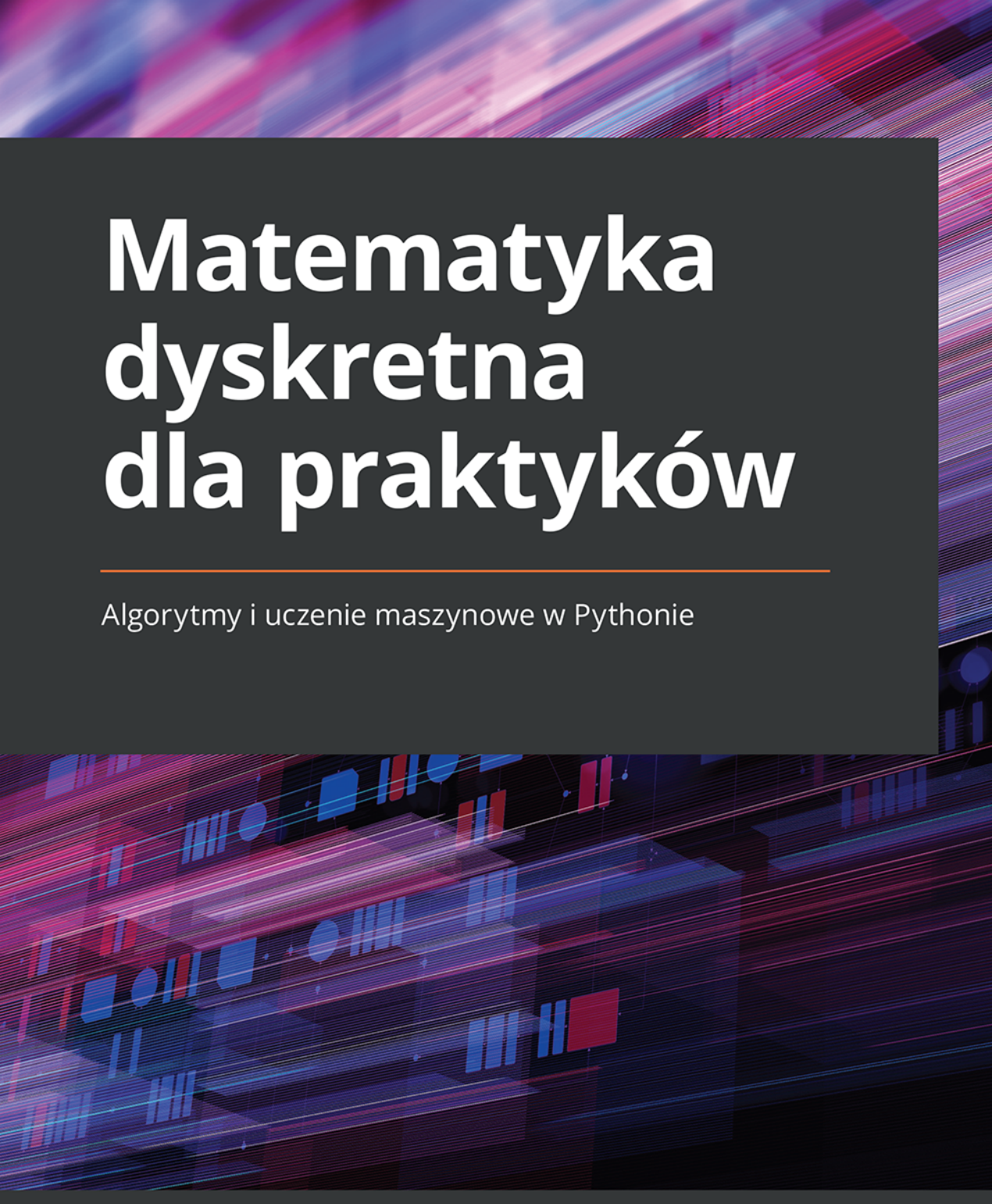


Matematyka dyskretna dla praktyków

Algorytmy i uczenie maszynowe w Pythonie

Ryan T. White
Archana Tikayat Ray

Helion

Tytuł oryginału: Practical Discrete Mathematics: Discover math principles that fuel algorithms for computer science and machine learning with Python

Tłumaczenie: Filip Kamiński

ISBN: 978-83-283-8396-8

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/madypr.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/madypr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 9

Przeszukiwanie struktur danych i znajdowanie najkrótszych ścieżek

W tym rozdziale omówimy techniki przeszukiwania grafów, drzew i sieci oraz praktyczne zastosowania przeszukiwania grafów. Przedstawimy i przeanalizujemy dwa popularne algorytmy: **algorytm przeszukiwania w głąb** (ang. *depth-first search*, DFS) oraz algorytm Dijkstry znajdowania najkrótszych ścieżek między wierzchołkami w sieciach. Oba algorytmy zostaną zaprezentowane na małych strukturach, które pozwolą Ci intuicyjnie zrozumieć ich działanie. Następnie zaprezentujemy implementacje w Pythonie, które można wykorzystać do rozwiązania rzeczywistych problemów.

W tym rozdziale omówimy następujące zagadnienia:

- Przeszukiwanie struktur grafowych i drzew.
- Algorytm przeszukiwania w głąb.
- Problem najkrótszej ścieżki i jego warianty.
- Znajdowanie najkrótszych ścieżek metodą siłową.
- Algorytm Dijkstry znajdowania najkrótszych ścieżek.
- Implementacja algorytmu Dijkstry w Pythonie.

Po przeczytaniu tego rozdziału będziesz umiał wyjaśnić cel przeszukiwania, będziesz wiedział, jak zaimplementować metodę przeszukiwania w głąb, zrozumiesz problemy związane z najkrótszą ścieżką i ich warianty oraz będziesz wiedział, jak zaimplementować algorytm Dijkstry do znajdowania najkrótszych ścieżek.

Przeszukiwanie struktur grafowych i drzew

W poprzednim rozdziale poznałeś grafy i drzewa. Pamiętaj, że w tym rozdziale, ilekroć będziemy odwoływać się do grafów, będziemy mieli na myśli też drzewa, które również są grafami, ale takimi bez cykli. Tematem tego rozdziału jest przeszukiwanie grafów. Przeszukiwanie to po prostu przechodzenie wzdłuż krawędzi grafu w celu znalezienia ścieżek do wierzchołków docelowych. Brzmi to bardzo prosto, ale ponieważ wiele grafów wykorzystywanych w praktyce jest ogromnych, chcemy to zrobić tak wydajnie, jak to tylko możliwe.

Istnieje wiele powodów, dla których możemy chcieć przeszukać graf. Załóżmy na przykład, że chcesz wysłać wiadomość przez sieć do pięciu znajomych mieszkających w pięciu różnych miastach. Z pewnością nie istnieje bezpośrednie połączenie między Twoim urządzeniem a urządzeniami Twoich znajomych. Dlatego zanim wiadomość dotrze do znajomych, musi przejść przez wiele urządzeń sieciowych, które mogą być reprezentowane przez wierzchołki w grafie. Ponieważ urządzenia sieciowe zmieniają strukturę połączeń w czasie, przechowywanie stałego grafu reprezentującego sieć nie jest możliwe. Oznacza to, że ścieżki muszą być wyznaczone w momencie wysłania wiadomości. Jest to przykład, który możemy rozwiązać za pomocą przeszukiwania grafów.

Osobną kwestią jest ustalenie, którą ścieżką należy przesłać wiadomość. Możesz chcieć wybrać ścieżki, które zabierają najmniej czasu na dostarczenie wiadomości, lub ścieżki, które przechodzą przez nieobciążone ruchem urządzenia sieciowe. Znajdowanie najkrótszej ścieżki zostanie omówione w dalszej części tego rozdziału. Na razie wystarczy, abyś wiedział, że przeszukiwanie grafów jest często ważnym elementem rozwiązywania takich problemów.

Algorytmy przeszukiwania grafów nie robią zbyt wiele same z siebie. Wykorzystuje się je zazwyczaj w roli elementów składowych bardziej złożonych programów, które rozwiązują wiele problemów, takich jak znajdowanie najkrótszych ścieżek i minimalnych drzew rozpinających, wykrywanie składowych spójnych, analizowanie przepływu w sieci, dopasowywanie wierzchołków pomiędzy grupami i rozwiązywanie dużych problemów planowania, w których istnieją złożone relacje pomiędzy zadaniami.

W następnym podrozdziale poznasz algorytm przeszukiwania w głąb — jeden z najpopularniejszych algorytmów przeszukiwania grafów.

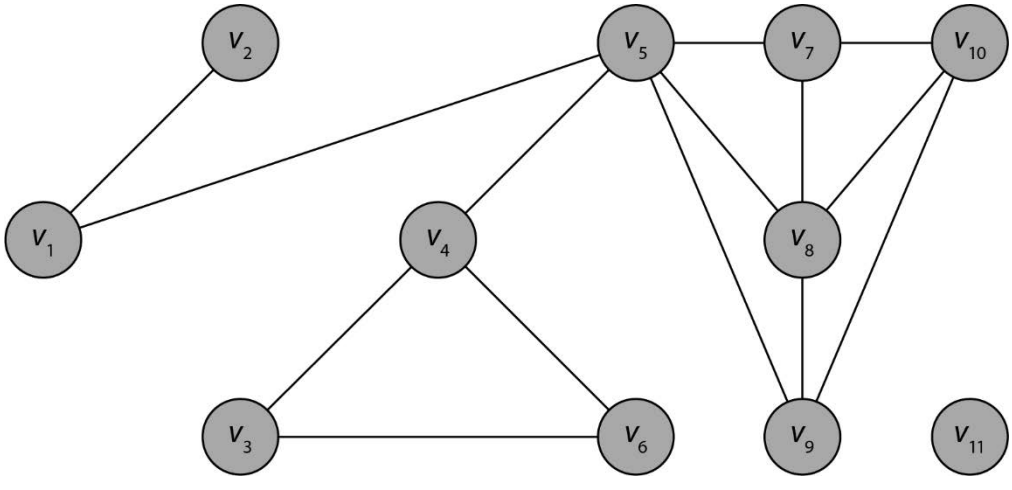
Algorytm przeszukiwania w głąb (DFS)

Przeszukiwanie grafu to w skrócie przejście przez jego elementy w usystematyzowany sposób. W tym podrozdziale poznasz algorytm pozwalający przeprowadzić taki proces. Chociaż przejście przez strukturę grafu może być istotne samo w sobie, jak już wspomnieliśmy, często jest to podproblem, który rozwiązuje się w jakimś bardziej złożonym zagadnieniu grafowym. Algorytm przeszukiwania w głąb (ang. *depth-first search*, DFS) jest prawdopodobnie najczęściej stosowanym algorytmem przeszukiwania grafów. Jest to wydajna metoda, często wykorzystywana w bardziej złożonych rozwiązaniach.

Algorytm przeszukiwania w głąb rozpoczyna pracę w wierzchołku startowym, a następnie przechodzi przez pierwszą związaną z nim krawędź do kolejnego wierzchołka i powtarza ten proces aż do momentu, gdy nie będzie krawędzi prowadzących do nieodwiedzonych jeszcze wierzchołków (to znaczy, dopóki nie uda mu się wejść najgłębiej, jak to możliwe). Po dotarciu do końcowego wierzchołka algorytm cofa się do ostatniego wierzchołka, który ma jeszcze nieodwiedzonych sąsiadów, i przechodzi od tego wierzchołka przez kolejne nieodwiedzone wierzchołki, aż do kolejnego ślepego zaułka. Następnie algorytm ponownie się cofa i odwiedza kolejne nieodwiedzone jeszcze wierzchołki aż do momentu, gdy wszystkie wierzchołki połączone z wierzchołkiem startowym zostaną odwiedzone.

Abyś to dobrze zrozumiał, prześledzimy działanie tej metody na małym grafie z rysunku 9.1. Rozpoczniemy od wierzchołka v_1 i przeszukamy graf za pomocą algorytmu przeszukiwana w głąb.

Zauważ, że nie określiliśmy, jak należy dokonać wyboru ścieżek. Ustalmy więc, że w przypadku więcej niż jednej opcji wybierzemy wierzchołek o najniższym numerze. Wierzchołki i krawędzie w obrębie aktualnie przetwarzanej ścieżki zaznaczymy na pomarańczowo, a poprzednio odwiedzone wierzchołki i wcześniej przebyte krawędzie na zielono (rysunek 9.1).

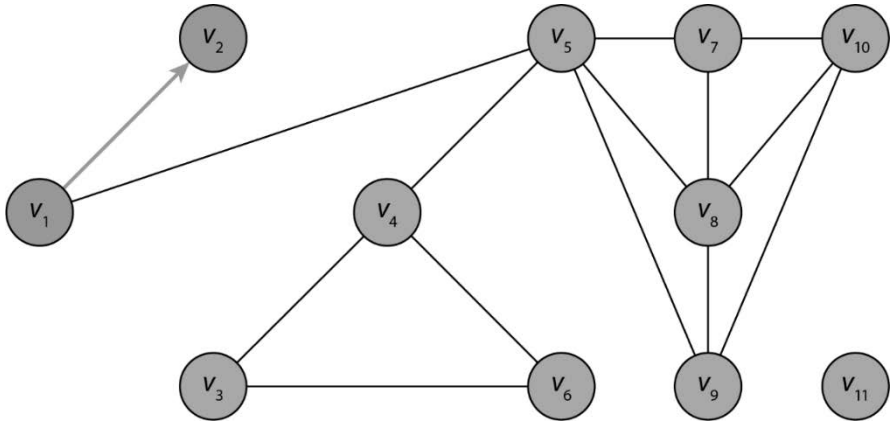


Rysunek 9.1. Graf

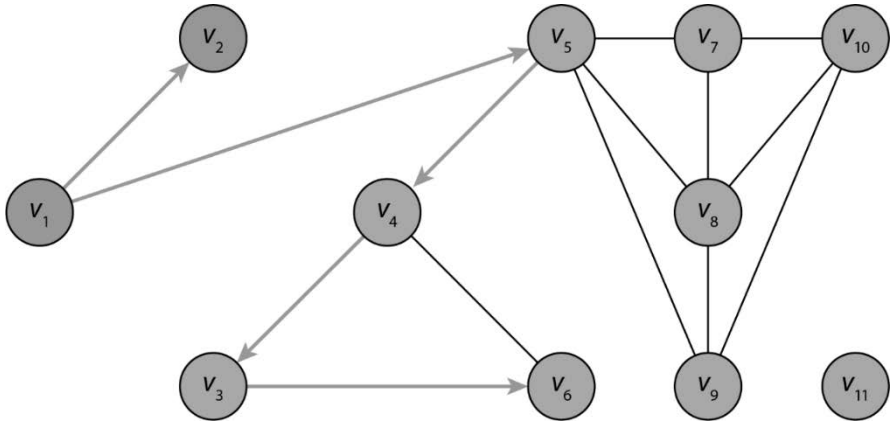
Krok 1. Pierwszym krokiem będzie przejście do wierzchołka v_2. Wierzchołek ten nie sąsiaduje z żadnymi wierzchołkami, których jeszcze nie odwiedziliśmy, więc algorytm zatrzyma się (rysunek 9.2).

Krok 2. Cofamy się do wierzchołka v_1. Następnie będziemy podążali kolejnymi ścieżkami, aż ponownie dotrzemy do ślepego zaułka. Przechodzimy więc z v_1 do v_5, potem do v_4 i dalej do v_3 oraz v_6, który nie ma już żadnych nieodwiedzonych sąsiadów. Ponownie następuje więc zatrzymanie (rysunek 9.3).

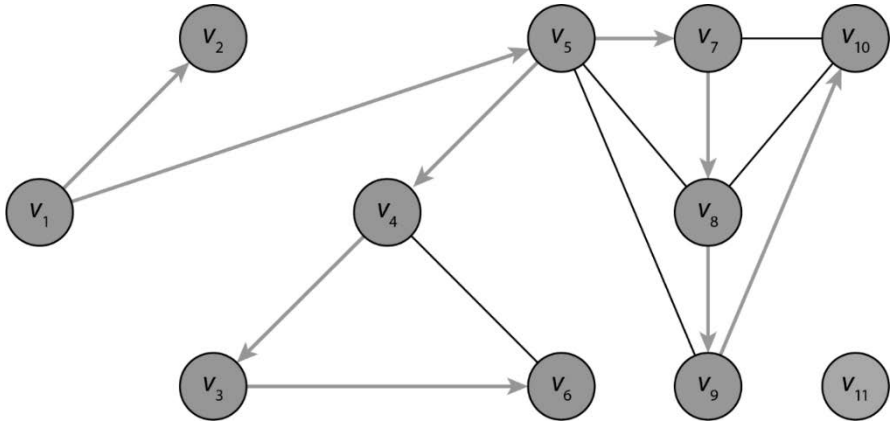
Krok 3. Cofamy się do wierzchołka v_5 będącego ostatnim wierzchołkiem na pomarańczowej ścieżce, który ma jeszcze nieodwiedzonych sąsiadów. Przechodzimy kolejno przez v_7, v_8, v_9 i v_{10}, gdzie następuje kolejne zatrzymanie (rysunek 9.4).



Rysunek 9.2. Pierwszy krok w algorytmie przeszukiwania w głąb



Rysunek 9.3. Drugi krok algorytmu przeszukiwania w głąb



Rysunek 9.4. Trzeci krok algorytmu przeszukiwania w głąb

W tym momencie wszystkie wierzchołki połączone z v_1 zostały pokolorowane, co oznacza, że wszystkie wierzchołki zostały odwiedzone, a więc przeszukiwanie grafu zostało zakończone.

Lista wierzchołków odwiedzonych przez algorytm jest następująca:

$$v_1, v_2, v_5, v_4, v_3, v_6, v_7, v_8, v_9, v_{10}$$

Zauważ, że nie odwiedziliśmy wierzchołka v_{11}, który nie jest połączony z wierzchołkiem źródłowym. Algorytm przeszukiwania w głąb nie pominie żadnego wierzchołka, który jest elementem tej samej spójnej składowej co wierzchołek startowy. Aby odwiedzić wszystkie wierzchołki grafu z wieloma składowymi spójnymi, konieczne jest uruchomienie algorytmu w każdej składowej osobno.

Zajmiemy się teraz implementacją algorytmu przeszukiwana w głąb w Pythonie.

Implementacja algorytmu przeszukiwania w głąb w Pythonie

Oczywiście w przypadku dużych problemów spotykanych w praktyce nie jesteśmy w stanie przeprowadzić ręcznego przeszukiwania w głąb! Stwórzmy więc jego implementację w Pythonie.

Zdefiniujemy funkcję o nazwie `DFS`, która przyjmie macierz sąsiedztwa i zwróci wszystkie wierzchołki połączone ścieżką z wierzchołkiem startowym.

Implementacja będzie się składała z kilku fragmentów, które postaramy się na bieżąco wyjaśniać. Zaczynamy od dokumentacji, która opisuje, co robi funkcja oraz jakie ma parametry wejściowe i wyjściowe:

```
# Algorytm przeszukiwania w głąb (Depth First Search)
#
# WEJŚCIE
# A — kwadratowa, symetryczna i binarna macierz sąsiedztwa
# source — numer wierzchołka startowego (w grafie)
#
# WYJŚCIE
# vertexList — uporządkowana lista odwiedzonych wierzchołków
```

Następnie zdefiniujemy funkcję, która przyjmuje macierz sąsiedztwa i numer wierzchołka źródłowego. W pierwszych wierszach odejmujemy od numeru wierzchołka startowego 1 (w Pythonie tablice są indeksowane od zera), ustalamy liczbę wierzchołków w grafie i inicjujemy kilka struktur danych, w tym tablicę wartości binarnych do przechowywania odwiedzonych wierzchołków, stos, który ma być użyty w algorytmie, oraz listę wierzchołków, które odwiedził algorytm:

```
def DFS(A, source):
    # Zmniejszamy wartość source o 1, aby uniknąć błędów
    source -= 1

    # Znajdowanie liczby wierzchołków
    n = A.shape[0]
```

```
# Stworzenie zbioru nieodwiedzonych wierzchołków. Na początek zbiór ten zawiera wszystkie wierzchołki
unvisited = [1] * n

# Inicjalizacja kolejki. Na początek kolejka zawiera tylko wierzchołek startowy
stack = [source]

# Inicjalizacja listy wierzchołków
vertexList = []
```

Następnie funkcja zdejmuje ostatni wierzchołek ze stosu i dodaje go (jeśli nie został jeszcze odwiedzony) do końca kolejki wraz z listą jego jeszcze nieodwiedzonych sąsiadów. Operacje te są powtarzane aż do momentu, gdy stos będzie pusty. Na zakończenie funkcja zwraca listę odwiedzonych wierzchołków:

```
# Dopóki stos (zmienna stack) nie jest pusty
while stack:
    # Usunięcie odwiedzonego wierzchołka ze stosu i zapisanie go do zmiennej v
    v = stack.pop()

    # Jeżeli v nie był wcześniej odwiedzony, oznaczamy go jako odwiedzony i dodajemy do listy
    # odwiedzonych wierzchołków
    if unvisited[v]:
        # Zapis i wyświetlenie numeru właśnie odwiedzonego wierzchołka
        vertexList.append(v)

        # Oznaczenie wierzchołka v jako odwiedzonego
        unvisited[v] = 0

    # Przejście przez wierzchołki
    for u in range(n - 1, 0, -1):
        # Dodanie każdego nieodwiedzonego wierzchołka do stosu
        if A[v,u] == 1 and unvisited[u] == 1:
            stack.append(u)

# Zwrócenie listy odwiedzonych wierzchołków
return vertexList
```

Kod jest już kompletny. Aby upewnić się, że działa on zgodnie z przeznaczeniem, przetestujemy go na przykładzie, który rozwiązaliśmy wcześniej ręcznie. Najpierw musimy zapisać naszą macierz sąsiedztwa:

```
# Macierz sąsiedztwa grafu z rysunku 9.1
A = numpy.array([[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                 [0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
                 [0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0],
                 [1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0],
                 [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                 [0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0],
                 [0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0],
                 [0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0],
                 [0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0],
                 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

Następnie należy wywołać funkcję `DFS` z parametrem `source=1`, czyli z numerem wierzchołka, od którego rozpoczęliśmy ręczne obliczenia. Ponieważ numerowaliśmy nasze wierzchołki od 1, a Python indeksuje tablice od 0, do elementów listy otrzymanej z wywołania musimy dodać 1:

```
# Uruchomienie algorytmu na macierzy sąsiedztwa A z parametrem source równym 1
vertexList = DFS(A,1)

# Dodanie 1 do numerów wierzchołków
[x + 1 for x in vertexList]
```

Po uruchomieniu kodu otrzymasz poniższy wynik:

```
[1, 2, 5, 4, 3, 6, 7, 8, 9, 10]
```

Zauważ, że podczas ręcznej implementacji algorytmu znaleźliśmy dokładnie tę samą listę z dokładnie tą samą kolejnością elementów. Najwyraźniej nasz kod robi dokładnie to samo, co zrobiliśmy ręcznie. Jedyna różnica jest taka, że wynik otrzymujemy niemal natychmiast. Nasza implementacja algorytmu przeszukiwania w głąb okazała się więc wielkim sukcesem!

W tym podrozdziale dowiedziałeś się, czym jest algorytm przeszukiwania w głąb oraz jakie są wybrane zastosowania. Zaprezentowaliśmy też jego działanie na przykładzie i zaimplementowaliśmy go w Pythonie. Na koniec pokazaliśmy, że otrzymane wyniki są zgodne z tymi z ręcznych obliczeń.

Pozostała część rozdziału koncentruje się na bardzo praktycznym problemie, jakim jest znalezienie najkrótszej ścieżki między dwoma wierzchołkami w sieci lub grafie ważonym.

Problem najkrótszej ścieżki i jego warianty

W tym podrozdziale skupimy się na innym problemie związanym z grafami, jakim jest znajdowanie najkrótszych ścieżek między wierzchołkami w sieci. Problem ten jest istotny w zagadnieniach związanych z routingiem, takich jak znalezienie najkrótszej trasy do celu lub najszybszego sposobu dostarczenia wiadomości przez sieć komputerową. Zagadnienie najkrótszej ścieżki zostało również wykorzystane w problemie dotyczącym minimalizacji spalania paliwa przy jednoczesnym precyzyjnym ustawieniu floty małych satelitów badawczych wykorzystywanych do przesyłania obrazów odległych gwiazd.

Problem znajdowania najkrótszej ścieżki w grafach bez ważonych krawędzi został omówiony w poprzednim rozdziale. Powróćmy do niego na chwilę, zanim przejdziemy do bardziej ogólnego problemu, który dotyczy sieci (grafów ważonych). W rozdziale 8. „Przechowywanie i wyodrębnianie cech z grafów, drzew i sieci" przedstawiliśmy sposób znalezienia najkrótszej (składającej się z najmniejszej liczby krawędzi) ścieżki między węzłami v_i i v_j w grafie i w grafie skierowanym. Była to po prostu najmniejsza liczba n, taka, że n-ta potęga macierzy sąsiedztwa zawiera dodatnią wartość w i-tym wierszu i j-tej kolumnie. Jest to dość oczywiste, ponieważ wartość z tej pozycji jest równa liczbie ścieżek z vi do v_j o długości n.

Ponieważ złożoność obliczeniowa mnożenia macierzy jest niewielka (poniżej $O(n^3)$), zależność ta jest niezwykle przydatna, pozwala bowiem w efektywny sposób znaleźć najkrótszą odległość między wierzchołkami w grafach i grafach skierowanych.

Najkrótsze ścieżki w sieciach

Problemem o znacznie szerszym zastosowaniu jest zagadnienie znajdowania najkrótszej ścieżki między węzłami w sieci, w której wagi krawędzi reprezentują odległość między węzłami. Jest to bardzo ważny problem, którego rozwiązanie pozwala usługom takim jak MapQuest, Mapy Google i Waze znajdować najkrótszą trasę między dwoma miastami. Zapewne wielu czytelników codziennie korzysta z tych rozwiązań! Równoważnym problemem jest znalezienie najkrótszej trasy na dostarczenie energii elektrycznej z elektrowni przez stacje przesyłowe aż do naszych gniazdek. Biorąc pod uwagę, że na dłuższych trasach dochodzi do większych strat energii, inteligentna sieć energetyczna powinna utrzymywać odległości na niskim poziomie, tak aby efektywnie wykorzystywać energię generowaną przez elektrownie.

Inne zastosowania najkrótszych ścieżek

Poza powyższymi zastosowaniami wagi można również interpretować jako coś innego niż odległości. Na przykład podczas korzystania z Map Google bardziej niż przebytą odległością możesz być zainteresowany czasem potrzebnym na dotarcie do jakiegoś punktu. Trasa piesza może być najkrótsza, ale jeśli odległość jest mierzona w dziesiątkach lub setkach kilometrów, informacja o najkrótszej trasie może być zupełnie bezużyteczna! Zamiast tego krawędziom sieci możemy przypisać wagi odpowiadające czasowi podróży między węzłami. Powiązanym zagadnieniem, wykorzystywanym w rozwiązaniach takich jak Waze, jest wykorzystanie danych o ruchu drogowym w czasie rzeczywistym do zapewnienia lepszego oszacowania czasu potrzebnego na przejazd między dwiema lokalizacjami. Ponownie odległość nie jest tu najważniejszym czynnikiem, bardziej interesuje nas najszybsza trasa (rysunek 9.5).

Rysunek 9.5. Na tej mapie widzimy dwie trasy. Pierwsza ma długość 503 mil, a druga 511 mil

Zwróć uwagę, że na rysunku 9.5 Mapy Google rekomendują dłuższą trasę. Dzieje się tak, ponieważ ich celem jest znalezienie najszybszej trasy, która bywa nieco dłuższa. Istnieje wiele powodów, dla których dłuższą trasę możemy pokonać szybciej: na krótszej trasie może być większy ruch, niższe ograniczenia prędkości lub więcej świateł.

Traktowanie wag jako czasów wiąże się z zupełnie nowym obszarem praktycznych zastosowań, w których interesują nas ścieżki o najkrótszym czasie. Załóżmy, że chcesz wysłać wiadomość tekstową ze swojego telefonu na komputer znajomego. W takim przypadku o wiele bardziej niż odległość, jaką musi pokonać sygnał, interesuje Cię opóźnienie w sieci lub czas dostarczenia wiadomości. Znajdowanie najkrótszych ścieżek pozwala inteligentnie sterować ruchem w sieci komputerowej lub w internecie.

Inną opcją jest użycie wag, które reprezentują koszt dodania krawędzi do ścieżki. Załóżmy, że z przejściem przez krawędź wiążą się koszty, takie jak cena paliwa lub wynagrodzenie kierowcy ciężarówki. Może interesować nas znalezienie ścieżki, której koszt jest najmniejszy, nawet jeśli odległość i czas nie są minimalne. Załóżmy, że chcielibyśmy wybudować drogę łączącą dwa miasta, która przebiega przez jakieś węzły pośrednie. Koszty budowy dla każdego odcinka pomiędzy każdą parą węzłów będą różne i zależne nie tylko od odległości między węzłami, ale także od ukształtowania terenu, konieczności przewiezienia materiałów, zatrudnienia pracowników oraz wielu innych kwestii.

Niezależnie od tego, czy szukamy najkrótszych ścieżek w kontekście odległości, czasu, kosztów, czy innych parametrów, wszystkie one sprowadzają się do tego samego problemu — znalezienia minimalnej sumy wag łączących dwa punkty w sieci. Możliwość połączenia tak wielu różnych problemów w jeden abstrakcyjny problem dotyczący sieci pokazuje moc matematyki w uogólnianiu i rozwiązywaniu wielu problemów naraz.

Definicja problemu najkrótszej ścieżki

Jak widzisz, w praktycznych zastosowaniach wagi mogą reprezentować wiele różnych wartości. Warto więc wprowadzić pewną abstrakcję od konkretnych założeń dotyczących tego, co reprezentują wagi, i sformalizować opis problemu znalezienia najkrótszych ścieżek w sieciach.

Niech $N = (V, E, W)$ będzie siecią, w której $V = \{v_1, v_2, ..., v_n\}$ to zbiór wierzchołków, E to zbiór krawędzi łączących pary wierzchołków, a W to zbiór wag krawędzi. Poszukamy najkrótszej drogi od wierzchołka v_i do wierzchołka v_j. Innymi słowy, chcemy znaleźć zbiór krawędzi łączących v_i z v_j, których suma wag będzie minimalna.

Zauważ, że między daną parą węzłów może istnieć wiele różnych ścieżek lub może nie istnieć żadna. Jeśli ścieżki istnieją, to może też istnieć więcej niż jedna ścieżka z minimalną sumą wag. W związku z tym powinniśmy pamiętać, że jest to problem, w którym rozwiązanie: może nie istnieć, może istnieć unikalne rozwiązanie lub może istnieć wiele rozwiązań.

W większości praktycznych zastosowań istotna jest jedynie możliwość nieistnienia rozwiązania, czyli sytuacja, w której v_i nie jest połączone z v_j żadną ścieżką. Do sprawdzenia, czy v_i jest połączone z v_j, możemy wykorzystać metody z poprzedniego rozdziału.

Sprawdzenie, czy istnieje rozwiązanie

Przypomnijmy, że ścieżkę z v_i do v_j z minimalną liczbą krawędzi możemy znaleźć, potęgując macierz sąsiedztwa, aż do otrzymania dodatniej wartości w i-tym wierszu i j-tej kolumnie macierzy. Oczywiście jeśli v_i nie jest połączone z v_j, to możemy nigdy nie otrzymać takiej wartości. Ponieważ wiemy, ile krawędzi ($|E|$) znajduje się w sieci, możemy sprawdzić, czy istnieje ścieżka pomiędzy interesującymi nas wierzchołkami. Ścieżka nie istnieje, jeżeli macierz A^n zawiera w i-tym wierszu i j-tej kolumnie zera dla wszystkich $n \leq |E|$. Tym samym wiemy też, że nawet gdybyśmy wykorzystali wszystkie krawędzie w sieci, to nie znaleźlibyśmy ścieżki (a więc i najkrótszej ścieżki) między wierzchołkami. W przypadku, gdy wierzchołki są połączone, istnieje jakaś ścieżka, a tym samym również ścieżka o minimalnej długości.

Przed zastosowaniem algorytmu znajdowania najkrótszej ścieżki warto najpierw sprawdzić poprzez potęgowanie macierzy sąsiedztwa, czy jakakolwiek ścieżka istnieje. Stwórzmy więc funkcję, która wykona takie sprawdzenie. Funkcja ta sprawdzi, czy wierzchołki sąsiadują ze sobą. Jeżeli nie, funkcja obliczy kolejne potęgi macierzy sąsiedztwa aż do otrzymania potwierdzenia lub przekroczenia $A^{|E|}$ bez wykrycia ścieżki (wtedy będziemy pewni, że nie istnieje żadna ścieżka z v_i do v_j). W takim przypadku będziemy wiedzieć, że najkrótsza ścieżka nie istnieje, i unikniemy kłopotów związanych z jej poszukiwaniem!

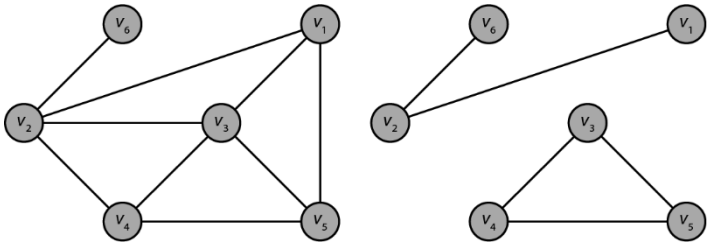
Jeżeli ścieżka istnieje, to nasza funkcja wyświetli jej długość i zwróci `True`. W przeciwnym razie funkcja zwróci `False` i wyświetli informację, że nie znaleziono ścieżki:

```
import numpy

# Funkcja zwraca True, jeżeli w macierzy sąsiedztwa istnieje połączenie pomiędzy wierzchołkami i i j
def isConnected(A, i, j):
    # Inicjalizacja macierzy ścieżek i przypisanie do niej macierzy sąsiedztwa
    paths = A

    # Ustalenie liczby wierzchołków w grafie
    numberOfVertices = A.shape[0]

    # Ustalenie liczby krawędzi w grafie
    numberOfEdges = numpy.sum(A)/2

    # Jeżeli vi sąsiaduje z vj, funkcja zwraca True
    if paths[i-1][j-1] > 0:
        print('Wierzchołek', i, 'i wierzchołek', j, 'sąsiadują ze sobą')
        return True

    else:
        # Pętla trwa do momentu znalezienia ścieżki
        for pathLength in range(2, numberOfVertices):
            # Potęgowanie macierzy sąsiedztwa
            paths = numpy.dot(paths, A)

            # Jeżeli element w i-tym wierszu i j-tej kolumnie jest większy od zera, to znaleźliśmy ścieżkę
            if paths[i-1][j-1] > 0:
                print('Pomiędzy wierzchołkami',i, 'i', j, 'istnieje ścieżka
                ↪o długości', pathLength)
                return True
```

```
    # Brak ścieżek, wierzchołki nie są połączone
    if pathLength == numberOfEdges:
        print('Nie istnieje żadna ścieżka z wierzchołka', i, 'do wierzchołka', j)
        return False
```

Ponieważ kod umieściliśmy w funkcji, jego uruchomienie nie da żadnego wyjścia. Aby coś obliczyć, musimy wywołać funkcję z macierzą sąsiedztwa i numerami wierzchołków *i* oraz *j*. Zapisanie kodu w postaci funkcji pozwala na jego ponowne wykorzystanie z różnymi danymi wejściowymi i daje nam możliwość sprawdzenia, czy różne wierzchołki są ze sobą połączone.

W ramach testów wykorzystamy powyższy kod do znalezienia długości ścieżek w małym grafie. Dzięki temu będziemy mogli łatwo sprawdzić poprawność otrzymanych wyników za pomocą diagramu sieci. W rozdziale 8. „Przechowywanie i wyodrębnianie cech z grafów, drzew i sieci" analizowaliśmy dwa grafy pokazane na rysunku 9.6.



Rysunek 9.6. Graf G_1 (po lewej) i graf G_2 (po prawej)

Nazwijmy graf z lewej części rysunku G_1, a graf z prawej G_2. Równie dobrze grafy te mogłyby być sieciami z ważonymi krawędziami, ale wagi nie mają znaczenia dla określenia, czy dwa wierzchołki są ze sobą połączone:

```
# Macierz sąsiedztwa grafu G1
A1 = numpy.array([[0, 1, 1, 0, 1, 0], [1, 0, 1, 1, 0, 1],
                  [1, 1, 0, 1, 1, 0], [0, 1, 1, 0, 1, 0],
                  [1, 0, 1, 1, 0, 0], [0, 1, 0, 0, 0, 0]])

# Sprawdzenie połączeń pomiędzy niektórymi wierzchołkami
print(isConnected(A1, 1, 4))
print(isConnected(A1, 2, 3))
print(isConnected(A1, 5, 6))
```

W powyższym kodzie zdefiniowaliśmy macierz sąsiedztwa dla grafu G_1 i sprawdziliśmy, czy kilka par wierzchołków jest ze sobą połączonych. Kod wyświetlił następujące wyniki:

```
Pomiędzy wierzchołkami 1 i 4 istnieje ścieżka o długości 2
True
Wierzchołek 2 i wierzchołek 3 sąsiadują ze sobą
True
Pomiędzy wierzchołkami 5 i 6 istnieje ścieżka o długości 3
True
```

Oczywiście wyniki te są zgodne z tym, co widzimy na rysunku 9.6. W grafie istnieje ścieżka z v_1 do v_4 składająca się z dwóch krawędzi, krawędź łącząca v_2 z v_3 oraz trzyelementowa ścieżka z v_5 do v_6.

W przypadku niektórych wierzchołków z grafu G_2 kod powinien zwrócić `False`. Sprawdźmy:

```
# Macierz sąsiedztwa grafu G2
A2 = numpy.array([[0, 1, 0, 0, 0, 0], [1, 0, 0, 0, 0, 1],
                  [0, 0, 0, 1, 1, 0], [0, 0, 1, 0, 1, 0],
                  [0, 0, 1, 1, 0, 0], [0, 1, 0, 0, 0, 0]])

print(isConnected(A2, 1, 6))
print(isConnected(A2, 2, 5))
print(isConnected(A2, 1, 4))
```

Powyższy kod wyświetli następujące informacje:

```
Pomiędzy wierzchołkami 1 i 6 istnieje ścieżka o długości 2
True

Nie istnieje żadna ścieżka z wierzchołka 2 do wierzchołka 5
False

Nie istnieje żadna ścieżka z wierzchołka 1 do wierzchołka 4
False
```

Ponownie wyniki są zgodne z rysunkiem prezentującym graf G_2. Wierzchołki v_1 i v_6 są ze sobą połączone, a wierzchołki v_2 i v_5 oraz v_1 i v_4 nie.

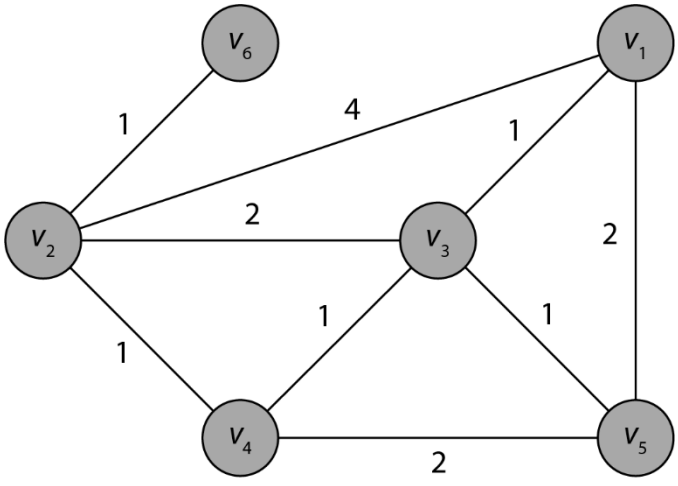
Teraz gdy znasz już metodę pozwalającą sprawdzić, czy rozwiązanie istnieje, zajmiemy się znajdowaniem najkrótszej ścieżki w małym problemie.

> Zwróć uwagę, że w przypadku dużych sieci sprawdzenie, czy połączenie istnieje, jest dość kosztowne. W takim przypadku warto przejść od razu do szukania najkrótszych ścieżek, chociaż należy zdawać sobie sprawę, że wyszukiwanie nie powiedzie się, jeśli wierzchołki v_i i v_j nie są ze sobą połączone.

Znajdowanie najkrótszych ścieżek metodą siłową

Jak wspomnieliśmy w poprzednim podrozdziale, interesują nas ścieżki od wierzchołka v_i do wierzchołka v_j z minimalną sumą wag krawędzi. Przyjrzyjmy się możliwości znalezienia najkrótszych ścieżek za pomocą algorytmu siłowego.

Rozważ sieć z rysunku 9.7, którą omówiliśmy w rozdziale 8. „Przechowywanie i wyodrębnianie cech z grafów, drzew i sieci". Niech V będzie zbiorem wierzchołków, E będzie zbiorem krawędzi, a W zbiorem wag.



Rysunek 9.7. Sieć

Przykładowym problemem, który spróbujemy rozwiązać, jest znalezienie najkrótszej ścieżki z v_1 do v_2. Istnieje wiele ścieżek pomiędzy tymi dwoma wierzchołkami. W tabeli 9.1 wymieniono je wraz z ich długościami.

Tabela 9.1. Wszystkie ścieżki od v_1 do v_2 i ich długości. Nie pokazano ścieżek, które ponownie przechodzą przez ten sam wierzchołek

Ścieżka z v1 do v2	Długość ścieżki
v_1-v_2	4
v_1-v_3-v_2	1 + 2 = 3
v_1-v_3-v_4-v_2	1 + 1 + 1 = 3
v_1-v_3-v_5-v_4-v_2	1 + 1 + 2 + 1 = 5
v_1-v_5-v_3-v_2	2 + 1 + 2 = 5
v_1-v_5-v_4-v_2	2 + 2 + 1 = 5
v_1-v_5-v_4-v_3-v_2	2 + 2 + 1 + 2 = 7

Analiza pełnej listy ścieżek z v_1 do v_2 pozwala łatwo zauważyć, że najkrótsze ścieżki to te znajdujące się w wyróżnionych wierszach. Ścieżki te mają długość trzech jednostek i są to v_1-v_3-v_2 oraz v_1-v_3-v_4-v_2.

Zauważ, że ścieżki te składają się z większej liczby krawędzi niż ścieżka o minimalnej liczbie krawędzi przechodząca bezpośrednio od v_1 do v_2, która ma długość 4 jednostek. Oczywiście długość ścieżki nie jest koniecznie związana z liczbą krawędzi. Nie powinniśmy oczekiwać, że najkrótsze ścieżki będą zawsze składały się z najmniejszej liczby krawędzi.

W tym przykładzie po prostu wymieniliśmy wszystkie możliwe ścieżki, ale w przypadku dużego grafu może to być niezwykle kosztowne. Załóżmy na przykład, że graf z n wierzchołkami jest kompletny, co oznacza, że pomiędzy każdą parą wierzchołków znajduje się krawędź. Tak

więc wierzchołek v_1 sąsiaduje z n–1 krawędziami. Wierzchołek v_2 ma n–2 sąsiednich krawędzi plus krawędź od v_1 do v_2, która została już wcześniej uwzględniona. Wierzchołek v_3 ma n–3 sąsiednich krawędzi plus dwie krawędzie łączące go z v_1 i v_2. Kontynuując ten wzorzec, w końcu znajdujemy tylko 1 nieuwzględnioną krawędź związaną z wierzchołkiem v_{n-1}. W ten sposób możemy obliczyć liczbę krawędzi w grafie. A zatem łączna liczba krawędzi jest równa:

$$1 + 2 + 3 + \cdots + n - 1$$

Zgodnie z dowodem indukcyjnym z rozdziału 2. „Logika formalna i dowody matematyczne" suma pierwszych n–1 nieujemnych liczb całkowitych jest równa:

$$\frac{(n-1)n}{2}$$

Jeśli graf ma na przykład 100 wierzchołków, to łącznie jest w nim $^{(100)(99)}/_{2} = 4950$ krawędzi. W takim grafie mogą istnieć miliony różnych ścieżek od jednego wierzchołka do drugiego!

Ile ścieżek będzie więc pomiędzy parą wierzchołków? Załóżmy, że chcemy ustalić liczbę ścieżek z v_1 do v_2, które przechodzą przez k dodatkowych wierzchołków. Zgodnie z informacjami z rozdziału 4. „Kombinatoryka z użyciem SciPy" mamy $|V|$–2 krawędzie do wyboru. Liczba ścieżek przechodzących przez k wierzchołków będzie więc równa:

$$\binom{|V|-2}{k} = \frac{(|V|-2)!}{k!\,(|V|-2-k)!}$$

Jest to prawda dla każdego k od 0 do $|V|$–2. Stąd liczba ścieżek przechodzących przez 5 dodatkowych wierzchołków w naszym 100-wierzchołkowym kompletnym grafie jest równa:

$$\binom{100-2}{5} = \binom{98}{5} = \frac{98!}{5!\,(98-5)!} = 67910864$$

Oczywiście nie ma powodu, dla którego w ścieżce miałoby być tylko pięć dodatkowych wierzchołków. Równie dobrze może ich być 2, 3, 4, ..., 98, co oznacza, że łączna liczba ścieżek to:

$$\binom{98}{0} + \binom{98}{1} + \binom{98}{2} + \cdots + \binom{98}{98}$$

Chociaż sposób wyliczenia tej wartości wykracza poza zakres niniejszej książki, wiadomo, że ta suma jest równa:

$$2^{98} \approx 3{,}17 \cdot 10^{29}$$

Tym samym podejście oparte na metodzie siłowej jest wyraźnie ograniczone! Przetestowanie tak wielu ścieżek zajęłoby całkowicie nierealną ilość czasu. Ponadto ten 100-wierzchołkowy graf jest dość mały, zwłaszcza jeśli weźmie się pod uwagę fakt, że na przykład mapy, takie jak Mapy Google, umieszczają wierzchołek na każdym skrzyżowaniu. Oznacza to, że w samym Nowym Jorku jest ponad 12 000 wierzchołków!

Chociaż ta metoda jest łatwa do zrozumienia, jej praktyczne wykorzystanie jest nie możliwe. Potrzebne nam będzie bardziej strategiczne podejście pozwalające znaleźć najkrótszą ścieżkę w sensownym czasie. Potrzebujemy skutecznego sposobu na znalezienie najkrótszych ścieżek między określonymi wierzchołkami w sieciach lub sieciach skierowanych przy założeniu, że rozwiązanie istnieje. To właśnie robi algorytm Dijkstry. Czas więc się z nim zapoznać!

Algorytm Dijkstry znajdowania najkrótszych ścieżek

W tym podrozdziale omówimy algorytm Dijkstry, który służy do znajdowania najkrótszych ścieżek. Przedstawimy jego działanie w prosty sposób i zastosujemy algorytm ręcznie na małej sieci.

Algorytm Dijkstry jest najpopularniejszym algorytmem znajdowania ścieżek w sieciach. Algorytm został nazwany na cześć holenderskiego informatyka Edsgera W. Dijkstry, który opracował go w 1956 roku. W tamtych czasach informatyka była nową dziedziną i na rynku istniało niewiele czasopism naukowych jej poświęconych. Dlatego autor nie opublikował swoich odkryć aż do 1959 roku.

Aby zrozumieć ideę algorytmu i wyrobić sobie intuicję, zaczniemy od prezentacji jego działania na małej sieci z rysunku 9.7. Zrozumienie sposobu pracy algorytmu jest ważne, ponieważ istnieje wiele jego odmian. Mamy nadzieję, że nauczysz się go dostosowywać do własnych problemów!

Tak jak w poprzednim podrozdziale, interesować nas będą najkrótsze ścieżki z v_1 do v_2. Ponieważ wykorzystana przez nas sieć była niewielka, takie ścieżki udało się nam znaleźć metodą siłową, są to:

$$v_1 - v_3 - v_2 \text{ i } v_1 - v_3 - v_4 - v_2$$

Każda z nich ma długość 3 jednostek. W praktyce, aby znaleźć najkrótszą ścieżkę z v_1 do v_2, znaleźlibyśmy najkrótsze ścieżki z v_1 do wszystkich wierzchołków po drodze do v_2 (tak działa typowa implementacja algorytmu Dijkstry).

Algorytm Dijkstry

Zaczniemy od wierzchołka v_1 i przejdziemy przez graf, realizując algorytm Dijkstry. W trakcie przejścia będziemy utrzymywali dwie listy: listę wierzchołków, które odwiedziliśmy, oraz listę wierzchołków, w których jeszcze nie byliśmy. Na początku zbiór odwiedzonych wierzchołków będzie pusty, a wszystkie wierzchołki znajdą się w zbiorze nieodwiedzonych:

- *odwiedzone wierzchołki* = { },
- *nieodwiedzone wierzchołki* = $\{v_1, v_2, v_3, v_4, v_5, v_6\}$.

W naszym problemie punktem wyjścia (źródłem, ang. *source*) będzie wierzchołek v_1. Algorytm Dijkstry składa się z następujących kroków:

- **Inicjalizacja**. Ustaw odległość od źródła do każdego wierzchołka na nieskończoność. Ustaw odległość źródła do niego samego na 0.
- Odwiedź najbliższy nieodwiedzony sąsiedni wierzchołek, który znajduje się najbliżej od źródła (w przypadku większej liczby wierzchołków o tej samej odległości można wybrać dowolny z nich):
  a. Jeśli jakiekolwiek ścieżki przechodzące przez aktualnie przetwarzany wierzchołek są krótsze niż dotychczas znane trasy, zaktualizuj tabele odległości.
  b. Dla każdej zmienionej odległości zapamiętaj bieżący wierzchołek jako „poprzedni".
  c. Dodaj bieżący wierzchołek do listy odwiedzonych wierzchołków.
- Powtarzaj poprzedni punkt, aż odwiedzone zostaną wszystkie wierzchołki.

Algorytm Dijkstry zwróci najkrótsze ścieżki od źródła v_1 do każdego innego wierzchołka grafu. Jest to znacznie więcej, niż oczekiwaliśmy, ale w wielu zagadnieniach i tak interesuje nas coś więcej niż tylko ścieżka pomiędzy określonymi wierzchołkami.

Algorytm Dijkstry jest przykładem **algorytmu zachłannego**, ponieważ w każdym kroku wybiera najkrótszą ścieżkę od źródła. Oczywiście generowanie ścieżki w oparciu o najkrótsze trasy z punktu startowego do kolejnych punktów na trasie nie musi prowadzić do najlepszej ścieżki, ale czasami się to zdarza. Jeśli dopisze nam szczęście, to dzięki niektórym z tych wczesnych wyborów znajdziemy najkrótszą trasę. Jeśli nie, algorytm i tak znajdzie rozwiązanie, ale zanim to zrobi, być może będzie musiał cofnąć się wiele razy. Proces ten nie jest szybki, ale i tak trwa to *znacznie* krócej niż znalezienie trasy metodą siłową. Dzięki temu mamy szansę znaleźć trasę w rzeczywistych problemach wielkoskalowych.

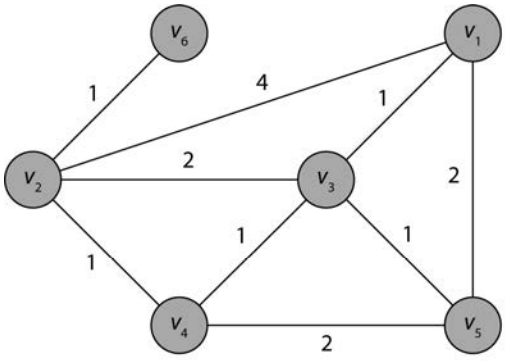
Algorytm Dijkstry zastosowany do małego problemu

Sprawdźmy, czy da się wykonać te kroki dla małej sieci z poprzedniego przykładu! W każdym kroku wyjaśnimy, co się dzieje, narysujemy zaktualizowaną sieć, zaznaczymy bieżący wierzchołek i nowe krawędzie, które mają zostać włączone do najkrótszej ścieżki, zaktualizujemy tabelę najkrótszych odległości i poprzednich wierzchołków oraz zaktualizujemy listy odwiedzonych i nieodwiedzonych wierzchołków.

Krok 0 (inicjalizacja). Ustaw najkrótszą odległość do każdego wierzchołka na nieskończoność (∞). Ustaw odległość od źródła do niego samego na 0 (rysunek 9.8).

Krok 1. Dodaj v_1 do zbioru odwiedzonych wierzchołków. Znajdź odległości od źródła do jego wszystkich sąsiadów ze zbioru nieodwiedzonych wierzchołków. Jeśli znaleziona odległość jest krótsza niż aktualnie znana, zapisz ją w tabeli (rysunek 9.9).

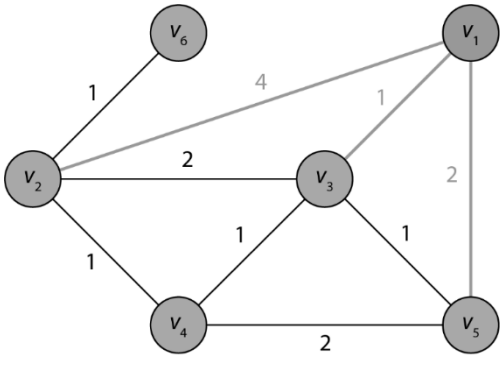
Krok 2. Odwiedź jeszcze nieodwiedzony wierzchołek o najkrótszej dotychczas znalezionej odległości od źródła, dodaj go do zbioru odwiedzonych wierzchołków, znajdź odległości od źródła przez ten wierzchołek do każdego nieodwiedzonego jeszcze wierzchołka i zapisz nowe wartości, o ile są one mniejsze niż dotychczasowe (w tym przykładzie będą one wynosić 1 plus długość nowej krawędzi; rysunek 9.10).



Wierzchołek	Najkrótsza odległość	Poprzedni wierzchołek
v_1	0	
v_2	∞	
v_3	∞	
v_4	∞	
v_5	∞	
v_6	∞	

Odwiedzone wierzchołki	Nieodwiedzone wierzchołki
	$v_1, v_2, v_3, v_4, v_5, v_6$

Rysunek 9.8. Zerowy krok algorytmu Dijkstry



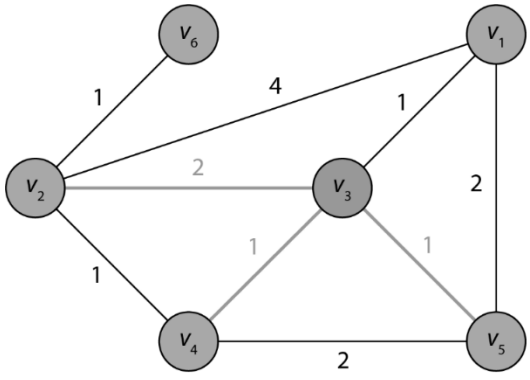
Wierzchołek	Najkrótsza odległość	Poprzedni wierzchołek
v_1	0	
v_2	4	v_1
v_3	1	v_1
v_4	∞	
v_5	2	v_1
v_6	∞	

Odwiedzone wierzchołki	Nieodwiedzone wierzchołki
v_1	v_2, v_3, v_4, v_5, v_6

Rysunek 9.9. Pierwszy krok algorytmu Dijkstry

Krok 3. Odwiedź nieodwiedzony wierzchołek o najkrótszej jak dotąd odległości od źródła (v_4). Dodaj go do zbioru odwiedzonych wierzchołków, znajdź odległości od źródła przez ten wierzchołek do każdego nieodwiedzonego wierzchołka i zapisz wszystkie odległości, które uległy skróceniu.

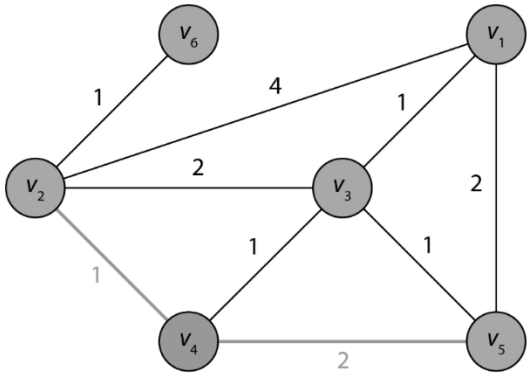
Tym razem zarówno v_4, jak i v_5 znajdują się w odległość 2 jednostek od źródła. Arbitralnie wybierzemy więc v_4 (rysunek 9.11).



Wierzchołek	Najkrótsza odległość	Poprzedni wierzchołek
v_1	0	
v_2	1 + 2 = 3	v_3
v_3	1	v_1
v_4	1 + 1 = 2	v_3
v_5	2	v_1
v_6	∞	

Odwiedzone wierzchołki	Nieodwiedzone wierzchołki
v_1, v_3	v_2, v_4, v_5, v_6

Rysunek 9.10. Drugi krok algorytmu Dijkstry



Wierzchołek	Najkrótsza odległość	Poprzedni wierzchołek
v_1	0	
v_2	3	v_3
v_3	1	v_1
v_4	2	v_3
v_5	2	v_1
v_6	∞	

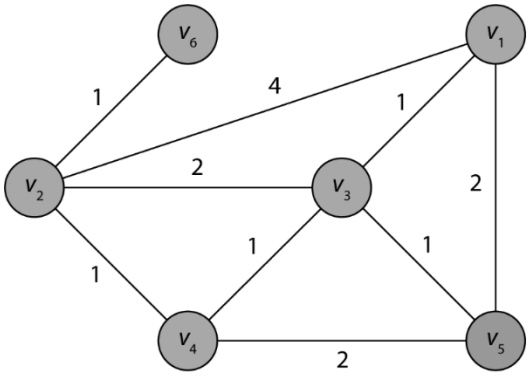
Odwiedzone wierzchołki	Nieodwiedzone wierzchołki
v_1, v_3, v_4	v_2, v_5, v_6

Rysunek 9.11. Trzeci krok algorytmu Dijkstry

Odległość do v_2 wynosi $2 + 1 = 3$, co nie jest lepszą wartością. Odległość do v_5 wynosi $2 + 2 = 4$, co też nie jest poprawne. W tym kroku nie dokonamy więc żadnych aktualizacji i po prostu przejdziemy do następnej najkrótszej odległości na liście.

Krok 4. Odwiedź kolejny nieodwiedzony jeszcze wierzchołek o najkrótszej znalezionej do tej pory odległości (v_5), dodaj go do zbioru odwiedzonych wierzchołków, znajdź odległości od źródła przez ten wierzchołek do każdego nieodwiedzonego wierzchołka i zapisz wszystkie odległości, które uległy skróceniu.

Wierzchołek v_5 nie ma już żadnych nieodwiedzonych sąsiadów (rysunek 9.12). Przejdźmy więc do kolejnego kroku.

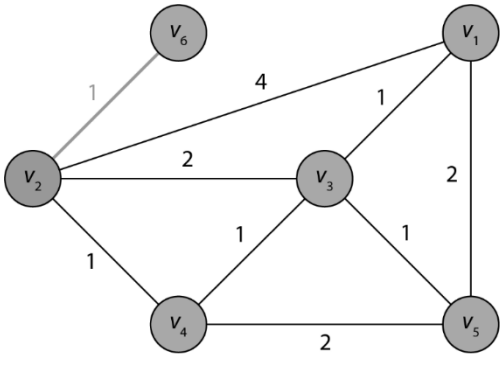


Wierzchołek	Najkrótsza odległość	Poprzedni wierzchołek
v_1	0	
v_2	3	v_3
v_3	1	v_1
v_4	2	v_3
v_5	2	v_1
v_6	∞	

Odwiedzone wierzchołki	Nieodwiedzone wierzchołki
v_1, v_3, v_4, v_5	v_2, v_6

Rysunek 9.12. Czwarty krok algorytmu Dijkstry

Krok 5. Odwiedź nieodwiedzony wierzchołek o najkrótszej znalezionej jak dotąd odległości od źródła (v_2), dodaj go do zbioru odwiedzonych wierzchołków, znajdź odległości od źródła przez ten wierzchołek do każdego nieodwiedzonego wierzchołka i zapisz wszystkie odległości, które uległy skróceniu (rysunek 9.13).

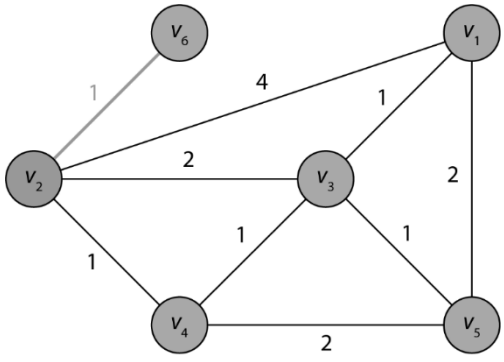


Wierzchołek	Najkrótsza odległość	Poprzedni wierzchołek
v_1	0	
v_2	3	v_3
v_3	1	v_1
v_4	2	v_3
v_5	2	v_1
v_6	3 + 1 = 4	v_2

Odwiedzone wierzchołki	Nieodwiedzone wierzchołki
v_1, v_2, v_3, v_4, v_5	v_6

Rysunek 9.13. Piąty krok algorytmu Dijkstry

Krok 6. Odwiedź nieodwiedzony wierzchołek o najkrótszej znalezionej jak dotąd odległości od źródła (v_6), dodaj go do zbioru odwiedzonych wierzchołków, znajdź odległości od źródła przez ten wierzchołek do każdego nieodwiedzonego wierzchołka i zapisz wszystkie odległości, które uległy skróceniu (rysunek 9.14).



Wierzchołek	Najkrótsza odległość	Poprzedni wierzchołek
v_1	0	
v_2	3	v_3
v_3	1	v_1
v_4	2	v_3
v_5	2	v_1
v_6	4	v_2

Odwiedzone wierzchołki	Nieodwiedzone wierzchołki
$v_1, v_2, v_3, v_4, v_5, v_6$	

Rysunek 9.14. Szósty krok algorytmu Dijkstry

Zbiór nieodwiedzonych wierzchołków jest teraz pusty, a najkrótsza odległość do v_2 wynosi 3 jednostki. Zgodnie z tabelą z rysunku 9.14 ostatnią krawędzią najkrótszej ścieżki jest krawędź od v_3 do v_2. Wierzchołkiem poprzedzającym v_3 w najkrótszej ścieżce jest v_1. Stąd najkrótsza ścieżka od v_1 do v_2 to:

$$v_1 - v_3 - v_2$$

Znaleziona ścieżka i jej długość odpowiadają wartościom, które znaleźliśmy wcześniej metodą siłową. Tym razem jednak zastosowaliśmy bardziej systematyczne podejście. Zapiszmy też dodatkowe wyniki, które daje nam algorytm Dijkstry, czyli najkrótsze ścieżki od v_1 do każdego innego węzła. Podsumowanie wyników znajduje się w tabeli 9.2.

Tabela 9.2. Najkrótsze ścieżki od v_1 do każdego innego wierzchołka znalezionego za pomocą algorytmu Dijkstry

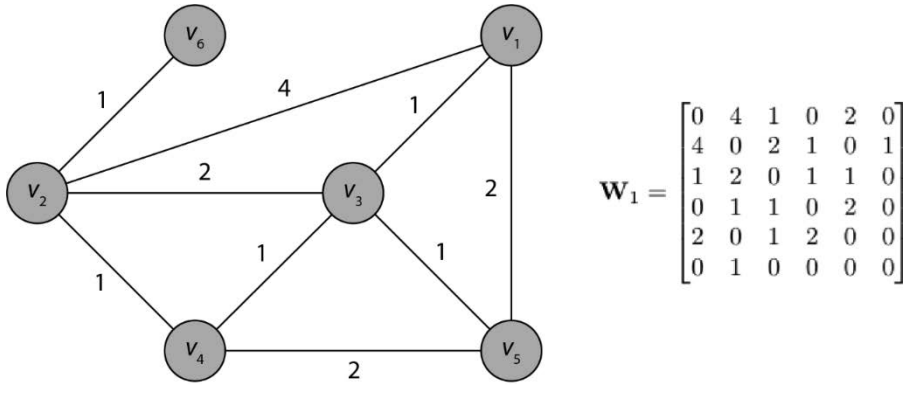
Wierzchołek końcowy	Ścieżka	Odległość
v_2	$v_2 \leftarrow v_3 \leftarrow v_1$	3
v_3	$v_3 \leftarrow v_1$	1
v_4	$v_4 \leftarrow v_3 \leftarrow v_1$	2
v_5	$v_5 \leftarrow v_1$	2
v_6	$v_6 \leftarrow v_2 \leftarrow v_3 \leftarrow v_1$	4

W tym podrozdziale poznałeś algorytm Dijkstry wykorzystywany do znajdowania najkrótszych ścieżek między wierzchołkami w sieci. Pokazaliśmy też, jak można zastosować go ręcznie w przypadku małej sieci. Teraz gdy rozumiesz już, jak działa ten algorytm, spróbujmy zaimplementować go w Pythonie, tak abyś mógł rozwiązywać jeszcze większe problemy!

Implementacja algorytmu Dijkstry w Pythonie

Wiesz już, jak działa algorytm Dijkstry, teraz zajmiemy się jego implementacją w Pythonie.

Dane wejściowe do algorytmu to sieć i wierzchołek źródłowy. Najprostszym sposobem reprezentacji sieci jest użycie macierzy wag, która została wprowadzona w rozdziale 8. „Przechowywanie i wyodrębnianie cech z grafów, drzew i sieci". Sieć z rysunku 9.7 wraz z odpowiadającą jej macierzą wag pokazano na rysunku 9.15.



Rysunek 9.15. Mała sieć i jej macierz wag

W kontekście problemu najkrótszej trasy macierz wag można nazwać macierzą odległości. My jednak powstrzymamy się od stosowania tego określenia, ponieważ jak miałeś okazję przekonać się w poprzednich podrozdziałach, problemy z najkrótszą ścieżką mogą, ale nie muszą odnosić się do odległości.

Wynikiem działania algorytmu będzie tabela, taka jak ta w prawym górnym rogu rysunku 9.14. Tabela ta będzie zawierała najkrótsze odległości z wierzchołka źródłowego do każdego z pozostałych wierzchołków.

Tabelę 9.2 można wygenerować również bezpośrednio w kodzie algorytmu, ale my zrobimy to poza nim.

Stworzymy funkcję, która przyjmie macierz wag i wierzchołek źródłowy, wykona algorytm Dijkstry i zwróci tabelę. Ponieważ kod będzie dość długi, pokażemy go we fragmentach wraz z ich wyjaśnieniem.

Najpierw zaimportujemy NumPy i stworzymy krótką dokumentację, która podsumuje działanie kodu. Jest to dobra praktyka podczas tworzenia nowej funkcji lub dużej porcji kodu:

```
import numpy

# Implementacja algorytmu Dijkstry znajdująca najkrótsze ścieżki do
# wszystkich innych wierzchołków w grafie
#
# WEJŚCIE
# W — kwadratowa macierz wag
# i — numer wierzchołka startowego
#
# WYJŚCIE
# shortestDistances — tablica najkrótszych odległości od wierzchołka startowego do pozostałych
#
# previousVertices — tablica przedostatnich wierzchołków w ścieżce od wierzchołka początkowego
#
```

Teraz zdefiniujemy funkcję o nazwie `Dijkstra`, która przyjmie macierz wag W i wierzchołek startowy v_i. Na początku musimy ustalić liczbę wierzchołków i zainicjować kilka tablic, które będą zawierały dane dotyczące wierzchołków, w tym informację, czy został on odwiedzony.

Ustawimy również odległości do wszystkich wierzchołków na ∞, odległość wierzchołka źródłowego do niego samego na 0 i oznaczymy wierzchołek źródłowy jako odwiedzony:

```
def Dijkstra(W, i):
    # Ustalenie liczby wierzchołków w grafie
    n = W.shape[0]

    # Inicjalizacja tablicy shortestDistances
    shortestDistances = numpy.array([numpy.inf] * n)

    # Inicjalizacja tablicy previousVertices
    previousVertices = numpy.array([numpy.inf] * n)

    # Inicjalizacja zbioru nieodwiedzonych wierzchołków. Na początku znajdą się w nim wszystkie wierzchołki
    unvisited = numpy.array([1] * n)

    # Oznaczenie wierzchołka startowego jako odwiedzonego
    unvisited[i - 1] = 0

    # Ustawienie odległości pomiędzy wierzchołkiem startowym i nim samym na zero
    shortestDistances[i - 1] = 0
```

Następnie stworzymy pętlę, która będzie przechodzić przez wierzchołki. W pętli znajdziemy najbliższy nieodwiedzony wierzchołek (`x`) i go odwiedzimy:

```
    # Pętla po wierzchołkach. Pętla zakończy się, gdy zbiór nieodwiedzonych wierzchołków będzie pusty
    for _ in range(n):
        # Znajdowanie odległości do wszystkich sąsiednich wierzchołków
```

```
    # Odległości do pozostałych ustawiane są na zero
    distances = shortestDistances * unvisited

    # Znajdowanie najbliższego nieodwiedzonego wierzchołka (distances > 0)
    x = numpy.argmin(numpy.ma.masked_where(
        distances == 0, distances))

    # Oznaczenie wierzchołka x jako odwiedzony
    unvisited[x] = 0
```

W następnym etapie przejdziemy przez wierzchołki kolejną pętlą. Jeśli jakikolwiek sąsiedni i nieodwiedzony jeszcze wierzchołek znajduje się bliżej wierzchołka źródłowego (przy przejściu przez bieżący), zapiszemy jego nową, krótszą odległość od źródła oraz oznaczymy bieżący wierzchołek jako przedostatni w tej najkrótszej ścieżce znalezionej do tej pory przez algorytm:

```
    # Przejście przez wierzchołki
    for v in range(n):

        oldDistance = shortestDistances[v]
        newDistance = shortestDistances[x] + W[v,x]
        adjacent = W[v,x] > 0
        unvis = unvisited[v]

        # Jeżeli v i x są ze sobą połączone, a wierzchołek v nie został znaleziony, to szukamy
        #najkrótszej trasy do v...
        if adjacent and unvis and oldDistance > newDistance:
            # Zapamiętanie najkrótszej znalezionej do tej pory ścieżki
            shortestDistances[v] = shortestDistances[x] + W[v,x]
            # Zapamiętanie poprzedniego wierzchołka
            previousVertices[v] = x
```

Na koniec wyświetlimy tabelę, taką jak ta w prawym górnym rogu rysunku 9.14. Zwróć uwagę, że do numerów wierzchołków dodajemy 1, aby poradzić sobie z faktem, że Python numeruje wierzchołki od 0 (my numerujemy je od 1). Na wypadek, gdybyśmy chcieli połączyć algorytm z inną funkcją, zwracamy również te same informacje w postaci, w jakiej przechowuje je Python:

```
    # Wyświetlenie tabeli podobnej do tej z książki
    print(numpy.array([numpy.arange(n) + 1, shortestDistances,
                       previousVertices + 1]).T)
    # Zwrócenie wartości
    return shortestDistances, previousVertices
```

Po stworzeniu implementacji algorytmu Dijkstry powinniśmy ją przetestować. Nasza implementacja powinna zadziałać nawet w przypadku dużych problemów, ale zawsze powinniśmy przetestować nowy kod (zwłaszcza długi) na problemie ze znanym rozwiązaniem, aby upewnić się co do poprawności jego działania.

Przykład — najkrótsze ścieżki

Weźmy macierz wag małej sieci z rysunku 9.15 i sprawdźmy, czy algorytm zwróci poprawne wyniki pokazane na rysunku 9.14.

Najpierw zapiszemy macierz wag w postaci tablicy NumPy:

```
# Stworzenie macierzy wag dla sieci z rysunku 9.15
W1 = numpy.array([[0, 4, 1, 0, 2, 0],
                  [4, 0, 2, 1, 0, 1],
                  [1, 2, 0, 1, 1, 0],
                  [0, 1, 1, 0, 2, 0],
                  [2, 0, 1, 2, 0, 0],
                  [0, 1, 0, 0, 0, 0]])
```

Następnie wywołamy funkcję `Dijkstra` z macierzą `W1` i wierzchołkiem v_1:

```
# Uruchomienie algorytmu Dijkstry z punktem startowym w wierzchołku v1
Dijkstra(W1, 1)
```

Kod wyświetli następujący wynik:

```
[[ 1.  0. inf]
 [ 2.  3.  3.]
 [ 3.  1.  1.]
 [ 4.  2.  3.]
 [ 5.  2.  1.]
 [ 6.  4.  2.]]

(array([0., 3., 1., 2., 2., 4.]), array([inf,  2.,  0.,  2.,  0.,  1.]))
```

Tablica wyświetlona jako pierwsza zawiera te same wartości, co znaleziona przez nas ręcznie tablica z rysunku 9.14.

Druga wyświetlona wartość (czyli to, co zwróciła funkcja) zawiera dwie prawe kolumny z tablicy z rysunku 9.14. Wartości te są jednak mniejsze o 1, ponieważ indeksowanie w Pythonie rozpoczyna się od zera.

Wyświetlone dane może i są ciekawe, ale co z właściwymi ścieżkami? Wygodnie byłoby wygenerować same ścieżki, tak jak zrobiliśmy to ręcznie w tabeli 9.2. W przypadku długiej ścieżki cały proces byłby jednak bardzo żmudny, dlatego stworzymy funkcję, która zrobi to za nas!

Najpierw zdefiniujemy nową funkcję i zainicjalizujemy niektóre zmienne i listy:

```
# Funkcja odtwarza najkrótsze ścieżki z source do destination na podstawie previousVertices i je wyświetla
def printShortestPath(shortestDistances, previousVertices, source, destination):
    # Uwzględnienie różnic w indeksowaniu
    source -= 1
    destination -= 1

    # Konwersja previousVertices na int
    previousVertices = previousVertices.astype(int)

    # Inicjalizacja listy elementów ścieżki. Przypisanie do niej wierzchołka docelowego
    path = [destination]
```

Następnie w pętli będziemy dodawać poprzednie wierzchołki z tabeli, aż dotrzemy do źródła:

```
# Dodawanie poprzednich wierzchołków z previousVertices aż do momentu osiągnięcia wierzchołka startowego
for _ in range(previousVertices.shape[0] - 1):
    # Jeżeli wierzchołek startowy znajduje się w ścieżce, pętla jest przerywana
    if path[-1] == source:
        break
    # Jeżeli wierzchołek startowy nie znajduje się w ścieżce, dodawany jest do niej kolejny wierzchołek
    else:
        path.append(previousVertices[path[-1]])
```

Na koniec stworzymy i wypiszemy łańcuch znaków podobny do drugiej kolumny tabeli 9.2:

```
# Inicjalizacja wyjściowego łańcucha znaków
output = []

# Przejście przez ścieżkę od tyłu (od wierzchołka startowego do wierzchołka docelowego)
for i in numpy.flip(path):
    # Stworzenie listy łańcuchów znaków
    if i > 0:
        output.append('->')

    output.append('v' + str(i + 1))

# Wyświetlenie łańcuchów znaków bez spacji
print('Ścieżka =', *output, '\t\t Odległość =',
      shortestDistances[destination])
```

Uruchomimy kod, aby znaleźć najkrótsze ścieżki od v_1 do innych wierzchołków:

```
for i in range(2,7):
    printShortestPath(shortestDistances, previousVertices, 1, i)
```

Po uruchomieniu kod wypisze następujące informacje:

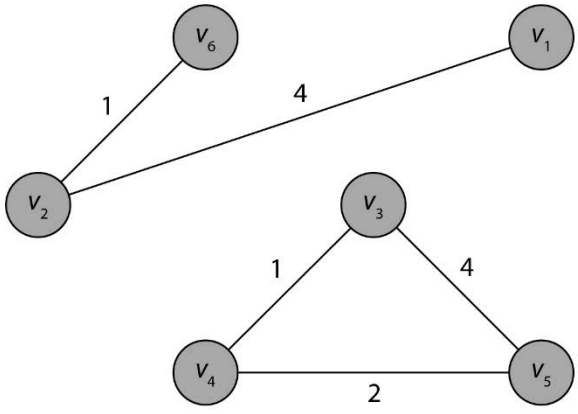
```
Ścieżka = v1 -> v3 -> v2                Odległość = 3.0
Ścieżka = v1 -> v3              Odległość = 1.0
Ścieżka = v1 -> v3 -> v4                Odległość = 2.0
Ścieżka = v1 -> v5              Odległość = 2.0
Ścieżka = v1 -> v3 -> v2 -> v6          Odległość = 4.0
```

Jak widzisz, otrzymane wartości są zgodne z tabelą 9.2.

W tym przykładzie wszystko wygląda świetnie. Spójrzmy jednak na trudniejszy — sieć, w której pomiędzy niektórymi parami wierzchołków nie istnieje żadna ścieżka.

Przykład — sieć bez połączenia

Rozważ sieć i macierz wag z rysunku 9.16.



$$\mathbf{W}_2 = \begin{bmatrix} 0 & 4 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 4 & 0 \\ 0 & 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 4 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Rysunek 9.16. Sieć bez połączenia

Ten graf składa się z dwóch składowych spójnych, które nie są połączone ze sobą żadnymi krawędziami, a zatem nie istnieje najkrótsza ścieżka między wierzchołkami pochodzącymi z różnych składowych. W związku z tym nasza implementacja algorytmu Dijkstry nie zadziała z taką macierzą wag. Aby znaleźć najkrótsze ścieżki, musimy najpierw dostosować nasze metody.

Zaczniemy od zapisania macierzy wag w postaci tablicy NumPy:

```
# Macierz wag dla sieci z rysunku 9.16
W2 = numpy.array([[0, 4, 0, 0, 0, 0],
                  [4, 0, 0, 0, 0, 1],
                  [0, 0, 0, 1, 4, 0],
                  [0, 0, 1, 0, 2, 0],
                  [0, 0, 4, 2, 0, 0],
                  [0, 1, 0, 0, 0, 0]])
```

Następnie stworzymy małą funkcję, która wykona kilka czynności: (1) za pomocą funkcji `isConnected` znajdzie wszystkie wierzchołki połączone z węzłem źródłowym; (2) uruchomi algorytm Dijkstry, aby znaleźć najkrótsze ścieżki:

```
# Znajdowanie najkrótszych ścieżek pomiędzy połączonymi ze sobą wierzchołkami
def distancesWithinComponent(source):
    # Inicjalizacja listy reprezentującej spójną składową
    component = [source]

    # Tworzenie spójnej składowej
    for i in range(1, W2.shape[0] + 1):
        if i != source and isConnected(W2, source, i):
            component.append(i)

    # Znajdowanie macierzy wag spójnej składowej
    subnetwork = W2[numpy.array(component) - 1,:][:,numpy.array(component) - 1]

    # Uruchomienie algorytmu Dijkstry
    return Dijkstra(subnetwork, 1)
```

Uruchommy algorytm z wierzchołkiem źródłowym v_1:

```
distancesWithinComponent(1)
```

Kod wyświetli następujący wynik:

```
Wierzchołek 1 i wierzchołek 2 sąsiadują ze sobą
Pomiędzy wierzchołkami 1 i 6 istnieje ścieżka o długości 2
[[ 1.  0. inf]
 [ 2.  4.  1.]
 [ 3.  5.  2.]]
(array([0., 4., 5.]), array([inf,  0.,  1.]))
```

Zauważ, że wynikowa tabela jest nieco zmodyfikowana. Pierwsza kolumna powinna zawierać wartości 1, 2 i 6. Jednak stworzyliśmy podsieć i zmieniliśmy numerację wierzchołków na 1, 2 i 3. Wyraźnie widzimy, że najkrótsza ścieżka od v_1 do v_2 przebiega po łączącej je krawędzi o długości 4. Najkrótsza ścieżka od v_1 do v_6 przechodzi zaś przez v_2 i ma długość 5.

Następnie wywołamy funkcję z wierzchołkiem startowym z drugiej składowej (v_3).

```
distancesWithinComponent(3)
```

Kod wyświetli następujący wynik:

```
Wierzchołek 3 i wierzchołek 4 sąsiadują ze sobą
Wierzchołek 3 i wierzchołek 5 sąsiadują ze sobą
[[ 1.  0. inf]
 [ 2.  1.  1.]
 [ 3.  3.  2.]]
(array([0., 1., 3.]), array([inf,  0.,  1.]))
```

W tym wywołaniu wierzchołkami są v_3, v_4 i v_5. Najkrótsza ścieżka od v_3 do v_4 przechodzi po prostu przez łączącą je krawędź i ma długość 1. Najkrótsza ścieżka od v_3 do v_4 przechodzi przez v_5 i ma długość 3. W przypadku małego grafu wyniki są dość oczywiste, ale dobrze wiedzieć, że możemy wykorzystać napisany przez nas kod do znajdowania ścieżek w grafach niespójnych.

Podsumowanie

W tym rozdziale wykorzystaliśmy struktury grafowe, w tym drzewa i sieci, które poznałeś w rozdziale 8. „Przechowywanie i wyodrębnianie cech z grafów, drzew i sieci". W rozdziale omówiliśmy kilka praktycznych problemów związanych z grafami i przedstawiliśmy popularne algorytmy ich rozwiązywania.

Zaczęliśmy od zagadnienia przeszukiwania grafów, w którym przechodzimy przez graf, aby odkryć jego strukturę i ewentualnie wykonać pewne obliczenia w każdym wierzchołku. Następnie zaprezentowaliśmy algorytm przeszukiwania w głąb — prawdopodobnie najpopularniejszy algorytm przeszukiwania grafów. Przed pokazaniem implementacji w Pythonie przeanalizowaliśmy jego ręczną implementację na małym grafie. Obie implementacje dały te same wyniki.

Następnie przeszliśmy do bardzo praktycznego problemu, jakim jest znajdowanie najkrótszych ścieżek między wierzchołkami w sieciach. Zagadnienie to odgrywa istotną rolę w znajdowaniu optymalnych tras podróży, trasowaniu wiadomości w sieciach komputerowych, wydajnym dostarczaniu energii elektrycznej w sieciach elektrycznych i w wielu innych obszarach. Ponieważ w niektórych sieciach nie istnieją ścieżki pomiędzy pewnymi wierzchołkami, stworzyliśmy krótką procedurę sprawdzania, czy wierzchołki są ze sobą połączone ścieżką. Następnie wykorzystaliśmy kilka metod z rozdziału 4. „Kombinatoryka z użyciem SciPy", aby pokazać, że znalezienie ścieżki metodą siłową jest praktycznie niewykonalne.

Ponieważ metoda siłowa nie dała oczekiwanych rezultatów, w kolejnym podrozdziale przedstawiliśmy algorytm Dijkstry do znajdowania najkrótszych tras z wierzchołka źródłowego do innego wierzchołka w sieci. Jest to zachłanny algorytm, który w każdej iteracji podejmuje najbardziej optymalną decyzję. Najpierw zaprezentowaliśmy działanie algorytmu krok po kroku na małym problemie, tak abyś lepiej zrozumiał jego działanie.

W ostatnim podrozdziale zaimplementowaliśmy od podstaw algorytm Dijkstry w Pythonie. Nasza implementacja działa tak samo jak w przykładzie, który rozwiązaliśmy ręcznie. Kod wygenerował dokładnie tę samą optymalną ścieżkę, ale dodatkowo pokazaliśmy, jak poprzez użycie innej macierzy wag i innego węzła źródłowego możemy natychmiast zastosować go do znalezienia ścieżki w innej sieci.

Teraz przejdziemy do trzeciej części książki, która koncentruje się na praktycznych zastosowaniach poznanych zagadnień, w tym na zastosowaniu regresji liniowej w uczeniu maszynowym, wyszukiwaniu w sieci za pomocą algorytmu Google PageRank oraz na analizie głównych składowych, czyli na metodzie redukcji wymiarów, która pozwala nam przechowywać duże zbiory danych za pomocą mniejszej liczby zmiennych.

Matematyka dyskretna — poznaj, zrozum, zastosuj!

Mimo że osiągnięcia matematyczne stały się podwalinami algorytmiki, wielu inżynierów nie w pełni rozumie reguły matematyki dyskretnej. Nawet jeśli nie stanowi to szczególnego problemu w codziennej pracy, w końcu okazuje się, że matematyka dyskretna jest niezbędna do osiągnięcia prawdziwej biegłości w operowaniu algorytmami i w pracy na danych. Co więcej, znajomość tej dziedziny bardzo ułatwia rozwiązywanie problemów z zakresu uczenia maszynowego. W ten sposób praktyczna biegłość w matematyce zauważalnie poprawia wyniki pracy inżynierów.

Ta książka jest kompleksowym wprowadzeniem do matematyki dyskretnej, przydatnym dla każdego, kto chce pogłębić i ugruntować swoje umiejętności informatyczne. W zrozumiały sposób przedstawiono tu metody matematyki dyskretnej i ich zastosowanie w algorytmach i analizie danych, włączając w to techniki uczenia maszynowego. Zaprezentowano również zasady oceny złożoności obliczeniowej algorytmów i używania wyników tej oceny do zarządzania pracą procesora. Omówiono także sposoby przechowywania struktur grafowych, ich przeszukiwania i znajdywania ścieżek między wierzchołkami. Pokazano też, jak wykorzystać przedstawione informacje podczas posługiwania się bibliotekami Pythona, takimi jak scikit-learn i NumPy.

W książce między innymi:

- terminologia i metody matematyki dyskretnej
- zastosowanie metod matematyki dyskretnej w algorytmach i analizie danych
- algebra Boole'a i kombinatoryka w podstawowych strukturach algorytmów
- rozwiązywanie problemów z dziedziny teorii grafów
- zadania związane z uczeniem maszynowym a matematyka dyskretna

Dr Ryan T. White — jest naukowcem specjalizującym się w uczeniu maszynowym i teorii prawdopodobieństwa. Wykłada matematykę w Florida Institute of Technology. Zajmuje się analizą stochastyczną i jej algorytmami, kieruje też projektami z zakresu uczenia maszynowego.

Archana Tikayat Ray — przygotowuje się do obrony doktoratu w Georgia Institute of Technology w Atlancie. Jej prace badawcze koncentrują się na uczeniu maszynowym i przetwarzaniu języka naturalnego (NLP).

ISBN 978-83-283-8396-8
9 788328 383968
Cena: 69,00 zł

Packt