Przechodzenie grafów

Graf jest to para zbiorów
zbiór wierzchołków *V*
zbiór krawędzi E

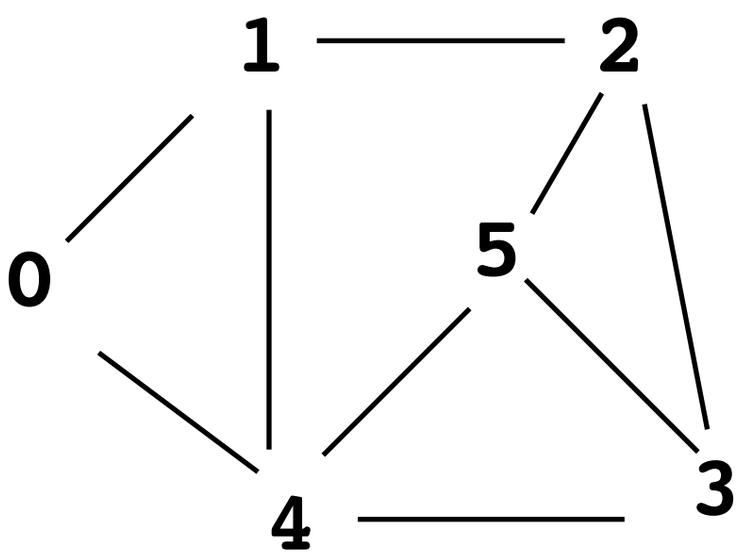
$G = (V, E)$

Krawędź, to
dla grafu nieskierowanego: dwuelementowy zbiór wierzchołków
dla grafu skierowanego: para wierzchołków

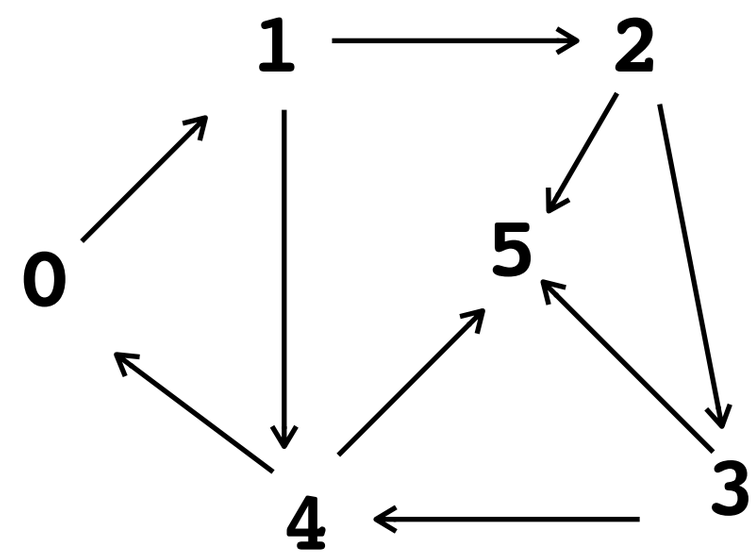
w jednym i w drugim przypadku krawędzie zapisujemy jako pary (v, w)
(matematyczny zapis dla grafu nieskierowanego powinien być $\{v, w\}$,
ale utarł się zapis (v, w) i w tej sytuacji)

Przykład: $G = (V, E)$

V = {0, 1, 2, 3, 4, 5}
E = {(0,1), (1,2), (1,4), (2,5), (2,3), (3,5)
(3,4), (4,5), (4,0)}



nieskierowany

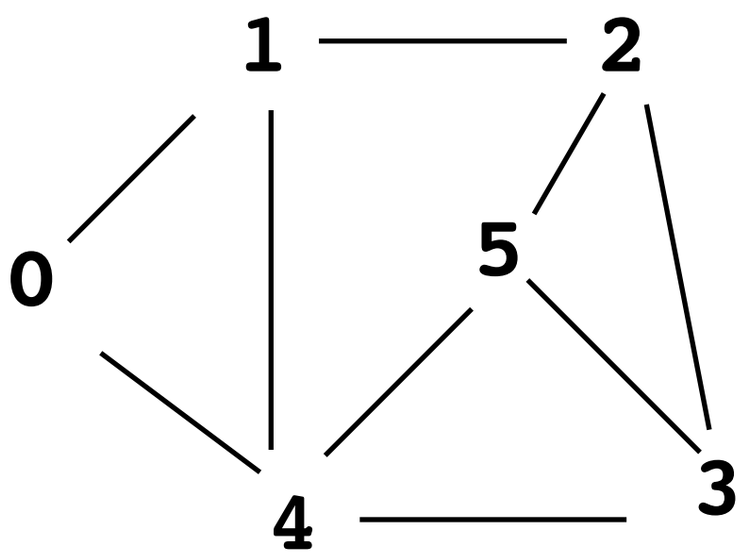


skierowany

reprezentowanie grafów w programach

- lista wierzchołków i krawędzi, jak w poprzednim przykładzie

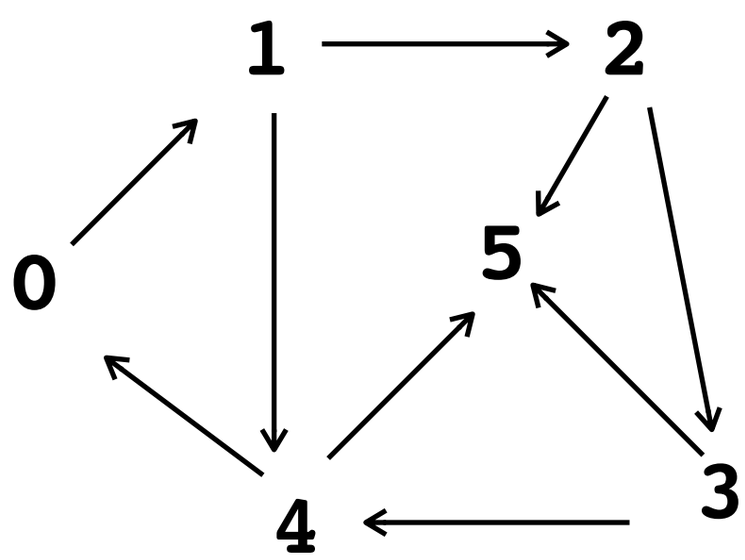
- tablica sąsiedztw (graf nieskierowany)



E:

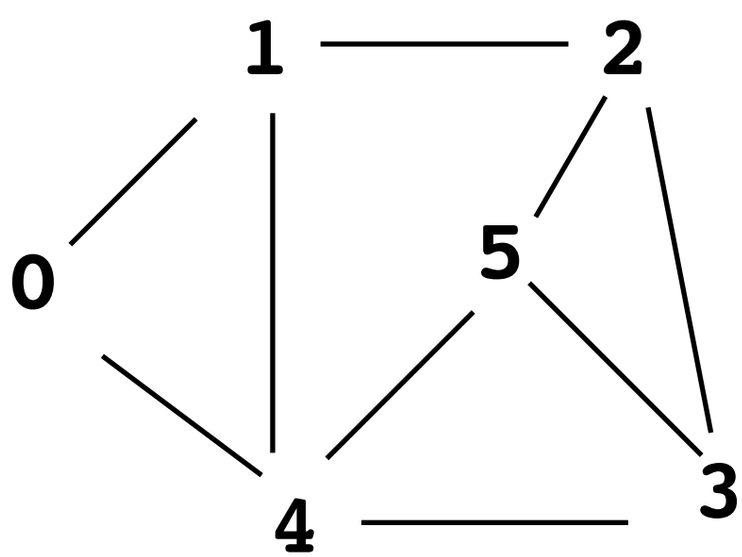
	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	1
3	0	0	1	0	1	1
4	1	1	0	1	0	1
5	0	0	1	1	1	0

- tablica sąsiedztw (graf skierowany)



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

- lista sąsiedztw (graf nieskierowany)



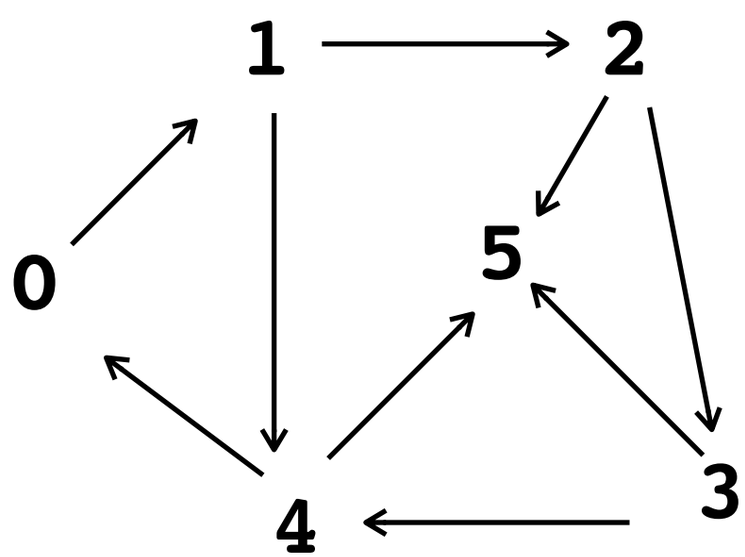
0	[1, 4]
1	[0, 2, 4]
2	[1, 5, 3]
3	[4, 5, 2]
4	[5, 1, 0]
5	[2, 3, 4]

Adj:

0	—>1--->4
1	—> 0—> 2 —> 4
2	—> 1 —> 5 —> 3
3	—> 4 —> 5 —> 2
4	—> 5 —> 1 —> 0
5	—> 2 —> 3 —> 4

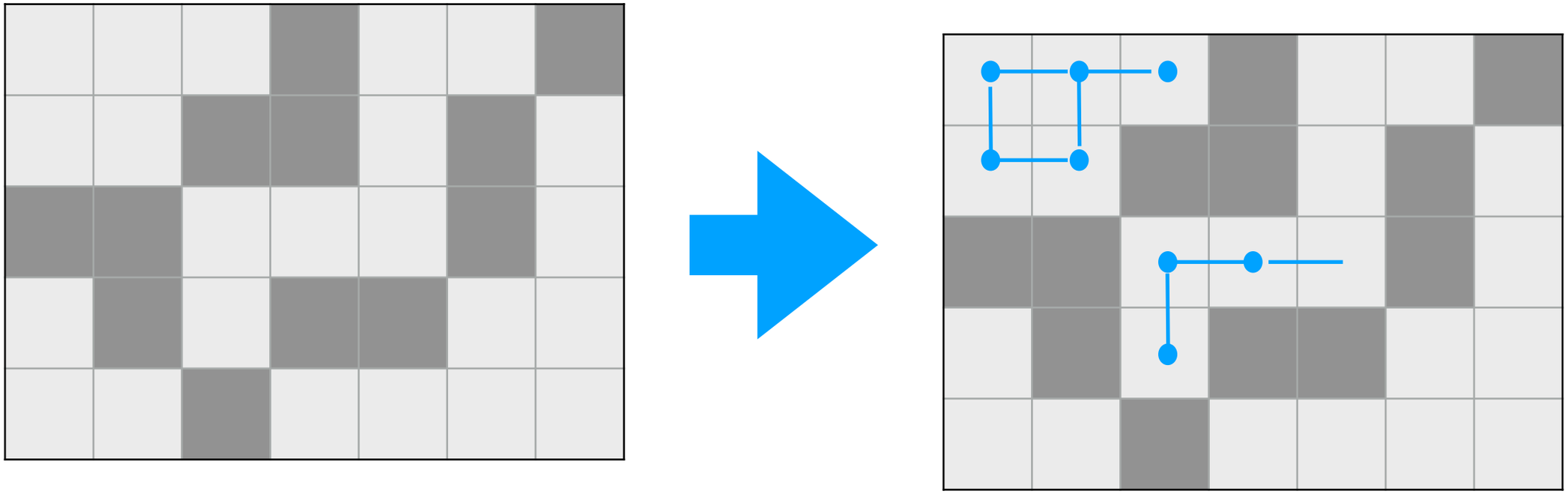
mogą to być listy dowiązaniowe lub tablicowe, ale o zmiennej długości

- lista sąsiedztw (graf skierowany)



0	—>1
1	—> 2 —> 4
2	—> 5 —> 3
3	—> 4 —> 5
4	—> 5 —> 0
5	—> NIL

Przykład zastosowania przeszukiwania grafów



- obejść wszystkie pola osiągalne z danego pola (ruchy: góra, dół, lewo, prawo)

- pokolorować rozłączne obszary na różne kolory

przechodzenie grafów w głąb
(depth first search)

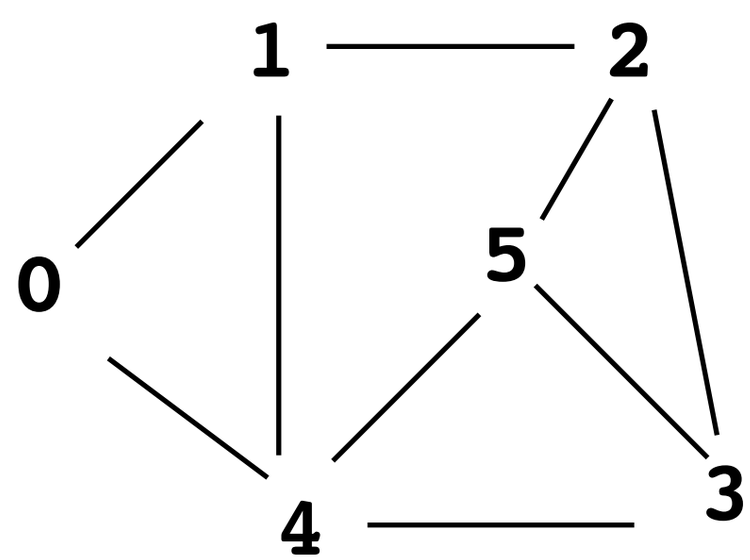
oznaczenia: niech v będzie wierzchołkiem
wtedy

`v.nr` oznacza numer wierzchołka
`v.Adj` to lista sąsiedztwa dla v
`v.visited` informacja boolowska, czy wierzchołek już był odwiedzony w czasie przeszukiwania grafu
`v.p` wierzchołek, z którego doszliśmy do wierzchołka v w trakcie przeszukiwania grafu

przechodzenie grafów w głąb

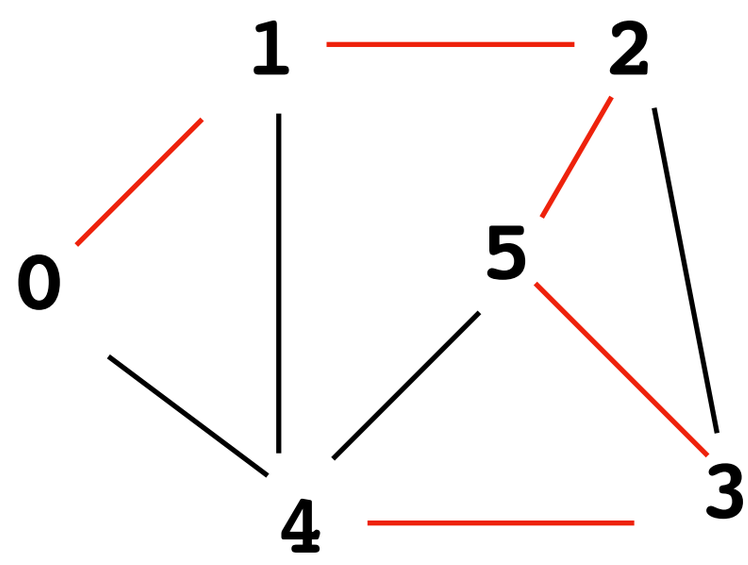
```
dfs(V,u)
// V - lista wierzchołków grafu
// u - wierzcholek startowy
// przeszukujemy graf w głąb zaczynając od wierzchołka
// startowego u
   print u.nr
   u.visited = True
   for  v in u.Adj
     if v.visited == False
        v.p=u
        dfs(V,v)
```

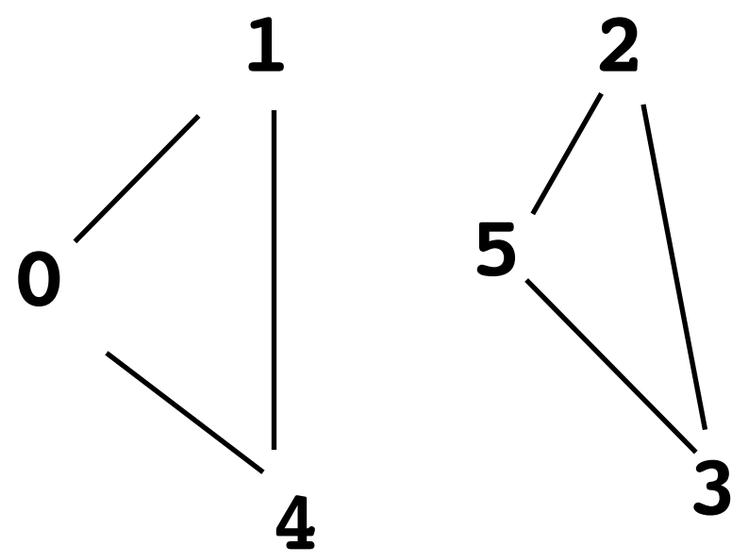
```
dfs(V,V[0]):

      0, 1, 2, 5, 3, 4
pi:       0  1  2  5  3
```



krawędzie postaci `(v.p, v)` tworzą drzewo spinające graf, nazywane też drzewem rozpinającym graf (drzewo pszeszukiwań w głąb)

```
dfs(V,V[0]):

v:     0, 1, 2, 5, 3, 4
v.p:      0  1  2  5  3

dfs(0)
  dfs(1)
    dfs(2)
      dfs(5)
        dfs(3)
          dfs(4)
```





graf może być niespójny

wtedy `dfs(0)` zwróci tylko `0,1,4`

Przeszukanie całego grafu:

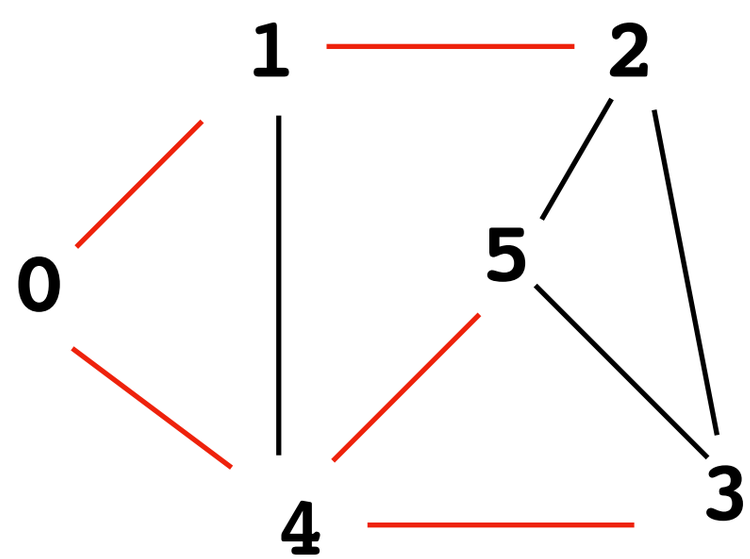
```
dfs(V)
// V - lista wierzchołków grafu
// przeszukujemy graf w głąb
   for  v in V
     if v.visited == False
        dfs(V,v)
```

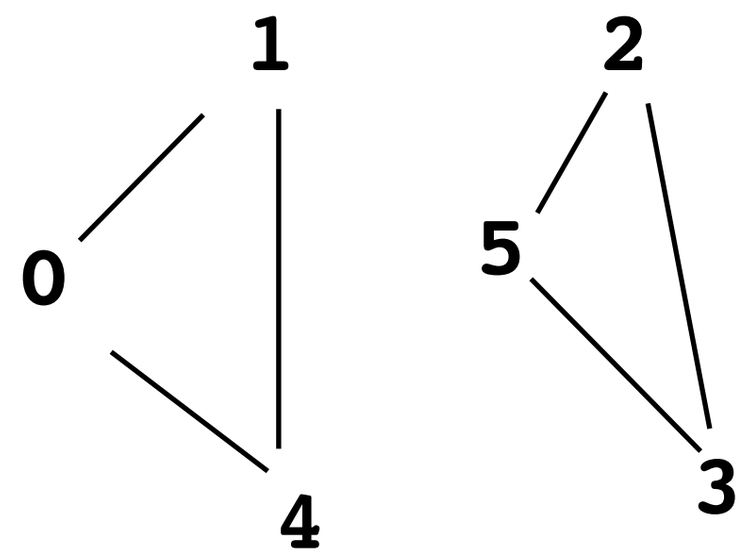
przechodzenie grafów wszerz
(breadth first search)

```
bfs(V,s)
// V - lista wierzchołków grafu; s - wierzchołek startowy
// przeszukujemy graf wszerz rozpoczynając od wierzchołka
// startowego s

   s.visited = True
   enQueue(Q,s)
   while notEmpty(Q)
     u=deQueue(Q)
     print u
     for v in u.Adj
       if v.visited==False
          v.visited=True
          v.p=u
          enQueue(Q,v)
```

krawędzie postaci `(v.p, v)` znowu tworzą drzewo spinające graf (drzewo pszeszukiwań wszerz)

```
bfs(V,V[0]):

v:          0,     1,     4,       2,     3, 5
Q:    [0] [1,4] [4,2] [2,3,5] [3,5] [5] []
p:          0,0    0,1    1,4,4    4,4    4
```





graf może być niespójny
wtedy `bfs(0)` zwróci tylko `0,1,4`

Przeszukanie całego grafu wymaga
wielokrotnego wywołania `bfs()`

```
bfs(V)
// V - lista wierzchołków grafu
// przeszukujemy graf wszerz
   for  v in V
     if v.visited == False
        bfs(V,v)
```

przechodzenie grafów w głąb
(wersja nierekurencyjna, ze stosem)

```
def dfs(V,s)
// V - lista wierzchołków grafu; s - wierzchołek startowy
// przeszukujemy graf wszerz rozpoczynając od s w głąb
// startowego s

   s.visited = True
   push(S,s)
   while notEmpty(S)
      u=pop(S)
      print u
      for v in u.Adj
        if v.visited==False
           v.visited=True
           v.p=u
           pop(S,v)
```

złożoność pesymistyczna

dla grafu $G = (V,E)$ oznaczmy przez V i E rozmiary zbiorów V i E

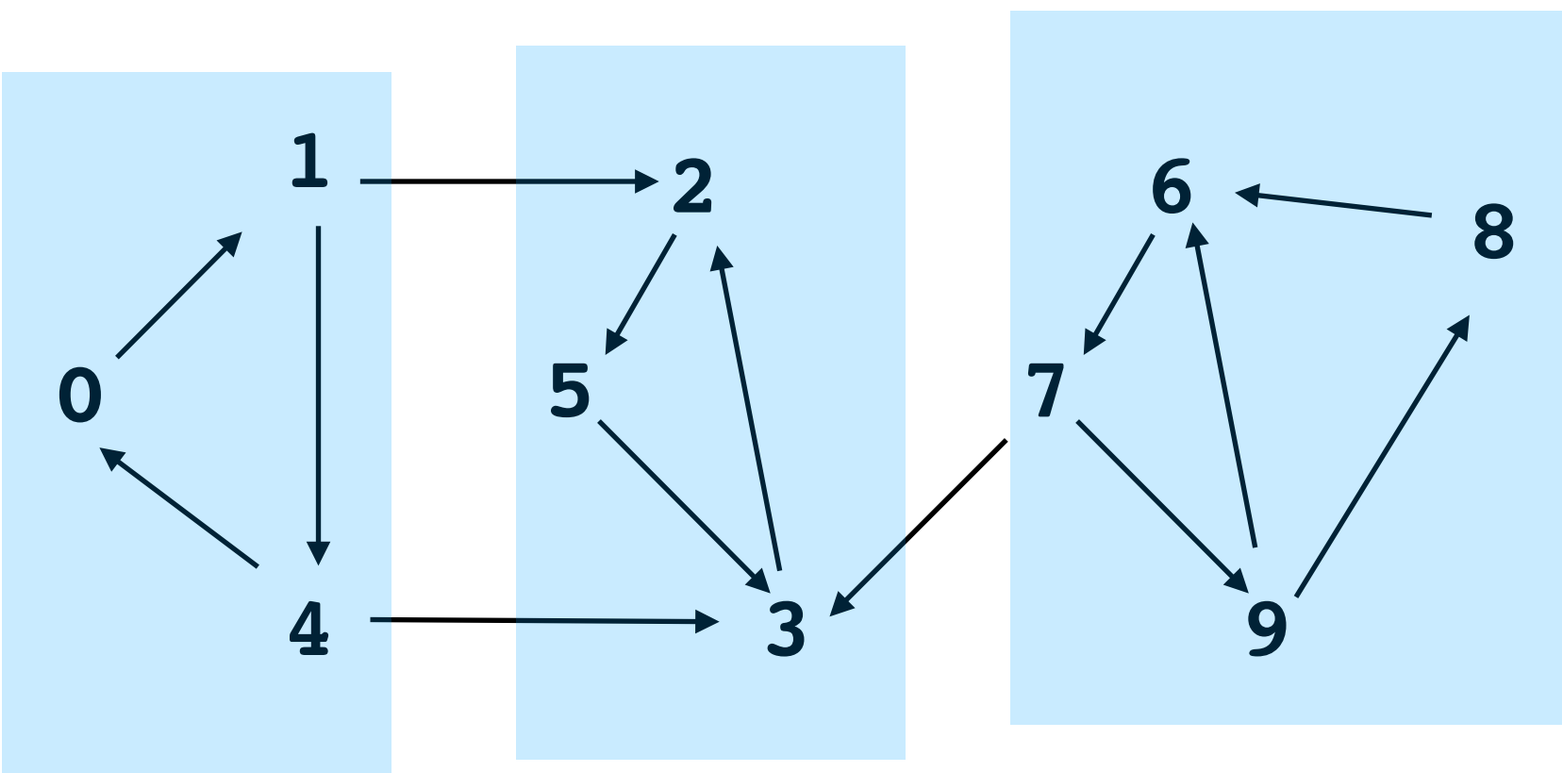
Stwierdzenie Złożoność pesymistyczna przechodzenia grafu w głąb i wszerz jest $O(V+E)$

Dowód Każdy wierzchołek tylko raz zaznaczamy jako odwiedzony (`v.visited=True`) i tylko wtedy wywołujemy rekurencyjnie `dfs` / wstawiamy go do kolejki / wkładamy na stos. Zatem ilość wywołań rekurencyjnych / obrotów pętli `while` jest $O(V)$.

Sumaryczna ilość wykonań ciała pętli `for v in u.Adj` jest nie większa niż E

silnie spójne składowe grafu skierowanego

Definicja. Silnie spójną składową skierowanego grafu jest maksymalny zbiór wierzchołków *U* taki, że dla każdych dwóch wierzchołków *u*, *v* należących do *U* istnieją w grafie ścieżki z *u* do *v* i z *v* do *u*.



Przypomnijmy przeszukanie całego grafu w głąb:

```
dfs(V)
// V - lista wierzchołków grafu
// przeszukujemy graf w głąb
   for  v in V
     if v.visited == False
        dfs(V,v)


dfs(V,u)
// V - lista wierzchołków grafu
// u - wierzcholek startowy
// przeszukujemy graf w głąb zaczynając od wierzchołka
// startowego u
   print u.nr
   u.visited = True
   for  v in u.Adj
     if v.visited == False
        v.p=u
        dfs(V,v)
```

Rozszerzamy funkcję `dfs(V,u)` o liczenie kolejności *odwiedzania* (`u.d`) i *zakończenia przetwarzania* (`u.f`) wierzchołków grafu.

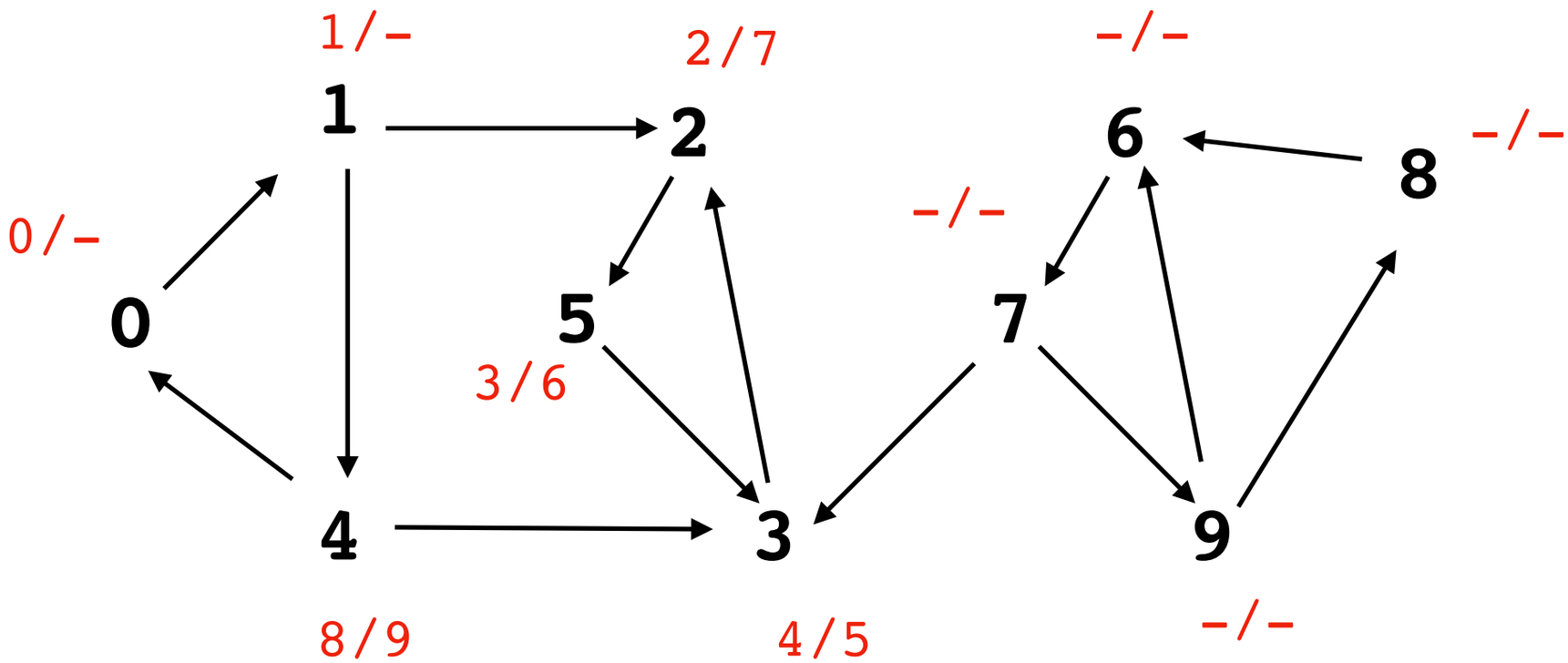
```
dfs(V,u)
// V - lista wierzchołków grafu
// u - wierzcholek startowy
// time - zmienna globalna
   time = time+1  // zwiększenie licznika "czasu"
   u.d = time  // zapamiętujemy czas odwiedzenia wierzchołka u
   u.visited = True
   for  v in u.Adj
     if v.visited == False
        v.p=u
        dfs(V,v)
   time = time+1 // zwiększenie licznika "czasu"
   u.f = time   // zapamiętujemy czas przetworzenia wierzchołka u
```

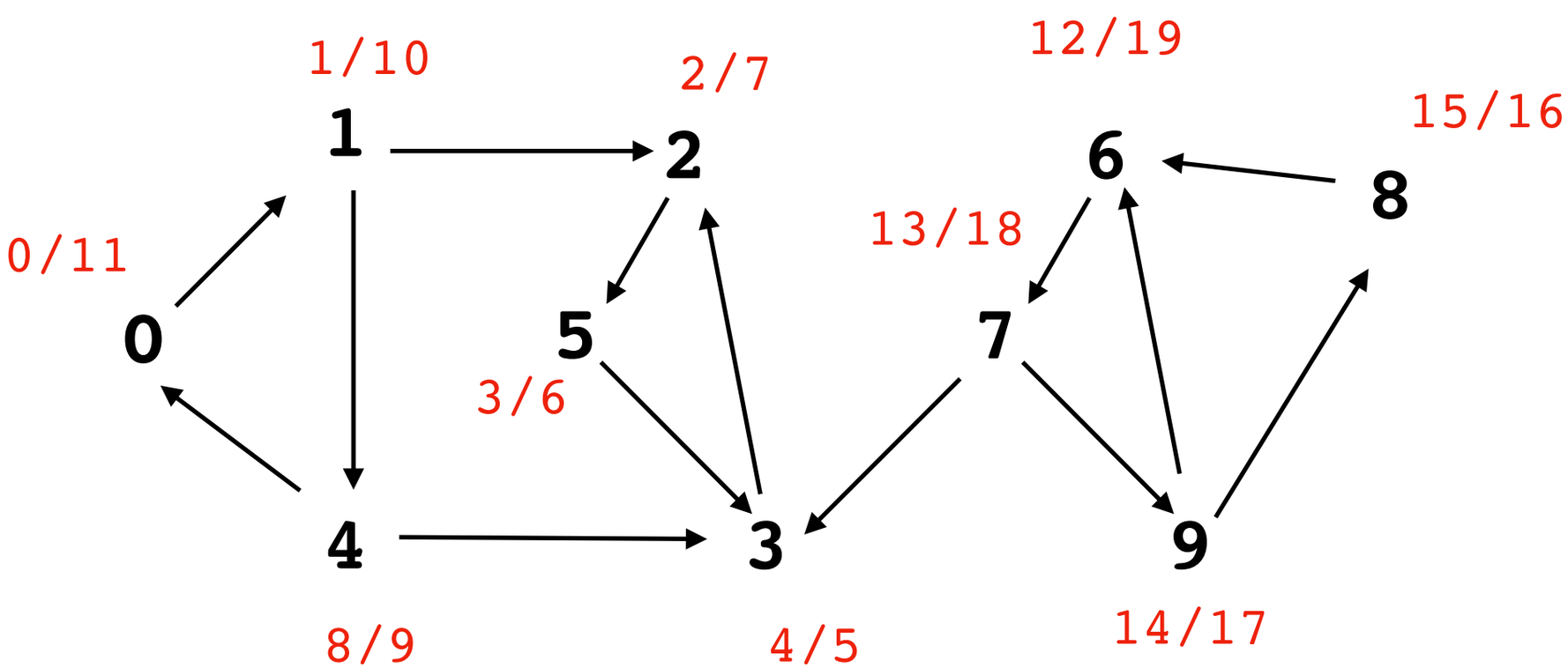
Teraz algorytm znajdowania silnie spójnych składowych grafu skierowanego

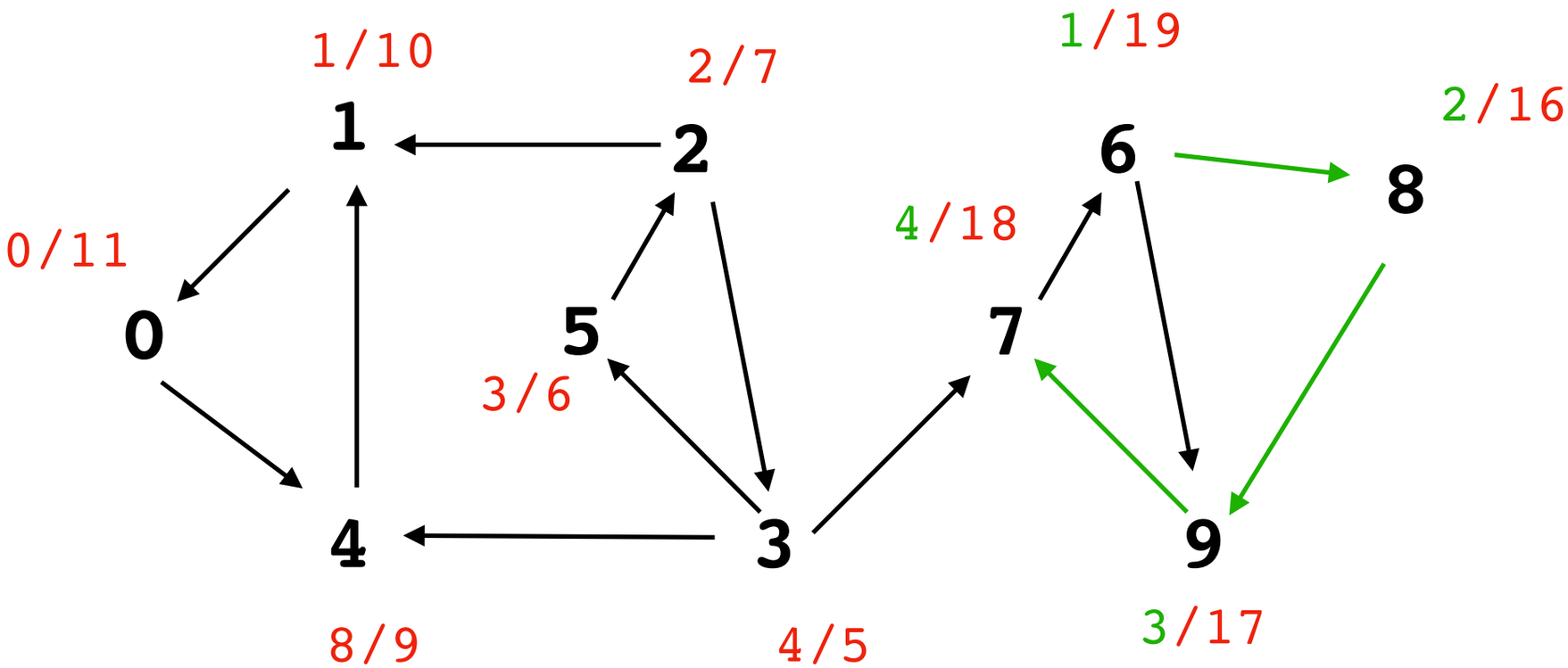
```
Strongly-Connected-Components(V)
  - wykonaj DFS(V) // wylicza u.f dla każdego wierzchołka
  - skonstruuj graf VT będący transpozycją grafu V
  - wykonaj DFS(VT) przebiegając pętlę główną w kolejności
    malejących u.f  // (u.f wyliczone w DFS(V) powżej)
  - uzyskane drzewa przeszukiwania w głąb to silnie spójne składowe
    grafu V; wypisz wierzchołki tych drzew
```

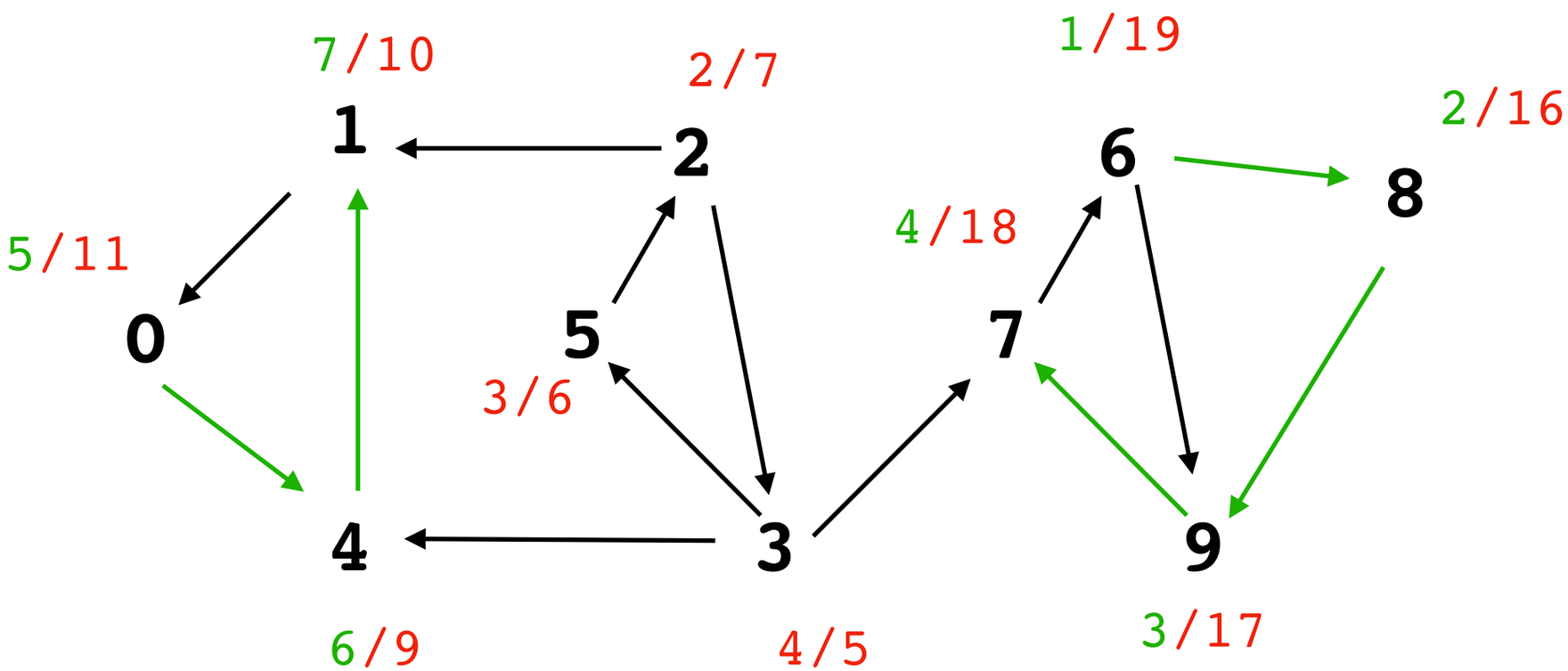
Graf transponowany to graf uzyskany przez odwrócenie kierunku krawędzi

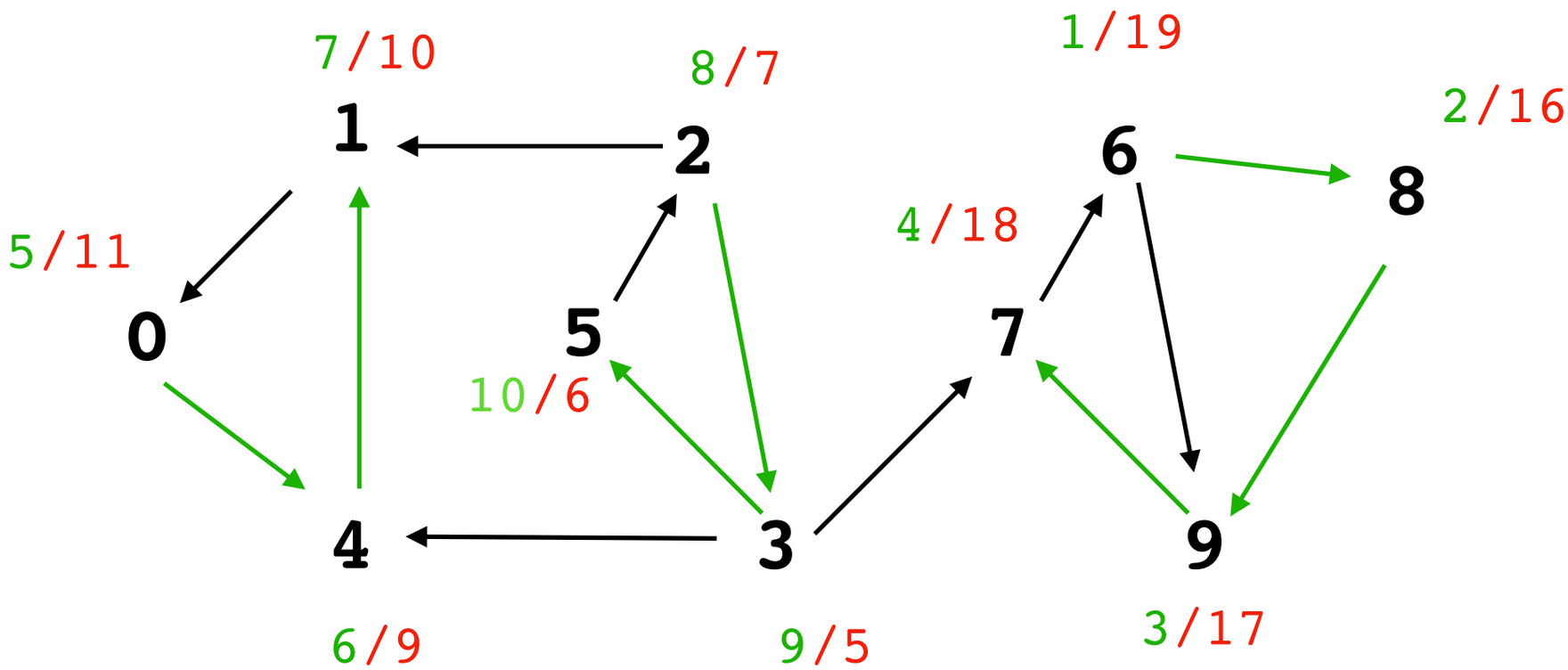
przykład


1/–
2/7
–/–
–/–
–/–
0/–
3/6
8/9
4/5
–/–
0
1
2
3
4
5
6
7
8
9


1/10
2/7
12/19
15/16
13/18
0/11
3/6
8/9
4/5
14/17
0
1
2
3
4
5
6
7
8
9


1/10
2/7
1/19
2/16
1
2
6
8
0/11
4/18
0
5
7
3/6
4
3
9
8/9
4/5
3/17


7/10
2/7
1/19
2/16
1
2
6
8
5/11
4/18
0
5
7
3/6
4
3
9
6/9
4/5
3/17


7/10
8/7
1/19
2/16
1
2
6
8
5/11
4/18
0
5
7
10/6
4
3
9
6/9
9/5
3/17

dla grafu $G = (V,E)$ oznaczmy przez V i E rozmiary zbiorów V i E

Stwierdzenie Złożoność pesymistyczna algorytmu `Strongly-Connected-Components` jest $O(V+E)$

Dowód.
przeszukiwanie grafów (wejściowego i transponowanego)
w czasie $O(V+E)$

zbudowanie listy sąsiedztw dla grafu transponowanego można zrobić w czasie liniowym

w czasie pierwszego przeszukiwania grafu wejściowego można zapisać listę wierzchołków w kolejności rosnących czasów przetworzenia `u.f`

wypisywanie składowych spójności można zrobić w czasie drugiego przeszukiwania grafu

Stwierdzenie Algorytm `Strongly-Connected-Components` poprawnie wyznacza silnie spójne składowe w grafie skierowanym.

Dowód:

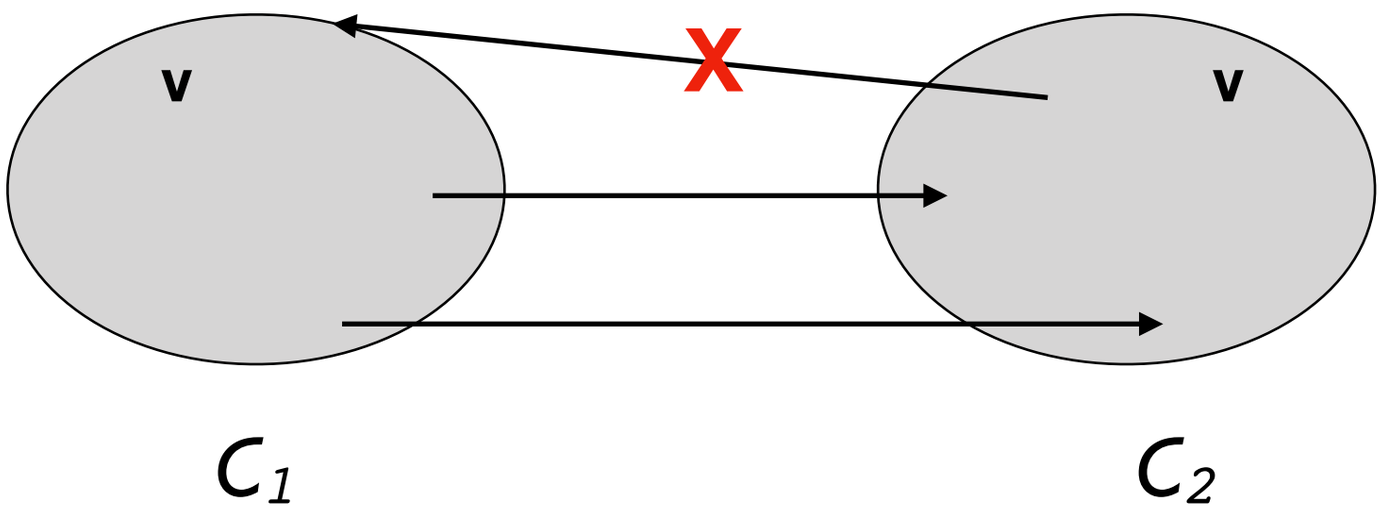
składowe silnej spójności są takie same w grafie wejściowym i w transponowanym

niech C będzie SSS (silnie spójną składową) w grafie
oznaczmy $f(C) = max\{ u.f \ : \ u \in C \}$

Własność. Jeżeli C_1, C_2 są SSS w grafie i jest krawędź
$u_1 \longrightarrow u_2$ gdzie $u_1 \in C_1$ $u_2 \in C_2$, to $f(C_1) > f(C_2)$

Własność. Jeżeli C_1, C_2 są SSS w grafie i jest krawędź $u_1 \longrightarrow u_2$ gdzie $u_1 \in C_1$ $u_2 \in C_2$, to $f(C_1) > f(C_2)$

Dowód Własności.

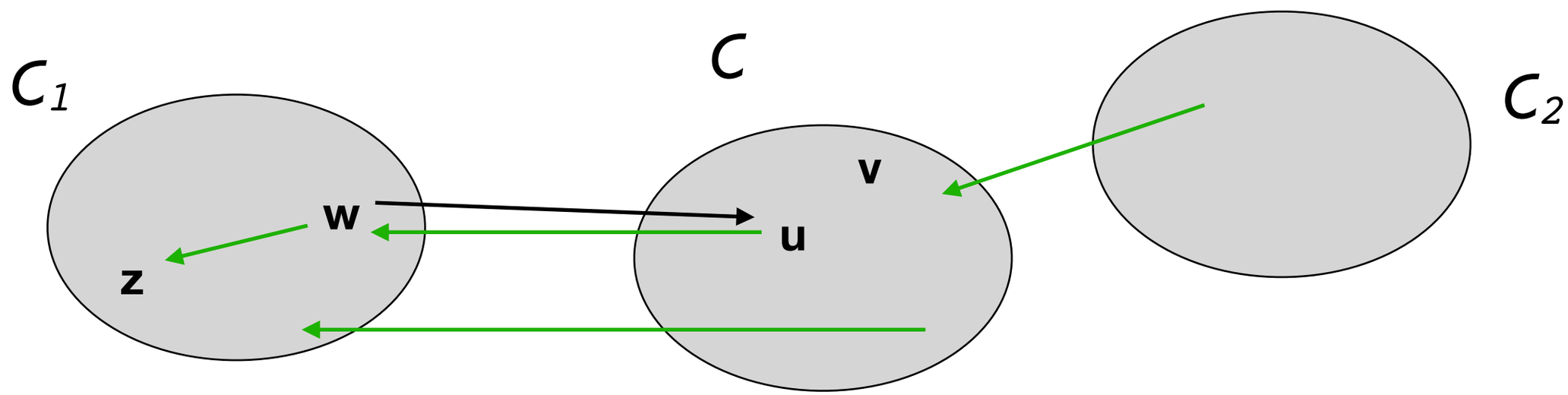


Niech v będzie pierwszym odwiedzonym wierzchołkiem w $C_1 \cup C_2$

(1) $v \in C_1$ to obejdziemy i zakończymy C_2 zanim powrócimy do C_1
(2) $v \in C_2$ to zakończymy C_2 zanim wejdziemy do C_1

Dowód Stwierdzenia.

Sprawdzimy, że jeżeli wywołujemy `dfs`(*v*) , gdzie *C* jest SSS do której należy *v*, to `dfs`(v) przejdzie przez dokładnie wszystkie wierzchołki *C*



$$f(C_1) > f(C) > f(C_2)$$

(1) obejdziemy wszystkie wierzchołki należące do *C* bo gdyby któryś był już wcześniej odwiedzony to *v* tez musiałby być odwiedzony i nie wywoływalibyśmy `dfs`(*v*)

(2) nie wyjdziemy poza *C*, bo dla krawędzi *u* —> *w* wychodzącej z *C* mamy $f(C_1) = z.f > v.f$ a więc `dfs`(z) musiał być już wcześniej zrobiony, zatem *w* już musiał być odwiedzony. (`dfs` wywołujemy w kolejności od najpóźniej odwiedzonych w czasie pierwszego przechodzenia w głąb)