Algorytmy i Struktury Danych

(wybrane algorytmy grafowe)

Michał Knapik

Polsko-Japońska Akademia Technik Komputerowych

27 czerwca 2021

Spis treści

Find-Union: Idea

Problem

Niech $\mathcal{U}$ będzie skończonym uniwersum elementów. Chcemy:

- reprezentować rodziny rozłącznych podzbiorów $\mathcal{U}$

oraz efektywnie wykonywać następujące operacje:

- **Find(x)**: znalezienie zbioru do którego należy **x**
- **Union(A, B)**: scalenie zbiorów **A** i **B**

Find-Union: Idea

Problem

Niech $\mathcal{U}$ będzie skończonym uniwersum elementów. Chcemy:

- reprezentować rodziny rozłącznych podzbiorów $\mathcal{U}$

oraz efektywnie wykonywać następujące operacje:

- **Find(x)**: znalezienie zbioru do którego należy **x**
- **Union(A, B)**: scalenie zbiorów **A** i **B**

(Dokładniejsze specyfikacje w wykładach online. Tu zakładamy, że **Union**$(\cdot,\cdot)$ *jest funkcją o efektach ubocznych, operującą w zbiorze potęgowym* $\mathcal{U}$*.)*

Find-Union: Idea

Przykład, $\mathcal{U} = \{1, 2, 3, 4, 5, 6\}$

Find-Union: Idea

Przykład, $\mathcal{U} = \{1, 2, 3, 4, 5, 6\}$

Załóżmy, że na pewnym etapie mamy: $\{1, 3\}, \{2, 4, 5\}, \{6\}$

Find-Union: Idea

Przykład, $\mathcal{U} = \{1, 2, 3, 4, 5, 6\}$

Załóżmy, że na pewnym etapie mamy: $\{1, 3\}, \{2, 4, 5\}, \{6\}$
I wtedy: **Find(2)** = $\{2, 4, 5\}$

Find-Union: Idea

Przykład, $\mathcal{U} = \{1, 2, 3, 4, 5, 6\}$

Załóżmy, że na pewnym etapie mamy: $\{1, 3\}, \{2, 4, 5\}, \{6\}$
I wtedy: **Find(2)** = $\{2, 4, 5\}$
Możemy wykonać np. **Union(**$\{1, 3\}$**,** $\{6\}$**)**

Find-Union: Idea

Przykład, $\mathcal{U} = \{1, 2, 3, 4, 5, 6\}$

Załóżmy, że na pewnym etapie mamy: $\{1, 3\}, \{2, 4, 5\}, \{6\}$
I wtedy: **Find(2)** = $\{2, 4, 5\}$
Możemy wykonać np. **Union(**$\{1, 3\}$**,** $\{6\}$**)**
I otrzymamy: $\{1, 3, 6\}, \{2, 4, 5\}$

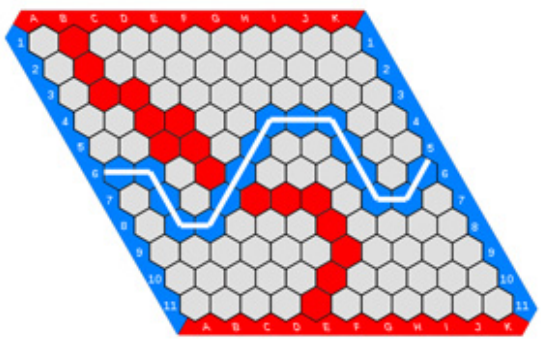
Find-Union: Idea

Przykład, $\mathcal{U} = \{1, 2, 3, 4, 5, 6\}$

Załóżmy, że na pewnym etapie mamy: $\{1, 3\}, \{2, 4, 5\}, \{6\}$
I wtedy: **Find(2)** = $\{2, 4, 5\}$
Możemy wykonać np. **Union(**$\{1, 3\}$**,** $\{6\}$**)**
I otrzymamy: $\{1, 3, 6\}, \{2, 4, 5\}$

Bardzo różne zastosowania:

- Symulacje fizyczne (perkolacje)
- Gry - np. Hex
- Budowa minimalnego drzewa rozpinającego
- I wiele innych...



Find-Union: Find

Podzbiory rozłączne $\mathcal{U}$ reprezentujemy jako drzewa, których wierzchołki mają strukturę:

- pole *val* elementu
- wskaźnik *fath* do ojca (NULL w korzeniu)
- tablica wskaźników *child* do synów (NULL w liściach)
- pomocniczo: wysokość *height* poddrzewa

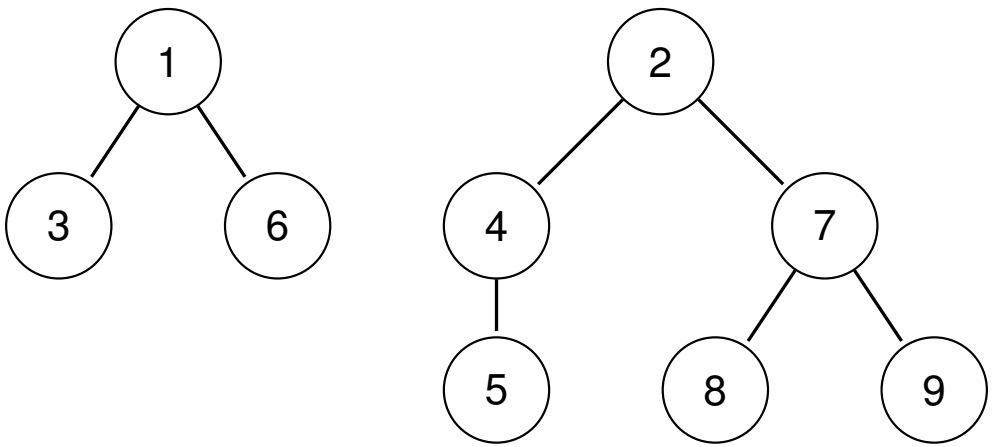
Etykietą zbioru zawierającego element **x** jest więc korzeń drzewa w którym się on znajduje.

Find(x): pierwsza przymiarka

```
curr := x
while curr.fath != NULL
    curr := curr.fath
end while
return curr
```

Find-Union: Ilustracja Find

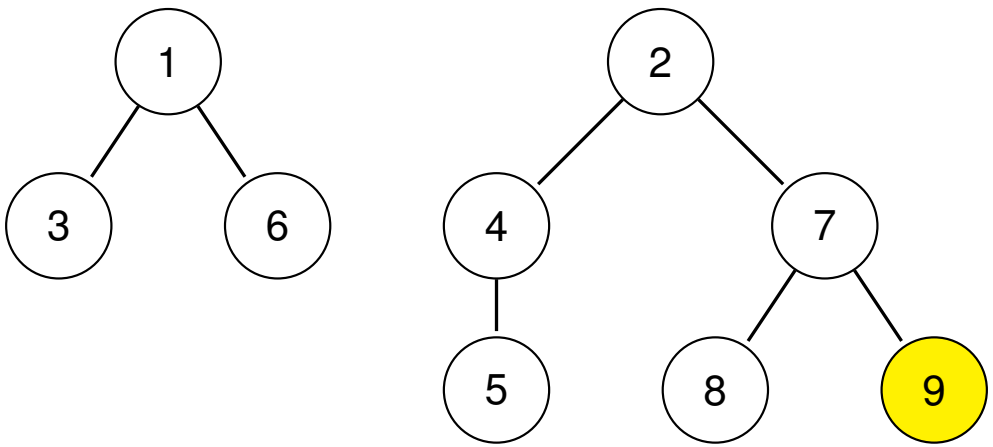
$$\{1, 3, 6\}, \{2, 4, 5, 7, 8, 9\}$$



Find(9)

Find-Union: Ilustracja Find

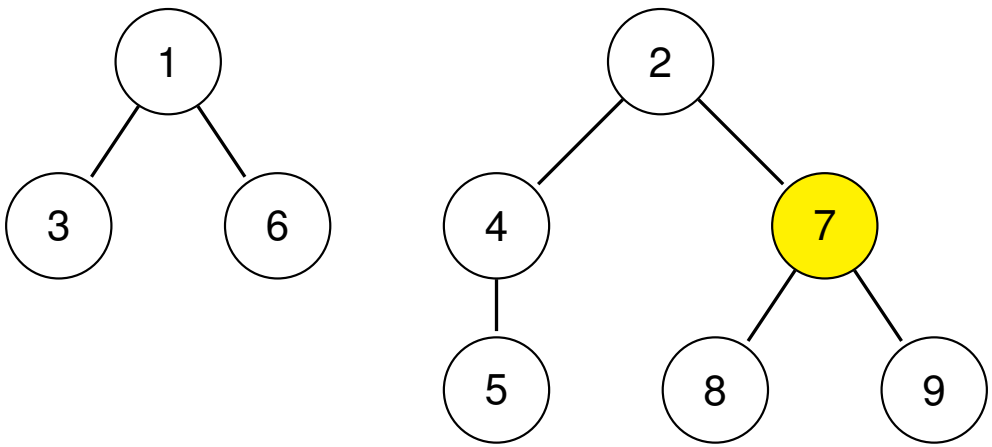
$$\{1, 3, 6\}, \{2, 4, 5, 7, 8, 9\}$$



Find(9)

Find-Union: Ilustracja Find

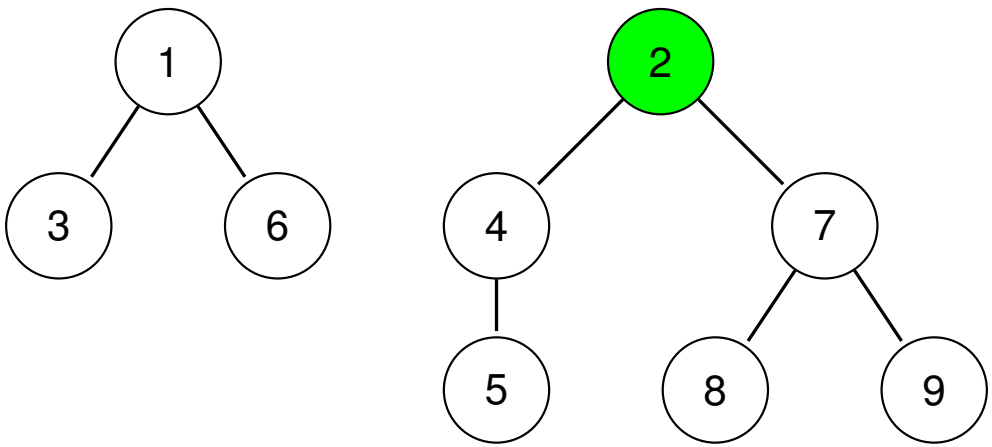
$$\{1, 3, 6\}, \{2, 4, 5, 7, 8, 9\}$$



Find(9)

Find-Union: Ilustracja Find

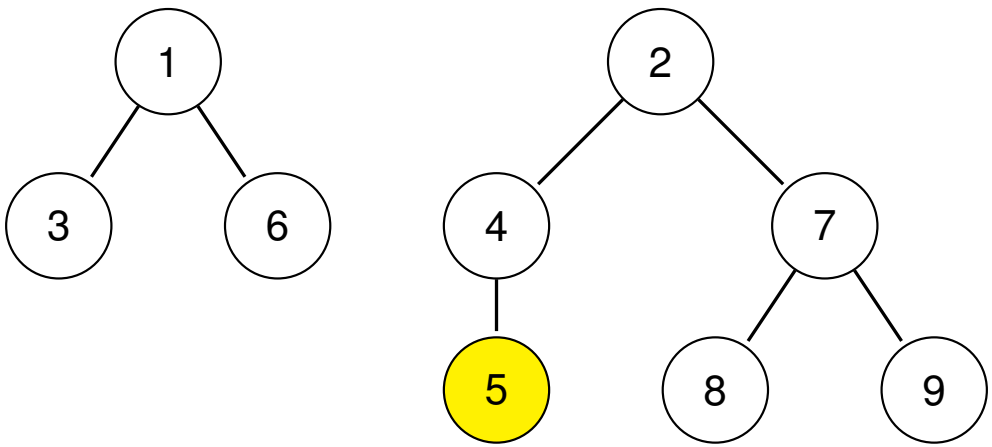
$$\{1,3,6\},\{2,4,5,7,8,9\}$$



Find(9)

Find-Union: Ilustracja Find

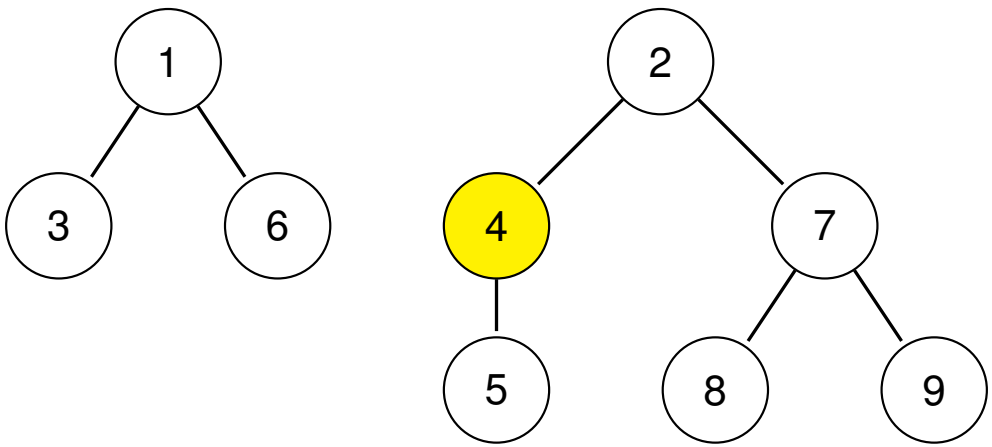
$$\{1,3,6\},\{2,4,5,7,8,9\}$$



Find(5)

Find-Union: Ilustracja Find

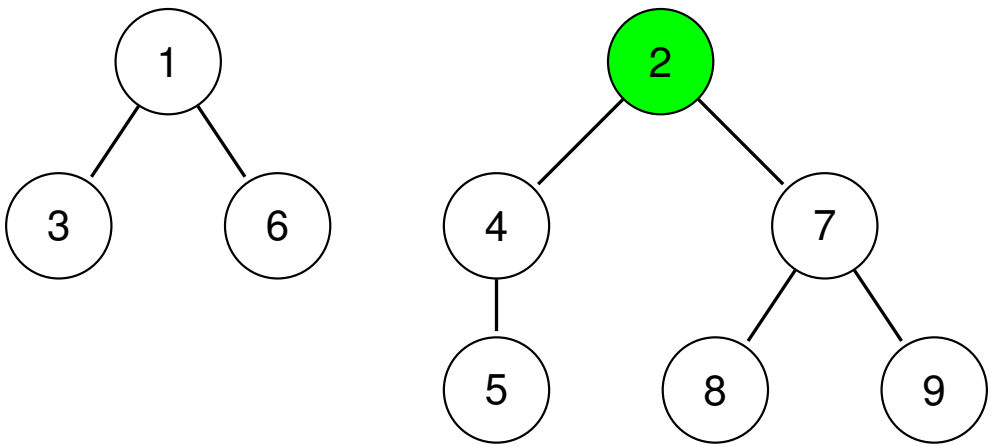
$$\{1, 3, 6\}, \{2, 4, 5, 7, 8, 9\}$$



Find(5)

Find-Union: Ilustracja Find

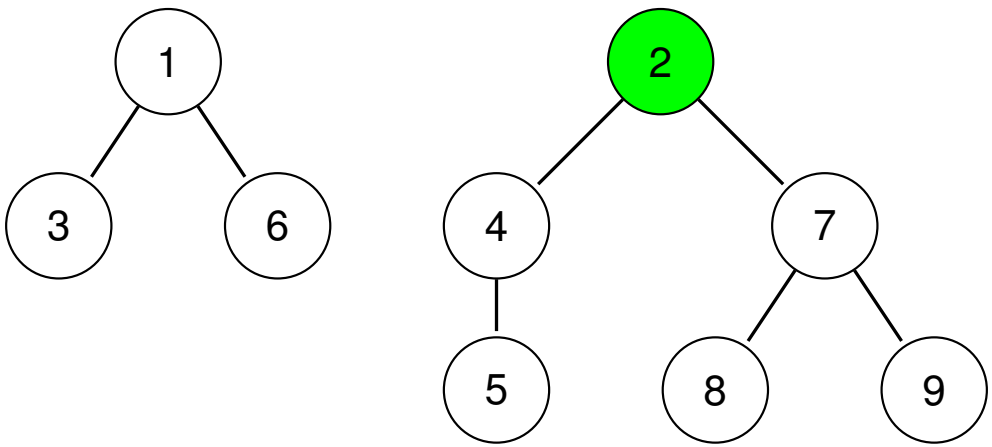
$$\{1,3,6\},\{2,4,5,7,8,9\}$$



Find(5)

Find-Union: Ilustracja Find

$$\{1, 3, 6\}, \{2, 4, 5, 7, 8, 9\}$$



Find(5)

Złożoność czasowa **Find(x)** zależy od wysokości drzewa. Ale **ponowne wyszukanie** tego elementu powinno być szybsze!

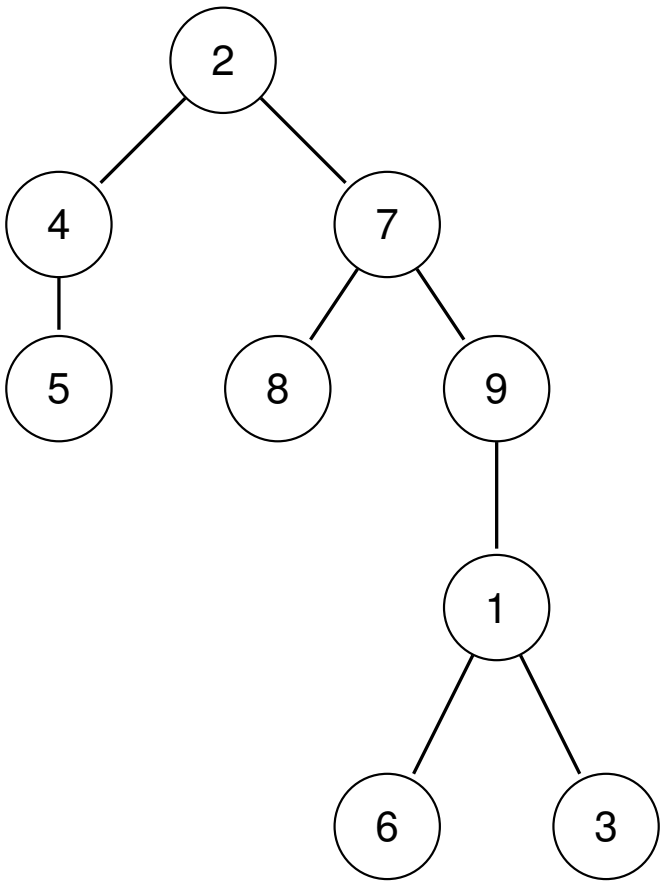
Find-Union: Poprawiamy Find

Find(x): kompresja ścieżek

```
curr := x  //pierwszy przebieg bez zmian
while curr.fath != NULL
    curr := curr.fath
end while
top := curr

curr := x  //ponownie przejdz sciezke od x
while curr.fath != NULL
    tmp := curr.fath
    curr.fath := top //podpinajac kazdy napotkany
    curr := tmp      //wierzcholek bezp. pod korzen
end while
top := curr

return top
```

Find-Union: Ilustracja Find z Kompresją

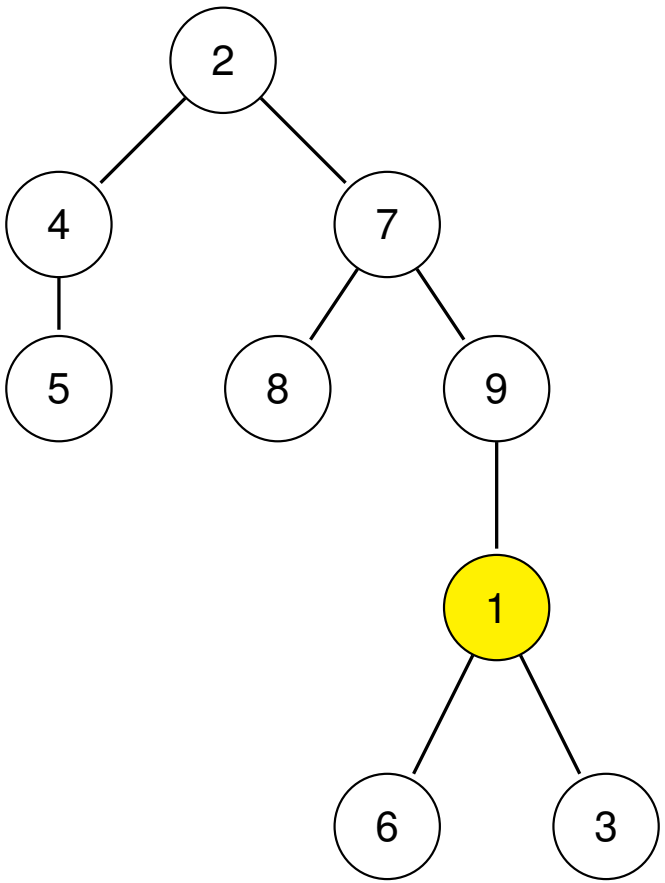
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

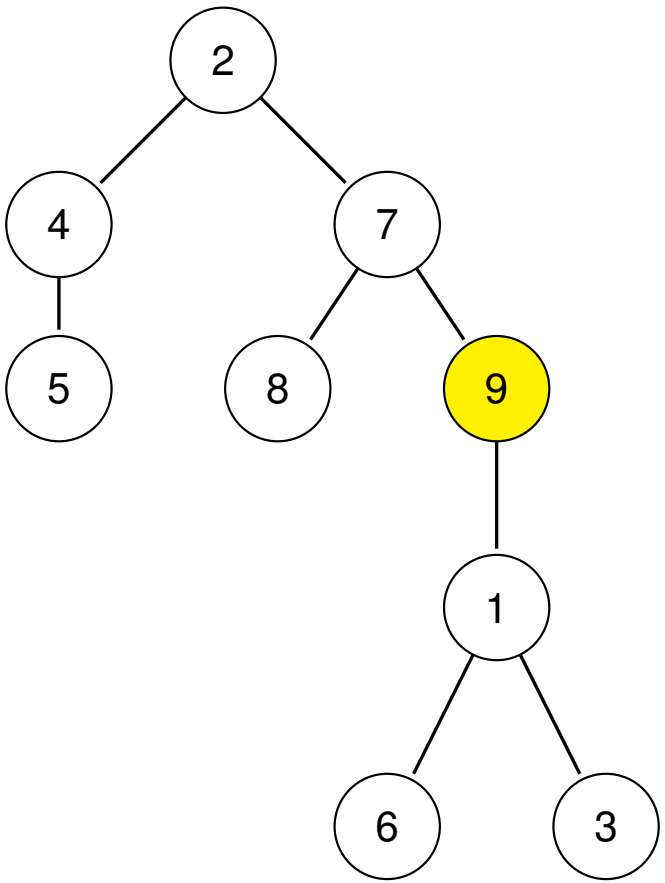
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

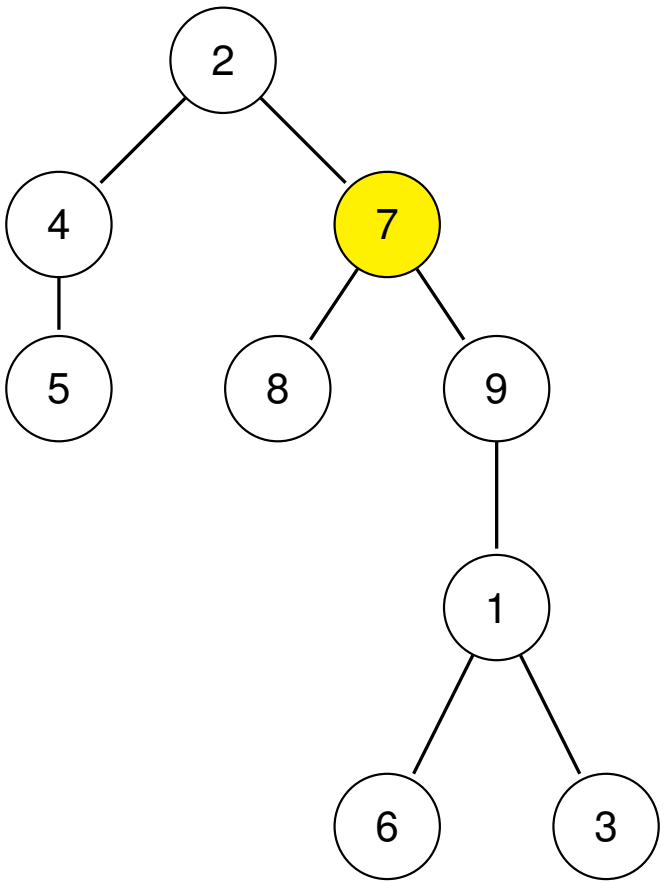
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

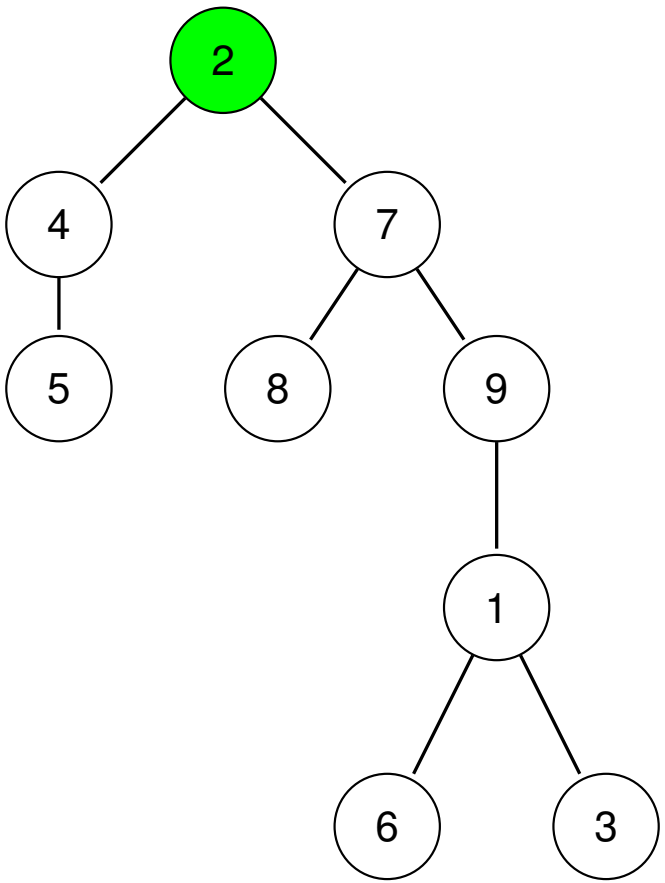
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

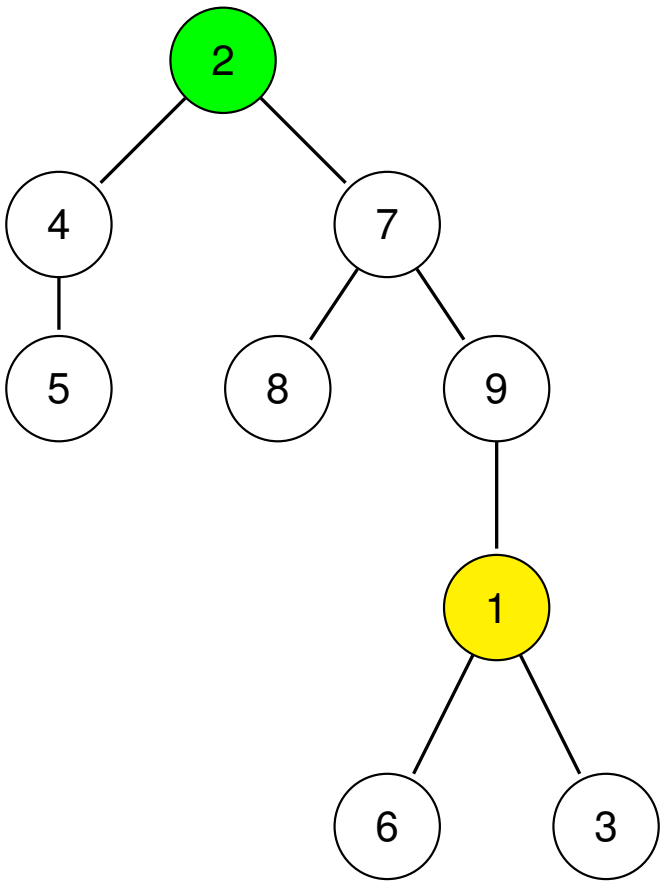
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

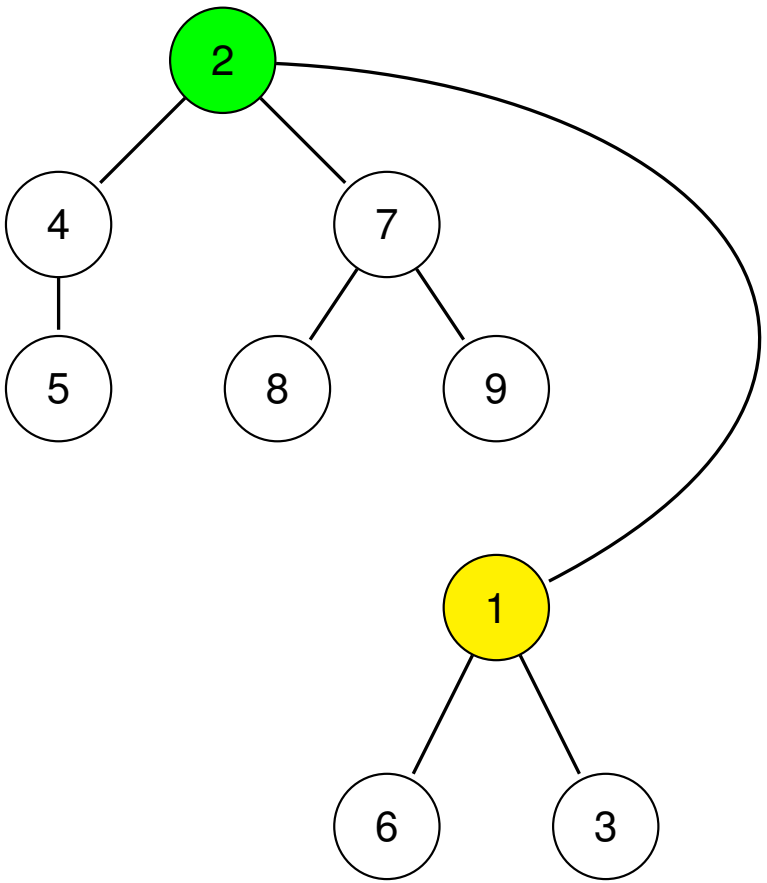
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

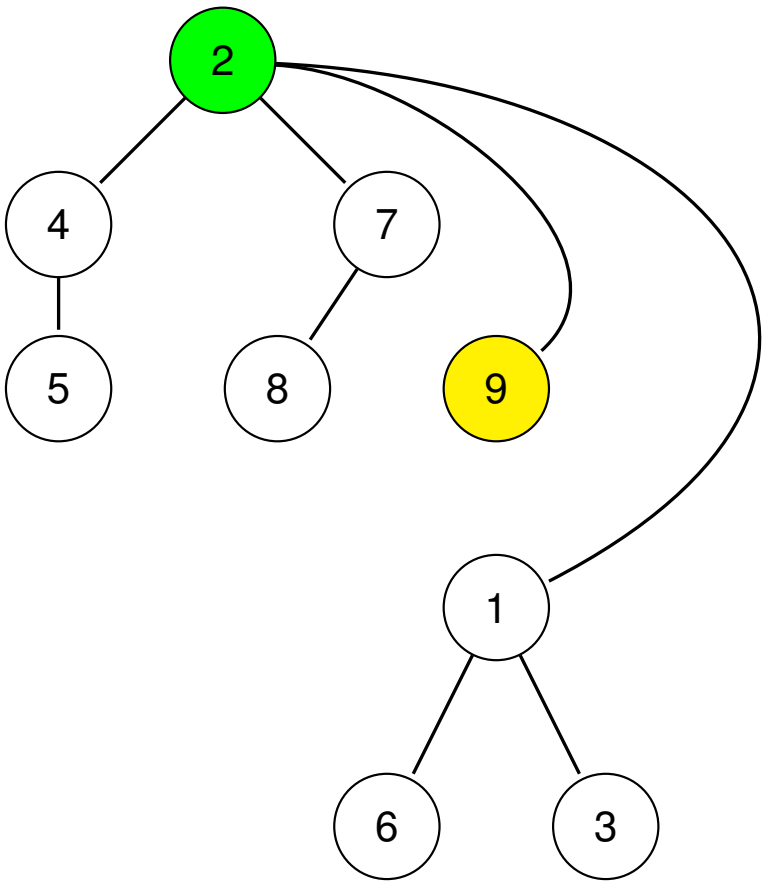
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

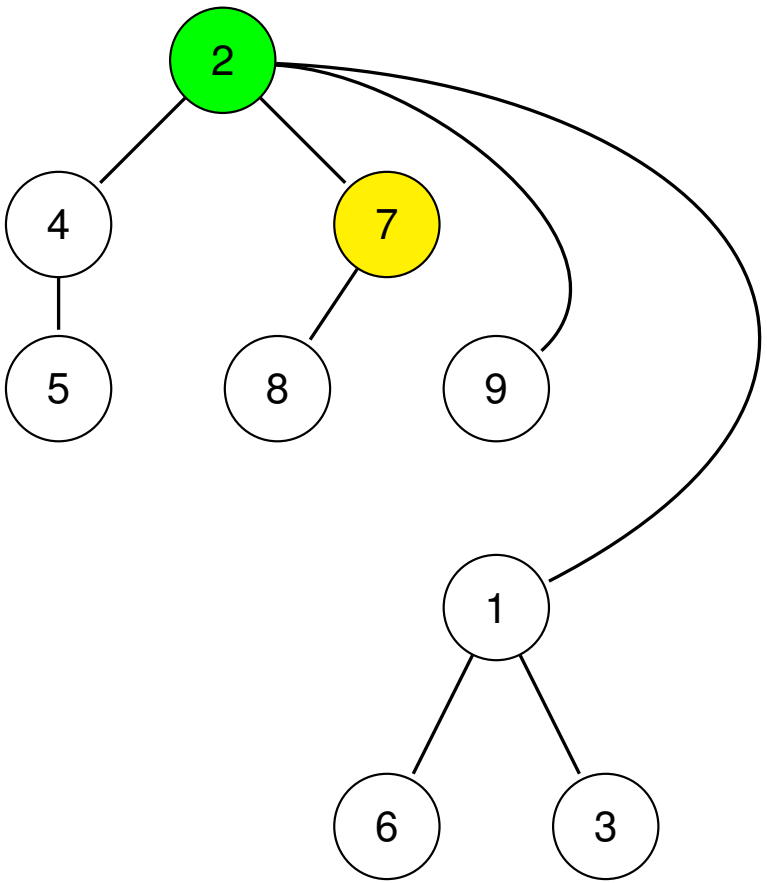
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

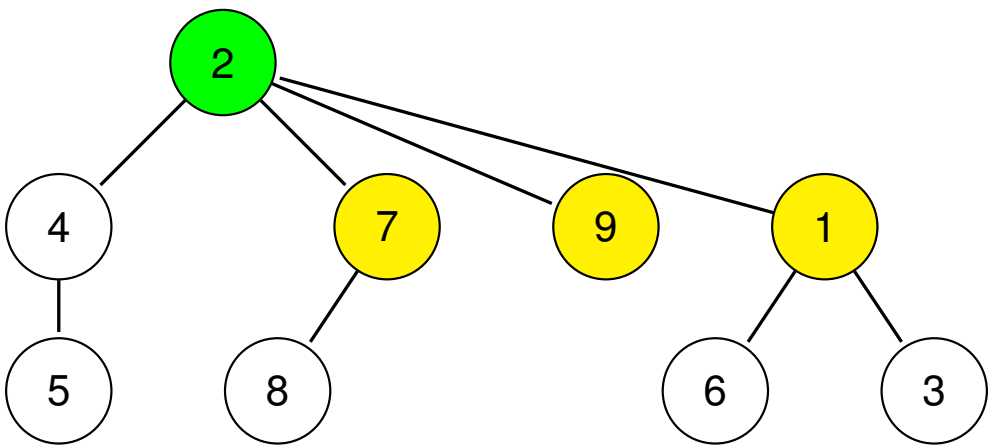
$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1)

Find-Union: Ilustracja Find z Kompresją

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$



Find(1) : kompresja zmniejszyła wysokość drzewa o połowę.

Find-Union: Union

Bardzo prosta idea: podpinamy drzewko mniejsze do korzenia drzewka większego.

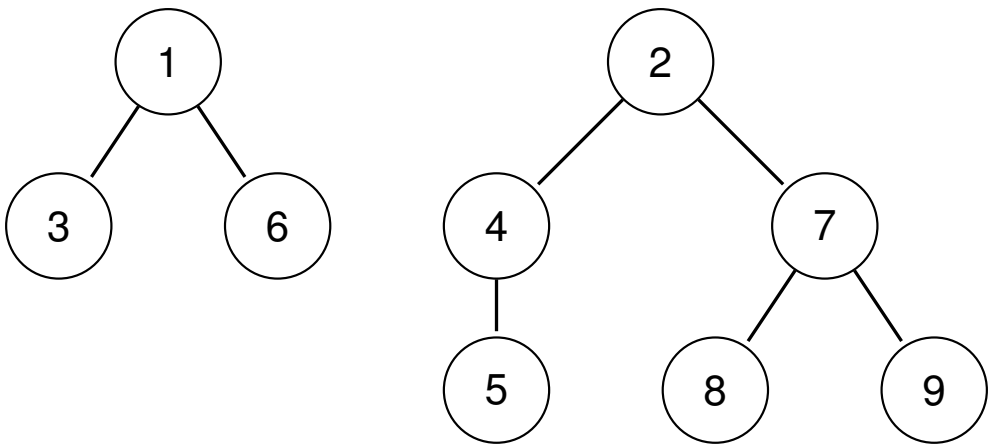
Union(A, B)

```
smaller := minheight(FindTop(A), FindTop(B))
bigger := maxheight(FindTop(A), FindTop(B))

//{smaller, bigger} = {A, B}
//smaller.height <= bigger.height

bigger.child.insert(smaller)

return bigger
```

Złożoność: stała.

Find-Union: Ilustracja Union

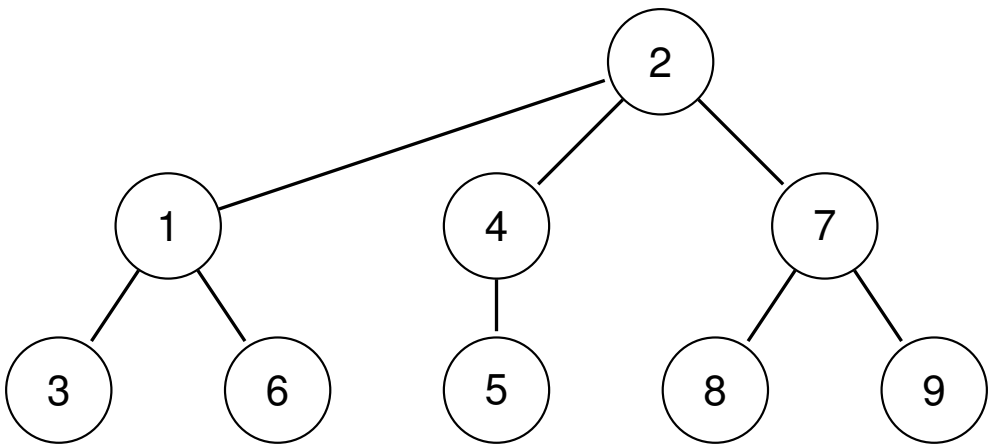
$$\{1, 3, 6\}, \{2, 4, 5, 7, 8, 9\}$$



Union($\{1, 3, 6\}$**,** $\{2, 4, 5, 7, 8, 9\}$**)**

Find-Union: Ilustracja Union

$$\{1, 3, 6, 2, 4, 5, 7, 8, 9\}$$



Union($\{1, 3, 6\}$**,** $\{2, 4, 5, 7, 8, 9\}$**)**

Find-Union: Uzupełnienia

Twierdzenie o złożoności czasowej Find-Union

Wykonanie **M** operacji **Find(·)** i **Union**$(\cdot, \cdot)$ na **N** elementach zajmuje **O(N + Mlog*N)**.

Uzupełnienia:

- log*N: logarytm iterowany; w tym wszechświecie < 6
- znane inne strategie poprawiania ścieżek (jednoprzebiegowe, itp)
- dowód powyższego twierdzenia jest trudny

Podstawowe Definicje: Drzewa Rozpinające

Minimalne Drzewo Rozpinające

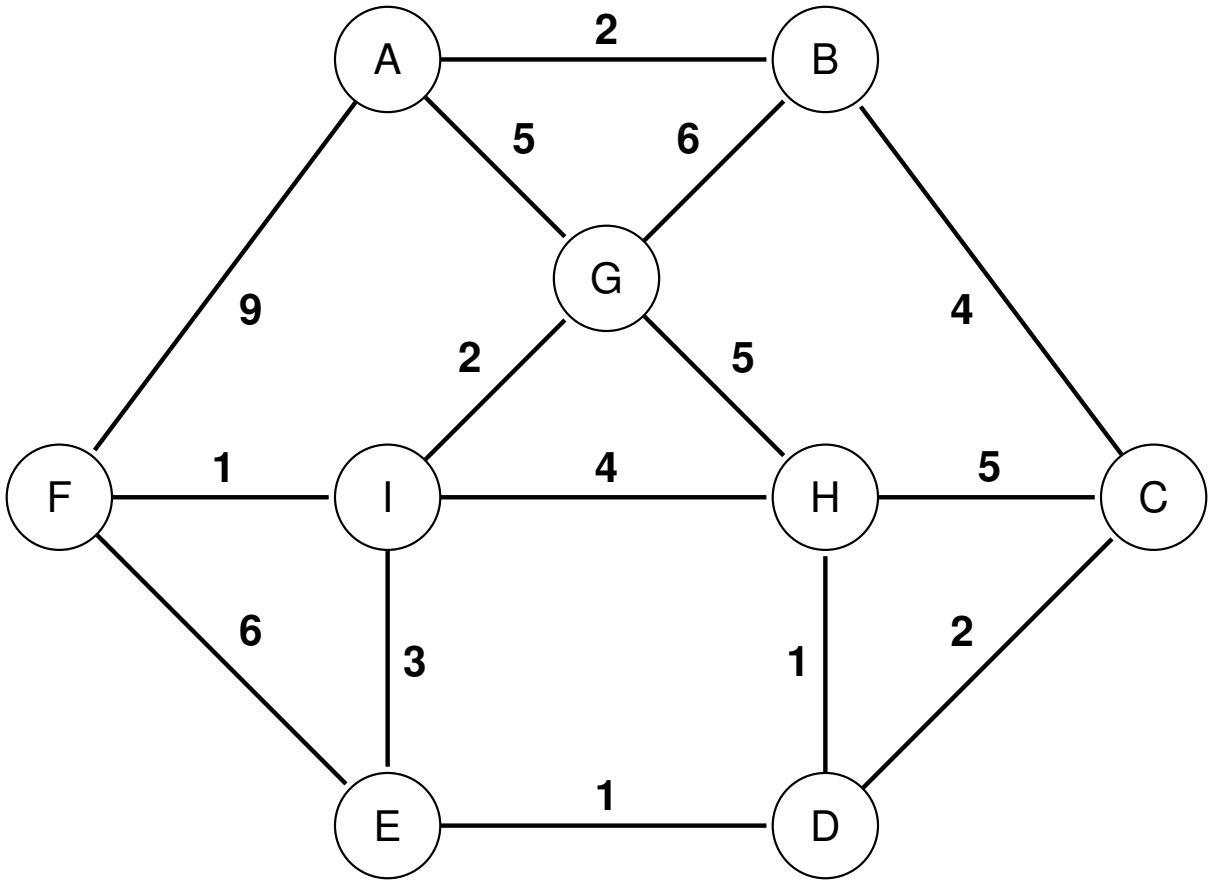
Niech $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ będzie grafem niezorientowanym, a $c\colon \mathcal{E} \to \mathbb{R}_+$ **funkcją kosztu**, przyporządkowującą krawędziom nieujemne liczby rzeczywiste.

Minimalne drzewo rozpinające $\mathrm{MST}(\mathcal{G})$ to drzewo rozpinające $\mathcal{G}$, w którym suma kosztów krawędzi jest najmniejsza.

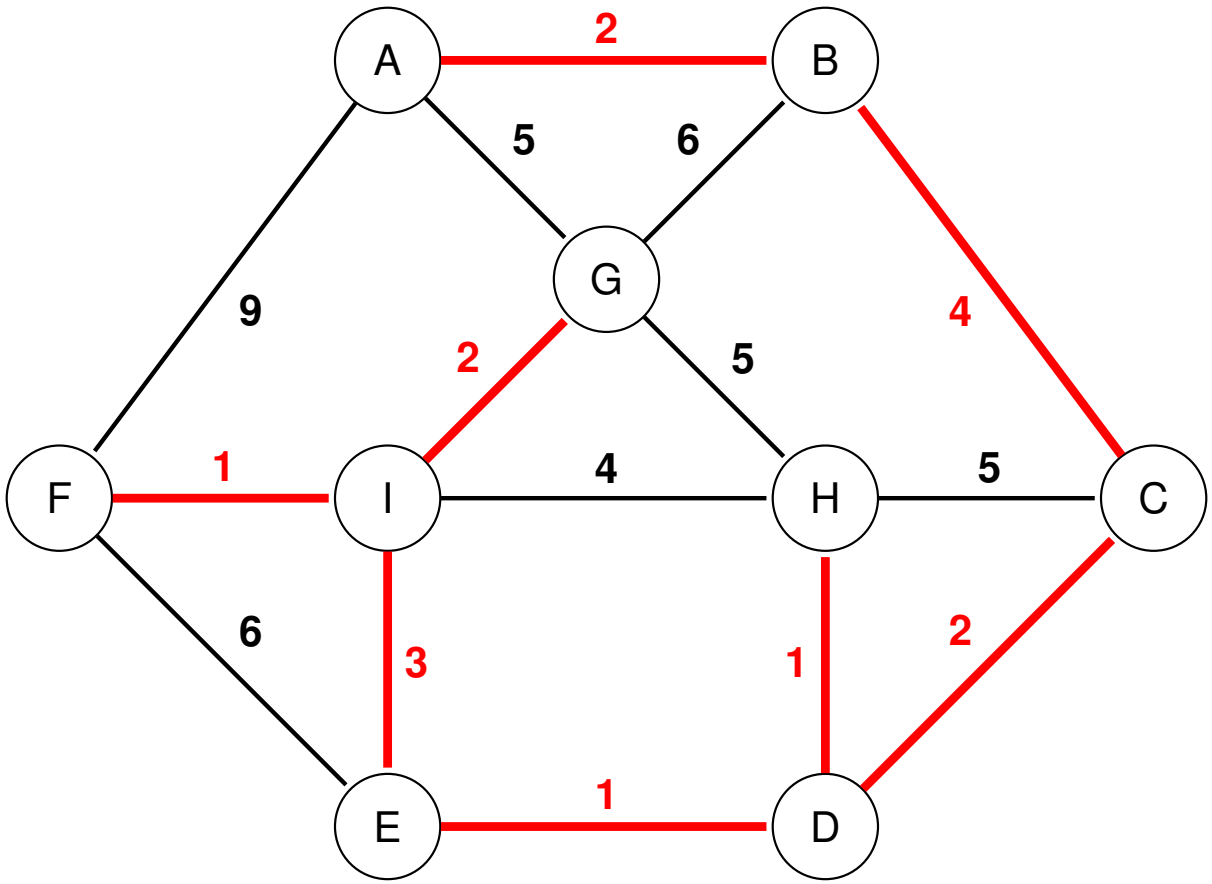
Różne zastosowania:

- Spanning Tree Protocol: budowa logicznej sieci minimalizującej energię (w warstwie 2 OSI)
- Segmentacja obrazu: budowa hierarchii obiektów
- Biologia obliczeniowa: taksonomia i inne
- I wiele innych...

Podstawowe Definicje: Drzewa Rozpinające



Podstawowe Definicje: Drzewa Rozpinające



MinCost = 16

Podstawowe Definicje: Drzewa Rozpinające

Lemat

Jeśli koszty krawędzi grafu $\mathcal{G}$ są różne, to istnieje tylko jedno minimalne drzewo rozpinające $\mathrm{MST}(\mathcal{G})$.

Podstawowe Definicje: Drzewa Rozpinające

Lemat

Jeśli koszty krawędzi grafu $\mathcal{G}$ są różne, to istnieje tylko jedno minimalne drzewo rozpinające $\mathrm{MST}(\mathcal{G})$.

Zarys dowodu:

Podstawowe Definicje: Drzewa Rozpinające

Lemat

Jeśli koszty krawędzi grafu $\mathcal{G}$ są różne, to istnieje tylko jedno minimalne drzewo rozpinające $\mathrm{MST}(\mathcal{G})$.

Zarys dowodu:

- Załóżmy, że są dwa takie drzewa i wypiszmy ich krawędzie, posortowane rosnąco

Podstawowe Definicje: Drzewa Rozpinające

Lemat

Jeśli koszty krawędzi grafu $\mathcal{G}$ są różne, to istnieje tylko jedno minimalne drzewo rozpinające $\mathrm{MST}(\mathcal{G})$.

Zarys dowodu:

- Załóżmy, że są dwa takie drzewa i wypiszmy ich krawędzie, posortowane rosnąco
- Pierwsza różnica w tych listach: wybierzmy mniejszą krawędź e

Podstawowe Definicje: Drzewa Rozpinające

Lemat

Jeśli koszty krawędzi grafu $\mathcal{G}$ są różne, to istnieje tylko jedno minimalne drzewo rozpinające $\mathrm{MST}(\mathcal{G})$.

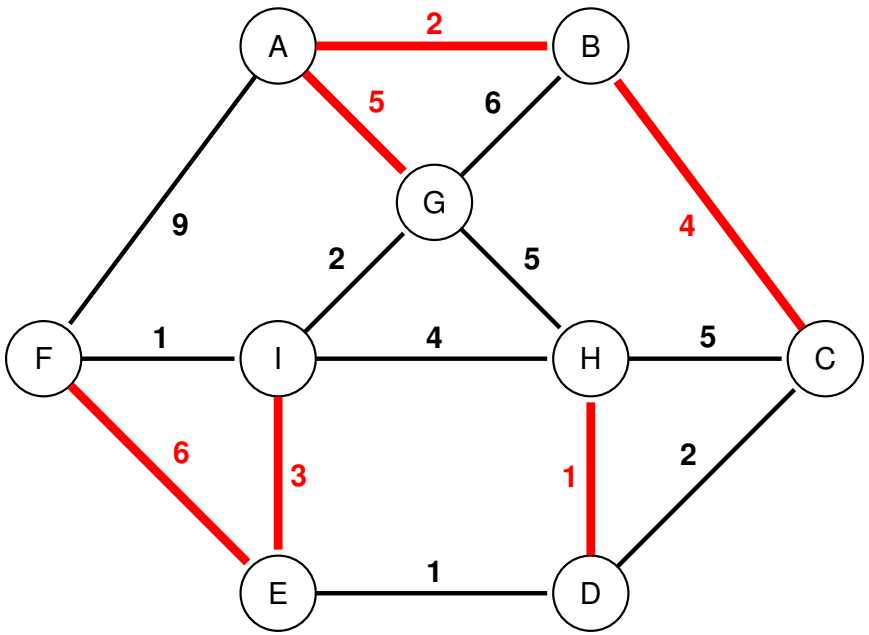
Zarys dowodu:

- Załóżmy, że są dwa takie drzewa i wypiszmy ich krawędzie, posortowane rosnąco
- Pierwsza różnica w tych listach: wybierzmy mniejszą krawędź e
- Dołączmy tę krawędź do drugiego drzewa; z definicji powstanie teraz cykl, który można zredukować usuwając krawędź większą niż e -> sprzeczność!

Podstawowe Definicje: Las Rozpinający

Las Rozpinający

Niech $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ będzie grafem niezorientowanym. **Las rozpinający** $\mathcal{G}$, to dowolna rodzina jego rozłącznych podgrafów będących drzewami, których wierzchołki pokrywają $\mathcal{V}$.



Algorytm Kruskala

Własności Algorytmu Kruskala

- **Kruskal($\mathcal{G}$)**: minimalne drzewo rozpinające grafu $\mathcal{G}$
- Algorytm zachłanny
- Bardzo prosty, dzięki dobrze dobranym strukturom danych
- Złożoność: **O(ElogE)** $=$ **O(ElogV)**, gdzie **E** $= |\mathcal{E}|$ i **V** $= |\mathcal{V}|$

Algorytm Kruskala

Algorytm Kruskala: Idea

F: rosnący las rozpinający $\mathcal{G}$, **S**: zbiór wszystkich krawędzi

Algorytm Kruskala

Algorytm Kruskala: Idea

F: rosnący las rozpinający $\mathcal{G}$, **S**: zbiór wszystkich krawędzi

1. **F** := $\mathcal{V}$ *(las jednowierzchołkowych drzew)*

Algorytm Kruskala

Algorytm Kruskala: Idea

F: rosnący las rozpinający $\mathcal{G}$, **S**: zbiór wszystkich krawędzi

1. **F** := $\mathcal{V}$ *(las jednowierzchołkowych drzew)*
2. **dopóki** **S** jest niepusty:

Algorytm Kruskala

Algorytm Kruskala: Idea

F: rosnący las rozpinający $\mathcal{G}$, **S**: zbiór wszystkich krawędzi

1. **F** := $\mathcal{V}$ *(las jednowierzchołkowych drzew)*
2. **dopóki** **S** jest niepusty:
3. usuń z **S** krawędź **e** o najmniejszym koszcie

Algorytm Kruskala

Algorytm Kruskala: Idea

F: rosnący las rozpinający $\mathcal{G}$, **S**: zbiór wszystkich krawędzi

1. **F** := $\mathcal{V}$ *(las jednowierzchołkowych drzew)*
2. **dopóki** **S** jest niepusty:
3. usuń z **S** krawędź **e** o najmniejszym koszcie
4. jeśli **e** łączy dwa różne drzewa w **F**, to połącz je w jedno

Algorytm Kruskala

Algorytm Kruskala: Idea

F: rosnący las rozpinający $\mathcal{G}$, **S**: zbiór wszystkich krawędzi

1. **F** := $\mathcal{V}$ *(las jednowierzchołkowych drzew)*
2. **dopóki** **S** jest niepusty:
3. usuń z **S** krawędź **e** o najmniejszym koszcie
4. jeśli **e** łączy dwa różne drzewa w **F**, to połącz je w jedno

Jak to zaimplementować?

Algorytm Kruskala

Algorytm Kruskala: Idea

F: rosnący las rozpinający $\mathcal{G}$, **S**: zbiór wszystkich krawędzi

1. **F** := $\mathcal{V}$ *(las jednowierzchołkowych drzew)*
2. **dopóki** **S** jest niepusty:
3. usuń z **S** krawędź **e** o najmniejszym koszcie
4. jeśli **e** łączy dwa różne drzewa w **F**, to połącz je w jedno

Jak to zaimplementować? Jakiej struktury danych użyć dla **S**?

Algorytm Kruskala

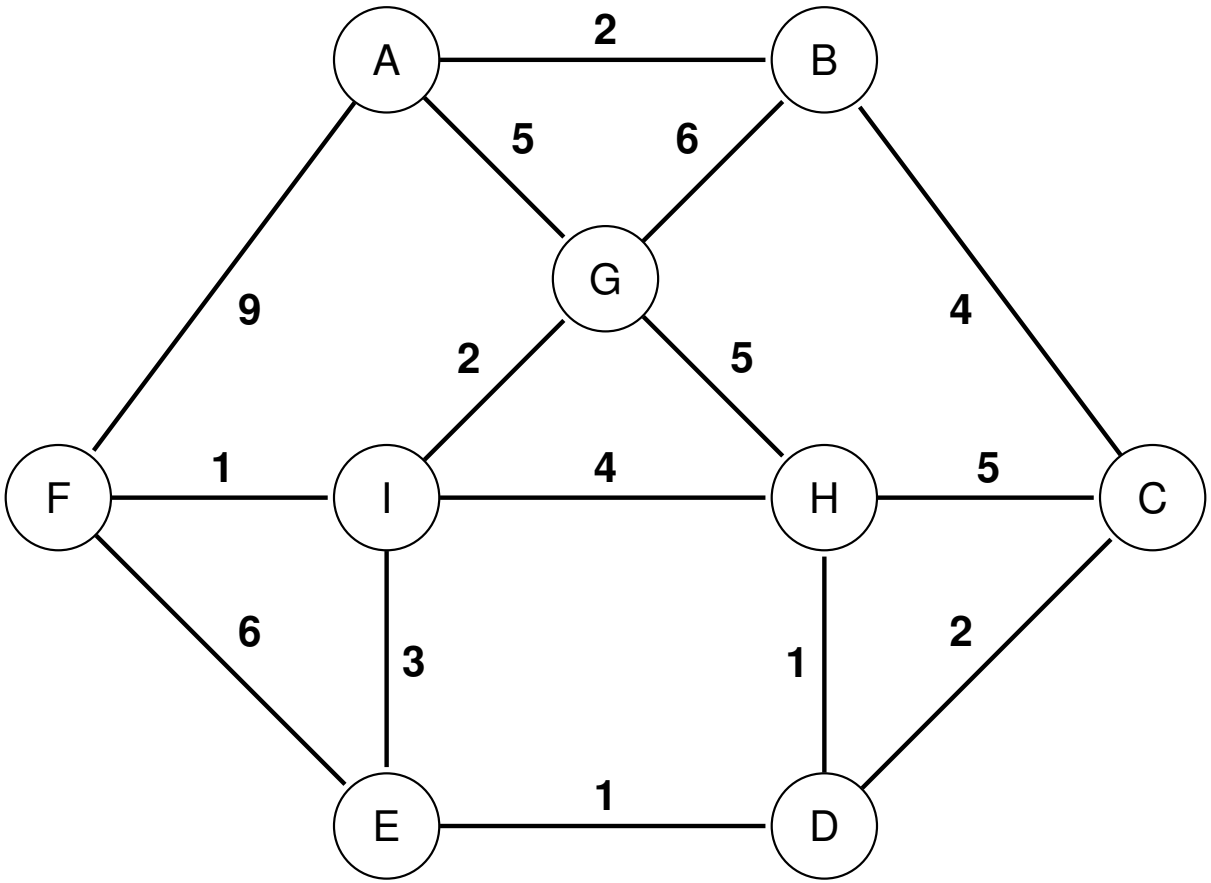
Algorytm Kruskala: Idea

F: rosnący las rozpinający $\mathcal{G}$, **S**: zbiór wszystkich krawędzi

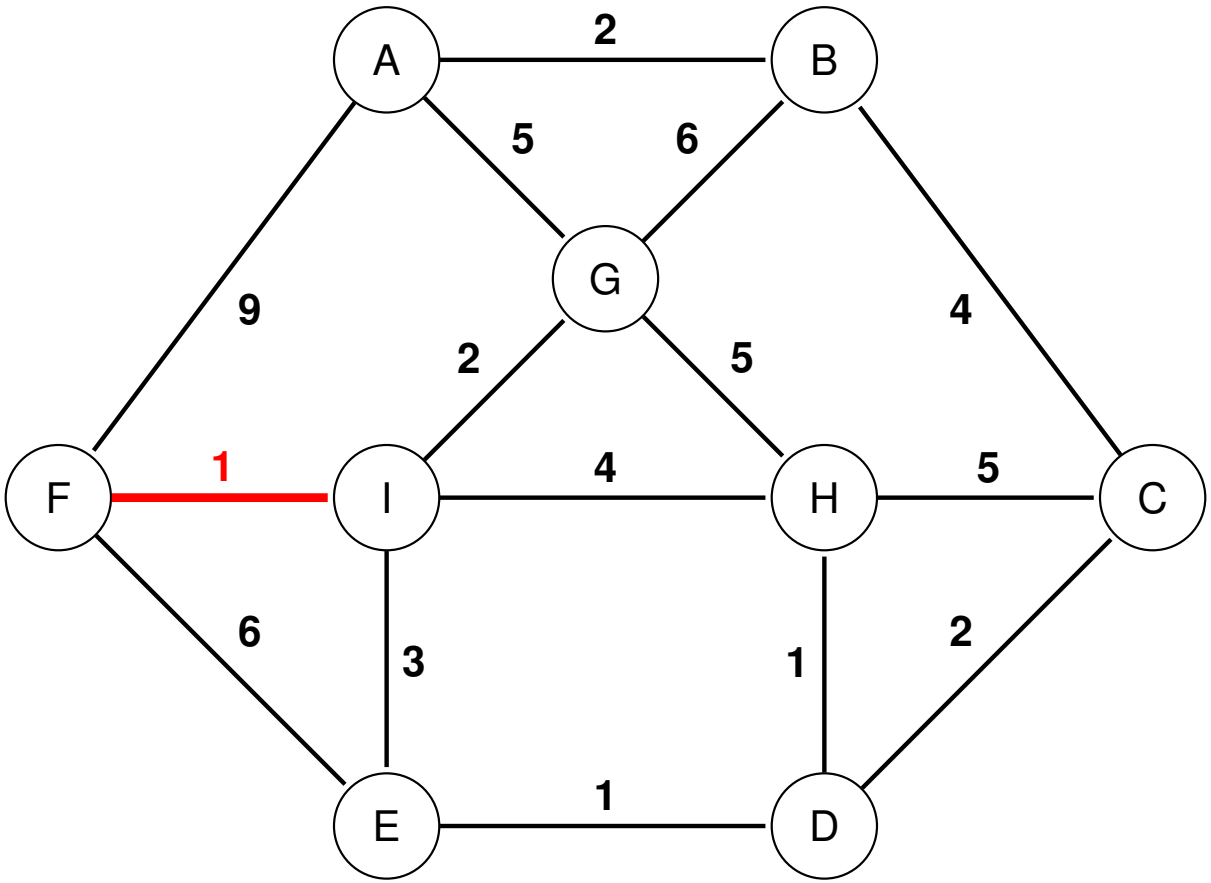
1. **F** := $\mathcal{V}$ *(las jednowierzchołkowych drzew)*
2. **dopóki** **S** jest niepusty:
3. usuń z **S** krawędź **e** o najmniejszym koszcie
4. jeśli **e** łączy dwa różne drzewa w **F**, to połącz je w jedno

Jak to zaimplementować? Jakiej struktury danych użyć dla **S**? A dla **F**?

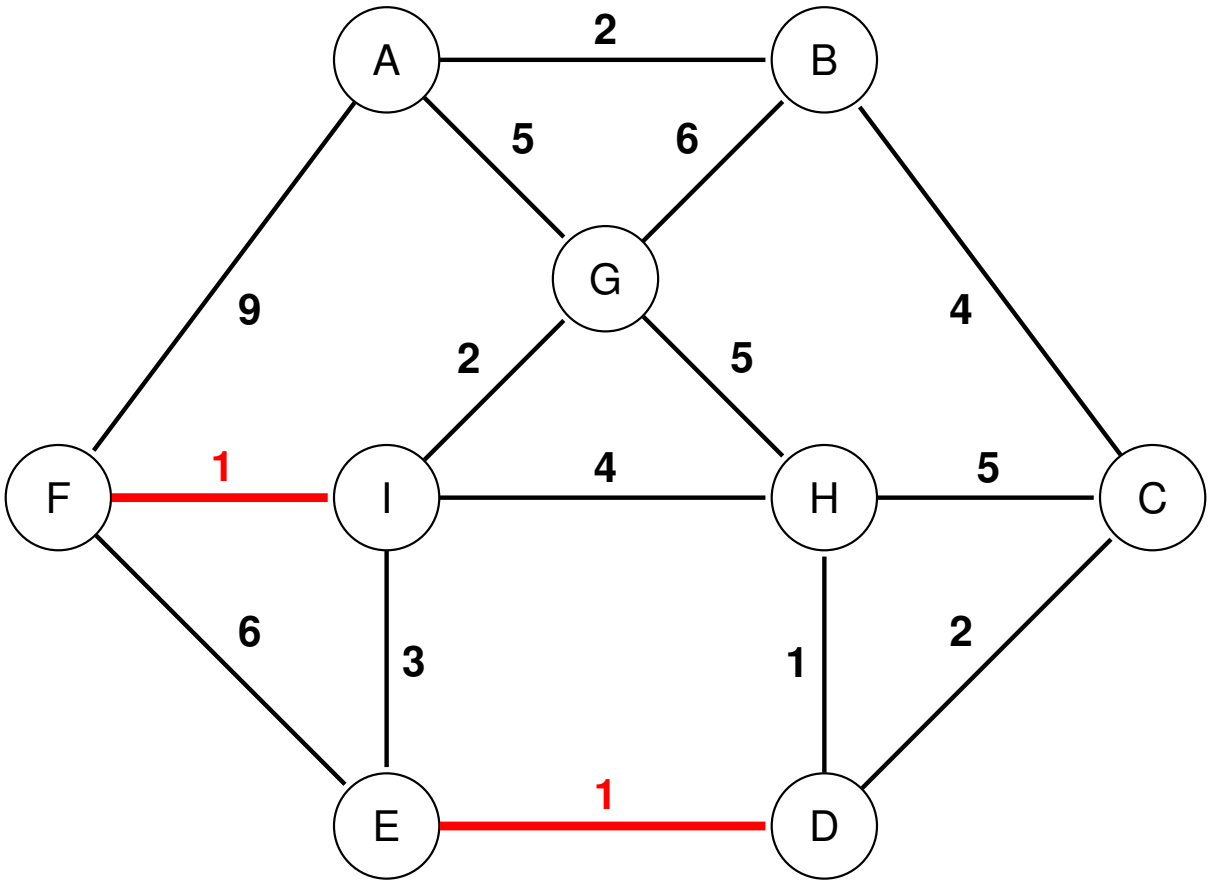
Algorytm Kruskala: Ilustracja



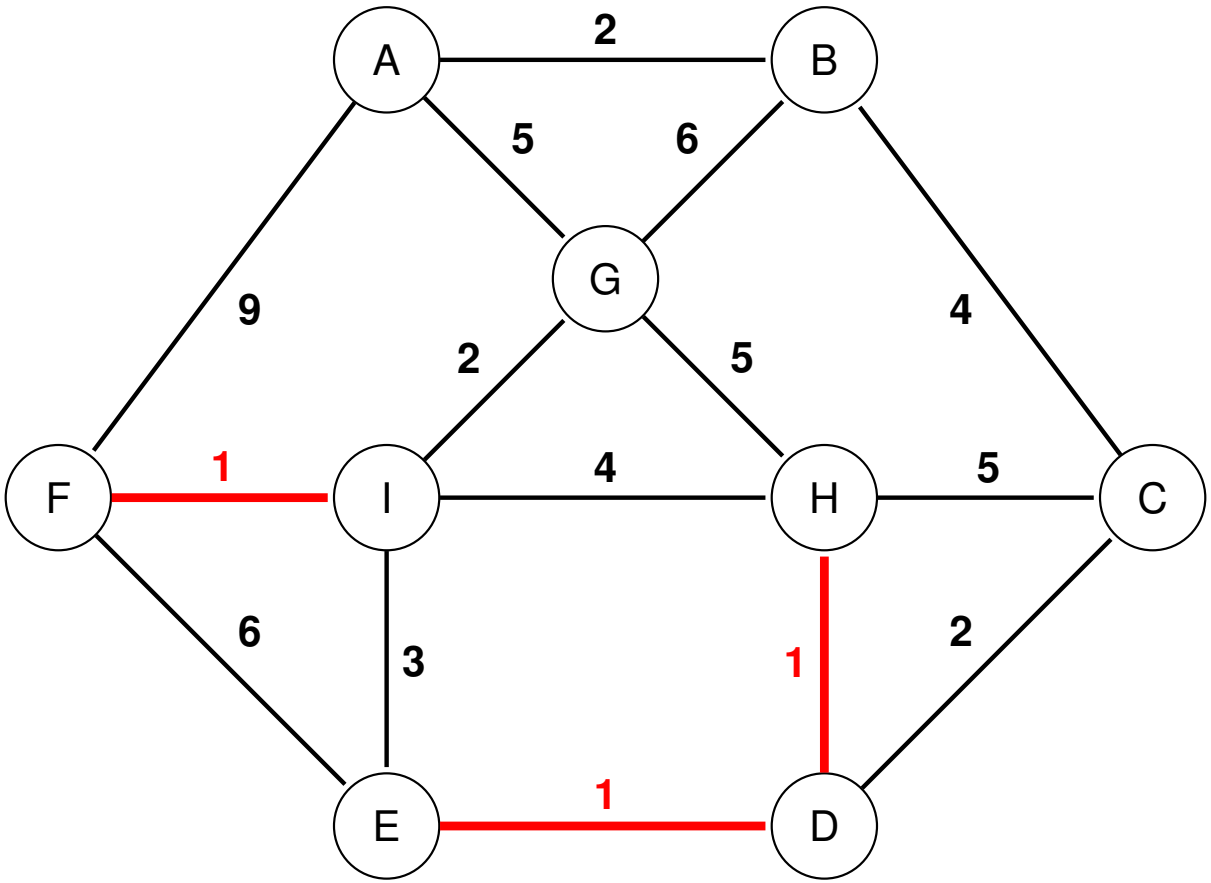
Algorytm Kruskala: Ilustracja



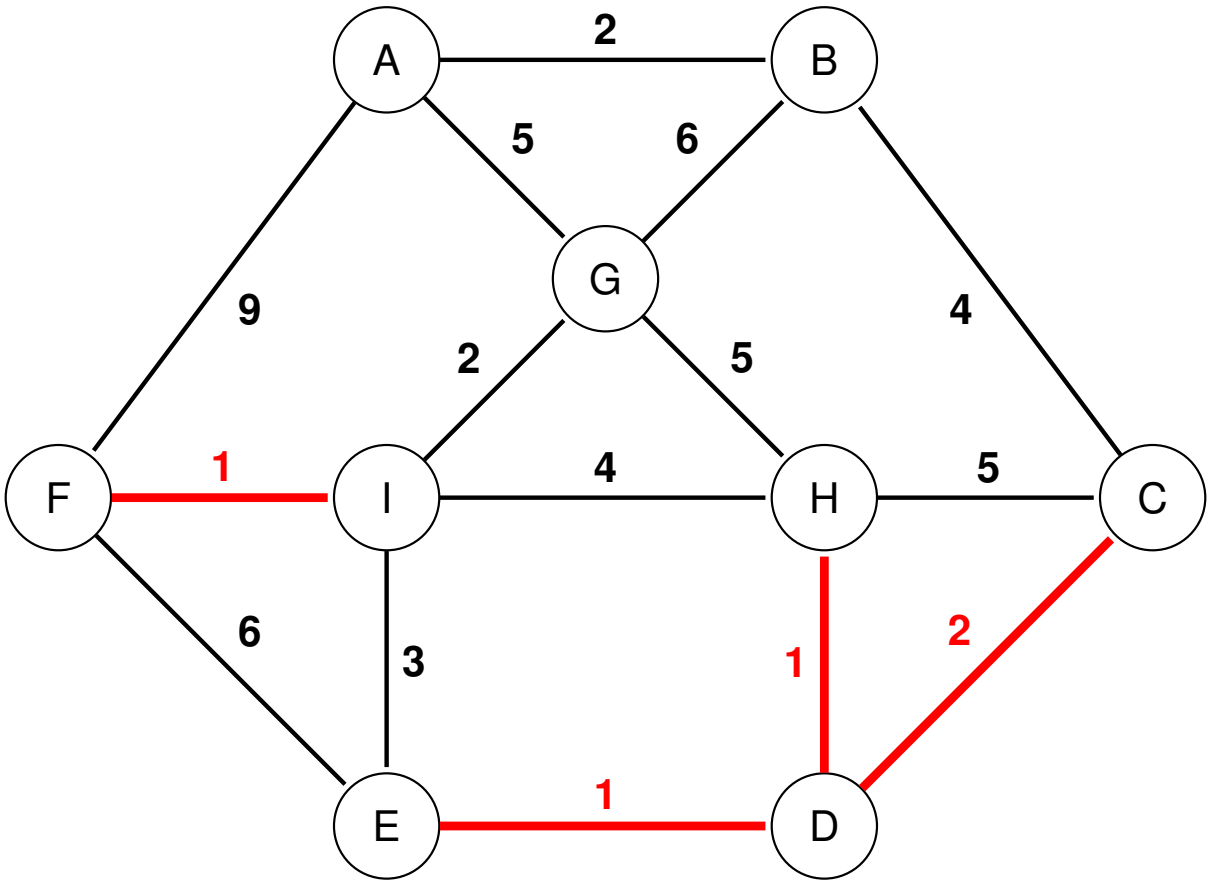
Algorytm Kruskala: Ilustracja



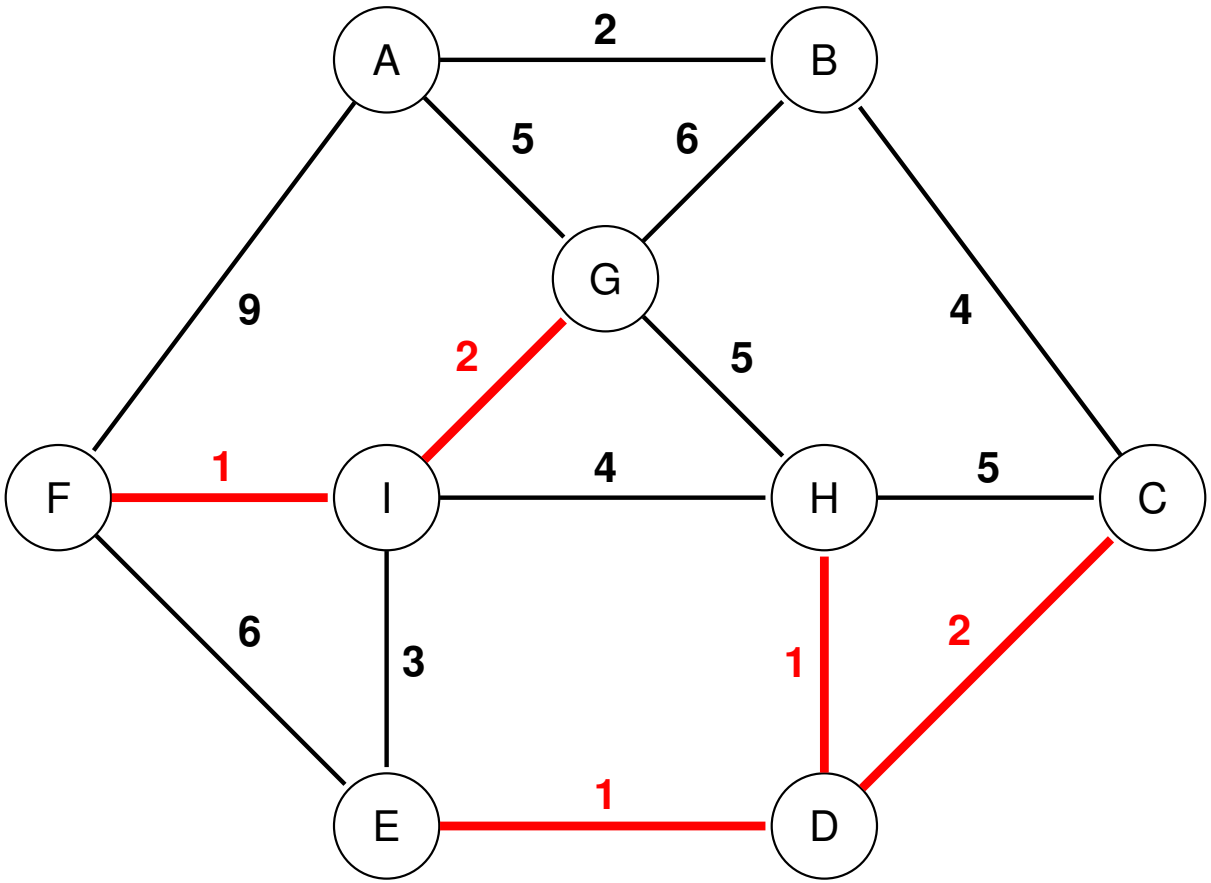
Algorytm Kruskala: Ilustracja



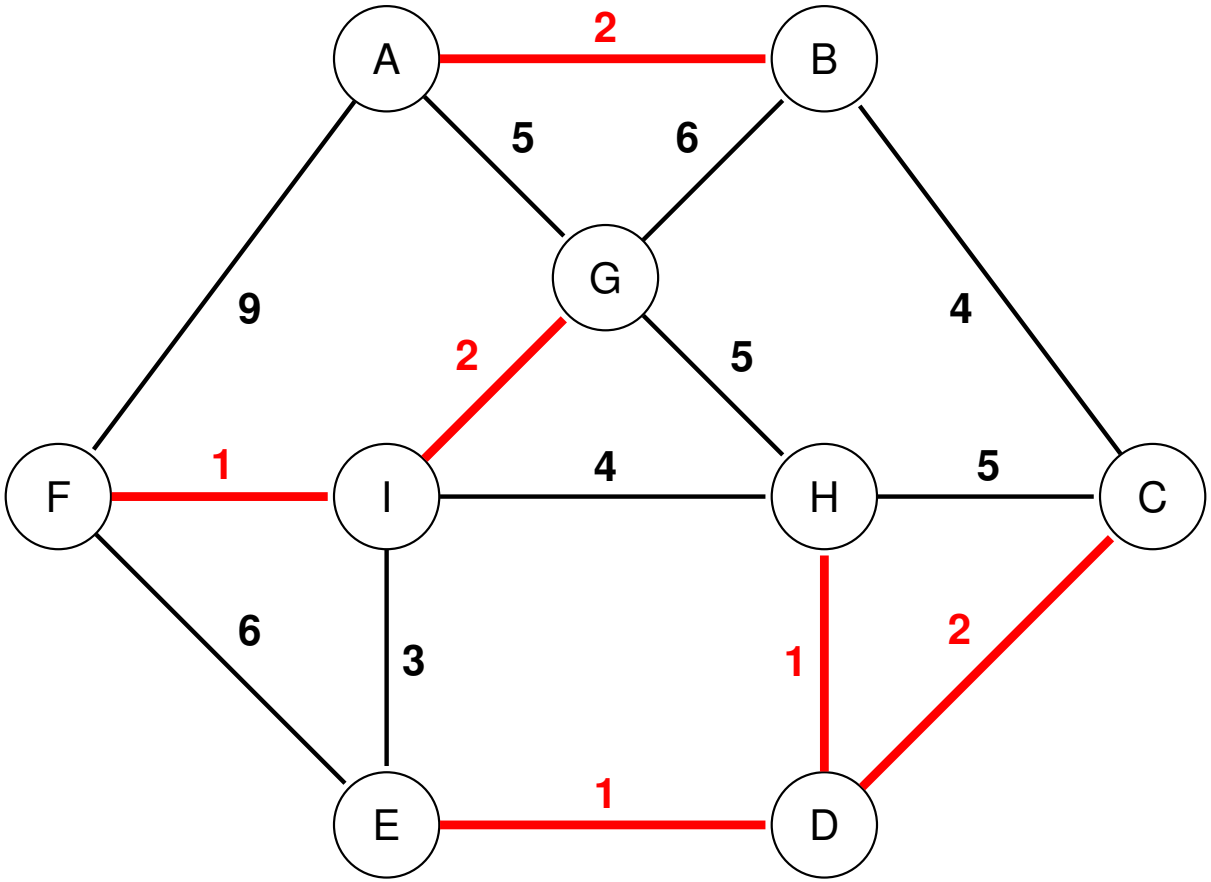
Algorytm Kruskala: Ilustracja



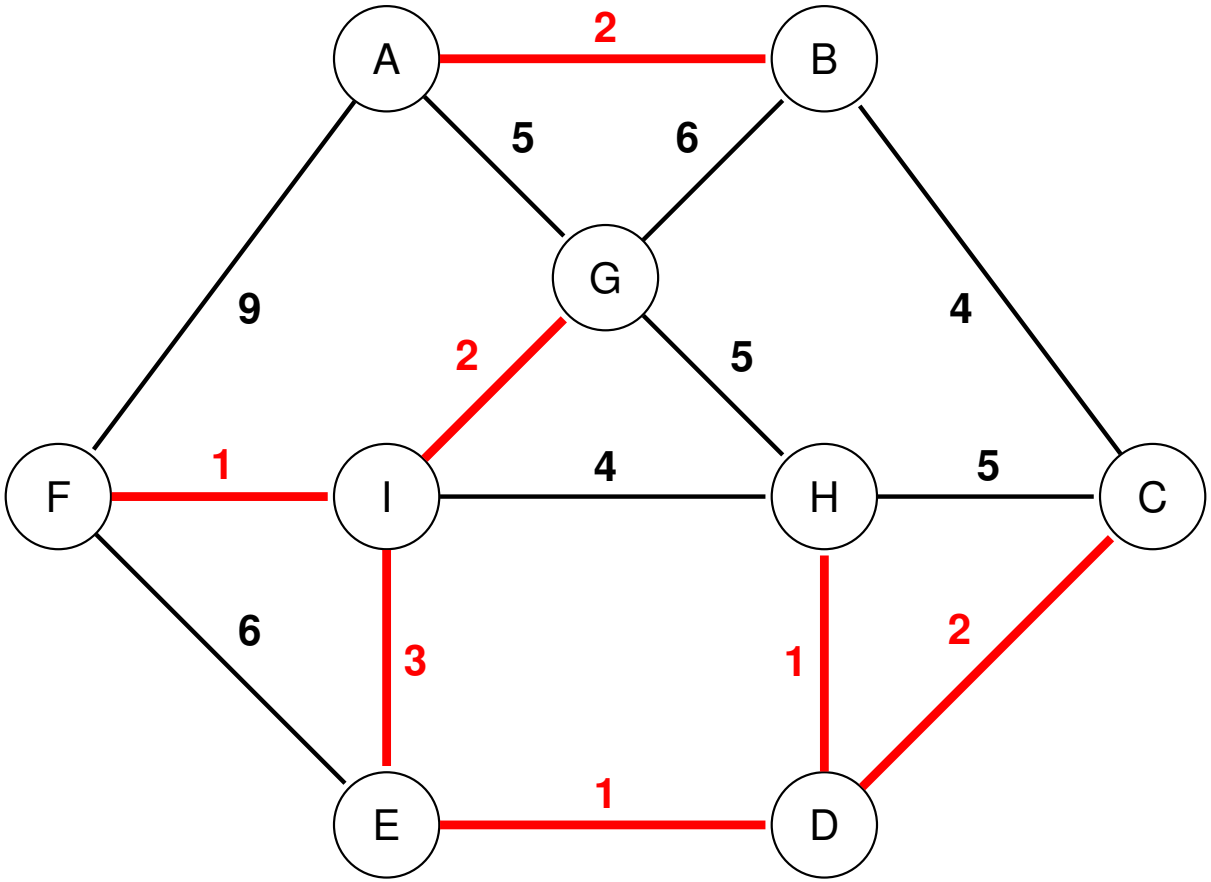
Algorytm Kruskala: Ilustracja



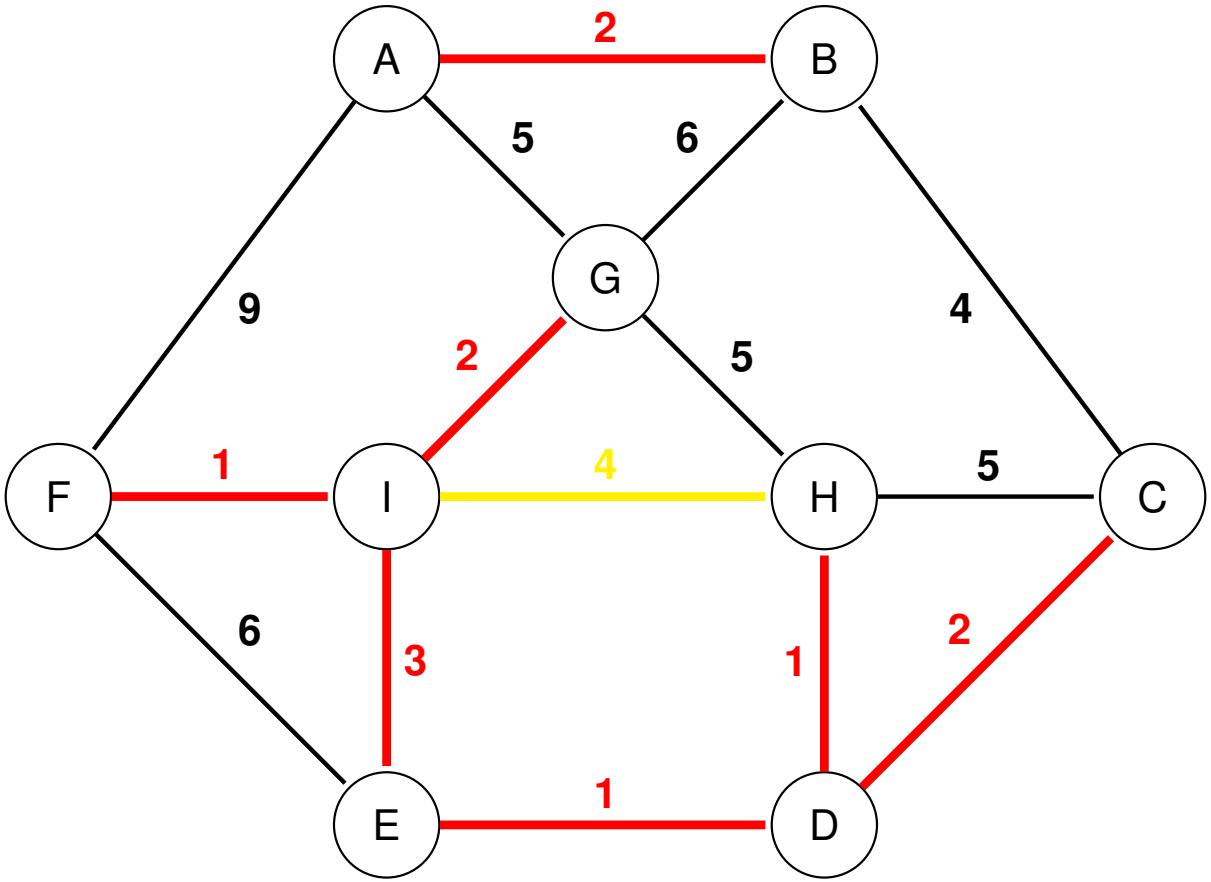
Algorytm Kruskala: Ilustracja



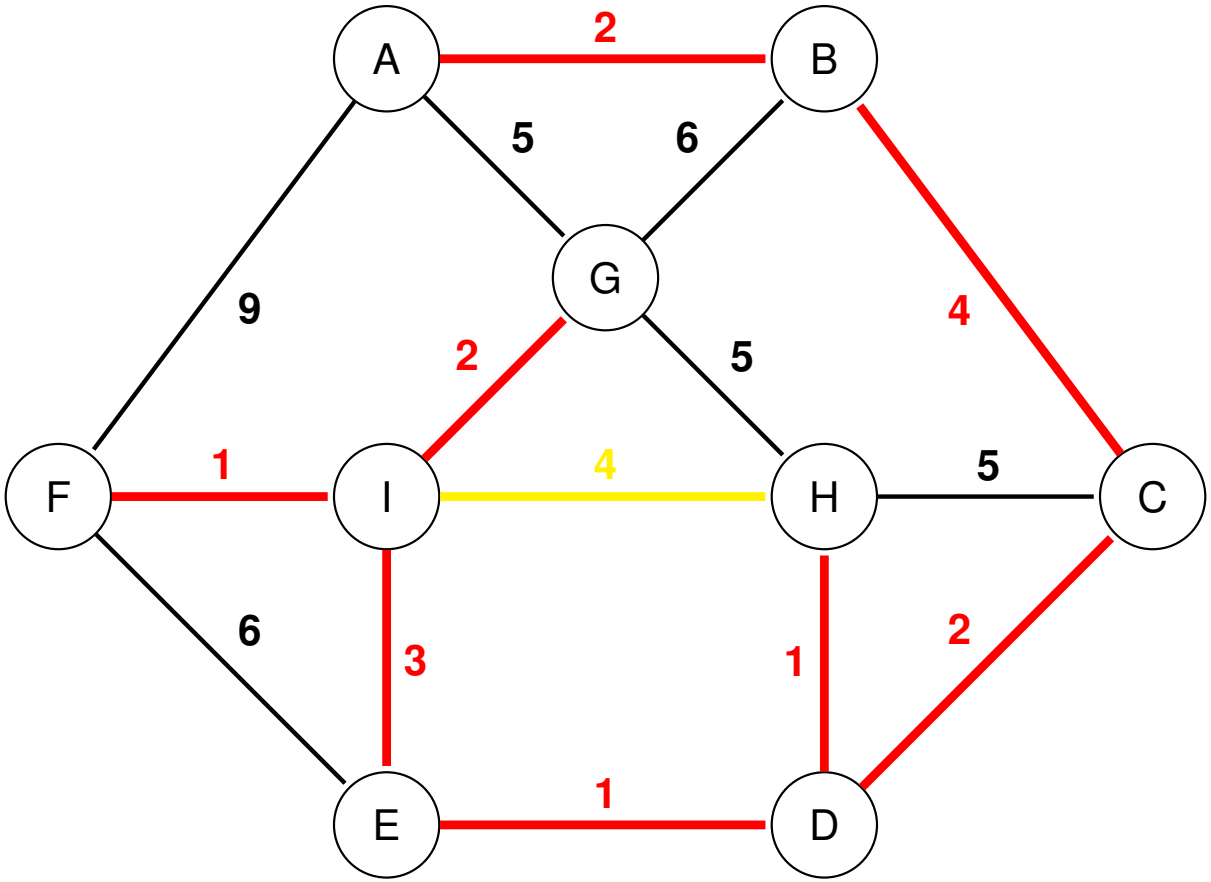
Algorytm Kruskala: Ilustracja



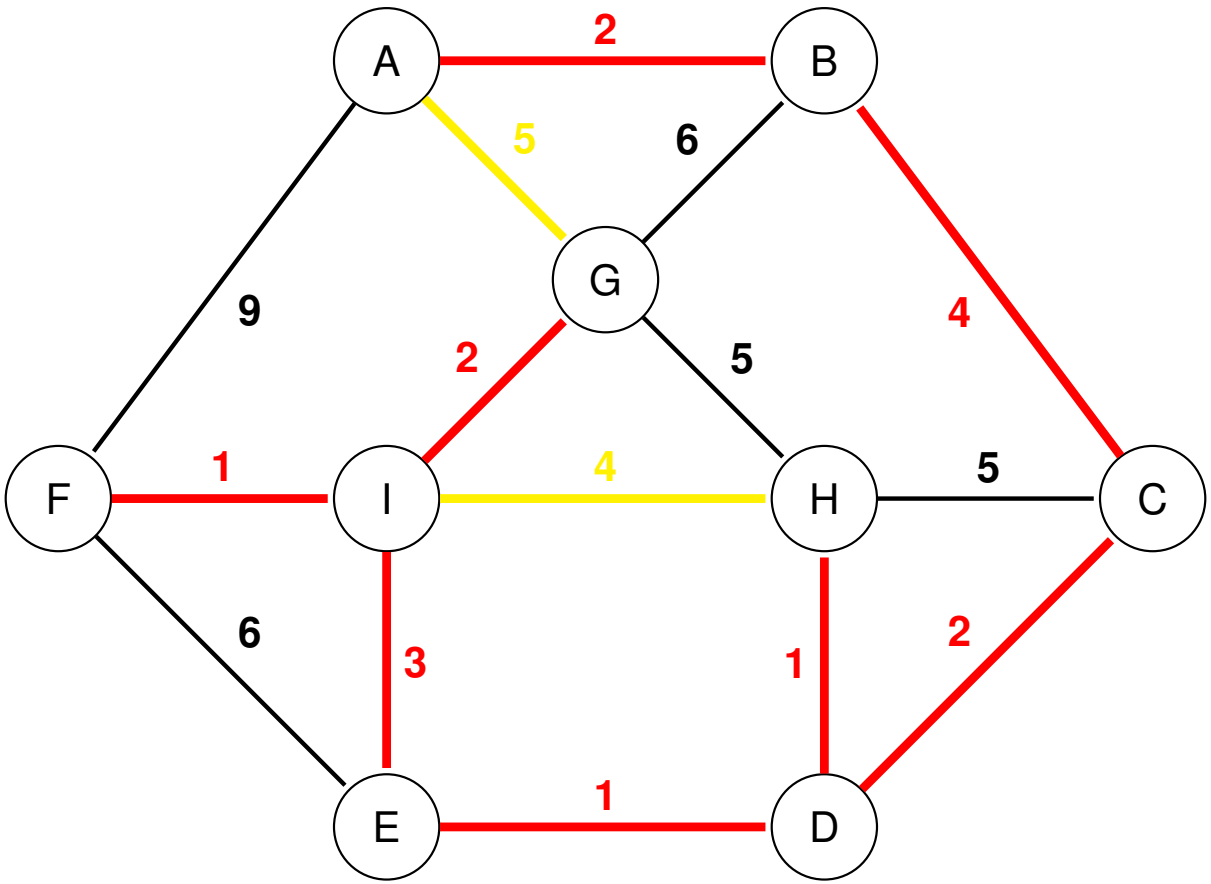
Algorytm Kruskala: Ilustracja



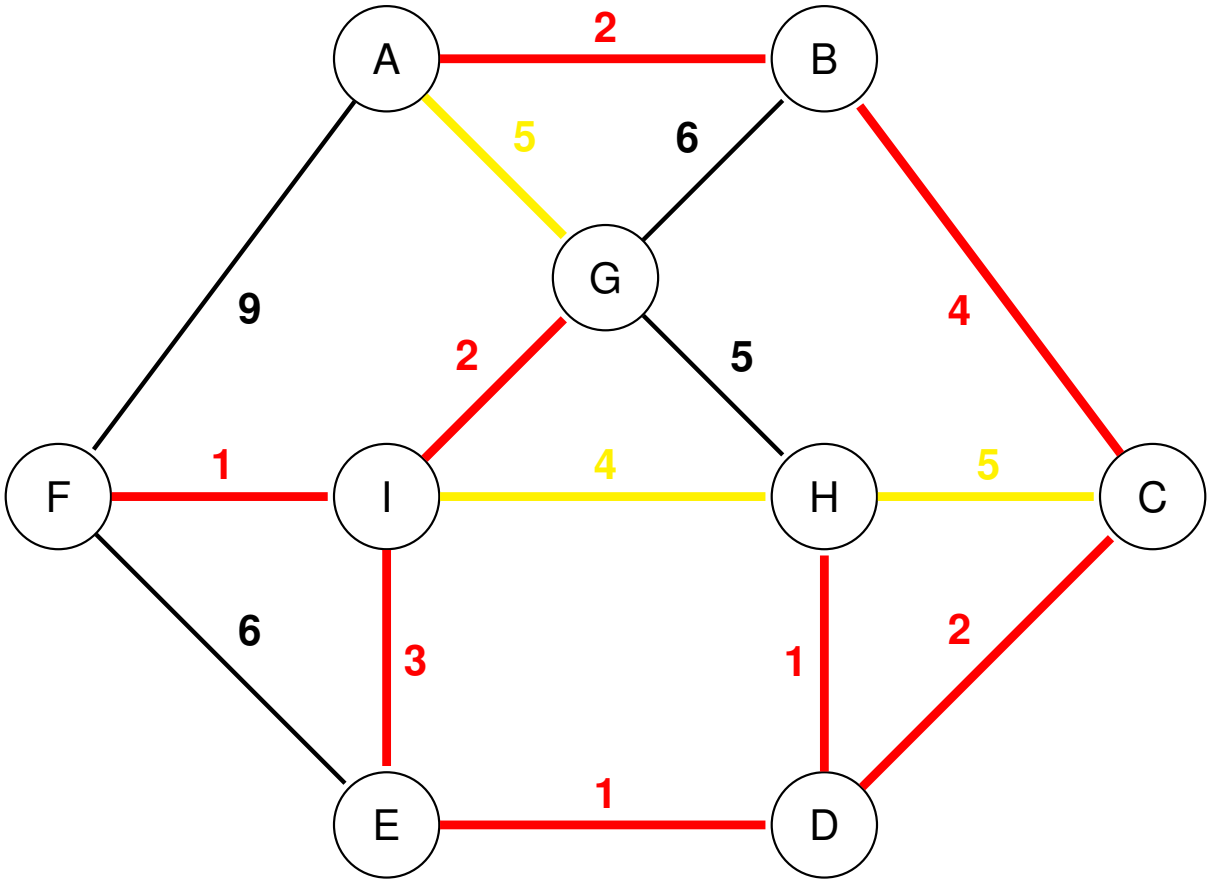
Algorytm Kruskala: Ilustracja



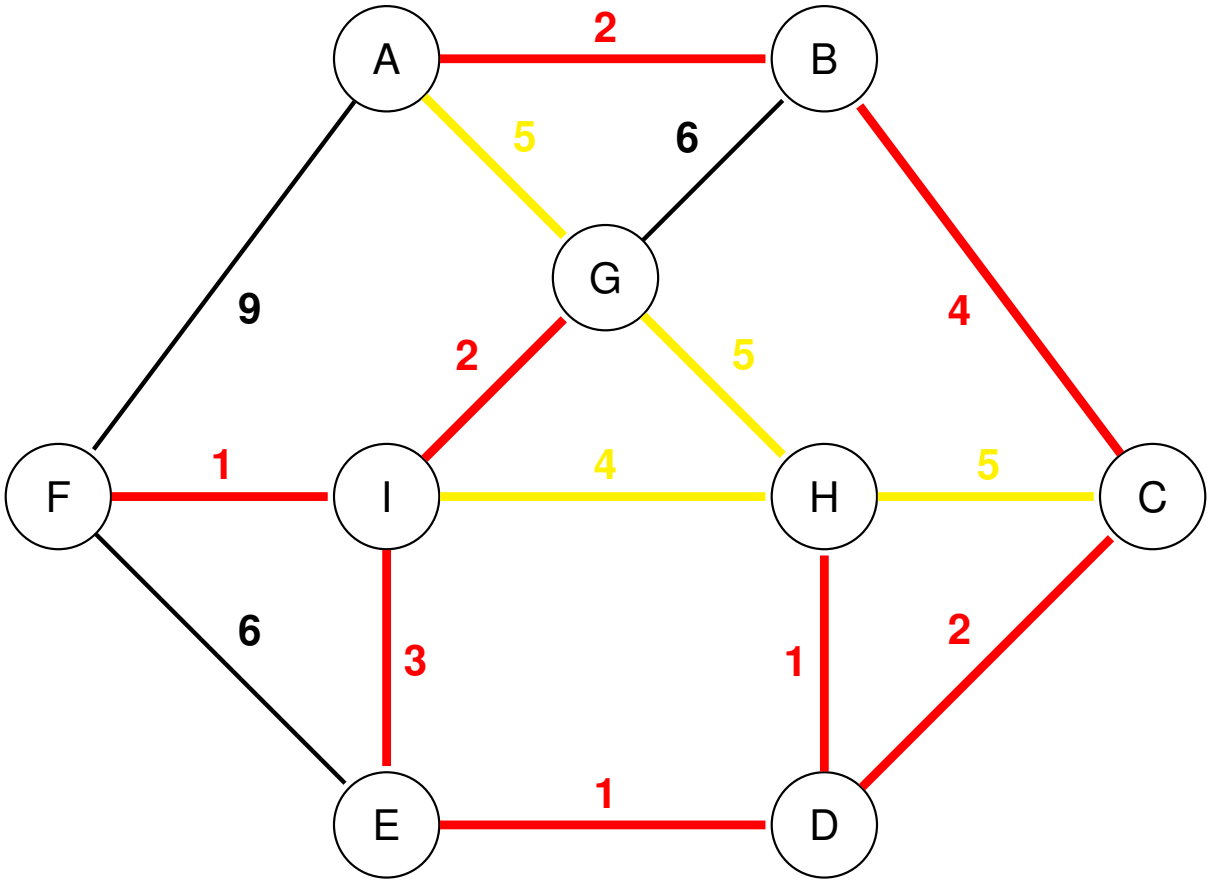
Algorytm Kruskala: Ilustracja



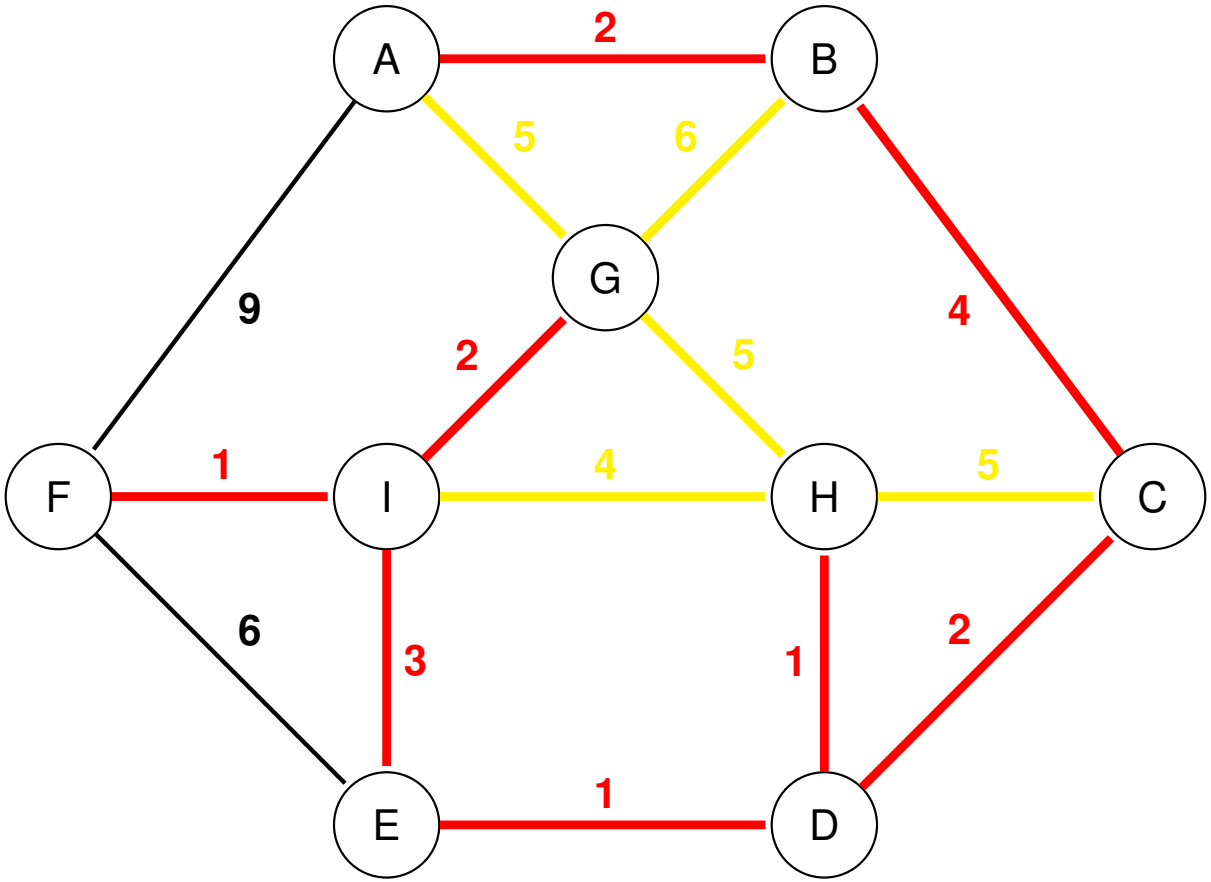
Algorytm Kruskala: Ilustracja



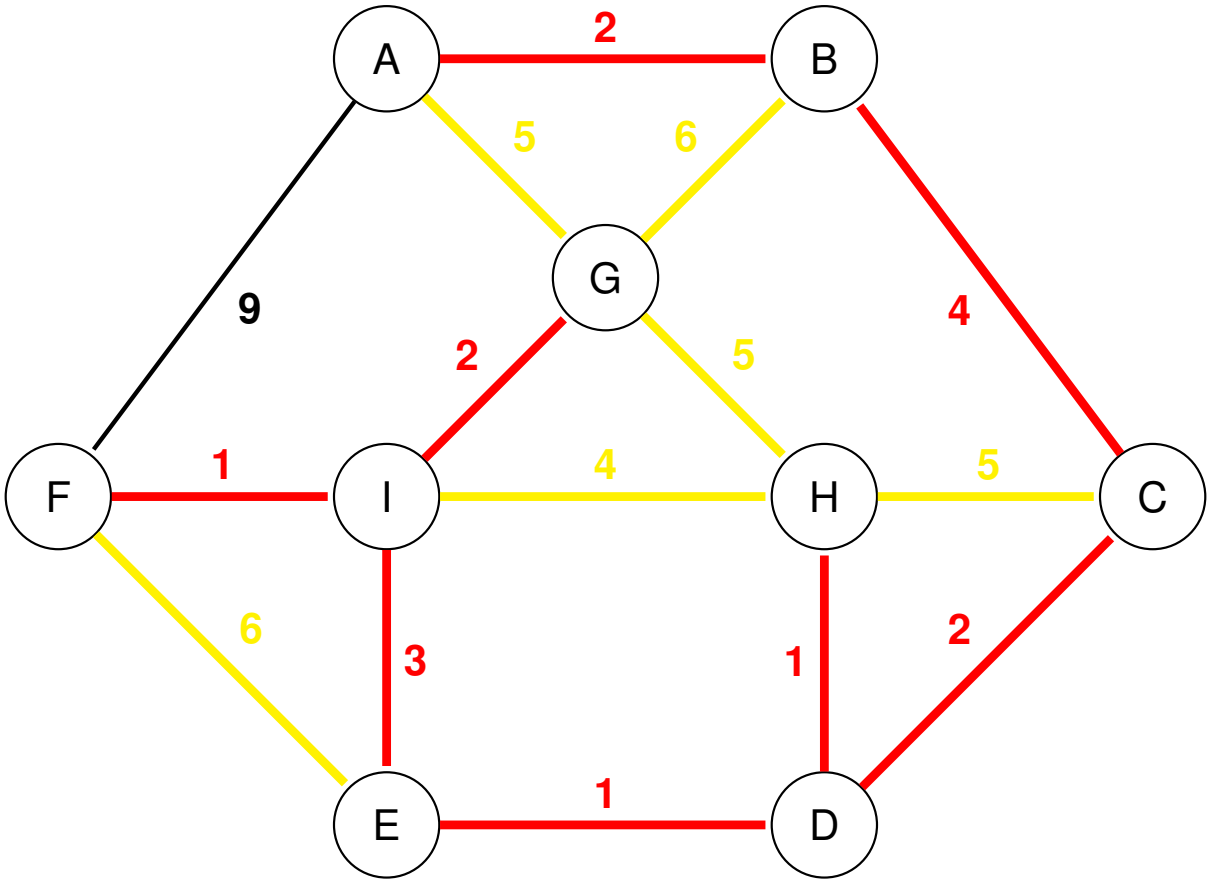
Algorytm Kruskala: Ilustracja



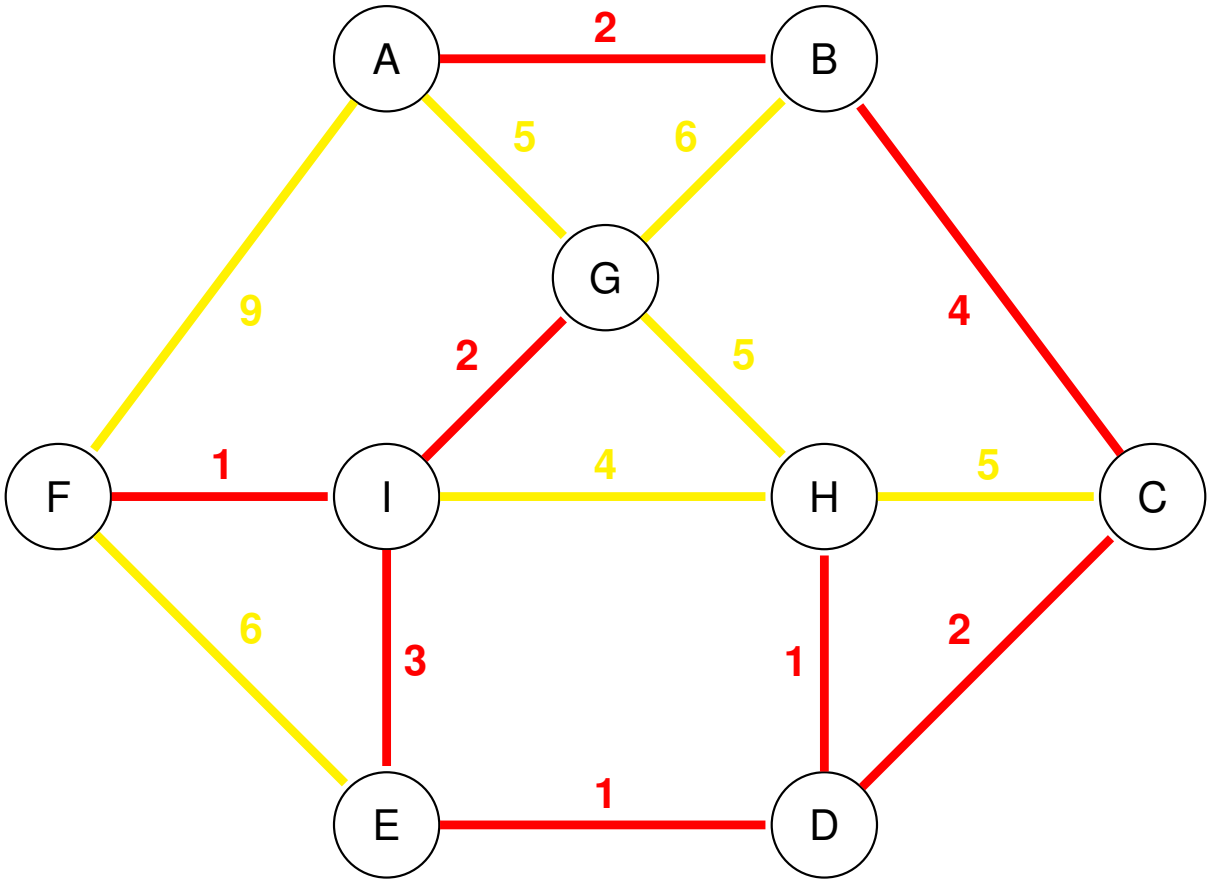
Algorytm Kruskala: Ilustracja



Algorytm Kruskala: Ilustracja



Algorytm Kruskala: Ilustracja



Algorytm Kruskala: Implementacja

EdgesPQ: kolejka priorytetowa dla krawędzi grafu
VertSet: struktura zbiorów rozłącznych dla wierzchołków
EdgeStack: struktura do zbierania krawędzi min. drzewa rozpin.

Kruskal($\mathcal{G}$)

```
KruskPQ.construct(ℰ)
VertSet := {{v} | v ∈ 𝒱}

while !empty(KruskPQ)
    e := MIN(KruskPQ)
    DELMIN(KruskPQ)

    A = VertSet.Find(e.leftEnd)
    B = VertSet.Find(e.rightEnd)

    if A != B then
        VertSet.Union(A,B)
        EdgeStack.PUSH(e)
    end if
end while
```

Algorytm Kruskala: Uzupełnienia

- Złożoność **O**(**ElogE**) = **O**(**ElogV**): dominuje wykonanie operacji usunięcia elementu z kolejki **EdgesPQ**, aż do opróżnienia; (A przy wykorzystaniu sortowania w czasie liniowym można uzyskać czas prawie liniowy!)

Algorytm Kruskala: Uzupełnienia

- Złożoność **O**(**ElogE**) $=$ **O**(**ElogV**): dominuje wykonanie operacji usunięcia elementu z kolejki **EdgesPQ**, aż do opróżnienia; (A przy wykorzystaniu sortowania w czasie liniowym można uzyskać czas prawie liniowy!)
- Dowód poprawności jest dość prosty: wystarczy wykazać, że jeśli las **F** przed scaleniem przy użyciu krawędzi **e** składa się z poddrzew $\mathrm{MST}(\mathcal{G})$, to po scaleniu też tak będzie

Algorytm Kruskala: Uzupełnienia

- Złożoność **O**(**ElogE**) = **O**(**ElogV**): dominuje wykonanie operacji usunięcia elementu z kolejki **EdgesPQ**, aż do opróżnienia; (A przy wykorzystaniu sortowania w czasie liniowym można uzyskać czas prawie liniowy!)
- Dowód poprawności jest dość prosty: wystarczy wykazać, że jeśli las **F** przed scaleniem przy użyciu krawędzi **e** składa się z poddrzew $\mathrm{MST}(\mathcal{G})$, to po scaleniu też tak będzie

  (czyli scalająca krawędź **e** należy zawsze do $\mathrm{MST}(\mathcal{G})$)

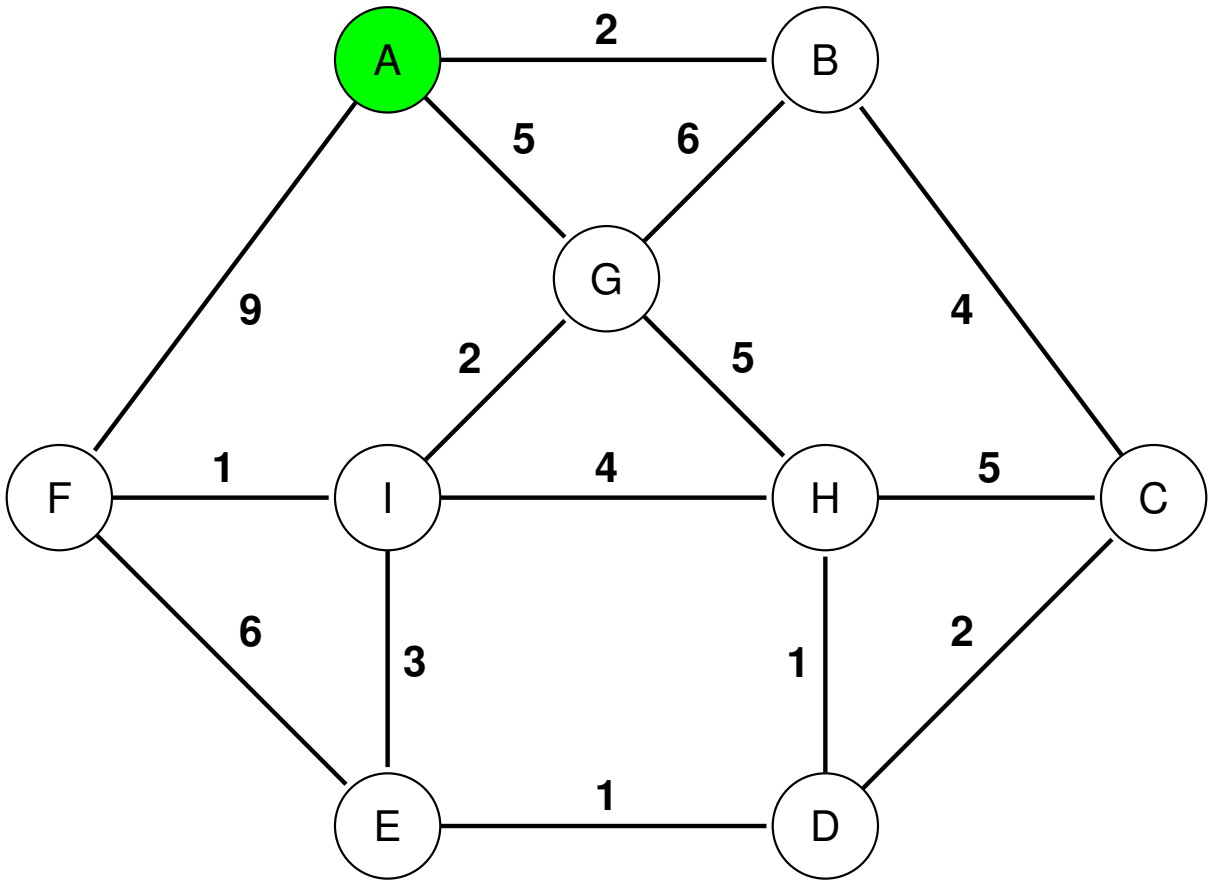
Podstawowe Definicje: Drzewa Najkrótszych Ścieżek

Drzewo Najkrótszych Ścieżek

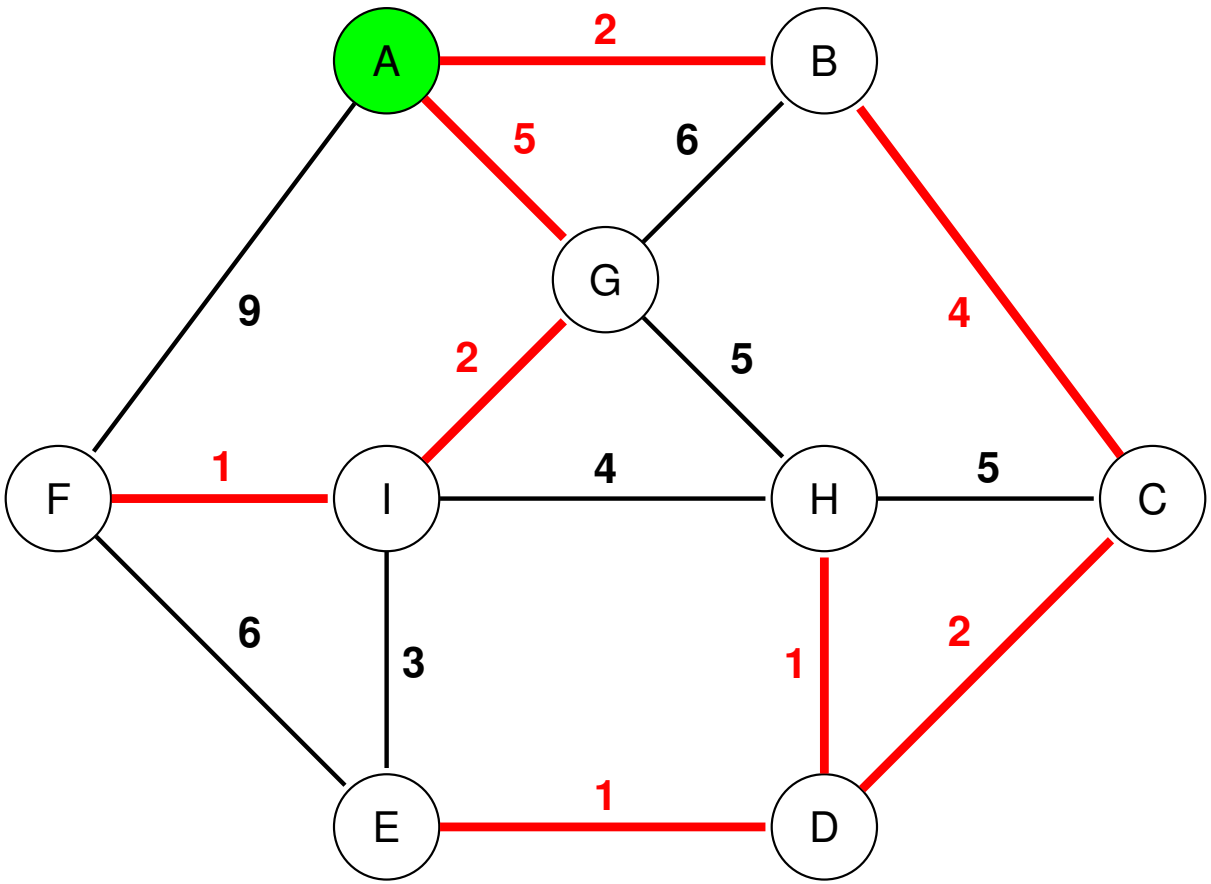
Niech $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ będzie grafem niezorientowanym z funkcją kosztu $c\colon \mathcal{E} \to \mathbb{R}_+$. Niech **src** $\in \mathcal{V}$ będzie **wierzchołkiem źródłowym**.

Drzewo najkrótszych ścieżek $\mathrm{SPT}(\mathcal{G}, \textbf{src})$ ze źródła **src** to drzewo rozpinające $\mathcal{G}$ w którym ścieżka pomiędzy **src** a dowolnym wierzchołkiem **v** jest najkrótszą ścieżką pomiędzy **src** a **v** w $\mathcal{G}$.

Podstawowe Definicje: Drzewa Najkrótszych Ścieżek



Podstawowe Definicje: Drzewa Najkrótszych Ścieżek



Podstawowe Definicje: Drzewa Najkrótszych Ścieżek

Różne zastosowania:

- Routing w sieciach komputerowych (OSPF, sieci z pełną informacją o strukturze)
- Global Positioning System
- Nawigacja w robotyce
- I wiele innych...

Algorytm Dijkstry

Algorytm Dijkstry: Idea

Wykorzystamy następujące abstrakcyjne operacje.

Algorytm Dijkstry

Algorytm Dijkstry: Idea

Wykorzystamy następujące abstrakcyjne operacje.

- Pytanie wierzchołka **v** o:

Algorytm Dijkstry

Algorytm Dijkstry: Idea

Wykorzystamy następujące abstrakcyjne operacje.

- Pytanie wierzchołka **v** o:
  - **dist**(**v**) – najkrótszą znaną już ścieżkę ze źródła **src** do **v** (ustawione na ∞, jeśli nie znamy żadnej ścieżki)

Algorytm Dijkstry

Algorytm Dijkstry: Idea

Wykorzystamy następujące abstrakcyjne operacje.

- Pytanie wierzchołka **v** o:
  - **dist**(**v**) – najkrótszą znaną już ścieżkę ze źródła **src** do **v** (ustawione na ∞, jeśli nie znamy żadnej ścieżki)
  - **prev**(**v**) – poprzednik **v** na tej ścieżce (ustawione na NULL, w przypadku j.w.)

Algorytm Dijkstry

Algorytm Dijkstry: Idea

Wykorzystamy następujące abstrakcyjne operacje.

- Pytanie wierzchołka **v** o:
  - **dist**(**v**) – najkrótszą znaną już ścieżkę ze źródła **src** do **v** (ustawione na ∞, jeśli nie znamy żadnej ścieżki)
  - **prev**(**v**) – poprzednik **v** na tej ścieżce (ustawione na NULL, w przypadku j.w.)

oraz następujący zbiór:

Algorytm Dijkstry

Algorytm Dijkstry: Idea

Wykorzystamy następujące abstrakcyjne operacje.

- Pytanie wierzchołka **v** o:
  - **dist**(**v**) – najkrótszą znaną już ścieżkę ze źródła **src** do **v** (ustawione na ∞, jeśli nie znamy żadnej ścieżki)
  - **prev**(**v**) – poprzednik **v** na tej ścieżce (ustawione na NULL, w przypadku j.w.)

oraz następujący zbiór:

- **unVisited**: wierzchołki, których jeszcze nie odwiedziliśmy.

Algorytm Dijkstry

Algorytm Dijkstry: Idea

1. Inicjalizacja: wstaw wszystkie wierzchołki do **unVisited**, ustaw **dist**(**src**) = 0

Algorytm Dijkstry

Algorytm Dijkstry: Idea

1. Inicjalizacja: wstaw wszystkie wierzchołki do **unVisited**, ustaw **dist**(**src**) = 0
2. **dopóki unVisited** jest niepusty:

Algorytm Dijkstry

Algorytm Dijkstry: Idea

1. Inicjalizacja: wstaw wszystkie wierzchołki do **unVisited**, ustaw **dist**(**src**) = 0
2. **dopóki unVisited** jest niepusty:
3. usuń z **unVisited** wierzchołek **v** o najmniejszym **dist**(**v**) *(czyli najmniejszej odległości od źródła)*

Algorytm Dijkstry

Algorytm Dijkstry: Idea

1. Inicjalizacja: wstaw wszystkie wierzchołki do **unVisited**, ustaw **dist**(**src**) = 0
2. **dopóki unVisited** jest niepusty:
3. usuń z **unVisited** wierzchołek **v** o najmniejszym **dist**(**v**) *(czyli najmniejszej odległości od źródła)*
4. **dla każdego sąsiada w wierzchołka v z unVisited:**

Algorytm Dijkstry

Algorytm Dijkstry: Idea

1. Inicjalizacja: wstaw wszystkie wierzchołki do **unVisited**, ustaw **dist**(**src**) = 0
2. **dopóki unVisited** jest niepusty:
3. usuń z **unVisited** wierzchołek **v** o najmniejszym **dist**(**v**) *(czyli najmniejszej odległości od źródła)*
4. **dla każdego sąsiada w wierzchołka v z unVisited:**
5. porównaj **dist**(**w**) z **dist**(**v**) $+ c(\mathbf{v}, \mathbf{w})$

Algorytm Dijkstry

Algorytm Dijkstry: Idea

1. Inicjalizacja: wstaw wszystkie wierzchołki do **unVisited**, ustaw **dist**(**src**) = 0
2. **dopóki unVisited** jest niepusty:
3. usuń z **unVisited** wierzchołek **v** o najmniejszym **dist**(**v**) *(czyli najmniejszej odległości od źródła)*
4. **dla każdego sąsiada w wierzchołka v z unVisited:**
5. porównaj **dist**(**w**) z **dist**(**v**) $+ c(\mathbf{v}, \mathbf{w})$
6. **jeśli większe, to:** **prev**(**w**) $:=$ **v**
   dist(**w**) $:=$ **dist**(**v**) $+ c(\mathbf{v}, \mathbf{w})$
   (znaleźliśmy krótszą drogę do **w** *przez* **v***)*

Algorytm Dijkstry

Algorytm Dijkstry: Idea

1. Inicjalizacja: wstaw wszystkie wierzchołki do **unVisited**, ustaw **dist**(**src**) = 0
2. **dopóki unVisited** jest niepusty:
3. usuń z **unVisited** wierzchołek **v** o najmniejszym **dist**(**v**) *(czyli najmniejszej odległości od źródła)*
4. **dla każdego sąsiada w wierzchołka v z unVisited:**
5. porównaj **dist**(**w**) z **dist**(**v**) $+ c(\mathbf{v}, \mathbf{w})$
6. **jeśli większe, to:** **prev**(**w**) $:=$ **v**
   dist(**w**) $:=$ **dist**(**v**) $+ c(\mathbf{v}, \mathbf{w})$
   (znaleźliśmy krótszą drogę do w przez v)

Jakiej struktury danych użyć dla **unVisited**?

Algorytm Dijkstry

Struktura Danych dla **unVisited**

Wierzchołki **v** $\in \mathcal{V}$ umieścimy w **kopcu** z priorytetem **dist**(**v**).

Algorytm Dijkstry

Struktura Danych dla **unVisited**

Wierzchołki **v** $\in \mathcal{V}$ umieścimy w **kopcu** z priorytetem **dist**(**v**).

Będą nam potrzebne dodatkowe operacje:

Algorytm Dijkstry

Struktura Danych dla **unVisited**

Wierzchołki $\mathbf{v} \in \mathcal{V}$ umieścimy w **kopcu** z priorytetem **dist**(**v**).

Będą nam potrzebne dodatkowe operacje:

- **Member**(**v**): wskaźnik do elementu **v** w kopcu albo NULL, gdy go tam nie ma

Algorytm Dijkstry

Struktura Danych dla **unVisited**

Wierzchołki $\mathbf{v} \in \mathcal{V}$ umieścimy w **kopcu** z priorytetem **dist**(**v**).

Będą nam potrzebne dodatkowe operacje:

- **Member**(**v**): wskaźnik do elementu **v** w kopcu albo NULL, gdy go tam nie ma
- **DecreaseKey**(**v**, **d**): zmniejszenie wartości priorytetu **v** na **d** i rekonstrukcja kopca

Algorytm Dijkstry

Struktura Danych dla **unVisited**

Wierzchołki $\mathbf{v} \in \mathcal{V}$ umieścimy w **kopcu** z priorytetem **dist**(**v**).

Będą nam potrzebne dodatkowe operacje:

- **Member**(**v**): wskaźnik do elementu **v** w kopcu albo NULL, gdy go tam nie ma
- **DecreaseKey**(**v**, **d**): zmniejszenie wartości priorytetu **v** na **d** i rekonstrukcja kopca

Member$(\cdot)$ - tablica, uaktualniana tylko po usunięciu elementu

Algorytm Dijkstry

Struktura Danych dla **unVisited**

Wierzchołki $\mathbf{v} \in \mathcal{V}$ umieścimy w **kopcu** z priorytetem **dist**(**v**).

Będą nam potrzebne dodatkowe operacje:

- **Member**(**v**): wskaźnik do elementu **v** w kopcu albo NULL, gdy go tam nie ma
- **DecreaseKey**(**v**, **d**): zmniejszenie wartości priorytetu **v** na **d** i rekonstrukcja kopca

Member$(\cdot)$ - tablica, uaktualniana tylko po usunięciu elementu
DecreaseKey$(\cdot, \cdot)$ - jak to zaimplementować?

Algorytm Dijkstry: Uzupełnienia

- Indukcyjny dowód poprawności algorytmu oparty o niezmiennik:
  - jeśli **v** nie należy do **unVisited** to **dist**(**v**) jest długością najkrótszej drogi z **src** do **v**, zaś **prev**(**v**) jego poprzednikiem na tej drodze
  - jeśli **v** należy do **unVisited** **dist**(**v**) jest długością najkrótszej drogi z **src** do **v** przechodzącej po wierzchołkach spoza **unVisited**, zaś **prev**(**v**) jego poprzednikiem na tej drodze
- Złożoność: **O**(**ElogV**), gdzie **E** $= |\mathcal{E}|$ i **V** $= |\mathcal{V}|$
- Pseudokod łatwy do napisania, implementacja też
- Popularne warianty z heurystyką **A***

Źródła

- HEX Game by Jean-Luc W - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6206668