# Języki Skryptowe

dokumentacja projektu „Hotele”

Grzegorz Koperwas

27 grudnia 2021

# 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 twożony jest słownik zwracający dystans do dannego węzła (miasta), dla dannego węzła[1].

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

Następnie, dla każdej odległości[2], jest zliczana liczba miast spełniających warunek zadania[3].

## Algorytmy

### Generacja słownika Miasto → 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 polę 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*.

[1] W dalszej części pracy będe stosował te wyrażenia wymiennie

[2] kluczy słownika

[3] W wynniku optymalizacji powtóżenia są eliminowane

[4] Zwany równierz słownikiem Miasto → 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 city2distace w bramie do
            if miasto nie jest w city2distace then
                city2distace[ miasto ] = dystans do bramy + dystans z bramy do
                 miasta
            end
        end
    end
    else if dict city2distance jest pusty then
        for połączenie w miasto do
            city2distace[ 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 city2distace do
            if miasto jest najbardziej oddalonym miastem then
                for dla sąsiadów miasta do
                    if sąsiad nie jest w city2distace 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
    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** *miasto 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**

**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 bez żadnych problemów.

## Eksperymenty

W trakcie tworzenia projektu najwięcej czasu spędziłem nad optymalizacją rozwiązania. Próbowane były algorytmy wieloprocesorowe[5]. 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.

[5]Wielowątkowość 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

```
import sys
from datetime import datetime
from miasto import Miasto
from connection import Connection

num_of_cities = input()
cities = list([Miasto(x) for x in range(int(num_of_cities))])
connections = list()
for line in sys.stdin:
    first, second = [int(x) for x in line.split(" ")]
    connection = Connection(cities[first - 1], cities[second - 1])
    connections.append(connection)

begin_time = datetime.now()
done_cities = list()
iteration = 1
while len(done_cities) != len(cities):
    iteration += 1
    for city in cities:
        if city in done_cities:
            continue
        elif len(city.city2distance.keys()) == len(cities):
            done_cities.append(city)
        else:
            if city.get_connections():
                # city is in quick_mode
                done_cities.append(city)
    print(len(done_cities), iteration, file=sys.stderr, end="\r")

really_done_cities = list()
iteration = 1
print(file=sys.stderr)
while len(really_done_cities) < len(done_cities):
    did_modify_something = False
    for city in done_cities:
        print(len(really_done_cities), iteration, file=sys.stderr, end="\r")
        if city in really_done_cities:
            continue
        elif len(city.city2distance.keys()) == len(cities):
            really_done_cities.append(city)
            did_modify_something = True
        else:
            # finish cities in quick_mode
            city.get_connections()
    if not did_modify_something:
        for city in done_cities:
            if city in really_done_cities:
                continue
            city.get_connections(True)

print(file=sys.stderr)
```

```
hotels = sum([len(city.get_hotels()) for city in cities])

print(file=sys.stderr)
print("ŁZajeo:", (datetime.now() - begin_time).total_seconds(), "s", file=sys.stderr)
print(hotels)

"""
with open("graph.dot", "w+") as f:
    f.write("graph {overlap=false;\n")
    f.write("\n\t".join([connection.get_dot_string() for connection in connections]))
    f.write("}")
"""
```

Listing 2: miasto.py

```
import math

class Miasto:
    def __init__(self, id) -> None:
        self.id = id
        self.connections = []
        self.city2distance = {}
        self.__quickmode = None
        self.locked = False

    def __repr__(self) -> str:
        return f"<Miasto {self.id + 1}>"

    def get_connections(self, force_slow = False):
        if self.__quickmode is not None and not force_slow:
            gateway, offset = self.__quickmode
            for city, distance in gateway.city2distance.items():
                if city not in self.city2distance.keys():
                    self.city2distance[city] = offset + distance
        elif len(self.city2distance.keys()) == 0:
            for connection in self.connections:
                self.city2distance[connection.get_other(self)] = 1
            if len(self.city2distance.keys()) == 1 and not self.locked:
                # encountered a node that connects us to everyone else
                self.__quickmode = list(self.city2distance.items())[0]
        else:
            max_distance = max(self.city2distance.values())
            tmp = {}
            for city, distance in self.city2distance.items():
                if distance == max_distance:
                    # only care about furthest cities
                    for candidate in city.get_surrounding_cities():
                        if candidate not in self.city2distance.keys():
                            tmp[candidate] = max_distance + 1
            if len(tmp.keys()) == 1 and not self.locked:
                # encountered a node that connects us to everyone else
                self.__quickmode = list(tmp.items())[0]
            for k, v in tmp.items():
                self.city2distance[k] = v

        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          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};"
```