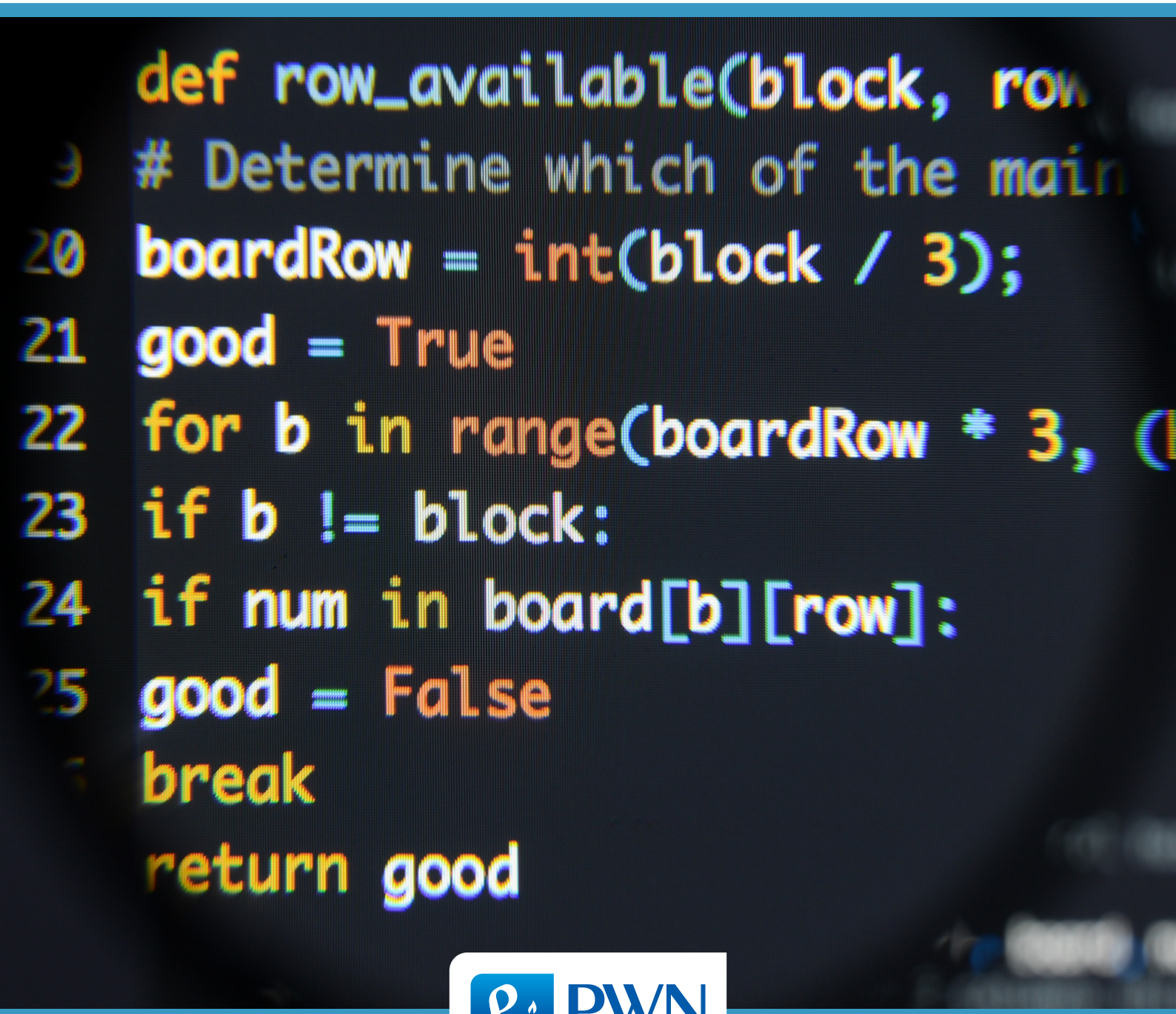
ROZWIĄZUJ PROBLEMY
WYSZUKIWANIA ZA POMOCĄ JĘZYKA

PYTHON



PWN

Rozwiązuj problemy wyszukiwania za pomocą języka **Python**

Publikacja opisuje problemy wyszukiwania i techniki ich rozwiązywania za pomocą języka Python. Wyszukiwanie to bardzo szeroki temat, w ebooku *Rozwiązuj problemy wyszukiwania za pomocą języka Python* zostały omówione te najważniejsze algorytmy wyszukiwania, takie jak wyszukiwanie binarne, przeszukiwanie wszerz, przeszukiwanie w głąb i A*.

Problemy i ich rozwiązania omawiane w tej publikacji mają pomóc doświadczonym programistom w odświeżeniu wiedzy ze studiów informatycznych, wprowadzając jednocześnie pewne zaawansowane elementy języka Python. Jest to idealny materiał np. do odświeżenia wiedzy przed rozmową kwalifikacyjną albo okazja do poznania nowych technik rozwiązywania problemów, które mogą się przydać w codziennej pracy. Osoby, które samodzielnie uczą się programowania, będą mogły przyspieszyć swoją edukację informatyczną, poznając klasyczne problemy informatyczne w Pythonie.

Z publikacji dowiesz się:

- na czym polega przeszukiwanie DNA,
- co ma wspólnego wychodzenie z labiryntu z rozwiązywaniem problemów informatycznych,
- ak przeprawić trzech misjonarzy i trzech kanibali na drugi brzeg rzeki, tak aby wszyscy przeżyli i jaki ma to związek z programowaniem,
- jak korzystać z algorytmu A*.

Opanuj techniki rozwiązywania problemów wyszukiwania w Pythonie i zwiększ szanse sukcesu realizowanych projektów webowych, przetwarzania danych, uczenia maszynowego i wielu innych.

Wioleta Szczygielska-Dybciak

Wioleta Szczygielska-Dybciak
Wydawca Redakcji IT

1.1. Przeszukiwanie DNA

Geny często są reprezentowane w oprogramowaniu ciągiem znaków A, C, G i T. Każda litera reprezentuje *nukleotyd*, a kombinacja trzech nukleotydów jest nazywana *kodonem*, jak pokazano na rysunku 1.1. Kodon koduje określony aminokwas, który w połączeniu z innymi aminokwasami może tworzyć *białko*. Klasyczne zadanie oprogramowania bioinformatycznego polega na znajdowaniu określonego kodonu w genie.

1.1.1. Przechowywanie DNA

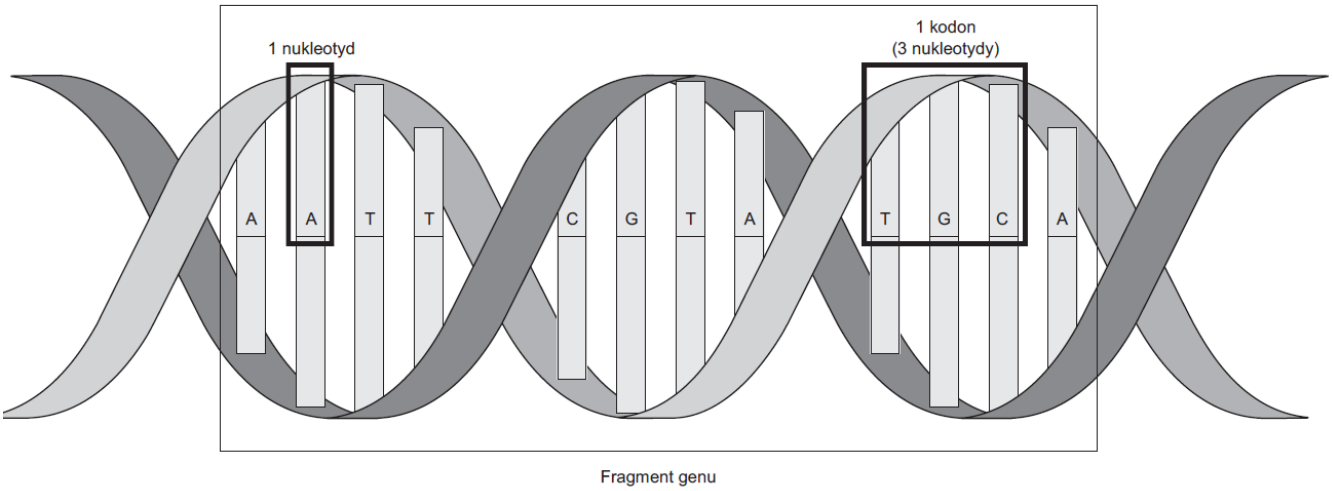
Do reprezentowania nukleotydu użyjemy prostego typu `IntEnum` z czterema przypadkami.

Listing 1.1. dna_search.py

```
from enum import IntEnum
from typing import Tuple, List

Nucleotide: IntEnum = IntEnum('Nucleotide', ('A', 'C', 'G', 'T'))
```

`Nucleotide` jest typu `IntEnum`, a nie `Enum`, ponieważ `IntEnum` oferuje „gratisowe" operatory porównań (<, >= itp.). Te operatory muszą być dostępne w typie danych, aby implementowane przez nas algorytmy wyszukiwania mogły wykonywać na nim działania



Rysunek 1.1. Nukleotyd jest reprezentowany przez jedną z czterech liter: A, C, G lub T. Kodon składa się z trzech nukleotydów, a gen z wielu kodonów

Ponadto importujemy typy `Tuple` (krotka) i `List` z pakietu typing z myślą o adnotacjach typów.

Kodony można definiować jako krotki składające się z trzech obiektów Nucleotide, natomiast gen można definiować jako listy obiektów `Codon`.

Listing 1.2. dna_search.py kontynuacja

```
Codon = Tuple[Nucleotide, Nucleotide, Nucleotide] # alias typu dla kodonów
Gene = List[Codon] # alias typu dla genów
```

UWAGA Chociaż później będziemy musieli porównywać ze sobą kodony, nie musimy definiować niestandardowej klasy z operatorem < zaimplementowanym specjalnie dla klasy `Codon`. Wynika to z faktu, że Python oferuje wbudowane wsparcie dla porównań między krotkami złożonymi z typów, które również wspierają porównania.Geny w internecie są zazwyczaj przechowywane w formacie plikowym, który zawiera bardzo długi łańcuch reprezentujący wszystkie nukleotydy w sekwencji tego genu. Zdefiniujemy taki łańcuch na potrzeby pewnego przykładowego genu o nazwie `gene _ str`.

Listing 1.3. dna_search.py kontynuacja

```
gene_str: str = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT"
```

Będziemy potrzebować również funkcji pomocniczej do przekształcania typu `str` do `Gene`.

Listing 1.4. dna_search.py kontynuacja

```
def string_to_gene(s: str) -> Gene:
    gene: Gene = []
    for i in range(0, len(s), 3):
        if (i + 2) >= len(s): # nie przekrocz końca!
            return gene
        # inicjalizuj kodon na bazie trzech nukleotydów
        codon: Codon = (Nucleotide[s[i]], Nucleotide[s[i + 1]],
       Nucleotide[s[i + 2]])
        gene.append(codon) # dodaj kodon do genu
    return gene
```

Funkcja `string _ to _ gene()` przechodzi przez cały otrzymany `str` i przekształca jego trzy kolejne znaki w `Codon`, który dodaje na końcu nowego genu. Gdy dwa miejsca za obecnie analizowaną pozycją nie można znaleźć już nukleotydu (patrz instrukcja `if` w pętli), funkcja wie, że osiągnęła koniec niepełnego genu i ignoruje jeden lub dwa ostatnie nukleotydy.

Funkcji `string _ to _ gene()` można użyć do przekształcania łańcucha `gene _ str` w obiekt `Gene`.

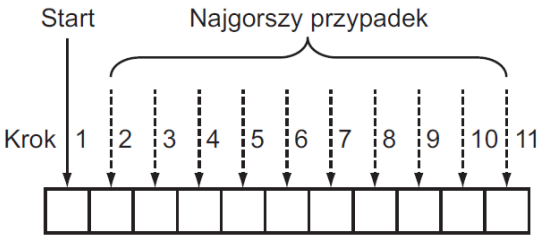
Listing 1.5. dna_search.py kontynuacja

```
my_gene: Gene = string_to_gene(gene_str)
```

1.1.2. Przeszukiwanie liniowe

Jedną z operacji, jakie możemy chcieć wykonać na genie, jest wyszukanie określonego kodonu. Celem jest po prostu ustalenie, czy kodon znajduje się w genie. Operacja przeszukiwania liniowego przechodzi przez wszystkie elementy w przeszukiwanym obszarze, w kolejności zgodnej z wewnętrzną strukturą danych, dopóki nie znajdzie tego, czego szuka lub nie dotrze do końca struktury danych. W konsekwencji przeszukiwanie liniowe stanowi najłatwiejszy, najbardziej naturalny i oczywisty sposób szukania czegoś. W najgorszym przypadku przeszukiwanie liniowe będzie wymagało przejścia przez wszystkie elementy w strukturze danych, a zatem ma złożoność O*(n)*, gdzie *n* to liczba elementów w strukturze. Zostało to zilustrowane na rysunku 1.2.

Definiowanie funkcji, która wykonuje przeszukiwanie liniowe, jest banalnie proste. Wystarczy przejść przez każdy z elementów w strukturze danych i porównać go z elementem, którego szukamy. Następujący kod definiuje taką funkcję dla obiektów `Gene` i `Codon`, a następnie testuje ją na obiekcie `my _ gene` i kodonach o nazwach `acg` oraz `gat`.



Rysunek 1.3. W najgorszym przypadku wyszukiwania binarnego musimy przejść jedynie przez `lg(n)` elementów listy

Listing 1.6. dna_search.py kontynuacja

```
def linear_contains(gene: Gene, key_codon: Codon) -> bool:
    for codon in gene:
        if codon == key_codon:
            return True
    return False

acg: Codon = (Nucleotide.A, Nucleotide.C, Nucleotide.G)
gat: Codon = (Nucleotide.G, Nucleotide.A, Nucleotide.T)
print(linear_contains(my_gene, acg)) # True
print(linear_contains(my_gene, gat)) # False
```

UWAGA Ta funkcja służy jedynie do celów ilustracyjnych. Wszystkie wbudowane typy sekwencyjne w Pythonie (`list`, `tuple`, `range`) implementują metodę `__contains__()`, która umożliwia wyszukiwanie w nich określonego elementu po prostu przy użyciu operatora in. Co więcej, operator in można zastosować na każdym typie, który implementuje metodę `__contains__()`. Na przykład do wyszukania `acg` w `my_gene` i wyświetlenia wyniku moglibyśmy użyć polecenia `print(acg w my_gene)`.

1.1.3. Wyszukiwanie binarne

Istnieje szybszy sposób wyszukiwania niż przechodzenie przez wszystkie elementy, ale aby go zastosować, trzeba znać kolejność danych w strukturze danych. Jeśli wiemy, że struktura jest posortowana i zapewnia natychmiastowy dostęp do dowolnego elementu za pomocą indeksu, możemy przeprowadzić wyszukiwanie binarne. W świetle tych kryteriów posortowana lista Pythona świetnie nadaje się do wyszukiwania binarnego.

Wyszukiwanie binarne polega na znajdowaniu środkowego elementu w posortowanym zakresie elementów, porównaniu go z wyszukiwanym elementem, zmniejszaniu zakresu o połowę na podstawie wyniku tego porównania i zaczynaniu procesu od nowa. Przeanalizujmy konkretny przykład.

Przypuśćmy, że mamy listę posortowanych alfabetycznie słów, na przykład [„`cat`", „`dog`", „`kangaroo`", „`llama`", „`rabbit`", „`rat`", „`zebra`"] i szukamy słowa „`rat`":

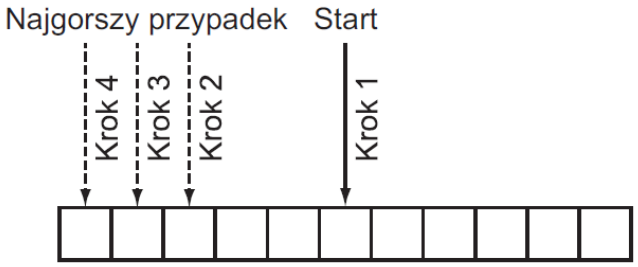
1 Ustalamy, że środkowym elementem tej listy zawierającej siedem słów jest słowo „llama".

2 Jak wiemy, słowo „rat" leży alfabetycznie za słowem „llama", dlatego musi znajdować się w tej (mniej więcej) połówce listy, która leży za słowem „llama" (gdybyśmy w tym kroku znaleźli słowo „rat", moglibyśmy zwrócić jego pozycję, natomiast gdyby szukane słowo leżało przed sprawdzanym środkowym słowem, wiedzielibyśmy, że leży w połówce listy znajdującej się przed słowem „llama").

3 Możemy ponownie wykonać kroki 1. i 2. dla połówki listy, w której potencjalnie znajduje się słowo „rat". W rezultacie ta połowa staje się nową podstawową listą. Powyższe kroki powtarzamy do znalezienia słowa

„rat” lub do momentu, gdy analizowany zakres nie zawiera już żadnych elementów do przeszukania, co oznacza, że słowo „rat” nie występuje na liście słów.

Przeszukiwanie DNA



Rysunek 1.3. W najgorszym przypadku wyszukiwania binarnego musimy przejść jedynie przez lg(*n*) elementów listy

Na rysunku 1.3 zilustrowano wyszukiwanie binarne. Jak widać, w odróżnieniu od przeszukiwania liniowego, nie wymaga ono sprawdzania każdego elementu.

Wyszukiwanie binarne stale zmniejsza obszar wyszukiwania o połowę, w związku z tym czas wykonania wynosi w najgorszym przypadku O(*lg n*). Istnieje jednak pewne ograniczenie. W odróżnieniu od wyszukiwania liniowego, wyszukiwanie binarne wymaga, aby przeszukiwana struktura danych była posortowana, a sortowanie zajmuje czas. Tak naprawdę czas sortowania wynosi O(*n lg n*) w przypadku najlepszych algorytmów sortowania. Jeśli będziemy przeprowadzać wyszukiwanie tylko raz, a nasza bazowa struktura danych nie jest posortowana, bardziej sensowne jest zwykle przeprowadzenie zwykłego przeszukiwania liniowego. Jeśli jednak planujemy przeprowadzać wyszukiwanie wielokrotnie, warto poświęcić czas na sortowanie, aby móc czerpać korzyści ze znacznie obniżonego kosztu poszczególnych operacji wyszukiwania.

Pisanie funkcji wyszukiwania binarnego kodonu w genie przypomina pisanie tej funkcji dla innych typów danych, ponieważ typ `Codon` można porównywać z innymi obiektami tego rodzaju, a typ `Gene` jest po prostu listą.

Listing 1.7. dna_search.py kontynuacja

```
def binary_contains(gene: Gene, key_codon: Codon) -> bool:
    low: int = 0
    high: int = len(gene) - 1
    while low <= high: # dopóki istnieje obszar do przeszukania
        mid: int = (low + high) // 2
        if gene[mid] < key_codon:
            low = mid + 1
        elif gene[mid] > key_codon:
            high = mid - 1
        else:
            return True
    return False
```

Prześledźmy tę funkcję wiersz po wierszu.

```
    low: int = 0
```

```
    high: int = len(gene) - 1
```

Zaczynamy od przeanalizowania zakresu, który zawiera całą listę (gene).

```
    while low <= high:
```

Kontynuujemy wyszukiwanie tak długo, jak długo istnieje zakres do przeszukania.

Gdy zmienna low jest mniejsza niż high, oznacza to, że na liście nie ma już miejsc do przeszukania.

```
    mid: int = (low + high) // 2
```

Wyznaczamy środkowy element mid, używając dzielenia całkowitego i prostego wzoru na średnią poznanego w szkole podstawowej.

```
    if gene[mid] < key_codon:

        low = mid + 1
```

Jeśli szukany element znajduje się za środkowym elementem aktualnego zakresu, modyfikujemy zakres stosowany w kolejnej iteracji pętli, przenosząc koniec low za obecny środkowy element. W tym miejscu dzielimy zakres na połowę z myślą o kolejnej iteracji.

```
    elif gene[mid] > key_codon:

        high = mid - 1
```

Analogicznie skracamy zakres o połowę w drugim kierunku, jeśli szukany element jest mniejszy niż środkowy.

```
    else:

        return True
```

Jeśli porównywany element nie jest ani mniejszy, ani większy od środkowego elementu, oznacza to, że znaleźliśmy go na liście! I oczywiście, jeśli skończą się iteracje pętli, zwracamy False (co nie zostało tu pokazane), wskazując, że element nie został znaleziony.

Możemy uruchomić naszą funkcję na tym samym genie i kodonie, ale najpierw musimy go posortować.

Listing 1.8. dna_search.py kontynuacja

```
my_sorted_gene: Gene = sorted(my_gene)
print(binary_contains(my_sorted_gene, acg)) # True
print(binary_contains(my_sorted_gene, gat)) # False
```

WSKAZÓWKA Można zaimplementować wydajne wyszukiwanie binarne przy użyciu modułu `bisect` z biblioteki standardowej Pythona: https://docs.python.org/3/library/bisect.html.

1.1.4. Uogólniony przykład

Możemy uogólnić funkcje l`inear_contains()` oraz `binary_contains()` w taki sposób, aby działały one na niemal dowolnej sekwencji Pythona. Następujące uogólnione wersje są niemal takie same jak wersje przedstawione wcześniej, zmienione zostały jedynie pewne nazwy i adnotacje typów.

UWAGA W poniższym listingu kodu zaimportowaliśmy wiele typów. Plik generic_search.py będziemy wykorzystywać w innych uogólnionych algorytmach wyszukiwania przedstawionych w dalszej części tego rozdziału i to polecenie importuje wszystkie typy, jakich będziemy potrzebować w przyszłości.

UWAGA Przed kontynuowaniem lektury trzeba zainstalować moduł typing_extensions za pomocą polecenia pip `install typing_extensions` lub `pip3 install typing_extensions`, w zależności od konfiguracji interpretera Pythona. Moduł ten umożliwia uzyskiwanie dostępu do typu `Protocol`, który

w przyszłej wersji Pythona będzie wchodził w skład biblioteki standardowej (zgodnie ze specyfikacją PEP 544). W związku z tym w przyszłej wersji Pythona importowanie modułu `typing _ extensions` powinno nie być konieczne i powinno wystarczyć polecenie from `typing import Protocol` zamiast from `typing _ extensions import Protocol`.

Listing 1.9. generic_search.py

```
from __future__ import annotations
from typing import TypeVar, Iterable, Sequence, Generic, List, Callable, Set,
    Deque, Dict, Any, Optional
from typing_extensions import Protocol
from heapq import heappush, heappop

T = TypeVar('T')

def linear_contains(iterable: Iterable[T], key: T) -> bool:
    for item in iterable:
        if item == key:
            return True
    return False

C = TypeVar("C", bound="Comparable")

class Comparable(Protocol):
    def __eq__(self, other: Any) -> bool:
        ...

    def __lt__(self: C, other: C) -> bool:
        ...

    def __gt__(self: C, other: C) -> bool:
        return (not self < other) and self != other

    def __le__(self: C, other: C) -> bool:
        return self < other or self == other

    def __ge__(self: C, other: C) -> bool:
        return not self < other

def binary_contains(sequence: Sequence[C], key: C) -> bool:
    low: int = 0
    high: int = len(sequence) - 1
    while low <= high: # dopóki istnieje zakres do przeszukania
        mid: int = (low + high) // 2
        if sequence[mid] < key:
            low = mid + 1
        elif sequence[mid] > key:
            high = mid - 1
        else:
            return True
    return False
```

```
if __name__ == "__main__":
    print(linear_contains([1, 5, 15, 15, 15, 15, 20], 5)) # True
    print(binary_contains(["a", "d", "e", "f", "z"], "f")) # True
    print(binary_contains(["john", "mark", "ronald", "sarah"], "sheila"))
     #False
```

Teraz możesz spróbować przeprowadzić wyszukanie na innych typach danych. Tych funkcji można używać na niemal dowolnej kolekcji Pythona. Na tym polega przewaga uogólnionego kodu. Jedynym słabym punktem tego przykładu są komplikacje związane z adnotacjami typów w Pythonie, w postaci klasy Comparable. Typ `Comparable` jest typem implementującym operatory porównania (<, >= itd.). Przyszłe wersje Pythona powinny oferować zwięźlejszy sposób tworzenia adnotacji typów dla typów, które implementują te standardowe operatory.

1.2. Rozwiązywanie labiryntów

Znajdowanie wyjścia z labiryntu stanowi w informatyce odpowiednik wielu popularnych problemów wyszukiwania. Dlatego do zilustrowania algorytmów przeszukiwania wszerz, w głąb i A* użyjemy funkcji znajdowania drogi wyjścia z fizycznego labiryntu.

Nasz labirynt będzie dwuwymiarową siatką komórek typu `Cell`. `Cell` będzie typem wyliczeniowym z różnymi wartościami str, gdzie „ „ reprezentuje pustą, a „X” zablokowaną komórkę. Użyjemy również innych wartości wykorzystywanych do wyświetlania labiryntu na ekranie.

Listing 1.10. maze.py

```
from enum import Enum
from typing import List, NamedTuple, Callable, Optional
import random
from math import sqrt
from generic_search import dfs, bfs, node_to_path, astar, Node

class Cell(str, Enum):
    EMPTY = " "
    BLOCKED = "X"
    START = "S"
    GOAL = "G" # cel
    PATH = "*" # droga do wyjścia
```

Ponownie zaczynamy od zaimportowania wielu klas. Warto zauważyć, że ostatnie polecenie importu (z `generic _ search`) dotyczy symboli, których jeszcze nie zdefiniowaliśmy. Dołączyliśmy je dla wygody, ale możesz je umieścić w komentarzu do momentu, aż będzie potrzebne.

Będziemy musieli się odwołać do określonej pozycji w labiryncie. Do tego celu użyjemy po prostu krotki `NamedTuple` z właściwościami reprezentującymi odpowiedni wiersz (`row`) i kolumnę (`column`).

Listing 1.11. maze.py kontynuacja

```
class MazeLocation(NamedTuple):
    row: int
    column: int
```

1.2.1. Generowanie losowego labiryntu

Nasza klasa `Maze` będzie zawierała siatkę (listę `list`) reprezentującą stan labiryntu. Ponadto będzie zawierała zmienne instancji reprezentujące liczbę wierszy, liczbę kolumn, pozycję początkową i pozycję celu. Siatka zostanie w losowy sposób wypełniona zablokowanymi komórkami.

Generowany labirynt powinien być stosunkowo pusty, aby prawie zawsze istniała droga z danej pozycji początkowej do danej pozycji docelowej (w końcu będzie on służył do testowania naszych algorytmów). Pozwolimy, aby osoba tworząca nowy labirynt mogła określić dokładny stopień zapełnienia (sparseness), ale określimy domyślną wartość 20% zablokowanych komórek. Gdy losowa liczba będzie mniejsza niż określony parametr sparseness, puste miejsce po prostu zastąpimy blokadą. Jeśli powtórzymy zabieg dla każdej pozycji w labiryncie, stopień zapełnienia labiryntu powinien statystycznie wynosić mniej więcej tyle, ile określony parametr sparseness.

Listing 1.12. maze.py kontynuacja

```
class Maze:
    def __init__(self, rows: int = 10, columns: int = 10, sparseness: float =
     0.2, start: MazeLocation = MazeLocation(0, 0), goal: MazeLocation =
     MazeLocation(9, 9)) -> None:
        # inicjalizuj podstawowe zmienne instancji
        self._rows: int = rows
        self._columns: int = columns
        self.start: MazeLocation = start
        self.goal: MazeLocation = goal
        # wypełnij siatkę pustymi komórkami
        self._grid: List[List[Cell]] = [[Cell.EMPTY for c in range(columns)]
    for r in range(rows)]
        # wypełnij siatkę zablokowanymi komórkami
        self._randomly_fill(rows, columns, sparseness)
        # wypełnij pozycje początkową i docelową w siatce
        self._grid[start.row][start.column] = Cell.START
        self._grid[goal.row][goal.column] = Cell.GOAL

    def _randomly_fill(self, rows: int, columns: int, sparseness: float):
        for row in range(rows):
            for column in range(columns):
                if random.uniform(0, 1.0) < sparseness:
                    self._grid[row][column] = Cell.BLOCKED
```

Labirynt już mamy, ale potrzebujemy jeszcze funkcji do wyświetlania go w konsoli w zwięzłej postaci. Chcemy, by znaki znajdowały się blisko siebie, aby wyglądały jak labirynt.

Listing 1.13. maze.py kontynuacja

```
# zwróć ładnie sformatowaną wersję labiryntu do wyświetlenia
def __str__(self) -> str:
    output: str = ""
    for row in self._grid:
        output += "".join([c.value for c in row]) + "\n"
    return output
```

A teraz przetestujmy te funkcje labiryntu.

```
maze: Maze = Maze()

print(maze)
```

1.2.2. Detale labiryntu

Przydałaby się nam funkcja do sprawdzania, czy w trakcie przeszukiwania osiągnęliśmy cel. Innymi słowy, chcemy sprawdzać, czy pozycja `MazeLocation`, do której dotarła operacja przeszukiwania, jest celem. Tę metodę możemy dodać do klasy `Maze`.

Listing 1.14. maze.py kontynuacja

```
def goal_test(self, ml: MazeLocation) -> bool:
    return ml == self.goal
```

W jaki sposób możemy poruszać się po labiryncie? Załóżmy, że możemy przesuwać się o jedno miejsce w pionie lub w poziomie z danego miejsca. Funkcja `successors()` może znajdować kolejne dozwolone pozycje z określonej pozycji `MazeLocation` z uwzględnieniem tych kryteriów. Funkcja `successors()` będzie inna dla każdego labiryntu, ponieważ każdy z nich ma rozmiar i zbiór ścian. W związku z tym zdefiniujemy ją w postaci metody klasy `Maze`.

Listing 1.15. maze.py kontynuacja

```
def successors(self, ml: MazeLocation) -> List[MazeLocation]:
    locations: List[MazeLocation] = []
    if ml.row + 1 < self._rows and self._grid[ml.row + 1][ml.column] !=
     Cell.BLOCKED:
        locations.append(MazeLocation(ml.row + 1, ml.column))
    if ml.row - 1 >= 0 and self._grid[ml.row - 1][ml.column] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row - 1, ml.column))
    if ml.column + 1 < self._columns and self._grid[ml.row][ml.column + 1] !=
     Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column + 1))
    if ml.column - 1 >= 0 and self._grid[ml.row][ml.column - 1] !=
     Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column - 1))
    return locations
```

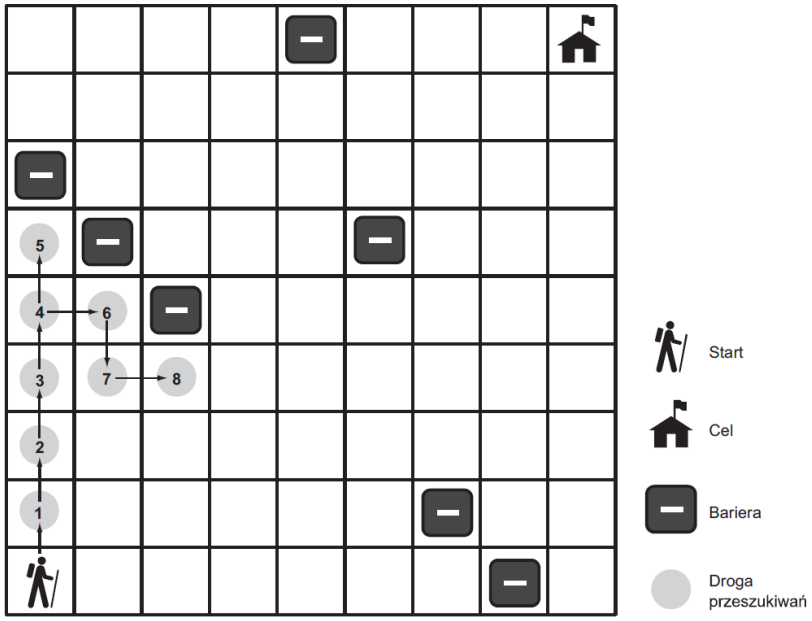
`successors()` po prostu sprawdza miejsca w labiryncie znajdujące się nad, pod, po prawej i po lewej stronie pozycji MazeLocation w poszukiwaniu pustego miejsca, na które można przejść z danej pozycji. Metoda nie sprawdza pozycji, które wykraczają poza brzegi labiryntu. Metoda umieszcza każdą znalezioną dozwoloną lokalizację `MazeLocation` na liście, którą na końcu zwraca do obiektu wywołującego.

1.2.3. Przeszukiwanie w głąb

Przeszukiwanie w głąb (ang. *depth-first serach*, *DFS*), jak sama nazwa wskazuje, to przeszukiwanie, które przechodzi tak głęboko, jak to możliwe, a następnie, jeśli weszło w ślepą uliczkę, wraca do ostatniego rozgałęzienia. Zaimplementujemy uogólnioną funkcję przeszukiwania w głąb, którą będziemy mogli wykorzystać do rozwiązywania naszego labiryntu. Aczkolwiek funkcja może służyć do rozwiązywania także innych problemów. Na rysunku 1.4 zilustrowano proces przeszukiwania labiryntu w głąb.

STOSY

Algorytm przeszukiwania w głąb bazuje na strukturze danych zwanej stosem. Stos to struktura danych, która działa zgodnie z zasadą *Last-In-First-Out* (*LIFO* – ostatni na wejściu, pierwszy na wyjściu). Wyobraź sobie stos kartek. Pierwsza kartka położona na górze stosu jest pierwszą kartką ściąganą ze stosu. Stos jest często implementowany na bazie prostszej struktury danych, takiej jak lista. Nasz stos zaimplementujemy przy użyciu typu `list` Pythona.



Rysunek 1.4. W przeszukiwaniu w głąb operacja jest kontynuowana w głąb, aż do napotkania ściany, która zmusza do powrotu do ostatniego rozgałęzienia

Stosy oferują zwykle przynajmniej dwie operacje:

- `push()` – umieszcza element na górze stosu,
- `pop()` – usuwa ze stosu górny element i go zwraca.

Zaimplementujemy obie metody, jak również właściwość empty do sprawdzania, czy stos jest pusty (nie ma już żadnych elementów). Kod stosu dodamy do pliku `generic _ search.py`, z którego korzystaliśmy we wcześniejszej części rozdziału. Wszystkie niezbędne polecenia importu dodaliśmy już wcześniej.

Listing 1.16. generic_search.py kontynuacja

```
class Stack(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []

    @property
    def empty(self) -> bool:
        return not self._container # not zwraca true dla pustego pojemnika

    def push(self, item: T) -> None:
        self._container.append(item)

    def pop(self) -> T:
        return self._container.pop() # LIFO
```

```
    def __repr__(self) -> str:
        return repr(self._container)
```

Jak widać, aby zaimplementować stos przy użyciu listy Pythona, wystarczy zawsze dołączać elementy do prawego końca listy i zawsze usuwać elementy z prawego końca. Wykonanie metody `pop()` na liście nie powiedzie się, jeżeli lista nie będzie zawierała żadnych elementów, dlatego, gdy stos jest pusty, wykonanie metody `pop()` na obiekcie `Stack` również zakończy się niepowodzeniem.

ALGORYTM DFS

Będziemy potrzebować jeszcze jednego detalu, zanim zajmiemy się implementowaniem algorytmu DFS. Potrzebujemy klasy `Node`, której użyjemy do śledzenia, w jaki sposób przeszliśmy z jednego stanu do drugiego (z jednego miejsca do drugiego) w czasie przeszukiwania. Klasa `Node` będzie stanowić coś w rodzaju otoki stanu. W naszym labiryncie typ stanu to `MazeLocation`. Obiekt `Node`, z którego doszliśmy do aktualnego stanu, będziemy nazywać jego rodzicem (`parent`). Nasza klasa `Node` będzie zawierała również właściwości koszt (`cost`) i heurystyka (`heuristic`) oraz implementowała metodę `__lt__()`, dzięki czemu będziemy mogli użyć jej później w algorytmie A*.

Listing 1.17. generic_search.py kontynuacja

```
class Node(Generic[T]):
    def __init__(self, state: T, parent: Optional[Node], cost: float = 0.0,
     heuristic: float = 0.0) -> None:
      self.state: T = state
      self.parent: Optional[Node] = parent
      self.cost: float = cost
      self.heuristic: float = heuristic

  def __lt__(self, other: Node) -> bool:
      return (self.cost + self.heuristic) < (other.cost + other.heuristic)
```

WSKAZÓWKA Typ `Optional` wskazuje, że do wartości sparametryzowanego typu można przypisać zmienną lub wartość `None`.

WSKAZÓWKA Polecenie `from __future__ import annotations` w górnej części pliku sprawia, że `Node` może odwoływać się do siebie w adnotacjach typów swoich metod. Bez niego musielibyśmy umieszczać adnotację typu w cudzysłowie w postaci łańcucha (np. ‚`Node`'). W przyszłych wersjach Pythona importowanie `annotations` nie będzie konieczne. Dodatkowe informacje można znaleźć w dokumencie PEP 563, *Postponed Evaluation of Annotations* pod adresem http://mng.bz/pgzR.

W procesie przeszukiwania w głąb trzeba śledzić dwie struktury danych: stos stanów (czyli „miejsc"), jakie planujemy przeszukać, któremu nadamy nazwę `frontier`, oraz zbiór stanów, jakie już przeszukaliśmy, któremu nadamy nazwę `explored`. Tak długo, jak `stos frontier` zawiera więcej stanów do przejrzenia, algorytm DFS będzie sprawdzał, czy są one celem (gdy stan jest celem, algorytm DFS się zatrzyma i go zwróci) oraz dodawał kolejne stany do stosu. Algorytm DFS będzie również oznaczał każdy przeszukany stan jako sprawdzony `explored`, aby operacja przeszukania się nie zapętliła, przechodząc przez stany, które odwiedziła już wcześniej. Jeśli `stos frontier` jest pusty, nie ma już nic do przeszukania.

Listing 1.18. generic_search.py kontynuacja

```
def dfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
     List[T]]) -> Optional[Node[T]]:
    # frontier to stos nieodwiedzonych jeszcze miejsc
    frontier: Stack[Node[T]] = Stack()
    frontier.push(Node(initial, None))
    # explored to zbiór miejsc już odwiedzonych
    explored: Set[T] = {initial}

    # kontynuuj przeszukanie, dopóki pozostaje coś do sprawdzenia
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # jeśli znaleźliśmy cel, kończymy
        if goal_test(current_state):
            return current_node
        # sprawdź, gdzie możemy przejść i czego nie sprawdziliśmy
        for child in successors(current_state):
            if child in explored: # pomiń sprawdzone już stany
                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None # wszystko przeszukane i cel nieodnaleziony
```

Jeśli wykonanie funkcji `dfs()` się powiedzie, zwróci ona obiekt `Node` reprezentujący stan celu. Możemy odtworzyć drogę od początku do celu, przechodząc wstecz przez ten obiekt `Node` i poprzedzające go węzły przy użyciu właściwości parent.

Listing 1.19. generic_search.py kontynuacja

```
def node_to_path(node: Node[T]) -> List[T]:
    path: List[T] = [node.state]
    # odtwarzanie drogi od końca do początku
    while node.parent is not None:
        node = node.parent
        path.append(node.state)
    path.reverse()
    return path
```

Na potrzeby wyświetlania warto oznaczyć w labiryncie odnalezioną drogę od startu do celu. Przydałby się również mechanizm usuwania drogi, abyśmy mogli przetestować na tym samym labiryncie inny algorytm przeszukiwania. Następujące dwie metody dodamy do klasy Maze w pliku `maze.py`.

Listing 1.20. maze.py kontynuacja

```
def mark(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.PATH
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL

def clear(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.EMPTY
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL
```

To była długa droga, ale wreszcie jesteśmy gotowi do rozwiązania labiryntu.

Listing 1.21. maze.py kontynuacja

```
if __name__ == "__main__":
    # Testuj DFS
    m: Maze = Maze()
    print(m)
    solution1: Optional[Node[MazeLocation]] = dfs(m.start, m.goal_test,
     m.successors)
    if solution1 is None: # Nie znaleziono rozwiązania
        print("No solution found using depth-first search!")
    else:
        path1: List[MazeLocation] = node_to_path(solution1)
```

Pomyślne rozwiązanie będzie wyglądało podobnie do następującego:

```
    S****X X
     X *****
          X*
     XX******X
      X*
      X**X
     X *****
           *
          X *X
            *G
```

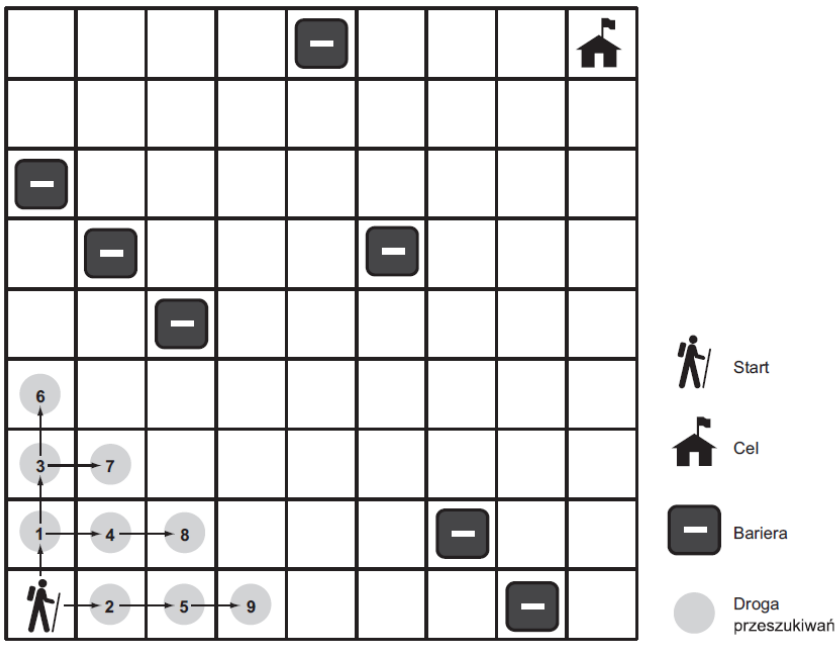
Gwiazdki reprezentują drogę od startu do celu znalezioną przez funkcję przeszukiwania w głąb. Należy pamiętać, że każdy labirynt jest generowany w sposób losowy i nie wszystkie będą miały rozwiązanie.

1.2.4. Przeszukiwanie wszerz

Jak można zauważyć, drogi wyjścia z labiryntu odnajdowane przez algorytm przeszukiwania w głąb wydają się dość nienaturalne. Zazwyczaj nie są one najkrótszymi drogami. Natomiast algorytm przeszukiwania wszerz (ang. *breadth-first serach*, *BFS*) zawsze znajduje najkrótszą drogę, systematycznie przeglądając w każdej iteracji kolejne warstwy węzłów coraz bardziej oddalone od stanu początkowego. To, który algorytm przeszukiwania daje większe szanse na szybsze znalezienie rozwiązania, zależy od rodzaju problemu. Wybór algorytmu czasami sprowadza się do decyzji, czy ważniejsza jest gwarancja znalezienia najkrótszej drogi do celu (jeśli taka istnieje), czy też szansa na szybkie znalezienie rozwiązania. Na rysunku 2.5 zilustrowano proces przeszukiwania labiryntu wszerz.

Aby zrozumieć, dlaczego przeszukiwanie w głąb czasami zwraca wynik szybciej niż przeszukiwanie wszerz, wyobraź sobie, że szukasz plamki znajdującej się na pewnej warstwie cebuli. Korzystanie ze strategii przeszukiwania w głąb przypomina wbicie noża w środek cebuli i chaotyczne analizowanie wycinanych kawałków. Jeśli plamka przypadkiem znajduje się blisko wyciętego fragmentu, jest szansa znalezienia jej szybciej niż gdybyśmy wybrali strategię przeszukiwania wszerz, która polega na pracochłonnym obieraniu cebuli z kolejnych pojedynczych warstw.

Aby jeszcze lepiej zrozumieć, dlaczego przeszukiwanie wszerz zawsze znajduje najkrótsze rozwiązanie (o ile ono istnieje), wyobraźmy sobie, że próbujemy znaleźć trasę kolejową z Bostonu do Nowego Jorku z najmniejszą liczbą przystanków. Jeśli będziemy zmierzać w tym samym kierunku, wracając dopiero po dotarciu do końca trasy (jak w przeszukiwaniu w głąb), pierwsza znaleziona trasa może prowadzić aż do Seattle przed połączeniem wracającym do Nowego Jorku. Natomiast stosując algorytm przeszukiwania wszerz, zaczniemy od sprawdzenia wszystkich stacji znajdujących jeden przystanek od Bostonu. Później sprawdzimy wszystkie stacje znajdujące się dwa przystanki od Bostonu.



Rysunek 1.5. W przeszukiwaniu wszerz elementy znajdujące się najbliżej początku zostają przeszukane najwcześniej

Następnie sprawdzimy wszystkie stacje trzy przystanki od Bostonu. I proces będzie kontynuowany aż do odnalezienia Nowego Jorku. W związku z tym, gdy odnajdziemy Nowy Jork, będziemy wiedzieli, że znaleźliśmy trasę z najmniejszą liczbą przystanków, ponieważ sprawdziliśmy wszystkie stacje oddalone od Nowego Jorku o mniejszą liczbę przystanków i żadna z nich nie docierała do Nowego Jorku.

KOLEJKI

Do zaimplementowania algorytmu BFS będziemy potrzebować struktury danych nazywanej *kolejką* (ang. *queue*). Podczas gdy stos działa zgodnie z regułą LIFO, kolejka działa zgodnie z regułą FIFO (ang. First-*In-First-Out* – pierwszy na wejściu, pierwszy na wyjściu). – podobnie jak w przypadku zwykłej kolejki, w której na początku jest obsługiwana osoba, która jako pierwsza ustawiła się w kolejce. Kolejka powinna oferować przynajmniej te same metody `push()` i `pop()` co stos. W rzeczywistości nasza implementacja klasy `Queue` (bazująca na typie Pythona `deque`) jest prawie taka sama jak implementacja `Stack` – z tą różnicą, że elementy są usuwane z lewej strony pojemnika `_container` (używając słowa lewej, mamy na myśli początek wewnętrznego magazynu) i że lista zostaje zastąpiona obiektem `Deque`. Elementy leżące po lewej stronie są najstarszymi elementami znajdującymi się nadal w obiekcie `Deque` (pod względem czasu dodania), dlatego są pierwszymi elementami do ściągnięcia.

Listing 1.22. generic_search.py kontynuacja

```
class Queue(Generic[T]):
    def __init__(self) -> None:
        self._container: Deque[T] = Deque()

    @property
    def empty(self) -> bool:
        return not self._container # not zwraca true dla pustego pojemnika

    def push(self, item: T) -> None:
        self._container.append(item)

    def pop(self) -> T:
        return self._container.popleft() # FIFO

    def __repr__(self) -> str:
        return repr(self._container)
```

WSKAZÓWKA Dlaczego implementacja klasy `Queue` wykorzystuje w roli wewnętrznego magazynu typ `Deque` zamiast listy jak w implementacji klasy `Stack`? Jest to związane ze sposobem ściągania elementów. W przypadku stosu dodajemy i ściągamy elementy z prawej strony. Natomiast w przypadku kolejki także dodajemy elementy z prawej strony, ale ściągamy je z lewej strony. Struktura danych `List` w Pythonie oferuje efektywny sposób ściągania elementów z prawej, ale nie z lewej. Natomiast typ `Deque` pozwala na efektywne ściąganie elementów z dowolnej strony. W rezultacie typ `Deque` ma wbudowaną metodę o nazwie `popleft()`, natomiast lista nie oferuje jej odpowiednika. Wprawdzie moglibyśmy znaleźć inne sposoby wykorzystania listy w roli wewnętrznego magazynu dla kolejki, ale byłyby one mniej wydajne. Koszt operacji ściągania z lewej z deque wynosi O(1), natomiast w przypadku listy wynosi O(*n*). W przypadku listy, po ściągnięciu z lewej, każdy kolejny element trzeba przesunąć o jedną pozycję w lewo, co obniża wydajność.

ALGORYTM BFS

Co zdumiewające, kod algorytmu przeszukiwania wszerz jest prawie taki sam jak algorytmu przeszukiwania w głąb – w roli `frontier` wystarczy użyć kolejki zamiast stosu. Zastąpienie stosu kolejką zmienia kolejność przeszukiwania stanów i zapewnia, że stany znajdujące się najbliżej początku zostają przeszukane jako pierwsze.

Listing 1.23. generic_search.py kontynuacja

```
def bfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
     List[T]]) -> Optional[Node[T]]:
    # frontier to lista miejsc jeszcze nieodwiedzonych
    frontier: Queue[Node[T]] = Queue()
    frontier.push(Node(initial, None))
    # explored to zbiór miejsc już odwiedzonych

    explored: Set[T] = {initial}

    # kontynuuj przeszukanie, dopóki pozostaje coś do sprawdzenia
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # jeśli znaleźliśmy cel, kończymy
        if goal_test(current_state):
            return current_node
        # sprawdź, gdzie możemy przejść i czego nie sprawdziliśmy
        for child in successors(current_state):
            if child in explored: # pomiń sprawdzone już stany
                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None # wszystko przeszukane i cel nieodnaleziony
```

Jeśli spróbujesz uruchomić funkcję `bfs()`, przekonasz się, że zawsze znajduje ona najkrótsze rozwiązanie dla przekazanego labiryntu. Następujący test dodajemy tuż pod poprzednim w części pliku `if __name__ == "__main__":`, aby można było porównywać wyniki dla tego samego labiryntu.

Listing 1.24. maze.py kontynuacja

```
# Testuj BFS
solution2: Optional[Node[MazeLocation]] = bfs(m.start, m.goal_test,
     m.successors)
if solution2 is None: # Nie znaleziono rozwiązania
    print("No solution found using breadth-first search!")
else:
    path2: List[MazeLocation] = node_to_path(solution2)
    m.mark(path2)
    print(m)
    m.clear(path2)
```

To niesamowite, że można uzyskać tak odmienny efekt, pozostawiając ten sam kod algorytmu i zmieniając jedynie strukturę danych. Poniżej przedstawiamy wynik wywołania funkcji `bfs()` na tym samym labiryncie, na którym wcześniej wywołaliśmy funkcję `dfs()`. Jak widać, droga oznaczona przy użyciu gwiazdek prowadzi bardziej bezpośrednio od początku do celu niż poprzednie rozwiązanie.

```
S  X X
*X
*   X
*XX   X
* X
* X X
*X
*
*  X  X
*********G
```

1.2.5. Algorytm A*

Obieranie cebuli warstwa po warstwie, jak w algorytmie przeszukiwania wszerz, może być bardzo czasochłonne. Podobnie jak BFS, algorytm A* próbuje znaleźć najkrótszą drogę ze stanu początkowego do docelowego. W odróżnieniu od przedstawionej implementacji przeszukiwania BFS algorytm A* wykorzystuje kombinację funkcji kosztów i funkcji heurystycznej do koncentrowania się na drogach, które dają największą szansę na szybkie znalezienie rozwiązania.

Funkcja kosztu, g(*n*), oblicza koszt dojścia do określonego stanu. W przypadku naszego labiryntu byłaby nim liczba poprzednich kroków, przez jakie musieliśmy przejść, aby dotrzeć do danego stanu. Funkcja heurystyczna, h(*n*), zwraca szacowany koszt przejścia z danego stanu do stanu docelowego. Można udowodnić, że jeśli heurystyka h(*n*) jest dopuszczalna, odnaleziona droga będzie optymalna. Dopuszczalna funkcja heurystyczna nigdy nie przeszacowuje kosztu drogi do celu. Przykładem na płaszczyźnie dwuwymiarowej może być odległość w linii prostej, ponieważ linia prosta zawsze jest najkrótszą drogą[1].

Całkowity koszt dla dowolnego analizowanego stanu reprezentuje funkcja f(*n*), która jest prostą kombinacją funkcji g(*n*) i h(*n*). Faktycznie, f(*n*) = g(*n*) + h(*n*). Wybierając ze zbioru `frontier` następny stan do sprawdzenia, algorytm przeszukiwania A* wybiera ten o najmniejszej wartości f(*n*). To odróżnia go od algorytmów BFS oraz DFS.

KOLEJKI PRIORYTETOWE

Aby ułatwiać wybieranie ze zbioru frontier stanu o najniższej wartości f(*n*), algorytm A* opiera ten zbiór na strukturze danych *kolejka priorytetowa*. Kolejka priorytetowa przechowuje elementy w określonej kolejności, aby pierwszy ściągany z niej element zawsze miał najwyższy priorytet (w naszym przypadku elementem o najwyższym priorytecie jest ten o najniższej wartości f(*n)*). Zazwyczaj wiąże się to z wykorzystaniem w roli pojemnika sterty binarnej, co oznacza operacje push o koszcie O(lg *n*) i *pop* o koszcie O(lg *n*).

[1] Dodatkowe informacje o heurystykach można znaleźć w książce Stuarta Russella i Petera Norviga zatytułowanej *Artificial Intelligence: A Modern Approach*. Wyd. 3. Pearson, 2010, s. 94.

Biblioteka standardowa Pythona oferuje funkcje `heappush()` i `heappop()`, które przechowują otrzymaną listę w postaci sterty binarnej. Możemy zaimplementować kolejkę priorytetową, budując cienką otokę wokół tych funkcji biblioteki standardowej. Nasza klasa `PriorityQueue` będzie przypominała klasy stosu typu `Stack` i kolejki typu `Queue`, z metodami `push()` i `pop()` zmodyfikowanymi w taki sposób, aby bazowały one na metodach `heappush()` oraz `heappop()`.

Listing 1.25. generic_search.py kontynuacja

```
class PriorityQueue(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []

    @property
    def empty(self) -> bool:
        return not self._container # not zwraca true dla pustego pojemnika

    def push(self, item: T) -> None:
        heappush(self._container, item) # dodawanie według priorytetu

    def pop(self) -> T:
        return heappop(self._container) # ściąganie według priorytetu

    def __repr__(self) -> str:
        return repr(self._container)
```

Do porównywania priorytetów różnych elementów metody `heappush()` i `heappop()` używają operatora <. Dlatego we wcześniejszej części rozdziału musieliśmy zaimplementować metodę `__lt__()` w klasie `Node`. Porównując ze sobą dwa obiekty `Node`, bazujemy na ich wartościach f(*n*), które są po prostu sumą właściwości `cost` i `heuristic`.

HEURYSTYKI

Heurystyka określa intuicyjny sposób rozwiązywania problemu[2]. W kontekście rozwiązywania labiryntów heurystyki dążą do wybrania najlepszego kolejnego miejsca do przeszukania, które doprowadzi nas do celu. Innymi słowy, służą do domyślania się, które węzły ze zbioru frontier znajdują się najbliżej celu. Jak już wspomnieliśmy, jeśli heurystyka użyta w algorytmie przeszukiwania A* zwraca trafny względny wynik i jest dopuszczalna (nigdy nie przeszacowuje odległości), algorytm A* znajdzie najkrótszą ścieżkę. Heurystyki, które zwracają mniejsze wartości, prowadzą do przeszukiwania większej liczby stanów, natomiast heurystyki bliższe prawdziwej rzeczywistej odległości (ale jej nieprzekraczające, ponieważ wtedy byłyby niedopuszczalne) prowadzą do przeszukania mniejszej liczby stanów. W związku z tym idealne heurystyki maksymalnie zbliżają się do rzeczywistej odległości, nigdy jej nie przekraczając.

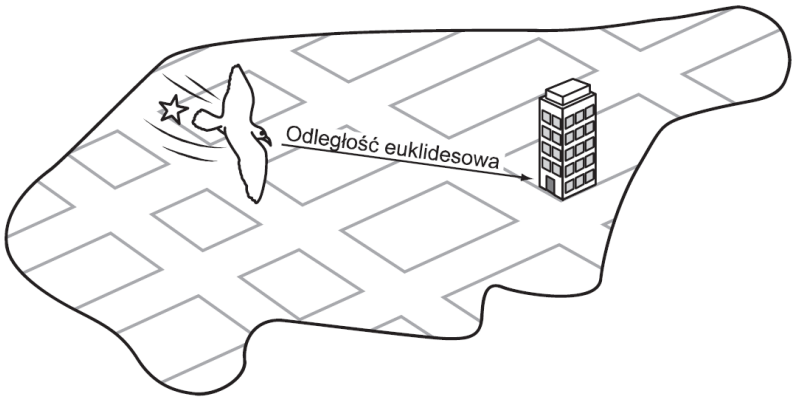
Odległość euklidesowa

Jak się nauczyliśmy na lekcjach matematyki, najkrótsza droga między dwoma punktami to linia prosta. W związku z tym heurystyka oparta na prostej linii zawsze będzie dopuszczalnym rozwiązaniem problemu labiryntu. Do obliczania odległości euklidesowej służy następujący wzór, wyprowadzony z twierdzenia Pitagorasa: `odległość = √((różnica pozycji x)² + (różnica pozycji y)²)`. W przypadku naszych labiryntów różnica *x* stanowi odpowiednik różnicy numerów kolumn między pozycjami w labiryncie, natomiast różnica *y* stanowi odpowiednik różnicy numerów wierszy. To rozwiązanie implementujemy w pliku maze.py.

[2] Dodatkowe informacje o heurystyce A* do szukania ścieżek można znaleźć w rozdziale Heuristics w dokumencie Amita Patel zatytułowanym *Amit's Thoughts on Pathfinding*, http://mng.bz/z7O4.

Listing 1.26. maze.py kontynuacja

```
def euclidean_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = ml.column - goal.column
        ydist: int = ml.row - goal.row
        return sqrt((xdist * xdist) + (ydist * ydist))
    return distance
```



Rysunek 1.6. Odległość euklidesowa jest równa długości prostej linii od początku do celu

`euclidean_distance()` to funkcja, która zwraca inną funkcję. Ten ciekawy wzorzec jest możliwy w Pythonie i innych językach, które traktują funkcje jak typ pierwszoklasowy. Funkcja `distance()` przechwytuje `goal`, czyli pozycję celu przekazywaną do funkcji `euclidean_distance()`. Przechwytywanie oznacza, że funkcja `distance()` może się odwoływać do tej zmiennej za każdym razem, gdy zostaje wywołana (na stałe). Zwracana funkcja wykorzystuje goal do przeprowadzenia własnych obliczeń. Ten wzorzec umożliwia tworzenie funkcji, która wymaga mniejszej liczby parametrów. Zwracana funkcja `distance()` ma tylko jeden parametr określający początkową pozycję w labiryncie i stale „zna" cel.

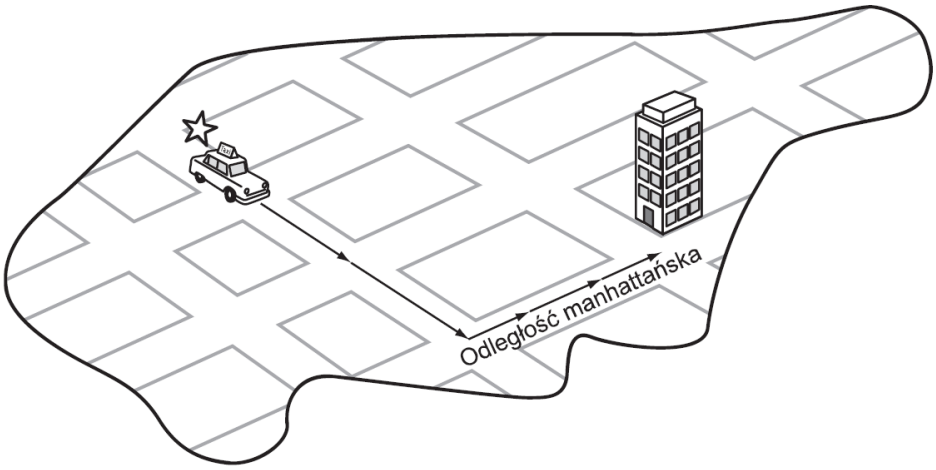
Na rysunku 1.6 zilustrowano odległość euklidesową w kontekście siatki, takiej jak ulice Manhattanu.

ODLEGŁ OŚĆ MANHATTAŃSKA

Odległość euklidesowa to przydatne narzędzie, ale do naszego konkretnego zadania (labiryntu, w którym można się poruszać tylko w jednym z czterech kierunków) możemy użyć czegoś jeszcze lepszego. Nazwa odległości manhattańskiej wywodzi się ze sposobu nawigowania po Manhattanie, najsłynniejszej dzielnicy Nowego Jorku, która jest zbudowana na planie siatki. Aby dostać się z dowolnego miejsca w inne dowolne miejsce na Manhattanie, trzeba przejść określoną liczbę przecznic w poziomie i określoną liczbę przecznic w pionie (na Manhattanie prawie nie ma ulic biegnących po ukośnej). Aby wyznaczyć odległość manhattańską, wystarczy znaleźć różnicę w wierszach między dwoma pozycjami w labiryncie i dodać do niej różnicę w kolumnach. Na rysunku 1.7 pokazano odległość manhattańską.

Listing 2.27. maze.py kontynuacja

```
def manhattan_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = abs(ml.column - goal.column)
        ydist: int = abs(ml.row - goal.row)
        return (xdist + ydist)
    return distance
```



Rysunek 1.7. W odległości manhattańskiej nie ma przekątnych. Droga musi się składać z odcinków równoległych lub prostopadłych

Ponieważ ta heurystyka lepiej odzwierciedla rzeczywisty sposób przechodzenia przez labirynt (poruszamy się pionowo i poziomo, a nie po skosie), jest dokładniejszym niż odległość euklidesowa przybliżeniem prawdziwej odległości między dwoma miejscami w labiryncie. W związku z tym algorytm przeszukiwania A* w połączeniu z odległością manhattańską pozwala na przeszukanie mniejszej liczby stanów w labiryncie niż algorytm przeszukiwania A* w połączeniu z odległością euklidesową. Uzyskane drogi również będą optymalne, ponieważ odległość manhattańska jest dopuszczalna (nigdy nie przeszacowuje odległości) dla labiryntów, w których dozwolone są tylko cztery kierunki ruchu.

ALGORYTM A*

Aby przejść z przeszukiwania BFS do algorytmu przeszukiwania A*, musimy wprowadzić kilka małych zmian. Pierwsza polega na zastosowaniu w roli struktury danych frontier kolejki priorytetowej zamiast kolejki. Dzięki temu będą ściągane węzły najmniejszej wartości f(*n*). Druga zmiana polega na zastosowaniu słownika w roli zbioru explored. Słownik pozwoli nam przechowywać najniższy koszt (g(*n*)) dla każdego odwiedzanego węzła. Ponieważ mamy do czynienia z funkcją heurystyczną, może się zdarzyć, że niektóre węzły zostaną odwiedzone dwukrotnie, jeśli heurystyka jest niespójna. Jeśli koszt dojścia do węzła znalezionego podczas badania nowego kierunku jest mniejszy niż gdy odwiedzaliśmy go poprzednio, będziemy preferować nową ścieżkę.

Dla uproszczenia, funkcja `astar()` nie zawiera parametru z funkcją obliczającą koszt. Zamiast tego przyjmujemy założenie, że każdy krok w naszym labiryncie ma koszt 1. Każdemu nowemu węzłowi przypisujemy koszt bazujący na tym prostym wzorze, jak również heurystyczną ocenę bazującą na nowej funkcji `heuristic()` przekazywanej w postaci argumentu do funkcji przeszukiwania. Poza tymi zmianami funkcja `astar()` jest zaskakująco podobna do funkcji `bfs()`. Porównaj je ze sobą.

Listing 1.28. generic_search.py

```
def astar(initial: T, goal_test: Callable[[T], bool], successors:
     Callable[[T], List[T]], heuristic: Callable[[T], float]) ->
     Optional[Node[T]]:
    # frontier to lista nieodwiedzonych jeszcze miejsc
    frontier: PriorityQueue[Node[T]] = PriorityQueue()
    frontier.push(Node(initial, None, 0.0, heuristic(initial)))
    # explored to zbiór miejsc już odwiedzonych
    explored: Dict[T, float] = {initial: 0.0}
```

```
    # kontynuuj przeszukanie dopóki pozostaje coś do sprawdzenia
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # jeśli znaleźliśmy cel, kończymy
        if goal_test(current_state):
            return current_node
        # sprawdź, gdzie możemy przejść i czego nie sprawdziliśmy
        for child in successors(current_state):
            new_cost: float = current_node.cost + 1 # 1 dla siatki,
            # bardziej zaawansowane przypadki potrzebują funkcji kosztu

            if child not in explored or explored[child] > new_cost:
                explored[child] = new_cost
                frontier.push(Node(child, current_node, new_cost,
     heuristic(child)))
    return None # wszystko przeszukane i cel nieodnaleziony
```

Gratulacje. Jeśli dotarłeś do tego miejsca, nie tylko się nauczyłeś rozwiązywać labirynt, ale również poznałeś wybrane uogólnione funkcje przeszukiwania, które można wykorzystywać do wielu celów. Algorytmy DFS i BFS stosujemy na mniejszych zbiorach danych i stanów, w których wydajność nie jest kluczowa. W niektórych sytuacjach algorytm DFS działa wydajniej niż BFS, ale algorytm BFS ma tę zaletę, że zawsze zwraca optymalną drogę. Co ciekawe, jedyna różnica w implementacjach algorytmów BFS i DFS polega na użyciu kolejki zamiast stosu w roli zbioru stanów do przeszukania. Nieco bardziej skomplikowany algorytm przeszukiwania A*, w połączeniu z dobrą, spójną i dopuszczalną funkcją heurystyczną, nie tylko dostarcza optymalne drogi, ale jest dużo wydajniejszy niż BFS. A ponieważ wszystkie trzy funkcje zostały zaimplementowane w uogólniony sposób, aby wykorzystać je do niemal dowolnego wyszukiwania, wystarczy użyć polecenia import `generic_search`.

Sprawdźmy działanie funkcji `astar()` na tym samym labiryncie, dodając następujący kod do sekcji testowania w pliku maze.py.

Listing 1.29. maze.py kontynuacja

```
# Testuj A*
distance: Callable[[MazeLocation], float] = manhattan_distance(m.goal)
solution3: Optional[Node[MazeLocation]] = astar(m.start, m.goal_test,
     m.successors, distance)
if solution3 is None: #Nie znaleziono rozwiązania
    print("No solution found using A*!")
else:
    path3: List[MazeLocation] = node_to_path(solution3)
    m.mark(path3)
    print(m)
```

Co ciekawe, wynik będzie się nieco różnił od funkcji `bfs()`, mimo że zarówno `bfs()`, jak i `astar()` znajdują optymalne drogi (tej samej długości). W wyniku zastosowania heurystyki funkcja `astar()` zmierza po przekątnej prosto do celu. W rezultacie przeszukuje mniej stanów niż `bfs()`, co skutkuje większą wydajnością. Aby to udowodnić, dodaj licznik stanów do każdej funkcji.

```
S** X X
X**
  * X
 XX*  X
 X*
 X**X
X ****
   *
  X * X
   **G
```

1.3. Misjonarze i kanibale

Trzech misjonarzy i trzech kanibali siedzi na zachodnim brzegu rzeki. Mają tylko jeden kajak, w którym mieszą się maksymalnie dwie osoby, i wszyscy muszą przeprawić się na drugi brzeg rzeki. Na żadnym brzegu nie może w żadnym momencie znajdować się więcej kanibali niż misjonarzy, w przeciwnym razie kanibale zjedliby misjonarzy. Dodatkowo, aby kajak mógł dotrzeć na drugi brzeg, musi mieć przynajmniej jedną osobę na pokładzie. Jaka kolejność przeprawy przez rzekę sprawi, że wszyscy bezpiecznie dostaną się na drugi brzeg? Problem został zilustrowany na rysunku 1.8.

1.3.1. Modelowanie problemu

Problem zamodelujemy przy użyciu struktury, która śledzi brzeg zachodni. Ilu misjonarzy i kanibali stoi na tym brzegu? Czy łódka znajduje się przy zachodnim brzegu? Na podstawie tych informacji możemy wywnioskować, co znajduje się na wschodnim brzegu, ponieważ przyjmujemy, że wszystko to, czego nie ma na brzegu zachodnim, znajduje się na brzegu wschodnim.

Zaczynamy od utworzenia małej pomocniczej zmiennej do śledzenia maksymalnej liczby misjonarzy i kanibali. A później zdefiniujemy główną klasę.

Listing 1.30. missionaries.py

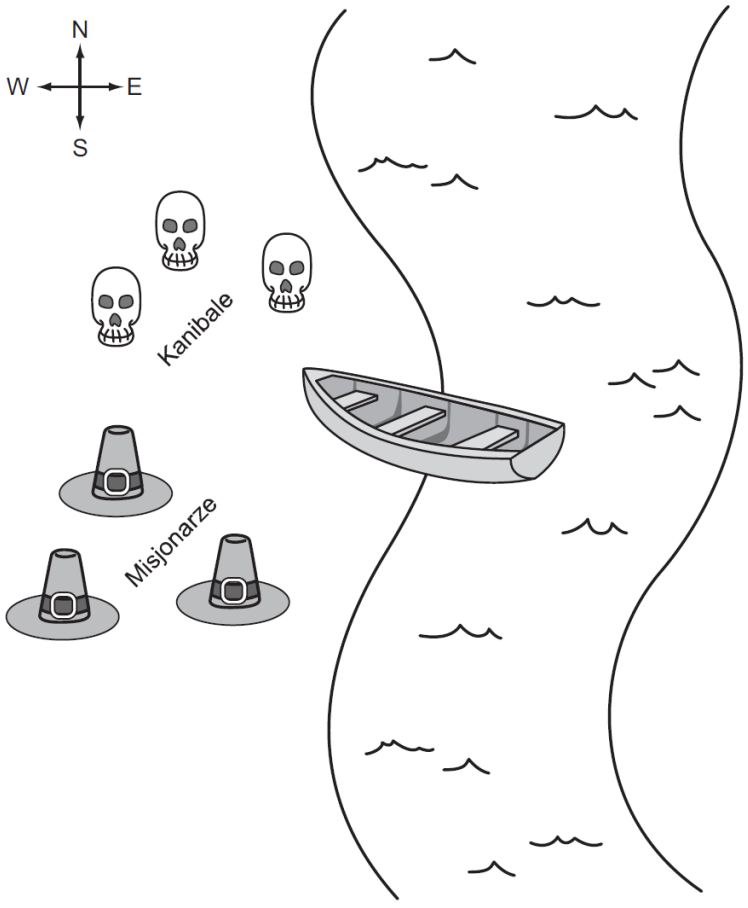
```
from __future__ import annotations
from typing import List, Optional
from generic_search import bfs, Node, node_to_path

MAX_NUM: int = 3
```

```
class MCState:
    def __init__(self, missionaries: int, cannibals: int, boat: bool) ->
     None:
        self.wm: int = missionaries # ilu misjonarzy na zachodnim (W) brzegu
        self.wc: int = cannibals # ilu kanibali na zachodnim brzegu
        self.em: int = MAX_NUM - self.wm # ilu misjonarzy na wschodnim (E) brzegu
        self.ec: int = MAX_NUM - self.wc # ilu kanibali na wschodnim brzegu
        self.boat: bool = boat # czy łódka na zachodnim brzegu

    def __str__(self) -> str:

        return ("On the west bank there are {} missionaries and {}
     cannibals.\n" # Na zachodnim brzegu jest {} misjonarzy i {} kanibali
                  "On the east bank there are {} missionaries and {}
     cannibals.\n" # Na wschodnim brzegu jest {} misjonarzy i {} kanibali
                  # Łódka jest na {} brzegu
                  "The boat is on the {} bank.") \
               .format(self.wm, self.wc, self.em, self.ec, ("west" if self.boat
     else "east"))
```



Rysunek 1.8. Misjonarze i kanibale muszą użyć jednego kajaka do przeprawienia wszystkich z zachodniego na wschodni brzeg rzeki. Jeśli w dowolnym momencie kanibale będą przewyższać liczebnie misjonarzy, to ich zjedzą

Klasa `MCState` jest inicjalizowana na podstawie liczby misjonarzy i kanibali na zachodnim brzegu, a także położenia łodzi. Oferuje również metodę do prezentowania aktualnego stanu na ekranie, co przyda się później podczas wyświetlania rozwiązania problemu.

Ponieważ chcemy korzystać z istniejących funkcji przeszukiwania, musimy zdefiniować funkcję goal_test do testowania, czy stan jest stanem docelowym, i funkcję successors do znajdowania potencjalnych następców dowolnego stanu. Funkcja testowania celu jest, jak w przypadku problemu rozwiązywania labiryntu, dość prosta. Celem jest osiągnięcie prawowitego stanu, w którym wszyscy misjonarze i kanibale znajdują się na wschodnim brzegu rzeki. Dodajemy odpowiednią metodę do klasy `MCState`.

Listing 1.31. missionaries.py kontynuacja

```
    def goal_test(self) -> bool:
        return self.is_legal and self.em == MAX_NUM and self.ec == MAX_NUM
```

Aby utworzyć funkcję znajdującą następców, trzeba przejść przez wszystkie możliwe ruchy, które można wykonać z jednego brzegu na drugi, a następnie sprawdzić, czy każdy z nich będzie skutkował prawidłowym stanem. Jak pamiętamy, w prawidłowym stanie liczba kanibali nie jest większa niż liczba misjonarzy na żadnym z brzegów. W związku z tym możemy zdefiniować pomocniczą właściwość (w postaci `metody is_legal` klasy `MCState`) do sprawdzania, czy stan jest prawowity.

Listing 1.32. missionaries.py kontynuacja

```
@property
def is_legal(self) -> bool:
    if self.wm < self.wc and self.wm > 0:
        return False
    if self.em < self.ec and self.em > 0:
        return False
    return True
```

Z uwagi na przejrzystość kod funkcji do znajdowania następców jest nieco rozwlekły. Próbujemy dodać wszelkie możliwe kombinacje jednej lub dwóch osób przeprawiających się przez rzekę z brzegu, przy którym aktualnie jest zacumowana łódka. Po dodaniu wszystkich możliwych ruchów, korzystając z wyrażenia listowego, wybieramy tylko te prawowite. Jak poprzednio, funkcja będzie metodą klasy `MCState`.

Listing 1.33. missionaries.py kontynuacja

```
    def successors(self) -> List[MCState]:
        sucs: List[MCState] = []
        if self.boat: # łódka na zachodnim brzegu
            if self.wm > 1:
                sucs.append(MCState(self.wm - 2, self.wc, not self.boat))
            if self.wm > 0:
                sucs.append(MCState(self.wm - 1, self.wc, not self.boat))
            if self.wc > 1:
                sucs.append(MCState(self.wm, self.wc - 2, not self.boat))
            if self.wc > 0:
                sucs.append(MCState(self.wm, self.wc - 1, not self.boat))
            if (self.wc > 0) and (self.wm > 0):
                sucs.append(MCState(self.wm - 1, self.wc - 1, not self.boat))
        else: # łódka na wschodnim brzegu
```

```
        if self.em > 1:
            sucs.append(MCState(self.wm + 2, self.wc, not self.boat))
        if self.em > 0:
            sucs.append(MCState(self.wm + 1, self.wc, not self.boat))
        if self.ec > 1:
            sucs.append(MCState(self.wm, self.wc + 2, not self.boat))
        if self.ec > 0:
            sucs.append(MCState(self.wm, self.wc + 1, not self.boat))
        if (self.ec > 0) and (self.em > 0):
            sucs.append(MCState(self.wm + 1, self.wc + 1, not self.boat))
    return [x for x in sucs if x.is_legal]
```

1.3.2. Rozwiązywanie problemu

Mamy już wszystkie komponenty potrzebne do rozwiązania zadania. Jak pamiętamy, gdy korzystaliśmy z funkcji przeszukiwania `bfs()`, `dfs()` i `astar()`, otrzymywaliśmy obiekt `Node`, który na końcu za pomocą funkcji `node_to_path()` przekształcaliśmy w listę stanów prowadzących do rozwiązania. Nie mamy jeszcze metody przekształcania tej listy w zrozumiałą, gotową do wyświetlenia sekwencję kroków reprezentujących rozwiązanie problemu misjonarzy i kanibali.

Funkcja `display_solution()` przekształca ścieżkę rozwiązania w wynik do wyświetlenia, czyli rozwiązanie zadania zrozumiałe dla człowieka. Jej działanie polega na przechodzeniu przez wszystkie stany w ścieżce rozwiązania i jednoczesnym śledzeniu stanu poprzedniego. Wyznacza różnicę między stanem poprzednim a aktualnym w celu sprawdzenia, ilu misjonarzy oraz kanibali zostało przeprawionych na drugą stronę rzeki i w jakim kierunku.

Listing 1.34. missionaries.py kontynuacja

```
def display_solution(path: List[MCState]):
    if len(path) == 0: # sanity check
        return
    old_state: MCState = path[0]
    print(old_state)
    for current_state in path[1:]:
        if current_state.boat:
            # {} misjonarzy i {} kanibali przeprawionych ze wschodu na zachód
            print("{} missionaries and {} cannibals moved from the east bank
     to the west bank.\n"
                  .format(old_state.em - current_state.em, old_state.ec -
     current_state.ec))
        else:
            # {} misjonarzy i {} kanibali przeprawionych z zachodu na wschód
            print("{} missionaries and {} cannibals moved from the west bank
     to the east bank.\n"
                  .format(old_state.wm - current_state.wm, old_state.wc -
     current_state.wc))
        print(current_state)
        old_state = current_state
```

W funkcji `display_solution()` wykorzystujemy to, że klasa `MCState` umie wyświetlić czytelną reprezentację za pomocą metody `__str__()`.

Na zakończenie musimy faktycznie rozwiązać problem misjonarzy i kanibali. Do tego celu możemy użyć jednej z napisanych wcześniej funkcji przeszukiwania, ponieważ zaimplementowaliśmy je w sposób uogólniony. To rozwiązanie wykorzystuje funkcję `bfs()` (ponieważ funkcja `dfs()` wymagałaby oznaczenia referencyjnie różnych stanów tej samej wartości jako równych, natomiast funkcja `astar()` wymagałaby heurystyki).

Listing 1.35. missionaries.py kontynuacja

```
if __name__ == "__main__":
    start: MCState = MCState(MAX_NUM, MAX_NUM, True)
    solution: Optional[Node[MCState]] = bfs(start, MCState.goal_test,
     MCState.successors)
    if solution is None: # Nie znaleziono rozwiązania
        print("No solution found!")
    else:
        path: List[MCState] = node_to_path(solution)
        display_solution(path)
```

Wspaniale jest zobaczyć, jak uniwersalne są nasze uogólnione funkcje przeszukiwania. Można łatwo je zaadoptować do rozwiązywania wielu różnych problemów. Wyświetlony wynik powinien być podobny do następującego (w skróconej postaci):

```
On the west bank there are 3 missionaries and 3 cannibals.
On the east bank there are 0 missionaries and 0 cannibals.
The boast is on the west bank.
0 missionaries and 2 cannibals moved from the west bank to the east bank.
On the west bank there are 3 missionaries and 1 cannibals.
On the east bank there are 0 missionaries and 2 cannibals.
The boast is on the east bank.
0 missionaries and 1 cannibals moved from the east bank to the west bank.
...
On the west bank there are 0 missionaries and 0 cannibals.
On the east bank there are 3 missionaries and 3 cannibals.

The boast is on the east bank.
```

1.4. Praktyczne zastosowania

Wyszukiwanie ma zastosowanie we wszystkich aplikacjach użytkowych. W niektórych jest elementem kluczowym (Google Search, Spotlight, Lucene), w innych służy do korzystania ze struktur, na których jest oparty magazyn danych. Znajomość odpowiedniego algorytmu wyszukiwania do zastosowania na wybranej strukturze danych jest niezbędna do uzyskania wysokiej wydajności. Na przykład stosowanie wyszukiwania liniowego, zamiast binarnego, na posortowanej strukturze danych byłoby bardzo kosztowne.

A* to jeden z najczęściej wdrażanych algorytmów znajdowania drogi. Większą popularnością cieszą się jedynie algorytmy, które wykonują wstępne obliczenia w obszarze wyszukiwania. W przypadku ślepego przeszukiwania nie znaleziono jeszcze niezawodnego algorytmu, który lepiej niż A* sprawdzałby się we wszystkich scenariuszach. To sprawia, że jest on kluczowym komponentem wielu systemów, począwszy od programów do planowania tras, po znajdowanie najkrótszego sposobu parsowania języka programowania. Większość oprogramowania do wyświetlania tras na mapie (jak Google Maps) wykorzystuje do nawigacji algorytm Dijkstra (którego wariantem jest algorytm A*). Dodatkowe informacje o algorytmie Dijkstra można znaleźć w rozdziale 4. Za każdym razem, gdy fikcyjna postać w grze znajduje najkrótszą drogę z jednego końca planszy na drugi bez ludzkiej interwencji prawdopodobnie jest wykorzystywany algorytm A*.

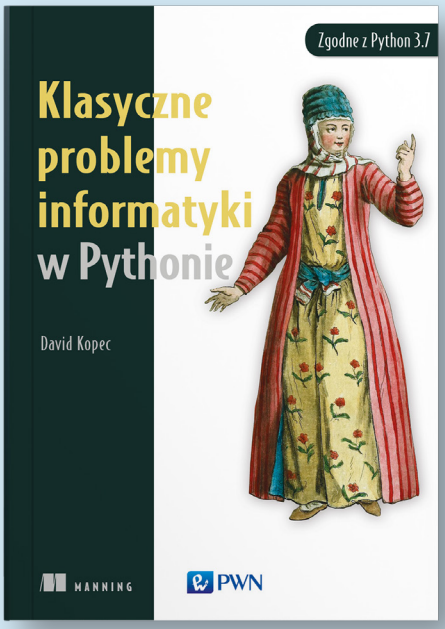
Algorytmy przeszukiwania wszerz i w głąb często są podstawą dla bardziej złożonych metod wyszukiwania, takich jak Uniform Cost Search czy algorytm z nawrotami (które pokażemy w kolejnym rozdziale). Algorytm przeszukiwania wszerz często wystarcza do znajdowania najkrótszej ścieżki w dość niewielkim grafie. A ponieważ jest on tak podobny do algorytmu A*, łatwo można go zastąpić algorytmem A*, jeśli istnieje dobra heurystyka dla większego grafu.

1.5. Ćwiczenia

1. Pokaż, że wyszukiwanie binarne działa wydajniej niż liniowe, tworząc listę miliona liczb i mierząc, ile czasu funkcje `linear_contains()` i `binary_contains()` zdefiniowane w tym rozdziale potrzebują do znalezienia różnych liczb z listy.

2. Dodaj licznik do funkcji `dfs()`, `bfs()` i `astar()`, aby sprawdzić, ile stanów przeszukuje każda z tych funkcji dla tego samego labiryntu. Określ wartości liczników dla stu różnych labiryntów, aby uzyskać wyniki istotne statystycznie.

3. Znajdź rozwiązanie problemu misjonarzy i kanibali dla innych początkowych liczb misjonarzy i kanibali. Wskazówka: być może trzeba będzie dodać nadpisania `metod __eq__()` i `__hash__()` do klasy `MCState`.



Ebook pochodzi z książki:

Klasyczne problemy informatyki w Pythonie

Autor: David Kopec

Wydawnictwo Naukowe PWN

SPRAWDŹ

Zainteresowały Cię nasze książki?

Znajdziesz je na:

IBUK libra

Ibuk Libra to czytelnia on-line czynna całą dobę. Dostępne w niej są tysiące e-booków oraz e-czasopism z niemal każdej dziedziny. Do IBUKA Libry możesz zalogować się z dowolnego miejsca, o każdej porze. Korzystanie z IBUKA Libry jest bezpłatne - poproś o dostęp w swojej bibliotece.

Przejdź do IBUK Libra

IBUK

IBUK.pl jest platformą pozwalającą kupować i wypożyczać e-booki. Można je wypożyczać zarówno pojedynczo - już od 4,92 PLN na dobę oraz w abonamentach - ceny zaczynają się nawet od 19,90 PLN miesięcznie. W ofercie dostępne sa także audiobooki.

Przejdź do Ibuk.pl

PWN KSIĘGARNIA INTERNETOWA

Księgarnia Internetowa PWN oferuje szeroki zakres publikacji: podręczniki akademickie, książki naukowe i popularnonaukowe, słowniki języka polskiego i słowniki języków obcych. Znajdziesz w niej zarówno publikacje papierowe, jak i książki w wersji elektronicznej e-booki i audiobooki.

Przejdź do Księgarni Internetowej PWN

Śledź nas na Facebooku:

PWN STEM