Bit Algo
START

Wstęp do grafów

Pojęcia podstawowe

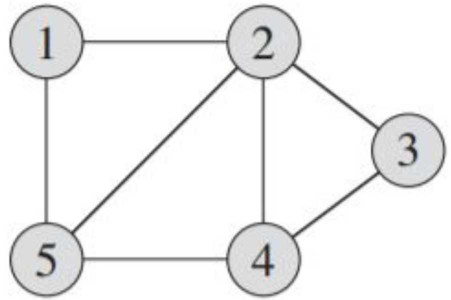
- **wierzchołek** - node, reprezentowany zwykle przez liczbę naturalną >= 0
- **krawędź** - edge, reprezentowana zwykle przez krawędź (u, v); w przypadku grafów skierowanych u -> v
- **graf** - zbiór wierzchołków i krawędzi
- **stopień wierzchołka** - liczba krawędzi, które wchodzą lub wychodzą z wierzchołka (są do niego **adjacentne**, adjacent); dla grafu skierowanego wyróżnia się out-degree i in-degree
- **V lub |V|** - liczba wierzchołków
- **E lub |E|** - liczba krawędzi
- **graf gęsty** - $|E| \sim |V|^2$, **graf rzadki** - $|E| << |V|^2$
- **multigraf** - dopuszczamy wielokrotne krawędzie między wierzchołkami, w tym pętle do siebie samego

Sposoby reprezentacji grafów

Lista krawędzi:

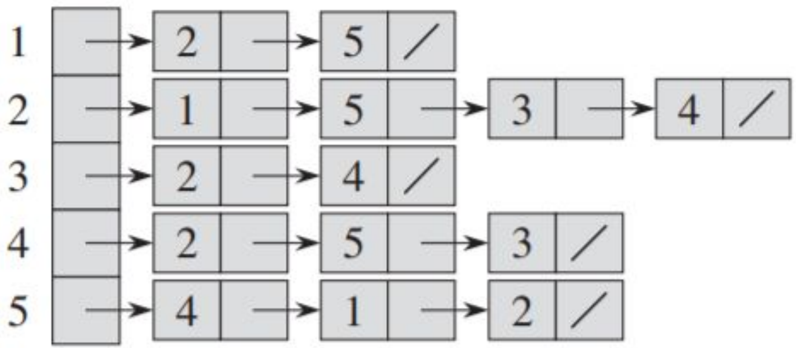
- lista np. [(0, 1), (2, 3), (4, 10), ...]
- zalety:
  - prostota
  - bardzo wygodne do zapisu/odczytu z plików
  - automatyczna eliminacja wierzchołków o stopniu 0
- wady:
  - niszczy graf, bo nie reprezentuje wierzchołków o stopniu 0
  - wolne, np. znalezienie konkretnego wierzchołka to O(E)

Sposoby reprezentacji grafów

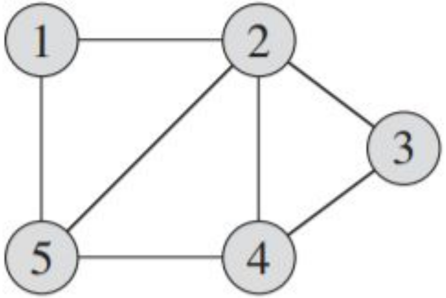


Listy adjacencji:

- dla każdego wierzchołka robimy listę jego sąsiadów, a same wierzchołki trzymamy w liście
- zalety:
  - dobre dla grafów rzadkich (stałe O(V+E) pamięci)
  - łatwe przeglądanie sąsiadów
- wady:
  - niewydajne dla grafów gęstych
  - przy operacjach na krawędziach trzeba pamiętać o zmianie stanu w listach obu wierzchołków
  - niewydajne dla operacji na krawędziach (np. sprawdzanie, czy krawędź jest w grafie)



Sposoby reprezentacji grafów



Macierz adjacencji:

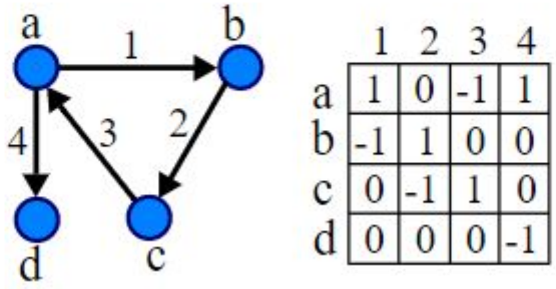
- robimy macierz każdy-z-każdym dla wierzchołków, gdzie [i, j] mówi, czy jest krawędź między tymi wierzchołkami (1 - jest, 0/None - nie ma)
- dla grafów nieskierowanych symetryczna, dla skierowanych nie
- zalety:
  - wydajne dla grafów gęstych (stałe $O(V^2)$ pamięci)
  - łatwe sprawdzanie, czy krawędź istnieje
  - łatwa reprezentacja grafów ważonych (waga zamiast 1, jak jest krawędź)
  - dla grafów nieskierowanych można zrobić macierz trójkątną
- wady:
  - niewydajne dla grafów rzadkich
  - przeglądanie sąsiadów to zawsze O(V)

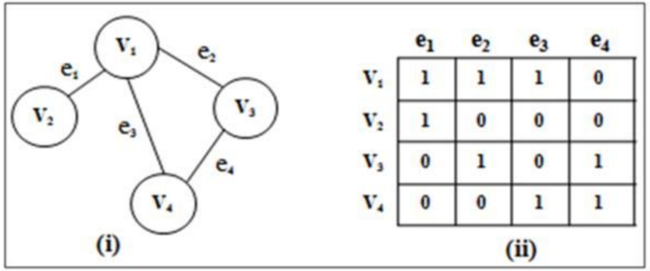
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Sposoby reprezentacji grafów

Macierz incydencji:

- wiersze - wierzchołki, kolumny - krawędzie
- [i, j] = 1, gdy i-ty wierzchołek jest incydentny z j-tą krawędzią, w przeciwnym przypadku 0
- dla grafów skierowanych 1, gdy krawędź wychodzi z wierzchołka, -1 gdy do niego wchodzi
- wymaga O(V+E) pamięci
- podobna w zaletach i wadach do macierzy adjacencji
- rzadko wykorzystywana





Sposoby reprezentacji grafów

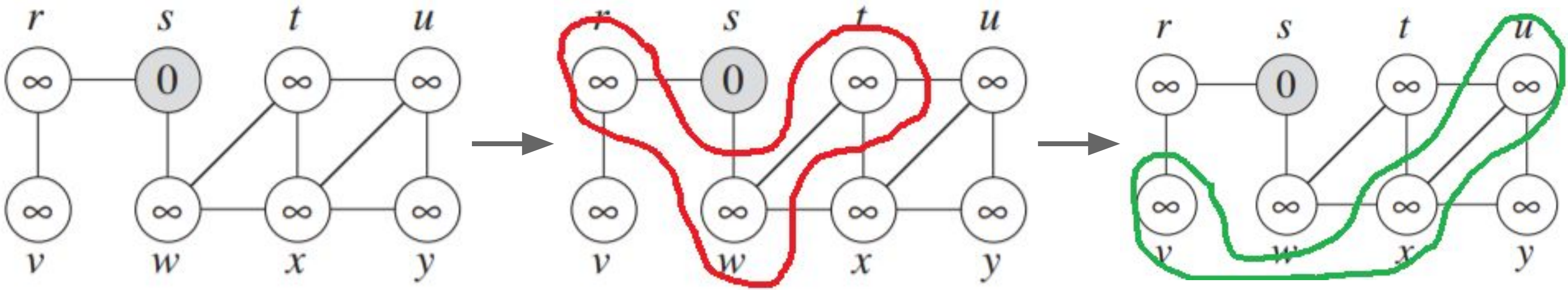
Inne struktury sąsiedztwa:

- wygodne, wykorzystywane w praktyce, **przeciętnie** bardzo wydajne
- zamiast list sąsiadów dla każdego wierzchołka zbiory sąsiadów:
  - dodawanie i usuwanie sąsiadów średnio O(1)
  - sprawdzanie, czy wierzchołek należy do sąsiadów średnio O(1)
  - łatwe robienie sumy (unii), różnicy etc.
- zamiast listy wierzchołków słownik (mapa) wierzchołków:
  - wierzchołek -> zbiór sąsiadów
  - dostęp do wierzchołka i jego zbioru sąsiadów średnio O(1)
  - łatwe stworzenie zbioru wierzchołków grafu (np. graph.keys())
  - łatwe robienie sumy (unii), różnicy etc. grafów
- można też wykorzystać zrównoważone struktury drzewiaste

Breadth-First Search (BFS)

- **problem:** na początku dostając graf jesteśmy “ślepi”, bo to dość nienaturalna dla komputera struktura, nie znamy jego kształtu, sąsiedztwa etc.
- **idea:** weźmy jakiś punkt początkowy (**źródło**), a później badajmy sukcesywnie najpierw swoich sąsiadów, później ich sąsiadów (kolejny “pierścień”), później jeszcze bardziej na zewnątrz ich sąsiadów itd.
- **breadth-first** - nazwa bierze się stąd, że najpierw “szeroko” sprawdzamy najbliższych sąsiadów (odległość 1), później “szerszy” zbiór dalszych sąsiadów (odległość 2) itd.
- **nadążanie za tym, co jest czym** - w wersji z Cormena nasze wierzchołki dostają atrybut color (white - nieodwiedzony, grey - w aktualnie zwiedzanym “pierścieniu” i jeszcze nieodwiedzony, black - już zbadany), w praktyce można też wykorzystywać zbiory (set)

BFS - “pierścienie” kolejnych sąsiadów

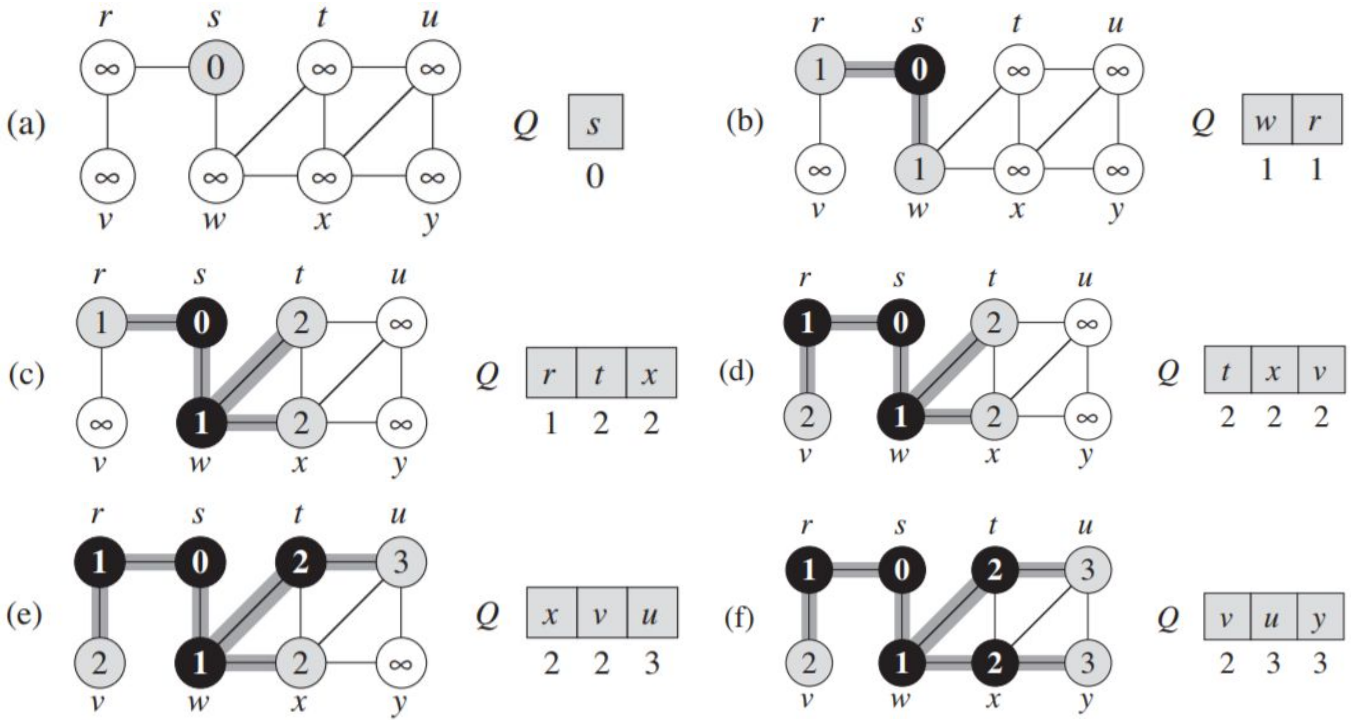


Algorytm BFS

- zakładamy, że od razu liczymy **odległość** każdego wierzchołka od źródła s (liczbę krawędzi po drodze), czyli `u.d`
- `u.π` oznacza **poprzednika** wierzchołka, czyli wierzchołek, poprzez który go poznaliśmy (jego sąsiada z poprzedniego “pierścienia”)
- szare wierzchołki (odwiedzane w tej chwili) trzymamy w kolejce FIFO, dzięki czemu “automatycznie” najpierw przechodzimy po pierwszym pierścieniu, a później od razu zaczynamy kolejny
- **odwiedzając** dany wierzchołek (linie 14-17) dla każdego jego jeszcze **nie odwiedzonego** sąsiada zmieniamy mu kolor, obliczamy odległość, zapisujemy poprzednika i wrzucamy go na koniec kolejki

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

BFS - przykład



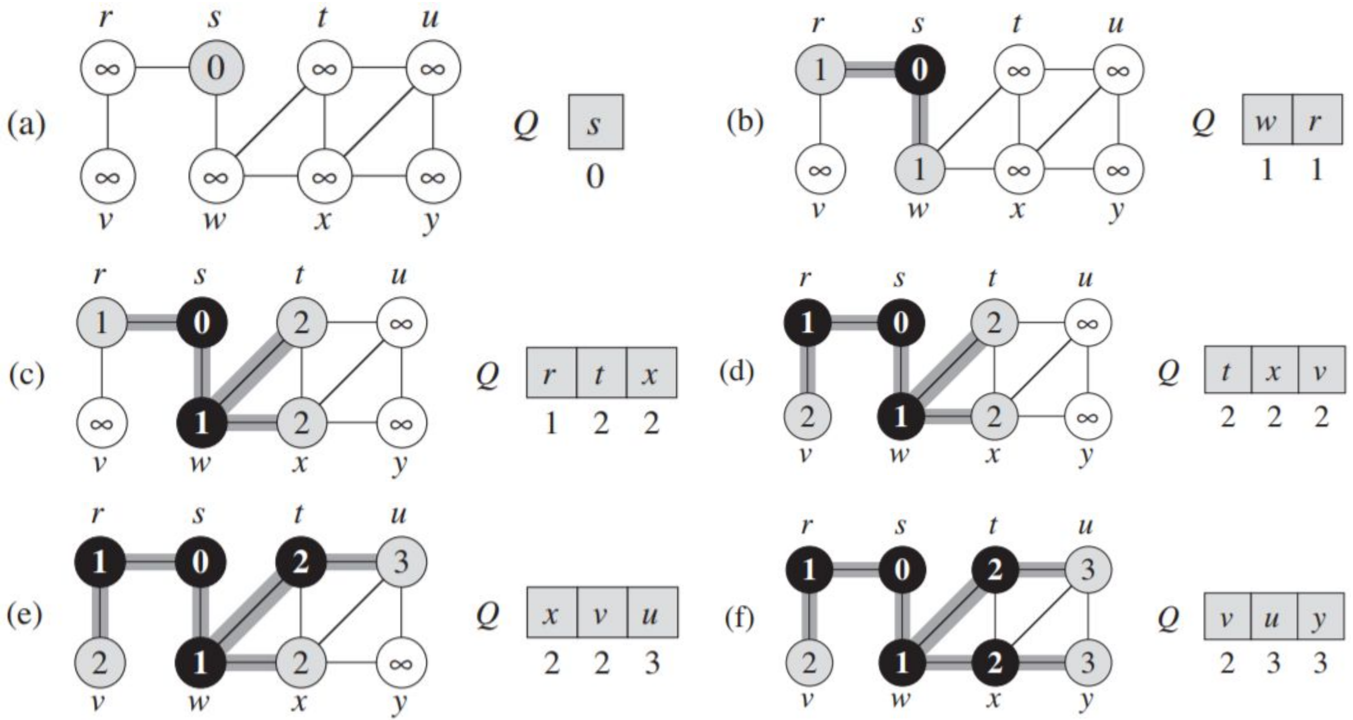

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

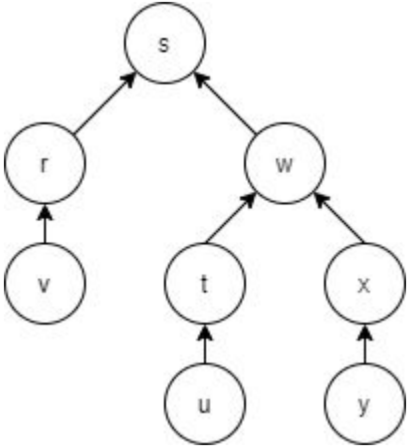
BFS - spostrzeżenia

- szare wierzchołki są niepotrzebne (ale przydatne do nauki!)
- “za darmo” mamy **najkrótszą odległość** od s do każdego wierzchołka z grafu
- “za darmo” mamy **poprzednika** dla każdego wierzchołka z grafu
- z powyższych wynika, że mamy też **drzewo BFS (breadth-first tree)**, a także **najkrótszą ścieżkę** od s do każdego wierzchołka z grafu (po prostu przechodzimy drzewo)
- odległość **nie** zależy od kolejności w listach adjacencji, ale kształt drzewa już **tak**
- BFS faktycznie przegląda krok po kroku całą część grafu **osiągalną** z wierzchołka s (w szczególności wykryje, jak coś nie jest z nim połączone)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Drzewo breadth-first





BFS - złożoność

- 1-4: O(V)
- 9-11: dla każdego wierzchołka będzie jedno ENQUEUE i jedno DEQUEUE, każda z tych operacji to O(1), czyli razem O(V)
- 12: suma długości wszystkich list adjacencji ma O(E), przez każdą listę przechodzimy tylko raz
- alternatywne 12: dla każdego z wierzchołków trzeba przejść po jego potencjalnych sąsiadach (wierszu macierzy) i odwiedzić sąsiadów

- **złożoność BFS dla listy adjacencji:** O(V+E)
- **złożoność BFS dla macierzy adjacencji:** O(V^2)

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

Depth-First Search (DFS)

- **problem:** często nie chcemy przeglądać całego grafu jak BFSem, zależy nam np. na szybkim dojściu do jakiegoś konkretnego wierzchołka
- **idea:** idźmy po kolejnych sąsiadach najdalej jak się da, w końcu dojdziemy do “rogu” (będziemy mieli odwiedzonych wszystkich sąsiadów aktualnego wierzchołka), później backtrackujemy
- **algorytm rekurencyjny:** wymaga rekurencji i stosu w przeciwieństwie do iteracyjnego BFSa
- **depth-first** - nazwa bierze się stąd, że najpierw idziemy “głęboko” w głąb grafu, eksplorujemy jego “głęboki koniec”, a później się cofamy coraz “płycej” (coraz bliżej źródła)

Drzewa i lasy depth-first, kolorowanie

- **depth-first tree:** analogiczne do drzewa BFS, bo też będziemy mieli poprzedników
- **depth-first forest:** ze względu na backtracking może nastąpić utworzenie depth-first tree, a później cofnięcie się do jego korzenia i stworzenie innego drzewa z tego wierzchołka; tworzy to wiele równoległych drzew, które nazywamy lasem
- **kolorowanie:** także będziemy korzystać z kolorów biały-szary-czarny, aby każdy wierzchołek odwiedzić dokładnie raz (gwarantuje rozdzielność drzew w lesie depth-first); szary ponownie jest niepotrzebny, ale dobry do nauki

Znaczniki czasowe (timestampy)

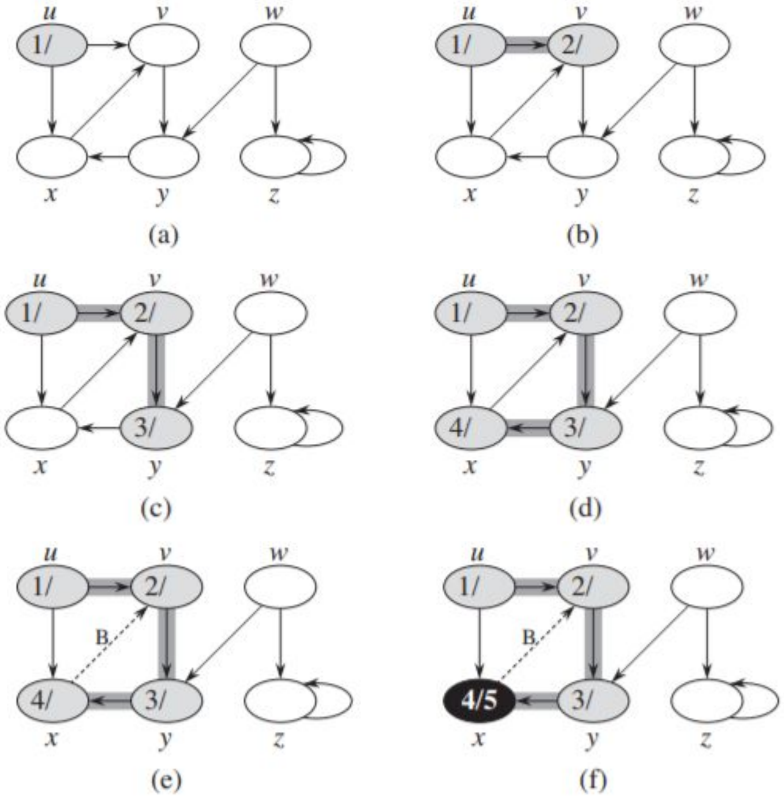
- jako **aktualny czas** traktujemy kolejne liczby naturalne
- **discovery** - v.d (**nie** distance!), zapisywany, gdy po raz pierwszy wchodzimy do wierzchołka (odkrywamy go) i zmieniamy kolor na szary
- **finish** - v.f, zapisywany, gdy odwiedzimy wszystkich sąsiadów danego wierzchołka i na pewno nigdy więcej go nie odwiedzimy (także backtrackingiem)
- znaczniki czasowe zapewniają informacje o grafie i działaniu w nim DFSa

Algorytm DFS

- uwaga: DFS wywoła się dla **każdego** fragmentu grafu, nawet, jeżeli nie są połączone!
- po raz pierwszy wchodząc do wierzchołka w DFS-Visit zapisujemy jego czas i zmieniamy kolor
- dla każdego z sąsiadów wierzchołka, jeżeli jeszcze go nie odwiedziliśmy, zapisujemy poprzednika i **od razu się dla niego wywołujemy (idziemy “głębiej)**
- po odwiedzeniu wszystkich sąsiadów kończymy wizytę, zapisujemy czas i zmieniamy kolor

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

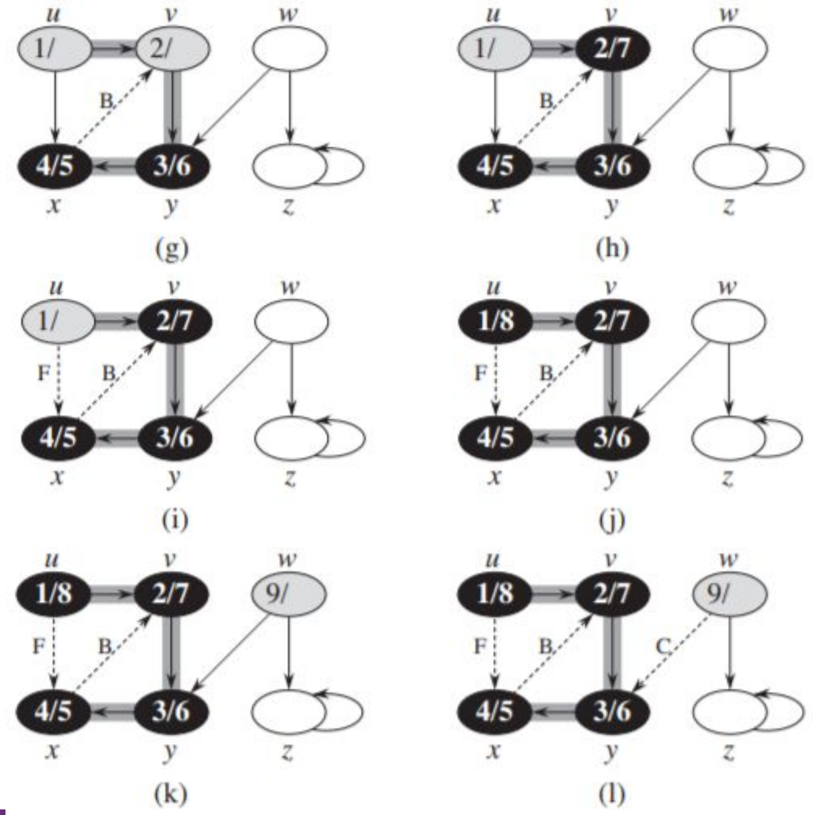
```
DFS-VISIT(G, u)
time = time + 1              // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]        // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK              // blacken u; it is finished
time = time + 1
u.f = time
```



DFS - przykład

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1                  // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]            // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                  // blacken u; it is finished
time = time + 1
u.f = time
```

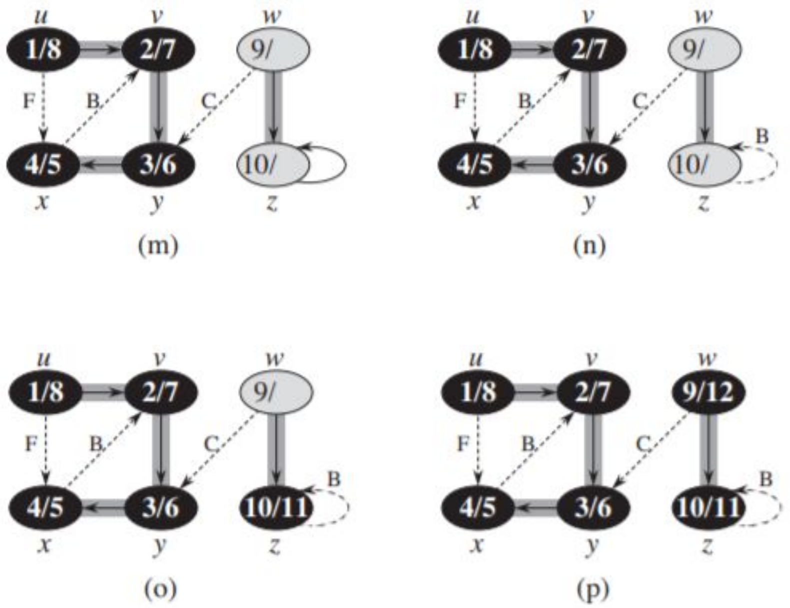


DFS - przykład

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1                 // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]           // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                 // blacken u; it is finished
time = time + 1
u.f = time
```

DFS - przykład




```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1                  // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]            // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                  // blacken u; it is finished
time = time + 1
u.f = time
```

DFS - złożoność

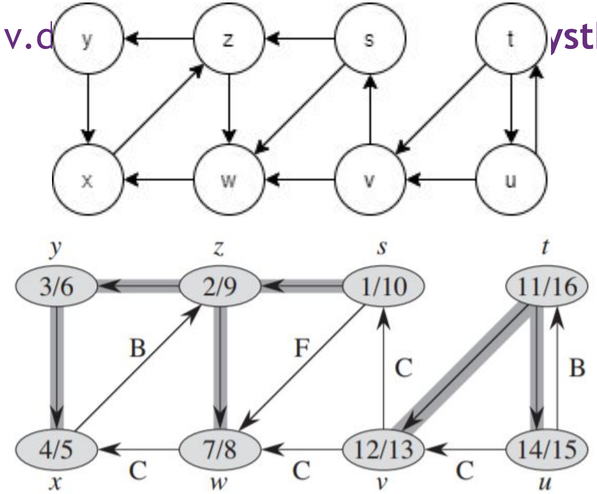
- 1-3 i 5-7 w DFS: O(V)
- DFS-Visit jest **wywoływane** tylko raz dla każdego wierzchołka (może się backtrackować, ale wywołanie jest tylko jedno)
- 4-7 w DFS-Visit: wywoła się zgodnie z powyższym tyle razy, ile mają w sumie listy adjacencji, czyli O(E)
- **złożoność DFS dla listy adjacencji:** O(V+E)
- **złożoność DFS dla macierzy adjacencji:** O(V^2)

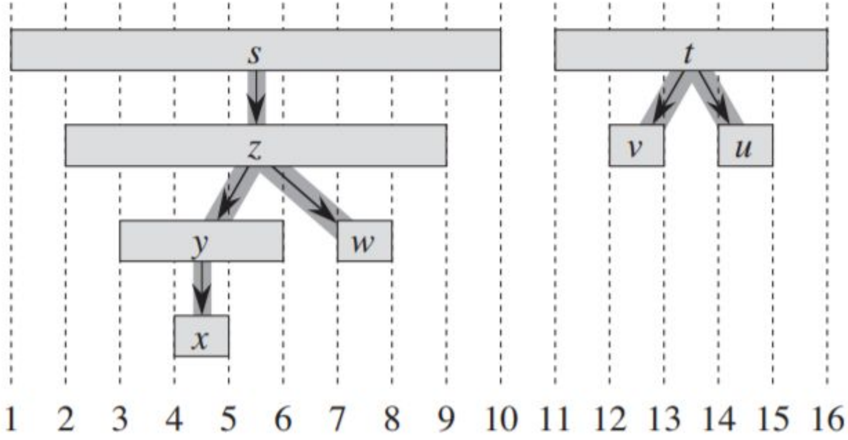
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1              // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]        // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK              // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

DFS - timestampy a drzewo depth-first

- z pewnego wierzchołka v idziemy w dół, “zagłębiając” się w graf, a później backtrackujemy
- mamy v.d (z chwili, kiedy weszliśmy) i v.f (kiedy zwiedziliśmy **wszystkie** poddrzewa i kończymy backtracking)
- **pomiędzy** v.d **ystkich dzieci**





Zastosowanie DFS - spójne składowe w grafie nieskierowanym

Cel: Mając na wejściu graf nieskierowany dany jako listy adjacencji przypisać każdemu wierzchołkowi etykietę spójnej składowej, do której należy.

Zastosowanie DFS - spójne składowe w grafie nieskierowanym

Podejście: Należy utworzyć strukturę danych, przechowującą etykietę spójnej składowej każdego wierzchołka. Może to być odpowiednie pole w strukturze wierzchołka, lub tablica.

```
class Vertex:
    def _init_(self, visited, adj, componentIndex)
```

Zastosowanie DFS - spójne składowe w grafie nieskierowanym

```
def findComponent(Graph):
    for vertex in Graph:
        vertex.visited = false
    componentIndex = 0
    for vertex in Graph:
        if not vertex.visited:
            DFSvisitcompnent(Graph, vertex, componentIndex)
            componentIndex += 1
```

Zastosowanie DFS - spójne składowe w grafie nieskierowanym

```
def DFSvisitcomponent(Graph, vertex, componentIndex):
    vertex.visited = true
    for v in vertex.adj:
        if not Graph[v].visited:
            Graph[v].componentIndex = componentIndex
            DFSvisitcomponent(Graph, Graph[v], componentIndex)
```

Zastosowanie DFS - cykl w grafie nieskierowanym

Cel: mając dany na wejściu graf reprezentowany jako listy adjacencji, sprawdź, czy graf zawiera cykl.
Tutaj przyda się nam DFS w wersji z trzema kolorami.

Zastosowanie DFS - cykl w grafie nieskierowanym

Rozwiązanie: zauważamy, 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.

Zastosowanie DFS - cykl w grafie nieskierowanym

```
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
```

Zastosowanie DFS - cykl w grafie nieskierowanym

```
def detectCycle(Graph):
    for vertex in Graph:
        vertex.color = WHITE
    isCycle = False
    for vertex in Graph:
        if vertex.color == WHITE:
            isCycle = isCycle or DFSvisit(GRaph, vertex)
    return isCycle
```

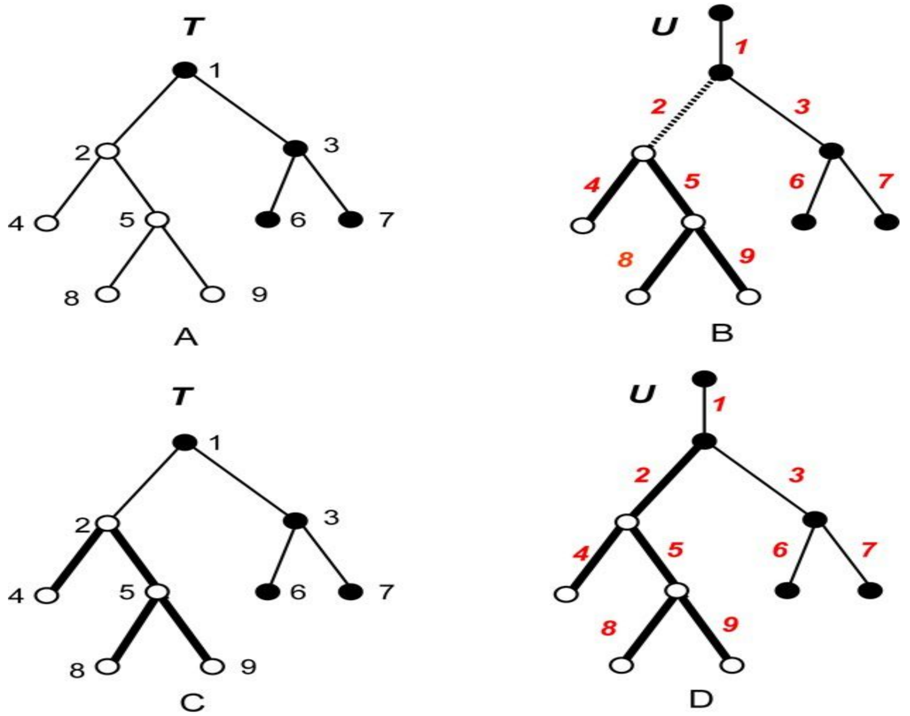
Warm up!

Dany jest spójny graf, potencjalnie bardzo gęsty. Podaj algorytm, który w czasie O(V), sprawdzi, czy graf zawiera cykl.

Zastosowanie timestamps z DFS

Dane jest nieskierowane drzewo, w postaci list adjacencji. Do zadanego drzewa zadajemy serię zapytań postaci i, j, gdzie i oraz j oznaczają indeksy wierzchołków.

Dla każdego zapytania chcemy informację, czy znajdują się na jednej ścieżce od roota, do jakiegoś liścia. Root jest dany. Przeprowadź liniowy (O(V + E)) preprocessing grafu, tak aby dało się na każde z zapytań odpowiedzieć w czasie O(1).

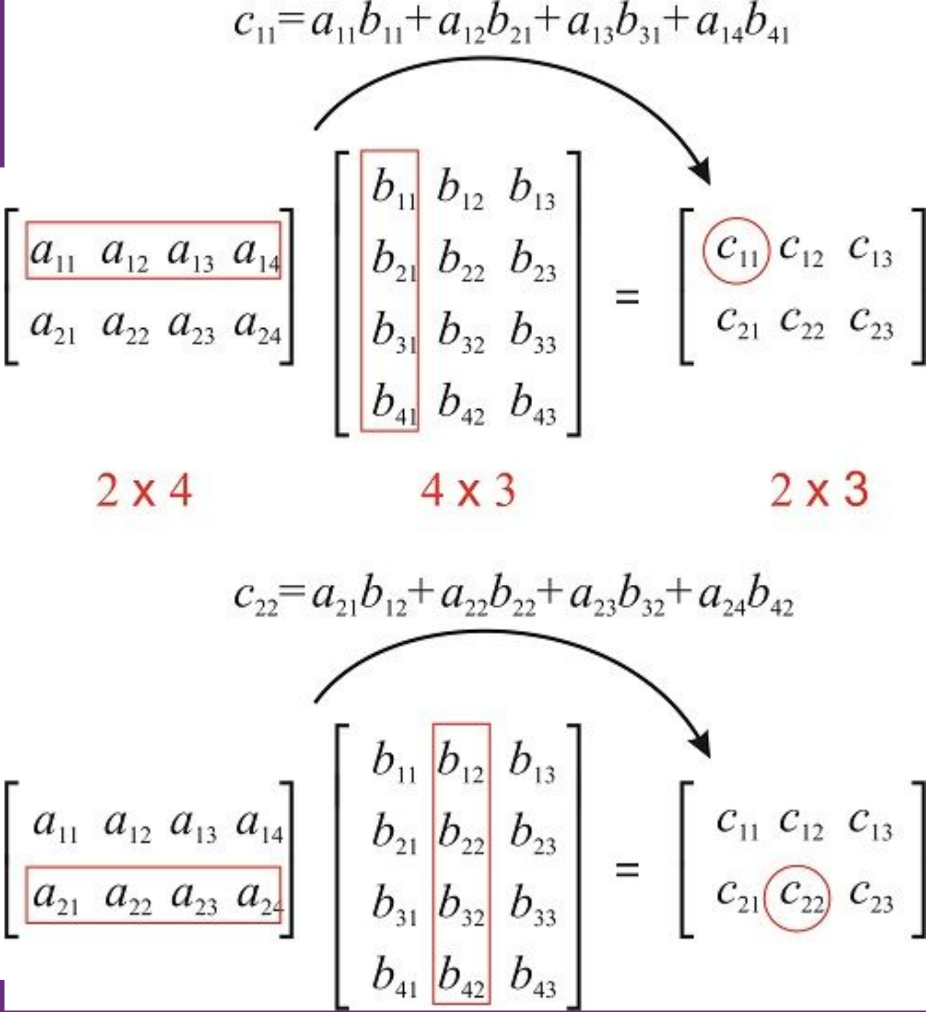


Zastosowanie timestamps z DFS

Rozwiązanie: Przeprowadzamy klasyczny DFS z dodaniem znaczników czasowych, zaczynając od wyróżnionego roota. Jeżeli czas wejścia wierzchołka i jest mniejszy, od czasu wejścia wierzchołka j, a jego czas wyjścia jest większy od czasu wejścia wierzchołka j, to j znajduje się w poddrzewie o roocie i. Czyli istnieje ścieżka. Jest to WKW. I odpowiedź na pytanie dokonuje się w O(1), ponieważ to tylko porównanie liczb.

Zastosowanie reprezentacji macierzowej grafu

Mając dany graf w postaci macierzy, sprawdź, czy istnieje w nim cykl o długości k.



Zastosowanie reprezentacji macierzowej grafu

Podnosząc macierz adjacencji do potęgi k, otrzymuję w indeksie [i][j] liczbę marszrut o długości k, z wierzchołka i do j. Należy zatem podnieść macierz adjacencji do potęgi k, i sprawdzić czy gdzieś w [i][i] mamy 1. Złożoność O(V^3*log(k)). W praktyce można osiągnąć złożoność O(V^2.7 * log(k)). W teorii… O(V^2.3 * log(k)).

Bit Algo

START