Bit Algo
START

Wstęp do grafów

Uwaga o reprezentacji grafu

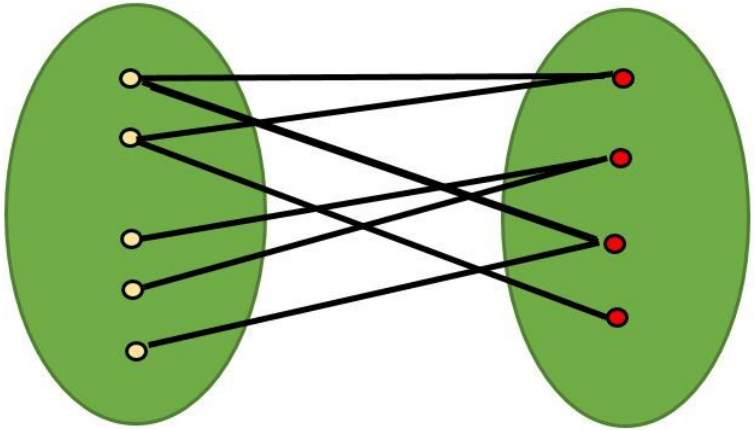
Jeżeli zadanie nie podaje explicite reprezentacji grafu, to zwykle jego pierwszą częścią jest wybranie najkorzystniejszej reprezentacji.

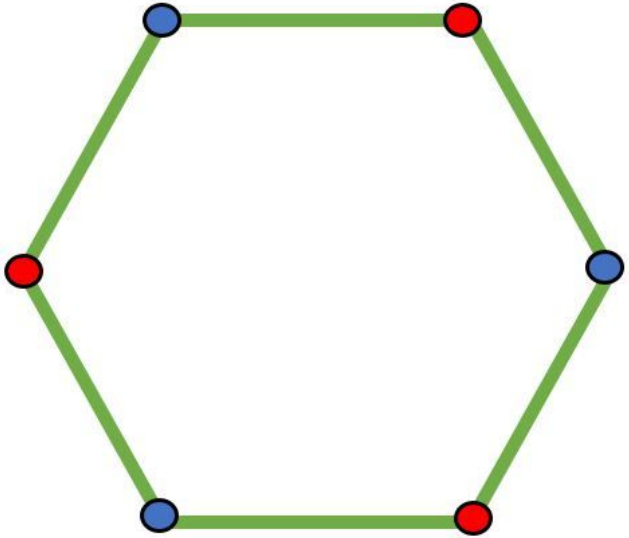
Wybór np. na kolokwium trzeba **uzasadnić**.

Zadanie 1

Napisz algorytm sprawdzający, czy graf jest dwudzielny (czyli czy da się podzielić jego wierzchołki na dwa zbiory, takie że krawędzie łączą jedynie wierzchołki z różnych zbiorów).

Zadanie 1 - wizualizacja





Zadanie 1 - rozwiązanie

Idea jest taka, żeby wykorzystać 2-kolorowanie grafu: jeżeli jest dwudzielny, to da się go rozdzielić na 2 części o różnych kolorach.

Będziemy kolorować wierzchołki na 2 kolory, np. czerwony i niebieski. Na początku wszystkie mają jakiś trzeci, “zerowy” kolor, np. biały.

Będziemy wykonywać BFS po grafie z pewnymi modyfikacjami. Wierzchołek-źródło kolorujemy na dowolny kolor (czerwony lub niebieski), a jego sąsiadów na “przeciwny” kolor.

Odwiedzając wierzchołek, musi być on już pokolorowany (bo jego poprzednik go pokolorował). Próbujemy kolorować jego sąsiadów na przeciwny kolor, niż wierzchołek. Jeżeli którykolwiek sąsiad ma “przeciwny” kolor, to wiemy, że graf **nie** jest dwudzielny.

Zadanie 2

Napisz algorytm sprawdzający, czy graf nieskierowany posiada cykl.

Zadanie 2 - rozwiązanie

Rozwiązanie: zauważmy, że jeżeli wejdziemy w cykl w grafie nieskierowanym, to przeglądając poddrzewo pierwszego wierzchołka cyklu do którego weszliśmy, będziemy ponownie próbowali się do niego dostać, przed wyjściem z niego.

Łopatologicznie: jak już coś pokolorowaliśmy, to tam byliśmy, więc jak tam chcemy wejść, to znaczy, że mamy cykl.

```
def DFSvisit(Graph, vertex):
    vertex.color = Gray
    isCycle = false
    for v in vertex.adj:
        if Graph[v].color == WHITE:
            isCycle = isCycle or DFSvisit(Graph, Graph[v])
        elif Graph[v].color == GRAY:
            isCycle = True
    vertex.color = BLACK
    return isCycle
```

Zadanie 3

Mówimy, że wierzchołek t w grafie skierowanym jest uniwersalnym ujściem (universal sink), gdy:

- z każdego innego wierzchołka v istnieje krawędź z v do t
- nie istnieje żadna krawędź wychodząca z t

Proszę podać algorytm znajdujący ujście (jeśli istnieje) przy reprezentacji macierzowej grafu.

Zadanie 3 - rozwiązanie naiwne

Optymalna reprezentacja: adjacency matrix (ponieważ łatwo sprawdzać krawędzie pomiędzy wierzchołkami).

Rozwiązanie: mamy znaleźć taki wierzchołek, że wszystko do niego wchodzi, a nic nie wychodzi. Musi mieć więc w macierzy adjacencji wiersz z samymi 0, a kolumnę z samymi 1 (poza "własnym" wierszem).

Sprawdzenie, czy wiersz ma same 0, to O(V); sprawdzenie, czy kolumna ma same 1 (poza jedną), to O(V).

Złożoność: $O(V^2)$

Zadanie 3 - rozwiązanie optymalne

Obserwacja: jeżeli A[x, y] = 1, to x nie może być ujściem (bo wychodzi z niego krawędź); jeżeli A[x, y] = 0, to y nie może być ujściem (bo nie wchodzi do niego krawędź z x).

Obserwacja 2: uniwersalne ujście może być tylko jedno.

Rozwiązanie: zrobić stos kandydatów na ujście, na początku są to wszystkie wierzchołki. Za każdym razem bierzemy 2 wierzchołki z góry i sprawdzamy powyższy warunek, zawsze eliminując jeden wierzchołek.

W n-1 krokach eliminujemy n-1 wierzchołków - O(n). Na koniec wystarczy dla jedynego pozostałego wierzchołka sprawdzić warunek w jego wierszu i kolumnie - O(n).

Złożoność: O(n)

Zadanie 4

Podaj algorytm, który mając na wejściu graf G reprezentowany przez listy sąsiedztwa sprawdza, czy dla każdej krawędzi u -> v istnieje także krawędź przeciwna.

Zadanie 4 - rozwiązanie naiwne

Rozwiązanie: sprawdzamy po kolei dla każdej krawędzi, czy istnieje krawędź w drugą stronę.

Problem: graf pełny, w którym każda lista ma długość V-1

Złożoność: O(V*E)

Zadanie 4 - rozwiązanie optymalne (tylko w C/C++)

Należy zaalokować (nie inicjalizować!) tablicę dwuwymiarową o rozmiarze V x V. Teraz wykonujemy 3 przejścia po krawędziach. W pierwszym przejściu mając indeksy wierzchołków porządkujemy je w parę (i, j), i<j i przypisujemy:

t[i][j] = 0. W drugim przejściu t[i][j]++. W trzecim przejściu sprawdzamy czy wszędzie t[i][j] == 2. Ostatecznie otrzymujemy złożoność O(V+E), ponieważ alokacja pamięci kwadratowej jest liniowa w językach C/C++, jeśli nie poprzedza całkowitej inicjalizacji tablicy.

W innych językach programowania (Java, Python) możemy uzyskać podobną złożoność stosując słownik, do którego wrzucamy krawędzie jako pary, i sprawdzamy, czy dla każdej pary (i,j) jest w słowniku para (j,i).

Zadanie 4 - rozwiązanie optymalne (tylko w C/C++)

```
#include<vector>
#include<algorithm> // min, max
using namespace std;

bool isUndirected(vector<vector<int>> & G){
  int V = G.size();
  int** edges = new int*[V];
  for (int i = 0; i < V; i++)
    edges[i] = new int[V];

  for (int i = 0; i < V; i++)
    for (int j = 0; j < G[i].size(); j++)
      edges[min(i, G[i][j])][max(i, G[i][j])] = 0;

  for (int i = 0; i < V; i++)
    for (int j = 0; k < G[i].size(); j++)
      edges[min(i, G[i][j])][max(i, G[i][j])]++;
```

```
for (int i=0; i<V; i++)
  for (int j=0; j<G[i].size(); j++)
    int x = min(i, G[i][j])
    int y = max(i, G[i][j])
    if (edges[x][y] != 2)
    {
      // zwolnij pamięć
      return False
    }
return True
```

Zadanie 4 - rozwiązanie na zbiorze krawędzi

```
def isUndirected(Graph):
    edgesSet = set()
    for v in range(len(Graph)):
        for neighbour in Graph[v]:
            edgesSet.add((v, neighbour))
    for v in range(len(Graph)):
        for neighbour in Graph[v]:
            if (neighbour, v) not in edgesSet:
                return False
    return True
```

Zadanie 5a

Dany jest ciąg przedziałów postaci $[a_i, b_i]$. Dwa przedziały można skleić, jeśli mają dokładnie jeden punkt wspólny. Podaj algorytm, który sprawdza, czy da się uzyskać przedział [a, b] poprzez sklejanie odcinków.

Zadanie 5a - stworzenie grafu

Na dobry początek trzeba w ogóle stworzyć graf z tych odcinków. Każdy wierzchołek to początek lub koniec odcinka, krawędzie są między tymi wierzchołkami, które tworzą odcinek (początek -> koniec).

Optymalna reprezentacja: zależy od grafu, listowo będzie pewnie prościej

Rozwiązanie: idziemy po odcinkach, z końca i początku tworzymy wierzchołek, jeżeli jeszcze go nie ma (trzymamy je np. w słowniku); po ew. utworzeniu dodajemy krawędź.

Zadanie 5a - uzyskanie przedziału [a, b]

Rozwiązanie: jeżeli w słowniku **nie** ma klucza a, to zwracamy False. Jeżeli klucz a jest, to puszczamy DFSa po grafie, aż dojdziemy do b i wtedy zwracamy True. Jeżeli po przeszukaniu całego grafu nie doszliśmy, to zwracamy False.

Złożoność: O(Vlog(V)) + O(V + E) = O(Vlog(V) + E), konstrukcja grafu + algorytm

Zadanie 5b

Dany jest ciąg przedziałów postaci $[a_i, b_i]$. Dwa przedziały można skleić, jeśli mają dokładnie jeden punkt wspólny. Podaj algorytm, który sprawdza, jaki najdłuższy odcinek można uzyskać, sklejając co najwyżej k odcinków.

To zadanie zrobimy, kiedy poznamy więcej algorytmów grafowych, bo wtedy będziemy mogli je zrobić w czasie O(k * V * E).

Zadanie 6

Znany operator telefonii komórkowej Pause postanowił zakończyć działalność w Polsce. Jednym z głównych elementów całej procedury jest wyłączenie wszystkich stacji nadawczych (które tworzą spójny graf połączeń). Ze względów technologicznych urządzenia należy wyłączać pojedynczo, a operatorowi dodatkowo zależy na tym, by podczas całego procesu wszyscy abonenci znajdujący się w zasięgu działających stacji mogli się ze sobą łączyć (czyli by graf pozostał spójny).

Proszę zaproponować algorytm podający kolejność wyłączania stacji.

Zadanie 6 - rozwiązanie

Rozwiązanie cz. 1: konstruujemy las drzew DFS w czasie O(V+E). Drzewa lasu rozpinają się na cały graf.

Rozwiązanie cz. 2: przechodzimy po każdym drzewie pre-order rekurencyjnie, jak dojdziemy do liścia, to dodajemy go jako kolejną stację do zamknięcia i usuwamy wierzchołek; gdy usuniemy wszystkie dzieci rodzica, to wtedy dodajemy go do usunięcia. Graf zachowa spójność z własności drzewa DFS (rozpina się na cały graf). Przejście po drzewie wszystkich wierzchołków to O(V).

Złożoność: O(V+E)

Bit Algo
START