                return self._create_path_dict(path)
        return {}

    def _visit(self, source):
        """Metoda oparta na dfs'ie szukajaca maksymalnego skojarzenia."""
        self.visited_v.append(source)
        for edge in self.graph.iteroutedges(source):
            if edge not in self.visited_e:
                # Tutaj decydujemy o dozwolonym kierunku krawedzi.
                if self._is_entering(edge) or self._is_outgoing(edge):
                    self.visited_e.add(edge)
                    self.visited_e.add(~edge)
                    last_node = self._visit(edge.target)
                    if last_node in self.v2free:
                        return last_node   # przekazujemy wyzej
        # Czyszczenie, chyba ze jestesmy w v2free.
        if source not in self.v2free:
            self.visited_v.pop()
        return source

    def _is_entering(self, edge):   # from v1 to v2, edge not in matching
        return edge.target in self.v2 and self.mate[edge.target] != edge.source

    def _is_outgoing(self, edge):   # from v2 to v1, edge in matching
        return edge.target in self.v1 and self.mate[edge.target] == edge.source

    def _remove_cycles(self):
        """Usuwanie cykli ze sciezki."""
        node_next = {self.visited_v[-1]: None}
        for i in xrange(len(self.visited_v)-1):
            node_next[self.visited_v[i]] = self.visited_v[i+1]
        new_path = []
        node = self.visited_v[0]
        while node is not None:
            new_path.append(node)
            node = node_next[node]
        return new_path

    def _create_path_dict(self, path):
        """Create dict to update matching."""
        path_dict = dict()
        i = 0
        while i < len(path):    # len(path) is even
            source = path[i]
```

```
            target = path[i + 1]
            path_dict[source] = target
            path_dict[target] = source
            i += 2
        return path_dict
```

4.6. Algorytm Kuhna-Munkresa

Algorytm Kuhna-Munkresa (metoda węgierska) zostanie przez nas wykorzystany do wyznaczenia skojarzenia o najmniejszej wadze w grafach dwudzielnych. Rozwiązanie tego problemu odpowiada zagadnieniu porządkowania zadań, które można przedstawić za pomocą grafu dwudzielnego pełnego w postaci macierzy sąsiedztwa. Graf ten posiada tę samą liczbę wierzchołków w ich dwóch rozłącznych podzbiorach, gdzie jeden podzbiór reprezentuje zadania do wykonania, a drugi dopasowywanych wykonawców zadań [28].

Dane wejściowe: Graf dwudzielny pełny z wagami.

Problem: Wyznaczenie skojarzenia o najmniejszej wadze.

Opis algorytmu: Algorytm wykonywany jest w sześciu krokach. Pierwsze dwa realizowane są jednokrotnie, na początku działania programu.

1. Odjęcie elementu najmniejszego od wszystkich pozostałych w każdym wierszu macierzy sąsiedztwa z wagami zadań.
2. Odnalezienie w zaktualizowanej macierzy współrzędnych wskazujących gdzie znajduje się zero i oznaczenie go gwiazdką (ZERO-STAR). Krok ten powtarzany jest dla każdego pozostałego zera w macierzy, jeżeli w jego wierszu bądź kolumnie nie znajduje się już inne zero oznaczone gwiazdką.
3. Zakrycie każdej kolumny macierzy posiadającej zero oznaczone gwiazdką przed jej dalszym sprawdzaniem. Jeżeli zakryto wszystkie kolumny macierzy, współrzędne macierzy gdzie znajdują się aktualnie zera oznaczone gwiazdką reprezentują optymalne skojarzenie. W przeciwnym razie przechodzimy do kroku numer 4.
4. Odnalezienie zer które nie zostały zakryte przed dalszym sprawdzaniem i oznaczenie ich jako zero prim (ZERO-PRIME). Jeżeli w wierszu gdzie znajduje się zero prim nie ma żadnego zera oznaczonego gwiazdką, przechodzimy do kroku numer 5. W przeciwnym razie zakrywamy ten wiersz przed dalszym sprawdzaniem, odsłaniając natomiast kolumnę z odnalezionym zerem oznaczonym gwiazdką. Kontynuujemy dopóki nie pozostanie żadne zasłonięte zero oznaczone gwiazdką. Zapamiętujemy najmniejszą, odsłoniętą w ten sposób wartość i przechodzimy do kroku numer 6.
5. Zaczynając od współrzędnych elementu zero prim, z którym trafiliśmy przechodząc z kroku numer 4, wyszukujemy naprzemiennie zero oznaczone gwiazdką w jego kolumnie, a następnie zero oznaczone prim w wierszu zera oznaczonego gwiazdką. Kontynuujemy dopóki nie odnajdziemy takiego zera oznaczonego prim, dla którego nie ma zera oznaczonego gwiazdką w tej samej kolumnie. Usuwamy gwiazdkę ze wszystkich odwiedzonych

po drodze elementów oznaczonych gwiazdką, odwiedzonym elementom oznaczonym prim przypisujemy zaś gwiazdkę zamiast prim. Odsłaniamy wszystkie elementy macierzy i powracamy do kroku numer 3.

6. Wartość odnalezioną w kroku 4 dodajemy do każdego elementu każdego zasłoniętego wiersza i odejmujemy ją od wszystkich elementów każdej niezasłoniętej kolumny. Powracamy do kroku numer 4.

Złożoność: Zastosowana implementacja to ulepszona wersja macierzowa oryginalnej metody węgierskiej, posiada złożoność obliczeniową $O(V^3)$ [28].

Uwagi: Graf wejściowy konwertowany jest do macierzy wag zadań przed rozpoczęciem działania algorytmu, zaś po jego wykonaniu wynik w postaci współrzędnych macierzy jest z powrotem dopasowywany do elementów grafu wejściowego. Wiersze macierzy reprezentują jeden rozłączny podzbiór pełnego grafu dwudzielnego, kolumny drugi.

Listing 4.6. Moduł hungarian.

```
#!/usr/bin/python

from bipartite import BipartiteGraphBFS

ZERO, ZERO_STAR, ZERO_PRIME = 0, 1, 2

class BipartiteMatchingMunkresGraph:
    """Find maximum cardinality matching using Munkres algorithm
    (Hungarian Method).
    Based on https://en.wikipedia.org/wiki/Hungarian_algorithm
    and detailed algorithm description from
    http://csclab.murraystate.edu/~bob.pilgrim/445/munkres.html

    Attributes
    ----------
    graph: input directed graph (graph representation)
    must be complete bipartite (biclique), with disjoint NxM node
    subsets of the same size

    Notes
    -----
    """

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("The graph is directed")
        bipartite_coloring = BipartiteGraphBFS(graph)
        # throws ValueError if graph is not bipartite
        bipartite_coloring.run()
        self.first_component = sorted((v for v in bipartite_coloring.color
                                       if bipartite_coloring.color[v] == 0))
        self.second_component = sorted((v for v in bipartite_coloring.color
                                        if bipartite_coloring.color[v] == 1))

        if len(self.first_component) != len(self.second_component):
            raise ValueError("Connected components do not match in size")
```

```
        second_component_set = set(self.second_component)
        for node in self.first_component:
            if graph[node].viewkeys() != second_component_set:
                raise ValueError("Input bipartite graph is not complete")

        self.matrix = list()
        for source in self.first_component:
            row = list()
            for target in sorted(graph[source]):
                row.append(graph[source][target].weight)
            self.matrix.append(row)
        self.mate = set()
        self.n = len(self.matrix)
        self.mask = [[ZERO for i in xrange(self.n)] for j in xrange(self.n)]
        self.row_cover = [False] * self.n
        self.column_cover = [False] * self.n
        self.starred_zeroes = list()
        self.primed_zeroes = list()

    def run(self):
        """Executable pseudocode."""
        self.substract_minimum()
        self.star_initial_zeroes()
        self.minimize_assignments()

        for i, row in enumerate(self.mask):
            for j, val in enumerate(row):
                if self.mask[i][j] == ZERO_STAR:
                    edge_source = self.first_component[i]
                    edge_target = self.second_component[j]
                    self.mate.add((edge_source, edge_target))

    def substract_minimum(self):
        """(Step I)
        Substract the value of the minimum element in every matrix row."""
        for row in self.matrix:
            row_min = min(row)
            for i in xrange(self.n):
                row[i] = row[i] - row_min

    def star_initial_zeroes(self):
        """(Step II)
        Find zeroes in matrix and star them if the rows or columns
        containing them are not already covered.
        Cover row and column after finding a zero to star."""
        for i, row in enumerate(self.matrix):
            for j, val in enumerate(row):
                if (self.matrix[i][j] == ZERO and not self.row_cover[i]
                                              and not self.column_cover[j]):
                    self.row_cover[i] = True
                    self.column_cover[j] = True
                    self.mask[i][j] = ZERO_STAR
        self.clear_covers()

    def clear_covers(self):
        """Clear covers placed during searching for zeroes to star/prime."""
```

```
        for i in xrange(self.n):
            self.row_cover[i] = False
            self.column_cover[i] = False

    def minimize_assignments(self):
        """(Step III and Step V)
        Find the optimum assignment by traversing the cost matrix."""
        while self.cover_starred_zeroes_columns():
            zero_elem = self.cover_all_zeroes()
            self.primed_zeroes.append(zero_elem)
            # here starts step V
            while zero_elem:
                zero_elem = self.find_starred_zero_in_column(zero_elem[1])
                if zero_elem:
                    self.starred_zeroes.append(zero_elem)
                    zero_elem = self.find_primed_zero_in_row(zero_elem[0])
                    self.starred_zeroes.append(zero_elem)
            self.star_primes_and_clear()

    def cover_starred_zeroes_columns(self):
        """Cover all columns where starred zero is present in any row.
        Return true if there are columns yet to be covered."""
        for i in xrange(self.n):
            self.column_cover[i] = any(
                mask_row[i] == ZERO_STAR for mask_row in self.mask)
        return not all(self.column_cover)

    def cover_all_zeroes(self):
        """(Step IV)
        Find all zeroes available, that can yet be covered in matrix.
        Return if any found element does not have a starred zero in row."""
        while True:
            zero_position = self.find_noncovered_zero()
            if zero_position is None:
                self.update_matrix_with_new_minimum()
            else:
                self.mask[zero_position[0]][zero_position[1]] = ZERO_PRIME
                starred_value_index = None
                for i, val in enumerate(self.mask[zero_position[0]]):
                    if val == ZERO_STAR:
                        starred_value_index = i
                        break
                if starred_value_index is not None:
                    self.row_cover[zero_position[0]] = True
                    self.column_cover[starred_value_index] = False
                else:
                    return zero_position

    def star_primes_and_clear(self):
        """Stars prime values and clears other markings."""
        for zero in self.starred_zeroes:
            self.mask[zero[0]][zero[1]] = ZERO
        for zero in self.primed_zeroes:
            self.mask[zero[0]][zero[1]] = ZERO_STAR

        self.starred_zeroes = list()
        self.primed_zeroes = list()
```

```
        for i, row in enumerate(self.mask):
            for j, val in enumerate(row):
                if val == ZERO_PRIME:
                    self.mask[i][j] = ZERO
        self.clear_covers()

    def update_matrix_with_new_minimum(self):
        """(Step VI)
        Update matrix based on the found minimum value."""
        minimum = self.find_smallest_uncovered_value()
        for i, row in enumerate(self.matrix):
            for j, val in enumerate(row):
                if self.row_cover[i]:
                    self.matrix[i][j] += minimum
                if not self.column_cover[j]:
                    self.matrix[i][j] -= minimum

    def find_smallest_uncovered_value(self):
        """Find the smallest value in cost matrix that is not covered."""
        minimum = None
        for i, row in enumerate(self.matrix):
            for j, val in enumerate(row):
                if not self.row_cover[i] and not self.column_cover[j]:
                    if minimum is None:
                        minimum = val
                    if val < minimum:
                        minimum = val
        return minimum

    def find_noncovered_zero(self):
        """If exists, return any (first) uncovered zero in the matrix."""
        for i, row in enumerate(self.matrix):
            for j, val in enumerate(row):
                if (val == ZERO and not self.row_cover[i]
                                and not self.column_cover[j]):
                    return (i, j)

    def find_starred_zero_in_column(self, column_index):
        """If exists, return the index of starred zero occuring in column."""
        for i, row in enumerate(self.mask):
            if row[column_index] == ZERO_STAR:
                return (i, column_index)

    def find_primed_zero_in_row(self, row_index):
        """If exists, return the index of starred zero occuring in row."""
        for j, val in enumerate(self.mask[row_index]):
            if val == ZERO_PRIME:
                return (row_index, j)
```

4.7. Algorytm kwiatowy Edmondsa

Algorytm kwiatowy służy do wyznaczania największego skojarzenia w dowolnych grafach. Został wynaleziony przez Jacka Edmondsa w 1961 roku, udowadniając że problem odnajdywania skojarzenia największego w dowol-

nym grafie posiada złożoność wielomianową. Działanie algorytmu opiera się na wyszukiwaniu ścieżek powiększających w grafie. Nazwa algorytmu związana jest z jego kluczowym krokiem, który realizowany jest podczas przeszukiwania grafu i następuje wtedy, gdy zostaje odnaleziony w nim cykl o nieparzystej długości. Wierzchołki należące do cyklu są "łączone" w jeden wierzchołek - kwiat - który po odnalezieniu ścieżki powiększającej w grafie jest z powrotem "rozwijany" do osobnych elementów [2].

Nasza implementacja jest adaptacją kodu podanego przez Davida Eppsteina, zamieszczonego w serwisie *ActiveState Code* [34].

Dane wejściowe: Dowolny nieskierowany graf.

Problem: Wyznaczenia skojarzenia największego w grafie.

Opis algorytmu: Działanie algorytmu opiera się na wyszukiwaniu ścieżek powiększających w grafie. Wyszukiwanie nowej ścieżki rozpoczyna się od odnalezienia wierzchołków grafu, które nie należą do skojarzenia. Lista ta sukcesywnie pomniejsza się wraz ze zwiększaniem wielkości skojarzenia. Wierzchołki te następnie są po kolei sprawdzane i mogą służyć za element nowej ścieżki powiększającej.

Wraz z kolejnymi krokami algorytmu, wierzchołki przypisywane są do dwóch zbiorów, reprezentujących naprzemienne pozycje na ścieżce. Zbiór T zawiera elementy nieparzyste na ścieżce powiększającej, poczynając od wierzchołka od którego została utworzona ścieżka, zaś zbiór S elementy parzyste. Ścieżki powiększające tworzone są poprzez odnajdywanie krawędzi zaczynających się od nieparzystego wierzchołka wolnego (nienależącego do skojarzenia) i kończących się na innym, nieparzystym wierzchołku wolnym. Częścią ścieżki powiększającej może być podzbiór już istniejącego skojarzenia.

W sytuacji, gdy podczas przeszukiwania struktury grafu odnaleziony zostanie cykl nieparzystej długości, jego wierzchołki zostają tymczasowo łączone w kwiat, którego elementy traktowane jak jeden wierzchołek. Kwiat w tej implementacji reprezentowany jest przez strukturę zbiorów rozłącznych, poprzez sprawdzanie przynależności elementów do obiektu leader. Działanie algorytmu kończy się, gdy podczas poszukiwań nowej ścieżki powiększającej w grafie nie zostanie odnaleziony nowy cykl nieparzystej długości, a równocześnie ścieżka powiększająca nie została powiększona.

Złożoność: Zastosowaną implementację charakteryzuje złożoność obliczeniowa $O(VE\alpha(V,E))$, gdzie $\alpha(V,E)$ to odwrócona funkcja Ackermanna [30]. Wartość tej funkcji dla zwiększającej się liczby wierzchołków i krawędzi rośnie bardzo powoli [2].

Uwagi: W implementacji algorytmu zastosowano strukturę zbiorów rozłącznych (klasa UnionFind w module trees). Posiada ona interfejs słownika, którego kluczami są wierzchołki grafu. Służy ona do efektywnego sprawdzania, do jakiego kwiatu należy dany element. W przypadku gdy kwiat, w którym znajduje się dany wierzchołek, jest elementem innego kwiatu, zwracany jest ten najbardziej zewnętrzny.

Listing 4.7. Moduł trees.

```
#!/usr/bin/python

class UnionFind:
    """UnionFind data structure with path compression and ranking.
    Uses basic dict interface for access to elements."""

    def __init__(self):
        self.parents = {}
        self.ranks = {}

    def __getitem__(self, vertex):
        if vertex not in self.parents:
            self.parents[vertex] = vertex
            self.ranks[vertex] = 1
            return vertex
        else:
            if vertex != self.parents[vertex]:
                #recursive call when accessing dict
                self.parents[vertex] = self[self.parents[vertex]]
            return self.parents[vertex]

    def union(self, *vertices):
        roots = [self[x] for x in vertices]
        highest_root = max([(self.ranks[root], root) for root in roots])[1]
        for root in roots:
            if root != highest_root:
                self.ranks[highest_root] += self.ranks[root]
                self.parents[root] = highest_root
```

Listing 4.8. Moduł blossom.

```
#!/usr/bin/python

from trees import UnionFind

class MaximumMatchingEdmonds:
    """Find maximum cardinality matching using Edmond's
    blossom-contraction algorithm.

    Attributes
    ----------
    graph: input undirected graph

    Notes
    -----
    Based on the algorithm recipe by David Eppstein:
    http://code.activestate.com/recipes/
    221251-maximum-cardinality-matching-in-general-graphs/
    """
    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.mate = dict()
```

```
def run(self):
    """Executable pseudocode."""
    while True:
        is_improved = self.find_augmenting_path()
        if not is_improved:
            break

def find_augmenting_path(self):
    """Search for a new augmenting path. Return False on failure."""

    # leader: union-find structure; the leader of a blossom is one
    # of its vertices (not necessarily topmost), and leader[u] always
    # points to the leader of the largest blossom containing u
    #
    # S: dictionary of leader at even levels of the structure tree.
    # Dictionary keys are names of leader (as returned by the union-find
    # data structure) and values are the structure tree parent of the
    # blossom (a T-node, or the top vertex if the blossom is a root of
    # a structure tree).
    #
    # T: dictionary of vertices at odd levels of the structure tree.
    # Dictionary keys are the vertices; T[x] is a vertex with an unmatched
    # edge to x. To find the parent in the structure tree, use leader[T[x]].
    #
    # unexplored: collection of unexplored vertices within leader of S
    #
    # base: if x was originally a T-vertex, but becomes part of
    # a blossom, base[t] will be the pair (u,v) at the base of the
    # blossom, where u and t are on the same side of the blossom
    # and v is on the other side.

    leader = UnionFind()
    S = {}
    T = {}
    unexplored = []
    base = {}

    for v in self.graph.iternodes():
        if v not in self.mate:
            S[v] = v
            unexplored.append(v)

    current = 0
    while current < len(unexplored):
        u = unexplored[current]
        current += 1

        for v in self.graph[u]:
            if leader[v] in S:  # blossom or augmenting path
                if self.contract_or_augment(u, v, leader,
                                            S, T, unexplored, base):
                    return True

            elif v not in T:  # previously unexplored node, add as T-node
                T[v] = u
                unexplored_node = self.mate[v]
                if leader[unexplored_node] not in S:
```

```
                    S[unexplored_node] = v  # add its match as an S-node
                    unexplored.append(unexplored_node)
        return False

    def contract_or_augment(self, u, v, leader, S, T, unexplored, base):
        """Handles either contracting edges to blossoms or adding them
        to matching. Return False if path was not augmented."""

        if leader[u] == leader[v]:
            # loop inside blossom
            return False

        path1, head1 = {}, u
        path2, head2 = {}, v

        while True:
            # move in parallel through graph to find common node on two paths
            head1 = self.find_next_step(path1, head1, leader, S, T)
            head2 = self.find_next_step(path2, head2, leader, S, T)

            if head1 == head2:
                self.create_blossom(u, v, head1, leader,
                                    S, T, unexplored, base)
                return False

            if leader[S[head1]] == head1 and leader[S[head2]] == head2:
                self.inverse_path(u, leader, S, T, base)
                self.inverse_path(v, leader, S, T, base)
                self.mate[u] = v
                self.mate[v] = u
                return True

            if head1 in path2:
                self.create_blossom(u, v, head1, leader,
                                    S, T, unexplored, base)
                return False

            if head2 in path1:
                self.create_blossom(u, v, head2, leader,
                                    S, T, unexplored, base)
                return False

    def find_next_step(self, path, head, leader, S, T):
        """Find next vertex on path when traversing graph."""
        head = leader[head]
        parent = leader[S[head]]
        if parent == head:
            return head
        path[head] = parent
        path[parent] = leader[T[parent]]
        return path[parent]

    def create_blossom(self, u, v, a, leader, S, T, unexplored, base):
        """Create a new blossom from edge u-v with common ancestor a."""
        a = leader[a]
        path1 = self.find_side_path(u, v, a, leader, S, T, unexplored, base)
        path2 = self.find_side_path(v, u, a, leader, S, T, unexplored, base)
```

```
            leader.union(*path1)
            leader.union(*path2)
            S[leader[a]] = S[a]

        def find_side_path(self, v, u, a, leader, S, T, unexplored, base):
            """Find a path to traverse the blossom (from one direction)."""
            path = [leader[u]]
            b = (u, v)
            while path[-1] != a:
                tnode = S[path[-1]]
                path.append(tnode)
                base[tnode] = b
                unexplored.append(tnode)
                path.append(leader[T[tnode]])
            return path

        def find_alternating_path(self, start, goal, leader, S, T, base):
            """Find alternating path from vertex start to goal.
            If goal is not defined, traverses until reaching the base
            of the tree of start."""
            path = []
            while True:
                while start in T:
                    u, v = base[start]
                    us = self.find_alternating_path(u, start, leader, S, T, base)
                    us.reverse()
                    path += us
                    start = v
                path.append(start)
                if start not in self.mate:
                    return path
                tnode = self.mate[start]
                path.append(tnode)
                if goal and tnode == goal:
                    return path
                start = T[tnode]

        def inverse_path(self, u, leader, S, T, base):
            """Alternate path from vertex to the base of its tree."""
            path = self.find_alternating_path(u, None, leader, S, T, base)
            path.reverse()
            pairs = zip(path[::2], path[1::2])
            for x, y in pairs:
                self.mate[x] = y
                self.mate[y] = x
```

5. Podsumowanie

W pracy przedstawiono szereg algorytmów wyznaczających skojarzenia w różnych typach grafów. Chyba najprostszymi grafami są drzewa, które posiadają strukturę rekurencyjną. Została ona wykorzystana do konstrukcji algorytmu znajdującego największe skojarzenie techniką programowania dynamicznego [33].

Drzewa są również grafami dwudzielnymi, a dla tych grafów istnieją osobne metody wyznaczania skojarzenia największego. Wykorzystuje się teorię sieci przepływowych i algorytmy wyznaczające maksymalny przepływ. W ramach pracy przygotowano nowe wersje algorytmu Forda-Fulkersona, algorytmu Edmondsa-Karpa, oraz implementację algorytmu Dynica. Podano także nową implementację algorytmu znajdującego największe skojarzenie metodą ścieżek powiększających.

W przypadku grafów ogólnych możemy szybko wyznaczyć skojarzenie maksymalne za pomocą prostego algorytmu zachłannego, działającego w czasie liniowym. Natomiast skojarzenie największe można wyznaczyć za pomocą słynnego algorytmu kwiatowego Edmondsa. Jest to najtrudniejszy algorytm omawiany w pracy, którego implementację dołączono do rozwijanej biblioteki grafowej.

Dla grafów dwudzielnych ważonych podano algorytm węgierski znajdujący skojarzenie największe o najmniejszej wadze. Algorytm jest także rozwiązaniem problemu przypisania.

Przypadek grafów ogólnych ważonych jest najtrudniejszy do rozwiązania. Mamy do dyspozycji szybki algorytm zachłanny, który znajduje rozwiązanie przybliżone. Algorytm dokładny jest trudniejszy od algorytmu kwiatowego Edmondsa i wymaga nowych metod. Jego pythonową implementację można znaleźć w bibliotece *NetworkX*, natomiast implementację w C++, znajdującą skojarzenie doskonałe o najmniejszej wadze, opisał Kolmogorov [36].

Implementacje wszystkich algorytmów posiadają zestaw testów sprawdzających poprawność kodu z użyciem modułu `unittest`. Testy czasowej złożoności obliczeniowej wykonano z użyciem modułu `timeit`.

A. Testy algorytmów

Rozdział zawiera wyniki testów algorytmów, których implementacje powstały w ramach niniejszej pracy.

A.1. Testy skojarzeń dla drzew

Testowanie algorytmu wyznaczającego największe skojarzenie dla drzew metodą programowania dynamicznego. Wyniki przedstawia wykres A.1. Dostajemy zależność $O(V)$ dla drzew przypadkowych. Drzewa są grafami dwudzielnymi, więc dostępne są algorytmy znajdujące największe skojarzenie dla ogólnych grafów dwudzielnych. Nasze testy pokazują, że algorytm dedykowany drzewom działa od 10 do 100 razy szybciej niż ogólne algorytmy.

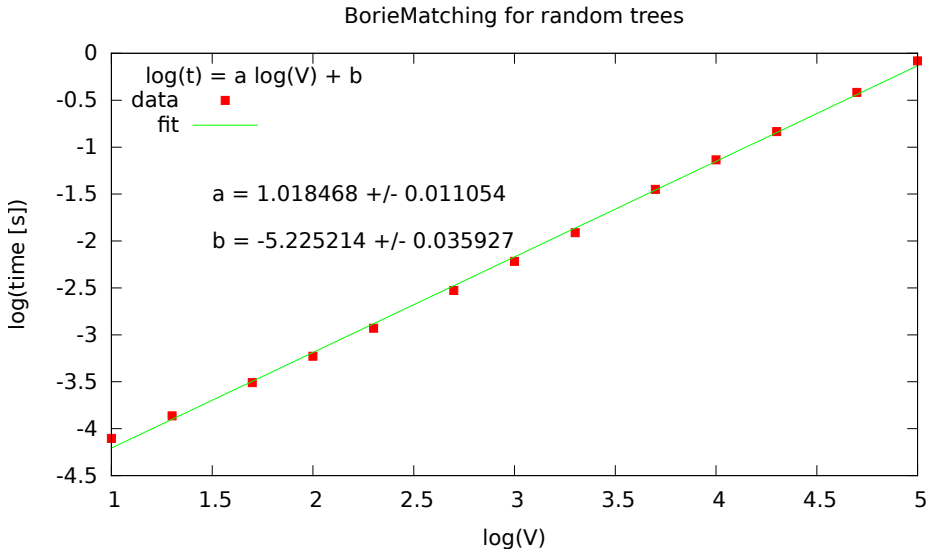
A.2. Testy skojarzeń dla grafów dwudzielnych (ścieżki powiększające)

Przeprowadzono testy algorytmu pozwalającego na znalezienie skojarzenia największego w grafach dwudzielnych, wykorzystującego metodę ścieżek powiększających. Wyniki przedstawia wykres A.2. Praktyczna złożoność obliczeniowa algorytmu trochę przekracza $O(VE)$.

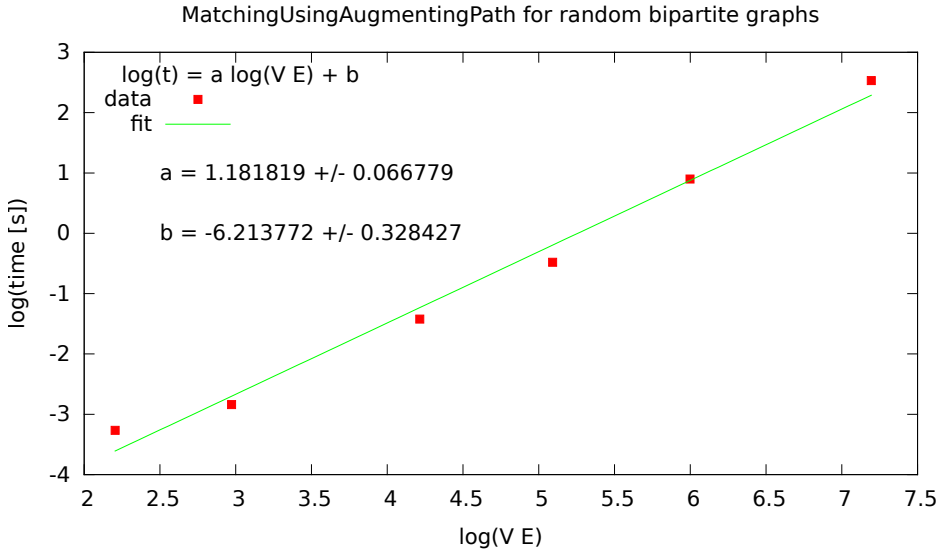
A.3. Testy skojarzeń dla grafów dwudzielnych (sieci przepływowe)

Testy algorytmów służących do odnajdywania największego skojarzenia w grafach dwudzielnych, poprzez sprowadzenie go do problemu wyszukiwania maksymalnego przepływu w sieci przepływowej. Metody sprawdzono na dwóch typach grafów: sieciach przepływowych o losowej strukturze, oraz w sieciach dwupoziomowych, odpowiadającym tym wykorzystywanym w poszukiwaniach skojarzenia największego.

Przetestowano następujące algorytmy:
— algorytm Edmondsa-Karpa (wykresy A.3, A.4),
— rekurencyjna wersję algorytmu Forda-Fulkersona (wykresy A.5, A.6),
— algorytm Dynica (wykresy A.7, A.8).
Wykresy pokazują zwykle mniejszą złożoność algorytmów niż przewiduje teoria, co prawdopodobnie wynika z właściwości użytych generatorów przypadkowych sieci przepływowych. Z drugiej strony, sieci dwuwarstwowe zbudowane do testowania grafów dwudzielnych mogą nie być najtrudniejszym



Rysunek A.1. Wykres wydajności algorytmu rekurencyjnego (Borie) dla drzew. Współczynnik a bliski 1 potwierdza zależność liniową.



Rysunek A.2. Wykres wydajności algorytmu wyznaczającego skojarzenia w grafach dwudzielnych bez wag. Współczynnik a wyższy niż 1 wskazuje na istnienie czynników pogarszających teoretyczną złożoność $O(VE)$.

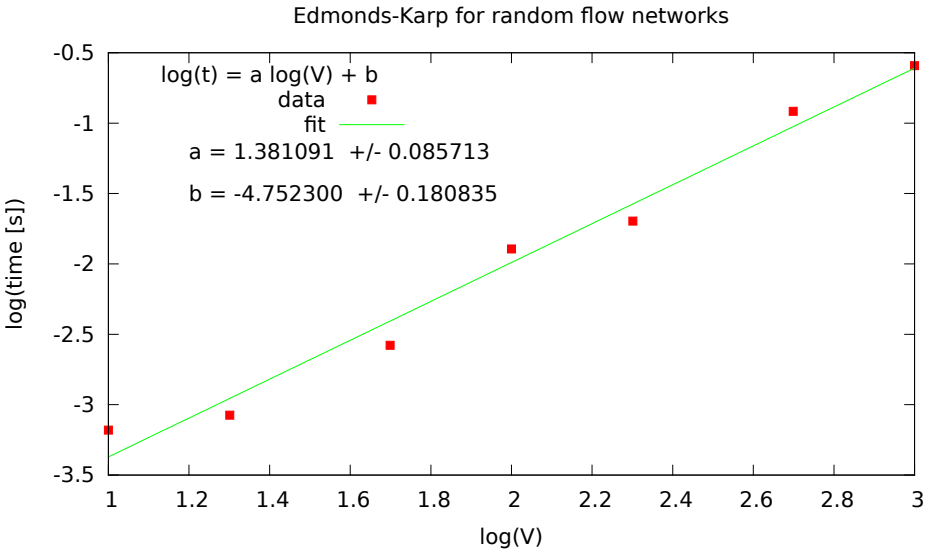
przypadkiem dla algorytmów sieci przepływowych, przez co czas pracy jest krótszy niż oszacowanie ogólne.

A.4. Testy skojarzeń dla grafów dwudzielnych ważonych

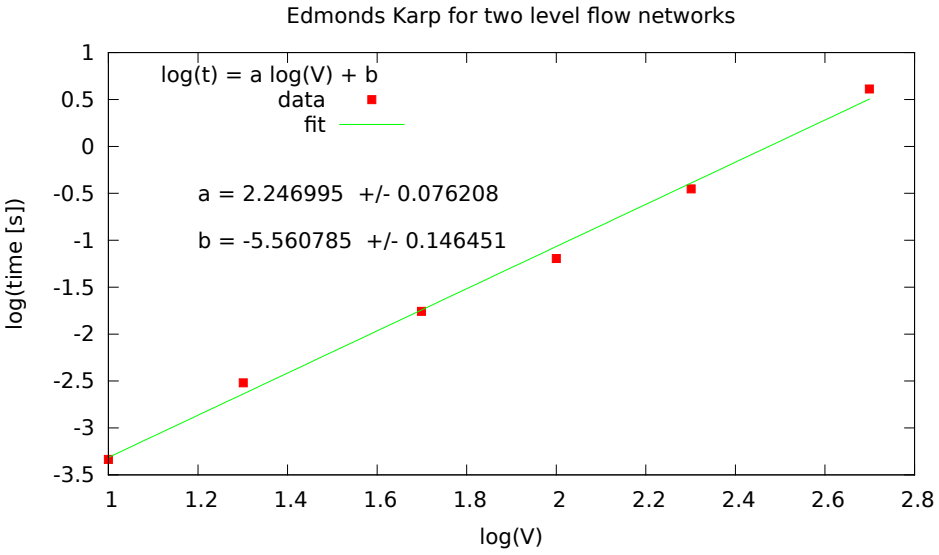
Przetestowano algorytm węgierski Kuhna-Munkresa, służący do znajdowania skojarzenia o największej wadze w pełnym grafie dwudzielnym. Wyniki przedstawia wykres A.9. Implementacja posiada złożoność nieco wyższą niż oczekiwana $O(V^3)$. Możliwą przyczyną jest sposób w jaki Python operuje na dynamicznych obiektach list (w przeciwieństwie do obiektów array o statycznym rozmiarze), których wydajność może maleć przy dużych zbiorach danych. Być może należałoby rozważyć użycie w implementacji algorytmu specjalizowanych macierzy z pakietu numpy, używanego w obliczeniach naukowych [35].

A.5. Testy skojarzeń dla grafów dowolnych

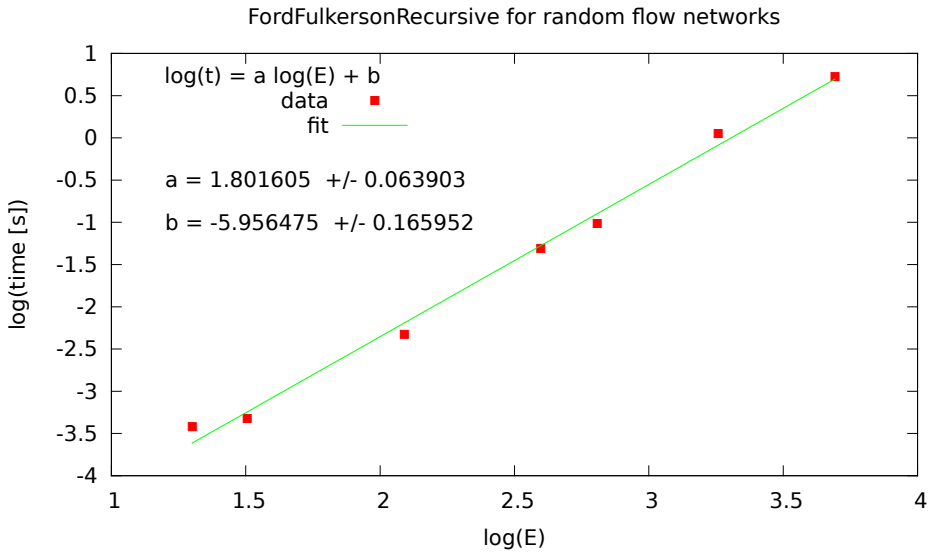
Przeprowadzono testy algorytmu Edmondsa służącego do odnajdywania skojarzenia największego w dowolnych grafach spójnych. Wyniki przedstawia rysunek A.10. Do celów testowych zastosowano metodę tworzącą grafy spójne, oraz posiadające ograniczoną liczbę krawędzi na węzeł, aby sprawdzić działanie algorytmu w warunkach, gdzie konieczne będzie rozwiązywanie problemu występowania nieparzystych cykli w grafie poprzez łączenie wierzchołków w kwiaty. Mimo to algorytm wykazał złożoność zasadniczo niższą niż oczekiwana, co może być spowodowane sposobem generowania danych testowych.



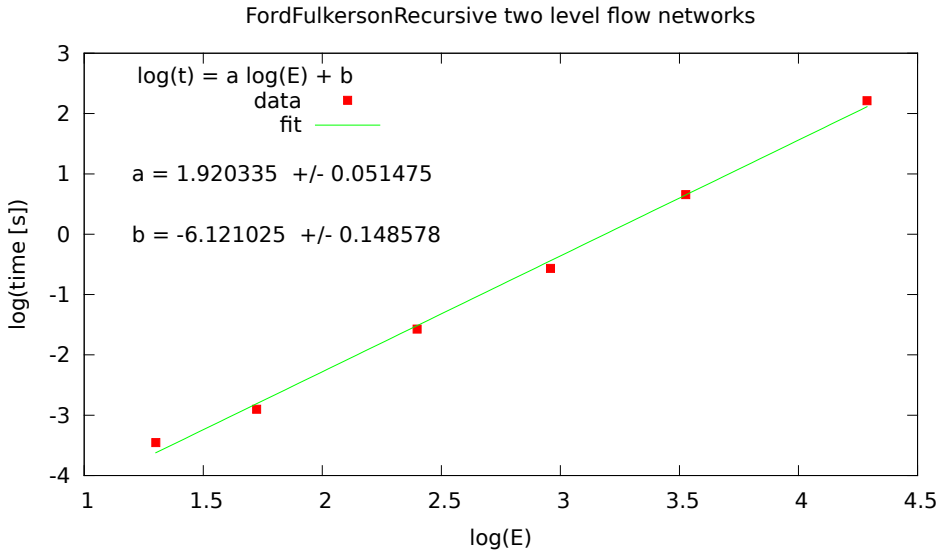
Rysunek A.3. Wykres wydajności algorytmu Edmondsa-Karpa dla losowych sieci przepływowych w zależności od liczby wierzchołków w grafie. Wpływ liczby krawędzi na złożoność wydaje się mniejszy niż oczekiwany względem złożoności teoretycznej $O(VE^2)$.



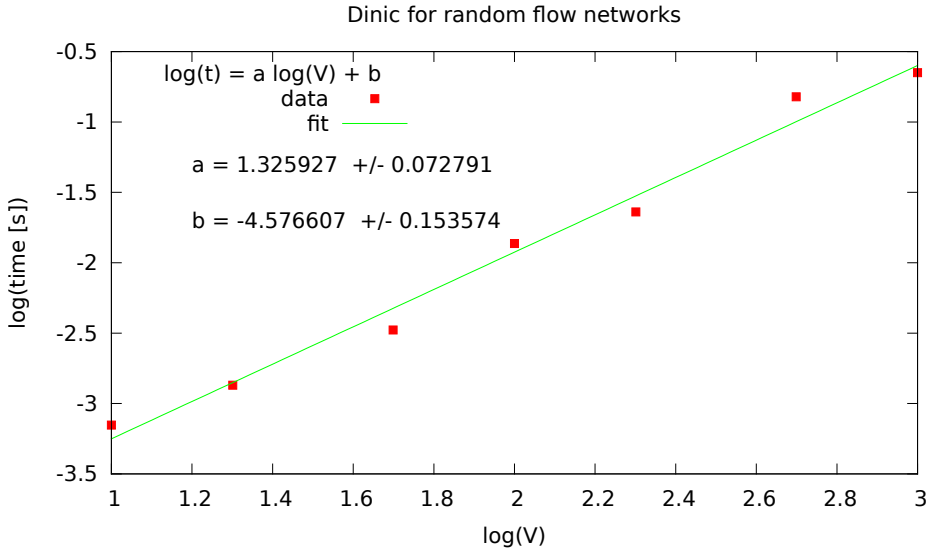
Rysunek A.4. Wykres wydajności algorytmu Edmondsa-Karpa dla dwuwarstwoych sieci przepływowych w zależności od liczby wierzchołków w grafie. Funkcja tworząca testowe sieci przepływowe generuje "płytkie" i bardzo gęste grafy, wpływ liczby krawędzi na złożoność wydaje się mniejszy niż oczekiwany względem złożoności teoretycznej $O(VE^2)$.



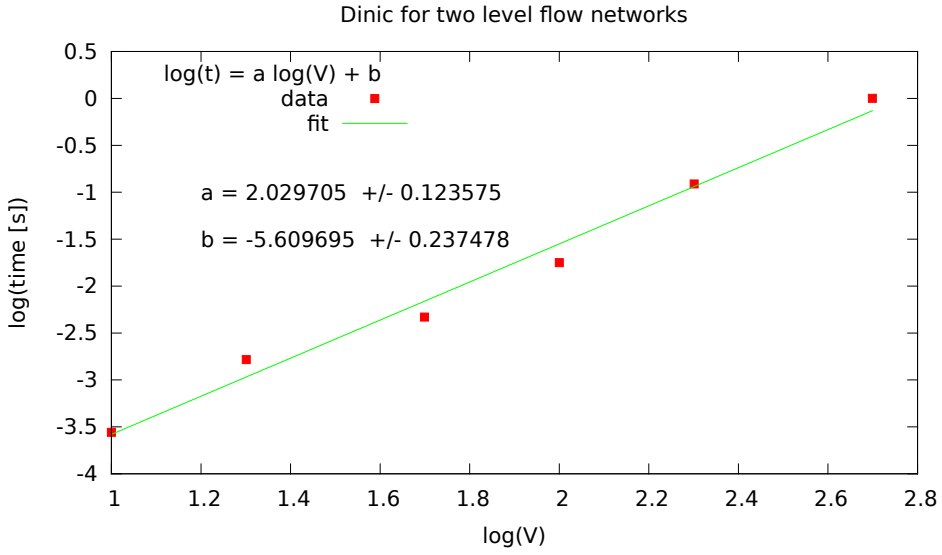
Rysunek A.5. Wykres wydajności rekurencyjnej wersji algorytmu Forda-Fulkersona dla losowych sieci przepływowych w zależności od liczby krawędzi w grafie. Złożoność teoretyczna algorytmu Forda-Fulkersona wynosi $O(E|f|)$, gdzie f odpowiada wartości maksymalnego przepływu i poza zależnością liniową od liczby krawędzi, nie jest bezpośrednio determinowana liczbą wierzchołków i krawędzi.



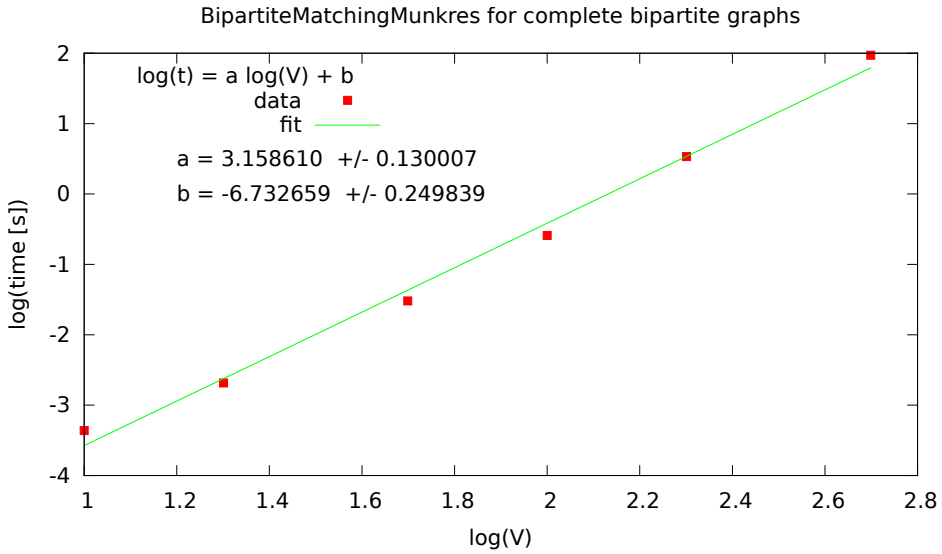
Rysunek A.6. Wykres wydajności rekurencyjnej wersji algorytmu Forda-Fulkersona dla dwuwarstwowych sieci przepływowych w zależności od liczby krawędzi w grafie. Funkcja tworząca testowe sieci przepływowe generuje "płytkie" i bardzo gęste grafy. Złożoność teoretyczna algorytmu Forda-Fulkersona wynosi $O(E|f|)$, gdzie f odpowiada wartości maksymalnego przepływu i poza zależnością liniową od liczby krawędzi, nie jest bezpośrednio determinowana liczbą wierzchołków i krawędzi.



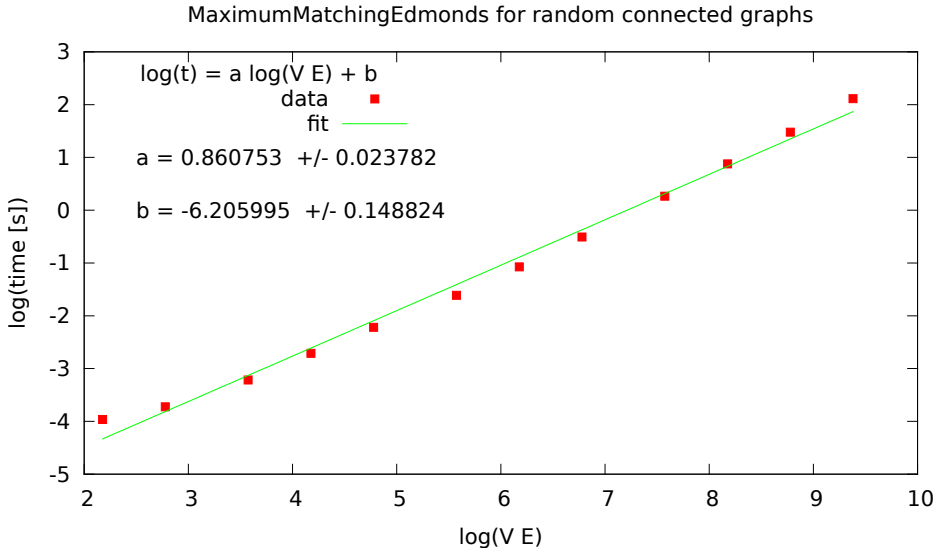
Rysunek A.7. Wykres wydajności algorytmu Dynica dla losowych sieci przepływowych w zależności od liczby wierzchołków w grafie. Złożoność teoretyczna algorytmu Dynica wynosi $O(V^2E)$, niższa niż kwadratowa zależność od liczby wierzchołków może wynikać ze sposobu generowania losowych sieci testowych.



Rysunek A.8. Wykres wydajności algorytmu Dynica dla dwuwarstwowych sieci przepływowych w zależności od liczby wierzchołków w grafie. Złożoność teoretyczna algorytmu Dynica wynosi $O(V^2E)$, współczynnik a zbliżony do 2 jest mały i prawdopodobnie wynika ze sposobu generowania losowych sieci testowych.



Rysunek A.9. Wykres wydajności algorytmu węgierskiego dla grafów dwudzielnych ważonych. Współczynnik a wyższy niż 3 wskazuje na istnienie czynników pogarszających teoretyczną złożoność $O(V^3)$.



Rysunek A.10. Wykres testu algorytmu Edmondsa dla grafów spójnych bez wag. Współczynnikiem nieuwzględnianym w oznaczonej złożoności $O(VE)$ jest odwrócona funkcja Ackermanna, która z powodu bardzo powolnego tempa wzrostu jest traktowana jak stała.

Bibliografia

[1] Wikipedia, Matching (graph theory), 2017, https://en.wikipedia.org/wiki/Matching_%28graph_theory%29.

[2] Wikipedia, Blossom algorithm, 2017, https://en.wikipedia.org/wiki/Blossom_algorithm.

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

[4] Laszló Lovász, Michael Plummer, *Matching Theory*, Volume 29 of Annals of Discrete Mathematics, North-Holland, 1986.

[5] Eugene Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.

[6] Christos H. Papadimitriou, Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, 1982.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

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

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

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

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

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

[13] Leonardo Sampaio, *Algorithmic aspects of graph colourings heuristics*, Université de Nice - Sophia Antipolis, 2012, https://tel.archives-ouvertes.fr/tel-00759408.

[14] Wikipedia, Independent set (graph theory), 2017, https://en.wikipedia.org/wiki/Independent_set_%28graph_theory%29.

[15] Wikipedia, Bipartite graph, 2017, https://en.wikipedia.org/wiki/Bipartite_graph.

[16] Wikipedia, Vertex cover, 2017, https://en.wikipedia.org/wiki/Vertex_cover.

[17] Wikipedia, Edge cover, 2017, https://en.wikipedia.org/wiki/Edge_cover.

[18] Krzysztof Diks, Wojciech Rytter, Piotr Sankowski, *Zaawansowane algorytmy i struktury danych*, 2017, http://wazniak.mimuw.edu.pl/.

[19] Wikipedia, Ford-Fulkerson algorithm, 2017, https://en.wikipedia.org/wiki/Ford-Fulkerson_algorithm.

[20] Wikipedia, Edmonds-Karp algorithm, 2017, https://en.wikipedia.org/wiki/Edmonds-Karp_algorithm.

[21] Wikipedia, Dinic's algorithm, 2017,
https://en.wikipedia.org/wiki/Dinic's_algorithm.

[22] V. M. Malhotra, M. Pramodh Kumar, S. N. Maheshwari, *An $O(|V|^3)$ algorithm for finding maximum flows in networks*, Information Processing Letters 7(6), 277-278 (1978).

[23] Wikipedia, Push-relabel maximum flow algorithm, 2017,
https://en.wikipedia.org/wiki/Push-relabel_maximum_flow_algorithm.

[24] Adam Paweł Wojda, *Wykłady z programowania liniowego*, Wydział Matematyki Stosowanej AGH, 2003,
http://wms.mat.agh.edu.pl/~wojda/Prog_lin/Pl2/Pl2.html.

[25] John E. Hopcroft, Richard M. Karp, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM Journal on Computing 2(4), 225-231 (1973).

[26] Tim Roughgarden, *Lectures Notes on Algorithmic Game Theory, Lecture 10: Kidney Exchange and Stable Matching*, Department of Computer Science, Stanford University (2014) http://theory.stanford.edu/~tim/f13/f13.pdf0.

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

[28] Wikipedia, Hungarian algorithm, 2017,
https://en.wikipedia.org/wiki/Hungarian_algorithm.

[29] Wikipedia, Christofides algorithm, 2017,
https://en.wikipedia.org/wiki/Christofides_algorithm.

[30] Zvi Galil, *Efficient Algorithms for Finding Maximum Matching in Graphs*, ACM Computing Surveys 18, 23-38 (1986).

[31] Aric Hagberg, Dan A. Schult, and Pieter J. Swart, NetworkX, 2017,
http://networkx.github.io/.

[32] Andrzej Kapanowski, *graphs-dict, GitHub repository, 2017*, https://github.com/ufkapano/graphs-dict/.

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

[34] David Eppstein, *Maximum cardinality matching in general graphs (Python recipe)*, ActiveState Code, 2017,
http://code.activestate.com/recipes/221251-maximum-cardinality-matching-in-general-graphs/.

[35] Travis Oliphant, NumPy, 2017,
http://www.numpy.org/.

[36] Vladimir Kolmogorov, *Blossom V: A new implementation of a minimum cost perfect matching algorithm*, Mathematical Programming Computation 1(1), 43-67 (2009).