UNIWERSYTET PEDAGOGICZNY
im. Komisji Edukacji Narodowej
W KRAKOWIE
Wydział Matematyczno-Fizyczno-Techniczny
Kierunek Informatyka, specjalność administracja systemami informatycznymi

JAKUB WĄSIK

ALGORYTMY WYZNACZAJĄCE NAJKRÓTSZE ŚCIEŻKI W GRAFACH WAŻONYCH

Praca licencjacka
napisana pod kierunkiem
dra Romana Czapli

KRAKÓW 2019

Spis treści

Wstęp

Celem niniejszej pracy jest przedstawienie algorytmów wyznaczających najkrótsze ścieżki w grafach ważonych oraz ich implementacji w języku Python. W pracy zostały zawarte podstawowe informacje na temat teorii grafów oraz wspominanych algorytmów. Teoria grafów to ważny i ciągle rozwijany dział matematyki i informatyki o wielu praktycznych zastosowaniach. Natomiast język Python to potężny i jednocześnie łatwy język programowania wspierający wiele paradygmatów programowania. Warto zaznaczyć, że w ostatnim czasie popularność języka Python rośnie bardzo dynamicznie. Ma to związek m.in. z rozwojem dziedziny wiedzy jaką jest sztuczna inteligencja, a przede wszystkim jej poddziedziny - uczenia maszynowego. Nie powinno zatem dziwić, że autor pracy wybrała właśnie ten język w celu implementacji omawianych algorytmów. W pracy możemy wyróżnić dwie główne części, pierwsza zawiera dokładne omówienie i teoretyczny opis tych algorytmów, w drugiej części zaprezentowano implementacje algorytmów oraz analizę i porównanie wspomnianych implementacji.

Rozdział 1

Podstawy teorii grafów

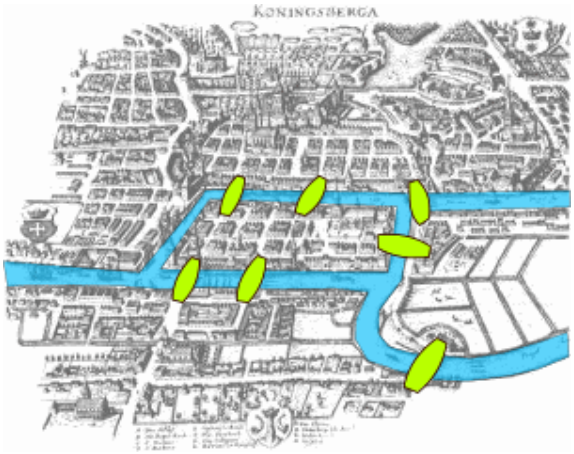
Zanim przejdziemy do opisu algorytmów wyznaczania najkrótszych ścieżek w niniejszym rozdziale wprowadzimy podstawowe pojęcia z teorii grafów, które będą stosowane w dalszej części pracy. Te informacje jak i ich uzupełnienie można znaleźć między innymi w [2, 3, 5].

1.1 Pojęcie grafu

Graf to struktura matematyczna opisująca pewien zbiór punktów i połączenia między nimi. Pojęcie grafu jest bardzo intuicyjne, dlatego w różnych źródłach można spotkać różne definicje grafu.

Historia powstania grafów sięga *XVIII* wieku, dokładniej w 1736 roku Leonhard Euler[1)] jako pierwszy przeprowadził badania rozwiązując *problem mostów królewieckich*. Problem ten dotyczył rzeki Pregoły przepływającej przez Królewiec (obecna, rosyjska nazwa to Kaliningrad) i siedmiu mostów pomiędzy brzegami rzeki oraz dwoma wyspami znajdującymi się w rozwidleniu rzeki (zob. rysunek 1.1). Problem, który postawił i rozwiązał Euler, był następujący: "*Czy można przejść kolejno przez wszystkie mosty tak, żeby każdy przekroczyć tylko raz?*". Euler dowiódł, że nie jest to możliwe ustalając przy tym pierwsze własności grafów.

[1)] Leonhard Euler (1707-1783) – szwajcarski matematyk i fizyk. Był prekursorem rozwoju w wielu obszarach obu tych nauk. Większą część swojego życia spędził w Rosji oraz Prusach. Jest uważany za jednego z najbardziej produktywnych matematyków w historii.

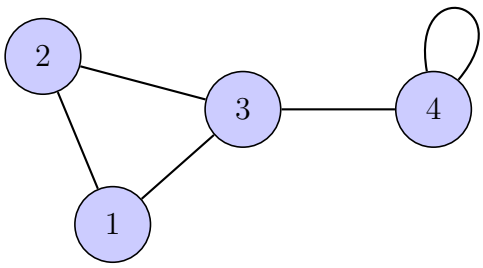


Rys. 1.1: Mapa przedstawiająca mosty królewieckie za życia Leonharda Eulera. Źródło: [7]

Definicja 1.1. *Grafem nieskierowanym* (*niezorientowanym*) G nazywamy parę (V, E) składającą się z niepustego i skończonego zbioru *wierzchołków* V oraz skończonego zbioru krawędzi E

$$E = \Big\{\{v_i, v_j\} \colon v_i, v_j \in V\Big\}.$$

Krawędzią w grafie nieskierowanym o wierzchołkach $u, v \in V$ nazywamy dwuelementowy zbiór $\{u, v\} \in E$ i mówimy, że krawędź $\{u, v\}$ jest *incydentna* do wierzchołków u i v. Do oznaczenia krawędzi grafu $\{u, v\}$ będziemy stosować notację (u, v) - w tym przypadku zapis (u, v) i (v, u) oznacza tą samą krawędź (zob. przykład na rysunku 1.2).

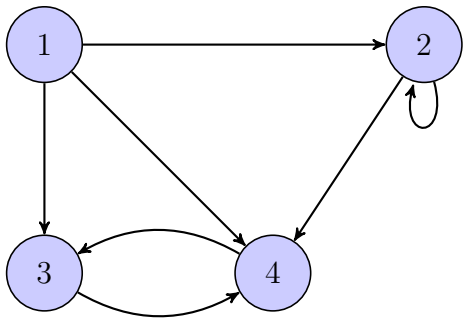


Rys. 1.2: Graf nieskierowany $G = (V, E)$, gdzie $V = \{1, 2, 3, 4\}$ i $E = \{(1, 2), (1, 3), (2, 3), (3, 4), (4, 4)\}$. Krawędź $(4, 4)$ jest pętlą własną

Definicja 1.2. *Grafem skierowanym* (*zorientowanym* lub *digrafem* z ang. *directed graph*) G nazywamy parę (V, E), gdzie E to zbiór krawędzi, a V to zbiór wierzchołków. W przypadku grafu skierowanego zbiór krawędzi definiujemy jako zbiór uporządkowanych par:

$$E = \Big\{(v_i, v_j) \colon v_i, v_j \in V\Big\}.$$

Krawędź grafu skierowanego nazywana jest też *łukiem*. Mówimy, że krawędź $e = (u, v) \in E$ jest *wychodząca* z wierzchołka u i *wchodząca* do wierzchołka v. (zob. przykład na rysunku 1.3).



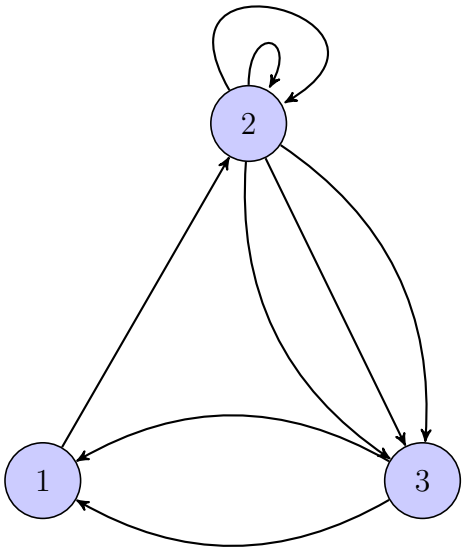
Rys. 1.3: Graf skierowany $G = (V, E)$, gdzie $V = \{1, 2, 3, 4\}$ i $E = \{(1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 4), (4, 3))\}$. Krawędź $(2, 2)$ jest pętlą własną, krawędzie $(3, 4)$ i $(4, 3)$ są różne.

Definicja 1.3. W grafie $G = (V, E)$, krawędź $e = (u, v) \in E$, dla której $u = v$ nazywamy *pętlą własną*. Powtarzające się krawędzie lub pętle grafu G (tzn. gdy E jest *multizbiorem*[2)]) nazywamy *równoległymi* lub *wielokrotnymi*. Graf bez krawędzi wielokrotnych i pętli nazywamy *grafem prostym*. Graf, który zawiera krawędzie lub pętle wielokrotne nazywamy *multigrafem* (zob. przykład na rysunku 1.4).

Definicja 1.4. *Drogą* (*ścieżką*) o długości k w grafie $G = (V, E)$ prowadzącą z wierzchołka u do wierzchołka u' nazywamy ciąg wierzchołków $(v_i)_{i=0}^{k}$ takich, że $u = v_0$, $u' = v_k$ i $(v_i, v_{i+1}) \in E$ dla $i = 0, 1, 2, \ldots, k-1$. *Droga prosta* to szczególny przypadek drogi, w której wszystkie wierzchołki są różne. *Długość drogi* to ilość jej krawędzi (lub wierzchołków nie licząc startowego). Jeżeli istnieje ścieżka p z wierzchołka u do wierzchołka v to mówimy, że v jest osiągalny z u, co zapisujemy symbolicznie $u \overset{p}{\rightsquigarrow} v$, jeżeli G jest grafem skierowanym.

Definicja 1.5. Jeżeli graf $G = (V, E)$ jest skierowany to drogę $p = (v_i)_{i=0}^{k}$ nazywamy *cyklem* jeżeli $v_0 = v_k$. Analogicznie, *cyklem prostym* nazywamy cykl, w którym wszystkie wierzchołki są różne poza wierzchołkiem początkowym i końcowym. W grafie nieskierowanym droga $p = (v_i)_{i=0}^{k}$ tworzy *cykl*, jeżeli $k > 0$, $v_0 = v_k$ oraz wszystkie krawędzie są na tej drodze różne. *Graf acykliczny* to taki graf, który nie zawiera cykli.

[2)] Multizbiór jest uogólnieniem pojęcia zbioru, w którym jeden element (w naszym wypadku krawędź grafu) może występować więcej niż jeden raz.



Rys. 1.4: Multigraf (skierowany) $G = (V, E)$, gdzie $V = \{1, 2, 3\}$ i $E = \{(1, 2), (2, 2), (2, 2), (2, 3), (2, 3), (2, 3), (3, 1), (3, 1))\}$. Przedstawiony multigraf zawiera między innymi trzy krawędzie $(2, 3)$, które są równoległe. W tym przypadku nie da się odróżnić tych krawędzi i należałoby nadać odpowiednie etykiety każdej krawędzi (mielibyśmy do czynienia tzw. *grafem etykietowanym*).

Definicja 1.6. *Grafem spójnym* nazywamy graf, w którym dla każdego wierzchołka $v \in V$ istnieje droga do każdego innego wierzchołka $v' \in V$. W takim przypadku graf nieskierowany nazywamy krótko *grafem spójnym*, natomiast graf skierowany nazywamy *grafem silnie spójnym*.

Definicja 1.7. *Stopień wierzchołka* w grafie nieskierowanym to suma krawędzi incydentnych do niego. W grafie skierowanym dla każdego wierzchołka wyróżniamy *stopień wejściowy* będący liczbą krawędzi do niego wchodzących oraz *stopień wyjściowy*, który jest liczbą krawędzi z niego wychodzących. *Stopniem wierzchołka* w grafie skierowanym nazywamy sumę stopni wchodzących i wychodzących. Pętle zwiększają stopień wierzchołka o 2. Wierzchołek stopnia 0 jest nazywany *wierzchołkiem izolowanym*, a wierzchołek stopnia 1 *wierzchołkiem końcowym* albo *wierzchołkiem wiszącym*.

Definicja 1.8. Gęstością grafu $G = (E, V)$ nazywamy stosunek liczby jego krawędzi do maksymalnej możliwej ilości krawędzi w grafie. W grafach prostych będą to więc odpowiednio liczby[3)]

[3)] Dla grafów prostych maksymalna ilość krawędzi w grafie skierowanym wynosi $|V|\big(|V| - 1\big)$ natomiast w grafie nieskierowanym jest to $|V|\big(|V| - 1\big)/2$.

- dla grafów nieskierowanych

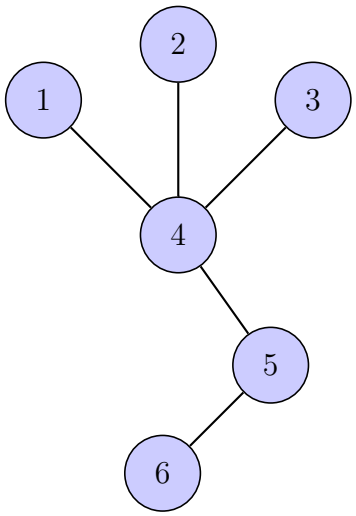
$$\rho(G) = \frac{2|E|}{|V|\Big(|V| - 1\Big)}$$

- dla grafów skierowanych:

$$\rho(G) = \frac{|E|}{|V|\Big(|V| - 1\Big)}$$

Grafem rzadkim będziemy określić graf, w którym ilość wszystkich krawędzi $|E|$ jest dużo mniejsza niż kwadrat ilości wierzchołków $|V|^2$. Natomiast gdy ilość krawędzi będzie bliska $|V|^2$ to mówimy, że graf jest *gęsty*[4)].

Definicja 1.9. *Drzewem* nazywamy nieskierowany graf, który jest acykliczny i spójny (zobacz rysunek 1.5). Wierzchołek drzewa o stopniu równym 1 nazywamy *liściem*. Drzewo, w którym jest wyróżniony jeden z wierzchołków nazywamy *drzewem ukorzenionym*, a wyróżniony wierzchołek – *korzeniem*. Graf nieskierowany i acykliczny, ale niekoniecznie spójny nazywamy *lasem*.

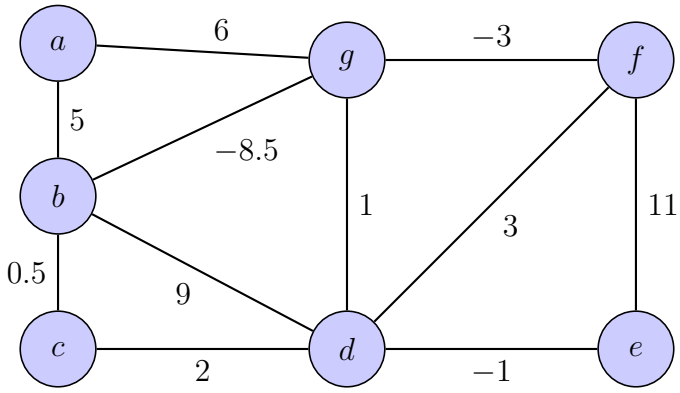


Rys. 1.5: Przykład grafu będącego drzewem.

Definicja 1.10. *Grafem z wagami* lub *grafem ważonym* nazywamy taki graf $G = (V, E)$, w którym z każdą krawędzią związana jest pewna liczba rzeczywista zwana *wagą* lub *kosztem*. Wagi krawędzi przeważnie określone są poprzez *funkcję wagową* $w\colon E \to \mathbb{R}$. Możemy zatem powiedzieć, że grafem ważonym nazywamy trójkę $G =$

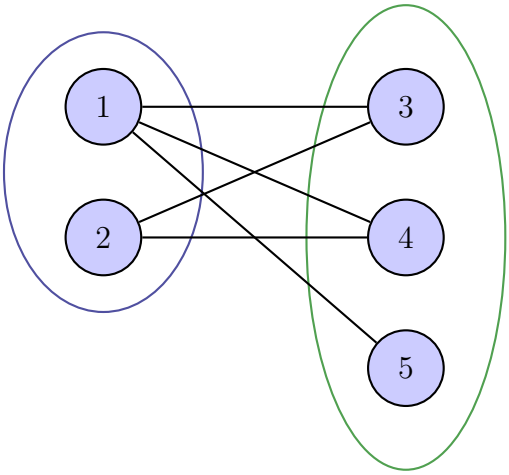
[4)] Jak można wywnioskować z tej definicji, pojęcie grafu rzadkiego i gęstego nie jest precyzyjne. Korzystając z pojęcia gęstości grafu możemy określać, dla jakiej wartości progowej graf uznajemy za rzadki lub gęsty.

(V, E, w), gdzie w jest pewną funkcją wagową (zob. przykład na rysunku 1.6). Jeżeli nie będzie to prowadzić do nieporozumień to w celu oznaczenia wartości funkcji wagowej (czyli wagi) dla $(u, v) \in E$ będziemy pisać $w(u, v)$ zamiast $w\Big((u, v)\Big)$.



Rys. 1.6: Przykład nieskierowanego grafu ważonego.

Definicja 1.11. *Graf dwudzielny* to taki graf, który można podzielić na dwa oddzielne zbiory wierzchołków, które nie będą połączone ze sobą wewnątrz tego zbioru. Można też stworzyć graf K-dzielny, zasada jest podobna, podział na K rozłącznych zbiorów wierzchołków (zob. przykład na rysunku 1.7).



Rys. 1.7: Przykład grafu dwudzielnego. Jak widać podany graf dwudzielny można podzielić na 2 zbiory wierzchołków $\{1, 2\}$ oraz $\{3, 4, 5\}$.

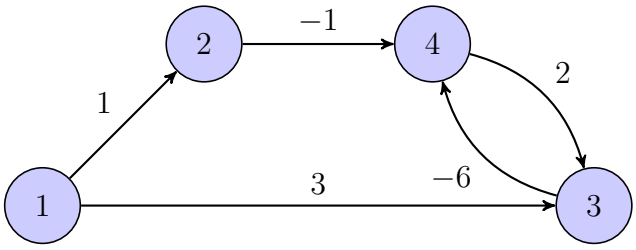
Definicja 1.12. Niech $G = (V, E, w)$ będzie grafem ważonym z funkcją wagową $w\colon E \to \mathbb{R}$. *Wagą drogi* (*długością drogi*) $\boldsymbol{p} = (v_i)_{i=0}^{k}$ nazywamy liczbę będącą sumą wag krawędzi tworzących tę drogę:

$$w(\boldsymbol{p}) = \sum_{j=0}^{k-1} w(v_i, v_{i+1}).$$

Cykl o wadze ujemnej nazywamy *cyklem ujemnym* (zob. przykład na rysunku 1.8). *Wagę najkrótszej ścieżki* $\delta(u, v)$ z wierzchołka u do wierzchołka v definiujemy następująco:

$$\delta(u,v) = \begin{cases} \min\{w(p)\colon\ u \overset{p}{\rightsquigarrow} v\}, & \text{gdy istnieje droga z } u \text{ do } v \\ \infty, & \text{gdy nie istnieje droga z } u \text{ do } v \end{cases}.$$

Każdą ścieżkę p z wierzchołka u do wierzchołka v, dla której zachodzi równość $w(p) = \delta(u, v)$ nazywamy *najkrótszą ścieżką*.



Rys. 1.8: Przykład skierowanego grafu ważonego zawierającego cykl ujemny. Na podanym grafie pomiędzy wierzchołkiem 3 oraz 4 występuje ujemny cykl o wadze równej -4.

Analizując algorytmy będziemy często używać oznaczeń zaczerpniętych z teorii zbiorów: $|V|$ i $|E|$ dla oznaczenia odpowiednio liczby wszystkich wierzchołków grafu oraz liczby wszystkich krawędzi w grafie[5)]. Często też, ilość wierzchołków w grafie G będziemy oznaczać symbolem n, a liczbę krawędzi symbolem m. Dodatkowo podczas analizy złożoności algorytmów zazwyczaj podaje się sam symbol bez dodatkowych znaków, na przykład V^2 zamiast $|V|^2$.

Istnieje wiele rodzajów grafów, ale najprostszym podziałem będzie podział na:

- Grafy nieskierowane (*niezorientowane*)
- Grafy skierowane (*zorientowane*)
- Grafy ważone
- Drzewa
- Multigrafy

[5)] Chodzi tu konkretnie o symbole oznaczające *moc zbioru*, czyli liczbę jego elementów.

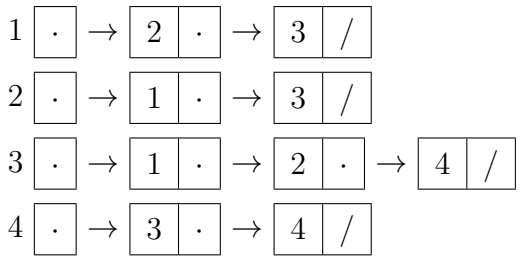
1.2 Reprezentacja grafów w komputerze

Istnieją dwa najpopularniejsze sposoby reprezentacji grafów w pamięci komputera:

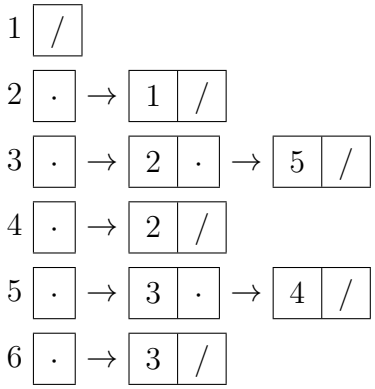
- Lista sąsiedztwa

- Macierz sąsiedztwa

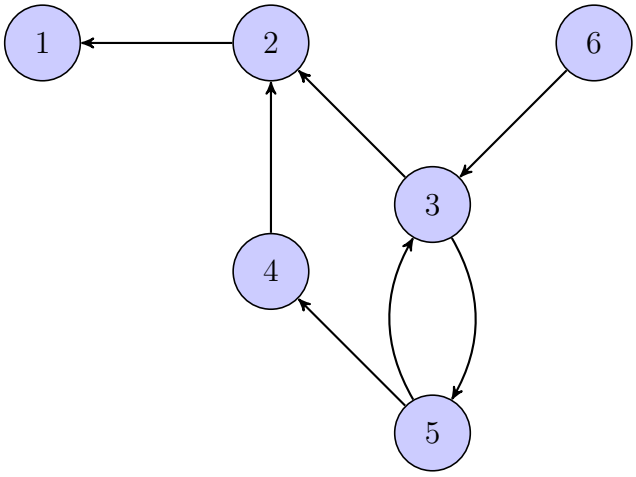
Lista sąsiedztwa to taka struktura danych, w której dla każdego wierzchołka w grafie tworzymy listę sąsiadujących z nim wierzchołków.



Rys. 1.9: Lista sąsiedztwa grafu nieskierowanego z rysunku 1.2.



Rys. 1.10: Lista sąsiedztwa przykładowego grafu skierowanego przedstawionego na rysunku 1.11.



Rys. 1.11: Przykładowy graf skierowany.

W celu zaimplementowania listy sąsiedztwa w języku Python musimy stworzyć listę wierzchołków, a każdy wierzchołek będzie listą sąsiednich wierzchołków więc otrzymujemy listę list. W przypadku grafu ważonego, listę zastępujemy słownikiem, aby dodatkowo opisać wagi na krawędziach. Najprostszy przykład kodu w języku Python dla grafu przedstawionego na rysunku 1.2 może obrazować kod źródłowy z rysunku 1.12. Bardziej złożony graf skierowany z rysunku 1.11 może być zaimplementowany w Pythonie jak na rysunku 1.13.

```
1 v1 = [2, 3]
2 v2 = [1, 3]
3 v3 = [1, 2, 4]
4 v4 = [3, 4]
5 vertices = [v1, v2, v3, v4]
```

Rys. 1.12: Kod źródłowy implementacji listy sąsiedztwa grafu nieskierowanego.

```
1 v1 = []
2 v2 = [1]
3 v3 = [2, 5]
4 v4 = [2]
5 v5 = [3, 4]
6 v6 = [3]
7 vertices = [v1, v2, v3, v4, v5, v6]
```

Rys. 1.13: Kod źródłowy implementacji listy sąsiedztwa grafu skierowanego.

Macierz sąsiedztwa to macierz kwadratowa $\boldsymbol{A} = [a_{ij}]$ o rozmiarze $n \times n$ (n to liczba wierzchołków grafu) taka, że $a_{ij} = 1$, jeżeli w grafie istnieje krawędź (i, j),

a $a_{ij} = 0$, jeżeli nie istnieje. Jeśli graf jest nieskierowany to macierz sąsiedztwa jest symetryczna.

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Rys. 1.14: Macierz sąsiedztwa dla grafu z rysunku 1.11

Aby zaimplementować macierz sąsiedztwa w języku Python należy stworzyć listę list n - wymiarowych, gdzie n to liczba wierzchołków w grafie, dla grafu z rysunku 1.11 macierz sąsiedztwa przedstawia rysunek 1.15

```
matrix = [[0, 0, 0, 0, 0, 0],
          [1, 0, 0, 0, 0, 0],
          [0, 0, 1, 0, 1, 0],
          [0, 0, 1, 0, 0, 0],
          [0, 0, 1, 1, 0, 0],
          [0, 0, 1, 0, 0, 0]]
```

Rys. 1.15: Kod źródłowy implementacji macierzy sąsiedztwa.

1.3 Podstawowe algorytmy badania grafów

Badanie grafu to algorytmiczne przejście każdego wierzchołka grafu i sprawdzenie (zbadanie) relacji z innymi wierzchołkami. Można dzięki temu znajdować m.in.: *spójne składowe grafu*, ścieżki między wierzchołkami (w tym dla grafów nieważonych te najkrótsze) lub badać czy dany graf jest dwudzielny.

Najprostszymi algorytmami badającymi grafy są 2 podstawowe algorytmy:

- Przeszukiwania wszerz (*BFS*)
- Przeszukiwania wgłąb (*DFS*)

Są one używane w wielu bardziej skomplikowanych algorytmach. Przykładem użycia przeszukiwania w głąb jest *sortowanie topologiczne skierowanych grafów acyklicznych* (z ang. *directed acyclic graph*, w skrócie *DAG*), polegające na uporządkowaniu w porządku liniowym wierzchołków w taki sposób, aby każdy kolejny wierzchołek

wymagał poprzedniego, czyli relacja między dwoma wierzchołkami przechodziła od poprzedniego do następnego. Popularnym zastosowaniem DAG-ów jest analiza sieci PERT[6)].

1.4 Problem najkrótszych ścieżek w grafie ważonym

Problem znajdowania najkrótszej ścieżki między dwoma wierzchołkami u i v w grafie ważonym sprowadza się do wyznaczenia drogi, która prowadzi z wierzchołka u do v, której waga (koszt) jest minimalna. Istnieją różne odmiany algorytmów wyznaczania najkrótszych ścieżek w grafach:

- **Najkrótsze ścieżki z jednym wierzchołkiem źródłowym.** W problemie tym wyróżniamy jeden wierzchołek s (zwany *źródłem*, ang. *source*) grafu G i poszukujemy najkrótszych ścieżek z wierzchołka s do pozostałych wierzchołków grafu.

- **Najkrótsze ścieżki z jednym wierzchołkiem docelowym.** Wyróżniamy w grafie G jeden wierzchołek docelowy t (ang. *target*) i poszukujemy najkrótszy ścieżek ze wszystkich pozostałych wierzchołków grafu G do wierzchołka t. Łatwo zauważyć, że gdy zmienimy zwroty wszystkich krawędzi (w grafach skierowanych) na przeciwne to problem ten sprowadza się do wyznaczenia najkrótszych ścieżek z jednym wierzchołkiem źródłowym.

- **Najkrótsze ścieżki między parą wierzchołków.** W tym problemie poszukujemy najkrótszej drogi prowadzącej z pewnego wierzchołka u do innego wierzchołka v. Niestety okazuje się, że w pesymistycznym przypadku żaden algorytm rozwiązujący ten problem nie jest szybszy od algorytmu rozwiązującego problem najkrótszych ścieżek z jednym źródłem.

- **Najkrótsze ścieżki między wszystkimi parami wierzchołków.** Poszukujemy najkrótszych ścieżek między wszystkimi parami wierzchołków danego grafu G.

Dla problemu z jednym wierzchołkiem źródłowym lub docelowym użyć można algorytmu *Djikstry*, wymaga on, by w całym grafie nie było krawędzi o ujemnych wagach. Bardziej ogólnym algorytmem dla problemu z jednym źródłem, w którym

[6)] *PERT* (z ang. *Program Evaluation and Review Technique*) - metoda planowania i zarządzania projektem, przedstawiana jako graf skierowany z wagami, którego wierzchołki stanowią zadania realizowane w ramach projektu oraz krawędzie z wagami mówiące o czasie potrzebnym na wykonanie zadania.

dopuszczamy występowanie wag ujemnych jest algorytm *Bellmana-Forda*. Dla problemu najkrótszych ścieżek między wszystkimi parami wierzchołków można oczywiście użyć algorytmu Djikstry lub Bellmana-Forda (kolejno traktując jako źródło wszystkie wierzchołki grafu). Rozwiązanie to jest jednak bardzo nieefektywne. Na szczęście istnieją lepsze algorytmy dla problemu wyznaczenia najkrótszych ścieżek między wszystkimi parami wierzchołków m.in. algorytm *Floyda-Warshalla* oraz algorytm *Johnsona* (efektywny dla grafów rzadkich).

Rozdział 2

Omówienie wybranych algorytmów

W tym rozdziale zostaną dokładnie przedstawione wspominane już algorytmy wyznaczania najkrótszych ścieżek w grafach ważonych. W szczególności podamy pseudokody tych algorytmów, które podobnie jak podane tu informacje można znaleźć m.in. w [1, 2].

2.1 Algorytmy wyznaczające ścieżki z jednego źródła

Na początek opiszemy algorytmy Dijkstry oraz Bellmana-Forda, które rozwiązują problem wyznaczania najkrótszej drogi w grafie z jednym wierzchołkiem źródłowym.

2.1.1 Algorytm Dijkstry

Algorytm Dijkstry przeznaczony jest do znajdowania najkrótszych ścieżek w skierowanych grafach ważonych o wagach nieujemnych. Zakładamy więc, że mamy graf ważony $G = (V, E)$ z funkcją wagową w taką, że $w(u, v) \geq 0$ dla każdej krawędzi $(u, v) \in E$. Poza tym w zbiorze wierzchołków V grafu G wyróżniamy wierzchołek źródłowy s. Algorytm wyznacza w grafie wszystkie najkrótsze ścieżki między wierzchołkiem źródłowym, a pozostałymi wierzchołkami grafu oraz wyznacza koszt przejścia każdej z tych ścieżek.

W algorytmie Dijkstry z każdym wierzchołkiem $v \in V$ grafu G wiążemy dwie wartości. Pierwszą z nich jest $d[v]$, która jest aktualnym górnym oszacowaniem kosztu najkrótszej drogi z wierzchołka źródłowego do wierzchołka v. Drugą wartością jest $p[v]$, która zawiera poprzedni wierzchołek (tzw. poprzednik) do v na chwilowo najkrótszej drodze ze źródła do v. Ponadto, w omawianym algorytmie wykorzystuje się tzw. operację *relaksacji* (inaczej *osłabiania ograniczeń*).

Definicja 2.1. Relaksacją krawędzi (u, v) grafu G przy uwzględnieniu wierzchołka źródłowego s nazywamy proces polegający na sprawdzeniu, czy przechodząc przez wierzchołek u można wyznaczyć krótszą ścieżkę (tj. o mniejszym koszcie) między v i s od dotychczas wyznaczonej. Jeżeli w wyniku tego sprawdzania taka możliwość istnieje to następuje odpowiednia aktualizacja wartości $d[v]$ i $p[v]$. W wyniku relaksacji może uleć zmienieniu wartość oszacowania kosztu najkrótszej drogi $d[v]$ i może uleć zmianie poprzednik $p[v]$ (zobacz rysunek 2.1).

```
1 Function relax(u, v, w):
2     if d[v] ≥ d[u] + w(u, v) then
3         d[v] = d[u] + w(u, v)
4         p[v] = u
```

Rys. 2.1: Pseudokod procedury relaksacji.

W prezentowanym algorytmie Dijkstry (rysunek 2.2) przyjmuje się początkowo, dla każdego wierzchołka $v \in V$, że $d[v] = \infty$ oraz $d[s] = 0$. Ponadto, dla każdego wierzchołka $v \in V$ przyjmujemy początkowo $p[v] = \text{Null}$, co oznacza brak poprzednika [1)]. W trakcie działania algorytmu pamiętany jest (początkowo pusty) zbiór S, w którym umieszczane są wierzchołki, dla których koszt najkrótszej drogi ze źródła s został już obliczony. Używana jest także kolejka priorytetowa (typu min) $Q = V/S$, w której znajdą się wierzchołki jeszcze nieprzetworzone (wierzchołki są zorganizowane według wartości d). Następnie algorytm wykonuje wielokrotnie następujące kroki:

- wybiera z kolejki Q wierzchołek u o najmniejszej wartości oszacowania $d[u]$ i usuwa go z niej (operacja `extractmin`);
- dodaje wierzchołek u do zbioru S;
- wykonuje relaksację dla wszystkich krawędzi wychodzących z wierzchołka u.

W wyniku działania algorytmu Dijkstry dla każdego wierzchołka v otrzymujemy wartości $d[v]$, która jest najmniejszym kosztem drogi z wierzchołka s do wierzchołka v (tj. wagą najkrótszej ścieżki z s do v). Z kolei, dzięki wartości $p[v]$ możemy odczytać drogę o koszcie $d[v]$, odczytując kolejno v, $p[v]$, $p[p[v]]$ itd. aż dojedziemy do s.

Warto zauważyć, że w algorytmie Dijskstry do zbioru S za każdym razem dodawany jest wierzchołek $v \in V/S$ o aktualnie najmniejszej wartości $d[v]$. Jest to

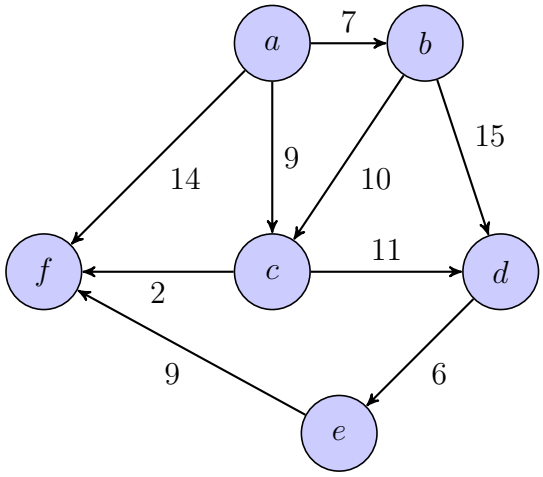
[1)] W zależności od implementacji może to być wartość 0, Null lub None.

```
1  Function dijkstra(G, w, s):
2      for all v ∈ V(G) do
3          d[v] = ∞
4          p[v] = Null
5      d[s] = 0
6      S = ∅
7      Q = V(G)
8      while Q ≠ ∅ do
9          u = extract_min(Q)
10         S = S ∪ {u}
11         for all v ∈ Adj[u] do
12             relax(u, v, w)
```

Rys. 2.2: Pseudokod algorytmu Dijkstry.

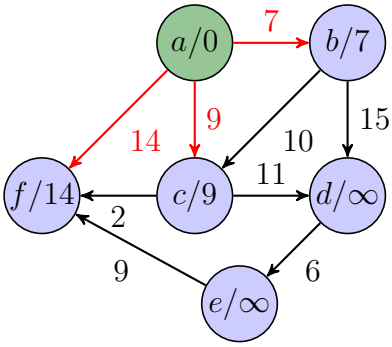
zatem przykład algorytmu zachłannego. Można wykazać, że wyniku działania tego algorytmu zawsze otrzymujemy optymalne rozwiązanie.

Przykład 1. Przeanalizujmy działanie algorytmu Dijkstry (ryunek 2.2) dla przykładowego grafu (rysunek 2.3), gdzie jako wierzchołek źródłowy wybieramy wierzchołek a. Etapy działania algorytmu (iteracje drugiej pętli) możemy zobaczyć na rysunku 2.4. Kolejne ilustracje prezentują przetwarzanie kolejnych wierzchołków: a, b, c, d oraz f.

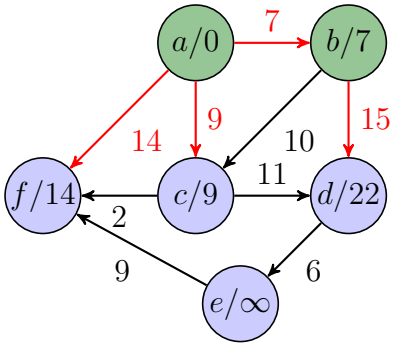


Rys. 2.3: Przykładowy graf ważony o wagach nieujemnych.

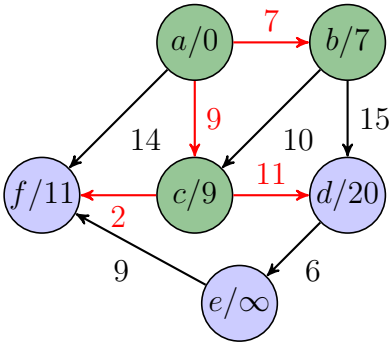
Wierzchołki zaznaczone na zielono to wierzchołki, które zostały już przetworzone (odwiedzone) przez algorytm. Kolorem czerwonym zaznaczamy krawędzie prowadzą-



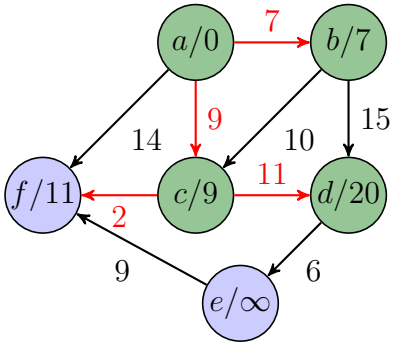
(a) Przetwarzanie wierzchołka "a".



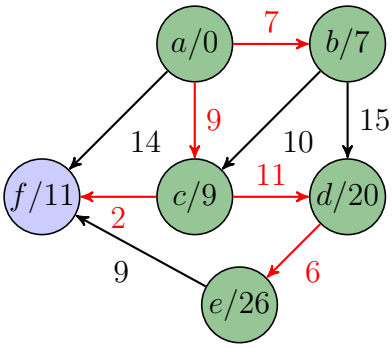
(b) Przetwarzanie wierzchołka "b".



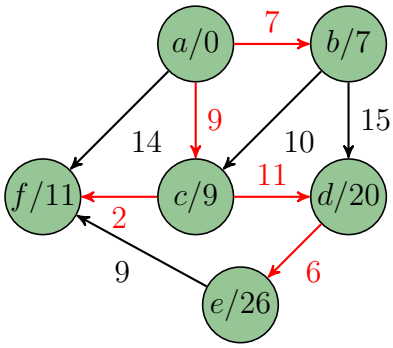
(c) Przetwarzanie wierzchołka "c".



(d) Przetwarzanie wierzchołka "d".



(e) Przetwarzanie wierzchołka "f".



(f) Przetwarzanie wierzchołka "e".

Rys. 2.4: Przebieg algorytmu Dijkstry dla grafu 2.3

ce od poprzednika do rozważanego wierzchołka, tworzymy z nich najkrótsze ścieżki, dla przykładu: jeżeli chcemy odczytać najkrótszą drogę od źródła (a) do wierzchołka f należy popatrzeć na rysunek 2.4f. Dla wierzchołka f poprzednikiem jest wierzchołek c więc krawędź (c, f) jest zaznaczona na czerwono, dalej poprzednikiem c jest a, wierzchołek a nie ma już poprzedników (bo jest startowy). Otrzymujemy zatem najkrótszą drogę: $a \to c \to f$. Z gotowych poprzedników powstaje tzw. *drzewo najkrótszych ścieżek* o korzeniu w wierzchołku źródłowym, jest to podgraf wejściowego grafu, który zawiera tylko krawędzie, z których można utworzyć najkrótsze ścieżki.

2.1.2 Algorytm Bellmana-Forda

Algorytm Bellmana-Forda jest uogólnioną wersją algorytmu Dijkstry. Podstawową wadą algorytmu Dijkstry jest niemożność zastosowania go do grafów z wagami ujemnymi. Wynika to z faktu, że korzysta on z założenia iż dodanie do ścieżki dodatkowej krawędzi może ją tylko wydłużyć - uwzględnienie wag ujemnych w naturalny sposób psuje to założenie. Algorytm Bellmana-Forda wyznacza najkrótsze ścieżki w gafie ważonym (również o wagach ujemnych) z ustalonego wierzchołka s do pozostałych wierzchołków grafu. W przypadku wystąpienia w grafie cyklu ujemnego, algorytm Bellmana-Forda nie może wyznaczyć szukanej drogi[2)], ale wykrywa taką sytuację - dlatego często jest używany do wykrywania cykli ujemnych w grafach ważonych.

```
1  Function bellman-ford(G, w, s):
2      for all v ∈ V(G) do
3          d[v] = ∞
4          p[v] = Null
5      d[s] = 0
6      for i ← 1 to |V[G]| − 1 do
7          for all (u, v) ∈ E(G) do
8              relax(u, v, w)
9      for all (u, v) ∈ E(G) do
10         if d[v] > d[u] + w(u, v) then
11             return False
12     return True
```

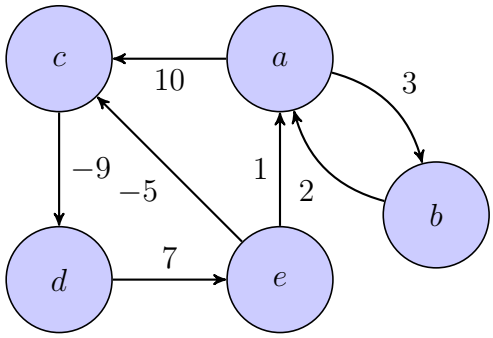
Rys. 2.5: Pseudokod algorytmu Bellmana-Forda.

[2)] Oczywiście, w takiej sytuacji w grafie nie da się w ogóle wyznaczyć najkrótszych ścieżek.

W omawianym algorytmie również używana jest relaksacja, lecz zamiast wykonywać relaksację tylko na tych krawędziach, które zapewniają minimalizację wagi, relaksacja jest wykonywana dla wszystkich $n-1$ krawędzi, gdzie n to liczba wierzchołków w grafie. Następnie sprawdzane jest wystąpienie cyklu ujemnego w badanym grafie. Jeżeli odległość od końcowego wierzchołka jest mniejsza niż odległość od wierzchołka początkowego powiększona o wagę danej krawędzi to musiał wystąpić cykl o ujemnej wadze, zwracany jest wtedy błąd w postaci zmiennej logicznej `False`.

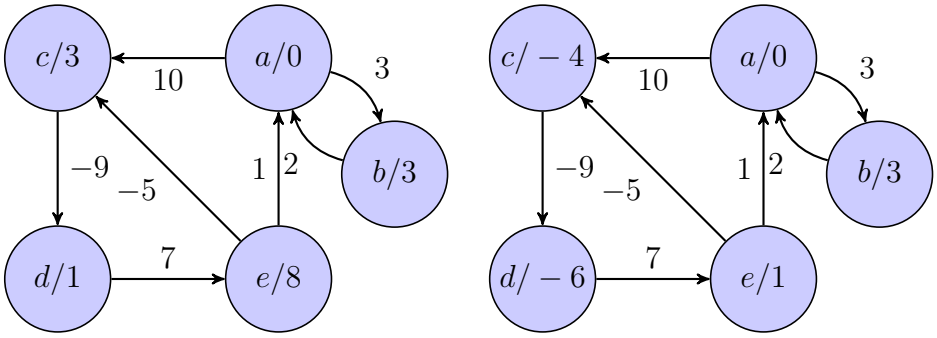
Zwracany wynik algorytmu Bellmana-Forda jest identyczny do wyniku algorytmu Dijkstry. W celu ilustracji działania algorytmu Bellmana-Forda przeanalizujemy dwa przykłady grafów - jeden z cyklem ujemnym, a drugi bez.

Przykład 2. Rozważmy graf przedstawiony na rysunku 2.6 z cyklem o wadze ujemnej ($c \to d \to e \to c$) równej -7. Przeanalizujmy działanie algorytmy Bellmana-Forda dla tego grafu, gdy jako wierzchołek źródłowy wybierzemy a.



Rys. 2.6: Przykładowy graf ważony zawierający ujemny cykl.

Rysunek 2.7d przedstawia wynik algorytmu. Żeby wynik był poprawny dla każdej krawędzi należy przeprowadzić sprawdzanie z 10 linii pseudokodu (**`if`** $d[v] > d[u] + w(u,v)$ **`then return`** `False`). W tym przypadku sprawdzanie to zakończy się zwróceniem wartości `False`.



(a) Pierwsza iteracja pętli.

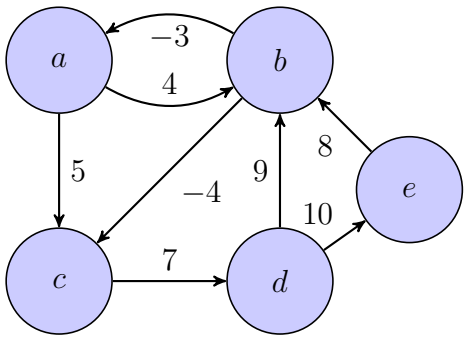
(b) Druga iteracja pętli.

(c) Trzecia iteracja pętli.

(d) Czwarta iteracja pętli.

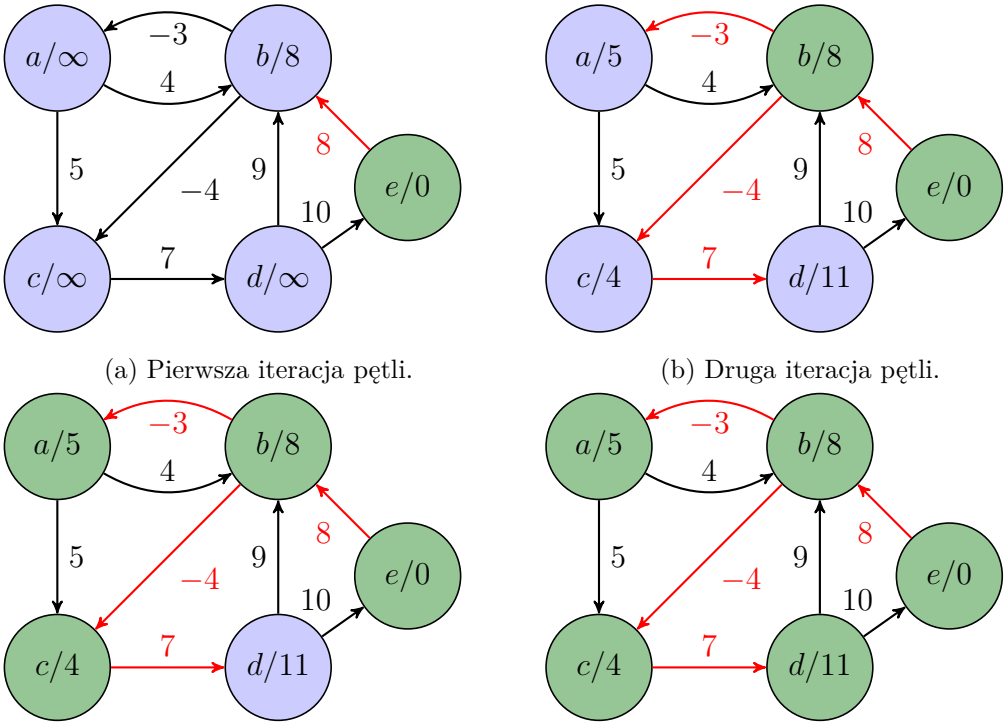
Rys. 2.7: Przebieg algorytmu Bellmana-Forda dla grafu 2.6

Przykład 3. Przeanalizujemy teraz działanie algorytmu Bellmana-Forda dla grafu, który nie zawiera cykli ujemnych (rysunek 2.8). Jako wierzchołek źródłowy wybieramy wierzchołek e.



Rys. 2.8: Przykładowy graf niezawierający ujemnego cyklu.

W wyniku działania algorytmu otrzymujemy najkrótsze ścieżki między wierzchołkiem źródłowym (e) i pozostałymi wierzchołkami oraz ich wagi - w takiej samej formie jak w przypadku algorytmu Dijkstry (oczywiście algorytm Dijkstry zwróciłby tu niepoprawne wyniki ze względu na ujemne wagi). Przykładowo, najkrótsza ścieżka między wierzchołkiem e i d to: $e \to b \to c \to d$, a jej waga wynosi 11.



(a) Pierwsza iteracja pętli.

(b) Druga iteracja pętli.

(c) Trzecia iteracja pętli.

(d) Czwarta iteracja pętli.

Rys. 2.9: Przebieg algorytmu Bellmana-Forda dla grafu z rysunku 2.8

2.2 Algorytmy wyznaczające ścieżki między wszystkimi wierzchołkami

W tym podrozdziale zajmiemy się algorytmami, które służą do rozwiązywania problemu wyznaczania najkrótszych ścieżek pomiędzy wszystkimi wierzchołkami ważonego grafu skierowanego. Oczywiście problem ten można rozwiązać wykonując $|V|$ razy algorytm rozwiązujący problem najkrótszych ścieżek z jednym źródłem. Jeżeli graf nie ma wag ujemnych możemy zastosować algorytm Dijkstry, w przeciwnym wypadku musimy wybrać wolniejszy algorytm Bellmana-Forda. Złożoność obliczeniowa takiego podejścia zostanie omówiona w rozdziale 4, w tej sekcji omówimy jak lepiej rozwiązać wcześniej wspomniany problem. Kolejno omówimy trzy algorytmy: algorytm z iloczynem odległości[3)], Floyda-Warshalla oraz Johnsona.

Ponieważ pierwsze dwa algorytmy, które będziemy omawiać działają wykorzystując jako reprezentację grafu $G = (V, E)$ macierze sąsiedztwa musimy wprowadzić pewne oznaczenia. Zakładamy, że wierzchołki w grafie są numerowane od 1 do

[3)] Będziemy stosować taką nazwę do określenia algorytmu, w którym wykorzystywana jest operacja podobna do mnożenia macierzy (zwana przez nas *iloczynem odległości*). Warto zaznaczyć, że w literaturze algorytm ten nie ma jakiejś specjalnej nazwy.

$n = |V|$. Będziemy zatem rozważać macierz $\mathbf{W} = [w_{ij}]$ o wymiarach $n \times n$, która reprezentuje wagi krawędzi n-wierzchołkowego grafu skierowanego, gdzie

$$w_{ij} = \begin{cases} 0, & \text{dla } i = j, \\ w(i,j), & \text{dla } i \neq j \text{ oraz } (i,j) \in E, \\ \infty, & \text{dla } i \neq j \text{ oraz } (i,j) \notin E. \end{cases}$$

Wynikiem działania przedstawionych w tym podrozdziale algorytmów będzie macierz (tablica) $\mathbf{D} = [d_{ij}]$ o wymiarze $n \times n$, gdzie d_{ij} jest wagą najkrótszej ścieżki od wierzchołka i do j tj. $d_{ij} = \delta(i,j)$. Macierz $\mathbf{D}$ będziemy nazywać *macierzą najkrótszych ścieżek*. W celu rozwiązania problemu najkrótszych ścieżek między wszystkimi parami wierzchołków oprócz wyznaczenia postaci macierzy $\mathbf{D}$ należy jeszcze wyznaczyć tzw. *macierz poprzedników* $\mathbf{P} = [p_{ij}]$, gdzie

$$p_{ij} = \begin{cases} \texttt{None}, & \text{dla } i = j, \\ k, & \begin{array}{l}\text{gdzie } k \text{ jest poprzednikiem } j \text{ na pewnej} \\ \text{najkrótszej ścieżce prowadzącej z } i \text{ do } j\end{array} \end{cases}.$$

Ponadto w algorytmach, pewne macierze będą miały indeks górny ujęty w nawiasy dla oznaczenia iteracji. Przykładowo $\mathbf{D}^{(m)} = [d_{ij}^m]$ oznacza macierz ścieżek wyznaczoną w m-tej iteracji algorytmu.

2.2.1 Wyznaczanie najkrótszych ścieżek a mnożenie macierzy - algorytm z iloczynem odległości

Pierwszy z algorytmów, który służy do wyznaczania najkrótszych ścieżek korzysta z bardzo ważnej własności najkrótszych ścieżek. Własność ta mówi, że wszystkie podścieżki każdej najkrótszej ścieżki są również najkrótszymi ścieżkami. Niech będzie dana macierz wag $\mathbf{W} = [w_{ij}]$ dla naszego grafu G. Weźmy najkrótszą ścieżkę p z wierzchołka i do wierzchołka j i załóżmy o niej, że składa się z co najwyżej m krawędzi. Jeżeli graf nie zawiera cykli ujemnych to $m < \infty$. Jeżeli $i = j$ to $w(p) = 0$ i ścieżka nie zawiera żadnej krawędzi. Natomiast, jeśli $i \neq j$ to ścieżkę p można rozłożyć na ścieżkę $i \overset{p'}{\rightsquigarrow} k$ i krawędź (k,j), gdzie p' ma już $m-1$ krawędzi. Ścieżka p' jest najkrótszą ścieżką prowadzącą z i do k i zachodzi równość $\delta(i,j) = \delta(i,k) + w_{k,j}$.

Załóżmy teraz, że $d_{ij}^{(m)}$ jest najmniejszą wagą ścieżki prowadzącej z wierzchołka i do wierzchołka j zawierającej co najwyżej m krawędzi. Oczywiście, jeśli $m = 0$ to istnieje najkrótsza droga z i do j wtedy i tylko wtedy, gdy $i = j$. Jest zatem

$$d_{ij}^{(0)} = \begin{cases} 0, & \text{dla } i = j, \\ \infty, & \text{dla } i \neq j. \end{cases}$$

Jeżeli mamy $m \geq 1$ to $d_{ij}^{(m)}$ obliczamy według następującej reguły

$$d_{ij}^{(m)} = \min\left(d_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\{d_{ik}^{(m-1)} + w_{kj}\}\right) = \min_{1 \leq k \leq n}\{d_{ik}^{(m-1)} + w_{kj}\}. \tag{2.1}$$

Zatem $d_{ij}^{(m)}$ to minimum wagi najkrótszej ścieżki z i do j składającej się z co najwyżej $m-1$ krawędzi oraz najmniejszej wagi spośród tych ścieżek, o co najwyżej m krawędziach, którą otrzymamy rozważając wszystkie możliwe poprzedniki k wierzchołka j.

Jeżeli graf nie ma cykli ujemnych to dla każdej pary wierzchołków i oraz j, dla których $\delta(i,j) < \infty$, istnieje ścieżka prosta mająca co najwyżej $n-1$ krawędzi. Oczywiście dłuższa ścieżka między i i j nie może mieć mniejszej wagi niż najkrótsza ścieżka z i do j. Zachodzi więc

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \ldots. \tag{2.2}$$

Ogólna idea algorytmu jest następująca. Danymi wejściowymi algorytmu jest macierz wag $\mathbf{W} = [w_{ij}]$, a w wyniku działania algorytmu otrzymujemy ciąg macierzy $\mathbf{D}^{(1)}, \mathbf{D}^{(2)}, \ldots, \mathbf{D}^{(n-1)}$, gdzie $\mathbf{D}^{(m)} = [d_{ij}^{(m)}]$ dla $m = 1, 2, \ldots, n-1$. Ostatnia w tym ciągu macierz reprezentuje wagi najkrótszych ścieżek między wszystkimi parami wierzchołków tj. $\mathbf{D}^{(n-1)} = \mathbf{D}$. Główna część algorytmu to procedura zaprezentowana na rysunku 2.10.

```
1 Function extend-shortest-path(D, W):
2     n = rows[D]
3     D' ← n × n matrix
4     for i ← 1 to n do
5         for j ← 1 to n do
6             d'_ij = ∞
7             for i ← 1 to n do
8                 d'_ij = min(d'_ij, d_ik + w_kj)
9     return D'
```

Rys. 2.10: Pseudokod algorytmu rozszerzania macierzy najkrótszych ścieżek.

Pokażemy teraz związek, a właściwie podobieństwo omówionej tu operacji wyznaczania macierzy najkrótszych ścieżek z operacją mnożenia macierzy kwadratowych. Niech $\mathbf{A}$ i $\mathbf{B}$ będą macierzami wag o wymiarach $n \times n$. *Iloczynem odległości* macierzy $\mathbf{A}$ i $\mathbf{B}$ nazywamy macierz $\mathbf{C} = \mathbf{A} *_{min} \mathbf{B}$, której wyrazy obliczamy według

wzoru:

$$c_{ij} = \min_{1 \leq k \leq n} \{a_{ik} + b_{kj}\}. \tag{2.3}$$

Tak zdefiniowane działanie iloczynu odległości ma następujące własności:

1. działanie $*_{min}$ jest łączne tzn. dla dowolnych macierzy $\mathbf{A}$, $\mathbf{B}$ i $\mathbf{E}$ o wymiarach $n \times n$ zachodzi równość:

$$(\mathbf{A} *_{min} \mathbf{B}) *_{min} \mathbf{E} = \mathbf{A} *_{min} (\mathbf{B} *_{min} \mathbf{E});$$

2. działanie $*_{min}$ jest przemienne względem dodawania tzn. dla dowolnych macierzy $\mathbf{A}$, $\mathbf{B}$ i $\mathbf{E}$ o wymiarach $n \times n$ jest:

$$\mathbf{A} *_{min} (\mathbf{B} + \mathbf{E}) = \mathbf{A} *_{min} \mathbf{B} + \mathbf{A} *_{min} \mathbf{E}$$

oraz

$$(\mathbf{A} + \mathbf{B}) *_{min} \mathbf{E} = \mathbf{A} *_{min} \mathbf{E} + \mathbf{B} *_{min} \mathbf{E};$$

3. elementem neutralnym działania $*_{min}$ jest macierz $\mathbf{I}$ o wymiarach $n \times n$, której elementy mają postać

$$c_{ij} = \begin{cases} 0, & \text{dla } i = j, \\ \infty, & \text{dla } i \neq j. \end{cases}$$

tzn. dla dowolnej macierzy $\mathbf{A}$ o wymiarze $n \times n$ zachodzą równości:

$$\mathbf{I} *_{min} \mathbf{A} = \mathbf{A} *_{min} \mathbf{I} = \mathbf{A}.$$

Pierwsza i druga własność wynika wprost z łączności operacji min oraz jej rozdzielności względem dodawania. Wykazując trzecią własność należy zauważyć, że można opuścić operację min dla tak zdefiniowanej macierzy $\mathbf{I}$. Zauważmy, że funkcja przedstawiona na rysunku 2.10 służy do wyznaczania iloczynu odległości macierzy $\mathbf{D}$ i $\mathbf{W}$. Jeżeli wykonamy teraz w równaniu (2.1) ciąg następujących podstawień:

$$\begin{aligned} d^{(m-1)} &\to a, \\ w &\to b, \\ d^{(m)} &\to c, \\ \min &\to +, \\ + &\to \cdot, \end{aligned}$$

to otrzymamy równanie

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj},$$

które wyznacza element iloczynu macierzy $\mathbf{C} = \mathbf{AB}$ na pozycji (i, j). Sama procedura przedstawiona na rysunku 2.10 będzie przedstawiała wtedy algorytm mnożenia macierzy kwadratowych $\mathbf{A}$ i $\mathbf{B}$ o wymiarach $n \times n$ (zob. rysunek 2.11).

```
1 Function matrix-multiply(A, B):
2     n = rows[A]
3     C ← n × n matrix
4     for i ← 1 to n do
5         for j ← 1 to n do
6             c_ij = 0
7             for i ← 1 to n do
8                 c_ij = c_ij + a_ik · b_kj
9     return C
```

Rys. 2.11: Pseudokod algorytmu mnożenia macierzy.

Wrócimy teraz do problemu wyznaczania najkrótszych ścieżek. W procedurze przedstawionej na rysunku 2.10, wagi najkrótszych ścieżek są obliczane poprzez rozszerzanie najkrótszych ścieżek krawędź po krawędzi. W procesie tym jest obliczany ciąg $n - 1$ macierzy

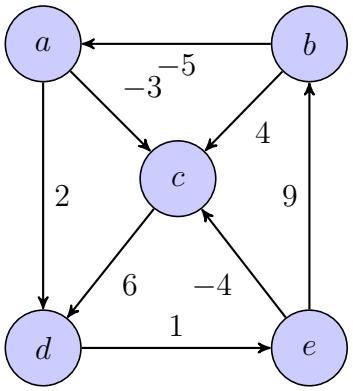
$$\begin{aligned}
\mathbf{D}^{(1)} &= \mathbf{D}^{(0)} *_{min} \mathbf{W} = \mathbf{W} \\
\mathbf{D}^{(2)} &= \mathbf{D}^{(1)} *_{min} \mathbf{W} = \mathbf{W}^2 \\
\mathbf{D}^{(3)} &= \mathbf{D}^{(2)} *_{min} \mathbf{W} = \mathbf{W}^3 \\
&\vdots \\
\mathbf{D}^{(n-1)} &= \mathbf{D}^{(n-2)} *_{min} \mathbf{W} = \mathbf{W}^{n-1}
\end{aligned}$$

Ostateczna macierz $\mathbf{D}^{(n-1)}$ reprezentuje wagi najkrótszych ścieżek, pseudokod z rysunku 2.12 ilustruje jak obliczyć taką macierz.

```
1 Function all-pairs-shortest-paths(W):
2     n = rows[W]
3     D^(1) ← W
4     for m ← 2 to n − 1 do
5         D^(m) ←extend-shortest-paths(D^(m−1), W)
6     return D^(n−1)
```

Rys. 2.12: Pseudokod algorytmu z iloczynem odległości wyznaczania najkrótszych ścieżek w grafie reprezentowanym przez macierz sąsiedztwa.

Przykład 4. Przeanalizujemy teraz działanie algorytmu z iloczynem odległości na przykładzie grafu przedstawionego na rysunku 2.13.



Rys. 2.13: Przykładowy skierowany graf ważony.

$$\mathbf{D}^{(1)} = \begin{pmatrix} 0 & \infty & -3 & 2 & \infty \\ -5 & 0 & 4 & \infty & \infty \\ \infty & \infty & 0 & 6 & \infty \\ \infty & \infty & 1 & 0 & 1 \\ \infty & 9 & -4 & \infty & 0 \end{pmatrix} \quad \mathbf{D}^{(2)} = \begin{pmatrix} 0 & \infty & -3 & 2 & 3 \\ -5 & 0 & -8 & -3 & \infty \\ \infty & \infty & 0 & 6 & 7 \\ \infty & 10 & -3 & 0 & 1 \\ 4 & 9 & -4 & 2 & 0 \end{pmatrix}$$

$$\mathbf{D}^{(3)} = \begin{pmatrix} 0 & 12 & -3 & 2 & 3 \\ -5 & 0 & -8 & -3 & -2 \\ \infty & 16 & 0 & 6 & 7 \\ 5 & 10 & -3 & 0 & 1 \\ 4 & 9 & -4 & 2 & 0 \end{pmatrix} \quad \mathbf{D}^{(4)} = \begin{pmatrix} 0 & 12 & -3 & 2 & 3 \\ -5 & 0 & -8 & -3 & -2 \\ 11 & 16 & 0 & 6 & 7 \\ 5 & 10 & -3 & 0 & 1 \\ 4 & 9 & -4 & 2 & 0 \end{pmatrix}$$

Rys. 2.14: Przebieg algorytmu z iloczynem odległości.

Rysunek 2.14 pokazuje jak zmienia się macierz odległości w kolejnych iteracjach pętli algorytmu.

Okazuje się, że można przyspieszyć ten algorytm i zamiast $n-1$ mnożeń macierzy wykonywać tylko $\lceil \lg(n-1) \rceil$ mnożeń[4]. Z równości (2.2) wynika, że $\mathbf{D}^{(m)} = \mathbf{D}^{(n-1)}$ dla wszystkich $m \geq n-1$. Ponieważ iloczyn odległości jest działaniem łącznym możemy wykorzystując szybkie potęgowanie macierzy wyznaczyć macierz $\mathbf{D}^{(n-1)}$właśnie w czasie $\lceil \lg(n-1) \rceil$:

$$\begin{aligned}
\mathbf{D}^{(1)} &= \mathbf{W}, \\
\mathbf{D}^{(2)} &= \mathbf{W}^2 = \mathbf{W} *_{min} \mathbf{W}, \\
\mathbf{D}^{(4)} &= \mathbf{W}^4 = \mathbf{W}^2 *_{min} \mathbf{W}^2, \\
&\vdots \\
\mathbf{D}^{\left(2^{\lceil \lg(n-1) \rceil}\right)} &= \mathbf{W}^{2^{\lceil \lg(n-1) \rceil}} = \mathbf{W}^{2^{\lceil \lg(n-1) \rceil - 1}} *_{min} \mathbf{W}^{2^{\lceil \lg(n-1) \rceil - 1}}.
\end{aligned}$$

Zgodnie z tym co zauważyliśmy, ponieważ $2^{\lceil \lg(n-1) \rceil} \geq n-1$, więc $\mathbf{D}^{\left(2^{\lceil \lg(n-1) \rceil}\right)} = \mathbf{D}^{(n-1)}$

Rysunek 2.15 przedstawia jak zmodyfikować algorytm 2.12 by uzyskać to przyśpieszenie.

```
1 Function faster-all-pairs-shortest-paths(W):
2     n = rows[W]
3     D^(1) ← W
4     m = 1
5     while n − 1 > m do
6         D^(2m) ←extend-shortest-paths(D^(m), D^(m))
7         m = 2 * m
8     return D^(m)
```

Rys. 2.15: Pseudokod ulepszonego algorytmu z iloczynem odległości wyznaczania najkrótszych ścieżek w grafie reprezentowanym przez macierz sąsiedztwa.

2.2.2 Algorytm Floyda-Warshalla

Algorytm Floyda-Warshalla wykorzystuje inną cechę najkrótszych ścieżek niż ta, z której korzysta algorytm opisany w poprzedniej sekcji. W algorytmie wyko-

[4] Symbol lg oznacza tu logarytm o podstawie równej 2. Symbolem $\lceil x \rceil$ oznaczamy funkcję zwaną sufitem lub cechą górną liczby rzeczywistej x tzn. $\lceil x \rceil = \min\{k \in \mathbb{Z} \colon k \geq x\}$.

rzystującym iloczyn odległości konstruuje się coraz to dłuższe ścieżki, natomiast tutaj ścieżki są konstruowane przechodząc przez coraz większy zbiór wierzchołków. Mówiąc prosto, idea algorytmu Floyda-Warshalla jest następująca: w celu znalezienia najkrótszej ścieżki prowadzącej z wierzchołka i do j przechodzimy po grafie próbując znaleźć ewentualnie pośredni wierzchołek k, którego wbudowanie w drogę umożliwi otrzymanie lepszego wyniku niż dotychczasowa wartość wagi ścieżki z i do j. Przejdźmy teraz do bardziej formalnego opisu. Na początek zdefiniujmy pewne ważne pojęcie. *Wierzchołkiem wewnętrznym* ścieżki $p = (v_m)_{m=1}^{l}$ nazywamy każdy wierzchołek tej ścieżki różny od jej początku i końca (tj. wierzchołka v_1 i v_l). Niech zbiorem wierzchołków grafu G będzie $V = \{1, 2, 3, \ldots, n\}$. Niech $d_{ij}^{(k)}$ dla $k = \{0, 1, \ldots, n\}$ oznacza najmniejszą wagę ścieżki z wierzchołka i do j spośród ścieżek, których wierzchołki wewnętrzne należą do zbioru $\{1, 2, \ldots, k\}$. Prawdziwy jest następujący wzór rekurencyjny:

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{jeśli } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right), & \text{jeśli } k \geq 1. \end{cases} \tag{2.4}$$

Istotnie, dla $k = 0$ prawdziwość wzoru wynika z definicji $\mathbf{D}^{(0)}$. Teraz pokażemy słuszność wzoru dla $k > 0$. Niech p będzie najkrótszą ścieżką prowadzącą z i do j której wierzchołki wewnętrzne należą do zbioru $\{1, 2, \ldots, k\}$ (p jest ścieżką prostą). Musimy rozważyć dwa przypadki. Pierwszy z nich zachodzi, gdy wierzchołek k nie leży na ścieżce p. Wtedy $d_{ij}^{(k)} = w(p) = d_{ij}^{(k-1)}$ oraz $w(p) \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$. W drugim przypadku zakładamy, że wierzchołek k leży na ścieżce p i wtedy występuje on na niej dokładnie raz. Możemy zatem podzielić ścieżkę p na dwie podścieżki p_1 i p_2 takie, że $i \overset{p_1}{\rightsquigarrow} k$ i $k \overset{p_2}{\rightsquigarrow} j$. Wierzchołek k nie jest wierzchołkiem wewnętrznym ścieżek p_1 i p_2. Ponieważ są to podścieżki najkrótszej ścieżki, więc same też muszą być najkrótszymi ścieżkami. Jest zatem $w(p_1) = d_{ik}^{(k-1)}$ oraz $w(p_2) = d_{kj}^{(k-1)}$. Otrzymujemy więc równość $d_{ij}^{(k)} = w(p) = w(p_1) + w(p_2) = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$, a ponieważ p to najkrótsza ścieżka to $w(p) \leq d_{ij}^{(k-1)}$ i wzór (2.4) zachodzi także w tym przypadku.

Korzystając z rekurencyjnego wzoru (2.4) możemy skonstruować algorytm przedstawiony na rysunku 2.16.

Algorytm oblicza ciąg macierzy $(\mathbf{D}^{(0)}, \mathbf{D}^{(1)}, \ldots, \mathbf{D}^{(n)})$ aż do wynaczynienia postaci macierzy szukanej $\mathbf{D}^{(n)}$, która jest macierzą wag najkrótszych ścieżek tj. $\mathbf{D} = \mathbf{D}^{(n)}$. W celu wyznaczania najkrótszych ścieżek należy zbudować macierz poprzedników $\mathbf{P}$. Możemy to zrobić w tym samym czasie, gdy budujemy macierz najkrótszych ścieżek. Będziemy więc wyznaczać ciąg macierzy $(\mathbf{P}^{(0)}, \mathbf{P}^{(1)}, \ldots, \mathbf{P}^{(n)})$, gdzie $\mathbf{P} = \mathbf{P}^{(n)}$ oraz $p_{ij}^{(k)}$ to poprzednik wierzchołka j znajdujący się na najkrótszej ścieżce z wierzchołka

```
1 Function floyd-warshall(W):
2     n = rows[W]
3     D^(0) ← W
4     for k ← 1 to n do
5         for i ← 1 to n do
6             for j ← 1 to n do
7                 d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8     return D^(n)
```

Rys. 2.16: Pseudokod algorytmu Floyda-Warshalla.

i, której wewnętrzne wierzchołki są ze zbioru $\{1, 2, \ldots, k\}$. Możemy ten poprzednik określić rekurencyjnie wzorami:

$$p_{ij}^{(0)} = \begin{cases} \texttt{None} & \text{jeśli } i = j \text{ lub } w_{ij} = \infty \\ i & \text{jeśli } i \geq j \text{ i } w_{ij} < \infty \end{cases}$$

oraz dla $k \geq 1$ tworzymy rekurencyjną funkcję wyznaczającą poprzedniki:

$$p_{ij}^{(k)} = \begin{cases} p_{ij}^{(k-1)} & \text{jeśli } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ p_{kj}^{(k-1)} & \text{jeśli } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Przykład 5. Sprawdzimy teraz działanie algorytmu Floyda-Warshalla dla przykładowego grafu z rusunku 2.13. Przebieg algorytmu obrazuje rysunek 2.17.

$$\mathbf{D}^{(0)} = \begin{pmatrix} 0 & \infty & -3 & 2 & \infty \\ -5 & 0 & 4 & \infty & \infty \\ \infty & \infty & 0 & 6 & \infty \\ \infty & \infty & 1 & 0 & 1 \\ \infty & 9 & -4 & \infty & 0 \end{pmatrix} \quad \mathbf{P}^{(0)} = \begin{pmatrix} None & None & 1 & 1 & None \\ 2 & None & 2 & None & None \\ None & None & None & 3 & 3 \\ None & None & 4 & None & 4 \\ None & 5 & 5 & None & None \end{pmatrix}$$

$$\mathbf{D}^{(1)} = \begin{pmatrix} 0 & \infty & -3 & 2 & \infty \\ -5 & 0 & -8 & -3 & \infty \\ \infty & \infty & 0 & 6 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ \infty & 9 & -4 & \infty & 0 \end{pmatrix} \quad \mathbf{P}^{(1)} = \begin{pmatrix} None & None & 1 & 1 & None \\ 2 & None & 1 & 1 & None \\ None & None & None & 3 & None \\ None & None & 4 & None & 4 \\ None & 5 & 5 & None & None \end{pmatrix}$$

$$\mathbf{D}^{(2)} = \begin{pmatrix} 0 & \infty & -3 & 2 & \infty \\ -5 & 0 & -8 & -3 & \infty \\ \infty & \infty & 0 & 6 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ 4 & 9 & -4 & 6 & 0 \end{pmatrix} \quad \mathbf{P}^{(2)} = \begin{pmatrix} None & None & 1 & 1 & None \\ 2 & None & 1 & 1 & None \\ None & None & None & 3 & None \\ None & None & 4 & None & 4 \\ 2 & 5 & 5 & 1 & None \end{pmatrix}$$

$$\mathbf{D}^{(3)} = \begin{pmatrix} 0 & \infty & -3 & 2 & \infty \\ -5 & 0 & -8 & -3 & \infty \\ \infty & \infty & 0 & 6 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ 4 & 9 & -4 & 2 & 0 \end{pmatrix} \quad \mathbf{P}^{(3)} = \begin{pmatrix} None & None & 1 & 1 & None \\ 2 & None & 2 & 2 & None \\ None & None & None & 3 & None \\ None & None & 4 & None & 4 \\ 2 & 5 & 5 & 3 & None \end{pmatrix}$$

$$\mathbf{D}^{(4)} = \begin{pmatrix} 0 & \infty & -3 & 2 & 3 \\ -5 & 0 & -8 & -3 & -2 \\ \infty & \infty & 0 & 6 & 7 \\ \infty & \infty & \infty & 0 & 1 \\ 4 & 9 & -4 & 2 & 0 \end{pmatrix} \quad \mathbf{P}^{(4)} = \begin{pmatrix} None & None & 1 & 1 & 4 \\ 2 & None & 1 & 1 & 4 \\ None & None & None & 3 & 4 \\ None & None & 4 & None & 4 \\ 2 & 5 & 5 & 3 & None \end{pmatrix}$$

$$\mathbf{D}^{(5)} = \begin{pmatrix} 0 & 12 & -3 & 2 & 3 \\ -5 & 0 & -8 & -3 & -2 \\ 11 & 16 & 0 & 6 & 7 \\ 5 & 10 & -3 & 0 & 1 \\ 4 & 9 & -4 & 2 & 0 \end{pmatrix} \quad \mathbf{P}^{(5)} = \begin{pmatrix} None & 5 & 1 & 1 & 4 \\ 2 & None & 1 & 1 & 4 \\ 2 & 5 & None & 3 & 4 \\ 2 & 5 & 4 & None & 4 \\ 2 & 5 & 5 & 3 & None \end{pmatrix}$$

Rys. 2.17: Przebieg algorytmu Floyda-Warshalla.

2.2.3 Algorytm Johnsona

Ostatnim algorytmem, który omówimy w tym rozdziale jest algorytm Johnsona. Podobnie jak algorytm z iloczynem odległości oraz algorytm Floyda-Warshalla służy on do wyznaczania najkrótszych ścieżek między wszystkim parami wierzchołków w grafie ważonym $G = (V, E, w)$. Algorytm jest efektywniejszy od poprzednich dwóch algorytmów, gdy stosujemy go do grafów rzadkich. Działanie algorytmu oparte jest na omówionych już wcześniej algorytmach Dijkstry oraz Bellmana-Forda. Algorytm Johnsona wykorzystuje implementację grafu w postaci list sąsiedztwa. Wynikiem jego działania jest macierz wag najkrótszych ścieżek, bądź informacja, że graf zawiera ujemny cykl. Algorytm Johnsona wykorzystuje tzw. *metodę zmieniających się wag*. Za pomocą algorytmu Bellmana-Forda wyznaczana jest nowa funkcja wagowa o nieujemnych wartościach oraz sprawdzane jest ewentualne występowanie cykli ujemnych[5)]. Mając do dyspozycji graf z nowymi wagami możemy wyznaczyć najkrótsze ścieżki między wszystkimi parami wierzchołków z wykorzystaniem algorytmu Dijkstry dla każdego wierzchołka oddzielnie (jako wierzchołka źródłowego).

Sama metoda zmieniania wag polega na tym, iż jeśli w grafie występują ujemne wagi krawędzi, to w celu zastosowania algorytmu Dijkstry należy stworzyć nową funkcję wagową w', która posiada dwie bardzo ważne cechy:

1. dla wszystkich krawędzi $(u, v) \in E$ zachodzi nierówność $w'(u, v) \geq 0$,
2. dla wszystkich par wierzchołków $u, v \in V$ najkrótsza ścieżka prowadząca z wierzchołka u do wierzchołka v, dla funkcji wagowej w, jest również najkrótszą ścieżką z u do v dla funkcji wagowej w'.

Warto w tym miejscu zauważyć, że w ogólnym przypadku dodanie do każdej wagi pewnej stałej liczby dodatniej (np. wartości bezwzględnej z minimalnej wagi w grafie) w celu uzyskania wagi w' nie spełnia powyższych warunków (zob. przykład 7).

Jeżeli dany jest skierowany graf ważony $G = (V, E)$ z funkcją wagową $w : E \to \mathbb{R}$ to możemy zdefiniować nową funkcję wagową w', która spełnia wspomniane założenia następująco:

$$w'(u, v) = w(u, v) + h(u) - h(v) \tag{2.5}$$

gdzie h jest funkcją odwzorowującą zbiór wierzchołków V w zbiór liczb rzeczywistych tj. $h : V \to \mathbb{R}$. Łatwo wykazać, że funkcja wagowa zdefiniowana równaniem (2.5) spełnia wspomniane dwa warunki.

[5)] Oczywiście w przypadku wystąpienia cyklu ujemnego działanie całego algorytmy jest przerywane.

Rozważmy dowolną ścieżkę $p = (v_i)_{i=0}^k$ prowadzącą z wierzchołka v_0 do wierzchołka v_k. Pokażemy, że jeżeli p jest najkrótszą ścieżką z funkcją wagową w to jest też najkrótszą ścieżką z v_0 do v_k z funkcją wagową w' oraz jeżeli p jest najkrótszą ścieżką z v_0 do v_k przy funkcji wagowej w' to musi być też najkrótszą ścieżką z wagą w. Inaczej mówiąc równość $w(p) = \delta(v_0, v_k)$ zachodzi wtedy i tylko wtedy, gdy zachodzi $w'(p) = \delta'(v_0, v_k)$, gdzie przez δ' oznaczymy wagi najkrótszych ścieżek grafu G z funkcją wagową w'. Ponadto, graf G ma cykl o ujemnej wadze przy funkcji wagowej w wtedy i tylko wtedy, gdy cykl ten występuje przy funkcji wagowej w'. Na początek pokażemy, że zachodzi równość

$$w'(p) = w(p) + h(v_0) - h(v_k) \tag{2.6}$$

Ponieważ $w'(p)$ jest sumą wag poszczególnych krawędzi na ścieżce p, więc

$$\begin{aligned} w'(p) &= \sum_{i=1}^{k} w'(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} \Big(w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \Big) \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\ &= w(p) + h(v_0) - h(v_k) \end{aligned}$$

co dowodzi równości (2.6). Teraz korzystając z dowodu nie wprost możemy wykazać, że $w(p) = \delta(v_0, v_k)$, wtedy i tylko wtedy gdy $w'(p) = \delta'(v_0, v_k)$. Ponieważ wartości $h(v_0)$ i $h(v_k)$ nie zależą od wyboru ścieżki, to jeżeli przypuścimy, że istnieje krótsza ścieżka p' z v_0 do v_k przy funkcji wagowej w', wtedy $w'(p') < w'(p)$. Z równości (2.6) wynika, że

$$\begin{aligned} w(p') + h(v_0) - h(v_k) &= w'(p') \\ &< w'(p) \\ &= w(p) + h(v_0) - h(v_k) \end{aligned}$$

przez co $w(p') < w(p)$, co prowadzi do sprzeczności.

Wykażemy teraz, że graf G zawiera cykl o ujemnej wadze przy funkcji wagowej w wtedy i tylko w wtedy, gdy przy funkcji w' ten cykl również ma wagę ujemną. Niech będzie dany w grafie G dowolna ścieżka $p = (v_i)_{i=0}^k$, dla której $v_0 = v_k$ (tzn. p jest cyklem). Z równości (2.6) wynika, że

$$\begin{aligned} w'(p) &= w(p) + h(v_0) - h(v_k) \\ &= w(p) \end{aligned}$$

co pokazuje, że cykl p posiada ujemną wagę przy funkcji wagowej w' wtedy i tylko wtedy, gdy przy funkcji wagowej w również jego waga jest ujemna.

Wykazaliśmy już, że funkcja wagowa w' określona równaniem (2.5) spełnia drugi z postulatów. Chcemy teraz tak zdefiniować funkcję h, aby funkcja wagowa w' przyjmowała wartości nieujemne. Mając skierowany graf ważony $G = (V, E)$ z funkcją wagową $w\colon E \to \mathbb{R}$ tworzymy nowy graf G', który zawiera wszystkie wierzchołki grafu G wraz z nowym wierzchołkiem $s \notin V$ oraz wszystkie krawędzie grafu G z dodanymi krawędziami prowadzącymi od wierzchołka s do każdego wierzchołka grafu G. Zatem, budujemy graf $G' = (V', E')$, gdzie $V' = V \cup \{s\}$ oraz $E' = E \cup \Big\{(s, v) : v \in V\Big\}$. Należy zauważyć, że z żadnego wierzchołka grafu G nie osiągniemy wierzchołka s, a stąd wynika, że żadna najkrótsza ścieżka w G' nie będzie zawierać wierzchołka s za wyjątkiem tych o początku w s. Dodatkowo graf G' nie będzie posiadał cykli ujemnych wtedy i tylko wtedy, gdy graf G ich nie ma.

Załóżmy, że grafy G i G' nie mają cykli ujemnych. Dla każdego wierzchołka $v \in V'$ zdefiniujmy $h(v) = \delta(s, v)$. Jedna z własności najkrótszych ścieżek mówi o tym, że spełniona jest tzw. *nierówności trójkąta* tzn. $\delta(s, v) \leq \delta(s, u) + w(u, v)$, gdzie s to wierzchołek źródłowy w skierowanym grafie ważonym G. Wykorzystując, wspomnianą nierówność trójkąta w kontekście naszej definicji funkcji h otrzymujemy, że $h(v) \leq h(u) + w(u, v)$ dla wszystkich krawędzi $(u, v) \in E'$. Jeśli więc zdefiniujemy w' zgodnie z tą definicją funkcji h to $w'(u, v) = w(u, v) + h(u) - h(v) \geq 0$ tzn. nasza funkcja wagowa będzie spełniać pierwszy postulat.

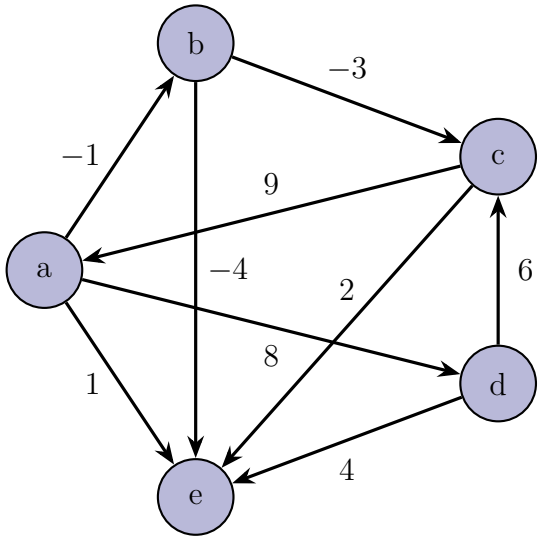
W algorytmie Johnsona, aby zbudować nową funkcję wagową wykorzystujemy algorytm Bellmana-Forda na nowo stworzonym grafie G', który jest opisanym rozszerzeniem grafu G. Mamy więc $h(v) = \delta(s, v)$, gdzie $\delta(s, v)$ zostały wyznaczone właśnie tym algorytmem. Zakładamy, że graf G nie ma cykli ujemnych (jeżeli takowe są to ma je również graf G' i zostaną wykryte przez algorytm Bellmana-Forda co będzie skutkowało przerwaniem całego algorytmu). Otrzymujemy graf G z nowymi wagami o nieujemnych wartościach. Teraz już możemy zastosować algorytm Dijkstry. Algorytm zwraca macierz najkrótszych ścieżek $\mathbf{D} = [d_{ij}]$. Pseudokod algorytmu Johnsona przedstawiony jest na rysunku 2.18.

```
1  Function johnson(G):
2      oblicz G', gdzie V[G'] = V[G] ∪ {s} i E[G'] = E[G] ∪ {(s, v) : v ∈ V[G]}
3      if bellman-ford(G', w, s) = False then
4          wypisz "graf wejściowy zawiera cykl o ujemnej wadze"
5          return
6      else
7          for all v ∈ V[G'] do
8              przypisz do h(v) wartość d[v] obliczoną w algorytmie
                Bellmana-Forda
9          for all (u, v) ∈ E[G'] do
10             w'(u, v) = w(u, v) + h(u) − h(v)
11         niech D = [d_uv] będzie nową macierzą |V| × |V|
12         for all u ∈ V[G] do
13             d[v] = dijkstra(G, w', u)
14             for all v ∈ V[G] do
15                 d_uv = d[v] + h(v) − h(u)
16     return D
```

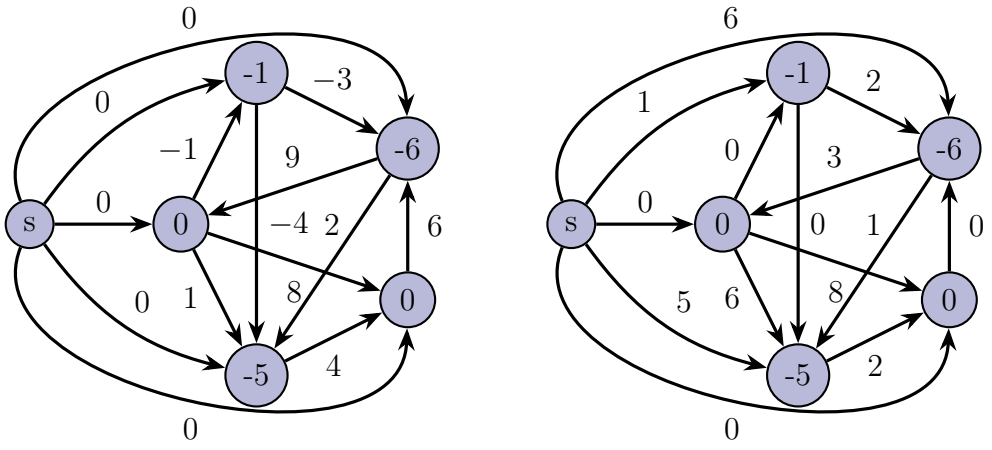
Rys. 2.18: Pseudokod algorytmu Johnsona.

Przykład 6. Przeanalizujemy działanie algorytmu Johnsona dla grafu zaprezentowanego na rysunku 2.19.

Najpierw utworzymy wierzchołek s i dodamy odpowiednie krawędzie oraz wykonamy algorytm Bellmana-Forda na tak zmodyfikowanym grafie (rysunek 2.20). Otrzymujemy nowe wagi, po czym wierzchołek s wraz z krawędziami wychodzącymi od niego, zostaje usunięty.



Rys. 2.19: Przykładowy graf.



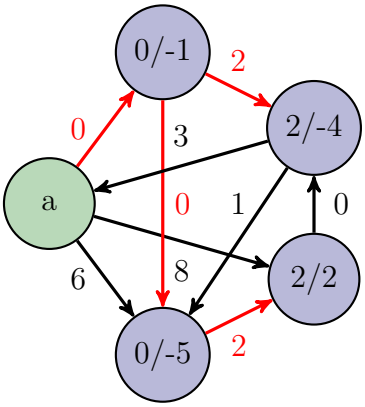
(a) Graf G' po dodaniu wierzchołka s oraz dodatkowych krawędzi.

(b) Graf G' po użyciu algorytmu Bellmana-Forda i określeniu nowych wag.

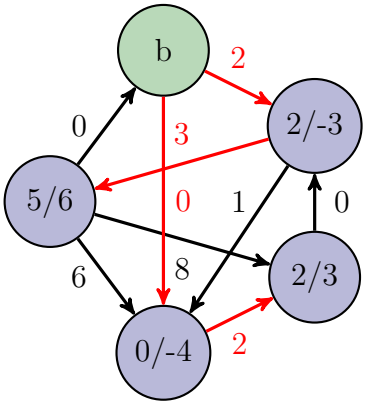
Rys. 2.20: Przebieg metody zmieniających się wag dla grafu z rysunku 2.19.

Po usunięciu ujemnych wag, algorytm Johnsona wykonuje algorytm Dijkstry dla każdego wierzchołka, proces ten możemy zobaczyć na rysunku 2.21. Na wierzchołkach narysowane są dwie wagi oddzielone ukośną kreską, pierwsza (po lewej) to waga zwrócona przez algorytm Dijkstry, druga (po prawej) to realna waga po zmianie funkcji wagowej z w' na w i wykonaniu działania $d_{ij} = d(u, v) + h(v) - h(u)$. Zwracana jest macierz $\mathbf{D} = [d_{ij}]$ zawierająca wagi najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków. Jako że wykorzystujemy algorytm Dijkstry, zwraca on też listę poprzedników więc możemy też dodatkowo zwracać zbiór takich list.

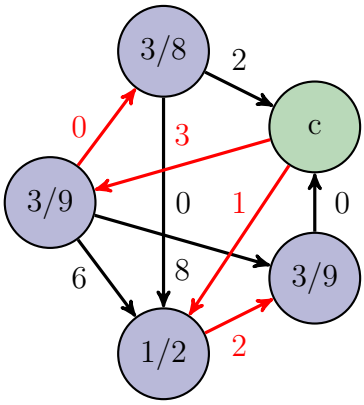
Taki zbiór może mieć też formę macierzy poprzedników.



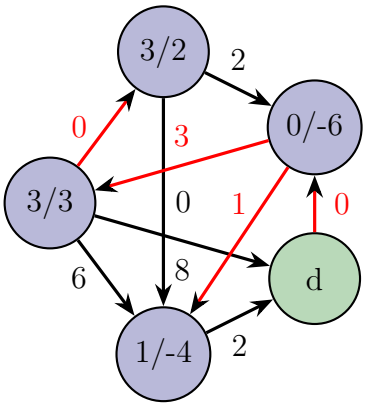
(a) Wynik algorytmu Dijkstry dla wierzchołka a.



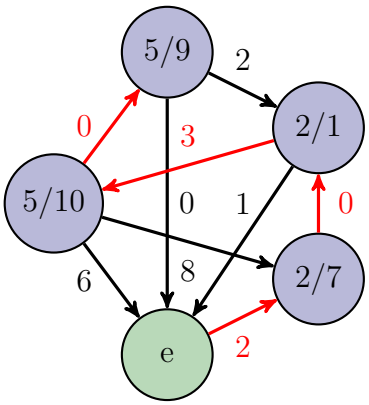
(b) Wynik algorytmu Dijkstry dla wierzchołka b.



(c) Wynik algorytmu Dijkstry dla wierzchołka c.



(d) Wynik algorytmu Dijkstry dla wierzchołka d.

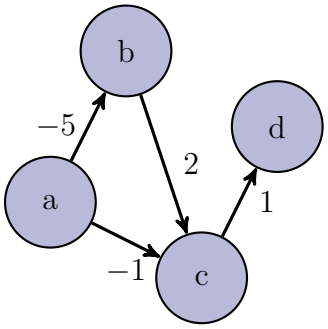


(e) Wynik algorytmu Dijkstry dla wierzchołka e.

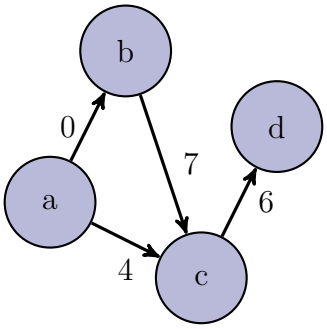
Rys. 2.21: Przebieg wyznaczania najkrótszych ścieżek algorytmem Dijkstry dla każdego wierzchołka na potrzeby algorytmu Johnsona.

Przykład 7. Podamy teraz pewien przykład grafu skierowanego G z funkcją wagową w, dla którego przy wykorzystaniu funkcji wagowej $\widehat{w}(u,v) = w(u,v) + x$, gdzie x

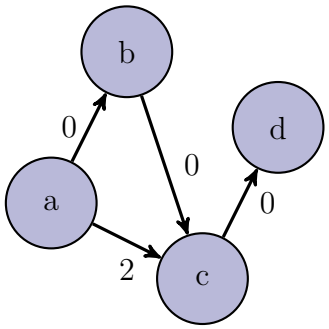
jest pewną stałą dodatnią, najkrótsze ścieżki będą inne niż najkrótsze ścieżki z wykorzystaniem funkcji wagowej w. Za x przyjmiemy liczbę przeciwną do najmniejszej wagi w grafie, usuniemy wtedy wszystkie krawędzie z ujemnymi wagami.



(a) Przykładowy graf.



(b) Użycie funkcji wagowej, $\widehat{w}$ gdzie $x = 5$.



(c) Graf z nowymi wagami wyznaczonymi zgodnie z funkcją wagową w' określoną w algorytmie Johnsona.

Rys. 2.22: Przykładowy graf i grafy ze zmienionymi wagami w celu uzyskania grafu o wagach nieujemnych.

Jak widać na rysunku 2.22b nowa funkcja wagowa $\widehat{w}$ prowadzi do grafu o wagach nieujemnych, lecz najkrótsze ścieżki ulegają zmianie. Przykładowo, w grafie 2.22a najkrótsza ścieżka z wierzchołka a do wierzchołka b prowadzi przez wierzchołek c i waga tej ścieżki wynosi -3. Użycie funkcji wagowej $\widehat{w}$ (dla $x = 5$) do budowy grafu o wagach nieujemnych powoduje, że najkrótszą ścieżką z a do c jest ta bezpośrednia o wadze równej 4 - ścieżka prowadząca przez wierzchołek b ma już wagę równą 7. Na rysunku 2.22c widzimy, że graf powstały z użyciem funkcji wagowej w' określonej w algorytmie Johnsona prowadzi do uzyskania grafu o wagach nieujemnych i nie zmienia najkrótszych ścieżek - np. najkrótsza ścieżka z a do c to ta prowadząca przez wierzchołek b.

Rozdział 3

Implementacja wybranych algorytmów w języku Python

W tym rozdziale zostaną zaprezentowane implementacje w języku Python algorytmów rozwiązujących problemy wyznaczania najkrótszych ścieżek, których pseudokody i omówienie znajduje się w rozdziale 2. Szczegóły samego języka Python można znaleźć w jego obszernej dokumentacji [6]. Część kodów źródłowych wykorzystanych w tym rozdziale zawiera dodatek B.

Implementacja grafu będzie się różnić od wersji, która została podana w rozdziale 1. Była ona bardzo prosta i podana jako przykład możliwej implementacji. Do zaimplementowania naszych algorytmów będzie potrzebna bardziej skomplikowana wersja. W tym celu utworzymy klasę `Graph` zawierającą:

- `vertices` (zbiór wierzchołków) - słownik, którego kluczami są nazwy wierzchołków (przeważnie oznaczane kolejnymi małymi literami alfabetu łacińskiego), a wartościami obiekty klasy `Vertex`. Sama klasa `Vertex` z dekoratorem `dataclass` wygląda jak nazwana krotka (`namedtuple`), ale w przeciwieństwie do niej jest obiektem mutowalnym, wierzchołki będą oznaczane standardowo małymi literami alfabetu, ale mogą też być słowami. Każdy pojedynczy wierzchołek zawiera pola, w których będą zapisywane najkrótsze ścieżki oraz poprzedniki.

- `edges` (zbiór krawędzi) - lista obiektów klasy `Edge` również z dekoratorem `dataclass`. Klasa `Edge` posiada 3 atrybuty: `start` (wierzchołek początkowy u), `end` (wierzchołek końcowy v) oraz `weight` (wartość wagi krawędzi w).

- `neighbors` (zbiór sąsiadów/lista sąsiedztwa) - słownik, którego kluczami są wierzchołki, a wartościami są listy składające się z krotek wierzchołków są-

siadujących z danym kluczem (wierzchołkiem) oraz wag krawędzi, którą są połączone.

- `indexes` - słownik odwzorowujący nazwy wierzchołków na ich pozycję w macierzowej implementacji grafu. Nazwy wierzchołków są posortowane alfabetycznie, przykładowo: `a` będzie pod indeksem 0 (w Pythonie indeksy list zaczynają się od 0), `b` będzie pod indeksem 1, itd.
- `distance_matrix` - lista list reprezentująca macierz wag grafu. Wartość każdego elementu może zawierać jedną z trzech wartości: wagę najkrótszej ścieżki jeśli istnieje ścieżka pomiędzy danymi wierzchołkami, wartość 0 na przekątnej macierzy oraz `None`, gdy nie istnieje ścieżka pomiędzy danymi wierzchołkami.
- `path_matrix` - lista list reprezentująca macierz poprzedników, zawiera poprzednie wierzchołki na najkrótszej ścieżce w grafie. Wartość `None`, oznacza koniec ścieżki lub jej brak.

```
1  from dataclasses import dataclass
2  from math import inf
3
4
5  @dataclass
6  class Edge:
7      start: str
8      end: str
9      weight: int
10
11
12 @dataclass
13 class Vertex:
14     name: str
15     distance: dict = None
16     path: dict = None
17
18
19 class Graph:
20     def __init__(self, vertices: dict, edges: list):
21         self.vertices = vertices
```

```
        self.edges = edges
        self.indexes = {}
        self.neighbors = {v: [(edge.end, edge.weight)
                              for edge in self.edges
                              if edge.start == v]
                          for v in self.vertices}
        for v in sorted(self.vertices):
            self.indexes[v] = i
            i += 1
        n = len(self.vertices)
        self.distance_matrix = [
            [inf] * n for _ in range(n)]
        self.path_matrix = [
            [None] * n for _ in range(n)]
        for vertex in self.vertices:
            for neighbor, weight \
                    in self.neighbors[vertex]:
                self.distance_matrix[
                    self.indexes[vertex]][
                    self.indexes[neighbor]] = weight
                self.path_matrix[
                    self.indexes[vertex]][
                    self.indexes[neighbor]] = neighbor
        for j in range(n):
            self.distance_matrix[j][j] = 0
```

Rys. 3.1: Kod źródłowy implementacji grafu.

3.1 Algorytm Dijkstry

```
import heapq
from math import inf


def dijkstra(g, s):
    s = g.vertices[s]
```

```
7       s.distance = {}
8       s.path = {}
9       for v in g.vertices:
10          s.distance[v] = inf
11          s.path[v] = None
12      s.distance[s.name] = 0
13      nodes = [v for v in g.vertices]
14      h = [(s.distance[s.name], s.name)]
15      while nodes and h:
16          try:
17              min_weight, nearest = heapq.heappop(h)
18              while nearest not in nodes:
19                  min_weight, nearest = heapq.heappop(h)
20          except IndexError:
21              for node in nodes:
22                  s.distance[node] = inf
23                  s.path[node] = None
24              break
25          nodes.remove(nearest)
26          for end, weight in [(x.end, x.weight) for x in
27                              g.edges if x.start == nearest]:
28              w = min_weight + weight
29              if w < s.distance[end]:
30                  s.distance[end] = w
31                  s.path[end] = nearest
32                  heapq.heappush(h,(s.distance[end],end))
33      return s.distance, s.path
```

Rys. 3.2: Kod źródłowy implementacji algorytmu Dijkstry.

Implementacja z rysunku 3.2 nie jest wierną realizacją pseudokodu umieszczonego na rysunku 2.2 z rozdziału 2. Rożnica polega na innej implementacji kolejki Q, gdzie początkowo zawiera ona jeden wierzchołek s (wraz z wagą $d[s]$), a w wyniku relaksacji dodawane są kolejne. Kolejną różnicą jest definicja zbioru S, który początkowo zawiera wszystkie wierzchołki, a następnie usuwane są z niego wierzchołki już przetworzone.

Na początek algorytm inicjuje zmienne, które będą wynikiem jego działania: `distance` oraz `path`, obie zmienne są słownikami języka Python (`dict`) są atry-

butami wierzchołka startowego s. Domyślnie wszystkie odległości wynoszą `inf`, co jest symbolem wartości nieskończonej w języku Python. Odległość od wierzchołka startowego będzie równa 0, ponieważ zakładamy, że w naszym grafie nie będzie pętli własnych. Tworzymy zmienną `nodes`, która zawiera listę (kopię) wszystkich wierzchołków grafu oraz zmienną `h` będącą listą, z której będzie korzystać kolejka priorytetowa - inicjujemy ją krotką zawierającą wagę (równą 0) i nazwę wierzchołka źródłowego.

Algorytm wykonuje pętlę `while` dopóki na liście wierzchołków do przetworzenia (zmienna `nodes`) znajdują się wierzchołki oraz lista h nie jest pusta. W pętli `while` "ściągamy" najbliższy wierzchołek (funkcją `heappop`) z kolejki i przypisujemy go do zmiennej `nearest`. Jeśli wierzchołek już został przetworzony i nie ma go na liście `nodes` bądź kolejka jest pusta, ustawiamy wszystkie nieprzetworzone wierzchołki jako nieosiągalne i kończymy algorytm. Następnie przetwarzamy ściągnięty wierzchołek oraz jego sąsiadów w pętli `for`. Do kolejki dodajemy (funkcja `heappush`) tylko wierzchołki, których aktualna dla algorytmu najkrótsza ścieżka jest najkrótsza (proces relaksacji). Dodatkowo obliczany jest zbiór poprzedników (zmienna `path`), z których można utworzyć najkrótsze ścieżki idąc rekurencyjnie od ostatniego wierzchołka do startowego.

Przykład 8. Rysunek 3.3 przedstawia kod programu testującego implementację algorytmu Dijkstry. Użyty został ten sam graf, który był w przykładzie działania algorytmu w pseudokodzie (rysunek 2.3), wierzchołkiem źródłowym jest wierzchołek `a`.

```
1  from described.dijkstra import dijkstra, print_dijkstra
2  from described.graph import Edge, Graph, Vertex
3
4  G = Graph(vertices={"a": Vertex("a"), "b": Vertex("b"),
5                      "c": Vertex("c"), "d": Vertex("d"),
6                     "e": Vertex("e"), "f": Vertex("f")},
7            edges=[Edge("a", "b", 7), Edge("a", "c", 9),
8                   Edge("a", "f", 14), Edge("b", "c", 10),
9                   Edge("b", "d", 15), Edge("c", "d", 11),
10                  Edge("c", "f", 2), Edge("d", "e", 6),
11                  Edge("e", "f", 9)])
12
13 dijkstra(G, "a")
```

```
14 print_dijkstra(G, "a")
```

Rys. 3.3: Kod źródłowy pokazujący użycie implementacji algorytmu Dijkstry z grafem przedstawionym na rysunku 2.3.

Wynikiem implementacji algorytmu będzie krotka zawierająca dwa słowniki. Pierwszy słownik zawiera odległości od źródła do poszczególnych wierzchołków, drugi słownik zawiera poprzedniki dla wszystkich wierzchołków. Na rysunku 3.4 został zaprezentowany wynik działania algorytmu sformatowany za pomocą funkcji pomocniczej `print_dijkstra`.

```
Algorytm Dijkstry

Droga z a do b (odległość: 7): a -> b
Droga z a do c (odległość: 9): a -> c
Droga z a do d (odległość: 20): a -> c -> d
Droga z a do e (odległość: 26): a -> c -> d -> e
Droga z a do f (odległość: 11): a -> c -> f
```

Rys. 3.4: Wynik uruchomienia programu z rysunku 3.3

3.2 Algorytm Bellmana-Forda

```
1  from math import inf
2
3
4  def bellman_ford(g, s):
5      s = g.vertices[s]
6      s.distance = {}
7      s.path = {}
8      for v in g.vertices:
9          s.distance[v] = inf
10         s.path[v] = None
11     s.distance[s.name] = 0
12     for _ in range(len(g.vertices) - 1):
13         for edge in g.edges:
14             if s.distance[edge.start] != inf and \
15                     s.distance[edge.start] + \
16                     edge.weight < s.distance[edge.end]:
```

```
17                  s.distance[edge.end] = \
18                      s.distance[edge.start] + edge.weight
19                  s.path[edge.end] = edge.start
20      for edge in g.edges:
21          if s.distance[edge.start] != inf and \
22                  s.distance[edge.start] + edge.weight \
23                  < s.distance[edge.end]:
24              return False
25      return s.distance, s.path
```

Rys. 3.5: Kod źródłowy implementacji algorytmu Bellmana-Forda.

W przypadku tej implementacji, kod źródłowy w Pythonie jest praktycznie identyczną adaptacją pseudokodu z rozdziału 2. Program wykonuje relaksację dla każdej krawędzi grafu (czyli `len(g.vertices) - 1`) razy).

Przykład 9. Rysunek 3.6 przedstawia kod programu testującego implementację algorytmu Bellmana-Forda. Użyty został ten sam graf jak na rysunku 2.6. Wynikiem działania algorytmu będzie zwrócenie wartości `False`, czyli graf zawiera cykl ujemny i nie da się wyznaczyć najkrótszych ścieżek.

```
1  from described.bellman_ford import bellman_ford, \
2      print_bellman_ford
3  from described.graph import Edge, Graph, Vertex
4
5  G = Graph(vertices={"a": Vertex("a"), "b": Vertex("b"),
6                      "c": Vertex("c"), "d": Vertex("d"),
7                      "e": Vertex("e")},
8            edges=[Edge("a", "c", 10), Edge("a", "b", 3),
9                   Edge("b", "a", 2), Edge("c", "d", -9),
10                  Edge("d", "e", 7), Edge("e", "a", 1),
11                  Edge("e", "c", -5)])
12
13 print_bellman_ford(G, "a", bellman_ford(G, "a"))
```

Rys. 3.6: Kod źródłowy pokazujący użycie implementacji algorytmu Bellmana-Forda z grafem zawierającym cykl ujemny przedstawionym na rysunku 2.6.

Przykład 10. Następny przykład (rysunek 3.7) będzie sprawdzał działanie dla grafu bez ujemnych cykli. W tym celu również użyty zostanie przykład z rozdziału 2 (rysunek 2.8).

```
from described.bellman_ford import bellman_ford, \
    print_bellman_ford
from described.graph import Edge, Graph, Vertex

G = Graph(vertices={"a": Vertex("a"), "b": Vertex("b"),
                    "c": Vertex("c"), "d": Vertex("d"),
                    "e": Vertex("e")},
          edges=[Edge("a", "b", 4), Edge("a", "c", 5),
                 Edge("b", "a", -3), Edge("b", "c", -4),
                 Edge("c", "d", 7), Edge("d", "b", 9),
                 Edge("d", "e", 10), Edge("e", "b", 8)])

print_bellman_ford(G, "e", bellman_ford(G, "e"))
```

Rys. 3.7: Kod źródłowy pokazujący użycie implementacji algorytmu Bellmana-Forda z grafem niezawierającym cyklu ujemnego przedstawionym na rysunku 2.8.

Wynikiem będzie taka sama krotka jak w przypadku algorytmu Dijkstry. Wynik został wyświetlony z użyciem pomocniczej funkcji `print_bellman_ford`.

```
Algorytm Bellmana-Forda

Droga z e do a (odległość: 5): e -> b -> a
Droga z e do b (odległość: 8): e -> b
Droga z e do c (odległość: 4): e -> b -> c
Droga z e do d (odległość: 11): e -> b -> c -> d
```

Rys. 3.8: Wynik uruchomienia programu z rysunku 3.7 po sformatowaniu.

3.3 Algorytm z iloczynem odległości

Algorytm podstawowy z iloczynem odległości oraz jego ulepszenie są bardzo podobne więc przedstawiona zostanie implementacja ulepszonej wersji. Funkcja została nazwana `faster_matrix_multiply` i została przedstawiona na rysunku 3.9.

```
from copy import deepcopy
from math import inf


```

```
5  def faster_matrix_multiply(g):
6      n = len(g.vertices)
7      distance = deepcopy(g.distance_matrix)
8      m = 1
9      while n - 1 > m:
10         old_distance = deepcopy(distance)
11         for i in range(n):
12             for j in range(n):
13                 distance[i][j] = inf
14                 for k in range(n):
15                     distance[i][j] = \
16                         min(distance[i][j],
17                             old_distance[i][k]
18                             + old_distance[k][j])
19         m = 2 * m
20     g.distance_matrix = deepcopy(distance)
21     return distance
```

Rys. 3.9: Kod źródłowy implementacji algorytmu z iloczynem odległości.

Algorytm wykorzystuje macierzową implementację grafu oraz zmienne zawierające macierze `distance_matrix` i `path_matrix`. Kolejne iteracje grafu powstają przez kopię poprzedniej wersji grafu (bądź grafu bazowego). Należy pamiętać o wykonaniu tzw. kopii głębokiej (deepcopy), gdyż zależy nam na wykonaniu kopii całej struktury, a nie tylko kopii referencji do obiektów. Główną operacją algorytmu jest wybieranie mniejszej odległości, funkcja `min`. Implementacja jest wiernym odwzorowaniem pseudokodu z rysunku 2.15.

Przykład 11. Rysunek 3.10 przedstawia kod programu testującego implementację algorytmu z iloczynem odległości. Użyty został ten sam graf jak na rysunku 2.13.

```
1  from described.graph import Edge, Graph, Vertex
2  from described.matrix_multiply\
3      import faster_matrix_multiply
4
5  G = Graph(vertices=["a", "b", "c", "d", "e"],
6            edges=[Edge("a", "c", -3), Edge("a", "d", 2),
7                   Edge("b", "a", -5), Edge("b", "c", 4),
8                   Edge("c", "d", 6), Edge("d", "e", 1),
```

```
                Edge("e", "b", 9), Edge("e", "c", -4)])

for line in faster_matrix_multiply(G):
    print(line)
```

Rys. 3.10: Kod źródłowy pokazujący użycie implementacji algorytmu z iloczynem odległośći z grafem przedstawionym na rysunku 2.13.

Wynikiem będzie macierz reprezentowana przez listę list najkrótszych ścieżek, wynik pokazany jest na rysunku 3.11.

```
[0, 12, -3, 2, 3]
[-5, 0, -8, -3, -2]
[11, 16, 0, 6, 7]
[5, 10, -3, 0, 1]
[4, 9, -4, 2, 0]
```

Rys. 3.11: Wynik uruchomienia programu z rysunku 3.10

3.4 Algorytm Floyda-Warshalla

Implementacja algorytmu Floyda-Warshalla również będzie korzystać z macierzowej implementacji grafu.

```
def floyd_warshall(g):
    n = len(g.vertices)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if g.distance_matrix[i][j] \
                        > g.distance_matrix[i][k] \
                        + g.distance_matrix[k][j]:
                    g.distance_matrix[i][j] \
                        = g.distance_matrix[i][k] \
                        + g.distance_matrix[k][j]
                    g.path_matrix[i][j] \
                        = g.path_matrix[i][k]
    for vertex in g.vertices:
        g.vertices[vertex].distance = {
            v: g.distance_matrix[g.indexes[vertex]][
```

```
17                  g.indexes[v]] for v in g.vertices}
18          g.vertices[vertex].path = {
19              v: g.path_matrix[g.indexes[vertex]][
20                  g.indexes[v]] for v in g.vertices}
21      return g.distance_matrix, g.path_matrix
```

Rys. 3.12: Kod źródłowy implementacji algorytmu Floyda-Warshalla

Przedstawiona implementacja używa macierzy, które są atrybutami instancji klasy grafu: `distance_matrix` oraz `path_matrix`. Będą one również zwracanymi wynikami algorytmu czyli macierzą najkrótszych ścieżek oraz macierzą poprzedników. Algorytm n razy przechodzi po wszystkich elementach macierzy i zmniejsza odległości przeszukując wszystkie krawędzie zgodnie z założeniami algorytmu.

Przykład 12. Rysunek 3.13 przedstawia kod programu testującego implementację algorytmu Floyda-Warshalla. Użyty został ten sam graf jak na rysunku 2.13.

```
1  from described.floyd_warshall import floyd_warshall, \
2      print_floyd_warshall
3  from described.graph import Edge, Graph, Vertex
4
5  G = Graph(vertices={"a": Vertex("a"), "b": Vertex("b"),
6                      "c": Vertex("c"), "d": Vertex("d"),
7                      "e": Vertex("e")},
8            edges=[Edge("a", "c", -3), Edge("a", "d", 2),
9                   Edge("b", "a", -5), Edge("b", "c", 4),
10                  Edge("c", "d", 6), Edge("d", "e", 1),
11                  Edge("e", "b", 9), Edge("e", "c", -4)])
12
13 print_floyd_warshall(G, floyd_warshall(G))
```

Rys. 3.13: Kod źródłowy pokazujący użycie implementacji algorytmu Floyda-Warshalla z grafem przedstawionym na rysunku 2.13.

Wynikiem działania algorytmu będą listy list reprezentujące macierz najkrótszych ścieżek oraz macierz poprzedników. Rezultat działania programu z rysunku 3.13 przedstawia rysunek 3.14 - również w tym przypadku, użyto do wyświetlenia, osobnej funkcji `print_floyd_warshall`.

```
Algorytm Floyda-Warshalla

Droga z a do b (odległość: 12): a -> d -> e -> b
Droga z a do c (odległość: -3): a -> c
Droga z a do d (odległość: 2): a -> d
Droga z a do e (odległość: 3): a -> d -> e
Droga z b do a (odległość: -5): b -> a
Droga z b do c (odległość: -8): b -> a -> c
Droga z b do d (odległość: -3): b -> a -> d
Droga z b do e (odległość: -2): b -> a -> d -> e
Droga z c do a (odległość: 11): c -> d -> e -> b -> a
Droga z c do b (odległość: 16): c -> d -> e -> b
Droga z c do d (odległość: 6): c -> d
Droga z c do e (odległość: 7): c -> d -> e
Droga z d do a (odległość: 5): d -> e -> b -> a
Droga z d do b (odległość: 10): d -> e -> b
Droga z d do c (odległość: -3): d -> e -> c
Droga z d do e (odległość: 1): d -> e
Droga z e do a (odległość: 4): e -> b -> a
Droga z e do b (odległość: 9): e -> b
Droga z e do c (odległość: -4): e -> c
Droga z e do d (odległość: 2): e -> c -> d
```

Rys. 3.14: Wynik uruchomienia programu z rysunku 3.13

3.5 Algorytm Johnsona

```
1  from described.bellman_ford import bellman_ford
2  from described.dijkstra import dijkstra
3  from described.graph import Graph, Edge, Vertex
4
5
6  def johnson(g: Graph):
7      g.vertices["s"] = Vertex("s")
8      g.neighbors["s"] = []
9      for vertex in g.vertices:
10         g.edges.append(Edge("s", vertex, 0))
11         g.neighbors["s"] += [(vertex, 0)]
12     bf = bellman_ford(g, "s")
13     if not bf:
14         print("Ujemny cykl")
15         return
16     distances_b_f, _ = bf
```

```
17      for edge in g.edges:
18          edge.weight = edge.weight + \
19                        distances_b_f[edge.start] \
20                        - distances_b_f[edge.end]
21      s_edges = [edge for edge in g.edges
22                 if edge.start == "s"]
23      for edge in s_edges:
24          g.edges.remove(edge)
25      g.vertices.pop("s")
26      g.neighbors.pop("s")
27      distances = {}
28      path = {}
29      for vertex in g.vertices:
30          distances[vertex], path[vertex] = \
31              dijkstra(g, vertex)
32      for v_1 in g.vertices:
33          for v_2 in g.vertices:
34              distances[v_1][v_2] += \
35                  distances_b_f[v_2] - distances_b_f[v_1]
36      for edge in g.edges:
37          edge.weight = edge.weight + \
38                        distances_b_f[edge.start] \
39                        - distances_b_f[edge.end]
40      return distances, path
```

Rys. 3.15: Kod źródłowy implementacji algorytmu Johnsona.

Podana implementacja dużo się nie różni od pseudokodu z rozdziału 2. Jedyną większą różnicą jest to, że algorytm Dijkstry zwraca nam odległości w postaci słownika. Zmienna `distances` jest słownikiem, którego kluczami są nazwy wierzchołków, a wartości to również słowniki - zawierają one wyniki algorytmu Dijkstry więc kluczami podsłownika są również nazwy wierzchołków, a wartościami są odległości najkrótszych ścieżek, dokładniej między kluczem słownika `distances`, a kluczem podsłownika.

Przykład 13. Rysunek 3.16 przedstawia kod programu testującego implementację algorytmu Johnsona. Użyty został ten sam graf jak na rysunku 2.19.

```
1  from described.graph import Edge, Graph, Vertex
2  from described.johnson import johnson, print_johnson
```

```
3
4  G = Graph(vertices={"a": Vertex("a"), "b": Vertex("b"),
5                      "c": Vertex("c"), "d": Vertex("d"),
6                      "e": Vertex("e")},
7            edges=[Edge("a", "b", -1), Edge("a", "d", 8),
8                   Edge("a", "e", 1), Edge("b", "c", -3),
9                   Edge("b", "e", -4), Edge("c", "a", 9),
10                  Edge("c", "e", 2), Edge("d", "c", -6),
11                  Edge("e", "d", 7)])
12
13 print_johnson(G, johnson(G))
```

Rys. 3.16: Kod źródłowy pokazujący użycie implementacji algorytmu Johnsona z grafem przedstawionym na rysunku 2.19.

Wynikiem będzie krotka, która zawiera dwie wartości: słownik najkrótszych ścieżek oraz słownik poprzedników. Wynik został wyświetlony z wykorzystaniem pomocniczej funkcji `print_johnson` (zob. rysunek 3.17).

```
Algorytm Johnsona

Droga z a do b (odległość: -1): a -> b
Droga z a do c (odległość: -4): a -> b -> c
Droga z a do d (odległość: 2): a -> b -> e -> d
Droga z a do e (odległość: -5): a -> b -> e
Droga z b do a (odległość: 6): b -> c -> a
Droga z b do c (odległość: -3): b -> c
Droga z b do d (odległość: 3): b -> e -> d
Droga z b do e (odległość: -4): b -> e
Droga z c do a (odległość: 9): c -> a
Droga z c do b (odległość: 8): c -> a -> b
Droga z c do d (odległość: 9): c -> e -> d
Droga z c do e (odległość: 2): c -> e
Droga z d do a (odległość: 3): d -> c -> a
Droga z d do b (odległość: 2): d -> c -> a -> b
Droga z d do c (odległość: -6): d -> c
Droga z d do e (odległość: -4): d -> c -> e
Droga z e do a (odległość: 10): e -> d -> c -> a
Droga z e do b (odległość: 9): e -> d -> c -> a -> b
Droga z e do c (odległość: 1): e -> d -> c
Droga z e do d (odległość: 7): e -> d
```

Rys. 3.17: Wynik uruchomienia programu z rysunku 3.16

Rozdział 4

Złożoność i porównanie algorytmów

Niniejszy rozdział jest poświęcony teoretycznej analizie złożoności poszczególnych algorytmów wyznaczania najkrótszych ścieżek w grafach ważonych przedstawionych w rozdziale 2 oraz potwierdzeniu tej złożoności w przypadku praktycznych implementacji przedstawionych w rozdziale 3. Ograniczmy się tylko do podania złożoności czasowych. W dodatku A umieszczone są podstawowe informacje dotyczące złożoności obliczeniowej i notacji asymptotycznej.

4.1 Złożoności algorytmów

Na początek przytoczymy pewne fakty dotyczące złożoności omawianych algorytmów, które zostały zaczerpnięte z pracy [2]. Następnie przejdziemy do sprawdzenia, czy złożoność przedstawionych w rozdziale 3 implementacji w języku Python tych algorytmów jest zgodna z ich złożonością teoretyczną. Złożoność czasowa danego algorytmu zależy od jego budowy oraz od reprezentacji grafu przyjętej w danym algorytmie i może zależeć od liczby wierzchołków, liczby krawędzi lub jednego i drugiego. W celu porównania rzeczywistej złożoności przedstawionych implementacji z ich teoretyczną złożonością wykorzystamy pewien generator grafów, dzięki któremu będziemy "produkować" grafy o zadanej liczbie wierzchołków, krawędzi, gęstości itp. Następnie korzystając z biblioteki *timeit* języka Python wyznaczymy czas działania algorytmów dla danego grafu, a zbiorcze wyniki przeanalizujemy na wykresach (stworzonych za pomocą biblioteki *matplotlib* języka Python) odnosząc je do założeń teoretycznych. Przypominamy, że będziemy rozważać graf $G = (V, E)$, gdzie V jest zbiorem wierzchołków, a E zbiorem krawędzi[1)].

[1)] W trakcie omawiania złożoności algorytmów podczas używania notacji asymptotycznej będziemy pisać krótko V i E zamiast odpowiednio $|V|$ i $|E|$ w celu oznaczenia liczby wierzchołków

4.1.1 Najkrótsze ścieżki z jednym źródłem

W tej sekcji omówimy złożoności algorytmów, które służą do rozwiązywania problemu wyznaczania najkrótszych ścieżek z jednym źródłem. Na zakończenie omówimy jakie byłyby złożoności algorytmów Dijkstry i Bellmana-Forda, gdyby je wykorzystać do rozwiązania problemu wyznaczania najkrótszych ścieżek między wszystkim parami wierzchołków grafu.

Algorytm Dijkstry

Istotnym elementem w przypadku szybkości działania algorytmu Dijkstry jest sposób zaimplementowania kolejki priorytetowej. Jeśli rozważymy najprostszy przypadek i kolejkę zaimplementujemy jako jednowymiarową listę (tablicę) przeszukiwaną liniowo, to wtedy każda operacji znajdowania elementu trwa $O(V)$. Ponieważ każdy z $|V|$ wierzchołków przechodzi przez tę kolejkę więc łączny czas wyboru wynosi $O(V^2)$. Każda z list sąsiedztwa jest analizowana raz, podobnie jak każda krawędź sąsiadująca z danym wierzchołkiem. Stad, łączny czas działania algorytmu Dijkstry wynosi $O(V^2 + E) = O(V^2)$, gdyż w najgorszym przypadku liczba krawędzi wynosi $O(V^2)$. Lepszą złożoność otrzymamy gdy kolejka będzie zaimplementowana w postaci kopca typu min[2)]. Kopiec jest budowany w czasie $O(V)$, następnie wykonywana jest instrukcja iteracyjna $|V|$ razy. Operacja wyboru każdego elementu minimalnego wynosi $O(\log V)$ (czas potrzebny do przywrócenia własności kopca)[3)]. Zatem, łącznie czas wyboru elementów minimalnych wynosi $O(V \log V)$. Dla każdej krawędzi przeprowadzana jest operacja relaksacji, która może spowodować zmianę wartości w kopcu - przywrócenie własności kopca zajmuje czas $O(\log V)$. Wszystkich krawędzi jest $|E|$, więc ten etap zajmuje $O(E \log V)$, cały algorytm działa w czasie $O\Big((V + E) \log V\Big)$. Dodatkowo, jeżeli zastosowalibyśmy tzw. kopce Fibonacciego to czas przywracania własności takiego kopca wynosi $O(1)$ (czas znajdywania i usuwania elementu minimalnego będzie równy $O(\log V)$), a co za tym idzie łączny czas działania algorytmu w tym przypadku będzie równy $O(V \log V + E)$

Na rysunku 4.1 został przedstawiony wykres zależności czasu wykonania algorytmu Dijkstry zaimplementowanego w rozdziale 3 (implementacja kolejki priorytetowej

i krawędzi, co nie powinno prowadzić do nieporozumień.

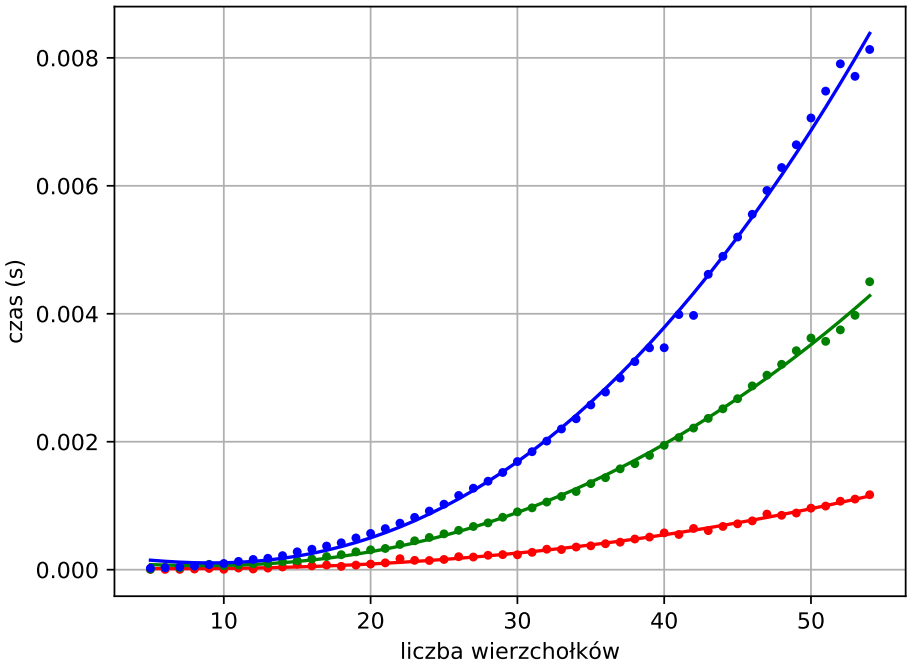
[2)] Jak łatwo zauważyć, po postaci ostatecznego oszacowania złożoności, jest to prawda w przypadku grafów, które nie są gęste.

[3)] Moglibyśmy użyć tu symbolu logarytmu o podstawie równej 2 (lg), ale w kontekście wzoru na zmianę podstawy logarytmu i użytej O-notacji możemy bezkarnie używać symbolu logarytmu dziesiętnego (log).

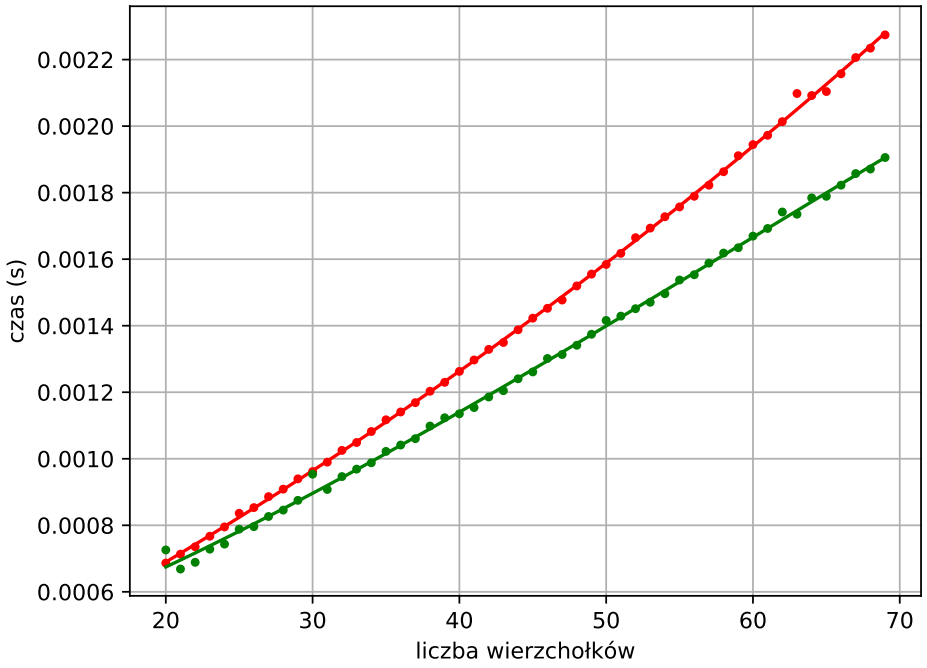
wykorzystuje kopce binarne) w zależności od liczby wierzchołków. Przyjmujemy stałą gęstość grafu. Zostały użyte trzy zbiory grafów o gęstościach odpowiednio 0.1, 0.4 i 0.8. Widać, że algorytm jest dość szybki, a wraz ze wzrostem liczby wierzchołków czas wykonywania rośnie zgodnie z teoretycznym oszacowaniem czasu działania.

Na wykresie 4.2 zostało przedstawione porównanie czasu działania dwóch implementacji algorytmu Dijkstry, w przypadku zmieniającej się liczby wierzchołków i stałej liczby krawędzi (gdyż w takiej sytuacji różnice miedzy wykresami są największe). W pierwszym przypadku zastosowano kolejkę zaimplementowaną za pomocą zwykłej listy z liniowym wyszukiwaniem elementu minimalnego, w drugim przypadku mamy do czynienia z implementacją algorytmu zaprezentowaną w rozdziale 3.

Widać, że algorytm wykorzystujący kolejkę zaimplementowaną jako prostą listę z liniowym wyszukiwaniem działa wolniej niż ten, w którym kolejka zaimplementowana jest przy użyci kopca. Mimo, że mogłoby się wydawać, że różnice w czasach działania są nieznaczne to jednak ta mała różnica może mieć istotne znaczenie na przykład w algorytmie Johnsona i ogólnie w badaniach grafów rzadkich.



Rys. 4.1: Zależność czasu działania algorytmu Dijkstry od liczby wierzchołków w grafie i stałej gęstości (czerwone punkty - gęstość 0.1, zielone punkty - gęstość 0.4, niebieskie punkty - gęstość 0.8). Linie ciągłe oznaczają aproksymacje odpowiednich wykresów punktowych.



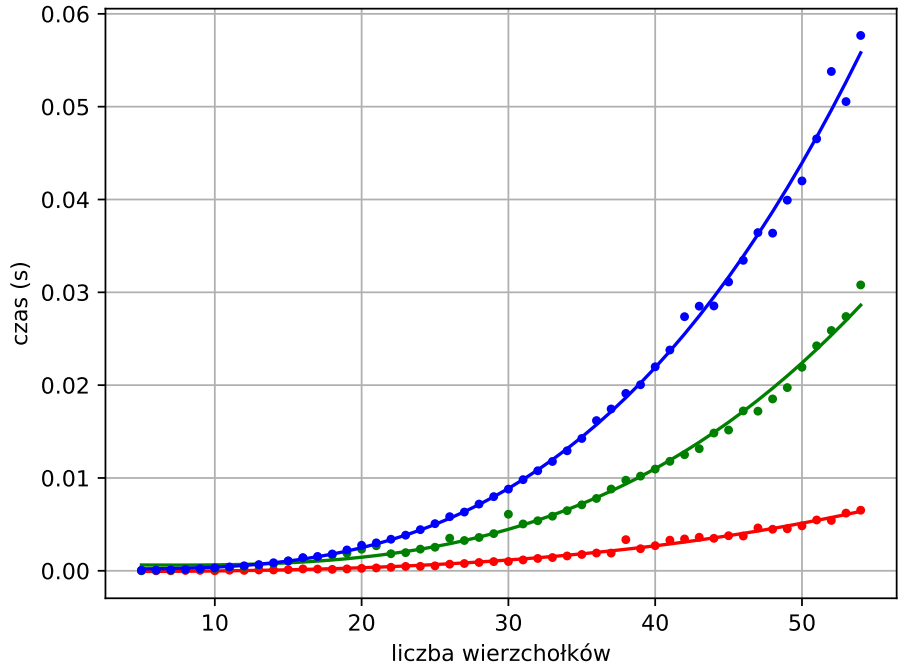
Rys. 4.2: Porównanie czasu działania dwóch implementacji algorytmu Dijkstry w zależności od zmiany liczby wierzchołków przy stałej liczbie krawędzi. Punkty czerwone - implementacja z użyciem kolejki zaimplementowanej jako zwykła lista, punkty zielone - implementacja z użyciem kolejki zaimplementowanej z użyciem kopców binarnych. Linie ciągłe - aproksymacje odpowiednich wykresów punktowych.

Algorytm Bellmana-Forda

Podczas działania algorytmu Bellmana-Forda dokonywana jest $\big(|V|-1\big)$-krotna relaksacja każdej krawędzi. Ponieważ graf zawiera $|E|$ krawędzi i dodatkowo wykonywane jest sprawdzanie wystąpienia cykli ujemnych ($|E|$-krotna iteracja), więc czas działania algorytmu wynosi $O(VE+E) = O(VE)$. Jeżeli mamy do czynienia z grafami gęstymi $|E| = O(V^2)$ to złożoność algorytmu Bellmana-Forda wynosi $O(V^3)$, a w przypadku grafów rzadkich $|E| = O(V)$ mamy do czynienia ze złożonością równą $O(V^2)$.

Na rysunku 4.3 został pokazany wykres zależności czasu wykonania algorytmu Bellmana-Forda (implementacja przestawiona w rozdziale 3) w zależności od liczby wierzchołków i stałej gęstości. Zostały użyte trzy zbiory grafów o gęstościach odpowiednio równych: 0.1, 0.4 i 0.8. Zależności czasu działania algorytmu od zmiany liczby wierzchołków i krawędzi (różne gęstości) są zgodne z teoretycznym oszaco-

waniem. Porównując szybko wykresy z rysunków 4.1 i 4.3 widzimy, że algorytm Bellmana-Forda jest ok. dziesięciokrotnie wolniejszy od algorytmu Dijkstry. Jest to oczywiście cena tego, że w przypadku algorytmu Bellmana-Forda możemy badać grafy o wagach ujemnych, jak również wykrywać cykle ujemne.



Rys. 4.3: Zależność czasu wykonywania algorytmu Bellmana-Forda od liczby wierzchołków przy stałej gęstości (punkty czerwone - gęstość 0.1, punkty zielone - gęstość 0.4, punkty niebieskie - gęstość 0.8).

Algorytm Dijkstry i Bellmana-Forda w kontekście problemu najkrótszych ścieżek między wszystkimi parami wierzchołków

Jasne jest, że problem najkrótszych ścieżek między wszystkimi parami wierzchołków może być rozwiązany, jeżeli wykonamy $|V|$ razy algorytm dla problemu najkrótszych ścieżek z jednym wierzchołkiem źródłowym. Jeżeli badamy graf ważony o wagach nieujemnych możemy zastosować algorytm Dijkstry. Jeżeli implementacja kolejki priorytetowej będzie oparta o jednowymiarową tablicę z wyszukiwaniem liniowym, to czas działania takiego algorytmu będzie wynosił $O(V^3 + VE) = O(V^3)$. Natomiast, jeżeli do implementacji kolejki użyjemy kopców binarnych to złożoność algorytmu będzie równa $O(VE \log V)$, a w przypadku użycia kopów Fibonaciego mamy czas działania równy $O(V^2 \log V + VE)$.

Z algorytmu Dijkstry nie możemy skorzystać, jeżeli w grafie dopuszczamy wa-

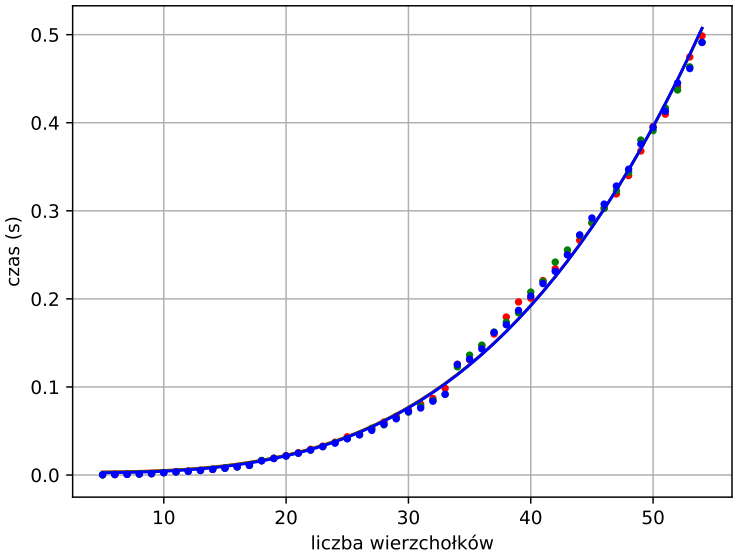
gi o wartościach ujemnych. W takim przypadku możemy wykorzystać wolniejszy algorytm Bellmana-Forda używając go po razie dla każdego wierzchołka grafu. Czas działania tak otrzymanego algorytmu wynosi $O(V^2E)$, co dla grafów gęstych $\left(|E| = O(V^2)\right)$ daje $O(V^4)$.

4.1.2 Najkrótsze ścieżki między wszystkimi parami wierzchołków

Przejdziemy teraz do omówienia czasu działania trzech algorytmów służących do rozwiązywania problemu wyznaczania najkrótszych ścieżek między wszystkimi parami wierzchołków w grafach ważonych.

Algorytm z iloczynem odległości

W rozdziale 3 wykazaliśmy, że algorytm z iloczynem odległości możemy w łatwy sposób sprowadzić do problemu mnożenia macierzy. Stąd, czas działania tego algorytmy wynosi $O(V^4)$ - pojedyncze wykonanie mnożenia (iloczynu odległości) macierzy ma złożoność $O(V^3)$, ponadto algorytm oblicza ciąg $(V-1)$ macierzy. W ulepszonej wersji algorytmu liczba obliczanych iloczynów jest zredukowana do $\log(V-1)$, co daje łączny czas działania algorytmu równy $O(V^3 \log V)$.



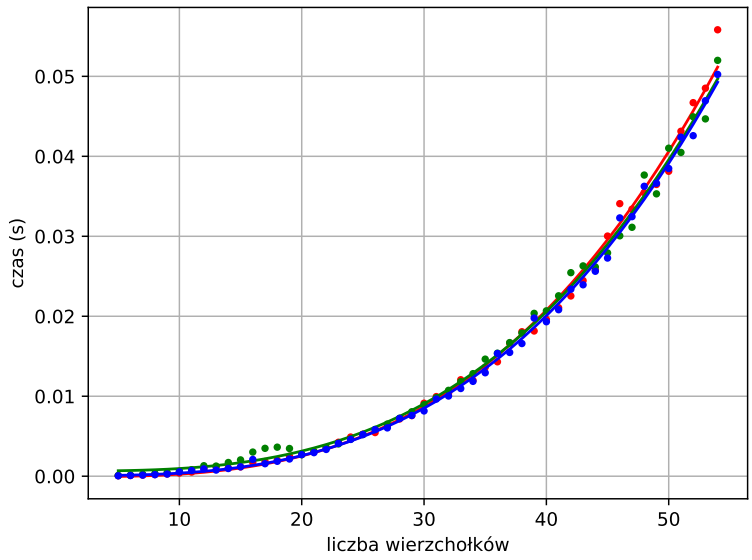
Rys. 4.4: Zależność czasu wykonywania algorytmu z iloczynem odległości od liczby wierzchołków grafu przy stałej gęstości (kolor czerwony - gęstość 0.1, kolor zielony - gęstość 0.4, kolor niebieski - gęstość 0.8).

Rysunek 4.4 przedstawia wykres, na którym widzimy zależność czasu wykonania

ulepszonej wersji algorytmu z iloczynem odległości zaimplementowanego w rozdziale 3 od liczby wierzchołków dla grafów o stałej gęstości. Tak jak we wcześniejszych analizach zostały użyte trzy zbiory grafów o gęstościach odpowiednio 0.1, 0.4 i 0.8. Złożoność czasowa tego algorytmu, jak i algorytmu Floyda-Warshalla nie jest zależna od liczby krawędzi (obydwa algorytmy wykorzystują reprezentację macierzową grafu), a co za tym idzie zmiana gęstości przy ustalonej liczbie wierzchołków nie ma wpływu na czas działania - widzimy, że wykresy dla wspominanych gęstości pokrywają się.

Algorytm Floyda-Warshalla

Złożoność czasowa algorytmu Floyda-Warshalla jest bardzo prosta do wyznaczenia. Wykonywanych jest $|V|$ iteracji dla macierzy o wymiarach $|V| \times |V|$ co daje złożoność $O(V^3)$.



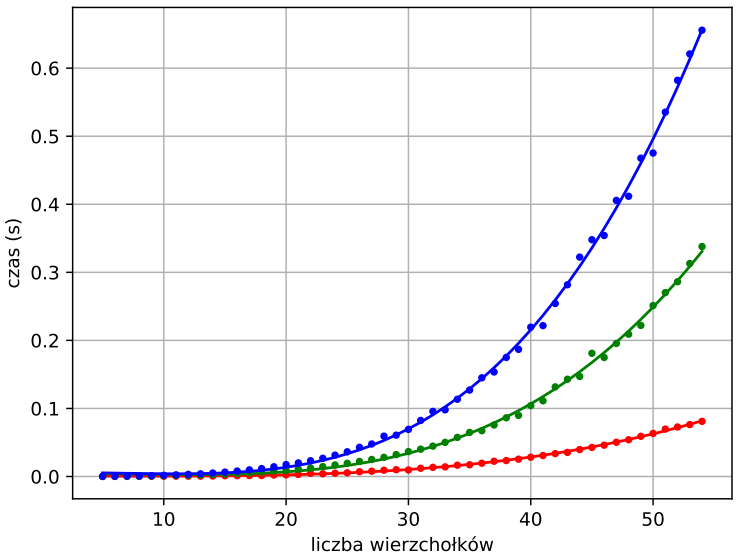
Rys. 4.5: Zależność czasu wykonywania algorytmu Floyda-Warshalla od liczby wierzchołków grafu przy stałej gęstości (kolor czerwony - gęstość 0.1, kolor zielony - gęstość 0.4, kolor niebieski - gęstość 0.8).

Na rysunku 4.5 został pokazany wykres zależności czasu wykonania algorytmu Floyda-Warshalla zaimplementowanego w rozdziale 3 od liczby wierzchołków dla grafów o stałej gęstości. Podobnie jak poprzednio zostały użyte trzy zbiory grafów o gęstościach 0.1, 0.4 i 0.8. Jak już wypominaliśmy przy okazji analizy algorytmu z iloczynem odległości, wykres ukazuje brak zależności czasu działania algorytmu Floyda-Warshall od liczby krawędzi grafu. Porównując wykresy na rysunkach 4.4

i 4.5 widzimy, że zgodnie z oczekiwaniami algorytm Floyda-Warshalla jest znacznie szybszy niż algorytm z iloczynem odległości.

Algorytm Johnsona

W algorytmie Johnsona budowa nowego grafu za pomocą algorytmu Bellmana-Forda wykonywane jest w czasie $O(VE)$. Dla tak zbudowanego grafu używamy algorytmu Dijkstry więc złożoność jest zależna od implementacji algorytmu Dijkstry. W przypadku implementacji używającej kolejki priorytetowej opartej na kopcach binarnych złożoność wyniesie $O(VE \log V + VE)$, używając bardziej skomplikowanej implementacji kolejki opartej na kopcach Fibonacciego uzyskamy złożoność $O(V^2 \log V + VE)$. Jak łatwo zauważyć algorytm Johnsona ma niższą złożoność dla grafów rzadkich niż algorytm Floyda-Warshalla. W przypadku grafów gęstych algorytm Floyda-Warshalla o złożoności $O(V^3)$ jest o wiele bardziej wydajny.



Rys. 4.6: Wykres zależności czasu działania algorytmu Johnsona od liczby wierzchołków przy stałej gęstości. (punkty czerwone - gęstość 0.1, punkty zielone - gęstość 0.4, punkty niebieskie - gęstość 0.8)

Na rysunku 4.6 został pokazany wykres zależności czasu wykonania algorytmu Johnsona zaimplementowanego w rozdziale 3 od liczby wierzchołków dla grafów o stałej gęstości. Tu również zostały użyte trzy zbiory grafów o gęstościach 0.1, 0.4 i 0.8. Z wykresu możemy zauważyć, że czas wykonania algorytmu Johnsona szybko rośnie dla dużych ilości wierzchołków i dużych gęstości grafów. Jeśli porównamy wykres dla algorytmu Floyda-Warshalla i wykres dla algorytmu Johnsona można

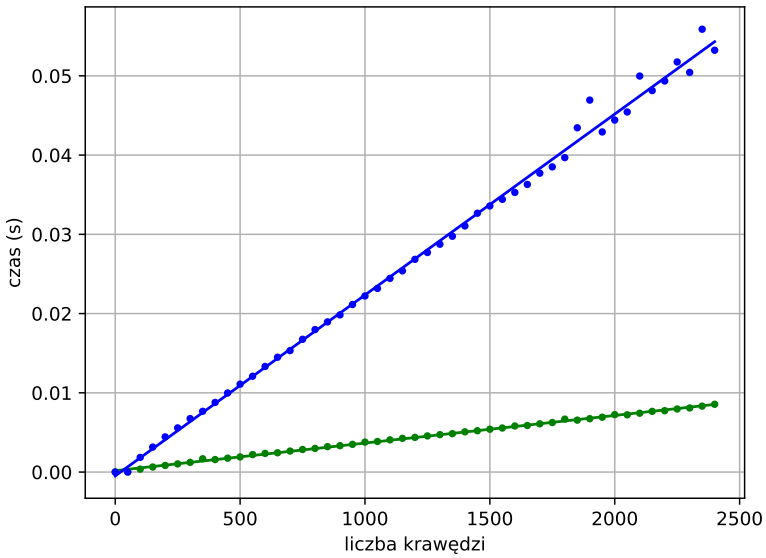
stwierdzić, że na ogół algorytm Floyda-Warshalla jest znacznie szybszy jednak dla grafów wystarczająco rzadkich, algorytm Johnsona działa efektywniej - wrócimy do tego problem w dalszej części tego rozdziału.

4.2 Dalsze porównania czasów działania algorytmów

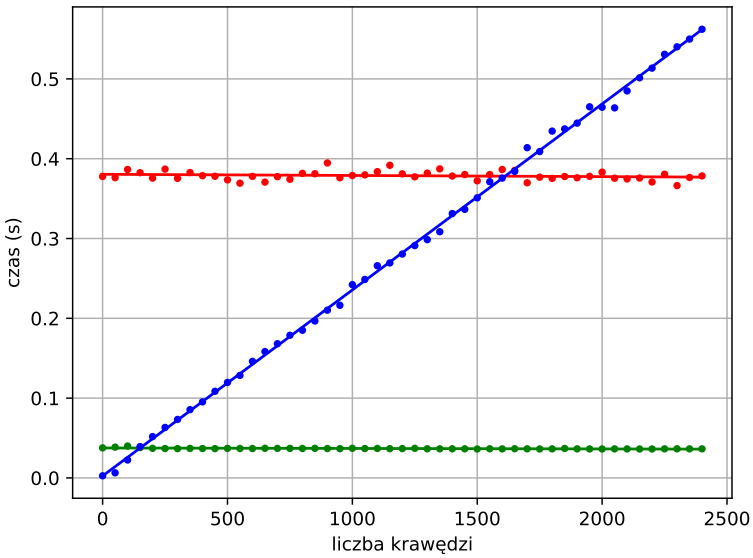
W tej sekcji skupimy się na dodatkowej analizie czasu działania algorytmów zaimplementowanych w rozdziale 3 w celu sprawdzenia czy złożoność czasowa tych algorytmów jest zgodna z założeniami. Podczas omawiania złożoności czasowej algorytmów analizowaliśmy już czas działania naszych implementacji dla zmiennej liczby wierzchołków, ale przy zachowaniu stałej gęstości grafu.

Porównanie dla stałej liczby wierzchołków

Porównamy teraz między sobą czasy działania wybranych algorytmów dla grafów o stałej liczbie wierzchołków, lecz zmieniającej się liczbie krawędzi. Liczba wierzchołków będzie stała i równa 50, liczba krawędzi będzie rosła od 0 do maksymalnej ilości krawędzi w grafie z 50 wierzchołkami tj. wartości równej 2450 z krokiem równym 50.



Rys. 4.7: Zależność czasu działania algorytmów Dijkstry i Bellmana-Forda od liczby krawędzi przy stałej liczbie wierzchołków (zielone punkty - algorytm Dijkstry, niebieski punkty - algorytm Bellmana-Forda). Linie ciągłe oznaczają aproksymacje odpowiednich wykresów punktowych.



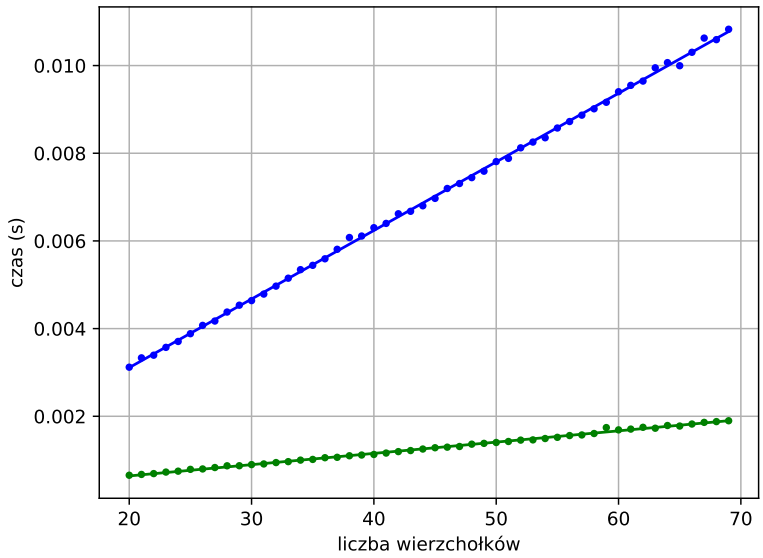
Rys. 4.8: Zależność czasu działania algorytmów z iloczynem odległości, Floyda-Warshalla i Johnsona od liczby krawędzi przy stałej liczbie wierzchołków (czerwone punkty - algorytm z iloczyn odległości, zielone punkty - algorytm Floyda-Warshalla, niebieskie punkty - algorytm Johnsona). Linie ciągłe oznaczają aproksymacje odpowiednich wykresów punktowych.

Porównanie algorytmu Dijkstry i Bellmana-Forda (rysunek 4.7) jest jednoznaczne. Algorytm Dijkstry jest stanowczo szybszy niż Bellmana-Forda. Porównanie algorytmów z iloczynem odległości, Floyda-Warshalla oraz algorytmu Johnsona (rysunek 4.8) również potwierdza teoretyczne założenia, algorytm Johnsona dla małej ilości krawędzi (czyli dla grafów rzadkich) będzie szybszy, lecz muszą to być naprawdę rzadkie grafy; po przekroczeniu pewnej ilości krawędzi czas wykonywania algorytmu Johnsona rośnie liniowo w przeciwieństwie do algorytmów z iloczynem odległości i Floyda-Warshalla, które nie są zależne od ilości krawędzi.

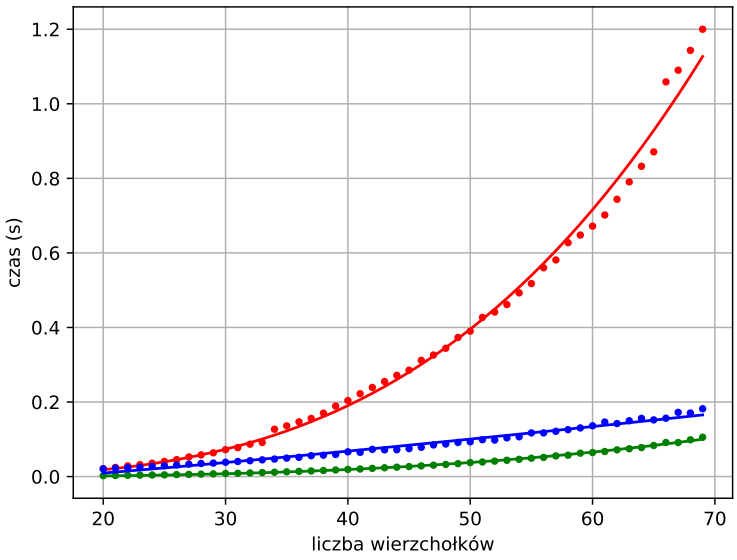
Porównanie dla stałej liczby krawędzi

Dokonamy teraz porównania między sobą czasów działania wybranych algorytmów dla grafów o stałej liczbie krawędzi, lecz zmieniającej się liczbie wierzchołków. W tym przypadku ilość wierzchołków będzie rosła od 20 do 70, ilość krawędzi będzie stała i równa 380 (maksymalna liczba krawędzi dla grafu o 20 wierzchołkach). Oczywiście w tym przypadku gęstość grafu będzie się stale zmniejszać.

Zgodnie z oczekiwaniami algorytm Dijkstry w tym przypadku również jest lepszy niż algorytm Bellmana-Forda (rysunek 4.9).

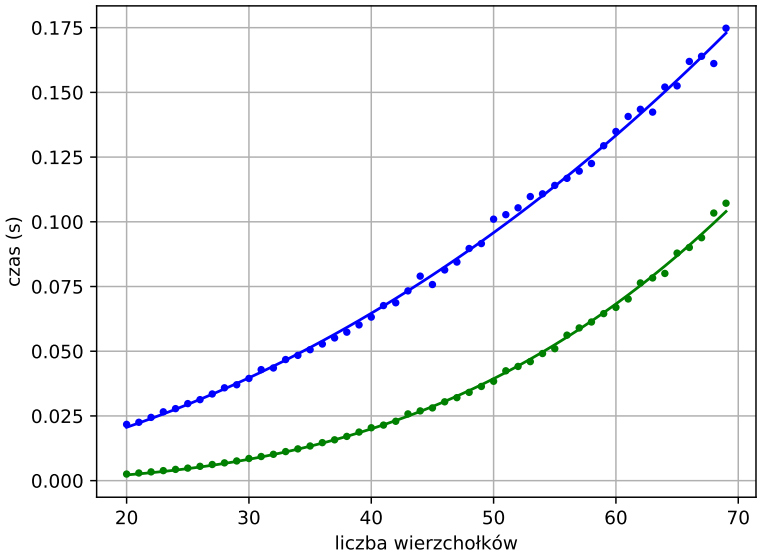


Rys. 4.9: Zależność czasu działania algorytmów Dijkstry i Bellmana-Forda od liczby wierzchołków przy ustalonej liczbie krawędzi (kolor zielony - algorytm Dijkstry, kolor niebieski - algorytm Bellmana-Forda).



Rys. 4.10: Zależność czasu działania algorytmów z iloczynem odległości, Floyda-Warshalla i Johnsona od liczby wierzchołków przy stałej liczbie krawędzi w grafie (czerwone punkty - algorytm z iloczynem odległości, zielone punkty - algorytm Floyda-Warshalla, niebieskie punkty - algorytm Johnsona). Linie ciągłe oznaczają aproksymacje odpowiednich wykresów punktowych.

Podobnie porównanie algorytmów z iloczynem odległości, Floyda-Warshalla oraz Johnsona pokazuje, że wraz ze wzrostem liczby wierzchołków czas wykonywania znacznie wzrasta zgodnie z teoretycznymi oczekiwaniami. Spośród tych trzech algorytmów najwolniejszy jest algorytm z iloczynem odległości. Wzrost czasu działania algorytmu Johnsona i Floyda-Warshalla jest podobny, lecz to ten drugi przeważnie jest szybszy. W celu lepszego porównania tych dwóch algorytmów użyjemy osobnego wykresu (rysunek 4.11). Widać, że w przypadku stałej ilości krawędzi czas działania algorytmu Floyda-Warshalla jest lepszy, ale rośnie z podobną szybkością jak czas działania algorytmu Johnsona.



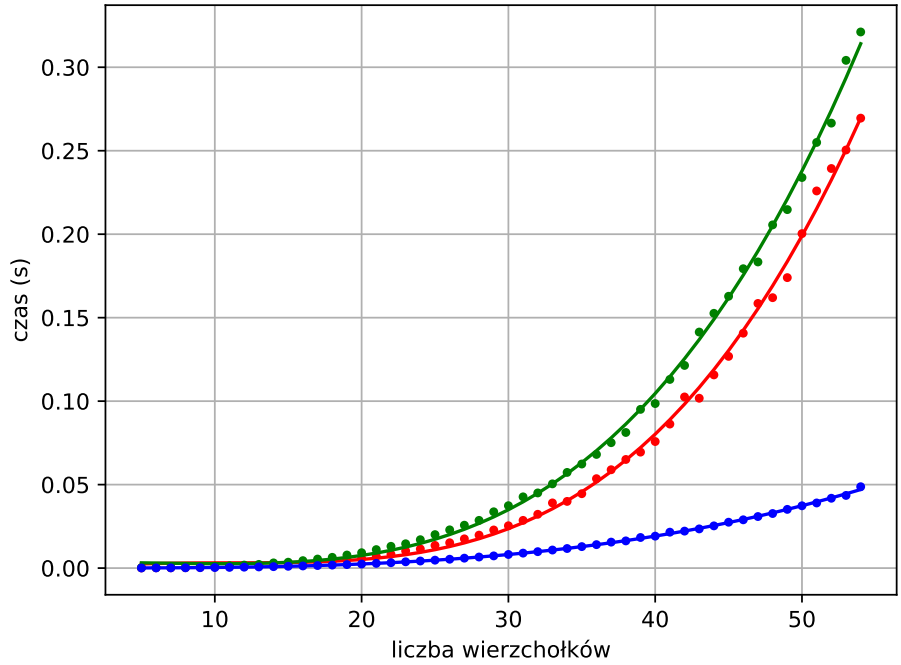
Rys. 4.11: Porównanie czasu działania algorytmów Floyda-Warshalla i Johnsona w zależności od liczby wierzchołków przy ustalonej liczbie krawędzi (zielone punkty - Floyda-Warshalla, niebieskie punkty - algorytm Johnsona). Linie ciągłe oznaczają aproksymacje odpowiednich wykresów punktowych.

Algorytm Dijkstry dla ścieżek między wszystkim parami wierzchołków

Przedstawimy teraz porównanie algorytmu Dijkstry w przypadku zastosowania go do wyznaczania najkrótszych ścieżek między wszystkimi parami wierzchołków z algorytmem Floyda-Warshalla oraz algorytmem Johnsona. W tym celu wykorzystamy grafy o stałej gęstości równej 0.4. Oczywiście należy uwzględnić fakt, że ponieważ używamy algorytmu Dijkstry to rozważamy grafy o nieujemnych wagach.

Rysunek 4.12 pokazuje porównanie czasów działania wspomnianych algorytmów.

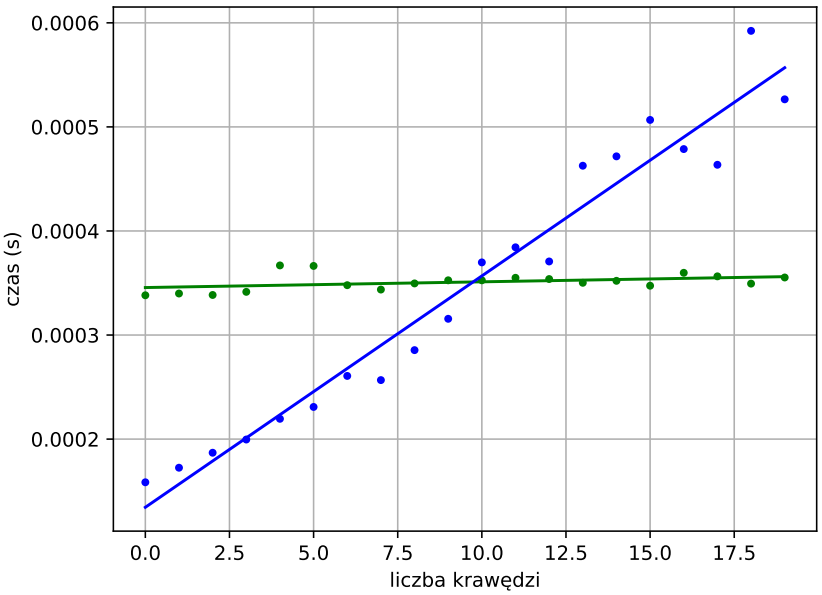
Analizując ten wykres mogłoby się wydawać, że tak użyty algorytm Dijsktry nie jest znacząco lepszy niż algorytm Johnsona. Pamiętajmy jednak, że algorytm Johnsona wykorzystuje algorytm Dijsktry właśnie w takiej postaci, a badamy tu czas działania tylko pod względem zmiany liczby wierzchołków. Ponadto po raz kolejny widać, że algorytm Floyda-Warshalla jest praktycznie zawsze najlepszym wyborem w przypadku problemu najkrótszych ścieżek miedzy wszystkimi parami wierzchołków.



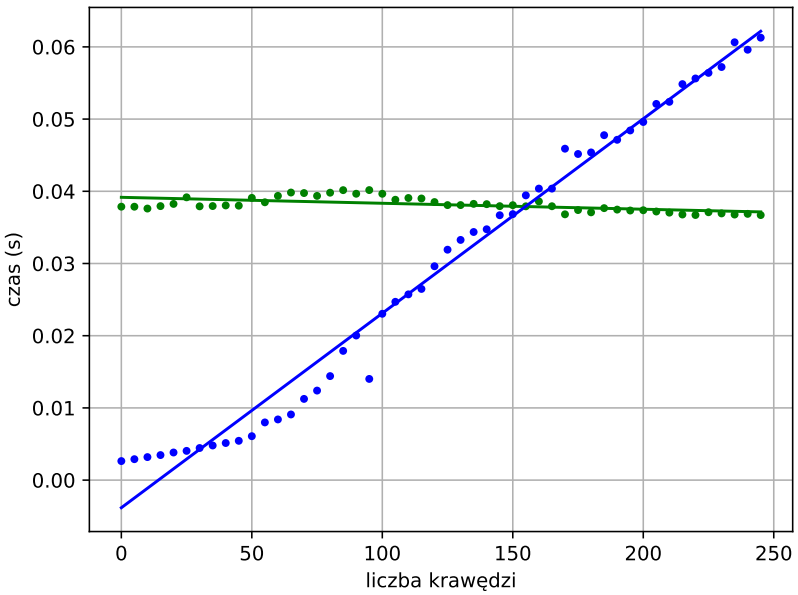
Rys. 4.12: Zależność czasu działania algorytmów Johnsona, Dijkstry dla wszystkich wierzchołków oraz Floyda-Warshalla od liczby wierzchołków przy stałej gęstości grafu (czerwone punkty - algorytm Dijkstry dla wszystkich wierzchołków, zielone punkty - algorytm Johnsona, niebieskie punkty - algorytm Floyd-Warshall).

Algorytm Johnsona a pojęcie grafu gęstego

Wspominaliśmy już w tym rozdziale, że algorytm Johnsona jest szybszy niż Floyda-Warshalla dla grafów rzadkich. Problemem jest jednak fakt, że nie ma ścisłej definicji kiedy dany graf możemy uznać za gęsty lub rzadki. W niniejszej sekcji przeprowadzimy krótką analizę, aby spróbować określić wartość graniczną gęstości dla której powiemy, że graf jest już gęsty. W tym celu będziemy badać czasy działania algorytmu Johnsona i Floyda-Warshalla.



Rys. 4.13: Zależność czasu działania algorytmów Johnsona i Floyda-Warshalla od liczby krawędzi przy stałej liczbie wierzchołków grafu równej 10 (kolor zielony - algorytm Floyda-Warshalla, kolor niebieski - algorytm Johnson).

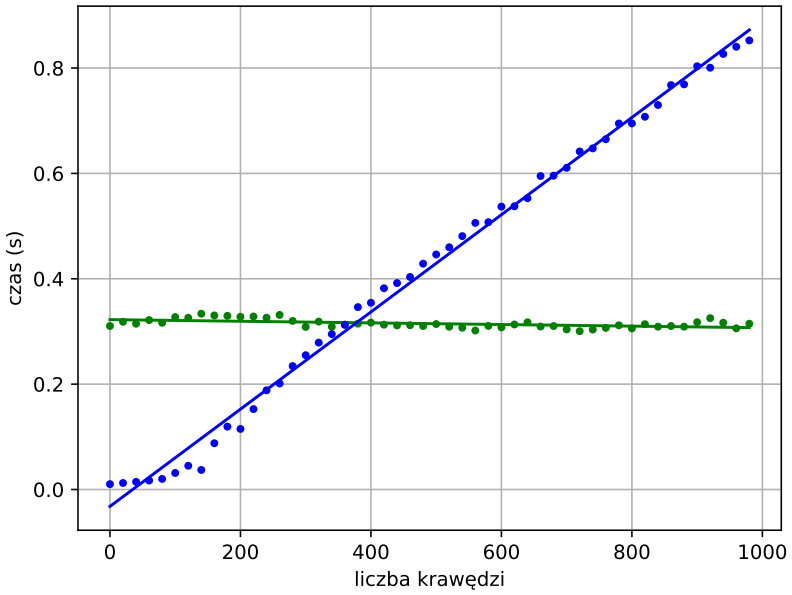


Rys. 4.14: Zależność czasu działania algorytmów Johnsona i Floyda-Warshalla od liczby krawędzi przy stałej liczbie wierzchołków grafu równej 50 (kolor zielony - algorytm Floyda-Warshalla, kolor niebieski - algorytm Johnsona).

Zacznijmy od przypadku grafu mającego 10 wierzchołków, w którym będziemy zwiększać liczbę krawędzi od 0 do 20. Stosowne porównanie czasów działania algorytmów przedstawia rysunek 4.13. Możemy w przybliżeniu stwierdzić, że dla grafów, które mają powyżej dziewięciu krawędzi algorytm Floyda-Warshalla jest już szybszy. Dla grafu o 10 wierzchołkach jest to gęstość w przybliżeniu równa 0.1, czyli bardzo mała.

Następny analizowany przypadek to grafy ze stałą liczbą wierzchołków równą 50 oraz liczbą krawędzi zmieniającą się od 0 do 250 z krokiem co 5. Porównanie czasu działania obu algorytmów przedstawia wykres na rysunku 4.14. Teraz, algorytm Johnsona staje się wolniejszy dla liczby krawędzi powyżej wartości ok. 150 krawędzi, co daje gęstość równą 0.06.

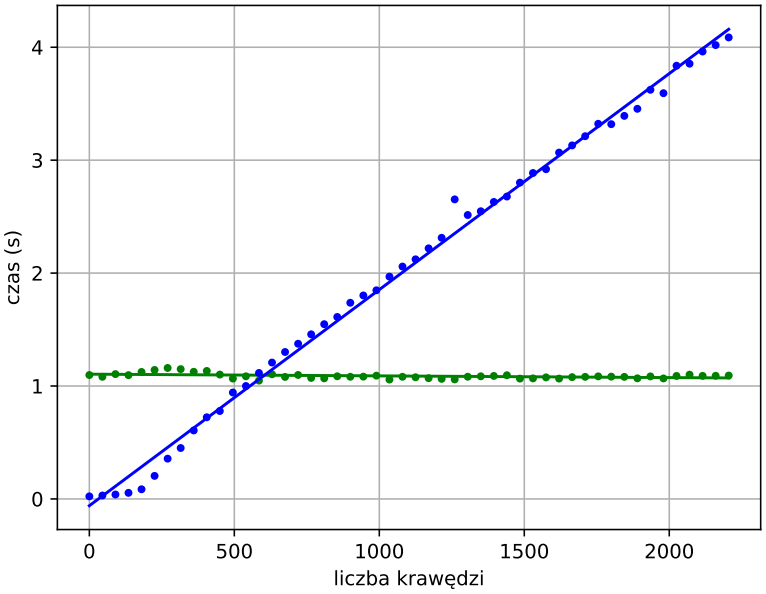
Kolejny przypadek to grafy o liczbie wierzchołków równej 100 i liczbie krawędzi zmieniających się od 0 do 1000 z krokiem co 20. Stosowane porównanie czasu działania algorytmów przedstawia rysunek 4.15. Z wykresu można odczytać, że dla liczby krawędzi większej niż 400 algorytm Floyda-Warshalla działa szybciej, co daje gęstość grafu równą ok. 0.04. Możemy już przypuszczać, że nasza szukana wartość gęstości nie jest stała i maleje.



Rys. 4.15: Zależność czasu działania algorytmów Johnsona i Floyda-Warshalla od liczby krawędzi przy stałej liczbie wierzchołków grafu równej 100 (kolor zielony - algorytm Floyda-Warshalla, kolor niebieski - algorytm Johnsona).

Ostatni przypadek, który rozważymy to grafy o stałej liczbie wierzchołków równej

150 i zmieniającej się liczbie krawędzi od 0 do 22350 z krokiem co 45. Porównanie czasu działania obu algorytmów dla takich grafów przedstawia wykres na rysunku 4.16. Teraz, algorytm Johnsona staje się wolniejszy dla liczby krawędzi powyżej wartości ok. 600, co daje gęstość równą 0.02. Wynik ten potwierdza poprzednie przypuszczenia.



Rys. 4.16: Zależność czasu działania algorytmów Johnsona i Floyda-Warshalla od liczby krawędzi przy stałej liczbie wierzchołków grafu równej 150 (kolor zielony - algorytm Floyda-Warshalla, kolor niebieski - algorytm Johnsona).

Na podstawie przeprowadzonej tej krótkiej analizy możemy stwierdzić, że mówienie o grafach rzadkich i gęstych chociażby tylko w kontekście algorytmu Johnsona jest dość niejednoznaczne i nie da się określić wartości granicznej gęstości dla której powiemy, że dany graf jest rzadki lub gęsty.

4.3 Podsumowanie

Do wyznaczania ścieżek z jednym wierzchołkiem źródłowym w grafie ważonym bez wag ujemnych najlepszym wyborem będzie algorytm Dijkstry. Jeżeli graf posiada ujemne wagi to musimy skorzystać z algorytmu Bellmana-Forda licząc się z tym, że jest to algorytm wolniejszy. Ponadto, algorytm Bellmana-Forda możemy wykorzystywać do wykrywania cykli ujemnych. W przypadku problemu wyznaczania najkrótszych ścieżek między wszystkimi parami wierzchołków grafu ważonego

najlepszym wyborem będzie algorytm Floyda-Warshalla. Jedynie w przypadku grafów odpowiednio rzadkich warto rozważyć użycie algorytmu Johnsona.

Zakończenie

W niniejszej pracy zostały przedstawione podstawowe pojęcia teorii grafów oraz najważniejsze algorytmy wyznaczające najkrótsze ścieżki w grafach ważonych, które są powszechnie wykorzystywane w praktyce. Można zauważyć podział pracy na część teoretyczną oraz praktyczną.

Cześć teoretyczna pracy, dokładniej są to rozdziały 1 i 2, skupia się na podstawowych informacjach z teorii grafów oraz na dokładnym opisie wspomnianych algorytmów bazujących na wybranych informacjach z literatury [1–3, 5].

Cześć praktyczna, rozdziały 3 i 4, skupia się na implementacji algorytmów w języku Python oraz analizie i porównaniu tych implementacji w celu niejako sprawdzania ich poprawności - badana była złożoność czasowa zaimplementowanych algorytmów.

Głównym osiągnięciem autora pracy jest przedstawienie w zwięzły sposób opisu teoretycznego omawianych algorytmów, a przede wszystkim samodzielna ich implementacja w języku Python. Autor pracy przeprowadził także w rozdziale 4 analizę złożoności czasowej swoich implantacji, która to zgadza się z teoretycznymi założeniami.

Na końcu pracy umieszczono dwa dodatki. Pierwszy z nich zawiera krótkie informacje na temat złożoności obliczeniowej i notacji asymptotycznej. Drugi zawiera wybrane kody źródłowe w języku Python wykorzystane w niniejszej pracy. Wszystkie kody źródłowe, które autor napisał na potrzeby niniejszej pracy zostały załączone na nośniku CD. Można je wykorzystać, by porównać wyniki na różnych maszynach obliczeniowych

Podsumowując, treść zawarta w niniejszej pracy pozwala w prosty i czytelny sposób zaznajomić się z tematyką grafów oraz algorytmów wyznaczających najkrótsze ścieżki w grafach ważonych i może zostać użyta jako podstawowe źródło wiedzy na ich temat.

Dodatki

A Elementy złożoności obliczeniowej i notacja asymptotyczna

W niniejszym dodatku podamy pewne definicje istotne z punktu widzenia analizy algorytmów. Te informacje jak i ich uzupełnienie można znaleźć między innymi w pracach [2, 4, 5].

Definicja A.1. *Złożonością obliczeniową* algorytmów nazywamy miarę, która określa ilość zasobów niezbędnych do wykonania danego algorytmu. Mówiąc o zasobach zwykle mamy na myśli czas i mówimy wtedy o *złożoności czasowej* lub zajętość pamięci i w takim przypadku mamy do czynienia ze *złożonością pamięciową*.

Większość zagadnień obliczeniowych ma wspólną cechę – im większe są rozmiary danych wejściowych, tym więcej zasobów (czasu, pamięci) jest koniecznych do wykonania danych obliczeń. Możemy zatem śmiało powiedzieć, że złożoność algorytmu jest funkcją rozmiaru danych wejściowych.

Przeważnie podczas analizy algorytmów to jego czas działania odgrywa większe znaczenie. Czas działania algorytmu zależy nie tylko od niego samego ale też od szybkości działania komputera, rodzaju danych i ich wielkości. Ponadto, program napisany w języku maszynowym będzie działał szybciej niż ten napisany w języku interpretowanym przez system operacyjny. Dlatego, w celu uzależnienia się od konkretnego komputera zamiast zamiast czasu określa się ilości wykonywanych operacji elementarnych (podstawowych) o których zakładamy, że wykonują się w jednakowym czasie. Inaczej mówiąc miarą złożoności czasowej jest liczba operacji podstawowych w zależności od rozmiaru wejścia. Za operacje elementarne przyjmuje się np.: operacje arytmetyczne, logiczne i relacyjne; postawienie; indeksowanie lub odwołanie do pola struktury; przekazywanie wartości do funkcji; inicjalizacja wywołania funkcji.

Definicja A.2. *Złożonością pesymistyczną* nazywamy złożoność algorytmu, która jest

wyznaczona w najgorszym przypadku. Dokładniej, jeżeli D jest zbiorem wszystkich możliwych danych wejściowych, d oznacza jeden z elementów zbioru D, a f jest funkcją, która dla danego zestawu danych wejściowych d zwraca liczbę operacji podstawowych to złożoność pesymistyczną możemy definiować następująco:

$$\sup\{f(d)\colon\ d \in D\}.$$

Definicja A.3. Niech D będzie zbiorem wszystkich możliwych danych wejściowych algorytmu, d oznacza pojedynczy zestaw danych wejściowych ze zbioru D, f jest funkcją, która dla danego d zwraca liczbę operacji podstawowych. *Złożoność oczekiwana* (*średnia*) określa złożoność średnią, czyli wartość oczekiwaną funkcji f traktowanej jako zmienną losową. Jeżeli wszystkie dane wejściowe są jednakowo prawdopodobne (z prawdopodobieństwem różnym od zera), wtedy złożoność oczekiwaną możemy wyznaczyć według wzoru:

$$\frac{\sum_{d \in D} f(d)}{|D|}$$

Wyznaczenie dokładnej postaci funkcji danych wejściowych opisującej złożoność obliczeniową algorytmu przeważnie nie jest zadaniem prostym, ale w praktyce wystarczą jej oszacowania. Najbardziej powszechną miarą do zapisu szacowania złożoności jet *O-notacja* dotycząca kresu górnego.

Definicja A.4. Niech będą dane dwie funkcje f i g o wartościach nieujemnych: $f, g\colon \mathbb{N} \to \mathbb{R}^+$. Mówimy, że funkcja $f = f(n)$ jest co najwyżej rzędu $g = g(n)$, czyli ma złożoność $O(g(n))$, jeżeli:

$$\exists_{c>0}\exists_{n_0>0}\forall_{n>n_0} f(n) \leq c \cdot g(n).$$

Inaczej:

$$f(n) = O(g(n)) \equiv \limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty.$$

Możemy powiedzieć nieformalnie, że funkcja g jest kresem górnym dla funkcji f. Ponieważ dla danej funkcji f może istnieć niekończenie wiele takich funkcji to w praktyce w celu uniknięcia niejednoznaczności wybiera się najmniejszą funkcję g o tej własności.

Innymi miarami do do szacowania złożoności są Ω-*notacja* i Θ-*notacja*, jednak są one rzadziej stosowane.

Definicja A.5. Niech będą dane dwie funkcje f i g o wartościach nieujemnych: $f, g\colon \mathbb{N} \to \mathbb{R}^+$. Mówimy, że funkcja $f = f(n)$ jest co najmniej rzędu $g = g(n)$,

czyli ma złożoność $\Omega(g(n))$, jeżeli:

$$\exists_{c>0}\exists_{n_0>0}\forall_{n>n_0} f(n) \geq c \cdot g(n).$$

Inaczej:

$$f(n) = \Omega(g(n)) \equiv \liminf_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0.$$

Możemy powiedzieć nieformalnie, że funkcja g jest kresem dolnym dla funkcji f. Ponieważ dla danej funkcji f może istnieć niekończenie wiele takich funkcji to w praktyce w celu uniknięcia niejednoznaczności wybiera się największą funkcję g o tej własności.

Definicja A.6. Notacja Θ jest połączeniem notacji O oraz Ω. Niech będą dane dwie funkcje f i g o wartościach nieujemnych: $f, g\colon \mathbb{N} \to \mathbb{R}^+$. Mówimy, że funkcja $f = f(n)$ jest co najmniej rzędu $g = g(n)$, czyli ma złożoność $\Theta(g(n))$, jeżeli $f(n) = \Omega(g(n))$ oraz $f(n) = O(g(n))$

Można by pomyśleć, że notacja Θ jest najlepsza i najbardziej dokładna ale jest to mylne przekonanie i w praktyce notacja ta jest rzadko używana.

B Wybrane kody źródłowe

```
1  def print_dijkstra(g: Graph, s: str):
2      s: Vertex = g.vertices[s]
3      for v in s.distance:
4          if s.path[v]:
5              print('Droga z {} do {} (odległość: {}): '
6                    .format(s.name, v, s.distance[v]),
7                            end='')
8              p: str = v
9              que = deque()
10             while s.path[p]:
11                 que.append(p)
12                 p = s.path[p]
13             que.append(s.name)
14             while que:
15                 vertex = que.pop()
16                 if vertex == v:
```

```
                    print('{}\n'.format(v), end='')
                    continue
                print('{} -> '.format(vertex), end='')
```

Rys. 1: Kod źródłowy funkcji pomocniczej wypisującej wyniki algorytmu Dijkstry.

```
def print_bellman_ford(g: Graph, s: str, result):
    if result == -1:
        print("Ujemny cykl")
        return
    elif result == -2:
        print("Wierzchołek '{}' nie posiada krawędzi \
              wychodzących (może być wierzchołkiem \
              izolowanym)".format(s))
        return
    s: Vertex = g.vertices[s]
    for v in s.distance:
        if s.path[v]:
            print('Droga z {} do {} (odległość: {}): '
                  .format(s.name, v, s.distance[v]),
                          end='')
            p: str = v
            que = deque()
            while s.path[p]:
                que.append(p)
                p = s.path[p]
            que.append(s.name)
            while que:
                vertex = que.pop()
                if vertex == v:
                    print('{}\n'.format(v), end='')
                    continue
                print('{} -> '.format(vertex), end='')
```

Rys. 2: Kod źródłowy funkcji pomocniczej wypisującej wyniki algorytmu Bellmana-Forda.

```
def print_floyd_warshall(g, result):
```

```
    if not result:
        return
    distance, path = result
    for u in g.vertices:
        for v in g.vertices:
            if path[g.indexes[u]][g.indexes[v]]:
                print('Droga z {} do {} (odległość: \
                      {}): '.format(u, v, distance[
                          g.indexes[u]][g.indexes[v]]),
                          end='')
                p: str = u
                while path[g.indexes[p]][g.indexes[v]]:
                    print('{} -> '.format(p), end='')
                    p = path[g.indexes[p]][g.indexes[v]]
                print('{}\n'.format(v), end='')
```

Rys. 3: Kod źródłowy funkcji pomocniczej wypisującej wyniki algorytmu Floyda-Warshalla.

```
def print_johnson(g, result):
    if not result:
        return
    distance, path = result
    for u in g.vertices:
        for v in g.vertices:
            if path[u][v]:
                print('Droga z {} do {} (odległość: \
                    {}): '.format(u, v,
                                  distance[u][v]), end='')
                p: str = v
                que = deque()
                while path[u][p]:
                    que.append(p)
                    p = path[u][p]
                que.append(u)
                while que:
                   vertex = que.pop()
                   if vertex == v:
```

```
                            print('{}\n'.format(v), end='')
                            continue
                        print('{}->'.format(vertex), end='')
```

Rys. 4: Kod źródłowy funkcji pomocniczej wypisującej wyniki algorytmu Johnsona.

```
import operator
import random
from described.graph import Edge, Graph, Vertex


def generate(vertices_number, edges_number):
    vertices = []
    for i in range(vertices_number):
        name = ""
        first_letter = int(i / 26)
        second_letter = i % 26
        if first_letter != 0:
            first_letter = chr(97 + first_letter - 1)
            name += first_letter
        second_letter = chr(97 + second_letter)
        name += second_letter
        vertices.append(Vertex(name))
    edges = []
    graph_edges = []
    available_edges = []
    for v_start in vertices:
        for v_end in vertices:
            if v_start == v_end:
                continue
            available_edges.append((v_start, v_end))
    for i in range(edges_number):
        random_edge = random.choice(available_edges)
        edges.append(random_edge)
        available_edges.remove(random_edge)
    for edge in edges:
        start_vertex, end_vertex = edge
```

```
32          weight = random.randint(0, 10)
33          graph_edges.append(Edge(start_vertex.name,
34                                  end_vertex.name, weight))
35      graph_edges.sort(key=operator.attrgetter('start',
36                                               'end'))
37      graph = Graph(vertices={vertex.name: vertex
38                              for vertex in vertices},
39                    edges=graph_edges)
40      return graph
```

Rys. 5: Kod źródłowy funkcji generującej losowe grafy.

```
1  import pickle
2  from described.graph import Graph
3  from generator.graph_generator import generate
4
5
6  with open("const_380_edges.graph", "wb") as graph_file:
7      graphs_list = []
8      edges_number = 380
9      for i in range(20, 70):
10         g: Graph = generate(i, edges_number)
11         graphs_list.append(g)
12     pickle.dump(graphs_list, graph_file)
```

Rys. 6: Kod źródłowy przykładowego zapisu wygenerowanych grafów do pliku.

```
1  import pickle
2  from timeit import Timer
3
4  import matplotlib.pyplot as plt
5  import numpy as np
6  from scipy.optimize import curve_fit
7
8
9  def make_plot(file_name: str, function_name: str,
10                repeat_number: int, color: str,
11                approximation_function=None,
12                count_edges: bool = False,
```

```
13              count_vertices: bool = False,
14              density: float = None,
15              use_polyfit: bool = False,
16              dim: int = 1):
17      with open(file_name, "rb") as graph_file:
18          graphs_list = pickle.load(graph_file)
19
20      data = {}
21      for graph in graphs_list:
22          function = getattr(graph, function_name)
23          t = Timer(lambda: function())
24          function_time = t.repeat(repeat_number, 1)
25          function_time = min(function_time)
26
27          if count_edges is False and count_vertices \
28                  is False:
29              count_vertices = True
30          if count_edges is True and count_vertices \
31                  is True:
32              print("Błąd, nie można liczyć krawędzi \
33              i wierzchołków jednocześnie")
34              return
35
36          if count_vertices:
37              data[len(graph.vertices)] = function_time
38          elif count_edges:
39              data[len(graph.edges)] = function_time
40
41      x_array = [x for x in data]
42      y_array = []
43      for x in x_array:
44          y_array.append(data[x])
45
46      plt.plot(x_array, y_array, '.')
47
48      if use_polyfit:
```

```
        z = np.polyfit(x_array, y_array, dim)
        p = np.poly1d(z)
        plt.plot(x_array, p(x_array), color, label="{}"
                 .format(str(function_name)))
    else:
        popt, pcov = curve_fit(approximation_function,
                               x_array, y_array)
        if density is None:
            plt.plot(x_array, approximation_function(
                x_array, *popt), color, label="{}"
                     .format(str(function_name)))
        else:
            plt.plot(x_array, approximation_function(
               x_array, *popt), color,
               label="gęstość {}".format(str(density)))
```

Rys. 7: Kod źródłowy funkcji tworzącej pojedynczy wykres dla jednej funkcji, na głównym wykresie (rysunku) może być wiele takich pojedynczych wykresów.

```
import matplotlib.pyplot as plt
from described.graph import Graph, Edge, Vertex
from plots.make_plot import make_plot

make_plot("const_380_edges.graph",
          "dijkstra_naive", 100, "r",
          use_polyfit=True, dim=3)
make_plot("const_380_edges.graph",
          "dijkstra", 100, "g",
          use_polyfit=True, dim=3)

plt.xlabel("liczba wierzchołków")
plt.ylabel("czas (s)")
plt.legend()
plt.grid()
plt.savefig(fname="Wykres_Dijkstra_naiwny_kolejka.pdf",
            dpi=300, quality=100, format="pdf")
```

```
18 plt.show()
```

Rys. 8: Kod źródłowy pokazujący użycie funkcji `make_plot`.

Literatura

[1] T.H. Cormen, *Algorytmy bez tajemnic*, Wydawnictwo Helion, Gliwice, 2013.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C.S. Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa, 2012.

[3] W. Kordecki, A. Łyczkowska-Hańćkowiak, *Matematyka dyskretna dla informatyków*, Wydawnictwo Helion, Gliwice, 2018.

[4] P. Kotowski, *Algorytmy + Struktury danych = Abstrakcyjne typy danych*, Wydawnictwo BTC, Warszawa 2006;

[5] K. Pieńkowski, J. Wojciechowski, *Grafy i sieci*, Wydawnictwo Naukowe PWN, Warszawa, 2013.

[6] https://docs.python.org/3/

[7] https://pl.wikipedia.org/wiki/Zagadnienie_mostów_królewieckich