

Języki Skryptowe

dokumentacja projektu „Hotele”

Grzegorz Koperwas

30 stycznia 2022

Część I

Opis programu

W Bajtocji jest n miast połączonych zaledwie $n - 1$ drogami. Każda z dróg łączy bezpośrednio dwa miasta. Wszystkie drogi mają taką samą długość i są dwukierunkowe. Wiadomo, że z każdego miasta da się dojechać do każdego innego dokładnie jedną trasą, złożoną z jednej lub większej liczby dróg. Inaczej mówiąc, sieć dróg tworzy drzewo.

Król Bajtocji, Bajtazar, chce wybudować trzy luksusowe hotele, które będą gościć turystów z całego świata.

Król chciałby, aby hotele znajdowały się w różnych miastach i były położone w tych samych odległościach od siebie.

Pomóż królowi i napisz program, który obliczy, na ile sposobów można wybudować takie trzy hotele w Bajtocji.

Instrukcja obsługi

Należy wykonać plik `./run.sh`, wygeneruje on automatycznie zestawy danych testowych, wykona plik `./projekt.py` dla nich, oraz wygeneruje skryptem `./raport.py` plik `./raport.html`.

Skrypt `./backup.sh` tworzy kopię plików wejściowych, wyjściowych oraz raportu i zapisuje je w archiwum `${data}.tar.gz`.

Dodatkowe informacje

Wymagania:

1. Biblioteka Jinja2
2. Python3 (gwarantowane działanie na wersji 3.9.7, wcześniejsze mogą nie działać)
3. System zgodny z POSIX

Część II

Opis działania

Dane jest drzewo n węzłów.

W pierwszej fazie, dla każdego węzła jest tworzony jest słownik zwracający dystans do danego węzła (miasta), dla danego węzła¹.

W drugiej fazie, dla każdego miasta, tworzony jest słownik zwracający listę wszystkich miast, których odległość jest równa, dla danej odległości.

Następnie, dla każdej odległości², jest zliczana liczba miast spełniających warunek zadania³.

Algorytmy

Generacja słownika Miasto \rightarrow odległość

Zrealizowany jest głównie w metodzie `get_connections` w klasie `Miasto`.

Każde „Miasto” jest węzłem w drzewie, zawiera ono domyślnie pusty słownik `city2distace`⁴, flagę trybu szybkiego, puste pole na bramę trybu szybkiego oraz listę połączeń z innymi miastami.

Tryb szybki jest uruchamiany jeżeli napotkamy na sytuację, gdzie występuje *dokładnie jeden* węzeł łączący dane miasto (brama) z resztą drzewa. Wtedy zamiast „chodzić” po drzewie, możemy przepisać słownik z *bramy trybu szybkiego*.

¹W dalszej części pracy będę stosował te wyrażenia wymiennie

²kluczy słownika

³W wyniku optymalizacji powtóżenia są eliminowane

⁴Zwany również słownikiem Miasto \rightarrow odległość

```

funkcja GetConnections(miasto, force_slow):
    if miasto jest w trybie szybkim i nie ma ustawionej flagi force_slow then
        /* W trybie szybkim dopisujemy nowe miasta z „bramy” */
        for miasto w city2distance w bramie do
            if miasto nie jest w city2distance then
                city2distance[ miasto ] = dystans do bramy + dystans z bramy do
                    miasta
            end
        end
    end
    else if dict city2distance jest pusty then
        for połączenie w miasto do
            | city2distance[ miasto z połączenia ] = 1
        end
        /* Sprawdź czy możemy wejść w tryb szybki */
        if miasto ma jednego sąsiada then
            | wejdź w tryb szybki i ustaw bramę na znalezione miasto
        end
    end
    else
        tmp = dict();
        for miasto w city2distance do
            if miasto jest najbardziej oddalonym miastem then
                for dla sąsiadów miasta do
                    if sąsiad nie jest w city2distance then
                        | tmp[ sąsiad ] = dystans do miasta + 1
                    end
                end
            end
        end
        if w tmp jest tylko jedno miasto then
            | wejdź w tryb szybki
        end
        dopisz elementy z tmp do city2distance
    end
    return czy jestem w trybie szybkim

```

Algorithm 1: Metoda pomocnicza do obliczania dystansu

funkcja *main*(*miasta*):

```
    skończoneMiasta = [] while len(skończoneMiasta) != len(miasta) do
        for miasto w miasta do
            if miasto w skończoneMiasta then
                | continue
            end
            else if miasto ma odległość do wszystkich innych miast then
                | dodaj miasto do skończoneMiasta
            end
            else
                GetConnections(miasto) if miasto jest w trybie szybkim then
                    | dodaj miasto do skończoneMiasta
                end
            end
        end
    end
    fin = [] /* Dokończ miasta w trybie szybkim */
    while len(fin) != len(skończoneMiasta) do
        for miasto w skończoneMiasta do
            if miasto w fin then
                | continue
            end
            else if Miasto ma odległość do wszystkich innych miast then
                | dodaj miasto do fin
            end
            else
                | GetConnections(miasto)
            end
        end
        if nie było zmian w fin then
            /* wymuś tryb powolny w miastach */
            for miasto w skończoneMiasta do
                if miasto w fin then
                    | continue
                end
                else
                    | GetConnections(miasto, true)
                end
            end
        end
    end
    end
    return fin
```

Algorithm 2: Obliczanie dystansu dla wszystkich miast

Algorytm wyznaczania ilości hoteli.

Każde miasto posiada unikalne id. Zatem w celu wyeliminowania powtórzeń tych samych konfiguracji hoteli, każde miasto wyznacza możliwe konfiguracje hoteli, gdzie jego „partnerzy” posiadają większe id.

Input: miasto

Output: res

```
distance2city = {} /* odwrócenie słownika city2distace */
for miasto w city2distace do
    if id miasta > id miasta rozważanego then
        | dodaj miasto do distance2city
    end
end
for miasta w distance2city do
    if conajmniej 2 miasta dla danego dystansu then
        for miasta poza pierwszym dla danego dystansu do
            for miasta po rozważanym miastem do
                if Dystans między miastami jest równy then
                    | dodaj hotele do res
                end
            end
        end
    end
end
end
```

Algorithm 3: Wyznaczanie ilości hoteli w jednym mieście

Implementacja

Projekt posiada tradycyjną strukturę w formie jednej klasy na plik, plus plik spinający całość. Projekt nie posiada struktury modułu.

- Plik `projekt.py` zawiera główną część projektu.
- Plik `miasto.py` zawiera definicje klasy `Miasto`.
- Plik `connection.py` zawiera definicje klasy `Connection`.
- Plik `generate.py` zawiera prosty skrypt generujący przypadki

Testy

Zgodność wyników była sprawdzana na manualnie określonych przypadkach. Dodatkowo w celu testów na większych problemach powstał skrypt `generate.py`.

Dodatkowo dla celów testowych była wprowadzona funkcjonalność eksportu zparsowanych drzew do aplikacji `graphviz`, która radziła sobie z eksportem drzew z tysiącami miast do postaci graficznej bez żadnych problemów.

Eksperymenty

W trakcie tworzenia projektu najwięcej czasu spędziłem nad optymalizacją rozwiązania. Próbowane były algorytmy wieloprocessorowe⁵. Jednak ostatecznie rozwiązanie oparte na porównywaniu `id` miast pozwoliło łatwo rozwiązać problem powtarzających się trójek hoteli jak i zmniejszyło znacząco czas wykonywania się programu dla większych drzew, jak i znacząco zmniejszyło ilość pamięci potrzebnej do zapisywania wyników (z około 700mb dla rozwiązania trzymającego wszystkie hotele w liście do pomijalnego dla obiektu `int`).

Dodatkowo problem wyznaczania słownika miasta do dystansu został przyspieszony przez wprowadzenie *trybu szybkiego*, który potrafi odrzucić połowę miast w pierwszej iteracji.

⁵Wielowątkowość w języku `python` nie daje wzrostu wydajności ze względu na *Global Interpreter Lock*, tzw. GIL. Problem w zadaniu nie jest ograniczany przez I/O, zatem wielowątkowość w tradycyjnym tego słowa znaczeniu nie ma sensu

Pełen kod aplikacji

Listing 1: projekt.py

```
1 import sys
2 from datetime import datetime
3 from miasto import Miasto
4 from connection import Connection
5
6 num_of_cities = input()
7 cities = list([Miasto(x) for x in range(int(num_of_cities))])
8 connections = list()
9 for line in sys.stdin:
10     first, second = [int(x) for x in line.split(" ")]
11     connection = Connection(cities[first - 1], cities[second - 1])
12     connections.append(connection)
13
14 begin_time = datetime.now()
15 done_cities = list()
16 iteration = 1
17 while len(done_cities) != len(cities):
18     iteration += 1
19     for city in cities:
20         if city in done_cities:
21             continue
22         elif len(city.city2distance.keys()) == len(cities):
23             done_cities.append(city)
24         else:
25             if city.get_connections():
26                 # city is in quick_mode
27                 done_cities.append(city)
28     print(len(done_cities), iteration, file=sys.stderr, end="\r")
29
30 really_done_cities = list()
31 iteration = 1
32 print(file=sys.stderr)
33 while len(really_done_cities) < len(done_cities):
34     did_modify_something = False
35     for city in done_cities:
36         print(len(really_done_cities), iteration, file=sys.stderr, end="\r")
37         if city in really_done_cities:
38             continue
39         elif len(city.city2distance.keys()) == len(cities):
40             really_done_cities.append(city)
41             did_modify_something = True
42         else:
43             # finish cities in quick_mode
44             city.get_connections()
45     if not did_modify_something:
46         for city in done_cities:
47             if city in really_done_cities:
48                 continue
49             city.get_connections(True)
50
51 print(file=sys.stderr)
```



```

52 hotels = sum([len(city.get_hotels()) for city in cities])
53
54 print(file=sys.stderr)
55 print("Zajętość:", (datetime.now() - begin_time).total_seconds(), "s", file=sys.stderr)
56 print(hotels)
57
58 """
59 with open("graph.dot", "w+") as f:
60     f.write("graph {overlap=false;\n")
61     f.write("\n\t".join([connection.get_dot_string() for connection in connections]))
62     f.write("}")
63 """

```

Listing 2: miasto.py

```

1 import math
2
3 class Miasto:
4     def __init__(self, id) -> None:
5         self.id = id
6         self.connections = []
7         self.city2distance = {}
8         self.__quickmode = None
9         self.locked = False
10
11     def __repr__(self) -> str:
12         return f"<Miasto {self.id + 1}>"
13
14     def get_connections(self, force_slow = False):
15         if self.__quickmode is not None and not force_slow:
16             gateway, offset = self.__quickmode
17             for city, distance in gateway.city2distance.items():
18                 if city not in self.city2distance.keys():
19                     self.city2distance[city] = offset + distance
20         elif len(self.city2distance.keys()) == 0:
21             for connection in self.connections:
22                 self.city2distance[connection.get_other(self)] = 1
23             if len(self.city2distance.keys()) == 1 and not self.locked:
24                 # encountered a node that connects us to everyone else
25                 self.__quickmode = list(self.city2distance.items())[0]
26         else:
27             max_distance = max(self.city2distance.values())
28             tmp = {}
29             for city, distance in self.city2distance.items():
30                 if distance == max_distance:
31                     # only care about furthest cities
32                     for candidate in city.get_surrounding_cities():
33                         if candidate not in self.city2distance.keys():
34                             tmp[candidate] = max_distance + 1
35             if len(tmp.keys()) == 1 and not self.locked:
36                 # encountered a node that connects us to everyone else
37                 self.__quickmode = list(tmp.items())[0]
38             for k, v in tmp.items():
39                 self.city2distance[k] = v
40
41         return self.__quickmode is not None

```

```

42
43
44     def get_surrounding_cities(self) -> list:
45         res = list()
46         for connection in self.connections:
47             res.append(connection.get_other(self))
48         return res
49
50     def get_hotels(self):
51         self.distance2city = {}
52         res = []
53         for city, distance in self.city2distance.items():
54             if city.id > self.id:
55                 array = self.distance2city.get(distance, list())
56                 array.append(city)
57                 self.distance2city[distance] = array
58         for distance, array in self.distance2city.items():
59             if len(array) >= 2:
60                 for i, city in enumerate(array[:-1]):
61                     if city is not self:
62                         for another_city in array[i:]:
63                             if (
64                                 city.city2distance[another_city] == distance and
65                                 another_city is not self and
66                                 city is not another_city):
67                                 hotel = [self.id, city.id, another_city.id]
68                                 hotel.sort()
69                                 hotel_str = ""
70                                 for x in hotel:
71                                     hotel_str += "-" + str(x)
72                                 if not hotel_str in res:
73                                     res.append(hotel_str)
74
75         return res

```

Listing 3: connection.py

```

1  import math
2
3  class Connection:
4      def __init__(self, city1, city2) -> None:
5          self.cities = (city1, city2)
6          for city in self.cities:
7              city.connections.append(self)
8
9      def __repr__(self) -> str:
10         return "{} <-> {}".format(
11             self.cities[0].id + 1,
12             self.cities[1].id + 1)
13
14     def get_other(self, city):
15         if not city in self.cities:
16             raise Exception("City not in connection!")
17         if city is self.cities[0]:
18             return self.cities[1]
19         else:
20             return self.cities[0]

```

```
21 ||
22 | def get_dot_string(self) -> str:
23 |     return f"{self.cities[0].id + 1} -- {self.cities[1].id + 1};"
```