Algorytmy grafowe – minimalne drzewo rozpinające grafu

Zadanie 1

Implementacja algorytmu Dijkstry-Prima poszukiwania minimalnego drzewa rozpinającego grafu (MST). W opisach numer w nawiasie oznacza numer linijki z pseudokodu, której instrukcje są realizowane w następnych paru linijkach kodu Pythona.

```
def DPA(G: Dict, a: List[List], s: int):
    # utworzenie sumy i zbiorów A i Q oraz list alfa i beta (01 - 06)
    suma = 0
    A = list()
    alfa = [0 for i in range(len(G))]
    beta = [inf for i in range(len(G))]
    Q = [i for i in range(len(G))]

    # zdjęcie z Q wierzchołka s oraz przypisanie mu kosztu 0 (07 - 08)
    beta[s] = 0
    Q.remove(s)

    # ustawienie s jako ostatnio wybranego wierzchołka (09)
    u_last = s

    # pętla główna (10)
    while Q:
        for el in Q:  # dla każdego u należącego do Q (11)
            for u in G[u_last]:  # i każdego u należącego do N[u*] (11)
                # jeśli a[u,u*]<beta[u] to alfa[u] ←u*; beta[u]←a[u,u*] (12)
                if a[u][u_last] < beta[u]:
                    alfa[u] = u_last
                    beta[u] = a[u][u_last]

        # dla każdego u∈Q u*←arg min(beta[u]) (13 - 14)
        # poszukiwanie minimum może sugerować użycie wbudowanej funkcji min()
        # jednak powoduje to problemy, ponieważ szukamy wtedy minimum z całego
        # beta, więc domyślnie zawsze będziemy znajdować wierzchołek startowy
        # o koszcie dotarcia 0
        mini = inf
        for u in Q:
            if beta[u] < mini:
                mini = beta[u]
                u_last = u

        # usunięcie z Q u*,
        # dodanie do A kolejnej krawędzi,
        # zwiększenie sumy (15 - 17)
        Q.remove(u_last)
        A.append((alfa[u_last], u_last))
        suma += a[alfa[u_last]][u_last]

    # (18)
    return A, suma
```

Przykład działania na podstawie grafu z wykładu:

```
# definicja grafu, macierzy kosztów oraz wywołanie funkcji
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 = [[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, 2, inf]]


print(DPA(graph, a, 0))
```

Wynik działania

```
([(0, 2), (2, 4), (0, 1), (2, 5), (1, 3), (5, 6)], 12)
```

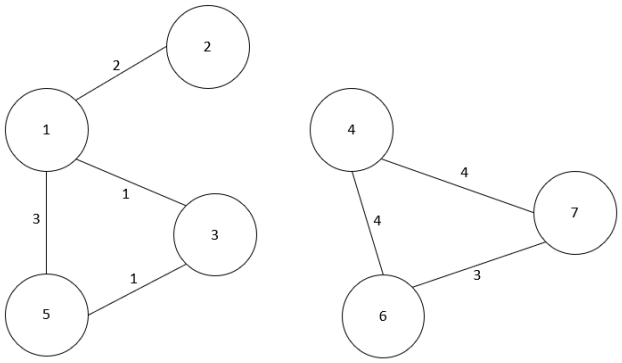
Zadanie 2

Z punktu widzenia działania algorytmu można wyróżnić następujące istotne własności:

a) Spójność
b) Nieskierowanie

W obu przypadkach wynikiem działania jest błąd programu w momencie usuwania z Q wierzchołka u*. Jest to spowodowane tym, że algorytm próbuje się wykonać dopóki istnieje zbiór Q, kiedy nie ma możliwości dotarcia do jakiegoś wierzchołka zbiór Q istnieje zawsze wobec tego w pewnym momencie dochodzi do sytuacji w której powtarza się wartość u*, ponieważ nie przypisano nowej i program próbuje usunąć z listy element, którego nie ma.

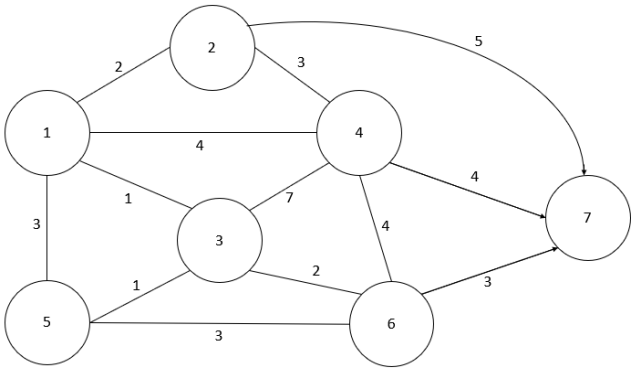
a) Przykład grafu niespójnego




```
graph2 = {
    0: [1, 2, 4],
    1: [0],
    2: [0, 4],
    3: [5, 6],
    4: [0, 5],
    5: [3, 6],
    6: [3, 5]
}

a2 = [[inf, 2, 1, inf, 3, inf, inf],
     [2, inf, inf, inf, inf, inf, inf],
     [1, inf, inf, inf, 1, inf, inf],
     [inf, inf, inf, inf, inf, 4, 4],
     [3, inf, 1, inf, inf, inf, inf],
     [inf, inf, inf, 4, inf, inf, 3],
     [inf, inf, inf, 4, inf, 3, inf]]
```

b) Przykład grafu skierowanego




```
graph3 = {
    0: [1, 2, 3, 4],
    1: [0, 3],
    2: [0, 3, 4, 5],
    3: [0, 1, 2, 5],
    4: [0, 2, 5],
    5: [2, 3, 4],
    6: [1, 3, 5]
}

a3 = [[inf, 2, 1, 4, 3, inf, inf],
     [2, inf, inf, 3, inf, inf, inf],
     [1, inf, inf, 7, 1, 2, inf],
     [4, 3, 7, inf, inf, 4, inf],
     [3, inf, 1, inf, inf, 3, inf],
     [inf, inf, 2, 4, 3, inf, inf],
     [inf, 5, inf, 4, inf, 3, inf]]
```

Zadanie 3

- Algorytm Kruskala rozpoczyna od utworzenia lasu z wierzchołków oryginalnego grafu, następnie działając na posortowanej liście krawędzi wybiera krawędź o najmniejszej wadze łączącą dwa drzewa (jeśli krawędź nie łączyłaby dwóch drzew to oznacza, że jej dodanie stworzy cykl). Kiedy L przestaje być lasem algorytm kończy działalność, a wybrane krawędzie tworzą drzewo rozpinające.

- Najbardziej kosztowną operacją algorytmu jest sortowanie zbioru krawędzi. Ma ona złożoność na poziomie O(E log(V)) i ze względu na bycie najbardziej kosztowną operacja definiuje złożoność działania całego algorytmu.

Zadanie 4

Praktyczne interpretacje problemu MST:

Problem MST odzwierciedla te praktyczne zagadnienia w których chcielibyśmy utworzyć sieć łączącą wszystkie punkty jak najmniejszym kosztem, będą to zatem zagadnienia takie jak: utworzenie sieci wodociągowej, ciepłowniczej, gazowej itp.

- Wagi mogą reprezentować szeroko pojęty koszt dotarcia do danego wierzchołka, nie koniecznie musi być to odległość. Może to np. być ilość surowca potrzebna do doprowadzenia rur do dane lokacji, trudność przekopu itp.
- W celu lepszego odzwierciedlenia problemu rzeczywistego zamiast wag w formie liczby całkowitej można użyć wektorów uwzględniających różne parametry tak jak wcześniej wymieniona trudność przekopu, łatwość późniejszej konserwacji i inne istotne czynniki wybrane podczas analizy zagadnienia.
- W przypadku wyboru wektorów do algorytmu należałoby dodać funkcję kosztów, która przeliczałaby parametry z wektorów na jedną wartość. W takim podejściu należy pamiętać o normalizacji danych w wektorach tak, aby np. ocena trudności przekopu w skali 0 – 10 nie została zmarginalizowana przez np. odległość mierzoną. w tysiącach metrów.