

# **Marcin Żołubowski**

Projekt zaliczeniowy na zajęcia z przedmiotu  
Algorytmy i struktury danych z językiem Python

## **Temat: algorytm Dijkstry**

### 1. Wprowadzenie

Algorytm Dijkstry służy do znajdowania najkrótszych ścieżek z pojedynczego źródła w grafie o nieujemnych wagach krawędzi. Algorytm znajduje w grafie wszystkie najkrótsze ścieżki pomiędzy wybranym wierzchołkiem a wszystkimi pozostałymi, przy okazji wyliczając również koszt przejścia każdej z tych ścieżek wykorzystując kolejkę priorytetową. Priorytet jest określany na podstawie minimalnej odległości od wierzchołka do wierzchołka startowego. Algorytm ten jest przykładem algorytmu zachłannego.

Pseudokod:

Dijkstra( $G, w, s$ ):

**dla każdego** wierzchołka  $v$  w  $V[G]$  **wykonaj**:

$d[v] := \text{nieskończoność}$

$\text{poprzednik}[v] := \text{niezdefiniowane}$

$d[s] := 0$

$Q := V$

**dopóki**  $Q$  niepuste **wykonaj**:

$v := \text{Zdejmij\_Min}(Q)$

**dla każdego**  $u$  - sąsiada wierzchołka  $v$  **wykonaj**:

**jeżeli**  $d[u] > d[v] + w(v, u)$  **to**:

$d[u] := d[v] + w(v, u)$

$\text{poprzednik}[u] := v$

$\text{Dodaj}(Q, u)$

## 2. Opis plików:

Projekt zawiera 8 plików:

- **priorityqueue.py** - implementacja kolejki priorytetowej na bazie modułu queue
- **graph.py** - implementacja grafu wraz z algorytmem
- **main.py** - program główny uruchamiający i wykonujący działania na grafie
- **graphtest.py** - program testujący graf
- **graphSmall.txt** - plik tekstowy z odpowiednio sformatowanymi danymi przygotowanymi na odczytanie za pomocą programu
- **graphSmall.png** - plik graficzny, dodany pomocniczo, aby łatwiej wprowadzać dane oraz aby łatwiej zweryfikować poprawność wyników
- **graphBig.txt** - jak dla graphSmall.txt, jednak tu mamy większe dane
- **graphBig.png** - jak dla graphSmall.png jednak tu mamy więcej danych

Zawartość plików **graphSmall.txt** oraz **graphBig.txt** wygląda w ten sposób, że zapisana jest pewna ilość krawędzi, gdzie każda krawędź znajduje się w osobnej linii, oraz jest zdefiniowana w sposób:

„OdWierzchołka DoWierzchołka Waga”

Przykład kawałka pliku **graphSmall.txt**

„Krakow Katowice 78

Krakow Zakopane 107

Krakow Rzeszow 168

itd...”

Obydwa pliki wyglądają analogicznie, różnią się jedynie ilością krawędzi.

Zawartość plików **graphSmall.png** oraz **graphBig.png** to nic innego jak screenshot mapy polski zrobiony z odpowiedniej odległości aby uchwycić mniej lub więcej miast polski z dodanymi w programie graficznym „krawędziami” z ich wagami pomiędzy odpowiednimi miastami. Najprościej jest ustalić wierzchołek początkowy na podstawie tych załączonych plików graficznych.

Pomaga to również zobrazować czy końcowe dane wyjściowe są zgodne z rzeczywistością. Pliki **graphSmall.png** oraz **graphBig.png** są powiązane z plikami odpowiednio **graphSmall.txt** oraz **graphBig.txt**

Więcej informacji o plikach znajduje się w punkcie 4. **Szczegóły implementacji.**

### 3. Uruchomienie oraz przebieg działania programu.

**Uwaga! Program wymaga kompilacji z pythonem 3!**

Będąc w folderze z projektem z poziomu konsoli, aby **uruchomić testy** należy wpisać: **python3 graphtest.py**. Wówczas otrzymamy wiadomość zwrotną informującą nas, że testy przeszły pomyślnie lub też nie. Program wówczas sam się zakończy.

```
marcin@Marcin-PC ~/Desktop/python/projekt $ python3 graphtest.py
.....
-----
Ran 5 tests in 0.000s

OK
marcin@Marcin-PC ~/Desktop/python/projekt $
```

Będąc w folderze z projektem z poziomu konsoli, aby **uruchomić program główny** należy wpisać: **python3 main.py**. Wówczas otrzymamy wiadomość:

```
marcin@Marcin-PC ~/Desktop/python/projekt $ python3 main.py
Skąd chcesz wczytać dane?
  1) mały graf polski
  2) duży graf polski
  3) własny graf
Wpisz 1, 2 lub 3: 
```

Program wówczas oczekuje na podanie z klawiatury cyfry: 1, 2 lub 3. Podanie innej cyfry skutkuje wypisaniem odpowiedniej informacji oraz ponowne wypisanie jak na obrazku powyżej z oczekiwaniem na wejście z klawiatury. Podanie innego znaku niż cyfra lub np. wciśnięcie enter, skutkuje wypisaniem na ekran odpowiedniej informacji i ponownie wypisuje na ekran jak na obrazku.

Opcja 1 oraz 2 wypisują na ekran to samo. Jedynie, opcja 1 pobiera dane z pliku **graphSmall.txt**, a opcja 2 z pliku **graphBig.txt**.

#### Po wybraniu opcji 1 lub 2:

Program żąda, aby podać wierzchołek startowy. Wierzchołek podajemy wpisując polską nazwę miasta, bez polskich znaków, które to miasto możemy wybrać np. korzystając z dołączonych plików graficznych: **graphSmall.png** dla wyboru **1** lub **graphBig.png** dla wyboru **2**. Podając wierzchołek nieistniejący, otrzymamy

zwrotną informację oraz program ponownie żąda, aby podać wierzchołek startowy. Tak wygląda odpowiedź programu dla grafu pobranego z pliku **graphSmall.txt**, a więc dla wyboru **1** oraz dla wyboru wierzchołka startowego „Krakow”:

```
marcin@Marcin-PC ~/Desktop/python/projekt $ python3 main.py
Skąd chcesz wczytać dane?
  1) mały graf polski
  2) duży graf polski
  3) własny graf
Wpisz 1, 2 lub 3: 1
Podaj wierzchołek startowy: Krakow
Czas Dijkstry:          0.0003714561462402344
Czas Bellmana-Forda:    0.0006728172302246094
Wyniki zostały zapisane w plikach:
resultDijkstra.txt
resultBellmanFord.txt
marcin@Marcin-PC ~/Desktop/python/projekt $
```

Po wpisaniu więc wierzchołka startowego otrzymujemy informację o czasie wykonania algorytmu Dijkstry oraz dla porównania algorytmu Bellmana-Forda, oczywiście dla tych samych danych. Na koniec program informuje nas, że pliki z wynikami działania obydwu algorytmów(pliki powinny być identyczne, jeżeli algorytmy zadziałały poprawnie) zostały zapisane w plikach, odpowiednio **resultDijkstra.txt** oraz **resultBellmanFord.txt**. Po tej informacji program kończy pracę. Możemy przejrzeć wyniki otwierając pliki(opis w jaki sposób należy je czytać znajduje się w punkcie 4. **Szczegóły implementacji**).

#### Po wybraniu opcji 3(dla własnego grafu):

Program żąda, aby podawać krawędzie(skierowane) w postaci:

**„OdWierzchołka DoWierzchołka Waga”**

np.: **„Krakow Katowice 78”**

Podobnie więc jak jest to zdefiniowane w plikach **graphSmall.txt** oraz **graphBig.txt**. Wpisujemy w takiej formie krawędzie, kończymy wpisywanie krawędzi wciskając enter w pustej linii. Jeżeli graf ma być „nieskierowany” krawędzie należy podać dla sytuacji odwrotnej jak dla przypadku powyżej czyli np.: **„Katowice Krakow 78”**. Dalszy przebieg programu jak dla opcji **1** lub **2**, czyli prośba o podanie wierzchołka startowego itd.

#### 4. Szczegóły implementacji

Kody źródłowe znajdują się w plikach:

- **priorityqueue.py**
- **graph.py**
- **graphtest.py**
- **main.py**

**priorityqueue.py** - plik z „moją” implementacją kolejki priorytetowej z metodami **pop** oraz **push**, które wykorzystują metody zdefiniowane z zaimportowanego modułu `queue`. Kolejka zdefiniowana w ten sposób ułatwia używanie metod **put**, **get** oraz ustalanie priorytetu przy każdym wkładaniu elementu do kolejki. Pomysł na takie jej zdefiniowanie znalazłem na stronie **stackoverflow.com**, którą również zamieszczę w wykorzystanej literaturze.

**graph.py** - plik ze zdefiniowanymi klasami **Edge**, **Node** oraz **Graph**. W pliku importuje „moją” klasę `PriorityQ` oraz importuję moduł `time`, do późniejszego przedstawienia czasu działania algorytmów. Klasa **Edge** reprezentuje krawędź, a ponieważ posiada pole reprezentujące jedynie cel, krawędź taką traktuję jako skierowaną. W klasie zdefiniowane jest również pole reprezentujące wagę krawędzi. Klasa ta nie posiada metod. Klasa **Node** reprezentuje wierzchołek grafu. Jego pola reprezentują nazwę wierzchołka (niepowtarzalną), listę krawędzi, pole reprezentujące poprzednika (który prowadzi w stronę wierzchołka startowego) oraz pole reprezentujące odległość od wierzchołka startowego. Klasa ta również nie posiada metod. Klasa **Graph** reprezentuje graf. Jego pola reprezentują listę wierzchołków oraz pomocniczo nazwę wierzchołka startowego. Klasa posiada metody:

- **addNode(nodeName)** - dzięki której możemy dodać wierzchołek o nazwie „nodeName” do grafu
- **getNode(nodeName)** - dzięki której możemy otrzymać do pracy wierzchołek o nazwie „nodeName”
- **addNeighbor(fromNode, toNode, weight)** - dzięki której możemy dodać krawędź do grafu prowadzącą od wierzchołka o nazwie „fromNode”, do wierzchołka o nazwie „toNode” o wadze „weight”
- **getStringNeighbors(node)** - pomocnicza metoda do wyświetlania listy par sąsiadów dla wierzchołka „node”
- **initialize(start)** - metoda inicjalizująca wierzchołki dla algorytmu dijkstry oraz bellmana-forda.

- **dijkstra(start)** – metoda z algorytmem dijkstry, przechożąca graf dla wierzchołka startowego o nazwie „start”
- **bellmanFord(start)** - metoda z algorytmem bellmana-forda, przechożąca graf dla wierzchołka startowego o nazwie „start”
- **getFromFile(filename)** – metoda dla pliku „filename” odczytuje jego zawartość oraz na jej podstawie tworzy graf, dodaje wierzchołki oraz krawędzie
- **writeToFile(filename)** – metoda zapisuje graf do pliku „filename” w odpowiedniej postaci

**graphtest.py** – plik testujący moduł graph, a dokładniej klasę **Graph** oraz jej metody: **addNode**, **getNode**, **addNeighbor**, **initialize** oraz **dijkstra**. Tester testuje graf, który przedstawiłem w samym pliku. Testy powinny wykonać się bezbłędnie a wyniki testów powinny być pozytywne.

**main.py** – plik główny wykorzystujący moduł graph z klasą **Graph**. Program nawiguje użytkownika prosząc, poprzez wyświetlanie odpowiednich informacji, o podanie odpowiednich danych do programu dotyczących wyboru grafu(pliku) do załadowania danych lub ładowania kolejnych krawędzi budujących graf. Następnie program uruchamia algorytmy podając zdefiniowany wcześniej przez użytkownika wierzchołek o pewnej nazwie, oraz przekazuje nazwę plików do zapisu grafu.

### **Pliki wynikowe resultDijkstra.txt oraz resultBellmanFord.txt**

Program główny zapisuje wyniki przejścia algorytmów do plików w ten sposób, że w pierwszej linii zawsze znajduje się napis:

„Od każdego wierzchołka do wierzchołka początkowego:”

co oznacza, że poniżej program wypisał wierzchołki oraz ich drogę poprzez poprzedników aż do wierzchołka początkowego wraz z sumaryczną wagą tego przejścia. Dla wyniku działania na małych danych wygląda to np.:

„Gdynia -> Gdansk -> Bydgoszcz -> Lodz -> Kielce -> Krakow koszt całkowity: 684”

Co oznacza, że żeby przejść z Gdyni do Krakowa należy odwiedzić wierzchołki kolejno: Gdynia, Gdansk, Bydgoszcz, Lodz, Kielce oraz Krakow. Koszt przejścia od wierzchołka Gdynia do Krakow wynosi 684.

Po wypisaniu ścieżek od wszystkich wierzchołków do wierzchołka startowego znajduje się linia:

„Zawartość list sąsiedztwa każdego wierzchołka:”

Co oznacza, że poniżej program wypisał bezpośrednich sąsiadów każdego wierzchołka wraz z kosztem dotarcia do nich. Np. dla wierzchołka „Warszawa”

„Warszawa

- > Olsztyn     koszt: 214
- > Białystok   koszt: 196
- > Lublin       koszt: 171
- > Rzeszów     koszt: 297
- > Kielce       koszt: 179
- > Łódź          koszt: 133
- > Bydgoszcz   koszt: 302”