

# Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

b.wozna@ujd.edu.pl

Wykład 1 i 2

# Cel przedmiotu

- Celem wykładów jest zapoznanie studentów z podstawowymi algorytmami grafowymi. Przedstawiona zostanie również ich poprawność i złożoność.
- Celem laboratorium jest implementacja algorytmów wprowadzonych na wykładzie.

# Program przedmiotu

- Reprezentacje grafu.
- Wyszukiwanie wszerz i wyszukiwanie w głąb.
- (Silnie) spójne komponenty.
- Sortowanie topologiczne.
- Minimalne drzewo rozpinające - Algorytmy Kruskala i Prima.
- Znajdowanie cyklu lub ścieżki Eulera. Algorytm Fleury'ego.
- Znajdowanie cyklu lub ścieżki Hamiltona
- Problem najkrótszej ścieżki: Algorytm Floyda-Warshalla
- Problem najkrótszej ścieżki: Algorytm Dijkstry
- Problem najkrótszej ścieżki: Algorytm Bellmana-Forda

# Efekty uczenia się

- E1 Student rozpoznaje złożoność obliczeniową wybranych algorytmów grafowych prezentowanych na wykładzie – efekt weryfikowany na egzaminie.
- E2 Student wyjaśnia wybrane algorytmy grafowe przedstawione na wykładzie – efekt weryfikowany na egzaminie.
- E3 Student implementuje algorytmy grafowe prezentowane na wykładzie w wybranym języku programowania – efekt weryfikowany na laboratorium.
- E4 Student potrafi samodzielnie pozyskiwać informacje z różnych źródeł oraz wykorzystywać je do rozwiązania postawionego problemu – efekt weryfikowany na laboratorium.
- E5 Student dostrzega potrzebę ciągłego aktualizowania i poszerzania wiedzy z zakresu algorytmiki – efekt weryfikowany na laboratorium.

# Literatura

- Cormen T.H., Leiserson Ch.E., Rivest R.L. Wprowadzenie do algorytmów. WNT, 1997 i późniejsze.
- Undirected Graphs:  
<https://algs4.cs.princeton.edu/41graph/>
- Directed Graphs:  
<https://algs4.cs.princeton.edu/42digraph/>

# Wprowadzenie

## Cel wykładu

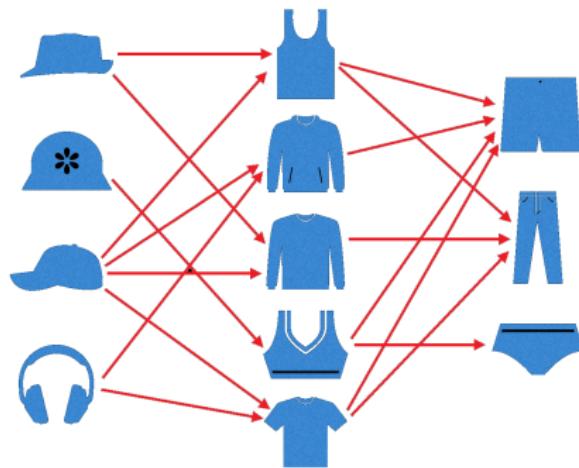
Celem wykładu jest zapoznanie z **teorią grafów** i ich zastosowaniem do rozwiązywania wybranych problemów praktycznych.

## Teoria grafów

Dział matematyki zajmujący się badaniem własności grafów oraz ich zastosowań do rozwiązywania rzeczywistych problemów.

# Przykład problemu z teorii grafów

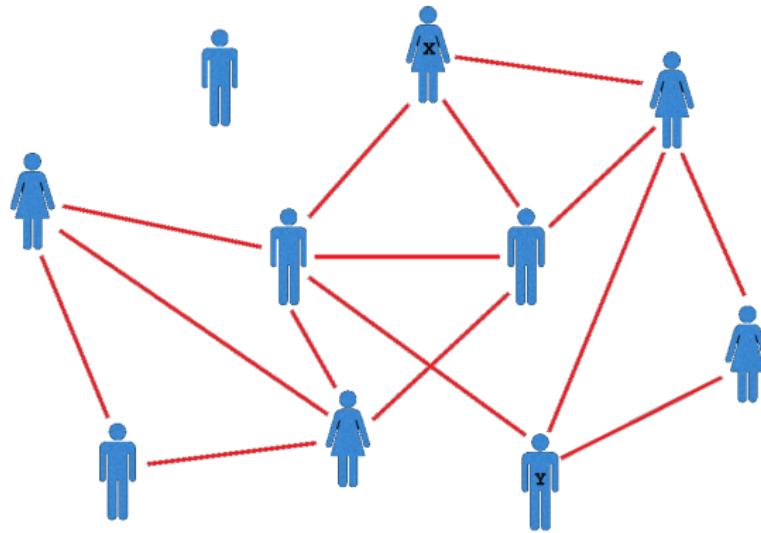
Na podstawie poniższej grafiki, ile różnych zestawów odzieży można skomponować, wybierając po jednym artykule z każdej kategorii?



## Sieć społecznościowa jako problem z teorii grafów

Reprezentacja grafowa pozwala odpowiedzieć na interesujące pytania, takie jak:

- ilu znajomych ma osoba X?
  - ile stopni separacji jest między osobą X a osobą Y?



# Leonhard Euler

- Za pierwszego teoretyka i badacza grafów uważa się Leonharda Eulera<sup>1</sup>, który rozstrzygnął tzw. zagadnienie mostów królewieckich.

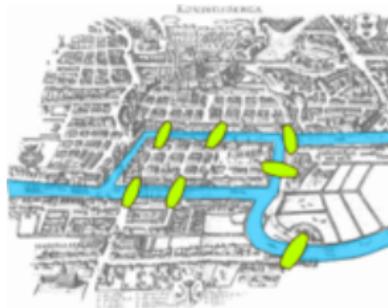


---

<sup>1</sup>Leonhard Euler (ur. 15 kwietnia 1707 r. w Bazylei - Szwajcaria, zm. 18 września 1783 r. w Petersburgu - Rosja) - szwajcarski matematyk, fizyk i astronom, jeden z twórców nowoczesnej matematyki.

# Mosty królewieckie

Przez Królewiec przepływała rzeka Pregole, w której rozwidleniach znajdowały się dwie wyspy. Ponad rzeką przerzucono siedem mostów, z których jeden łączył obie wyspy, a pozostałe mosty łączyły wyspy z brzegami rzeki. Plan mostów pokazuje rysunek:

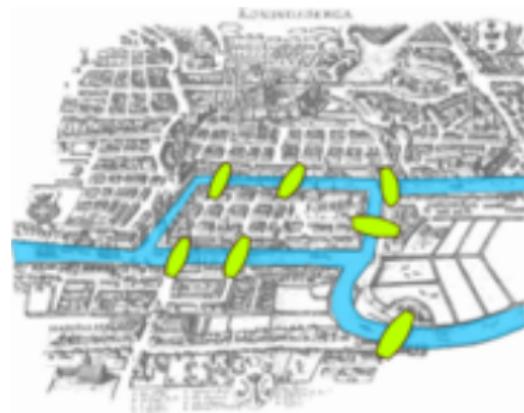
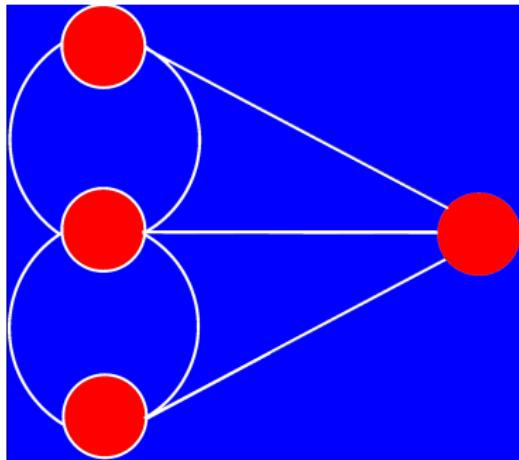


# Mosty królewieckie

- Zwykłe spacerowanie szybko się znudziło mieszkańców Królewca i zaczęli zastanawiać się, czy istnieje taka trasa spacerowa, która przechodzi przez każdy most dokładnie raz, żadnego nie omija, i pozwala wrócić do punktu wyjścia.
- Mieszkańcy nie potrafili rozwiązać postawionego problemu samodzielnie, więc postanowili napisać do matematyka **Leonharda Eulera**.

# Mosty królewieckie

- Euler wykazał, że rozwiązanie problemu mieszkańców nie jest możliwe, a decyduje o tym nieparzysta liczba wylotów mostów zarówno na każdą z wysp, jak i na oba brzegi rzeki – jeśli wejdzie się po raz trzeci na wyspę, nie ma jak z niej wyjść.
- Sytuację tę można przedstawić za pomocą następującego grafu:



# Rodzaje grafów

Istnieje wiele różnych rodzajów grafów. Ważne jest, aby móc rozpoznać, z jakim typem grafu pracujemy, zwłaszcza kiedy programujemy i rozwiązujeemy przy pomocy grafów konkretny problem.

## Rodzaje grafów rozważane na wykładach

- grafy nieskierowane (nieorientowane)
- grafy skierowane (orientowane)
- grafy z wagami

## Inne rodzaje grafów

- multigrafy
- hipergrafy

# Graf nieskierowany

## Definicja

**Grafem nieskierowanym** (grafem) nazywamy parę  $G = (V, E)$ , gdzie:

- $V$  to zbiór **wierzchołków** oraz
- $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$  to zbiór *nieuporządkowanych par* wierzchołków, zwanych **krawędziami**.

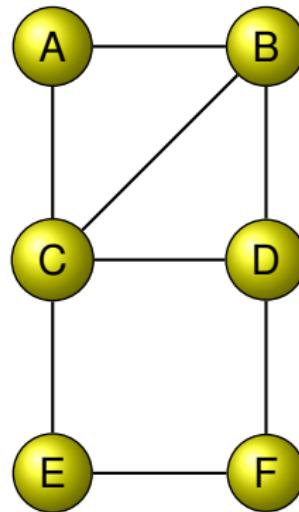
W grafie nieskierowanym nie mogą występować pętle.

## Intuicyjnie

Graf nieskierowany to graf, w którym krawędzie nie mają orientacji. To znaczy, krawędź od węzła  $u$  do węzła  $v$  jest identyczna z krawędzią od węzła  $v$  do węzła  $u$ .

# Graf nieskierowany $G = (V, E)$ - przykład

- $V = \{A, B, C, D, E, F\}$
- $E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}, \{C, D\}, \{E, F\}\}$
- $|V| = n = 6, |E| = m = 6$



## Uwaga!

Na powyższym grafie węzły mogą reprezentować miasta, a krawędzie drogi dwukierunkowe.

# Graf skierowany

## Definicja

**Grafem skierowanym** (lub **digrafem**) nazywamy parę  $G = (V, E)$ :

- $V$  to skończony zbiór **wierzchołków** oraz
- $E \subseteq \{(u, v) : u, v \in V\}$  to zbiór uporządkowanych par wierzchołków ze zbioru  $V$ , zwanych **krawędziami**.

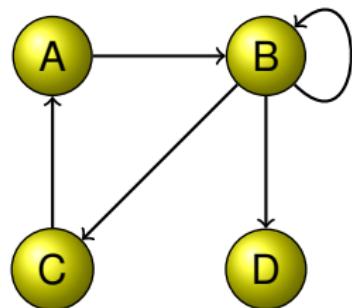
Możliwe jest istnienie **pętli** od danego wierzchołka do niego samego.

## Intuicyjnie

Graf skierowany to graf, w którym krawędzie mają określoną orientację. Oznacza to, że krawędź prowadząca od węzła  $u$  do węzła  $v$  różni się od krawędzi biegnącej od węzła  $v$  do węzła  $u$ , co podkreśla kierunek relacji między węzłami.

# Graf skierowany $G = (V, E)$ - przykład

- $V = \{A, B, C, D\}$
- $E = \{(A, B), (B, B), (B, C), (B, D), (C, A), (C, D), (D, C)\}$



W przedstawionym grafie:

- Węzły mogą reprezentować osoby.
- Krawędź  $(u, v)$  oznacza, że osoba  $u$  wręczyła prezent osobie  $v$ .
- Krawędź przychodząca oznacza otrzymanie prezentu.
- Krawędź wychodząca symbolizuje wręczenie prezentu.

Przykład:

- Osoba  $A$  wręczyła prezent osobie  $B$ .
- Osoba  $B$  wręczyła prezent samej sobie oraz osobom  $C$  i  $D$ .

# Etykietowany graf nieskierowany

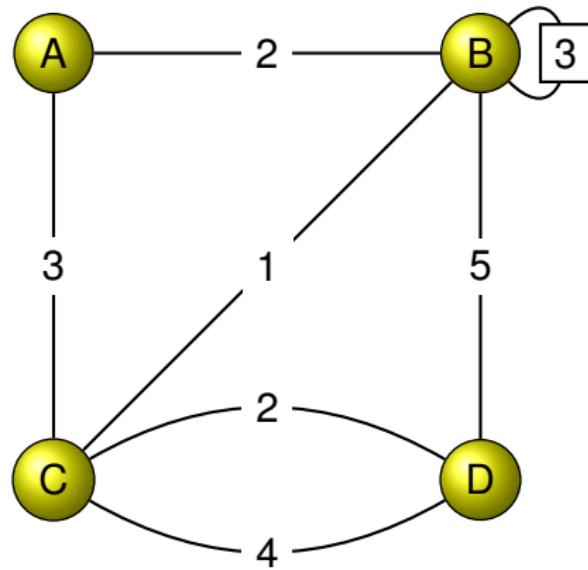
## Definicja

**Etykietowanym grafem nieskierowanym** nazywamy strukturę

$G = (V, E, w : E \rightarrow R)$ , gdzie

- $V$  to zbiór wierzchołków,
- $E \subseteq \{\{u, v\} : u, v \in V\}$  to zbiór par wierzchołków ze zbioru  $V$ , zwanych krawędziami.
- $w : E \rightarrow R$  to funkcja **wagi**; wagi reprezentują pewne wielkości (np. koszt, odległość, ilość, itp.).

# Etykietowany graf nieskierowany - przykład



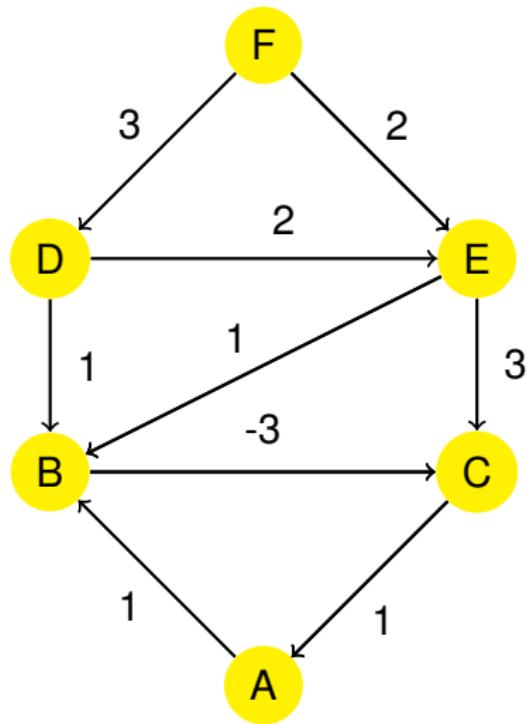
# Etykietowany graf skierowany

## Definicja

**Etykietowanym grafem skierowanym** nazywamy strukturę  $G = (V, E, w : E \rightarrow R)$ , gdzie

- $V$  to zbiór wierzchołków,
- $E \subseteq \{(u, v) : u, v \in V\}$  to zbiór uporządkowanych par wierzchołków ze zbioru  $V$ , zwanych krawędziami.
- $w : E \rightarrow R$  jest funkcją **wagi**; wagi reprezentują pewne wielkości (np. koszt, odległość, ilość, itp.).

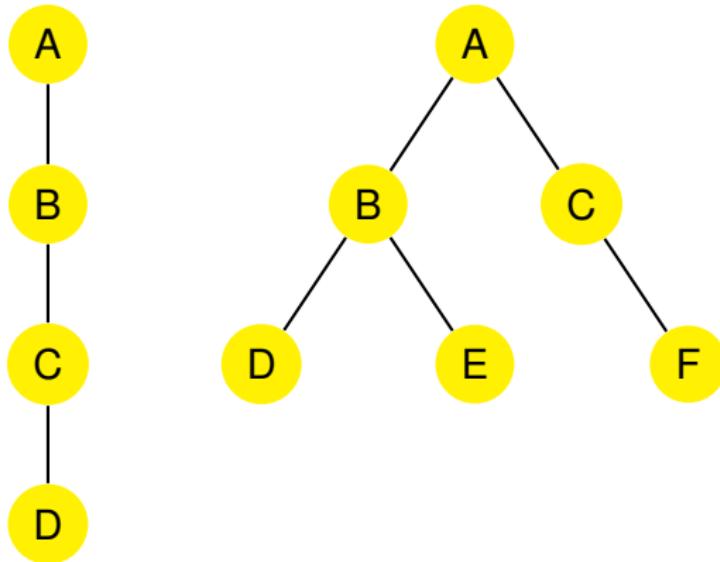
# Etykietowany graf skierowany - przykład



# Drzewa

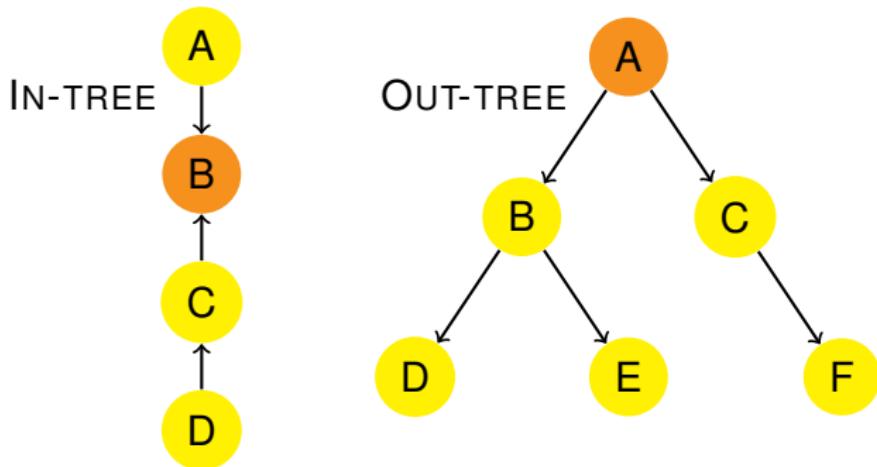
**Drzewo** - graf nieskierowany bez cykli.

Równoważnie, drzewo to **graf spójny** o  $n$  węzłach i  $n - 1$  krawędziach.



# Drzewa ukorzenione

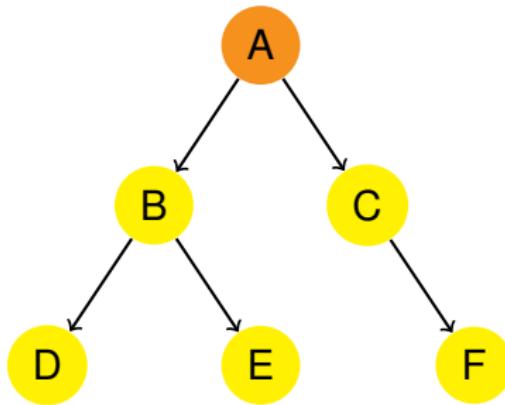
**Drzewo ukorzenione** to drzewo z wyróżnionym węzłem, zwanym **korzeniem**, w którym każda krawędź jest skierowana albo **w kierunku od** albo **w kierunku do** węzła korzenia.



Gdy krawędzie są odwrócone od korzenia, to mówimy, że graf jest typu **out-tree**, w przeciwnym przypadku mówimy, że graf jest typu **in-tree**.

# Skierowane grafy acykliczne - ang. Directed Acyclic Graphs (DAG)

DAG to grafy skierowane bez cykli. Grafy te odgrywają ważną rolę w reprezentacji struktur danych z zależnościami. Wszystkie out-trees są DAG-ami. Odwrotna zależność nie zachodzi.



# Grafy dwudzielne I

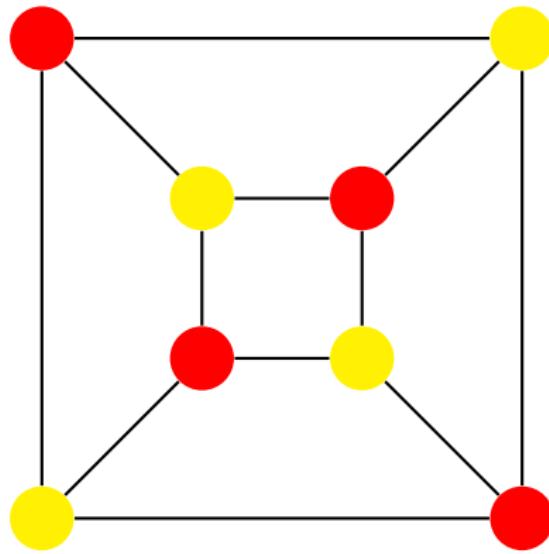
**Graf dwudzielny** (ang. bipartite graph lub bigraph) to graf, którego wierzchołki możemy podzielić na dwa rozłączne zbiory  $U$  i  $V$ .

Wierzchołki należące do zbioru  $U$  mogą się łączyć krawędziami tylko z wierzchołkami ze zbioru  $V$  i na odwrót.

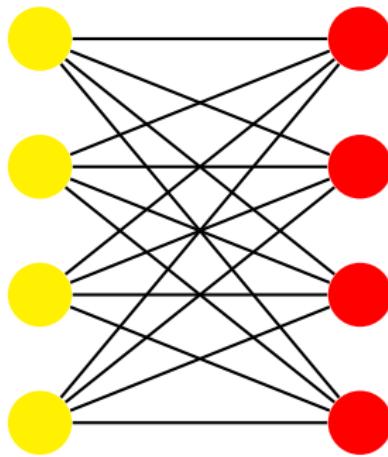
Wierzchołki należące do zbioru  $U$  możemy traktować jako pokolorowane, np. na żółto, a wierzchołki należące do zbioru  $V$  jako pokolorowane, np. na czerwono.

Graf jest dwudzielny, jeśli możemy pokolorować wszystkie jego wierzchołki w taki sposób, aby żaden z sąsiadów danego wierzchołka nie miał tego samego koloru co on.

# Grafy dwudzielne II



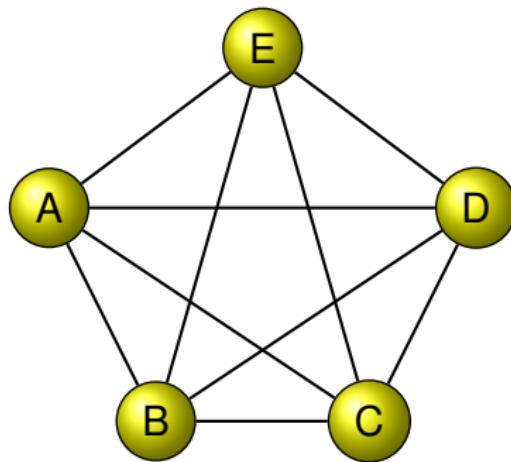
# Grafy dwudzielne III



# Grafy pełne (ang. complete graph)

**Graf pełny** to graf nieskierowany, w którym każda para wierzchołków jest połączona krawędzią.

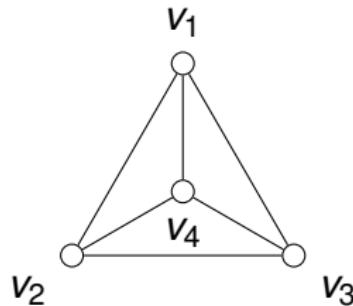
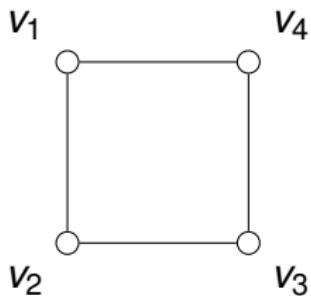
Graf pełny o  $n$  wierzchołkach oznacza się przez  $K_n$  (poniżej  $K_5$ )



Graf pełny o  $n$  wierzchołkach posiada  $\frac{n \cdot (n-1)}{2}$  krawędzi – przykładowo graf  $K_5$  posiada 10 krawędzi.

# Grafy planarny

- **Graf planarny** - graf nieskierowany, który można narysować na płaszczyźnie tak, by krawędzie grafu nie przecinały się ze sobą. Planarność ma duże zastosowanie w informatyce, m.in., w graficznej reprezentacji różnego rodzaju układów (np. scalonych, bramek, etc.).



# Reprezentacja grafów w komputerze

- Macierz sąsiedztwa
- Listy sąsiedztwa
- Lista krawędzi

# Macierz sąsiedztwa

## Definicja

Dany jest graf  $G = (V, E)$ , gdzie  $V = \{v_1, \dots, v_n\}$  (tj. zakłada się, że wierzchołki są ponumerowane w pewien dowolny sposób) oraz  $|V| = n$ . **Macierz sąsiedztwa** grafu  $G$  to macierz  $M(G) = (a_{ij})$  o wymiarze  $n \times n$  taka, że

$$a_{ij} = \begin{cases} 1, & \text{jeśli } (v_i, v_j) \in E \\ 0, & \text{w przeciwnym razie} \end{cases}$$

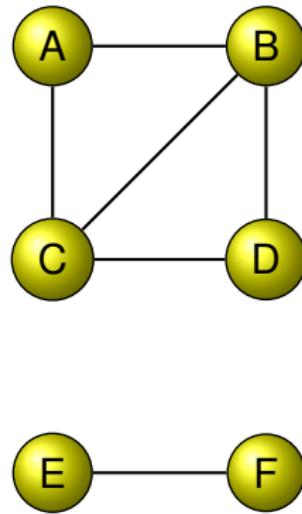
# Macierz sąsiedztwa - graf nieskierowany, przykład

- $V = \{A, B, C, D, E, F\}$

- $E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}, \{C, D\}, \{E, F\}\}$

- $M(G) =$

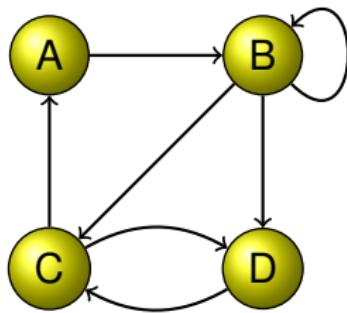
$$\begin{array}{cccccc} & A & B & C & D & E & F \\ A & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$



# Macierz sąsiedztwa - Graf skierowany, przykład

- $V = \{A, B, C, D\}$
- $E = \{(A, B), (B, B), (B, C), (B, D), (C, A), (C, D), (D, C)\}$
- $M(G) =$

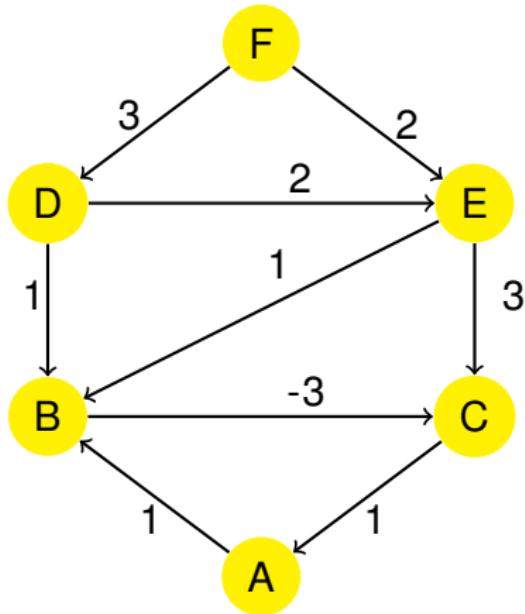
$$\begin{array}{l} A \quad B \quad C \quad D \\ \begin{matrix} A & \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \\ B & \\ C & \\ D & \end{matrix} \end{array}$$



# Macierz sąsiedztwa - Etykietowany graf skierowany, przykład

$$M(G) =$$

$$\begin{array}{cccccc} & A & B & C & D & E & F \\ \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix} & \left( \begin{array}{cccccc} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -3 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 2 & 0 \end{array} \right) \end{array}$$



# Macierz sąsiedztwa - własności

- Macierz sąsiedztwa dla grafów bez wag jest macierzą binarną.
- Macierz sąsiedztwa wymaga  $\Theta(|V|^2)$  pamięci, niezależnie od liczby krawędzi w tym grafie.
- Dla grafów nieskierowanych macierz sąsiedztwa jest symetryczna, dlatego w pewnych zastosowaniach opłaca się pamiętać tylko elementy na i powyżej głównej przekątnej macierzy sąsiedztwa.

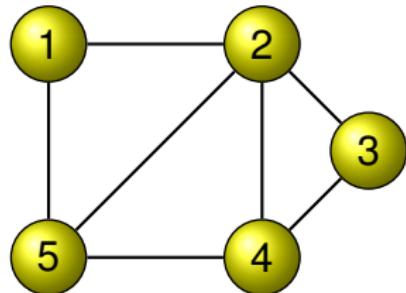
$$M(G) = \begin{pmatrix} & A & B & C & D & E & F \\ A & 0 & 1 & 1 & 0 & 0 & 0 \\ B & 1 & 0 & 1 & 1 & 0 & 0 \\ C & 1 & 1 & 0 & 1 & 0 & 0 \\ D & 0 & 1 & 1 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 0 & 1 \\ F & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

# Listy sąsiedztwa

## Definicja

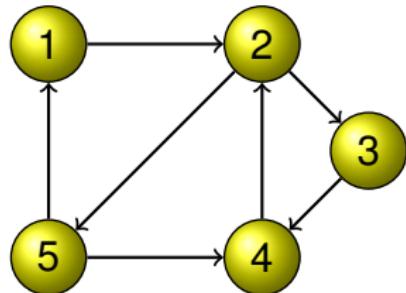
- W reprezentacji grafu  $G = (V, E)$  za pomocą **list sąsiedztwa** dana jest tablica  $A$  zawierająca  $|V|$  list, po jednej dla każdego wierzchołka z  $V$ .
- Dla każdego  $u \in V$  elementami listy sąsiedztwa  $A[u]$  są wszystkie wierzchołki  $v$  takie, że krawędź  $(u, v) \in E$ , tzn. w  $A[u]$  przechowujemy zbiór wierzchołków połączonych krawędzią z  $u$ .

# Listy sąsiedztwa - przykład



- 1 – >[(2),(5)]
- 2 – >[(1),(3),(4),(5)]
- 3 – >[(2),(4)]
- 4 – >[(2),(3),(5)]
- 5 – >[(1),(2),(4)]

# Listy sąsiedztwa - przykład



- 1 – >[(2)]
- 2 – >[(3),(5)]
- 3 – >[(4)]
- 4 – >[(2)]
- 5 – >[(1),(4)]

# Listy sąsiedztwa

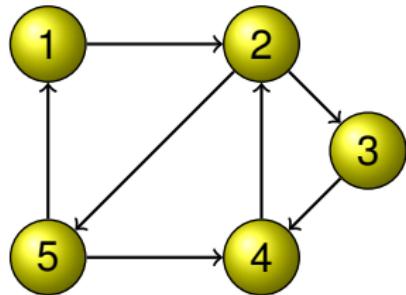
- Jeżeli  $G$  jest grafem skierowanym, to suma długości wszystkich list sąsiedztwa wynosi  $|E|$ , ponieważ krawędź postaci  $(u, v)$  jest reprezentowana przez wystąpienie  $v$  na liście  $A[u]$ .
- Jeżeli  $G$  jest grafem nieskierowanym, to suma długości wszystkich list sąsiedztwa wynosi  $2 \cdot |E|$ , ponieważ dla nieskierowanej krawędzi  $\{u, v\}$  wierzchołek  $u$  występuje na liście sąsiedztwa  $v$  i odwrotnie,  $v$  występuje na liście  $u$ .
- Reprezentacja listowa do reprezentacji grafu wymaga  $\Theta(|V| + |E|)$  pamięci.
- Reprezentacja listowa jest preferowana dla grafów **rzadkich** – tj. takich, dla których  $|E|$  jest dużo mniejsze niż  $|V|^2$ .
- Reprezentacja macierzy sąsiedztwa jest preferowana dla grafów **gęstych** – tj. takich, dla których  $|E|$  jest bliskie  $|V|^2$ .

# Lista Krawędzi

Dany jest graf  $G = (V, E)$ ,  $|V| = n$ ,  $|E| = m$

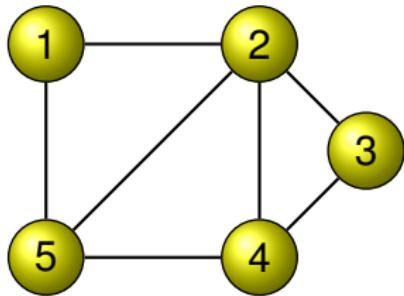
- Lista krawędzi to lista, na której przechowujemy wszystkie krawędzie występujące w grafie.
- Dla grafu skierowanego, może to być lista (tablica) zawierająca uporządkowane pary wierzchołków.
- Dla grafu nieskierowanego, może to być lista (tablica) zawierająca nieuporządkowane pary wierzchołków.
- Ponieważ każda krawędź zawiera tylko dwie liczby (tj. zakładamy, że wierzchołki są ponumerowane), całkowita zajętość pamięci dla listy krawędzi jest rzędu  $\Theta(|E|)$ .

# Lista krawędzi - graf skierowany



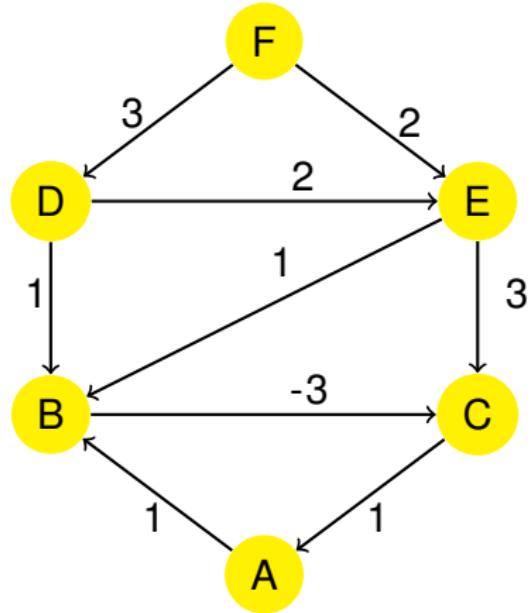
$$L = [(1, 2), (2, 3), (3, 4), (4, 2), (2, 5), (5, 4), (5, 1)].$$

# Lista krawędzi - graf nieskierowany



$$L = [(1, 2), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5)].$$

# Lista krawędzi - etykietowany graf skierowany, przykład



$$\begin{aligned} L = \\ [(A, B, 1), (B, C, -3), (C, A, 1), \\ (D, B, 1), (E, B, 1), (E, C, 3), \\ (D, E, 2), (F, D, 3), (F, E, 2)]. \end{aligned}$$

# Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

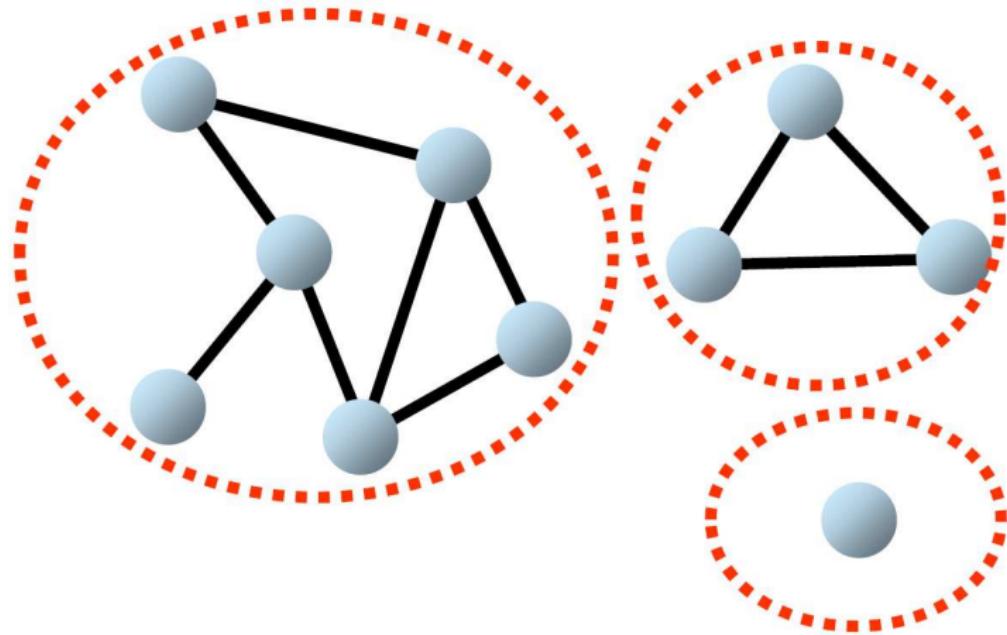
b.wozna@ujd.edu.pl

Wykład 3 i 4

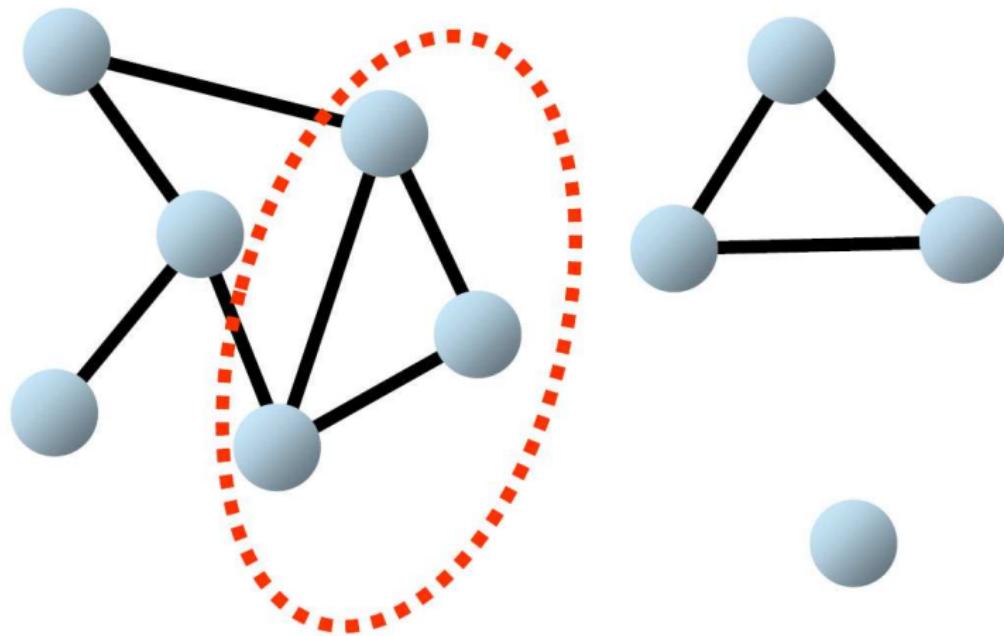
# Spójność grafu

- 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 jedną lub większą liczbę spójnych składowych (ang. connected components).
- Graf skierowany jest **siłnie spójny**, jeśli każde dwa wierzchołki są 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.

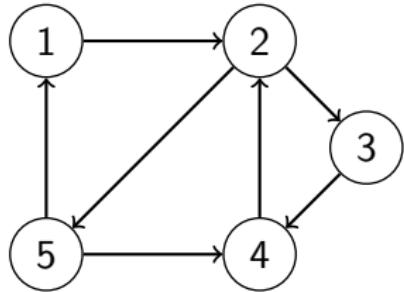
# Przeszukiwania grafu w głąb - idea

**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):*

```

1: for each vertex  $u \in V$  do
2:    $color[u] = WHITE$ 
3:    $pre[v] = NILL$ 
4: end for
5:  $time = 0$ 
6: for each vertex  $u \in V$  do
7:   if  $color[u] == WHITE$  then
8:      $VISIT(G, u)$ 
9:   end if
10: end for

```

*VISIT( $G, u$ )*

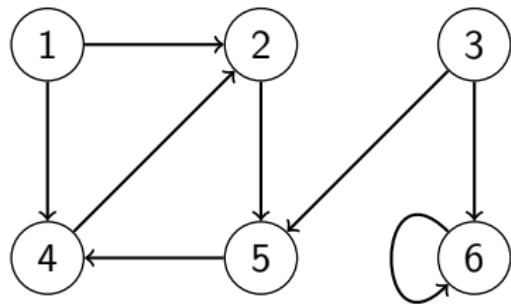
```

1:  $time = time + 1$ 
2:  $d[u] = time$ 
3:  $color[u] = GRAY$ 
4: for each  $v \in Adj[u]$  do
5:   if  $color[v] == WHITE$  then
6:      $pre[v] = u$ 
7:      $VISIT(G, v)$ 
8:   end if
9: end for
10:  $color[u] = RED$ 
11:  $time = time + 1$ 
12:  $f[u] = time$ 

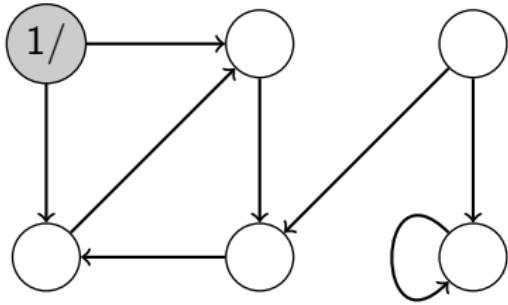
```

# Działanie algorytmu - Przykład

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

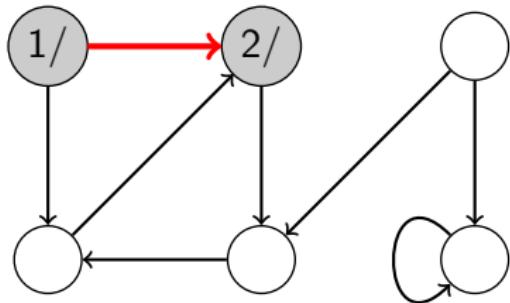


Krok: 1

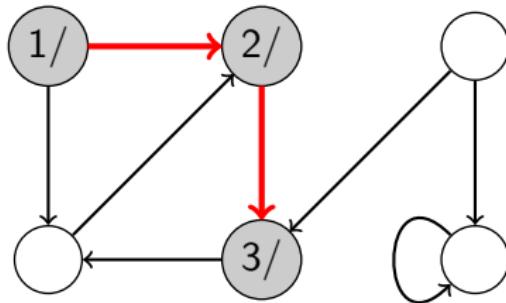


# Działanie algorytmu - Przykład

Krok: 2

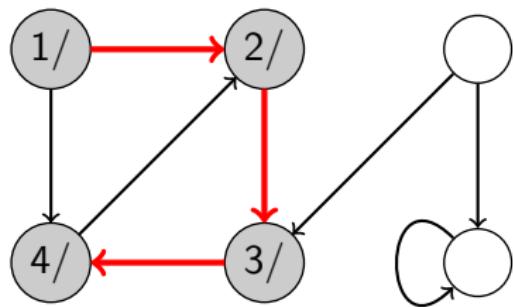


Krok: 3

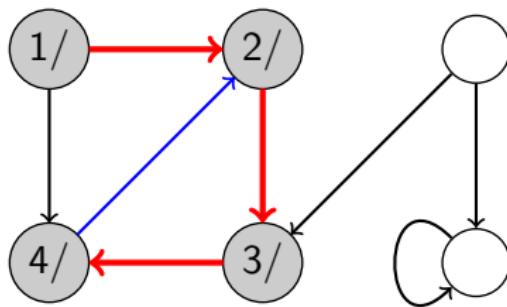


# Działanie algorytmu - Przykład

Krok: 4

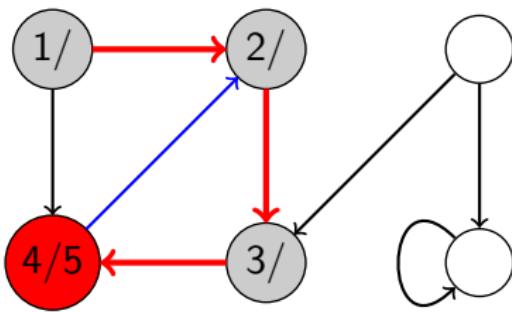


Krok: 5

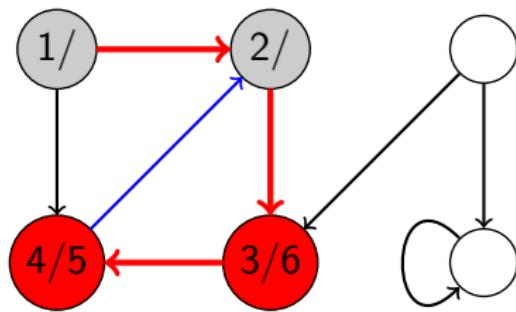


# Działanie algorytmu - Przykład

Krok: 6

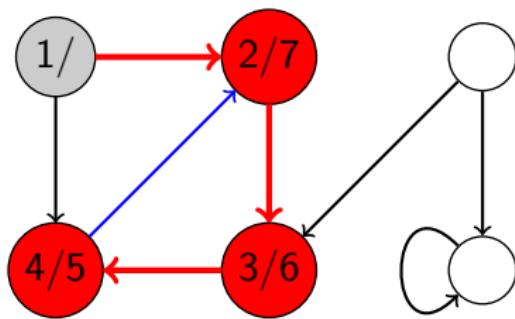


Krok: 7

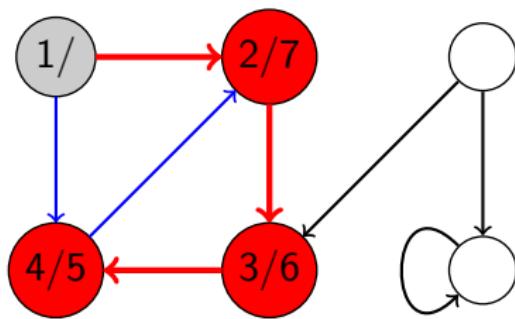


# Działanie algorytmu - Przykład

Krok: 8

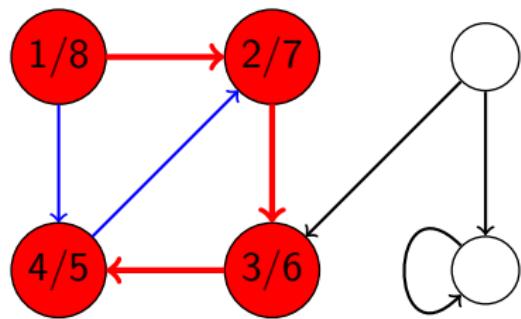


Krok: 9

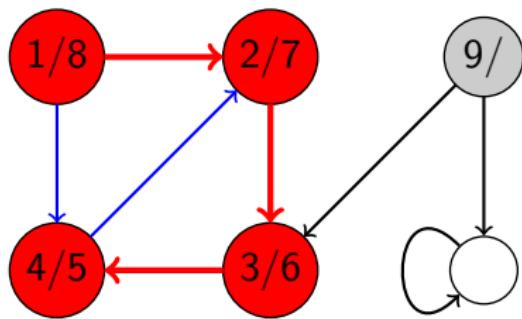


# Działanie algorytmu - Przykład

Krok: 10

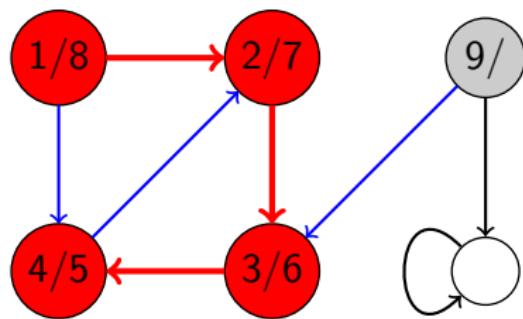


Krok: 11

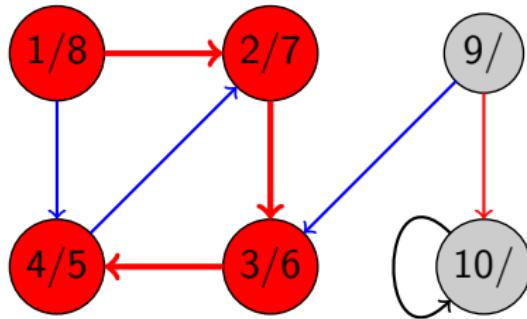


# Działanie algorytmu - Przykład

Krok: 12

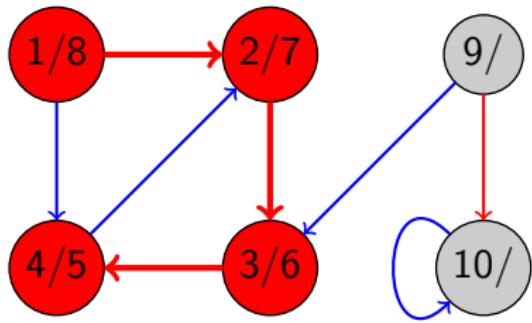


Krok: 13

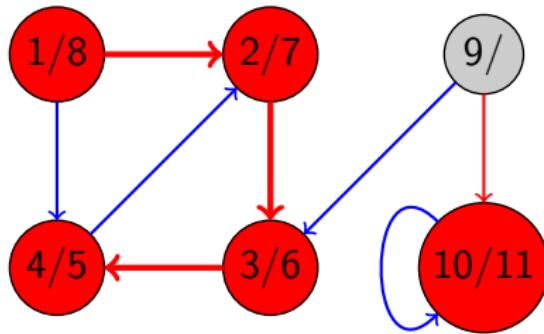


# Działanie algorytmu - Przykład

Krok: 14

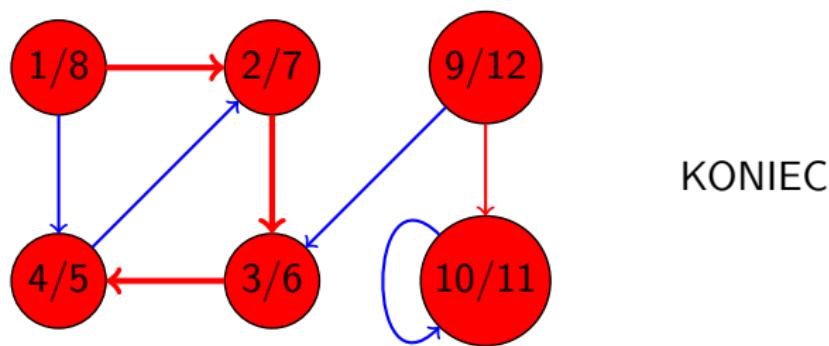


Krok: 15



# 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 wyjątkiem 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  $\Theta(V + E)$ .

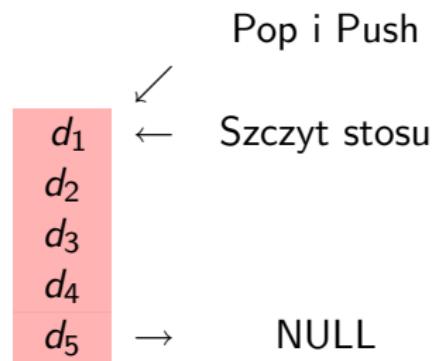
# Stos

- 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 góre.

# Stos - operacje

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

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



# DFS - reprezentacja macierzowa grafu

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

```
1: function getUnVisitedVertex(Vertex  $a$ , BoolVector  $visited$ )
2: for  $b = 0$  to  $n - 1$  do
3:   if  $edge[a][b] = \text{true}$  and  $visited[b] = \text{false}$  then
4:     return  $b$ 
5:   end if
6: end for
7: 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                                // rozpoczęj 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:

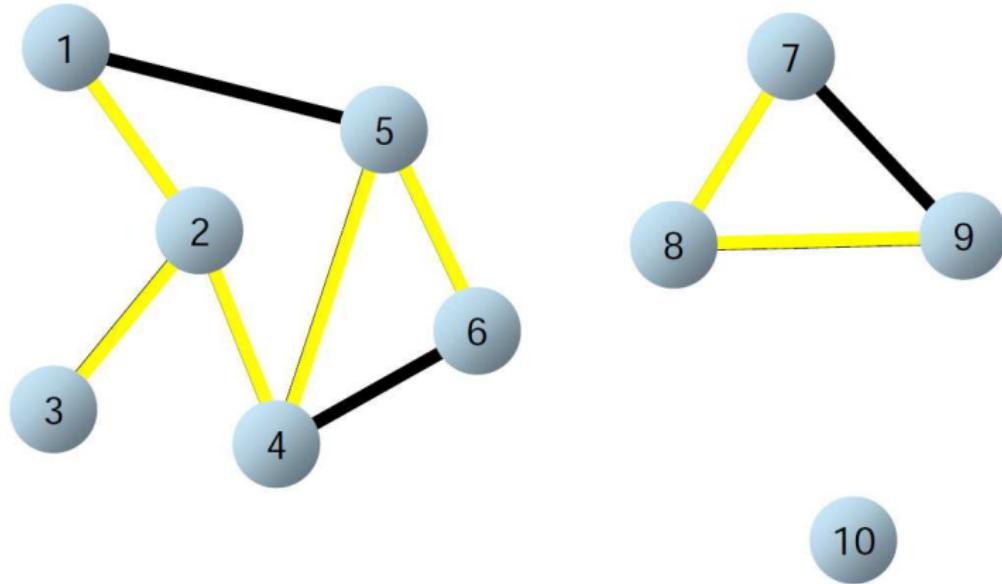
- graf reprezentowany jako listy sąsiedztwa:  $\Theta(|V| + |E|)$ , na co składa się początkowa inicjalizacja (zabiera  $\Theta(|V|)$ ) i przechodzenie po wszystkich sąsiadach każdego wierzchołka (zabiera  $\Theta(|E|)$ ) ponieważ suma długości wszystkich list sąsiedztwa wynosi  $O(|E|)$ .
- graf reprezentowany 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  $K = (d_1, d_2, \dots, d_n)$  oznacza kolejkę, wtedy:

- Wstawianie elementu do kolejki:  $\text{enqueue}(K, d) = (d_1, d_2, \dots, d_n, d)$
- Pobieranie elementu z kolejki:  $\text{dequeue}(K) = (d_2, \dots, d_n)$ , o ile  $n > 1$
- Obsługiwanie pierwszego elementu z kolejki bez jego usuwania :  
 $\text{first}(K) = d_1$
- Sprawdzanie niepustosci kolejki:  $\text{empty}(K)$  wtw., gdy  $n = 0$

Początek kolejki → 

$d_1$	$d_2$	$d_3$	$d_4$	$d_5$
-------	-------	-------	-------	-------

 ← 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 wierzchołkami 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

$BFS(G, s)$ :

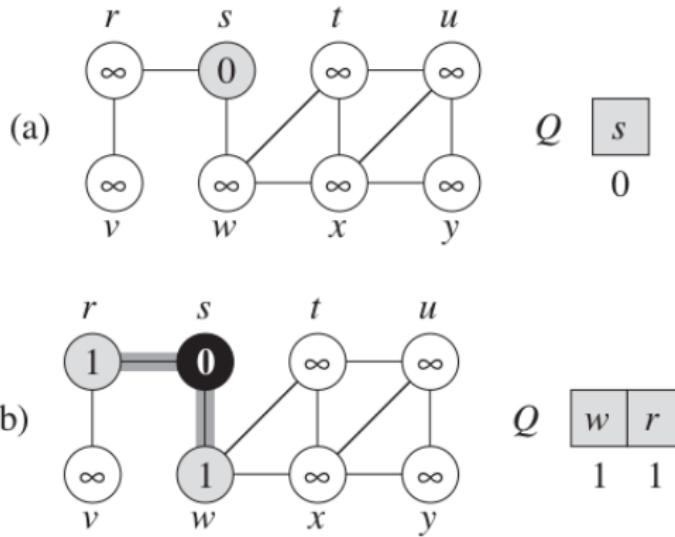
```

1: for each vertex  $u \in V - \{s\}$  do
2:    $color[u] = WHITE$ 
3:    $d[u] = \infty$ 
4:    $pre[u] = NIL$ 
5: end for
6:  $color[s] = GRAY$ 
7:  $d[s] = 0$ 
8:  $pre[s] = NIL$ 
9:  $EnQueue(Q, s)$ 
```

```

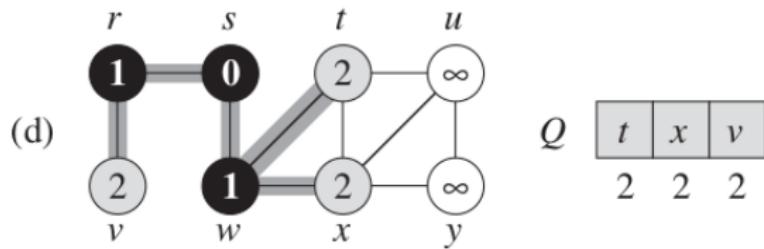
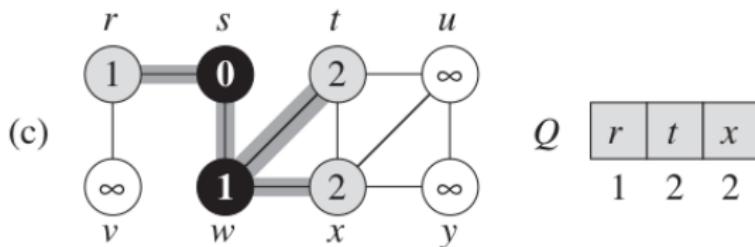
1: while  $Q \neq \emptyset$  do
2:    $u = DeQueue(Q)$ 
3:   for each  $v \in Adj[u]$  do
4:     if  $color[v] == WHITE$ 
      then
         $color[v] = GRAY$ 
         $d[v] = d[u] + 1$ 
         $pre[v] = u$ 
         $EnQueue(Q, v)$ 
      end if
    end for
11:  $DeQueue(Q)$ 
12:  $color[u] = BLACK$ 
13: end while
```

# Algorytm BFS - przykład



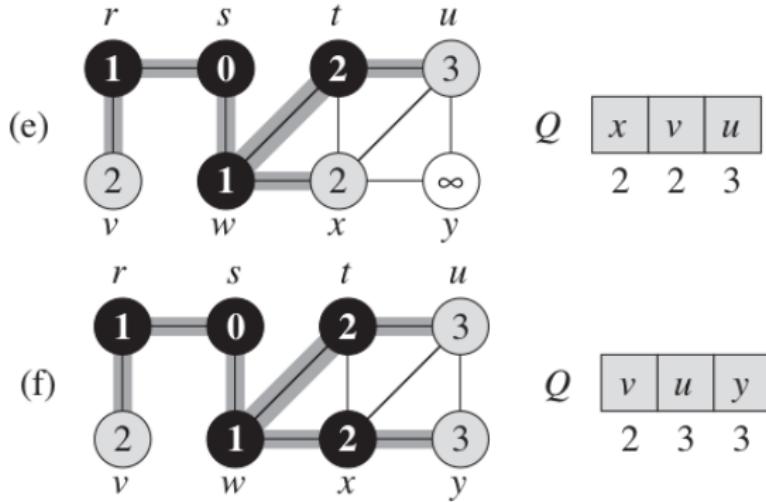
Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# Algorytm BFS - przykład



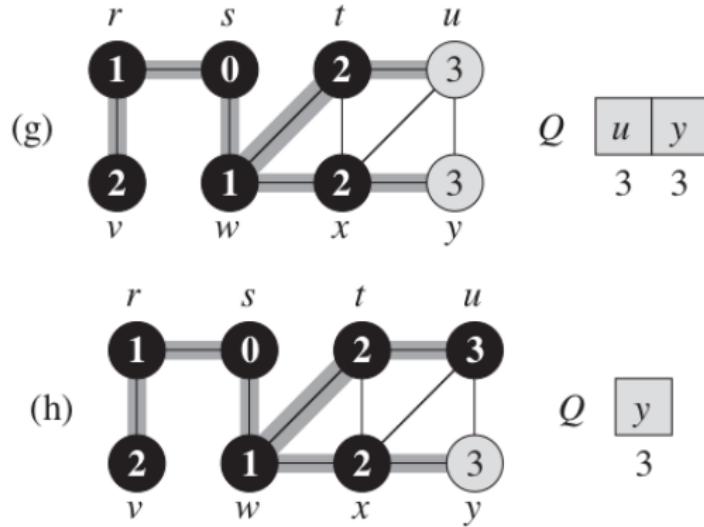
Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# Algorytm BFS - przykład



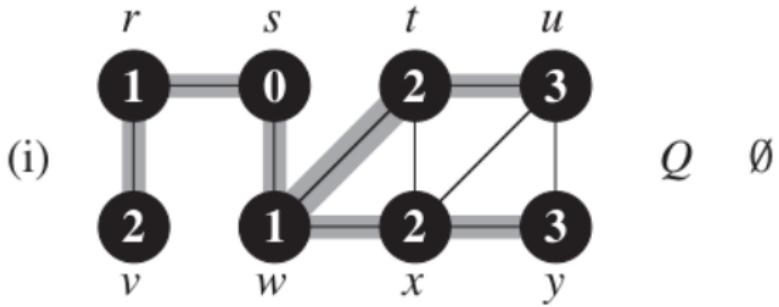
Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# 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

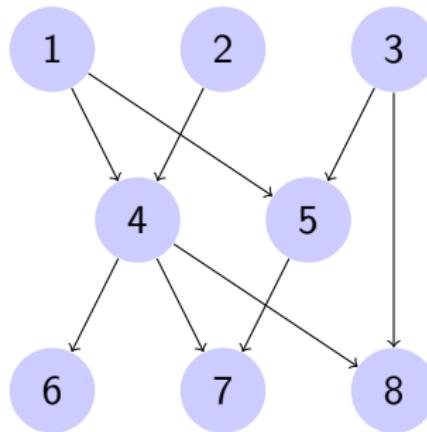


**Rysunek:** Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

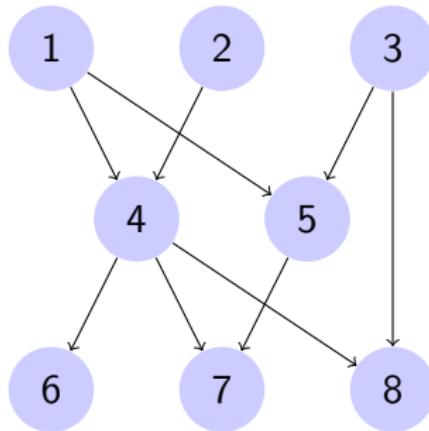
# BFS - reprezentacja macierzowa grafu

```
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 żywierzchoków
visited[a] = true; // rozpoczęń od wierzchołka a
displayVertex(a); // wyświetl wierzchołek
q.Enqueue(a); // wstaw na końcu
while (not q.empty()) { // do opróżnienia kolejki,
    b = q.Front(); // pobierz pierwszy wierzchołek
    q.DeQueue(); // usun go z kolejki
    // dopóki ma nie odwiedzonych sąsiadów
    c = getUnVisitedVertex(b, visited); // pobierz sąsiada
    while (c != -1)
    {
        visited[c] = true; // oznacz
        displayVertex(c); // wyświetl
        q.Enqueue(c); // wstaw do kolejki
        c = getUnVisitedVertex(b, visited); // pobierz sąsiada
    } // 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  $O(|E|)$ .
- graf reprezentowany jako macierz sąsiedztwa:  $\Theta(|V|^2)$ .

# Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

b.wozna@ujd.edu.pl

Wykład 5 i 6

# Spis treści

1 Sortowanie Topologiczne

2 Badanie acykliczności

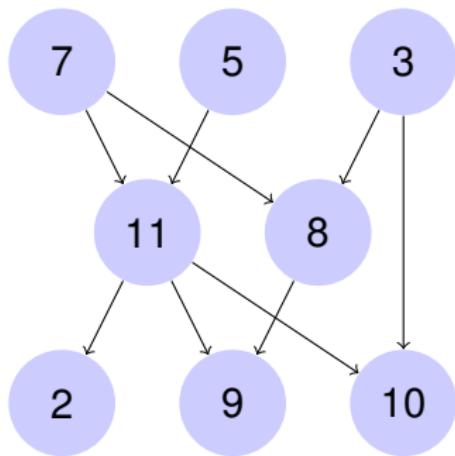
# Sortowanie Topologiczne - sformułowanie problemu

- Wejście: Acykliczny graf skierowany  $G = (V, E)$ , tzw. DAG (ang. *directed acyclic graph*).
- Wyjście: Liniowy porządek wierzchołków z  $V$  taki, że jeśli graf  $G$  zawiera krawędź  $(u, v)$ , to w tym porządku wierzchołek  $u$  występuje przed wierzchołkiem  $v$ .

# Sortowanie topologicznie - przykład

Wierzchołki w każdym grafie acyklicznym skierowanym można posortować topologicznie na jeden lub więcej sposobów:

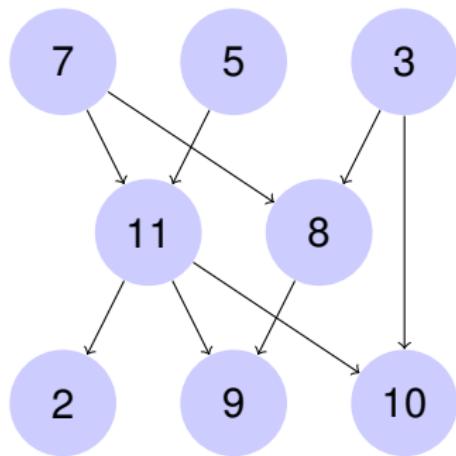
- 7,5,3,11,8,2,9,10



# Sortowanie topologicznie - przykład

Wierzchołki w każdym grafie acyklicznym skierowanym można posortować topologicznie na jeden lub więcej sposobów:

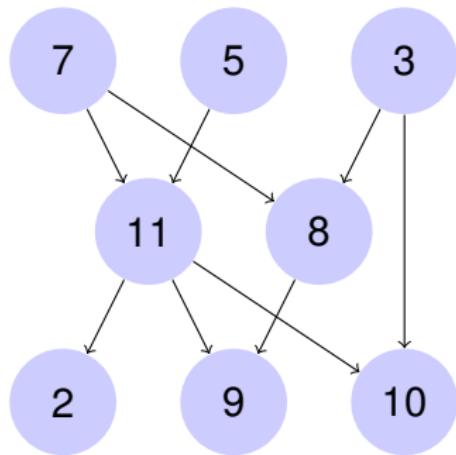
- 7,5,3,11,8,2,9,10
- 7,5,11,2,3,10,8,9



# Sortowanie topologicznie - przykład

Wierzchołki w każdym grafie acyklicznym skierowanym można posortować topologicznie na jeden lub więcej sposobów:

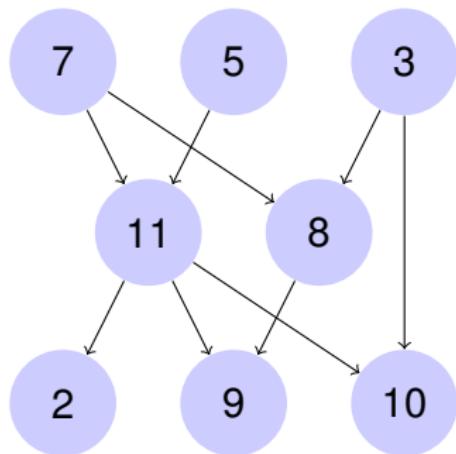
- 7,5,3,11,8,2,9,10
- 7,5,11,2,3,10,8,9
- 3,7,8,5,11,10,9,2



# Sortowanie topologicznie - przykład

Wierzchołki w każdym grafie acyklicznym skierowanym można posortować topologicznie na jeden lub więcej sposobów:

- 7,5,3,11,8,2,9,10
- 7,5,11,2,3,10,8,9
- 3,7,8,5,11,10,9,2
- 5,7,11,2,3,8,9,10



# Algorytm DFS - graf $G=(V,E)$ reprezentowany przez listy sąsiedztwa

$DFS(G = (V, E))$ :

```

1: for each vertex  $u \in V$  do
2:    $color[u] = WHITE$ 
3: end for
4:  $time = 0$ 
5: for each vertex  $u \in V$  do
6:   if  $color[u] == WHITE$  then
7:      $VISIT(G, u)$ 
8:   end if
9: end for
```

$VISIT(G, u)$

```

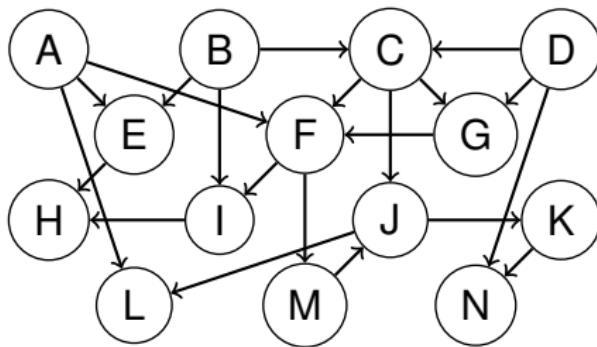
1:  $time = time + 1$ 
2:  $d[u] = time$ 
3:  $color[u] = GRAY$ 
4: for each  $v \in Adj[u]$  do
5:   if  $color[v] == WHITE$  then
6:      $VISIT(G, v)$ 
7:   end if
8: end for
9:  $color[u] = RED$ 
10:  $time = time + 1$ 
11:  $f[u] = time$ 
```

# Algorytm sortowania topologicznego bazujący na DFS

## TOPOLOGICAL-SORT( $G$ )

- Wykonaj algorytm  $DFS(G)$  na wejściowym DAG-u  $G = (V, E)$ , (reprezentowanym przez listy sąsiedztwa) w celu obliczenia czasów przetworzenia  $f[v]$  dla wszystkich wierzchołków  $v$ .
- Wypisz wierzchołki w porządku malejącym ze względu na ich “czas przetworzenia”, umieszczony w tablicy  $f$ .
- Złożoność:  $\Theta(|V| + |E|)$  – ponieważ DFS można wykonać w czasie  $\Theta(|V| + |E|)$ .

# Sortowanie topologiczne - przykład



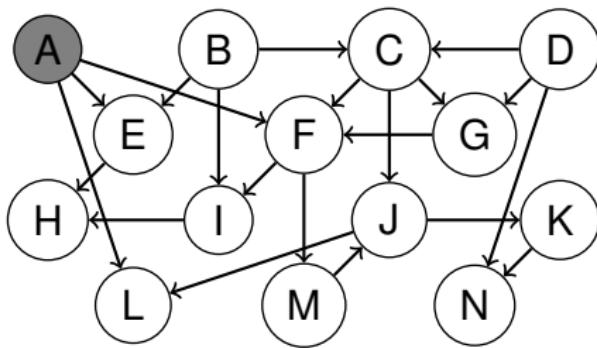
time = 0

A	B	C	D	E	F	G	H	I	J	K	L	M	N
W	W	W	W	W	W	W	W	W	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$d[v]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0

# Sortowanie topologiczne - przykład



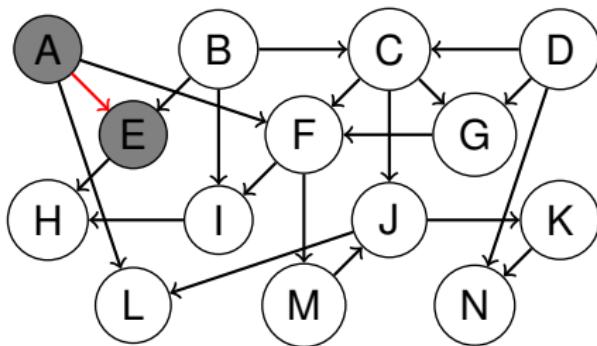
time = 1

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	W	W	W	W	W	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$d[v]$	1	0	0	0	0	0	0	0	0	0	0	0	0	0

# Sortowanie topologiczne - przykład



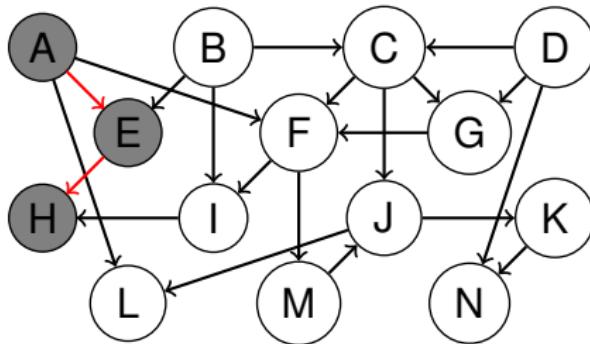
time = 2

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	G	W	W	W	W	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$d[v]$	1	0	0	0	2	0	0	0	0	0	0	0	0	0

# Sortowanie topologiczne - przykład



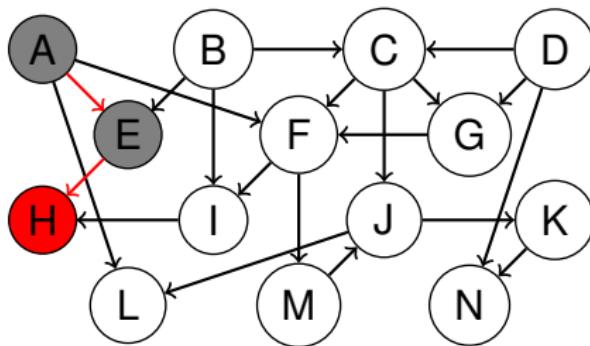
time = 3

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	G	W	W	G	W	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$d[v]$	1	0	0	0	2	0	0	3	0	0	0	0	0	0

# Sortowanie topologiczne - przykład



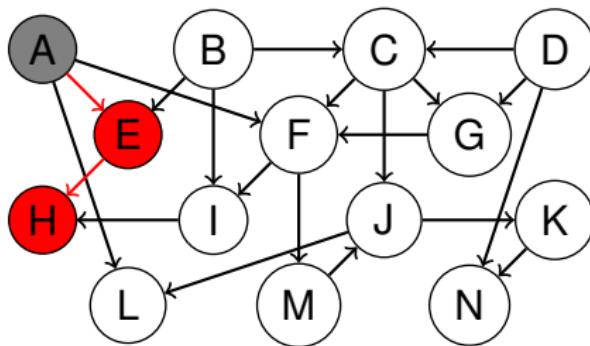
time = 4

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	G	W	W	R	W	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	0	0	0	4	0	0	0	0	0	0
$d[v]$	1	0	0	0	2	0	0	3	0	0	0	0	0	0

# Sortowanie topologiczne - przykład



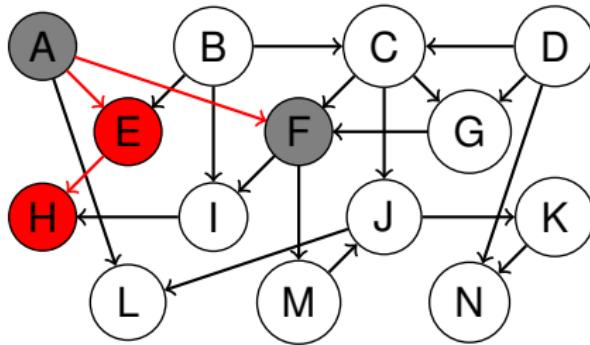
time = 5

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	W	W	R	W	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	0	0	0	0	0	0
$d[v]$	1	0	0	0	2	0	0	3	0	0	0	0	0	0

# Sortowanie topologiczne - przykład



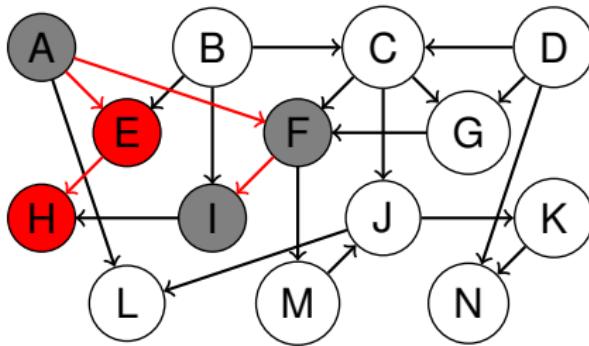
time = 6

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	W	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	0	0	0	0	0	0
$d[v]$	1	0	0	0	2	6	0	3	0	0	0	0	0	0

# Sortowanie topologiczne - przykład



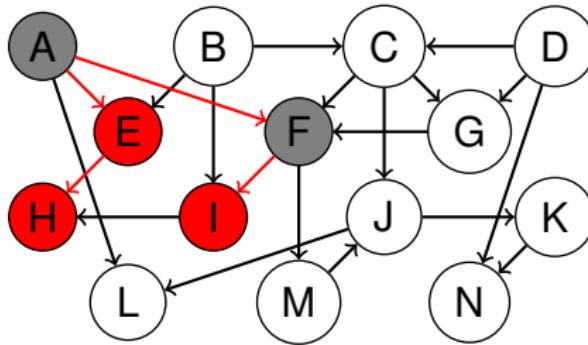
time = 7

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	G	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	0	0	0	0	0	0
$d[v]$	1	0	0	0	2	6	0	3	7	0	0	0	0	0

# Sortowanie topologiczne - przykład



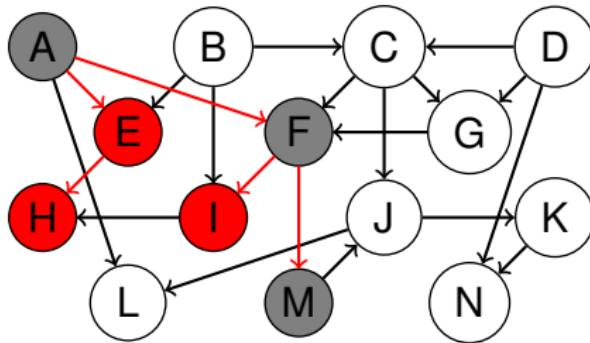
time = 8

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	W	W	W	W	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	0	0	0	0
$d[v]$	1	0	0	0	2	6	0	3	7	0	0	0	0	0

# Sortowanie topologiczne - przykład



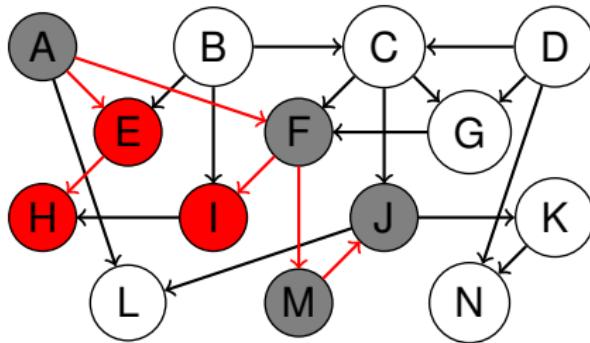
time = 9

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	W	W	W	G	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	0	0	0	0
$d[v]$	1	0	0	0	2	6	0	3	7	0	0	0	9	0

# Sortowanie topologiczne - przykład



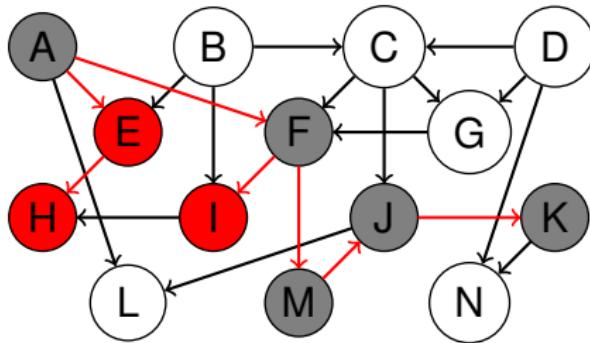
time = 10

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	G	W	W	G	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	0	0	0	0
$d[v]$	1	0	0	0	2	6	0	3	7	10	0	0	9	0

# Sortowanie topologiczne - przykład



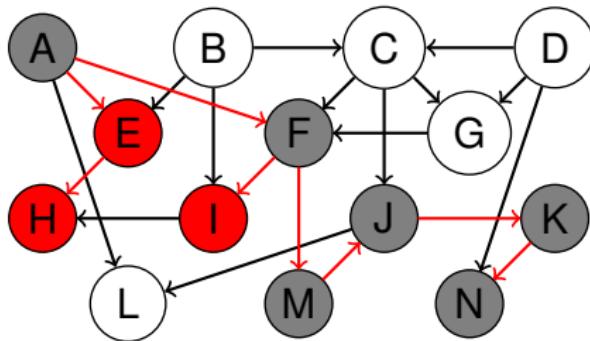
time = 11

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	G	G	W	G	W

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	0	0	0	0
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	0	9	0

# Sortowanie topologiczne - przykład



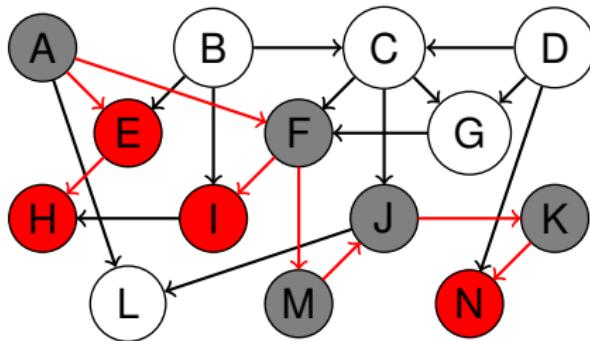
time = 12

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	G	G	W	G	G

Tabela: Parametr color. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	0	0	0	0
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	0	9	12

# Sortowanie topologiczne - przykład



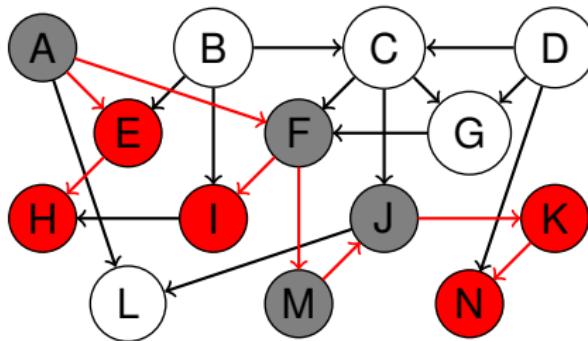
time = 13

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	G	G	W	G	R

Tabela: Parametr color. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	0	0	0	13
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	0	9	12

# Sortowanie topologiczne - przykład



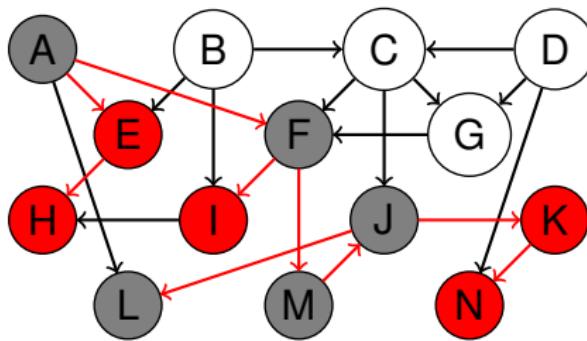
time = 14

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	G	R	W	G	R

Tabela: Parametr color. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	14	0	0	13
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	0	9	12

# Sortowanie topologiczne - przykład



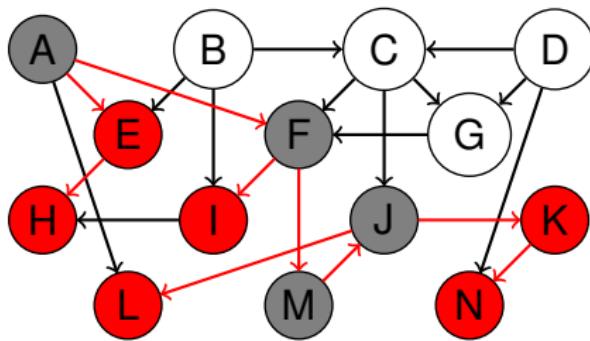
time = 15

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	G	R	G	G	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	14	0	0	13
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



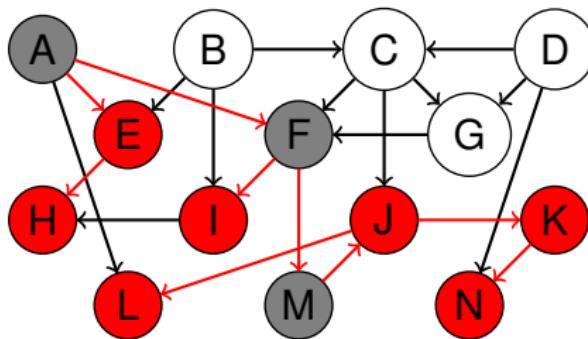
time = 16

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	G	R	R	G	R

Tabela: Parametr color. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	0	14	16	0	13
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



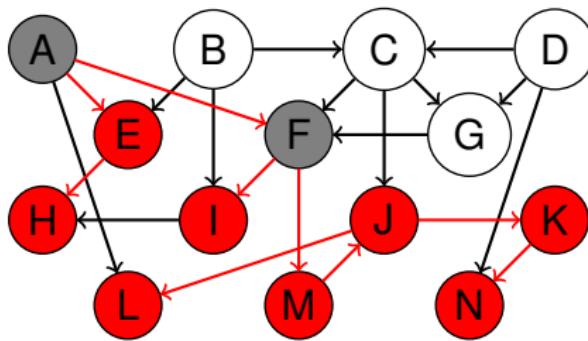
time = 17

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	R	R	R	G	R

Tabela: Parametr color. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	17	14	16	0	13
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



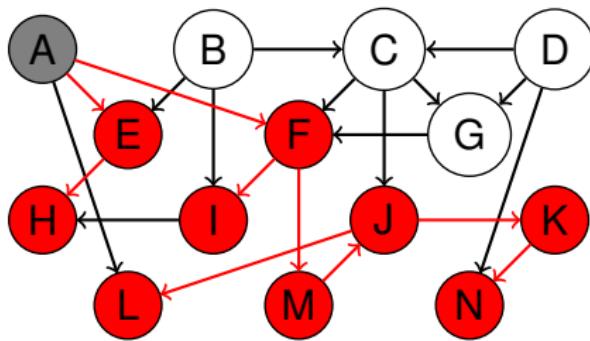
time = 18

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	G	W	R	R	R	R	R	R	R

Tabela: Parametr color. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	0	0	4	8	17	14	16	18	13
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



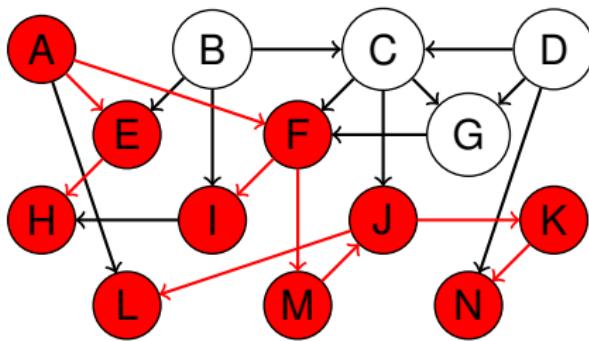
time = 19

A	B	C	D	E	F	G	H	I	J	K	L	M	N
G	W	W	W	R	R	W	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	0	0	0	0	5	19	0	4	8	17	14	16	18	13
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



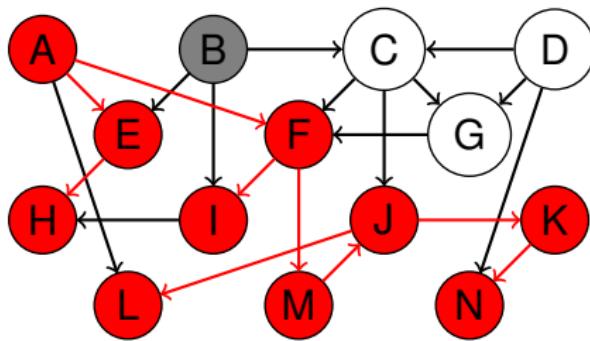
time = 20

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	W	W	W	R	R	W	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	0	0	0	5	19	0	4	8	17	14	16	18	13
$d[v]$	1	0	0	0	2	6	0	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



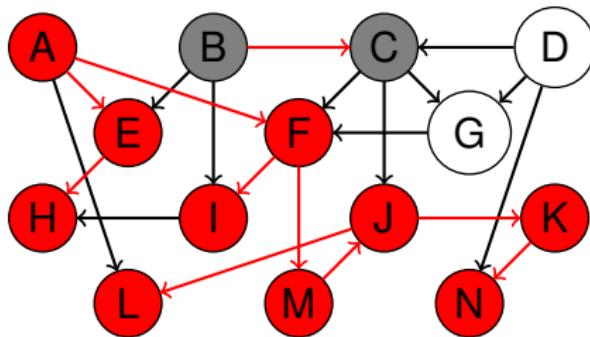
time = 21

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	G	W	W	R	R	W	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	0	0	0	5	19	0	4	8	17	14	16	18	13
$d[v]$	1	21	0	0	2	6	0	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



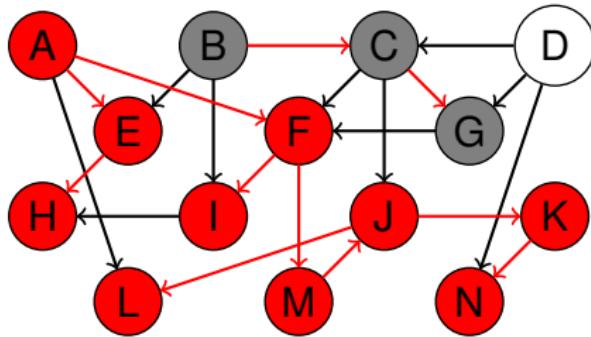
time = 22

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	G	G	W	R	R	W	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	0	0	0	5	19	0	4	8	17	14	16	18	13
$d[v]$	1	21	22	0	2	6	0	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



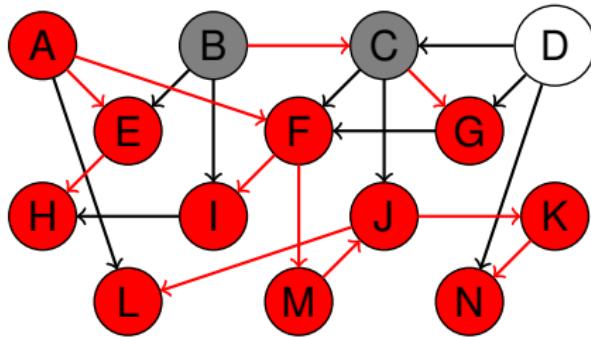
time = 23

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	G	G	W	R	R	G	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	0	0	0	5	19	0	4	8	17	14	16	18	13
$d[v]$	1	21	22	0	2	6	23	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



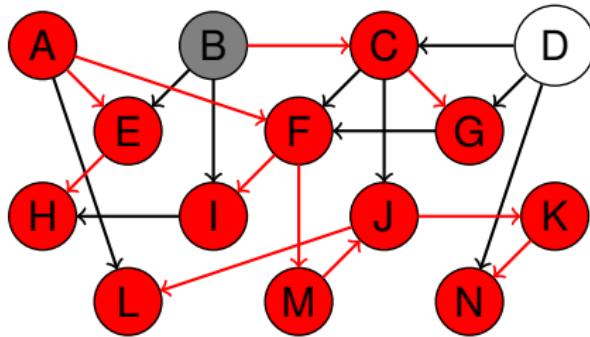
time = 24

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	G	G	W	R	R	R	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	0	0	0	5	19	24	4	8	17	14	16	18	13
$d[v]$	1	21	22	0	2	6	23	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



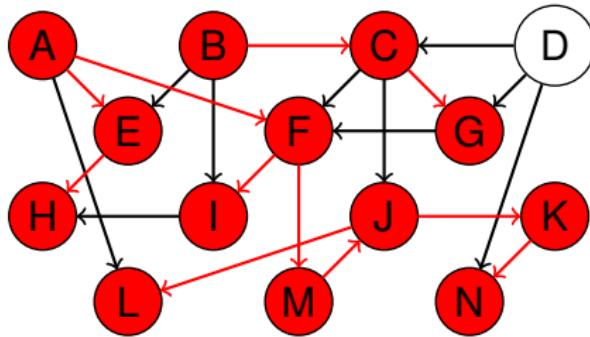
time = 25

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	G	R	W	R	R	R	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	0	25	0	5	19	24	4	8	17	14	16	18	13
$d[v]$	1	21	22	0	2	6	23	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



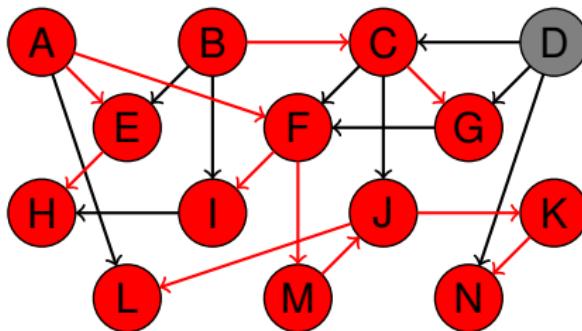
time = 26

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	R	R	W	R	R	R	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	26	25	0	5	19	24	4	8	17	14	16	18	13
$d[v]$	1	21	22	0	2	6	23	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



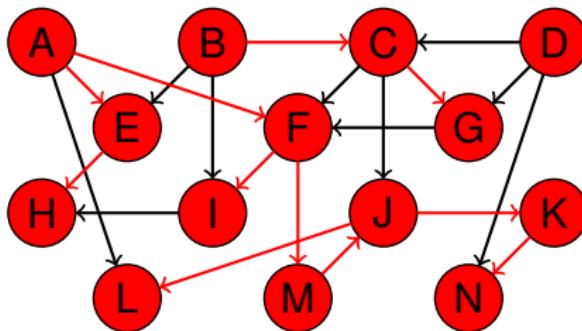
time = 27

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	R	R	G	R	R	R	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	26	25	0	5	19	24	4	8	17	14	16	18	13
$d[v]$	1	21	22	27	2	6	23	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład



time = 28

A	B	C	D	E	F	G	H	I	J	K	L	M	N
R	R	R	R	R	R	R	R	R	R	R	R	R	R

Tabela: Parametr `color`. W - oznacza WHITE, G - GRAY, R - RED

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
$f[v]$	20	26	25	28	5	19	24	4	8	17	14	16	18	13
$d[v]$	1	21	22	27	2	6	23	3	7	10	11	15	9	12

# Sortowanie topologiczne - przykład

Wynikowe sortowanie topologiczne:

D B C G A F M J L K N I E H

**Zauważ, że** wierzchołki do listy wchodzą w porządku malejącym ze względu na „końcowy czas przetwarzania” ich listy sąsiadów - tj. czas umieszczony w etykiecie  $f[u]$ :

A	B	C	D	E	F	G	H	I	J	K	L	M	N
20	26	25	28	5	19	24	4	8	17	14	16	18	13

# Sortowanie topologiczne. Reprezentacja macierzowa

## Badanie nieodwiedzonego wierzchołka

```
// zwraca nieodwiedzony wierzchołek przyległy do a
// zwraca -1, jeżeli takiego wierzchołka nie ma
int getUnVisitedVertex(Vertex a, BoolVector visited)
{
    for (b = 0; b < n; b=b+1) do
        if (edge[a][b] == true // jest krawędź
            and visited[b]==false) // b nie był odwiedzony
            then return b;
        endif
    endfor
    return -1;
}
```

# Sortowanie topologiczne. Reprezentacja macierzowa

```
topological_DFS(DAG G) {
    BoolVector visited = [n];
    //końcowy czas przetworzenia wierzchołków
    int fin[n];
    time = 0;
    for (k = 0; k < n; k=k+1) do
        visited[k] = false;
        fin[k] = 0;
    endfor
    for (k = 0; k < n; k=k+1) do
        if (visited[k] == false)
            G.topological_visit(k,time,visited,fin);
        endif
    endfor
    for (k = 0; k < n; k=k+1) do
        idx = max(fin,n); // wyszukuje maksymalny w tablicy
        displayVertex(idx);
        fin[idx] = -1;
    endfor
}
```

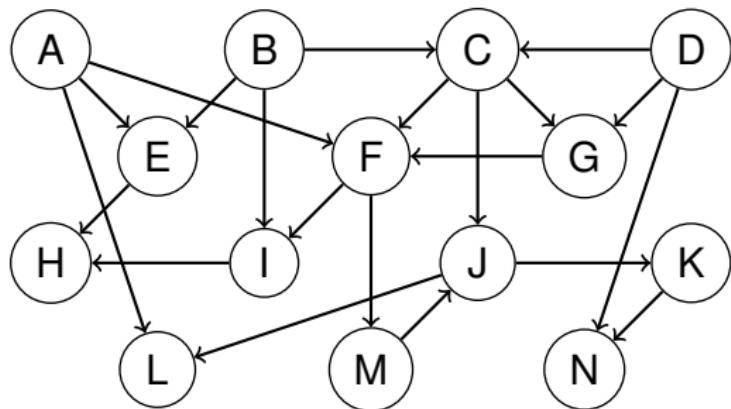
# Sortowanie topologiczne. Reprezentacja macierzowa

```
topological_visit(int a,int time,
                    BoolVector visited, int fin[])
{
    visited[a] = true;
    c = getUnVisitedVertex(a, visited);
    while (c != -1) do
        if (visited [c] == false)
            topological_visit(G,c,time,visited,fin);
        endif
        c = getUnVisitedVertex(a, visited);
    endwhile
    time = time+1;
    fin[a] = time;
}
```

# Działanie algorytmu - Przykład

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
A	0	0	0	0	1	1	0	0	0	0	0	1	0	0
B	0	0	1	0	1	0	0	0	1	0	0	0	0	0
C	0	0	0	0	0	1	1	0	0	1	0	0	0	0
D	0	0	1	0	0	0	1	0	0	0	0	0	0	1
E	0	0	0	0	0	0	0	1	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	1	0	0	0	1	0
G	0	0	0	0	0	1	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	1	0	0
K	0	0	0	0	0	0	0	0	0	0	0	0	0	1
L	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0	0	0	1	0	0	0	0
N	0	0	0	0	0	0	0	0	0	0	0	0	0	0

# Działanie algorytmu - Przykład I



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Nr. v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
visited	0	0	0	0	0	0	0	0	0	0	0	0	0	0
fin	0	0	0	0	0	0	0	0	0	0	0	0	0	0

# Działanie algorytmu - Przykład II

- 1  $time = 0;$
- 2  $visited[0] == 0 ? true;$
- 3  $topological\_visit(0, time, visited, fin)$
- 4  $visited[0] = 1;$
- 5  $c = getUnVisitedVertex(0, visited) = 4; // (0 \rightarrow 4)$
- 6  $visited[4] == 0 ? true;$
- 7  $topological\_visit(4, time, visited, fin);$
- 8  $visited[4] = 1;$
- 9  $c = getUnVisitedVertex(4, visited) = 7; // (4 \rightarrow 7)$
- 10  $visited[7] == 0 ? true;$
- 11  $topological\_visit(7, time, visited, fin);$
- 12  $visited[7] = 1;$

## Działanie algorytmu - Przykład III

- 13  $c = \text{getUnVisitedVertex}(7, \text{visited}) = -1;$
- 14  $\text{time} = \text{time} + 1 = 1; \text{fin}[7] = 1;$
- 15  $c = \text{getUnVisitedVertex}(4, \text{visited}) = -1$
- 16  $\text{time} = \text{time} + 1 = 2; \text{fin}[4] = 2;$
- 17  $c = \text{getUnVisitedVertex}(0, \text{visited}) = 5; // (0 \rightarrow 5)$
- 18  $\text{visited}[5] == 0 ? \text{true};$
- 19  $\text{topological\_visit}(5, \text{time}, \text{visited}, \text{fin});$
- 20  $\text{visited}[5] = 1;$
- 21  $c = \text{getUnVisitedVertex}(5, \text{visited}) = 8; // (5 \rightarrow 8)$
- 22  $\text{visited}[8] == 0 ? \text{true};$
- 23  $\text{topological\_visit}(8, \text{time}, \text{visited}, \text{fin});$
- 24  $\text{visited}[8] = 1;$

## Działanie algorytmu - Przykład IV

- 25  $c = \text{getUnVisitedVertex}(8, \text{visited}) = -1;$
- 26  $\text{time} = \text{time} + 1 = 3; \text{fin}[8] = 3;$
- 27  $c = \text{getUnVisitedVertex}(5, \text{visited}) = 12; // (5 \rightarrow 12)$
- 28  $\text{visited}[12] == 0 ? \text{true};$
- 29  $\text{topological\_visit}(12, \text{time}, \text{visited}, \text{fin});$
- 30  $\text{visited}[12] = 1;$
- 31  $c = \text{getUnVisitedVertex}(12, \text{visited}) = 9; // (12 \rightarrow 9)$
- 32  $\text{visited}[9] == 0 ? \text{true};$
- 33  $\text{topological\_visit}(9, \text{time}, \text{visited}, \text{fin});$
- 34  $\text{visited}[9] = 1;$
- 35  $c = \text{getUnVisitedVertex}(9, \text{visited}) = 10; // (9 \rightarrow 10)$
- 36  $\text{visited}[10] == 0 ? \text{true};$

# Działanie algorytmu - Przykład V

- 37 `topological_visit(10, time, visited, fin);`
- 38 `visited[10] = 1;`
- 39 `c = getUnVisitedVertex(10, visited) = 13; // (10->13)`
- 40 `visited[13] == 0 ? true;`
- 41 `topological_visit(13, time, visited, fin);`
- 42 `visited[13] = 1;`
- 43 `c = getUnVisitedVertex(13, visited) = -1;`
- 44 `time = time + 1 = 4; fin[13] = 4;`
- 45 `c = getUnVisitedVertex(10, visited) = -1;`
- 46 `time = time + 1 = 5; fin[10] = 5;`
- 47 `c = getUnVisitedVertex(9, visited) = 11; // (9->11)`
- 48 `visited[11] == 0 ? true;`

# Działanie algorytmu - Przykład VI

- 49 `topological_visit(11, time, visited, fin);`
- 50 `visited[11] = 1;`
- 51 `c = getUnVisitedVertex(11, visited) = -1;`
- 52 `time = time + 1 = 6; fin[11] = 6;`
- 53 `c = getUnVisitedVertex(9, visited) = -1;`
- 54 `time = time + 1 = 7; fin[9] = 7;`
- 55 `c = getUnVisitedVertex(12, visited) = -1;`
- 56 `time = time + 1 = 8; fin[12] = 8;`
- 57 `c = getUnVisitedVertex(5, visited) = -1;`
- 58 `time = time + 1 = 9; fin[5] = 9;`
- 59 `c = getUnVisitedVertex(0, visited) = -1;`
- 60 `time = time + 1 = 10; fin[0] = 10;`

# Działanie algorytmu - Przykład VII

- 61  $\text{visited}[1] == 0 ? \text{true};$
- 62 `topological_visit(1, time, visited, fin)`
- 63  $\text{visited}[1] = 1;$
- 64  $c = \text{getUnVisitedVertex}(1, \text{visited}) = 2; // (1 \rightarrow 2)$
- 65  $\text{visited}[2] == 0 ? \text{true};$
- 66 `topological_visit(2, time, visited, fin)`
- 67  $\text{visited}[2] = 1;$
- 68  $c = \text{getUnVisitedVertex}(2, \text{visited}) = 6; // (2 \rightarrow 6)$
- 69  $\text{visited}[6] == 0 ? \text{true};$
- 70 `topological_visit(6, time, visited, fin)`
- 71  $\text{visited}[6] = 1;$
- 72  $c = \text{getUnVisitedVertex}(6, \text{visited}) = -1;$

# Działanie algorytmu - Przykład VIII

- 73  $time = time + 1 = 11; fin[6] = 11;$
- 74  $c = getUnVisitedVertex(2, visited) = -1;$
- 75  $time = time + 1 = 12; fin[2] = 12;$
- 76  $c = getUnVisitedVertex(1, visited) = -1;$
- 77  $time = time + 1 = 13; fin[2] = 11;$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Nr. v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
fin	10	13	12	14	2	9	11	1	3	7	5	6	8	4

Wynikowe sortowanie topologiczne:

D B C G A F M J L K N I E H

# Sortowanie topologiczne - wersja nierekurencyjna

Metoda usuwania wierzchołków o stopniu wejściowym równym zero

- **Wykorzystywana własność:** jeśli graf jest acyklicznym grafem skierowanym, to posiada przynajmniej jeden wierzchołek o stopniu wejściowym równym zero.
- **Idea:** Dopóki graf posiada wierzchołki o stopniu wejściowym zero, znajdujemy taki wierzchołek, usuwamy go z grafu wraz ze wszystkimi wychodzącymi z niego krawędziami i umieszczamy go na liście wierzchołków posortowanych topologicznie.
- Jeśli w grafie pozostaną jakieś wierzchołki, to graf posiada cykle i sortowania topologicznego nie można wykonać.
- Złożoność:  $O(|V| + |E|)$

# Sortowanie topologiczne - Metoda usuwania wierzchołków I

- $G = (V, E)$  - acykliczny graf skierowany, tzw DAG
- $G$  - reprezentowany jest przez listy sąsiedztwa, tj.  $G.\text{Adj}(v)$  oznacza listę sąsiadów w wierzchołka  $v$  w grafie  $G$ .
- Rozmiar  $V$  jest  $n$ .
- Wierzchołki ponumerowane są od 0 do  $n - 1$ .
- $\text{in\_degree}(v)$  - stopień wejściowy wierzchołka  $v \in V$
- $\text{order}$  - kolejka, która będzie zawierać wynikowy porządek topologiczny, jeśli istnieje.

# Sortowanie topologiczne - Metoda usuwania wierzchołków II

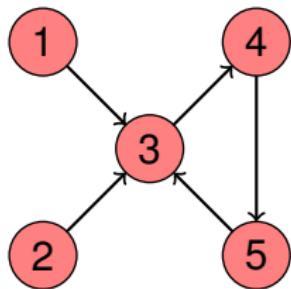
```
TopologicalSort (DAG G) {  
    Queue Q, Order;  
    count = 0;  
    int indegree [n];  
    for (v = 0; v < n; v = v+1)  
        indegree[v] = in_degree(v);  
    endfor  
    //wstaw do kolejki Q wszystkie wierzchołki  
    //ze stopniem wejściowym = 0  
    for (v = 0; v < n; v++)  
        if (indegree[v] == 0) then Q.Enqueue(v);  
    endfor  
    while (not Q.Empty()) do
```

# Sortowanie topologiczne - Metoda usuwania wierzchołków III

```
v = Q.DeQueue();
Order.Enqueue(v);
count = count + 1;
forall( w in G.Adj(v))
    indegree[w] = indegree[w]-1;
    if (indegree[w] == 0) then Q.Enqueue(w);
endforall
endwhile
// Istnieje cykl w grafie.
if (count != n) then Order = null;
else return Order;
}
```

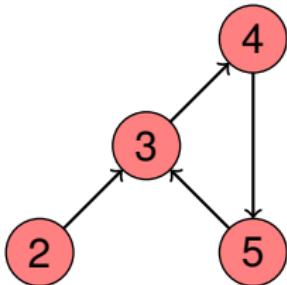
# Nierekurencyjne sortowanie topologiczne - przykład

- ➊  $order = \text{pusta}$ ;  $\text{count} = 0$
- ➋  $\text{indegree}[1] = 0, \text{indegree}[2] = 0,$   
 $\text{indegree}[3] = 3, \text{indegree}[4] = 1,$   
 $\text{indegree}[5] = 1.$
- ➌  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta



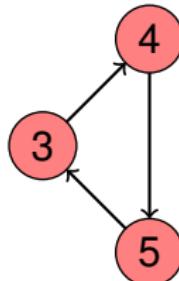
# Nierekurencyjne sortowanie topologiczne - przykład

- ➊  $order$  – pusta;  $count = 0$
- ➋  $indegree[1] = 0, indegree[2] = 0,$   
 $indegree[3] = 3, indegree[4] = 1,$   
 $indegree[5] = 1.$
- ➌  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta
  - ➊  $v = 1; Q = \{2\};$
  - ➋  $order = \{1\};$
  - ➌  $count = 1;$
  - ➍ Krawędź  $(1, 3); indegree[3] = 2;$



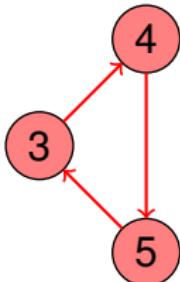
# Nierekurencyjne sortowanie topologiczne - przykład

- ➊  $order$  – pusta;  $count = 0$
- ➋  $indegree[1] = 0, indegree[2] = 0,$   
 $indegree[3] = 3, indegree[4] = 1,$   
 $indegree[5] = 1.$
- ➌  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta
  - ➊  $v = 1; Q = \{2\};$
  - ➋  $order = \{1\};$
  - ➌  $count = 1;$
  - ➍ Krawędź  $(1, 3); indegree[3] = 2;$
  - ➎  $v = 2; Q = \{\};$
  - ➏  $order = \{1, 2\};$
  - ➐  $count = 2;$
  - ➑ Krawędź  $(2, 3); indegree[3] = 1;$



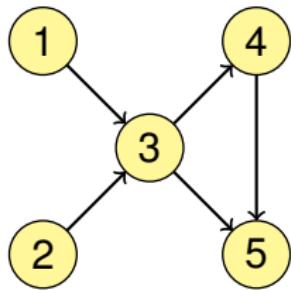
# Nierekurencyjne sortowanie topologiczne - przykład

- ➊  $order$  – pusta;  $count = 0$
- ➋  $indegree[1] = 0, indegree[2] = 0,$   
 $indegree[3] = 3, indegree[4] = 1,$   
 $indegree[5] = 1.$
- ➌  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta
  - ➊  $v = 1; Q = \{2\};$
  - ➋  $order = \{1\};$
  - ➌  $count = 1;$
  - ➍ Krawędź  $(1, 3); indegree[3] = 2;$
  - ➎  $v = 2; Q = \{\};$
  - ➏  $order = \{1, 2\};$
  - ➐  $count = 2;$
  - ➑ Krawędź  $(2, 3); indegree[3] = 1;$
- ➒  $Q = \{\}$  jest pusta. Wychodzimy w WHILE
- ➓  $count = 2, n = 5 \Rightarrow$  w grafie jest cykl.



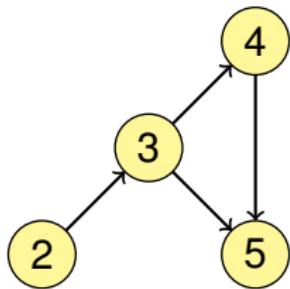
# Nierekurencyjne sortowanie topologiczne - przykład

- ➊ *order* – pusta; count = 0
- ➋ *indegree*[1] = 0, *indegree*[2] = 0,  
*indegree*[3] = 2, *indegree*[4] = 1,  
*indegree*[5] = 2.
- ➌  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta

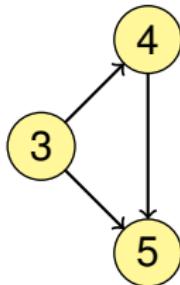


# Nierekurencyjne sortowanie topologiczne - przykład

- ① *order* – pusta; *count* = 0
- ②  $\text{indegree}[1] = 0, \text{indegree}[2] = 0,$   
 $\text{indegree}[3] = 2, \text{indegree}[4] = 1,$   
 $\text{indegree}[5] = 2.$
- ③  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta
  - ①  $v = 1; Q = \{2\}; \text{order} = \{1\}; \text{count} = 1;$
  - ② Krawędź (1, 3);  $\text{indegree}[3] = 1;$



# Nierekurencyjne sortowanie topologiczne - przykład



- ➊ *order* – pusta; *count* = 0
- ➋ *indegree*[1] = 0, *indegree*[2] = 0,  
*indegree*[3] = 2, *indegree*[4] = 1,  
*indegree*[5] = 2.
- ➌  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta
  - ➊  $v = 1; Q = \{2\}; order = \{1\}; count = 1;$
  - ➋ Krawędź (1, 3); *indegree*[3] = 1;
  - ➌  $v = 2; Q = \{\}; order = \{1, 2\}; count = 2;$
  - ➍ Krawędź (2, 3); *indegree*[3] = 0;  $Q = \{3\}$ ;

# Nierekurencyjne sortowanie topologiczne - przykład



- ➊ *order* – pusta; *count* = 0
- ➋ *indegree*[1] = 0, *indegree*[2] = 0,  
*indegree*[3] = 2, *indegree*[4] = 1,  
*indegree*[5] = 2.
- ➌  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta
  - ➍  $v = 1; Q = \{2\}; order = \{1\}; count = 1;$
  - ➎ Krawędź (1, 3); *indegree*[3] = 1;
  - ➏  $v = 2; Q = \{\}; order = \{1, 2\}; count = 2;$
  - ➐ Krawędź (2, 3); *indegree*[3] = 0;  $Q = \{3\};$
  - ➑  $v = 3; Q = \{\}; order = \{1, 2, 3\}; count = 3;$
  - ➒ Krawędź (3, 4); *indegree*[4] = 0;  $Q = \{4\};$
  - ➓ Krawędź (3, 5); *indegree*[5] = 1;  $Q = \{4\};$

# Nierekurencyjne sortowanie topologiczne - przykład

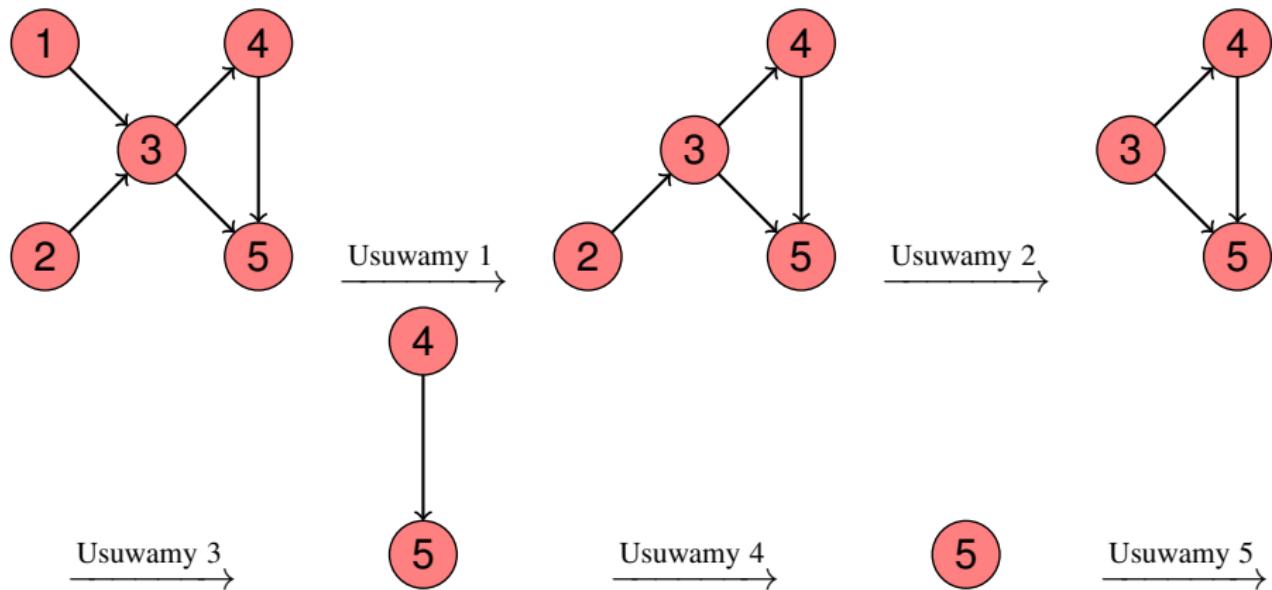
- ①  $order = \text{pusta}$ ;  $count = 0$
- ②  $indegree[1] = 0, indegree[2] = 0,$   
 $indegree[3] = 2, indegree[4] = 1,$   
 $indegree[5] = 2.$
- ③  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta
  - ①  $v = 1; Q = \{2\}; order = \{1\}; count = 1;$
  - ② Krawędź  $(1, 3); indegree[3] = 1;$
  - ③  $v = 2; Q = \{\}; order = \{1, 2\}; count = 2;$
  - ④ Krawędź  $(2, 3); indegree[3] = 0; Q = \{3\};$
  - ⑤  $v = 3; Q = \{\}; order = \{1, 2, 3\}; count = 3;$
  - ⑥ Krawędź  $(3, 4); indegree[4] = 0; Q = \{4\};$
  - ⑦ Krawędź  $(3, 5); indegree[5] = 1; Q = \{4\};$
  - ⑧  $v = 4; Q = \{\}; order = \{1, 2, 3, 4\}; count = 4;$
  - ⑨ Krawędź  $(4, 5); indegree[5] = 0; Q = \{5\};$

5

# Nierekurencyjne sortowanie topologiczne - przykład

- ➊ *order* – pusta; *count* = 0
- ➋ *indegree*[1] = 0, *indegree*[2] = 0,  
*indegree*[3] = 2, *indegree*[4] = 1,  
*indegree*[5] = 2.
- ➌  $Q = \{1, 2\}$  // Dopóki kolejka nie jest pusta
  - ➍  $v = 1; Q = \{2\}; order = \{1\}; count = 1;$
  - ➎ Krawędź (1, 3); *indegree*[3] = 1;
  - ➏  $v = 2; Q = \{\}; order = \{1, 2\}; count = 2;$
  - ➐ Krawędź (2, 3); *indegree*[3] = 0;  $Q = \{3\};$
  - ➑  $v = 3; Q = \{\}; order = \{1, 2, 3\}; count = 3;$
  - ➒ Krawędź (3, 4); *indegree*[4] = 0;  $Q = \{4\};$
  - ➓ Krawędź (3, 5); *indegree*[5] = 1;  $Q = \{4\};$
  - ➔  $v = 4; Q = \{\}; order = \{1, 2, 3, 4\}; count = 4;$
  - ➕ Krawędź (4, 5); *indegree*[5] = 0;  $Q = \{5\};$
  - ➖  $v = 5; Q = \{\}; order = \{1, 2, 3, 4, 5\}; count = 5;$
- ➍  $count = n //$   
TRUE
- ➎ Porządek topologiczny:  
1,2,3,4,5.

# Nierekurencyjne sortowanie topologiczne - przykład



Sortowanie topologiczne: 1, 2, 3, 4, 5

# Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

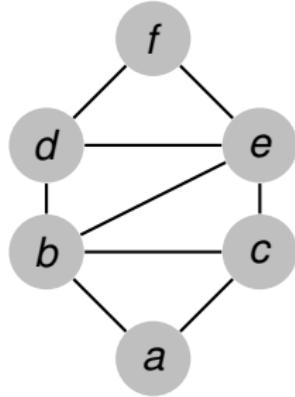
b.wozna@ujd.edu.pl

Wykład 7 i 8

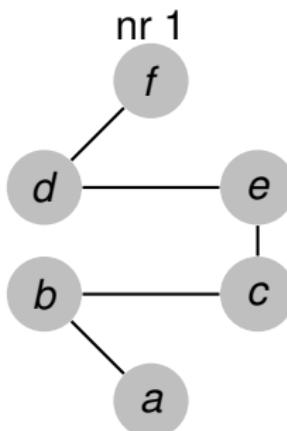
# Drzewo rozpinające grafu I

**Drzewem rozpinającym** grafu  $G$  nazywamy spójny i acykliczny podgraf grafu  $G$  zawierający wszystkie jego wierzchołki.  
Dany graf może posiadać wiele różnych drzew rozpinających.

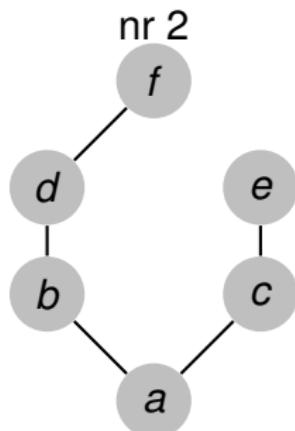
Graf wejściowy



Drzewo rozpinające nr 1



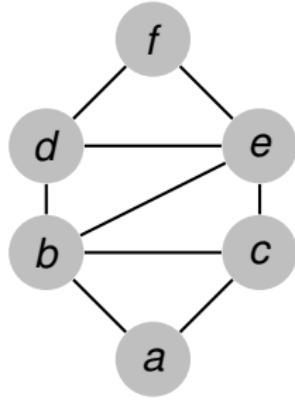
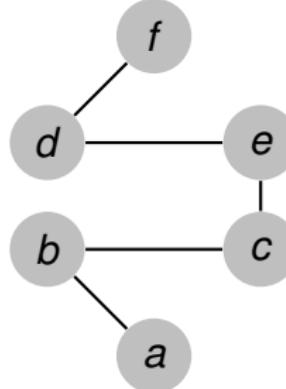
Drzewo rozpinające nr 2



# Drzewo rozpinające grafu II

- Drzewo rozpinające powstaje poprzez usunięcie z grafu krawędzi tworzących cykl.
- Drzewo rozpinające można utworzyć przy pomocy algorytmu DFS.

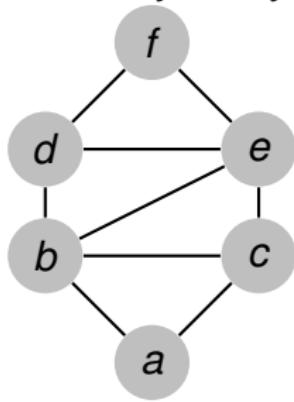
Graf wejściowy

Drzewo rozpinające nr 1.  
Wierzchołek początkowy a.

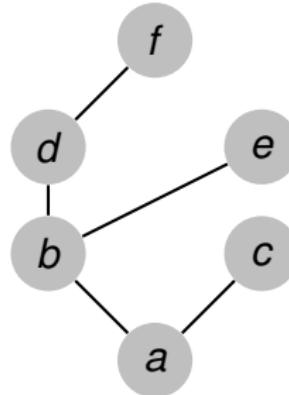
# Drzewo rozpinające grafu III

- Drzewo rozpinające można utworzyć przy pomocy algorytmu BFS.

Graf wejściowy



Drzewo rozpinające nr 2.  
Wierzchołek początkowy a.



# Etykietowany graf skierowany

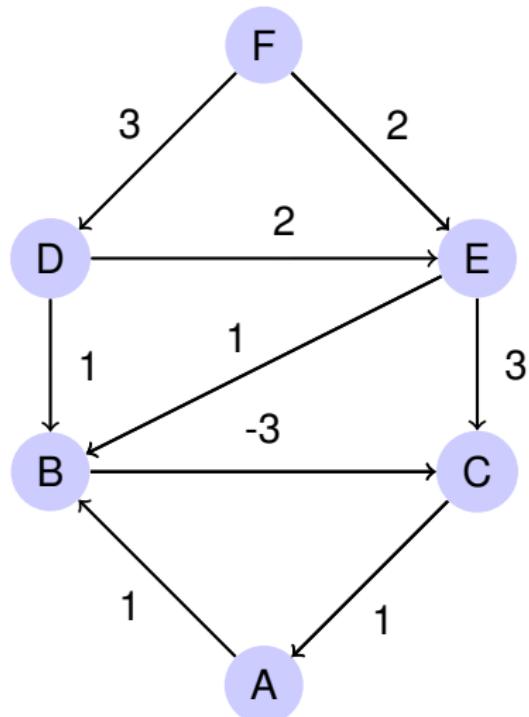
## Definicja

**Etykietowanym grafem skierowanym** nazywamy strukturę

$G = (V, E, w : E \rightarrow R)$  gdzie

- $V$  to zbiór wierzchołków,
- $E \subseteq \{(u, v) : u, v \in V\}$  to zbiór uporządkowanych par wierzchołków ze zbioru  $V$ , zwanych krawędziami.
- $w : E \rightarrow R$  jest funkcją **wagi**; wagi reprezentują pewne wielkości (np. długość drogi).

# Etykietowany graf skierowany - przykład



Macierz sąsiedztwa:

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	-3	0	0	0
C	1	0	0	0	0	0
D	0	1	0	0	2	0
E	0	1	3	0	0	0
F	0	0	0	3	2	0

# Etykietowany graf nieskierowany

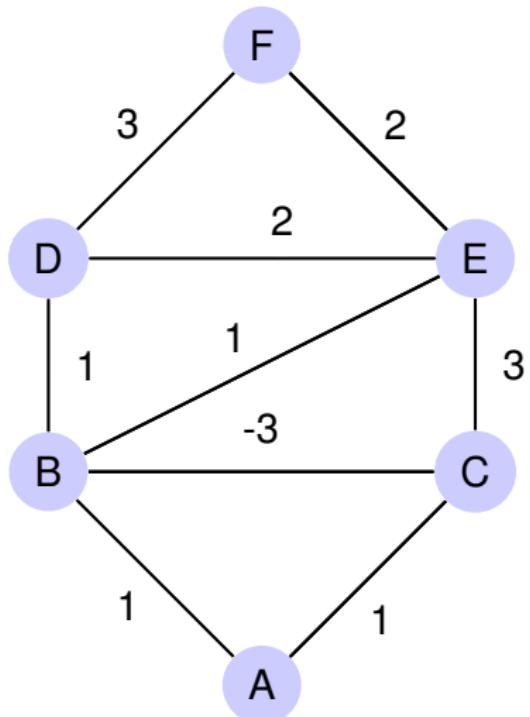
## Definicja

**Etykietowanym grafem nieskierowanym** nazywamy strukturę

$G = (V, E, w : E \rightarrow R)$  gdzie

- $V$  to zbiór wierzchołków,
- $E \subseteq \{\{u, v\} : u, v \in V\}$  to zbiór par wierzchołków ze zbioru  $V$ , zwanych krawędziami.
- $w : E \rightarrow R$  jest funkcją **wagi**; wagi reprezentują pewne wielkości (np. długość drogi).

# Etykietowany graf skierowany - przykład



Macierz sąsiedztwa:

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	1	0	-3	1	1	0
C	1	-3	0	0	3	0
D	0	1	0	0	2	3
E	0	1	3	2	0	2
F	0	0	0	3	2	0

# Struktury danych dla zbiorów rozłącznych I

- Niektóre realizacje algorytmów wymagają grupowania  $n$  różnych elementów w pewiną liczbę **zbiór rozłączny**.
- Dwie podstawowe operacje do wykonania na tach zbiorach to:
  - znajdowanie zbioru zawierającego dany element,
  - łączenia dwóch zbiorów.
- **Struktura danych dla zbiorów rozłącznych** umożliwia zarządzanie rodziną  $\mathbb{S} = \{S_1, S_2, \dots, S_n\}$  rozłącznych zbiorów dynamicznych.
- W takiej strukturze każdy zbiór jest identyfikowany przez **reprezentanta**, którym jest pewnym elementem tego zbioru.
- Założmy, że każdy element zbioru jest reprezentowany przez pewien obiekt, oznaczony jako  $x$ .
- Struktura danych dla zbiorów rozłącznych powinna wspierać następujące operacje:

# Struktury danych dla zbiorów rozłącznych II

- **MakeSet( $x$ )** - tworzy nowy zbiór, którego jedynym elementem (reprezentantem) jest  $x$ . Ponieważ zbiory mają być rozłączne,  $x$  nie może być elementem innego zbioru.
- **Union( $x, y$ )** - łączy dwa rozłączne zbiory dynamiczne zawierające odpowiednio  $x$  i  $y$ , powiedzmy  $S_x$  i  $S_y$  w nowy zbiór  $S_{x \cup y}$  będący ich sumą. Reprezentantem otrzymanego zbioru  $S_{x \cup y}$  może być dowolny element z  $S_x \cup S_y$ . Ponieważ zbiory w rodzinie mają być rozłączne, to „niszczymy” zbiory  $S_x$  i  $S_y$ , usuwając je z rodziny  $\mathbb{S}$ , a w ich miejsce dodajemy zbiór  $S_{x \cup y}$ .
- **FindSet( $x$ )** - zwraca wskaźnik (adres) do reprezentanta zbioru zawierającego  $x$ .

## Zastosowania struktur danych dla zbiorów rozłącznych I

- Jednym z wielu zastosowań struktur danych dla zbiorów rozłącznych jest **wyznaczanie spójnych składowych w grafie nieskierowanym.**
- Algorytm CONNECTED-COMPONENTS służy do obliczenia spójnych składowych grafu i wykorzystuje operacje na zbiorach rozłącznych.
- **Uwaga !** Gdy krawędzie grafu są statyczne – nie zmieniają się w czasie – wyznaczanie spójnych składowych w grafie nieskierowanym można wykonać szybciej, korzystając z algorytmu wyszukiwania w głąb.

## Zastosowania struktur danych dla zbiorów rozłącznych II

- Jeżeli jednak krawędzie są dodawane dynamicznie i konieczne jest zachowanie złączonych komponentów po dodaniu każdej krawędzi, to zastosowanie algorytmu CONNECTED-COMPONENTS może być znacznie bardziej wydajne, niż uruchamianie DFS dla każdej nowo dodanej krawędzi.

## Zastosowania struktur danych dla zbiorów rozłącznych III

### Algorytm CONNECTED-COMPONENTS

CONNECTED-COMPONENTS( $G=(V,E)$ )

```
1: for każdy wierzchołek  $v \in V$  do
2:    $MakeSet(v)$ 
3: end for
4: for każda krawędź  $(u, v) \in E$  do
5:   if  $FindSet(u) \neq FindSet(v)$  then
6:      $Union(u, v)$ 
7:   end if
8: end for
```

- Po przetworzeniu wszystkich krawędzi, dwa wierzchołki należą do tej samej składowej grafu wtedy i tylko wtedy, gdy odpowiadające im obiekty znajdują się w tym samym zbiorze.

## Zastosowania struktur danych dla zbiorów rozłącznych IV

- Po wykonaniu procedury CONNECTED-COMPONENTS, procedura SAME-COMPONENT odpowiada na pytanie, czy dwa wierzchołki należą do tej samej spójnej składowej.

### Algorytm SAME-COMPONENT

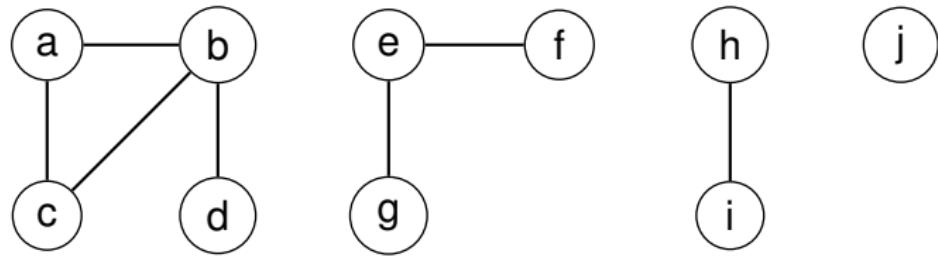
SAME-COMPONENT( $u, v$ )

```
1: if FindSet( $u$ ) == FindSet( $v$ ) then
2:   return true
3: else
4:   return false
5: end if
```

## Zastosowania struktur danych dla zbiorów rozłącznych V

Przykład: graf o czterech spójnych składowych

$\{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\}$



## Zastosowania struktur danych dla zbiorów rozłącznych VI

Rodzina zbiorów rozłącznych po przetworzeniu każdej krawędzi

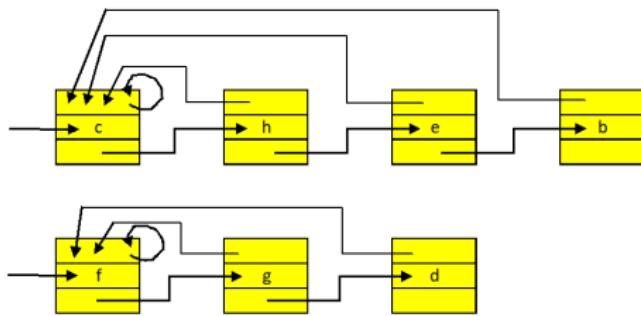
Przetworzone krawędzie	Rodzina zbiorów rozłącznych
Zbiory początkowe	$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}$
$\{b, d\}$	$\{a\}, \{b, d\}, \{c\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}$
$\{e, g\}$	$\{a\}, \{b, d\}, \{c\}, \{e, g\}, \{f\}, \{h\}, \{i\}, \{j\}$
$\{a, c\}$	$\{a, c\}, \{b, d\}, \{e, g\}, \{f\}, \{h\}, \{i\}, \{j\}$
$\{h, i\}$	$\{a, c\}, \{b, d\}, \{e, g\}, \{f\}, \{h, i\}, \{j\}$
$\{a, b\}$	$\{a, c, b, d\}, \{e, g\}, \{f\}, \{h, i\}, \{j\}$
$\{e, f\}$	$\{a, c, b, d\}, \{e, g, f\}, \{h, i\}, \{j\}$
$\{b, c\}$	$\{a, c, b, d\}, \{e, g, f\}, \{h, i\}, \{j\}$

# Listowa reprezentacja zbiorów rozłącznych I

- Każdy zbiór jest reprezentowany za pomocą listy.
- Pierwszy element na każdej liście służy jako reprezentant swojego zbioru.
- Każdy obiekt na liście składa się z elementu zbioru, wskaźnika do obiektu zawierającego następny element zbioru oraz wskaźnika do reprezentanta.

## Listowa reprezentacja zbiorów rozłącznych II

Reprezentacja dwóch zbiorów rozłącznych:  $S_1 = \{c, h, e, b\}$  oraz  $S_2 = \{f, g, d\}$ .



- W reprezentacji listowej, wykonanie procedury *MakeSet* wymaga czasu  $O(1)$ .
  - Aby wykonać  $\text{MakeSet}(x)$ , wystarczy utworzyć nową listę, której jedynym elementem jest  $x$ .

## Listowa reprezentacja zbiorów rozłącznych III

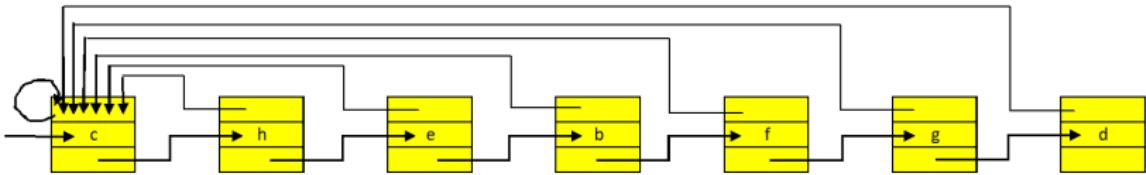
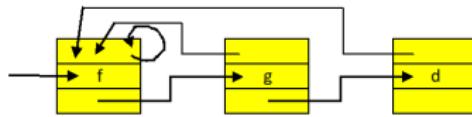
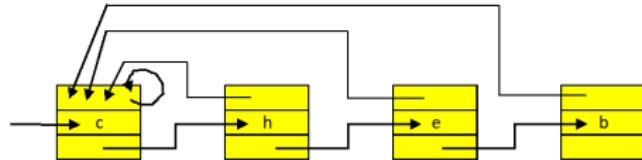
- W reprezentacji listowej, wykonanie procedury *FindSet* wymaga czasu  $O(1)$ .
  - $FindSet(x)$  zwraca wskaźnik od  $x$  do reprezentanta zbioru.
- Najprostsza implementacja operacji UNION przy użyciu reprezentacji listy zajmuje znacznie więcej czasu niż MAKE-SET lub FIND-SET.
- Operację  $Union(x, y)$  wykonujemy dołączając listę  $y$  na końcu listy  $x$ . Reprezentant listy  $x$  staje się reprezentantem zbioru wynikowego. Można użyć wskaźnika ogona dla listy  $x$ , aby szybko znaleźć miejsce dołączenia listy  $y$ . Ponieważ jednak wszystkie elementy listy  $y$  zostają dołączone do listy  $x$ , musimy zaktualizować wskaźnik do reprezentanta zbioru dla każdego obiektu pierwotnie znajdującego się na liście  $y$ , co zajmuje czas liniowy w stosunku do długości listy  $y$ .

# Listowa reprezentacja zbiorów rozłącznych IV

- Przykład:

# Listowa reprezentacja zbiorów rozłącznych V

Wynik wykonania:  $\text{Union}(e, g)$ . Reprezentantem zbioru wynikowego jest  $c$ .



# Drzewa rozpinające o minimalnej wadze

- Jeżeli mamy do czynienia z grafem z funkcją wagą, to najczęściej interesuje nas znalezienie **drzewa rozpinającego o minimalnej wadze**, tzn., drzewa z najmniejszą sumą wag jego krawędzi.
- Aby znaleźć drzewo o żądanych własnościach można zastosować dwa algorytmy:
  - Kruskala
  - Prima

# Algorytmy Kruskala

*Algorytm jest oparty o **metodę zachłanną** i polega na łączeniu wielu poddrzew w jedno za pomocą krawędzi o najmniejszej wadze.*

Założenia:

- Zastosowanie struktury danych reprezentującej zbiory rozłączne do pamiętania kilku rozłącznych zbiorów wierzchołków.
- FIND-SET( $u$ ) zwraca reprezentanta zbioru zawierającego wierzchołek  $u$ .
- UNION( $u, v$ ) - łączy drzewa zawierające  $u$  i  $v$  w jedno drzewo.
- **Wejście:** Spójny graf nieskierowany z funkcją wagą  
 $G = (V, E, w : E \mapsto R)$ .

# Algorytm Kruskala

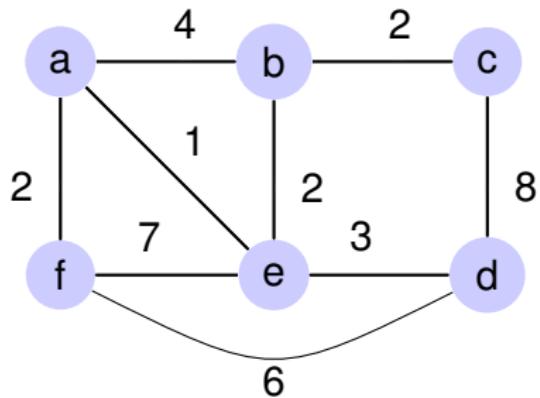
**Require:**  $KRUSKAL(G)$

- 1:  $A = \emptyset$
- 2: **for** każdy wierzchołek  $v \in V[G]$  **do**
- 3:    $Make - Set(v)$
- 4: **end for** {Utworzenie  $|V|$  drzew jednowierzchołkowych}
- 5: posortuj krawędzie z  $E$  niemalejąco względem wag.
- 6: **for** każda krawędź  $(u, v) \in E$ , w kolejności niemalejących wag **do**
- 7:   **if**  $FIND - SET(u) \neq FIND - SET(v)$  **then**
- 8:      $A = A \cup \{(u, v)\}$
- 9:      $Union(u, v)$
- 10:   **end if**
- 11: **end for**
- 12: **return**  $A$

# Algorytm Kruskala - Złożoność obliczeniowa

- Algorytm można podzielić na dwa etapy:
  - w pierwszym etapie sortujemy krawędzie według wag w czasie  $O(|E| \cdot \log(|E|))$ .
  - w drugim etapie budujemy rozpięte drzewo poprzez wybór najkrótszych krawędzi ze zbioru krawędzi  $E$ ; ten etap można wykonać w czasie  $O(|E| \cdot \log(|V|))$ .
- Sumaryczny czas pracy algorytmu Kruskala wynosi:  
 $O(|E| \cdot \log(|V|))$

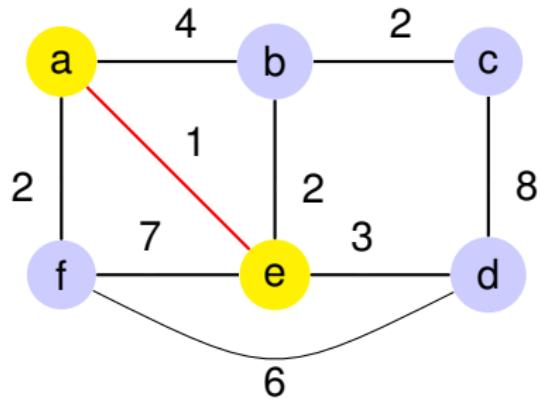
# Algorytm Kruskala - Przykład



- Po posortowaniu krawędzi wg. wag otrzymujemy:  
 $ae=1$  ,  $af=2$  ,  $bc=2$  ,  $be=2$  ,  
 $de=3$  ,  $ab=4$  ,  $fd=6$  ,  $ef=7$  ,  
 $cd=8$

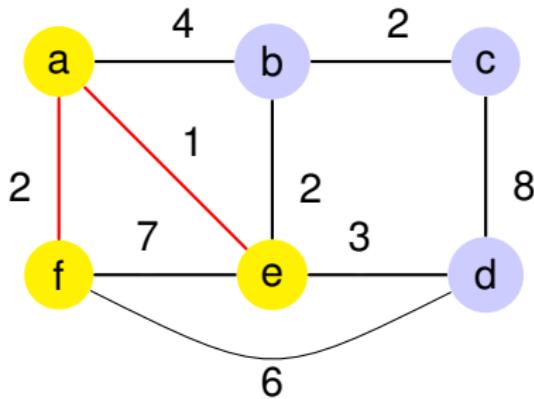
# Algorytm Kruskala - Przykład

Krok 1.



$ae=1$ ,  $af=2$ ,  $bc=2$ ,  $be=2$ ,  $de=3$ ,  
 $ab=4$ ,  $fd=6$ ,  $ef=7$ ,  $cd=8$

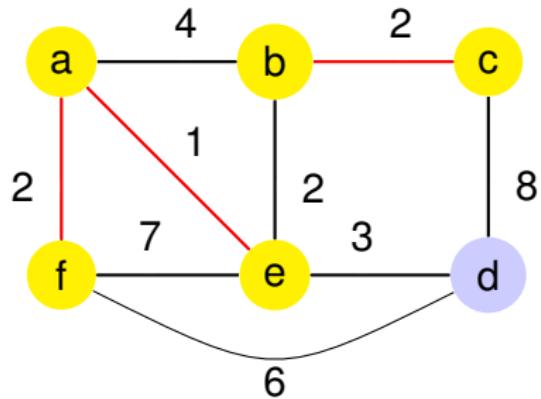
Krok 2.



$ae=1$ ,  $af=2$ ,  $bc=2$ ,  $be=2$ ,  $de=3$ ,  
 $ab=4$ ,  $fd=6$ ,  $ef=7$ ,  $cd=8$

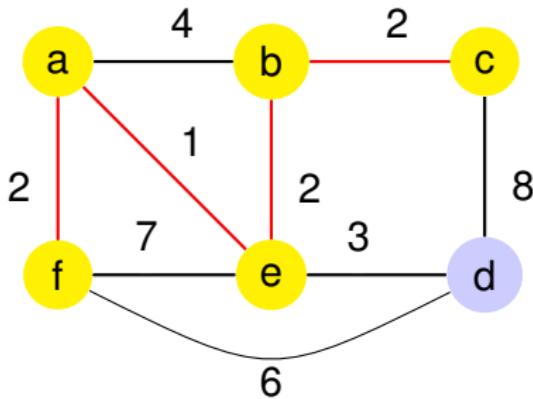
# Algorytm Kruskala - Przykład

Krok 3.



$ae=1$ ,  $af=2$ ,  $bc=2$ ,  $be=2$ ,  $de=3$ ,  
 $ab=4$ ,  $fd=6$ ,  $ef=7$ ,  $cd=8$

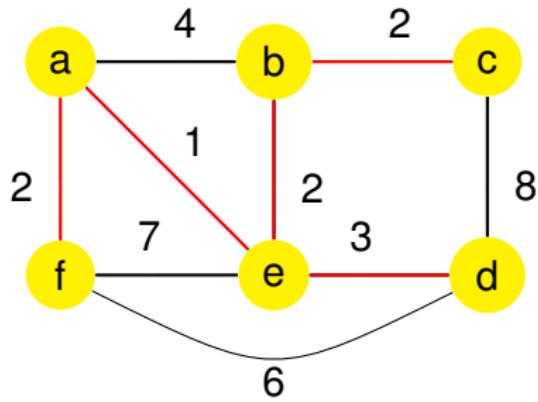
Krok 4 - scalenie.



$ae=1$ ,  $af=2$ ,  $bc=2$ ,  $be=2$ ,  $de=3$ ,  
 $ab=4$ ,  $fd=6$ ,  $ef=7$ ,  $cd=8$

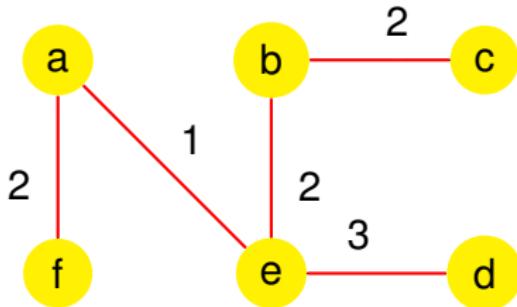
# Algorytm Kruskala - Przykład

Krok 5.

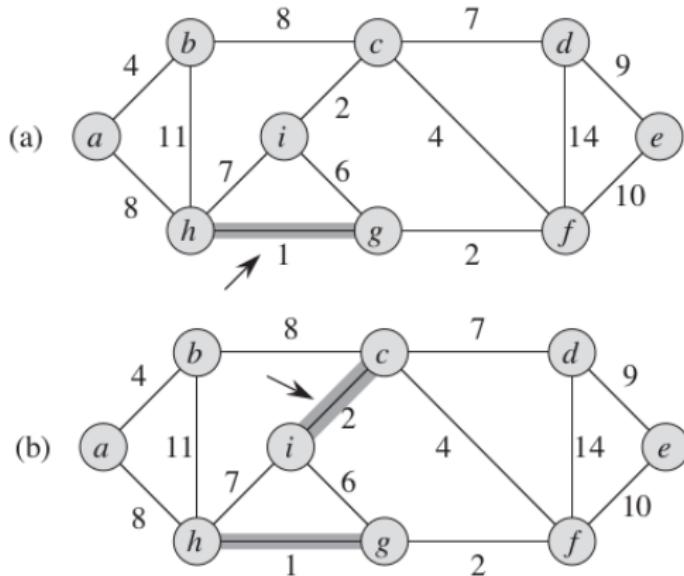


$ae=1$ ,  $af=2$ ,  $bc=2$ ,  $be=2$ ,  $de=3$ ,  
 $ab=4$ ,  $fd=6$ ,  $ef=7$ ,  $cd=8$

Minimalne drzewo.

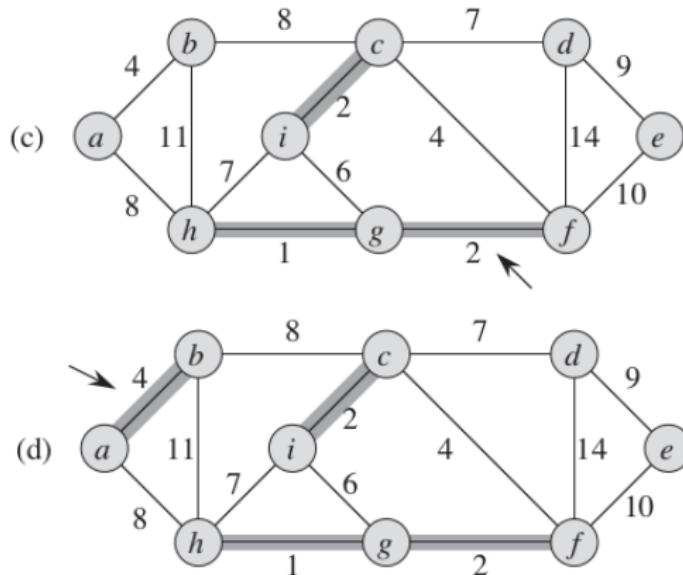


# Algorytm Kruskala - przykład



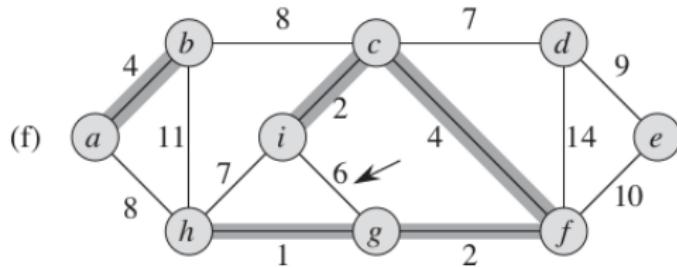
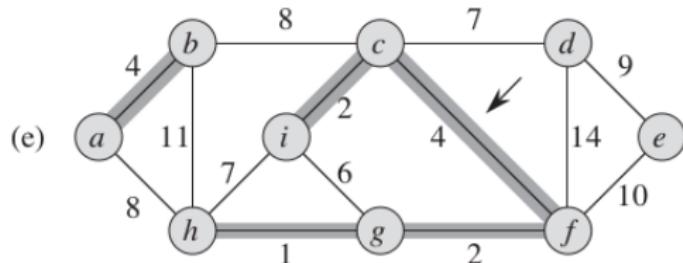
Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# Algorytm Kruskala - przykład



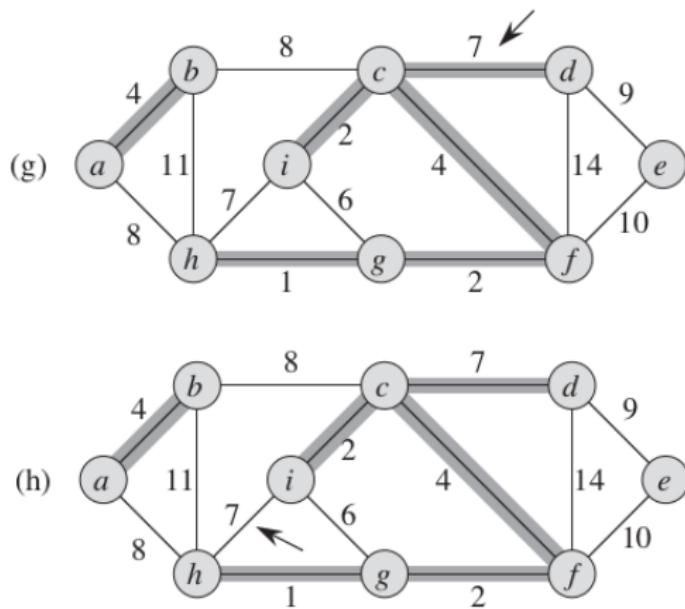
Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# Algorytm Kruskala - przykład



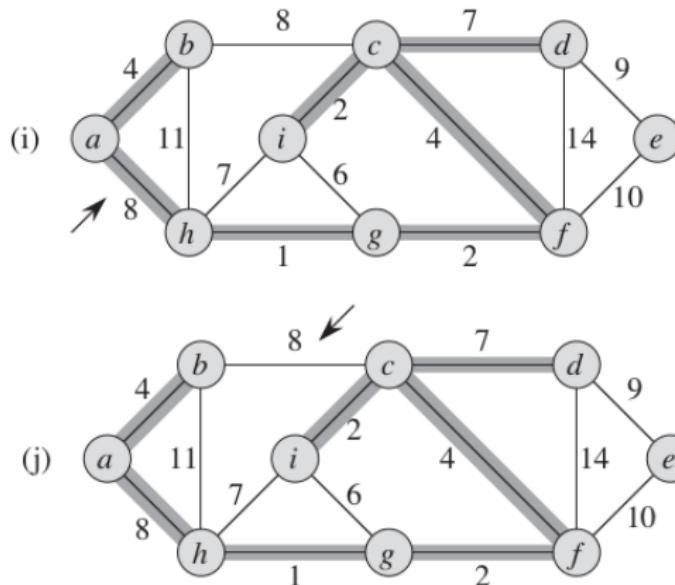
Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# Algorytm Kruskala - przykład



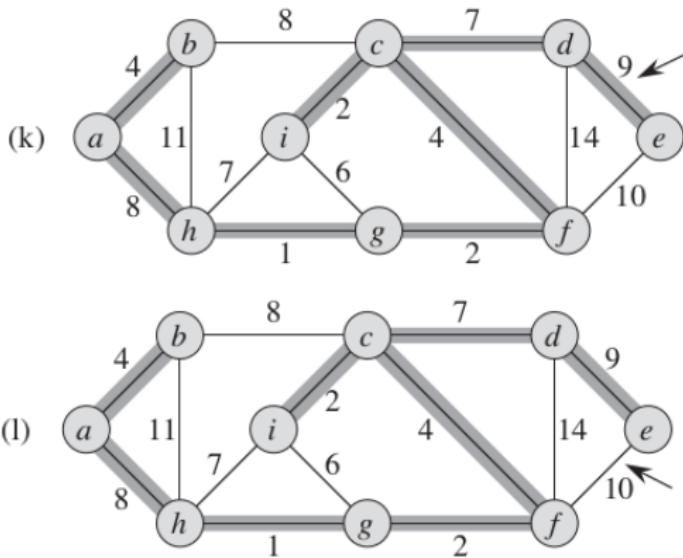
Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# Algorytm Kruskala - przykład



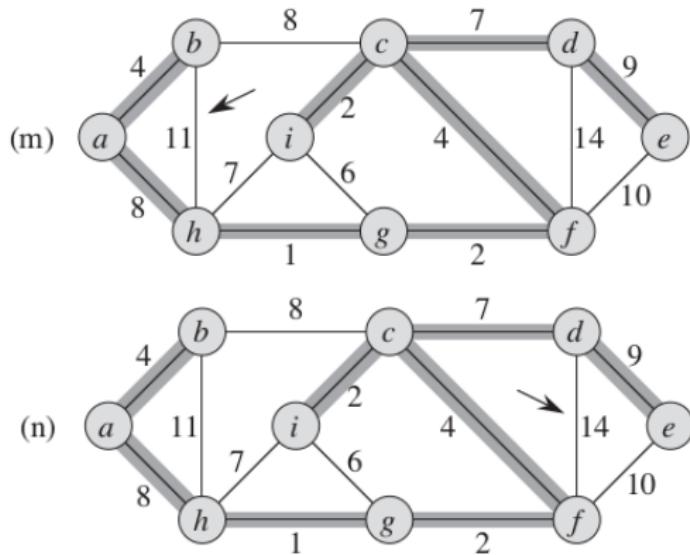
Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# Algorytm Kruskala - przykład



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

# Algorytm Kruskala - przykład



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

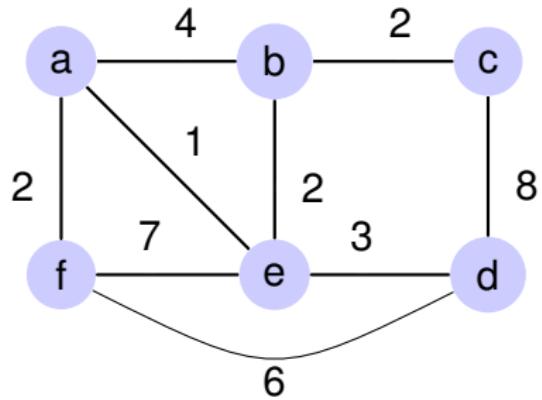
# Algorytm Prima

- Algorytm został wynaleziony w 1930 przez czeskiego matematyka **Vojtěcha Jarníka**, a następnie w 1957 odkryty na nowo przez informatyka **Roberta C. Prima** oraz niezależnie w 1959 przez **Edsgera Dijkstrę**. Z tego powodu algorytm nazywany jest również algorytmem **Dijkstry-Prima**, algorytmem **Jarníka**, albo algorytmem **Prima-Jarníka**.

# Algorytm Prima

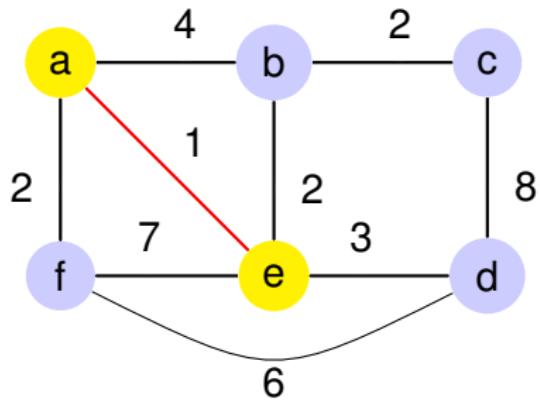
- Budowę minimalnego drzewa rozpinającego zaczynamy od dowolnego wierzchołka, np. od pierwszego. Dodajemy wierzchołek do drzewa, a wszystkie krawędzie incydentne umieszczamy na posortowanej wg. wag liście.
- Następnie zdejmujemy z listy pierwszy element (o najmniejszej wadze) i jeżeli wierzchołek, który łączy nie należy do drzewa, dodajemy go do drzewa a na liście znów umieszczamy wszystkie krawędzie incydentne z wierzchołkiem, który dodaliśmy.
- Jednym zdaniem: zawsze dodajemy do drzewa krawędź o najmniejszej wadze, osiągalną (w przeciwnieństwie do Kruskala) z jakiegoś wierzchołka tego drzewa.

# Algorytm Prima - Przykład



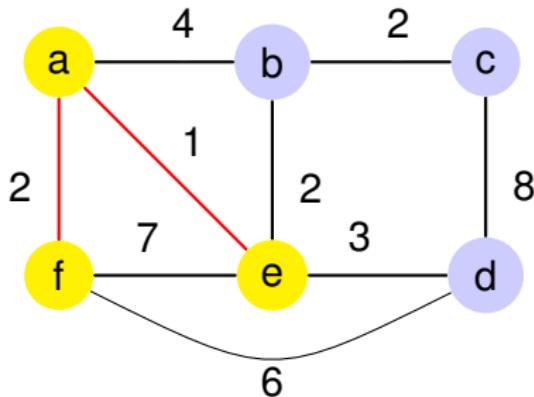
# Algorytm Prima - Przykład

Krok 1.



Wybieramy wierzchołek a.  
Tworzymy posortowaną listę  
 $L=[a,e,1],[a,f,2],[a,b,4]$ . Wybieramy krawędź (a,e).

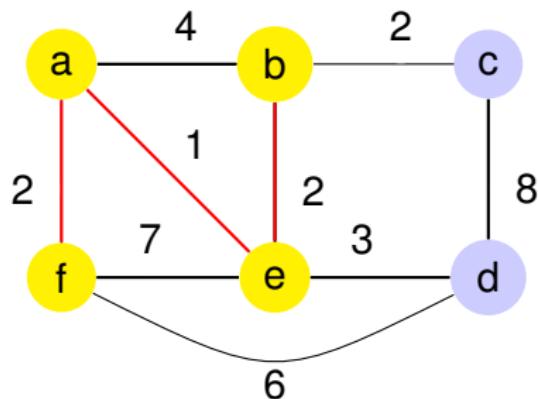
Krok 2.



Dodajemy nowe krawędzie:  
 $L=[a,f,2],[e,b,2],[e,d,3],[a,b,4],[e,f,7]$ .  
Wybieramy krawędź (a,f).

# Algorytm Prima - Przykład

Krok 3.

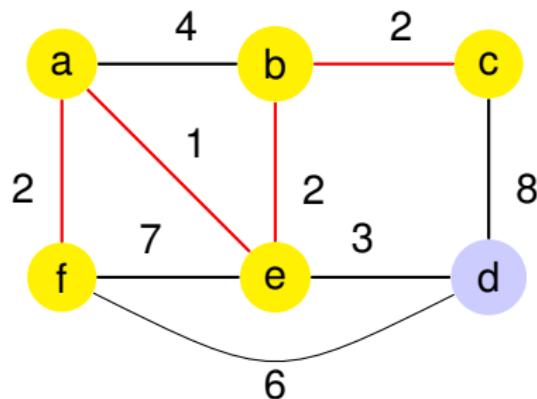


Krawędź [f,e,7] jest już na liście:

$$L=[e,b,2],[e,d,3],[a,b,4],[f,d,6],[e,f,7].$$

Wybieramy krawędź (e,b).

Krok 4



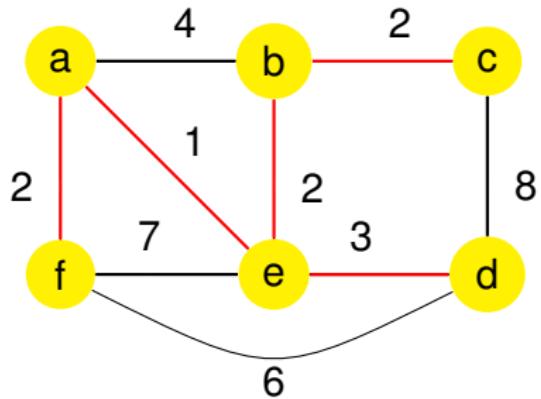
Dodajemy krawędź [b,c,2]:

$$L=[b,c,2],[e,d,3],[a,b,4],[f,d,6],[e,f,7]$$

Wybieramy krawędź (b,c).

# Algorytm Prima - Przykład

Krok 5.

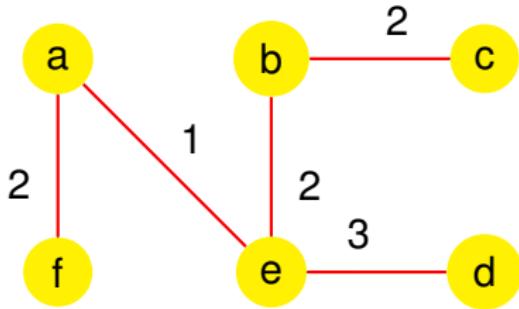


Dodajemy krawędź [c,d,8]:

$$L = [e,d,3], [a,b,4], [f,d,6], [e,f,7], [c,d,8]$$

Wybieramy krawędź (e,d).

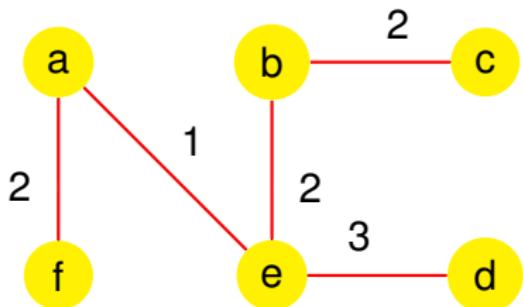
Minimalne Drzewo



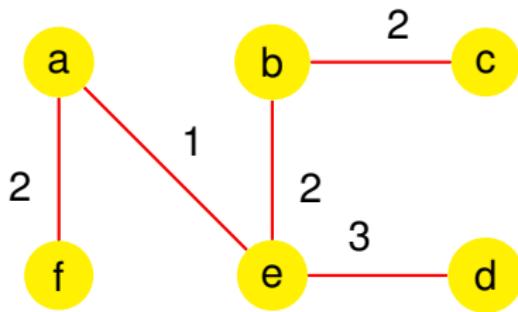
Drzewo utworzone.

# Algorytm Prima a Algorytm Kruskala - Przykład

Minimalne Drzewo wg. Algorytmu Kruskala.



Minimalne Drzewo wg. Algorytmu Prima



# Algorytm Prima I

Założenia:

- Wejście: graf oraz wierzchołek od którego rozpoczynamy budowę minimalnego drzewa rozpinającego.
- $Q$  - kolejka priorytetowa.
- $\text{key}(v)$  - klucz wyznaczający pozycję wierzchołka  $V$  w kolejce. Jest nim minimalna waga spośród wag krawędzi łączących  $v$  z wierzchołkami drzewa.  $\text{key}(v) = \infty$ , jeśli nie ma takiego wierzchołka.
- $\pi(v)$  - rodzic wierzchołka  $v$  w obliczonym drzewie.

$\text{Prim}(G = (V, E), r)$

- 1: **for** każdy  $u \in Q$  **do**
- 2:    $\text{key}(u) := \infty; \pi(u) = \text{NULL}$
- 3: **end for**
- 4:  $\text{key}(r) := 0$

# Algorytm Prima II

```

5:  $Q := V$ 
6: while  $Q \neq \emptyset$  do
7:    $u := ExtractMin(Q)$ 
8:   for każdy  $v \in Adj[u]$  do
9:     if  $v \in Q$  i  $w(u, v) < key(v)$  then
10:       $\pi(v) := u$ 
11:       $key(v) := w(u, v)$ 
12:    end if
13:  end for
14: end while

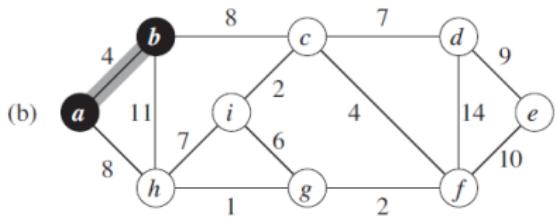
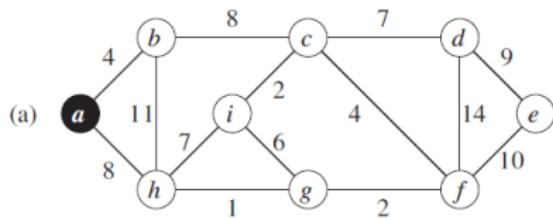
```

- Wiersz 1-5:inicjalizacja kolejki priorytetowej. Początkowo zawarte są w niej wszystkie wierzchołki, a kluczem każdego wierzcholka, poza korzeniem  $r$ , jest  $\infty$ .

# Algorytm Prima III

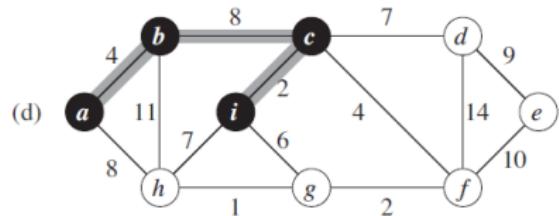
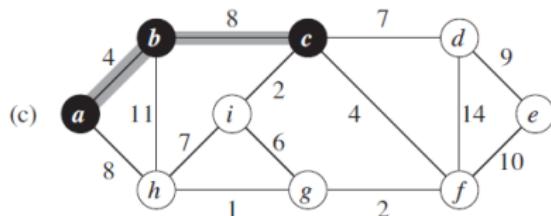
- W trakcie wykonywania algorytmu zbiór  $V - Q$  zawiera wierzchołki budowanego drzewa.

# Algorytm Prima -przykład I

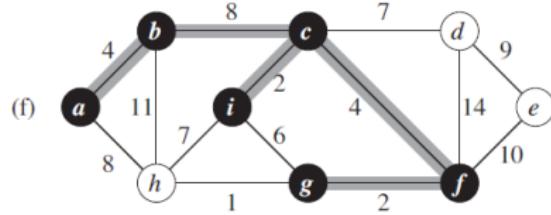
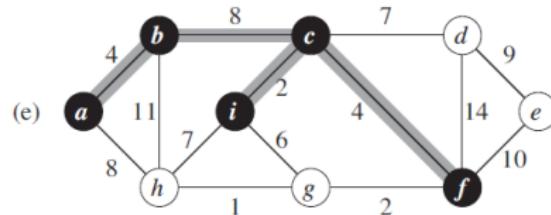


Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

# Algorytm Prima -przykład II

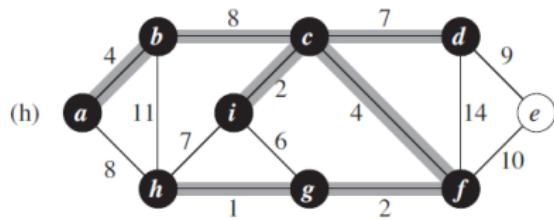
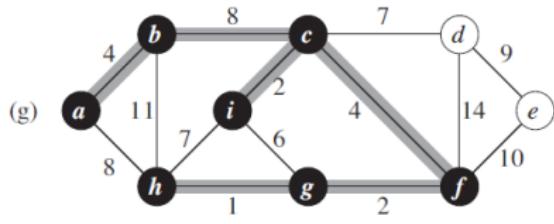


Rysunek: Źródło: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.



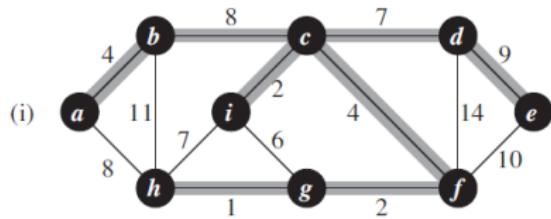
# Algorytm Prima -przykład III

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



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

# Algorytm Prima -przykład IV



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

# Złożoność algorytmu Prima I

- Czas działania algorytmu Prim zależy od tego, w jaki sposób implementujemy kolejkę priorytetową  $Q$ .
- Jeśli kolejka priorytetowa  $Q$  implementowana jest jako **kopiec minimalny**, wiersze 1-5 algorytmu możemy wykonać w czasie  $O(V)$ .

Zawartość pętli *while* (linie 6-14) wykonujemy  $|V|$  razy, a ponieważ każda operacja EXTRACT-MIN (operacja usunięcia korzenia kopca) zajmuje  $O(\log(|V|))$  czasu, to łączny czas wszystkich wywołań procedury *EXTRACT – MIN* wynosi  $O(|V|\log(|V|))$ .

Pętla *for* w wierszach 8 – 11 wykonuje się w czasie  $O(E)$ , ponieważ suma długości wszystkich list sąsiedztwa wynosi  $2|E|$ . W pętli *for* test należenia do  $Q$  w linii 9 można zaimplementować w stałym czasie, zachowując bit dla każdego wierzchołka, który

## Złożoność algorytmu Prima II

mówi, czy jest w  $Q$ , czy też nie, i aktualizuje bit, gdy wierzchołek jest usuwany z  $Q$ .

Przypisanie w linii 11 obejmuje niejawną operację  $DECREASE - KEY$  na kopcu minimalnym, która jest realizowana w czasie  $O(\log(V))$ .

Zatem całkowity czas pracy dla algorytmu Prima wynosi  $O(|V|\log(|V|) + |E|\log(|V|)) = O(|E|\log(|V|))$ , czyli asymptotycznie czas jest taki sam jak dla algorytmu Kruskala.

- Można poprawić asymptotyczny czas działania algorytmu Prima, poprzez zastosowanie kopców Fibonacciego. Wówczas całkowity czas pracy dla algorytmu Prima wynosi  $O(|E| + |V|\log(|V|))$ .

# Algorytmy Grafowe

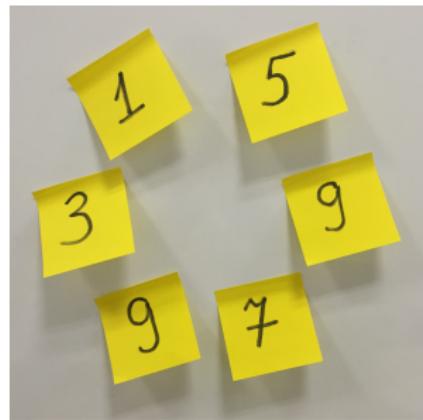
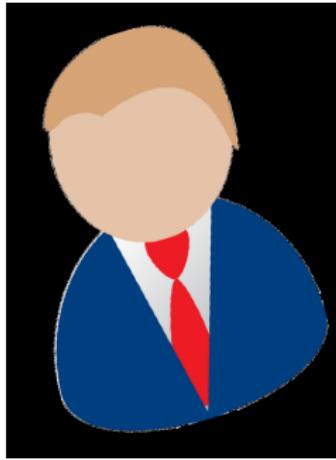
dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

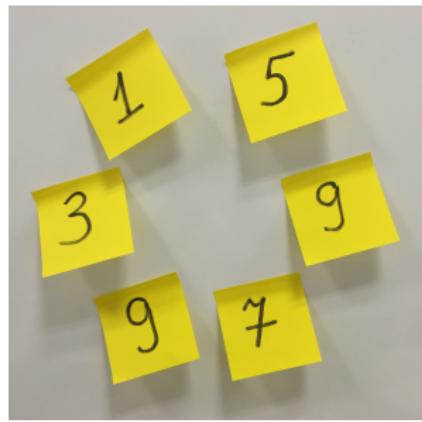
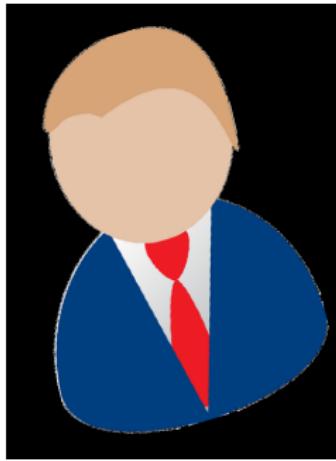
b.wozna@ujd.edu.pl

Wykład 9

# Rozmowa kwalifikacyjna



# Rozmowa kwalifikacyjna



# Największa liczba

## Definicja problemu

Jaka jest największa liczba, która składa się z cyfr 3, 9, 5, 9, 7, 1?  
Użyj wszystkich cyfr.

# Największa liczba

## Definicja problemu

Jaka jest największa liczba, która składa się z cyfr 3, 9, 5, 9, 7, 1?  
Użyj wszystkich cyfr.

## Przykład

359179, 537991, 913579, ...

# Największa liczba

## Definicja problemu

Jaka jest największa liczba, która składa się z cyfr 3, 9, 5, 9, 7, 1?  
Użyj wszystkich cyfr.

## Przykład

359179, 537991, 913579, ...

## Poprawna odpowiedź

**997531** ...

# Strategia zachłanna

- Dany jest zbiór cyfr:

5	7	3	9	1	9
---	---	---	---	---	---

# Strategia zachłanna

- Dany jest zbiór cyfr:

5	7	3	9	1	9
---	---	---	---	---	---

- Znajdź maksymalną cyfrę

5	7	3	9	1	9
---	---	---	---	---	---

# Strategia zachłanna

- Dany jest zbiór cyfr:

5	7	3	9	1	9
---	---	---	---	---	---

- Znajdź maksymalną cyfrę

5	7	3	9	1	9
---	---	---	---	---	---

- Dołącz ją do poszukiwanej liczby

5	7	3	9	1	9	→	9
---	---	---	---	---	---	---	---

# Strategia zachłanna

- Dany jest zbiór cyfr:

5	7	3	9	1	9
---	---	---	---	---	---

- Znajdź maksymalną cyfrę

5	7	3	9	1	9
---	---	---	---	---	---

- Dołącz ją do poszukiwanej liczby

5	7	3	9	1	9	→	9
---	---	---	---	---	---	---	---

- Usuń ją z listy cyfr

5	7	3	1	9	→ 9
---	---	---	---	---	-----

# Strategia zachłanna

- Dany jest zbiór cyfr:

5	7	3	9	1	9
---	---	---	---	---	---

- Znajdź maksymalną cyfrę

5	7	3	9	1	9
---	---	---	---	---	---

- Dołącz ją do poszukiwanej liczby

5	7	3	9	1	9	→	9
---	---	---	---	---	---	---	---

- Usuń ją z listy cyfr

5	7	3	1	9	→ 9
---	---	---	---	---	-----

- Powtarzaj, dopóki na liście znajdują się cyfry

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----

## Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------

## Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------

## Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------
- Powtarzaj, dopóki na liście znajdują się cyfry.

## Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	7	3	1	→ 99
---	---	---	---	------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	7	3	1	→ 99
---	---	---	---	------
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	→ 997
---	---	---	---	-------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	7	3	1	→ 99
---	---	---	---	------
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	→ 997
---	---	---	---	-------
- Usuń ją z listy cyfr: 

5	3	1	→ 997
---	---	---	-------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	7	3	1	→ 99
---	---	---	---	------
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	→ 997
---	---	---	---	-------
- Usuń ją z listy cyfr: 

5	3	1	→ 997
---	---	---	-------
- Powtarzaj, dopóki na liście znajdują się cyfry.

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	7	3	1	→ 99
---	---	---	---	------
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	→ 997
---	---	---	---	-------
- Usuń ją z listy cyfr: 

5	3	1	→ 997
---	---	---	-------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	3	1	→ 997
---	---	---	-------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	7	3	1	→ 99
---	---	---	---	------
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	→ 997
---	---	---	---	-------
- Usuń ją z listy cyfr: 

5	3	1	→ 997
---	---	---	-------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	3	1	→ 997
---	---	---	-------
- Dołącz ją do poszukiwanej liczby: 

5	3	1	→ 9975
---	---	---	--------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	→ 9
---	---	---	---	---	-----
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	→ 99
---	---	---	---	---	------
- Usuń ją z listy cyfr: 

5	7	3	1	→ 99
---	---	---	---	------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	7	3	1	→ 99
---	---	---	---	------
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	→ 997
---	---	---	---	-------
- Usuń ją z listy cyfr: 

5	3	1	→ 997
---	---	---	-------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	3	1	→ 997
---	---	---	-------
- Dołącz ją do poszukiwanej liczby: 

5	3	1	→ 9975
---	---	---	--------
- Usuń ją z listy cyfr: 

3	1	→ 9975
---	---	--------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

5	7	3	1	9	
---	---	---	---	---	--

 → 9
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	9	
---	---	---	---	---	--

 → 99
- Usuń ją z listy cyfr: 

5	7	3	1	
---	---	---	---	--

 → 99
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	7	3	1	
---	---	---	---	--

 → 99
- Dołącz ją do poszukiwanej liczby: 

5	7	3	1	
---	---	---	---	--

 → 997
- Usuń ją z listy cyfr: 

5	3	1	
---	---	---	--

 → 997
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

5	3	1	
---	---	---	--

 → 997
- Dołącz ją do poszukiwanej liczby: 

5	3	1	
---	---	---	--

 → 9975
- Usuń ją z listy cyfr: 

3	1	
---	---	--

 → 9975
- Powtarzaj, dopóki na liście znajdują się cyfry.

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------
- Dołącz ją do poszukiwanej liczby: 

3	1		→ 99753
---	---	--	---------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------
- Dołącz ją do poszukiwanej liczby: 

3	1		→ 99753
---	---	--	---------
- Usuń ją z listy cyfr: 

1	→ 99753
---	---------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------
- Dołącz ją do poszukiwanej liczby: 

3	1		→ 99753
---	---	--	---------
- Usuń ją z listy cyfr: 

1	→ 99753
---	---------
- Powtarzaj, dopóki na liście znajdują się cyfry.

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------
- Dołącz ją do poszukiwanej liczby: 

3	1		→ 99753
---	---	--	---------
- Usuń ją z listy cyfr: 

1	→ 99753
---	---------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

1	→ 99753
---	---------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------
- Dołącz ją do poszukiwanej liczby: 

3	1		→ 99753
---	---	--	---------
- Usuń ją z listy cyfr: 

1	→ 99753
---	---------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

1	→ 99753
---	---------
- Dołącz ją do poszukiwanej liczby: 

1	→ 997531
---	----------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------
- Dołącz ją do poszukiwanej liczby: 

3	1		→ 99753
---	---	--	---------
- Usuń ją z listy cyfr: 

1	→ 99753
---	---------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

1	→ 99753
---	---------
- Dołącz ją do poszukiwanej liczby: 

1	→ 997531
---	----------
- Usuń ją z listy cyfr: 

→ 997531
----------

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------
- Dołącz ją do poszukiwanej liczby: 

3	1		→ 99753
---	---	--	---------
- Usuń ją z listy cyfr: 

1	→ 99753
---	---------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

1	→ 99753
---	---------
- Dołącz ją do poszukiwanej liczby: 

1	→ 997531
---	----------
- Usuń ją z listy cyfr: 

→ 997531
----------
- Powtarzaj, dopóki na liście znajdują się cyfry.

# Strategia zachłanna

- Znajdź maksymalną cyfrę: 

3	1		→ 9975
---	---	--	--------
- Dołącz ją do poszukiwanej liczby: 

3	1		→ 99753
---	---	--	---------
- Usuń ją z listy cyfr: 

1	→ 99753
---	---------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Znajdź maksymalną cyfrę: 

1	→ 99753
---	---------
- Dołącz ją do poszukiwanej liczby: 

1	→ 997531
---	----------
- Usuń ją z listy cyfr: 

→ 997531
----------
- Powtarzaj, dopóki na liście znajdują się cyfry.
- Wynik: **997531**

# Strategia zachłanna

Tankowanie samochodu

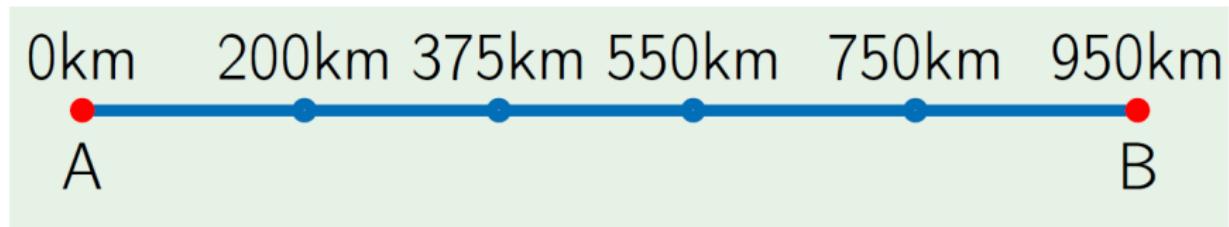
Dystans z pełnym bakiem = 400km



# Strategia zachłanna

Tankowanie samochodu

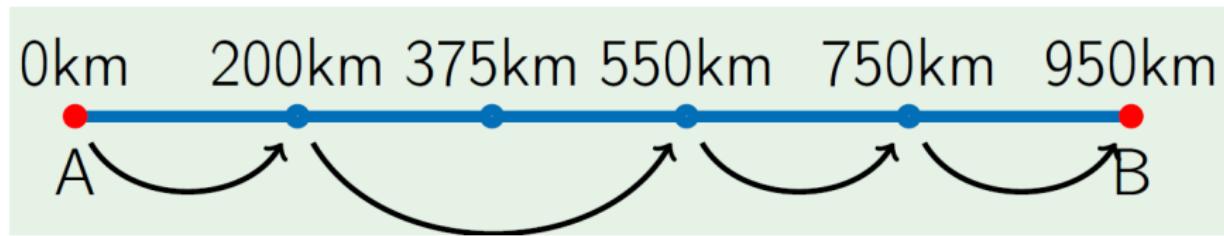
Dystans z pełnym bakiem = 400km



# Strategia zachłanna

Tankowanie samochodu

Dystans z pełnym bakiem = 400km



# Strategia zachłanna

Tankowanie samochodu

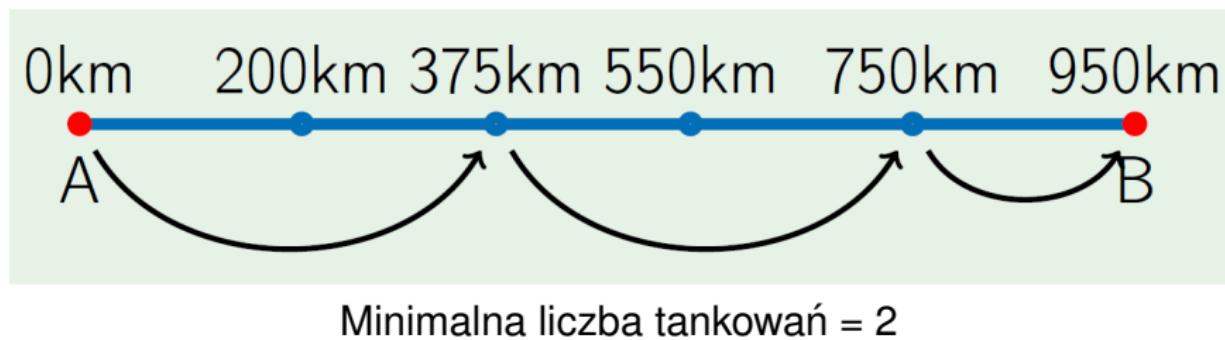
Dystans z pełnym bakiem = 400km



# Strategia zachłanna

Tankowanie samochodu

Dystans z pełnym bakiem = 400km



# Tankowanie samochodu

## Wejście:

Samochód, który może przejechać co najwyżej  $L$  kilometrów z pełnym bakiem, punkt początkowy  $A$ , punkt docelowy  $B$  i  $n$  stacji benzynowych w odległościach  $x_1 \leq x_2 \leq x_3 \leq \dots \leq x_n$  w kilometrach od  $A$  wzdłuż drogi od  $A$  do  $B$ .

## Wyjście:

Minimalna liczba tankowań, wymagana aby dostać się z punktu  $A$  do punktu  $B$ , oprócz uzupełnienia w punkcie  $A$ .

# Strategia zachłanna

- Dokonaj wyboru zachłannego
- Zredukuj do mniejszego problemu
- Powtarzaj postępowanie dla mniejszego problemu

# Strategia zachłanna

- Dokonaj wyboru zachłannego
- Zredukuj do mniejszego problemu
- Powtarzaj postępowanie dla mniejszego problemu

Która strategię wybrać?

- Uzupełnij paliwo na najbliższej stacji benzynowej
- Uzupełnij paliwo na najdalej dostępnej stacji benzynowej
- Jedź, aż zabraknie paliwa

# Strategia zachłanna

- Dokonaj wyboru zachłannego
- Zredukuj do mniejszego problemu
- Powtarzaj postępowanie dla mniejszego problemu

Która strategię wybrać?

- Uzupełnij paliwo na najbliższej stacji benzynowej
- **Uzupełnij paliwo na najdalej dostępnej stacji benzynowej**
- Jedź, aż zabraknie paliwa

# Algorytm zachłanny

- Zaczni od A;
- Uzupełnij paliwo na najdalej dostępnej stacji benzynowej G;
- Ustaw G jako nowe A;
- Przejdź z nowego punktu A do punktu B przy minimalnej liczbie uzupełnień;

# Algorytm zachłanny

$$A = x_0 \leq x_1 \leq x_2 \leq \dots \leq x_n \leq x_{n+1} = B$$

MinRefills( $x$ ,  $n$ ,  $L$ )

```
numRefills ← 0, currentRefill ← 0
while currentRefill ≤ n:
    lastRefill ← currentRefill
    while (currentRefill ≤ n and
           x[currentRefill + 1] - x[lastRefill] ≤ L):
        currentRefill ← currentRefill + 1
    if currentRefill == lastRefill:
        return IMPOSSIBLE
    if currentRefill ≤ n:
        numRefills ← numRefills + 1
return numRefills
```

# Algorytm zachłanny

## Złożoność

Czas działania algorytmu  $\text{MinRefills}(x, n, L)$  wynosi  $O(n)$ .

## Dowód

- `currentRefill` zmienia się od 0 do  $n + 1$ , co jeden.
- `numRefills` zmienia się od 0 do maksymalnie  $n$ , co jeden.
- Zatem mamy  $O(n)$  iteracji.

# Strategia zachłanna

## Podproblem

Podproblem to problem podobny do oryginalnego, ale o mniejszym rozmiarze.

## Przykład

- $\text{LargestNumber}(3, 9, 5, 9, 7, 1) =$   
„9” +  $\text{LargestNumber}(3, 5, 9, 7, 1)$
- Minimalna liczba tankowań z A do B = pierwsze tankowanie w G + minimalna liczba tankowań z G do B

# Strategia zachłanna

## Podproblem

Podproblem to problem podobny do oryginalnego, ale o mniejszym rozmiarze.

## Przykład

- $\text{LargestNumber}(3, 9, 5, 9, 7, 1) =$   
„9” +  $\text{LargestNumber}(3, 5, 9, 7, 1)$
- Minimalna liczba tankowań z A do B = pierwsze tankowanie w G + minimalna liczba tankowań z G do B

## Bezpieczny wybór

Wybór zachłanny nazywany jest **bezpiecznym wyborem**, jeśli istnieje optymalne rozwiązanie zgodne z pierwszym wyborem.

# Algorytmy zachłanne

- Algorytmy zachłanne stosowane do rozwiązywania problemów optymalizacyjnych, w których osiągnięcie optymalnego rozwiązania wymaga podejmowania wielu decyzji.
- Algorytm zachłanny przy podejmowaniu jednej z wielu możliwych decyzji zawsze wykonuje działanie, które wydaje się w danej chwili najkorzystniejsze.
- Algorytm zachłanny wybiera lokalnie optymalną możliwość w nadziei, że doprowadzi ona do globalnie optymalnego rozwiązania (działa zachłannie)
- **Algorytmy zachłanne nie zawsze prowadzą do optymalnych rozwiązań.**

# Problem wyboru zajęć

- Problem wyboru zajęć, to problem przydzielenia dostępu do zasobu wykorzystywanego podczas wykonywania pewnych zajęć.
- **Założenia:**
  - Niech będzie dany zbiór proponowanych zajęć  $S = \{1, \dots, n\}$ , do których ma być przydzielona sala wykładowa (zasoby), w której może się odbywać w danej chwili tylko jedno z tych zajęć.
  - Każde zajęcie ma swój czas rozpoczęcia  $s_i$  oraz czas zakończenia  $f_i$  takie, że  $s_i \leq f_i$ . Jeżeli zajęcie o numerze  $i$  zostanie wytypowane, to zajmuje zasób  $[s_i, f_i]$ . Zajęcia o numerach  $i$  oraz  $j$  są zgodne, jeśli  $[s_i, f_i] \cap [s_j, f_j] = \emptyset$
- **Problem:** Wyznaczyć największy podzbiór parami zgodnych zajęć.

# Jak działa algorytm wyczerpujący (naiwny) ?

- Generujemy wszystkie podzbiory zbioru zajęć
- Wybieramy te podzbiory, które są parami zgodne
- Wybieramy ten podzbiór, który ma najwięcej elementów
- **Obserwacja:** Algorytm wyczerpujący ma złożoność  $O(2^n)$

## Strategia zachłanna: przykład

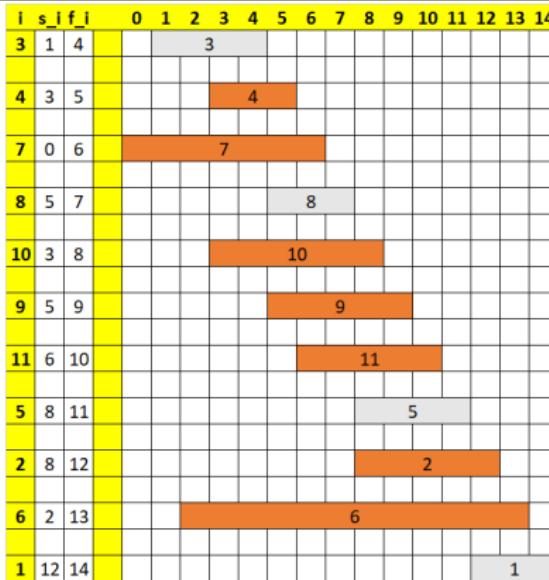
- Źródło:
- Dane są zajęcia:

i	1	2	3	4	5	6	7	8	9	10	11
$s_i$	12	8	1	3	8	2	0	5	5	3	6
$f_i$	14	12	4	5	11	13	6	7	9	8	10

- Porządkujemy zajęcia ze względu na czas zakończenia  
 $f_1 \leq f_2 \leq \dots \leq f_n$
- Wybieramy na każdym kroku to zajęcie, które ma najwcześniejszy czas zakończenia wśród zajęć, które mogą być dołączone do zbioru - wybór ten maksymalizuje ilość nie zajętego czasu po jego dokonaniu.

# Strategia zachłanna: przykład

i	1	2	3	4	5	6	7	8	9	10	11
$s_i$	12	8	1	3	8	2	0	5	5	3	6
$f_i$	14	12	4	5	11	13	6	7	9	8	10



# Algorytm: GREEDY-ACTIVITY-SELECTOR

- Dane wejściowe stanowią tablice  $s$  (czasów rozpoczęcia) oraz  $f$  (czasów zakończenia).
- Zakładamy, że zajęcia są posortowane ze względu na czas zakończenia.

## GREEDY-ACTIVITY-SELECTOR ( $s, f$ )

```
1:  $n := \text{length}[s]$ 
2:  $A := \{1\}$ 
3:  $j := 1$ 
4: for  $i := 2$  to  $n$  do
5:   if  $s_i \geq f_j$  then
6:      $A := A \cup \{i\}$ 
7:      $j := i$ 
8:   end if
9: end for
10: return  $A$ 
```

- Zbiór  $A$  zawiera wybrane zajęcia, a zmienna  $j$  zawiera numer ostatnio dodanego do  $A$  zajęcia.
- Zajęcia są rozpatrywane w porządku rosnącego czasu zakończenia, zatem  $f_j$  jest zawsze największym czasem zakończenia zajęcia należącego do  $A$ .
- W wierszach 2-3 wybieramy zajęcie 1, zbiór  $\{1\}$  staje się wartością zmiennej  $A$ , a zmiennej  $j$  przypisujemy numer tego zajęcia.
- W wierszach 4-9 rozpatrywane są wszystkie zajęcia; zajęcie zostaje dołączone, jeżeli jest zgodne ze wszystkimi dołączonymi dotychczas zajęciami.
- Aby stwierdzić, czy zajęcie  $i$  jest zgodne z każdym zajęciem ze zbioru  $A$ , wystarczy sprawdzić czy jego czas rozpoczęcia  $s_i$  nie jest wcześniejszy niż czas zakończenia  $f_j$  zajęcia ostatnio dodanego do  $A$ .
- Jeśli zajęcie  $i$  jest zgodne, to w wierszach 6-7 zostaje ono dodane do zbioru  $A$  oraz jest aktualizowana

## Algorytm: GREEDY-ACTIVITY-SELECTOR - Uwagi

- Zajęcie wybrane przez GREEDY-ACTIVITY-SELECTOR ma zawsze najwcześniejszy czas zakończenia wśród zajęć, które mogą być dołączone bez zakłócenia zgodności zbioru A.
- Wybór jest **zachłanny** w tym sensie, że pozostawia możliwie najwięcej swobody przy wyborze pozostałych zajęć.
- **Twierdzenie:** Algorytm zachłanny generuje rozwiązanie problemu wyboru zajęć o największym rozmiarze.
- Złożoność  $O(n)$ .

# Charakterystyczne cechy problemów poddających się strategii zachłannej

- **optymalna podstruktura** - optymalne rozwiązanie jest funkcją optymalnych rozwiązań podproblemów. Na przykład dla problemu wyboru zajęć własność optymalnej podstruktury polega na tym, że: jeżeli optymalne rozwiązanie A tego problemu rozpoczyna się od zajęć o numerze 1, to  $A' = A - \{1\}$  jest optymalnym rozwiązaniem problemu optymalnego wyboru zajęć dla zbioru  $S' = \{i \in S : s_i \geq f_1\}$ .
- **własność wyboru zachłanego** - za pomocą lokalnie optymalnych (zachłannych) wyborów można uzyskać globalnie optymalne rozwiązanie. Wybory podejmowane w algorytmie zachłannym nie są zależne od wyborów przeszłych. Można formalnie udowodnić (stosując metodę indukcji), że dany problem ma własność wyboru zachłanego.

# Problem przydziału zajęć do minimalnej liczby sal

- **Problem:** Dany jest zbiór zajęć, które mają się odbyć w pewnej liczbie sal wykładowych. Należy wyznaczyć taki przydział zajęć do sal, aby liczba użytych sal była najmniejsza.
- **Metoda zachłanna rozwiązująca ten problem:**
  - Niech będzie dany zbiór zajęć  $S = 1, \dots, n$ , do których ma być przydzielona sala wykładowa (zasoby), w której może się odbywać w danej chwili tylko jedno z tych zajęć.
  - Każde zajęcie ma swój czas rozpoczęcia  $s_i$  oraz czas zakończenia  $f_i$  takie, że  $s_i \leq f_i$ . Jeżeli zajęcie o numerze  $i$  zostanie wytypowane, to zajmuje zasób  $[s_i, f_i]$ . Zajęcia o numerach  $i$  oraz  $j$  są zgodne jeśli  $[s_i, f_i] \cap [s_j, f_j] = 0$ .
  - Niech zajęcia w  $S$  będą uporządkowane niemalejąco ze względu na ich czas rozpoczęcia; Jeżeli znajdą się dwa zajęcia  $i$  oraz  $j$  takie, że  $s_i = s_j$ , to wówczas pierwsze będzie to zajęcie, które ma dłuższy czas trwania.

# Algorytm: GREEDY-ROOM-SELECTOR

- Dane wejściowe stanowią tablice  $s$  (czasów rozpoczęcia) oraz  $f$  (czasów zakończenia).
- Zakładamy, że zajęcia są posortowane ze względu na czas rozpoczęcia.

## **GREEDY-ROOM-SELECTOR ( $s, f$ )**

```
1:  $k := 1$ 
2:  $A := \{1, 2, \dots, \text{length}[s]\}$ 
3: while  $A \neq \emptyset$  do
4:    $p[k] := \text{GREEDY-SELECTOR}(s, f)$ 
5:    $A := A - p[k]$ 
6:   if  $A \neq \emptyset$  then
7:      $k := k + 1$ 
8:   end if
9: end while
10: return  $k$ 
```

- Zadaniem procedury pomocniczej GREEDY-SELECTOR jest wybór ze zbioru  $S$  podzbioru zawierającego wszystkie parami zgodne zajęcia, wśród których jest zajęcie nr 1.
- Działanie tej procedury jest analogiczne jak działanie procedury GREEDY-ACTIVITY-SELECTOR. Różnica polega jedynie na tym, że GREEDY-SELECTOR pobiera zajęcia uporządkowane zgodnie z czasem ich rozpoczęcia, a GREEDY-ACTIVITY-SELECTOR pobiera zajęcia uporządkowane zgodnie z czasem ich zakończenia.
- Złożoność  $O(n^2)$

## Algorytm: GREEDY-ROOM-SELECTOR - Uwagi

- Liczba sal wyznaczona GREEDY-ROOM-SELECTION jest zawsze najmniejsza
- Wybór jest zachłanny w tym sensie, że pozostawia w danej Sali minimalną ilość niewykorzystanego czasu
- **Twierdzenie:** Algorytm GREEDY-ROOM-SELECTION generuje optymalne rozwiązanie problemu przydziału zajęć do jak najmniejszej liczby sal.

# Problem wydawania reszty za pomocą jak najmniejszej liczby monet

- **Problem:** Mamy resztę R i N monet o nominałach o określonych wartościach całkowitych. Monet o określonym nominale jest nieograniczona liczba. Należy podać jak najmniejszą ilość monet potrzebną do wydania reszty.
- **Przykład:** Wydać 98 PLN, dysponując nominałami:

50, 20, 10, 5, 2, 1.

- **Rozwiązanie:** Porządkujemy nominały malejąco ze względu na ich wartości. Wydajemy następująco:

- 50 - pozostało 48
- 20 - pozostało 28
- 20 - pozostało 8
- 5 - pozostało 3
- 2 - pozostało 1
- 1 - pozostało 0

# Algorytm: GREEDY-GIVE-CHANGE

- **Założenie:**  $c$  jest uporządkowaną malejąco tablicą nominałów,  $sum$  jest resztą do wydania,  $k$  jest liczbą nominałów różnego rodzaju.

## GREEDY-GIVE-CHANGE ( $sum, c, k$ )

```
1: for  $i := 0$  to  $k$  do
2:   if  $(sum \text{ div } c[i]) \neq 0$  then
3:      $change[i] := sum \text{ div } c[i]$ 
4:      $sum := sum \text{ mod } c[i]$ 
5:   else
6:      $change[i] := 0$ 
7:   end if
8: end for
9: return  $change$ 
```

- Do tablicy  $change$  wpisujemy największą możliwą liczbę nominału  $c^k > c^{k-1} > \dots > c^0$ .
- Jeżeli jakiegoś nominału nie można użyć do wydania określonej reszty, tzn. nie jest spełniony warunek  $(sum \text{ div } c[i])! = 0$ , to do tablicy  $change$  wpisywana jest wartość zero - wiersz 6 algorytmu.
- Złożoność  $O(n)$

# Problem wydawania reszt - uwagi

- Istnieją zbiory nominałów, dla których podany algorytm nie daje optymalnego rozwiązania.
- **Przykład 1:** Wydać 10PLN za pomocą nominałów 1, 5 i 6.
  - Rozwiązanie algorytmu zachłannego:  $6 + 1 + 1 + 1 + 1$ , czyli 5 monet.
  - Rozwiązanie optymalne:  $5 + 5$
- **Przykład 2:** Wydać 66 PLN za pomocą nominałów 50, 20 i 1.
  - Rozwiązanie algorytmu zachłannego:  $50 + 16 * 1$ , czyli 17 nominałów.
  - Rozwiązanie optymalne:  $3 * 20 + 6 * 1$ , czyli 9 nominałów.

# Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

b.wozna@ujd.edu.pl

Wykład 9 i 10

# Spis treści

- 1 Mosty królewieckie
- 2 Ścieżka i cykl Eulera

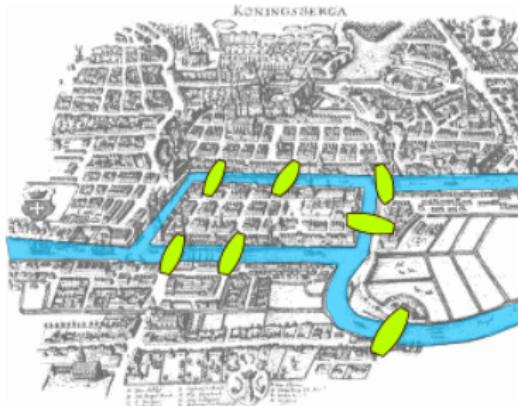
# Spis treści

1 Mosty królewieckie

2 Ścieżka i cykl Eulera

# Leonhard Euler i Mosty królewieckie I

Przez Królewiec przepływała rzeka Pregole, w której rozwidleniach znajdowały się dwie wyspy. Ponad rzeką przerzucono siedem mostów, z których jeden łączył obie wyspy, a pozostałe mosty łączyły wyspy z brzegami rzeki.



Źródło: <https://commons.wikimedia.org/w/index.php?curid=112920>

# Leonhard Euler i Mosty królewieckie II

**Pytanie:** czy można przejść przez każdy z siedmiu mostów dokładnie jeden raz tak, aby powrócić do punktu wyjścia ?

Wybitny szwajcarski matematyk Leonhard Euler (1707-1783) zainteresował się problemem mostów Królewieckich około 1735 roku i opublikował rozwiązanie („Solutio problematis ad geometriam situs pertinentis”) w 1741 roku.

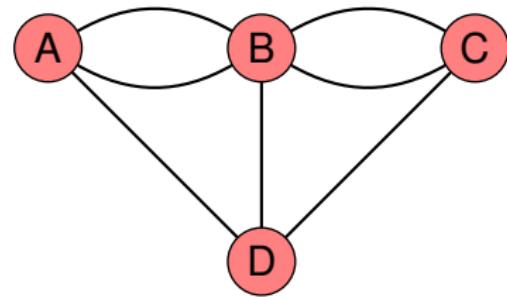
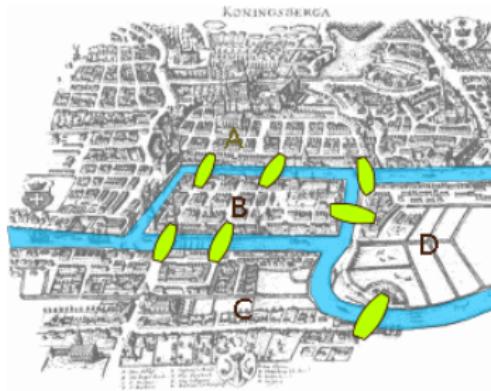


Leonhard Euler (1707 – 1783)

# Leonhard Euler i Mosty królewieckie III

Intuicja Eulera: fizyczna mapa nie ma znaczenia. Liczy się tylko lista regionów połączonych mostami.

**Odpowiedź:** problem mostów królewieckich równoważny jest pytaniu, czy graf pokazany na rysunku po prawej stronie jest grafem eulerowskim.



# Spis treści

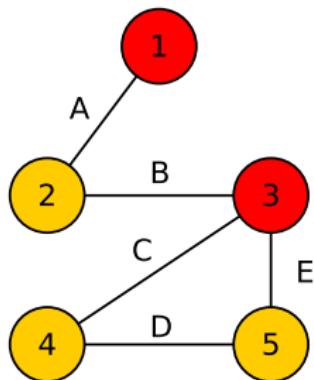
- 1 Mosty królewieckie
- 2 Ścieżka i cykl Eulera

# Ścieżka Eulera

- **Ścieżka Eulera** w grafie (skierowanym), to ścieżka (droga) prosta, która zawiera każdą krawędź grafu dokładnie jeden raz.
- Warunkiem istnienia ścieżki są:
  - 1 spójność grafu.
  - 2 dla grafu skierowanego należy sprawdzić, czy dla każdego wierzchołka, za wyjątkiem dwóch, stopień wyjściowy jest równy stopniu wejściowemu.
  - 3 dla grafu nieskierowanego z każdego wierzchołka, za wyjątkiem dwóch, musi wychodzić parzysta liczba krawędzi.
- Graf, który posiada ścieżkę Eulera nazywamy **grafem półeulerowskim**

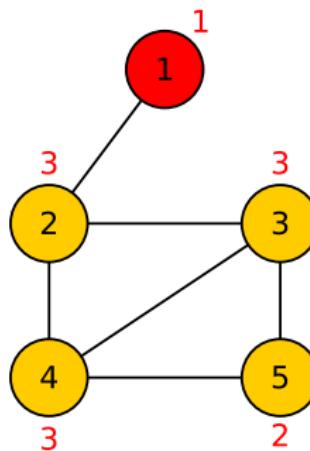
# Ścieżka Eulera - przykład

Graf ze ścieżką Eulera:



Ścieżka: 1-> ABCDE -> 3

Graf bez ścieżki Eulera:



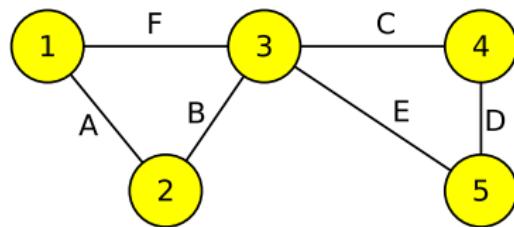
Nie spełniony warunek 3 istnienia  
ścieżki Eulera

# Cykl Eulera

- Cykl Eulera to taki cykl w grafie, który zawiera każdą krawędź grafu dokładnie jeden raz.
- Warunkiem istnienia cyklu są:
  - spójność grafu.
  - dla grafu skierowanego należy sprawdzić, czy dla każdego wierzchołka stopień wyjściowy jest równy stopniu wejściowemu.
  - dla grafu nieskierowanego z każdego wierzchołka musi wychodzić parzysta liczba krawędzi.
- Graf, który posiada cykl Eulera nazywamy **grafem eulerowskim**

# Cykl Eulera - przykład

Graf z cyklem Eulera:



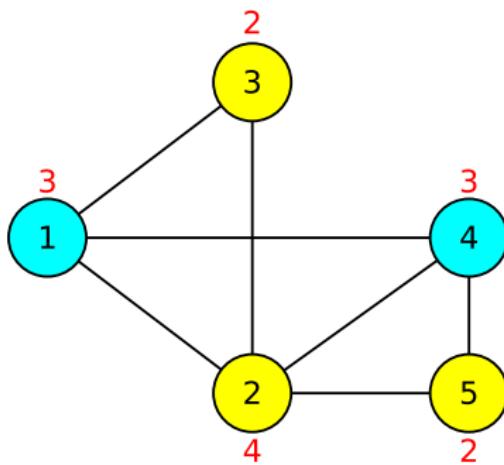
Cykl 1: 1->ABCDEF -> 1

Cykl 2: 1->ABEDCF -> 1

Cykl 3: 1->FCDEBA -> 1

Cykl 4: 1->FEDCBA -> 1

Graf bez cyklu Eulera:



Nie spełniony warunek 3 istnienia cyklu Eulera

# Twierdzenie Eulera, 1736

*"This question is so banal, but seemed to me worthy of attention in that neither geometry, nor algebra, nor even the art of counting was sufficient to solve it."*

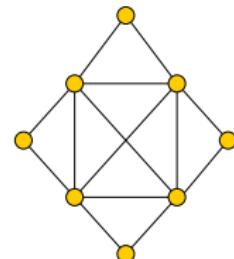
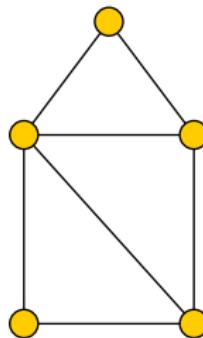
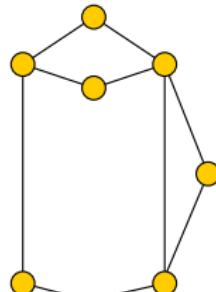
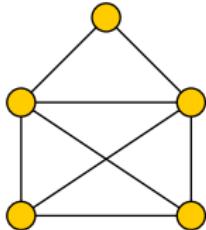
(Leonhard Euler, Marzec 1736)

## Twierdzenie

Spójny graf  $G$  (nieskierowany) ma cykl Eulera wtedy i tylko wtedy, gdy stopień każdego wierzchołka w  $G$  jest parzysty.

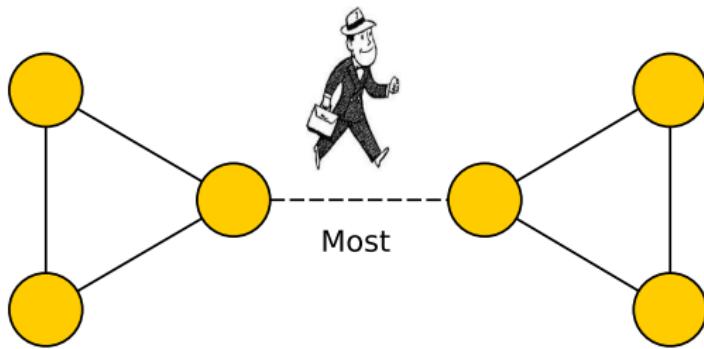
# Zagadki

- Sprawdź, w którym z poniższych grafów istnieje cykl lub droga Eulera. Jeśli w grafie istnieje cykl lub droga Eulera, to można ją narysować nie odrywając ołówka od papieru.



# Most

**Mostem** nazywamy taką krawędź grafu, której usunięcie zwiększa liczbę spójnych składowych tego grafu.



# Wyznaczanie cyklu Eulera

- Do wyznaczania cyklu Eulera służy algorytm **Fleury'ego**:
  - działa zarówno dla grafów skierowanych jak i nieskierowanych.
  - jest rekurencyjny.
  - zakłada, że graf jest Eulerowski.
- Algorytm Fleury'ego opiera się na prostej zasadzie: aby znaleźć cykl Eulera lub ścieżkę Eulera, mosty są ostatnimi krawędziami, które należy przejść.

# Algorytm Fleury'ego

- **Warunek wstępny:**

- Wejściowy graf jest grafem eulerowskim, czyli:
  - jest spójny
  - w przypadku poszukiwania ścież Eulera - posiada co najwyżej dwa wierzchołki o nieparzystym stopniu
  - w przypadku poszukiwania cyklu Eulera - wszystkie wierzchołki muszą mieć parzysty stopień

- **Warunek startowy:**

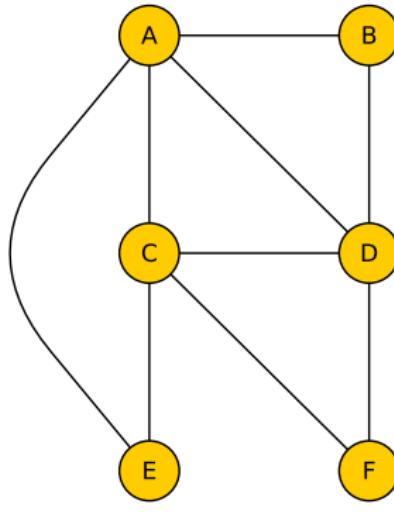
- W przypadku cyklu: wybierz dowolny wierzchołek
- W przypadku ścieżki: wybierz jeden z wierzchołków nieparzystych

# Algorytm Fleury'ego

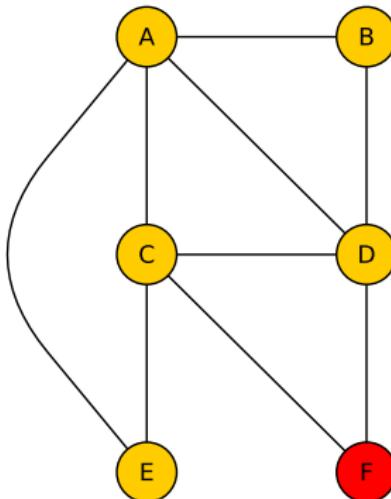
- **Kroki pośrednie:** Na każdym kroku, jeśli jest wybór, nie wybieraj **mostu** występującego w części grafu, która nie została jeszcze odwiedzona. Jednakże, jeśli jest tylko jeden wybór, to weź go.
- **Warunek Końcowy:** Kiedy nie można przechodzić już dalej, cykl (ścieżka) jest kompletna. [W przypadku cyklu, wracamy do wierzchołka wyjściowego; w przypadku ścieżki dochodzimy do drugiego wierzchołka o nieparzystym stopniu.]

# Algorytm Fleury'ego

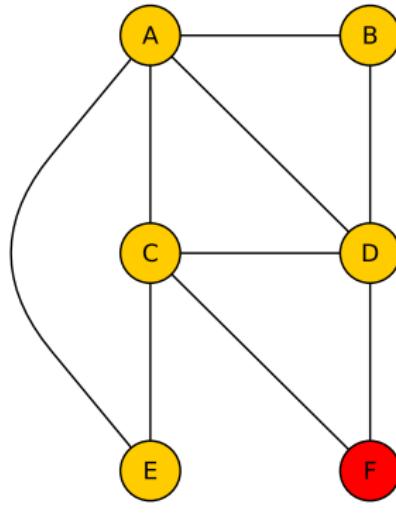
Graf eulerowski:



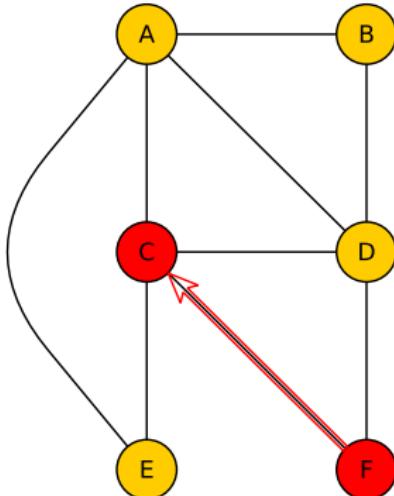
Wybieramy wierzchołek F jak startowy



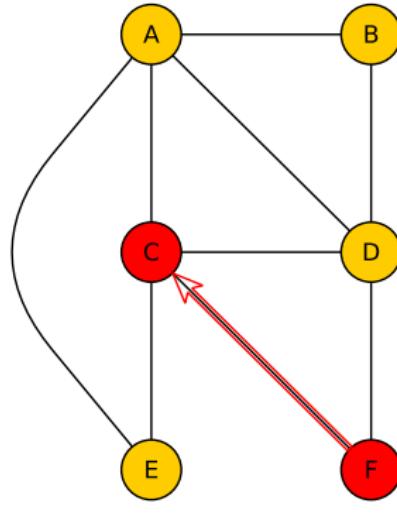
# Algorytm Fleury'ego



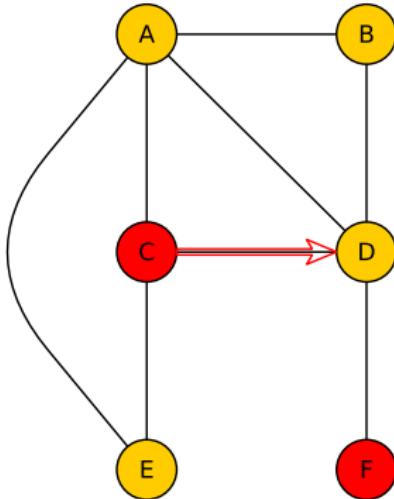
Idziemy z F do C



# Algorytm Fleury'ego

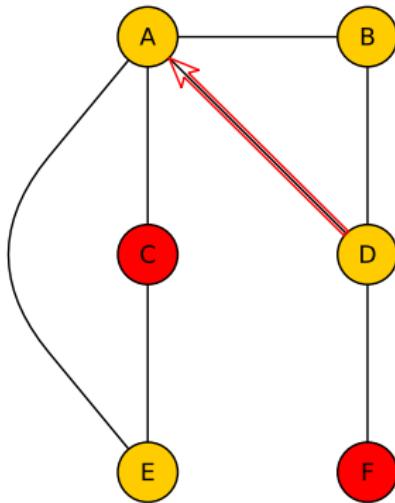
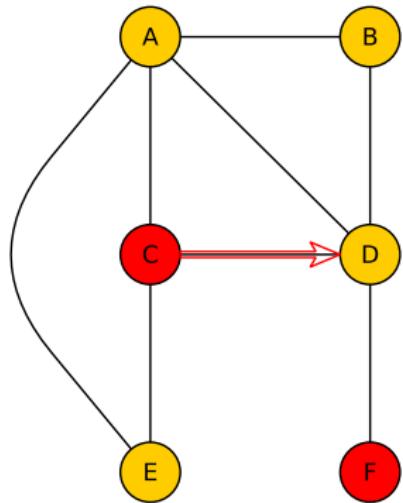


Idziemy z C do D



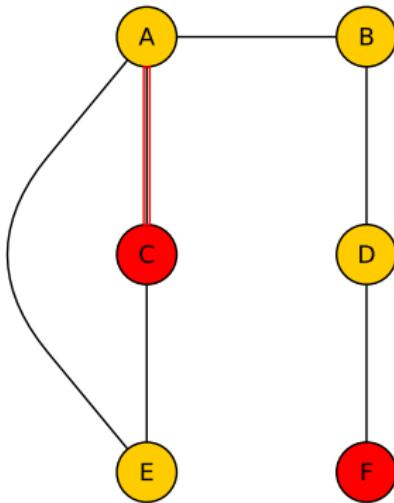
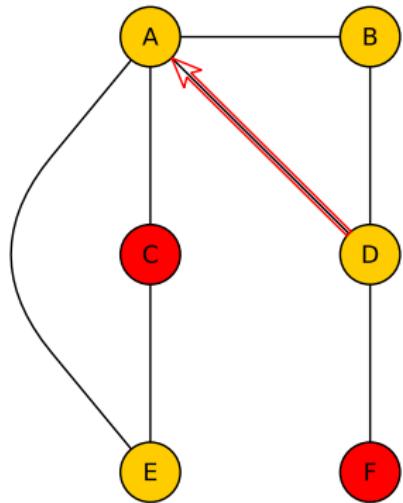
# Algorytm Fleury'ego

Idziemy z D do A. (Można również iść z D do B, ale nie można z D do F, bo DF jest mostem)



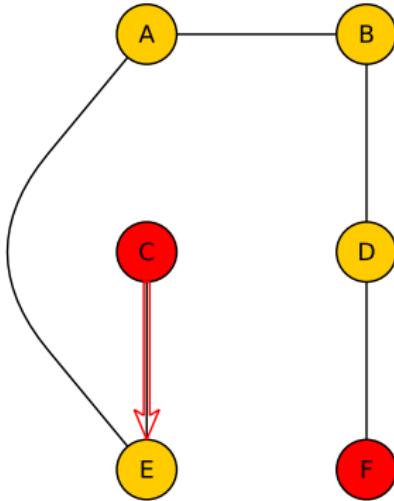
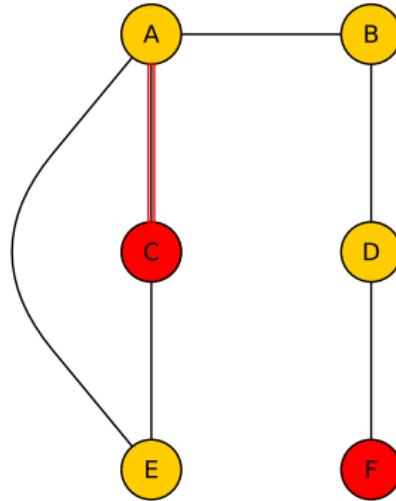
# Algorytm Fleury'ego

Idziemy z A do C. (Można również iść z A do E, ale nie można z A do B, bo AB jest mostem)



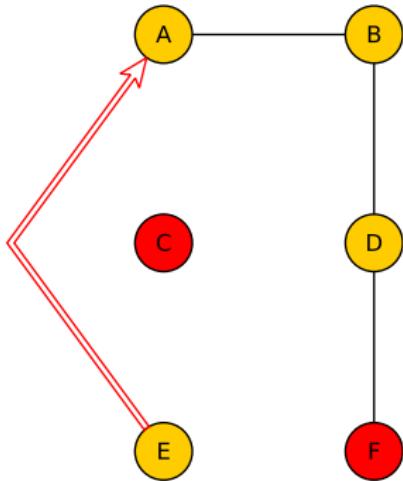
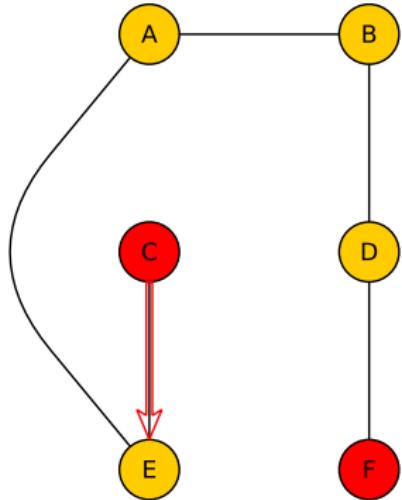
# Algorytm Fleury'ego

Idziemy z C do E. Nie ma wyboru.



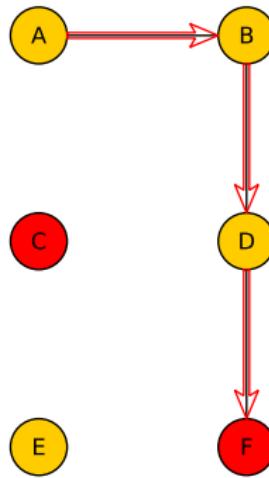
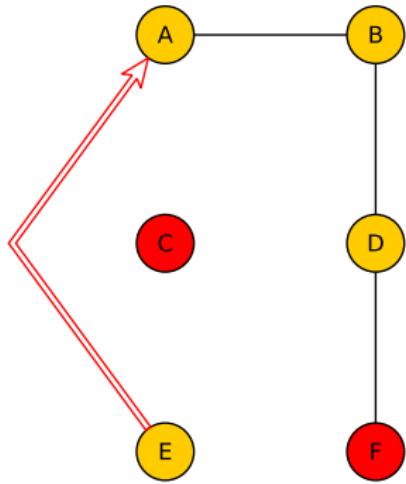
# Algorytm Fleury'ego

Idziemy z E do A. Nie ma wyboru.



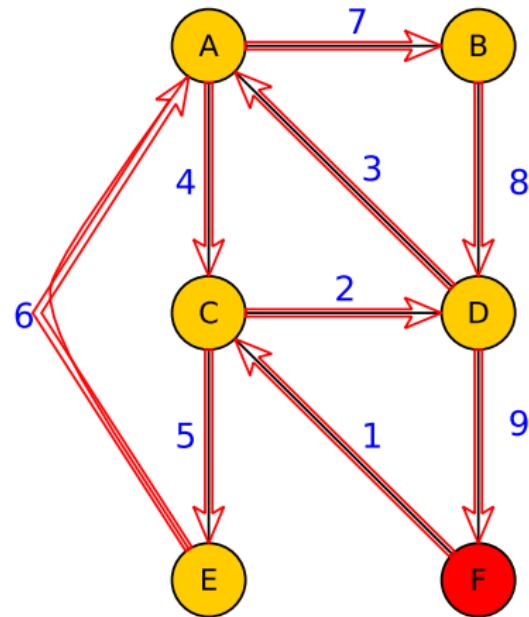
# Algorytm Fleury'ego

Idziemy z A do B, potem z B do D i na koniec z D do F. Jednoznaczna ścieżka.



# Algorytm Fleury'ego

Kolejne kroki algorytmu - podsumowanie.



# Algorytm Fleury'ego - Java I

```
/* Zakładamy, że:  
 (1) Graf jest reprezentowany przez  
 macierz sąsiedztwa  
 (2) w grafie są 2 lub 0 wierzchołki  
 o nieparzystym stopniu */  
public void printEulerTour() {  
     //Znajdź wierzchołek z nieparzystym stopniem  
     //Domyślnie startujemy od wierzchołka z indeksem 0  
     int u = 0;  
     for (int row = 0; row < V; ++row) {  
         int odd = 0;  
         for (int col = 0; col < V; ++col) {  
             if (adj[row][col] == true) odd++;  
         }  
     }
```

# Algorytm Fleury'ego - Java II

```
        if (odd % 2 == 1) {
            u = row; break;
        }
    }
// funkcja pomocnicza drukująca ścieżkę/cykl.
// poczynając od wierzchołka u
this.printEulerUtil(u);
}

void printEulerUtil(int u) {
    for (int v = 0; v < V; v++) {
        // jeśli krawędź u-v nie jest usunieta
        // i jest "poprawna"
        if (adj[u][v] == true &&
            this.isValidNextEdge(u, v) == true) {
```

# Algorytm Fleury'ego - Java III

```
        System.out.println(u + "-" + v) ;
        this.removeEdge(u,v);
        this.printEulerUtil(v);
    }
}

boolean isValidNextEdge(int u, int v)
{
    // liczba sasiadow wierzcholka u
    int count = 0;
    for (int i = 0; i < V; ++i) {
        if (adj[u][i] == true) count++;
    }
    //krawedz jest poprawna jesli zachodzi:
```

# Algorytm Fleury'ego - Java IV

```
//(1) v jest jedynym sasiadem wierzchołka u
if (adj[u][v] == true && count == 1)
    return true;
//Jesli u ma wielu sąsiadów, to należy
//sprawdzić, czy u-v jest mostem.
//liczba wierzchołków osiągalnych z u
int count1 = this.DFSCount(u);
this.removeEdge(u, v);
int count2 = this.DFSCount(u);
this.addEdge(u, v);
if (count1 > count2)
    return false; //u-v jest mostem
else
    return true;
}
```

# Algorytm Fleury'ego - Java V

```
// Funkcja bazujaca na algorytmie DFS
// zliczajaca wierzcholki osiągalne z wierzchołka a
int DFSCount(int a) {
    boolean[] visited = new boolean[V];
    for (int k = 0; k < V; ++k) visited[k] = false;
    Stack<Integer> s = new Stack<Integer>();
    visited[a] = true;
    s.push(a);
    int count = 1;
    while (!s.empty()) {
        int b = getUnVisitedVertex(s.peek(), visited);
        if (b == -1) {
            s.pop();
        } else {
```

# Algorytm Fleury'ego - Java VI

```
        visited[b] = true;
        count++;
        s.push(b);
    }
}
return count;
}
```

# Agorytm Fleury'ego - wykonanie I

```
bwozna@vostro:~/Graf-Fleury$ java Graph Koperta.in  
Wierzchołków: 5  
Krawędzi: 8  
Krawędzie grafu:  
0-1 0-2  
1-0 1-2 1-3 1-4  
2-0 2-1 2-3 2-4  
3-1 3-2 3-4  
4-1 4-2 4-3
```

Macierz sąsiedztwa:

```
false true true false false  
true false true true true  
true true false true true
```

# Agorytm Fleury'ego - wykonanie II

```
false true true false true  
false true true true false
```

Ścieżka startuje z:3

```
3-1 1-0 0-2 2-1 1-4  
4-2 2-3 3-4
```

```
bwozna@vostro:~/Graf-Fleury$ java Graph Euler0.in  
Wierzchołków: 6  
Krawędzi: 9  
Krawędzie grafu:  
0-1 0-4  
1-0 1-2 1-3 1-4  
2-1 2-3  
3-1 3-2 3-4 3-5
```

# Agorytm Fleury'ego - wykonanie III

4-0 4-1 4-3 4-5

5-3 5-4

Macierz sąsiedztwa:

```
false true false false true false  
true false true true true false  
false true false true false false  
false true true false true true  
true true false true false true  
false false false true true false
```

Ścieżka startuje z: 0

0-1 1-2 2-3 3-1

1-4 4-3 3-5 5-4 4-0

# Pseudokod algorytmu Fleury'ego I

Pseudokod na podstawie opracowania:

<https://www.geeksforgeeks.org/fleury-algorithm-for-printing-eulerian-path/>

```
/*
Funkcja drukuje ścieżkę/cykl Eulera.
Najpierw znajdowany jest nieparzysty wierzchołek
(jeśli istnieje), a następnie wywoływana jest
pomocnicza funkcja printEulerUtil(), aby wydrukować
ścieżkę. Zakłada się, że graf G=(V,E)
reprezentowany jest przez listy sąsiedztwa,
rozmiar V jest n, oraz że są 2 lub 0 wierzchołków
o stopniu nieparzystym.
adj[i] - lista sąsiedztwa dla wierzchołka i.
*/
```

# Pseudokod algorytmu Fleury'ego II

```
printEulerTour(Graph G)
{
    // Znajdź wierzchołek z nieparzystym stopniem
    // Domyślnie startujemy od wierzchołka
    // z indeksem 0.
    vertex u = 0;
    for (each v in V)
        if ( length of adj[v] is odd) then
            u = v;
            break;
        endif
    endfor
    printEulerUtil(G, u);
}
```

# Pseudokod algorytmu Fleury'ego III

```
//Drukuję ścieżkę/cykl Eulera poczynając
//od wierzchołka u
printEulerUtil(Graph G, vertex u)
{
    for (each v in adj[u]) do
        //jeśli krawędź u-v nie jest usunięta i jest
        //"poprawna" następną krawędzią.
        if (v != -1 and isValidNextEdge(G, u, v)) then
            print(u, "-", v) ;
            removeEdge(G, u, v);
            printEulerUtil(G, v);
        endif
    endfor
}
```

# Pseudokod algorytmu Fleury'ego IV

```
// Funkcja usuwa krawędź u-v z grafu.  
// Usuwa krawędź, zastępując wartość  
// sąsiadniego wierzchołka wartością -1.  
removeEdge(Graph G, vertex u, vertex v)  
{  
    //Znajdź v na liście sąsiedztwa u i zastąp go -1  
    for (each i in adj[u]) do  
        if (i==v) then i = -1;  
    endfor  
    //Znajdź u na liście sąsiedztwa v i zastąp go -1  
    for (each i in adj[v]) do  
        if (i==u) then i = -1;  
    endfor  
}
```

# Pseudokod algorytmu Fleury'ego V

```
//Funkcja sprawdzająca, czy krawędź u-v może
//być uważana za następną krawędź
//w cyklu/ścieżce Eulera
bool isValidNextEdge(Graph G, vertex u, vertex v)
{
    //liczba sąsiadów wierzchołka u
    int count = 0;
    for (each i in adj[u]) do
        if (i != -1) then
            count=count+1;
    endif
    endfor
    //Krawędź u-v jest poprawna, jeśli zachodzi
    //jeden z następujących przypadków:
    // 1) v jest jedynym sąsiadem wierzchołka u
```

# Pseudokod algorytmu Fleury'ego VI

```
if (count == 1) then
    return true;
endif
// 2) Jeśli u ma wielu przyległych sąsiadów, to
// wykonaj następujące kroki, aby sprawdzić,
// czy u-v jest mostem
// 2.a) liczba wierzchołków osiągalnych z u
boolvector visited[n];
for(each v in V) do
    visited[v]=false;
endfor;
count1 = DFSCount (G,u,visited);
// 2.b) Usuń krawędź (u,v), a po jej usunięciu,
// policz liczbę wierzchołków osiągalnych z u
removeEdge (G,u,v);
```

# Pseudokod algorytmu Fleury'ego VII

```
for(each v in V) do
    visited[v]=false;
endfor;
count2 = DFSCount (G, u, visited);
// 2.c) Dodaj krawędź z powrotem do grafu.
addEdge(u, v);
// 2.d) Jeśli count1 > count2, to
// krawędź (u, v) jest mostem
if (count1 > count2) then
    return false;
else
    return true;
endif
}
```

# Pseudokod algorytmu Fleury'ego VIII

```
// Funkcja bazująca na algorytmie DFS
// zliczająca wierzchołki osiągalne z wierzchołka v
int DFSCount(Graph G, verex v, boolvector visited)
{
    visited[v] = true;
    int count = 1;
    for (each i in adj[v])
        if (i != -1 and visited[i]==false) then
            count = count+ DFSCount (G,i, visited);
        endif
    endfor
    return count;
}
```

# Złożoność czasowa algorytmu Fleury'ego

- Złożoność czasowa powyższej implementacji wynosi  $O(|V|^4)$ .  
**Dlaczego ?** Funkcja `printEulerUtil()` działa jak DFS i wywołuje metodę `isValidNextEdge()`, która również wykonuje DFS dwa razy. Złożoność czasowa DFS dla reprezentacji macierzowej wynosi  $O(|V|^2)$ . Zatem, ogólna złożoność czasowa wynosi  $O(|V|^4)$ .
- Jeżeli zmodyfikujemy powyższą implementację do pracy z grafami reprezentowanymi przez listy sąsiedztwa, to jej złożoność czasowa wyniesie  $O((|V| + |E|)^2)$ .  
**Dlaczego ?** Funkcja `printEulerUtil()` działa jak DFS i wywołuje metodę `isValidNextEdge()`, która również wykonuje DFS dwa razy. Złożoność czasowa DFS dla reprezentacji listy sąsiedztwa wynosi  $O(|V| + |E|)$ . Zatem ogólna złożoność czasowa wynosi  $O((|V| + |E|)^2)$ .

# Cykl Eulera - Zastosowania

- Rysowanie/wycinanie figur przy pomocy plotera
- Problem chińskiego listonosza – W roku 1962 chiński matematyk Mei-Ko Kwan sformułował następujący problem:  
*Listonosz roznosząc listy musi przejść przez wszystkie ulice w swojej dzielnicy co najmniej jeden raz i wrócić na pocztę. Ponieważ jest człowiekiem leniwym, chciałby mieć jak najkrótszą do przejścia trasę. Znalezienie takiej trasy jest problemem, który nazwano problemem chińskiego listonosza (ang. Chinese postman problem - CPP).*

# Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

b.wozna@ujd.edu.pl

Wykład 11 i 12

# Spis treści

- 1 Cykle i ścieżki Hamiltona
- 2 Grafy pełne
- 3 Problem komiwojażera

# Spis treści

1 Cykle i ścieżki Hamiltona

2 Grafy pełne

3 Problem komiwojażera

# Cykle i ścieżki Hamiltona - definicje

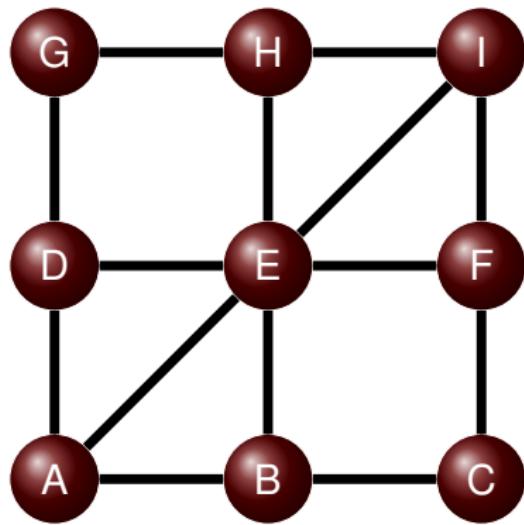
- Ścieżki Eulera i cykle Eulera to ścieżki i krawędzie, które przechodzą przez każdą krawędź grafu.
- Co, jeśli celem jest odwiedzenie każdego wierzchołka zamiast każdej krawędzi?

**Ścieżka Hamiltona** to ścieżka, która zawiera **każdy wierzchołek** grafu dokładnie jeden raz.

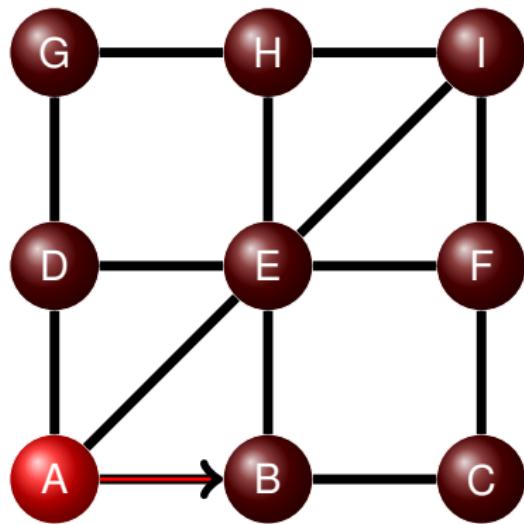
**Cykla Hamiltona** to cykl, która zawiera **każdy wierzchołek** grafu dokładnie jeden raz.

**Uwaga !** Ścieżka oznacza, że początkowe i końcowe wierzchołki są różne; Cykl oznacza, że są one takie same.

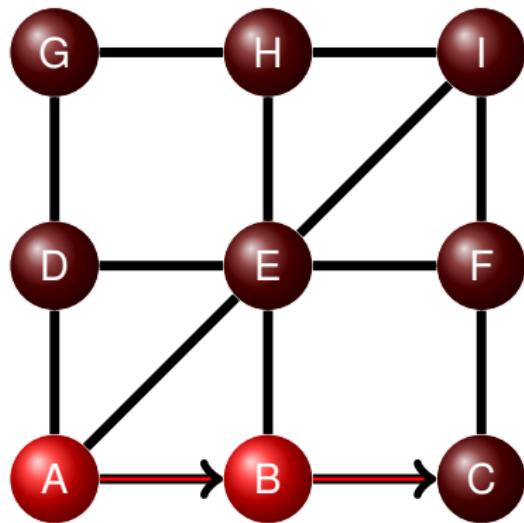
# Cykl Hamiltona - przykład



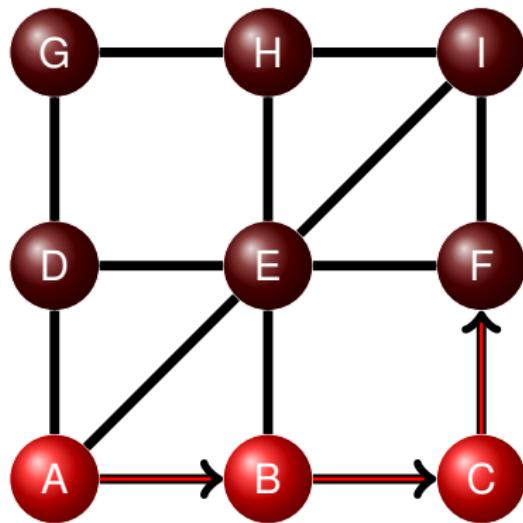
# Cykl Hamiltona - przykład I



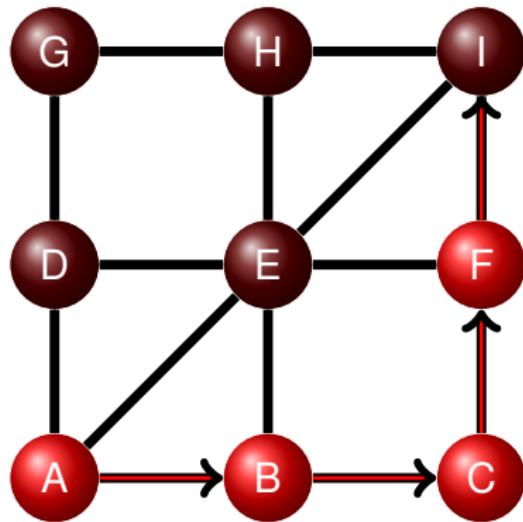
# Cykl Hamiltona - przykład II



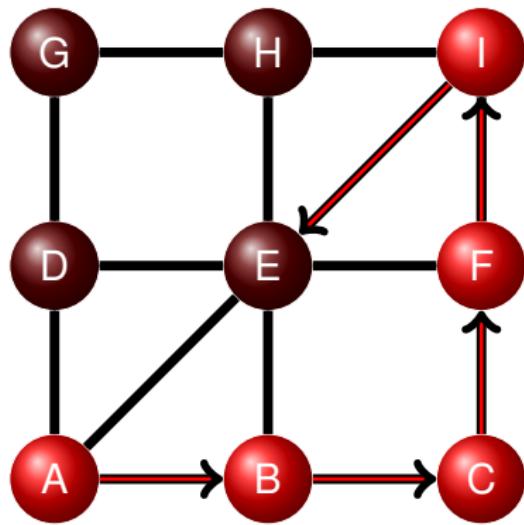
# Cykl Hamiltona - przykład III



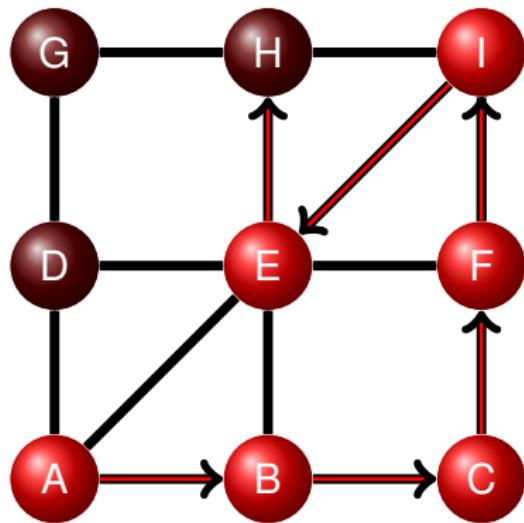
# Cykl Hamiltona - przykład IV



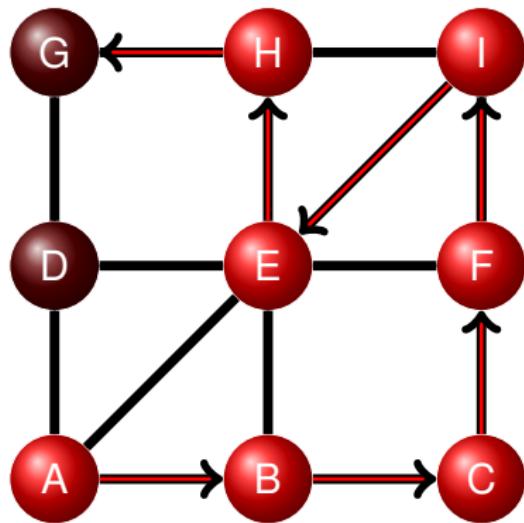
## Cykl Hamiltona - przykład V



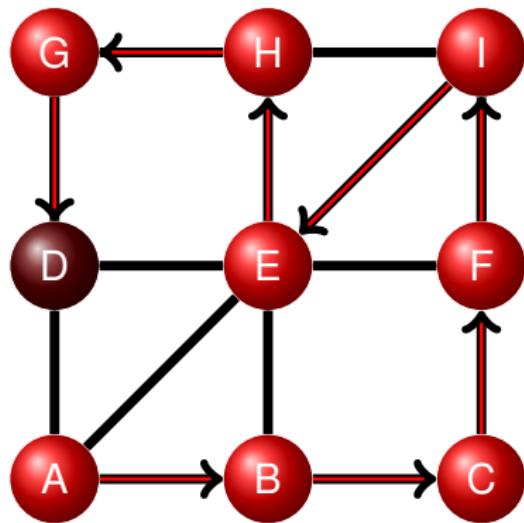
# Cykl Hamiltona - przykład VI



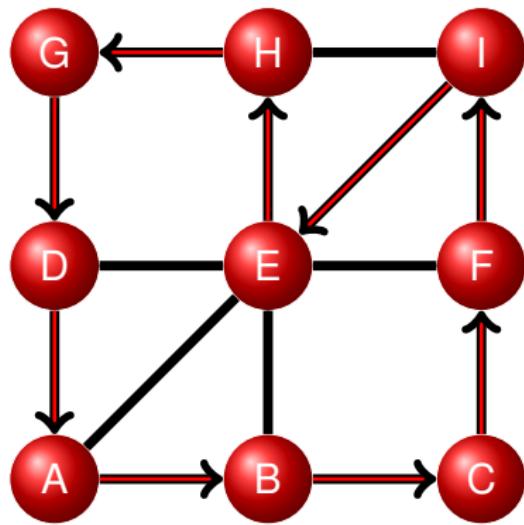
# Cykl Hamiltona - przykład VII



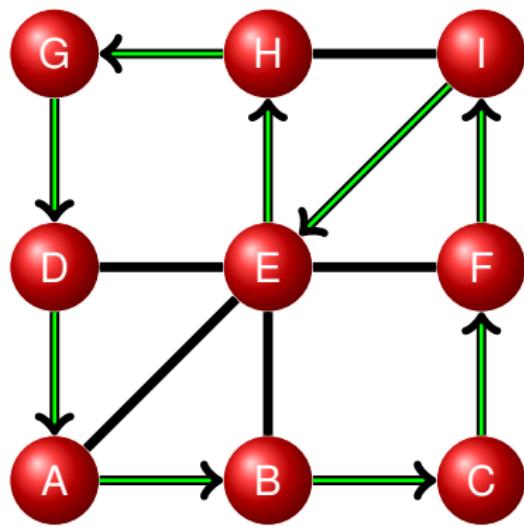
# Cykl Hamiltona - przykład VIII



## Cykl Hamiltona - przykład IX



# Cykl Hamiltona - przykład X



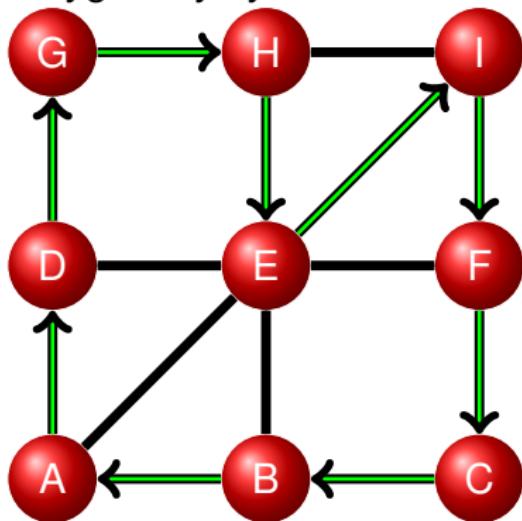
# Cykl Hamiltona

Zmiana wierzchołka początkowego **nie zmienia** cyklu Hamiltona, ponieważ te same krawędzie są odwiedzane w tych samych kierunkach.

- A,B,C,F,I,E,H,G,D,A
- B,C,F,I,E,H,G,D,A,B
- C,F,I,E,H,G,D,A,B,C
- F,I,E,H,G,D,A,B,C,F
- I,E,H,G,D,A,B,C,F,I
- E,H,G,D,A,B,C,F,I,E
- H,G,D,A,B,C,F,I,E,H
- G,D,A,B,C,F,I,E,H,G
- D,A,B,C,F,I,E,H,G,D

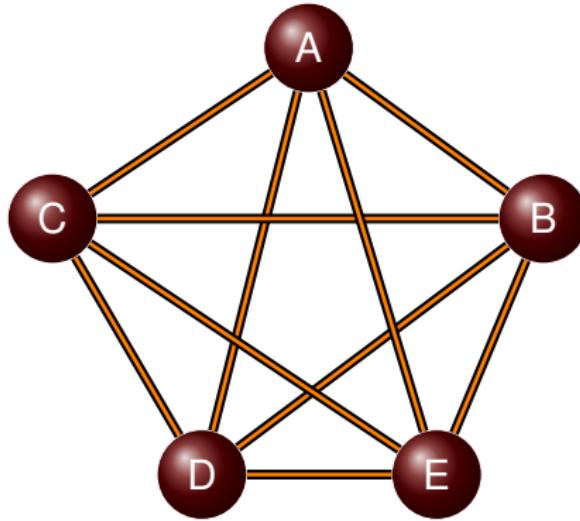
# Cykl Hamiltona

Można również odwrócić cykl Hamiltona w jego **odbicie lustrzane** poprzez odwrócenie kierunku. Obraz lustrzany używa tych samych krawędzi, ale do tyłu, zatem nie jest uważany za taki sam jak oryginalny cykl Hamiltona.



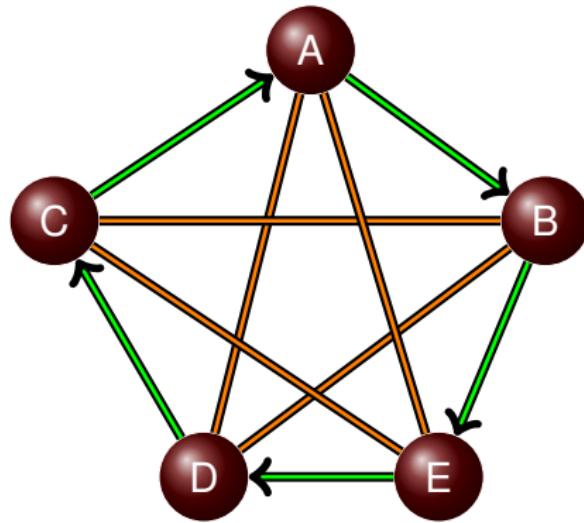
# Hamilton vs. Euler I

- Czy graf może posiadać zarówno cykl Hamiltona jak i cykl Eulera?



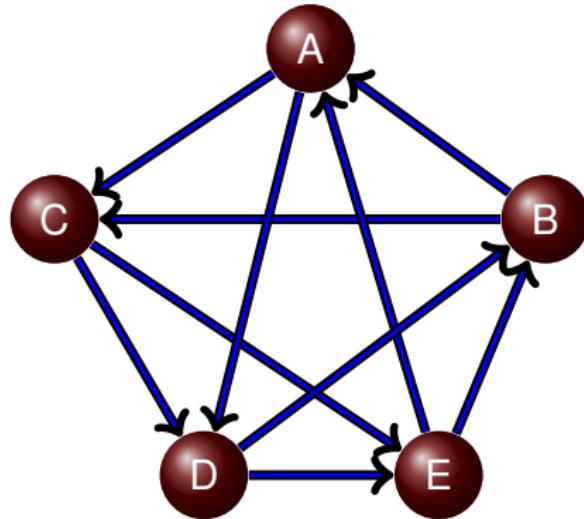
# Hamilton vs. Euler II

Cykl Hamiltona: (A,B,E,D,C,A)



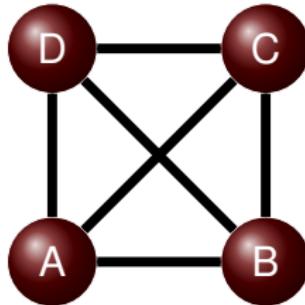
# Hamilton vs. Euler III

Cykl Eulera: (A,C,D,E,B,C,E,A,D,B,A)



# Hamilton vs. Euler

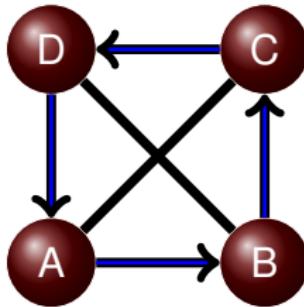
- Czy graf może mieć cykl Hamiltona, ale nie mieć cyklu Eulera?



# Hamilton vs. Euler

- Czy graf może mieć cykl Hamiltona, ale nie mieć cyklu Eulera?

Cykl Hamiltona: (A,B,C,D,A)

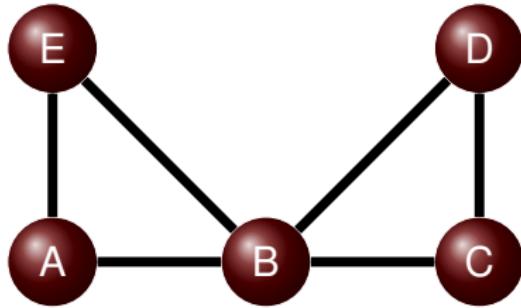


Cykl Eulera: brak – dlaczego ?

stopień (A) = stopień (B) = stopień (C) = stopień (D) = 3

# Hamilton vs. Euler

- Czy graf może mieć cykl Eulera, ale nie mieć cyklu Hamiltona?

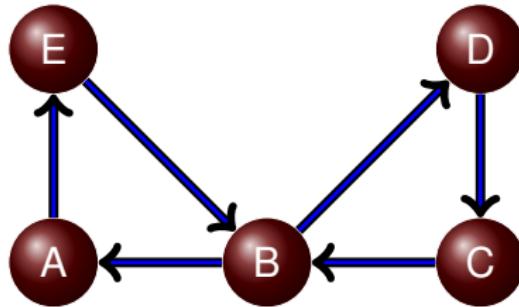


# Hamilton vs. Euler

- Czy graf może mieć cykl Eulera, ale nie mieć cyklu Hamiltona?

Cykl Hamiltona: Brak

Cykl Eulera: A-E-B-D-C-B-A

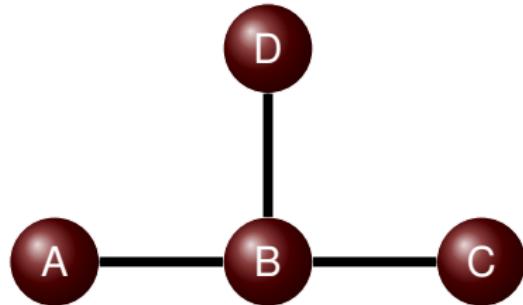


# Hamilton vs. Euler

- Czy graf może nie mieć ani cyklu Eulera, ani cyklu Hamiltona?

# Hamilton vs. Euler

- Czy graf może nie mieć ani cyklu Eulera, ani cyklu Hamiltona?



# Hamilton vs. Euler

**Wniosek:** to, czy graf zawiera cykl Hamiltona, czy też nie, nie mówi nic o tym, czy graf ten ma cykl Eulera, i odwrotnie. To samo dotyczy ścieżek Hamiltona / Eulera.

# Jaki graf posiada cykl Hamiltona ?

- Jak określić, czy graf ma ścieżkę lub cykl Eulera – wystarczy policzyć liczbę wierzchołków o nieparzystym stopniu.

# Jaki graf posiada cykl Hamiltona ?

- Jak określić, czy graf ma ścieżkę lub cykl Eulera – wystarczy policzyć liczbę wierzchołków o nieparzystym stopniu.
- Niestety nie ma prostego sposobu na stwierdzenie, czy dany graf zawiera ścieżkę lub cykl Hamiltona.

# Jaki graf posiada cykl Hamiltona ?

- Jak określić, czy graf ma ścieżkę lub cykl Eulera – wystarczy policzyć liczbę wierzchołków o nieparzystym stopniu.
- Niestety nie ma prostego sposobu na stwierdzenie, czy dany graf zawiera ścieżkę lub cykl Hamiltona.
- Zamiast pytać, czy dany graf ma cykl Hamiltona, lepiej rozważyć grafy z dużą ilością cykli Hamiltona i spróbować znaleźć najkrótszy z nich.

# Spis treści

1 Cykle i ścieżki Hamiltona

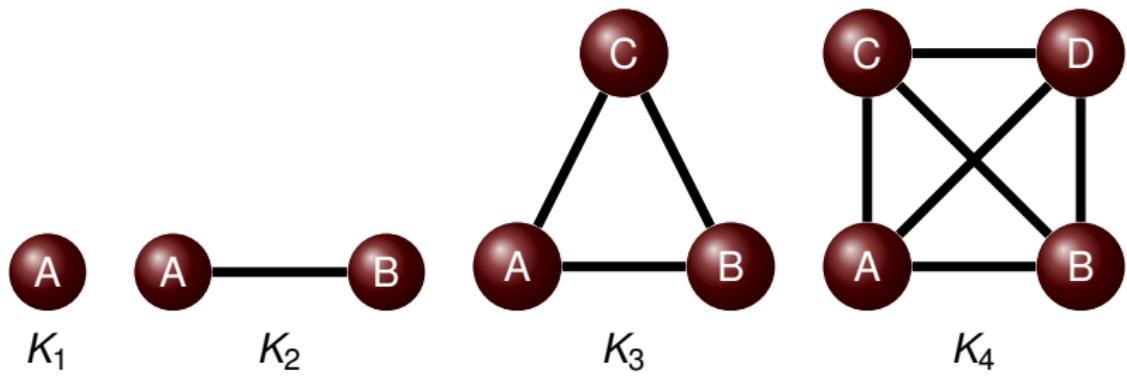
2 Grały pełne

3 Problem komiwojażera

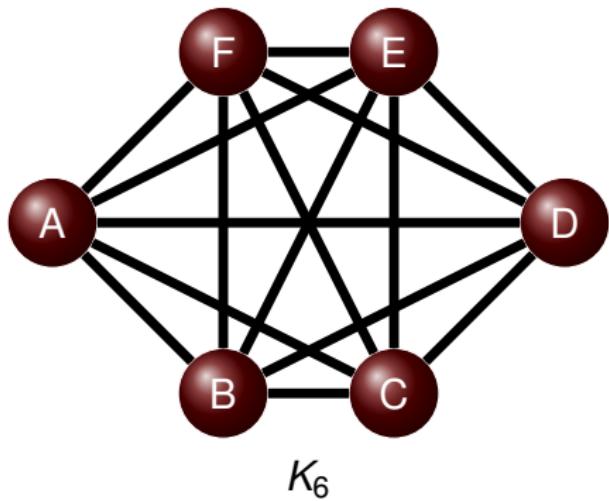
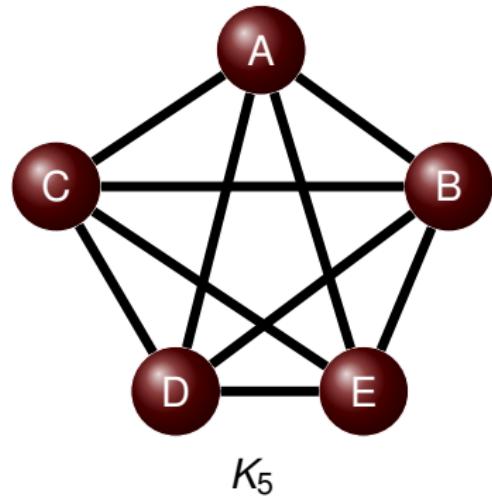
# Grafy pełne - definicja

- **Graf pełny** - graf nieskierowany, w którym każda para wierzchołków jest połączona krawędzią.
  - Graf pełny nie zawiera pętli.
  - W grafie pełnym każde dwa wierzchołki wspólnie dzielą dokładnie jedną krawędź.
- Graf pełny o  $n$  wierzchołkach oznacza się przez  $K_n$ .
- Graf pełny o  $n$  wierzchołkach posiada  $\frac{n \cdot (n-1)}{2}$  krawędzi.

# Grały pełne - przykłady



# Grały pełne - przykłady



# Ile różnych cykli Hamiltona ma graf $K_n$ ? I

- Cykl Hamiltona można przedstawić za pomocą uporządkowanej listy wierzchołków.
  - Pierwszy i ostatni wierzchołek na liście muszą być takie same. Wszystkie pozostałe wierzchołki pojawiają się dokładnie raz.
  - Pierwszy / ostatni wierzchołek nazywany jest **punktem odniesienia**.
- Zmiana punktu odniesienia **nie zmienia** cyklu Hamiltona, ponieważ te same krawędzie są odwiedzane w tych samych kierunkach. Przykładowo wszystkie poniższe trasy reprezentują ten sam cykl Hamiltona w  $K_4$ :

A,C,D,B,A (punkt odniesienia: A)

B,A,C,D,B (punkt odniesienia: B)

D,B,A,C,D (punkt odniesienia: D)

C,D,B,A,C (punkt odniesienia: C)

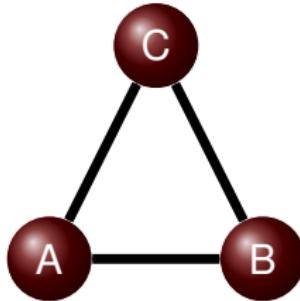
# Ile różnych cykli Hamiltona ma graf $K_n$ ? II

- Ponieważ  $K_n$  ma  $n$  różnych punktów odniesienia, to każdy cykl Hamiltona w  $K_n$  może być opisany przez dokładnie  $n$  różnych tras.

- Dla każdego  $n \geq 3$  liczba cykli Hamiltona w grafie  $K_n$  jest:

$$(n - 1) \times (n - 2) \times \dots \times 2 \times 1 = (n - 1)!$$

- Liczba tras Hamiltona w grafie  $K_n$  jest:  $n \times (n - 1)! = n!$
- Przykładowo, dla  $K_3$  mamy 2 cykle:



(A,B,C,A), (A,C,B,A), ale 6 różnych tras:  
 (A,B,C,A), (B,C,A,B), (C,A,B,C), (A,C,B,A),  
 (C,B,A,C), (B,A,C,B).

# Ile różnych cykli Hamiltona ma graf $K_n$ ? III

Wierzchołki $n$	Krawędzie $n(n - 1)/2$	Cykle Hamiltona $(n - 1)!$
1	0	
2	1	
3	3	2
4	6	6
5	10	24
6	15	120
7	21	620
...	...	...
16	120	1307674368000

# Spis treści

1 Cykle i ścieżki Hamiltona

2 Grafy pełne

3 Problem komiwojażera

# Problem komiwojażera

**Problem komiwojażera** (ang. travelling salesman problem, TSP):

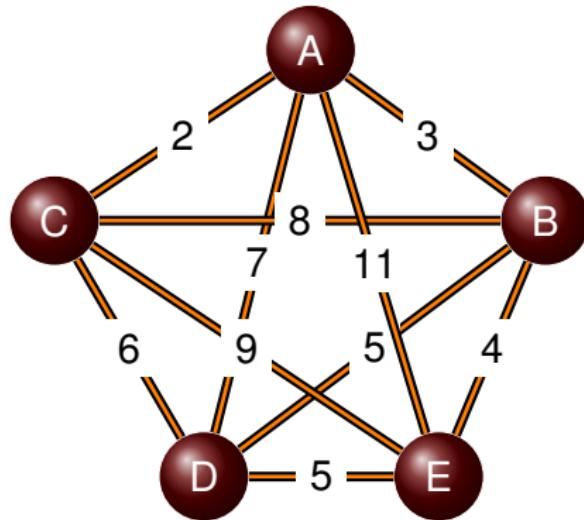
Franek, akwizytor, musi odwiedzić każde z kilku miast (powiedzmy, wszystkie miasta wojewódzkie w Polsce – tj. 16 miast). Franek chciałby, aby jego podróż był jak najkrótsza. **W jakiej kolejności Franek powinien odwiedzić 16 miast wojewódzkich?**

TSP pojawia się w wielu innych kontekstach:

- Prom kosmiczny musi sprowadzić z orbity kilka sztucznych satelitów i innych ładunków. Paliwo w kosmosie jest bardzo drogie.
- Gminny autobus szkolny musi dowieźć gimnazjalistów z kilku wiosek do Gimnazjum. Czas przejazdu jest tutaj bardzo istotny.

# TSP jako problem grafowy I

- Założymy, że mamy graf z funkcją wagi, tzn. graf na którym każda krawędź ma wagę (reprezentującą jego koszt, czas lub odległość).



# TSP jako problem grafowy II

- Problem komiwojażera polega na znalezieniu ścieżki lub cyklu, który:
  - zawiera każdy wierzchołek grafu; i
  - całkowita waga ścieżki/cyklu jest najmniejsza z możliwych.
- Innymi słowy:

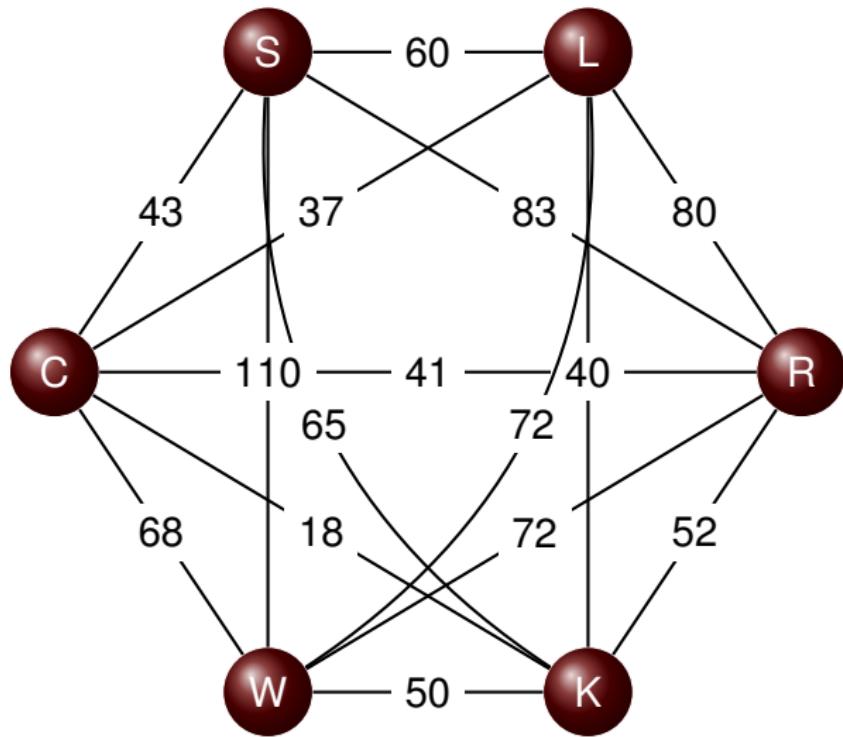
Problem komiwojażera polega na znalezieniu **minimalnego cyklu Hamiltona w pełnym grafie ważonym**.

# Problem komiwojażera - Przykład I

## Przykład

Dany jest zbiór miejscowości (*Częstochowa (C)*, *Wieluń (W)*, *Kłobuck(K)*, *Radomsko (R)*, *Siewierz (S)*, *Lubliniec (L)*) oraz odległości między nimi. Znaleźć drogę zamkniętą, która ma najkrótszą długość oraz przechodzi przez każdą miejscowości dokładnie jeden raz.

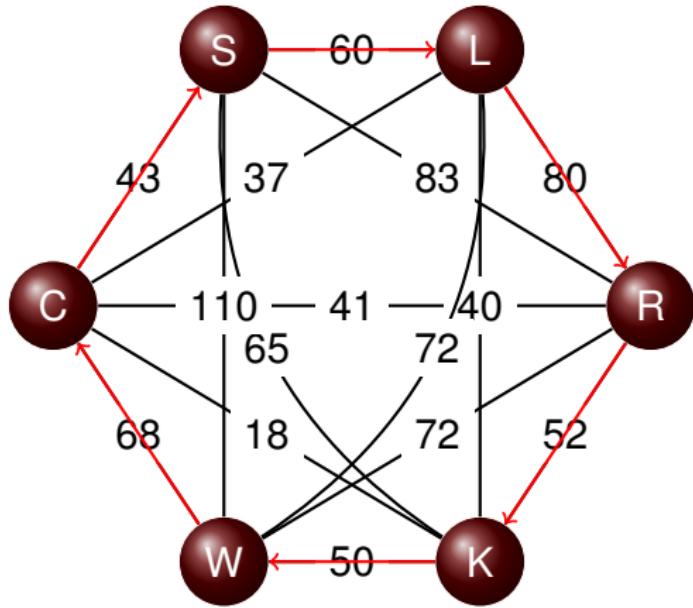
# Problem komiwojażera - Przykład II



# Problem komiwojażera - Przykład III

Cykl Hamiltona: (C,S,L,R,K,W,C).

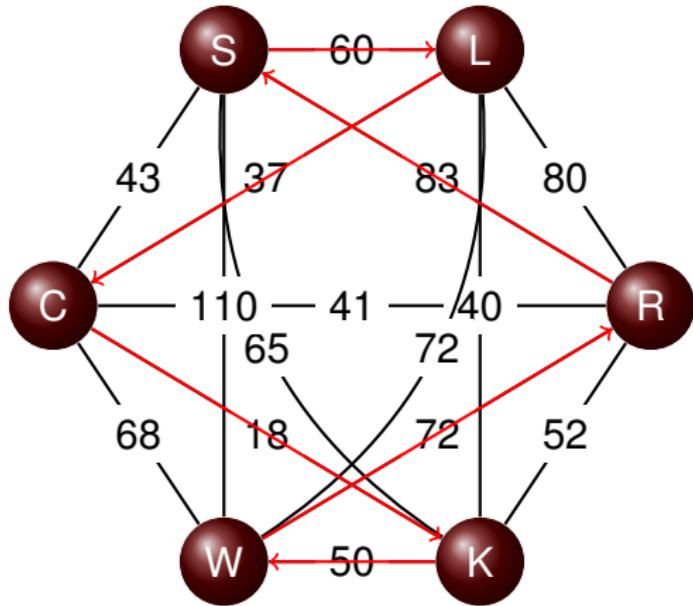
Odległość:  $43+60+80+52+50+68 = 353$



# Problem komiwojażera - Przykład IV

Cykl Hamiltona: (C,K,W,R,S,L,C).

Odległość:  $18+50+72+83+60+37=320$



# Problem komiwojażera - Przykład V

	C	W	K	R	S	L
Częstochowa(C)	0	68	18	41	43	37
Wieluń (W)	68	0	50	72	110	72
Kłobuck(K)	18	50	0	52	65	40
Radomsko (R)	41	72	52	0	83	80
Siewierz (S)	43	110	65	83	0	60
Lubliniec (L)	37	72	40	80	60	0

- Odległość (C,S,L,R,K,W,C):  $43+60+80+52+50+68 = \textcolor{blue}{353}$ .
- Odległość (C,K,W,R,S,L,C):  $18+50+72+83+60+37=\textcolor{red}{320}$ .
- Odległość (C,S,K,L,W,R,C):  $43+65+40+72+72+41= \textcolor{blue}{333}$ .
- Liczba wierzchołków (miast) jest 6, zatem **liczba cykli Hamiltona** wynosi:  $\textcolor{red}{5! = 5 \times 4 \times 3 \times 2 \times 1 = 120}$ .

# Problem komiwojażera - Przykład VI

- Można wymienić wszystkie cykle, obliczyć dla nich odległości, a następnie wybrać tę najmniejszą – czyli wykonać **algorytm siłowy**, tzw. **Brute-Force**.

- C, W, K, R, L, S, C : 353
- C, W, K, R, S, L, C : 350
- C, W, K, L, R, S, C : 364
- C, W, K, L, S, R, C : 342
- C, W, K, S, R, L, C : 383
- C, W, K, S, L, R, C : 364
- C, W, R, K, L, S, C : 335
- C, W, R, K, S, L, C : 354
- C, W, R, L, K, S, C : 368
- C, W, R, L, S, K, C : 363
- C, W, R, S, K, L, C : 365
- C, W, R, S, L, K, C : 341
- C, W, L, K, R, S, C : 358

# Problem komiwojażera - Przykład VII

- C, W, L, K, S, R, C : 369
- C, W, L, R, K, S, C : 380
- C, W, L, R, S, K, C : 386
- C, W, L, S, K, R, C : 358
- C, W, L, S, R, K, C : 353
- C, W, S, K, R, L, C : 412
- C, W, S, K, L, R, C : 404
- C, W, S, R, K, L, C : 390
- C, W, S, R, L, K, C : 399
- C, W, S, L, K, R, C : 371
- C, W, S, L, R, K, C : 388
- C, K, W, R, L, S, C : 323
- C, K, W, R, S, L, C : 320
- C, K, W, L, R, S, C : 346
- C, K, W, L, S, R, C : 324
- C, K, W, S, R, L, C : 378

# Problem komiwojażera - Przykład VIII

- C, K, W, S, L, R, C : 359
- C, K, R, W, L, S, C : 317
- C, K, R, W, S, L, C : 349
- C, K, R, L, W, S, C : 375
- C, K, R, L, S, W, C : 388
- C, K, R, S, W, L, C : 372
- C, K, R, S, L, W, C : 353
- C, K, L, W, R, S, C : 328
- C, K, L, W, S, R, C : 364
- C, K, L, R, W, S, C : 363
- C, K, L, R, S, W, C : 399
- C, K, L, S, W, R, C : 341
- C, K, L, S, R, W, C : 341
- C, K, S, W, R, L, C : 382
- C, K, S, W, L, R, C : 386
- C, K, S, R, W, L, C : 347

# Problem komiwojażera - Przykład IX

- C, K, S, R, L, W, C : 386
- C, K, S, L, W, R, C : 328
- C, K, S, L, R, W, C : 363
- **C, R, W, K, L, S, C : 306**
- C, R, W, K, S, L, C : 325
- C, R, W, L, K, S, C : 333
- C, R, W, L, S, K, C : 328
- C, R, W, S, K, L, C : 365
- C, R, W, S, L, K, C : 341
- C, R, K, W, L, S, C : 318
- C, R, K, W, S, L, C : 350
- C, R, K, L, W, S, C : 358
- C, R, K, L, S, W, C : 371
- C, R, K, S, W, L, C : 377
- C, R, K, S, L, W, C : 358
- C, R, L, W, K, S, C : 351

# Problem komiwojażera - Przykład X

- C, R, L, W, S, K, C : 386
- C, R, L, K, W, S, C : 364
- C, R, L, K, S, W, C : 404
- C, R, L, S, W, K, C : 359
- C, R, L, S, K, W, C : 364
- C, R, S, W, K, L, C : 361
- C, R, S, W, L, K, C : 364
- C, R, S, K, W, L, C : 348
- C, R, S, K, L, W, C : 369
- C, R, S, L, W, K, C : 324
- C, R, S, L, K, W, C : 342
- C, L, W, K, R, S, C : 337
- C, L, W, K, S, R, C : 348
- C, L, W, R, K, S, C : 341
- C, L, W, R, S, K, C : 347
- C, L, W, S, K, R, C : 377

# Problem komiwojażera - Przykład XI

- C, L, W, S, R, K, C : 372
- C, L, K, W, R, S, C : 325
- C, L, K, W, S, R, C : 361
- C, L, K, R, W, S, C : 354
- C, L, K, R, S, W, C : 390
- C, L, K, S, W, R, C : 365
- C, L, K, S, R, W, C : 365
- C, L, R, W, K, S, C : 347
- C, L, R, W, S, K, C : 382
- C, L, R, K, W, S, C : 372
- C, L, R, K, S, W, C : 412
- C, L, R, S, W, K, C : 378
- C, L, R, S, K, W, C : 383
- C, L, S, W, K, R, C : 350
- C, L, S, W, R, K, C : 349
- C, L, S, K, W, R, C : 325

# Problem komiwojażera - Przykład XII

- C, L, S, K, R, W, C : 354
- C, L, S, R, W, K, C : 320
- C, L, S, R, K, W, C : 350
- C, S, W, K, R, L, C : 372
- C, S, W, K, L, R, C : 364
- C, S, W, R, K, L, C : 354
- C, S, W, R, L, K, C : 363
- C, S, W, L, K, R, C : 358
- C, S, W, L, R, K, C : 375
- C, S, K, W, R, L, C : 347
- C, S, K, W, L, R, C : 351
- C, S, K, R, W, L, C : 341
- C, S, K, R, L, W, C : 380
- C, S, K, L, W, R, C : 333
- C, S, K, L, R, W, C : 368
- C, S, R, W, K, L, C : 325

## Problem komiwojażera - Przykład XIII

- C, S, R, W, L, K, C : 328
  - C, S, R, K, W, L, C : 337
  - C, S, R, K, L, W, C : 358
  - C, S, R, L, W, K, C : 346
  - C, S, R, L, K, W, C : 364
  - C, S, L, W, K, R, C : 318
  - C, S, L, W, R, K, C : 317
  - C, S, L, K, W, R, C : 306
  - C, S, L, K, R, W, C : 335
  - C, S, L, R, W, K, C : 323
  - C, S, L, R, K, W, C : 353
- Algorytm Brute-Force jest **optymalny**: gwarantuje znalezienie rozwiązania.
- C, S, L, K, W, R, C : 306
  - C, R, W, K, L, S, C : 306

## Problem komiwojażera - Przykład XIV

- Algorytm Brute-Force jest **nieefektywny**: musi sprawdzić **wszystkie** cykle Hamiltona, tj.  $(n - 1)!$ , a to może zająć dużo czasu, gdyż wartości funkcji silnia dla kolejnych  $n$  rosną bardzo szybko:

n	$n!$
1	1
5	120
10	3 628 800
15	$1\ 307\ 674\ 368\ 000 = 1,307674368 \cdot 10^{12}$
50	$3\ 041\ 409\ 320 \cdot 10^{64}$

# Problem komiwojażera - Przykład XV

Używając komputera wykonującego  $10^9$  **operacji na sekundę** oraz **algorytmu typu Brute-Force** wyznaczającego najkrótszą trasę dla problemu komiwojażera, rozwiązanie dla przypadku, gdy komiwojażer chce odwiedzić **16 miast wojewódzkich** – wszystkich możliwych wyborów jest **15!** – zajęłoby około **21 min.**

- Czy istnieje lepszy sposób rozwiązania problemu komiwojażera?
- To znaczy, czy istnieje optymalny algorytm, który również jest wydajny?

# Problem komiwojażera bez algorytmu siłowego I

Algorytm najbliższego sąsiada.

**Idea:** Na każdym etapie podróży należy wybrać najbliższy wierzchołek, który nie był jeszcze odwiedzony.

	C	W	K	R	S	L
Częstochowa(C)	0	68	18	41	43	37
Wieluń (W)	68	0	50	72	110	72
Kłobuck(K)	18	50	0	52	65	40
Radomsko (R)	41	72	52	0	83	80
Siewierz (S)	43	110	65	83	0	60
Lubliniec (L)	37	72	40	80	60	0

# Problem komiwojażera bez algorytmu siłowego II

	C	W	K	R	S	L
Częstochowa(C)	0	68	18	41	43	37
Wieluń (W)	68	0	50	72	110	72
Kłobuck(K)	18	50	0	52	65	40
Radomsko (R)	41	72	52	0	83	80
Siewierz (S)	43	110	65	83	0	60
Lubliniec (L)	37	72	40	80	60	0

- Jeśli komiwojażer wyrusza z Częstochowy, to najbliższym celem jest Kłobuck.
- Może zatem cykl Hamiltona powinien zacząć się od (C,K)

# Problem komiwojażera bez algorytmu siłowego III

	C	W	K	R	S	L
Częstochowa(C)	0	68	18	41	43	37
Wieluń (W)	68	0	50	72	110	72
Kłobuck(K)	18	50	0	52	65	40
Radomsko (R)	41	72	52	0	83	80
Siewierz (S)	43	110	65	83	0	60
Lubliniec (L)	37	72	40	80	60	0

- Najbliższym kolejnym nieodwiedzonym celem z Kłobucka jest Lubliniec: (C,K,L)

# Problem komiwojażera bez algorytmu siłowego IV

	C	W	K	R	S	L
Częstochowa(C)	0	68	18	41	43	37
Wieluń (W)	68	0	50	72	110	72
Kłobuck(K)	18	50	0	52	65	40
Radomsko (R)	41	72	52	0	83	80
Siewierz (S)	43	110	65	83	0	60
Lubliniec (L)	37	72	40	80	60	0

- Najbliższym kolejnym nieodwiedzonym celem z Lublinca jest Siewierz: (C,K,L,S)

# Problem komiwojażera bez algorytmu siłowego V

	C	W	K	R	S	L
Częstochowa(C)	0	68	18	41	43	37
Wieluń (W)	68	0	50	72	110	72
Kłobuck(K)	18	50	0	52	65	40
Radomsko (R)	41	72	52	0	83	80
Siewierz (S)	43	110	65	83	0	60
Lubliniec (L)	37	72	40	80	60	0

- Najbliższym kolejnym nieodwiedzonym celem z Siewierza jest Radomsko: (C,K,L,S,R)

# Problem komiwojażera bez algorytmu siłowego VI

	C	W	K	R	S	L
Częstochowa(C)	0	68	18	41	43	37
Wieluń (W)	68	0	50	72	110	72
Kłobuck(K)	18	50	0	52	65	40
Radomsko (R)	41	72	52	0	83	80
Siewierz (S)	43	110	65	83	0	60
Lubliniec (L)	37	72	40	80	60	0

- Najbliższym kolejnym nieodwiedzonym celem z Radomska jest Wieluń: (C,K,L,S,R,W)
- A potem już tylko powrót do Częstochowy: (C,K,L,S,R,W,C):
  - Odległość wygenerowanego cyklu Hamiltona: 341
  - Odległość optymalna: 306
  - Odległość średnia: 356,4

# Algorytm najbliższego - pseudokod I

**Nearest-neighbor-algorithm ( $G = (V, E)$ ,  $v \in V$ ):**

**Input:** Graf reprezentowany przez macierz sąsiedztwa z wagami o rozmiarze  $n \times n$  oraz indeks wierzchołka początkowego  $v$ .

**Output:** Lista odwiedzonych wierzchołków.

```
1: for  $i = 2$  to  $n$  do
2:   visited[i] = false;
3: end for
4: lista_int path = v;
5: visited[v] = true;
6: current = v;
7: for  $i = 2$  to  $n$  do
8:   Znajdź najmniejszy nieodwiedzony element w wierszu current  
   oraz kolumnie i.
```

# Algorytm najbliższego - pseudokod II

```
9:   current = i;  
10:  visited[i] = true;  
11:  Dodaj i na koniec listy path.  
12: end for  
13: Dodaj v na koniec listy path.  
14: return path.
```

Złożoność czasowa:  $O(n^2)$ .

# Algorytm najbliższego sąsiada vs algorytm siłowy

- Algorytm Brute-Force jest **optymalny**, ale **nieefektywny**.
  - Gwarantuje znalezienie optymalnego rozwiązania, ale może to zająć zbyt długi czas.
- Algorytm najbliższego sąsiada jest **wydajny**, ale **nieoptymalny**.
  - Jest szybki i łatwy w implementacji, ale nie zawsze znajduje cykl Hamilton o najmniejszej wadze.

# Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

b.wozna@ujd.edu.pl

Wykład 13 i 14

# Spis treści

- 1 Grafy skierowane z wagami - przypomnienie
- 2 Algorytm Bellmana-Forda
  - Przykład
- 3 Algorytm Dijkstry
  - Przykład 1
  - Przykład 2
- 4 Algorytm Floyda-Warshalla
  - Twórcy
  - Informacje wstępne
  - Pseudokod
  - Przykład
- 5 Algorytm Johnsona
  - Pseudokod
  - Przykład

# Graf skierowany z wagami

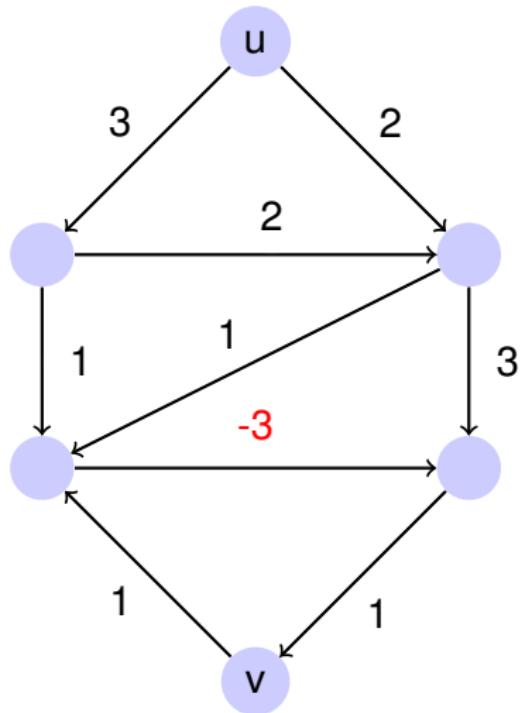
## Definicja

**Grafem skierowanym z wagami** nazywamy strukturę

$G = (V, E, \text{weight} : E \rightarrow Z)$  gdzie

- $V$  to zbiór wierzchołków,
- $E \subseteq \{(u, v) : u, v \in V\}$  to zbiór uporządkowanych par wierzchołków ze zbioru  $V$ , zwanych krawędziami.
- $\text{weight} : E \rightarrow Z$  jest funkcją wagi (odległości).

# Graf skierowany z wagami - przykład



# Drzewo najkrótszych ścieżek

## Sformułowanie problemu

### • Wejście:

- Graf skierowany z wagami  $G = (V, E, \text{weight} : E \rightarrow Z)$
- Wierzchołek  $r \in V$ , zwany **korzeniem**.

### • Wyjście:

- Drzewo  $T$  o korzeniu  $r$  takie, że ścieżka z  $r$  do każdego wierzchołka  $u$  w  $T$  jest najkrótszą ścieżką z  $r$  do  $u$  w grafie  $G$ .

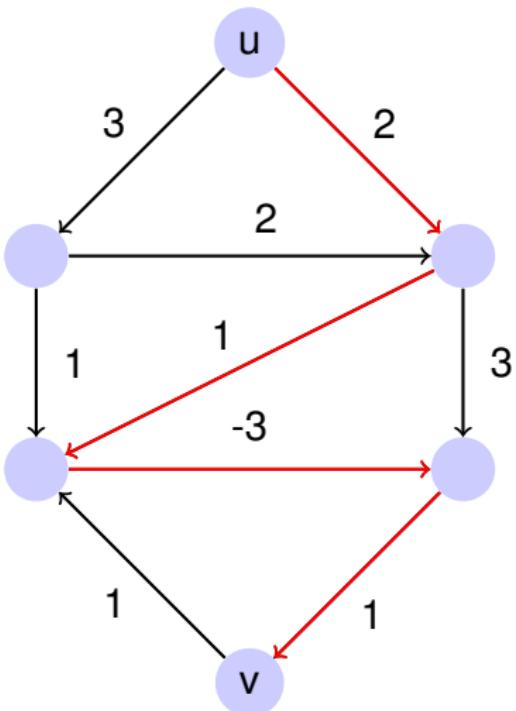
### • Założenie:

Rozważane grafy mają wierzchołki osiągalne z wybranego wierzchołka (korzenia)  $r$ <sup>a</sup>.

---

<sup>a</sup>Dlaczego? Wierzchołki nieosiągalne mogą być usunięte w czasie liniowym

# Graf skierowany z wagami - przykład



## Pytanie:

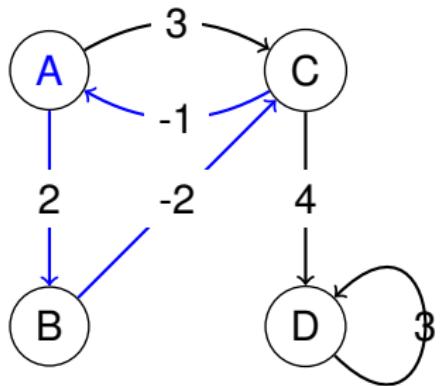
- Czy najkrótsza ścieżka pomiędzy wierzchołkami  $u$  i  $v$  może zawierać cykl?

## Odpowiedź:

- Jeśli w grafie istnieje najkrótsza ścieżka z  $u$  do  $v$ , to w grafie tym również istnieje najkrótsza ścieżka z  $u$  do  $v$ , która nie zawiera cykli.

# Ujemne cykle

- Niech  $G = (V, E, \text{weight} : E \rightarrow Z)$  będzie grafem skierowanym z wagami. **Cykł ujemny** w grafie  $G$ , to cykl  $C$ , którego waga  $\text{weight}(C)$  jest ujemna.



- $C=(A,B,C,A)$ ,  $w(C) = -1$ .

# Richard Ernest Bellman - (26.08.1920-19.03.1984)

- Richard Ernest Bellman znany jako twórca:
  - programowania dynamicznego.
  - algorytmu Bellmana–Fordy.
- W 1979 roku otrzymał IEEE Medal of Honor za "wkład w teorie sterowania i procesów decyzyjnych, szczególnie za opracowanie teorii programowania dynamicznego".



Źródło: <http://logistyka.math.uni.lodz.pl/Bellman.jpeg>

- Więcej na [http://en.wikipedia.org/wiki/Richard\\_E.\\_Bellman](http://en.wikipedia.org/wiki/Richard_E._Bellman)

# Lester Randolph Ford, junior (23.09.1927-26.02.2017)

- Amerykański matematyk specjalizujący się w algorytmach przepływu w sieci.
- Syn matematyka Lester R. Forda, seniora.
- Autor algorytmu Bellmana-Forda, służącego do znajdowania najkrótszej ścieżki w grafach z wagami.



Źródło:<http://wazniak.mimuw.edu.pl/index.php?title=Grafika:Ford-portret.jpg>

# Algorytm Bellmana-Forda

- Algorytm obliczający najkrótsze ścieżki z danego wierzchołka źródłowego do wszystkich pozostałych wierzchołków w skierowanym grafie z wagami.
- Wolniejszy od **algorytmu Dijkstry** dla tego samego problemu, ale bardziej uniwersalny, ponieważ jest w stanie obsługiwać grafy, które zawierają krawędzie o ujemnej wadze, ale nie zawierają ujemnych cykli.
- Algorytm po raz pierwszy został zaproponowany przez **Alfonso Shimbela** w 1955 roku, ale został nazwany imieniem **Richarda Bellmana i Lester Forda, Jr.**, gdyż to oni opublikowali ten algorytm odpowiednio w 1958 i 1956 roku.
- **Edward F. Moore** opublikował również ten sam algorytm w 1957 r., Dlatego też algorytm ten nazywany jest również **algorytmem Bellmana-Forda-Moore'a**<sup>1</sup>.

<sup>1</sup>Bang-Jensen, Jorgen; Gutin, Gregory (2000). Digraphs: Theory, Algorithms and Applications

# Algorytm Bellmana-Forda - pseudokod

**Input:** Graf

$G(V, E, \text{weight})$ ,  
wierzchołek początkowy  
*src.*

```
1: for all v in V do
2:   v.distance =  $\infty$ ;
3:   v.prev = Null;
4: end for
5: src.distance = 0
6: for i=1 to |V| - 1 do
7:   for all (u, v) in E do
8:     relax(u, v);
9:   end for
10: end for
```

# Algorytm Bellmana-Forda - pseudokod

**Input:** Graf

$G(V, E, \text{weight})$ ,

wierzchołek początkowy  
*src*.

```

1: for all v in V do
2:   v.distance =  $\infty$ ;
3:   v.prev = Null;
4: end for
5: src.distance = 0
6: for i=1 to |V| - 1 do
7:   for all (u, v) in E do
8:     relax(u, v);
9:   end for
10: end for
```

- L.1-4: Pętla **for** ustawia dla każdego wierzchołka grafu parametr **distance** (odległość) na nieskończoność oraz wskaźnik **prev** na zero. **prev** jest wskaźnikiem do poprzednika danego wierzchołka – pozwala na odtworzenie najkrótszej ścieżki.

# Algorytm Bellmana-Forda - pseudokod

**Input:** Graf

$G(V, E, \text{weight})$ ,

wierzchołek początkowy  
*src*.

```

1: for all v in V do
2:   v.distance =  $\infty$ ;
3:   v.prev = Null;
4: end for
5: src.distance = 0
6: for i=1 to |V| - 1 do
7:   for all (u, v) in E do
8:     relax(u, v);
9:   end for
10: end for
```

- L.1-4: Pętla **for** ustawia dla każdego wierzchołka grafu parametr **distance** (odległość) na nieskończoność oraz wskaźnik **prev** na zero. **prev** jest wskaźnikiem do poprzednika danego wierzchołka – pozwala na odtworzenie najkrótszej ścieżki.
- L.5: Odległość dla wierzchołka początkowego ustawiana jest na zero.

# Algorytm Bellmana-Forda - pseudokod

**Input:** Graf

$G(V, E, \text{weight})$ ,

wierzchołek początkowy  
*src*.

```

1: for all v in V do
2:   v.distance =  $\infty$ ;
3:   v.prev = Null;
4: end for
5: src.distance = 0
6: for i=1 to |V| - 1 do
7:   for all (u, v) in E do
8:     relax(u, v);
9:   end for
10: end for
```

- L.1-4: Pętla **for** ustawia dla każdego wierzchołka grafu parametr **distance** (odległość) na nieskończoność oraz wskaźnik **prev** na zero. **prev** jest wskaźnikiem do poprzednika danego wierzchołka – pozwala na odtworzenie najkrótszej ścieżki.
- L.5: Odległość dla wierzchołka początkowego ustawiana jest na zero.
- L.7-9: Pętla **for** przechodzi przez każdą krawędź grafu  $(u, v)$  i dokonuje jej tzw. **relaksacji**. Proces ten wykonywany jest  $|V| - 1$  razy

# Relaksacja krawędzi

- **Relaksacja krawędzi** jest najważniejszym krokiem w algorytmie Bellmana-Forda.
- **Relaksacja krawędzi** to sprawdzenie, czy przy przejściu daną krawędzią grafu  $(u, v)$  (tj. z ' $u$ ' do ' $v$ ') , nie otrzymamy krótszej ścieżki niż dotychczasowa ze źródła ' $src$ ' do ' $v$ '. Jeżeli tak, to zmniejszamy oszacowanie wagi  $v.distance$  najkrótszej ścieżki.

`relax( $u, v$ ):`

- 1: **if**  $v.distance > u.distance + \text{weight}(u, v)$  **then**
- 2:    $v.distance = u.distance + \text{weight}(u, v)$
- 3:    $v.prev = u$
- 4: **end if**

# Wykrywanie ujemnych cykli - pseudokod

**Input:** Graf  $G(V, E, \text{weight})$ , wierzchołek początkowy  $\text{src}$ .

```

1: for all  $v$  in  $V$  do
2:    $v.\text{distance} = \infty;$ 
3:    $v.\text{prev} = \text{Null};$ 
4: end for
5:  $\text{src}.\text{distance} = 0$ 
6: for  $i=1$  to  $|V| - 1$  do
7:   for all  $(u, v)$  in  $E$  do
8:     relax( $u, v$ );
9:   end for
10: end for
11: for all  $(u, v)$  in  $E$  do
12:   if  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$ 
    then
13:     "Istnieje ujemny cykl!"
14:   end if
15: end for
```

Krótki i prosty dodatek do algorytmu Bellmana-Forda pozwala mu wykrywać **ujemne cykle**.

# Złożoność algorytmu Bellmana-Forda

- Algorytm Bellman-Forda wykonuje  $|E|$  relaksacji dla każdej iteracji.
- Iteracji jest  $|V| - 1$ .
- Dlatego w najgorszym przypadku, algorytm Bellmana-Forda działa w czasie  $O(|V| \cdot |E|)$ .

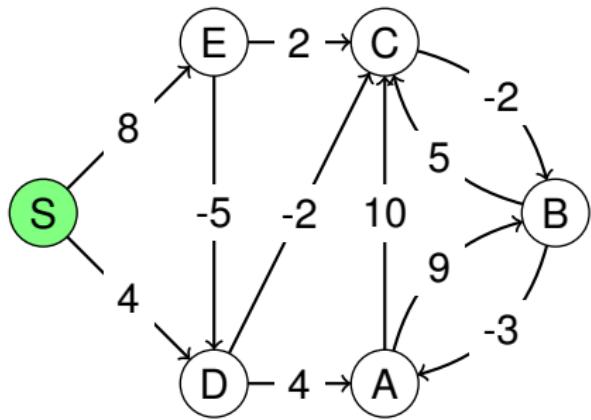
# Ustawienie początkowe parametru distance

**Input:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

```

1: for all  $v$  in  $V$  do
2:    $v.\text{distance} = \infty$ ;
3:    $v.\text{prev} = \text{Null}$ ;
4: end for
5:  $\text{src}.\text{distance} = 0$ 
6: for  $i=1$  to  $|V| - 1$  do
7:   for all  $(u, v)$  in  $E$  do
8:     relax( $u, v$ );
9:   end for
10: end for
11: for all  $(u, v)$  in  $E$  do
12:   if  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$  then
13:     "Istnieje ujemny cykl!"
14:   end if
15: end for

```



	s	a	b	c	d	e
I.1-4:	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
I.5:	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$

# Proces relaksacji, iteracja 1, slajd I

**Require:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

- 1: ...
- 2: **for**  $i=1$  to  $|V| - 1$  **do**
- 3:   **for all**  $(u, v)$  in  $E$  **do**
- 4:      $\text{relax}(u, v);$
- 5:   **end for**
- 6: **end for**
- 7: ...

W pętli l.3-5 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se$ .

- Krawędź  $ab$ :

$$b.\text{distance} > a.\text{distance} + \text{weight}(a, b)$$

$$\infty > \infty + 9 - \text{false}$$

- Krawędź  $ac$ :

$$c.\text{distance} > a.\text{distance} + \text{weight}(a, c)$$

$$\infty > \infty + 10 - \text{false}$$

# Proces relaksacji, iteracja 1, slajd II

- Krawędź  $ba$ :

$a.distance > b.distance + weight(b, a)$

$\infty > \infty + 3$  – false

- Krawędź  $bc$ :

$c.distance > b.distance + weight(b, c)$

$\infty > \infty + 5$  – false

- Krawędź  $cb$ :

$b.distance > c.distance + weight(c, b)$

$\infty > \infty + 2$  – false

- Krawędź  $da$ :

$a.distance > d.distance + weight(d, a)$

$\infty > \infty + 4$  – false

## Proces relaksacji, iteracja 1, slajd III

- Krawędź  $dc$ :

$c.distance > d.distance + \text{weight}(d, c)$

$\infty > \infty - 2$  – false

- Krawędź  $ec$ :

$c.distance > e.distance + \text{weight}(e, c)$

$\infty > \infty + 2$  – false

- Krawędź  $ed$ :

$d.distance > e.distance + \text{weight}(e, d)$

$\infty > \infty - 5$  – false

- Krawędź  $sd$ :

$d.distance > s.distance + \text{weight}(s, d)$

$\infty > 0 + 4$  – true;  $d.distance = 4$

- Krawędź  $se$ :

$e.distance > s.distance + \text{weight}(s, e)$

$\infty > 0 + 8$  – true;  $e.distance = 8$

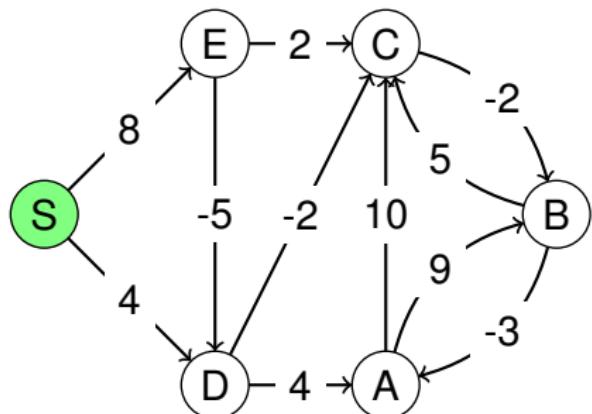
# Proces relaksacji, iteracja 1, slajd IV

**Input:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

```

1: ...
2: for  $i=1$  to  $|V| - 1$  do
3:   for all  $(u, v)$  in  $E$  do
4:     relax( $u, v$ );
5:   end for
6: end for
7: for all  $(u, v)$  in  $E$  do
8:   if  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$  then
9:     "Istnieje ujemny cykl!"
10:   end if
11: end for
```

W pętli l.3-5 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se$ .



	s	a	b	c	d	e
	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
i=1	0	$\infty$	$\infty$	$\infty$	4	8

# Proces relaksacji, iteracja 2, slajd I

**Require:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

- 1: ...
- 2: **for**  $i=1$  to  $|V| - 1$  **do**
- 3:   **for all**  $(u, v)$  in  $E$  **do**
- 4:      $\text{relax}(u, v);$
- 5:   **end for**
- 6: **end for**
- 7: ...

W pętli l.3-5 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se$ .

- Krawędź  $ab$ :

$$\begin{aligned} b.\text{distance} &> a.\text{distance} + \text{weight}(a, b) \\ \infty &> \infty + 9 - \text{false} \end{aligned}$$

- Krawędź  $ac$ :

$$\begin{aligned} c.\text{distance} &> a.\text{distance} + \text{weight}(a, c) \\ \infty &> \infty + 10 - \text{false} \end{aligned}$$

# Proces relaksacji, iteracja 2, slajd II

- Krawędź  $ba$ :

$$a.distance > b.distance + \text{weight}(b, a)$$

$$\infty > \infty - 3 - \text{false}$$

- Krawędź  $bc$ :

$$c.distance > b.distance + \text{weight}(b, c)$$

$$\infty > \infty + 5 - \text{false}$$

- Krawędź  $cb$ :

$$b.distance > c.distance + \text{weight}(c, b)$$

$$\infty > \infty - 2 - \text{false}$$

- Krawędź  $da$ :

$$a.distance > d.distance + \text{weight}(d, a)$$

$$\infty > 4 + 4 - \text{true}; \textcolor{red}{a.distance = 8}$$

## Proces relaksacji, iteracja 2, slajd III

- Krawędź  $dc$ :

$$c.distance > d.distance + \text{weight}(d, c)$$

$$\infty > 4 - 2 - \text{true}; \quad c.distance = 2$$

- Krawędź  $ec$ :

$$c.distance > e.distance + \text{weight}(e, c)$$

$$2 > 8 + 2 - \text{false}$$

- Krawędź  $ed$ :

$$d.distance > e.distance + \text{weight}(e, d)$$

$$4 > 8 - 5 - \text{true}; \quad d.distance = 3$$

- Krawędź  $sd$ :

$$d.distance > s.distance + \text{weight}(s, d)$$

$$3 > 0 + 4 - \text{false};$$

- Krawędź  $se$ :

$$e.distance > s.distance + \text{weight}(s, e)$$

$$8 > 0 + 8 - \text{false};$$

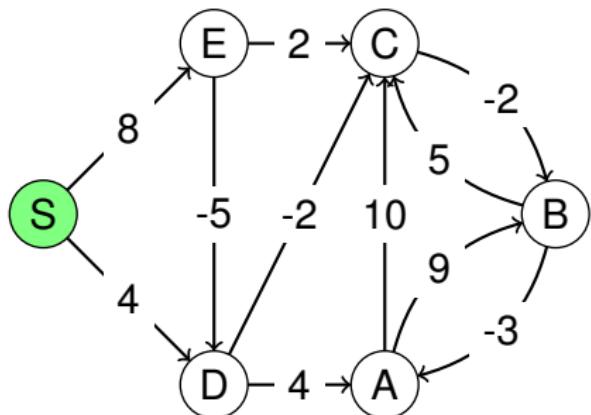
# Proces relaksacji, iteracja 2, slajd IV

**Input:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

```

1: ...
2: for  $i=1$  to  $|V| - 1$  do
3:   for all  $(u, v)$  in  $E$  do
4:     relax( $u, v$ );
5:   end for
6: end for
7: for all  $(u, v)$  in  $E$  do
8:   if  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$  then
9:     "Istnieje ujemny cykl!"
10:   end if
11: end for
```

W pętli l.2-6 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se$ .



	s	a	b	c	d	e
	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
i=1	0	$\infty$	$\infty$	$\infty$	4	8
i=2	0	8	$\infty$	2	3	8

# Proces relaksacji, iteracja 3, slajd I

**Require:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

- 1: ...
- 2: **for**  $i=1$  to  $|V| - 1$  **do**
- 3:   **for all**  $(u, v)$  in  $E$  **do**
- 4:      $\text{relax}(u, v);$
- 5:   **end for**
- 6: **end for**
- 7: ...

W pętli l.3-5 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se.$

- Krawędź  $ab$ :

$$\begin{aligned} b.\text{distance} &> a.\text{distance} + \text{weight}(a, b) \\ \infty &> 8 + 9 - \text{true}; b.\text{distance} = 17 \end{aligned}$$

- Krawędź  $ac$ :

$$\begin{aligned} c.\text{distance} &> a.\text{distance} + \text{weight}(a, c) \\ 2 &> 8 + 10 - \text{false} \end{aligned}$$

## Proces relaksacji, iteracja 3, slajd II

- Krawędź  $ba$ :

$$a.distance > b.distance + \text{weight}(b, a)$$

$$8 > 17 - 3 - \text{false}$$

- Krawędź  $bc$ :

$$c.distance > b.distance + \text{weight}(b, c)$$

$$2 > 17 + 5 - \text{false}$$

- Krawędź  $cb$ :

$$b.distance > c.distance + \text{weight}(c, b)$$

$$17 > 2 - 2 - \text{true}; \textcolor{red}{b.distance = 0}$$

- Krawędź  $da$ :

$$a.distance > d.distance + \text{weight}(d, a)$$

$$8 > 3 + 4 - \text{true}; \textcolor{red}{a.distance = 7}$$

## Proces relaksacji, iteracja 3, slajd III

- Krawędź  $dc$ :

$$c.distance > d.distance + \text{weight}(d, c)$$
$$2 > 3 - 2 - \text{true}; \quad c.distance = 1$$

- Krawędź  $ec$ :

$$c.distance > e.distance + \text{weight}(e, c)$$
$$1 > 8 + 2 - \text{false}$$

- Krawędź  $ed$ :

$$d.distance > e.distance + \text{weight}(e, d)$$
$$3 > 8 - 5 - \text{false};$$

- Krawędź  $sd$ :

$$d.distance > s.distance + \text{weight}(s, d)$$
$$3 > 0 + 4 - \text{false};$$

- Krawędź  $se$ :

$$e.distance > s.distance + \text{weight}(s, e)$$
$$8 > 0 + 8 - \text{false};$$

# Proces relaksacji, iteracja 3, slajd IV

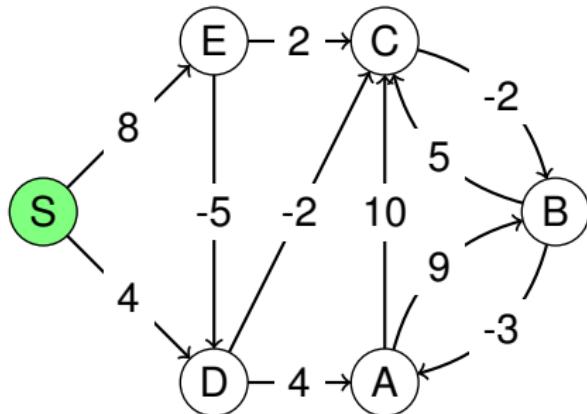
**Input:**  $G(V, E, \text{weight})$ ,  $\text{src.}$

```

1: ...
2: for  $i=1$  to  $|V| - 1$  do
3:   for all  $(u, v)$  in  $E$  do
4:     relax( $u, v$ );
5:   end for
6: end for
7: for all  $(u, v)$  in  $E$  do
8:   if  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$  then
9:     "Istnieje ujemny cykl!"
10:   end if
11: end for

```

W pętli l.2-6 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se.$



	s	a	b	c	d	e
	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
i=1	0	$\infty$	$\infty$	$\infty$	4	8
i=2	0	8	$\infty$	2	3	8
i=3	0	7	0	1	3	8

# Proces relaksacji, iteracja 4, slajd I

**Require:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

- 1: ...
- 2: **for**  $i=1$  to  $|V| - 1$  **do**
- 3:   **for all**  $(u, v)$  in  $E$  **do**
- 4:      $\text{relax}(u, v);$
- 5:   **end for**
- 6: **end for**
- 7: ...

W pętli l.3-5 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se.$

- Krawędź  $ab$ :

$$b.\text{distance} > a.\text{distance} + \text{weight}(a, b)$$

$$0 > 7 + 9 - \text{false};$$

- Krawędź  $ac$ :

$$c.\text{distance} > a.\text{distance} + \text{weight}(a, c)$$

$$1 > 7 + 10 - \text{false}$$

## Proces relaksacji, iteracja 4, slajd II

- Krawędź  $ba$ :

$$a.distance > b.distance + \text{weight}(b, a)$$

$$7 > 0 - 3 - \text{true}; \quad a.distance = -3$$

- Krawędź  $bc$ :

$$c.distance > b.distance + \text{weight}(b, c)$$

$$1 > 0 + 5 - \text{false}$$

- Krawędź  $cb$ :

$$b.distance > c.distance + \text{weight}(c, b)$$

$$0 > 1 - 2 - \text{true}; \quad b.distance = -1$$

- Krawędź  $da$ :

$$a.distance > d.distance + \text{weight}(d, a)$$

$$-3 > 3 + 4 - \text{false};$$

## Proces relaksacji, iteracja 4, slajd III

- Krawędź  $dc$ :

$$c.distance > d.distance + \text{weight}(d, c)$$

$1 > 3 - 2 - \text{false};$

- Krawędź  $ec$ :

$$c.distance > e.distance + \text{weight}(e, c)$$

$1 > 8 + 2 - \text{false}$

- Krawędź  $ed$ :

$$d.distance > e.distance + \text{weight}(e, d)$$

$3 > 8 - 5 - \text{false};$

- Krawędź  $sd$ :

$$d.distance > s.distance + \text{weight}(s, d)$$

$3 > 0 + 4 - \text{false};$

- Krawędź  $se$ :

$$e.distance > s.distance + \text{weight}(s, e)$$

$8 > 0 + 8 - \text{false};$

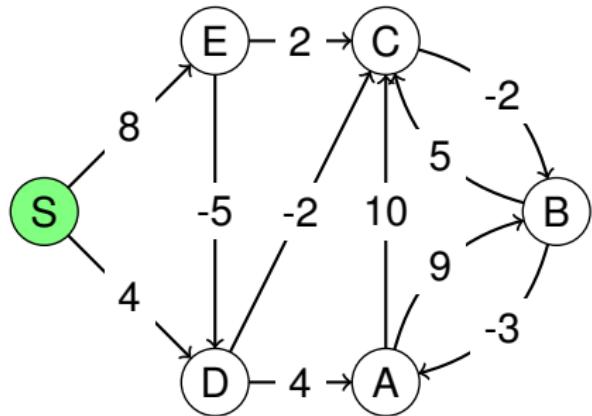
# Proces relaksacji, iteracja 4, slajd IV

**Input:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

```

1: ...
2: for  $i=1$  to  $|V| - 1$  do
3:   for all  $(u, v)$  in  $E$  do
4:     relax( $u, v$ );
5:   end for
6: end for
7: for all  $(u, v)$  in  $E$  do
8:   if  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$  then
9:     "Istnieje ujemny cykl!"
10:   end if
11: end for
```

W pętli l.2-6 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se$ .



	s	a	b	c	d	e
	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
i=1	0	$\infty$	$\infty$	$\infty$	4	8
i=2	0	8	$\infty$	2	3	8
i=3	0	7	0	1	3	8
i=4	0	-3	-1	1	3	8

# Proces relaksacji, iteracja 5, slajd I

**Require:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

- 1: ...
- 2: **for**  $i=1$  **to**  $|V| - 1$  **do**
- 3:   **for all**  $(u, v)$  in  $E$  **do**
- 4:      $\text{relax}(u, v);$
- 5:   **end for**
- 6: **end for**
- 7: ...

W pętli l.3-5 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se$ .

- Krawędź  $ab$ :

$$b.\text{distance} > a.\text{distance} + \text{weight}(a, b)$$
$$-1 > -3 + 9 - \text{false};$$

- Krawędź  $ac$ :

$$c.\text{distance} > a.\text{distance} + \text{weight}(a, c)$$
$$1 > -3 + 10 - \text{false};$$

## Proces relaksacji, iteracja 5, slajd II

- Krawędź  $ba$ :

$$a.distance > b.distance + \text{weight}(b, a)$$

$$-3 > -1 - 3 - \text{true}; \quad a.distance = -4$$

- Krawędź  $bc$ :

$$c.distance > b.distance + \text{weight}(b, c)$$

$$1 > -1 + 5 - \text{false};$$

- Krawędź  $cb$ :

$$b.distance > c.distance + \text{weight}(c, b)$$

$$-1 > 1 - 2 - \text{false};$$

- Krawędź  $da$ :

$$a.distance > d.distance + \text{weight}(d, a)$$

$$-3 > 3 + 4 - \text{false};$$

## Proces relaksacji, iteracja 5, slajd III

- Krawędź  $dc$ :

$$c.distance > d.distance + \text{weight}(d, c)$$

$1 > 3 - 2 - \text{false};$

- Krawędź  $ec$ :

$$c.distance > e.distance + \text{weight}(e, c)$$

$1 > 8 + 2 - \text{false}$

- Krawędź  $ed$ :

$$d.distance > e.distance + \text{weight}(e, d)$$

$3 > 8 - 5 - \text{false};$

- Krawędź  $sd$ :

$$d.distance > s.distance + \text{weight}(s, d)$$

$3 > 0 + 4 - \text{false};$

- Krawędź  $se$ :

$$e.distance > s.distance + \text{weight}(s, e)$$

$8 > 0 + 8 - \text{false};$

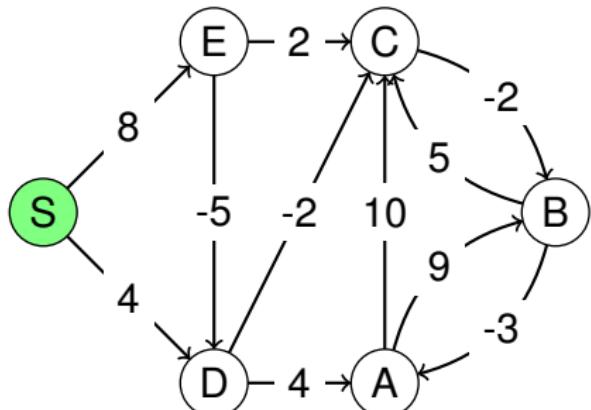
# Proces relaksacji, iteracja 5, slajd IV

**Input:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

```

1: ...
2: for  $i=1$  to  $|V| - 1$  do
3:   for all  $(u, v)$  in  $E$  do
4:     relax( $u, v$ );
5:   end for
6: end for
7: for all  $(u, v)$  in  $E$  do
8:   if  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$  then
9:     "Istnieje ujemny cykl!"
10:   end if
11: end for
```

W pętli l.2-6 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se$ .



	s	a	b	c	d	e
	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
i=1	0	$\infty$	$\infty$	$\infty$	4	8
i=2	0	8	$\infty$	2	3	8
i=3	0	7	0	1	3	8
i=4	0	-3	-1	1	3	8
i=5	0	-4	-1	1	3	8

# Badanie istnienia ujemnych cykli - slajd I

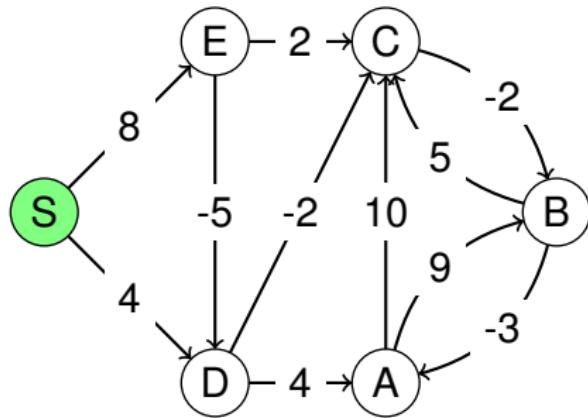
**Input:**  $G(V, E, \text{weight})$ ,  $\text{src.}$

```

1: ...
2: for  $i=1$  to  $|V| - 1$  do
3:   for all  $(u, v)$  in  $E$  do
4:     relax( $u, v$ );
5:   end for
6: end for
7: for all  $(u, v)$  in  $E$  do
8:   if  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$  then
9:     "Istnieje ujemny cykl!"
10:   end if
11: end for

```

W pętli l.7-11 krawędzie rozważane są w porządku leksykograficznym:  
 $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se.$



s	a	b	c	d	e
0	-4	-1	1	3	8

# Badanie istnienia ujemnych cykli - slajd II

**Require:**  $G(V, E, \text{weight})$ ,  $\text{src}$ .

- 1: ...
- 2: **for all**  $(u, v)$  in  $E$  **do**
- 3:   **if**  $v.\text{distance} > u.\text{distance} + \text{weight}(u, v)$  **then**
- 4:     "Istnieje ujemny cykl!"
- 5:   **end if**
- 6: **end for**

- Krawędź  $ab$ :

$$b.\text{distance} > a.\text{distance} + \text{weight}(a, b)$$

$$-1 > -3 + 9 - \text{false};$$

- Krawędź  $ac$ :

$$c.\text{distance} > a.\text{distance} + \text{weight}(a, c)$$

$$1 > -4 + 10 - \text{false};$$

W pętli l.3-6 krawędzie rozważane są w porządku leksykograficznym:  $ab, ac, ba, bc, cb, da, dc, ec, ed, sd, se$ .

# Badanie istnienia ujemnych cykli - slajd III

- Krawędź  $ba$ :

$$a.distance > b.distance + \text{weight}(b, a)$$

$-4 > -1 - 3$  – false;

- Krawędź  $bc$ :

$$c.distance > b.distance + \text{weight}(b, c)$$

$1 > -1 + 5$  – false;

- Krawędź  $cb$ :

$$b.distance > c.distance + \text{weight}(c, b)$$

$-1 > 1$  – 2 – false;

- Krawędź  $da$ :

$$a.distance > d.distance + \text{weight}(d, a)$$

$-4 > 3 + 4$  – false;

# Badanie istnienia ujemnych cykli - slajd IV

- Krawędź  $dc$ :

$$c.distance > d.distance + \text{weight}(d, c)$$

$1 > 3 - 2 - \text{false};$

- Krawędź  $ec$ :

$$c.distance > e.distance + \text{weight}(e, c)$$

$1 > 8 + 2 - \text{false}$

- Krawędź  $ed$ :

$$d.distance > e.distance + \text{weight}(e, d)$$

$3 > 8 - 5 - \text{false};$

- Krawędź  $sd$ :

$$d.distance > s.distance + \text{weight}(s, d)$$

$3 > 0 + 4 - \text{false};$

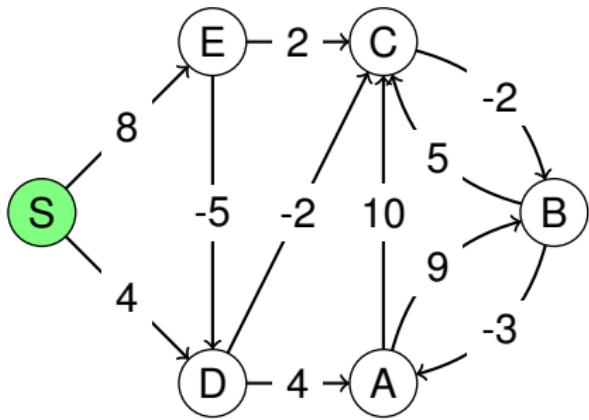
# Badanie istnienia ujemnych cykli - slajd V

- Krawędź se:

$$e.distance > s.distance + \text{weight}(s, e)$$
$$8 > 0 + 8 - \text{false};$$

**Wniosek:** Brak ujemnych cykli !

# Przykład - podsumowanie



	s	a	b	c	d	e
	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
i=1	0	$\infty$	$\infty$	$\infty$	4	8
i=2	0	8	$\infty$	2	3	8
i=3	0	7	0	1	3	8
i=4	0	-3	-1	1	3	8
i=5	0	-4	-1	1	3	8

# Algorytm Dijkstry I

- **Algorytm Dijkstry** został opublikowany w 1959 roku i nazwany na cześć swojego twórcy holenderskiego informatyka **Edsgera Dijkstry**.
- **Algorytm Dijkstry** można zastosować do grafu z wagami  $G(V, E, \text{weight})$ . Graf ten może być skierowany lub nieskierowany.
- **Algorytm Dijkstry** służy do wyznaczania najmniejszej odległości od ustalonego wierzchołka **scr** do wszystkich pozostałych w grafie.
- W odróżnieniu jednak od **Algorytmu Bellmana-Forda**, graf wejściowy **nie może zawierać krawędzi o ujemnych wagach**.

# Algorytm Dijkstry II

- W algorytmie pamiętany jest **zbiór wierzchołków  $Q$** , dla których nie obliczono jeszcze najkrótszych ścieżek, oraz wektor *distance*, odległości od wierzchołka *src* do pozostałych wierzchołków.
- Początkowo zbiór  $Q$  zawiera wszystkie wierzchołki, a wektor *distance* jest ustawiony na wartość "nieznana".
- Dopóki zbiór  $Q$  nie jest pusty wykonuj:
  - Pobierz ze zbioru  $Q$  wierzchołek  $v$  o najmniejszej wartości  $v.distance$  i usuń go ze zbioru.
  - Dla każdego następnika  $u$  wierzchołka  $v$  sprawdź, czy  $u.distance > v.distance + w((v, u))$ , tzn. czy aktualne oszacowanie odległości do wierzchołka  $u$  jest większe od oszacowania odległości do wierzchołka  $v$  plus waga krawędzi  $(v, u)$ .

# Algorytm Dijkstry - pseudokod I

**Require:** Graf  $G(V, E, \text{weight})$  taki, że  $\text{weight}(e) \geq 0$  dla każdego  $e \in E$  oraz wierzchołek źródłowy  $src \in V$ .

**Ensure:** Parametr  $v.distance$  dla każdego  $v \in V$  taki że  $v.distance$  jest równe minimalnej odległości od  $src$  do  $v$ .

- 1:  $src.distance = 0$  ;
- 2:  $Q = \{src\}$ ;
- 3: **for all**  $v$  in  $V \setminus \{src\}$  **do**
- 4:    $v.distance = \infty$ ;
- 5:    $Q = Q \cup \{v\}$ ; {Dodaj  $v$  do  $Q$ }.
- 6: **end for**
- 7: **while**  $Q \neq \emptyset$  **do**
- 8:    $v =$  wierzchołek z  $Q$  o najmniejszej wartości  $v.distance$ ;
- 9:    $Q = Q \setminus \{v\}$ ; {Usuń  $v$  z  $Q$ }.
- 10:   **for all**  $u$  in  $adj[v]$  **do**

# Algorytm Dijkstry - pseudokod II

```
11:   if  $u.distance > v.distance + weight(v, u)$  then
12:      $u.distance = v.distance + weight(v, u);$ 
13:   end if
14: end for
15: end while
16: return wartości  $distance$  dla każdego wierzchołka  $v \in V$ ;
```

## Złożoność<sup>a</sup>

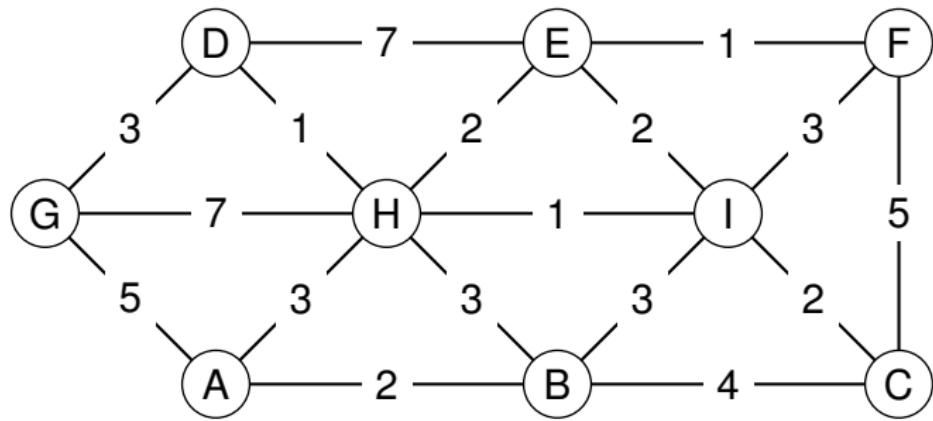
---

<sup>a</sup>M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. J. Assoc. Computer Machinery, 34(3):596–615, 1987

Algorytm Dijkstry można zaimplementować (przy użyciu tak zwanych kopków Fibonacciego) w czasie  $O(|V| \cdot \log(|V|) + |E|)$ .

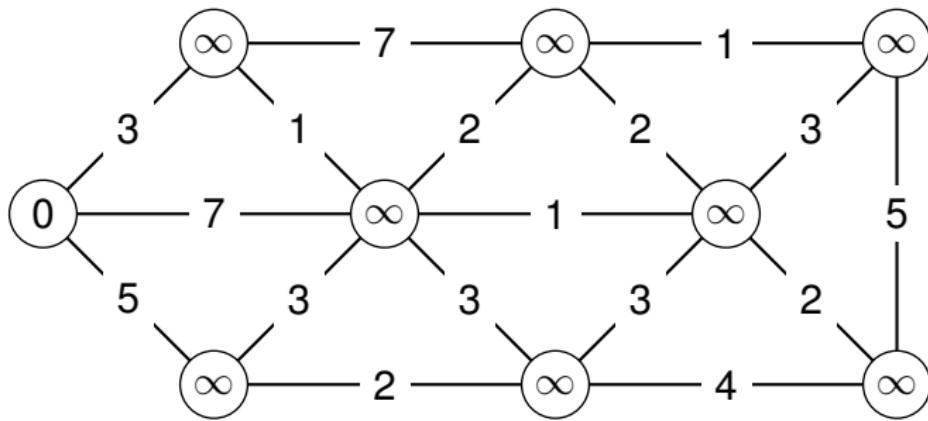
# Slajd I

Przykład zaczerpnięty z <https://brilliant.org/wiki/dijkstras-short-path-finder/>.



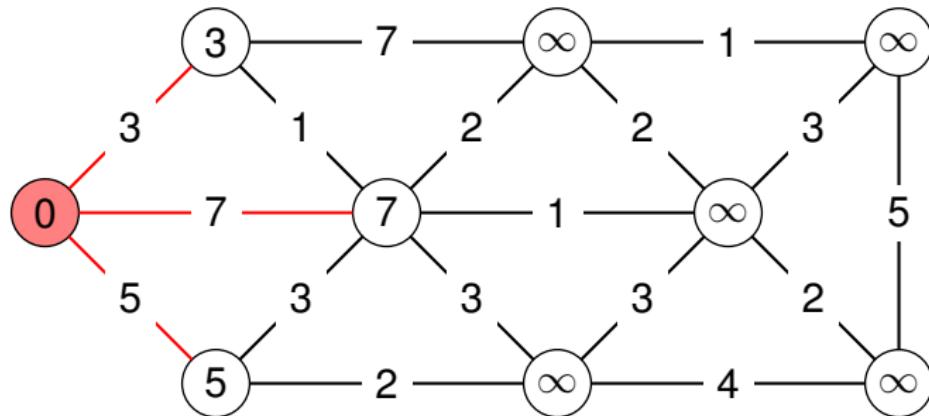
## Slajd II

- Krok 1: Odległości zainicjalizowane zgodnie z algorytmem.
- $scr = G$
- $Q = \{A, B, C, D, E, F, G, H, I\}$ .



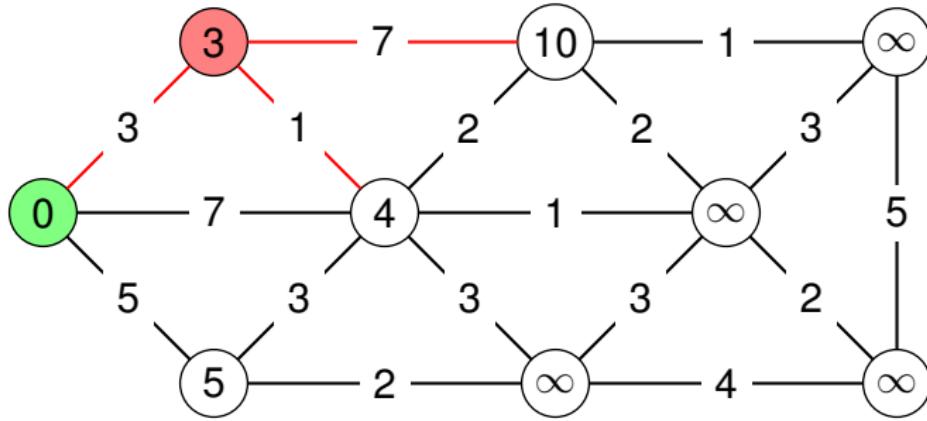
# Slajd III

- Krok 2: Wybierz pierwszy wierzchołek i obliczyć odległości do jego sąsiadów.
- $v = G; Q = Q \setminus \{G\} = \{A, B, C, D, E, F, H, I\}$ .
- relaksacja krawędzi incydentnych z  $G$ .



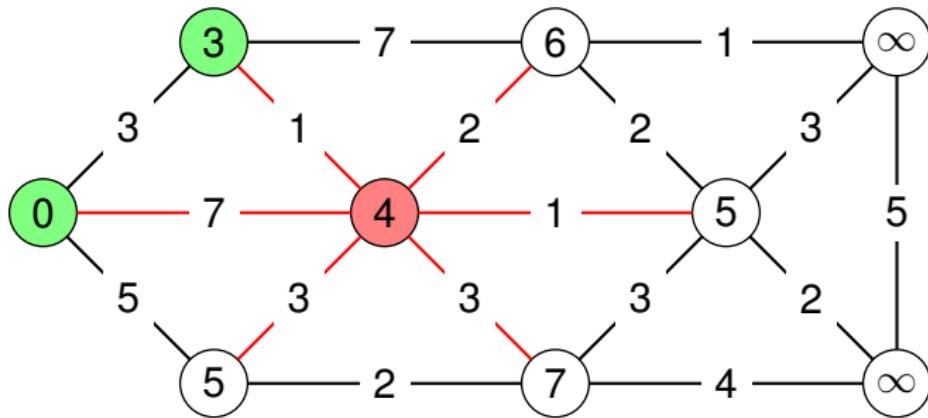
# Slajd IV

- Krok 3: Wybieramy kolejny węzeł o minimalnej odległości; następnie powtarzamy obliczenia odległości dla sąsiadujących wierzchołków.
- $v = D; Q = Q \setminus \{D\} = \{A, B, C, E, F, H, I\}$ .
- relaksacja krawędzi incydentnych z  $D$ .



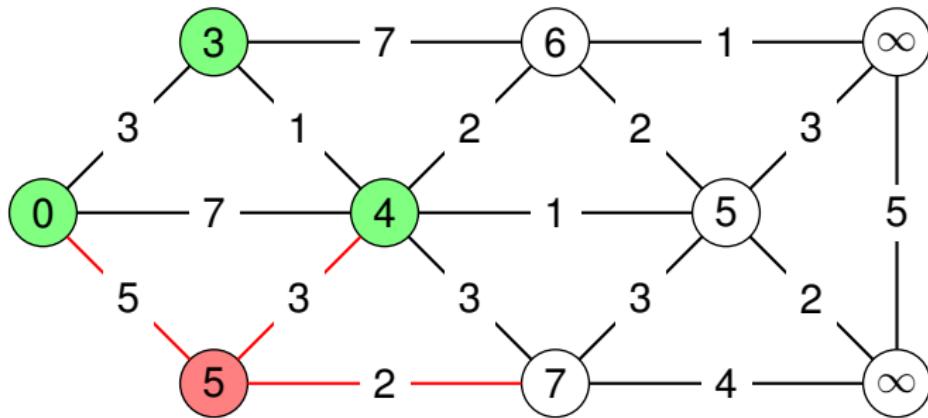
# Slajd V

- Krok 4: Wybieramy kolejny węzeł o minimalnej odległości; następnie powtarzamy obliczenia odległości dla sąsiadujących wierzchołków.
- $v = H; Q = Q \setminus \{H\} = \{A, B, C, E, F, I\}$ .
- relaksacja krawędzi incydentnych z  $H$ .



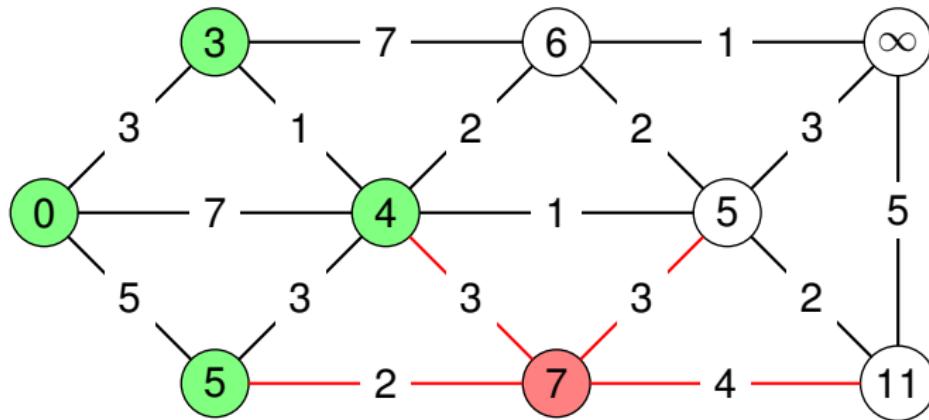
# Slajd VI

- Krok 5: Wybieramy kolejny węzeł o minimalnej odległości; następnie powtarzamy obliczenia odległości dla sąsiadujących wierzchołków.
- $v = A; Q = Q \setminus \{A\} = \{B, C, E, F, I\}$ .
- relaksacja krawędzi incydentnych z  $A$ .



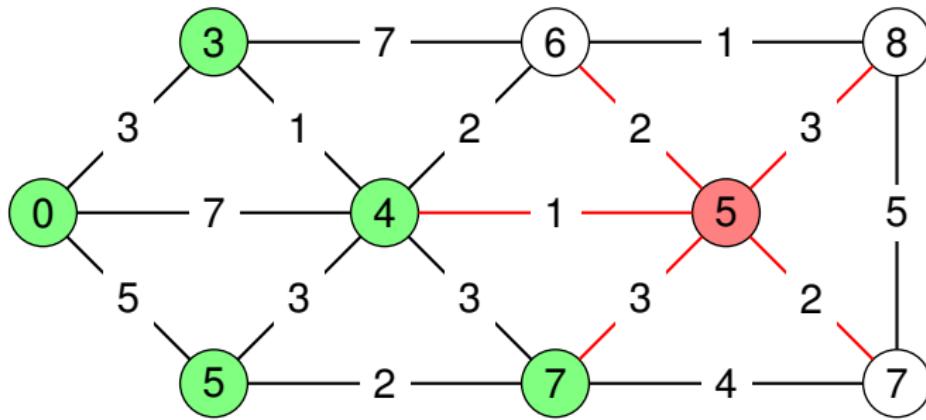
# Slajd VII

- Krok 6: Wybieramy kolejny węzeł o minimalnej odległości; następnie powtarzamy obliczenia odległości dla sąsiadujących wierzchołków.
- $v = B; Q = Q \setminus \{B\} = \{C, E, F, I\}$ .
- relaksacja krawędzi incydentnych z  $B$ .



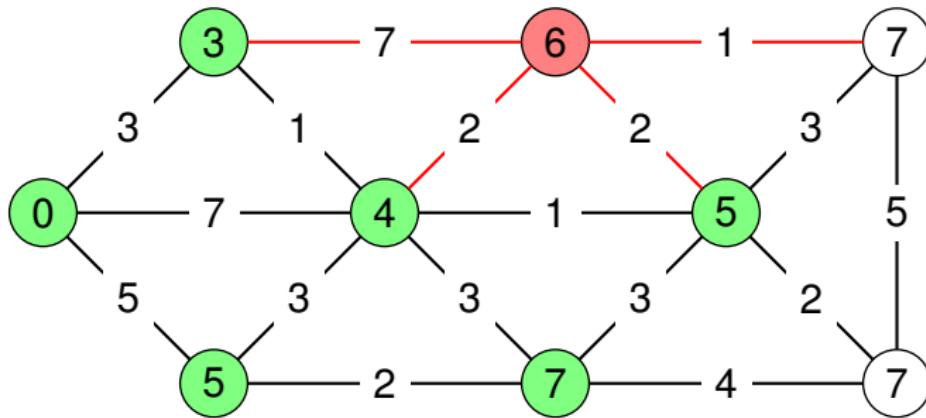
# Slajd VIII

- Krok 7: Wybieramy kolejny węzeł o minimalnej odległości; następnie powtarzamy obliczenia odległości dla sąsiadujących wierzchołków.
- $v = I; Q = Q \setminus \{I\} = \{C, E, F\}$ .
- relaksacja krawędzi incydentnych z  $I$ .



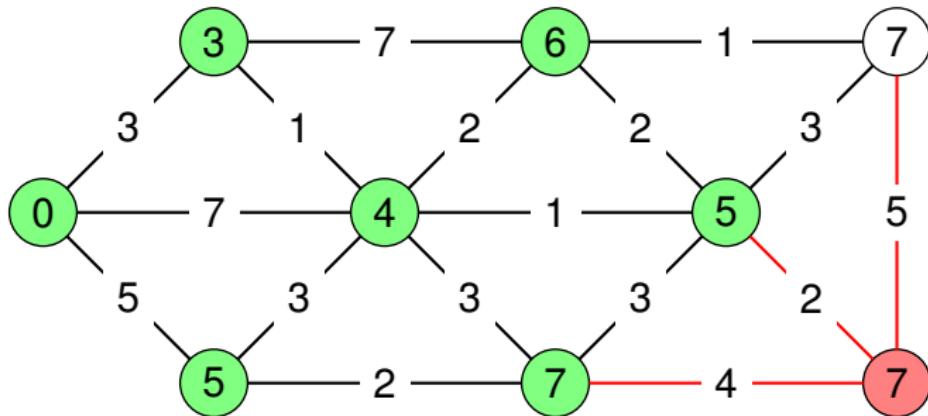
# Slajd IX

- Krok 8: Wybieramy kolejny węzeł o minimalnej odległości; następnie powtarzamy obliczenia odległości dla sąsiadujących wierzchołków.
- $v = E$ ;  $Q = Q \setminus \{E\} = \{C, F\}$ .
- relaksacja krawędzi incydentnych z  $E$ .



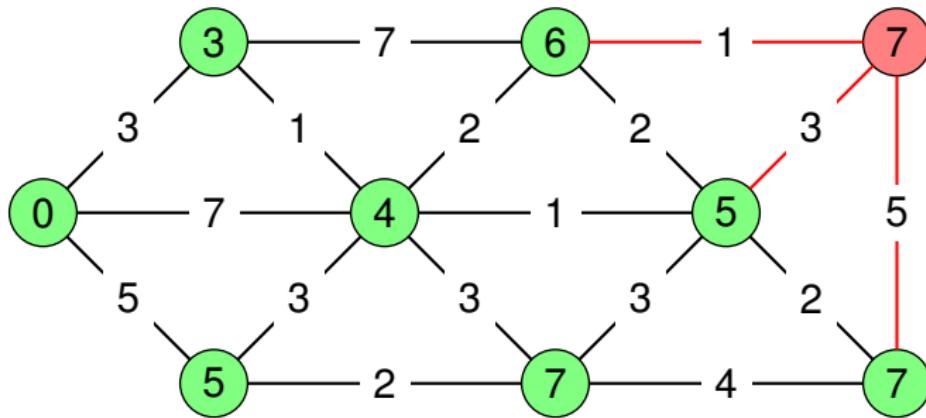
# Slajd X

- Krok 9: Wybieramy kolejny węzeł o minimalnej odległości; następnie powtarzamy obliczenia odległości dla sąsiadujących wierzchołków.
- $v = C; Q = Q \setminus \{C\} = \{F\}$ .
- relaksacja krawędzi incydentnych z  $C$ .



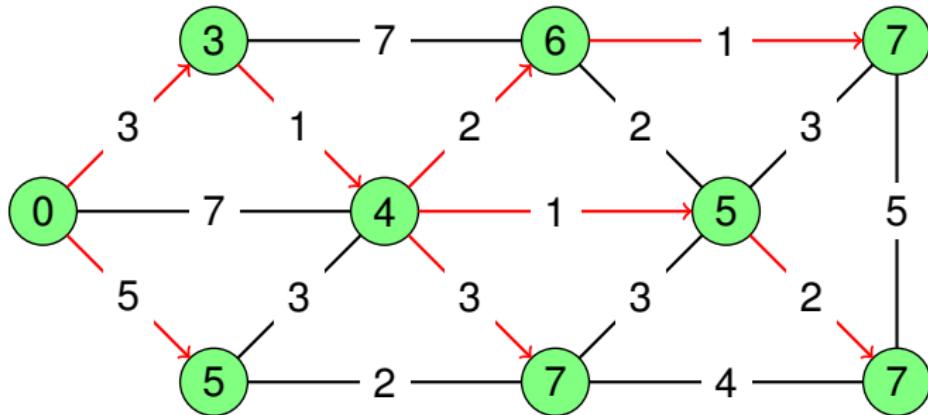
# Slajd XI

- Krok 10: Wybieramy kolejny węzeł o minimalnej odległości; następnie powtarzamy obliczenia odległości dla sąsiadujących wierzchołków.
- $v = F; Q = Q \setminus \{F\} = \emptyset$ .
- relaksacja krawędzi incydentnych z  $F$ .



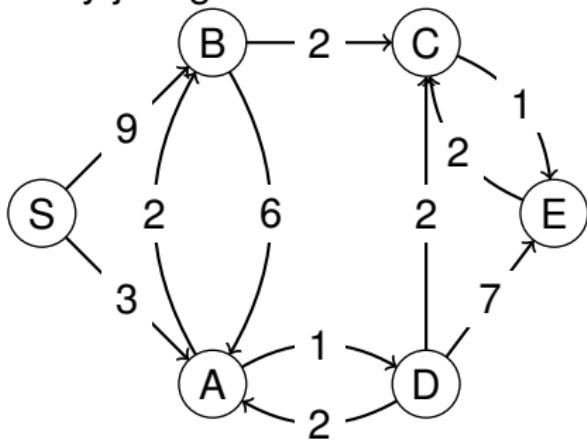
# Slajd XII

Wynikowe drzewo najkrótszych ścieżek:



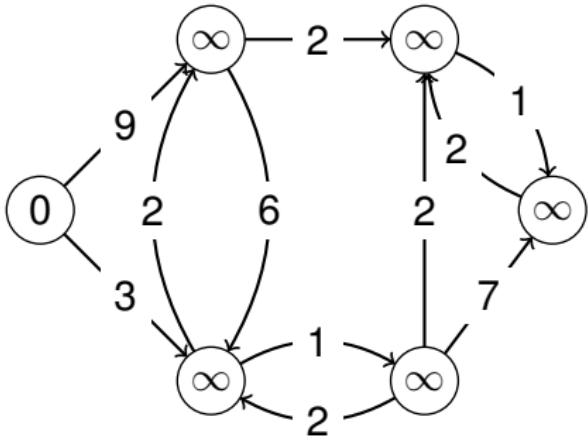
# Slajd I

Dany jest graf:



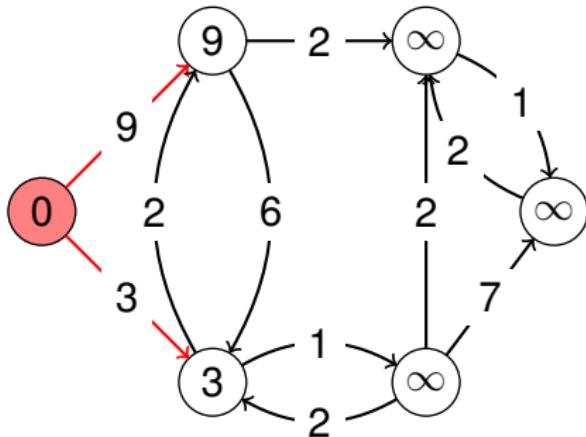
- Krok 1: Odległości zainicjalizowane zgodnie z algorytmem.

- $src = S;$   
 $Q = \{S, A, B, C, D, E\}$

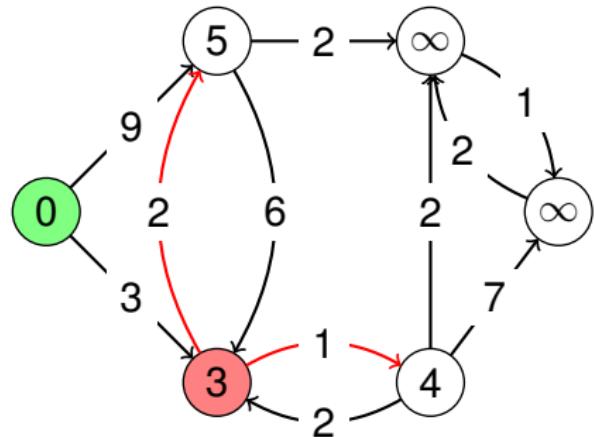


# Slajd II

- Krok 2:

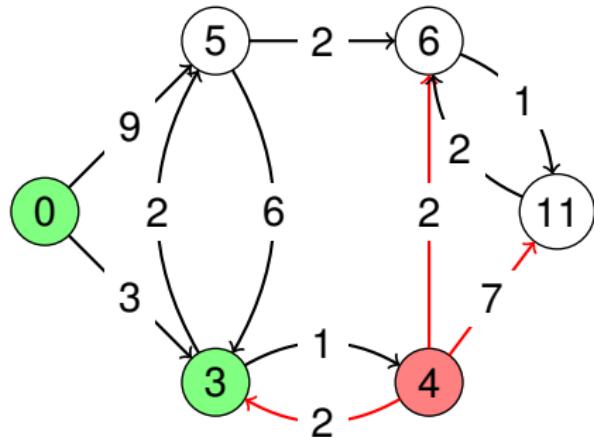


- Krok 3:

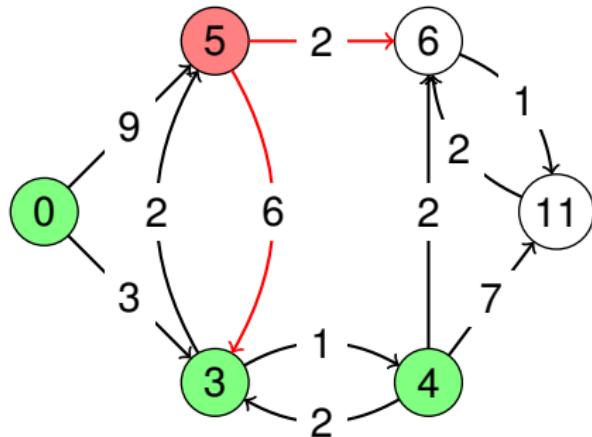


# Slajd III

- Krok 4:

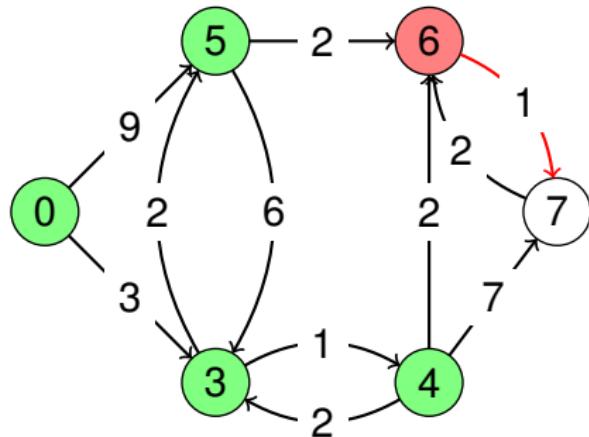


- Krok 5:

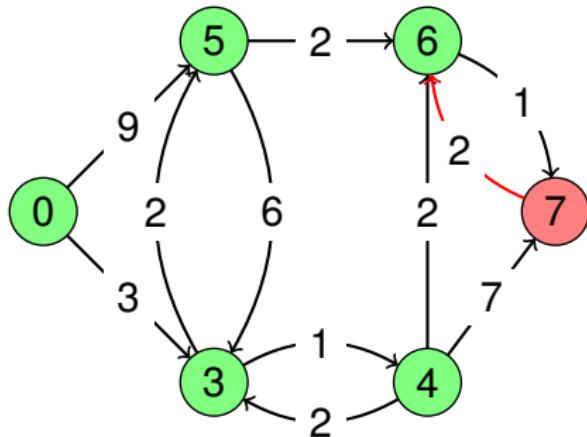


# Slajd IV

- Krok 6:

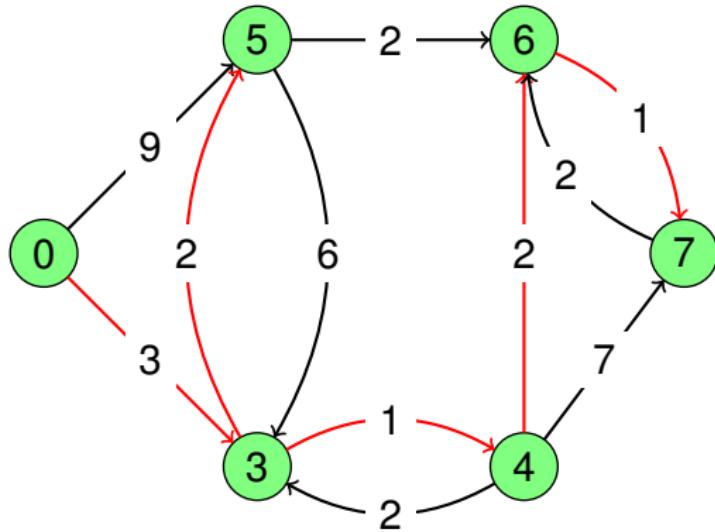


- Krok 7:



# Slajd V

Wynikowe drzewo najkrótszych ścieżek:



# Robert W. Floyd

- Robert W (Bob) Floyd, ur. 9.06.1936, zm. 26/09.2001 – amerykański informatyk.
- Znany z algorytmu Floyda-Warshalla, który efektywnie znajduje wszystkie najkrótsze ścieżki w grafie.
- Laureat nagrody Turinga nadawanej przez organizację ACM – 1978.
- Laureat nagrody Pioniera Informatyki, nadawanej przez organizację IEEE Computer Society – 1991
- Więcej na:  
[https://en.wikipedia.org/wiki/Robert\\_W.\\_Floyd](https://en.wikipedia.org/wiki/Robert_W._Floyd)

# Stephen Warshall

- Stephen Warshall, ur. 15.11.1935, zm. 11.12.2006 – amerykański informatyk.
- Znany z algorytmu Floyda-Warshalla, który efektywnie znajduje wszystkie najkrótsze ścieżki w grafie.
- Podczas swojej kariery naukowej S. Warshall prowadził badania w zakresie systemów operacyjnych, projektowania kompilatorów, projektowania języków oraz analizy operacyjnej.
- Więcej na:

[https://en.wikipedia.org/wiki/Stephen\\_Warshall](https://en.wikipedia.org/wiki/Stephen_Warshall)

# Algorytm Floyda-Warshalla

- **Algorytm Floyda-Warshalla** został zaproponowany niezależnie przez **Roberta W. Floyda i Stephena Warshalla** w pracach:
  - R. W. Floyd. *Algorithm 97: shortest path.* Communications of the ACM, 5:345, 1962.
  - S. Warshall. A theorem on boolean matrices. J. ACM, 9:11–12, 1962
- **Algorytm Floyda-Warshalla**, podobnie jak algorytm Bellmana-Forda lub algorytm Dijkstry, oblicza najkrótszą ścieżkę w grafie.
- Algorytmy Bellmana-Forda oraz Dijkstry obliczają najkrótszą ścieżkę tylko z jednego źródła. Algorytm Floyda-Warshalla oblicza **najkrótsze odległości pomiędzy każdą parą wierzchołków w grafie.**

# Slajd I

**Input:** Graf  $G(V, E, \text{weight})$ .

**Output:** Macierz  $D$  taka, że  $D[i][j]$  jest najkrótszą odległością od wierzchołka  $i$  do  $j$

```
1: for all  $D[i][j] \in D$  do
2:   if  $i == j$  then
3:      $D[i][j] = 0;$ 
4:   end if
5:   if  $(i, j)$  jest krawędzią z  $E$  then
6:      $D[i][j] = \text{weight}(i, j);$ 
7:   else
8:      $D[i][j] = \infty;$ 
9:   end if
10: end for
11: for  $k=1$  to  $|V|$  do
```

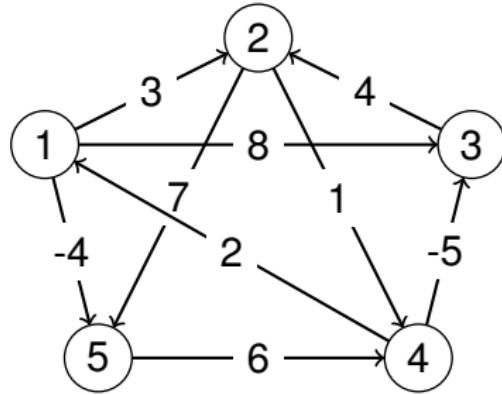
# Slajd II

```
12:   for all  $i=1$  to  $|V|$  do
13:     for all  $j=1$  to  $|V|$  do
14:       if  $D[i][j] > D[i][k] + D[k][j]$  then
15:          $D[i][j] = D[i][k] + D[k][j];$ 
16:       end if
17:     end for
18:   end for
19: end for
```

# Złożoność obliczeniowa I

- Algorytm Floyda-Warshalla można zaimplementować w czasie  $O(|V|^3)$ .
- Algorytm Floyda-Warshall jest zależny tylko od liczby wierzchołków w grafie. Czyni go to szczególnie użytecznym dla grafów gęstych, gdyż w ogóle nie zależy od liczby krawędzi.

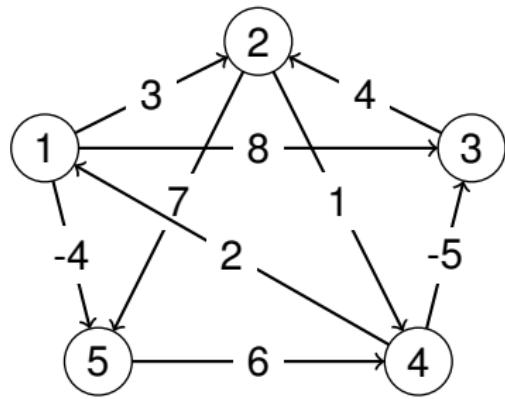
## Slajd I



	1	2	3	4	5
1	0	3	8	$\infty$	-4
2	$\infty$	0	$\infty$	1	7
3	$\infty$	4	0	$\infty$	$\infty$
4	2	$\infty$	-5	0	$\infty$
5	$\infty$	$\infty$	$\infty$	6	0

Macierz  $D$  po wykonaniu linii 1–10 algorytmu Floyda-Warshalla.

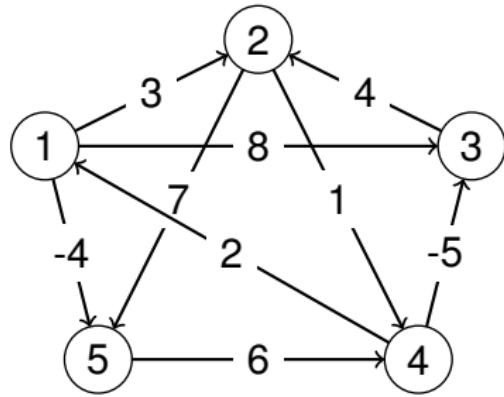
## Slajd II



	1	2	3	4	5
1	0	3	8	$\infty$	-4
2	$\infty$	0	$\infty$	1	7
3	$\infty$	4	0	$\infty$	$\infty$
4	2	5	-5	0	-2
5	$\infty$	$\infty$	$\infty$	6	0

Macierz  $D$  po wykonaniu linii 11–19 algorytmu Floyda-Warshalla, dla  $k=1$ .

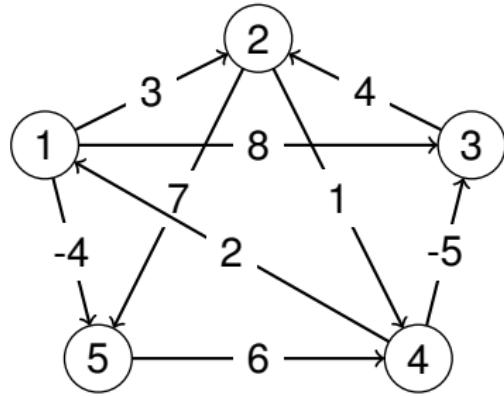
# Slajd III



	1	2	3	4	5
1	0	3	8	4	-4
2	$\infty$	0	$\infty$	1	7
3	$\infty$	4	0	5	11
4	2	5	-5	0	-2
5	$\infty$	$\infty$	$\infty$	6	0

Macierz  $D$  po wykonaniu linii 11–19 algorytmu Floyda-Warshalla, dla  $k=2$ .

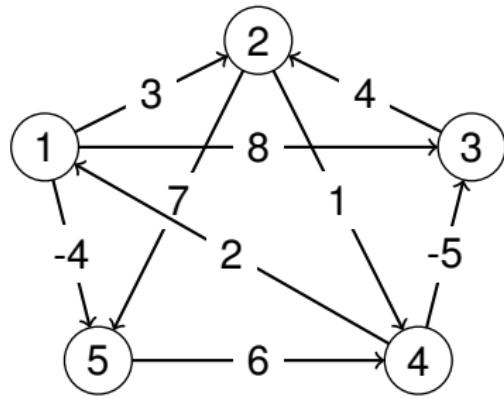
## Slajd IV



	1	2	3	4	5
1	0	3	8	4	-4
2	$\infty$	0	$\infty$	1	7
3	$\infty$	4	0	5	11
4	2	-1	-5	0	-2
5	$\infty$	$\infty$	$\infty$	6	0

Macierz  $D$  po wykonaniu linii 11–19 algorytmu Floyda-Warshalla, dla  $k=3$ .

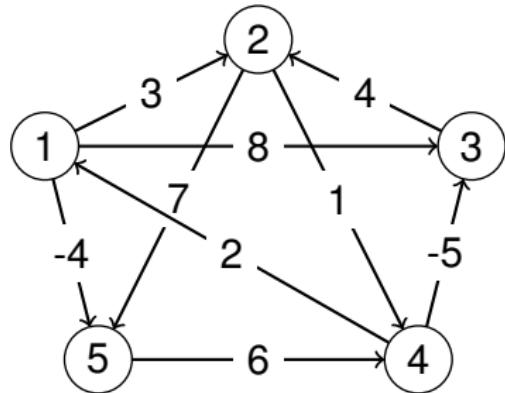
## Slajd V



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

Macierz  $D$  po wykonaniu linii 11–19 algorytmu Floyda-Warshalla, dla  $k=4$ .

## Slajd VI



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

Macierz  $D$  po wykonaniu linii 11–19 algorytmu Floyda-Warshalla, dla  $k=5$ .

# Wprowadzenie, slajd I

- **Algorytm Johnsona** podobnie jak algorytm Floyda-Warshalla zajmuje się znalezieniem najkrótszych ścieżek dla wszystkich par wierzchołków.
- **Algorytm Floyda-Warshalla** jest najbardziej efektywny w przypadku **grafów gęstych** (wiele krawędzi), natomiast gdy **algorytm Johnsona** jest najbardziej skuteczny w przypadku **grafów rzadkich** (niewiele krawędzi).
- Powodem, dla którego **algorytm Johnsona** jest lepszy dla grafów rzadkich, jest to, że jego **złożoność czasowa zależy również od liczby krawędzi w grafie**, podczas gdy w algorytmie Floyda-Warshalla nie.

# Wprowadzenie, slajd II

- Algorytm Johnsona działa w czasie  $O(|V|^2 \cdot \log(|V|) + |V| \cdot |E|)$ . Zatem, jeśli liczba krawędzi jest mała (tzn. graf jest rzadki), będzie on działał szybciej niż algorytm Floyd-Warshall, który działa w czasie  $O(|V|^3)$ .

Algorytm Johnsona składa się z trzech głównych kroków.

- Dodajemy nowy wierzchołek do grafu i łączymy go krawędziami o zerowej wadze z wszystkimi pozostałymi wierzchołkami grafu.
- Wykonujemy dla wszystkich krawędzi proces ponownego ważenia, który eliminuje ujemne wagi, przy pomocy algorytmu Bellmana-Forda.
- Usuwamy wierzchołek dodany w kroku 1 i uruchamiamy algorytm Dijkstry dla każdego węzła w grafie.

# Slajd I

**Require:** Graf  $G(V, E, \text{weight})$ .

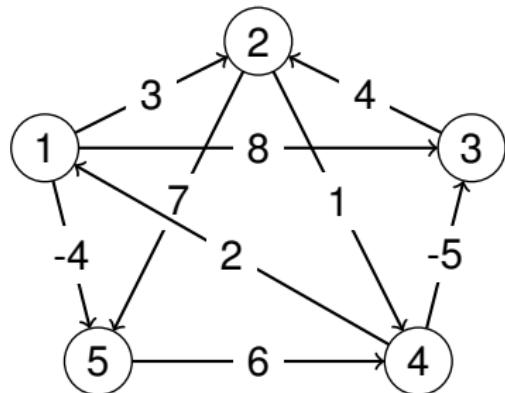
**Ensure:** Macierz  $D$  taka, że  $D[i][j]$  jest najkrótszą odległością od wierzchołka  $i$  do  $j$

- 1: Utwórz  $G'$  taki, że  $G'.V = G.V + \{s\}$ ,  $G'.E = G.E + \{(s, u) |$  dla każdego  $u \in G.V\}$  oraz  $\text{weight}(s, u) = 0$  dla każdego  $u \in G.V$
- 2: **if**  $\text{Bellman} - \text{Ford}(G', s) == \text{False}$  **then**
- 3:   **return** Graf ma ujemny cykl.
- 4: **else**
- 5:   **for all**  $v \in G'.V$  **do**
- 6:      $h(v) = \text{distance}(s, v)$  obliczony przez algorytm Bellmana-Forda
- 7:   **end for**
- 8:   **for all**  $(u, v) \in G'.E$  **do**
- 9:      $\text{weight}'(u, v) = \text{weight}(u, v) + h(u) - h(v);$

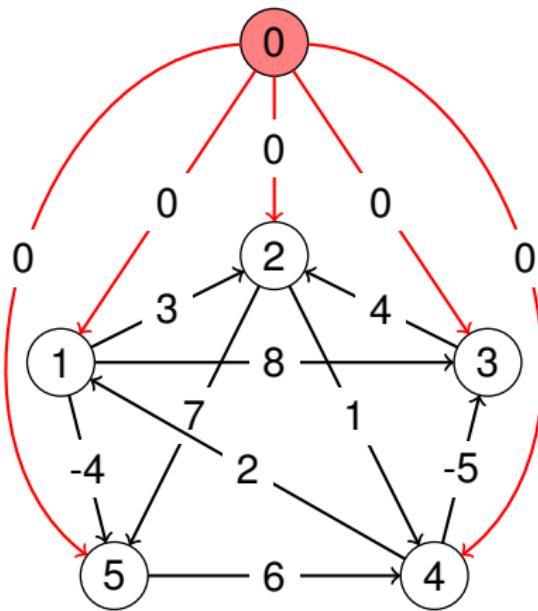
## Slajd II

```
10: end for
11: for all  $D[i][j] \in D$  do
12:    $D[i][j] = \infty;$ 
13: end for
14: for all  $u \in G.V$  do
15:   wykonaj  $Dijkstra(G, weight', u)$  aby obliczyć  $distance'(u, v)$  dla
      każdego  $v \in G.V$ .
16:   for all  $v \in G.V$  do
17:      $D[u][v] = distance'(u, v) + h(v) - h(u);$ 
18:   end for
19: end for
20: end if
```

# Linia 1 algorytmu.



Graf wejściowy  $G$ .



Graf  $G'$  z dodanym wierzchołkiem i krawędziami. L. 1 algorytmu.

# Linie 2-7 algorytmu. Slajd I

- Wiemy, że graf nie ma ujemnych cykli.
- Poniżej wykonanie L. 5-7 algorytmu.
- Wierzchołek źródłowy: **0**.
- Lista krawędzi:  $(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 2), (1, 3), (1, 5), (2, 4), (2, 5), (3, 2), (4, 1), (4, 3), (5, 4)$ .
- Relaksacja, wykonana  $|V'| - 1 = 5$  razy:



①  $i=1$ :

0	1	2	3	4	5
0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$

- $(0, 1)$ :  $1.distance > 0.distance + weight(0, 1)$ .  
 $\infty > 0 + 0$ . **1.distance = 0**.

## Linie 2-7 algorytmu. Slajd II

- $(0, 2): 2.\text{distance} > 0.\text{distance} + \text{weight}(0, 2).$   
 $\infty > 0 + 0.$   **$2.\text{distance} = 0.$**
- $(0, 3): 3.\text{distance} > 0.\text{distance} + \text{weight}(0, 3).$   
 $\infty > 0 + 0.$   **$3.\text{distance} = 0.$**
- $(0, 4): 4.\text{distance} > 0.\text{distance} + \text{weight}(0, 4).$   
 $\infty > 0 + 0.$   **$4.\text{distance} = 0.$**
- $(0, 5): 5.\text{distance} > 0.\text{distance} + \text{weight}(0, 5).$   
 $\infty > 0 + 0.$   **$5.\text{distance} = 0.$**
- $(1, 2): 2.\text{distance} > 1.\text{distance} + \text{weight}(1, 2).$   
 $0 > 0 + 3.$  **brak korekty.**
- $(1, 3): 3.\text{distance} > 1.\text{distance} + \text{weight}(1, 3).$   
 $0 > 0 + 8.$  **brak korekty.**
- $(1, 5): 5.\text{distance} > 1.\text{distance} + \text{weight}(1, 5).$   
 $0 > 0 + (-4).$   **$5.\text{distance} = -4.$**
- $(2, 4): 4.\text{distance} > 2.\text{distance} + \text{weight}(2, 4).$   
 $0 > 0 + 1.$  **brak korekty.**

## Linie 2-7 algorytmu. Slajd III

- $(2, 5): 5.\text{distance} > 2.\text{distance} + \text{weight}(2, 5).$   
 $-4 > 0 + 7.$  brak korekty.
- $(3, 2): 2.\text{distance} > 3.\text{distance} + \text{weight}(3, 2).$   
 $0 > 0 + 4.$  brak korekty.
- $(4, 1): 1.\text{distance} > 4.\text{distance} + \text{weight}(4, 1).$   
 $0 > 0 + 2.$  brak korekty.
- $(4, 3): 3.\text{distance} > 4.\text{distance} + \text{weight}(4, 3).$   
 $0 > 0 + (-5).$  3.distance = -5.
- $(5, 4): 4.\text{distance} > 5.\text{distance} + \text{weight}(5, 4).$   
 $0 > -4 + 6 = 2.$  brak korekty.

② i=2:

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

- $(0, 1): 1.\text{distance} > 0.\text{distance} + \text{weight}(0, 1).$   
 $0 > 0 + 0 = 0.$  brak korekty.

# Linie 2-7 algorytmu. Slajd IV

- $(0, 2)$ :  $2.distance > 0.distance + weight(0, 2)$ .  
 $0 > 0 + 0 = 0$ . brak korekty.
- $(0, 3)$ :  $3.distance > 0.distance + weight(0, 3)$ .  
 $-5 > 0 + 0 = 0$ . brak korekty.
- $(0, 4)$ :  $4.distance > 0.distance + weight(0, 4)$ .  
 $0 > 0 + 0 = 0$ . brak korekty.
- $(0, 5)$ :  $5.distance > 0.distance + weight(0, 5)$ .  
 $-4 > 0 + 0 = 0$ . brak korekty.
- $(1, 2)$ :  $2.distance > 1.distance + weight(1, 2)$ .  
 $0 > 0 + 3 = 3$ . brak korekty.
- $(1, 3)$ :  $3.distance > 1.distance + weight(1, 3)$ .  
 $-5 > 0 + 8 = 8$ . brak korekty.
- $(1, 5)$ :  $5.distance > 1.distance + weight(1, 5)$ .  
 $-4 > 0 - 4 = -4$ . brak korekty.
- $(2, 4)$ :  $4.distance > 2.distance + weight(2, 4)$ .  
 $0 > 0 + 1 = 1$ . brak korekty.

# Linie 2-7 algorytmu. Slajd V

- $(2, 5)$ :  $5.distance > 2.distance + weight(2, 5)$ .  
 $-4 > 0 + 7 = 7$ . brak korekty.
- $(3, 2)$ :  $2.distance > 3.distance + weight(3, 2)$ .  
 $0 > -5 + 4 = -1$ .  **$2.distance = -1$** .
- $(4, 1)$ :  $1.distance > 4.distance + weight(4, 1)$ .  
 $0 > 0 + 2 = 2$ . brak korekty.
- $(4, 3)$ :  $3.distance > 4.distance + weight(4, 3)$ .  
 $-5 > 0 - 5 = -5$ . brak korekty.
- $(5, 4)$ :  $4.distance > 5.distance + weight(5, 4)$ .  
 $0 > -4 + 6 = 2$ . brak korekty.

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

③ i=3:

- $(0, 1)$ :  $1.distance > 0.distance + weight(0, 1)$ .  
 $0 > 0 + 0 = 0$ . brak korekty.

# Linie 2-7 algorytmu. Slajd VI

- $(0, 2)$ :  $2.distance > 0.distance + weight(0, 2)$ .  
 $-1 > 0 + 0 = 0$ . brak korekty.
- $(0, 3)$ :  $3.distance > 0.distance + weight(0, 3)$ .  
 $-5 > 0 + 0 = 0$ . brak korekty.
- $(0, 4)$ :  $4.distance > 0.distance + weight(0, 4)$ .  
 $0 > 0 + 0 = 0$ . brak korekty.
- $(0, 5)$ :  $5.distance > 0.distance + weight(0, 5)$ .  
 $-4 > 0 + 0 = 0$ . brak korekty.
- $(1, 2)$ :  $2.distance > 1.distance + weight(1, 2)$ .  
 $-1 > 0 + 3 = 3$ . brak korekty.
- $(1, 3)$ :  $3.distance > 1.distance + weight(1, 3)$ .  
 $-5 > 0 + 8 = 8$ . brak korekty.
- $(1, 5)$ :  $5.distance > 1.distance + weight(1, 5)$ .  
 $-4 > 0 - 4 = -4$ . brak korekty.
- $(2, 4)$ :  $4.distance > 2.distance + weight(2, 4)$ .  
 $0 > -1 + 1 = 0$ . brak korekty.

# Linie 2-7 algorytmu. Slajd VII

- $(2, 5)$ :  $5.distance > 2.distance + weight(2, 5)$ .  
 $-4 > -1 + 7 = 6$ . brak korekty.
- $(3, 2)$ :  $2.distance > 3.distance + weight(3, 2)$ .  
 $-1 > -5 + 4 = -1$ . brak korekty.
- $(4, 1)$ :  $1.distance > 4.distance + weight(4, 1)$ .  
 $0 > 0 + 2 = 2$ . brak korekty.
- $(4, 3)$ :  $3.distance > 4.distance + weight(4, 3)$ .  
 $-5 > 0 - 5 = -5$ . brak korekty.
- $(5, 4)$ :  $4.distance > 5.distance + weight(5, 4)$ .  
 $0 > -4 + 6 = 2$ . brak korekty.

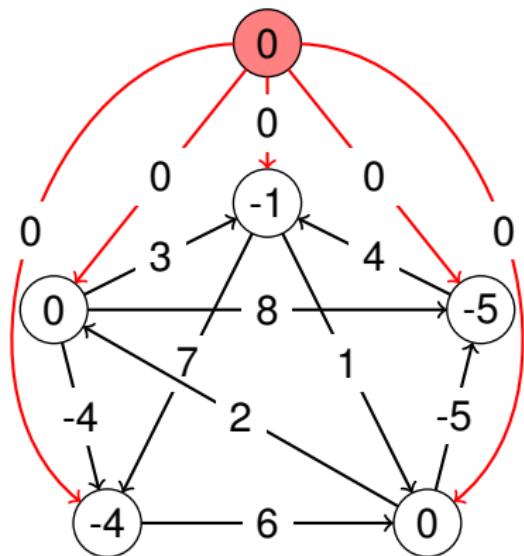
0	1	2	3	4	5
0	0	-1	-5	0	-4

- ④  $i=4, i=5$ : nic nie zmieni.

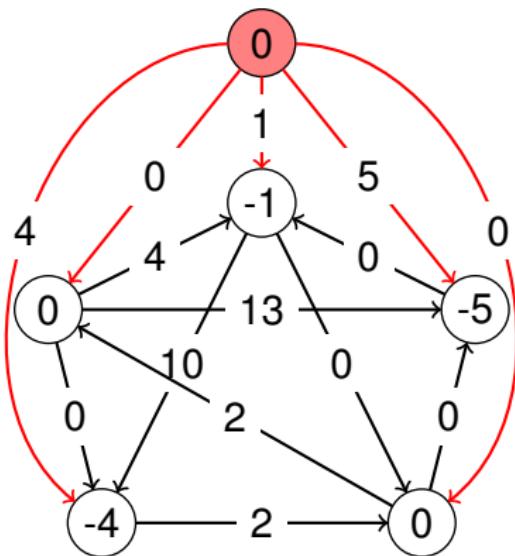
# Linie 8-10 algorytmu. Slajd I

- $\text{weight}'(0, 1) = \text{weight}(0, 1) + h(0) - h(1) = 0 + 0 - 0 = 0,$
- $\text{weight}'(0, 2) = \text{weight}(0, 2) + h(0) - h(2) = 0 + 0 - (-1) = 1,$
- $\text{weight}'(0, 3) = \text{weight}(0, 3) + h(0) - h(3) = 0 + 0 - (-5) = 5,$
- $\text{weight}'(0, 4) = \text{weight}(0, 4) + h(0) - h(4) = 0 + 0 - 0 = 0,$
- $\text{weight}'(0, 5) = \text{weight}(0, 5) + h(0) - h(5) = 0 + 0 - (-4) = 4,$
- $\text{weight}'(1, 2) = \text{weight}(1, 2) + h(1) - h(2) = 3 + 0 - (-1) = 4,$
- $\text{weight}'(1, 3) = \text{weight}(1, 3) + h(1) - h(3) = 8 + 0 - (-5) = 13,$
- $\text{weight}'(1, 5) = \text{weight}(1, 5) + h(1) - h(5) = -4 + 0 - (-4) = 0,$
- $\text{weight}'(2, 4) = \text{weight}(2, 4) + h(2) - h(4) = 1 - 1 - 0 = 0,$
- $\text{weight}'(2, 5) = \text{weight}(2, 5) + h(2) - h(5) = 7 - 1 - (-4) = 10,$
- $\text{weight}'(3, 2) = \text{weight}(3, 2) + h(3) - h(2) = 4 - 5 - (-1) = 0,$
- $\text{weight}'(4, 1) = \text{weight}(4, 1) + h(4) - h(1) = 2 + 0 - 0 = 2,$
- $\text{weight}'(4, 3) = \text{weight}(4, 3) + h(4) - h(3) = -5 + 0 - (-5) = 0,$
- $\text{weight}'(5, 4) = \text{weight}(5, 4) + h(5) - h(4) = 6 - 4 - 0 = 2.$

## Linie 8-10 algorytmu. Slajd II



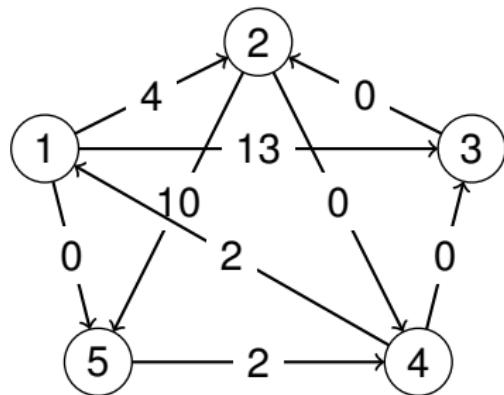
Graf  $G'$  z aktualizacją wag w wierzchołkach po uruchomieniu algorytmu Bellmana-Forda dla wierzchołka 0.



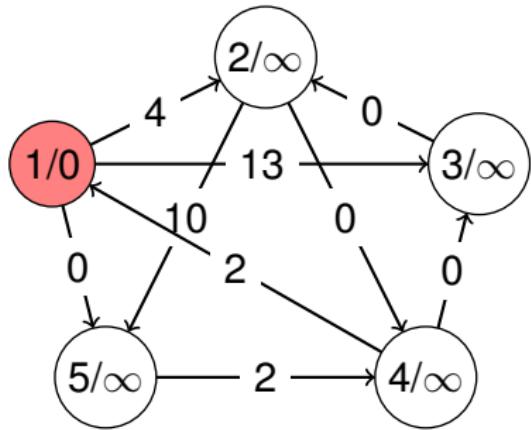
Korekta wag na krawędziach po uruchomieniu algorytmu Bellmana-Forda dla każdego wierzchołka w  $G'$ .

# Linie 11-19 algorytmu. Slajd I

Teraz dla każdego wierzchołka należy wykonać algorytm Dijkstry.  
Niech wierzchołkiem źródłowym będzie: 1

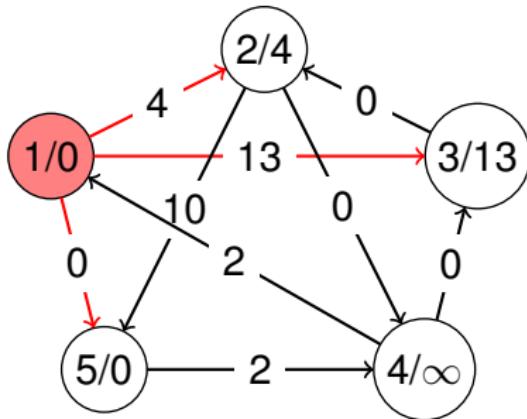


Graf wejściowy

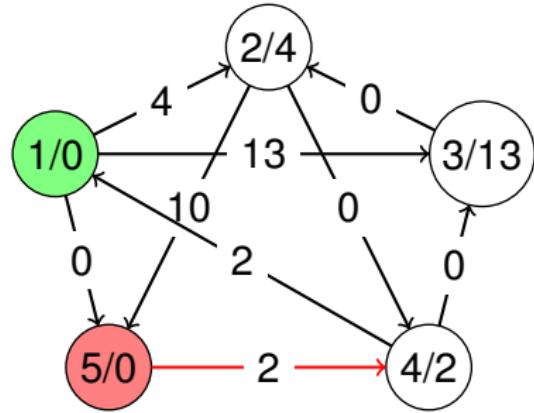


Krok 1. Etykieta  $x/y$  oznacza:  $x$  – nr wierzchołka,  $y$  – aktualną odległość od wierzchołka źródłowego.

## Linie 11-19 algorytmu. Slajd II

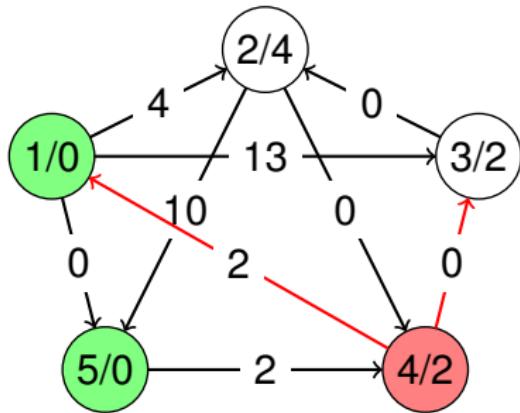


Krok 2:

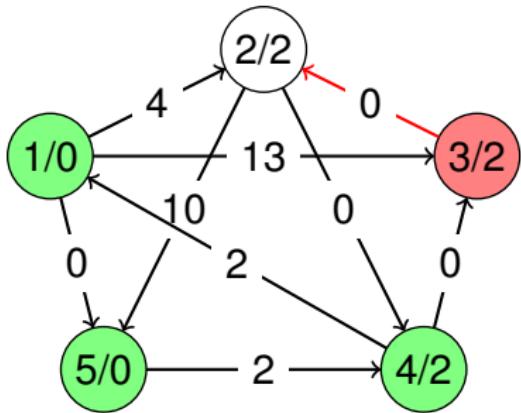


Krok 3:

# Linie 11-19 algorytmu. Slajd III

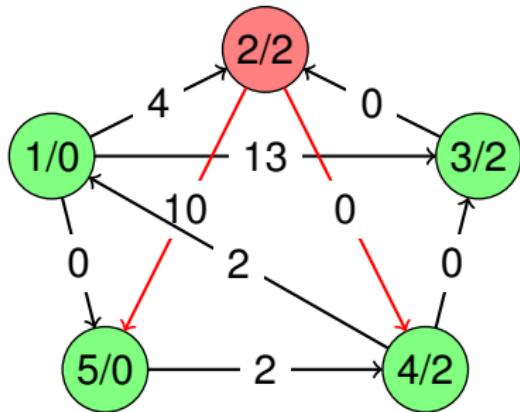


Krok 4:

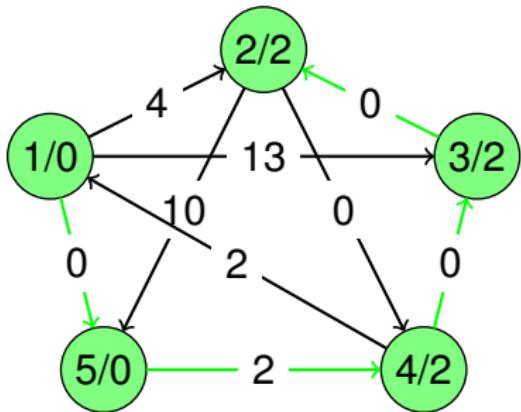


Krok 5:

# Linie 11-19 algorytmu. Slajd IV



Krok 6:



Krok 7:

## Linie 11-19 algorytmu. Slajd V

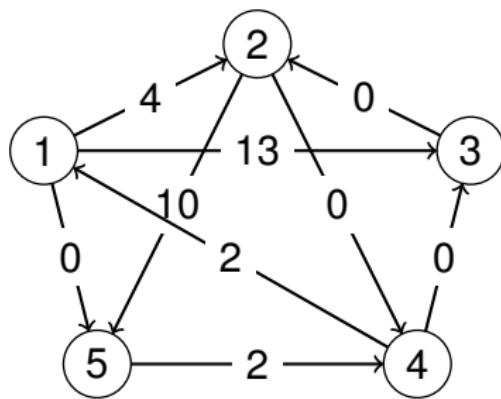
Obliczamy 1 wiersz macierz  $D_{5 \times 5}$ :

- $D[1][1] = \text{distance}'(1, 1) = 0$
- $D[1][2] = \text{distance}'(1, 2) + h(2) - h(1) = 2 + (-1) - 0 = 1$
- $D[1][3] = \text{distance}'(1, 3) + h(3) - h(1) = 2 + (-5) - 0 = -3$
- $D[1][4] = \text{distance}'(1, 4) + h(4) - h(1) = 2 + 0 - 0 = 2$
- $D[1][5] = \text{distance}'(1, 5) + h(5) - h(1) = 0 + (-4) - 0 = -4$

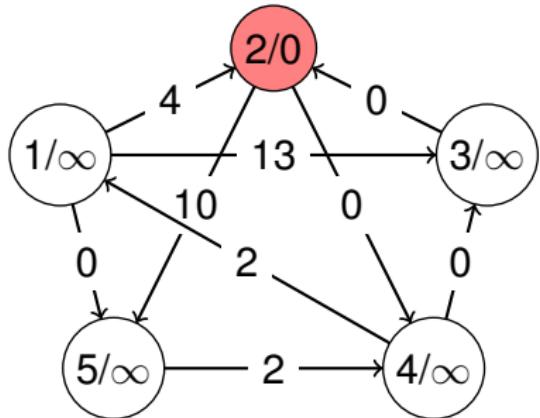
		1	2	3	4	5
1		0	1	-3	2	-4

# Linie 11-19 algorytmu. Slajd VI

Teraz dla każdego wierzchołka należy wykonać algorytm Dijkstry.  
Niech wierzchołkiem źródłowym będzie: 2

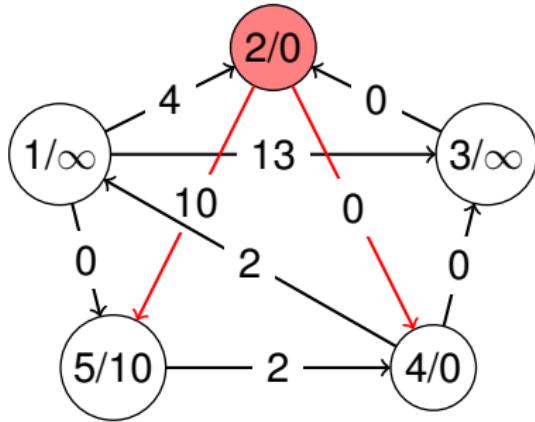


Graf wejściowy

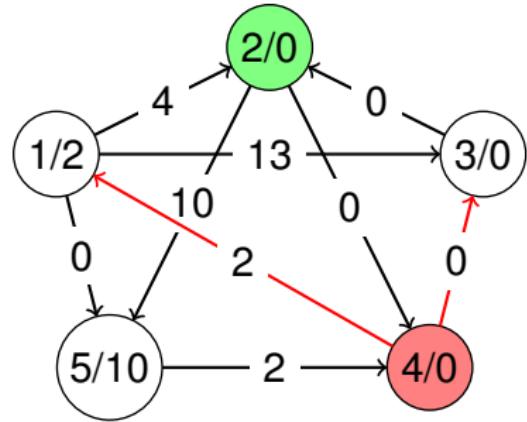


Krok 1. Etykieta  $x/y$  oznacza:  $x$  – nr wierzchołka,  $y$  – aktualną odległość od wierzchołka źródłowego.

# Linie 11-19 algorytmu. Slajd VII

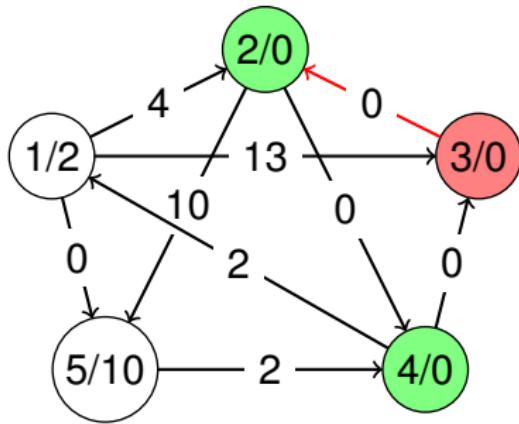


Krok 2.

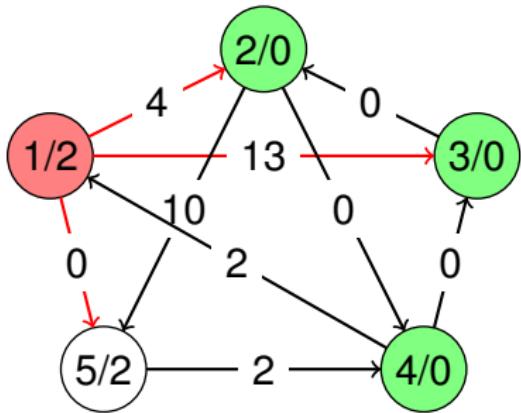


Krok 3.

# Linie 11-19 algorytmu. Slajd VIII

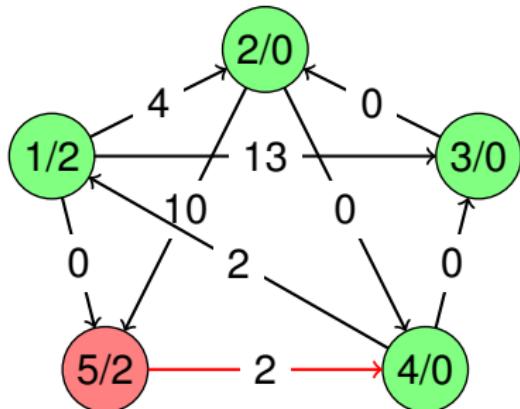


Krok 4.

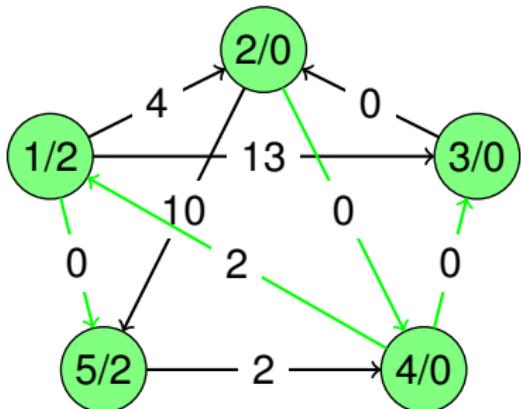


Krok 5.

# Linie 11-19 algorytmu. Slajd IX



Krok 6.



Krok 7.

## Linie 11-19 algorytmu. Slajd X

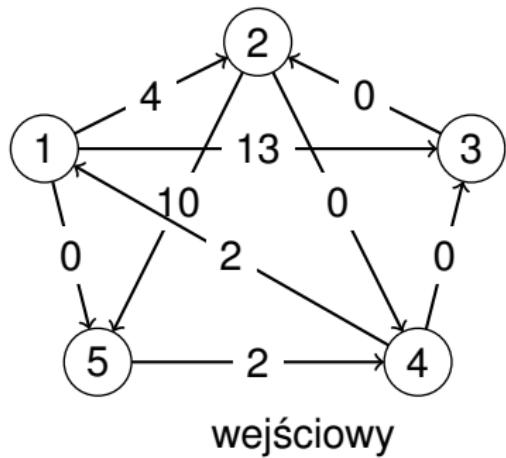
Obliczamy 2 wiersz macierz  $D_{5 \times 5}$ :

- $D[2][1] = \text{distance}'(2, 1) + h(1) - h(2) = 2 + 0 - (-1) = 3$
- $D[2][2] = \text{distance}'(2, 2) + h(2) - h(2) = 0$
- $D[2][3] = \text{distance}'(2, 3) + h(3) - h(2) = 0 + (-5) - (-1) = -4$
- $D[2][4] = \text{distance}'(2, 4) + h(4) - h(2) = 0 + 0 - (-1) = 1$
- $D[2][5] = \text{distance}'(2, 5) + h(5) - h(2) = 2 + (-4) - (-1) = -1$

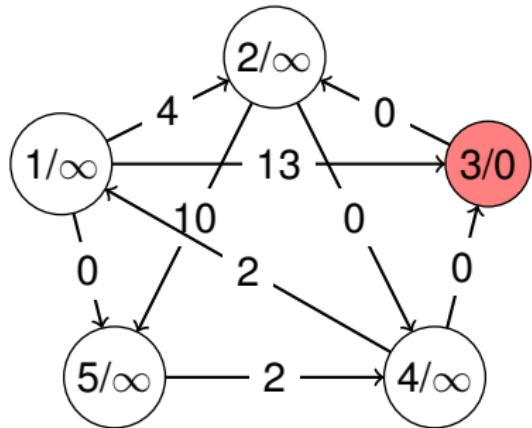
	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1

# Linie 11-19 algorytmu. Slajd XI

Teraz dla każdego wierzchołka należy wykonać algorytm Dijkstry.  
Niech wierzchołkiem źródłowym będzie: 3

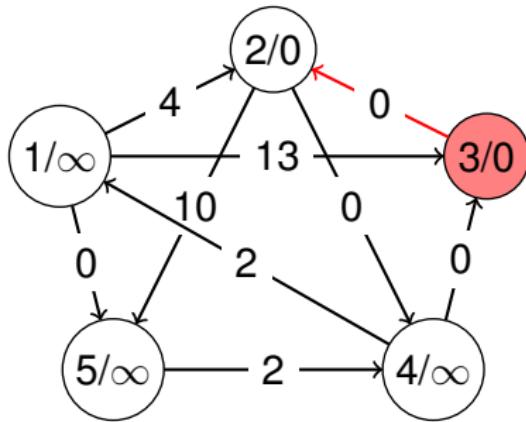


Graf

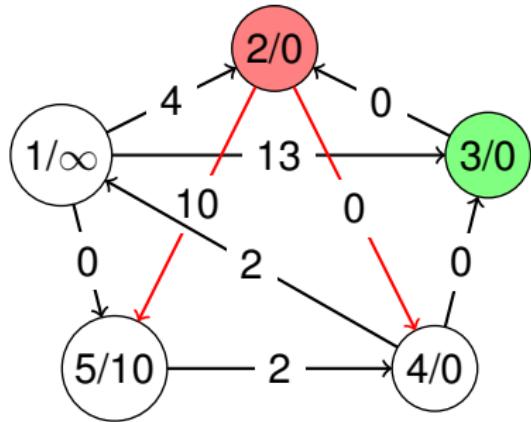


Krok 1. Etykieta  $x/y$  oznacza:  $x$  – nr wierzchołka,  $y$  – aktualną odległość od wierzchołka źródłowego.

# Linie 11-19 algorytmu. Slajd XII

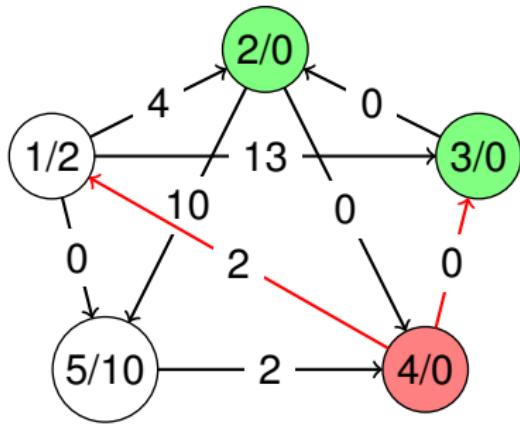


Krok 2.

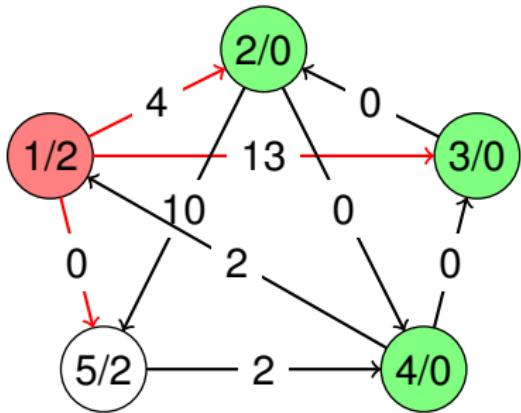


Krok 3.

# Linie 11-19 algorytmu. Slajd XIII

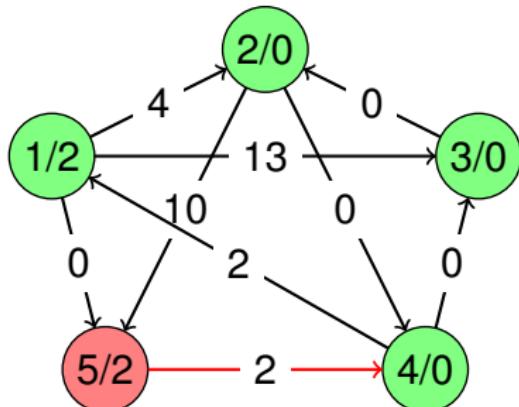


Krok 4.

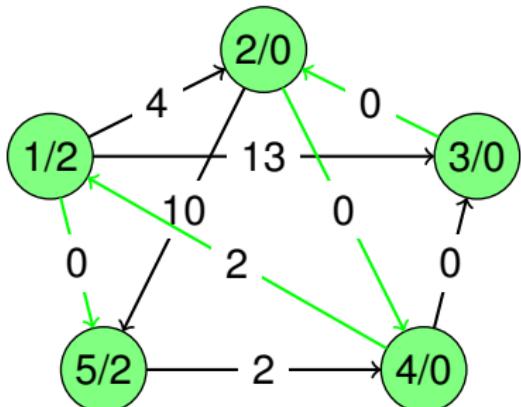


Krok 5.

# Linie 11-19 algorytmu. Slajd XIV



Krok 6.



Krok 7.

## Linie 11-19 algorytmu. Slajd XV

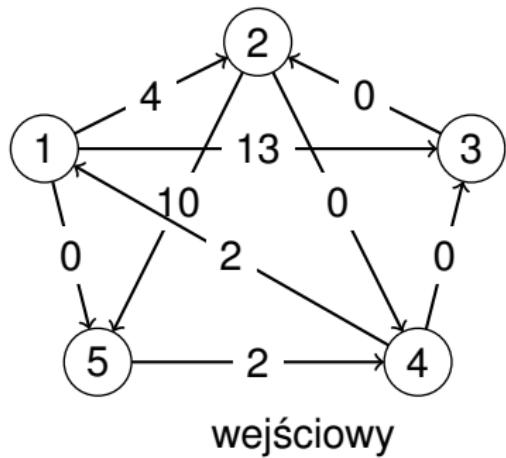
Obliczamy 3 wiersz macierz  $D_{5 \times 5}$ :

- $D[3][1] = \text{distance}'(3, 1) + h(1) - h(3) = 2 + 0 - (-5) = 7$
- $D[3][2] = \text{distance}'(3, 2) + h(2) - h(3) = 0 + (-1) - (-5) = 4$
- $D[3][3] = \text{distance}'(3, 3) + h(3) - h(3) = 0$
- $D[3][4] = \text{distance}'(3, 4) + h(4) - h(3) = 0 + 0 - (-5) = 5$
- $D[3][5] = \text{distance}'(3, 5) + h(5) - h(3) = 2 + (-4) - (-5) = 3$

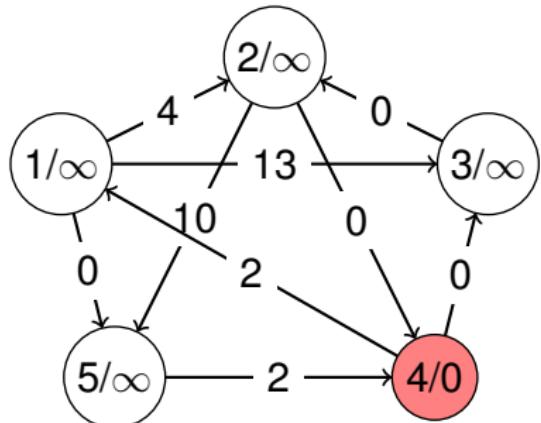
	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3

# Linie 11-19 algorytmu. Slajd XVI

Teraz dla każdego wierzchołka należy wykonać algorytm Dijkstry.  
Niech wierzchołkiem źródłowym będzie: 4

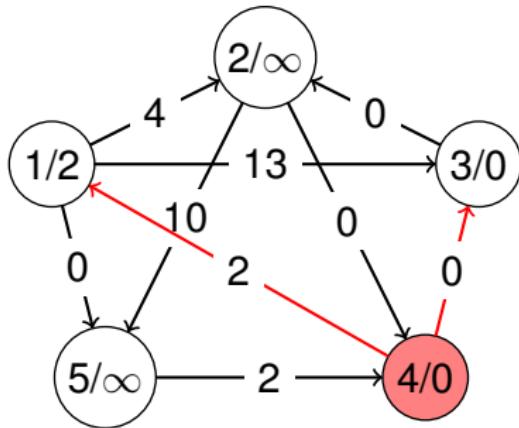


Graf

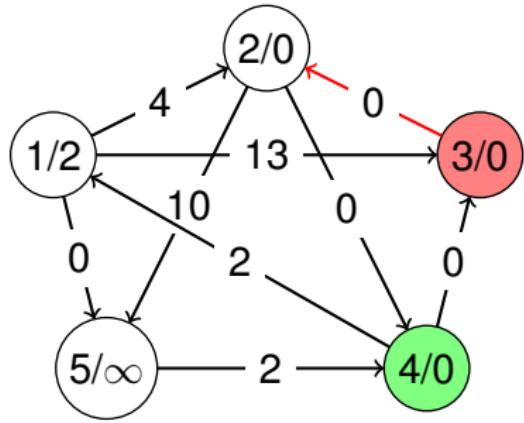


Krok 1. Etykieta  $x/y$  oznacza:  $x$  – nr wierzchołka,  $y$  – aktualną odległość od wierzchołka źródłowego.

# Linie 11-19 algorytmu. Slajd XVII

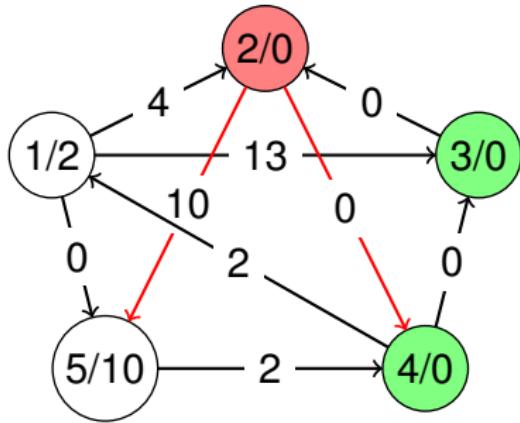


Krok 2.

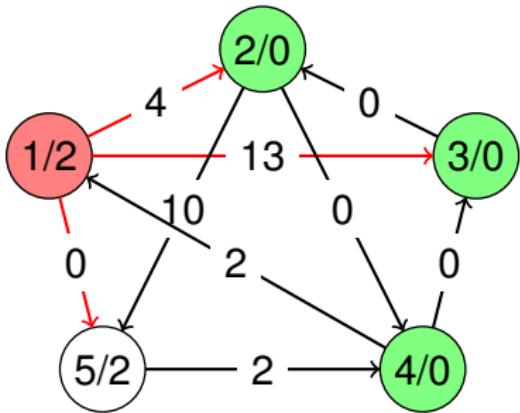


Krok 3.

# Linie 11-19 algorytmu. Slajd XVIII

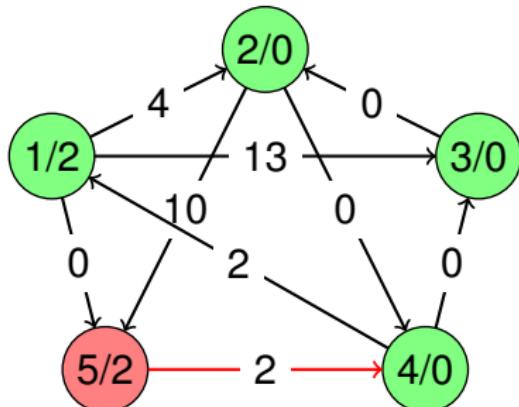


Krok 4.

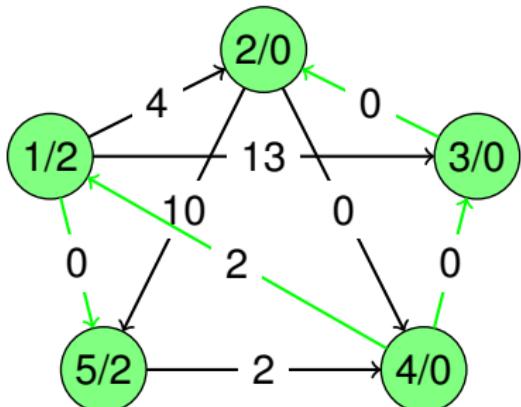


Krok 5.

# Linie 11-19 algorytmu. Slajd XIX



Krok 6.



Krok 7.

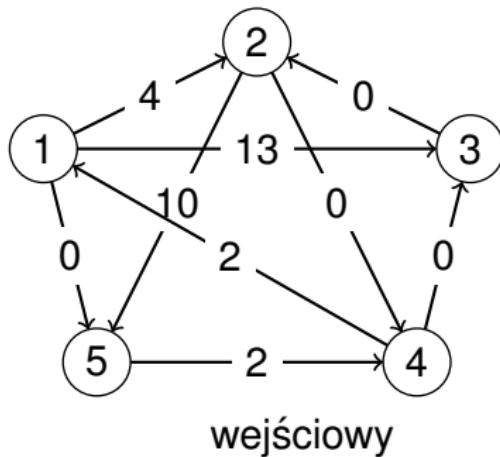
# Linie 11-19 algorytmu. Slajd XX

Obliczamy 4 wiersz macierz  $D_{5 \times 5}$ :

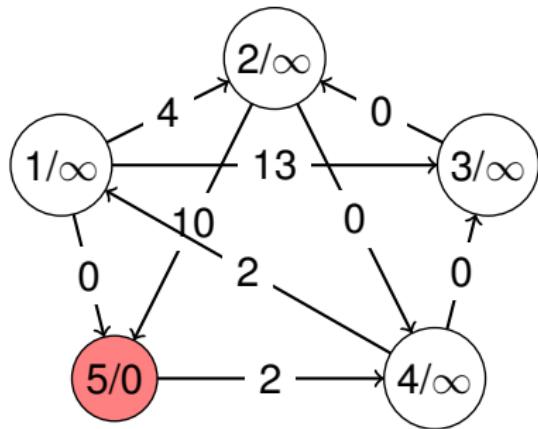
- $D[4][1] = \text{distance}'(4, 1) + h(1) - h(4) = 2 + 0 - 0 = 2$
- $D[4][2] = \text{distance}'(4, 2) + h(2) - h(4) = 0 + (-1) - 0 = -1$
- $D[4][3] = \text{distance}'(4, 3) + h(3) - h(4) = 0 + (-5) - 0 = -5$
- $D[4][4] = \text{distance}'(4, 4) + h(4) - h(4) = 0 + 0 - 0 = 0$
- $D[4][5] = \text{distance}'(4, 5) + h(5) - h(4) = 2 + (-4) - 0 = -2$

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2

# Linie 11-19 algorytmu. Slajd XXI

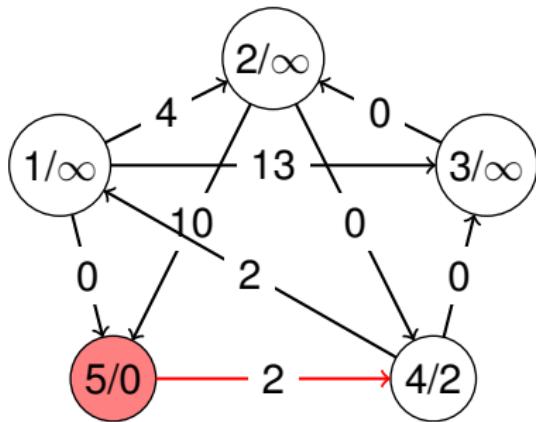


Graf

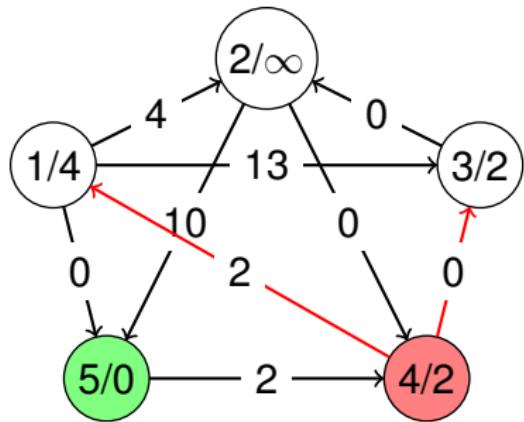


Krok 1. Etykieta  $x/y$  oznacza:  $x$  – nr wierzchołka,  $y$  – aktualną odległość od wierzchołka źródłowego.

# Linie 11-19 algorytmu. Slajd XXII

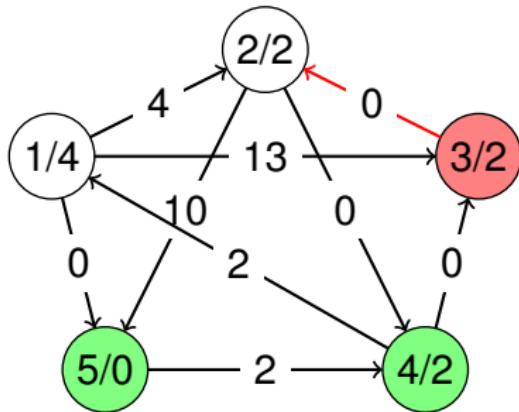


Krok 2.

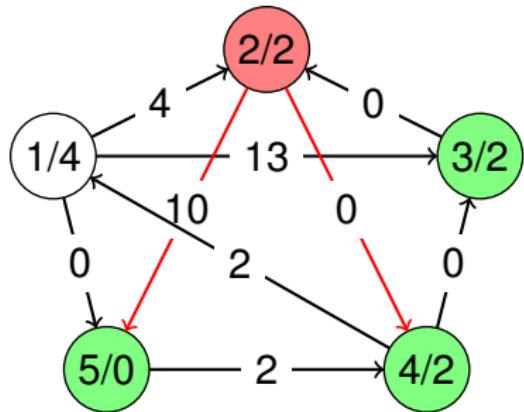


Krok 3.

# Linie 11-19 algorytmu. Slajd XXIII

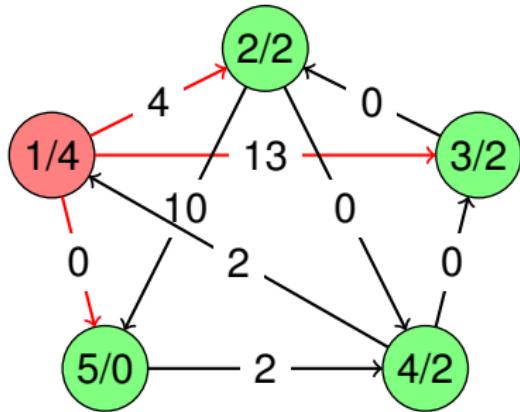


Krok 4.

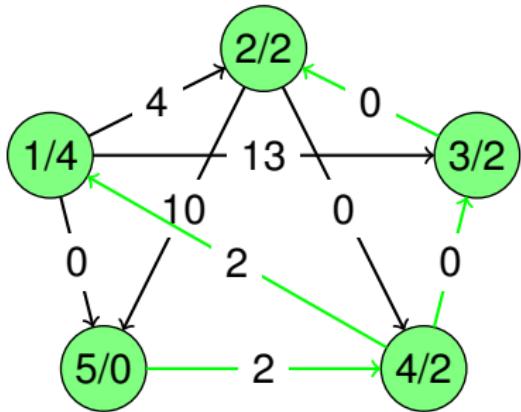


Krok 5.

# Linie 11-19 algorytmu. Slajd XXIV



Krok 6.



Krok 7.

# Linie 11-19 algorytmu. Slajd XXV

Obliczamy 5 wiersz macierz  $D_{5 \times 5}$ :

- $D[5][1] = \text{distance}'(5, 1) + h(1) - h(5) = 4 + 0 - (-4) = 8$
- $D[5][2] = \text{distance}'(5, 2) + h(2) - h(5) = 2 + (-1) - (-4) = 5$
- $D[5][3] = \text{distance}'(5, 3) + h(3) - h(5) = 2 + (-5) - (-4) = 1$
- $D[5][4] = \text{distance}'(5, 4) + h(4) - h(5) = 2 + 0 - (-4) = 6$
- $D[5][5] = \text{distance}'(5, 5) + h(5) - h(5) = 0$

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