Algorytmy grafowe – reprezentacja, algorytmy przeszukiwania

1. Wady i zalety macierzy oraz listy sąsiedztwa

Macierz sąsiedztwa:

a. Zalety:
   - Stały czas dodania krawędzi
   - Stały czas sprawdzenia istnienia krawędzi i jej usunięcia
   - Na podstawie własności macierzy można wnioskować o cechach grafu, np. macierz symetryczna oznacza, że graf jest nieskierowany

b. Wady:
   - Duże wymagania pamięciowe $O(V^2)$ – V to liczba wierzchołków

Lista sąsiedztwa

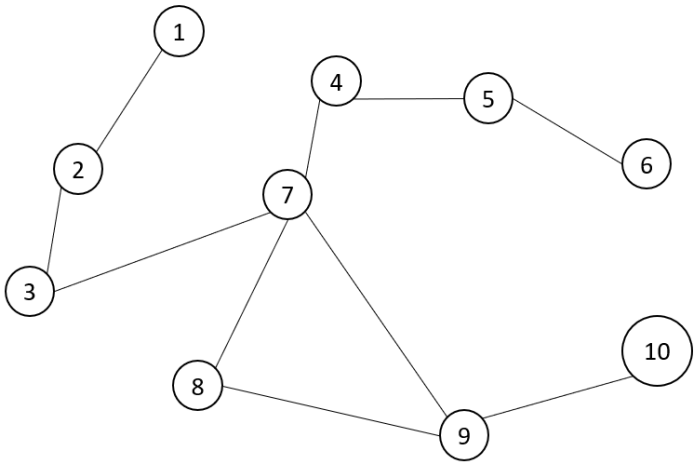
a. Zalety:
   - Mniejsze niż macierz wymagania pamięciowe O(V + E) - V to liczba wierzchołków, E to liczba krawędzi
   - Stały czas dodania krawędzi
   - Przejrzenie następników/poprzedników danego wierzchołka: maksymalnie O(V) (bo tyle sąsiadów może mieć wierzchołek), ale średnio O(E/V), bo tylu średnio sąsiadów będzie miał wierzchołek

b. Wady:
   - Złożoność O(E) dla: sprawdzenia istnienia krawędzi, jej usunięcia

Podsumowując, dla grafów rzadkich zdecydowanie lepiej sprawdza się lista sąsiedztwa, która znacząco ogranicza zużycie pamięci (grafy rzadkie mają mało krawędzi), w przypadku grafów o niskim rzędzie lub grafów gęstych bardziej przydatna okazać się może macierz sąsiedztwa głównie ze względu na prostotę implementacji.

2. Zdefiniowanie struktury danych dla grafu nieskierowanego rzędu 10

Przykładowy graf nieskierowany o 10 wierzchołkach.



Rys. 1. Graficzna reprezentacja przykładowego grafu nieskierowanego 10 rzędu.

Zadany graf prezentuję w postaci listy sąsiedztwa. W języku Python wykorzystuję do tego słownik posiadający jako klucze etykiety kolejnych wierzchołków, a jako wartości listy z etykietami wierzchołków sąsiednich

```
graph = {
    1: [2],
    2: [1, 3],
    3: [2, 7],
    4: [5, 7],
    5: [4, 6],
    6: [5],
    7: [3, 4, 8, 9],
    8: [7, 9],
    9: [7, 8, 10],
    10: [9]
}
```

Rys. 2. Reprezentacja grafu z rys. 1 w postaci listy sąsiedztwa z wykorzystaniem struktury słownika w języku Python.

3. Implementacja algorytmu przeszukiwania w głąb

Do implementacji wybieram algorytm DFS – przeszukiwania w głąb, jako nową numerację przyjmuję kolejność dodania wierzchołka do listy No (w celu otrzymania na ekranie numerów wystarczy dopisać parę linijek z właściwym wypisywaniem z tablicy, nie dodaję ich do funkcji, aby nie zaśmiecać jej logiki). Aby wykryć cykliczność oraz spójność przyjmuję następujące warunki:

- Dla cykliczności – jeżeli przy odkładaniu sąsiadów danego wierzchołka natrafimy na wierzchołek już odwiedzony więcej niż jeden raz to znaczy, że w grafie występuje cykl. Jednokrotne natrafienie na wierzchołek odwiedzony jest dopuszczalne ponieważ rozważamy graf nieskierowany i oznacza wybranie wierzchołka z którego przyszliśmy.
- Dla spójności – jeżeli na koniec działania funkcji liczba wierzchołków dodanych do tablicy No jest mniejsza niż liczba wierzchołków w grafie (czyli w przypadku wybranej struktury danych mniejsza niż liczba kluczy w słowniku) to znaczy, że graf jest nie spójny.

```
def dfs(G: Dict[int, List[int]], s: int):
    # No - lista kolejno odwiedzonych, czyli ponumerowanych wierzchołków
    # wierzchołek na początku listy ma numer 1, kolejny 2 itd.
    No = list()

    # zmienna opisująca istnienie cykli - domyślnie zakładamy graf acykliczny
    cycle_exists = False

    # zmienna opisująca spójność - domyślnie zakładamy graf spójny
    is_consistent = True

    stos = list()    # utworzenie zmiennej stos - bufor LIFO
    stos.append(s)   # odłożenie na stos wierzchołka startowego
    while stos:      # dopóki są elementy w stosie wykonuj:
        v = stos.pop()  # zdejmij ostatni element ze stosu
        if v not in No:    # jeżeli wierzchołek nie został ponumerowany wykonaj:
            No.append(v)    # dodaj wierzchołek do listy - nadaj mu numer
            no_of_neighbours_visited = 0 # liczba sąsiadów "v", którzy już są ponumerowani
            for u in G[v][::-1]:  # dla każdego "u" w liście sąsiedztwa "v"
                stos.append(u)    # odłóż "u" na stos
                if u in No:       # jeżeli "u" otrzymało numer
                    no_of_neighbours_visited += 1 # zwiększ o 1 liczbę odwiedzonych sąsiadów
                    if no_of_neighbours_visited > # jeżeli liczba już odwiedzonych sąsiadów
                                                  przekracza 1
                        cycle_exists = True     # zaznacz, że istnieje cykl
    # sprawdzenie spójności
    if len(No) < len(G.keys()):     # jeżeli liczba wierzchołków ponumerowanych
                                    # jest mniejsza niż liczba kluczy w słowniku
        is_consistent = False       # zaznacz, że graf nie jest spójny
    # zwróć krotkę zawierającą:
    # 1) listę odwiedzonych wierzchołków - nowy numer odpowiada pozycji na liście
    # 2) zmienną typu bool stwierdzającą czy istnieją cykle
    # 3) zmienną typu bool stwierdzającą czy graf jest spójny
    return No, cycle_exists, is_consistent
```

Rys. 3. Kod funkcji DFS.

4. Wyniki działania algorytmu dla trzech przypadków

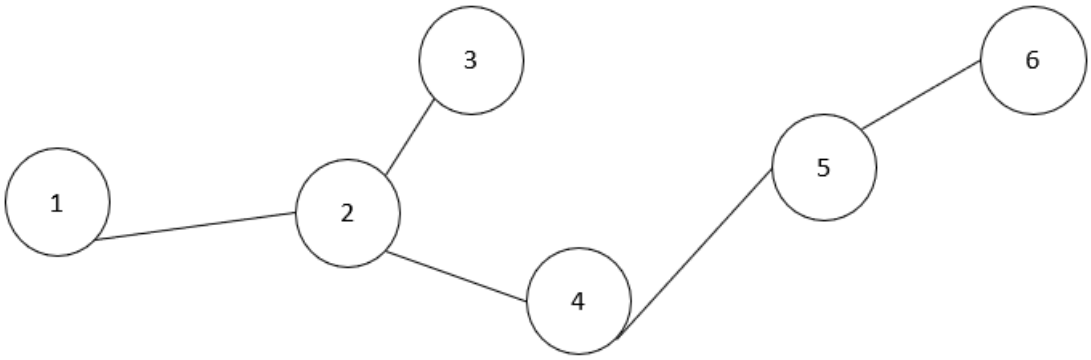
W celu eleganckiego wyświetlania wyników poza funkcją dopisałem następujące linijki kodu:

```
wynik1 = "1: " + str(ans1[0][0])
for i in range(1, len(ans1[0])):
    wynik1 += ", " + str(i+1) + ": " + str(ans1[0][i])

wynik1 += "\nCycle exists: " + str(ans1[1])
wynik1 += "\nGraph is consistent: " + str(ans1[2])

print(wynik1)
```

a. Graf spójny acykliczny

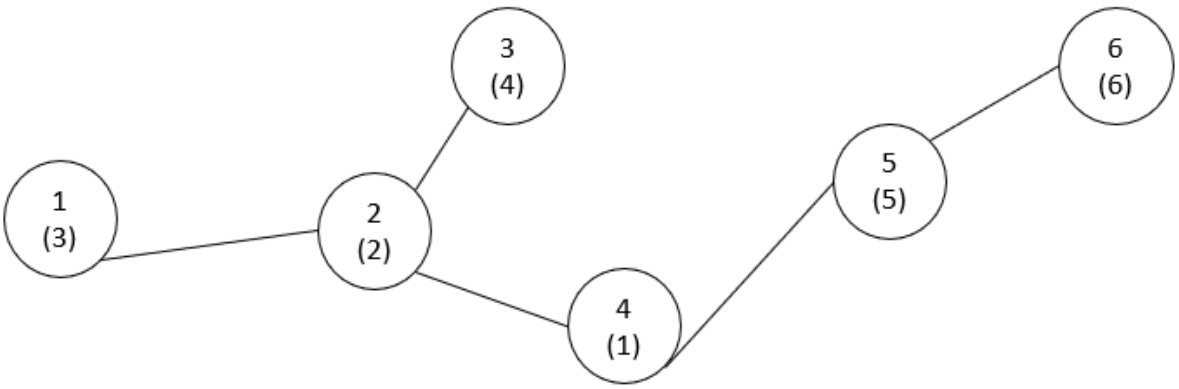


Rys. 4 Graf spójny acykliczny

Wynik działania programu dla wierzchołka 4 jako wierzchołka startowego:

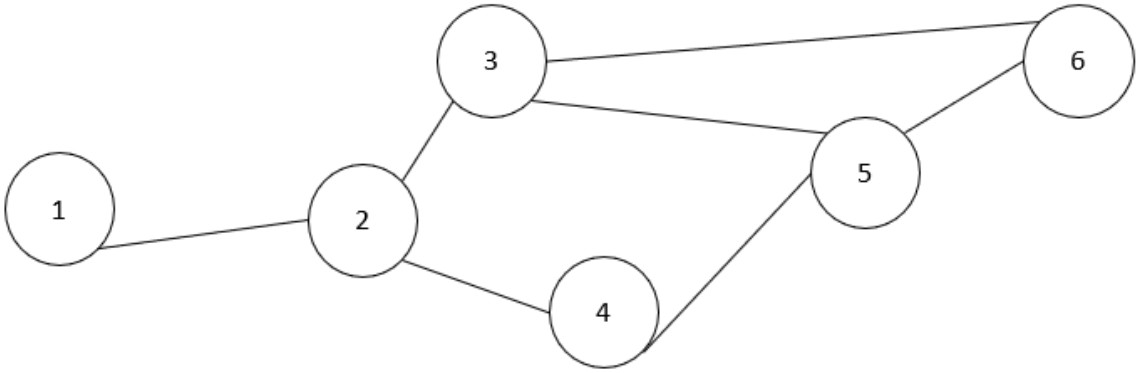
```
1: 4, 2: 2, 3: 1, 4: 3, 5: 5, 6: 6
Cycle exists: False
Graph is consistent: True
```

Jak widać program poprawnie wskazał, że mamy do czynienia z grafem spojonym pozbawionym cykli. Numeracja wierzchołków prezentowana jest w następujący sposób: *nowa_etykieta: stara_etykieta,*



Rys. 5 Graf z rysunku 4. z zaznaczoną w nawiasach nową numeracją

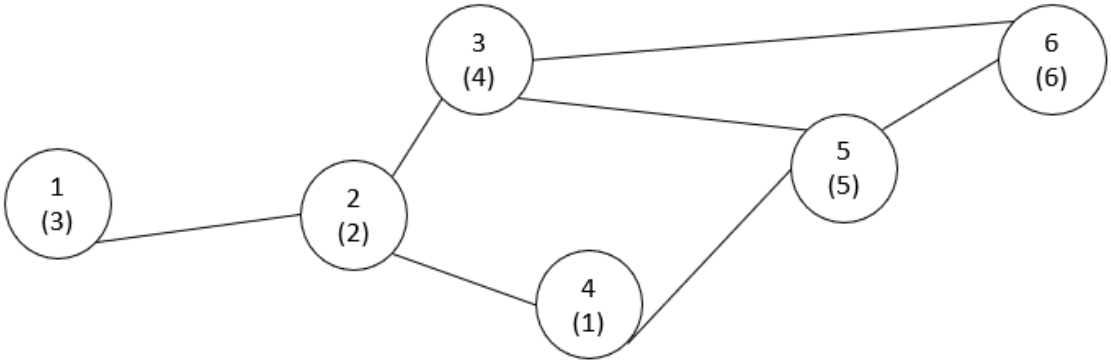
b. Graf spójny z cyklami



Rys. 6. Graf spójny z cyklami
Wynik:

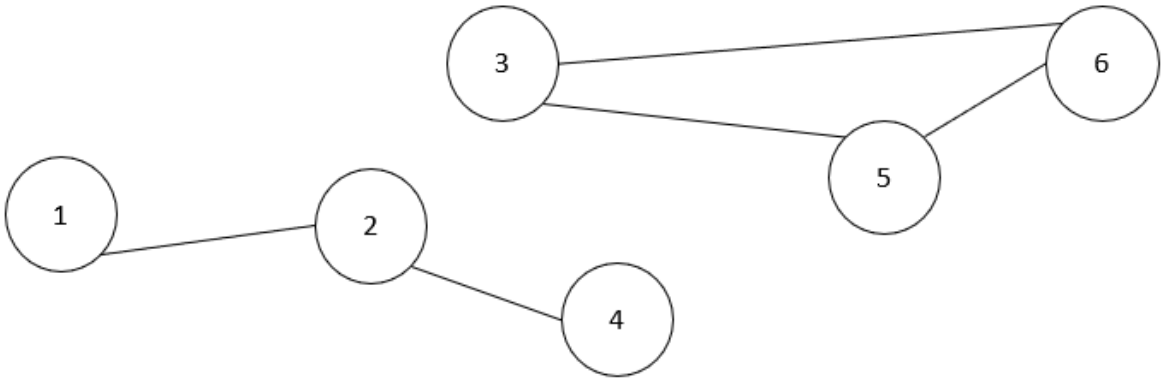
```
1: 4, 2: 2, 3: 1, 4: 3, 5: 5, 6: 6
Cycle exists: True
Graph is consistent: True
```

Tym razem jak widać program poprawnie wykrył istnienie cykli w grafie.



Rys. 7. Graf z rysunku 6. z zaznaczoną w nawiasach nową numeracją będącą wynikiem działania algorytmu

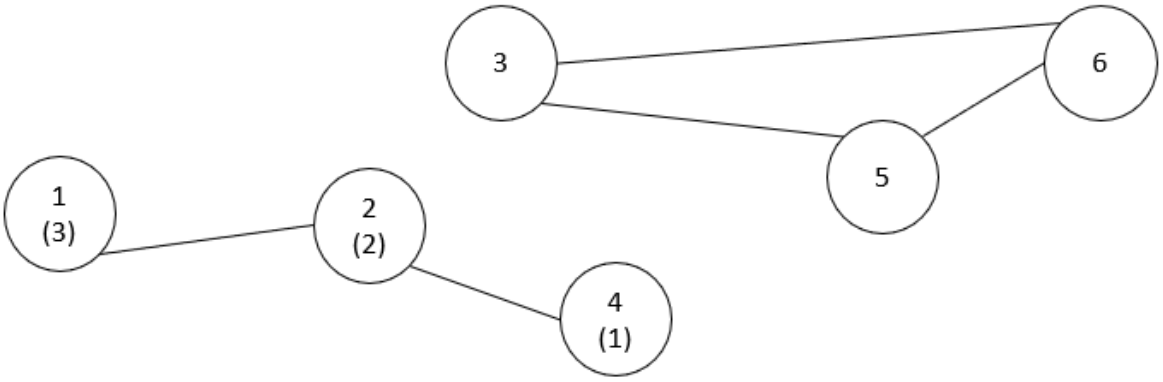
c. Graf niespójny z cyklami



Rys. 8 . Graf niespójny z cyklami

W tym przypadku wynik działania programu będzie uzależniony od wybranego wierzchołka startowego i tak np. dla wierzchołka 4 otrzymujemy następujący wynik:

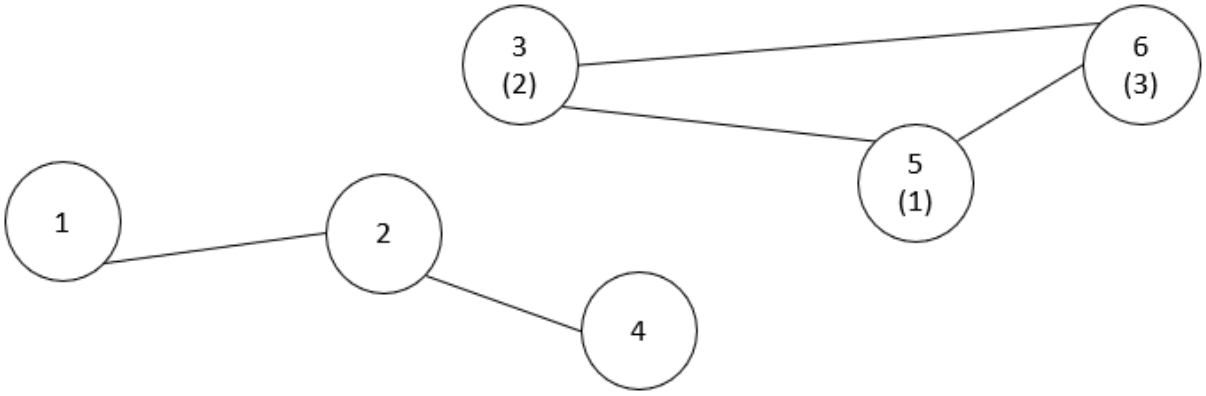
```
1: 4, 2: 2, 3: 1
Cycle exists: False
Graph is consistent: False
```



Rys. 9. Wynik działania dla grafu z rys. 8, dla wierzchołka startowego 4

Jak widać program nie wychwycił istnienia cykli w grafie oraz nie ponumerował wierzchołków w oddzielonej części grafu. Sprawdźmy jaki wynik uzyskamy zmieniając punkt startu na np. 5:

```
1: 5, 2: 3, 3: 6
Cycle exists: True
Graph is consistent: False
```



Rys. 10. Wynik działania dla grafu z rys. 8, dla wierzchołka startowego 5

W tym wypadku program zauważył istnienie cykli jednak nadal część wierzchołków pozostała nieponumerowana. Takiego zachowania można uniknąć nakazując programowi rozpocząć przeszukiwanie podgrafu po wykryciu niespójności, ponieważ jednak o takim mechanizmie nie było mowy we wcześniejszych rozważaniach oraz w pseudokodzie pominąłem jego implementacje.

5. Propozycje wykrywania innych własności grafu za pomocą przeszukiwania:

1. Wierzchołek rozpajający grafu – (za Wikipedią) Wierzchołek jest punktem rozpajającym, gdy jest korzeniem i ma więcej niż jednego syna, nie jest korzeniem, a dla przynajmniej jednego syna *s* zachodzi $low(s) \geq d(w)$, gdzie d() określa czas odwiedzenia *w* natomiast low(s) oznacza najmniejszy numer wierzchołka z którego można dojść do s inną drogą niż przez poprzednika w procedurze DFS, czyli w naszym przypadku drogą inną niż przez *w*.
2. Centrum grafu – zgodnie z definicją centrum grafu to wierzchołek, dla którego maksymalna odległość od innego wierzchołka jest najmniejsza, zatem aby wykryć taki wierzchołek za pomocą DFS należałoby uruchomić tę procedurę dla wszystkich wierzchołków. W czasie jej wykonywania zapamiętywać ilość przejść potrzebnych do dotarcia do wierzchołka rzędu 1, wybrać największą z tych wartości, a następnie porównać z wartościami uzyskanymi dla innych wierzchołków startowych
3. Sprawdzenie czy graf jest drzewem – z definicji drzewo to graf spójny, w którym nie istnieją cykle, skoro możemy sprawdzić te dwie własności możemy zatem stwierdzić czy graf jest drzewem.
4. Znajdowanie ścieżki między wierzchołkami *u* i *v* – należy uruchomić DFS dla u, użyć stosu do zapamiętania ścieżki pomiędzy startem, a obecnym wierzchołkiem, w chwili natrafienie na wierzchołek v zwrócić stos.
5. Sprawdzenie dwudzielności – za każdym razem gdy natrafiamy na nowy wierzchołek należy go pokolorować przeciwnie do jego rodzica, na koniec działania programu należy sprawdzić czy istnieje krawędź łącząca wierzchołki tego samego koloru , jeśli nie to znaczy, że graf jest dwudzielny
6. Dla grafu nieważonego DFS tworzy minimalne drzewo rozpinające.
7. Za pomocą BFS można odnaleźć wszystkich sąsiadów danego wierzchołka

8. W przypadku grafów skierowanych można wyszukać składowe silnie spójne

Inne zastosowania algorytmów przeszukiwania:

- Sortowanie topologiczne
- Rozwiązywanie labiryntów (z angielskiego „maze”)