Algorytmy grafowe –najkrótsza ścieżka w grafie

Zadanie 1

W celu implementacji algorytmu Johnsona przystępuję najpierw do napisanie wykorzystywanych w nim algorytmów Dijkstry oraz Bellmana-Forda.

```
def dijkstra(G, a, s):
    d = dict()
    p = dict()
    Q = dict()
    for u in G.keys():
        d[u] = inf
        p[u] = None
        Q[u] = u
    Q.pop(s)
    d[s] = 0
    u_last = s
    while Q:
        for el in Q.values():
            for u in G[u_last]:
                if (d[u_last]+a[u_last][u]) < d[u]:
                    d[u] = d[u_last]+a[u_last][u]
                    p[u] = u_last
        mini = inf
        for u in Q.values():
            if d[u] < mini:
                u_last = u
                mini = d[u]
        Q.pop(u_last)
    return d, p
```

Rys. 1. Implementacja algorytmu Djikstry

```
def bellman_ford(G, a, s):
    cost_of_reaching = {i: inf for i in G.keys()}
    previous = {i: -1 for i in G.keys()}
    cost_of_reaching[s] = 0

    for i in range(len(G) - 1):
        for u in G.keys():
            for v in G[u]:
                if cost_of_reaching[v] > cost_of_reaching[u] + a[u][v]:
                    cost_of_reaching[v] = cost_of_reaching[u] + a[u][v]
                    previous[v] = u

    for u in G.keys():
        for v in G[u]:
            if cost_of_reaching[v] > cost_of_reaching[u] + a[u][v]:
                print("Negative cycle exists")
                return None, None

    return cost_of_reaching, previous
```

Rys. 2. Implementacja algorytmu Bellamana-Forda

```
def johnson(G, a):

    # dodanie dodatkowego wierzchołka
    keys = list(G.keys())
    additional_node = keys[-1] + 1 # o jeden więcej niż ostatni wierzchołek
    G[additional_node] = G.keys()

    # zabronienie połączeń z wszytkich wierzchołków do nowego wierzołka
    for el in a:
        el.append(inf)

    # dodanie połączeń z nowego wierzołka do wszytkich wierzchołków
    a.append([0 for i in G.keys()])
    # usunięcie połączenie wierzchołka dodanego z samym sobą ()
    a[-1][-1] = inf

    # wywołanie algorytmu Bellmana-Forda oraz sprawdzenie istninia ew.
cykli ujemnych
    cost_of_reaching, previous = bellman_ford(G, a, additional_node)
    if cost_of_reaching is None and previous is None:
        print("Negative cycle exists")
        return None

    # modyfikacja wag krawędzi
    for i in range(len(a[0])):
        for j in range(len(a)):
            if a[i][j] != inf:
                a[i][j] = a[i][j] + cost_of_reaching[i] -
cost_of_reaching[j]

    # usunięcie dodatkowego wierzchołka
    G.pop(additional_node)
    # usunięcie połączeń z nowego wierzołka do wszytkich wierzchołków
    a.pop(-1)
    # usunięcie połączeń z wszytkich wierzchołków do nowego wierzołka
    for el in a:
        el.pop(-1)

    D = dict()
    for node in G.keys():
        D[node] = dijkstra(G, a, node)
        for destination in D[node][0].keys():
            # wyznaczenie długości dla grafu G
            D[node][0][destination] += - cost_of_reaching[node] +
cost_of_reaching[destination]

    return D
```

Rys. 3. Implementacja algorytmu Johnsona

Algorytm zwraca słownik D, który dla każdego wierzchołka *u* przechowuje krotkę składającą się z dwóch elementów:

1) Słownik zawierający jako klucze wierzchołki, a jako wartości koszty dotarcia do nich startując z *u*
2) Drugi słownik przechowujący znowu jako klucze wierzchołki, a jako wartości numer poprzednika w najkrótszej ścieżce do danego wierzchołka startując od *u.*

W celu otrzymania ścieżki z danego wierzchołka do innego wierzchołka końcowego można posłużyć się następującą funkcją

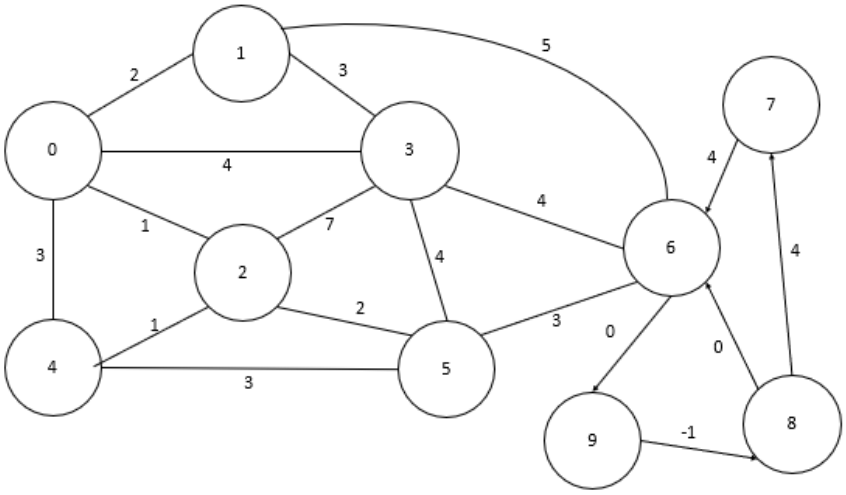
```
def path(D: dict(), start, end):
    D = D[start][1]
    path = [end]
    current = end
    while current != start:
        current = D[current]
        path.append(current)
    return path[::-1]
```

Zadanie 2

Z punktu działania rozważanego algorytmu poza oczywistą własnością jak spójność grafu istotne znaczenie mają także:

- **Brak cykli ujemnych** – przedstawiony algorytm może operować na grafie z wagami ujemnymi jednak otrzymamy wynik niewłaściwy, gdy w grafie znajdzie się cykl ujemny. Z tego powodu w algorytmie zaimplementowałem kod wykrywający istnienie takich cykli i w przypadku ich wystąpienia przerywający działanie programu.

Przykładowy graf z ujemnym cyklem:



Jego zapis w Pythonie:

```
graph_negative = {
    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_negative = [
     [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]
]
```

Wynik programu:

```
D:\Programy\Python\python.exe D:/studia_zadania/BO/kody/04_najkrotsza_sciezka.py
Negative cycle exists
Negative cycle exists

Process finished with exit code 0
```

Na ekranie można zauważyć podwójny komentarz o wystąpieniu ujemnego cyklu jest to spowodowane istnieniem mechanizmu wykrywania ujemnych cykli również w funkcji bellman_ford. Być może lepszym rozwiązaniem zamiast wypisywania byłoby użycie wyjątków z języka Python, jednak to w pewien sposób bardziej komplikowałoby kod czego chciałem uniknąć.

Przykładowa ścieżka jeśli pozbędziemy się ujemnego cyklu np. ustawiając wagę 3 dla krawędzi (8,6)

```
D:\Programy\Python\python.exe D:/studia_zadania/BO/kody/04_najkrotsza_sciezka.py
[0, 2, 5, 6, 9, 8]

Process finished with exit code 0
```

- **Rzadkość grafu –** rzadkość grafu jest własnością istotną ze względu na zasadność użycia alg. Johnsona, który będąc konkurencyjnym algorytmem do algorytmu Floyda-Warshalla osiąga mniejsze złożoności obliczeniowe właśnie dla grafów rzadkich. W przypadku grafu pełnego złożoność metody Johnsona wyrównuje się ze złożonością algorytmu Floyda -Warshalla

Zadanie 3

W przypadku algorytmu Johnsona złożoność obliczeniowa jest jednym z głównych powodów jego użycia, ponieważ tak jak zostało wspomniane w poprzednim punkcie o ile nie mamy do czynienia z przypadkiem pesymistycznym (grafu pełnego), w którym złożoność wynosi $O(V^3)$ to oczekiwana złożoność wynosi: $\mathrm{O}(\mathrm{V}^2\log(\mathrm{V}) + \mathrm{VE})$. Wynika to z tego, że:

- Dodanie nowego wierzchołka wymaga O(V) operacji
- Algorytm Bellmana-Forda działa ze złożonością O(VE)
- Zoptymalizowany algorytm Dijkstry działa ze złożonością O(Vlog(V) + E), więc wykonanie go na wszystkich wierzchołkach będzie wymagać $O(V^2 \log(V) + VE)$ operacji

Zatem najbardziej kosztowną czynnością algorytmu Johnsona jest wykonanie algorytmu Dijkstry na wszystkich wierzchołkach.