Algorytmy grafowe – algorytmy zachłanne dla zagadnienia komiwojażera

Zadanie 1

```
def GTSP(G, a):
    # utworzenie słownika z dodanymi krawędziami
    # (klucz) wierzchołek początkowy: (wartość) wierzchołek końcowy
    V = dict()

    suma = 0

    # utworzenie kopii macierzy a, aby nie modyfikowaćoryginału
    a_temp = deepcopy(a)
    while len(V.keys()) < len(G.keys()):
        if (a_temp == inf).all() and len(V.keys()) < len(G.keys()):
            raise RuntimeError("Can't find solution")

        # odnalezienie indeksu minimalnej wartość w macierzy a, czyli de facto
        # odczytanie jakie dwa wierzchołki łączy krawędź o minimalnej wartości
        x, y = np.unravel_index(np.argmin(a_temp), a_temp.shape)

        # usunięcie wybranej krawędzi z macierzy krawędzi
        a_temp[x][y] = inf

        # jeżeli kiedyś x było początkiem krawędzi
        # lub y było końcem to przejdź do kolejnej iteracji
        if x in V.keys() or y in V.values():
            continue


        # sprawdzenie czy nie tworzymy podcyklu
        # jeżeli y kiedyś było początkiem, a x końcem
        # i jednocześnie dodanie krwędzi nie kończy działania
        # sprawdź czy istnieje ścieżka, jeśli tak to
        # przejdź do następnej iteracji
        if (y in V.keys() and x in V.values()) and (len(V.keys()) < (len(G.keys()) - 1)):
            if path_exist(V, x, y):
                continue

        # dodanie krawędzi
        V[x] = y

        # dodanie wagi krawędzi do sumy
        suma += a[x][V[x]]

    return V, suma
```

Rys. 1. Implementacja algorytmu G-TSP

Ponieważ funkcja bazuje głównie na macierzy sąsiedztwa, to traktuje ona poniekąd każdy graf jako graf skierowany i zakłada, że każdy wierzchołek w cyklu Hamiltona może tylko raz pełnić każdą z dostępnych ról - początku i końca krawędzi. Nie umniejsza to ogólności takiego rozwiązania, gdyż w przypadku użycia tej funkcji na grafie nieskierowanym dana krawędź zostanie po prostu potraktowana jako dwa łuki o tej samej wadze prowadzące w przeciwne strony.

```
def path_exist(V, x, y):
    current = y
    for i in range(len(V.keys())):
        if current not in V.keys():
            return False
        current = V[current]
```

```
        if current == x:
            return True
```

Rys. 2. Funkcja sprawdzająca możliwość dojścia, jest poprawna ze względu na warunek pełnienia tylko raz danej roli przez wierzchołek

W celu bardziej przejrzystego wypisania wyniku korzystam z następującej funkcji pomocniczej:

```
def print_tsp(V, suma):
    string = f"{list(V.keys())[0]}"
    current = list(V.keys())[0]
    for i in range(len(V.keys())):
        string += f" -> {V[current]}"
        current = V[current]

    print(f"Koszt: {suma} \nCykl:\n{string}")
```

Zadanie 2

Ze względu na bycie algorytmem zachłannym, algorytm GTPS zazwyczaj nie daje rozwiązania optymalnego. Na działanie algorytmów nie ma wpływu skierowanie i nieskierowanie, mogą również istnieć wagi i cykle ujemne. Istotną właściwością jest pełność grafu, ponieważ dla grafów pełnych jest zdefiniowany problem, jednak czasami możliwe jest poprawne wykonanie się algorytmu nawet dla grafu niepełnego. Nie będzie to natomiast możliwe, gdy w grafie występują wierzchołki rozspajające lub graf jest niespójny.

1) Przykład wykonania na grafie pełnym

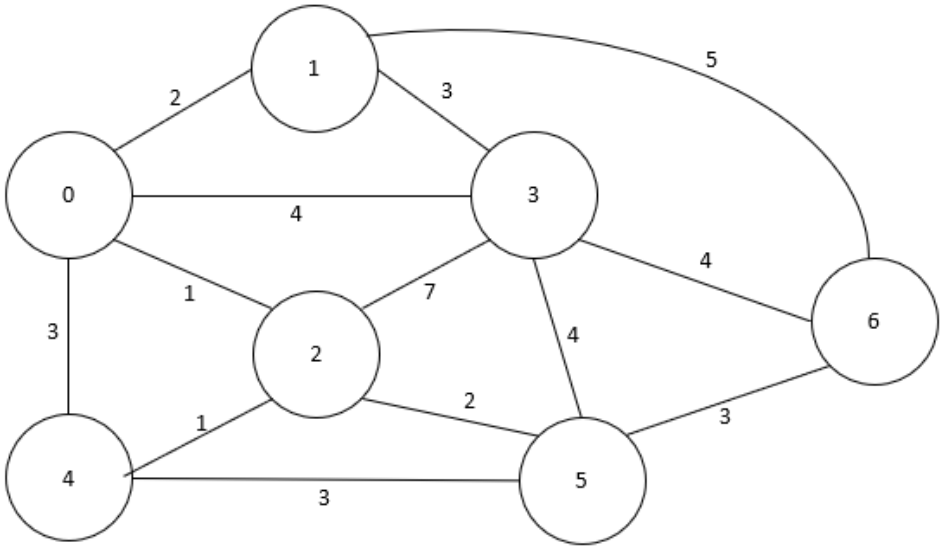
```
a3 = np.array([[inf, 2, 1, 4, 3, 6, 10],
               [2, inf, 2, 3, 3, 3, 5],
               [1, 2, inf, 7, -1, 2, 4],
               [4, 3, 7, inf, 6, 4, 4],
               [3, 3, -1, 6, inf, 3, 8],
               [6, 3, 2, 4, 3, inf, 3],
               [10, 5, 4, 4, 8, 3, inf]])
```

```
Koszt: 15.0
Cykl:
2 -> 4 -> 5 -> 6 -> 3 -> 1 -> 0 -> 2
```

2) Przykład wykonania na grafie nie pełnym

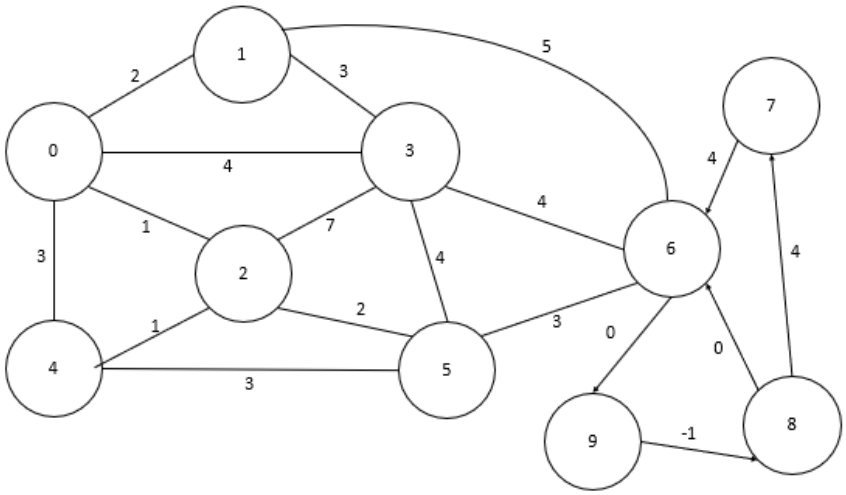
```
graph = {
    0: [1, 2, 3, 4],
    1: [0, 3, 6],
    2: [0, 3, 4, 5],
    3: [0, 1, 2, 5, 6],
    4: [0, 2, 5],
    5: [2, 3, 4, 6],
    6: [1, 3, 5]
}

a = np.array([[inf, 2, 1, 4, 3, inf, inf],
              [2, inf, inf, 3, inf, inf, 5],
              [1, inf, inf, 7, 1, 2, inf],
              [4, 3, 7, inf, inf, 4, 4],
              [3, inf, 1, inf, inf, 3, inf],
              [inf, inf, 2, 4, 3, inf, 3],
              [inf, 5, inf, 4, inf, 3, inf]])
```




```
Koszt: 17.0
Cykl:
0 -> 2 -> 4 -> 5 -> 6 -> 3 -> 1 -> 0
```

3) Przykład wykonania dla grafu z wierzchołkiem rozspajającym:




```
graph = {
    0: [1, 2, 3, 4],
    1: [0, 3, 6],
    2: [0, 3, 4, 5],
    3: [0, 1, 2, 5, 6],
    4: [0, 2, 5],
    5: [2, 3, 4, 6],
    6: [1, 3, 5, 9],
    7: [6],
    8: [6, 7],
    9: [8]
}

a = np.array([
     [inf, 2, 1, 4, 3, inf, inf, inf, inf, inf],
     [2, inf, inf, 3, inf, inf, 5, inf, inf, inf],
     [1, inf, inf, 7, 1, 2, inf, inf, inf, inf],
     [4, 3, 7, inf, inf, 4, 4, inf, inf, inf],
     [3, inf, 1, inf, inf, 3, inf, inf, inf, inf],
     [inf, inf, 2, 4, 3, inf, 3, inf, inf, inf],
     [inf, 5, inf, 4, inf, 3, inf, inf, inf, 0],
     [inf, inf, inf, inf, inf, inf, 4, inf, inf, inf],
     [inf, inf, inf, inf, inf, inf, 0, 4, inf, inf],
     [inf, inf, inf, inf, inf, inf, inf, inf, -1, inf]
])
```

```
Traceback (most recent call last):
  File "D:/studia_zadania/BO/kody/06_TSP.py", line 179, in <module>
    V, suma = GTSP(graph_negative, a_negative)
  File "D:/studia_zadania/BO/kody/06_TSP.py", line 42, in GTSP
    raise RuntimeError("Can't find solution")
RuntimeError: Can't find solution
```

Zadanie 3

Porównanie różnych algorytmów:

a) **Algorytm najbliższego sąsiada** – algorytm zachłanny, średnio dający rozwiązania o 25% gorsze od optymalnych, polegający na odwiedzaniu wierzchołka znajdującego się najbliżej wierzchołka ostatnio odwiedzonego. Dla grafu pełnego złożoność $O(n^2)$

b) **Algorytm zachłanny (G-TSP)** – algorytm polegający na przeglądaniu posortowanej listy krawędzi i stopniowym dodawaniu krawędzi o najmniejszej wadze, jednocześnie sprawdzając czy jej dodanie nie spowoduje powstania cyklu. Ma złożoność na poziomie $O(N^2 \log2(N))$

c) **Algorytm k-optymalny** – algorytm służący do ulepszania już wyznaczonej trasy, polegający na usuwaniu z cyklu k krawędzi i zastępowaniu ich innymi krawędziami tak, aby utworzyć inny prawidłowy cykl. Jeśli osiągnięte w ten sposób rozwiązanie jest lepsze od dotychczas znalezionego, zapamiętujemy je. Za każdym razem modyfikujemy cykl początkowy (nie rozwiązanie z poprzedniego kroku). Złożoność $O(n^k)$, wybranie k=n jest równoważne ze sprawdzeniem wszystkich możliwych rozwiązań.

d) **Algorytm Nearest-Insertion-Heuristik (NEARIN) / Farthest Insertion Heuristik (FARIN)** – algorytm polegający na utworzeniu cyklu składającego się z jednego wierzchołka, a następnie dodawaniu do cyklu wierzchołka o najmniejszej odległości od najbliższego/najdalszego wierzchołka będącego już w cyklu. Obie wersje mają złożoność obliczeniową na poziomie $O(n^2)$

e) **Algorytm Christofidesa** – algorytm bazujący na minimalnym drzewie rozpinającym, następnie wyznaczaniu cyklu Eulera w multigrafie stworzonym z wierzchołków o nieparzystym stopniu oraz krawędzi tworzących minimalne skojarzenie doskonałe na tych wierzchołkach. Cykl Hamiltonowski jest uzyskiwany przez skrócenie odwiedzonych wierzchołków. Jest algorytmem 1,5 optymalnym o złożoności $O(n^3)$

Złożoność obliczeniowa:

Złożoność obliczeniowa implementowanego algorytmu plasuje się na poziomie $O(N^2 \log2(N))$ nie jest to najlepszy, ale również nie najgorszy wynik spośród przedstawionych algorytmów. Największym minusem opisywanej metody w jest trudność w oszacowaniu jak bliskie do optymalnego jest otrzymane rozwiązanie.