Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

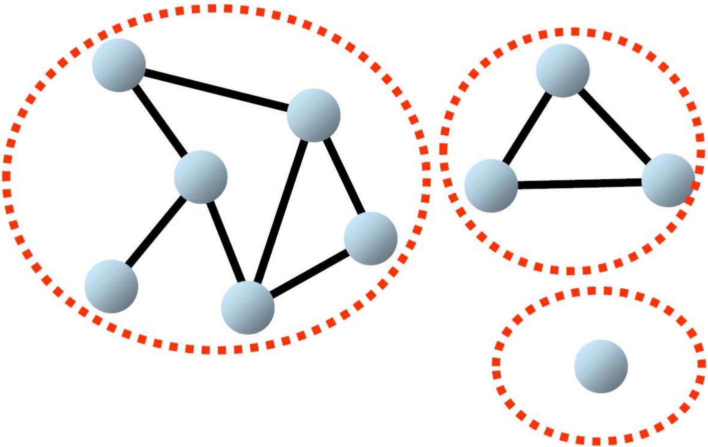
b.wozna@ujd.edu.pl

Wykład 3 i 4

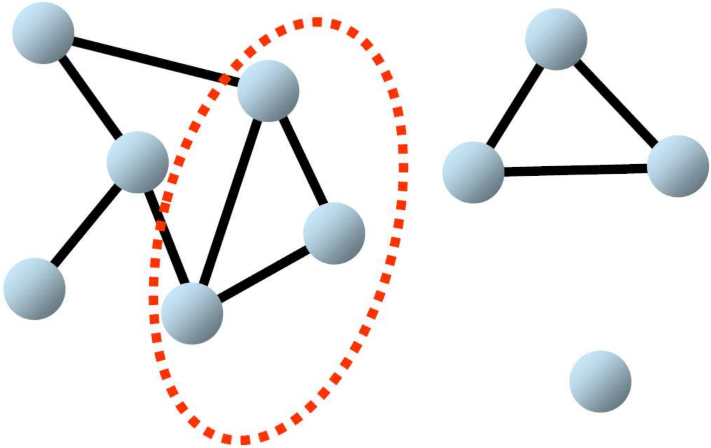
Spójność grafu

- Graf nieskierowany jest **spójny**, jeśli każda para wierzchołków jest połączona ścieżką.
- Każda **spójna składowa** grafu $G = (V, E)$ jest maksymalnym podzbiorem wierzchołków U zbioru V takim, że dla dowolnych dwóch wierzchołków z U istnieje łącząca je ścieżka w G.
- Jeżeli graf składa się z jednej spójnej składowej to mówimy, że jest **spójny** (ang. connected).
- Każdy graf nieskierowany można podzielić na jedna lub większą liczbę spójnych składowych (ang. connected components).
- Graf skierowany jest **silnie spójny**, jeśli każde dwa wierzchołki sa osiągalne jeden z drugiego.

Spójne składowe



Spójne podgrafy



Przeszukiwania grafu w głąb

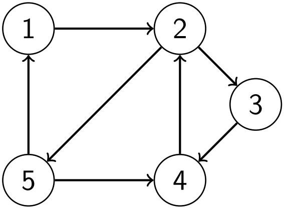
- Algorytm przeszukiwania grafu w głąb (ang. Depth-first search, DFS) to podstawowa metoda badania grafów.
- Algorytm DFS wykorzystuje się do badania spójności grafu - jeśli procedura wywołana dla pierwszego wierzchołka "dotrze" do wszystkich wierzchołków grafu to graf jest spójny.

Przeszukiwania grafu w głąb - idea

Dane wejściowe: Dowolny graf $G = (V, E)$, opcjonalnie wierzchołek początkowy $v \in V$.
Idea algorytmu $DFS(v)$**:**

1. Zaznacz wszystkie wierzchołki grafu jako nieodwiedzone.
2. Przejdź do początkowego wierzchołka, np. wskazanego v, zaznacz v jako odwiedzony i sprawdź, czy posiada nieodwiedzonych sąsiadów.
3. Jeżeli v posiada nieodwiedzonych sąsiadów, to przechodzimy to pierwszego nieodwiedzonego i powtarzamy całą procedurę.
4. Jeżeli wszyscy sąsiedzi pewnego wierzchołka u są zaznaczeni jako odwiedzeni, lub nie ma on sąsiadów, to algorytm wraca do wierzchołka, z którego u został osiągnięty.

Przeszukiwania grafu w głąb - idea



- DFS (1): 1,2,3,4,5
- DFS (2): 2,3,4,5,1
- DFS (3): 3,4,2,5,1
- DFS (4): 4,2,5,1,3
- DFS (5): 5,1,2,3,4

- Jeżeli podano wierzchołek początkowy, to algorytm zbada spójną składową zawierającą ten wierzchołek (**przypadek powyżej**).
- W przeciwnym razie algorytm przeszukuje kolejne spójne składowe, wybierając losowy wierzchołek z nowej spójnej składowej.

Przeszukiwanie w głąb - algorytm

- Za każdym razem, gdy przeszukiwanie w głąb odkrywa wierzchołek v podczas skanowania listy sąsiedztwa wcześniej odkrytego wierzchołka u, rejestruje to zdarzenie, ustawiając atrybut poprzednika v na u (i.e. $pre[v] = u$)
- Przeszukiwanie w głąb (DFS) koloruje wierzchołki podczas przeszukiwania, aby wskazać ich stan.
- Każdy wierzchołek jest **początkowo biały**, szarzeje, gdy zostaje odkryty w trakcie przeszukiwania, a czerwienieje, gdy jego lista sąsiedztwa została całkowicie zbadana.

Przeszukiwanie w głąb - Znaczniki czasowe w algorytmie

- Przeszukiwanie w głąb (DFS) przypisuje także znaczniki czasowe każdemu wierzchołkowi. Każdy wierzchołek v ma dwa takie znaczniki:
  - pierwszy znacznik $d[v]$ rejestruje moment, w którym v zostaje odkryty (i szarzeje),
  - drugi znacznik $f[v]$ rejestruje moment, w którym przeszukiwanie kończy badanie listy sąsiedztwa v (i czerwieni v).
- Znaczniki czasowe dostarczają ważnych informacji o strukturze grafu i są zwykle pomocne w analizowaniu zachowania przeszukiwania w głąb.
- Pseudokod na kolejnym slajdzie przedstawia klasyczny algorytm przeszukiwania w głąb. Zakładamy, że graf wejściowy G może być nieskierowany lub skierowany.
- Zmienna *time* jest zmienną globalną, której używamy do przypisywania znaczników czasowych.

Algorytm DFS - Graf reprezentowany przez listy sąsiedztwa

$DFS(G)$:

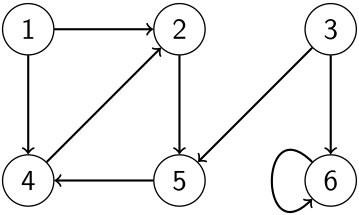
```
for each vertex u ∈ V do
    color[u] = WHITE
    pre[v] = NILL
end for
time = 0
for each vertex u ∈ V do
    if color[u] == WHITE then
        VISIT(G, u)
    end if
end for
```

$VISIT(G, u)$

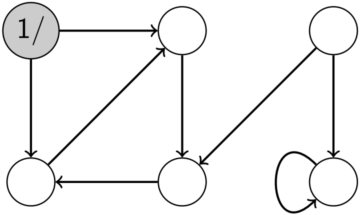
```
time = time + 1
d[u] = time
color[u] = GRAY
for each v ∈ Adj[u] do
    if color[v] == WHITE then
        pre[v] = u
        VISIT(G, v)
    end if
end for
color[u] = RED
time = time + 1
f[u] = time
```

Działanie algorytmu - Przykład

DFS: 3,6,1,2,5,4

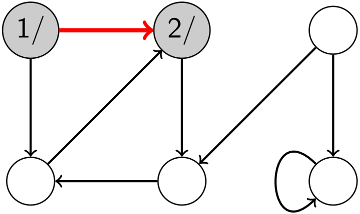


Krok: 1

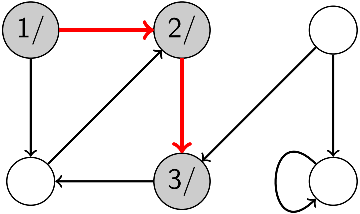


Działanie algorytmu - Przykład

Krok: 2

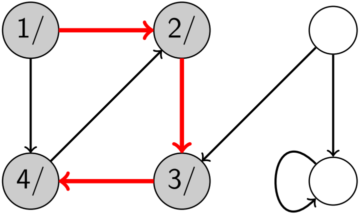


Krok: 3

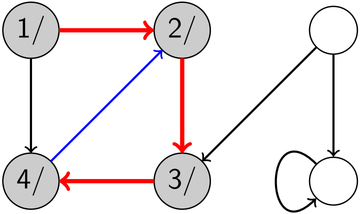


Działanie algorytmu - Przykład

Krok: 4

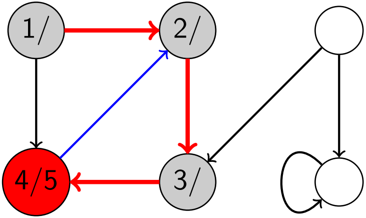


Krok: 5

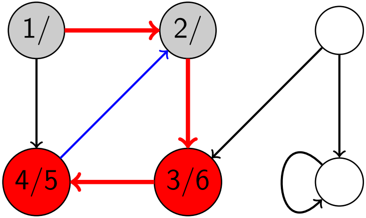


Działanie algorytmu - Przykład

Krok: 6

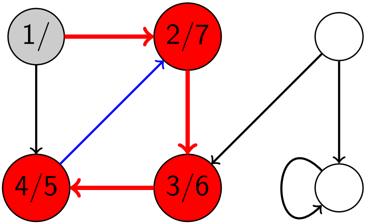


Krok: 7

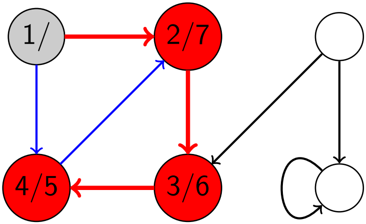


Działanie algorytmu - Przykład

Krok: 8

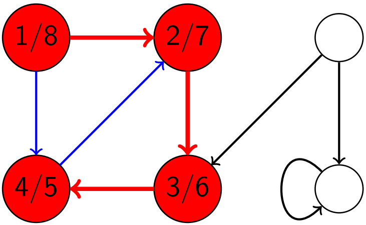


Krok: 9

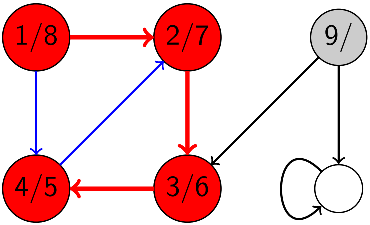


Działanie algorytmu - Przykład

Krok: 10

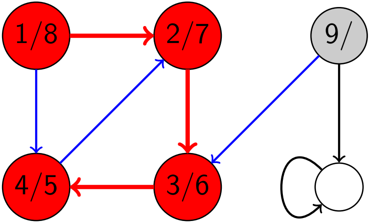


Krok: 11

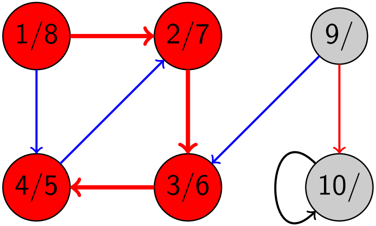


Działanie algorytmu - Przykład

Krok: 12

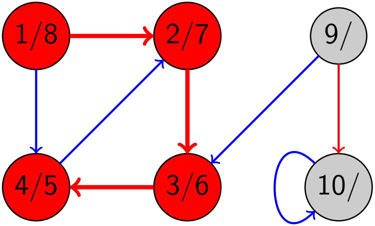


Krok: 13

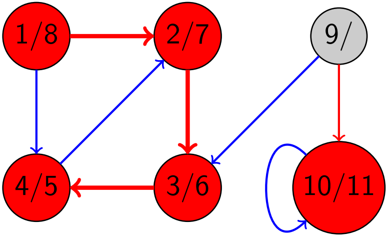


Działanie algorytmu - Przykład

Krok: 14

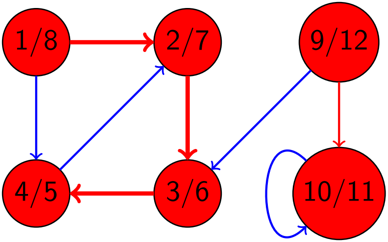


Krok: 15



Działanie algorytmu - Przykład

Krok: 16



KONIEC

Algorytm DFS - analiza złożoności

- Pętle w liniach 1-4 i 6-10 wymagają czasu proporcjonalnego do rozmiaru zbioru V, z wyłączeniem czasu potrzebnego na wykonanie wywołań procedury VISIT.
- Procedura VISIT jest wywoływana dokładnie raz dla każdego wierzchołka z V, ponieważ wierzchołek u, na którym wywoływany jest VISIT, musi być biały, a pierwszą rzeczą, jaką robi VISIT, jest malowanie wierzchołka u na szaro.
- Podczas realizacji VISIT pętla w liniach 4–9 wykonuje się $|Adj[u]|$ razy.
- Ponieważ $\sum_{v \in V} |Adj[v]| = \Theta(E)$ łączny koszt wykonania linii 4-9 procedury VISIT wynosi $\Theta(E)$.
- Czas działania DFS wynosi zatem $\Theta(V + E)$.

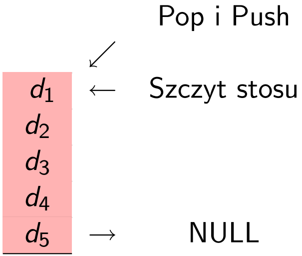
Stos

- Stos jest strukturą liniowo uporządkowanych danych, z których jedynie ostatni element, zwany wierzchołkiem, jest w danym momencie dostępny.
- W wierzchołku odbywa się dołączanie nowych elementów, również jedynie wierzchołek można usunąć.
- Cechą charakterystyczną stosu jest to, że dane są zapisywane i pobierane metodą **Last-In-First-Out (LIFO)** (pierwszy wchodzi, ostatni wychodzi).
- Działanie stosu jest często porównywane do stosu talerzy:
    - nie można usunąć talerza znajdującego się na dnie stosu nie usuwając wcześniej wszystkich innych.
    - nie można także dodać nowego talerza gdzieś indziej, niż na samą górę.

Stos - operacje

Niech $S = (d_1, d_2, \ldots, d_n)$ oznacza stos, wtedy:

- Odkładanie elementu na stos: $push(S, d) = (d, d_1, d_2, \ldots, d_n)$
- Pobieranie elementu ze stosu: $pop(S) = (d_2, \ldots, d_n)$, o ile $n > 1$
- Pobieranie elementu ze szczytu stosu bez jego usuwania: $top(S) = d_1$
- Sprawdzanie niepustości stosu: $empty(S)$ wtw., gdy $n = 0$



DFS - reprezentacja macierzowa grafu

Algorytm: Zwraca nieodwiedzony wierzchołek przyległy do wierzchołka *a*. Zwraca -1, jeżeli takiego wierzchołka nie ma.

```
function getUnVisitedVertex(Vertex a, BoolVector visited)
for b = 0 to n − 1 do
    if edge[a][b] = true and visited[b] = false then
        return b
    end if
end for
return -1
```

DFS dla grafu G, gdzie G jest macierzą sąsiedztwa

```
function DFS(Graph G, vertex a)
    declare visited[n]
    for k = 0 to n − 1 do visited[k] = false
    declare STACK s
    visited[a] = true                          // rozpocznij od wierzchołka a
    displayVertex(a)                           // wyświetl wierzchołek
    s.push(a)                                  // zapisz na stos
    while not s.empty() do
        b = getUnVisitedVertex(s.top(), visited)
        if b = −1 then    s.pop()    else
            visited[b] = true      // oznacz wierzchołek jako odwiedzony
            displayVertex(b)                   // wyświetl wierzchołek
            s.push(b)                          // zapisz na stos
        end if
    end while
end function
```

Przeszukiwania grafu w głąb - złożoność

Złożoność czasowa algorytmu DFS:

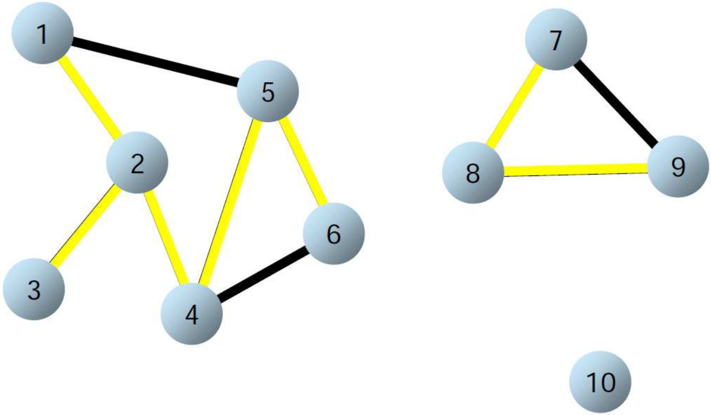
- graf reprezentowany jako listy sąsiedztwa: $\Theta(|V| + |E|)$, na co składa się początkowa inicjalizacja (zabiera $\Theta(|V|)$) i przechodzenie po wszystkich sąsiadach każdego wierzchołka (zabiera $\Theta(|E|)$ ponieważ suma długości wszystkich list sąsiedztwa wynosi $0(|E|)$).
- graf reprezentowany jako macierz sąsiedztwa: $\Theta(|V|^2)$.

Zastosowania DFS

Podstawowe zastosowania:

- Sprawdzanie spójności grafu.
- Wyznaczanie spójnych składowych grafu.
- Wyznaczanie silnie spójnych składowych (w wersji dla grafu skierowanego).
- Znajdowanie drogi w labiryncie.

Spójne składowe



Kolejka

- **Kolejka FIFO (First In First Out)** jest strukturą liniowo uporządkowanych danych, w której dołączać nowe dane można jedynie na koniec, a usuwać z początku.
- Procedura usunięcia danych z końca kolejki jest taka sama, jak w przypadku stosu, z tą różnicą, że usuwamy dane od początku a nie od końca.
- Działanie na kolejce jest intuicyjnie jasne, gdy skojarzymy ją z kolejką ludzi np. w sklepie. Każdy nowy klient staje na jej końcu, obsługa odbywa się jedynie na początku.

Kolejka - operacje

Niech $K = (d_1, d_2, \ldots, d_n)$ oznacza kolejkę, wtedy:

- Wstawianie elementu do kolejki: $enqueuq(K, d) = (d_1, d_2, \ldots, d_n, d)$
- Pobieranie elementu z kolejki: $dequeuq(K) = (d_2, \ldots, d_n)$, o ile $n > 1$
- Obsługiwanie pierwszego elementu z kolejki bez jego usuwania : $first(K) = d_1$
- Sprawdzanie niepustości kolejki: $empty(K)$ wtw., gdy $n = 0$

Początek kolejki $\rightarrow$ | d_1 | d_2 | d_3 | d_4 | d_5 | $\leftarrow$ Koniec kolejki

Przeszukiwanie grafu wszerz

- Z ustalonego wierzchołka źródłowego s przeglądamy kolejne wierzchołki z niego osiągalne.
- Wierzchołki w odległości k od źródłowego s odwiedzane są przed wierzchoakami w odległości $k+1$.
- Jeżeli po powyższym procesie pozostanie jakikolwiek nieodwiedzony wierzchołek u, kontynuujemy przeszukiwanie traktując go jako nowy wierzchołek źródłowy. Cały proces powtarzamy, aż wszystkie wierzchołki w grafie zostaną odwiedzone.
- Odwiedzane wierzchołki przechowywane są w kolejce FIFO.

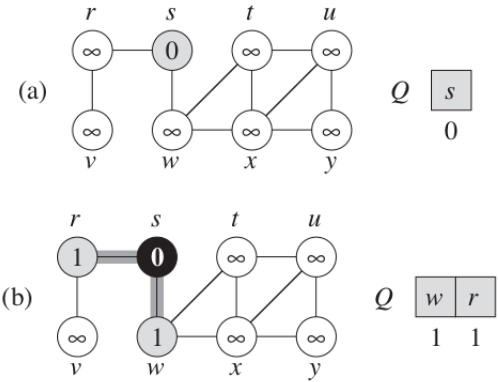
Algorytm BFS - graf G=(V,E) reprezentowany przez listy sąsiedztwa

$BFS(G, s)$**:**

for each vertex $u \in V - \{s\}$ **do**
    $color[u] = WHITE$
    $d[u] = \infty$
    $pre[u] = NIL$
end for
$color[s] = GRAY$
$d[s] = 0$
$pre[s] = NIL$
$EnQueue(Q, s)$

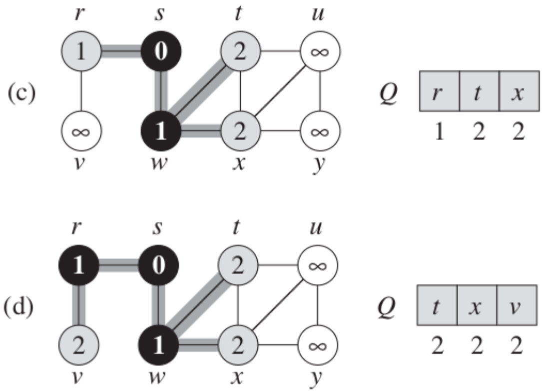
while $Q \neq \emptyset$ **do**
    $u = DeQueue(Q)$
    for each $v \in Adj[u]$ **do**
        if $color[v] == WHITE$ **then**
            $color[v] = GRAY$
            $d[v] = d[u] + 1$
            $pre[v] = u$
            $EnQueue(Q, v)$
        end if
    end for
    $DeQueue(Q)$
    $color[u] = BLACK$
end while

Algorytm BFS -przykład



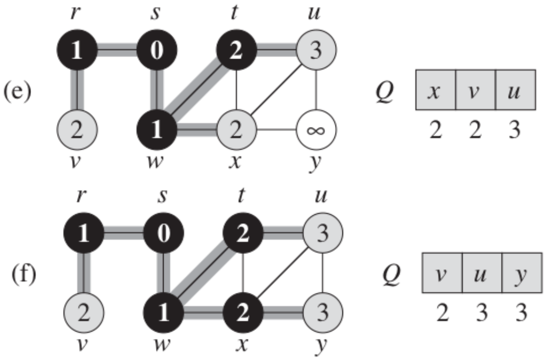
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm BFS -przykład



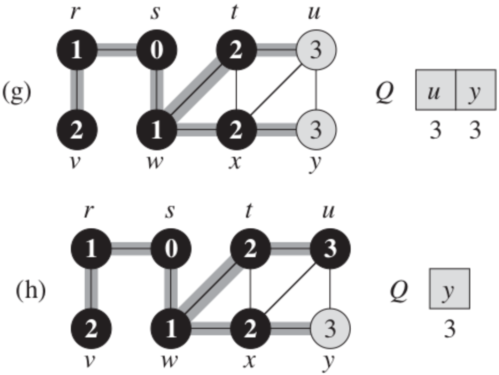
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm BFS -przykład



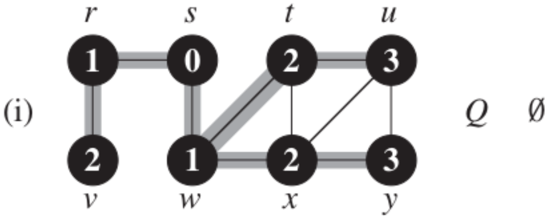
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm BFS -przykład



Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm BFS -przykład

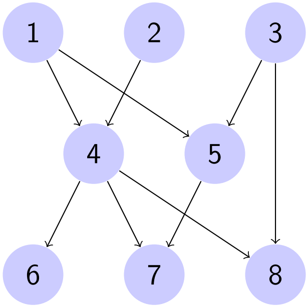


Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

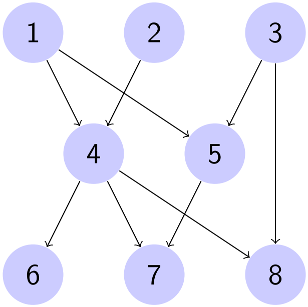
BFS - reprezentacja macierzowa grafu

```
BFS(Graph G,vertex a) { //G jest macierzą sąsiedztwa
 bool visited [n];
 for (k = 0; k < n; ++k) visited[k] = false;
 Queue q; // kolejka do przechowywania łwierzchoków
 visited[a] = true;  // rozpocznij od wierzcholka a
 displayVertex(a);   // wyswietl wierzcholek
 q.EnQueue(a);         // wstaw na koncu
 while (not q.empty()) {       // do opróżnienia kolejki,
   b = q.Front();        // pobierz piewszy wierzcholek
   q.DeQueue();            // usun go z kolejki
  // dopóki ma nie odwiedzonych sasiadów
  c = getUnVisitedVertex(b, visited);  // pobierz sasiada
  while (c != -1)
  {
    visited[c] = true;  // oznacz
    displayVertex(c);   // wyswietl
    q.EnQueue(c);            // wstaw do kolejki
    c = getUnVisitedVertex(b, visited);  // pobierz sasiada
  } // while
 } // while(kolejka nie jest pusta)
}
```

Algorytm BFS -przykład



Algorytm BFS -przykład



BFS(1): 1, 4, 5, 6, 7, 8
BFS(2): 2, 4, 6, 7, 8
BFS(3): 3, 5, 8, 7
BFS: 1, 4, 5, 6, 7, 8, 2, 3

Przeszukiwania grafu wszerz - złożoność

Złożoność czasowa algorytmu BFS:

- graf reprezentowany jako listy sąsiedztwa: $\Theta(|V| + |E|)$, na co składa się początkowa inicjalizacja (zabiera $\Theta(|V|)$) i przechodzenie po wszystkich sąsiadach każdego wierzchołka (zabiera $\Theta(|E|)$ ponieważ suma długości wszystkich list sąsiedztwa wynosi $0(|E|)$).
- graf reprezentowany jako macierz sąsiedztwa: $\Theta(|V|^2)$.