Algorytmy grafowe – reprezentacja, algorytmy przeszukiwania

Zadanie 1:

Macierz sąsiedztwa:

Zalety:

- Sprawdzenie czy istnieje krawędź oraz sprawdzenie kosztu danej krawędzi ma złożoność czasową O(1)
- Złożoność usunięcia krawędzi O(1)
- Szybki sposób sprawdzenia czy dany graf jest skierowany
- Jeżeli graf jest symetryczny można używać macierzy trójkątnej górnej (nie ma potrzeby zapisu dwukrotnie tej samej krawędzi)

Wady:

- Złożoność pamięciowa $O(|V|^2)$ gdzie $|V|$ to moc zbioru wierzchołków

Lista sąsiedztwa:

Zalety:

- Złożoność czasowa dodania krawędzi O(1)
- Znacznie mniejsza złożoność pamięciowa $O(|V|+|E|)$ gdzie $|V|$ to moc zbioru wierzchołków a $|E|$ to moc zbioru krawędzi

Wady:

- Większa złożoność sprawdzenia czy istnieje lub usunięcie krawędzi $O(|E|)$ gdzie $|E|$ to moc zbioru krawędzi
- Przy grafach nieskierowanych potrzeba zapisu krawędzi w dwóch miejscach.

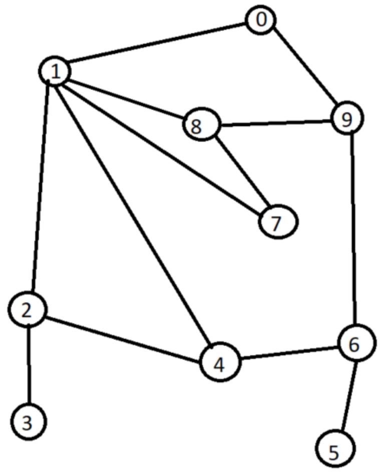
Podsumowując zapis macierzowy grafu jest opłacalny w przypadku bardzo gęstych grafów. Jeżeli chodzi o grafy rzadkie o dużo lepszym wyborem zapisu byłaby macierz sąsiedztwa. Z uwagi na pewne ograniczenia sprzętowe pod względem pamięciowym do zapisu grafów można posłużyć się zapisem przewidzianych dla macierzy rzadkich takich jak: Lista list (LIL), Współrzędne i wartość (COO), Format Yale.

Zadanie 2

Zdefiniowany graf:

```python
# przykładowy graf:
graf_list = {   0:[1,9] ,
                1:[0,1,2,4,7,8] ,
                2:[1,3,4],
                3:[2],
                4:[1,2,4,6],
                5:[6],
                6:[4,5,9],
                7:[1,8],
                8:[1,7,9],
                9:[0,6,8]}
```

Rys. 1. Reprezentacja grafu za pomocą list sąsiedztwa



Rys. 2. Rysunek przykładowego grafu

Zadanie 3

Algorytm przeszukiwania wszerz:

Warunek na spójność grafu:

- Za pomocą BFS zostały odwiedzone wszystkie wierzchołki.

Warunek na acykliczność grafu:

Jeżeli graf jest spójny:

- Należy badać czy w trakcie fazy 4c (Dodanie nieponumerowanych sąsiadów v do FIFO) algorytmu BFS natknięto się na ponumerowany wierzchołek. Jeżeli tak to czy jest on poprzednikiem przerabianego aktualnie wierzchołka jeżeli tak to należy kontynuować sprawdzanie acykliczności. Jeżeli nie to w danym grafie istnieje cykl

Jeżeli graf jest niespójny i nie wykryto cyklu w pierwszym kawałku grafu(jeżeli znaleziono to automatycznie dany graf nie jest acykliczny):

- Należy wykonać BFS z odpowiednią flagą acykl_check = True dla wszystkich nieodwiedzonych wierzchołków i sprawdzić czy istnieją jakieś cykle jak dla grafów spójnych. Dodatkowo można bardziej zoptymalizować dany kod pod względem współdzielenia pewnej listy z rejestrem odwiedzonych wierzchołków dla różnych instancji wywoływanego algorytmu BFS z flagą acykl_check = True.

Kod źródłowy:

```
import numpy as np

def BFS_lista(G : dict,s , acykl_check = False):
    #inicjalizacja zmiennych:
    FIFO = []
    #lista kolejnych odwiedzonych wierzchołków
    no_visited = []
    #nadanie wieszchołkowi v=s numeru No = s (numeracja) oraz dodanie poprzednika (potrzebne przy wyzn
    v = [s,None]
    acykl = True
    #koniec inicjalizacji zmiennych
    no_visited.append(v[0])
    #Umieszczenie w FIFO sąsiadów v oraz ich poprzedników
    for i in G[v[0]]:
        if not i in no_visited:
            FIFO.append([i,v[0]])
    #dopóki FIFO nie jest pusta:
    while FIFO != []:
        #pobranie z FIFO wierzchołka v z informacją o jego poprzedniku (z usunięciem)
        v = FIFO[0]
        FIFO.pop(0)
        # pominięcie wierzchołka jeżeli był już odwiedzony (a jest w stacku)
        if v[0] in no_visited:
            continue
        no_visited.append(v[0])
        #Dodanie nieponumerowanych sąsiadów v do FIFO
        for i in G[v[0]]:
            if not (i in no_visited):
                FIFO.append([i,v[0]])
            else:
                #Jeżeli trafiono na odwiedzony wierzchołek i nie jest to wierzchołek poprzedni to
                if i != v[1]:
                    acykl = False
    # Analiza:
    #Czy wszystkie wierzchołki odwiedzone:
```

Rys. 3. Pierwsza część programu[1]

[1] Ucięty komentarz „nadanie wierzchołkowi v=s numeru No = s (numeracja) oraz dodanie poprzednika (potrzebne przy wyznaczaniu cykli)”

```python
    # Analiza:
    #Czy wszystkie wierzchołki odwiedzone:

    if all([ i in no_visited for i in G.keys()]):
        spojny = "spojny"
    else:
        spojny = "niespojny"
        # sprawdzenie cykli ze wszystkich nieodwiedzonych wierzchołków
        if acykl and not acykl_check:
            for i in G.keys():
                if not (i in no_visited):
                    _ , acy , _ = BFS_lista(G, i, True)
                    acykl = acykl and (False if acy == "zawiera cykle" else True)

    # czy acykliczny
    if acykl:
        acykl = "acykliczny"
    else:
        acykl = "zawiera cykle"
    return no_visited, acykl, spojny
```

Rys. 4. Druga część programu

Wynik dla przykładowego grafu (**Rys. 2.**). kolejne wartości z listy określają kolejne odwiedzone wierzchołki:

```
([0, 1, 9, 2, 4, 7, 8, 6, 3, 5], 'zawiera cykle', 'spojny')
```

Rys. 5. Wynik operacji BFS dla przykładowego grafu

Zadanie 4

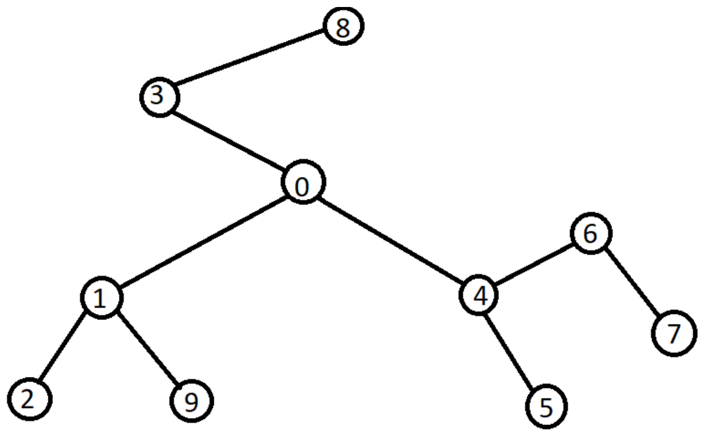
```python
# lista sąsiedztwa
# acykliczny spojny
G = {0:[1,3,4], 1:[2,9], 2:[1], 3:[0,8], 4:[0,5,6], 5:[4], 6:[4,7], 7:[6], 8:[3], 9:[1]}
print(BFS_lista(G,0))

#  spojny z cyklami
G = {0:[1,3,4], 1:[2,9], 2:[1], 3:[0,8], 4:[0,5,6], 5:[4,9], 6:[4,7], 7:[6], 8:[3], 9:[1]}
print(BFS_lista(G,0))

# niespojny zawiera cykle
G = {1:[2,9], 2:[1], 3:[8], 4:[5,6], 5:[4,7], 6:[4,7], 7:[5,6], 8:[3,10], 9:[1], 10 : [8]}
print(BFS_lista(G,1))
```

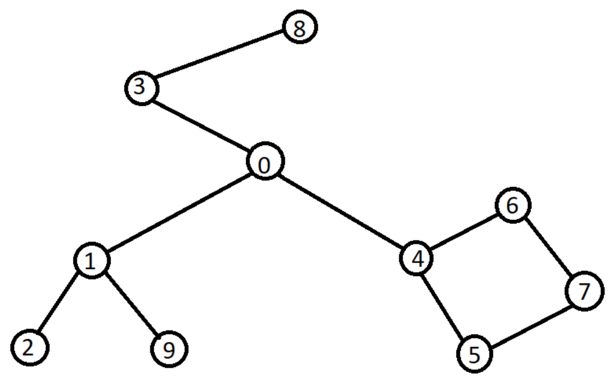
Rys. 6. Reprezentacja grafów dla różnych przypadków

Graf acykliczny spójny:



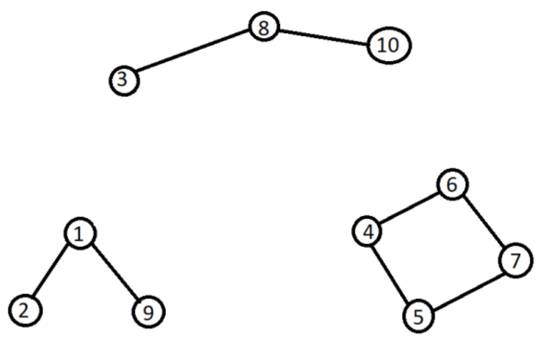
Rys. 7. Graf acykliczny spójny

Graf spójny zawierający cykle:



Rys. 7. Graf spójny zawierający cykle

Graf nie spójny zawierający cykle:



Rys. 8. Graf nie spójny zawierający cykle

Kolejno wyniki dla danych grafów(**Rys. 6.** ,**Rys. 7.** ,**Rys. 8.**):

```
([0, 1, 3, 4, 2, 9, 8, 5, 6, 7], 'acykliczny', 'spojny')
([0, 1, 3, 4, 2, 9, 8, 5, 6, 7], 'zawiera cykle', 'spojny')
([1, 2, 9], 'zawiera cykle', 'niespojny')
```

Rys. 9. Wyniki algorytmu dla poszczególnych grafów.

Zadanie 5

- Wierzchołki rozpajające grafu możemy znaleźć w następujący sposób:
  dla każdego wierzchołka v z V -zbiór wierzchołków grafu G wykonujemy bfs lub dfs lecz pomijamy istnienie wierzchołka v. (tzn. pomijamy każdą krawędź incydentną z wierzchołkiem v) Następnie sprawdzamy czy wszystkie wierzchołki (bez v) zostały odwiedzone. Jeżeli tak się nie stało to wierzchołek v jest wierzchołkiem rozpajającym.

- Centrum grafu możemy znaleźć w następujący sposób:
  dla każdego wierzchołka v z V -zbiór wierzchołków grafu G wykonujemy bfs przy czym każda wartość w kolejce będzie miała dodatkową wartość która jest określana odległość od grafu początkowego. Gdy w bfs będzie rozpatrywany nowy wierzchołek z kolejki to wszystkie nowo dodane wierzchołki będą posiadały odległość (w kolejce) o 1 większą od rozpatrywanego wierzchołka. Przy układaniu listy odwiedzonych wierzchołków przypisujemy odległość danego wierzchołka do maksymalnego kosztu jeżeli ten jest większy od poprzedniego maksymalnego kosztu. Porównujemy wszystkie maksymalne koszty dla wierzchołków i wybieramy wierzchołek dla którego koszt wynosi minimum. Ten wierzchołek jest centrum grafu
- Za pomocą dfs można znajdować drogę pomiędzy dwoma wierzchołkami odpowiednio wprowadzając zapamiętywanie drogi
- Za pomocą dfs i bfs można sprawdzić czy graf zawiera cykle lub czy jest spójny.(zadanie 3)
- Za pomocą dfs (dla grafu nieważonego) można stworzyć minimalne drzewo rozpinające odpowiednio dodając odwiedzone krawędzie pomiędzy poprzednikiem a rozpatrywanym wierzchołkiem do pewnego zbioru.