algorytmy-na-grafach

November 4, 2024

W celu zapoznania się działaniem struktury: - `deque` : deque - `heapq` : heapq

[2]:
```python
from collections import deque
import heapq
```

Przykładowy graf reprezentowany jako słownik list sąsiedztwa

[4]:
```python
graf = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```

0.1 Algorytm BFS

[6]:
```python
from collections import deque

def przeszukiwanie_wszerz(graf, wezel_startowy):
    odwiedzone = set()
    kolejka = deque([wezel_startowy])
    odwiedzone.add(wezel_startowy)

    while kolejka:
        wezel = kolejka.popleft()
        print(wezel, end=" ")

        for sasiad in graf.get(wezel, []):
            if sasiad not in odwiedzone:
                odwiedzone.add(sasiad)
                kolejka.append(sasiad)
```

[7]:
```python
przeszukiwanie_wszerz(graf, 'A')
```

```
A B C D E F
```

0.2 Algorytm DFS

[9]:
```
def przeszukiwanie_w_glab(graf, wezel_startowy, odwiedzone=None):
    if odwiedzone is None:
        odwiedzone = set()

    odwiedzone.add(wezel_startowy)
    print(wezel_startowy, end=" ")

    for sasiad in graf.get(wezel_startowy, []):
        if sasiad not in odwiedzone:
            przeszukiwanie_w_glab(graf, sasiad, odwiedzone)
```

[10]:
```
przeszukiwanie_w_glab(graf, 'A')
```

```
A B D E F C
```

1 Algorytm Dijkstry

[12]:
```
graf_wazony = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
```

[13]:
```
def dijkstra(graf, start):

    odleglosci = {wezel: float('inf') for wezel in graf}
    odleglosci[start] = 0
    kolejka = [(0, start)]

    while kolejka:
        obecna_odleglosc, obecny_wezel = heapq.heappop(kolejka)

        if obecna_odleglosc > odleglosci[obecny_wezel]:
            continue

        for sasiad, waga in graf[obecny_wezel].items():
            odleglosc = obecna_odleglosc + waga

            if odleglosc < odleglosci[sasiad]:
                odleglosci[sasiad] = odleglosc
                heapq.heappush(kolejka, (odleglosc, sasiad))

    return odleglosci
```

[14]:
```
odleglosci = dijkstra(graf_wazony, 'A')
print("Najkrótsze odległości od węzła 'A':", odleglosci)
```

```
Najkrótsze odległości od węzła 'A': {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

2 Biblioteka NetworkX - praca z grafami

Dokumentacja

[16]:
```
import networkx as nx
```

2.1 1. Tworzenie grafu

[18]:
```
Graf = nx.Graph()
```

2.2 2. Wierzchołki

W bibliotece NetworkX, która służy do pracy z grafami w Pythonie, wierzchołkami mogą być prawie dowolne obiekty – mogą to być na przykład napisy (jak nazwy miast), obrazy, elementy XML, inne grafy, a nawet specjalnie stworzone obiekty wierzchołków. Jedyny wymóg to to, żeby dany obiekt był "haszowalny" (czyli mógłby być używany jako klucz w słowniku Pythonowym). Dzięki temu możemy tworzyć grafy z różnymi typami danych, a nie tylko liczbami czy napisami.

[20]:
```
Graf.nodes
```

[20]:
```
NodeView(())
```

2.2.1 Dodawanie wierzchołków:

1. Dodawanie pojedynczych wierzchołków

[23]:
```
Graf.add_node(1)
Graf.nodes
```

[23]:
```
NodeView((1,))
```

2. Dodawanie wierzchołków z kontenera iterowalnego

[25]:
```
Graf.add_nodes_from(range(2, 5))
Graf.add_nodes_from([(5, {"Miasto": "Krawkow"}), (6, {"Miasto": "Katowice"})])
Graf.nodes
```

[25]:
```
NodeView((1, 2, 3, 4, 5, 6))
```

3. Przenoszenie wierzchołków pomiędzy grafami

[27]:
```
Graf1 = nx.Graph()
Graf1.add_nodes_from(range(7, 9))
```

```
Graf1.nodes
```

[27]: NodeView((7, 8))

[28]:
```
Graf.add_nodes_from(Graf1)
Graf.nodes
```

[28]: NodeView((1, 2, 3, 4, 5, 6, 7, 8))

2.3 3. Krawędzie

Kiedy w grafie NetworkX sprawdzamy, które węzły są połączone z danym węzłem (czyli jego sąsiadów, np. `Graph.adj`, `Graph.successors`, `Graph.predecessors`), NetworkX wyświetli te połączenia w takiej kolejności, w jakiej były dodawane do grafu.

Natomiast jeśli sprawdzamy wszystkie krawędzie w grafie za pomocą `Graph.edges`, kolejność będzie zależeć od kolejności węzłów oraz kolejności ich sąsiadów (czyli najpierw posortowane są węzły, a potem ich połączenia).

Przykład:

1. Jeśli dodamy krawędzie `A -> B`, potem `A -> C`, a potem `B -> C`, to przy sprawdzaniu sąsiadów (np. `Graph.adj`) zobaczymy je w takiej właśnie kolejności: `B` i `C` dla `A`.
2. Ale `Graph.edges` może uporządkować to najpierw według węzłów, a potem według ich sąsiadów, co może wyglądać trochę inaczej, jeśli chodzi o całą kolejność krawędzi w grafie.

[31]:
```
Graf.edges
```

[31]: EdgeView([])

1. Dodawanie pojedynczych krawędzi

[33]:
```
Graf.add_edge(1, 2)
Graf.edges
```

[33]: EdgeView([(1, 2)])

2. Dodawanie listy krawędzi

[35]:
```
Graf.add_edges_from([(2, 3), (1, 3)])
Graf.edges
```

[35]: EdgeView([(1, 2), (1, 3), (2, 3)])

3. Importowanie krawędzi z innego grafu

[37]:
```
Graf1.add_edge(7,8)
Graf.add_edges_from(Graf1.edges)
Graf.edges
```

[37]: EdgeView([(1, 2), (1, 3), (2, 3), (7, 8)])

2.4 4. Operacje na grafach

2.4.1 Usuwanie wierzchołków

1. Usuwanie jednego wierzchołka

[41]:
```
Graf.remove_node(2)
```

2. Usuwanie listy wierzchołków

[43]:
```
Graf.remove_nodes_from([7,8])
```

[44]:
```
Graf.edges
```

[44]: EdgeView([(1, 3)])

2.4.2 Usuwanie krawędzi

1. Usuwanie jednej krawędzi

[47]:
```
Graf.remove_edge(1,3)
```

2. Usuwanie listy krawędzi odbywa się analogicznie jak w przypadku wierzchołków: `Graph.remove_edges_from()`

2.5 5. Atrybuty elementów grafu

[50]:
```
Graf = nx.Graph(dzien = "Poniedziałek")
Graf.graph
```

[50]: {'dzien': 'Poniedziałek'}

[51]:
```
Graf.graph["dzien"] = "Wtorek"
Graf.graph
```

[51]: {'dzien': 'Wtorek'}

2.5.1 Atrybuty wierzchołków

Dodanie węzła bezpośrednio do `Graf.nodes` (czyli listy węzłów grafu Graf) nie dodaje go do grafu w NetworkX. Jeśli chcesz dodać nowy węzeł, użyj metody `Graf.add_node()`. Podobnie, aby dodać krawędź między węzłami, musisz skorzystać z metody `Graf.add_edge()`

[53]:
```
Graf.add_node(1, godzina='5')
Graf.add_nodes_from([3], godzina='2')
Graf.nodes[1]
```

[53]: {'godzina': '5'}

[54]:
```
Graf.nodes[1]['pokój'] = 714
Graf.nodes.data()
```

[54]: NodeDataView({1: {'godzina': '5', 'pokój': 714}, 3: {'godzina': '2'}})

2.5.2 Atrybuty krawędzi

Atrybut specjalny `weight` (waga) powinien być liczbą, ponieważ wiele algorytmów w grafach ważonych korzysta z tej wartości do obliczeń. Waga reprezentuje koszt, odległość lub siłę połączenia między węzłami, więc dla poprawnego działania takich algorytmów, jak Dijkstra czy Bellman-Ford, potrzebna jest wartość numeryczna.

[56]:
```
Graf.add_edge(1, 2, weight=4.7 )
Graf.add_edges_from([(3, 4), (4, 5)], kolor='czerwony')
Graf.add_edges_from([(1, 2, {'kolor': 'niebieski'}), (2, 3, {'weight': 8})])
Graf[1][2]['weight'] = 4.7
Graf.edges[3, 4]['weight'] = 4.2
```

2.6 6. Grafy skierowane

[58]:
```
DG = nx.DiGraph()
DG.add_weighted_edges_from([(1, 2, 0.5), (3, 1, 0.75)])
DG.out_degree(1, weight='weight')
```

[58]: 0.5

[59]:
```
DG.degree(1, weight='weight')
```

[59]: 1.25

2.7 7. Multi grafy

[61]:
```
transport_network = nx.MultiGraph()

transport_network.add_node("MiastoA")
transport_network.add_node("MiastoB")

transport_network.add_edge("MiastoA", "MiastoB", transport="pociąg", czas=120)
transport_network.add_edge("MiastoA", "MiastoB", transport="autobus", czas=180)
transport_network.add_edge("MiastoA", "MiastoB", transport="samolot", czas=45)

print("Połączenia między MiastoA i MiastoB w sieci transportowej:")
for edge in transport_network.edges(data=True):
    print(edge)
```

```
Połączenia między MiastoA i MiastoB w sieci transportowej:
('MiastoA', 'MiastoB', {'transport': 'pociąg', 'czas': 120})
('MiastoA', 'MiastoB', {'transport': 'autobus', 'czas': 180})
('MiastoA', 'MiastoB', {'transport': 'samolot', 'czas': 45})
```

2.8 8. Reprezentacja grafów

2.8.1 Lista sąsiedztwa

[64]:
```python
G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3), (2, 4)])

# Wyświetlanie listy sąsiedztwa
print("Lista sąsiedztwa:")
for node, neighbors in G.adjacency():
    print(f"{node}: {list(neighbors)}")
```

```
Lista sąsiedztwa:
1: [2, 3]
2: [1, 4]
3: [1]
4: [2]
```

2.8.2 Macierz sąsiedztwa

[66]:
```python
import numpy as np
adj_matrix = nx.adjacency_matrix(G).todense()
adj_matrix
```

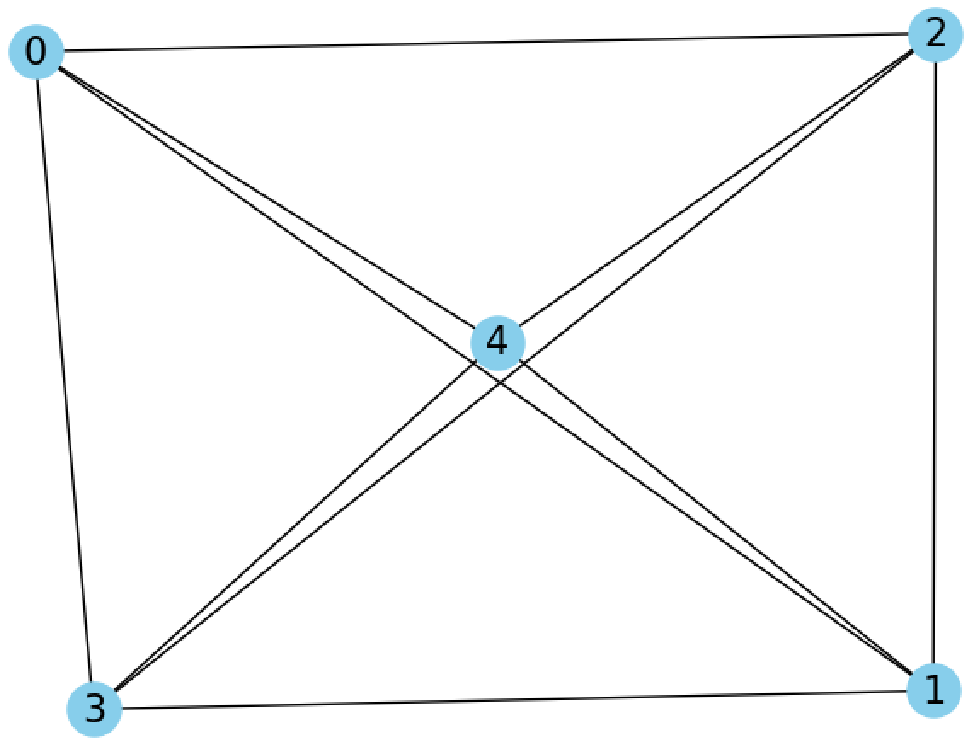
[66]:
```
array([[0, 1, 1, 0],
       [1, 0, 0, 1],
       [1, 0, 0, 0],
       [0, 1, 0, 0]])
```

2.9 9. Generatory grafów

[68]:
```python
import matplotlib.pyplot as plt
```

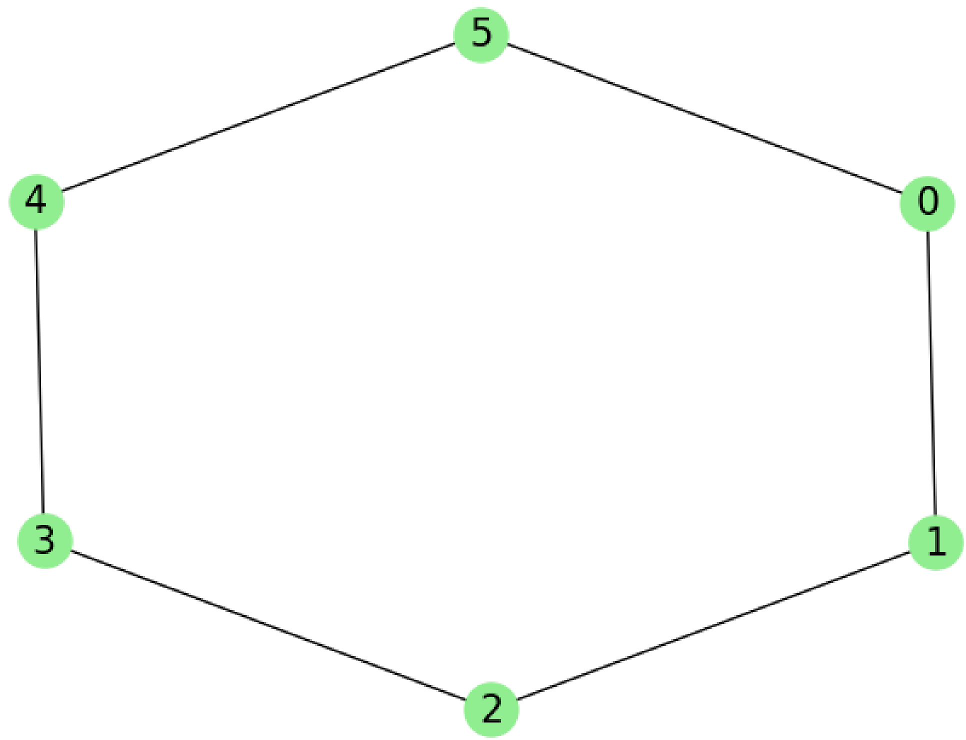
2.9.1 1.Graf pełny

[70]:
```python
G = nx.complete_graph(5)

nx.draw(G, with_labels=True, node_color="skyblue", node_size=500, font_size=16)
plt.show()
```



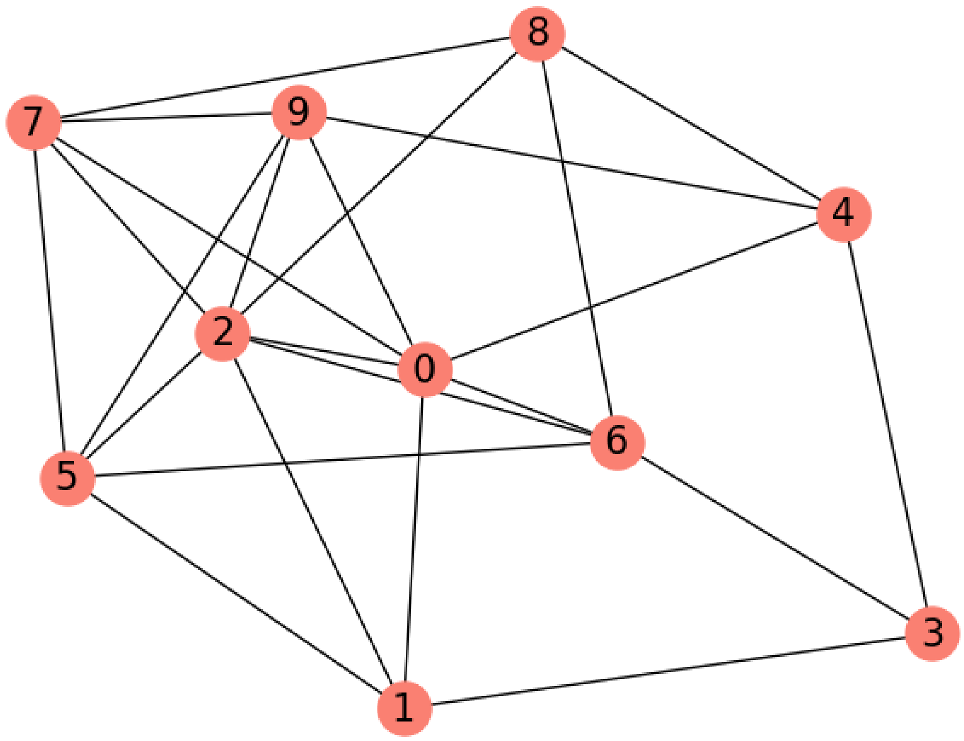
2.9.2 2. Graf cykliczny

[72]:
```
G = nx.cycle_graph(6)

nx.draw(G, with_labels=True, node_color="lightgreen", node_size=500,␣
  ↪font_size=16)
plt.show()
```



2.9.3 3. Graf losowy Erdos-Renyi

[74]:
```
G = nx.erdos_renyi_graph(10, 0.3)

nx.draw(G, with_labels=True, node_color="salmon", node_size=500, font_size=16)
plt.show()
```



3 Przykład

Wyznaczanie najkrótszej trasy pomiędzy uczelniami w Katowicach: (UE -> UŚ -> PŚ)

[76]:
```
import osmnx as ox
import networkx as nx
import matplotlib.pyplot as plt

miasto = "Katowice, Polska"
G = ox.graph_from_place(miasto, network_type='drive')

wspolrzedne_uczelni = {
    "Uniwersytet Ekonomiczny": (50.2599, 19.0242),
    "Uniwersytet Śląski": (50.2612, 19.0247),
    "Politechnika Śląska": (50.2655, 19.0179)
}

wezly_uczelni = {nazwa: ox.distance.nearest_nodes(G, wsp[1], wsp[0]) for nazwa,␣
↪wsp in wspolrzedne_uczelni.items()}
```

```
pary_polaczen = [("Uniwersytet Ekonomiczny", "Uniwersytet Śląski"),
                 ("Uniwersytet Śląski", "Politechnika Śląska"),
                 ("Politechnika Śląska", "Uniwersytet Ekonomiczny")]

trasy = [nx.dijkstra_path(G, wezly_uczelni[start], wezly_uczelni[end],␣
 ↪weight='length') for start, end in pary_polaczen]

fig, ax = ox.plot_graph_routes(G, trasy, route_linewidth=3, node_size=0,␣
 ↪bgcolor='white', route_color='blue')
plt.show()
```

4 Zadania do realizacji

Proszę wykonać minimum 3 zadania korzystając jedynie z dokumentacji, bez korzystania z LLM.

4.1 Zadanie 1

- Stwórz graf z 10 węzłami
- Dodaj losowe połączenia między osobami
- Oblicz stopień każdego węzła (liczbę połączeń każdej osoby) i wypisz osoby z najwyższą i najniższą liczbą znajomych.

4.2 Zadanie 2

- Stwórz graf z 8-10 węzłami, gdzie węzły to miasta, a krawędzie to drogi między nimi.
- Dodaj do każdej krawędzi wagę reprezentującą dystans między miastami.
- Znajdź najkrótszą ścieżkę pod względem odległości między dwoma wybranymi miastami, np. miasto_A i miasto_B.

4.3 Zadanie 3

- Stwórz graf skierowany z losowymi wagami na krawędziach, reprezentującymi przepustowość.
- Zdefiniuj węzeł źródłowy i węzeł ujściowy.
- Użyj funkcji `nx.maximum_flow` do obliczenia maksymalnego przepływu w grafie między źródłem a ujściem.
- Wyświetl krawędzie o największej przepustowości w znalezionym maksymalnym przepływie.

4.4 Zadanie 4

- Stwórz graf z 20-30 węzłami, reprezentującymi komputery lub serwery, z losowymi połączeniami między nimi.
- Usuń kilka losowych krawędzi, aby zasymulować awarie połączeń.
- Sprawdź, czy graf jest nadal spójny po awariach, używając funkcji do znajdowania komponentów spójnych.
- Wyznacz średnią długość najkrótszej ścieżki w grafie przed i po usunięciu krawędzi.