algograf

Lab 1: Rozgrzewka

Spis treści

W ramach laboratorium należy zaimplementować program (lub kilka jego wariantów) rozwiązujący zadanie “przewodnik turystyczny”.

Zadanie

Dany jest graf nieskierowany *G = (V,E)*, funkcja *c: E -> N* dająca wagi krawędziom, oraz wyróżnione wierzchołki *s* i *t*.
Szukamy scieżki z *s* do *t* takiej, że najmniejsza waga krawędzi na tej ścieżce jest jak największa.
Należy zwrócić najmniejszą wagę krawędzi na znalezionej ścieżce.
(W praktyce ścieżki szukamy tylko koncepcyjnie.)

Podejścia algorytmiczne:

1. wykorzystanie struktury find-union,
2. wyszukiwanie binarne + przegląd grafu metodami BFS/DFS,
3. algorytm a’la Dijkstra.

W ramach laboratorium należy zaimplementować jeden, dowolny z tych algorytmów (a jeśli zostanie czas, to także kolejne).

Proponowana kolejność prac

W ramach laboratorium proponujemy realizować zadanie w następujących krokach.

1. Napisz skrypt wczytujący przykładowy graf i wypisujący krawędzie.
    i. graphs-lab1.zip – grafy testowe
    ii. dimacs.py – funkcja wczytująca grafy
    iii. Więcej o wczytywaniu grafów znajdziesz w tej sekcji.

iv. **UWAGA**: Przyjmujemy, że w grafach testowych wierzchołek *s* ma numer 1 a wierzchołek *t* ma numer 2.

2. Rozwiązanie oparte of find-union.
   i. Zaimplementuj funkcje `find` i `union` realizujące zbiory rozłączne (przetestuj je, np. z poziomu konsoli).
   ii. Zaimplementuj sortowanie krawędzi grafu malejąco.
   iii. Użyj powyższych do zaimplementowania całego rozwiązania.
3. Rozwiąznie oparte o wyszukiwanie binarne + BFS/DFS.
   i. Zaimplementuj konwersję grafu na listy (lub zbiory) sąsiedztwa.
      a. W plikach wejściowych wierzchołki są indeksowane od 1 ale dla wygody rozważ konwersję na indeskowanie od 0.
   ii. Zaimplementuj algorytm DFS operujący na listach sąsiedztwa.
   iii. Zaadoptuje DFS na potrzeby rozwiązania opartego o wyszukiwanie binarne.
      a. Albo możesz za każdym razem tworzyć graf wejściowy do DFS-a tak, żeby posiadał tylko odpowiednie krawędzie.
      b. Albo możesz zmodyfikować DFS tak, żeby sam wybierał jedynie odpowiednie krawędzie.
   iv. Zrealizuj wyszukiwanie binarne które powinno dać całe rozwiązanie.
   v. (Opcjonalnie) Zamień DFS na BFS.
4. Zrealizuj rozwiązanie oparte o algorytm Dijkstry.
   i. Zaimplementuj całe rozwiązanie.

Wczytywanie grafów

Grafy są zapisane w formacie DIMACS ascii (+ modyfikacje pozwalające zapisywać wagi).

- Plik zaczyna się od 0 lub więcej linii zaczynających się od znaku `c` i odstępu.
  Są to komentarze, w których można umieścić dodatkowe informacje o grafie.
- Następnie występuje linia postaci `p edge V E` gdzie:
  - `V` to liczba wierzchołków w grafie,
  - `E` to liczba krawędzi.
- Następnie występuje `E` linii postaci `e X Y C` oznaczających krawędź nieskierowaną
  - między wierzchołkami o numerach `X` i `Y`
  - i o wadze `C`.

`dimacs.py` dostarcza funkcję `loadWeightedGraph(name)`, która wczytuje graf w formacie DIMACS ascii i zwraca parę postaci `(V,L)`, gdzie:

- `V` to liczba wierzchołków (numerowane od `1` do `V`),
- `L` to lista krawędzi w postaci trójek postaci `(X,Y,C)`.

```
from typing import Tuple

V: int # Liczba wierzchołków
L: Tuple[int, int, int] # List krawędzi
V, L = loadWeightedGraph("g1") # Wczytanie grafu z pliku "g1"
```

Przykład:

Poniższy kod wypisuje jakie krawędzie występują w grafie `g1` :

```
from dimacs import *

(V,L) = loadWeightedGraph( "g1" )        # wczytaj graf
for (x,y,c) in L:                        # przeglądaj krawędzie z listy
  print( "krawedz miedzy", x, "i", y,"o wadze", c )   # wypisuj
```

Podstawy Pythona - pomocne fragmenty kodu

Poniżej przedstawione są przydatne fragmenty kodu w Pythonie realizujące typowe zadania.

Stworzenie tablicy (listy) o zadanym rozmiarze, odczytanie rozmiaru

```
T = [ None ] * 10   # Tworzy listę o 10 obiektach "None"
A = [ 0 ] * 100     # Tworzy listę wypełnioną setką zer
B = list(range(10))       # Tworzy listę [0,1,2, ... 9]

l = len(A)          # odczytanie długości listy
```

Listy są indeksowane od 0.

Stworzenie funkcji

```
def function(x, y, z):
  return (x*y)+z

print(function(1,2,3))
```

Pętle i instrukcje warunkowe

```
i = 1
while i < 10:    # wykonuj dopóki i < 10
  print(i)
```

```
  i *= 2
  if i == 4:
    print( "Woo!" )
  elif i == 8:
    print("Ho ho ho!")
  else:
    print("Boooring!")

for i in range(10): # odpowiednik typowej konstrukcji pętli for( int i = 0; i < 10
  print(i)

# można interować też po bardziej zawiłych listach (i innych obiektach takich jak
L = [(1, 2), (3, 4), (5, 6)]
for x, y in L:
  print(x, y)
```

Posortowanie tablicy

```
A = [3, 1, 2]       # przykładowa tablica
B = sorted(A)       # stwórz posortowaną KOPIĘ tablicy A
print(A)            # wypisze [3,1,2]
print(B)            # wypisze [1,2,3]

A.sort()            # posortuj tablicę A
print(A)            # wypisze [1,2,3]

T = [None] * 10     # Tworzy tablicę o 10 obiektach "None"
A = [0] * 100       # Tworzy tablicę wypełnioną 100 zer

# sortowanie po elemencie tuple
A = [("ma", 2), ("kota", 3), ("Ala", 1)]

def second(x): return x[1]      # stwórz funkcję wybierającą element, po którym na

A.sort(key=second)              # posortuj używając funkcji second do wyliczenia k
print(A)                        # wypisz wynik

# j.w. używając funkcji anonimowych (lambda)
A.sort(key=lambda x: x[1])      # lambda to słowo kluczowe rozpoczynające
                                # funkcję anonimową ( lambda argumenty : zwracana
```

List comprehensions

Tworzenie list można sobie uprościć stosując zapisz zbliżony do matematycznego

```
A = [x**2 for x in range(5)]    # tworzy listę [0,1,4,9,16]
B = [(i, 0) for i in range(V+1)] # tworzy listę par [(0,0), (1,0), (2,0), ... ] za
```

Stworzenie zbioru elementów

```
X = set()                       # pusty zbiór
X.add(1)                        # dodaj 1 do zbioru
Y = set([3,1,2])                # stwórz zbiór z elementów listy
Y = {3, 1, 2}                   # stwórz zbiór bezpośrednio z elementów

for y in Y:
  print(y)                      # wypisz elementy zbioru

if 2 in Y:
  print("liczba 2 jest w zbiorze Y")
```

Wskazówki implementacyjne

Listy/zbiory sąsiedztwa

Stworzenie reprezentacji grafu przez zbiory sąsiedztwa

Możemy reprezentować graf jako listę indeksowaną numerami wierzchołków, której elementami są zbiory krawędzi wychodzących z danego wierzchołka. Razem z wierzchołkiem przechowujemy wagę powiązanej krawędzi jako krotkę `(v, c)`.

```
V, L = loadWeightedGraph(...)
G = [set() for _ in range(V)]   # tworzymy po jednym pustym zbiorze na wierzchołek

# # wersja prostsza koncepcyjnie, ale nieidiomatyczna w Pythonie
# G = [None]*(V+1)
# for i in range(V+1):
#  G[i] = set()

for u, v, c in L:
  u -= 1                # zmiana numerowania wierzchołków z "od 1" na "od 0"
  v -= 1
  G[u].add((v, c))      # dodaj krawędź z u do v
  G[v].add((u, c))      # dodaj krawędź z v do u
```

Stworzenie reprezentacji grafu przez listy sąsiedztwa

```
G = [[] for i in range(V)]     # tworzymy po jednej pustej liście na wierzchołek

for u, v, c in L:
  u -= 1                        # zmiana numerowania wierzchołków z "od 1" na "od
  v -= 1
  G[u].append((v, c))           # dodaj krawędź z x do y
  G[v].append((u, c))           # dodaj krawędź z y do x
```

Wypisanie grafu

```
for u in range(1, V+1):   # przeglądamy tylko prawdziwe wierzchołki, od 1 do V
  s = f"{u}: "
  for v in G[u]:
    s += f"{v}, "
  print(s)
```

DFS/BFS

Na nasze potrzeby możemy założyć, że DFS i BFS zwrócą wartość typu `bool` sygnalizującą czy istnieje ścieżka od wierzchołka źródłowego do docelowego.

Implementacja DFS

Algorytm DFS można zaimplementować w naturalny rekurencyjny sposób.
Należy jednak pamiętać o tym, że trzeba śledzić, które wierzchołki zostały odwiedzone.
W tym celu można sobie stworzyć np. listę `visited`, która jest przenoszona przez kolejne wywołania rekurencyjne.
UWAGA: Możliwe, że przekroczymy limit rekurencji w Pythonie. Wtedy można go zmodyfikować korzystając z `sys.setrecursionlimit(new_limit)` albo zaimplementować DFSa iteracyjnie (z własnym stosem).

```
def DFS(G, s, ...):     # DFS w grafie G z wierzchołka s
  V = len(G)            # liczba wierzchołków w grafie (zakładając powyższą implem
  visited = [False] * V
  ...
  DFSVisit(G, s, visited, ...)


def DFSVisit(G, s, visited, ...):  # rekurencyjna funkcja realizująca DFS
  visited[s] = True
  ...
  # przeglądaj wierzchołki osiągalne z s
  # przed wejściem do danego wierzchołka sprawdź, czy jego pole visited jest True
```

Implementacja BFS

W tym zadaniu korzystanie z BFS nie jest konieczne, ale jeśli ktoś chce, to może przydać się kolejka.

```
from collections import deque   # użyj odpowiedniej biblioteki

Q = deque()         # stwórz pustą kolejkę

Q.append("Ala")     # dopisz coś do kolejki
```

```
Q.append("ma")      # j.w.
Q.append("kota")    # j.w.

Q.popleft()         # wyjmij pierwszy element z kolejki (zwróci "Ala")

if Q:
  print("Q is not empty")
```

Implementacja algorytmu a'la Dijkstra

Aby zrealizować algorytm podobny do algorytmu Dijkstry, będziemy potrzebować kolejki priorytetowej.

```
from queue import PriorityQueue

Q = PriorityQueue()          # stwórz pustą kolejkę

Q.put(10)                    # wstaw 10 do kolejki
Q.put(5)                     # wstaw 5  do kolejki
Q.put(20)                    # wstaw 20 do kolejki
Q.get()                      # wyjmij z kolejki (da 5)

Q.empty()                    # sprawdza czy kolejka jest pusta
```

W naszym przypadku potrzebujemy kolejki wyjmującej elementy od największych do najmniejszych (czyli inaczej niż w Pythonie).
Najłatwiej to osiągnąć wstawiając liczby ze znakiem ujemnym.
Będziemy też potrzebować, żeby kolejka przechowywała nie tylko priorytety, ale numery wierzchołków.
W tym celu możemy wkładać do niej krotki, gdzie na pierwszej pozycji jest priorytet and drugiej numer wierzchołka.

UWAGA: `queue.PriorityQueue` nie wspiera operacji aktualizowania wag już znajdujących się w niej elementów przez co będziemy musieli wkładać do niej wielokrotnie ten sam wierzchołek wiekrotnie z różnymi wagami.