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->p[v]->p[p[v]]-> p[p[p[v]]]->…. aż do w. startowego

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

Najkrótsza droga w grafie

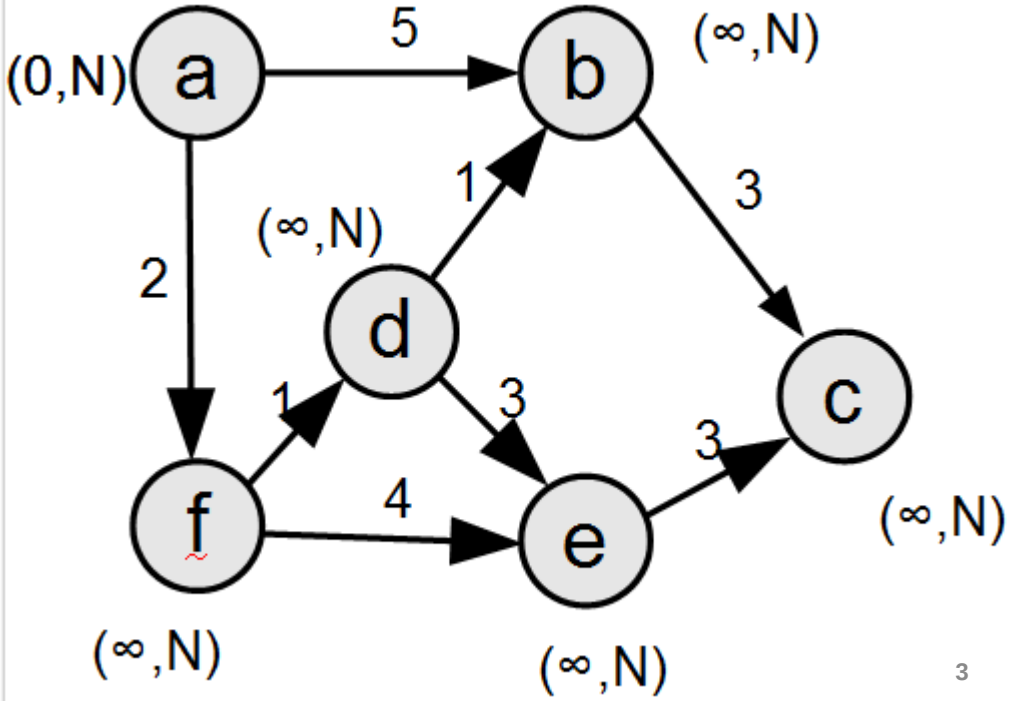
```
RELAKSACJA KRAWĘDZI
Jeżeli dochodząc do danego wierzchołka v z wierzchołka u
poprawiamy dotychczasową drogę (p[v])

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]**)

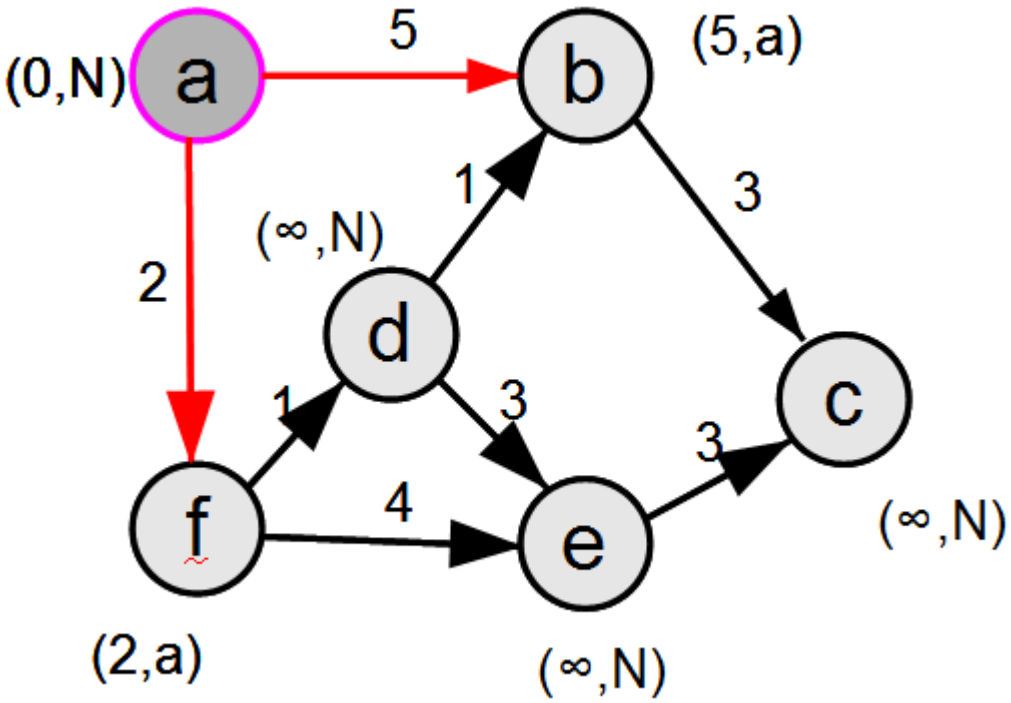
Dwa zbiory wierzchołków : przebadane i nieprzebadane (na początku wszystkie są nieprzebadane)

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



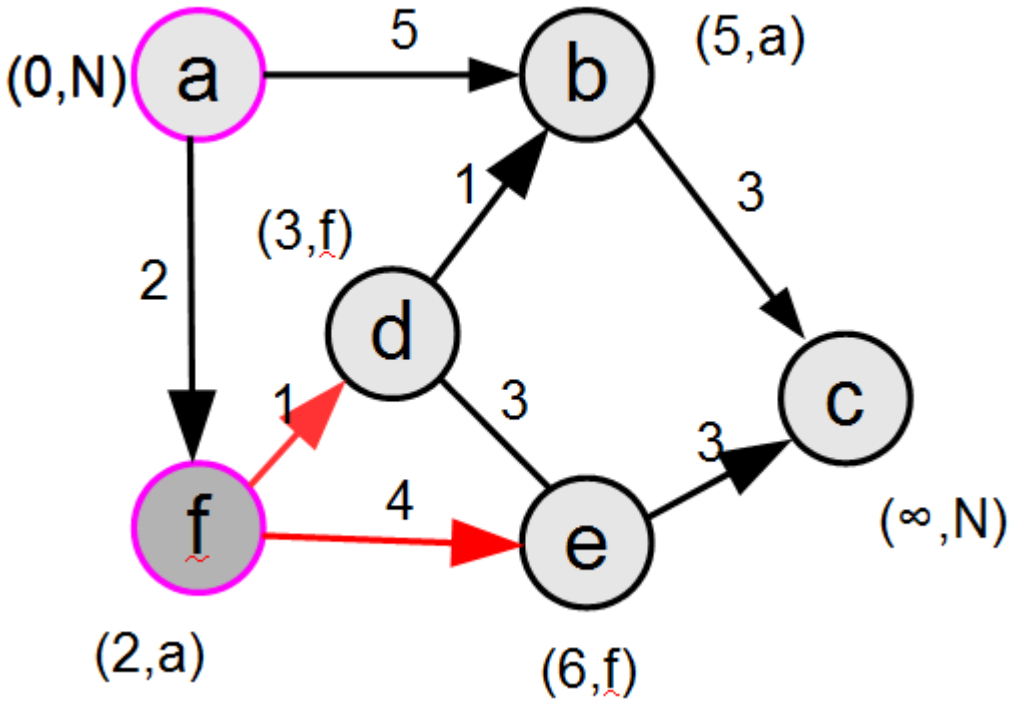
Najkrótsza droga w grafie – algorytm Dijkstry

- 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



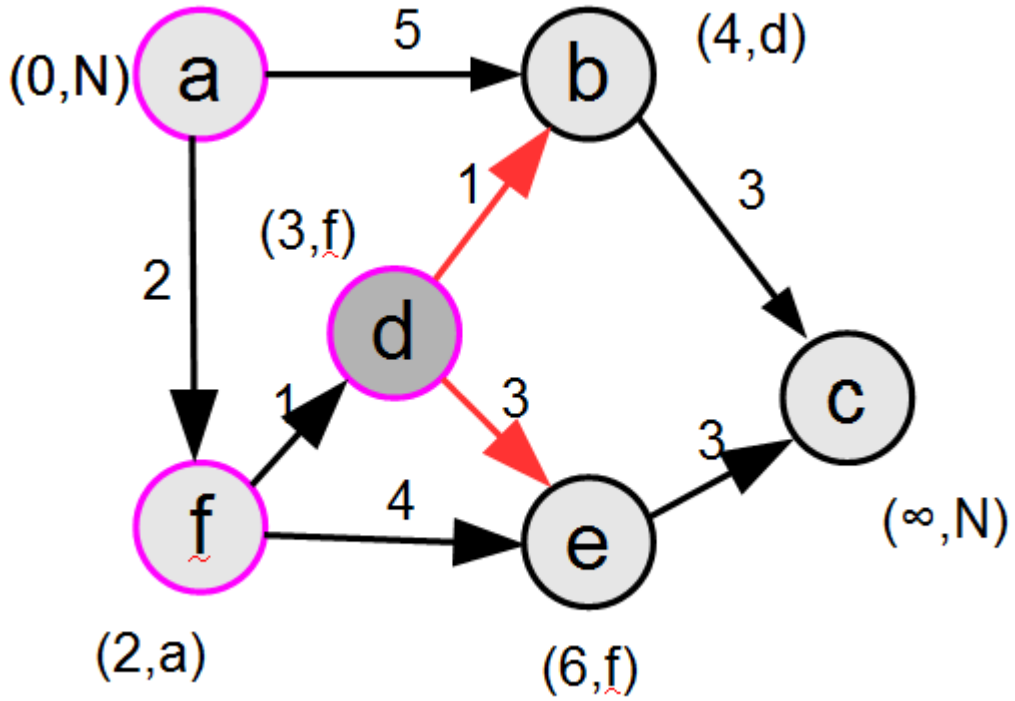
Najkrótsza droga w grafie – algorytm Dijkstry

- 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



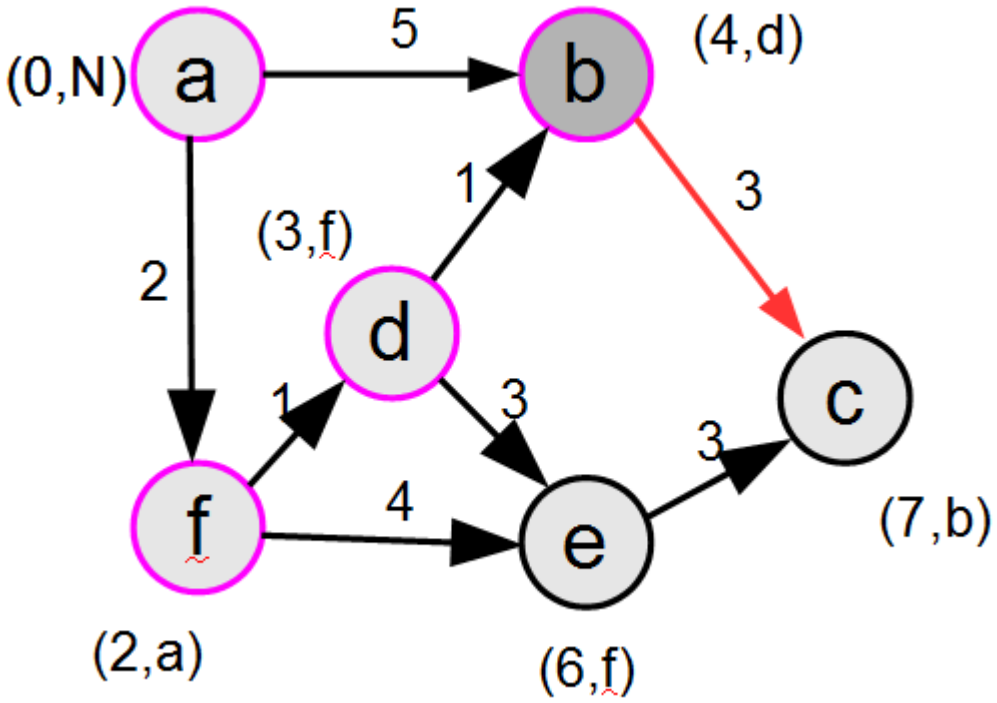
Najkrótsza droga w grafie – algorytm Dijkstry

- 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



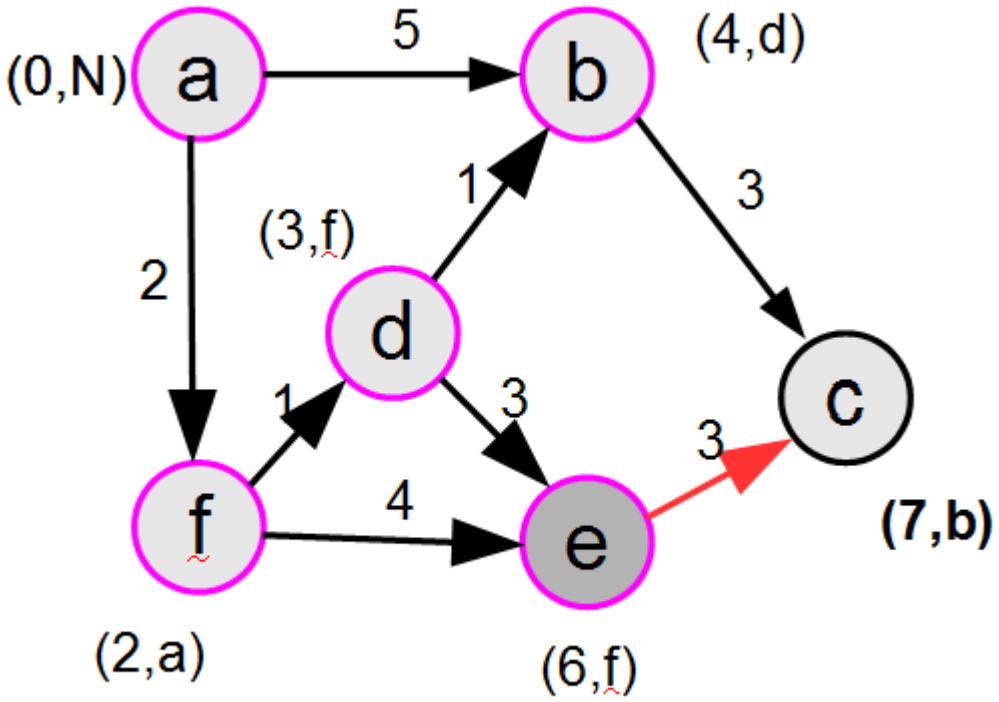
Najkrótsza droga w grafie – algorytm Dijkstry

- 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



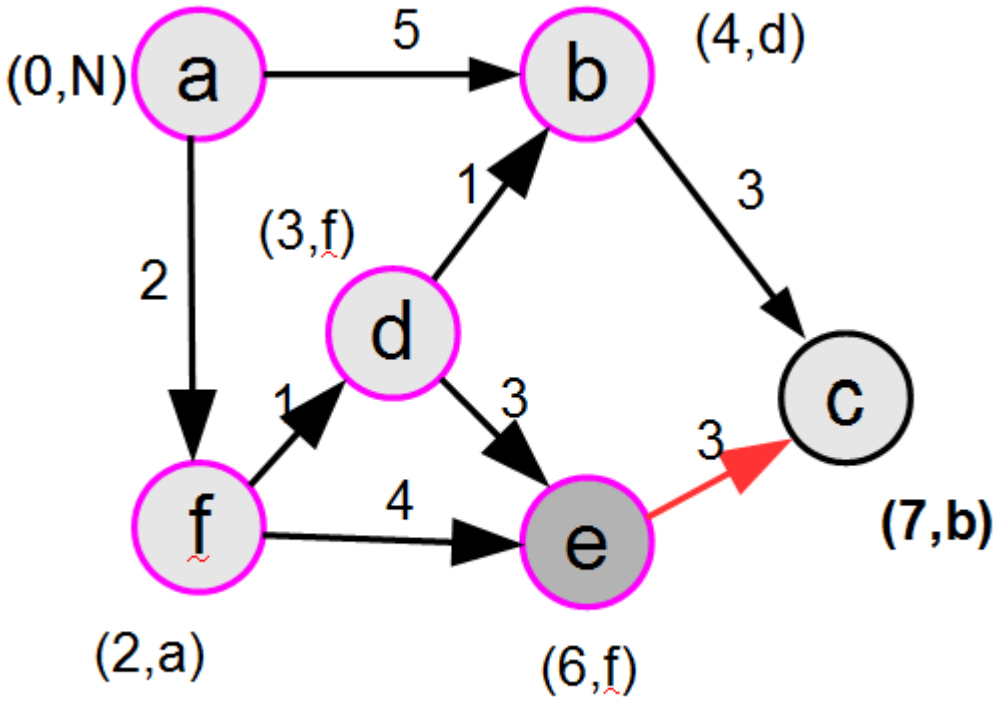
Najkrótsza droga w grafie – algorytm Dijkstry

- 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

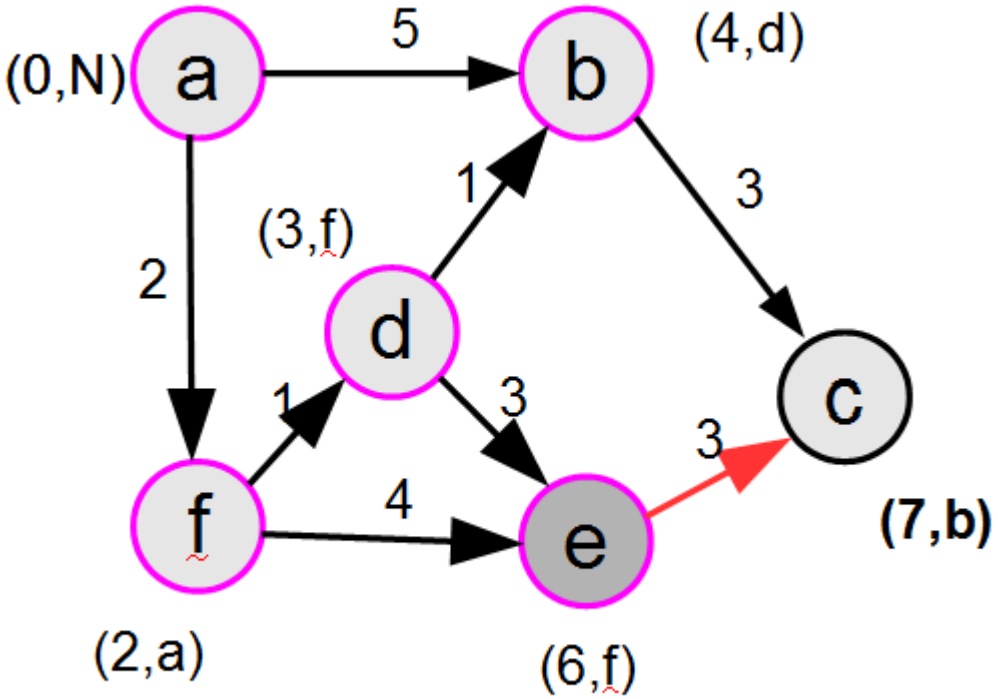
- Ze zbioru nieprzebadanych wierzchołków(c) wybieramy wierzchołek o najmniejszej wartości odległości (wierzchołek **c**)
- Dokonujemy relaksacji jego sąsiadów – brak sąsiadów
- Wierzchołek **c** usuwamy ze zbioru nieprzebadanych



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
Adj[v] – zbiór sąsiadów wierzchołka v
w(u,v) – waga,  Q – kolejka (zbiór wierzchołków)

Dijkstra(G, w, s)
      for each  v∈V  do
            d[v]=∞
            p[v] =NULL
      d[s]=0;
      Q = V[G]
      while Q ≠ ∅ do
            //usuń wierzchołek o najmniejszej d[i]
            u := Extract-Min(Q)
            for each  v ∈ Adj[u]  do
                  if d[v] > d[u] + w(u, v) then
                        //relaksacja krawędzi
                        d[v] =  d[u] + w(u, v)
                        p[v] = u;
```

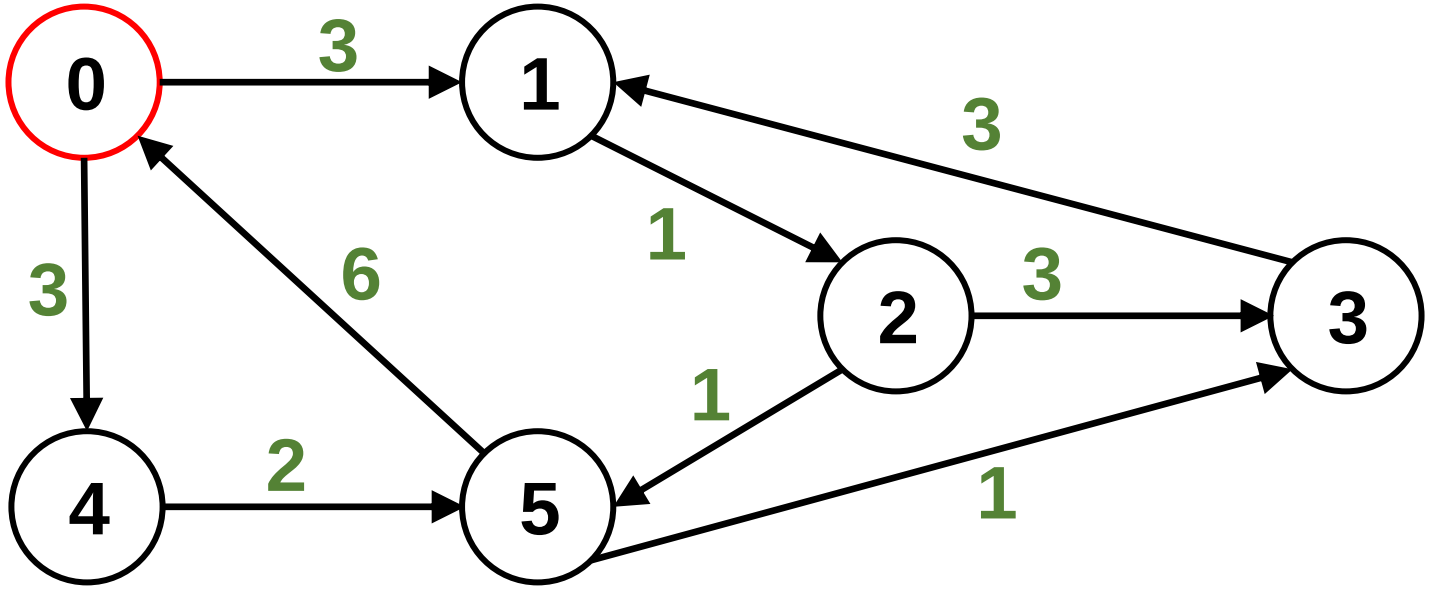
Najkrótsza droga w grafie – algorytm Dijkstry

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(ElogV)**
- po zastąpieniu zwykłego kopca kopcem Fibonacciego złożoność zmniejsza się do **O(E+VlogV)**

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*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

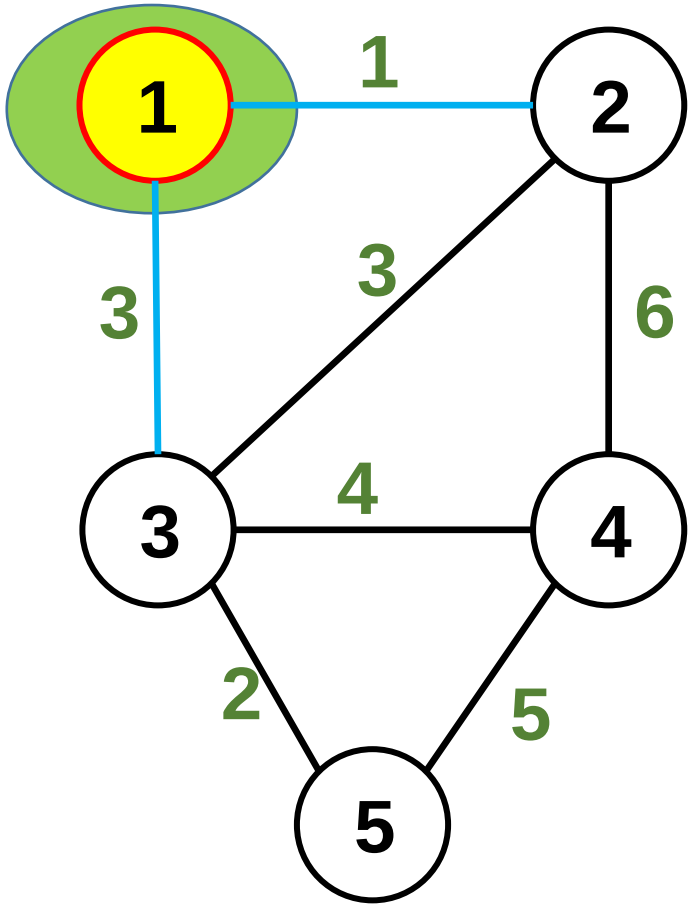
Algorytm Prima

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 o najmniejszej wadze i dodaj wierzchołek do zbioru rozwiązań.
4. Jeżeli zbiór rozwiązań zawiera wszystkie wierzchołki to koniec, w przeciwnym wypadku przejdź do pkt. 2.

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

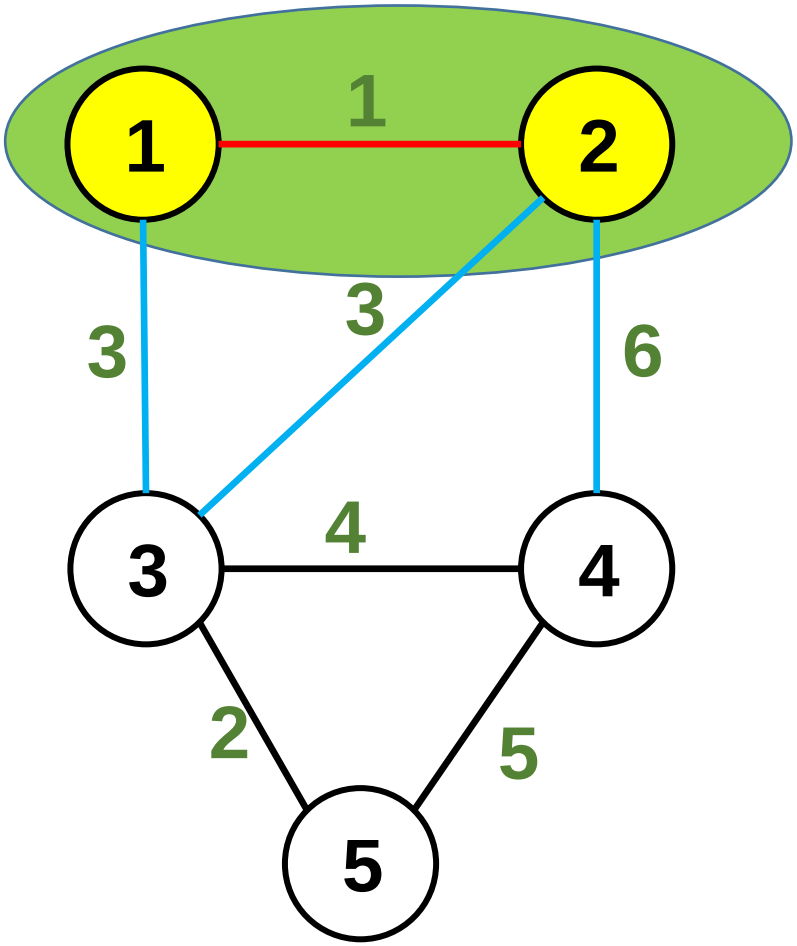
MST – graf

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



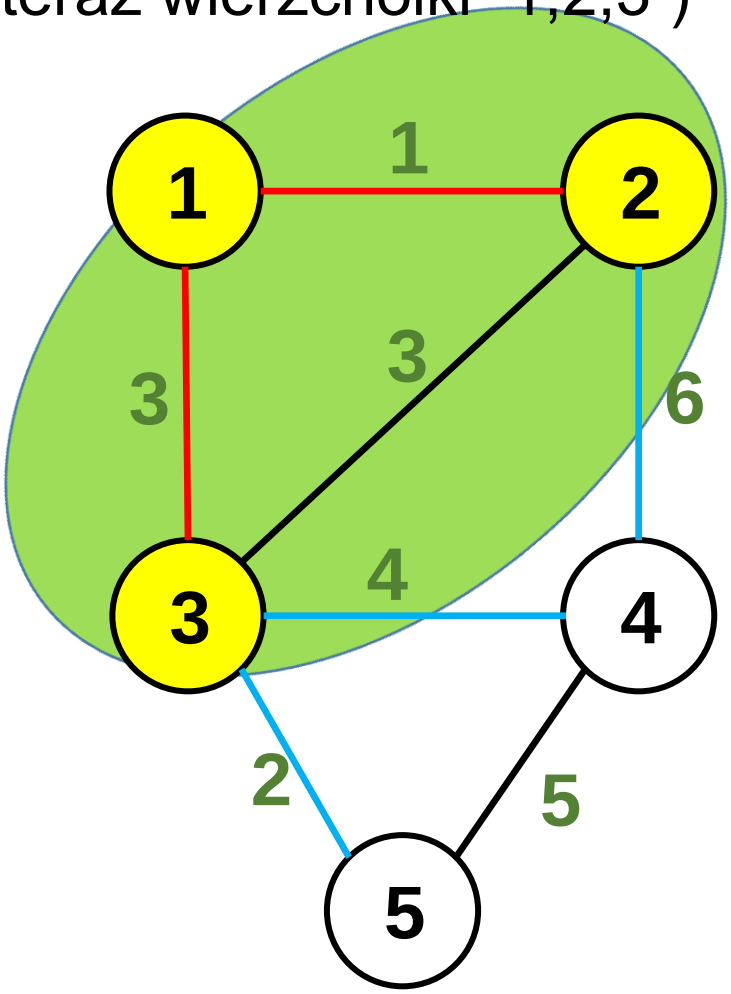
MST – graf

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



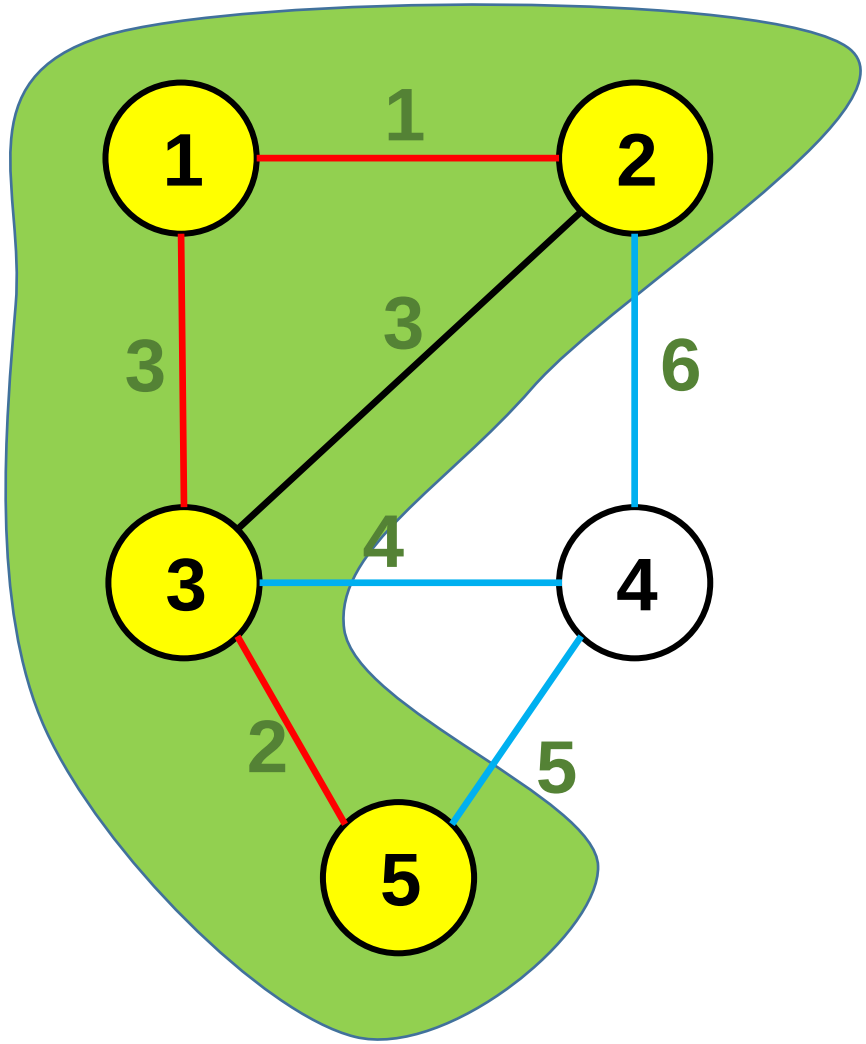
MST – graf

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)



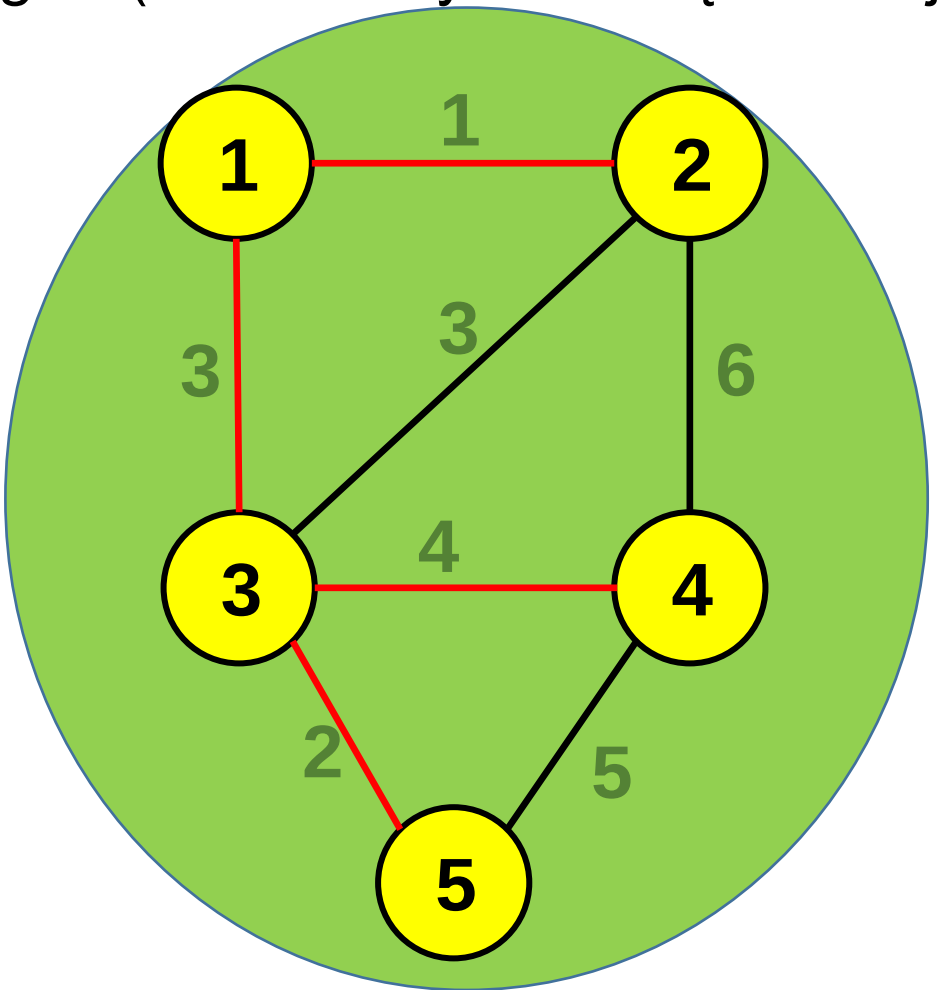
MST – graf

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)



MST – graf

Wybieramy najbliższy wierzchołek dla 1,2,3,5 o najmniejszej wadze tj. 4 (zbiór rozwiązań zawiera teraz wierzchołki 1,2,3,4,5). Ponieważ wszystkie wierzchołki należą do zbioru rozwiązań zatem uzyskany graf (z czerwonymi krawędziami jest rozwiązaniem).



MST – algorytm Prima

Algorytm Prima wg Cormena

```
Adj[u] – lista sąsiadów wierzchołka u
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 ∈ V[G] do  key(v)= ∞
key[r] = 0; p[r] = NIL
Q ← V[G]
while Q ≠ ∅ do
  u = Extract-Min(Q)
  for each v ∈ Adj[u] do
     if v ∈ Q AND w(u,v) < key[v] then
         key[v]= w(u,v); p[v]=u;
```

MST – algorytm Kruskala (1)

Algorytm Kruskala
A – zbiór krawędzi tworzących rozwiązanie
sE – posortowany zbiór krawędzi grafu

```
Kruskal(G,w)
A ← ∅
For each v ∈ 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) ≠ FindSet(v) do
      A ← A ∪ ({u, w)} //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 FindSet(u)=FindSet(v) to oznacza, że wierchołki u i v należą do tego samego zbioru

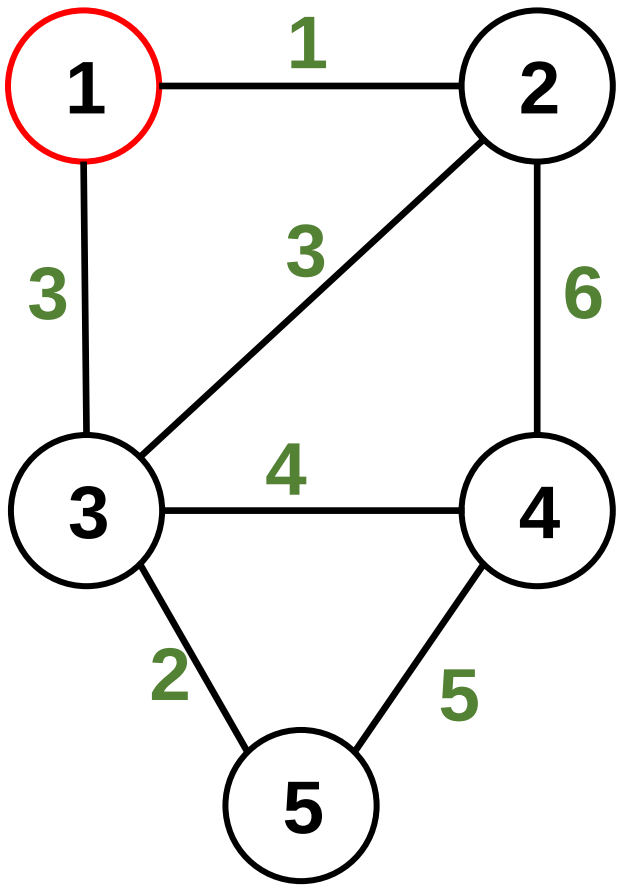
Union(u,v) – łączy dwa zbiory, z których jeden zawiera wierzchołek u, a drugi wierzchołek v

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

MST – algorytm Kruskala (3)

Lista posortowanych krawędzi:

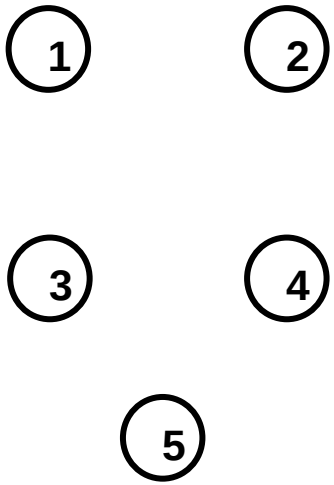
(1,2):1
(3,5):2
(1,3):3
(2,3):3
(3,4):4
(4,5):5
(2,4):6



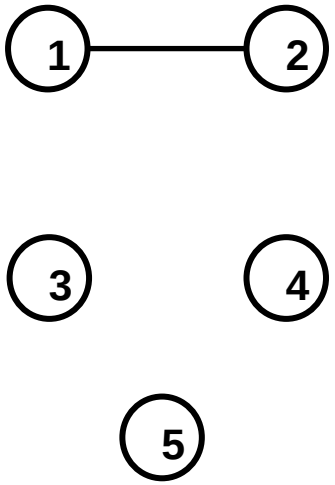
MST – algorytm Kruskala (4)

Lista posortowanych krawędzi:
(1,2):1
(3,5):2
(1,3):3
(2,3):3
(3,4):4
(4,5):5
(2,4):6

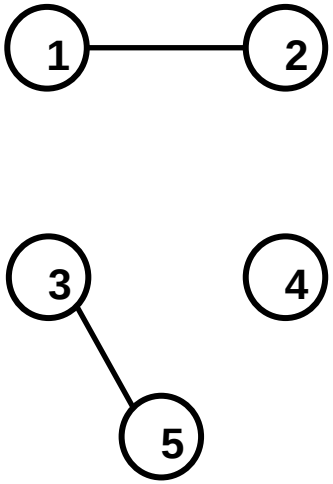
1. Tworzymy las



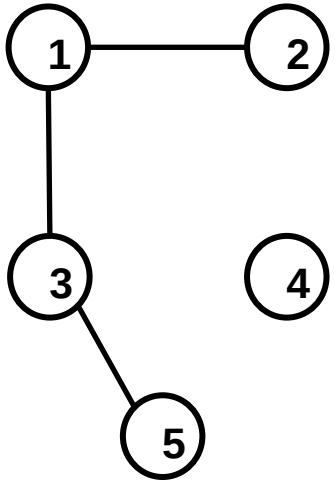
2. (1,2) -dodajemy



3. (3,5)-dodajemy

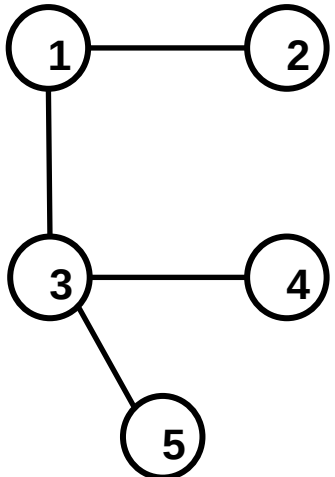


4. (1,3)-dodajemy



5. (2,3) –nie dod.

6. (3,4) -dodajemy



7. (4,5) –nie dod.

8. (2,4) –nie dod.

MST

MST – reprezentacja zbiorów rozłącznych

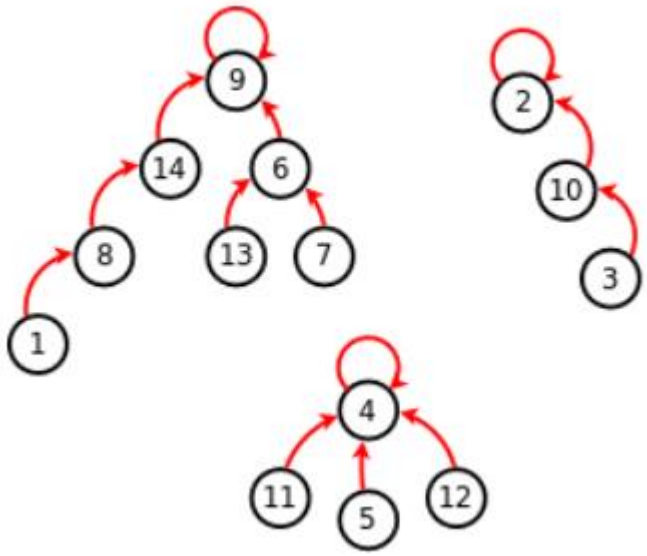
Wersja naiwna

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

Find(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

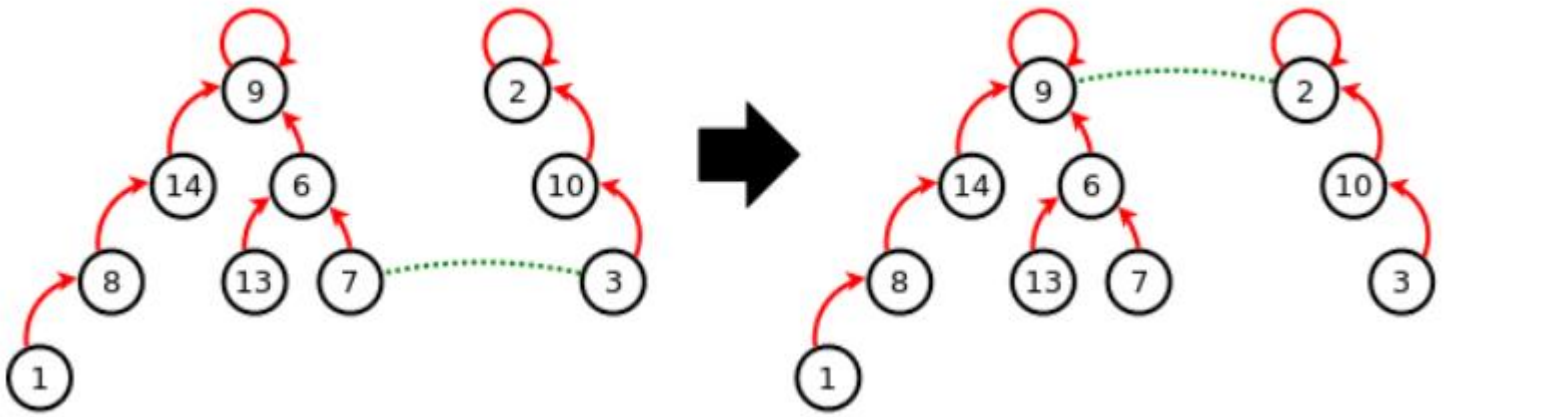



```
Find(x)
  a = x
  while a<>parent[a]
     a = parent[a]
   return a
```

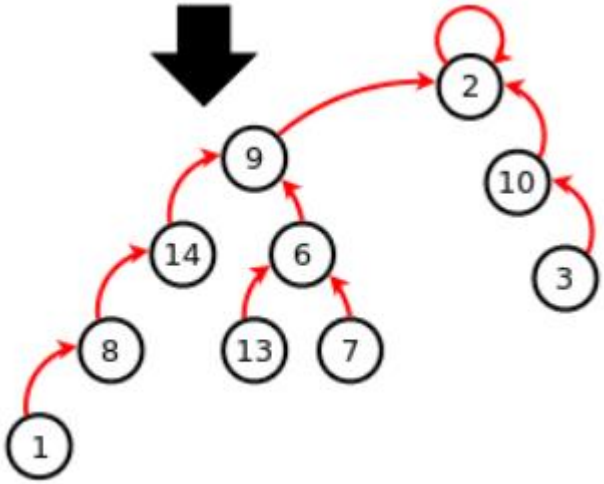
Rekursja:

```
Find(x)
  if parent[x] = x
    return x
  else
    return Find(parent[x])
```

MST – reprezentacja zbiorów rozłącznych

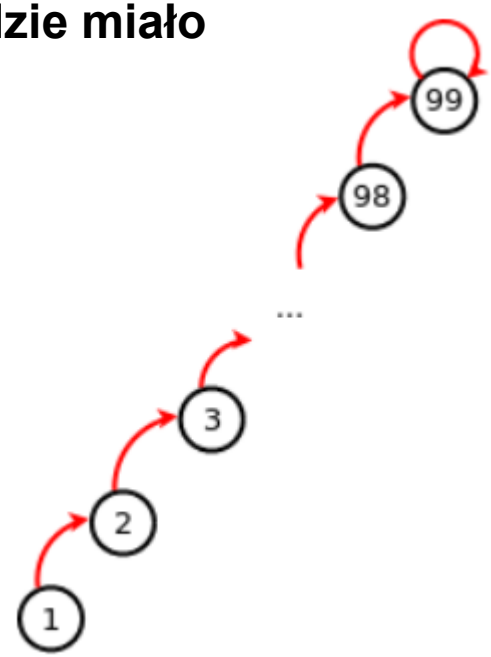



```
Union(x,y)
    a = Find(x)
    b = Find(y)
    parent[a] = b
```



MST – reprezentacja zbiorów rozłącznych

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

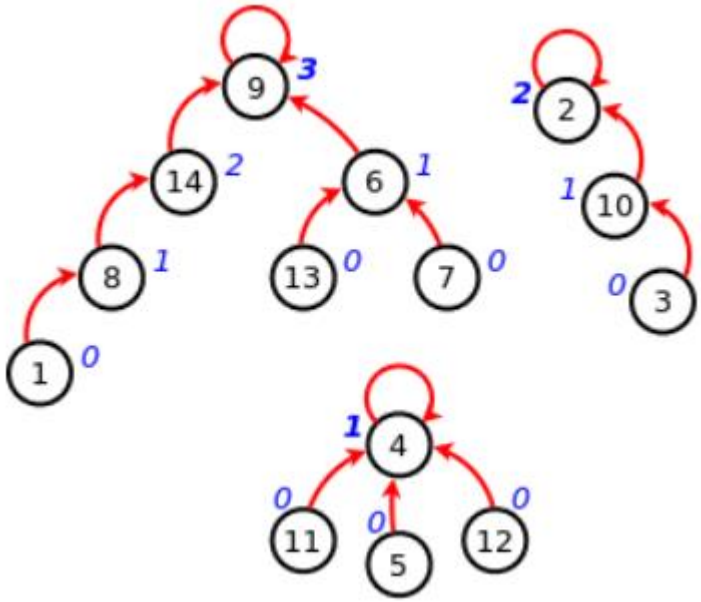


Rozwiązanie – łączenie wg rangi i kompresja ścieżek

MST – reprezentacja zbiorów rozłącznych

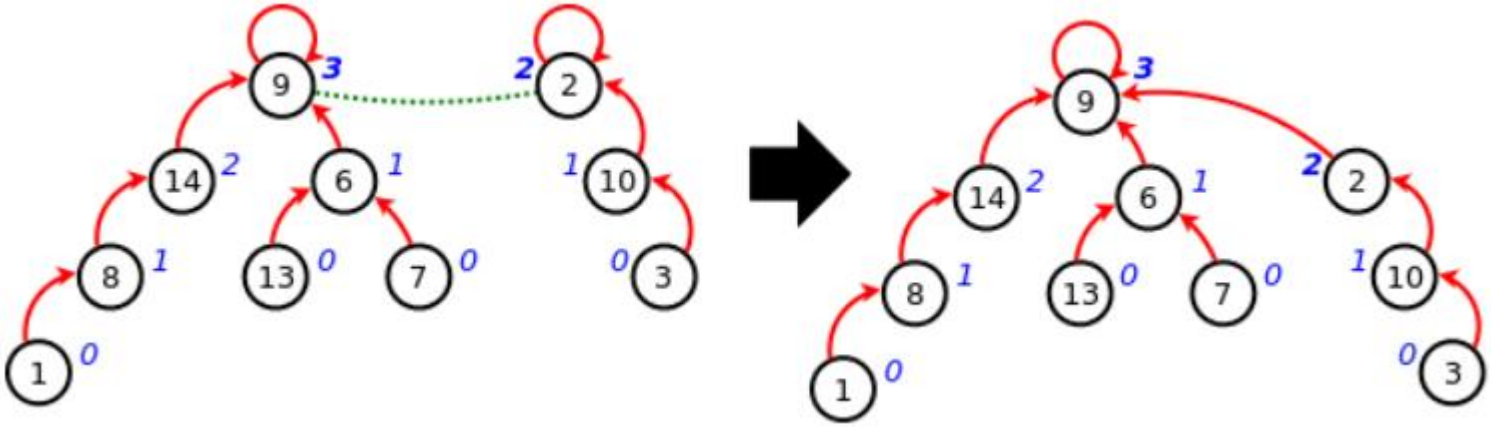
Ranga elementu x – oznacza wysokość drzewa podpiętego do x (oznaczmy ja jako rank[x]). Ranga w przypadku reprezentanta mówi ile operacji przechodzenia należy wykonać w najgorszym przypadku dla operacji Find()

W praktyce istotna jest tylko ranga reprezentantów

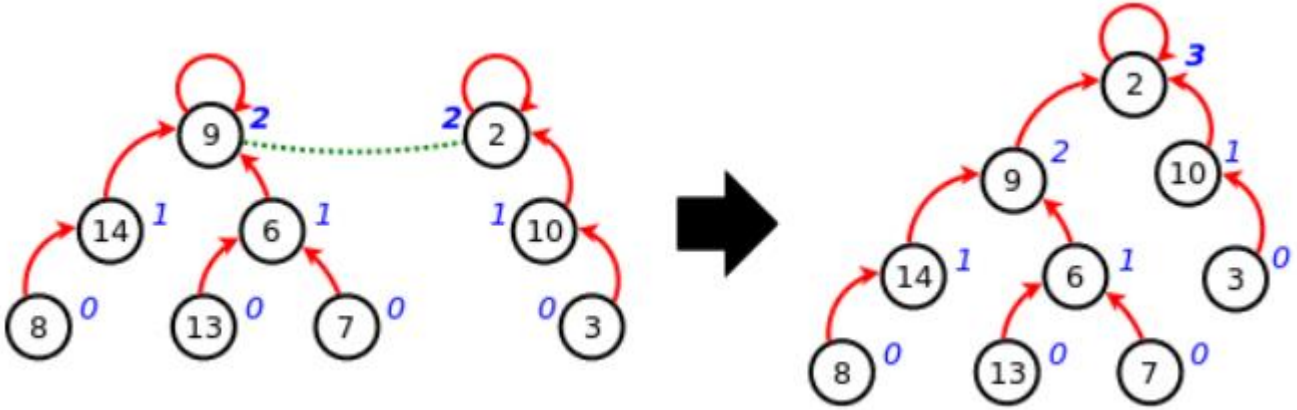


MST – reprezentacja zbiorów rozłącznych

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 = Find(x)
    b = Find(y)
    if rank[a] < rank[b]
       parent[a] = b
    else
       parent[b] = a;

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

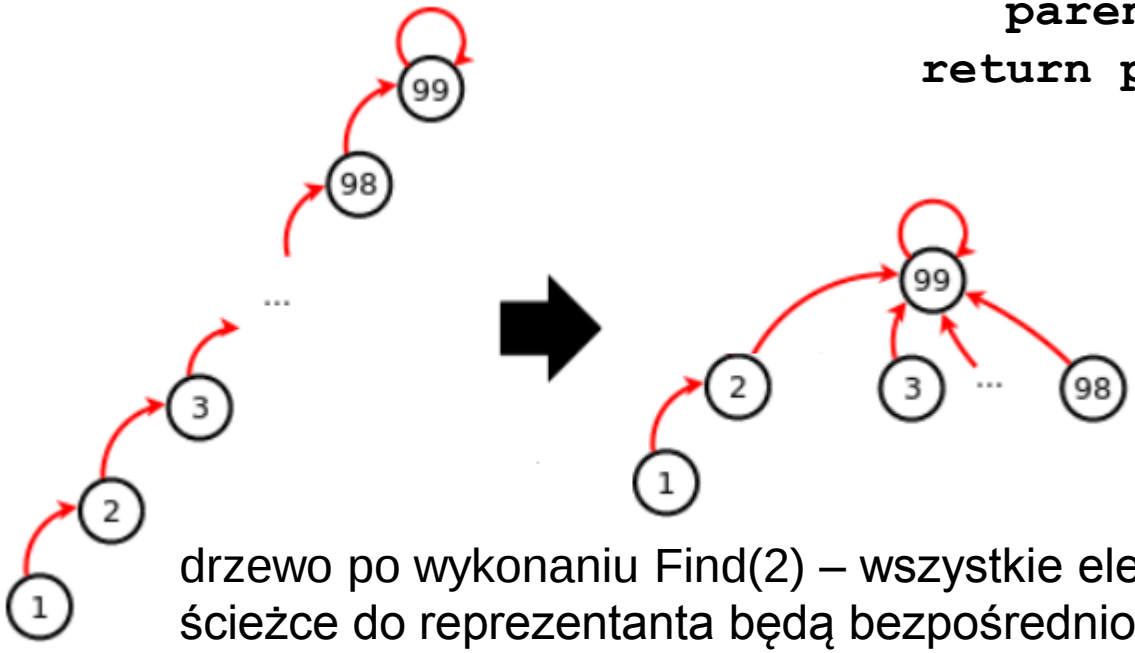
Złożoność Find – O(logn)

Złożoność Union – taka sama jak Find

MST – reprezentacja zbiorów rozłącznych

Kompresja ścieżek – jeżeli podczas operacji Find odkryliśmy, że reprezentantem x jest y to możemy od razy wpisać parent[x]=y, aby skrócić następne poszukiwania.

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



drzewo po wykonaniu Find(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 ∈ do V - {s} do
2     color[u]:= WHITE
3  color[s]:= GREY
4  Q:= s  //Q-zwykła kolejka FIFO – wstaw s do kolejki
5  while Q != ∅ do
6     u:=head[Q]
7     Q:= Q – u // usuń z kolejki u
8     for each v ∈ Adjv[u]
9       if color[v] = WHITE then
10          color[v]:= GREY
11          Q:= Q+v  //wstaw do kolejki wierzchołek v
12    color[u]=BLACK
13    wykonaj opercję na u
```

MST – przeszukiwanie grafu w głąb (DFS)

```
DFS(G)
1 for each u ∈ V do
2   color[u]:=WHITE
3   d[u]:= ∞
4 time:=0
5 for each u ∈ V do
6   if color[u]=WHITE then
7       DFS-VISIT(u)


DFS-VISIT(u)
1  color[u]:=GRAY
2  d[u]:=time:=time+1
3  for each v ∈ Adj[u] do
4     if color[v]= WHITE do then
5         p[v]:=u
6         DFS-VISIT(v)
7  color[u]:=BLACK
8  f[u]=time:=time+1
```