PPPD - Lab. 07

Zadanie punktowane, lab 07, 2020/2021

Temat: Problem komiwojażera

Treść zadania

W zadaniu nie można korzystać z `append`*,* `in` *(w rozumieniu* `5 in [1, 2, 3, 4, 5]`*, w pętli* `for` *można spokojnie używać),* `slice`*,* `random.shuffle()` *i indeksowania ujemnego.*

Wstęp

Zadanie traktuje o rozwiązywaniu problemu komiwojażera. Problem ten możemy sformułować tak: mamy dane n miast i odległości pomiędzy nimi (odległości można np. policzyć jako odległość Euklidesową pomiędzy współrzędnymi (x, y) tychże miast). W jakiej kolejności odwiedzać miasta, tak aby odwiedzić każde miasto raz, wrócić na koniec do miasta startowego (jest to jedyne miasto które odwiedzamy niejako dwa razy), a długość takiego cyklu (suma wszystkich pokonanych odległości) była najmniejsza? Tak naprawdę nie da się uzyskać poprawnej odpowiedzi inaczej, niż sprawdzając wszystkie $n!$ możliwości. Jednak takie rozwiązanie jest bardzo drogie czasowo, dlatego stosuje się także algorytmy przybliżone. W tym zadaniu będziesz musiał zaimplementować zarówno rozwiązanie dokładne, jak i przybliżone (elementy algorytmu genetycznego).

Spójrzmy na rysunek. Przypiszmy miastom numery: 0 - Warszawa, 1 - Kraków, 2 - Wrocław, 3 - Poznań, 4 - Gdańsk. Przykładowy cykl po którym miasta będą odwiedzane można zapisać jako `[3,4,2,0,1]`. Oznacza to, że pokonamy $304 + 486 + 341 + 299 + 403$ km. Taka lista `[3,4,2,0,1]` jest w kodzie nazywana ścieżką.

Taka scieżka może być postrzegana jako sekwencja genów: być może lekka modyfikacja takiej ścieżki (tzw. mutacja) spowoduje polepszenie wyniku (krótszy cykl). Tak samo można skrzyżować dwie ścieżki (tak jak krzyżuje się osobniki w naturze), aby uzyskać nową ścieżkę, która być może łączy najlepsze cechy swoich rodziców.

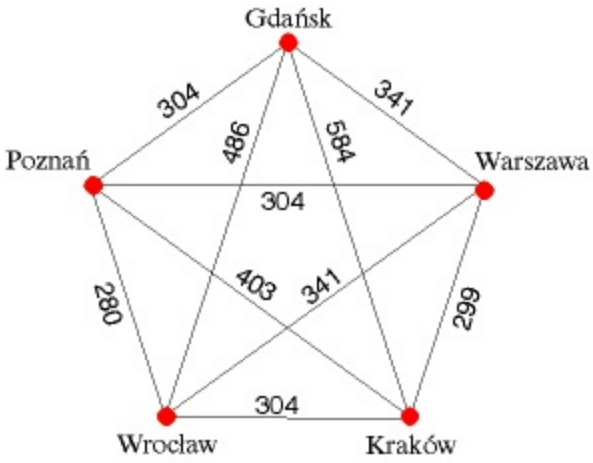
Zadanie

Do zadania dołączony jest kod źródłowy, który należy uzupełnić (uzupełnić ciała funkcji oraz fragment `main()`). Nagłówki funkcji oraz elementy funkcji `main()` są już napisane. Każda funkcja ma docstringa, który mówi, co funkcja ma robić, a w `main()` w komentarzach są pseudokody tego, co należy uzupełnić.

W funkcji dokonującej mutacji zalecane jest użycie funkcji `random.sample()`. Funkcja ta służy do losowania z zadanej listy paru elementów bez zwracania. Przykładowo:

```
print(random.sample([11,22,33,44,55], 2))
[55, 33] # niekoniecznie musi być zachowana kolejność, ale to nam nie przeszkadza
```

Dla algorytmu dokładnego (ostatni etap) posłuż się następującym pseudokodem (nie trzeba go w 100% rozumieć). Pseudokod pokazuje, jak wygenerować wszystkie permutacje w sposób iteracyjny. Moment gdy mamy nową permutację jest oznaczony poprzez `output(A)`. Tu należy napisać własny kod, który wykonuje naszą logikę dla danej permutacji miast.



Rysunek 1: Problem komiwojażera

```
procedure generate(n : integer, A : array of any):
    //c is an encoding of the stack state.
    //c[k] encodes the for-loop counter for when generate(k - 1, A) is called
    c : array of int

    for i := 0; i < n; i += 1 do
        c[i] := 0
    end for

    output(A)

    //i acts similarly to the stack pointer
    i := 0;
    while i < n do
        if  c[i] < i then
            if i is even then
                swap(A[0], A[i])
            else
                swap(A[c[i]], A[i])
            end if
            output(A)
            //Swap has occurred ending the for-loop.
            //Simulate the increment of the for-loop counter
            c[i] += 1
            //Simulate recursive call reaching the base case by bringing
            //the pointer to the base case analog in the array
            i := 0
        else
            //Calling generate(i+1, A) has ended as the for-loop terminated.
            //Reset the state and simulate popping the stack by incrementing the pointer.
            c[i] := 0
            i += 1
        end if
    end while
```

Przykładowy output:

```
0. Generujemy miasta
[-4.528727610341501, -1.334824126966415, 3.1107899015652514,
-0.17418163969465805, 2.6687795703643182, -1.9650500860165203, -2.7935298273414997]
[-3.2153199022963093, -3.0306978855540487, -3.6566170050790983,
-1.0102207134128816, -2.563038429502271, -0.9958324483394025, -3.9849650177869944]


1. Szukamy prostą mutacją
Prosta sciezka: [0, 1, 2, 3, 4, 5, 6]
Dla prostej ścieżki długość ścieżki to: 25.038161940885516
mutuj_A([0,1,2,3,4,5,6], 3) = [4, 1, 2, 0, 3, 5, 6]
mutuj_A([0,1,2,3,4,5,6], 4) = [5, 1, 3, 0, 4, 2, 6]
mutuj_A([0,1,2,3,4,5,6], 5) = [4, 1, 2, 0, 6, 3, 5]
Po mutacjach długość ścieżki [1, 6, 0, 4, 2, 3, 5] to: 20.187372315247863


2. Szukamy przez krzyzowanie


Po krzyżowaniu długość ścieżki [3, 4, 2, 6, 1, 0, 5] to: 20.45659414839204


3. Optymalny wynik to (17.73160413409294, [6, 1, 2, 3, 4, 5, 0])
```

Punktacja

Za poszczególne etapy można uzyskać następującą liczbę punktów:

- Etap 0 - `wygeneruj_prosta_sciezke()` (1p) + `wygeneruj_miasta()` (2p) = 3p
- Etap 1 - `oblicz_dlugosc_sciezki()` (1p) + `mutuj()` (1p) + fragment `main()` (1p) = 3p
- Etap 2 - `krzyzuj()` (1p) + fragment `main()` (1p) = 2p
- Etap 3 - `znajdz_rozwiazanie_optymalne()` = 2p

Uwaga

- Jeśli program się nie kompiluje (interpretuje), ocena jest zmniejszana o połowę.
- Jeśli kod programu jest niskiej jakości (nieestetycznie formatowanie, mylące nazwy zmiennych itp.), ocena jest zmniejszana o 2 p.