

Najkrótsza droga w grafie

Algorytm **Dijkstry** i **Bellmana-Forda** służy do wyznaczenia najkrótszej drogi pomiędzy wierzchołkiem startowym (zwanym często źródłowym) do wszystkich wierzchołków

Elementy obu algorytmów

d[v] – aktualna odległość wierzchołka **v** od wierzchołka startowego

p[v] – wierzchołek z którego osiągnięto wierzchołek **v**

Ciąg poprzedników począwszy od **v** wyznacza drogę do wierzchołka początkowego

$v \rightarrow p[v] \rightarrow p[p[v]] \rightarrow p[p[p[v]]] \rightarrow \dots$ aż do w. startowego

w(u, v) – waga krawędzi pomiędzy wierzchołkami **u** i **v**

Najkrótsza droga w grafie

RELAKSACJA KRAWĘDZI

Jeżeli dochodząc do danego wierzchołka v z wierzchołka u poprawiamy dotychczasową drogę ($p[v]$)

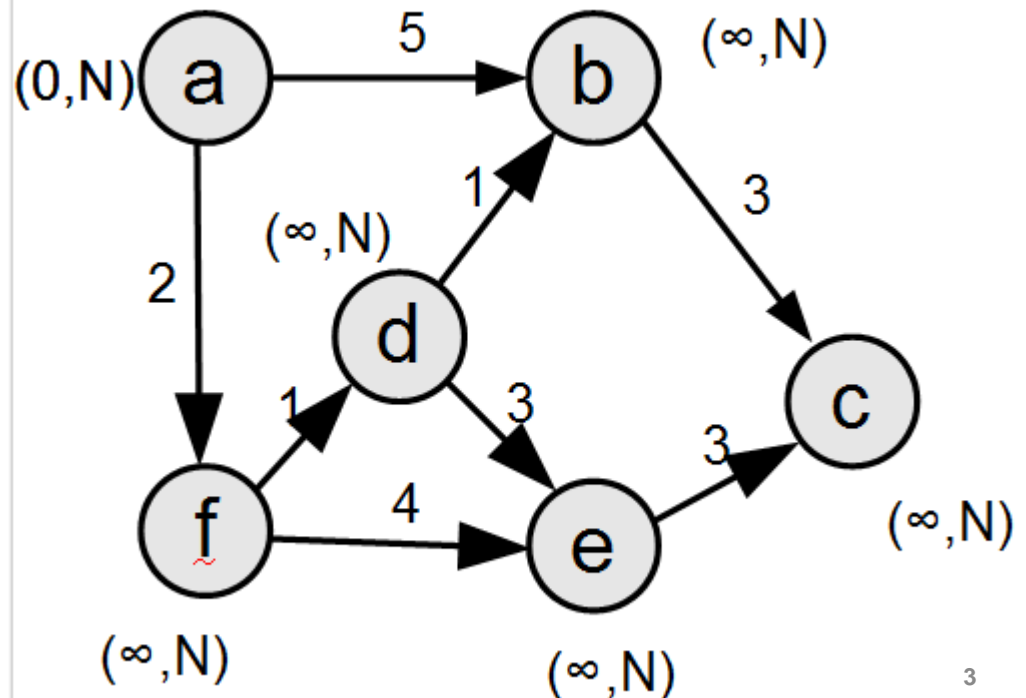
```
if  $d[v] > d[u] + w(u, v)$  then           //relaksacja krawędzi  
     $d[v] = d[u] + w(u, v)$   
     $p[v] = u;$ 
```

Najkrótsza droga w grafie – algorytm Dijkstry

Dla każdego wierzchołka pamiętamy dwie dane (d, p)
-odległość od wierzchołka startowego ($d[v]$),
-wierzchołek poprzedzający ($p[v]$)

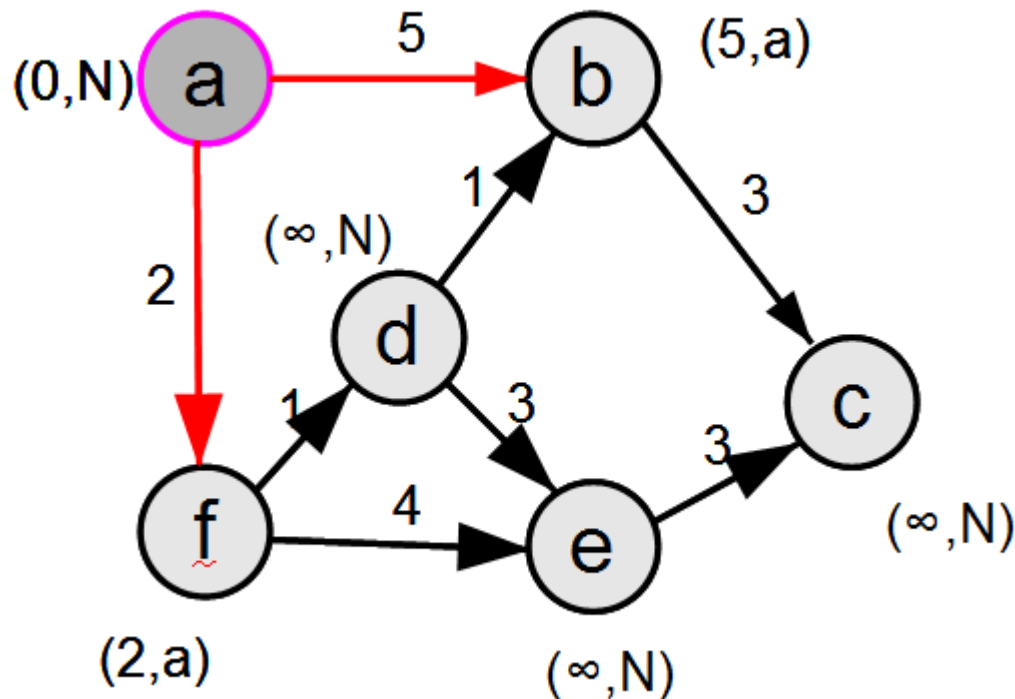
Dwa zbiory wierzchołków : przebadane i nieprzebadane (na początku wszystkie są nieprzebadane)

a – wierzchołek startowy, **N** – oznacza poprzednika nieokreślonego



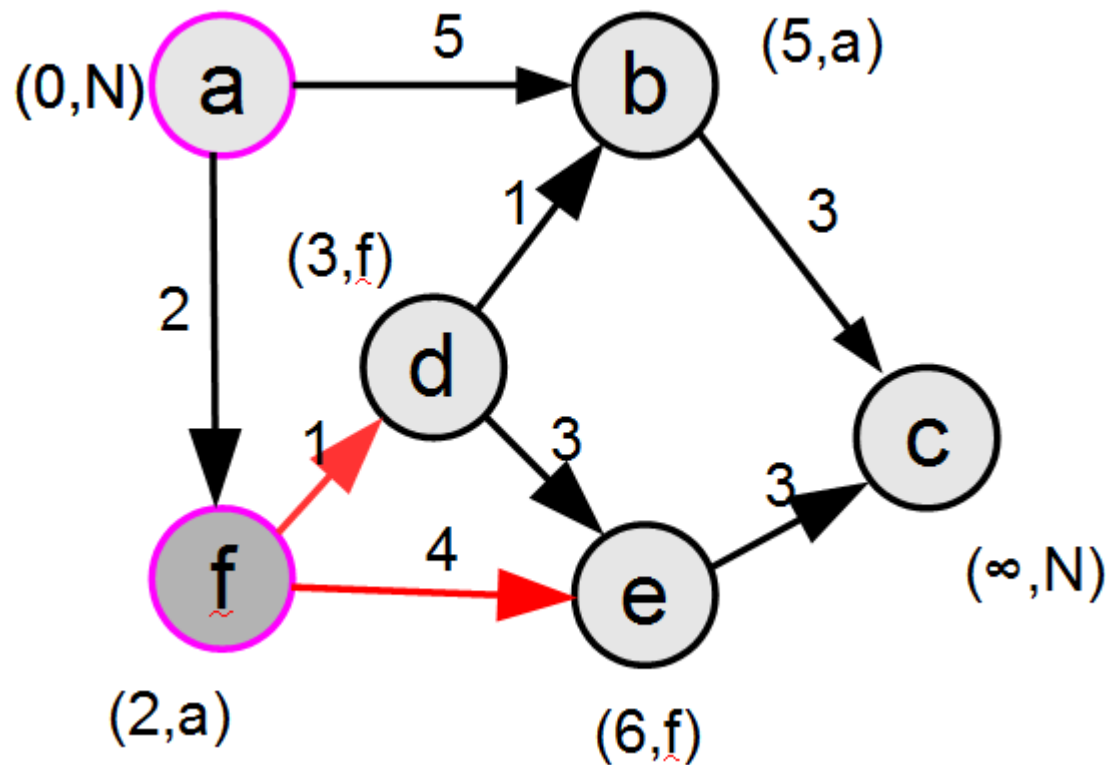
Najkrótsza droga w grafie – algorytm Dijkstry

- Ze zbioru nieprzebadanych wierzchołków (a, b, c, d, e, f) wybieramy wierzchołek o najmniejszej wartości odległości (wierzchołek **a**)
- Dokonujemy relaksacji jego sąsiadów (b, f)
- Wierzchołek **a** usuwamy ze zbioru nieprzebadanych



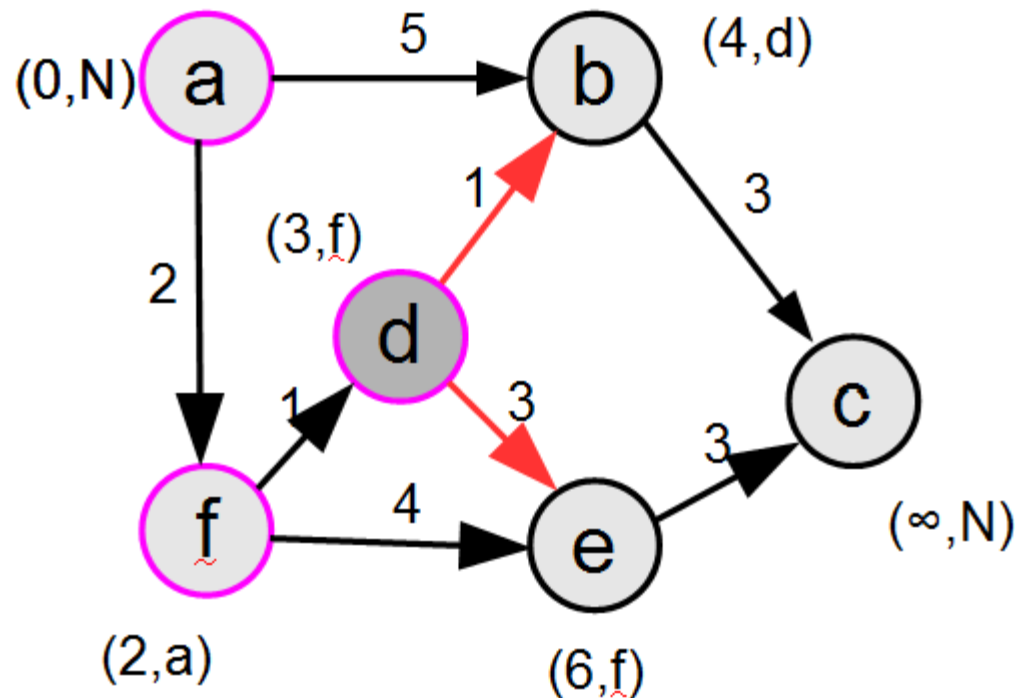
Najkrótsza droga w grafie – algorytm Dijkstry

- Ze zbioru nieprzebadanych wierzchołków (b, c, d, e, f) wybieramy wierzchołek o najmniejszej wartości odległości (wierzchołek f)
- Dokonujemy relaksacji jego sąsiadów (d, e)
- Wierzchołek f usuwamy ze zbioru nieprzebadanych



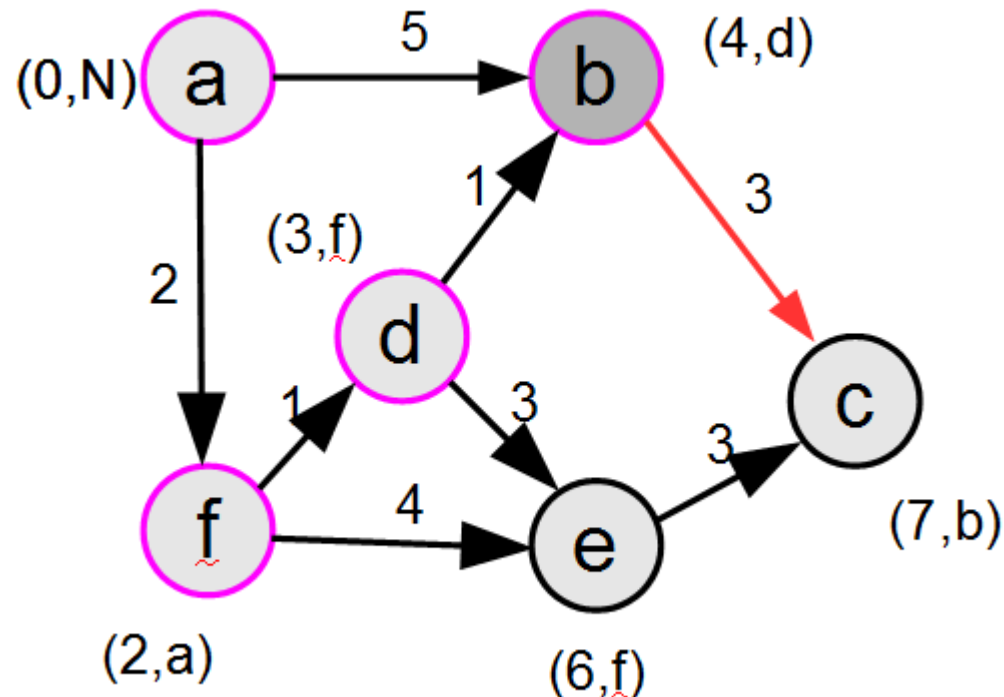
Najkrótsza droga w grafie – algorytm Dijkstry

- Ze zbioru nieprzebadanych wierzchołków (b, c, d, e) wybieramy wierzchołek o najmniejszej wartości odległości (wierzchołek **d**)
- Dokonujemy relaksacji jego sąsiadów (b, e)
- Wierzchołek **d** usuwamy ze zbioru nieprzebadanych



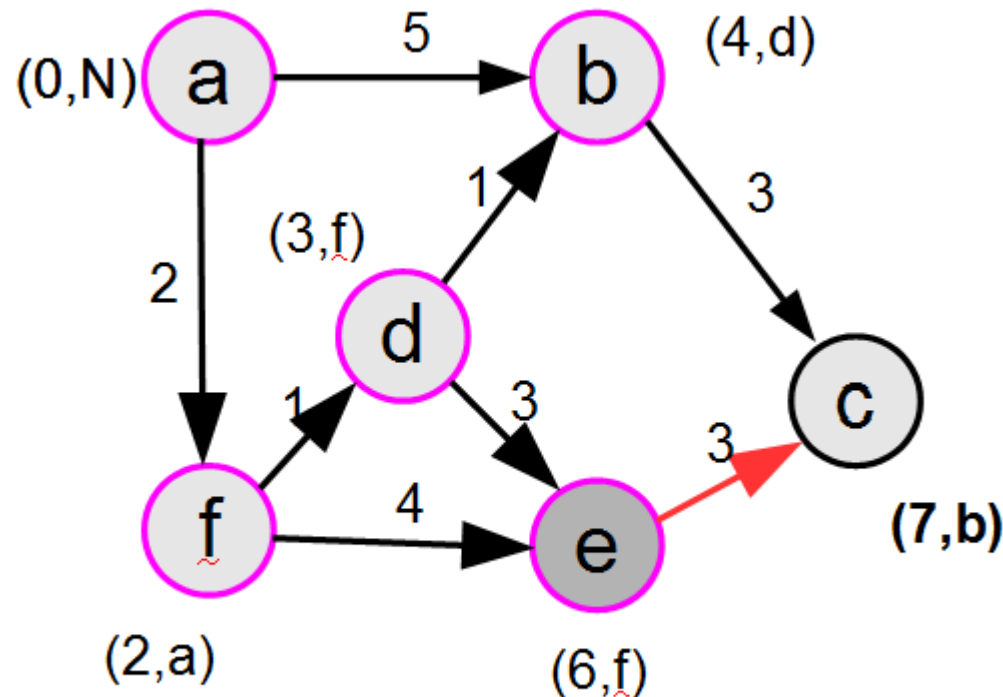
Najkrótsza droga w grafie – algorytm Dijkstry

- Ze zbioru nieprzebadanych wierzchołków (b, c, e) wybieramy wierzchołek o najmniejszej wartości odległości (wierzchołek **b**)
- Dokonujemy relaksacji jego sąsiadów (c)
- Wierzchołek **b** usuwamy ze zbioru nieprzebadanych



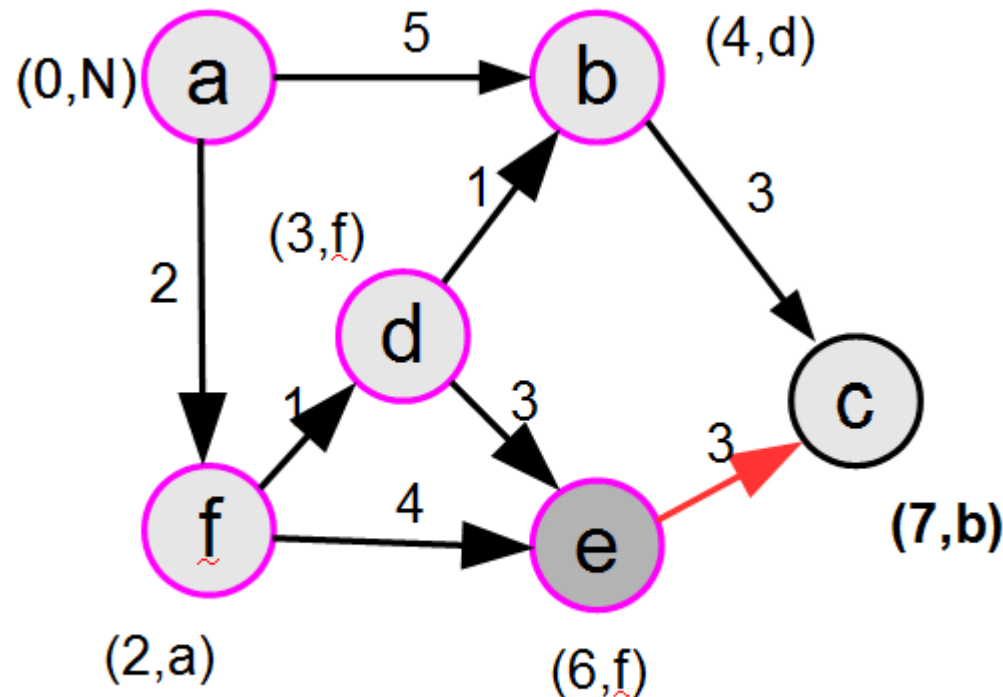
Najkrótsza droga w grafie – algorytm Dijkstry

- Ze zbioru nieprzebadanych wierzchołków (c, e) wybieramy wierzchołek o najmniejszej wartości odległości (wierzchołek **e**)
- Dokonujemy relaksacji jego sąsiadów (b, e)
- Wierzchołek **e** usuwamy ze zbioru nieprzebadanych



Najkrótsza droga w grafie – algorytm Dijkstry

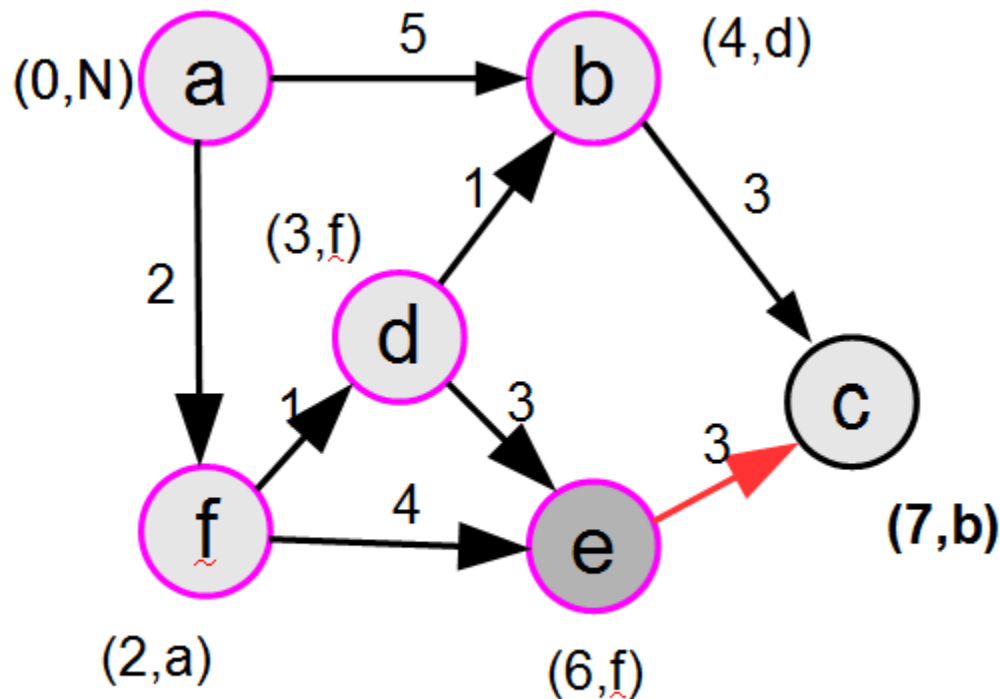
- Ze zbioru nieprzebadanych wierzchołków(**c**) wybieramy wierzchołek o najmniejszej wartości odległości (wierzchołek **c**)
- Dokonujemy relaksacji jego sąsiadów – brak sąsiadów
- Wierzchołek **c** usuwamy ze zbioru nieprzebadanych



Najkrótsza droga w grafie – algorytm Dijkstry

- Zbiór nieprzebadanych jest pusty więc STOP
- Przykładowa droga dla wierzchołka c od wierzchołka startowego (odczytywana od tyłu czyli od **c** do **a** na podstawie poprzedników

c->b->d->f->a



Najkrótsza droga w grafie – algorytm Dijkstry

G – graf, V – zbiór wierzchołków

$d[v]$ – aktualna odległość od wierzchołka startowego

$p[v]$ – wierzchołek z którego osiągnięto wierzchołek v

$\text{Adj}[v]$ – zbiór sąsiadów wierzchołka v

$w(u, v)$ – waga, Q – kolejka (zbiór wierzchołków)

Dijkstra(G, w, s)

 for each $v \in V$ do

$d[v] = \infty$

$p[v] = \text{NULL}$

$d[s] = 0$;

$Q = V[G]$

 while $Q \neq \emptyset$ do

 //usuń wierzchołek o najmniejszej $d[i]$

$u := \text{Extract-Min}(Q)$

 for each $v \in \text{Adj}[u]$ do

 if $d[v] > d[u] + w(u, v)$ then

 //relaksacja krawędzi

$d[v] = d[u] + w(u, v)$

$p[v] = u$;

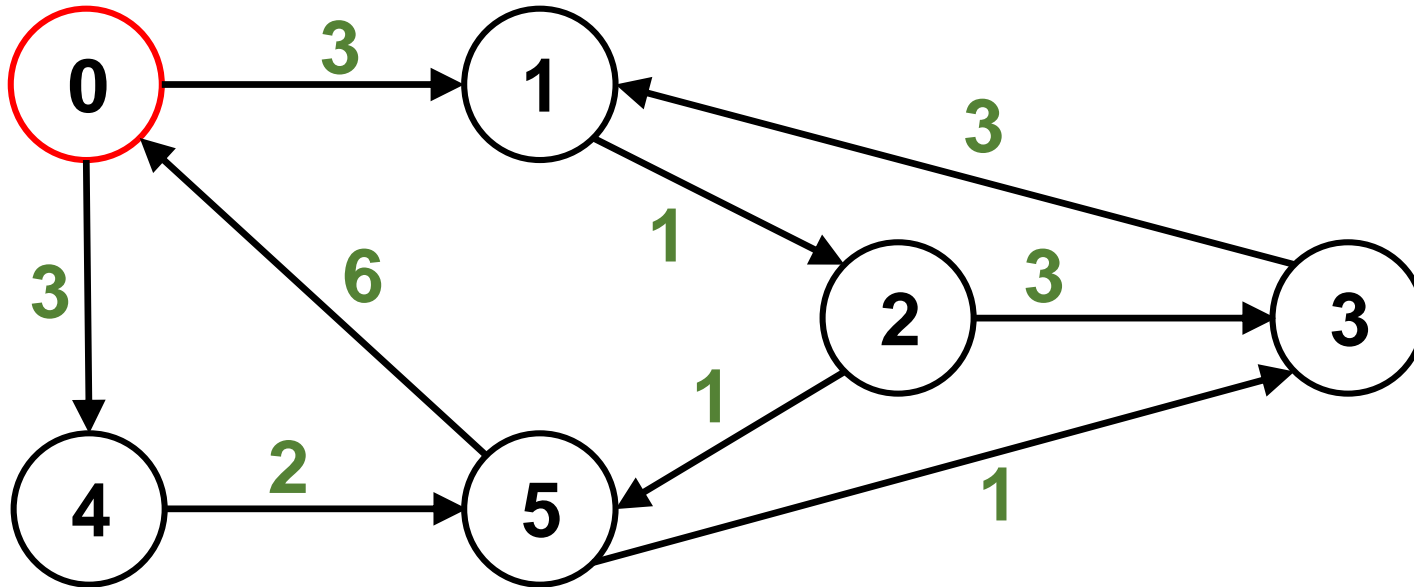
Najkrótsza droga w grafie – algorytm Dijkstry

O rzędzie złożoności decyduje implementacja kolejki priorytetowej:

- wykorzystując "naiwną" implementację poprzez zwykłą tablicę, otrzymujemy algorytm o złożoności **$O(V^2)$**
- w implementacji kolejki poprzez kopiec, złożoność wynosi **$O(E \log V)$**
- po zastąpieniu zwykłego kopca kopcem Fibonacciego złożoność zmniejsza się do **$O(E + V \log V)$**

Pierwszy wariant jest optymalny dla grafów gęstych, drugi jest szybszy dla grafów rzadkich, trzeci jest bardzo rzadko używany ze względu na duży stopień skomplikowania i niewielki w porównaniu z nim zysk czasowy.

Najkrótsza droga w grafie – algorytm Dijkstry



PRZERYSOWAĆ DO NOTATEK

Najkrótsza droga w grafie – algorytm Bellmana-Forda

Idea algorytmu opiera się na metodzie relaksacji (dokładniej następuje relaksacja $V-1$ razy każdej z krawędzi).

W odróżnieniu od algorytmu Dijkstry, poprawność algorytmu Bellmana-Forda nie opiera się na założeniu, że wagi w grafie są nieujemne (nie może jednak występować cykl o łącznej ujemnej wadze osiągalny ze źródła). Za tę ogólność płaci się jednak wyższą złożonością czasową.

Działa on w czasie $O(V \cdot E)$.

Najkrótsza droga w grafie – algorytm Bellmana-Forda

```
BelmannFord(G, w, s)
  for each v ∈ V[G] do
    d[v] := ∞
    p[v] := NULL
  d[s] := 0;
  for i := 1 to |V[G]| - 1 do
    for each (u,v) ∈ E do
      if d[v] > d[u] + w(u, v) then
        //relaksacja krawędzi
        d[v] := d[u] + w(u, v)
        p[v] := u;

  //sprawdzanie czy nie ma cyklu ujemnego
  for each (u,v) ∈ E do
    if d[v] > d[u] + w(u, v) then
      STOP - cykl ujemny
```

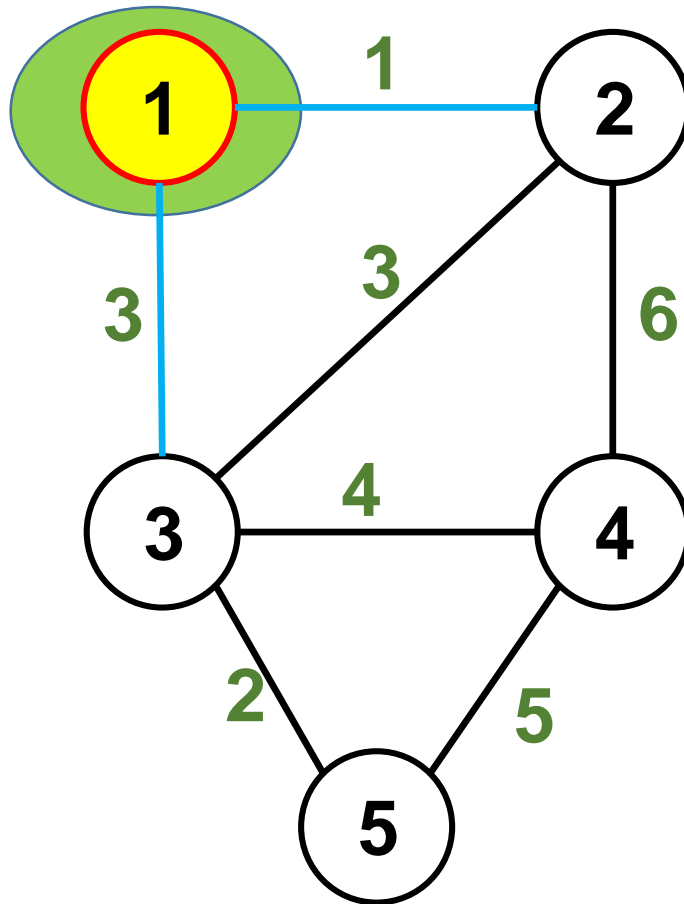
MST – algorytm Prima

Algorytm Prima

1. Wybierz wierzchołek startowy i dodaj do zbioru rozwiązań.
2. Utwórz listę wszystkich wierzchołków połączonych z wierzchołkami ze zbioru rozwiązań.
3. Z listy wybierz połączenie (krawędź) o najmniejszej wadze i dodaj wierzchołek do zbioru rozwiązań, a krawędź dołącz do zbioru krawędzi MST
4. Jeżeli zbiór rozwiązań zawiera wszystkie wierzchołki to koniec, w przeciwnym wypadku przejdź do pkt. 2.

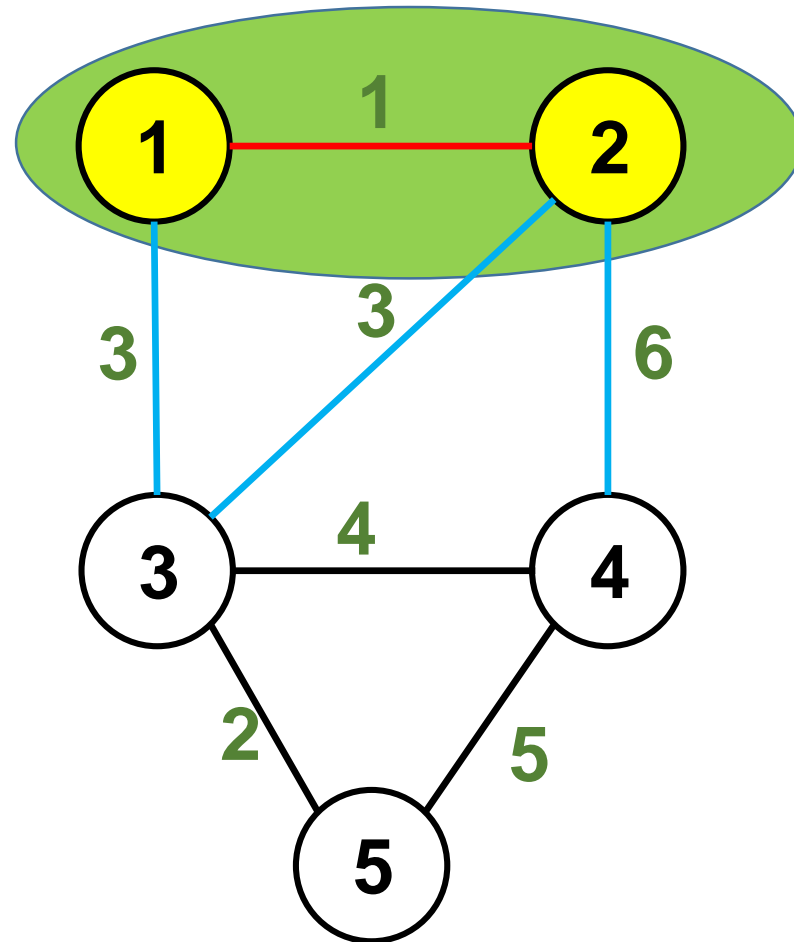
MST – graf

Wybieramy wierzchołek startowy np. 1 i przenosimy go do zbioru rozwiązań (zbiór rozwiązań zawiera teraz tylko wierzchołek 1)



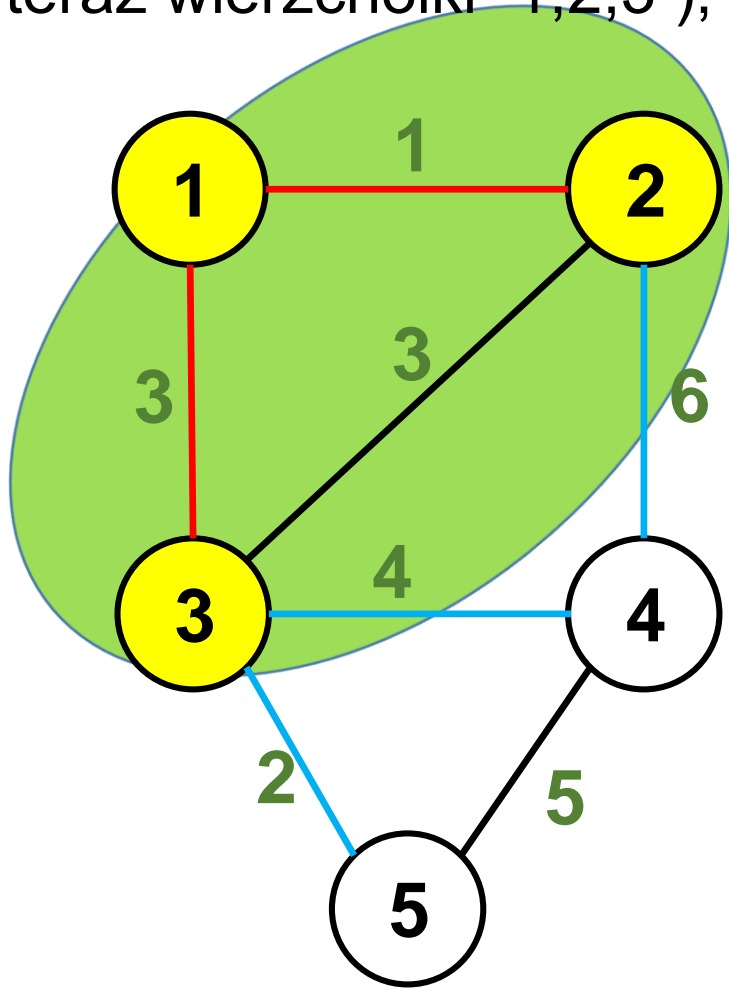
MST – graf

Wybieramy wierzchołek połączony z wierzchołkiem 1 o najmniejszej wadze tj. 2 i dodajemy go do zbioru rozwiązań (zbiór rozwiązań zawiera teraz wierzchołki 1,2 a krawędź 1-2 do drzewa MST)



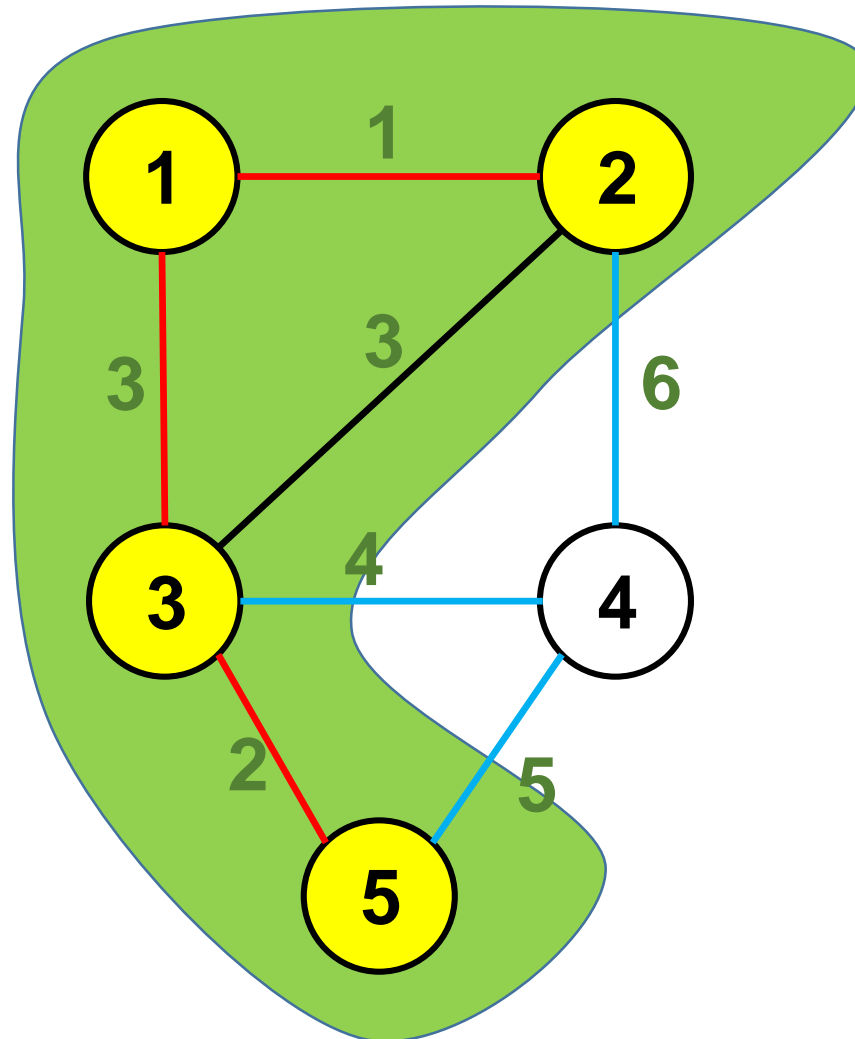
MST – graf

Wybieramy najbliższy wierzchołek dla 1,2 o najmniejszej wadze tj. 3 i dodajemy go do zbioru rozwiązań (zbiór rozwiązań zawiera teraz wierzchołki 1,2,3), a krawędź 1-3 do zbioru krawędzi MST



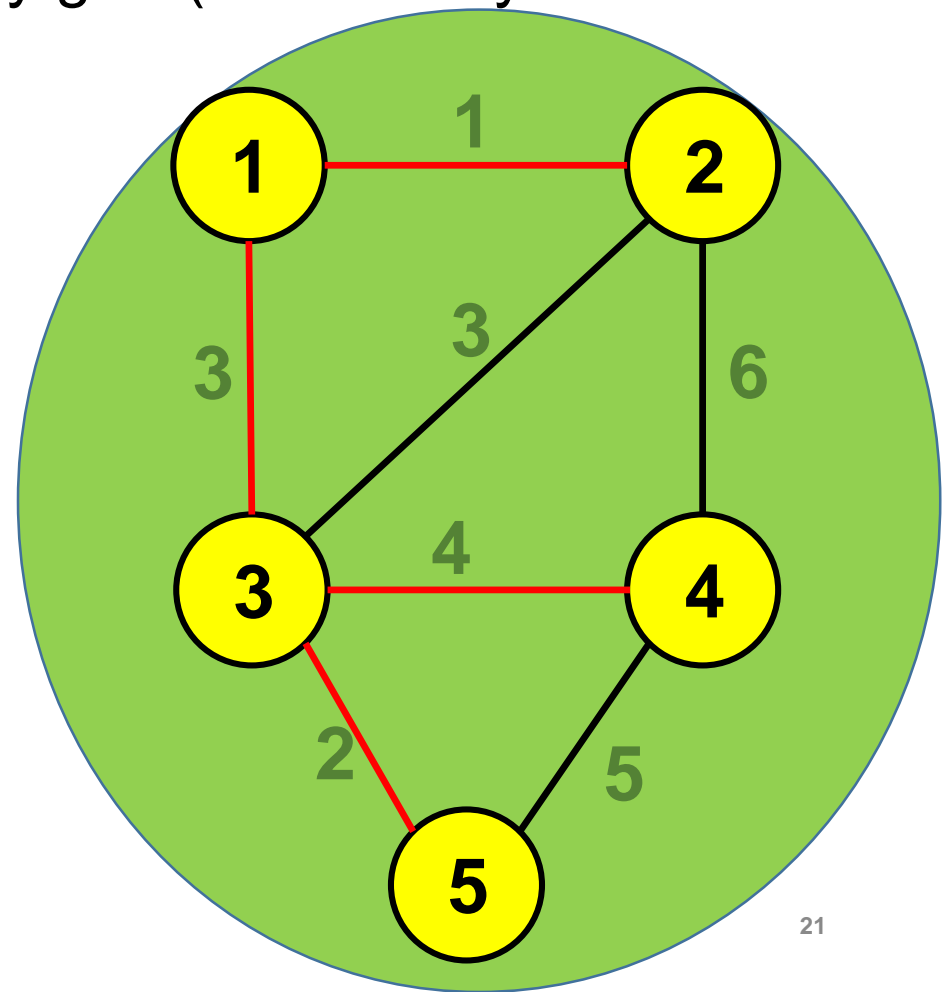
MST – graf

Wybieramy najbliższy wierzchołek dla 1,2,3 o najmniejszej wadze tj. 5 i dodajemy go do zbioru rozwiązań (zbiór rozwiązań zawiera teraz wierzchołki 1,2,3,5), a krawędź 3-5 do zbioru krawędzi MST



MST – graf

Wybieramy najbliższy wierzchołek dla 1,2,3,5 o najmniejszej wadze tj. 4 i dodajemy go do zbioru rozwiązań, a krawędź 3-4 do zbioru krawędzi MST. Ponieważ wszystkie wierzchołki należą do zbioru rozwiązań zatem uzyskany graf (z czerwonymi krawędziami, reprezentującymi zbiór krawędzi MST jest rozwiązaniem).



MST – algorytm Prima

Algorytm Prima wg Cormena

$\text{Adj}[u]$ – lista sąsiadów wierzchołka u

$\text{key}[u]$ – waga najmniejszej krawędzi dla u

$p[u]$ – wierzchołek z którym jest połączony u

Q – kolejka priorytetowa posortowana wg key

r – wierzchołek startowy

Prim(G, w, r)

For each $v \in V[G]$ do $\text{key}(v) = \infty$

$\text{key}[r] = 0$; $p[r] = \text{NIL}$

$Q \leftarrow V[G]$

while $Q \neq \emptyset$ do

$u = \text{Extract-Min}(Q)$

 for each $v \in \text{Adj}[u]$ do

 if $v \in Q$ AND $w(u, v) < \text{key}[v]$ then

$\text{key}[v] = w(u, v)$; $p[v] = u$;

Złożoność obliczeniowa (E to liczba krawędzi a V to liczba wierzchołków) przy użyciu kopca binarnego: $O(E \log(V))$

MST – algorytm Kruskala (1)

Algorytm Kruskala

A – zbiór krawędzi tworzących rozwiązanie

sE – posortowany zbiór krawędzi grafu

Kruskal (G, w)

A $\leftarrow \emptyset$

For each $v \in V[G]$ do

 MakeSet(v) //tworzy poddrzewo wierzchołka

posortuj krawędzie niemalejąco względem wag sE

for $i:=1$ to $i=|sE|$ do

$(u, v) = sE[i];$ //pobierz kolejną krawędź

 //sprawdź czy u i v należy do różnych poddrzew

 if FindSet(u) \neq FindSet(v) do

 A $\leftarrow A \cup (\{u, v\})$ //dodaj krawędź do rozw.

 Union(u, v) //połącz poddrzewa w jedno

return A

MST – algorytm Kruskala (2)

MakeSet(v) – tworzy zbiór składający się tylko z jednego elementu v

FindSet(v) - zwraca identyfikator zbioru do którego należy v (jeżeli $\text{FindSet}(u) = \text{FindSet}(v)$ to oznacza, że wierzchołki u i v należą do tego samego zbioru)

Union(u,v) – łączy dwa zbiory, z których jeden zawiera wierzchołek u , a drugi wierzchołek v

Sortowanie działa w czasie $O(E \log E)$, efektywna implementacja zbiorów rozłącznych ma złożoność $O(E \alpha(E, V))$, gdzie α jest odwrotnością funkcji Ackermana. Zatem złożoność całkowita wynosi $O(E \log E)$

MST – algorytm Kruskala (3)

Lista posortowanych krawędzi:

(1,2):1

(3,5):2

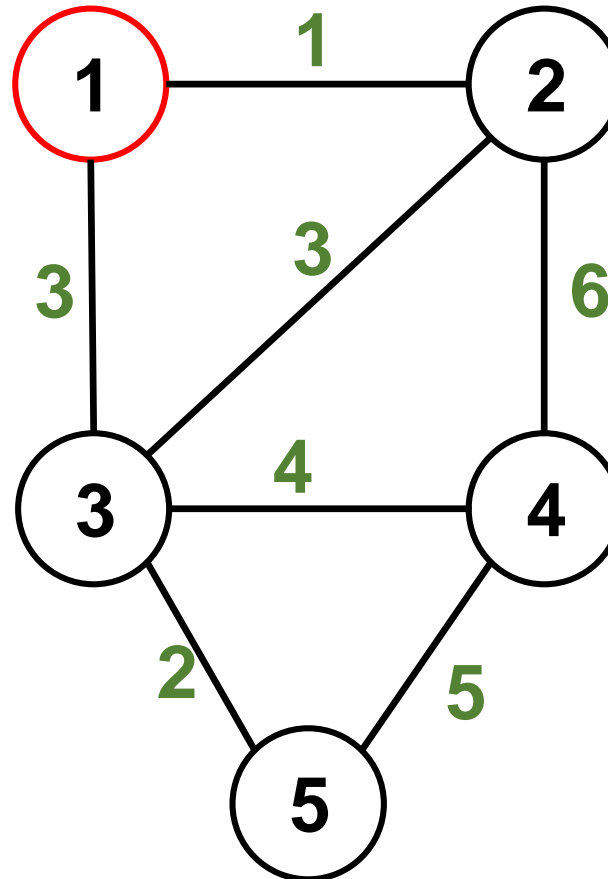
(1,3):3

(2,3):3

(3,4):4

(4,5):5

(2,4):6

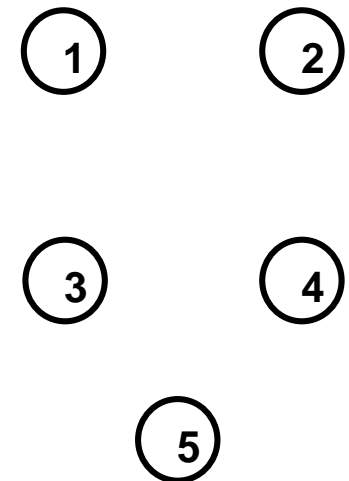


MST – algorytm Kruskala (4)

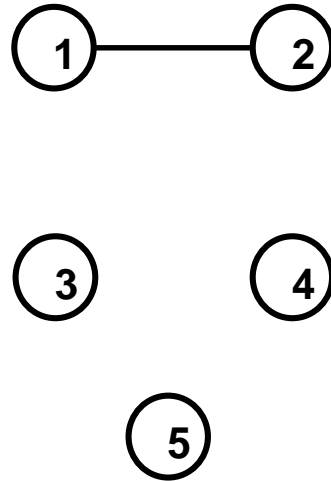
**Lista
posortowanych
krawędzi:**

(1,2):1
(3,5):2
(1,3):3
(2,3):3
(3,4):4
(4,5):5
(2,4):6

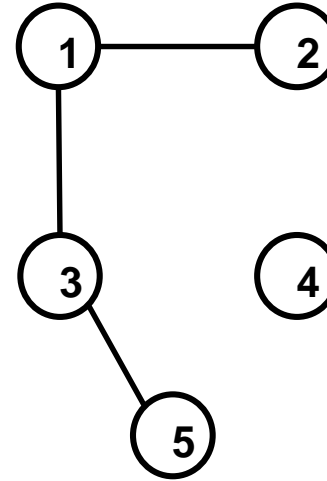
1. Tworzymy las



2. (1,2) -dodajemy



4. (1,3)-dodajemy

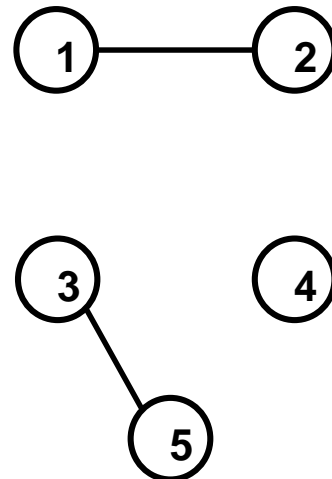


7. (4,5) –nie dod.

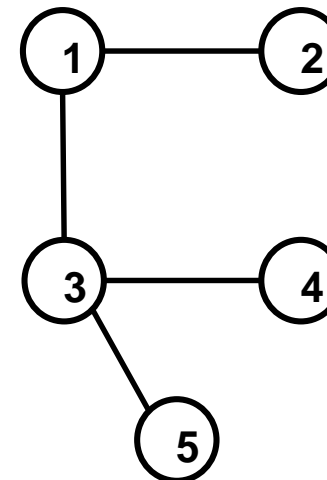
8. (2,4) –nie dod.

5. (2,3) –nie dod.

3. (3,5)-dodajemy



6. (3,4) -dodajemy



 **MST**

MST – reprezentacja zbiorów rozłącznych

Wersja naiwna

MakeSet()

```
for i:= 1 to n do  
    grupa[x] = x
```

FindSet(x)

```
return grupa[x]
```

Union(x,y)

```
for i:= 1 to n do  
    if grupa[i] = grupa[y] then  
        grupa[y] = grupa[x]
```

Złożoność Union – $O(n)$

MST – reprezentacja zbiorów rozłącznych

parent[x] – rodzic x

(np. **parent[8]=14**, **parent[14]=9**,

parent[9]=9 <-reprezentant zbioru tzn. **parent[x]=x**)

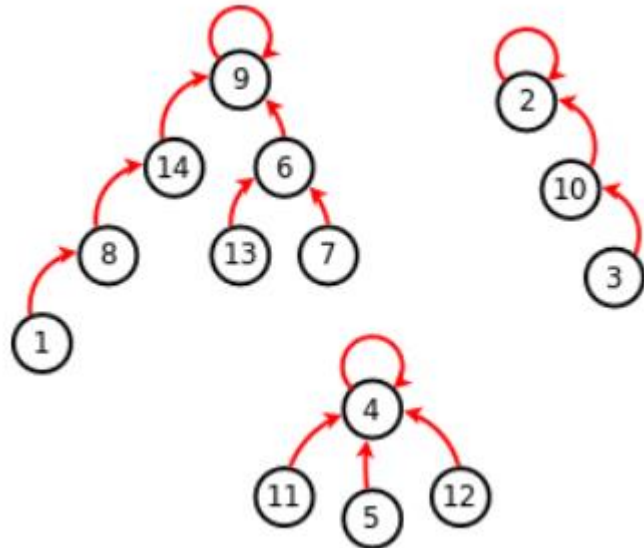
FindSet(x)

a = x

while a <> parent[a]

a = parent[a]

return a



Rekursja:

FindSet(x)

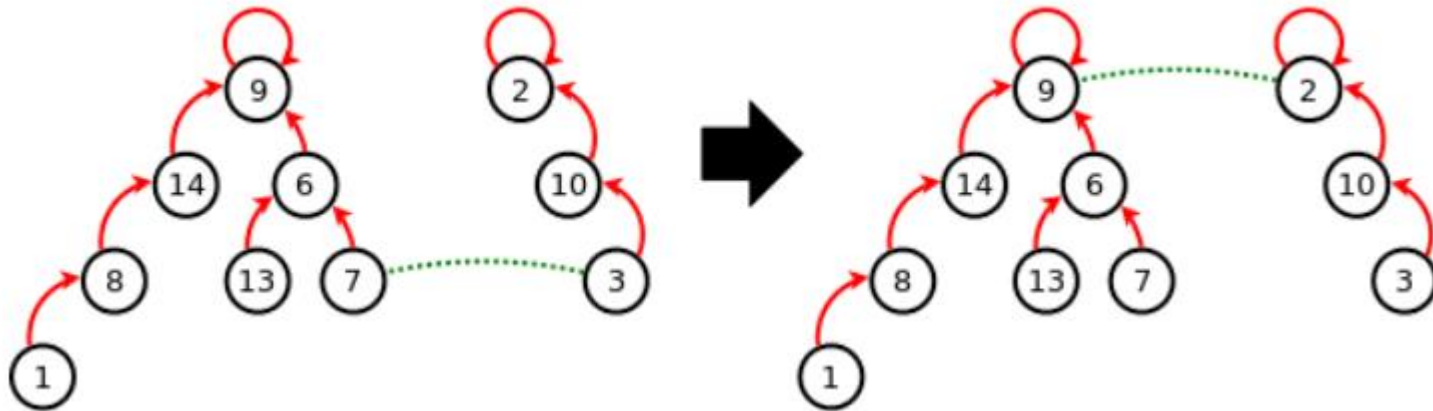
if parent[x] = x

return x

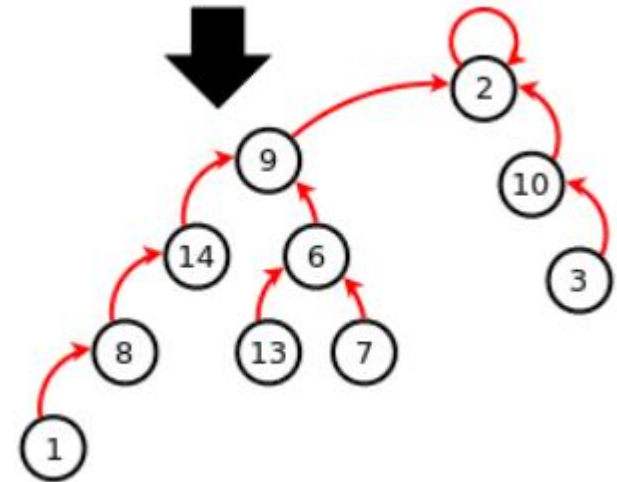
else

return FindSet(parent[x])

MST – reprezentacja zbiorów rozłącznych

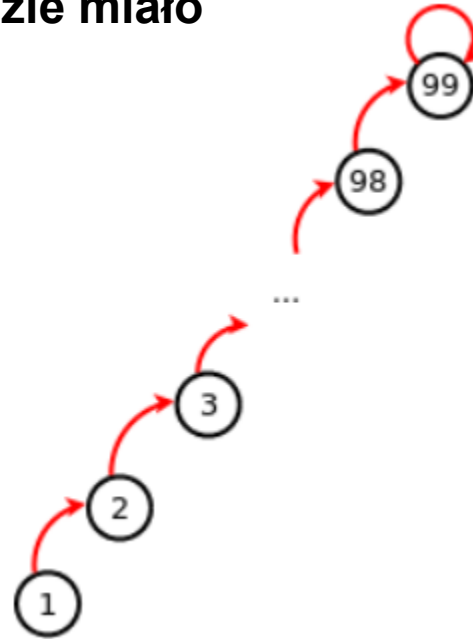


```
Union(x,y)  
  a = FindSet(x)  
  b = FindSet(y)  
  parent[a] = b
```



MST – reprezentacja zbiorów rozłącznych

Wersja nieoptymistyczna – *Find* będzie miało złożoność $O(n)$

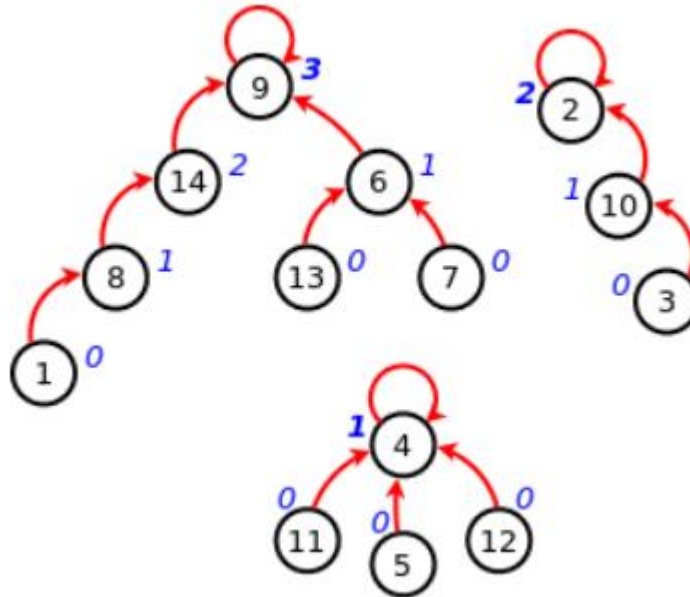


Rozwiązanie – łączenie wg rangi i kompresja ścieżek

MST – reprezentacja zbiorów rozłącznych

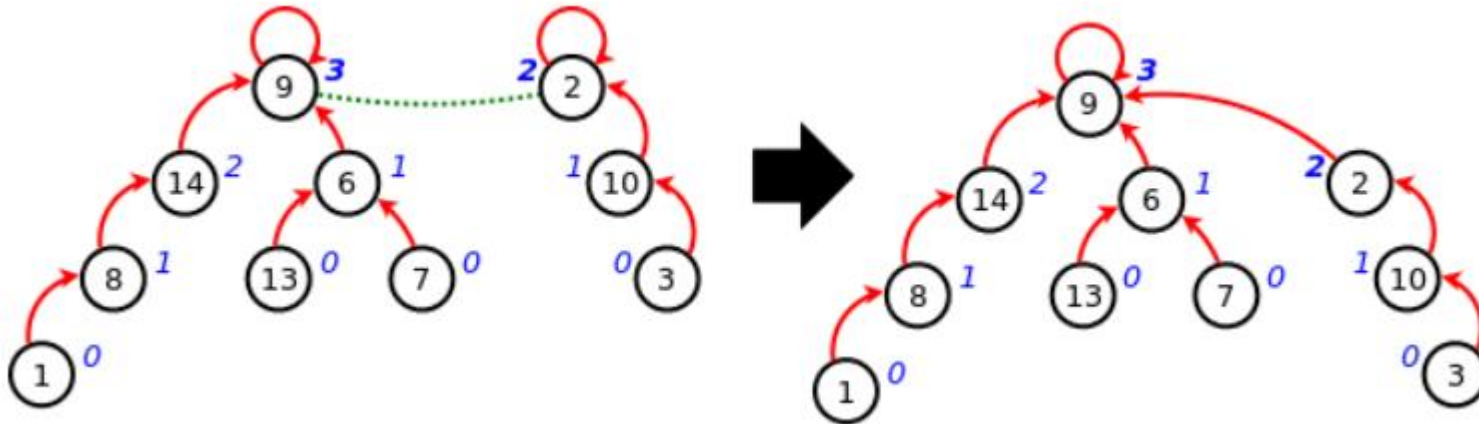
Ranga elementu x – oznacza wysokość drzewa podpiętego do x (oznaczymy ją jako $\text{rank}[x]$). Ranga w przypadku reprezentanta mówi ile operacji przechodzenia należy wykonać w najgorszym przypadku dla operacji `FindSet()`

W praktyce istotna jest tylko ranga reprezentantów

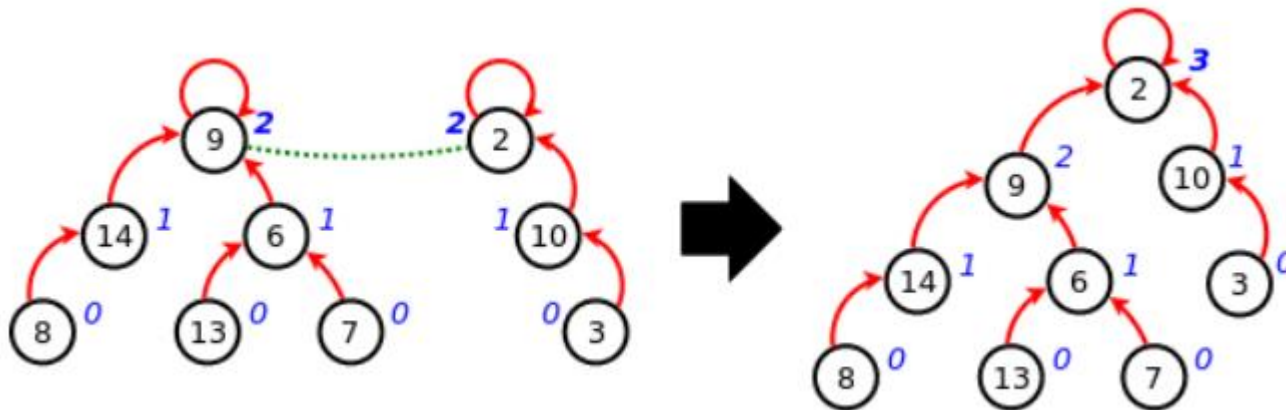


MST – reprezentacja zbiorów rozłącznych

Łączenie – zawsze o mniejszej randze do drzewa o większej randze



Drzewa o różnych rangach – ranga drzewa wynikowego nie zmienia się



Drzewa o równych rangach – ranga drzewa wynikowego większa o 1

MST – reprezentacja zbiorów rozłącznych

```
Union(x,y)
  a = FindSet(x)
  b = FindSet(y)
  if rank[a] < rank[b]
    parent[a] = b
  else
    parent[b] = a;

  if rank[a]==rank[b]
    rank[a] = rank[a]+1
```

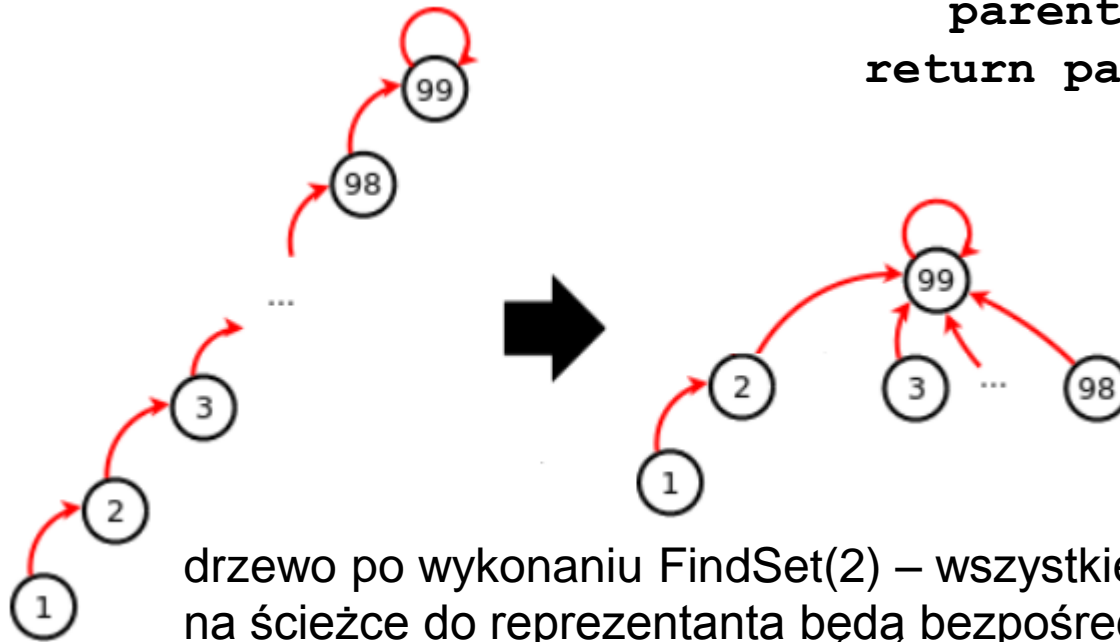
Złożoność FindSet – $O(\log n)$

Złożoność Union – taka sama jak FindSet

MST – reprezentacja zbiorów rozłącznych

Kompresja ścieżek – jeżeli podczas operacji FindSet odkryliśmy, że reprezentantem x jest y to możemy od razu wpisać $\text{parent}[x]=y$, aby skrócić następne poszukiwania.

```
FindSet(x)
  if parent[x] <> x
    parent[x] = FindSet(parent[x])
  return parent[x]
```



drzewo po wykonaniu FindSet(2) – wszystkie elementy znajdujące się na ścieżce do reprezentanta będą bezpośrednio po operacji wskazywać na reprezentanta.

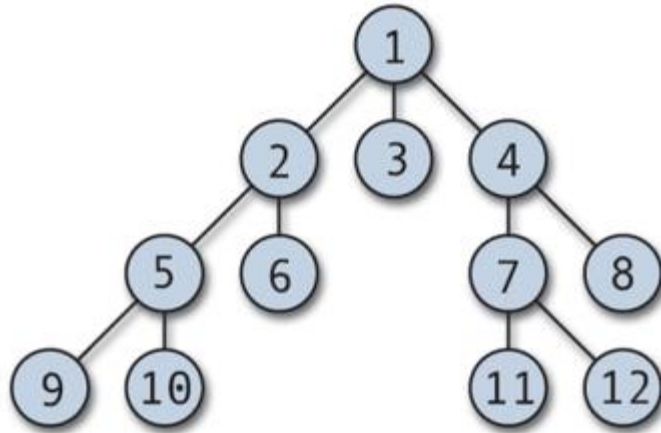
Kompresja ścieżek odrobinę "psuje" pojęcie rangi — nie oznacza ona już wysokości drzewa. Ranga pozostaje jednak większa lub równa wysokości, co wystarczy do naszych celów.

MST – przeszukiwanie grafu wszerz (BFS)

BFS(V, s)

```
1  for each  $u \in V - \{s\}$  do
2       $color[u] := WHITE$ 
3       $d[u] := \infty$     //-czas odwiedzenia
4       $p[u] := NIL$     //-poprzednik (jak w alg. Disjkstry)
5   $color[s] := GREY$ 
6   $d[s] := 0$ ;
7   $p[s] = NIL$ 
8   $Q := s$     //-Q-zwykła kolejka FIFO - wstaw s do kolejki
9  while  $Q \neq \emptyset$  do
10      $u := head[Q]$ 
11      $Q := Q - u$     //-usuń z kolejki u
12     for each  $v \in Adjv[u]$ 
13         if  $color[v] = WHITE$  then
14              $color[v] := GREY$ 
15              $d[v] := d[u] + 1$ 
16              $p[v] := u$ 
17              $Q := Q + v$     //-wstaw do kolejki wierzchołek v
18      $color[u] = BLACK$ 
19     wykonaj operację w u
```

MST – przeszukiwanie grafu wszerz (BFS)



Kolejność odwiedzanych wierzchołków w drzewie (preferujemy odwiedzanie od lewego do prawego potomka) metodą BFS

MST – przeszukiwanie grafu w głąb (DFS)

DFS(G)

```
1 for each  $u \in V$  do
2   color[u] := WHITE
3   d[u] :=  $\infty$     //-czas odwiedzenia
4   p[u] := NIL    //-poprzednik (jak w alg. Dijkstry)
5 time := 0
6 for each  $z \in V$  do
7   if color[z] = WHITE then
8     DFS-VISIT(z)
```

DFS-VISIT(u)

```
1 color[u] := GRAY
2 d[u] := time := time + 1
3 for each  $v \in \text{Adj}[u]$  do
4   if color[v] = WHITE do then
5     p[v] := u
6     DFS-VISIT(v)
7 color[u] := BLACK
8 wykonaj jakąś operację w u (np. wyświetl u)
9 f[u] = time := time + 1    //-czas przetworzenia
```

MST – przeszukiwanie grafu w głąb (DFS)

Wersja z dwoma kolorami – w tym przypadku przetwarzanie wierzchołka jest w momencie odwiedzenia – stąd kolejność przetwarzania będzie odwrotna

DFS(G)

```
1 for each  $u \in V$  do
2   color[u] := WHITE
3 for each  $z \in V$  do
4   if color[z] = WHITE then
5     DFS-VISIT(z)
```

DFS-VISIT(u)

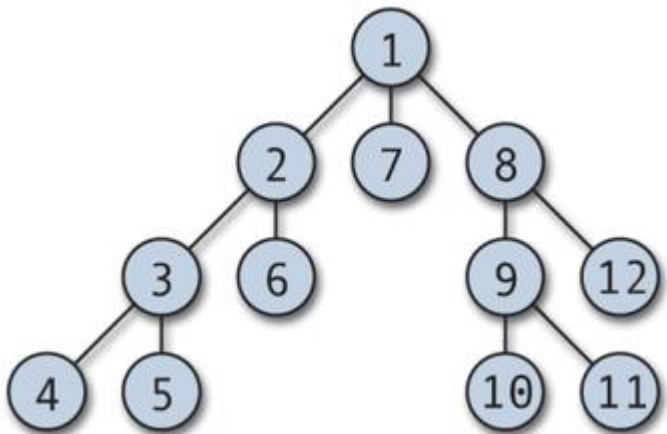
```
1 color[u] := BLACK
2 wykonaj operację w u (np. wyświetl symbol u)
3 for each  $v \in \text{Adj}[u]$  do
4   if color[v] = WHITE then
5     DFS-VISIT(v)
```

MST – przeszukiwanie grafu w głąb (DFS)

Wersja z dwoma kolorami, wierzchołkiem początkowym i stosem

DFS-iterative(G, u)

```
1 for each  $u \in V$  do
2   color[u] := WHITE
3 Q.push(u) //Q - stos (pusty na początku)
4 while  $Q \neq \emptyset$  do
5    $u = Q.pop()$ 
6   if color[u] = WHITE then
7     color[u] := BLACK
8     wykonaj operację dla u
9     for each  $v \in \text{Adj}[u]$  do
10      Q.push(v)
```



Kolejność odwiedzanych wierzchołków w drzewie metodą DFS (przedstawionej na tej stronie)
- preferujemy odwiedzanie od lewego do prawego potomka – czyli do stosu wstawiamy potomków od prawego do lewego