MATEMATYKA W KONKURSACH ALGORYTMICZNO-PROGRAMISTYCZNYCH

OTWARTY WEB-KURS

Webinarium przeprowadzone
w ramach projektu
"Mistrzostwa w Algorytmice
i Programowaniu - Uczniowie",
finansowanego przez:

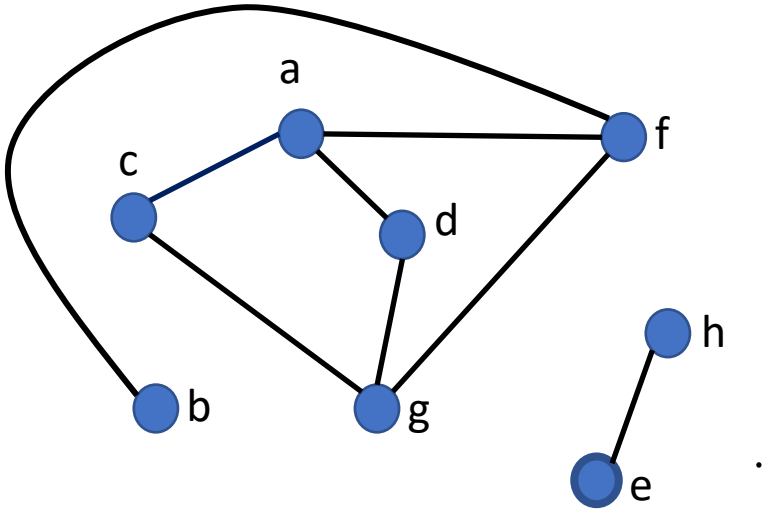
Algorytmy grafowe - podstawy

Krzysztof Diks

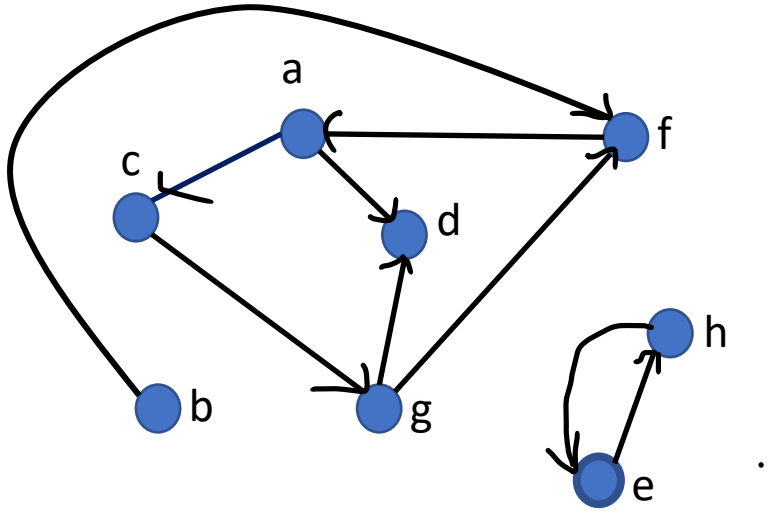
Grafem (skończonym) nazywamy każdą parę zbiorów (V,E), w której V to skończony zbiór elementów zwanych **wierzchołkami** grafu, natomiast E jest podzbiorem zbioru wszystkich par różnych wierzchołków.

Elementy zbioru E nazywamy **krawędziami** grafu. W przypadku, gdy pary są nieuporządkowane mówimy o **grafie nieskierowanym**.

W przypadku, gdy kolejność wierzchołków w parze jest istotna mówimy o **grafie skierowanym**



graf nieskierowany
V = {a, b, c, d, e, f, g, h}
E = {{b,f}, {a,c}, {a,d}, {a,f}, {g,c}, {g,f}, {d,g}, {e,h}}



graf skierowany
V = {a, b, c, d, e, f, g, h}
E = {(b,f), (a,c), (a,d), (f,a), (c,g), (g,f), (g,d),
(e,h), (h,e)}

Reprezentacja macierzowa grafu – macierz (czasami specjalizowana) sąsiedztwa

G = (V,E) – graf nieskierowany
$|V| = n$, $V = \{1, 2, \ldots, n\}$
E – zbiór dwuelementowych podzbiorów V
$|E| = m$, $0 \le m \le n(n-1)/2$
Oznaczenie: dla $\{u,v\} \in E$, piszemy $u - v$

A – macierz n x n

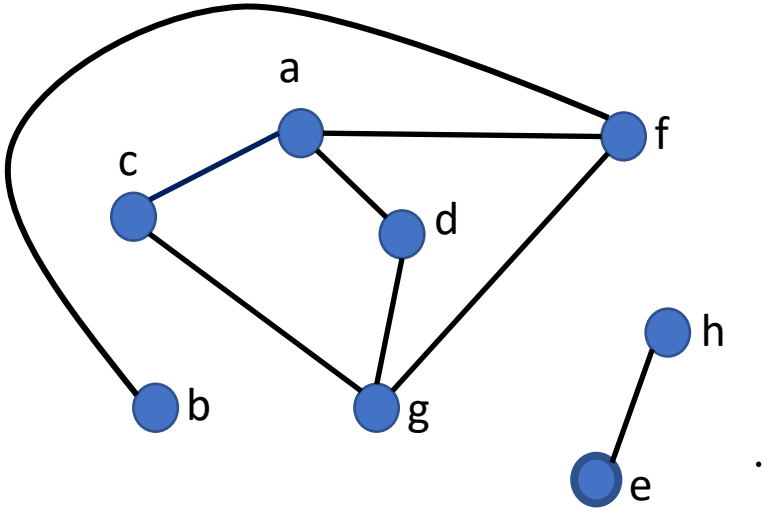
$$A[u,v] = \begin{cases} 0 & u - v \notin E \\ 1 & u - v \in E \end{cases}$$

G = (V,E) – graf skierowany
$|V| = n$, $V = \{1, 2, \ldots, n\}$
E – zbiór dwuelementowych uporządkowanych par różnych elementów z V
$|E| = m$, $0 \le m \le n(n-1)$
Oznaczenie: dla $(u,v) \in E$, piszemy $u \to v$

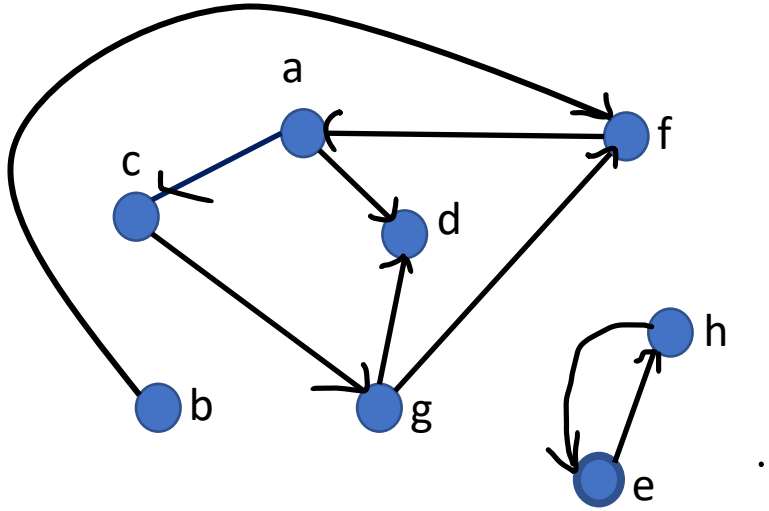
A – macierz n x n

$$A[u,v] = \begin{cases} 0 & u \to v \notin E \\ 1 & u \to v \in E \end{cases}$$

Rozmiar reprezentacji grafu – $\Theta(n^2)$



.



.

1	a
2	b
3	c
4	d
5	e
6	f
7	g
8	h

A	1	2	3	4	5	6	7	8
1	0	0	1	1	0	1	0	0
2	0	0	0	0	0	1	0	0
3	1	0	0	0	0	0	1	0
4	1	0	0	0	0	0	1	0
5	0	0	0	0	0	0	0	1
6	1	1	0	0	0	0	1	0
7	0	0	1	1	0	1	0	0
8	0	0	0	0	1	0	0	0

A	1	2	3	4	5	6	7	8
1	0	0	1	1	0	0	0	0
2	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1
6	1	0	0	0	0	0	0	0
7	0	0	0	1	0	1	0	0
8	0	0	0	0	1	0	0	0

Przykład zastosowania macierzy sąsiedztwa – problem najlżejszych ścieżek pomiędzy wszystkimi parami wierzchołków

Dane

G = (V,E) – graf skierowany, w: E → Z^+ (dodatnie liczby całkowite)

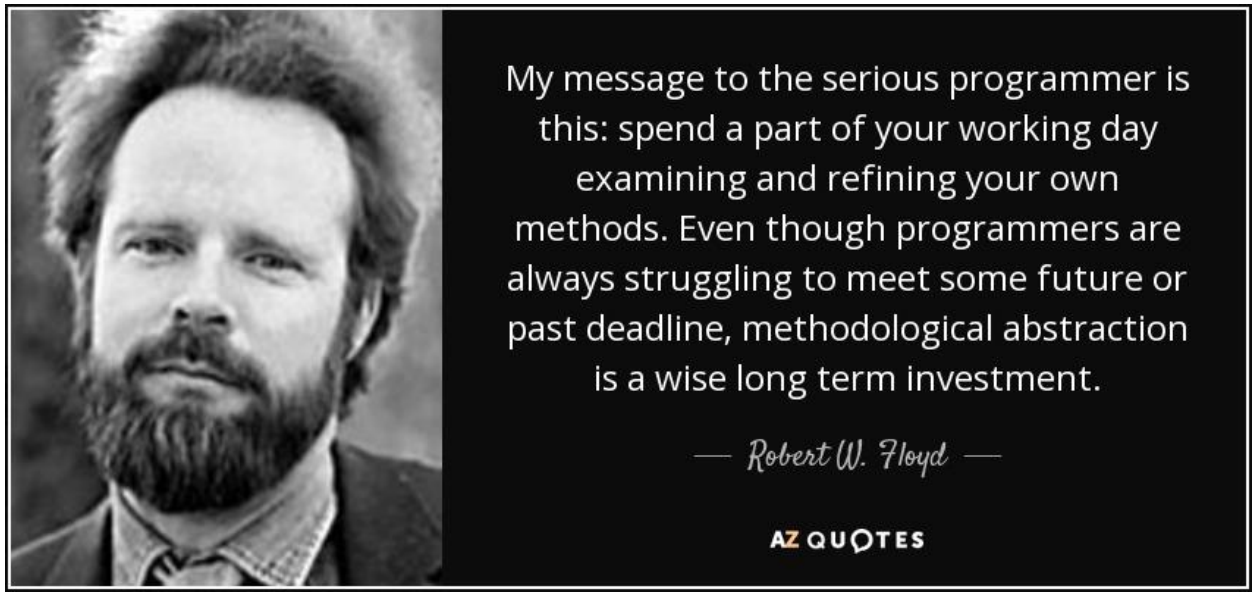
Wynik

W[1..n,1..n] – macierz wag najlżejszych ścieżek pomiędzy wierzchołkami;
jeśli w grafie nie ma ścieżki z i do j to W[i,j] = +∞

Ponieważ rozmiar wyniku jest kwadratowy w rozwiązaniu założymy, że graf jest zadany przez zmodyfikowaną macierz sąsiedztwa A[1..n,1..n] zdefiniowaną jak następuje:

$$A[i,j] = \begin{cases} 0 & i = j \\ w(i \to j) & i \to j \in E \\ +\infty & i \neq j, i \to j \notin E \end{cases}$$

Algorytm Floyda-Warshalla, 1962



1936 - 2001

1935 – 2006
(https://en.wikipedia.org/wiki/Stephen_Warshall)

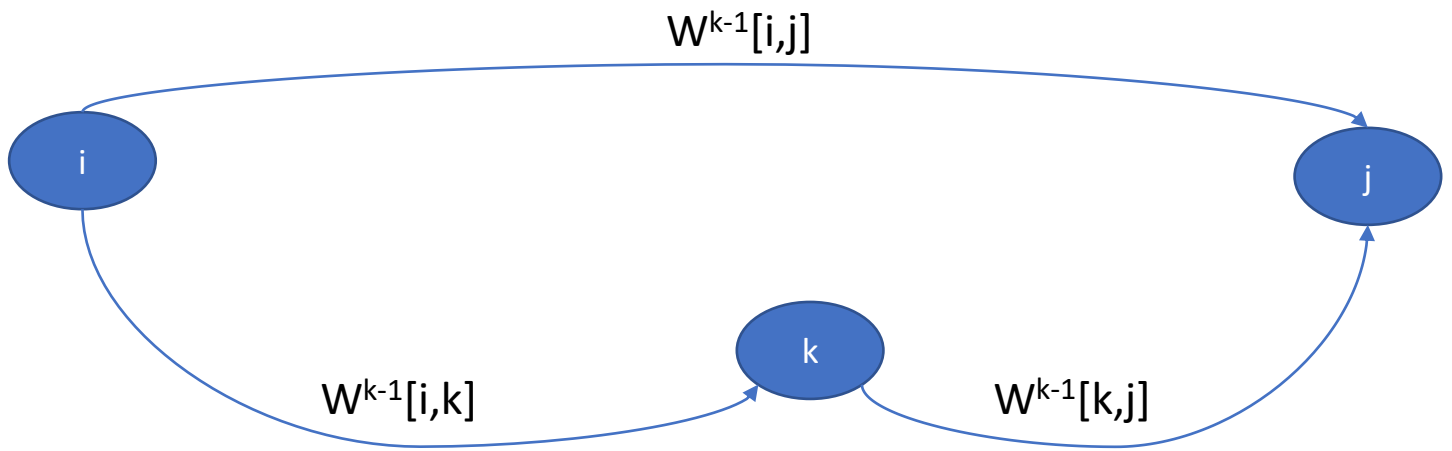
Idea:

$W^k[i,j]$ = waga najlżejszej ścieżki z i do j, na której każdy wewnętrzny wierzchołek (poza i oraz j) jest nie większy od k

Zauważmy, że $W^0[i,j] = A[i,j]$.

$W^{k-1} \longrightarrow W^k$

$$W^k[i,j] = MIN(W^{k-1}[i,k] + W^{k-1}[k,j], W^{k-1}[i,j])$$



Algorytm Floyda-Warshalla – wprost z wcześniejszych rozważań

W^0 := A;
for k ∈ [1, 2, …,n] do
  for i ∈ [1, 2, …,n] do
    for j ∈ [1, 2, …,n] do
      $W^k[i,j] = MIN(W^{k-1}[i,k] + W^{k-1}[k,j], W^{k-1}[i,j])$;

return W^n;

Czas – $\Theta(n^3)$
Pamięć - $\Theta(n^3)$

Naturalna poprawka oszczędzająca pamięć

W := A;
for k ∈ [1, 2, …,n] do
  for i ∈ [1, 2, …,n] do
    for j ∈ [1, 2, …,n] do
      W[i,j] = MIN(W[i,k] + W[k,j], W[i,j]);

return W;

Czas – $\Theta(n^3)$
Pamięć - $\Theta(n^2)$

Reprezentacja listowa grafu (czasami specjalizowana) – listy sąsiedztwa

graf nieskierowany

L[1..n] – tablica list sąsiadów wierzchołków

L[u] – lista sąsiadów wierzchołka u;

w przypadku grafu ważonego z każdym wierzchołkiem $v \in L[u]$ pamiętamy wagę w(v) - wagę krawędzi $u - v$

graf skierowany

$L^+[1..n]$, $L^-[1..n]$– tablice list sąsiadów wierzchołków

$L^+[u]$ – lista sąsiadów wierzchołka u, do których prowadza krawędzie o początku w wierzchołku u

$L^-[u]$ – lista sąsiadów wierzchołka u, od których prowadza krawędzie o końcu w wierzchołku u

rozmiar reprezentacji – $\Theta(n+m)$; czas inicjacji - $\Theta(n+m)$

Jedna z metod stosowanych w projektowaniu wydajnych algorytmów grafowych polega na ściśle określonym **przeszukiwaniu grafu** w celu zdobycia informacji o grafie, niezbędnej do rozwiązania zadanego problemu.

Przeszukiwanie grafów to wędrówka po jego wierzchołkach i krawędziach.

Dopóki nie powiemy inaczej będziemy rozważali grafy nieskierowane.

Niech G = (V,E) będzie grafem nieskierowanym o n wierzchołkach i m krawędziach.

Bez straty ogólności zakładamy, że V = {1, 2, …, n}.

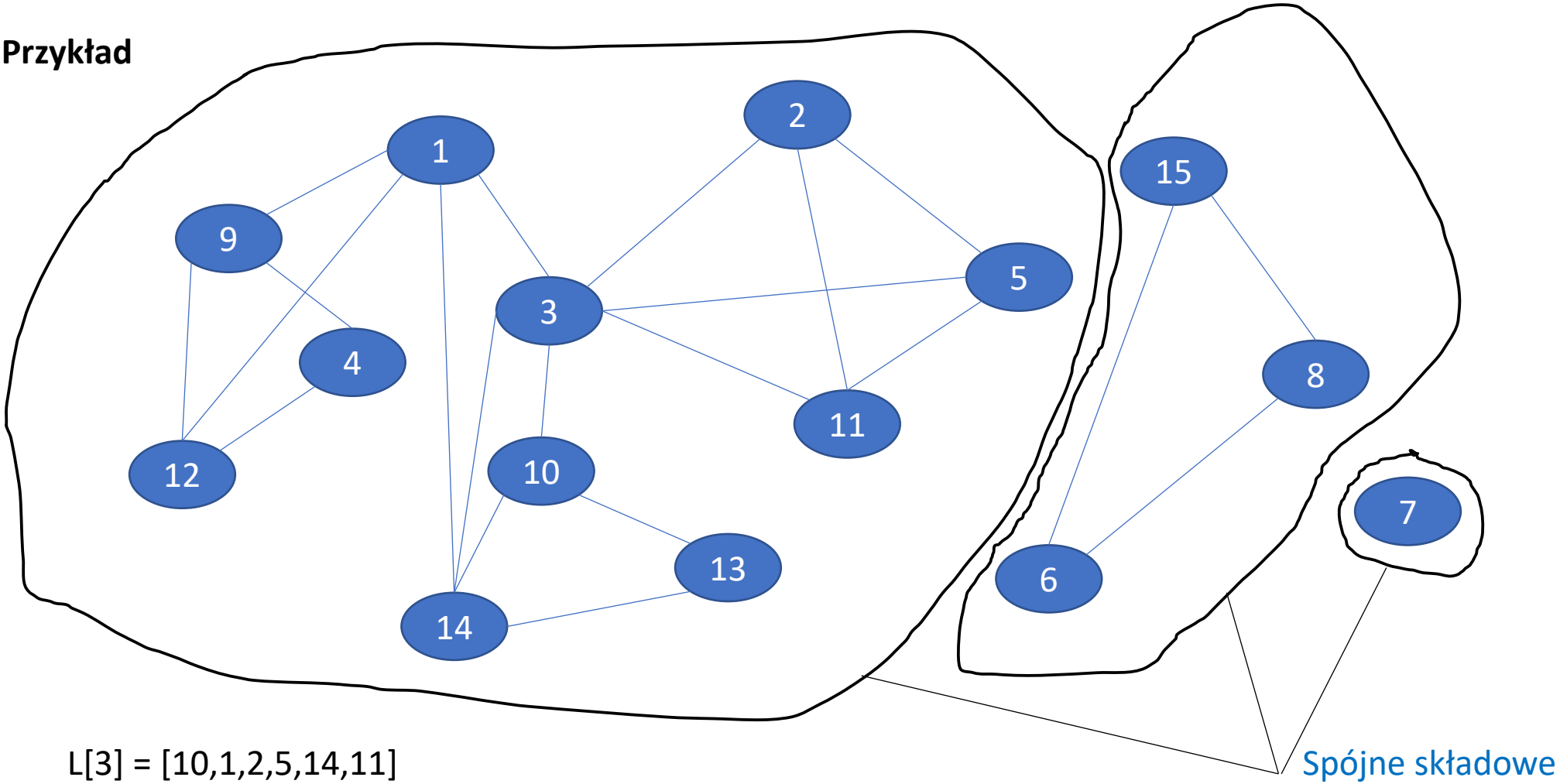
Graf G reprezentujemy przez listy sąsiedztwa L[1..n], gdzie L[u] to lista sąsiadów wierzchołka u.

Jeśli v jest na liście L[u], to Prev(v), Next(v) oznaczają odpowiednio poprzednika i następnika v na tej liście.

Rozmiar reprezentacji: rozmiar tablicy plus suma długości list sąsiedztwa = n + 2m.

Wiadomo, że $0 \leq m \leq n(n-1)/2$.

Przykład



L[3] = [10,1,2,5,14,11]
Deg(u) – stopień wierzchołka u, czyli liczba sąsiadów u w grafie
Deg(3) = 6
Deg(7) = 0, o wierzchołku 7 mówimy, że jest wierzchołkiem izolowanym

Abstrakcyjne przeszukiwanie grafu

visited[1..n] – tablica logiczna informująca, czy wierzchołek był już widziany podczas przeszukiwania

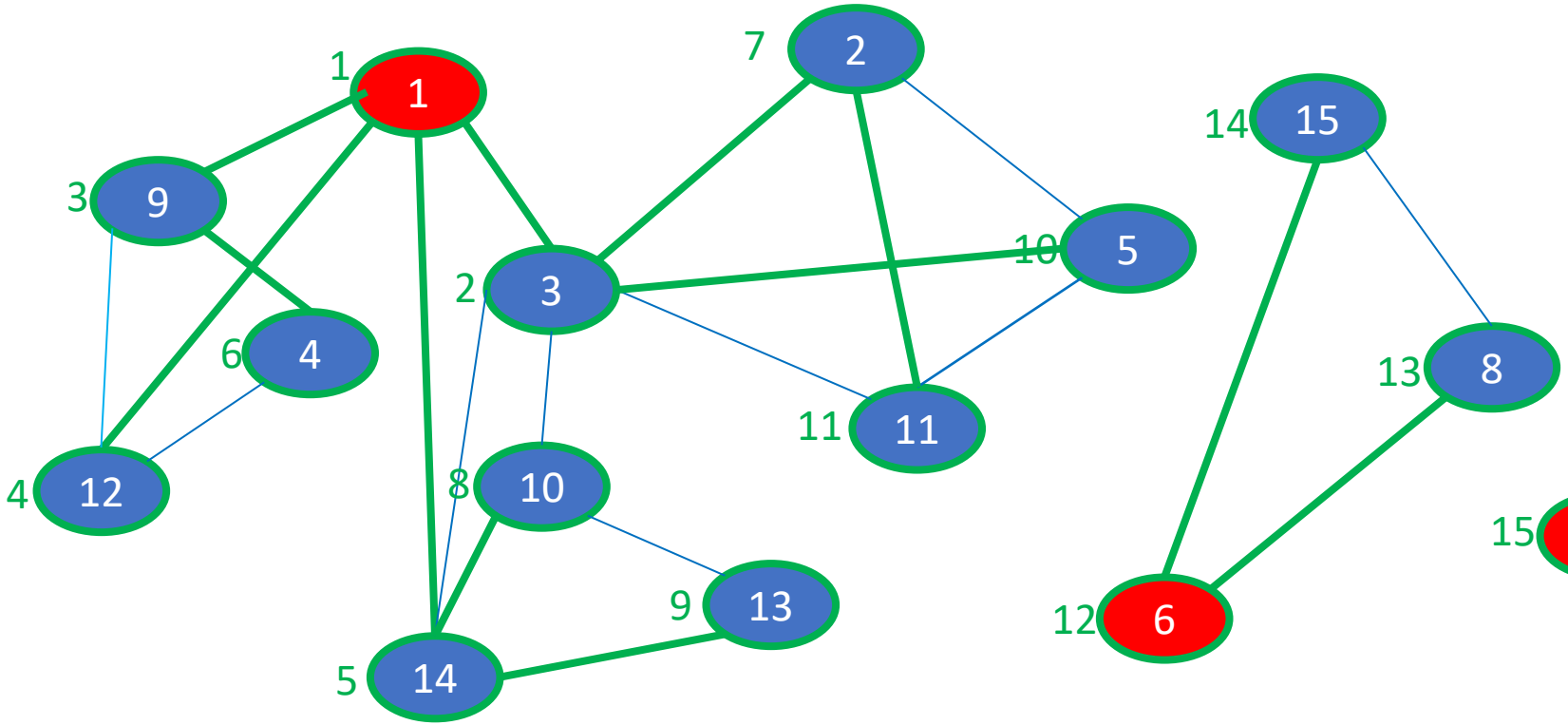
```
Search(G)::
begin
   S := ∅;  {zbiór wierzchołków, które zostały już zobaczone, ale nie obejrzeliśmy jeszcze w całości ich sąsiedztw}
   for u ∈ [1, …,n] do begin
        visited[u] := FALSE;
        Curent[u] := L[u]; {przeglądamy po kolei sąsiadów u, Current[u] to pierwszy nie obejrzany wierzchołek na liście L[u]}
   end;
   for u ∈ [1, …,n] do
   begin
      if Not visited[u] then
      begin
          visited[u] := TRUE; S := S ∪ {u};
          repeat
            v := (dowolny) wierzchołek z S;
            if Current[v] = NULL then S := S \ {v}; {przeszliśmy po wszystkich krawędziach wychodzących z v}
            else
            begin
              w := Current[v]; Current[v] := Next(Current[v]);
              if Not visited[w] then begin visited[w] := TRUE; S := S ∪ {w} end;
          until S = ∅
      end
end;
```

Jeśli operacje na zbiorze S są wykonywane w czasie stałym, to czas przeszukiwania wynosi O(n+m).

Przykład

Zielona numeracja to numeracja w kolejności odwiedzania wierzchołków po raz pierwszy – **numeracja przeszukiwania**. Zielona krawędź wskazuje pierwsze odwiedziny wierzchołka – zawsze prowadzi od wierzchołka odwiedzonego wcześniej (z mniejszym numerem).

Dla uproszczenia w przykładach przyjmujemy, że listy sąsiedztwa są uporządkowane rosnąco.



Graf rozpięty na zielonych krawędziach jest lasem. Drzewa w lesie można traktować jak drzewa ukorzenione, z korzeniem w najmniejszym wierzchołku. W takim drzewie zielone krawędzie prowadzą zawsze od rodzica do dziecka.

Spójne składowe w czasie O(n+m)

Dane

G=(V,E) – graf n-wierzchołkowy

Wynik

funkcja C: V →V taka, że C[u] = C[v] wtedy i tylko wtedy, gdy u i v są w tej samej spójne składowej

```
SearchCC(G)::
begin
   S := ∅;
   for u ∈ [1, …,n] do begin C[u] := 0; Current[u] := L[u] end;
   {C[u]= 0 oznacza, że wierzchołek nie został jeszcze odwiedzony}

   for u ∈ [1, …,n] do
   begin
      if C[u] = 0 then
      begin
         C[u] := u; S := S ∪ {u};
         repeat                                  {Identyfikatorem spójnej składowej jest
           v := (dowolny) wierzchołek z S;        najmniejszy spośród jej wierzchołków.}
           if Current[v] = NULL then S := S \ {v}; {przeszliśmy po wszystkich krawędziach wychodzących z v}
           else
           begin
             w := Current[v]; Current[v] := Next(Current[v]);
             if C[w] = 0 then begin C[w] := u; S := S ∪ {w} end
           end
         until S = ∅
      end
end;
```

Najkrótsze ścieżki w czasie O(n+m)

Dane
G = (V,E) – graf spójny, n = |V|
s – wyróżniony wierzchołek w G
Wynik
D[1..n] – tablica, w której D[u] długość najkrótszej ścieżki z wierzchołka s do wierzchołka u mierzona liczbą krawędzi
Metoda
przeszukiwanie grafu, w którym zbiór S implementujemy jako kolejkę FIFO (First In First Out)
jest to tzw. **przeszukiwanie wszerz** (ang. **BFS – Breadth First Search**)

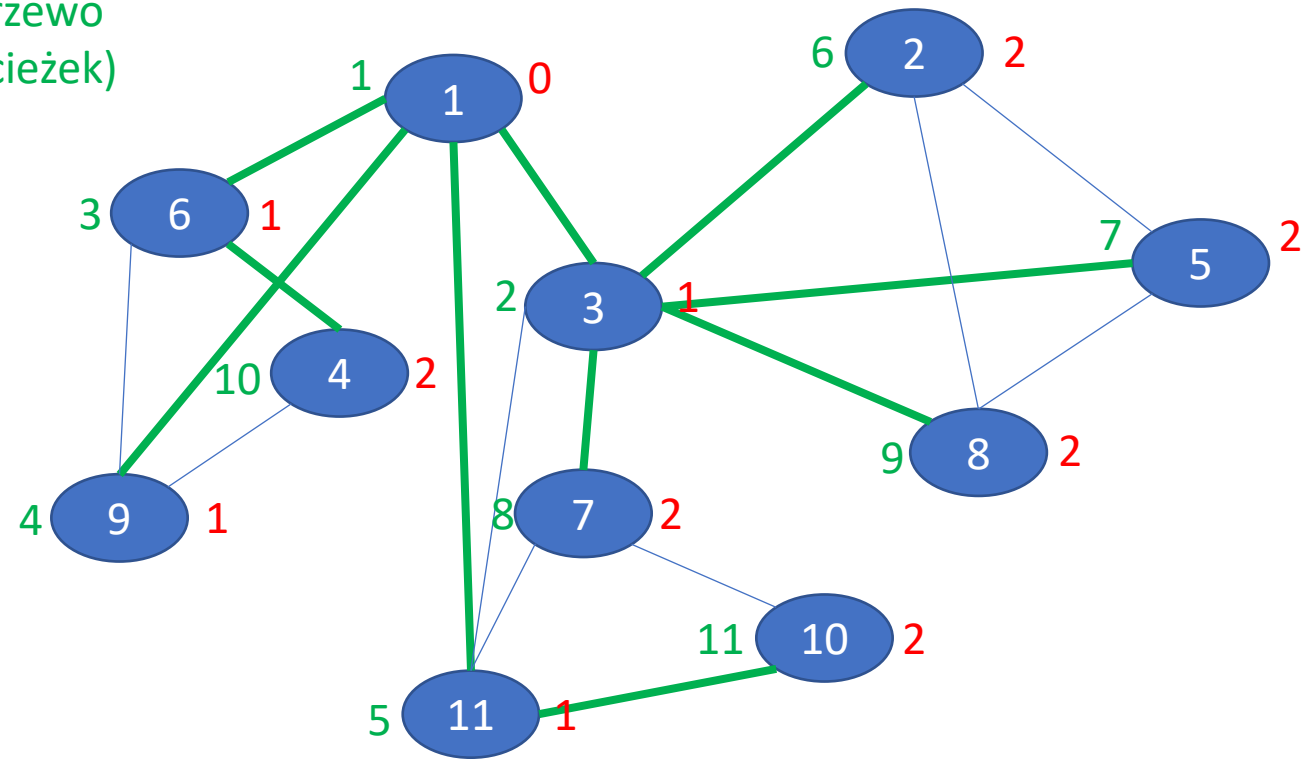
```
ShortestPathsBFS(G)::
begin
   S := {s};
   for u ∈ [1, …,n] do D[u] := -1; D[s] := 0; {D[u] = -1 oznacza, że wierzchołek u nie był jeszcze odwiedzony}
   while S ≠ ∅ do
   begin
      v := Front(S); Pop(S); w := Front(L[v]);
      while w ≠ NULL do
      begin
         if (D[w] = -1) then begin D[w] := D[v] + 1; Inject(S,w) end
         w := Next(w) {następny wierzchołek na liście L[v]}
      end
   end
end;
```

Przykład
graf spójny, listy sąsiedztwa uporządkowane rosnąco
s = 1

numeracja w porządku BFS – kolor zielony

odległości od s – kolor czerwony

drzewo rozpięte na zielonych krawędziach – drzewo przeszukiwania wszerz (drzewo najkrótszych ścieżek)



Przeszukiwanie grafu w głąb (ang. Depth First Search – DFS)

W tym przypadku zbiór S implementujemy jako stos.

Stos pojawia się niejawnie, ponieważ przeszukiwanie grafu dokonujemy z pomocą rekurencyjnej procedury DFS(u) – procedurę wywołujmy, gdy u jest nowo odkrytym wierzchołkiem, kolejnym, któremu należy nadać numer.

Podczas przeszukiwania wierzchołki grafu będziemy numerowali liczbami naturalnymi 1, 2, …
w kolejności pierwszych odwiedzin – numeracja dfs zapisywana jest w tablicy dfs_nr[1..n].

Wierzchołek, któremu zostanie nadany numer uważa się za zobaczony (odkryty).

Tablicę dfs_nr inicjujemy zerami – zero oznacza, że wierzchołek nie był jeszcze widziany.

W przeszukiwaniu użyjemy zmiennej globalnej last_nr, której wartością jest ostatni nadany numer.

Inicjalnie last_nr = 0.

```
DFS(u)::
begin
    last_nr := last_nr + 1;
    dfs_nr[u] := last_nr;
    v := Front(L[u]);
    while v ≠ NULL do begin
        if dfs_nr[v] = 0 then {wierzchołek nie był jeszcze widziany}
            DVS(v);
        v := Next(v);
    end
end;
```

Przykład adaptacji przeszukiwania w głąb do rozwiązania problemu spójnych składowych

cc_id – zmienna globalna, identyfikator aktualnie odwiedzanej spójnej składowej

C[1..n] – tablica identyfikatorów składowych, do których należą wierzchołki, C[u] = 0 oznacza, że u nie był odwiedzony

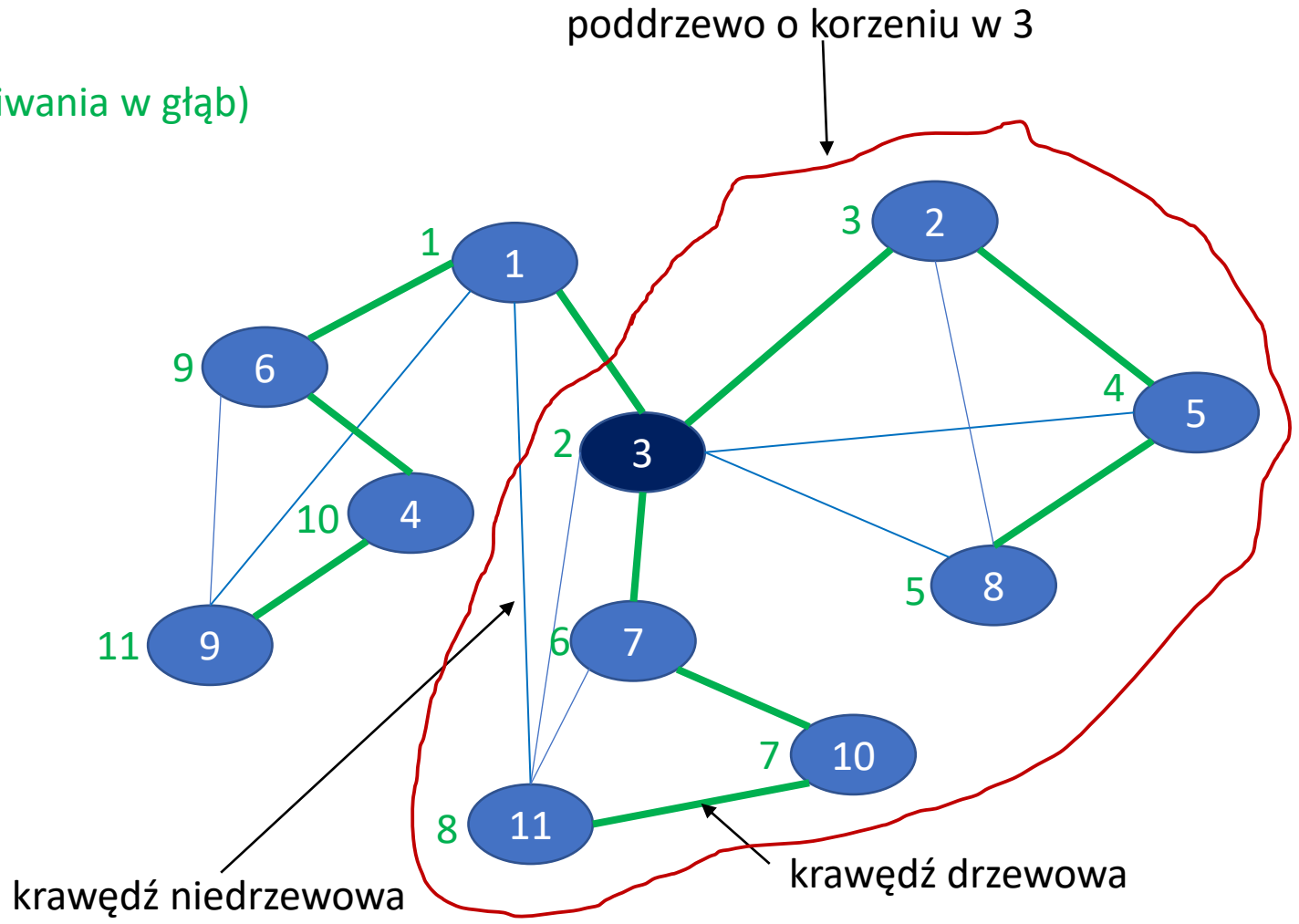
```
DFS_CC(u)::
begin
    C[u] := cc_id; v := Front(L[u]);
    while v ≠ NULL do begin
        if C[v] = 0 then DFS_CC(v);
        v :- Next(v)
    end
end;
```

```
Inicjacja
  for u ∈[1, …,n] do C[u] := 0

  for u ∈[1, …,n] do
  if C[u] = 0 then begin cc_id := u; DFS_CC(u) end;
```

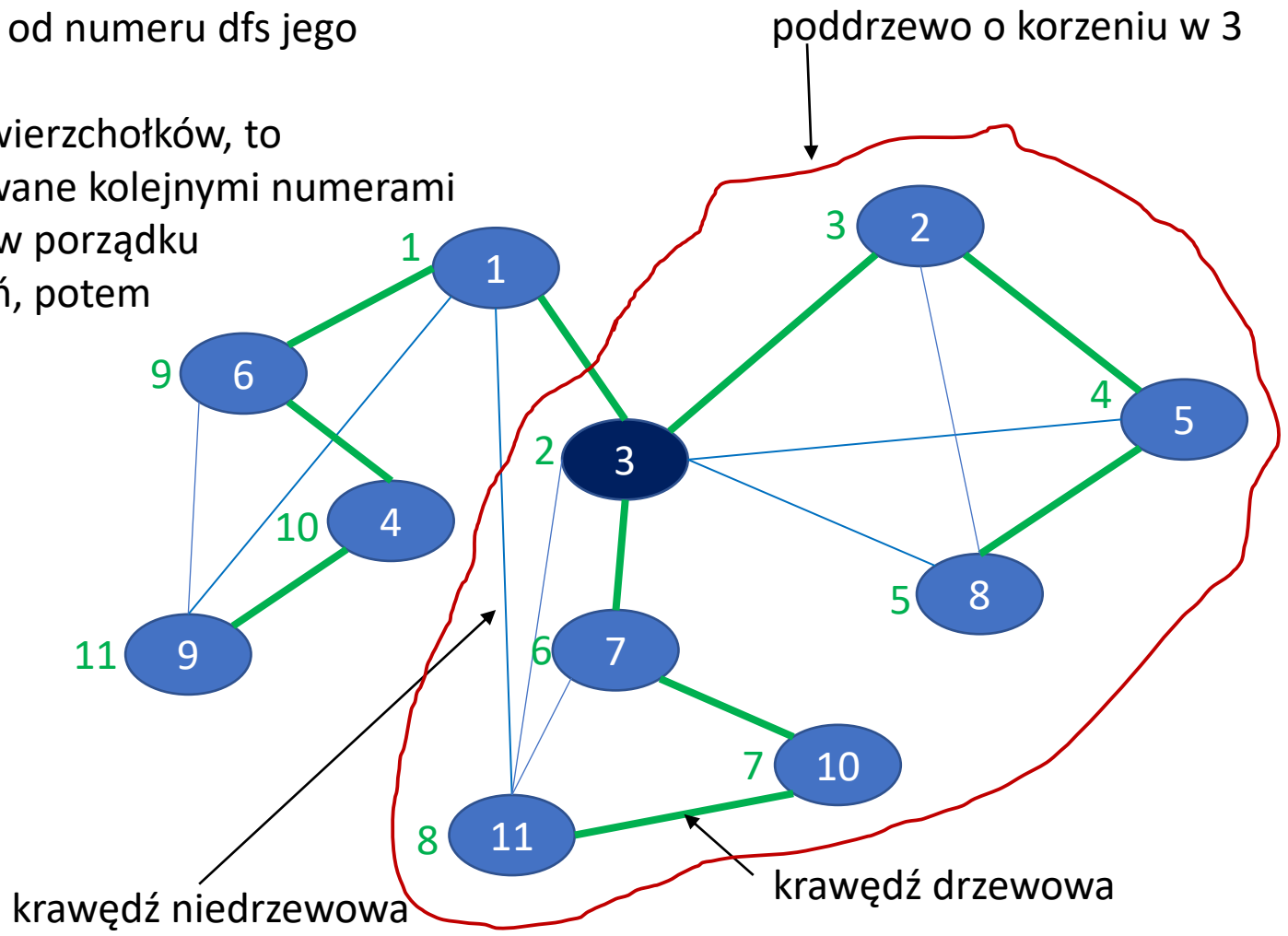
Przykład przeszukiwania w głąb

- listy sąsiedztwa uporządkowane rosnąco (na potrzeby przykładu)
- pierwsze wywołanie DFS(1)
- numeracja DFS – kolor zielony
- drzewo zielone – drzewo DFS (przeszukiwania w głąb)
- wierzchołek 1 – korzeń drzewa



Własności przeszukiwania w głąb

- krawędź niedrzewowa łączy zawsze potomka z przodkiem w w drzewie przeszukiwania w głąb
- numer dfs wierzchołka jest zawsze większy od numeru dfs jego właściwego przodka
- jeśli poddrzewo o korzeniu v zawiera d[v] wierzchołków, to wierzchołki tego poddrzewa są ponumerowane kolejnymi numerami dfs_nr[v], dfs_nr[v]+1, …, dfs_nr[v]+d[v]-1 w porządku prefiksowym („preorder” – najpierw korzeń, potem jego poddrzewa)



Dwuspójne składowe

Wierzchołek v w grafie G nazywamy **rozdzielającym** (**punktem artykulacji**) wtedy i tylko wtedy, gdy jego usunięcie z G (wraz z incydentnymi z nim krawędziami) zwiększa liczbę spójnych składowych w G.

Podobnie, **mostem** w grafie G nazywamy krawędź, której usunięcie zwiększa liczbę spójnych składowych grafu.

Powiem, że spójny graf G jest grafem **dwuspójnym wierzchołkowo** (krawędziowo) wtedy i tylko wtedy, gdy nie zawiera wierzchołków rozdzielających (mostów).

Uwaga: jeśli nie powiemy inaczej, mówiąc graf dwuspójny będziemy mieli zawsze na myśli graf dwuspójny wierzchołkowo.

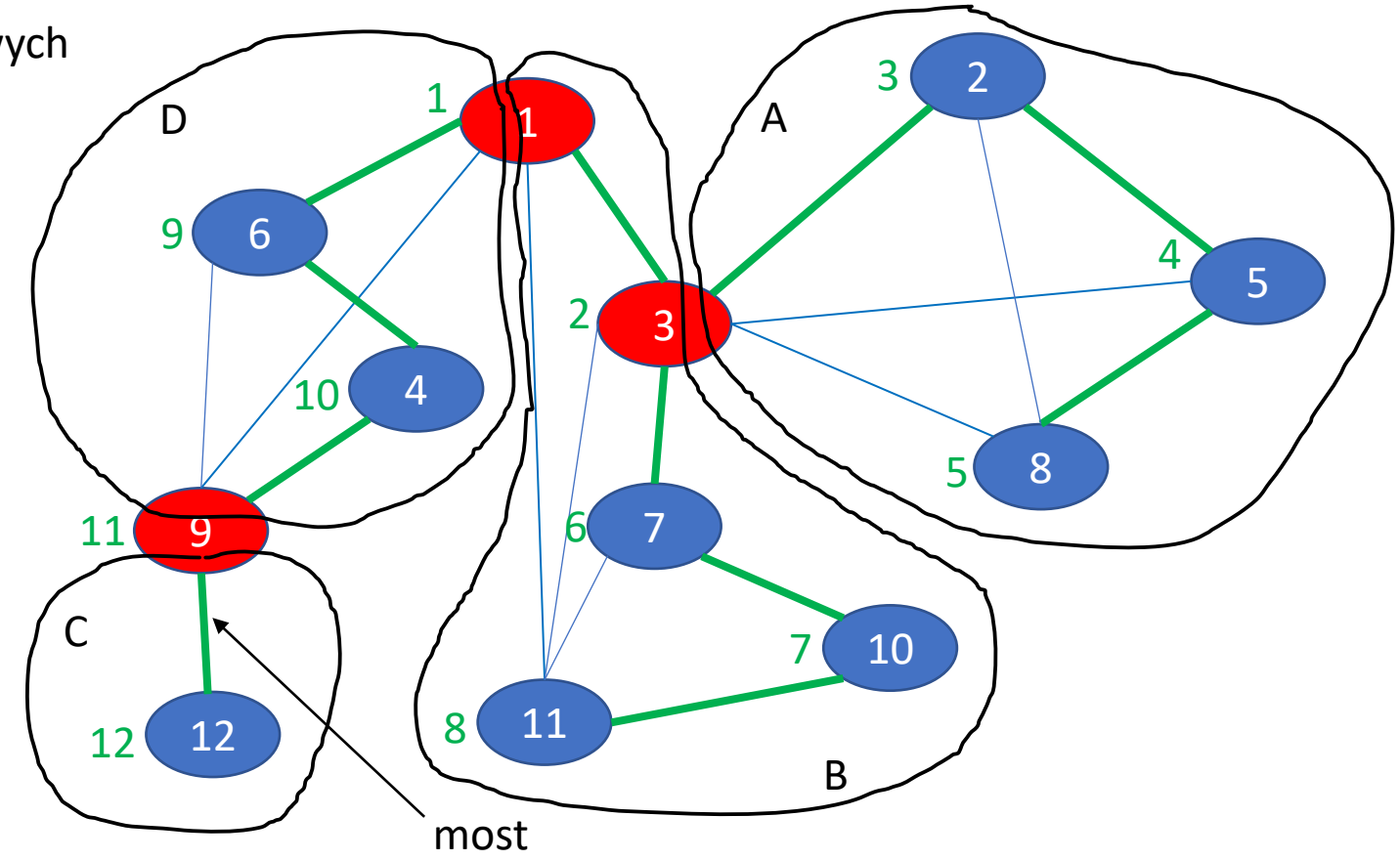
Dwuspójną składową grafu G nazywamy każdy jego maksymalny podgraf (z maksymalną możliwą liczbą wierzchołków i krawędzi).

A, B, C, D – cztery dwuspójne składowe

wierzchołek rozdzielający należy do
co najmniej dwóch dwuspójnych składowych

każda krawędź należy do dokładnie
Jednej dwuspójnej składowej

wierzchołek rozdzielający -



Problem dwuspójnych składowych

Dane
G = (V,E) – graf spójny zadany przez listy sąsiedztwa
Wynik
podział zbioru krawędzi na maksymalne podzbiory odpowiadające dwuspójnym składowym

Łatwiejszy problem – dwuspójność grafu

Dane
G = (V,E) – graf spójny zadany przez listy sąsiedztwa
Wynik
Informacja, czy graf składa się z 1, czy z większej liczby dwuspójnych składowych

Uwaga: problem dwuspójności łatwo rozwiązać w czasie O(nm), testując spójność grafu po usunięciu każdego wierzchołka

CEL – czas O(n+m) !!!

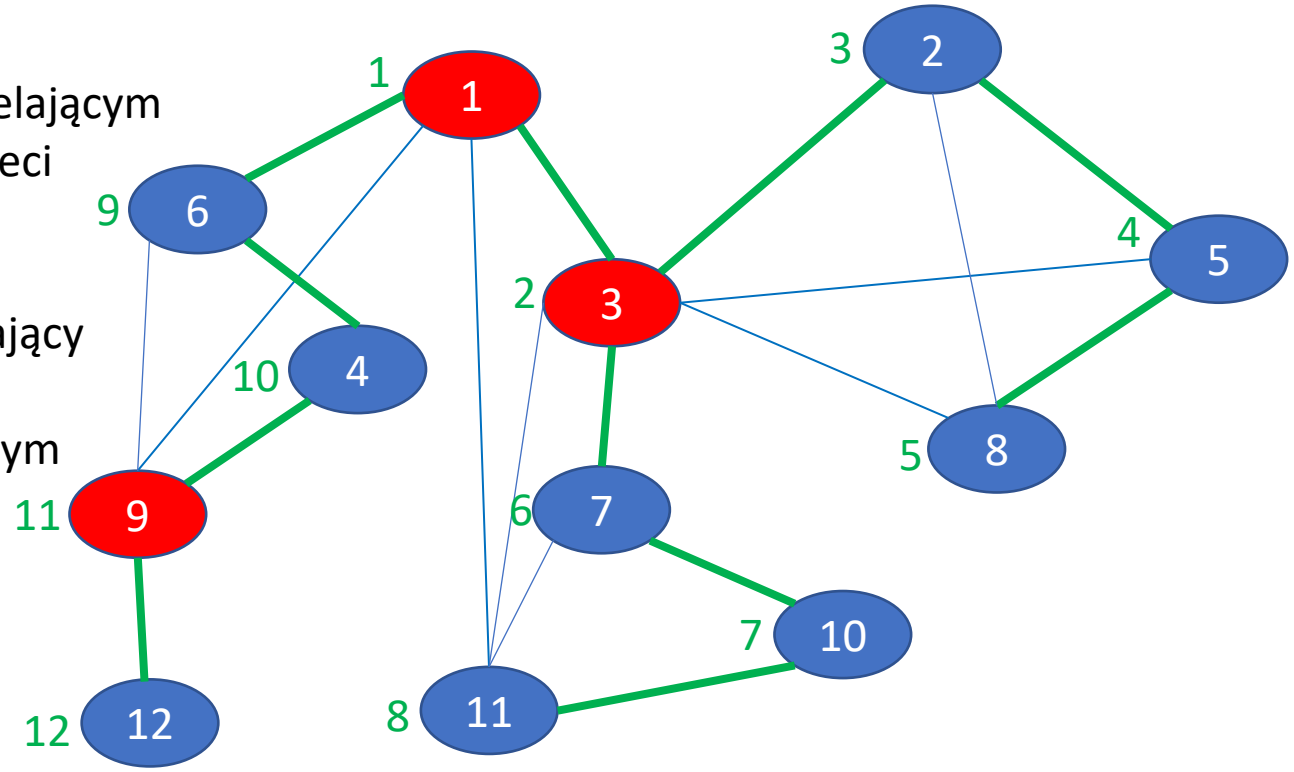
Dwuspójność grafu w czasie O(n+m)

Załóżmy, że przeszukaliśmy graf w głąb i znane jest drzewo przeszukiwania.

Spostrzeżenia

1. korzeń drzewa DFS jest wierzchołkiem rozdzielającym wtedy i tylko, gdy ma co najmniej dwójkę dzieci w tym drzewie

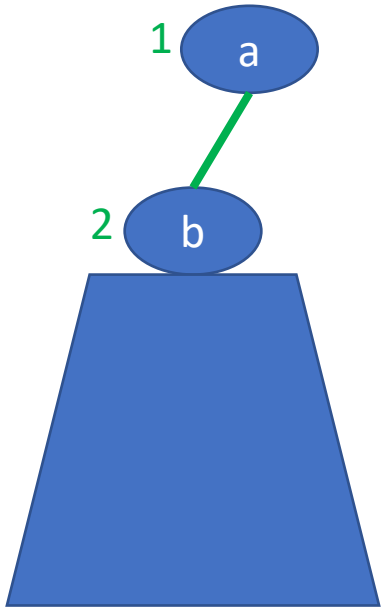
2. wierzchołek v różny od korzenia jest rozdzielający wtedy i tylko wtedy, gdy posiada dziecko u takie, że każda krawędź niedrzewowa o jednym z końców w poddrzewie o korzeniu u ma drugi koniec w poddrzewie o korzeniu w wierzchołku v



1. korzeń drzewa DFS jest wierzchołkiem rozdzielającym wtedy i tylko, gdy ma co najmniej dwójkę dzieci w tym drzewie

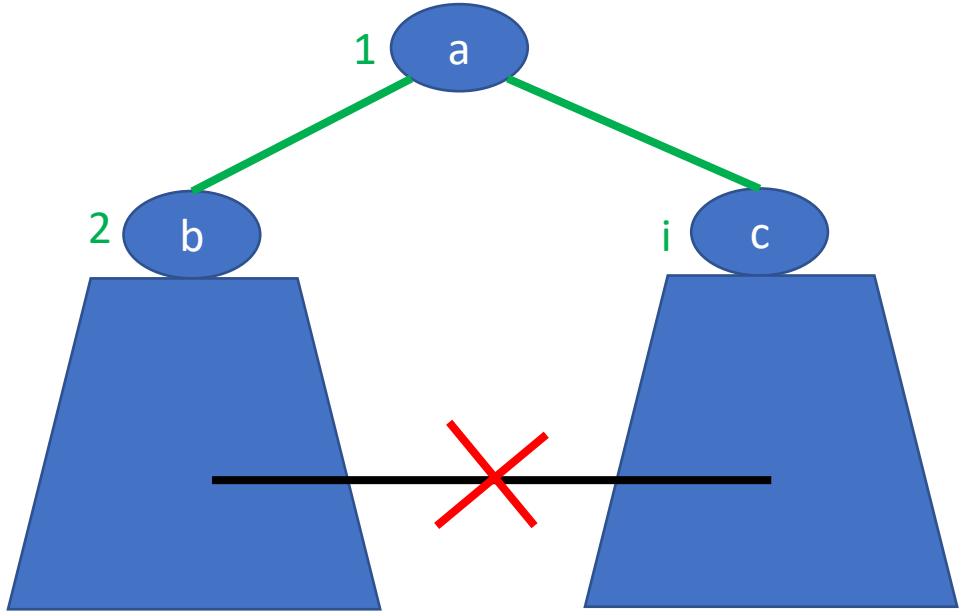
$\Rightarrow$

- korzeń ma tylko jedno dziecko w drzewie wyszukiwania w głąb
- wszystkie wierzchołki grafu poza a są w poddrzewie o korzeniu w b
- zatem usunięcie a nie rozspójnia grafu



$\Leftarrow$

- korzeń drzewa ma co najmniej dwójkę dzieci w drzewie wyszukiwania w głąb
- krawędzie nie drzewowe łączą tylko potomków z przodkami (nie ma krawędzi poprzecznych)
- z b do c można przejść tylko przez a, zatem usunięcie a rozspójnia graf



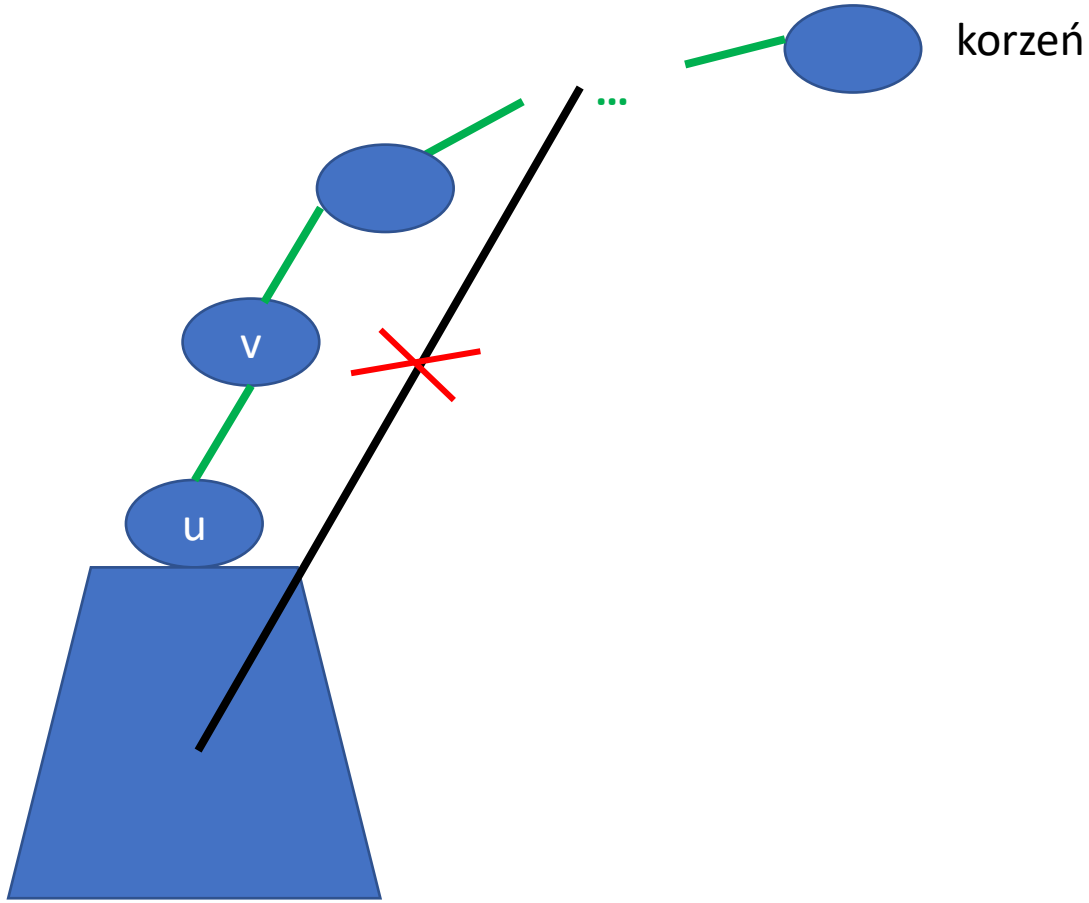
2. wierzchołek v różny od korzenia jest rozdzielający wtedy i tylko wtedy, gdy posiada dziecko u takie, że każda krawędź niedrzewowa o jednym z końców w poddrzewie o korzeniu u ma drugi koniec w poddrzewie o korzeniu w wierzchołku v

$\Rightarrow$

- jeżeli wierzchołek v jest rozdzielający, to posiada dziecko u takie, że każda ścieżka z u do korzenia drzewa prowadzi przez v
- ponieważ każda krawędź niedrzewowa łączy potomka z przodkiem, to jeśli jeden z końców jest w poddrzewie u, drugi koniec może „sięgać co najwyżej do" v

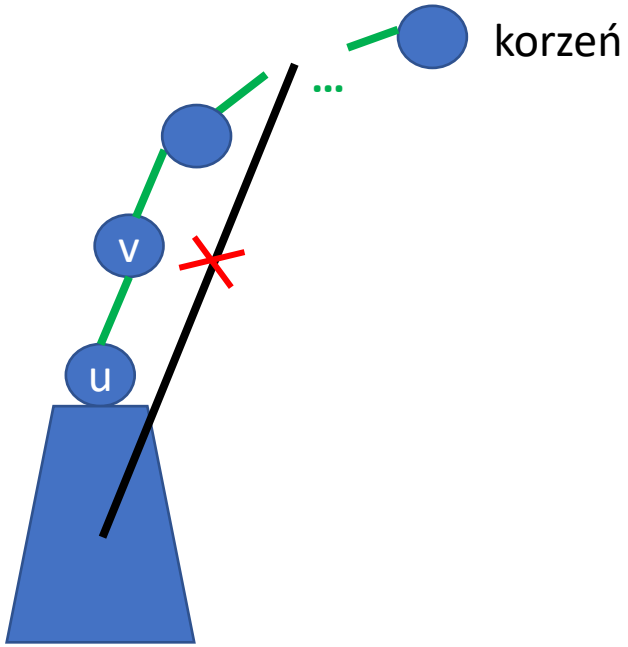
$\Leftarrow$

- ponieważ każda krawędź niedrzewowa o jednym z końców w poddrzewie o korzeniu u prowadzi co najwyżej do v, to każda ścieżka z u do korzenia musi prowadzić przez v, zatem v jest rozdzielający



To czy korzeń drzewa ma co najmniej dwójkę dzieci łatwo zweryfikować.

W jaki sposób sprawdzić warunek 2?



Definiujemy
low[u] = Min({dfs_nr[u]} ∪ {dfs_nr[x]: istnieje krawędź niedrzewowa x—y taka, że y jest w poddrzewie o korzeniu u})

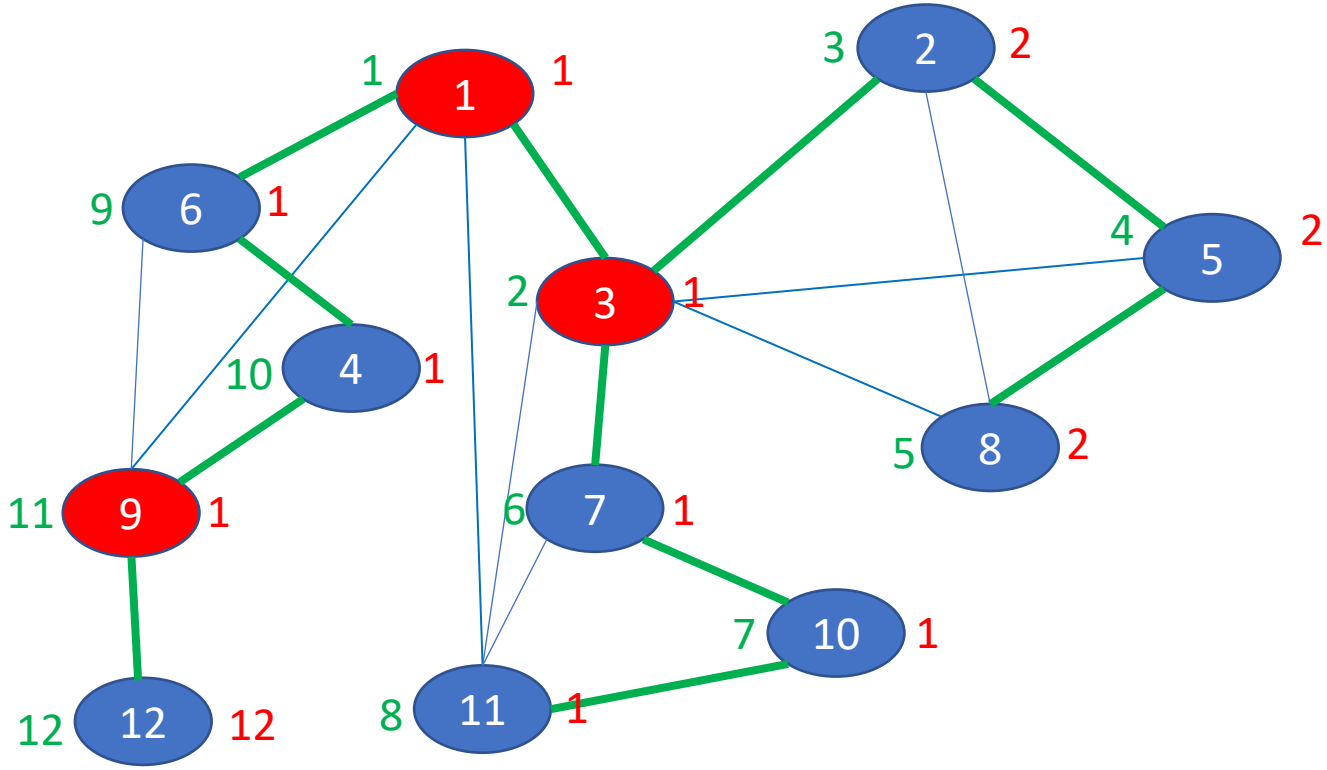
Innymi słowy low[u] mówi jak „wysoko” można uciec z poddrzewa o korzeniu u idąc w dół drzewa krawędziami drzewowymi i na końcu skacząc w górę po krawędzi niedrzewowej.

Warunek 2 na nowo
Wierzchołek v różny od korzenia jest rozdzielający wtedy i tylko wtedy, gdy posiada dziecko u takie, że low[u] ≥ dfs_nr[v].

Przydatna rekurencyjna definicja low:
low[u] = Min({dfs_nr[u]} ∪
{dfs_nr[x]: istnieje krawędź niedrzewowa x—u} ∪
{low[w]: w jest dzieckiem u w drzewie przeszukiwania w głąb})

Przykład

- na czerwono wartości low



Algorytm obliczania liczby dwuspójnych składowych w grafie spójnym

BC_nr – zmienna globalna, na której liczymy liczbę dwuspójnych składowych
Children – zmienna globalna, na której liczymy liczbę dzieci korzenia
ponadto zmienne globalne last_nr, dfs_nr[1..n], low[1..n]

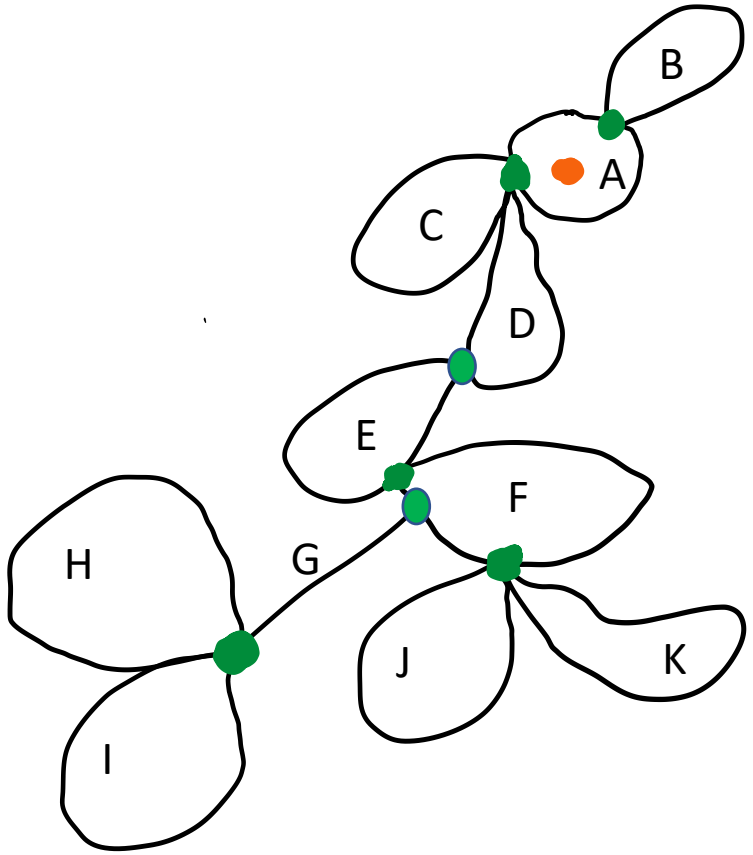
```
DFS_BicNr(v,f):: {f – rodzic v w drzewie przeszukiwania w głąb; dla korzenia f = 0}
begin
    last_nr := last_nr + 1; dfs_nr[v] := last_nr; low[v] := last_nr;
    u := Front(L[v]);
    while u ≠ NULL do begin
        if dfs_nr[u] = 0 then
        begin {wchodzimy do poddrzewa o korzeniu u}
            DFS_BicNr(u,v);
            if v = korzeń then Children := Children + 1
            else
              if low[u] ≥ dfs_nr[v] then BC_nr := BC_nr + 1;
              else low[v] := Min(low[v],low[u])
        end
        else
            if u ≠ f then low[v] := Min(low[v],dfs_nr[u]);
       u := Next(u)
    end
end;
```

Główny program:

```
begin
  BC_nr := 0; last_nr := 0; korzeń := 1;
  Children := 0;
  for v ∈ [1, …,n] do dfs_nr[v] := 0;
  DFS_BicNr(1,0);
  BC_nr := BC_nr + Children
end;
```

Struktura dwuspójnych składowych

- przypomina drzewo
- czerwona kropa, to korzeń drzewa przeszukiwania w głąb
- zielone kropy to wierzchołki rozdzielające
- składowe B, C, H, I, J, K to „liście”



Ważne spostrzeżenie:

- podczas przeszukiwania w głąb, jeśli wejdziemy do liścia-dwuspójnej składowej, to przed jego opuszczeniem „przejrzane” zostaną wszystkie krawędzie tej dwuspójnej składowej

- pierwszą krawędzią przeglądana jest zawsze krawędź drzewowa, którą wchodzimy z wierzchołka rozdzielającego w głąb tej składowej

- jeśli odkładamy krawędzie grafu na stos w kolejności ich przeglądania (krawędź niedrzewowa jest najpierw odkrywana w potomku jej drugiego końca), to po odkryciu wierzchołka rozdzielającego dla liścia w chwili wychodzenia z odpowiadającej jej dwuspójnej składowej, wszystkie krawędzie tej dwuspójnej składowej znajdują się na stosie, a najgłębiej jest położona krawędź drzewowa prowadząca w głąb składowej z wierzchołka rozdzielającego

Algorytm obliczania dwuspójnych składowych w grafie spójnym

Globalne: last_nr, dfs_nr[1..n], low[1..n], stos S (początkowo pusty)

```
DFS_BComp(v,f):: {f – rodzic v w drzewie przeszukiwania w głąb; dla korzenia f = 0}
begin
    last_nr := last_nr + 1; dfs_nr[v] := last_nr; low[v] := last_nr;
    u := Front(L[v]);
    while u ≠ NULL do begin
        if dfs_nr[u] = 0 then
        begin {wchodzimy do poddrzewa o korzeniu u}
            Push(S,v−u); DFS_BComp(u,v);
            if low[u] ≥ dfs_nr[v] then
            begin {kolejna dwuspójna składowa, v−u najgłębiej na stosie położona krawędź z tej składowej}
              output kolejna dwuspójna składowa;
              repeat x−y := Top(S); Pop(S); output x−y until x−y = v−u;
            end
            else low[v] := Min(low[v],low[u])
        end
        else
        begin
            if u ≠ f then
              if dfs_nr[u] < dfs_nr[v] then begin  low[v] := Min(low[v],dfs_nr[u]); Push(S, u−v) end
        end;
        u := Next(u)
    end
end;
```

Główny program:

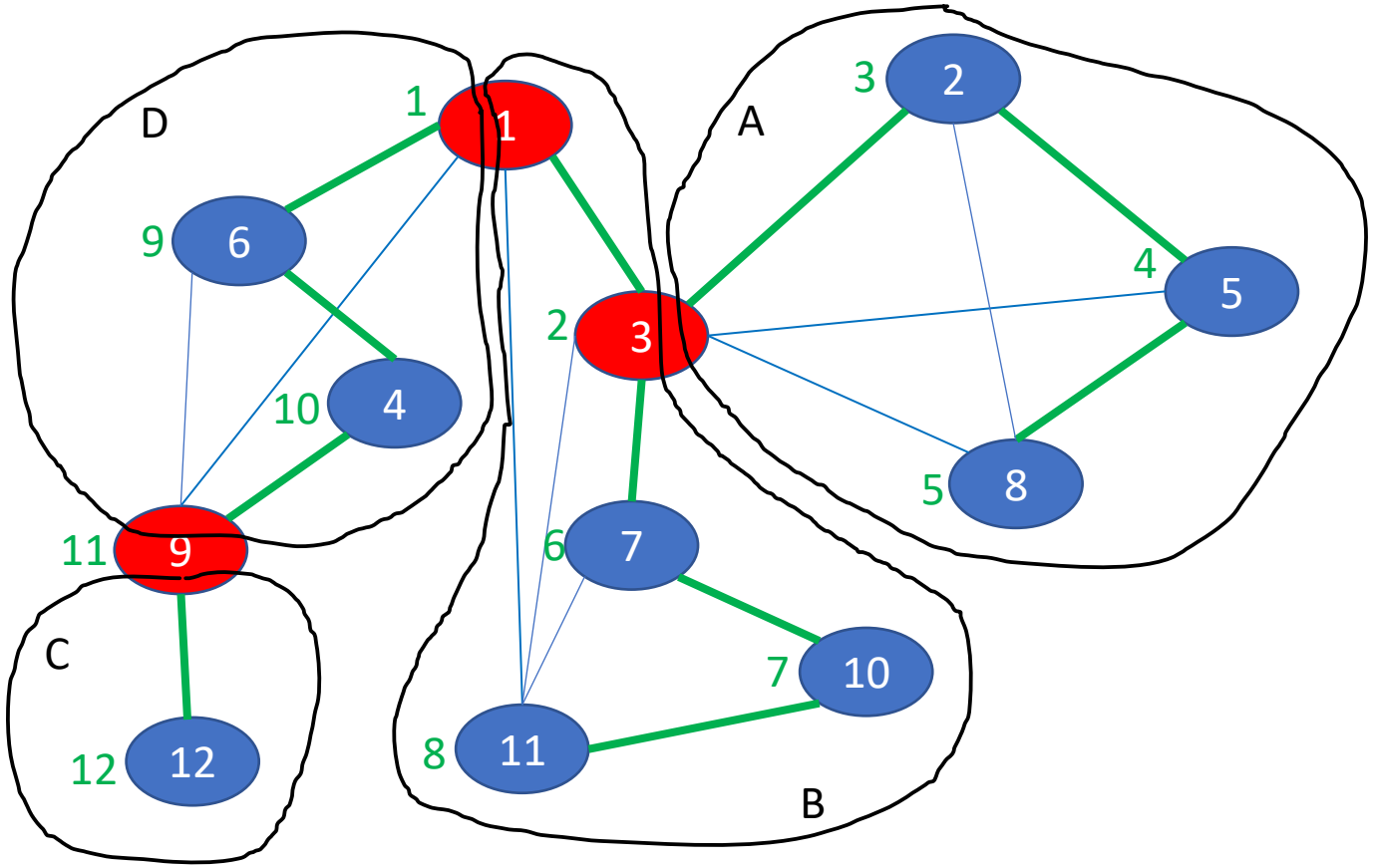
```
begin
  last_nr := 0; S := Θ;
  for v ∈[1, …,n] do dfs_nr[v] := 0;
  DFS_BComp(1,0);
end;
```

Złożoność: O(m)

Przykład

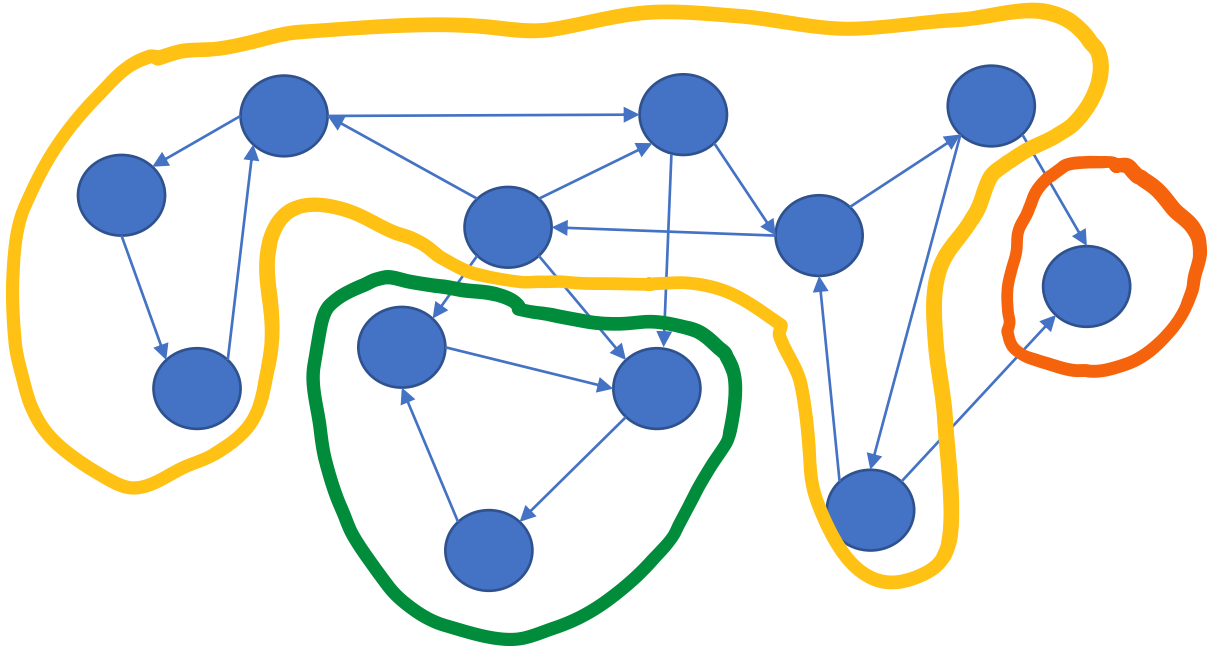
Oto zawartości stosów po wykryciu dwuspójnej składowej a przed wypisaniem jej krawędzi (listy sąsiedztwa uporządkowane rosnąco):

A	B	C	D
8-3	11-7		
8-2	11-3	9-12	
5-8	11-1	9-6	9-6
5-3	10-11	9-1	9-1
2-5	7-10	4-9	4-9
3-2	3-7	6-4	6-4
1-3	1-3	1-6	1-6



Problem silnie spójnych składowych

Niech G = (V,E) będzie grafem skierowanym beż pętli (krawędzi postaci v→v). Powiemy, że graf G jest **silnie spójny** wtedy i tylko wtedy, gdy dla każdej uporządkowanej pary wierzchołków (u,v) istnieje w G ścieżka skierowana z u do v. **Silnie spójną** składową grafu G nazywamy każdy jego maksymalny podgraf silnie spójny. Graf skierowany jest **słabo spójny**, gdy jest spójny po zamienieniu każdej krawędzi skierowanej na krawędź nieskierowaną.



Trzy silnie spójne składowe.

Reprezentacja grafu skierowanego przez listy sąsiedztwa

L+, L-: tablice list

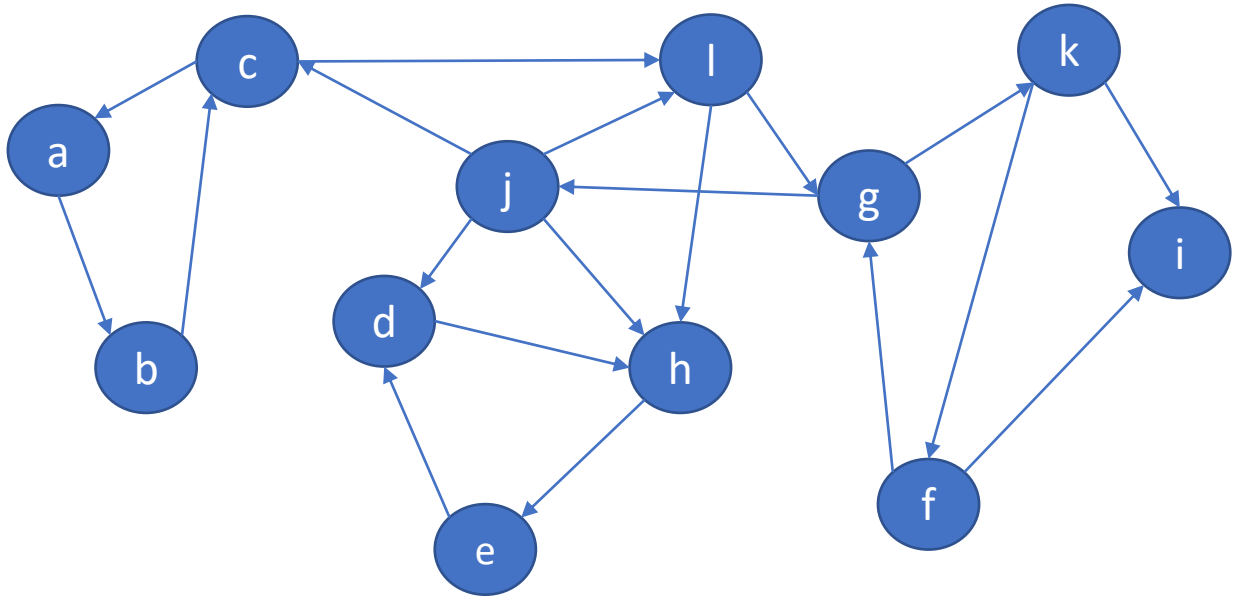
L+[v]: lista wierzchołków, do których prowadzą krawędzie z wierzchołka v

L-[v]: lista wierzchołków, od których prowadzą krawędzie do wierzchołka v

Rozmiar reprezentacji – O(n+m)

Przykład

L+[j]: d, h, l, c
L-[j]: g



Problem I – testowanie silnej spójności

Dane

G=(V,E) – słabo spójny graf skierowany

Wynik

odpowiedź na pytanie, czy G jest silnie spójny

Algorytm

1. weź dowolny wierzchołek s ∈ V
2. wykonaj przeszukiwanie w głąb (**w przód**) z wierzchołka s przechodząc po krawędziach zgodnie z ich orientacją (wykorzystując listy L+) i kolorując wszystkie odwiedzone wierzchołki na biało
3. wykonaj przeszukiwanie w głąb (**w tył**) z wierzchołka s przechodząc po krawędziach przeciwnie do ich orientacji (wykorzystując listy L-) i kolorując wszystkie odwiedzone wierzchołki na czarno
4. jeśli każdy wierzchołek został pokolorowany dwoma kolorami graf jest silnie spójny, w przeciwnym graf nie jest grafem silnie spójnym

Poprawność

Poprawność wynika z faktu, że graf jest silnie spójny wtedy i tylko wtedy, gdy z dowolnego, ustalonego wierzchołka s dojdziemy do każdego innego wierzchołka oraz z każdego wierzchołka dojdziemy do s, idąc po krawędziach zgodnie z ich orientacjami.

Złożoność

O(n+m) - dwa przeszukiwania w głąb

Problem II – silnie spójne składowe

Dane

G=(V={1, 2, …,n},E) – słabo spójny graf skierowany zadany przez
listy sąsiedztwa L+[1..n], L-[1..n]

Wynik

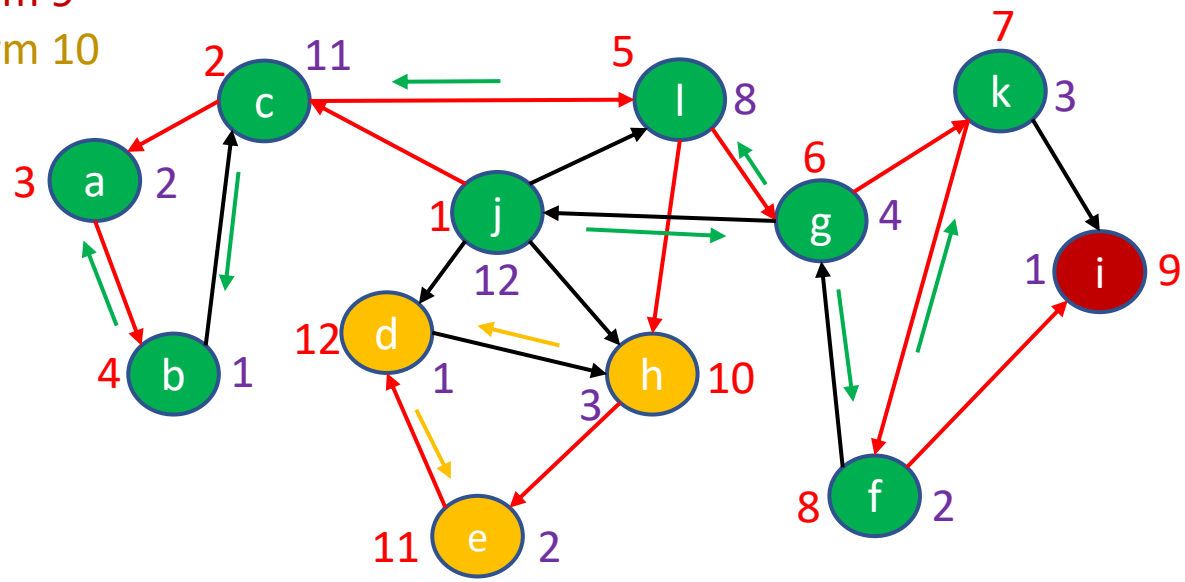
tablica s[1..n] – funkcja s: V → V taka, że s[u] = s[v] wtedy i tylko wtedy, gdy u i v są w tej samej silnie spójnej składowej grafu G

Algorytm

1. przeszukaj graf w przód metodą w głąb numerując wierzchołki w kolejności odwiedzania i obliczając dla każdego wierzchołka rozmiar poddrzewa w lesie przeszukiwania w głąb, o o korzeniu w tym wierzchołku
2. przeglądaj wierzchołki w kolejności odwiedzania w przód (punkt 1) i jeśli aktualnie oglądany wierzchołek v nie został jeszcze przypisany do żadnej silnie spójnej składowej (s[v] nie jest jeszcze określone), uruchom przeszukiwanie w tył (metoda w głąb) z wierzchołka v, oznaczając wszystkie wierzchołku osiągane z v i należące do poddrzewa w przód o korzeniu w tym wierzchołku, jako należące do silnie spójnej składowej o etykiecie v

Przykład działania algorytmu

numeracja czerwona – numeracja dfs w przód
(listy sąsiedztwa uporządkowane alfabetycznie)
krawędzie czerwone – krawędzie lasu przeszukiwania w przód
liczby fioletowe – rozmiary poddrzew
przeszukiwanie w tył z wierzchołka o numerze dfs równym 1
przeszukiwanie w tył z wierzchołka o numerze dfs równym 9
przeszukiwanie w tył z wierzchołka o numerze dfs równym 10



Poprawność

- jeśli w przeszukiwaniu w przód wierzchołek v jest odwiedzany jako pierwszy w swojej silnie spójnej składowej, to wszystkie wierzchołki z tej składowej znajdują się w poddrzewie drzewa przeszukiwania o korzeniu v

Złożoność

O(|V| + |E|)

Wykład opracowano między innymi na podstawie:

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Wprowadzenie do algorytmów, PWN 2012
- Lech Banachowski, Krzysztof Diks, Wojciech Rytter, Algorytmy i struktury danych, PWN 2018

<Mistrzostwa
w Algorytmice
i Programowaniu>

Projekt „Mistrzostwa w Algorytmice i Programowaniu – Uczniowie” jest finansowany ze środków pochodzących z „Programu Rozwoju Talentów Informatycznych na lata 2019-2029”

Dofinansowanie Projektu: 4.887.850,50 zł

Całkowita wartość Projektu: 5.460.850,50 zł