ZANURZ SIĘ W ALGORYTMACH

PRZYGODA DLA POCZĄTKUJĄCYCH ODKRYWCÓW PYTHONA

BRADFORD TUCKFIELD

no starch press

Helion

Tytuł oryginału: Dive Into Algorithms: A Pythonic Adventure for the Intrepid Beginner

Tłumaczenie: Ludwika Majchrzak

ISBN: 978-83-283-8344-9

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/zanalg.zip

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

Printed in Poland.

Spis treści

6

Zaawansowana optymalizacja

OPTYMALIZACJA NIE JEST JUŻ DLA CIEBIE OBCYM TERMINEM. W ROZDZIALE 3. OMÓWILIŚMY METODĘ GRADIENTU PROSTEGO, KTÓRA POZWALA NA ZNAJDYWANIE EKSTREMÓW LOKALNYCH FUNKCJI. Mówiliśmy wtedy w przenośni o wspinaniu się na pagórki funkcji. W rzeczywistości każdy problem optymalizacyjny można opisać jako wersję takiego wspinania się: staramy się znaleźć najlepsze rozwiązanie pośród szerokiego spektrum dostępnych możliwości. Metoda gradientu prostego jest prosta i elegancka, posiada jednak niezaprzeczalnie piętę achillesową: może zaprowadzić nas do wierzchołka, który będzie optymalny jedynie lokalnie, natomiast globalnie istnieć będzie inna, lepsza opcja. Używając ponownie analogii z pagórkami, można powiedzieć, że gradient prosty może zaprowadzić nas na szczyt niewielkiego pagórka, podczas gdy schodząc chwilowo niewiele niżej, możemy zacząć wspinaczkę na ogromną górę, która pozwoli nam zobaczyć okolicę z dużo większej wysokości. Uniknięcie problemu ekstremum lokalnego jest jednym z najtrudniejszych i zarazem najistotniejszych aspektów zaawansowanej optymalizacji.

W tym rozdziale omówimy bardziej zaawansowane algorytmy optymalizacyjne na przykładzie konkretnego problemu optymalizacyjnego. Zajmiemy się problemem komiwojażera. Podamy kilka rozwiązań tego problemu i zbadamy, jakie mają niedociągnięcia. Na koniec przedstawimy symulowane wyżarzanie, zaawansowany algorytm optymalizacyjny, który eliminuje te niedociągnięcia i wykonuje globalną optymalizację zamiast optymalizacji lokalnej.

Życie komiwojażera

Jednym z najbardziej znanych zagadnień optymalizacyjnych w informatyce i kombinatoryce jest **problem komiwojażera** (ang. *traveling salesman problem*). Wyobraź sobie sprzedawcę, który chcąc sprzedać swoje produkty, musi odbyć podróż do kilku miast. Podróże między miastami są dla niego kosztowne, a koszt ten może objawiać się na wiele sposobów — czy to w postaci utraconych możliwości zarobku, czy w koszcie paliwa do samochodu, czy nawet w postaci bólu głowy od noszenia towarów podczas długiej wędrówki (rysunek 6.1).

Rysunek 6.1. Obwoźny sprzedawca (komiwojażer) w Neapolu

Problem komiwojażera to zadanie optymalizacyjne, którego celem jest znalezienie najlepszej drogi łączącej wszystkie miasta, które chcemy odwiedzić. Jak wszystkie najlepsze problemy w nauce, zadanie jest bardzo łatwe do sformułowana, natomiast niezwykle trudne do rozwiązania.

Sformułowanie problemu

Odpalmy od razu Pythona i zagłębmy się w temat. Na początek wygenerujemy losową mapę miejsc, które ma odwiedzić nasz komiwojażer. Zaczniemy od ustalenia liczby N, która reprezentować będzie liczbę miast do wylosowania. Przyjmijmy $N = 40$. Musimy teraz ustalić współrzędne 40 punktów na mapie: jedną wartość x i jedną wartość y dla każdego miasta. Użyjemy do tego celu modułu `numpy` pozwalającego generować liczby losowe:

```
import numpy as np
random_seed = 1729
np.random.seed(random_seed)
N = 40
x = np.random.rand(N)
y = np.random.rand(N)
```

Powyższy fragment kodu wykorzystuje funkcję `random.seed()`, która służy do ustawienia ziarna w generatorze liczb pseudolosowym (aby dowiedzieć się więcej o generatorach liczb pseudolosowych, przeczytaj rozdział 5.). Jeśli użyjemy tego samego ziarna, to wynikiem działania generatora liczb pseudolosowych będzie zawsze ten sam ciąg liczb. Jeśli użyjesz w swoim kodzie tego samego ziarna, uzyskasz dokładnie takie same wyniki i wykresy jak te przedstawione w niniejszym rozdziale.

Użyjemy teraz funkcji `zip`, która połączy wartości x i y, a następnie stworzymy listę tych par, będącą listą współrzędnych naszych 40 losowo wygenerowanych miast.

```
points = zip(x, y)
cities = list(points)
```

Jeśli uruchomisz `print(cities)` w konsoli Pythona, otrzymasz listę zawierającą nasze losowo wygenerowane punkty. Każdy z punktów odpowiada jednemu miastu. Miasta te nie mają nazw, ale nie stanowi to dla nas żadnej przeszkody, pierwsze z nich nazwiemy `cities[0]`, drugie `cities[1]` i tak dalej.

Mamy teraz wszystko, czego potrzebujemy, żeby przystąpić do rozwiązywania problemu komiwojażera. Pierwsza propozycja rozwiązania tego zadania to po prostu odwiedzenie wszystkich miast w kolejności, w jakiej pojawiają się one w naszej liście. Możemy zatem stworzyć plan podróży, który zawierać będzie kolejność, w jakiej odwiedzimy nasze miasta. Zapiszmy ten plan podróży w zmiennej `itinerary`:

```
itinerary = list(range(0, N))
```

Powyższa linia kodu jest innym sposobem zapisu bardziej rozwlekłego zdefiniowania tej zmiennej:

```
itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20, \
             21,22,23,24,25,26,  27,28,29,30,31,32,33,34,35,36,37,38,39]
```

Kolejność liczb w naszym planie podróży odpowiada sugerowanej kolejności, w jakiej zalecamy komiwojażerowi odwiedzić wszystkie miasta: jako pierwsze miasto z indeksem 0, następnie miasto z indeksem 1 i tak dalej.

Musimy teraz ocenić, czy plan podróży, który chcemy zaproponować sprzedawcy, jest dobry lub przynajmniej akceptowalny jako rozwiązanie problemu komiwojażera. Pamiętaj, że naszym zadaniem jest zminimalizowanie kosztu ponoszonego przez komiwojażera podczas podróży między miastami. Jaki zatem jest koszt tak zaprojektowanej podróży? Możemy użyć dowolnej funkcji kosztu: być może na części z dróg pojawiają się większe korki niż na innych, a może są gdzieś rzeki, które są trudne do pokonania, a może trudniej jest podróżować na północ niż na wschód. Zacznijmy jednak od prostej miary: załóżmy, że pokonanie odcinka o długości 1 kosztuje jeden złoty bez względu na to, w jakim kierunku podróżujemy, i bez względu na to, między jakimi miastami. Nie podajemy tutaj żadnej jednostki długości, ponieważ nasz algorytm działać będzie w ten sam sposób dla długości wyrażonych w metrach, kilometrach, milach czy latach świetlnych. Przyjmując wspomnianą funkcję kosztu, minimalizowanie kosztu sprowadzamy do minimalizowania odległości, jaką musi pokonać komiwojażer.

Aby obliczyć długość podróży odpowiadającą konkretnemu planowi podróży, musimy zdefiniować dwie dodatkowe funkcje. Pierwsza z funkcji wygeneruje zbiór linii łączących następujące po sobie miasta. Druga funkcja obliczy sumę odległości odpowiadających wszystkim liniom. Zacznijmy od stworzenia pustej listy, do której wpisywać będziemy informacje o naszych liniach:

```
lines = []
```

Przejdziemy teraz kolejno po wszystkich miastach z naszego planu podróży, w każdym kroku dodając linię odpowiadającą połączeniu między aktualnym miastem i miastem znajdującym się tuż po nim.

```
for j in range(0, len(itinerary) - 1):
    lines.append([cities[itinerary[j]], cities[itinerary[j + 1]]])
```

Wywołując `print(lines)`, zobaczysz, w jaki sposób przechowujemy informacje o liniach w Pythonie. Każda linia przechowywana jest w postaci listy zawierającej współrzędne obu miast. Na przykład wypisując na ekran `print(lines[0])`, otrzymasz następujący wynik:

```
[(0.21215859519373315, 0.1421890509660515), (0.25901824052776146,
0.4415438502354807)]
```

Zbierzmy teraz wszystkie te fragmenty kodu w jedną funkcję o nazwie `genlines` (skrót od ang. *generate lines* — generuj linie). Argumentami wejściowymi tej funkcji jest lista miast `cities` i plan podróży `itinerary`, a wynikiem jej działania jest lista linii łączących wszystkie miasta w kolejności zgodnej z wejściowym planem podróży:

```
def genlines(cities, itinerary):
    lines = []
    for j in range(0, len(itinerary) - 1):
        lines.append([cities[itinerary[j]], cities[itinerary[j + 1]]])
    return(lines)
```

Mamy teraz listę złożoną z linii łączących każde następujące po sobie dwa miasta z planu podróży, możemy zatem stworzyć funkcję mierzącą całkowity dystans pokonywany podczas takiej podróży. Na początek zdefiniujemy nasz całkowity dystans na poziomie zera, a następnie będziemy dodawać długość każdego z odcinków odpowiadających poszczególnym elementom z listy `lines`. Do obliczenia długości poszczególnych odcinków skorzystamy z twierdzenia Pitagorasa.

UWAGA *Wykorzystanie twierdzenia Pitagorasa do obliczania odległości na kuli ziemskiej nie jest do końca prawidłowe. Powierzchnia Ziemi jest zakrzywiona, więc do obliczenia odległości między dwoma punktami na kuli ziemskiej należałoby skorzystać z bardziej zaawansowanej geometrii. Ignorujemy tutaj tę krzywiznę. Możemy założyć, że nasz komiwojażer może przemieszczać się podkopami pod zakrzywioną skorupą ziemską, przebiegającymi w linii prostej między dwoma punktami. Inne podejście to założenie, że nasz sprzedawca żyje w jakiejś utopijnej z punku widzenia geometrii Płaskolandii, w której odległości można obliczać, korzystając z tego prostego wzoru wyznaczonego przez Pitagorasa już w starożytnej Grecji. W rzeczywistości jednak dla niedużych odległości wartości wyliczone za pomocą twierdzenia Pitagorasa są bardzo dobrym przybliżeniem prawdziwej odległości.*

```
import math
def howfar(lines):
    distance = 0
    for j in range(0, len(lines)):
        distance += math.sqrt(abs(lines[j][1][0] - lines[j][0][0])**2 +
        ↪abs(lines[j][1][1] - lines[j][0][1])**2)
    return(distance)
```

Powyższa funkcja przyjmuje jako argument listę linii i zwraca w wyniku swojego działania sumę długości tych wszystkich linii. Mając dostępne obie funkcje, możemy wykorzystać je równocześnie, aby obliczyć całkowity dystans pokonany

przez komiwojażera podczas podążania za zaproponowanym przez nas planem podróży:

```
totaldistance = howfar(genlines(cities, itinerary))
        print(totaldistance)
```

Gdy uruchamiam ten kod na swoim komputerze, otrzymuję wynik około 16,81. Jeśli użyłeś tego samego ziarna, powinieneś otrzymać ten sam wynik. Jeśli użyłeś innego ziarna do wygenerowania listy miast, wtedy Twoje wyniki najprawdopodobniej będą się trochę różnić.

Spróbujmy teraz wyobrazić sobie, co znaczy uzyskany przez nas wynik. Narysujmy w tym celu wykres podróży naszego komiwojażera:

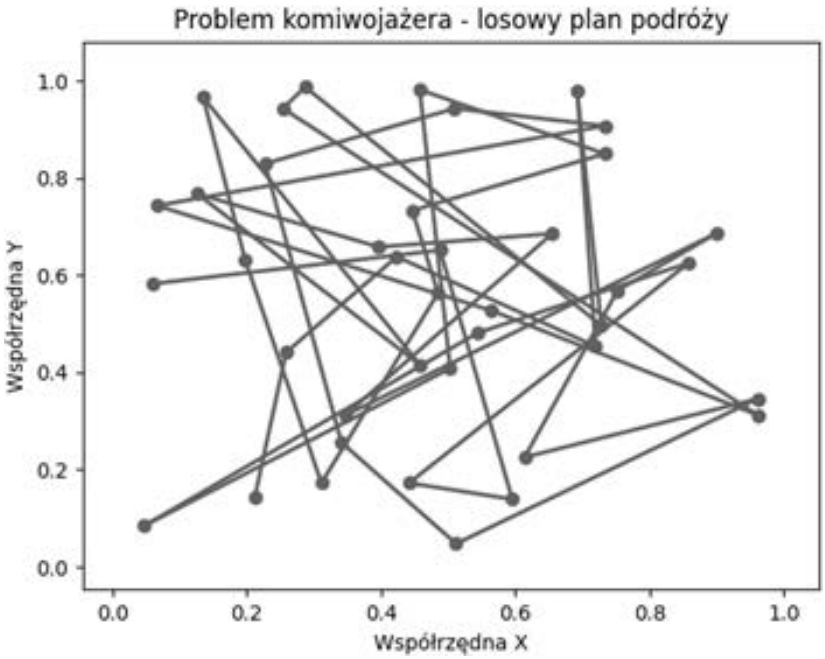
```
import matplotlib.collections as mc
import matplotlib.pylab as pl
def plotitinerary(cities, itin, plottitle, thename):
    lc = mc.LineCollection(genlines(cities, itin), linewidths=2)
    fig, ax = pl.subplots()
    ax.add_collection(lc)
    ax.autoscale()
    ax.margins(0.1)
    pl.scatter(x, y)
    pl.title(plottitle)
    pl.xlabel('Współrzędna X')
    pl.ylabel('Współrzędna Y')
    pl.savefig(str(thename) + '.png')
    pl.close()
```

Funkcja `plotitinerary()` ma cztery argumenty wejściowe: zmienna `cities` odpowiada liście miast, `itin` to plan podróży, który chcemy narysować (kolejność odwiedzania miast), `plottitle` jest tytułem, który pojawi się na górze wykresu, a `thename` zostanie wykorzystane jako nazwa pliku z rozszerzeniem png, w którym zostanie zapisany wygenerowany wykres. Do tworzenia wykresu funkcja wykorzystuje funkcje z modułu `pylab`, a do tworzenia listy zawierającej poszczególne linie — funkcje z modułu `collections`; oba te moduły należą do biblioteki `matplotlib`.

Wywołując tę funkcję z parametrami `plotitinerary(cities, itinerary, 'Problem komiwojażera - losowy plan podróży', 'rysunek2')`, uzyskasz wykres pokazany na rysunku 6.2.

Zapewne patrząc na ten wykres, widzisz od razu, że zaproponowany przez nas plan podróży nie jest w tym przypadku najlepszym rozwiązaniem problemu komiwojażera. Aktualny plan podróży powodowałby, że nasz biedny sprzedawca musiałby przejeżdżać wzdłuż i wszerz wielokrotnie po całej mapie. Na pewno jesteśmy w stanie stworzyć lepszy plan podróży, w którym przemierzając mapę z jednego jej krańca na drugi, sprzedawca zatrzyma się po drodze w miastach znajdujących się wzdłuż trasy przejazdu. W kolejnych sekcjach tego rozdziału skupimy się właśnie na celu minimalizowania długości trasy, którą będzie musiał pokonać nasz komiwojażer.

Pierwsze z rozwiązań, które omówimy, jest zdecydowanie najprostsze, jednak ma ono równocześnie najgorszą wydajność. W kolejnych sekcjach omówimy więc rozwiązania, w których odrobina złożoności rekompensuje znaczne zwiększenie wydajności obliczeniowej.



Rysunek 6.2. Wykres podróży odpowiadający kolejności, w jakiej poszczególne miasta zostały wylosowane

Mózg kontra mięśnie

Mogło Ci przyjść do głowy, że najłatwiej będzie stworzyć listę wszystkich możliwych planów podróży łączących miasta z naszej listy, obliczyć całkowitą długość trasy dla każdego z tych planów i zobaczyć, który z tych planów będzie najlepszy. Stwórzmy więc taką listę wszystkich planów podróży dla przypadku, gdy komiwojażer musi odwiedzić tylko trzy miasta:

- 1, 2, 3
- 1, 3, 2
- 2, 3, 1
- 2, 1, 3

- 3, 1, 2
- 3, 2, 1

W tym przypadku obliczenie długości trasy dla każdego z tych planów i sprawdzenie, który z tych planów jest najlepszy, nie powinno zająć zbyt długo. Takie podejście nazywamy *atakiem brute force*. Nazwa tego algorytmu nie wywodzi się od siły fizycznej (ang. *force*), ale od tego, że w przypadku sprawdzania wszystkich możliwych opcji wykorzystujemy muskuły naszego procesora, zamiast korzystać z inteligencji projektanta algorytmów, który mógłby znaleźć bardziej elegancie i szybsze w działaniu podejście.

Czasami atak *brute force* jest właściwym rozwiązaniem. Zwykle napisanie kodu stosującego tę metodę jest łatwe, a dodatkowo działa on skutecznie. Natomiast główną wadą metod bazujących na ataku *brute force* jest długi czas działania, który nigdy nie jest szybszy — a zwykle jest znacznie gorszy — od rozwiązań bazujących na algorytmach.

W przypadku problemu komiwojażera czas działania *brute force* rośnie tak szybko w miarę wzrostu liczby miast, że korzystanie z tej metody byłoby niepraktyczne dla liczby miast większej niż około 20. Aby to zauważyć, weź pod uwagę, jak duża jest liczba planów podróży (możliwych wariantów kolejności odwiedzanych miast), które trzeba sprawdzić dla tylko czterech miast:

1. Podczas wybierania pierwszego miasta, do którego mielibyśmy pojechać, nie odwiedziliśmy jeszcze żadnego z czterech miast, możemy więc dokonać tego wyboru na dokładnie cztery sposoby. Mamy więc 4 możliwości rozpoczęcia planu podróży.
2. Podczas wybierania drugiego miasta, do którego mielibyśmy pojechać, odwiedziliśmy już jedno miasto, pozostały więc nam trzy opcje do wyboru. Zatem liczba możliwych opcji wyboru dwóch pierwszych miast wynosi $4 \cdot 3 = 12$.
3. Podczas wybierania trzeciego miasta, do którego mielibyśmy pojechać, odwiedziliśmy już dwa miasta, mamy zatem już tylko dwie opcje do wyboru. Zatem liczba możliwych planów podróży dla pierwszych trzech miast wynosi $4 \cdot 3 \cdot 2 = 24$.
4. Podczas wybierania czwartego miasta, do którego mielibyśmy pojechać, nie mamy już wyboru. Odwiedziliśmy trzy miasta i pozostało nam ostatnie. Zatem całkowita liczba możliwych planów podróży wynosi $4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Zauważyłeś tu pewnie pewien wzór: mając N miast do odwiedzenia, obliczamy możliwą liczbę planów podróży według wzoru $N \cdot (N–1) \cdot (N–2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 = N!$ („N silnia”). Funkcja silnia rośnie niesamowicie szybko: 3! wynosi jedynie 6 (do takiej liczby opcji możemy użyć metody *brute force* nawet bez wykorzystania komputera), natomiast 10! wynosi już ponad 3 miliony (jest to liczba możliwości, która jest w miarę łatwa do przepracowania za pomocą *brute force* z użyciem nowoczesnego komputera), 18! wynosi już ponad 6 biliardów, 25! to ponad 15 kwadrylionów, a 35! i więcej zaczyna przechodzić możliwości zastosowania *brute force* z użyciem aktualnych technologii.

Tak szybki wzrost liczby możliwych przypadków nosi nazwę *eksplozji kombinatorycznej*. Zjawisko to nie ma ścisłej matematycznej definicji, natomiast odnosi się do przypadków analogicznych to tego, który tu zaobserwowaliśmy, gdzie rozważenie możliwych kombinacji i permutacji powoduje, że zwiększenie liczby początkowych elementów *N* prowadzi do niewspółmiernego wzrostu przypadków, które należałoby rozważyć. Wzrostu przypadków zwykle powodującego brak możliwości użycia metody *brute force*.

Na przykład liczba możliwych tras przejazdu między wszystkimi placówkami pocztowymi w Łodzi (których jest aktualnie 106 w samym mieście Łodzi, bez uwzględniania okolicznych miejscowości) jest większa od prawdopodobnej liczby atomów w całym wszechświecie, a uwzględniliśmy tutaj tylko jedno miasto. Podobnie liczba różnych partii szachów, jakie możemy rozegrać na szachownicy, jest również większa od liczby atomów we wszechświecie, a szachownica jest zwykle dużo mniejsza od powierzchni standardowego stołu jadalnego. To, że uzyskujemy prawie nieskończoną liczbę możliwości, mimo że rozpoczynamy z obiektem/listą z całą pewnością ograniczoną, jest sytuacją paradoksalną. W przypadku takich sytuacji nie ma żadnej możliwości wykorzystania metody *brute force*, co powoduje, że tym istotniejsze staje się dobre projektowanie algorytmów znajdujących rozwiązanie dla tego typu problemów. Eksplozja kombinatoryczna powoduje, że do rozwiązania problemu komiwojażera zamiast *brute force* zastosujemy podejście algorytmiczne. Nie mielibyśmy bowiem wystarczającej mocy obliczeniowej, aby przeprowadzić metodę *brute force* dla tego zadania.

Algorytm najbliższego sąsiada

Rozważmy zatem dość prosty i intuicyjny algorytm o nazwie *algorytm najbliższego sąsiada* (NN — ang. *nearest neighbour algorithm*). Zaczynamy od pierwszego miasta na liście. Następnie znajdujemy miasto, w którym jeszcze nie byliśmy, a które znajduje się najbliżej aktualnej miejscowości. W każdym kolejnym kroku szukamy miasta, w którym jeszcze nie byliśmy, a które znajduje się najbliżej miejsca, w którym jesteśmy. Taki proces minimalizuje odległość, którą mamy do pokonania w kolejnym kroku, przy czym nie mamy niestety pewności, że całkowity dystans również uda się zminimalizować. Zauważ, że zamiast sprawdzać każdy możliwy plan podróży, jak robilibyśmy w przypadku *brute force*, w każdym kroku szukamy jedynie najbliższego sąsiada. Dzięki takiemu podejściu uzyskujemy bardzo szybko działający algorytm, nawet dla dużych wartości *N*.

Implementacja algorytmu najbliższego sąsiada

Zacznijmy od napisania funkcji, która będzie w stanie odszukać najbliższego sąsiada dla wybranego miasta. Zaczynamy z punktem o nazwie `point` i listą miast reprezentowaną przez listę `cities`. Odległość między naszym punktem `point` i elementem listy `cities` o indeksie `j` obliczać będziemy, używając twierdzenia Pitagorasa:

```
point = [0.5, 0.5]
j = 10
distance = math.sqrt((point[0] - cities[j][0])**2 + (point[1] - cities[j][1])**2)
```

Chcąc teraz dowiedzieć się, który z punktów listy `cities` jest najbliżej punktu `point` (najbliższy sąsiad punktu `point`), musimy obliczyć odległość do każdego z dostępnych miast i porównać uzyskane wartości (listing 6.1).

Listing 6.1. Funkcja znajdująca najbliższego sąsiada wybranego punktu

```
def findnearest(cities, idx, nnitinerary):
    point = cities[idx]
    mindistance = float('inf')
    minidx = -1
    for j in range(0, len(cities)):
        distance = math.sqrt((point[0] - cities[j][0])**2 + (point[1] -
cities[j][1])**2)
        if distance < mindistance and distance > 0 and j not in nnitinerary:
            mindistance = distance
            minidx = j
return(minidx)
```

Po zdefiniowaniu funkcji `findnearest()` odszukującej najbliższego sąsiada możemy przejść do implementacji algorytmu NN. Zmienna przechowująca docelowy plan podróży nazywać się będzie `nnitinerary`. Załóżmy, że nasz komiwojażer mieszka w pierwszym mieście na naszej liście miast i w tym miejscu zaczyna swoją podróż:

```
nnitinerary = [0]
```

Nasz plan podróży będzie zwierał wszystkie *N* miast, zatem musimy przejść kolejno po wszystkich liczbach z zakresu od 0 do *N*–1, sprawdzić, który indeks odpowiada najbliższemu sąsiadowi miasta, w którym aktualnie jesteśmy, i dodać to miasto do naszego planu podróży. Zadanie to wykona funkcja o nazwie `donn()` (skrót od ang. *do nearest neighbour* — wykonaj algorytm najbliższego sąsiada), pokazana w listingu 6.2. Funkcja ta zaczyna od pierwszego miasta z listy `cities` i w każdym kroku dodaje do planu podróży najbliższego sąsiada ostatnio dodanego miasta, aż doda do planu podróży wszystkie miasta.

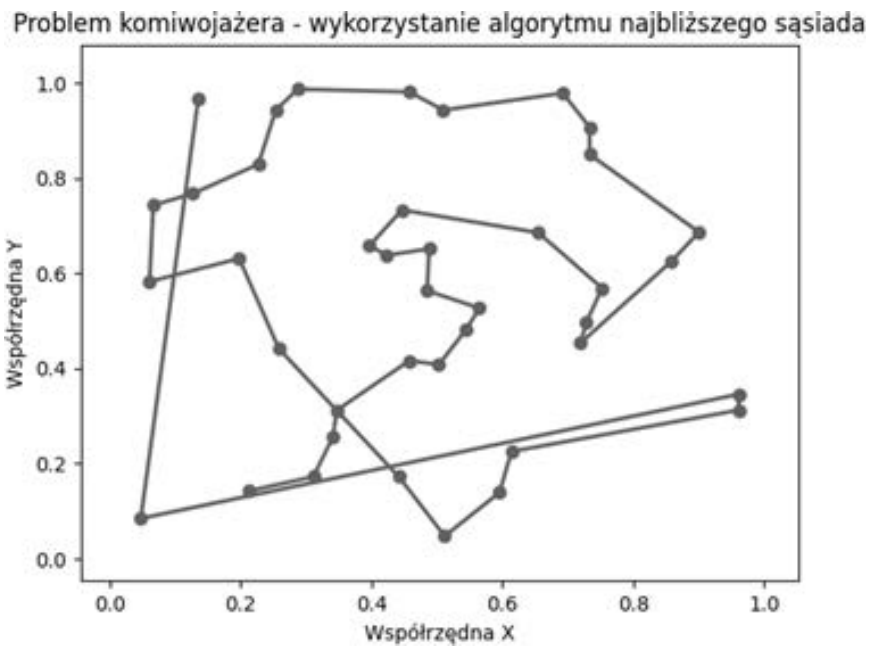
Listing 6.2. Funkcja znajdująca kolejnych najbliższych sąsiadów i zwracająca pełny plan podróży

```
def donn(cities, N):
    nnitinerary = [0]
    for j in range(0, N - 1):
        next = findnearest(cities, nnitinerary[len(nnitinerary) - 1],
        nnitinerary)
        nnitinerary.append(next)
    return(nnitinerary)
```

Mamy już wszystko, czego potrzebujemy do przeprowadzenia algorytmu najbliższego sąsiada i sprawdzenia jego wydajności. Na początek narysujmy wykres pokazujący podróż wyznaczoną przez nasz algorytm:

```
plotitinerary(cities, donn(cities, N), 'Problem komiwojażera - wykorzystanie
algorytmu najbliższego sąsiada', 'rysunek3')
```

Rezultat działania tego kodu możesz zobaczyć na rysunku 6.3.



Rysunek 6.3. Plan podróży wygenerowany za pomocą algorytmu najbliższego sąsiada

Możemy teraz sprawdzić, jak długa jest trasa całej podróży naszego komiwojażera w przypadku tego nowego planu podróży:

```
print(howfar(genlines(cities, donn(cities, N))))
```

Widzimy ogromną różnicę w porównaniu z początkowym, losowym planem podróży. W przypadku losowym nasz sprzedawca musiał pokonać dystans o długości 16,81, nasz algorytm skrócił tę trasę aż do 6,29. Pamiętaj, że nie używamy tutaj żadnych jednostek, zatem można to interpretować jako 6,29 kilometra (lub mili, lub parseka). Najistotniejsze jest, że uzyskaliśmy wartość istotnie mniejszą od początkowej wartości 16,81 uzyskanej podczas losowo wygenerowanego planu podróży. Udało nam się uzyskać tak znacznie lepszy wynik, wykorzystując bardzo prosty i zarazem intuicyjny algorytm. Jakość działania tego algorytmu jest

dobrze widoczna na rysunku 6,3, jest na nim znacznie mniej ścieżek zaczynających się po jednej stronie i kończących po drugiej, a dodatkowo jest bardzo dużo krótkich podróży między miastami leżącymi blisko siebie.

Poszukiwanie dalszych ulepszeń

Przyglądając się dokładniej rysunkom 6.2 i 6.3, możesz wyobrazić sobie kilka konkretnych ulepszeń, jakie można wprowadzić. Możesz spróbować wprowadzić te ulepszenia samodzielnie i za pomocą funkcji `howfar()` sprawdzić, czy rzeczywiście polepszyłeś wynik. Spójrzmy jeszcze raz na nasz początkowy plan podróży:

```
initial_itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,\
                     21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39]
```

Przykładowe ulepszenie, jakiego możemy dokonać, to zamiana miejscami miast o indeksach 6 i 30. Po takiej zamianie otrzymamy nowy plan podróży, w którym te dwie liczby będą zamienione miejscami (w listingu są one oznaczone pogrubioną czcionką):

```
new_itinerary = [0,1,2,3,4,5,30,7,8,9,10,11,12,13,14,15,16,17,18,19,20,\
                 21,22,23,24,25,26,27,28,29,6,31,32,33,34,35,36,37,38,39]
```

Sprawdźmy teraz, czy ta zmiana zmniejszyła całkowity dystans pokonywany przez naszego sprzedawcę:

```
print(howfar(genlines(cities, initial_itinerary)))
print(howfar(genlines(cities, new_itinerary)))
```

Jeśli nasz zmieniony plan podróży `new_itinerary` jest lepszy od początkowego planu `initial_itinerary`, będziemy mogli zapomnieć o początkowym planie podróży i zostawić tylko ten zmieniony. W tym przypadku nasz nowy plan podróży ma długość około 16,79, co jest drobnym polepszeniem oryginalnego wyniku. Po znalezieniu jednego drobnego ulepszenia możemy powtórzyć ten proces ponownie: wybrać dwa miasta, zamienić je miejscami w planie podróży i sprawdzić, czy całkowity dystans zwiększył się, czy zmniejszył. Możemy powtarzać ten proces w nieskończoność, w każdym kroku mając dużą szansę na znalezienie sposobu na obniżenie całkowitego dystansu.

Metoda ta jest na tyle prosta, że możemy od razu przystąpić do implementacji funkcji, która będzie dokonywać takiej zamiany i sprawdzania automatycznie (listing 6.3).

Parametrami wejściowymi funkcji `perturb()` jest lista miast `cities` i plan podróży `itinerary`. Wewnątrz funkcji definiujemy dwie zmienne: `neighborids1` i `neighborids2`, które są losowo wybranymi liczbami naturalnymi z przedziału między 0 a długością planu podróży. W kolejnym kroku funkcja tworzy nowy plan podróży o nazwie `itinerary2`, który jest kopią oryginalnego planu, ale z tą różnicą,

że miasta `neighborids1` i `neighborids2` są w nim zamienione miejscami. Następnie w funkcji liczony jest całkowity dystans do pokonania w obu planach podróży. Jeśli dystans odpowiadający oryginalnemu planowi podróży jest mniejszy, wynikiem działania funkcji jest oryginalny plan podróży. W przeciwnym razie wynikiem działania funkcji jest nowy plan podróży. Wysyłając do tej funkcji plan podróży, zawsze dostajemy plan podróży taki sam lub lepszy. Funkcja ta zmienia odrobinę (dokonuje perturbacji — ang. *perturb*) istniejący plan podróży, starając się go ulepszyć.

Listing 6.3. Funkcja dokonująca drobnej zmiany w planie podróży, porównująca długości tras i zwracająca ten plan podróży, w którym długość trasy jest mniejsza

```
def perturb(cities, itinerary):
    neighborids1 = math.floor(np.random.rand() * (len(itinerary)))
    neighborids2 = math.floor(np.random.rand() * (len(itinerary)))

    itinerary2 = itinerary.copy()

    itinerary2[neighborids1] = itinerary[neighborids2]
    itinerary2[neighborids2] = itinerary[neighborids1]

    distance1 = howfar(genlines(cities, itinerary))
    distance2 = howfar(genlines(cities, itinerary2))

    itinerarytoreturn = itinerary.copy()

    if(distance1 > distance2):
        itinerarytoreturn = itinerary2.copy()

        return(itinerarytoreturn.copy())
```

Możemy teraz wywołać tę funkcję wielokrotnie. Spróbujmy dokonać 2 milionów takich drobnych zmian na naszym losowym planie podróży i zobaczmy, czy uda nam się uzyskać plan o najmniejszym z możliwych dystansie:

```
itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20, \
             21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39]

np.random.seed(random_seed)
itinerary_ps = itinerary.copy()
for n in range(0, len(itinerary) * 50000):
    itinerary_ps = perturb(cities, itinerary_ps)

print(howfar(genlines(cities, itinerary_ps)))
```

Napisany przed chwilą algorytm moglibyśmy nazwać *algorytmem wyszukującym perturbacyjnie*. Algorytm ten przeszukuje tysiące możliwych planów podróży w nadziei znalezienia tego najlepszego, analogicznie jak w przypadku metody *brute force*. Nasz nowy algorytm jest jednak lepszy od metody *brute force*, ponieważ nie sprawdza wszystkich możliwości na oślep, a w zamian dokonuje drobnych

zmian w planie podróży, uzyskując ciąg, w którym całkowity pokonywany dystans monotonicznie maleje. Sugeruje to, że algorytm ten powinien osiągnąć dobre rozwiązanie szybciej niż metoda *brute force*. Wprowadzając do tego algorytmu kilka drobnych ulepszeń, uzyskamy zwieńczenie tej części rozdziału, czyli algorytm o nazwie symulowane wyżarzanie.

Zanim jednak przejdziemy do kodu symulowanego wyżarzania, omówimy, jakie są jego zalety w porównaniu z już omówionymi algorytmami. Wprowadzimy również funkcję temperatury, pozwalającą nam odzwierciedlić w Pythonie właściwości symulowanego wyżarzania.

Algorytmy zachłanne

Algorytm najbliższego sąsiada i algorytm wyszukujący perturbacyjnie, które omówiliśmy powyżej, należą do grupy algorytmów zwanych *algorytmami zachłannymi*. Algorytmy zachłanne w każdym kroku podejmują decyzję, która jest lokalnie optymalna. Jest ona najlepsza, jeśli weźmiemy pod uwagę ten konkretny krok, natomiast nie musi być najlepsza z globalnego punktu widzenia. W przypadku algorytmu najbliższego sąsiada w każdym kroku szukamy miasta, które leży najbliżej miasta, w którym się aktualnie znajdujemy, nie zwracając uwagi na pozostałe miasta. Przejazd do miasta leżącego najbliżej jest optymalny lokalnie, ponieważ minimalizuje odległość, jaką trzeba pokonać w najbliższym kroku. Algorytm nie bierze jednak pod uwagę wszystkich miast równocześnie, co może powodować, że ostateczne rozwiązanie nie jest optymalne globalnie — po odwiedzeniu większości miast te, które pozostały, mogą leżeć bardzo daleko od siebie i przejazd między nimi może znacznie zwiększać całkowity koszt przejazdu między wszystkimi miastami.

Zachłanność algorytmów odnosi się do tej właśnie krótkowzroczności, powodującej podejmowanie decyzji z uwzględnieniem tylko tych bardzo lokalnych informacji. Możemy spojrzeć na tę zachłanność przy rozwiązywaniu problemu optymalizacyjnego jak na problem znalezienia najwyższego punktu na rozległym pagórkowatym terenie, którego „wysokie" punkty odpowiadają lepszym, bardziej optymalnym rozwiązaniom (w odniesieniu do problemu komiwojażera byłyby to plany podróży z krótkimi całkowitymi dystansami), a „niskie" punkty odpowiadają gorszym, nieoptymalnym rozwiązaniom (w odniesieniu do problemu komiwojażera byłyby to plany podróży z długimi dystansami). Zachłanny algorytm poszukujący najwyższego punktu na rozważanym terenie sugerowałby zawsze kierowanie się ku górze, takie działanie może jednak zaprowadzić nas na szczyt jednego z mniejszych pagórków, a nie na szczyt góry, która na tym terenie jest najwyższa. Czasem lepiej jest na chwilę zejść na niższy poziom, żeby w konsekwencji wspiąć się na lepszą, wyższą górę. Ponieważ algorytmy zachłanne kierują się zawsze lokalnym ulepszeniem, nie będą nigdy w stanie zejść z małego pagórka i spowodują, że utkniemy w ekstremum lokalnym. Jest to dokładnie ten sam problem, z którym zetknęliśmy się w rozdziale 3.

Mając to na uwadze, możemy wprowadzić koncepcję, która pozwoli nam rozwiązać problem ekstremum lokalnego spowodowany przez zachłanność naszych algorytmów. Pomysł polega na tym, żeby porzucić naiwne przywiązanie do

podążania w górę za wszelką cenę. W przypadku problemu komiwojażera sprowadzać się to będzie do dokonania perturbacji powodującej gorszy plan podróży, który jednak pozwoli nam dotrzeć do jeszcze lepszych planów podróży, podobnie jak zejście z małego pagórka daje szansę na wspięcie się na inną, większą górę. Innymi słowy, w celu osiągnięcia lepszego rozwiązania musimy zgodzić się na chwilowe pogorszenie.

Funkcja temperatury

Chwilowe pogarszanie z zamiarem osiągnięcia lepszego rezultatu jest delikatną procedurą. Jeśli podejdziemy do pogarszania ze zbytnim oddaniem, możemy schodzić niżej z każdym krokiem, aż dotrzemy do punktu najniższego na danym terenie, zamiast osiągnąć najwyższy. Musimy znaleźć sposób, aby pogarszać rozwiązanie tylko odrobinę i tylko czasami, a co ważniejsze, tylko w kontekście uczenia się, jak otrzymać na koniec lepsze rozwiązanie.

Wyobraź sobie, że znajdujesz się na rozległym pagórkowatym terenie. Zaczynasz późnym popołudniem i wiesz, że masz tylko dwie godziny, żeby znaleźć najwyższy punkt znajdujący się gdzieś na tym obszarze. Przypuśćmy, że nie masz zegarka, który umożliwiałby Ci śledzenie upływu czasu. Wiesz jednak, że wraz z nadchodzącym zmrokiem powietrze będzie się stopniowo ochładzać. Podejmujesz zatem decyzję, aby używać temperatury jako sposobu na szacowanie, ile czasu zostało Ci jeszcze na poszukiwania.

Na początku tych dwóch godzin, gdy jest jeszcze względnie ciepło, będziesz zapewne otwarty na swobodną eksplorację otaczającego Cię terenu. Mając dużo czasu, możesz pozwolić sobie na ryzyko podróżowania na niższe obszary. Dzięki tej eksploracji lepiej zrozumiesz otaczający Cię teren i odwiedzisz wiele nowych miejsc. Gdy zrobi się zimniej i będziesz zbliżał się do końca dwóch godzin przeznaczonych na poszukiwania, będziesz zapewne mniej otwarty na tak szerokie poszukiwania. Będziesz chciał ograniczyć i ukierunkować swoje działania i będziesz wtedy mniej skłonny schodzić z pagórka, na którym w danym momencie będziesz.

Poświęć chwilę, aby przemyśleć przedstawioną tutaj taktykę, i zastanów się, dlaczego jest ona najlepszym sposobem na znalezienie najwyższego punktu. Omówiliśmy już, dlaczego czasem chcemy zejść niżej: chcemy w ten sposób zapobiec problemowi maksimum lokalnego i zatrzymania się na szczycie pagórka znajdującego się u stóp większej góry. Kiedy jednak powinniśmy zrezygnować ze schodzenia w dół? Weź pod uwagę ostatnie 10 sekund Twojego dwugodzinnego czasu poszukiwań. Nieważne, gdzie wtedy jesteś, powinieneś iść tylko ku górze. W tym momencie nie ma już sensu szukać innych pagórków czy większych gór. W czasie ostatnich 10 sekund powinieneś już tylko wdrapywać się na tę górę, na której właśnie jesteś. Schodząc w dół i nawet przechodząc na wyższą górę, nie miałbyś już czasu, aby wejść na jej szczyt, więc rozwiązanie mogłoby nie być nawet lokalnie najlepsze. Dlatego w ostatnich sekundach poszukiwań powinniśmy iść już tylko w górę i nie rozważać wcale tymczasowego schodzenia w dół.

Z drugiej strony, rozważając pierwsze 10 sekund poszukiwań, nie musisz się spieszyć i wspinać od razu w górę. Na początku warto poświęcić czas na poznanie otaczającego terenu. Jeśli popełnisz błąd w tych pierwszych sekundach poszukiwań, będziesz miał mnóstwo czasu, by ten błąd naprawić, a wiedza zdobyta podczas tej początkowej eksploracji zaowocuje w późniejszym momencie. Podczas tych pierwszych 10 sekund warto być otwartym na wycieczki w dół i nie upierać się przy ciągłym wchodzeniu w górę.

Rozważając dłuższe przedziały czasowe niż 10 sekund, będziesz zapewne stosować tę samą strategię. Będąc 10 minut przed końcem poszukiwań, będziesz miał bardziej umiarkowane nastawienie niż 10 sekund przed końcem czasu. Dwugodzinny przedział prawie się skończył i będziesz chciał iść już tylko w górę. 10 minut jest jednak dłuższym okresem niż 10 sekund, dlatego będziesz miał w sobie odrobinę chęci, żeby zejść trochę w dół, by móc sprawdzić, czy przez przypadek nie odkryjesz innego, bardziej obiecującego pagórka. Z tego samego powodu 10 minut po rozpoczęciu poszukiwań będziesz miał bardziej umiarkowane nastawienie niż 10 sekund po rozpoczęciu poszukiwań. Cały dwugodzinny przedział będzie charakteryzował się stopniową zmianą nastawienia: od początkowej gotowości, by czasem schodzić niżej, aż do rosnącej ochoty, by wspinać się tylko w górę.

W celu zamodelowania tego scenariusza w Pythonie zdefiniujemy funkcję odzwierciedlającą zmiany temperatury. Zaczniemy od wysokiej temperatury odpowiadającej gotowości do eksploracji i schodzenia na niższe tereny, a skończmy na niskiej temperaturze odpowiadającej brakowi gotowości do schodzenia w dół. Nasza funkcja temperatury jest względnie prosta, a jako parametr wejściowy przyjmuje zmienną `t` oznaczającą upływający czas:

```
temperature = lambda t: 1/(t + 1)
```

Możemy teraz wygenerować wykres temperatury, uruchamiając poniższy kod Pythona:

```
import matplotlib.pyplot as plt
ts = list(range(0,100))
plt.plot(ts, [temperature(t) for t in ts])
plt.title('Funkcja temperatury')
plt.xlabel('Czas')
plt.ylabel('Temperatura')
plt.show()
```

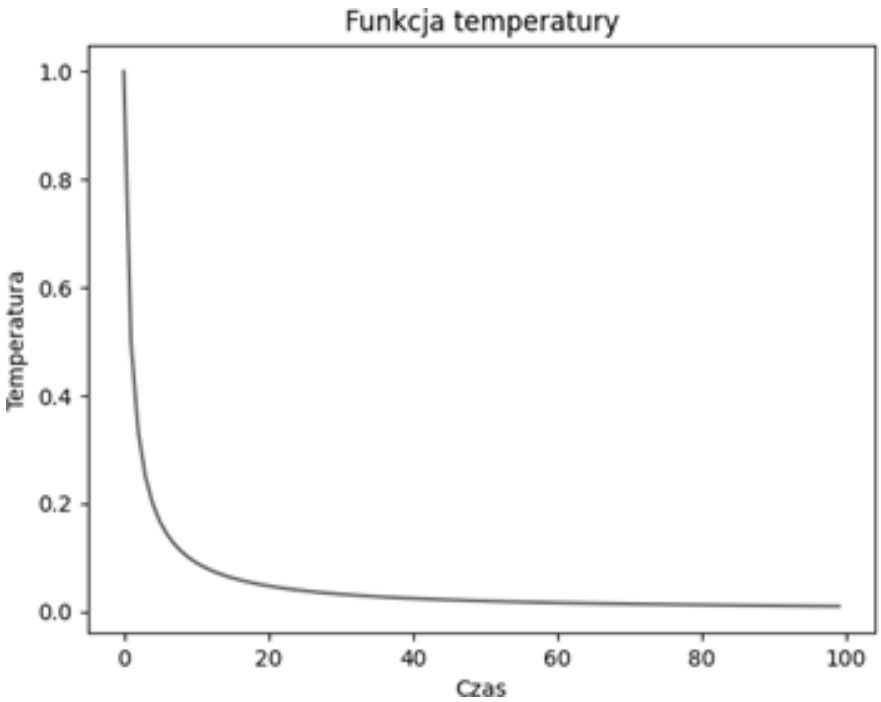
Powyższy kod zaczyna się od zaimportowania modułu `matplotlib`. Następnie definiujemy zmienną `ts` jako przedział od 1 do 100. Później rysujemy wykres temperatury dla każdego `t` z przedziału `ts`. Ponownie nie zwracamy tu uwagi na jednostki czy dokładną skalę, gdyż jest to hipotetyczna sytuacja, mająca jedynie pokazać ogólny kształt funkcji związanej ze stygnięciem. Dlatego używamy po prostu 0 do oznaczenia najniższej temperatury, 1 do oznaczenia najwyższej

temperatury, 0 do oznaczenia początku czasu i 99 do oznaczenia końca przedziału czasowego, bez uwzględniania konkretnych jednostek.

Wykres uzyskany za pomocą tego kodu przedstawiony jest na rysunku 6.4.

Rysunek ten pokazuje, jak zmienia się temperatura podczas naszej hipotetycznej optymalizacji. Tak zdefiniowana temperatura będzie odpowiadać za zarządzanie naszą optymalizacją: nasza gotowość do schodzenia w dół będzie proporcjonalna do aktualnej temperatury.

Mamy teraz wszystkie składniki potrzebne, żeby zaimplementować algorytm symulowanego wyżarzania. Nie zatrzymuj się tutaj — nie rozmyślaj zbyt długo, tylko szybko zanurkuj do następnej części tego rozdziału.



Rysunek 6.4. Spadek temperatury w miarę upływu czasu

Rysunek ten pokazuje, jak zmienia się temperatura podczas naszej hipotetycznej optymalizacji. Tak zdefiniowana temperatura będzie odpowiadać za zarządzanie naszą optymalizacją: nasza gotowość do schodzenia w dół będzie proporcjonalna do aktualnej temperatury.

Mamy teraz wszystkie składniki potrzebne, żeby zaimplementować algorytm symulowanego wyżarzania. Nie zatrzymuj się tutaj — nie rozmyślaj zbyt długo, tylko szybko zanurkuj do następnej części tego rozdziału.

Symulowane wyżarzanie

Połączmy teraz wszystkie omawiane w tym rozdziale koncepcje: funkcję temperatury, poszukiwanie najwyższego punktu na pagórkowatym terenie, algorytm wyszukujący perturbacyjnie i problem komiwojażera. W kontekście problemu komiwojażera nasz pagórkowaty obszar to wszystkie możliwe wersje planu podróży. Możemy sobie wyobrazić, że lepszy plan podróży odpowiada wyższym punktom na naszym obszarze, a gorsze plany podróży — niższym. Stosując funkcję `perturb()`, przenosimy się do kolejnego punktu, mając nadzieję, że będzie on najwyżej, jak to możliwe.

Do kierowania naszą eksploracją używać będziemy funkcji temperatury. Na początku wysoka temperatura będzie narzucać dużą otwartość na wybieranie gorszych planów podróży. Pod koniec poszukiwań będziemy mniej otwarci na pogarszanie się planów podróży i będziemy bardziej skupieni na zachłannej optymalizacji.

Tak skonstruowany algorytm nosi nazwę **symulowanego wyżarzania** (ang. SA — *simulated annealing*). Jest on bardzo podobny do znanego Ci już algorytmu wyszukującego perturbacyjnie. Główną różnicą między symulowanym wyżarzaniem a algorytmami perturbacyjnymi jest to, że symulowane wyżarzanie czasem akceptuje pogorszenie wyniku. Dzięki tej właściwości symulowane wyżarzanie ma szansę być odporne na problem ekstremum lokalnego. Otwartość symulowanego wyżarzania na akceptowanie gorszego wyniku zależy od aktualnej temperatury.

Zmodyfikujmy zatem naszą funkcję `perturb()` o wspomnianą zmianę. Dodamy nowy argument wejściowy o nazwie `time` reprezentujący czas. Argument ten będzie informował funkcję, jak długo wykonujemy nasze poszukiwania. Przy pierwszym wywołaniu funkcji `perturb()` zaczniemy z czasem o wartości 1, przy kolejnym użyjemy wartości 2, następnie 3 i tak dalej. Dodamy również fragment kodu obliczający aktualną temperaturę i wybierający liczbę losową. Jeśli wylosowana liczba będzie niższa niż aktualna temperatura, będziemy otwarci na zmianę w kierunku gorszego wyniku. Jeśli wylosowana liczba będzie wyższa niż aktualna temperatura, nie będziemy akceptować gorszego planu podróży. W ten sposób od czasu do czasu będziemy akceptować gorsze plany podróży, a prawdopodobieństwo zaakceptowania zmiany na gorsze będzie malało z czasem, zgodnie ze spadkiem temperatury. Nasza nowa funkcja przedstawiona jest w listingu 6.4, a dodane fragmenty oznaczone zostały pogrubieniem.

Listing 6.4. Zmieniona wersja funkcji perturb() uwzględniająca temperaturę i losowo wybieraną liczbę

```
def perturb_sa1(cities, itinerary, time):
    neighborids1 = math.floor(np.random.rand() * (len(itinerary)))
    neighborids2 = math.floor(np.random.rand() * (len(itinerary)))

    itinerary2 = itinerary.copy()

    itinerary2[neighborids1] = itinerary[neighborids2]
    itinerary2[neighborids2] = itinerary[neighborids1]
```

```
    distance1 = howfar(genlines(cities,itinerary))
    distance2 = howfar(genlines(cities,itinerary2))

    itinerarytoreturn = itinerary.copy()

    randomdraw = np.random.rand()
    temperature = 1/((time/1000) + 1)


    if ((distance2 > distance1 and (randomdraw) < (temperature)) or (distance1 >
    ↪distance2)):
        itinerarytoreturn = itinerary2.copy()

        return(itinerarytoreturn.copy())
```

Dzięki dodaniu tych dwóch linii kodu, nowego parametru wejściowego i warunku `if` otrzymaliśmy prosty algorytm symulowanego wyżarzania. Jak widzisz, użyliśmy tutaj zmodyfikowanej funkcji temperatury — dzielimy tu czas przez 1000, ponieważ w trakcie działania algorytmu zmienna `time` będzie przyjmowała bardzo duże wartości. Możemy teraz porównać działanie algorytmu symulowanego wyżarzania z algorytmem wyszukującym perturbacyjnie i algorytmem najbliższego sąsiada:

```
itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20, \
             21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39]
np.random.seed(random_seed)

itinerary_sa = itinerary.copy()
for n in range(0, len(itinerary) * 50000):
    itinerary_sa = perturb_sa1(cities, itinerary_sa, n)

print(howfar(genlines(cities, itinerary)))         # losowy plan podróży
print(howfar(genlines(cities, itinerary_ps)))      # wyszukiwanie perturbacyjnie
print(howfar(genlines(cities, itinerary_sa)))      # symulowane wyżarzanie
print(howfar(genlines(cities, donn(cities, N))))   # algorytm najbliższego sąsiada
```

Gratulacje! Umiesz przeprowadzić symulowane wyżarzanie. Jak widzisz, dystans pokonywany przez komiwojażera przy użyciu losowego planu podróży wynosi 16,81, a w przypadku algorytmu najbliższego sąsiada dystans wynosi zaledwie 6,29. Gdy użyjemy algorytmu wyszukującego perturbacyjnie, dystans wyniesie 7,38, a przy użyciu symulowanego wyżarzania otrzymujemy tylko 5,92. W naszym przypadku algorytm wyszukujący perturbacyjnie daje lepsze wyniki niż losowe wybranie kolejności miast w planie podróży, algorytm najbliższego sąsiada działa lepiej niż algorytm wyszukujący perturbacyjnie, a algorytm symulowanego wyżarzania daje najlepsze rezultaty z wszystkich rozważanych algorytmów. Możesz sprawdzić działanie tych algorytmów przy użyciu innego ziarna. W niektórych przypadkach algorytm symulowanego wyżarzania będzie działał gorzej niż algorytm najbliższego sąsiada. Wynika to z delikatności procesu wykorzystywanego

podczas symulowanego wyżarzania. Żeby algorytm ten działał prawidłowo i niezawodnie, wymagane jest zaistnienie równocześnie wielu elementów i musi on być precyzyjnie dostrojony do danego problemu. Natomiast dobrze dostrojony algorytm symulowanego wyżarzania będzie zawsze dawał lepsze wyniki od prostszych od niego algorytmów zachłannych. Na tym właśnie skupimy się w dalszej części tego rozdziału — omówimy szczegóły związane ze symulowanym wyżarzaniem oraz poznamy metody modyfikacji pozwalające uzyskać najlepszą wydajność tego typu algorytmów. Gdy wykonamy te ulepszenia i dostroimy nasz algorytm, będzie on dawał znacznie lepsze wyniki niż prostsze od niego algorytmy zachłanne. Pozostała część tego rozdziału poświęcona będzie szczegółom symulowanego wyżarzania i metodom strojenia pozwalającym uzyskać najlepsze wyniki.

METAHEURYSTYKI BAZUJĄCE NA METAFORACH

Algorytmy symulowanego wyżarzania są dość osobliwe. Łatwiej jest je zrozumieć, jeśli pozna się źródło ich pochodzenia. Wyżarzanie jest procesem występującym w metalurgii. Metoda ta polega na nagrzaniu materiału do wysokiej temperatury i następnie powolnym studzeniu go. Gdy metal jest gorący, wiązania między cząsteczkami są osłabione, a nawet pozrywane. Podczas studzenia powstają nowe wiązania, dzięki którym metal uzyskuje nowe, bardziej pożądane właściwości. Symulowane wyżarzanie przypomina wyżarzanie stosowane w metalurgii w tym sensie, że gdy temperatura jest wysoka, „osłabiamy" swoje poszukiwania, akceptując gorsze rozwiązania, w nadziei, że gdy temperatura opadnie, będziemy w stanie uzyskać rozwiązania lepsze niż wcześniej.

Metafora, która daje początek symulowanemu wyżarzaniu, wydaje się nieco sztuczna i dla osób niezajmujących się metalurgią jest zwykle mało intuicyjna. Istnieje wiele innych metaheurystyk bazujących na metaforach, część jest bardziej wyszukana, a część bardzo intuicyjna. Wszystkie takie metaheurystyki, które adaptują proces istniejący w naturze lub proces stworzony przez człowieka w sposób pozwalający rozwiązywać problemy optymalizacyjne, nazywamy *metaheurystykami bazującymi na metaforach*. Przykładem takich algorytmów są: algorytm mrówkowy (ang. *ant colony optimization*), algorytm kukułki (ang. *cuckoo search*), algorytm optymalizacji mątwy (ang. *cuttlefish optimization*), optymalizacja roju kotów (ang. *cat swarm optimization*), algorytm żabich skoków (ang. *shuffled frog leaping*), algorytm przeszukiwania harmonicznego (ang. *harmony search*) zainspirowany procesem improwizacji jazzowej i algorytm wody deszczowej (ang. *rain water algorithm*). Część z tych przenośni jest sztuczna i niezbyt przydatna, jednak czasem wiedza o tej metaforze inspiruje i pozwala uzyskać lepsze zrozumienie problemu. Każdy z tych algorytmów, bez względu na to, czy intuicyjny, czy nie, jest zwykle bardzo interesujący i implementacja ich w Pythonie byłaby dobrą zabawą.

Strojenie naszego algorytmu

Jak już wiesz, symulowane wyżarzanie jest procesem bardzo delikatnym. Kod z listingu 6.4 pokazuje podstawową wersję tego algorytmu, w której będziemy wprowadzać drobne zmiany. Proces wprowadzania takich drobnych zmian w celu ulepszenia algorytmu nazywa się *strojeniem* algorytmu. Prawidłowe strojenie algorytmu może zrobić ogromną różnicę w przypadku skomplikowanych problemów optymalizacyjnych.

Wykorzystywana przez nas funkcja `perturb()` dokonywała drobnych zmian planu podróży: zamieniała miejscami dwa miasta. Jest to tylko jeden z możliwych sposobów na wprowadzanie drobnych zmian w planie podróży. Trudno jest przewidzieć, która z metod wprowadzania drobnych zmian zadziała najlepiej. Dlatego przetestujemy kilka z nich.

Jedna z metod wprowadzania zmian w planie podróży to odwrócenie jakiejś części tego planu: weźmiemy podzbiór miast i odwiedzimy je w odwrotnej kolejności. W Pythonie takie odwrócenie można zapisać za pomocą jednej linii kodu. Zaczniemy od wybrania dwóch miast w naszym planie podróży. Mniejszy z indeksów oznaczymy jako `small`, a większy jako `big`. Następnie odwrócimy kolejność miast znajdujących się pomiędzy tymi dwoma wybranymi miastami:

```
small = 10
big = 20
itinerary = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20, \
             21,22,23,24,25,26,27,28,29, 30,31,32,33,34,35,36,37,38,39]
itinerary[small:big] = itinerary[small:big][::-1]
print(itinerary)
```

Gdy uruchomisz ten fragment kodu, zobaczysz, że miasta od 10 do 19 pojawiły się w odwrotnej kolejności:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
```

Innym sposobem wprowadzania zmian w planie podróży jest wycięcie wybranego fragmentu z miejsca, w którym się znajduje, i wstawienie go w inne miejsce w planie podróży. Możemy to zrobić na przykład w następujący sposób:

```
itinerary = [0,1,2,3,4,5,6,7,8,9]
```

przeniesiemy teraz cały fragment `[1,2,3,4]` na dalsze miejsce w planie podróży, tworząc następujący, nowy plan podróży:

```
itinerary = [0,5,6,7,8,1,2,3,4,9]
```

Powyższe zadanie wycinania i przenoszenia w inne miejsce wykonuje poniższy fragment kodu Pythona:

```
small = 1
big = 5
itinerary = [0,1,2,3,4,5,6,7,8,9]
tempitin = itinerary[small:big]
del(itinerary[small:big])
np.random.seed(random_seed + 1)
neighborids3 = math.floor(np.random.rand() * (len(itinerary)))
```

```
for j in range(0, len(tempitin)):
    itinerary.insert(neighborids3 + j, tempitin[j])
```

Możemy teraz zaktualizować naszą funkcję `perturb()` tak, aby losowo wybierała jedną z tych trzech metod wprowadzania drobnych zmian w planie podróży. Tego losowego wyboru dokonamy za pomocą wybrania losowej liczby z przedziału od 0 do 1. Jeśli wylosowana liczba będzie leżała w przedziale 0 – 0,45, to zastosowana zostanie metoda odwracania kolejności miast w wybranej części planu podróży. Jeśli wylosowana liczba będzie leżała w przedziale 0,45 – 0,55, to zastosowana zostanie metoda zamiany miejscami dwóch miast. Jeśli wylosowana liczba będzie w przedziale 0,55 – 1, to zastosowana zostanie metoda przenoszenia wybranej części w inne miejsce w planie podróży. W ten sposób nasza funkcja będzie wykorzystywać wszystkie trzy metody, losowo decydując, która metoda zostanie użyta w danym momencie. Ten losowy wybór metody i nowe sposoby wprowadzania drobnych zmian zostały uwzględnione w funkcji `perturb_sa2()` (listing 6.5).

Listing 6.5. Wykorzystanie kilku metod wprowadzania zmian w planie podróży

```
def perturb_sa2(cities, itinerary, time):
    neighborids1 = math.floor(np.random.rand() * (len(itinerary)))
    neighborids2 = math.floor(np.random.rand() * (len(itinerary)))

    itinerary2 = itinerary.copy()

    randomdraw2 = np.random.rand()
    small = min(neighborids1, neighborids2)
    big = max(neighborids1, neighborids2)
    if (randomdraw2 >= 0.55):
        itinerary2[small:big] = itinerary2[small:big][::-1]
    elif (randomdraw2 < 0.45):
        tempitin = itinerary[small:big]
        del(itinerary2[small:big])
        neighborids3 = math.floor(np.random.rand() * (len(itinerary)))
        for j in range(0, len(tempitin)):
            itinerary2.insert(neighborids3 + j, tempitin[j])
    else:
        itinerary2[neighborids1] = itinerary[neighborids2]
        itinerary2[neighborids2] = itinerary[neighborids1]

    distance1 = howfar(genlines(cities, itinerary))
    distance2 = howfar(genlines(cities, itinerary2))

    itinerarytoreturn = itinerary.copy()

    randomdraw = np.random.rand()
    temperature = 1/((time/1000) + 1)

    if ((distance2 > distance1 and (randomdraw) < (temperature)) or (distance1 >
    distance2)):
        itinerarytoreturn = itinerary2.copy()
```

```
return(itinerarytoreturn.copy())
```

Nasza funkcja `perturb()` jest teraz bardziej skomplikowana i zarazem bardziej elastyczna: może wykonywać kilka różnych rodzajów zmian w planie podróży, bazując na losowym wyborze. Nie zawsze warto dążyć do tego, aby algorytm był bardziej elastyczny, a zwiększanie skomplikowania zwykle jest tym, czego staramy się unikać. W niektórych jednak przypadkach warto dodać elastyczność lub większe skomplikowanie. Ocena, czy w danym przypadku jest to dobrą drogą, zależy od tego, czy dodanie elastyczności lub zwiększenie skomplikowania poprawi wydajność danego algorytmu. Na tym właśnie polega strojenie algorytmu: podobnie jak podczas strojenia instrumentu muzycznego nie wiesz z góry, jak bardzo naciągnięta będzie musiała być struna — musisz trochę ją napiąć lub trochę poluzować, posłuchać, jak brzmi po tej zmianie, i dopasować następne działania do tego, co słyszysz. Zatem żeby dowiedzieć się, czy zmiany dodane do naszej funkcji (oznaczone na listingu 6.5 pogrubieniem) poprawiły działanie algorytmu, musimy przetestować działanie naszego zmienionego algorytmu i porównać uzyskane wyniki z wcześniejszą wersją kodu.

Unikanie zbyt dużego pogarszania

Istotą algorytmu symulowanego wyżarzania jest to, że musimy zgodzić się na pogorszenie, żeby w ostatecznym rozrachunku otrzymać lepsze rozwiązanie. Nie chcemy jednak dopuścić, żeby zmiany powodowały *zbyt duże* pogarszanie rozwiązań. Aktualnie nasza funkcja `perturb()` zaakceptuje gorsze rozwiązania, jeśli losowa wartość jest niższa niż aktualna temperatura. Decyzja ta zapada w poniższej linii kodu (nie jest ona przeznaczona do uruchamiania jako samodzielny fragment kodu):

```
if ((distance2 > distance1 and randomdraw < temperature) or (distance1 >
distance2)):
```

Zmienimy ten warunek, aby nasza skłonność do zaakceptowania gorszego planu podróży zależała nie tylko od temperatury, ale również od tego, jak duża zmiana na gorsze mogłaby się dokonać w naszym planie podróży. Będziemy bardziej skłonni akceptować niewielkie pogorszenia niż zmiany na dużo gorszy wynik. Wprowadzimy więc do powyższego warunku miarę mówiącą o stopniu pogorszenia naszego planu podróży. Zmiana ta pokazana jest w poniższym fragmencie kodu (który również nie jest przeznaczony do uruchamiania w oderwaniu od reszty kodu):

```
scale = 3.5
if ((distance2 > distance1 and (randomdraw) < (math.exp(scale*(distance1 -
distance2)) * temperature)) or (distance1 > distance2)):
```

Po wprowadzeniu tego warunku do kodu funkcji początek funkcji `perturb()` nie zmieni się, natomiast jej końcówka będzie miała poniższą postać:

```
--fragment pominięty--
# tu znajduje się początek funkcji perturb()
    scale = 3.5
if ((distance2 > distance1 and (randomdraw) < (math.exp(scale*(distance1 -
distance2)) * temperature)) or (distance1 > distance2)):
    itinerarytoreturn = itinerary2.copy()
return(itinerarytoreturn.copy())
```

Umożliwienie powrotu do wcześniejszego rozwiązania

Podczas symulowanego wyżarzania możemy nieświadomie zaakceptować zmianę, która będzie wyraźnie gorsza od uzyskiwanych wcześniej rozwiązań. Warto w związku z tym monitorować, jakie było najlepsze z uzyskanych do tej pory rozwiązań. Będziemy wtedy mogli dodać do algorytmu opcję powrotu do tego najlepszego z uzyskanych dotychczas rozwiązań. Listing 6.6 zawiera kod uwzględniający takie powracanie do najlepszego rozwiązania (dodane fragmenty zostały oznaczone pogrubioną czcionką).

Listing 6.6. Pełna forma symulowanego wyżarzania zwracająca optymalny plan podróży

```
def perturb_sa3(cities, itinerary, time, maxitin):
    neighborids1 = math.floor(np.random.rand() * (len(itinerary)))
    neighborids2 = math.floor(np.random.rand() * (len(itinerary)))
    global mindistance
    global minitinerary
    global minidx
    itinerary2 = itinerary.copy()
    randomdraw = np.random.rand()

    randomdraw2 = np.random.rand()
    small = min(neighborids1, neighborids2)
    big = max(neighborids1, neighborids2)
    if (randomdraw2 >= 0.55):
        itinerary2[small:big] = itinerary2[small:big][::-1]
    elif (randomdraw2 < 0.45):
        tempitin = itinerary[small:big]
        del(itinerary2[small:big])
        neighborids3 = math.floor(np.random.rand() * (len(itinerary)))
        for j in range(0, len(tempitin)):
            itinerary2.insert(neighborids3 + j, tempitin[j])
    else:
        itinerary2[neighborids1] = itinerary[neighborids2]
        itinerary2[neighborids2] = itinerary[neighborids1]

    temperature = 1/(time/(maxitin/10)+1)

    distance1 = howfar(genlines(cities, itinerary))
    distance2 = howfar(genlines(cities, itinerary2))

    itinerarytoreturn = itinerary.copy()
```

```
scale = 3.5
if ((distance2 > distance1 and (randomdraw) < (math.exp(scale*(distance1 –
distance2)) * temperature)) or (distance1 > distance2)):
    itinerarytoreturn = itinerary2.copy()

reset = True
resetthresh = 0.04
if (reset and (time - minidx) > (maxitin * resetthresh)):
    itinerarytoreturn = minitinerary
    minidx = time

if (howfar(genlines(cities, itinerarytoreturn)) < mindistance):
    mindistance = howfar(genlines(cities, itinerary2))
    minitinerary = itinerarytoreturn
    minidx = time

if (abs(time - maxitin) <= 1):
    itinerarytoreturn = minitinerary.copy()

return(itinerarytoreturn.copy())
```

Zdefiniowaliśmy tutaj zmienne globalne odpowiadające najkrótszemu dystansowi, jaki udało się do tej pory uzyskać, planowi podróży, który odpowiada temu dystansowi, i czasowi, w jakim udało się osiągnąć to rozwiązanie. Jeśli upływa długi przedział czasu i nie udaje się znaleźć rozwiązania lepszego od zapisanego rozwiązania, które odpowiada minimalnemu dystansowi, to możemy uznać, że wszystkie poczynione przez ten czas zmiany były niekorzystne, i powrócić do tego najlepszego rozwiązania. Przy czym taki powrót do najlepszego ze znalezionych do tej pory rozwiązań będzie wykonywany, tylko jeśli wprowadziliśmy wiele drobnych korekt i żadna z nich nie spowodowała polepszenia rozwiązania w porównaniu z tym właśnie zapamiętanym najlepszym rozwiązaniem. Zmienna `resetthresh` wyznacza, jak długo godzimy się szukać, zanim zdecydujemy się na ewentualny powrót do wcześniej znalezionego najlepszego rozwiązania. Dodaliśmy tu również nowy parametr wejściowy o nazwie `maxitin`. Parametr ten determinuje, jak wiele razy planujemy wywołać naszą funkcję. Dzięki temu argumentowi wiemy, jak daleko zaszliśmy w naszych poszukiwaniach. Używamy tej zmiennej również w naszej funkcji temperatury, dzięki czemu może ona dopasować się do planowanej przez nas długości poszukiwań. Gdy nasz czas się skończy, wynikiem działania funkcji będzie najlepsze z rozwiązań, jakie udało się napotkać w całym czasie działania algorytmu.

Testowanie wydajności

Teraz gdy dokonaliśmy już tylu zmian i poprawek, jesteśmy w stanie napisać funkcję wykonującą symulowane wyżarzanie od początku do końca. Funkcja ta najpierw definiuje zmienne globalne, następnie wywołuje naszą najnowszą wersję funkcji `perturb()`, aby na koniec uzyskać plan podróży z bardzo krótkim dystansem (listing 6.7).

Listing 6.7. Funkcja przeprowadzająca pełne symulowane wyżarzanie i zwracająca optymalny plan podróży

```
def siman(itinerary, cities):
    newitinerary = itinerary.copy()
    global mindistance
    global minitinerary
    global minidx
    mindistance = howfar(genlines(cities, itinerary))
    minitinerary = itinerary
    minidx = 0

    maxitin = len(itinerary) * 50000
    for t in range(0,maxitin):
        newitinerary = perturb_sa3(cities, newitinerary, t, maxitin)

    return(newitinerary.copy())
```

Możemy teraz wywołać funkcję `siman()` i porównać wyniki jej działania z algorytmem najbliższego sąsiada:

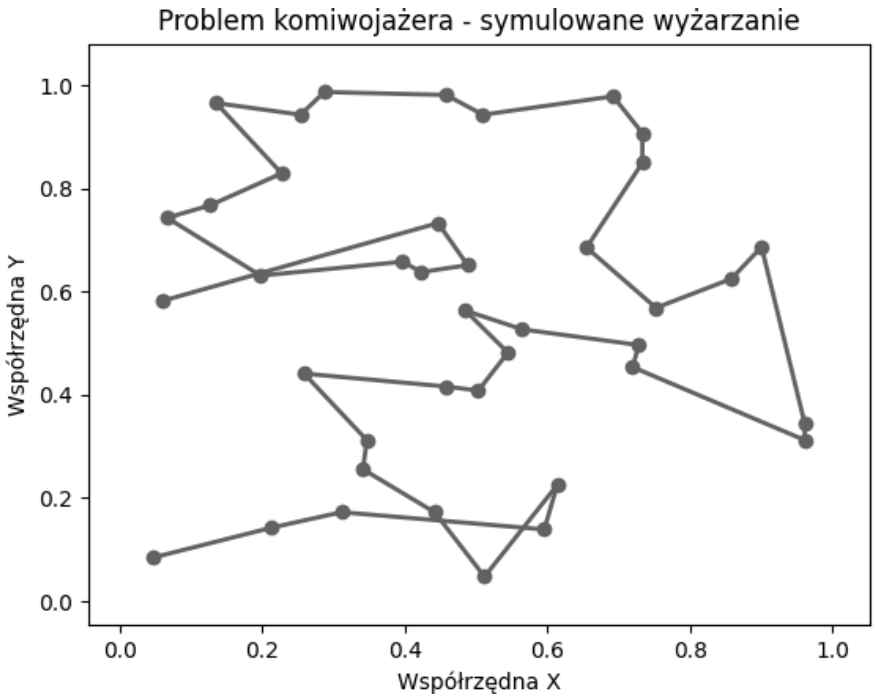
```
np.random.seed(random_seed)
itinerary = list(range(N))
nnitin = donn(cities, N)
nnresult = howfar(genlines(cities, nnitin))
simanitinerary = siman(itinerary, cities)
simanresult = howfar(genlines(cities, simanitinerary))
print(nnresult)
print(simanresult)
print(simanresult/nnresult)
```

Uruchamiając powyższy kod, przekonasz się, że nasza ostateczna wersja symulowanego wyżarzania znajduje plan podróży, którego dystans wynosi 5,32. Udało nam się zatem uzyskać wynik lepszy o 15% w porównaniu z algorytmem najbliższego sąsiada, dla którego dystans wynosił 6,29. Taki wynik może wydawać Ci się mało imponujący: nasze zmagania zajęły pół rozdziału, a osiągnęliśmy wynik lepszy tylko o 15%. Jest to w miarę racjonalne zastrzeżenie i może się okazać, że nigdy nie będziesz potrzebować wydajności lepszej niż ta uzyskiwana przy użyciu algorytmu najbliższego sąsiada. Wyobraź sobie jednak, że spotykasz się z CEO globalnej firmy logistycznej typu UPS czy DHL i składasz mu propozycję obniżenia kosztów przejazdów o 15%. Najprawdopodobniej zobaczyłbyś w jego oczach iskierki oznaczające, że myśli on o miliardach złotych, które udałoby się zaoszczędzić jego firmie. Logistyka jest aktualnie jednym z głównych czynników zwiększających koszty przedsiębiorstw, a dodatkowo powoduje ogromne zanieczyszczenie środowiska, dlatego znajdowanie dobrego rozwiązania problemu komiwojażera zawsze będzie miało dużo zastosowań praktycznych. Poza tym problem komiwojażera jest bardzo ważny z naukowego punktu widzenia, daje on nam punkt odniesienia przy porównywaniu metod optymalizacyjnych i jest dobrym

wstępem przed zagłębieniem się w inne, bardziej zaawansowane zagadnienia teoretyczne.

Wywołując `plotitinerary(cities, simanitinerary, 'Problem komiwojażera - symulowane wyżarzanie', 'rysunek5')`, uzyskasz wykres planu podróży, który jest wynikiem naszego ulepszonego symulowanego wyżarzania. Wykres ten możesz zobaczyć na rysunku 6.5.

Z jednej strony mamy tutaj wykres losowo wygenerowanych punktów i łączących je linii. Z drugiej jednak strony jest to wynik procesu optymalizacji, który przeprowadziliśmy za pomocą ponad setek, a być może nawet tysięcy iteracji, dążąc zawzięcie do uzyskania perfekcyjnego wyniku wśród niemalże nieskończonej liczby możliwości, i pod tym względem jest on wspaniały.



Rysunek 6.5. Efekt końcowy symulowanego wyżarzania

Podsumowanie

W tym rozdziale omówiliśmy problem komiwojażera jako przykład zastosowania zaawansowanych algorytmów optymalizacyjnych. Rozważyliśmy kilka metod rozwiązywania tego problemu, takich jak atak *brute force*, algorytm najbliższego sąsiada i na koniec symulowane wyżarzanie — bardzo wydajną metodę, która dopuszcza pogarszanie rozwiązania, aby w ostatecznym rozrachunku uzyskać

rozwiązanie dużo lepsze. Mam nadzieję, że zapoznając się z tym skomplikowanym zagadnieniem, zdobyłeś narzędzia, które pozwolą Ci lepiej rozwiązywać problemy optymalizacyjne, gdyż zaawansowane metody optymalizacji będą zawsze miały wiele zastosowań praktycznych zarówno w naukach technicznych, jak i w świecie biznesu.

W kolejnym rozdziale skierujemy naszą uwagę w stronę geometrii i omówimy algorytmy umożliwiające tworzenie konstrukcji geometrycznych i dokonywanie przekształceń na płaszczyźnie. Niech nasza podróż przez algorytmy trwa!

Skorowidz


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

ZANURZ SIĘ W ALGORYTMACH: ŁAGODNIE I Z ZACIEKAWIENIEM!

Bez znajomości algorytmów nie można się nauczyć programowania. Algorytmy są też przydatne w pracy naukowców i inżynierów. Właściwie każdy z nas codziennie z nich korzysta: gotując, wypełniając urzędowe formularze czy przeprowadzając mniej lub bardziej złożone procedury. Co więcej, ludzki organizm również wykonuje skomplikowane działania bez udziału świadomości, ale zgodnie z ukrytymi algorytmami. Łatwo się przekonać, że dzięki umiejętności zastosowania odpowiedniego algorytmu albo nawet zaprojektowania nowego można rozwiązać bardzo złożone problemy. Ta książka jest praktycznym wprowadzeniem do algorytmów i ich implementacji w Pythonie. Omówiono w niej wiele najciekawszych algorytmów służących do przeszukiwania, sortowania i optymalizacji. Zaprezentowano także te bazujące na... ludzkiej podświadomości. Nie zabrakło dość zaawansowanych tematów, takich jak algorytmy służące do uczenia maszynowego, przetwarzania języka naturalnego i wdrażania technik sztucznej inteligencji. Omówiono też algorytmy znane od starożytności, służące do mnożenia liczb, obliczania największego wspólnego dzielnika czy generowania kwadratów magicznych. Pokazano ponadto, w jaki sposób zaimplementować poszczególne algorytmy w Pythonie, aby uzyskać możliwie najwyższą wydajność kodu.

Dzięki książce dowiesz się, jak:

- generować i wykorzystywać diagramy Woronoja
- wykorzystywać algorytmy podczas pisania chatbota
- stosować algorytm wyżarzania do optymalizacji globalnej
- tworzyć drzewa decyzyjne
- projektować algorytmy przydatne w pisaniu programów
- mierzyć wydajność i prędkość działania algorytmów

Dr Bradford Tuckfield

jest analitykiem danych, konsultantem i autorem książek. Publikował artykuły z zakresu matematyki, zarządzania i medycyny w wielu renomowanych czasopismach. Kieruje założoną przez siebie firmą konsultingową Kmbara. Zajmował się też kulturą i polityką publiczną.

ISBN 978-83-283-8344-9
9 788328 383449
Cena: 69,00 zł
no starch press