Algorytmy Grafowe

dr hab. Bożena Woźna-Szcześniak, prof. UJD

Uniwersytet Jana Długosza w Częstochowie

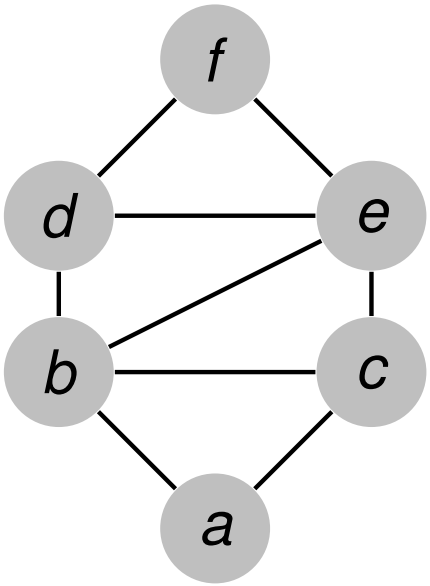
b.wozna@ujd.edu.pl

Wykład 7 i 8

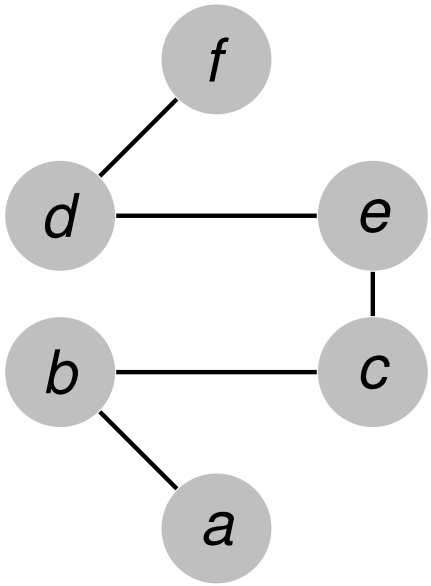
Drzewo rozpinające grafu I

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

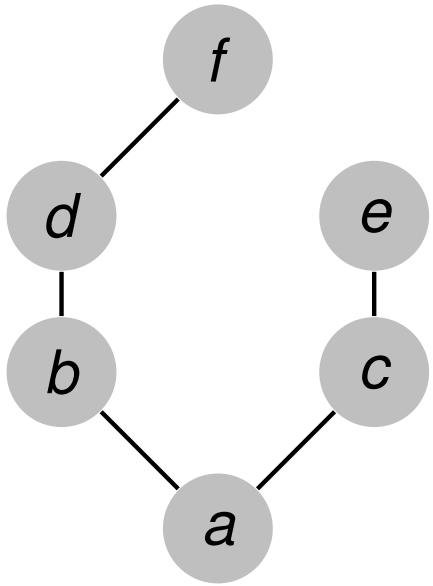
Graf wejściowy



Drzewo rozpinające nr 1



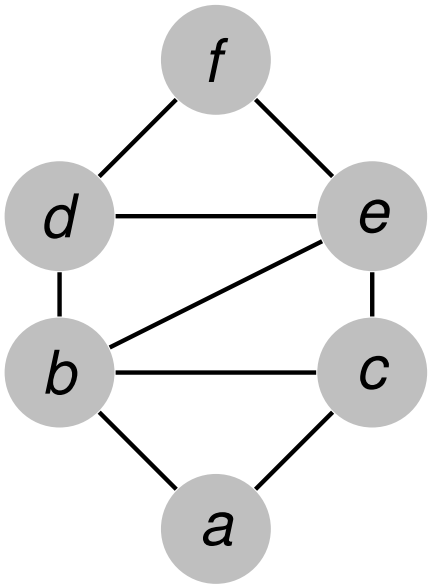
Drzewo rozpinające nr 2



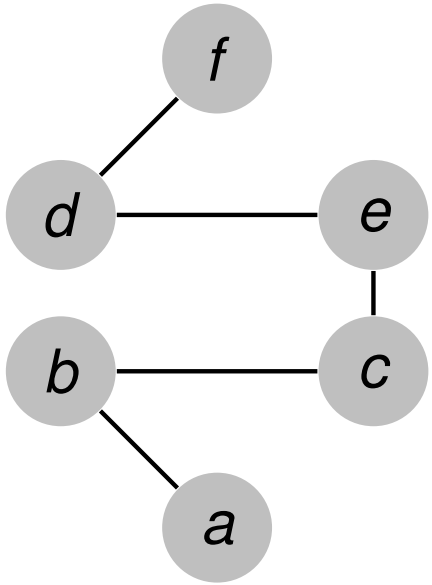
Drzewo rozpinające grafu II

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

Graf wejściowy



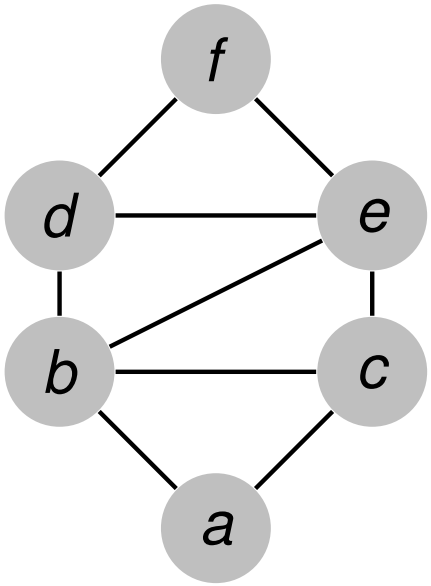
Drzewo rozpinające nr 1.
Wirzchołek początkowy *a*.



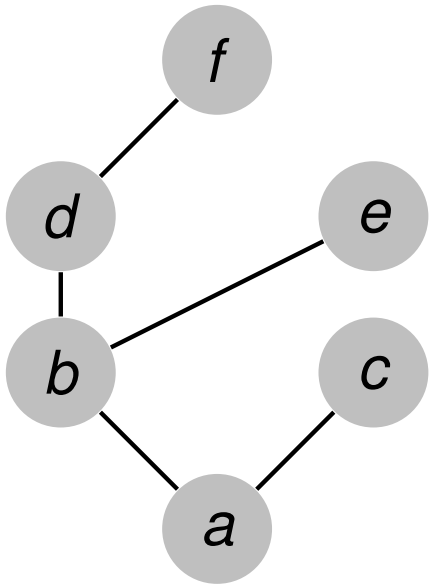
Drzewo rozpinające grafu III

- Drzewo rozpinające można utworzyć przy pomocy algorytmu BFS.

Graf wejściowy



Drzewo rozpinające nr 2.
Wirzchołek początkowy *a*.



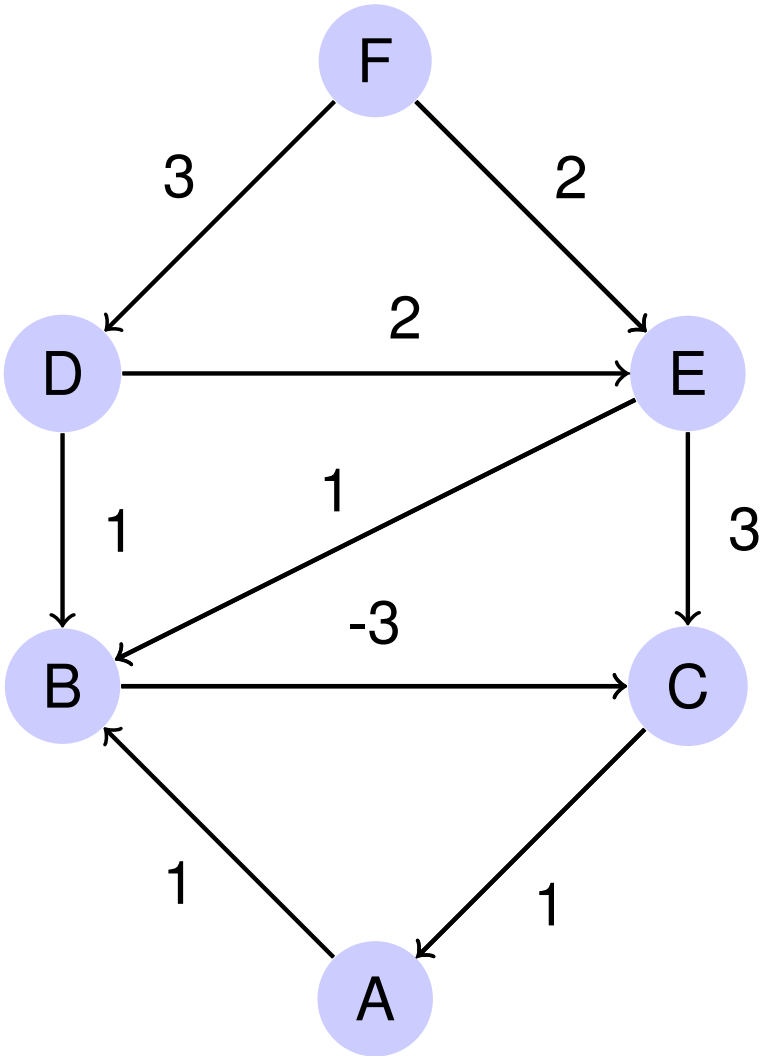
Etykietowany graf skierowany

Definicja

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

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

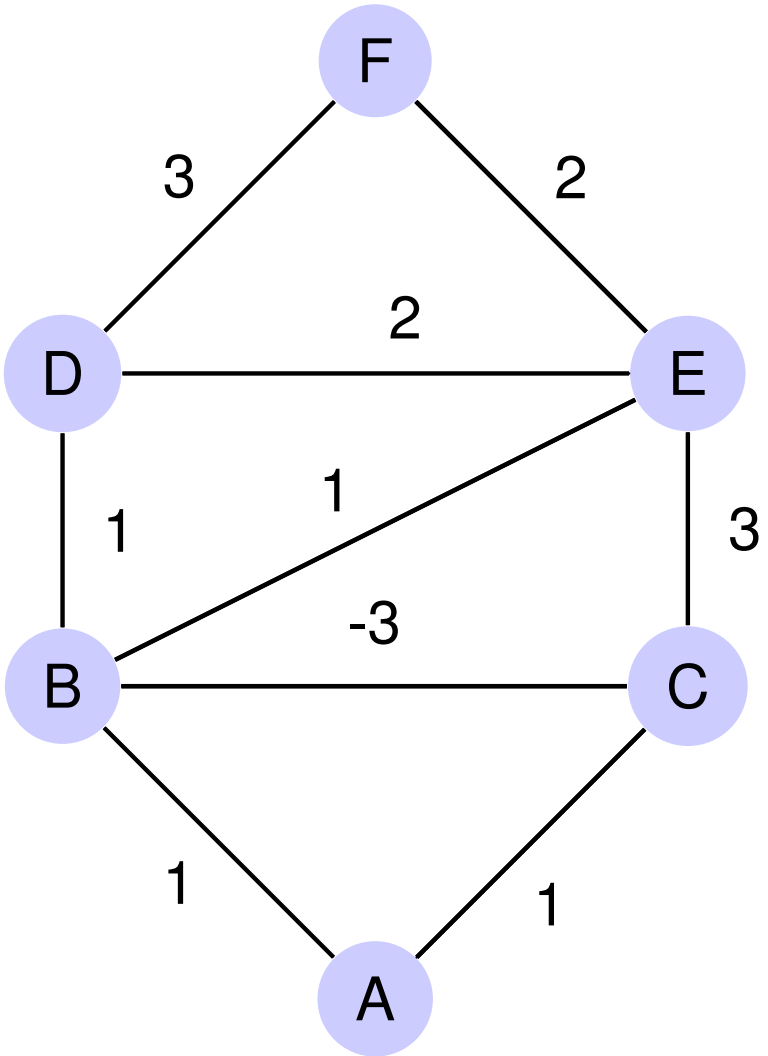
Etykietowany graf nieskierowany

Definicja

Etykietowanym grafem nieskierowanym nazywamy strukturę $G = (V, E, w : E \to R)$ gdzie

- V to zbiór wierzchołków,
- $E \subseteq \{\{u, v\} : u, v \in V\}$ to zbiór par wierzchołków ze zbioru V, zwanych krawędziami.
- $w : E \to R$ 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 rozłącznych**.
- 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 $\mathbb{S} = \{S_1, S_2, \ldots, S_n\}$ rozłącznych zbiorów dynamicznych.
- W takiej strukturze każdy zbiór jest identyfikowany przez **reprezentanta**, którym jest pewnym elementem tego zbioru.
- Załóżmy, 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) - łaczy 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 ∈ V do
2:     MakeSet(v)
3: end for
4: for każda krawędź (u, v) ∈ E do
5:     if FindSet(u) ≠ 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.

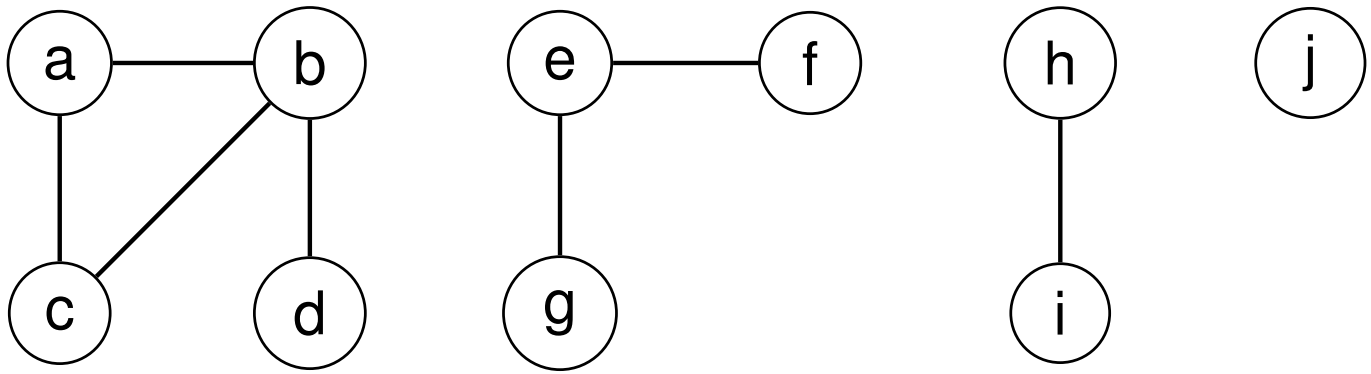
Algorytm SAME-COMPONENT

```
   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

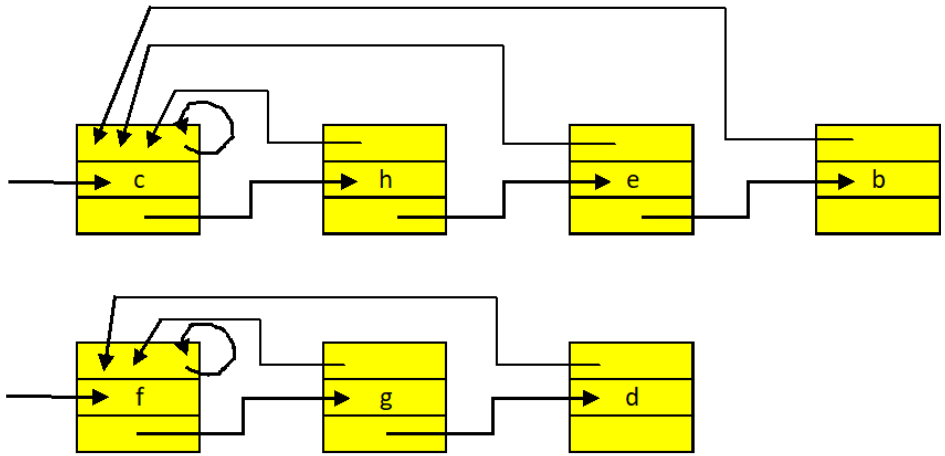
Przetworzone krawędzie	**Rodzina zbiorów rozłącznych**
Zbiory początkowe	$\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\},\{i\},\{j\}$
$\{b,d\}$	$\{a\},\{b,d\},\{c\},\{e\},\{f\},\{g\},\{h\},\{i\},\{j\}$
$\{e,g\}$	$\{a\},\{b,d\},\{c\},\{e,g\},\{f\},\{h\},\{i\},\{j\}$
$\{a,c\}$	$\{a,c\},\{b,d\},\{e,g\},\{f\},\{h\},\{i\},\{j\}$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

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 następy 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 *MakeSet*(x), wystarczy utworzyć nową listę, której jedynym elementem jest x.

Listowa reprezentacja zbiorów rozłącznych III

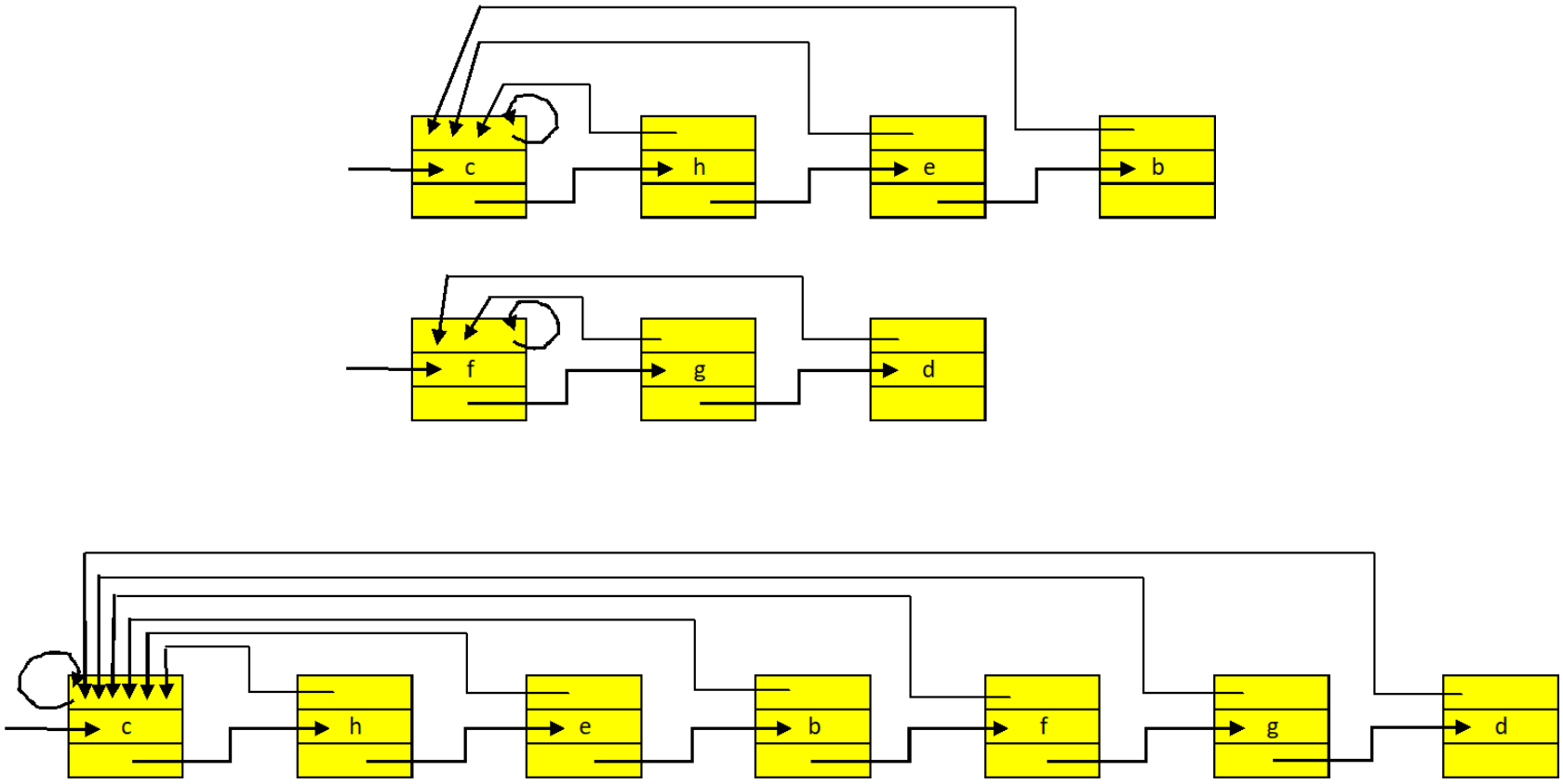
- W reprezentacji listowej, wykonanie procedury *FindSet* wymaga czasu $O(1)$.
  - *FindSet*(x) zwraca wskaźnik od x do reprezentanta zbioru.
- Najprostsza implementacja operacji UNION przy użyciu reprezentacji listy zajmuje znacznie więcej czasu niż MAKE-SET lub FIND-SET.
- Operację *Union*(x, y) wykonujemy 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: *Union*(e, g). Reprezentantem zbioru wynikowego jest c.



Drzewa rozpinające o minimalnej wadze

- Jeżeli mamy do czynienia z grafem z funkcją wagi, 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 wagi $G = (V, E, w : E \mapsto R)$.

Algorytm Kruskala

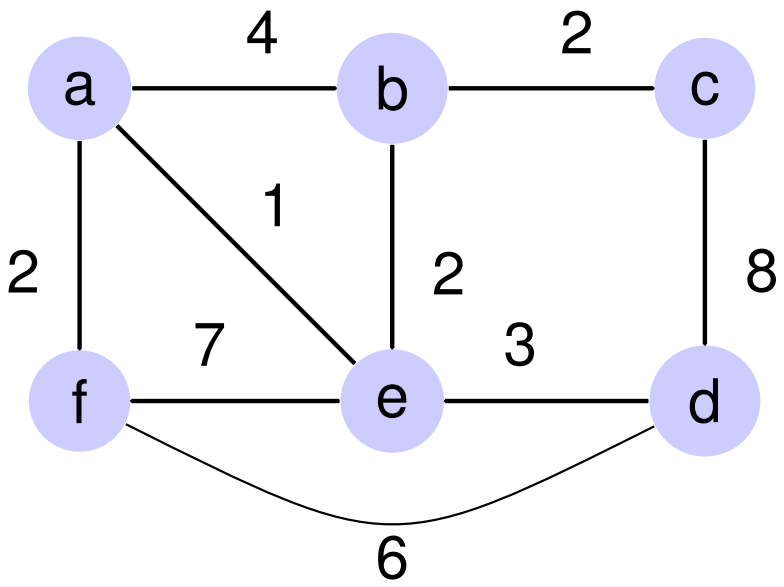
Require: $KRUSKAL(G)$

1: $A = \emptyset$
2: **for** każdy wierzchołek $v \in V[G]$ **do**
3: $\quad Make - Set(v)$
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: $\quad$ **if** $FIND - SET(u) \neq FIND - SET(v)$ **then**
8: $\quad\quad A = A \cup \{(u, v)\}$
9: $\quad\quad Union(u, v)$
10: $\quad$ **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: $O(|E| \cdot log(|V|))$

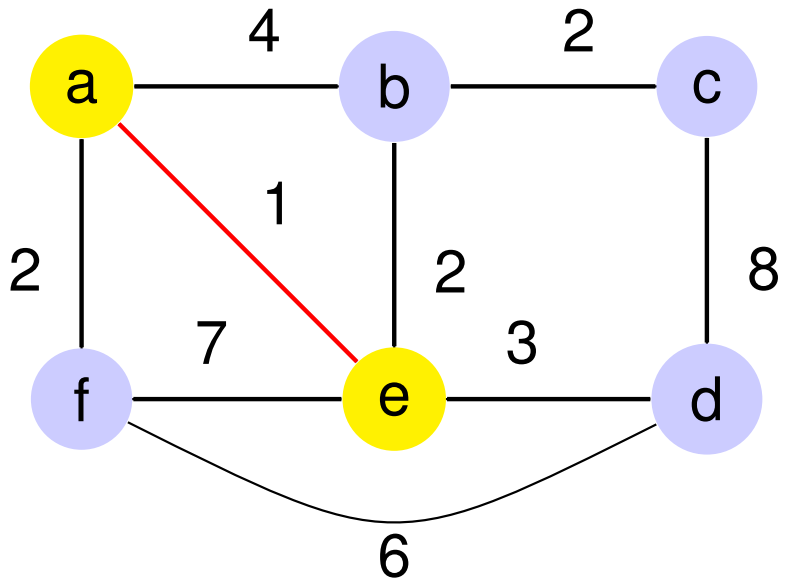
Algorytm Kruskala - Przykład



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

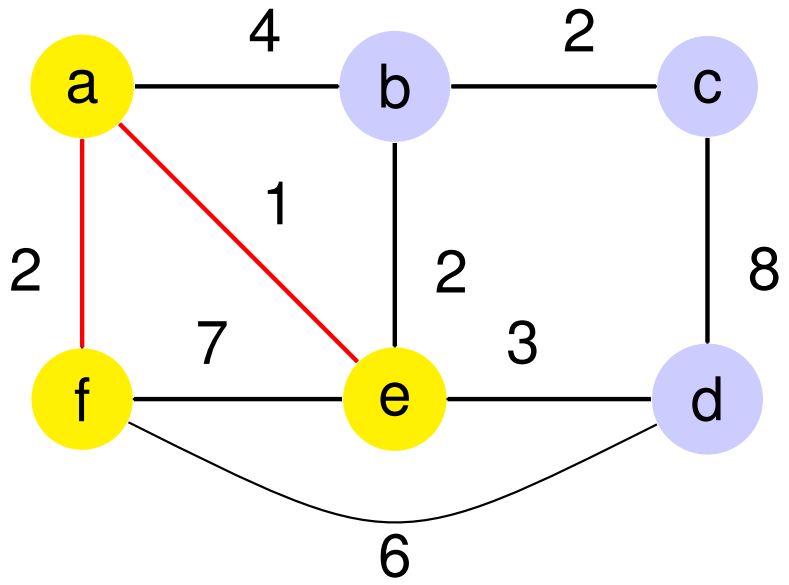
Algorytm Kruskala - Przykład

Krok 1.



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

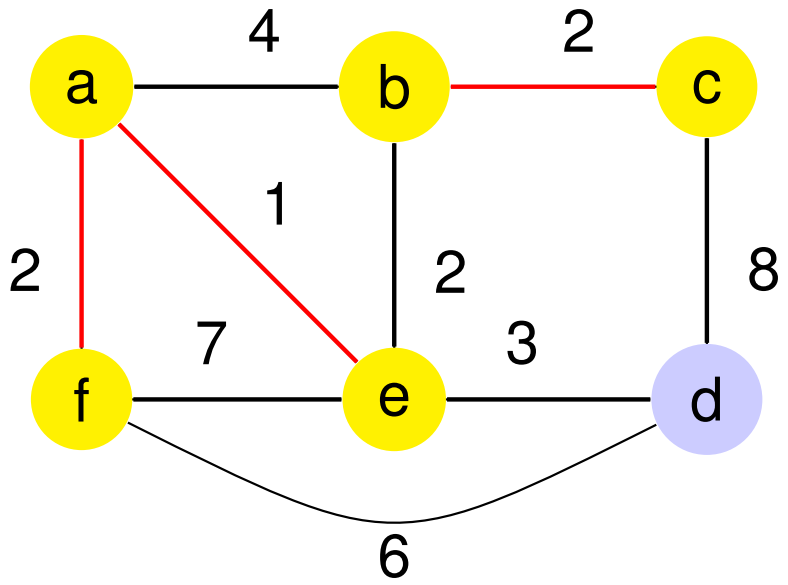
Krok 2.



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

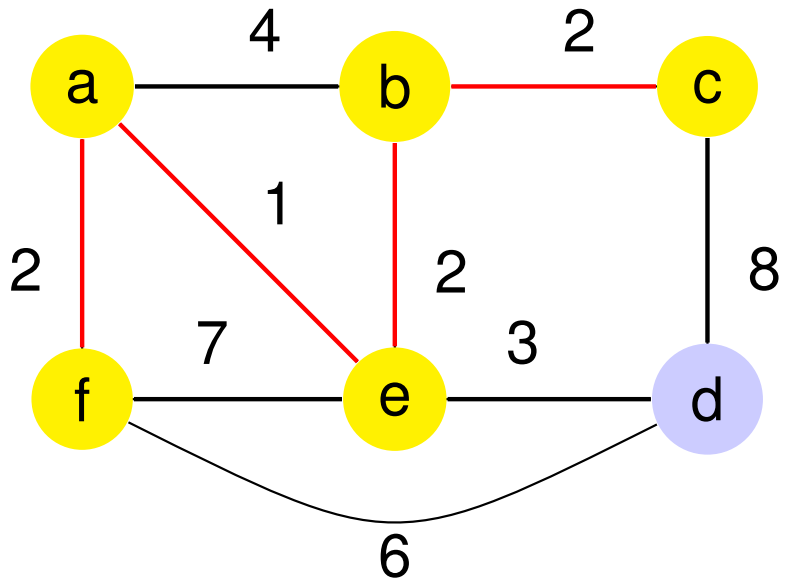
Algorytm Kruskala - Przykład

Krok 3.



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

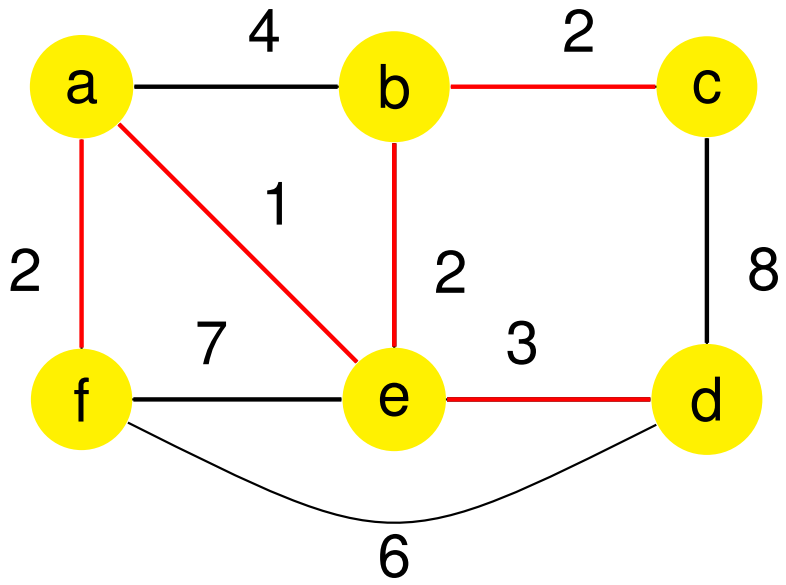
Krok 4 - scalenie.



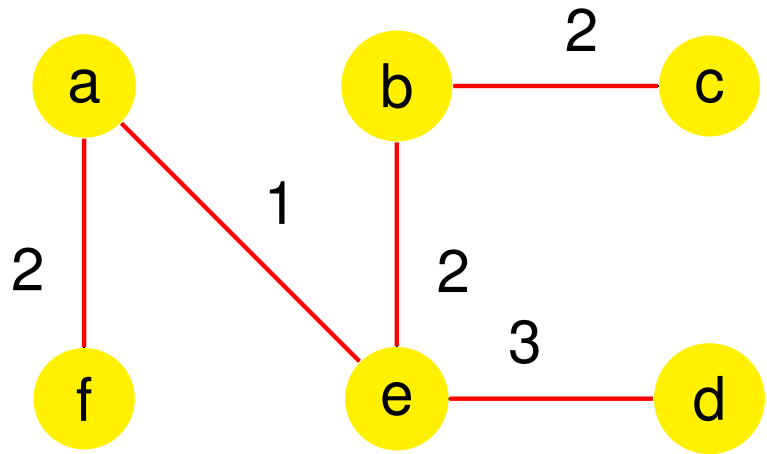
ae=1, af=2, bc=2, be=2, de=3,
ab=4, fd=6, ef=7, cd=8

Algorytm Kruskala - Przykład

Krok 5.

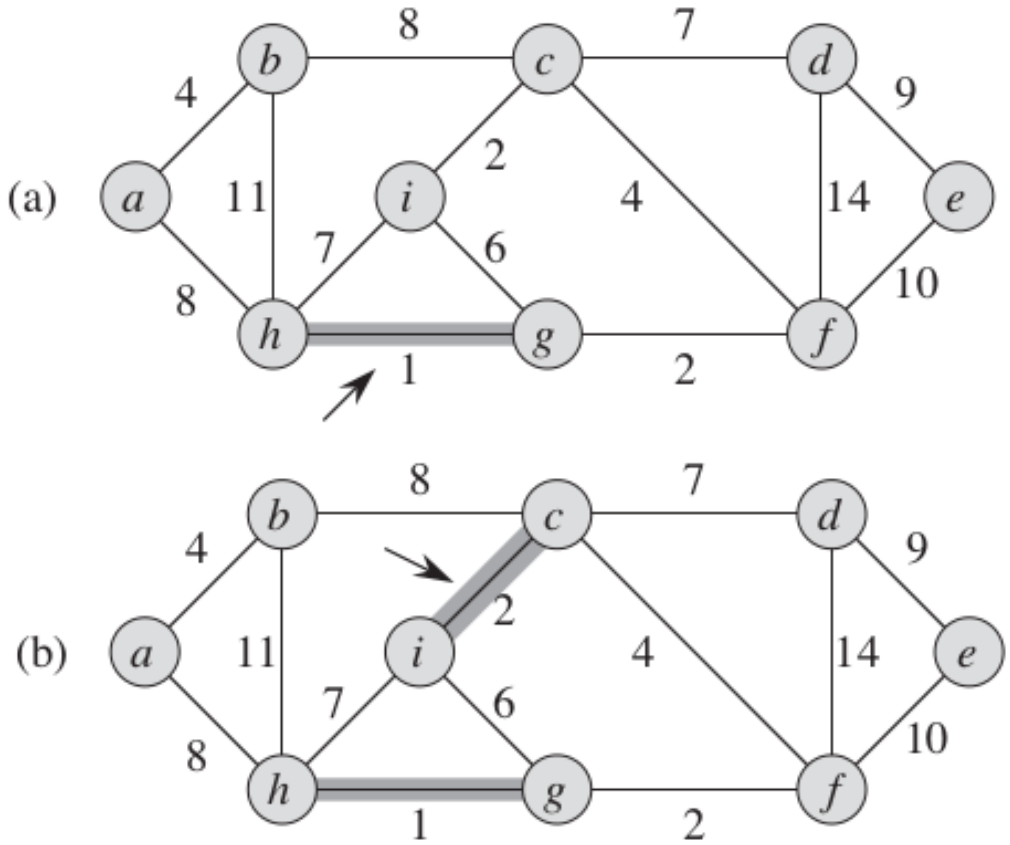


Minimalne drzewo.



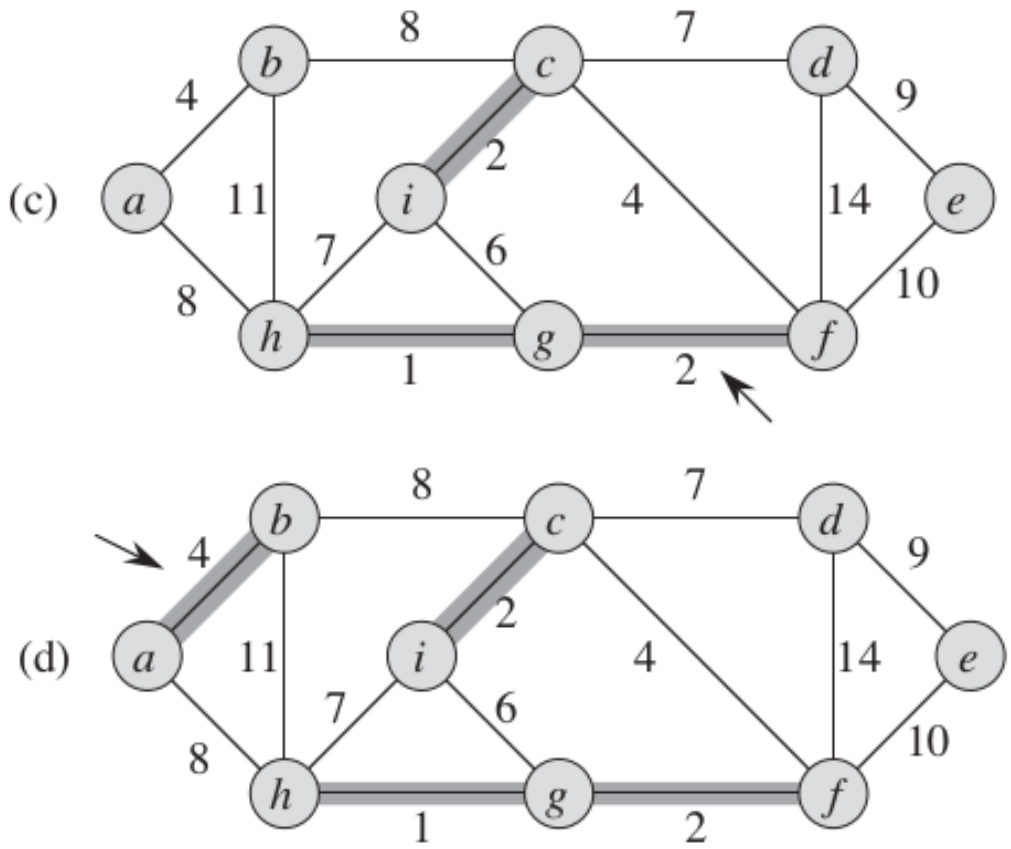
ae=1, af=2, bc=2, be=2, de=3,
ab=4, fd=6, ef=7, cd=8

Algorytm Kruskal -przykład



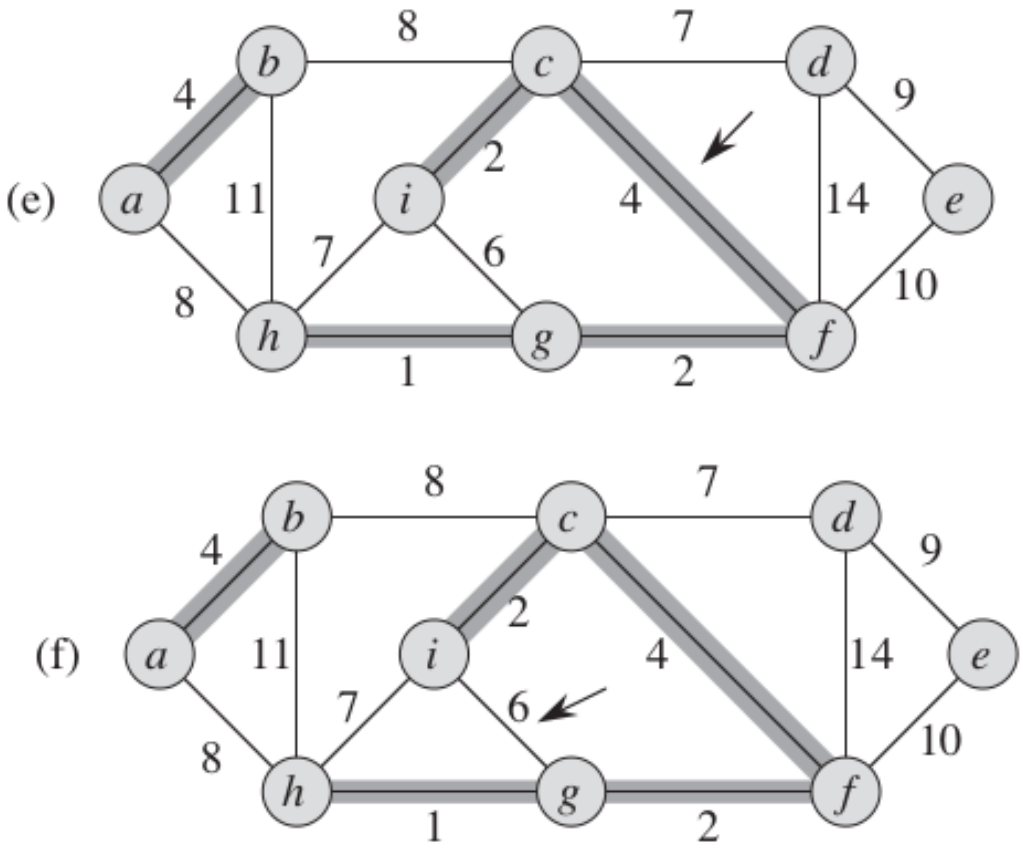
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm Kruskal -przykład



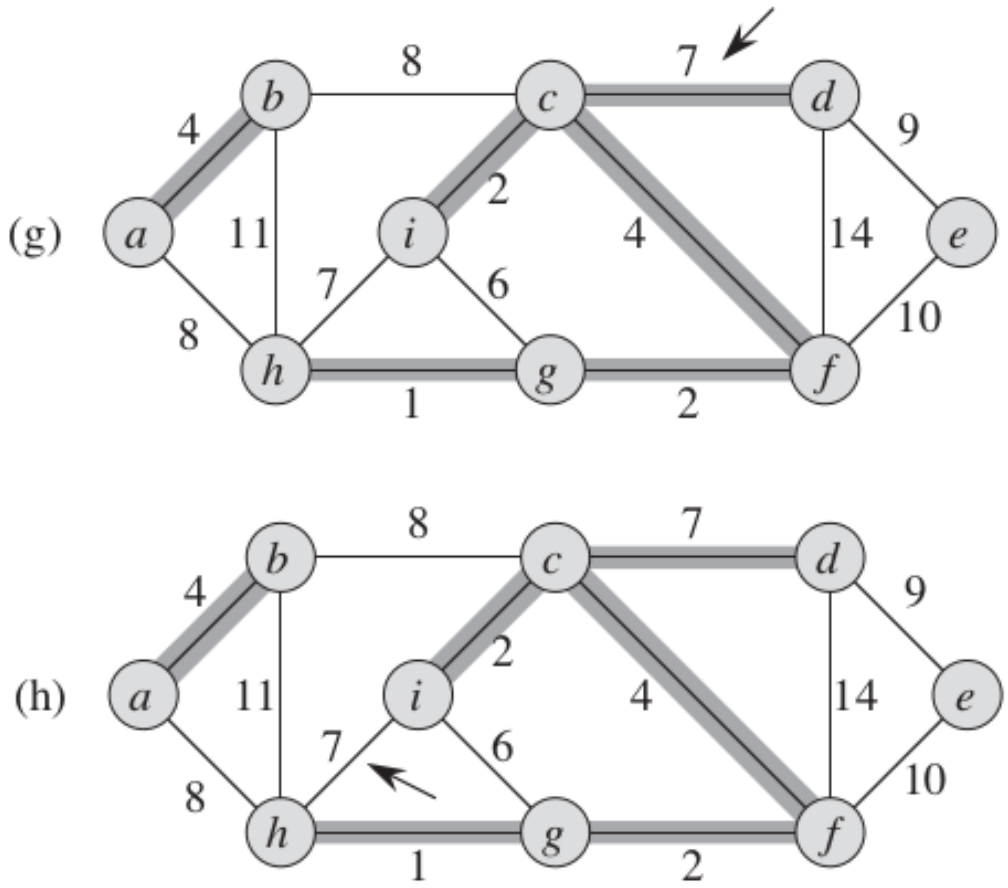
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm Kruskal -przykład



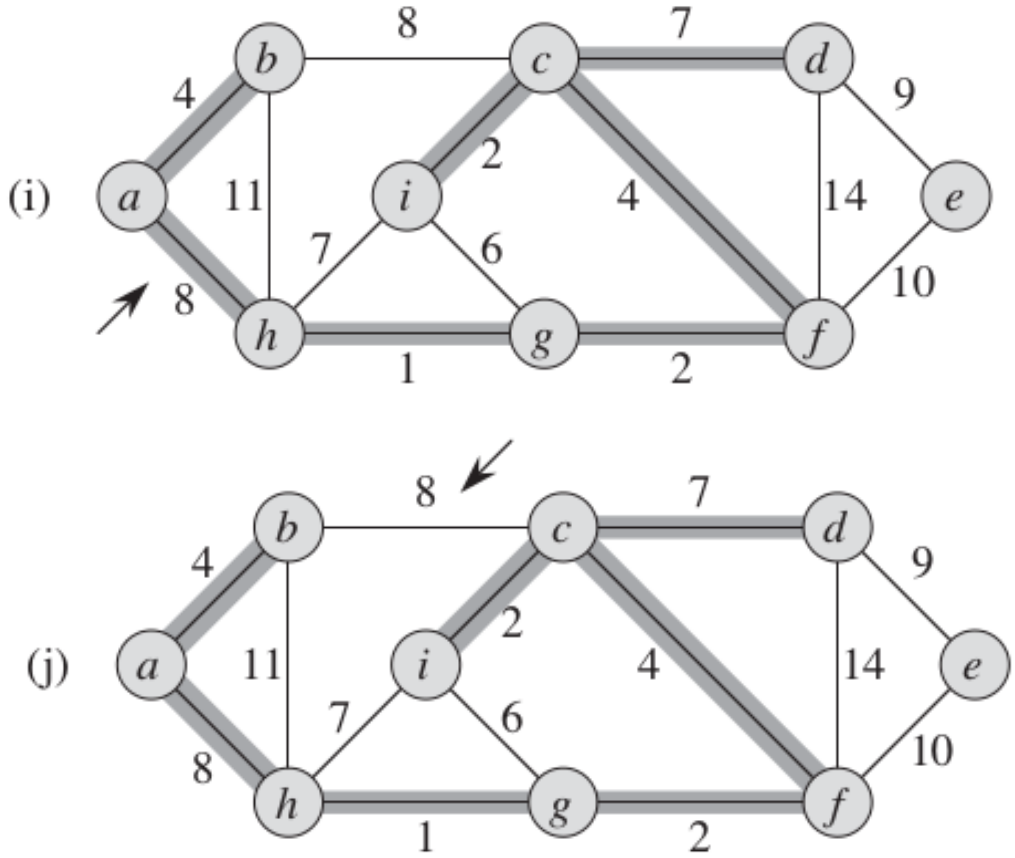
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm Kruskal -przykład



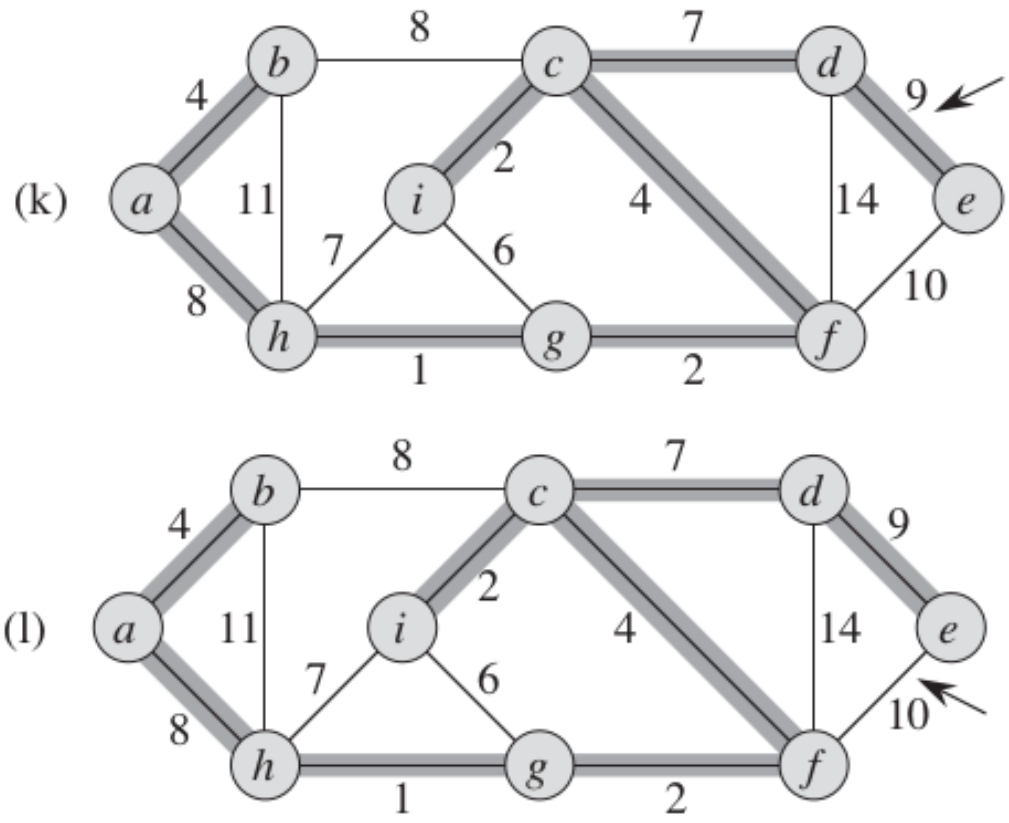
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm Kruskal -przykład



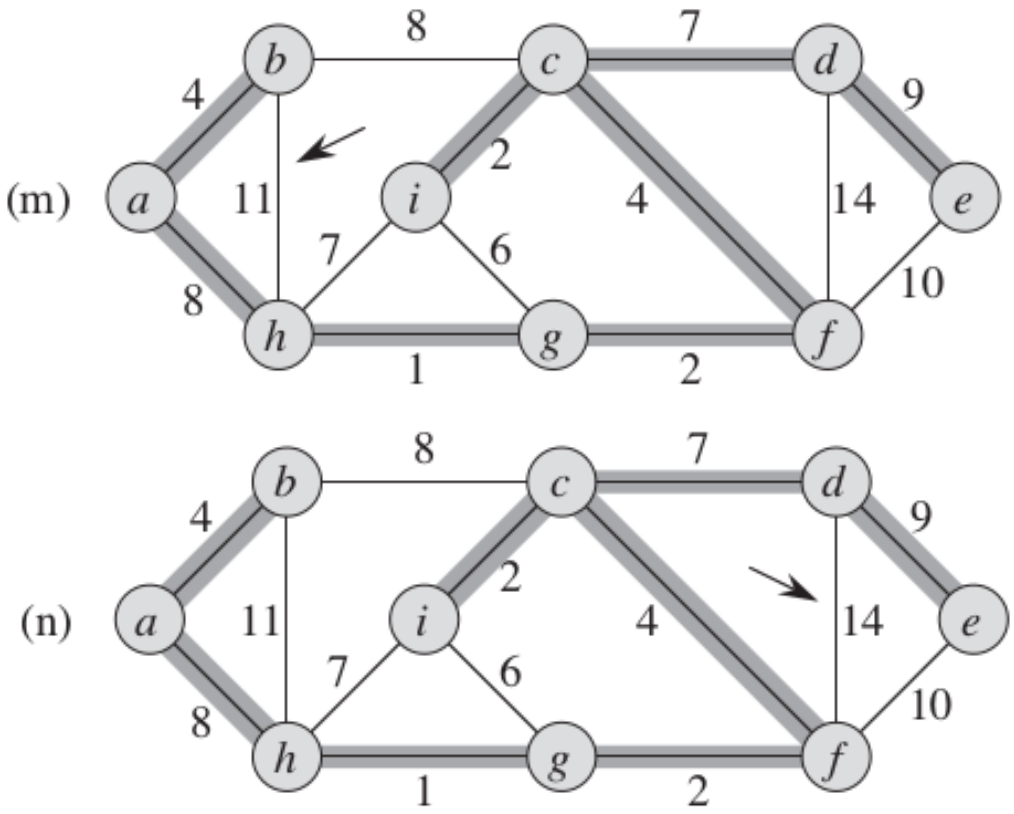
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm Kruskal -przykład



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

Algorytm Kruskal -przykład



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

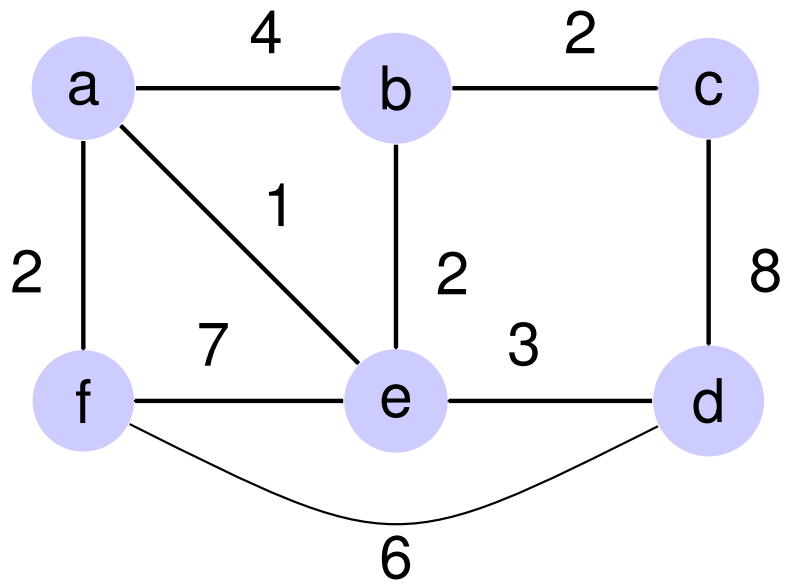
Algorytm Prima

- 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

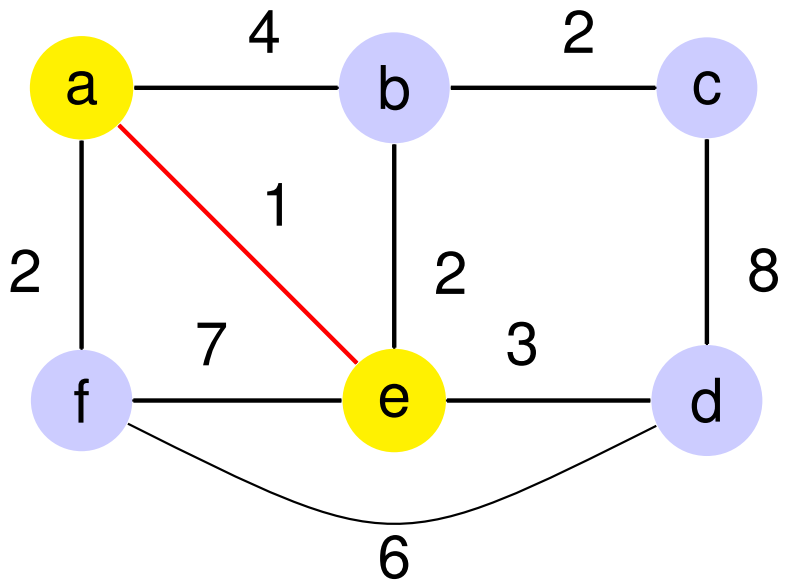
- Budowę minimalnego drzewa rozpinającego zaczynamy od dowolnego wierzchołka, np. od pierwszego. Dodajemy wierzchołek do drzewa, a wszystkie krawędzie incydentne umieszczamy na posortowanej wg. wag liście.
- Następnie zdejmujemy z listy pierwszy element (o najmniejszej wadze) i jeżeli wierzchołek, który 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 przeciwieństwie do Kruskala) z jakiegoś wierzchołka tego drzewa.

Algorytm Prima - Przykład



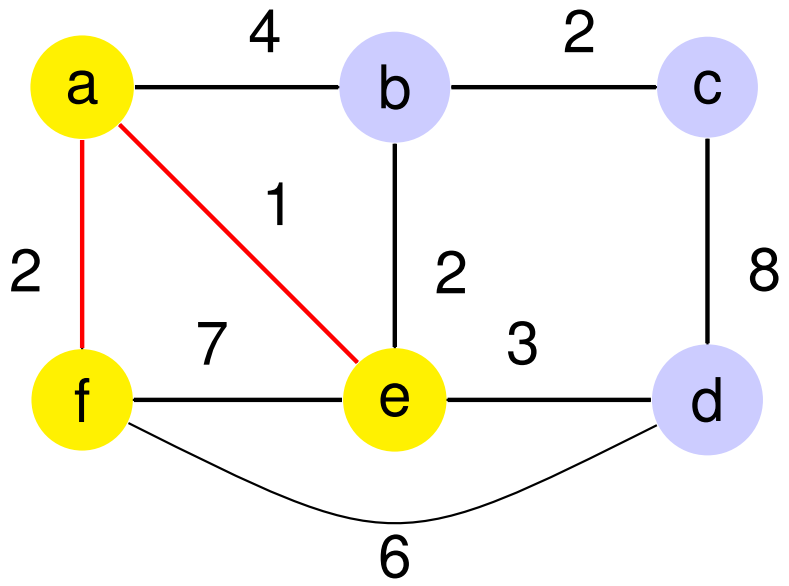
Algorytm Prima - Przykład

Krok 1.



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

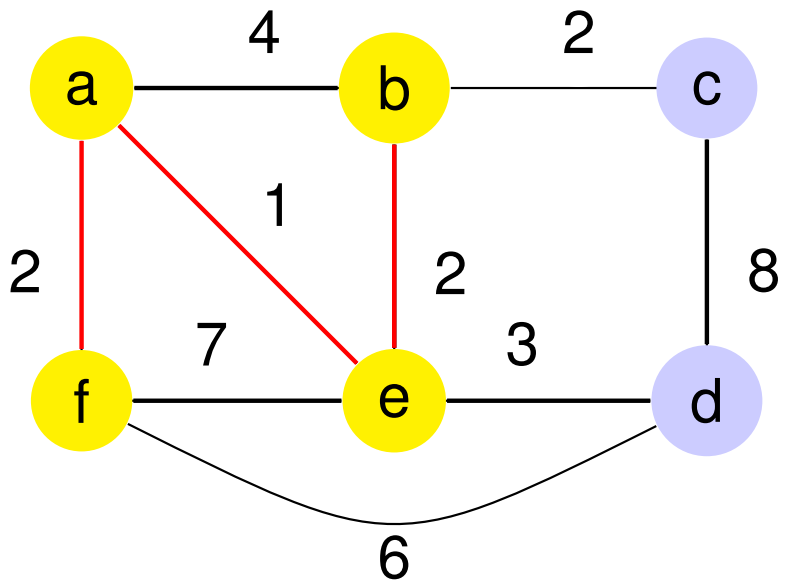
Krok 2.



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

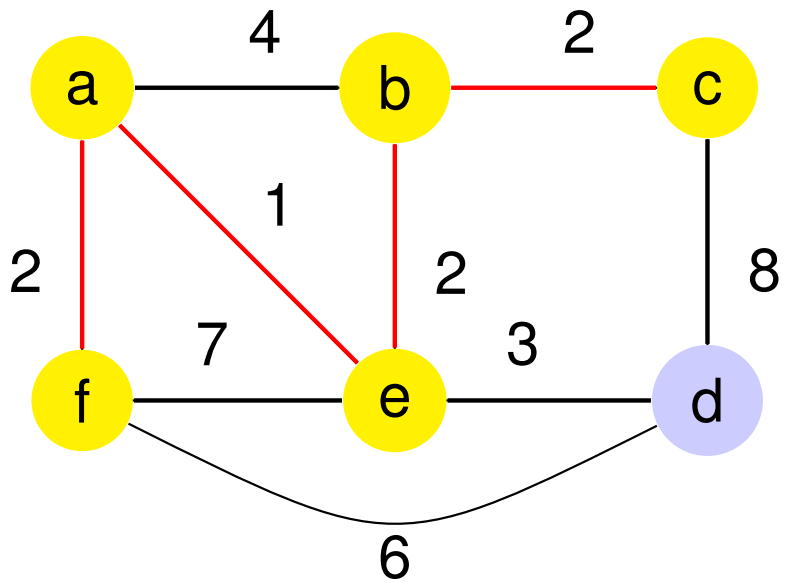
Algorytm Prima - Przykład

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).

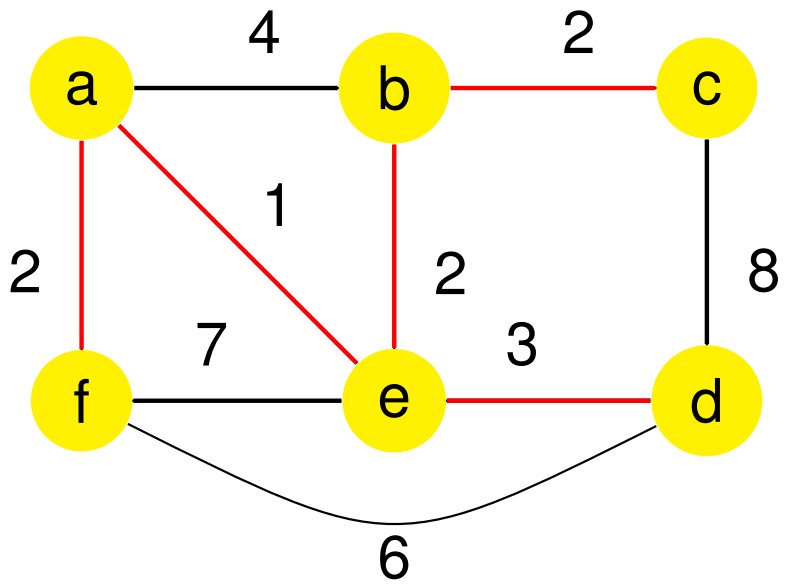
Krok 4



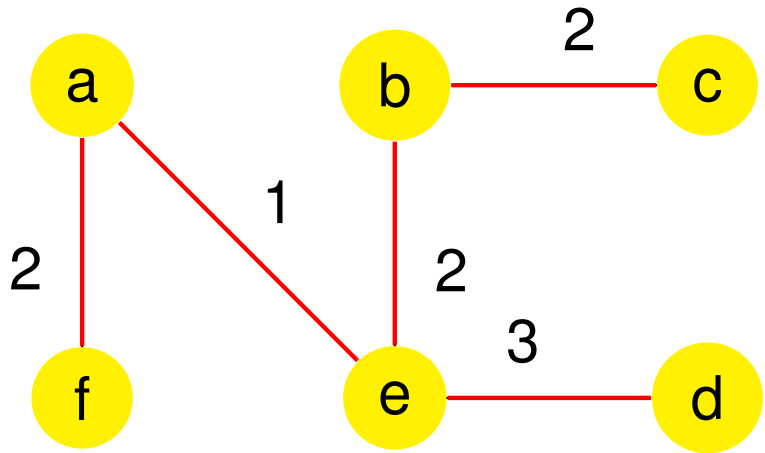
Dodajemy krawędź [b,c,2]:
L=[b,c,2],[e,d,3],[a,b,4],[f,d,6],[e,f,7]
Wybieramy krawędź (b,c).

Algorytm Prima - Przykład

Krok 5.



Minimalne Drzewo

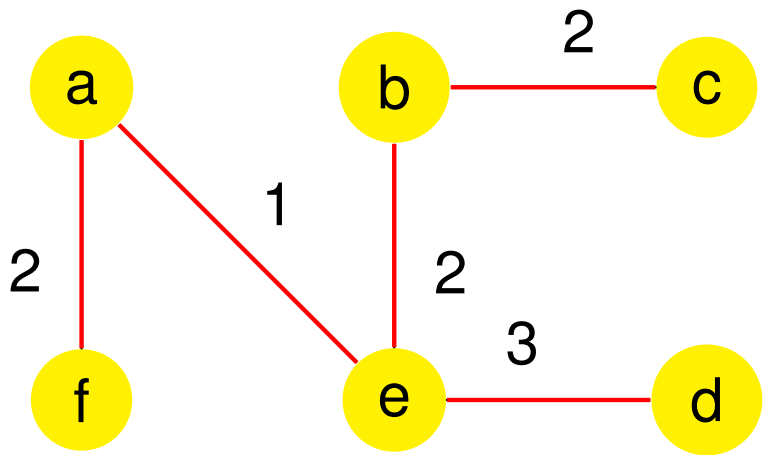


Dodajemy krawędź [c,d,8]:
L=[e,d,3],[a,b,4],[f,d,6],[e,f,7],[c,d,8]
Wybieramy krawędź (e,d).

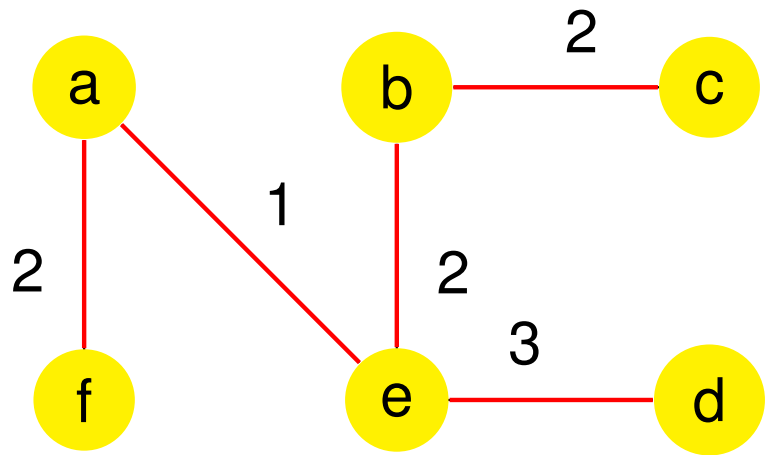
Drzewo utworzone.

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.
- $key(v)$ - klucz wyznaczający pozycję wierzchołka V w kolejce. Jest nim minimalna waga spośród wag krawędzi łączących v z wierzchołkami drzewa. $key(v) = \infty$, jeśli nie ma takiego wierzchołka.
- $\pi(v)$ - rodzic wierzchołka v w obliczanym drzewie.

Prim($G = (V, E), r$)

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

Algorytm Prima II

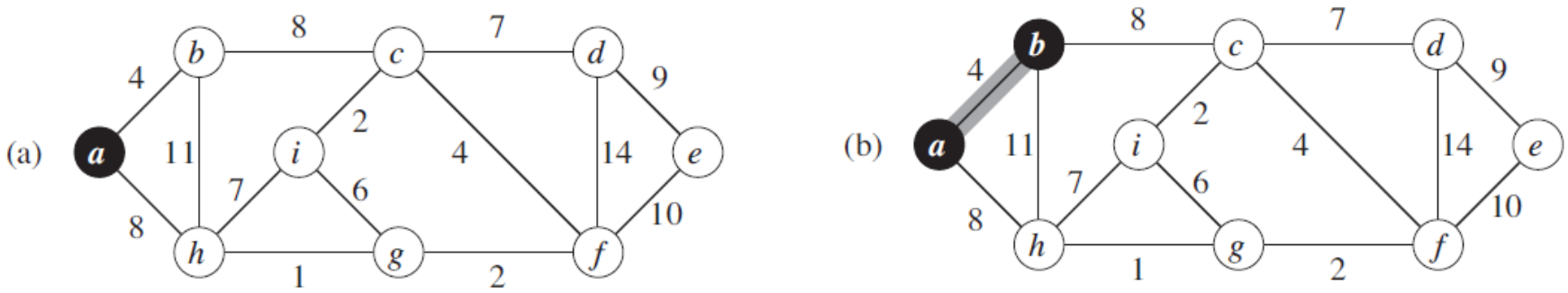
5: $Q := V$
6: **while** $Q \neq \emptyset$ **do**
7: $u := ExtractMin(Q)$
8: **for** każdy $v \in Adj[u]$ **do**
9: **if** $v \in Q$ i $w(u, v) < key(v)$ **then**
10: $\pi(v) := u$
11: $key(v) := w(u, v)$
12: **end if**
13: **end for**
14: **end while**

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

Algorytm Prima III

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

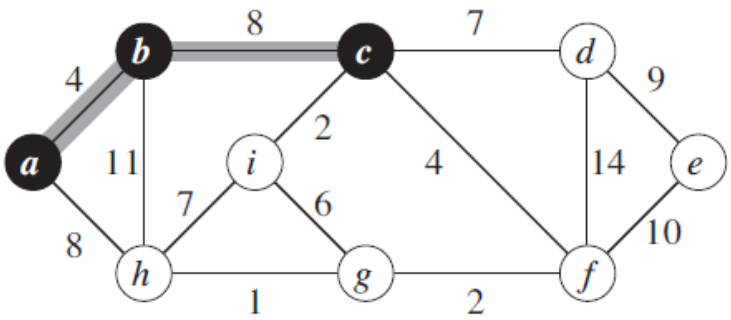
Algorytm Prima -przykład I



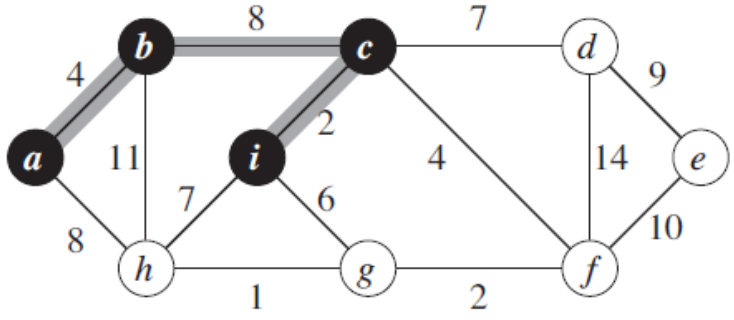
Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

Algorytm Prima -przykład II

(c)

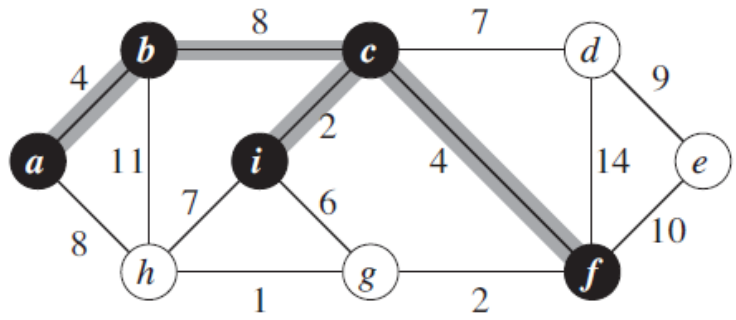


(d)

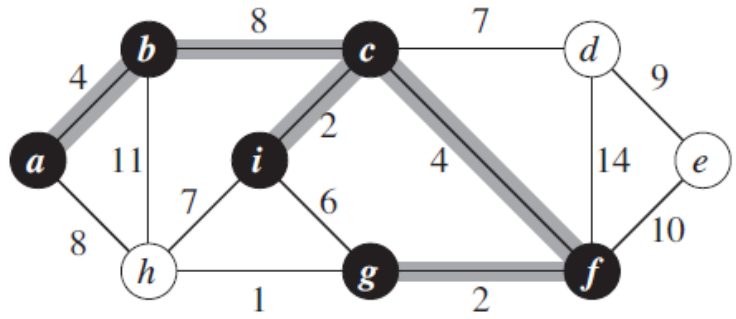


Rysunek: Źródło:Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

(e)

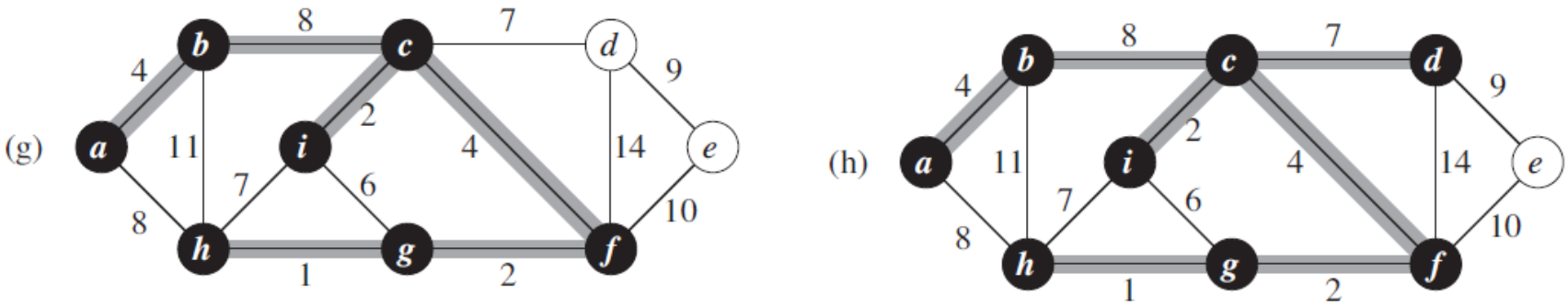


(f)



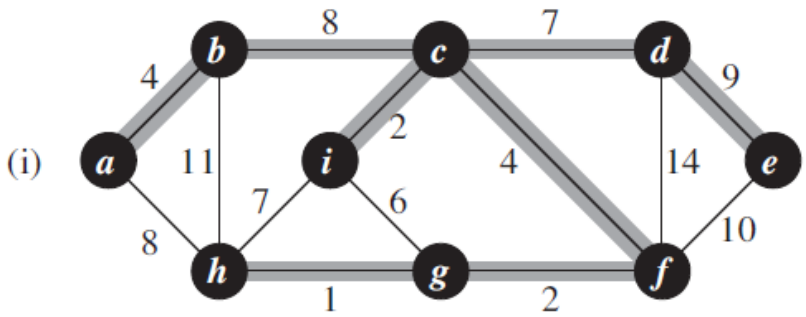
Algorytm Prima -przykład III

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 prac 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 $O(|E| + |V|log(|V|))$.