Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Paweł Konefał

Nr albumu: 1184429

Sieci przepływowe w teorii grafów

Praca magisterska na kierunku Informatyka stosowana

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Informatyki Stosowanej

Kraków 2026

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

Chciałbym złożyć podziękowania Promotorowi, Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu, za poświęcony czas, zaangażowanie i cenne wskazówki, które umożliwiły powstanie pracy magisterskiej, a także za nieocenioną pomoc oraz wsparcie przy analizie algorytmów i udoskonalaniu ostatecznej wersji tej pracy.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów związanych z sieciami przepływowymi. Sieć przepływowa to graf skierowany, w którym wszystkie krawędzie mają nieujemną przepustowość i wyróżnia się dwa wierzchołki: źródło oraz ujście. Przyjmuje się też, że każdy wierzchołek leży na pewnej ścieżce od źródła do ujścia. Modelowanie przepływu w sieciach pozwala na formalne sformułowanie problemów z różnych dziedzin, takich jak transport, logistyka czy zarządzanie zasobami.

Do podstawowych problemów związanych z sieciami przepływowymi zalicza się problem maksymalnego przepływu, w którym należy wyznaczyć przepływ o maksymalnej wartości ze źródła do ujścia uwzględniając przepustowości krawędzi. Następny jest problem najtańszego przepływu, czyli zadanie optymalizacyjne polegające na znalezieniu możliwie najmniejszego sumarycznego kosztu przesyłu przez sieć dla zadanej wartości przepływu.

W części teoretycznej pracy zostały podane definicje i pojęcia związane z teorią grafów oraz sieciami przepływowymi, a także zastosowania tych sieci w problemach z różnych obszarów informatycznych i inżynieryjnych wraz z przykładami. Omówiono podstawowe algorytmy związane z problemem maksymalnego przepływu, jak również bardziej zaawansowane rozwiązania, które były później implementowane w ramach niniejszej pracy. Zaprezentowano podstawowy algorytm służący do rozwiązania problemu najtańszego przepływu oraz wspomniano o planarnych sieciach przepływowych, które pojawiły się w innej pracy dyplomowej związanej z grafami planarnymi.

Część praktyczna poświęcona jest implementacji zaawansowanych algorytmów w języku Python, które zostały przetestowane pod kątem poprawnościowym i wydajnościowym. Zaimplementowane zostały algorytmy typu "prześlij-przemianuj" związane z zagadnieniem maksymalnego przepływu - algorytm generyczny, "przemianuj i przesuń na początek" oraz modyfikacje wykorzystujące kolejkę typu FIFO czy przeszukiwanie wstecz w porządku BFS. Opracowano również algorytm Busackera-Gowena do znajdowania najtańszego przepływu, gdzie wykorzystuje się wyszukiwanie najkrótszych ścieżek z jednego źródła przy obecności ujemnych wag krawędzi. Z tego powodu zaimplementowano algorytm SPFA do znajdowania najkrótszych ścieżek, który jest bardziej wydajny niż tradycyjny algorytm Bellmana-Forda.

Poprawność wszystkich przedstawionych algorytmów została sprawdzona i potwierdzona testami. W testach wydajnościowych złożoność obliczeniowa została wyznaczona eksperymentalnie oraz porównana ze złożonością teoretyczną, a otrzymane wyniki opisane. Dokonano kilku interesujących obserwacji związanych z działaniem oraz wydajnością zaimplementowanych algorytmów, a także możliwości ich dalszej optymalizacji.

Słowa kluczowe: grafy, grafy ważone, sieci przepływowe, problem maksymalnego przepływu, problem najtańszego przepływu

English title: Flow networks in graph theory

Abstract

Python implementation of selected graph algorithms for flow networks is presented. A flow network is a directed graph in which every edge has a non-negative capacity and two distinguished vertices are defined: the source and the sink. It is also assumed that every vertex lies on some path from the source to the sink. Modeling flow in networks allows for the formal formulation of problems from various fields, such as transportation, logistics and resource management.

Among the fundamental problems related to flow networks are the maximum flow problem, which focuses on determining the greatest possible amount of flow from the source to the sink while respecting edge capacities, and the minimum-cost flow problem, an optimization problem that consists of finding the smallest possible total network transmission cost for a given flow value.

The theoretical part of this thesis presents the definitions and concepts related to graph theory and flow networks, as well as applications of flow networks to problems from various areas of computer science and engineering, accompanied by examples. Fundamental algorithms for solving the maximum flow problem are discussed, together with more advanced approaches that were subsequently implemented as part of this thesis. The Busacker-Gowen algorithm for solving the minimum-cost flow problem is also presented. In addition, planar flow networks are mentioned with basic results.

The practical part focuses on the implementation of advanced algorithms in Python, which were evaluated in terms of both correctness and performance. Several push-relabel algorithms for the maximum flow problem were implemented, including the generic push-relabel algorithm, the relabel-to-front algorithm and variants using a FIFO queue or backward breadth-first search. The Busacker-Gowen algorithm for solving the minimum-cost flow problem was developed, relying on shortest-path computations. Therefore, the thesis also includes an implementation of the Shortest Path Faster Algorithm (SPFA) for finding shortest paths from a single source, where negative edge weights are possible. According to the literature, SPFA is more efficient than the Bellman-Ford algorithm, which was also confirmed in performance tests.

The correctness of all presented algorithms was successfully verified. During the performance evaluation, their computational complexity was determined experimentally and compared with the corresponding theoretical complexity and the obtained results were analyzed. Several noteworthy observations were made regarding the behaviour and efficiency of the implemented algorithms, as well as their potential for further optimization.

Keywords: graphs, weighted graphs, flow networks, the maximum flow problem, the minimum-cost flow problem

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy są sieci przepływowe, czyli grafy skierowane ważone spełniające dodatkowe warunki [1]. Stanowią one jeden z podstawowych modeli wykorzystywanych do opisu przepływu zasobów w sieciach oraz rozwiązywania problemów optymalizacyjnych. Jako szczególny rodzaj grafów, sieci przepływowe stanowią jeden z obszarów badań teorii grafów. W literaturze polskiej można znaleźć pozycje częściowo lub w całości poświęcone grafom, które wykorzystano w tej pracy [2], [3], [4].

Modelowanie przepływu w sieciach pozwala na formalne sformułowanie problemów związanych z transportem, logistyką, telekomunikacją czy zarządzaniem zasobami, gdzie istotne jest efektywne przesyłanie określonych wielkości pomiędzy węzłami pewnego systemu, uwzględniając ograniczenia przepustowości poszczególnych kanałów (połączeń między danymi węzłami). W zależności od przyjętego celu optymalizacji, rozważane są różne problemy przepływowe, spośród których do najważniejszych należą problem maksymalnego przepływu oraz problem najtańszego przepływu. Istotą pierwszego z nich jest wyznaczenie maksymalnej ilości przepływu, jaka może zostać przesłana ze źródła do ujścia w sieci przepływowej, z uwzględnieniem ograniczeń przepustowości nałożonych na poszczególne krawędzie. Z kolei problem najtańszego przepływu polega na wyznaczeniu przepływu spełniającego zadane ograniczenia jednocześnie minimalizując całkowity koszt jego realizacji.

1.1. Cel pracy

Celem pracy jest przedstawienie tematyki sieci przepływowych w teorii grafów, implementacji w języku Python [5] algorytmów rozwiązujących problemy maksymalnego i najtańszego przepływu, do rozwiązania których wykorzystywane są sieci przepływowe oraz stworzenie pracy w pełni poświęconej temu zagadnieniu. Zaimplementowane algorytmy posłużą w rozwoju biblioteki graphtheory [6], będą uwzględniały więc obecne w niej struktury.

Tematyka sieci przepływowych była poruszana w kilku wcześniejszych pracach dyplomowych, które również są związane z biblioteką graphtheory [7], [8], [9], jednak jej ujęcie ograniczało się do zakresu i kontekstu poszczególnych prac. Warto również zauważyć, że wspomniane prace dyplomowe powstały pewien czas temu. Biorąc pod uwagę dynamiczny rozwój języka Python oraz dobrych praktych programistycznych, przedstawione implementacje w tych pracach mogą wymagać unowocześnienia.

W niniejszej pracy pojawią się również wydajniejsze algorytmy związane z problemem maksymalnego przepływu czy nowe algorytmy dotyczące problemu najtańszego przepływu, nieujęte w żadnej ze wspomnianych prac.

1.2. Zastosowania sieci przepływowych

Sieci przepływowe znajdują zastosowania w problemach z różnych obszarów informatycznych i inżynieryjnych. Wybrane zastosowania sieci przepływowych:

— Modelowanie wielu problemów przepływu pewnej wielkości w sieci, na przykład obliczanie maksymalnego przepływu cieczy w rurach, części na liniach produkcyjnych lub pojazdów na drogach [7].
— Symulowanie obiektów ze świata rzeczywistego, na przykład linie transportowe, sieci energetyczne, telefoniczne i informatyczne, procesy produkcyjne [10].
— Wyznaczanie skojarzeń w grafach dwudzielnych [11], [12], [13], [14].
— Optymalizacja przepływu zasobów z określonymi ograniczeniami / kosztami.
  - Przykład problemu: Dane jest rozmieszczenie pokoi w domu strachów. Niektóre pokoje mają wejścia, przez które odwiedzający mogą wejść do środka, a inne wyjścia, przez które mogą oni opuścić dom. Pokoje są połączone drzwiami, a odwiedzający mogą poruszać się różnymi ścieżkami (ścieżki nie muszą zawierać wszystkich pokoi). Celem jest minimalizacja kosztów instalacji *specjalnych* podłóg (ang. *shaker floors*) jednocześnie zapewniając, że wszyscy odwiedzający przejdą przez minimum jeden pokój z taką podłogą, niezależnie od wybranej trasy. Koszt zainstalowania podłogi w każdym pokoju jest znany [11].
— Wyznaczanie miejsc w sieci, które uniemożliwiają przesłanie większej ilości towaru (to tak zwane "wąskie gardła"). Ich eliminacja pozwala na uzyskanie lepszej wydajności.
— Przypisywanie pracy do ludzi.
  - Przykład problemu: Problem muzealnych strażników. Pewne muzeum zatrudnia strażników do pilnowania eksponatów. Pojedyncza warta trwa 30 minut. Należy wyznaczyć maksymalną liczbę strażników na daną zmianę uwzględniając pory, kiedy strażnicy nie mogą pracować (przykładowo: strażnik X nie może pracować od 3:30 do 7:30) oraz maksymalną liczbę godzin pracy na dzień [11].
  - Przykład problemu: Znalezienie takiego przypisania, że każda praca ma przydzielonego pracownika oraz żaden pracownik nie jest przepracowany (nie jest przypisany do zbyt wielu prac) [12].
— Problemy związane z segmentacją obrazu.
  - Przykład problemu: *2013 South Central: Drop Zone*. Dany jest obrazek (mapa) będący dwuwymiarową siatką wartości. Należy określić najmniejszą liczbę *barykad*, którą trzeba umieścić, aby zamknąć pewien obszar lub przedmiot. Dla problemu z 2013 roku mamy następujące wartości: D (dane pole jest strefą zrzutu, którą trzeba chronić za wszelką cenę), X (służy jako ściana, po której nie można się poruszać) oraz *kropkę* (. - otwarta przestrzeń, po takich polach można się poruszać). Barykady mogą być pionowe (|) lub poziome (-) [11].
— Problem eliminacji.
  - Przykład problemu: *Baseball Elimination*. Mamy podaną tabelę roz-

grywek ligi bejsbolowej, w której dla każdej drużyny zapisana jest liczba zwycięstw oraz pozostałe spotkania do rozegrania. Z pomocą sieci przepływowych możemy określić, czy dana drużyna X ma szanse na zwycięstwo w tych rozgrywkach uwzględniając aktualny etap sezonu [11], [12], [13].

— Wyznaczanie ścieżek rozłącznych krawędziowo w grafach skierowanych [12], [13], [14].
  - Przykładowe zastosowanie: Odporność na błędy w routingu. Krawędzie / węzły w sieciach mogą ulec awarii. Rozłączne ścieżki umożliwiają planowanie tras zapasowych na wypadek awarii [12].

— Wybór projektów (ang. *Project Selection*, *Project Scheduling*). Dany jest pewien zbiór N projektów, pomiędzy którymi mogą występować zależności (przykładowo: nie można rozpocząć projektu j dopóki nie zostanie zakończony projekt i). Ukończenie projektu wiąże się z wygenerowaniem zysku lub straty. Celem jest wybór takiego podzbioru projektów, który pozwoli na maksymalizację całkowitego zysku [12], [13].

— Przydział sal (ang. *Room Scheduling*). Dany jest zbiór N pomieszczeń i M wydarzeń. Każde pomieszczenie charakteryzowane jest przez liczbę miejsc, natomiast każde wydarzenie przez liczbę uczestników. Należy przydzielić wszystkie wydarzenia do jak najmniejszej liczby sal w taki sposób, aby żadne dwa wydarzenia nie nachodziły na siebie [13], [14].
  - Można również rozszerzyć problem o dodatkowe kryteria, na przykład dodając przedziały czasowe czy opiekunów wydarzeń, którzy mogą mieć własne ograniczenia (ten sam opiekun może być tylko na jednym wydarzeniu jednocześnie, danego dnia jeden opiekun może uczestniczyć maksymalnie w pięciu spotkaniach) [13].

1.3. Struktura pracy

Rozdział 1 zawiera wprowadzenie do tematyki pracy, jej cel i strukturę oraz zastosowania sieci przepływowych w problemach z różnych dziedzin. W rozdziale 2 przedstawiono najważniejsze pojęcia i definicje związane z teorią grafów oraz sieciami przepływowymi, potrzebne do zrozumienia przedstawianych algorytmów. Rozdział 3 przedstawia implementacje i opisy nowych algorytmów związanych z sieciami przepływowymi. Algorytmy te zostaną wykorzystane w rozwoju biblioteki graphtheory. Rozdział 4 zawiera podsumowanie niniejszej pracy i jej wyników. W dodatku A zamieszczono testy wydajnościowe zaimplementowanych algorytmów.

2. Teoria grafów

2.1. Podstawowe definicje i twierdzenia

Grafem (prostym) nazywamy uporządkowaną parę $G = (V, E)$, gdzie V to zbiór wierzchołków (skończony i niepusty), a $E \subset V \times V$ to zbiór krawędzi. W zbiorze krawędzi zabronione są pętle, czyli pary typu (u, u). Krawędzie mogą być skierowane lub nieskierowane, a wtedy utożsamia się pary (u, v) i (v, u).

Graf skierowany lub **digraf** (ang. *directed graph*) to graf, gdzie wszystkie krawędzie w zbiorze E są skierowane. Każda taka krawędź to uporządkowana para dwóch różnych wierzchołków ze zbioru V. Jeżeli (u, v) jest krawędzią z E, to zapis ten oznacza, że krawędź jest wychodząca z wierzchołka u (który jest początkiem krawędzi skierowanej) i wchodząca do wierzchołka v (który jest jej końcem).

Graf ważony to graf, w którym każda krawędź ze zbioru E ma przypisaną dodatkową wartość zwaną **wagą**, oznaczaną jako $w(u, v) \in \mathbf{R}$, gdzie $(u, v) \in E$. W grafie ważonym krawędzie zapisujemy często za pomocą trzech elementów, gdzie dwa pierwsze określają wierzchołki, które są połączone krawędzią, natomiast trzeci określa wagę tej krawędzi. Dla skierowanego grafu ważonego, zapis (u, v, w) oznacza, że krawędź jest wychodząca z wierzchołka u i wchodząca do wierzchołka v oraz o wadze w.

2.2. Sieci przepływowe

Sieć przepływowa (ang. *flow network*) to graf skierowany $G = (V, E)$, w którym wszystkie krawędzie $(u, v) \in E$ mają nieujemną przepustowość $c(u, v) \geqslant 0$. W sieci przepływowej nie występują krawędzie przeciwne (przeciwnie skierowane) ani pętle. Jeśli para wierzchołków (u, v) nie należy do zbioru E to przyjmujemy, że przepustowość krawędzi $c(u, v) = 0$. W sieci przepływowej wyróżniamy dwa wierzchołki: **źródło** s i **ujście** t. Przyjmujemy, że każdy wierzchołek leży na pewnej ścieżce od źródła do ujścia [2].

W przypadku, gdy graf skierowany modelujący pewien problem ma krawędzie przeciwne (na przykład towar może być przesyłany w obie strony pomiędzy danymi wierzchołkami), należy go przekształcić na równoważny graf,

który takich krawędzi nie posiada, otrzymując sieć przepływową. Przekształcenie polega na wybraniu jednej z dwóch krawędzi przeciwnie skierowanych, a następnie dokonaniu jej podziału dodając nowy wierzchołek i zastępując wcześniej wybraną krawędź parą nowych krawędzi. Każda z nich ma przepustowość równą przepustowości krawędzi pierwotnej.

W problemie maksymalnego przepływu zamiast jednego źródła i jednego ujścia może pojawić się wiele źródeł i wiele ujść. Należy wtedy taką sieć sprowadzić do zwykłego problemu z jednym źródłem i jednym ujściem. Tworzony jest nowy wierzchołek zwany **superźródłem** S oraz drugi nowy wierzchołek zwany **superujściem** T. Między superźródłem S i źródłem s_i dodawana jest krawędź skierowana (S, s_i) o przepustowości $c(S, s_i) = \infty$ dla każdego $i = 1, 2, \ldots, n_s$, gdzie n_s to liczba źródeł. Analogicznie, między superujściem T i ujściem t_j dodawana jest krawędź skierowana (t_j, T) o przepustowości $c(t_j, T) = \infty$ dla każdego $j = 1, 2, \ldots, n_t$, gdzie n_t to liczba ujść.

Przepływ to każda funkcja rzeczywista $f : V \times V \to \mathbf{R}$ w sieci przepływowej G spełniająca warunki:

— Warunek przepustowości (ang. *capacity constraints*):
dla każdego $u, v \in V$ zachodzi $f(u, v) \leqslant c(u, v)$.
Przepływ z jednego wierzchołka do drugiego nie może przekroczyć jego przepustowości. Jeśli pomiędzy wierzchołkami u i v nie ma krawędzi, to przepływ $f(u, v) = 0$, ponieważ $c(u, v) = 0$.
— Warunek skośnej symetryczności (ang. *skew symmetry*):
dla każdego $u, v \in V$ zachodzi $f(u, v) = -f(v, u)$.
Oznacza to, że przepływ w odwrotnym kierunku jest ujemny.
— Warunek zachowania przepływu (ang. *flow conservation*):
dla każdego $u \in V - \{s, t\}$ zachodzi $\sum_{v \in V} f(u, v) = 0$.
Łączny przepływ wchodzący do wierzchołka u niebędącego źródłem ani ujściem jest równy łącznemu przepływowi wychodzącemu z tego wierzchołka.

Funkcja $f(u, v)$ nazywana jest **przepływem netto** (ang. *net flow*) z wierzchołka u do wierzchołka v i może przyjmować wartości dodatnie jak i ujemne. **Wartość przepływu** f definiuje się jako sumę przepływów netto, które opuszczają źródło: $|f| = \sum_{v \in V} f(s, v)$.

W problemach maksymalnego i najtańszego przepływu dana jest sieć przepływowa G z źródłem s i ujściem t. W problemie maksymalnego przepływu należy odnaleźć przepływ od źródła do ujścia posiadający maksymalną wartość. Problem najtańszego przepływu to zadanie optymalizacyjne, gdzie dla podanej wartości przepływu należy znaleźć najmniejszy możliwy sumaryczny koszt przesyłu przez sieć. Co ważne, rozwiązanie istnieje tylko wtedy, gdy wartość przepływu nie jest większa niż wartość maksymalnego przepływu.

Przepustowością residualną krawędzi (ang. *residual capacity*) w odniesieniu do przepływu f definuje się jako różnicę między przepustowością krawędzi

oraz wartością przepływu: $c_f(u,v) = c(u,v) - f(u,v)$. Korzystając z własności sieci przepływowych, że przepustowość krawędzi przeciwnej w oryginalnej sieci wynosi 0, można wyznaczyć przepustowość residualną dla krawędzi przeciwnej: $c_f(v,u) = 0 - f(v,u) = f(u,v)$.

Przy użyciu przepustowości residualnych krawędzi, dla sieci przepływowej $G = (V,E)$ można utworzyć **sieć residualną** $G_f = (V,E_f)$ (ang. *residual network*) indukowaną przez przepływ f: $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$ reprezentującą dostępną przepustowość na krawędziach w grafie G. Sieć residualna posiada wszystkie wierzchołki oryginalnej sieci i może wykorzystywać krawędzie nie występujące w niej, czyli krawędzie przeciwne. Przepływ w sieci residualnej może być dozwolony, mimo że w oryginalnej sieci jest zabroniony. Pozwala to na przesłanie z powrotem przepływu co jest równoważne zmniejszeniu przepływu na danej krawędzi.

Ścieżka powiększająca lub rozszerzająca (ang. *augmenting path*) to każda ścieżka w sieci residualnej G_f ze źródła do ujścia. Każda krawędź (u,v) na ścieżce powiększającej musi posiadać niezerowe przepustowości residualne i umożliwiać dodatni przepływ między wierzchołkami u i v bez naruszania przepustowości dla tej krawędzi. Przepustowość residualna ścieżki powiększającej P to najmniejsza z przepustowości jej krawędzi: $c_f(P) = \min\{c_f(u,v) : (u,v) \in P\}$. Przepływ w sieci G jest maksymalny, gdy w sieci residualnej G_f nie istnieje żadna ścieżka powiększająca.

Sieć warstwowa (ang. *level graph*) - niech $\mathrm{dist}(v)$ będzie długością najkrótszej ścieżki od źródła s do wierzchołka v w sieci residualnej. Sieć warstwowa G_L na bazie sieci residualnej G_f to graf z takimi krawędziami, gdzie $\mathrm{dist}(v) = \mathrm{dist}(u) + 1$ dla $(u,v) \in E_f$.

Przepływ blokujący (ang. *blocking flow*) - przepływ w sieci warstwowej, dla którego nie ma ścieżki powiększającej.

2.3. Problem maksymalnego przepływu

Problem maksymalnego przepływu (ang. *the maximum flow problem*) polega na określeniu największej ilości przepływu ze źródła s do ujścia t, jaki można osiągnąć bez naruszania ograniczeń przepustowości nałożonych na poszczególne kanały. Istnieje wiele algorytmów przeznaczonych do rozwiązywania problemu maksymalnego przepływu, niektóre z nich mają już swoje implementacje w bibliotece graphtheory.

— **Algorytm Forda-Fulkersona** [15]. Algorytm zachłanny, który wyznacza maksymalny przepływ poprzez znajdowanie ścieżek powiększających w sieci residualnej. Złożoność obliczeniowa jest ograniczona przez $O(E|f|)$, gdzie $|f|$ to wartość maksymalnego przepływu. Warianty implementacji algorytmu opisano w pracach dyplomowych Łukasza Gałuszki [7] i Krzysztofa Niedzielskiego [8]. Praca Gałuszki dodaje do biblioteki graphtheory

moduł graphtheory.flow.fordfulkerson, gdzie znajdowanie ścieżek powiększających odbywa się z pomocą DFS. Jest to realizowane przy użyciu metody _find_path() wywoływanej w nieskończonej pętli **while**. W pracy Niedzielskiego, biblioteka graphtheory została rozszerzona o rekurencyjną implementację tego algorytmu.

— **Algorytm Edmondsa-Karpa** [16]. Specjalizacja algorytmu Forda-Fulkersona, która poprawia jego złożoność. W każdej iteracji wybierana jest najkrótsza ścieżka powiększająca - ścieżka o najmniejszej liczbie krawędzi. Najkrótsza ścieżka jest wybierana z wykorzystaniem BFS. Złożoność obliczeniowa algorytmu wynosi $O(VE^2)$. Implementacja algorytmu również znajduje się w pracach dyplomowych Gałuszki i Niedzielskiego. Praca Gałuszki dodaje do biblioteki graphtheory moduł graphtheory.flow.edmondskarp, gdzie znajdowanie ścieżek powiększających z pomocą BFS jest realizowane przy użyciu metody _find_path(), wywoływanej w nieskończonej pętli **while**. W pracy Niedzielskiego została przedstawiona nowa implementacja algorytmu zaimplementowanego przez Gałuszkę, wykorzystując nowsze praktyki języka Python oraz przedstawiając kod w bardziej czytelny i ujednolicony sposób.

— **Algorytm Dinica** [17]. Również wykorzystuje najkrótsze ścieżki powiększające do wyznaczenia maksymalnego przepływu. W przeciwieństwie do Edmondsa-Karpa, algorytm ten wykorzystuje dodatkowo sieć warstwową oraz przepływ blokujący, co pozwala na osiągnięcie złożoności obliczeniowej $O(V^2E)$. Algorytm Dinica dzieli graf na warstwy z pomocą BFS, a następnie znajduje przepływ blokujący z wykorzystaniem DFS. Po zwiększeniu przepływu, graf ponownie jest dzielony na warstwy i szukany jest kolejny przepływ blokujący. Algorytm działa do momentu, aż po podziale grafu ujście nie jest osiągalne. Implementacja algorytmu została opisana w pracy Niedzielskiego i dodana do biblioteki graphtheory jako moduł graphtheory.flow.dinic. W implementacji algorytmu znajdują się dwie pętle **while**, gdzie jedna jest zagnieżdżona w drugiej. Zewnętrzna pętla ma za zadanie sprawdzać, czy po podziale grafu na warstwy osiągalne jest ujście – jeśli tak, to w wewnętrznej pętli zwiększany jest przepływ, aż do momentu znalezienia przepływu blokującego, co powoduje powrót do pętli zewnętrznej.

— **Algorytmy typu "prześlij-przemianuj"** [18]. Algorytmy te również wykorzystują sieci residualne, a po zakończeniu działania wszystkie wierzchołki, oprócz źródła i ujścia, mają przepływ netto równy 0 (warunek zachowania przepływu jest spełniony). W porównaniu do algorytmów opartych na metodzie Forda-Fulkersona są jednak bardziej lokalne. Zamiast badać całą sieć residualną, aby znaleźć ścieżkę powiększającą, podczas jednego kroku działają tylko na jednym wierzchołku jednocześnie. Dzięki temu podejściu, złożoność obliczeniowa algorytmu wynosi $O(V^2E)$ (Goldberg-Tarjan). Algorytmy typu "prześlij-przemianuj" w czasie swojego działania łamią warunek zachowania przepływu. W tych algorytmach występuje pojęcie **przedprzepływu** (ang. *preflow*), a z każdym wierzchołkiem powiązane są dwie wielkości: wysokość i nadmiar przepływu. Szczegółowy opis algorytmu będzie podany w rozdziale 3.

— **Algorytm typu "przemianuj i przesuń na początek"**. Modyfika-

cja algorytmu typu "prześlij-przemianuj", gdzie struktura danych jest zarządzana efektywniej, a podstawowe operacje *prześlij* i *przemianuj* są wykonywane w odpowiedniej kolejności. Algorytm skanuje listę wierzchołków wybierając przepełnione wierzchołki i je "rozładowuje" (ang. *discharging*), wykonując operacje *prześlij* i *przemianuj*. Po przemianowaniu wierzchołka, przesuwany jest on na początek listy. Złożoność obliczeniowa algorytmu typu "prześlij i przesuń na początek" wynosi $O(V^3)$, co oznacza, że złożoność nie zależy od liczby krawędzi. Algorytm jest przydatny dla grafów gęstych, które mają dużą liczbę krawędzi. Szczegółowy opis algorytmu będzie podany w rozdziale 3.

Listing 2.1. Przykład sesji interaktywnej - wyznaczanie maksymalnego przepływu w sieci przepływowej z użyciem algorytmów Forda-Fulkersona, Edmondsa-Karpa i Dinica.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from fordfulkerson import FordFulkerson, FordFulkersonRecursive
>>> from edmondskarp import EdmondsKarp
>>> from dinic import Dinic
>>> V = 100
>>> gf = GraphFactory(Graph)
>>> G = gf.make_flow_network(V)
>>> ff = FordFulkerson(G)
>>> ff.run(0, V-1)
>>> ff.flow      # przeplywy przez poszczegolne krawedzie
(dict)
>>> ff.max_flow      # wartosc maksymalnego przeplywu
(number)
>>> ffr = FordFulkersonRecursive(G)
>>> ffr.run(0, V-1)
>>> ffr.flow
(dict)
>>> ffr.max_flow
(number)
>>> ek = EdmondsKarp(G)
>>> ek.run(0, V-1)
>>> ek.flow
(dict)
>>> ek.max_flow
(number)
>>> dinic = Dinic(G)
>>> dinic.run(0, V-1)
>>> dinic.flow
(dict)
>>> dinic.max_flow
(number)
```

2.4. Problem najtańszego przepływu

Problem najtańszego przepływu (ang. *the minimum-cost flow problem*) polega na znalezieniu przepływu o zadanej wartości F ze źródła s do ujścia t w sieci przepływowej z jak najmniejszym kosztem przesyłu. Oprócz pojemności, z każdą krawędzią związany jest nieujemny koszt $w(u, v)$ przesyłu jednej jednostki z wykorzystaniem danej krawędzi [4]. Aby istniało rozwiązanie, wartość F nie może być większa niż wartość maksymalnego przepływu. Algorytmem rozwiązującym problem najtańszego przepływu jest algorytm Busackera-Gowena, który działa podobnie jak algorytm Forda-Fulkersona. Złożoność obliczeniowa algorytmu Busackera-Gowena jest ograniczona przez $O(VE|f|)$ [13] (szczegółowy opis w rozdziale 3).

2.5. Planarne sieci przepływowe

W pewnych zastosowaniach sieci przepływowe mogą być grafami planarnymi skierowanymi, a to można wykorzystać do stworzenia bardziej wydajnych algorytmów rozwiązujących problem maksymalnego przepływu (ang. *maximum st-flow problem*). W roku 2009 Borradaile i Klein [19] podali algorytm działający w czasie $O(n \log n)$.

Szczególnym przypadkiem planarnych sieci przepływowych są sieci, w których źródło s i ujście t są na brzegu tej samej ściany. Przyjmuje się, że jest to ściana zewnętrzna grafu, a grafy nazywa się *st*-planarnymi (ang. *st-planar graphs*). Rozważania teoretyczne związane ze znajdowaniem najkrótszych ścieżek w grafach planarnych doprowadziły koło roku 1997 do powstania algorytmu działającego w czasie $O(n)$ [20].

Kolejnym szczególnym przypadkiem planarnych sieci przepływowych są sieci acykliczne, do których należą skierowane grafy szeregowo-równoległe (dsp-grafy) rozważane w pracy licencjackiej Magdaleny Stępień [9]. Przedstawiono tam generator dsp-grafów przypadkowych oraz algorytm rozpoznający dsp-grafy.

3. Algorytmy

W tym rozdziale zostaną dokładnie opisane implementacje nowych algorytmów przygotowanych w ramach niniejszej pracy. Zwykle na wejściu algorytmów mamy sieć przepływową, czyli graf skierowany ważony, spełniający dodatkowe warunki opisane w rozdziale 2. Następnie tworzymy sieć residualną (obiekt residual), czyli graf skierowany ważony z dodatkowymi krawędziami przeciwnymi o zerowej przepustowości. Przepływ maksymalny jest wyznaczany w sieci residualnej przy wykorzystaniu obiektu flow. Koncepcyjnie obiekt ten jest tablicą z zapisaną chwilową wartością przepływu między parami wierzchołków. W naszej implementacji jest to słownik słowników.

W opisach z literatury zwykle przyjmuje się, że krawędzie w sieci residualnej pojawiają się i znikają, stosownie do niezerowej lub zerowej wartości przepustowości residualnej $c_f(u, v)$. W naszych implementacjach krawędzie sieci residualnej nie znikają, tylko na bieżąco jest obliczana ich chwilowa przepustowość residualna w instrukcji cap = edge.weight − **self**.flow[u][v]. Dodatnia przepustowość residualna krawędzi pozwala na wykorzystanie krawędzi w budowie ścieżki powiększającej lub na wykonanie operacji przesłania przepływu.

3.1. Algorytm typu "prześlij-przemianuj"

Jak zostało wspomniane, generyczny algorytm typu "prześlij-przemianuj" działa w bardziej lokalny sposób niż algorytmy oparte na metodzie Forda-Fulkersona. Zamiast badać całą sieć residualną, pracuje on na jednym wierzchołku jednocześnie, analizując tylko sąsiadów danego wierzchołka. W czasie działania algorytmu łamie też warunek zachowania przepływu oraz używa pojęcia przedprzepływu, czyli przepływu z osłabionym warunkiem zachowania przepływu: dla każdego $u \in V - \{s\}$ zachodzi

$$\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \geqslant 0.$$

Oznacza to, że łączny przepływ wchodzący do wierzchołka u niebędącego źródłem może być większy niż łączny przepływ wychodzący z tego wierzchołka. Różnica ta nazywana jest **nadmiarem** (ang. *excess*).

Działanie generycznego algorytmu opiera się na zainicjalizowaniu przedprzepływu, a następnie wyborze i wykonaniu odpowiedniej operacji *prześlij* lub *przemianuj* dopóki taka istnieje. Algorytm nie mówi nic o kolejności tych operacji, ani o kolejności wierzchołków, w jaki mają zostać wybrane, co zostawia dowolność w wyborze.

Dane wejściowe: Sieć przepływowa, źródło i ujście.

Problem: Wyznaczenie maksymalnego przepływu.

Opis algorytmu: Algorytm rozpoczynamy od zainicjalizowania przedprzepływu. Dla każdego wierzchołka początkowy nadmiar oraz wysokość ustawiane są na zero. Jedynym wyjątkiem jest źródło, którego wysokość jest równa liczbie wierzchołków w grafie. Następnie przesyłamy przedprzepływ do każdego sąsiada źródła oraz tworzymy zbiór, którego elementami są wierzchołki z dodatnim nadmiarem i niebędące źródłem lub ujściem. Po inicjalizacji pobierany jest dowolny wierzchołek ze zbioru i wykonywana na nim metoda _push_or_relabel(u). Metoda ta przesyła nadmiar do kolejnych wierzchołków, dopóki nadmiar jest dodatni lub zostały przejrzane wszystkie wierzchołki. Jeżeli nadmiar nadal jest dodatni, wykonywane jest *przemianowanie*, a wierzchołek zostaje dodany ponownie do zbioru. Do zbioru dodawany jest również sąsiad przetwarzanego wierzchołka, jeżeli został mu przesłany przedprzepływ.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V^2E)$.

Uwagi: Przedprzepływ może zostać przesłany do sąsiedniego wierzchołka tylko wtedy, gdy przepustowość residualna jest dodatnia, natomiast wysokość sąsiada jest o 1 jednostkę niższa niż przetwarzanego wierzchołka.

Listing 3.1. Algorytm typu "prześlij-przemianuj".

```
#!/usr/bin/env python3

from graphtheory.structures.edges import Edge


class PushRelabelGeneric:
    """
    Notes
    -----
    Based on:

    Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., 2009,
        Introduction to Algorithms, third edition, The MIT Press,
        Cambridge, London.

    https://en.wikipedia.org/wiki/Push-relabel_maximum_flow_algorithm

    https://cp-algorithms.com/graph/push-relabel.html

    https://www.youtube.com/watch?v=_QCWBZ_3Ci0
    """

    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__(n=self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            self.residual.add_node(node)
```

```
        # Initial capacities for the residual network.
        for edge in self.graph.iteredges():
            self.residual.add_edge(edge)  # original capacity
            self.residual.add_edge(Edge(edge.target, edge.source, 0))

        # Flow (at the end must be legal).
        self.flow = dict()
        for source in self.graph.iternodes():
            self.flow[source] = dict()
            for target in self.graph.iternodes():
                self.flow[source][target] = 0

        # Initial flow is zero.
        self.max_flow = 0
        self.height = dict()
        self.excess = dict()

    def run(self, source, sink):
        """Executable pseudocode."""
        if source == sink:
            raise ValueError("source and sink are the same")
        self.source = source
        self.sink = sink
        self._initialize_preflow()

        # While we have nodes with excess flow, do applicable operation.
        while self.node_set:
            u = self.node_set.pop()
            self._push_or_relabel(u)

        self.max_flow = self.excess[self.sink]

    def _initialize_preflow(self):
        for u in self.residual.iternodes():
            self.height[u] = 0
            self.excess[u] = 0

        self.height[self.source] = self.residual.v()

        for edge in self.graph.iteroutedges(self.source):
            self.excess[edge.target] += edge.weight
            self.excess[edge.source] -= edge.weight
            self.flow[edge.source][edge.target] += edge.weight
            self.flow[edge.target][edge.source] -= edge.weight

        self.node_set = set(u for u in self.residual.iternodes()
            if self.excess[u] > 0 and u != self.source and u != self.sink)

    def _push_or_relabel(self, u):
        for edge in self.residual.iteroutedges(u):
            v = edge.target
            cap = edge.weight - self.flow[u][v]

            # We can add flow only if c_f > 0 and h[u] = h[v] + 1.
            if cap > 0 and self.height[u] == self.height[v] + 1:
                self._push(edge)
```

```
                if self.excess[u] == 0:
                    break

        if self.excess[u] > 0:
            self._relabel(u) # cannot push, so we must relabel
            self.node_set.add(u) # still have some excess flow

    def _push(self, edge):
        cap = edge.weight - self.flow[edge.source][edge.target]
        d_flow = min(self.excess[edge.source], cap)
        self.flow[edge.source][edge.target] += d_flow
        self.flow[edge.target][edge.source] -= d_flow
        self.excess[edge.source] -= d_flow
        self.excess[edge.target] += d_flow

        if edge.target != self.source and edge.target != self.sink:
            self.node_set.add(edge.target)

    def _relabel(self, u):
        min_h = float('inf')
        for edge in self.residual.iteroutedges(u):
            cap = edge.weight - self.flow[edge.source][edge.target]
            if cap > 0: # residual capacity exists
                min_h = min(min_h, self.height[edge.target])

        if min_h != float('inf'):
            self.height[u] = 1 + min_h
```

3.2. Algorytm typu "przemianuj i prześlij na początek"

Generyczna metoda "prześlij-przemianuj" pozwala na stosowanie operacji *prześlij* i *przemianuj* w dowolnej kolejności. Jednak gdy kolejność tych operacji będzie wybierana odpowiednio, a struktura danych sieci zarządzana efektywnie, problem maksymalnego przepływu można rozwiązać szybciej niż $O(V^2E)$. Przedstawiony poniżej algorytm typu "przemianuj i prześlij na początek" działa w czasie $O(V^3)$, co jest asymptotycznie przynajmniej tak dobre jak $O(V^2E)$, natomiast dla gęstych sieci jest lepsze, ponieważ nie zależy od liczby krawędzi w sieci przepływowej. W algorytmie tym znajduje się też lista wierzchołków sieci, którą algorytm skanuje i wybiera przepełniony wierzchołek. Jest on następnie *rozładowywany*, dopóki ma dodatni nadmiar. Za każdym razem, gdy wierzchołek zostanie przemianowany, przesuwany jest na początek listy, a następnie kontynuowane jest skanowanie listy w poszukiwaniu kolejnego przepełnionego wierzchołka.

Dane wejściowe: Sieć przepływowa, źródło i ujście.

Problem: Wyznaczenie maksymalnego przepływu.

Opis algorytmu: Algorytm rozpoczynamy od zainicjalizowania przedprzepływu, tak samo jak w wersji generycznej, z tą różnicą, że zamiast zbioru używamy listy podwójnie wiązanej do przechowywania wszystkich wierzchołków

oprócz źródła i ujścia (niezależnie od posiadanego nadmiaru). Po inicjalizacji wczytujemy pierwszy element z listy i go *rozładowujemy*. Jeśli podczas operacji *rozładowania* choć raz *przemianowaliśmy* wierzchołek, ustawiamy go na początku listy, w przeciwnym razie zostaje na aktualnym miejscu w liście. Niezależnie od *przemianowania*, po przetworzeniu wierzchołka wczytujemy kolejny z listy, będący aktualnie za przetworzonym dopiero co wierzchołkiem. Postępujemy tak, aż do momentu, kiedy dotrzemy do końca listy. Metoda _discharge(u) wykonuje operacje *prześlij* lub *przemianuj* do momentu, aż z wierzchołka pozbędziemy się całego nadmiaru.

Złożoność: Złożoność czasowa algorytmu wynosi $O(V^3)$.

Uwagi: Do przechowywania listy wierzchołków zaimplementowano osobną klasę DoubleList reprezentującą listę podwójnie wiązaną, aby operacje wstawiania i usuwania wierzchołków listy miały czas $O(1)$.

Listing 3.2. Algorytm typu "przemianuj i prześlij na początek".

```
#!/usr/bin/env python3

from graphtheory.structures.edges import Edge
from doublelist import DoubleList


class PushRelabelToFront:
    """
    Notes
    -----
    Based on:

    Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., 2009,
        Introduction to Algorithms, third edition, The MIT Press,
        Cambridge, London.

    https://en.wikipedia.org/wiki/Push-relabel_maximum_flow_algorithm

    https://cp-algorithms.com/graph/push-relabel.html
    """

    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__(n=self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            self.residual.add_node(node)

        # Initial capacities for the residual network.
        for edge in self.graph.iteredges():
            self.residual.add_edge(edge)  # original capacity
            self.residual.add_edge(Edge(edge.target, edge.source, 0))

        # Flow (at the end must be legal).
```

```
        self.flow = dict()
        for source in self.graph.iternodes():
            self.flow[source] = dict()
            for target in self.graph.iternodes():
                self.flow[source][target] = 0

        # Initial flow is zero.
        self.max_flow = 0
        self.height = dict()
        self.excess = dict()
        # Neighbor lists and current indices, from Cormen.
        self.neighbors = dict()
        for u in self.residual.iternodes():
            self.neighbors[u] = list(self.residual.iteroutedges(u))
        self.neighbor_idx = dict((u, 0) for u in self.residual.iternodes())

    def run(self, source, sink):
        """Executable pseudocode."""
        if source == sink:
            raise ValueError("source and sink are the same")
        self.source = source
        self.sink = sink
        self._initialize_preflow()

        node_list = DoubleList()
        for u in self.residual.iternodes():
            if u != self.source and u != self.sink:
                node_list.insert_tail(u)

        current = node_list.head
        # While we have nodes with excess flow, discharge them.
        while current:
            u = current.data
            old_height = self.height[u]

            self._discharge(u)

            if self.height[u] > old_height:
                # move u to the front of the list...
                node_list.remove(current)
                node_list.insert_head(current.data)
                # ...and go to the next node after u
                current = node_list.head.next
            else:
                current = current.next # go to the next node after u

        self.max_flow = self.excess[self.sink]

    def _initialize_preflow(self):
        for u in self.residual.iternodes():
            self.height[u] = 0
            self.excess[u] = 0

        self.height[self.source] = self.residual.v()

        for edge in self.graph.iteroutedges(self.source):
            self.excess[edge.target] += edge.weight
```

```
            self.excess[edge.source] -= edge.weight
            self.flow[edge.source][edge.target] += edge.weight
            self.flow[edge.target][edge.source] -= edge.weight

    def _discharge(self, u):
        while self.excess[u] > 0:
            idx = self.neighbor_idx[u]
            if idx == len(self.neighbors[u]):
                # Cannot push, so we must relabel and reset index.
                self._relabel(u)
                self.neighbor_idx[u] = 0
                continue

            edge = self.neighbors[u][idx]
            v = edge.target
            cap = edge.weight - self.flow[u][v]

            # We can add flow only if c_f > 0 and h[u] = h[v] + 1.
            if cap > 0 and self.height[u] == self.height[v] + 1:
                self._push(edge)
            else:
                self.neighbor_idx[u] += 1

    def _push(self, edge):
        cap = edge.weight - self.flow[edge.source][edge.target]
        d_flow = min(self.excess[edge.source], cap)
        self.flow[edge.source][edge.target] += d_flow
        self.flow[edge.target][edge.source] -= d_flow
        self.excess[edge.source] -= d_flow
        self.excess[edge.target] += d_flow

    def _relabel(self, u):
        min_h = float('inf')
        for edge in self.residual.iteroutedges(u):
            cap = edge.weight - self.flow[edge.source][edge.target]
            if cap > 0: # residual capacity exists
                min_h = min(min_h, self.height[edge.target])

        if min_h != float('inf'):
            self.height[u] = 1 + min_h
```

3.3. Modyfikacje algorytmu typu "prześlij-przemianuj"

Można zmodyfikować też wersję generyczną znajdującą się w klasie PushRelabelGeneric, aby wybierała do przetworzenia wierzchołki w uporządkowany sposób oraz ustalała wysokości wierzchołków przed rozpoczęciem *rozładowywania* wierzchołków. Często przyśpiesza to działanie algorytmu i wpływa pozytywnie na jego złożoność. Zaimplementowano więc dwie nowe klasy: PushRelabelFIFO i PushRelabelFIFOWithBFS, będące modyfikacją klasy PushRelabelGeneric.

Dane wejściowe: Sieć przepływowa, źródło i ujście.

Problem: Wyznaczenie maksymalnego przepływu.

Opis algorytmu: Oba algorytmy działają w bardzo podobny sposób, co wersja generyczna. Od wersji generycznej różnią się tym, że zamiast zbioru do przechowywania wierzchołków wykorzystywana jest kolejka typu FIFO (porządek "pierwszy przyjęty, pierwszy obsłużony"), dzięki czemu wierzchołki są przetwarzane w uporządkowanej kolejności. Drugą zmianą jest zastąpienie metody _push_or_relabel() metodą _discharge() z algorytmu typu "przemianuj i prześlij na początek", aby od razu *rozładować* cały nadmiar nagromadzony w wierzchołku. Na potrzeby tej metody inicjalizowane są dodatkowo w __init__() listy sąsiadów i aktualne pozycje na liście. Klasa PushRelabelFIFO różni się od klasy PushRelabelFIFOWithBFS tym, że w metodzie _initialize_preflow() wywoływana jest dodatkowo metoda _backward_bfs_heights(). Metoda ta przechodzi graf residualny w porządku BFS, ale w odwrotnej kolejności. Zaczynając od ujścia t, oblicza dokładne wysokości wierzchołków [18].

Złożoność: Złożoność czasowa algorytmów wynosi $O(V^2E)$.

Uwagi: Dla przyspieszenia metody _backward_bfs_heights() znajdującej się w klasie PushRelabelFIFOWithBFS użyto transponowanego grafu residualnego. Pozwala to na wykorzystanie metody iteroutedges(u), która jest szybsza niż iterinedges(u). Używanie transponowanego grafu residualnego jest szczególnie przydatne, gdy metoda _backward_bfs_heights() wykorzystywana będzie wielokrotnie, na przykład przy dalszym usprawnieniu algorytmu polegającym na użyciu heurystyki globalnego przemianowania (ang. *global relabeling heuristic*). Zgodnie z tą heurystyką, okresowo wykonuje się przeszukiwanie wstecz BFS, aby aktualizować wysokości wierzchołków, przyspieszając pracę algorytmu. Wraz z heurystyką luk (ang. *gap heuristic*), pozwalają one na pomijanie niepotrzebnych operacji *przemianowania*, które są wąskim gardłem algorytmu [18].

Listing 3.3. Modyfikacje algorytmu typu "prześlij-przemianuj".

```
    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__(n=self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            self.residual.add_node(node)

        # Initial capacities for the residual network.
        for edge in self.graph.iteredges():
            self.residual.add_edge(edge)  # original capacity
            self.residual.add_edge(Edge(edge.target, edge.source, 0))

        # Flow (at the end must be legal).
        self.flow = dict()
        for source in self.graph.iternodes():
            self.flow[source] = dict()
            for target in self.graph.iternodes():
```

```
                self.flow[source][target] = 0

        # Initial flow is zero.
        self.max_flow = 0
        self.height = dict()
        self.excess = dict()
        self.queue = collections.deque()
        self.in_queue = set()
        # Neighbor lists and current indices, from Cormen.
        self.neighbors = dict()
        for u in self.residual.iternodes():
            self.neighbors[u] = list(self.residual.iteroutedges(u))
        self.neighbor_idx = dict((u, 0) for u in self.residual.iternodes())

    def run(self, source, sink):
        """Executable pseudocode."""
        if source == sink:
            raise ValueError("source and sink are the same")
        self.source = source
        self.sink = sink
        self._initialize_preflow()

        # While we have nodes with excess flow, discharge them.
        while self.queue:
            u = self.queue.popleft()
            self.in_queue.remove(u)
            self._discharge(u)

        self.max_flow = self.excess[self.sink]

    def _discharge(self, u):
        while self.excess[u] > 0:
            idx = self.neighbor_idx[u]
            if idx == len(self.neighbors[u]):
                # Cannot push, so we must relabel and reset index.
                self._relabel(u)
                self.neighbor_idx[u] = 0
                continue

            edge = self.neighbors[u][idx]
            v = edge.target
            cap = edge.weight - self.flow[u][v]

            # We can add flow only if c_f > 0 and h[u] = h[v] + 1.
            if cap > 0 and self.height[u] == self.height[v] + 1:
                self._push(edge)
            else:
                self.neighbor_idx[u] += 1
```

Listing 3.4. Metoda _backward_bfs_heights().

```
    def _backward_bfs_heights(self):
        visited = dict()
        for node in self.residual.iternodes():
            self.height[node] = 0
            visited[node] = False
```

```
        visited[self.sink] = True
        queue = collections.deque([self.sink])

        reverse = self.residual.transpose()
        while queue:
            v = queue.popleft()

            for edge in reverse.iteroutedges(v):
                u = edge.target
                cap = edge.weight - self.flow[u][v]

                if cap > 0 and not visited[u]:
                    self.height[u] = self.height[v] + 1
                    visited[u] = True
                    queue.append(u)
```

3.4. Algorytm Busackera-Gowena

W tym rozdziale zostanie przeanalizowany algorytm rozwiązujący problem najtańszego przepływu, który ma wiele zastosowań praktycznych. Algorytm Busackera-Gowena, tak jak inne oparte na metodzie Forda-Fulkersona, wykorzystuje ścieżki powiększające [21]. Uwzględniając pojemności i koszty krawędzi, używa najtańszych ścieżek do wyznaczenia najtańszego przepływu o ustalonej wartości.

Dane wejściowe: Sieć przepływowa, zadana wartość przepływu, źródło i ujście.

Problem: Wyznaczenie najtańszego przepływu.

Opis algorytmu: Algorytm w nieskończonej pętli **while** wyszukuje najtańsze ścieżki powiększające i aktualizuje zmienne przechowujące aktualną wartość przepływu oraz minimalny koszt. Pętla wykonuje się do momentu, aż nie osiągniemy przepływu o zadanej wartości F lub nie istnieje żadna ścieżka powiększająca. Brak nowej ścieżki oznacza, że aktualny przepływ jest maksymalny. Jeżeli wartość F jest większa niż osiągnięty przepływ maksymalny, algorytm kończy się błędem, ponieważ rozwiązanie dla podanego F nie istnieje.

Złożoność: Złożoność czasowa algorytmu jest ograniczona przez $O(VE|f|)$, gdzie $|f|$ jest wartością zadanego przepływu.

Uwagi: Przygotowano dwie implementacje algorytmu Busackera-Gowena. Pierwsza implementacja znajduje najtańszą ścieżkę algorytmem Bellmana-Forda. Druga implementacja wykorzystuje szybszy algorytm SPFA (ang. *Shortest Path Faster Algorithm*). Listing przedstawia implementację z algorytmem SPFA.

Listing 3.5. Moduł busackergowen.

```
#!/usr/bin/env python3

import collections
from graphtheory.structures.edges import Edge


class BusackerGowenSPFA:
    """The Busacker–Gowen algorithm for computing the minimum cost flow
    with given flow value.

    Attributes
    ----------
    graph : input directed graph (flow network)
    residual : directed graph (residual network)
    flow : dict-of-dict
    source : node
    sink : node
    current_flow : number
    flow_value : number

    Notes
    -----
    Based on:

    Jacek Wojciechowski, Krzysztof Pienkosz, Grafy i sieci,
    Wydawnictwo Naukowe PWN, Warszawa 2013.
    """

    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__(n=self.graph.v(), directed=True)
        for node in self.graph.iternodes():
            self.residual.add_node(node)
        # Initial capacities and costs for the residual network.
        for edge in self.graph.iteredges():
            # Original capacity and cost
            self.residual.add_edge(edge)
            # Reverse edge with 0 capacity and negated cost (weight)
            rev_edge = Edge(edge.target, edge.source, -edge.weight)
            rev_edge.capacity = 0
            self.residual.add_edge(rev_edge)
        # Legal flow.
        self.flow = dict()
        for source in self.graph.iternodes():
            self.flow[source] = dict()
            for target in self.graph.iternodes():
                self.flow[source][target] = 0
        # Initial flow and cost are zeros.
        self.current_flow = 0
        self.min_cost = 0

    def run(self, source, sink, flow_value):
        """Executable pseudocode for finding the minimum cost flow from
```

```
        source to sink with a given flow value."""
        if source == sink:
            raise ValueError("source and sink are the same")
        if flow_value < 0:
            raise ValueError("flow value can't be negative")
        self.source = source
        self.sink = sink
        self.flow_value = flow_value
        while True:
            # Cost for flow with flow_value found, stop the algorithm
            if self.current_flow == self.flow_value:
                break

            min_capacity, min_cost = self._find_minimum_path()
            if min_capacity > 0:
                self.current_flow += min_capacity
                self.min_cost += min_cost
            # No augmenting paths and flow_value > max_flow
            elif self.flow_value > self.current_flow:
                raise ValueError(
                    f"flow_value cannot exceed max_flow ({self.current_flow})")

    def _find_minimum_path(self):
        """Finding the minimum cost augmenting path in the residual network."""
        # Shortest Path Faster Algorithm.
        parent = dict((node, None) for node in self.residual.iternodes())
        cost = dict((node, float("inf")) for node in self.residual.iternodes())
        cost[self.source] = 0

        # Capacity of found path to node.
        capacity = {self.source: float("inf")}

        queue = collections.deque()
        queue.append(self.source)
        in_queue = dict((node, False) for node in self.residual.iternodes())
        in_queue[self.source] = True

        while len(queue) > 0:
            node = queue.popleft()
            in_queue[node] = False
            for edge in self.residual.iteroutedges(node):
                cap = edge.capacity - self.flow[edge.source][edge.target]
                if cap > 0:
                    alt = cost[edge.source] + edge.weight
                    if cost[edge.target] > alt:
                        cost[edge.target] = alt
                        parent[edge.target] = edge.source
                        capacity[edge.target] = min(capacity[edge.source], cap)

                        if not in_queue[edge.target]:
                            queue.append(edge.target)
                            in_queue[edge.target] = True

        if parent[self.sink] is None:
            assert cost[self.sink] == float("inf")
            return 0, 0   # no augmenting path
```

```
        # Backtrack search and update flow
        new_flow = capacity[self.sink]
        if self.current_flow + new_flow > self.flow_value:
            new_flow = self.flow_value - self.current_flow

        target = self.sink
        while target != self.source:
            node = parent[target]
            self.flow[node][target] += new_flow
            self.flow[target][node] -= new_flow
            target = node

        return new_flow, new_flow * cost[self.sink]
```

3.5. Algorytm SPFA do wyznaczania najkrótszej ścieżki

Wyznaczanie najtańszej ścieżki jest tożsame z wyznaczaniem najkrótszej ścieżki, gdzie odległość między wierzchołkami może być traktowana jako koszt pomiędzy nimi. W problemach związanych z przepływem, wagi krawędzi mogą być ujemne. Algorytmem służącym do znajdowania najkrótszych ścieżek z ujemnymi wagami krawędzi jest algorytm Bellmana-Forda. Algorytm ten jest już zaimplementowany w bibliotece graphtheory i znajduje się w module graphtheory.shortestpaths. Algorytm SPFA to ulepszenie algorytmu Bellmana-Forda, gdzie zamiast przechodzić przez wszystkie krawędzie $|V|-1$ razy, przetwarzane są tylko wierzchołki, które zostały poddane relaksacji. Proces relaksacji służy do skrócenia aktualnie estymowanej najkrótszej ścieżki. W procesie relaksacji do ścieżki dodawany jest wierzchołek, jeżeli przechodząc przez niego nowa ścieżka jest krótsza niż estymowana. W ramach niniejszej pracy, oprócz użycia algorytmu SPFA w module busackergowen do wyznaczania najtańszych ścieżek, ten algorytm zaimplementowano również jako osobny moduł.

Dane wejściowe: Graf ważony skierowany (dozwolone są ujemne wagi), wierzchołek początkowy s (źródło).

Problem: Wyznaczenie najkrótszych ścieżek z wierzchołka początkowego s do pozostałych wierzchołków grafu. W przypadku, gdy w grafie występuje ujemny cykl, algorytm kończy się komunikatem o wykryciu takiego cyklu.

Opis algorytmu: Algorytm rozpoczyna się od stworzenia kolejki do przechowywania wierzchołków, które były końcami relaksowanych krawędzi. Kolejka działa w porządku FIFO, czyli "pierwszy przyjęty, pierwszy obsłużony". Na początku, jedynym elementem tej kolejki jest wierzchołek początkowy s (źródło). Następnie w każdej iteracji pętli wyciągany jest jeden wierzchołek z kolejki. Każda krawędź wychodząca z tego wierzchołka poddawana jest procesowi relaksacji. W przypadku, gdy relaksacja zakończy się pomyślnie, wierzchołek końcowy tej krawędzi zostaje dodany do kolejki tylko jeśli jeszcze nie znajduje się w kolejce. Pętla kończy się w momencie, gdy kolejka stanie się pusta, co oznacza że nie ma kolejnych wierzchołków do przetworzenia. Na

końcu działania algorytmu sprawdzany jest brak występowania ujemnego cyklu. Ten krok sprawdzający nie wpływa na końcową złożoność algorytmu.

Złożoność: Pesymistyczna złożoność czasowa algorytmu wynosi $O(VE)$, ponieważ w najgorszym przypadku konieczne będzie przetworzenie wszystkich krawędzi tak jak w algorytmie Bellmana-Forda. W dużej liczbie przypadków nie jest konieczne jednak przechodzenie przez wszystkie krawędzie i algorytm zakończy się szybciej. Średnia złożoność czasowa jest mniejsza i wynosi $O(E)$.

Uwagi: Najkrótsza ścieżka jest zdefiniowana tylko wtedy, gdy w grafie nie występuje ujemny cykl. Przed zakończeniem działania algorytmu sprawdzany jest brak istnienia takiego cyklu. Dodatkowo, jako że kolejka `collections.deque` nie posiada możliwości sprawdzenia, czy dany element jest w tej kolejce, użyto słownika `in_queue` do przechowywania informacji, czy wierzchołki są w kolejce.

Listing 3.6. Moduł `spfa`.

```
#!/usr/bin/env python3

import collections


class ShortestPathFasterAlgorithm:
    """The SPFA algorithm for the shortest path problem.

    Attributes
    ----------
    graph : input directed weighted graph
    parent : dict with nodes (shortest path tree)
    distance : dict with nodes (distances to source node)
    source : node

    Examples
    --------
    >>> from graphtheory.structures.edges import Edge
    >>> from graphtheory.structures.graphs import Graph
    >>> from graphtheory.shortestpaths.spfa import ShortestPathFasterAlgorithm
    >>> G = Graph(n=10, directed=True)    # an exemplary directed graph
    # Add nodes and edges here.
    >>> algorithm = ShortestPathFasterAlgorithm(G)     # initialization
    >>> algorithm.run(source)     # calculations
    >>> algorithm.parent   # shortest path tree as a dict
    >>> algorithm.distance[target]   # distance from source to target
    >>> algorithm.path(target)   # path from source to target
    >>> algorithm.path_iter(target)   # path from source to target

    Notes
    -----
    Based on:

    https://en.wikipedia.org/wiki/Bellman-Ford_algorithm#Improvements

    https://www.geeksforgeeks.org/dsa/shortest-path-faster-algorithm/
```

```
    """

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

        Parameters
        ----------
        graph : directed weighted graph
        """
        if not graph.is_directed():
            raise ValueError("the graph is not directed")
        self.graph = graph
        # Shortest path tree as a dictionary.
        self.parent = dict(((node, None) for node in self.graph.iternodes()))
        self.distance = dict(((node, float("inf"))
                              for node in self.graph.iternodes()))
        self.source = None

    def run(self, source):
        """Finding shortest paths from the source.

        Parameters
        ----------
        source : node
        """
        self.source = source
        self.distance[source] = 0

        queue = collections.deque()
        queue.append(self.source)
        in_queue = dict((node, False) for node in self.graph.iternodes())
        in_queue[self.source] = True

        while len(queue) > 0:   # worstcase: O(V) time
            node = queue.popleft()
            in_queue[node] = False
            for edge in self.graph.iteroutedges(node):   # O(E) time
                if self._relax(edge):
                    if not in_queue[edge.target]:
                        queue.append(edge.target)
                        in_queue[edge.target] = True
        # Check for negative cycles.
        for edge in self.graph.iteredges():   # O(E) time
            if (self.distance[edge.target] >
                    self.distance[edge.source] + edge.weight):
                raise ValueError("negative cycle detected")

    def _relax(self, edge):
        """Edge relaxation."""
        alt = self.distance[edge.source] + edge.weight
        if self.distance[edge.target] > alt:
            self.distance[edge.target] = alt
            self.parent[edge.target] = edge.source
            return True
        return False

    def path(self, target):
```

```
        """Construct a path from source to target."""
        if self.source == target:
            return [self.source]
        elif self.parent[target] is None:
            raise ValueError("no path to target")
        else:
            return self.path(self.parent[target]) + [target]

    def path_iter(self, target):
        """Construct a path from source to target."""
        if self.distance[target] == float("inf"):
            raise ValueError("no path to target")
        path = [target]
        while self.parent[target] is not None:
            target = self.parent[target]
            path.append(target)
        path.reverse()
        return path
```

4. Podsumowanie

W niniejszej pracy rozważano dwa główne problemy związane z sieciami przepływowymi: problem maksymalnego przepływu i problem najtańszego przepływu. W obu przypadkach zaprezentowano wydajne algorytmy służące do rozwiązywania tych problemów oraz przetestowano implementacje algorytmów w języku Python.

Dla problemu maksymalnego przepływu zbadano zaawansowany algorytm typu "prześlij-przemianuj", działający w czasie $O(V^2E)$, oraz algorytm typu "przemianuj i prześlij na początek", działający w czasie $O(V^3)$. W testach potwierdzono teoretyczną złożoność obliczeniową, ale także ujawniono słabe strony algorytmów. Okazało się, że dla gęstych sieci przepływowych i sieci z dużymi różnicami między pojemnościami krawędzi, algorytmy wykorzystujące ścieżki powiększające (np. algorytm Edmondsa-Karpa) działają stabilniej i często szybciej, mimo teoretycznie gorszej złożoności. Być może to była przyczyna, dla której te dwa algorytmy (oznaczone pierwotnie gwiazdką jako trudne) usunięto z nowych wydań słynnej książki Cormena i innych [2].

Dla problemu najtańszego przepływu zbadano algorytm Busackera-Gowena. Tutaj krawędzie sieci przepływowej miały dwa ważne atrybuty: pojemność i koszt. Jednym z etapów algorytmu Busackera-Gowena jest wyszukiwanie najtańszej ścieżki w oparciu o atrybut kosztu, co wymaga użycia algorytmu znajdującego najkrótsze ścieżki z jednego źródła. Zbadano zastosowanie algorytmu Bellmana-Forda (możliwe są ujemne koszty krawędzi) i algorytmu SPFA. Zostało potwierdzone, że algorytm SPFA jest szybszy. Przygotowano niezależną implementację algorytmu SPFA, która dołączyła do implementacji innych algorytmów wyznaczających najkrótsze ścieżki z jednego źródła.

Ważnym elementem badania zachowania algorytmów są praktyczne testy wydajnościowe. W ramach pracy przygotowano dwa nowe generatory gęstych sieci przepływowych, które dały lepszy wgląd w pracę algorytmów. Inna ciekawa obserwacja to elastyczność klasy `Graph` z biblioteki `graphtheory`. Klasa przechowuje krawędzie grafu niezależnie od wzbogacenia krawędzi o dodatkowe atrybuty. Domyślnie krawędzie mają trzy atrybuty: `source`, `target` oraz `weight`. Algorytmy dla problemu najtańszego przepływu po raz pierwszy wykorzystały dodatkowy atrybut `capacity` do przechowywania pojemności krawędzi, a atrybut `weight` przechowywał koszt krawędzi.

A. Testy algorytmów

Dodatek A poświęcony jest wynikom testów przeprowadzonych dla algorytmów związanych z problemami maksymalnego i najtańszego przepływu, a także algorytmów Bellmana-Forda oraz SPFA do wyszukiwania najkrótszych ścieżek, które zostały użyte w algorytmie Busackera-Gowena. Dla wszystkich algorytmów i ich modyfikacji, zaimplementowanych w ramach niniejszej pracy, wykonano testy poprawnościowe i wydajnościowe. Do testów poprawnościowych wykorzystano testy jednostkowe z modułu unittest [22], natomiast wydajność została sprawdzona z pomocą modułu timeit [23].

Testy wydajnościowe wykonano również dla algorytmów związanych z problemem maksymalnego przepływu, które aktualnie znajdują się w bibliotece graphtheory, czyli algorytmów Forda-Fulkersona, Edmondsa-Karpa i Dinica, aby mieć spójne dane pochodzące z tego samego urządzenia.

W związku z losowością i przypadkowością wyników, na które mogą mieć wpływ procesy zachodzące w systemie operacyjnym, dla każdego algorytmu wykonano trzy próby, natomiast przy analizie wyników uwzględniano tylko medianę. Do generowania sieci przepływowych wykorzystano metodę make_flow_network z klasy GraphFactory będącą częścią biblioteki graphtheory. Z uwagi na to, że metoda ta tworzy rzadkie sieci, w ramach tej pracy zaimplementowano także dwa nowe generatory tworzące gęste sieci przepływowe - make_complete_flow oraz make_transitive_tournament.

A.1. Testowanie algorytmu typu "prześlij-przemianuj"

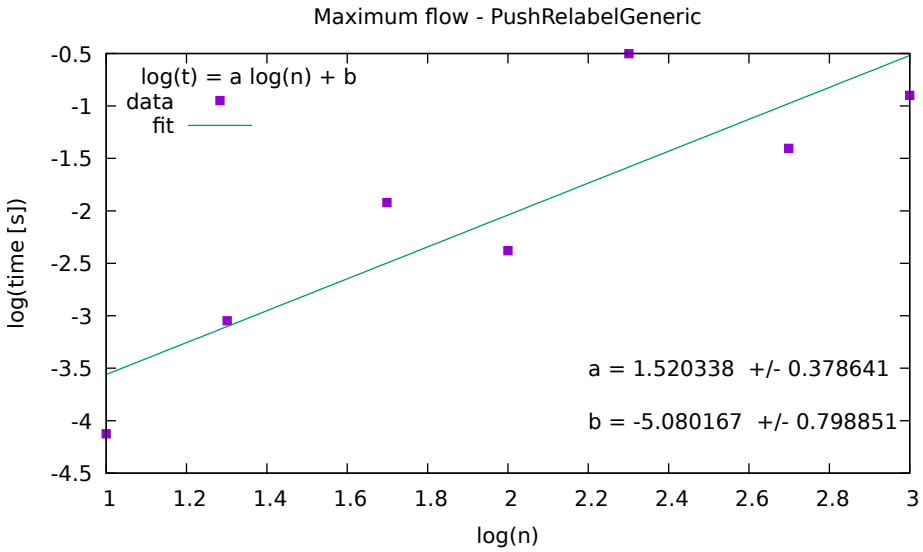
Algorytm generyczny typu "prześlij-przemianuj" testowano z wykorzystaniem trzech generatów wymienionych we wstępie tego dodatku:

— make_flow_network, którego wyniki znajdują się na rys. A.1.
— make_complete_flow, którego wyniki znajdują się na rys. A.2.
— make_transitive_tournament, którego wyniki znajdują się na rys. A.3.

Uzyskano wyniki znacznie lepsze niż zakładałaby teoria. Zgodnie z teorią, złożoność algorytmu wynosi $O(V^2E)$. Wyniki porównano z czasami otrzymanymi dla algorytmu Dinica, który również ma taką samą złożoność teoretyczną i znajduje się już w bibliotece graphtheory. Rezultaty testów dla algorytmu Dinica z wykorzystaniem wymienionych generatorów znajdują się odpowiednio na rys. A.4, A.5 i A.6. Algorytm Dinica również uzyskał wartości mniejsze niż wynikałoby to z teorii. Może to wskazywać, że generatory nie tworzą wystarczająco trudnych sieci dla tych algorytmów, a teoretyczna złożoność mówi o przypadkach pesymistycznych, które nie występują często.

Porównując wyniki dla PushRelabelGeneric oraz Dinic, dla dwóch pierwszych generatorów zaobserwowano poprawę, szczególnie różnica dla make_complete_flow

jest znacząca. Dla make_transitive_tournament algorytm Dinica uzyskał nieznacznie lepszy współczynnik.



Rysunek A.1. Wykres wydajności algorytmu PushRelabelGeneric do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_flow_network. Dla sieci rzadkich, gdzie liczba krawędzi jest zbliżona do liczby wierzchołków, algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 3.

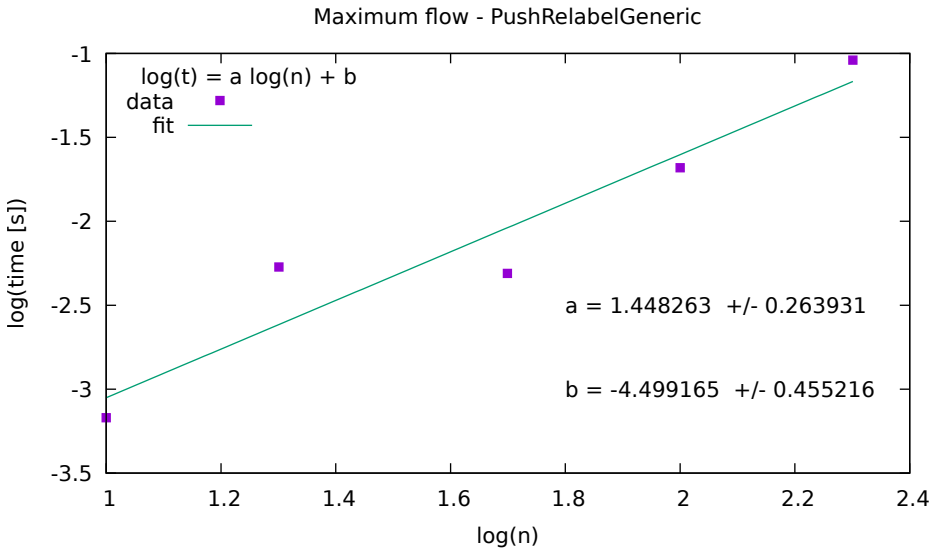
A.2. Testowanie algorytmu typu "przemianuj i prześlij na początek"

Również algorytm typu "przemianuj i prześlij na początek" testowano z wykorzystaniem trzech generatorów:

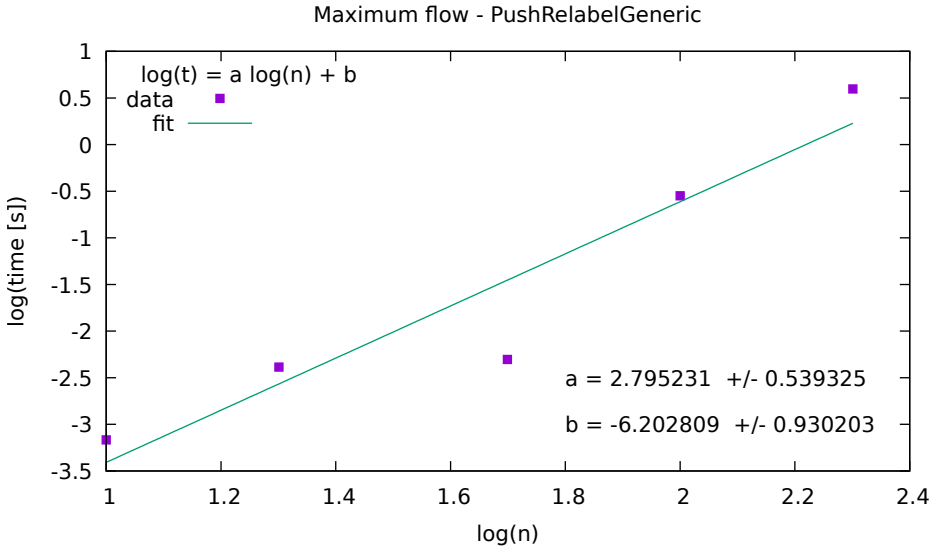
— make_flow_network (rys. A.7),
— make_complete_flow (rys. A.8),
— make_transitive_tournament (rys. A.9).

Dla dwóch pierwszych generatorów uzyskano wyniki znacznie lepsze niż zakładałaby teoria, zgodnie z którą złożoność wynosi $O(V^3)$. Dla ostatniego generatora wyniki dalej są lepsze, ale tutaj bardziej zbliżone do złożoności teoretycznej.

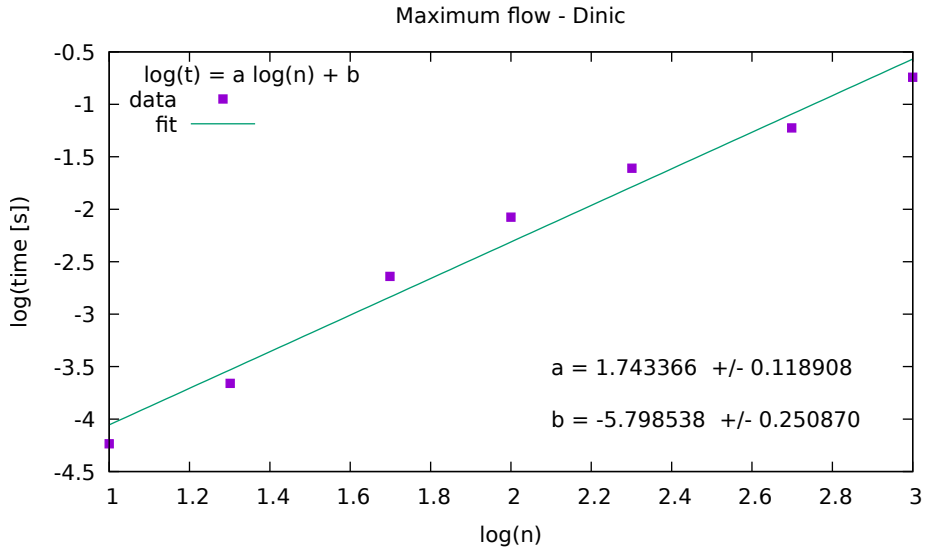
Porównując wyniki do rezultatów algorytmu generycznego, dla rzadkich sieci uzyskiwanych z make_flow_network, PushRelabelToFront osiąga zauważalnie gorszą złożoność niż wersja generyczna. Dla make_complete_flow wyniki są porównywalne, natomiast dla make_transitive_tournament osiąga lepszą złożoność. Zgodnie z teorią, dla gęstych sieci algorytm typu "przemianuj i prześlij na początek" powinien osiągać znacznie lepsze wyniki w porównaniu do wersji generycznej i niż pokazują to testy, które nie potwierdziły przewagi tego algorytmu dla grafów, w których liczba krawędzi jest znacząco większa od liczby



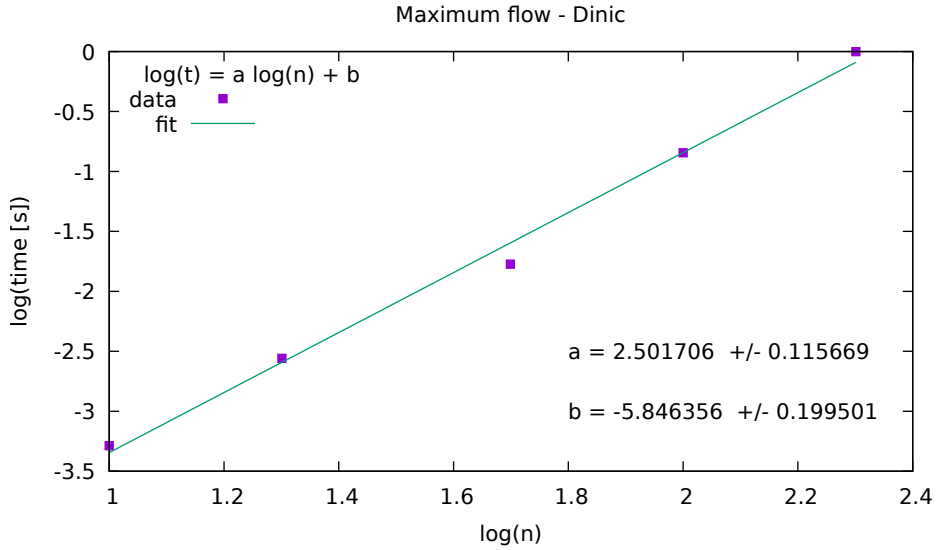
Rysunek A.2. Wykres wydajności algorytmu PushRelabelGeneric do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_complete_flow. Dla sieci gęstych generowanych przez tę metodę, gdzie liczba krawędzi jest w przybliżeniu równa kwadratowi liczby wierzchołków, algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 4.



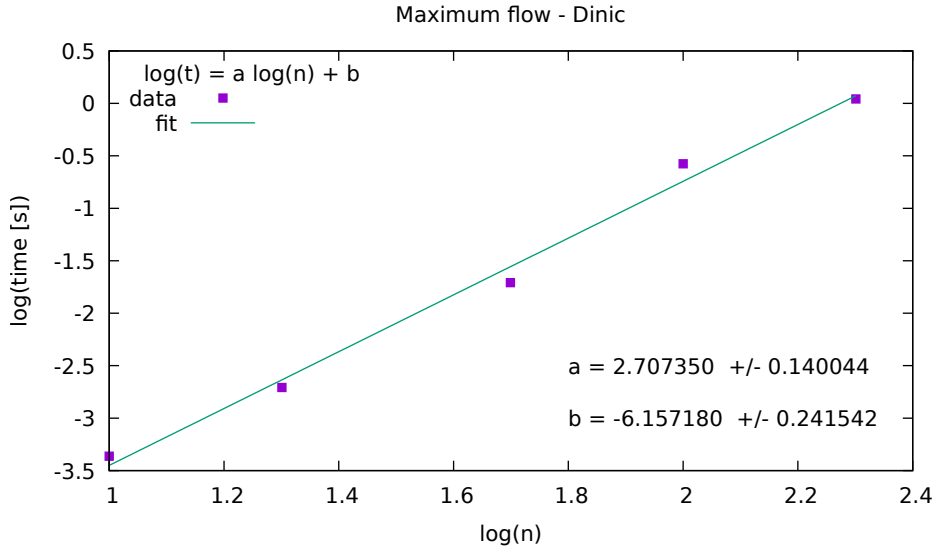
Rysunek A.3. Wykres wydajności algorytmu PushRelabelGeneric do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_transitive_tournament. Dla sieci gęstych generowanych przez tę metodę, gdzie liczba krawędzi jest w przybliżeniu równa kwadratowi liczby wierzchołków, algorytm osiąga złożoność mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 4.



Rysunek A.4. Wykres wydajności algorytmu Dinic do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_flow_network. Dla sieci rzadkich, gdzie liczba krawędzi jest zbliżona do liczby wierzchołków, algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 3.



Rysunek A.5. Wykres wydajności algorytmu Dinic do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_complete_flow. Dla sieci gęstych generowanych przez tę metodę, gdzie liczba krawędzi jest w przybliżeniu równa kwadratowi liczby wierzchołków, algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 4.



Rysunek A.6. Wykres wydajności algorytmu Dinic do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_transitive_tournament. Dla sieci gęstych generowanych przez tę metodę, gdzie liczba krawędzi jest w przybliżeniu równa kwadratowi liczby wierzchołków, algorytm osiąga złożoność mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 4.

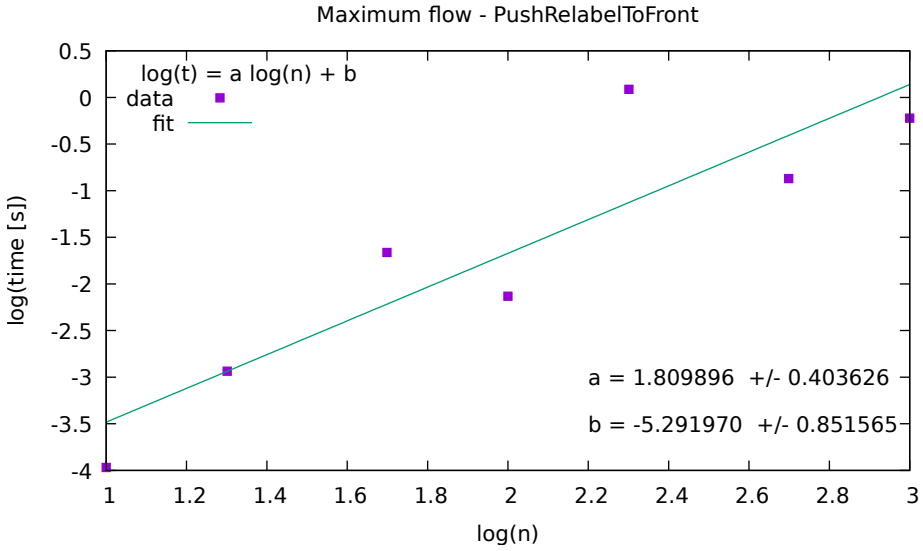
wierzchołków. Wpływ na to może mieć wykonywanie niepotrzebnych operacji *przemianowania*, które są *wąskim gardłem* algorytmu, jak wspomniano w rozdziale 3.3 i możliwe, że wpływa to na wyniki bardziej niż w przypadku wersji generycznej.

A.3. Testowanie modyfikacji algorytmu typu "prześlij-przemianuj"

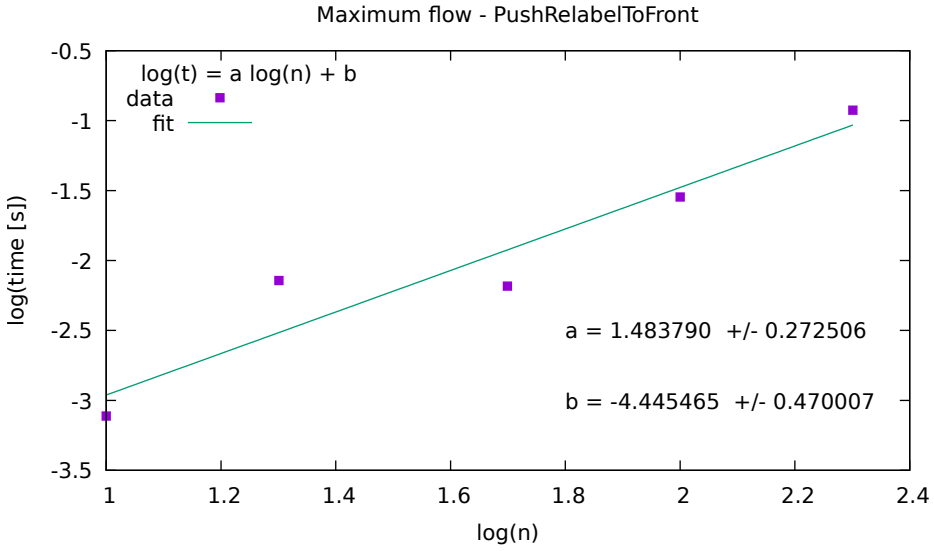
Tak jak w przypadku pozostałych algorytmów typu "prześlij-przemianuj", do testowania użyto trzech generatorów. Dla PushRelabelFIFO wyniki znajdują się na rys. A.10 (make_flow_network), A.11 (make_complete_flow) i A.12 (make_transitive_tournament), natomiast dla PushRelabelFIFOWithBFS odpowiednio na rys. A.13, A.14 oraz A.15. Zgodnie z teorią, złożoności tych algorytmów wynoszą $O(V^2E)$, tak samo jak dla wersji generycznej.

Wyniki testów wydajnościowych obu modyfikacji porównano z rezultatami otrzymanymi dla wersji generycznej. W przypadku PushRelabelFIFO osiągnięto porównywalne czasy dla wszystkich generatorów, z zaniedbywalną przewagą algorytmu generycznego. Dla PushRelabelFIFOWithBFS wyniki dla generatorów make_flow_network i make_transitive_tournament również są porównywalne, natomiast dla make_complete_flow widać, że wersja generyczna osiągała nieznacznie lepsze wyniki.

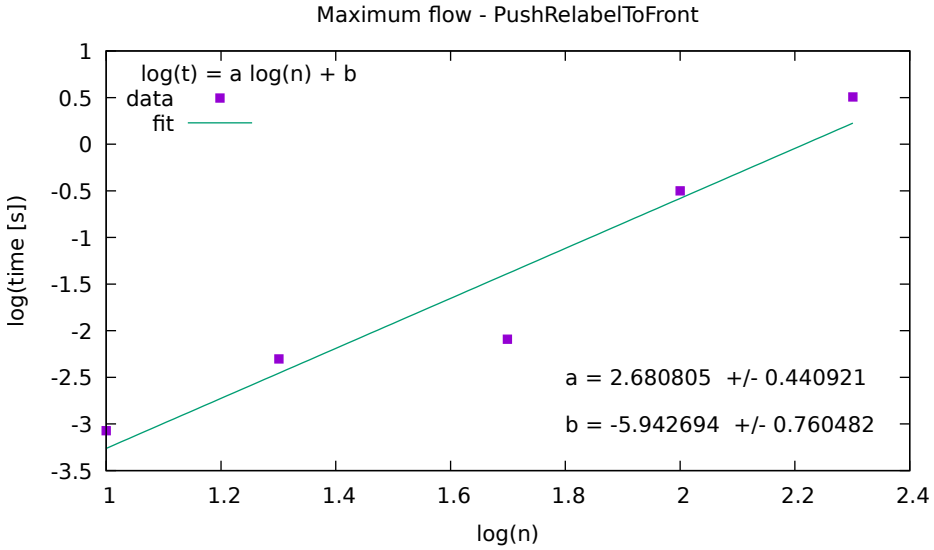
Nie zauważono więc poprawy szybkości względem wersji generycznej, co może wskazywać, że modyfikacje nie mają pozytywnego wpływu na wersję



Rysunek A.7. Wykres wydajności algorytmu PushRelabelToFront do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_flow_network. Algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^3)$. Współczynnik a powinien wynosić w przybliżeniu 3.



Rysunek A.8. Wykres wydajności algorytmu PushRelabelToFront do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_complete_flow. Algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^3)$. Współczynnik a powinien wynosić w przybliżeniu 3.



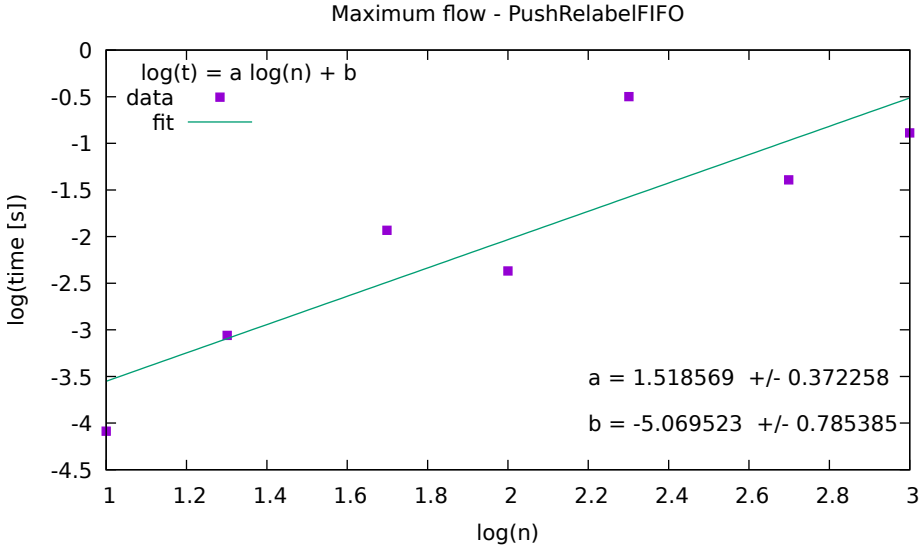
Rysunek A.9. Wykres wydajności algorytmu PushRelabelToFront do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_transitive_tournament. Algorytm osiąga złożoność mniejszą niż teoretyczne $O(V^3)$, lecz do niej zbliżoną, ponieważ współczynnik a powinien wynosić w przybliżeniu 3.

generyczną, a wskazówki zawarte w literaturze [18] mogą dotyczyć algorytmu typu "przemianuj i prześlij na początek", aby polepszyć jego wydajność.

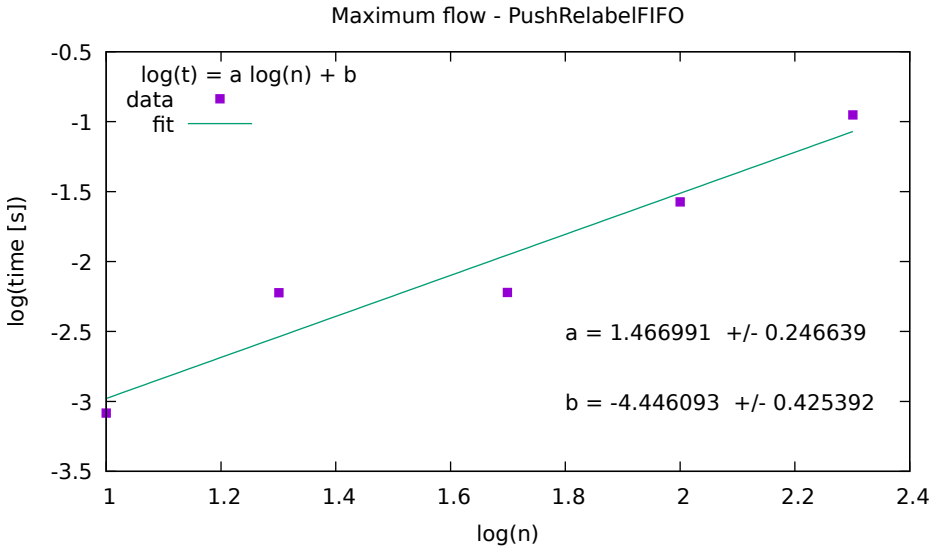
A.4. Testowanie najtańszego przepływu

Algorytmy BusackerGowenBF i BusackerGowenSPFA testowano z wykorzystaniem dwóch generatów wymienionych we wstępie tego dodatku - make_flow_network oraz make_complete_flow. Wyniki dla BusackerGowenBF znajdują się odpowiednio na rys. A.16 i A.17, a dla BusackerGowenSFPA odpowiednio na rys. A.18 i A.19.

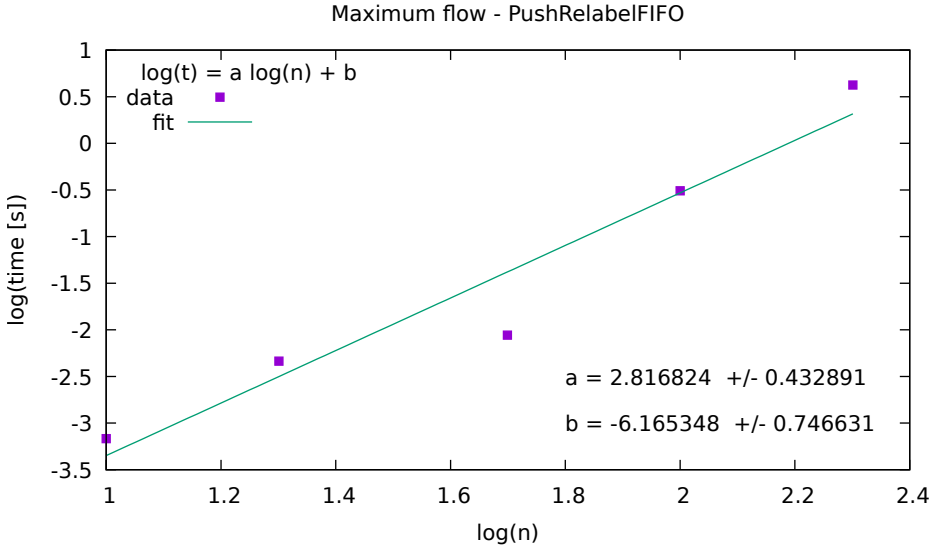
Uzyskane wyniki dla obu algorytmów są zgodne z teorią, która mówi, że algorytm Busackera-Gowena ma złożoność $O(VE|f|)$. Porównując współczynniki między tymi algorytmami, dla obu generatorów jest widoczne, że użycie algorytmu SPFA do wyznaczania najkrótszych ścieżek jest zdecydowanie lepsze i przyśpiesza pracę algorytmu Busackera-Gowena, ponieważ w obu przypadkach współczynnik a dla algorytmu SPFA jest mniejszy niż odpowiadający mu współczynnik dla algorytmu Bellmana-Forda.



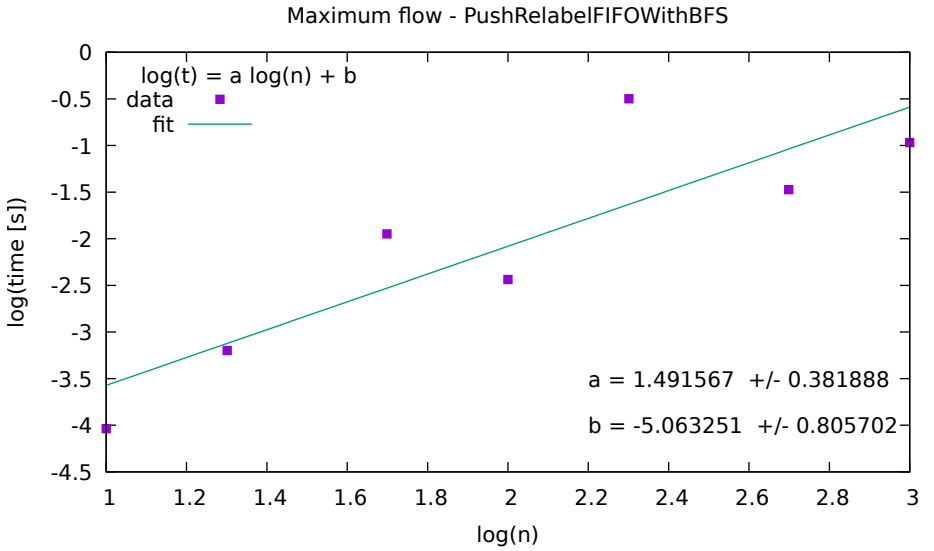
Rysunek A.10. Wykres wydajności algorytmu PushRelabelFIFO do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_flow_network. Dla sieci rzadkich, gdzie liczba krawędzi jest zbliżona do liczby wierzchołków, algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 3.



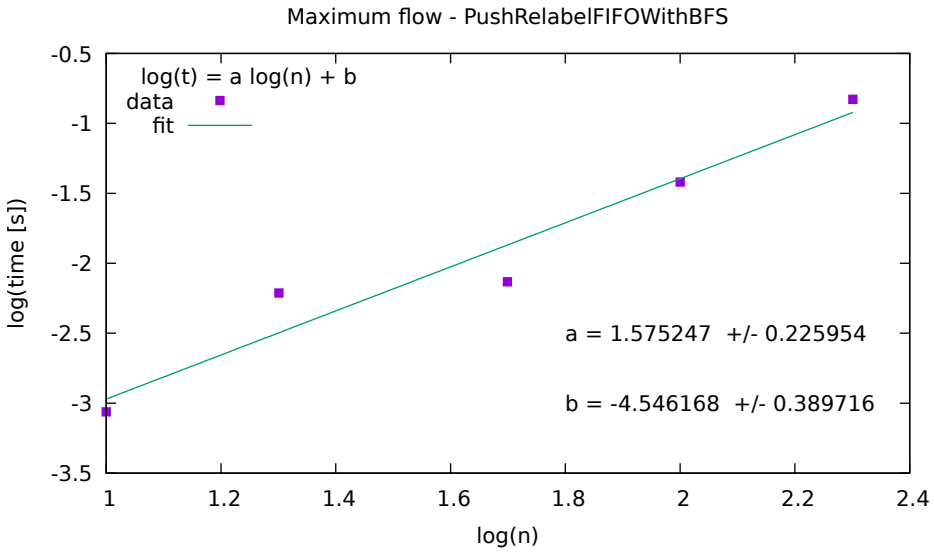
Rysunek A.11. Wykres wydajności algorytmu PushRelabelFIFO do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_complete_flow. Dla sieci gęstych generowanych przez tę metodę, gdzie liczba krawędzi jest w przybliżeniu równa kwadratowi liczby wierzchołków, algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 4.



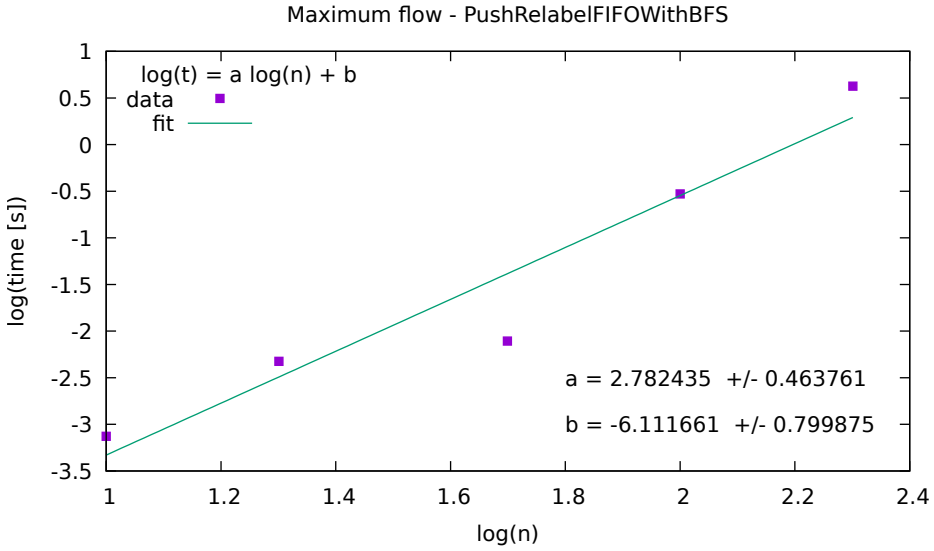
Rysunek A.12. Wykres wydajności algorytmu PushRelabelFIFO do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_transitive_tournament. Dla sieci gęstych generowanych przez tę metodę, gdzie liczba krawędzi jest w przybliżeniu równa kwadratowi liczby wierzchołków, algorytm osiąga złożoność mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 4.



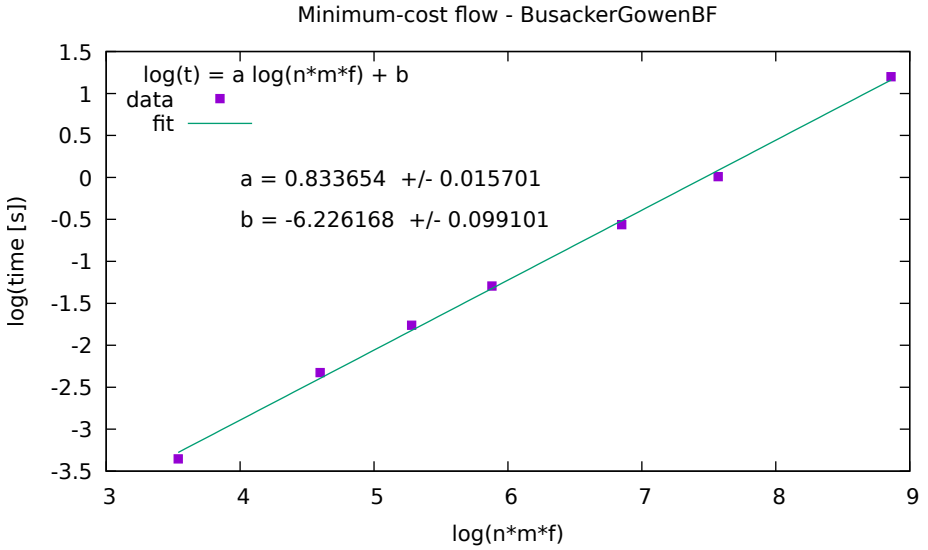
Rysunek A.13. Wykres wydajności algorytmu PushRelabelFIFOWithBFS do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_flow_network. Dla sieci rzadkich, gdzie liczba krawędzi jest zbliżona do liczby wierzchołków, algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 3.



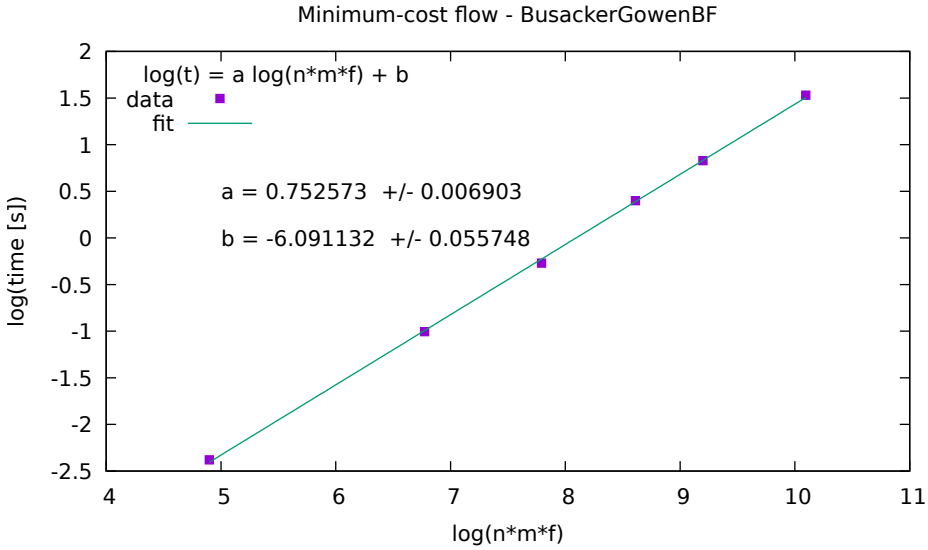
Rysunek A.14. Wykres wydajności algorytmu PushRelabelFIFOWithBFS do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_complete_flow. Dla sieci gęstych generowanych przez tę metodę, gdzie liczba krawędzi jest w przybliżeniu równa kwadratowi liczby wierzchołków, algorytm osiąga złożoność znacznie mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 4.



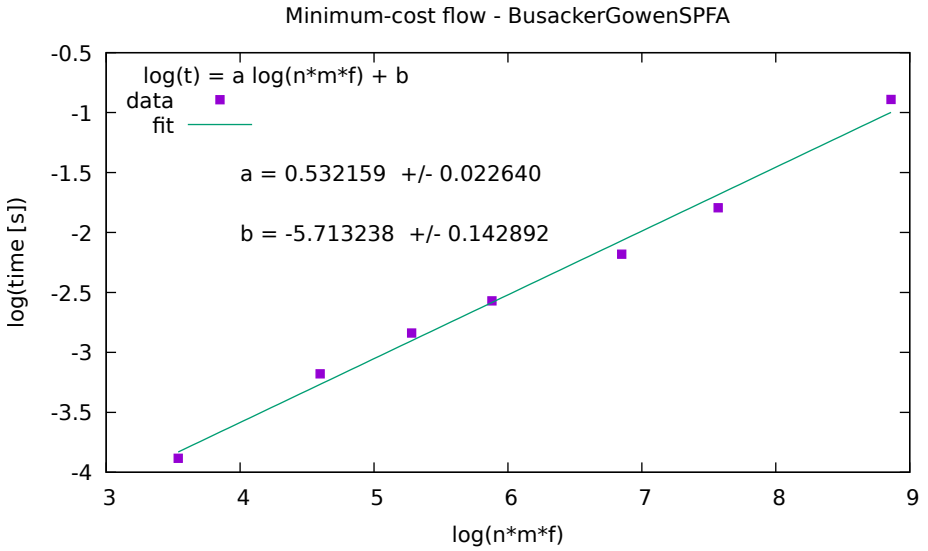
Rysunek A.15. Wykres wydajności algorytmu PushRelabelFIFOWithBFS do znajdowania maksymalnego przepływu wykorzystując sieci przepływowe generowane przez make_transitive_tournament. Dla sieci gęstych generowanych przez tę metodę, gdzie liczba krawędzi jest w przybliżeniu równa kwadratowi liczby wierzchołków, algorytm osiąga złożoność mniejszą niż teoretyczne $O(V^2E)$. Współczynnik a powinien wynosić w przybliżeniu 4.



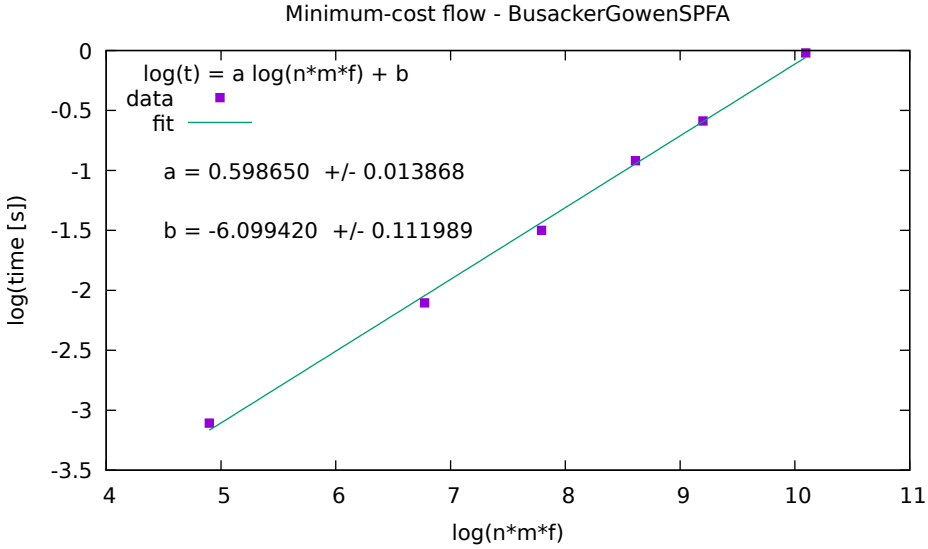
Rysunek A.16. Wykres wydajności algorytmu BusackerGowenBF do znajdowania najtańszego przepływu wykorzystując sieci przepływowe generowane przez make_flow_network. Złożoność tego algorytmu wynosi $O(VE|f|)$, dlatego więc współczynnik a powinien wynosić nie więcej niż 1. Współczynnik a = 0.834(16) potwierdza teoretyczną złożoność algorytmu.



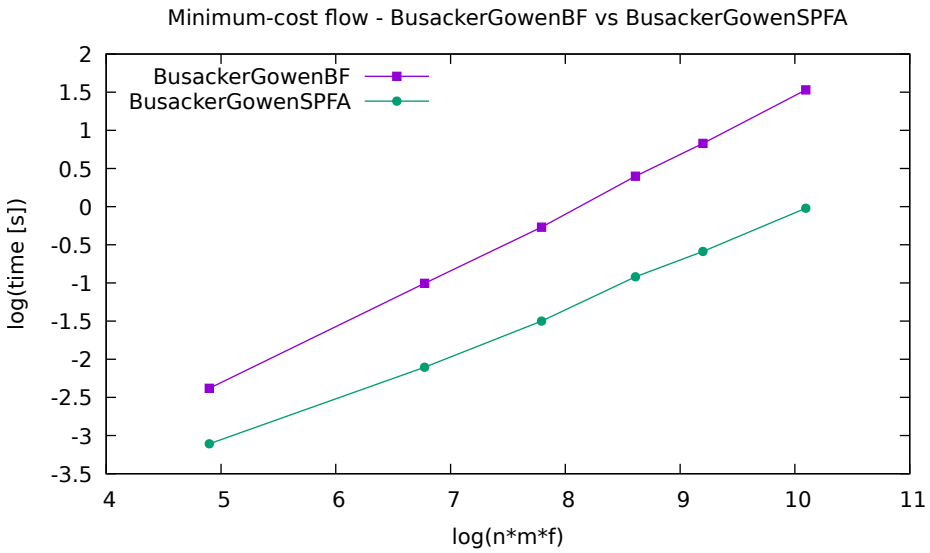
Rysunek A.17. Wykres wydajności algorytmu BusackerGowenBF do znajdowania najtańszego przepływu wykorzystując sieci przepływowe generowane przez make_complete_flow. Złożoność tego algorytmu wynosi $O(VE|f|)$, dlatego więc współczynnik a powinien wynosić nie więcej niż 1. Współczynnik a = 0.7526(70) potwierdza teoretyczną złożoność algorytmu.



Rysunek A.18. Wykres wydajności algorytmu BusackerGowenSPFA do znajdowania najtańszego przepływu wykorzystując sieci przepływowe generowane przez make_flow_network. Złożoność tego algorytmu wynosi $O(VE|f|)$, dlatego więc współczynnik a powinien wynosić nie więcej niż 1. Współczynnik a = 0.532(23) potwierdza teoretyczną złożoność algorytmu i wyższość nad BusackerGowenBF (rys. A.16).



Rysunek A.19. Wykres wydajności algorytmu BusackerGowenSPFA do znajdowania najtańszego przepływu wykorzystując sieci przepływowe generowane przez make_complete_flow. Złożoność tego algorytmu wynosi $O(VE|f|)$, dlatego więc współczynnik a powinien wynosić nie więcej niż 1. Współczynnik a = 0.599(14) potwierdza teoretyczną złożoność algorytmu i wyższość nad BusackerGowenBF (rys. A.17).



Rysunek A.20. Porównanie wydajności algorytmu BusackerGowen do znajdowania najtańszego przepływu w zależności od użytej metody wyznaczania najkrótszych ścieżek. Wersja algorytmu BusackerGowenSPFA osiąga lepsze wyniki, co potwierdza wyższość algorytmu SPFA nad algorytmem Bellmana-Forda.

Bibliografia

[1] Wikipedia, Flow network, 2026, https://en.wikipedia.org/wiki/Flow_network.

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

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

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

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

[6] Andrzej Kapanowski, graphtheory, GitHub repository, 2026, https://github.com/ufkapano/graphtheory/.

[7] Łukasz Gałuszka, *Implementacja wybranych algorytmów dla grafów z wagami w języku Python*, Uniwersytet Jagielloński, Kraków 2014.

[8] Krzysztof Niedzielski, *Skojarzenia w teorii grafów*, Uniwersytet Jagielloński, Kraków 2017.

[9] Magdalena Stępień, *Wybrane algorytmy dla grafów planarnych*, Uniwersytet Jagielloński, Kraków 2019.

[10] Serwis Edukacyjny w I-LO w Tarnowie, https://eduinf.waw.pl/inf/alg/001_search/0144.php.

[11] Applications of Network Flow from Arup Guha's teachings at the University of Central Florida, Department of Computer Science, https://www.cs.ucf.edu/~dmarino/progcontests/modules/netflow2/NetFlow-Apps.pdf.

[12] Sariel Har-Peled's *Fundamental Algorithms* course at the University of Illinois Urbana-Champaign, The Grainger College of Engineering, Spring 2011, https://courses.grainger.illinois.edu/CS473/sp2011/Lectures/index.html.

[13] Jeff Erickson, *Algorithms*, 2019, https://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf.

[14] Lecture 20 from Shikha Singh's *Algorithm Design and Analysis* course at Williams College, Department of Computer Science, Spring 2020, https://www.cs.williams.edu/~shikha/teaching/spring20/cs256/lectures/Lecture20.pdf.

[15] Wikipedia, Ford–Fulkerson algorithm, 2026, https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm.

[16] Wikipedia, Edmonds–Karp algorithm, 2026, https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm.

[17] Wikipedia, Dinic's algorithm, 2026, https://en.wikipedia.org/wiki/Dinic%27s_algorithm.

[18] Wikipedia, Push–relabel maximum flow algorithm, 2026,

https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm.

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

[20] M. Henzinger, P. Klein, S. Rao, S. Subramanian, *Faster Shortest-Path Algorithms for Planar Graphs*, Journal of Computer and System Sciences 55(1), 3-23 (1997).

[21] R. G. Busacker, P. J. Gowen, *A Procedure for Determining a Family of Minimum-Cost Network Flow Patterns*, Operations Research Office Technical Report 15, Johns Hopkins University, Baltimore, Maryland, 1961.

[22] Python Programming Language - Official Documentation Website, https://docs.python.org/3/library/unittest.html.

[23] Python Programming Language - Official Documentation Website, https://docs.python.org/3/library/timeit.html.