Drzewa rozpinające

Wyobraźmy sobie problem projektowania układu elektronicznego. Na pewnym etapie trzeba połączyć elektrycznie szereg końcówek (pinów) różnych podzespołów. To połączenie może rozprowadzić masę, zasilanie, albo jakiś sygnał. Chcemy je zrealizować za pomocą połączeń lub ścieżek na płytce unikając pętli. Załóżmy dodatkowo, że chcemy zminimalizować sumaryczną długość tych połączeń.

Możemy ten problem zamodelować za pomocą grafów. Załóżmy że utworzymy graf nieskierowany $G = (V, E)$, gdzie V będzie zbiorem wszystkich końcówek, a E będzie zbiorem wszystkich możliwych połączeń między nimi (zatem będzie to graf **pełny**). Acykliczny podzbiór tego grafu jest **drzewem**, a drzewo zawarte w danym grafie i obejmujące jego wszystkie wierzchołki nazywa się **drzewem rozpinającym** (*spanning tree*). Można udowodnić, że każde drzewo rozpinające grafu ma $|V| - 1$ łuków.

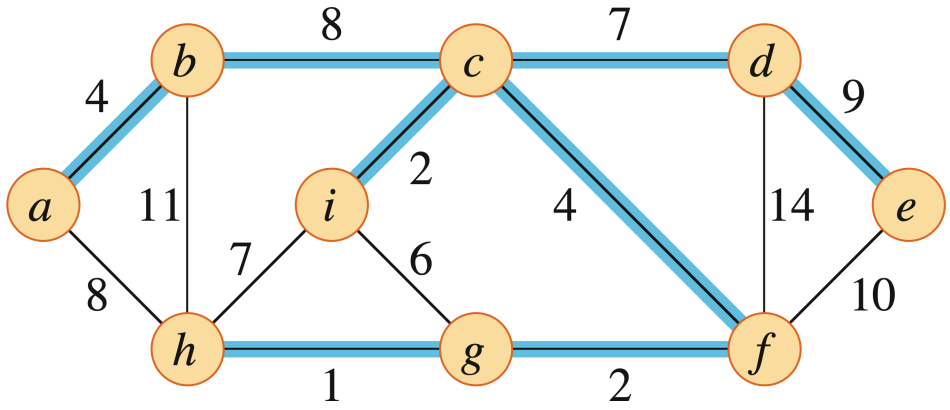
Uwaga: oczywiście nie każdy graf ma drzewo rozpinające. Gdyby graf miał izolowane węzły, lub kilka spójnych składowych, to nie dałoby się wybrać podzbioru węzłów stanowiącego drzewo rozpinające. Jednak w rozważanym przypadku wyjściowy graf jest pełny, a więc na pewno ma takie drzewo, i co więcej, w zbiorze łuków grafu zawarte są wszystkie możliwe drzewa rozpinające.

Aby uwzględnić długości połączeń elektrycznych wprowadzamy je jako **wagi** wszystkich łuków grafu $w(u, v)$. Celem jest znalezienie acyklicznego podgrafu $T \subseteq E$, który łączy wszystkie wierzchołki, i który minimalizuje sumaryczną wagę:

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

Problem znalezienia drzewa minimalizującego to kryterium nazywa się **problemem minimalnego drzewa rozpinającego**.

Na przykład, dla poniższego grafu minimalne drzewo rozpinające jest zaznaczone kolorem niebieskim. Jego całkowita waga wynosi 37. Jednak to minimalne drzewo rozpinające nie jest jedynym takim. Gdyby usunąć z niego łuk (b, c) i w jego miejsce dodać łuk (a, h) to otrzymalibyśmy inne drzewo rozpinające również o wadze 37.



Algorytm Kruskala

Idea algorytmu Kruskala jest prosta: poczynając od pustego zbioru łuków A, rozpatruj wszystkie łuki grafu w kolejności rosnących wag, i dodawaj do zbioru A tylko te łuki, które łączą (niepołączone) składowe.

```
MST-KRUSKAL(G, w)
 1  A = ∅
 2  for each vertex v ∈ G, V
 3        MAKE-SET(v)
 4  create a single list of the edges in G.E
 5  sort the list of edges into monotonically increasing order by weight w
 6  for each edge (u, v) taken from the sorted list in order
 7        if FIND-SET(u) ≠ FIND-SET(v)
 8              A = A ∪ {(u, v)}
 9              UNION(u, v)
10  return A
```

MAKE-SET — tworzy zbiór zawierający wyłącznie dany wierzchołek
FIND-SET — znajduje zbiór zawierający dany wierzchołek
UNION — łączy dwa zbiory w jeden

Pseudokod algorytmu MST-Kruskal nie ujawnia całego mechanizmu jego działania, ponieważ algorytm wykorzystuje operacje na zbiorach, których implementacja jest ukryta, i niekoniecznie jest trywialna. W szczególności procedury Find-Set i Union nie są jednokrokowe, i czas działania całego algorytmu zależy od wykonywania tych procedur. Nie wdając się w szczegóły można udowodnić, że ten czas jest $O(E \log V)$.

Podobnie sprawa się ma z poprawnością algorytmu. Oczywiste jest, że algorytm zbuduje drzewo, ponieważ nigdy nie doda do budowanego grafu łuku, który łączy składowe już połączone. Jednak, że jest to drzewo rozpinające, i w szczególności minimalne drzewo rozpinające, już takie oczywiste nie jest. Dowód poprawności działania tego algorytmu tu pomijamy.

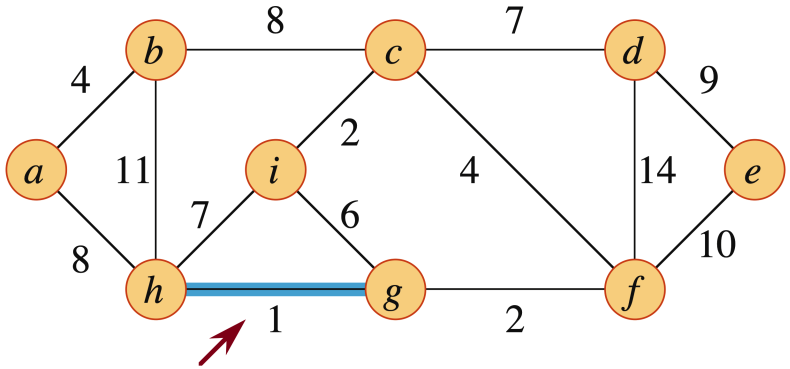
Algorytm Kruskala należy do klasy algorytmów **zachłannych**, czyli takich, które powtarzalnie wykonują krok w danym momencie optymalny, i otrzymują w tym procesie rozwiązanie globalnie optymalne.

Jednak, jak jeszcze nie raz przekonamy się, nie wszystkie problemy można rozwiązać za pomocą algorytmów zachłannych, i ogólnie nie ma gwarancji, że algorytm zachłanny wybierający kroki lokalnie optymalne zawsze wygeneruje globalnie optymalne rozwiązanie.

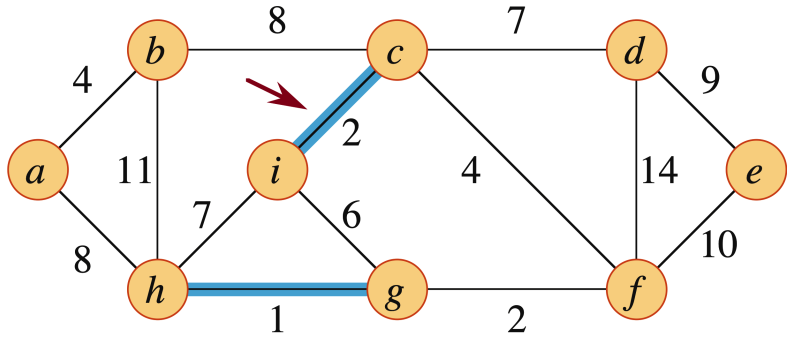
Algorytm Kruskala — przykład działania

Czerwona strzałka wskazuje kolejno analizowane łuki. Te dodane do tworzonego lasu A stają się niebieskie. Nowo dodawany łuk zawsze łączy istniejące dwa drzewa.

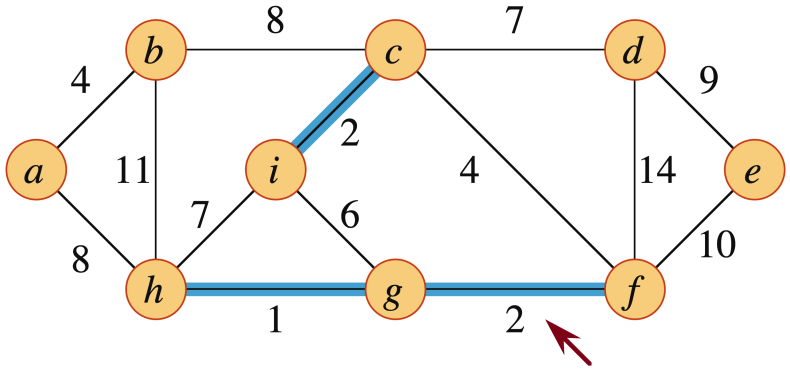
(a)



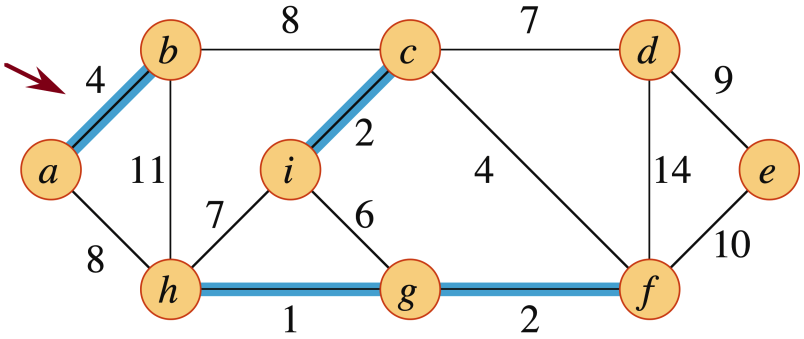
(b)



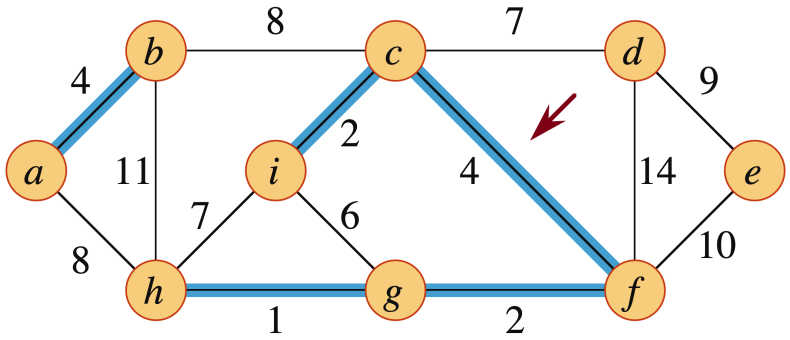
(c)



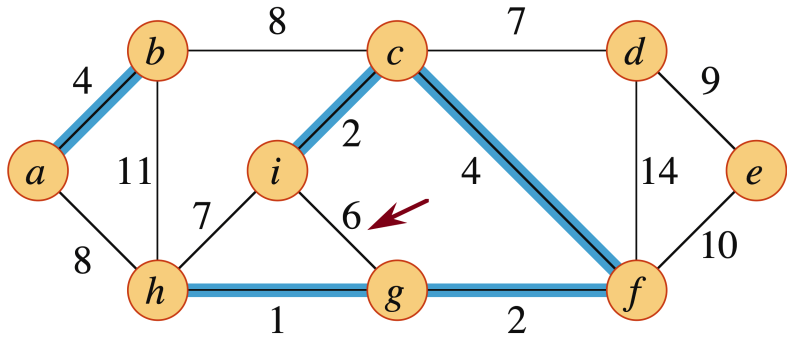
(d)



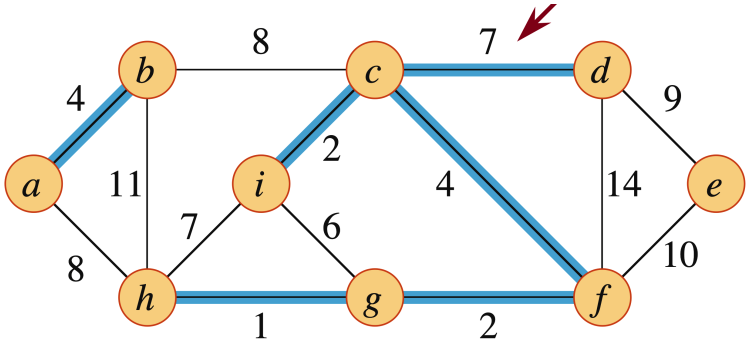
(e)



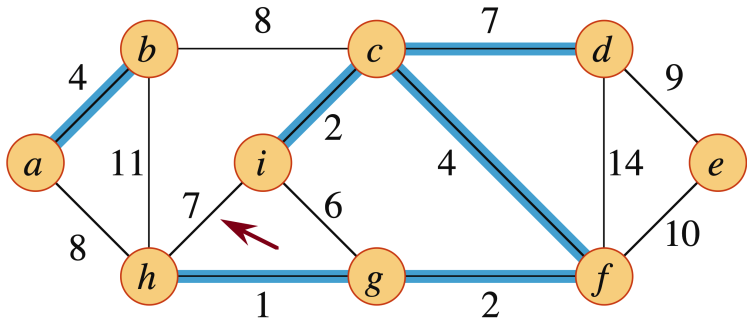
(f)



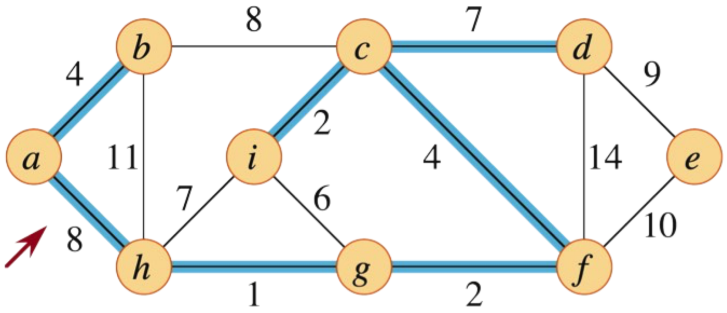
(g)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

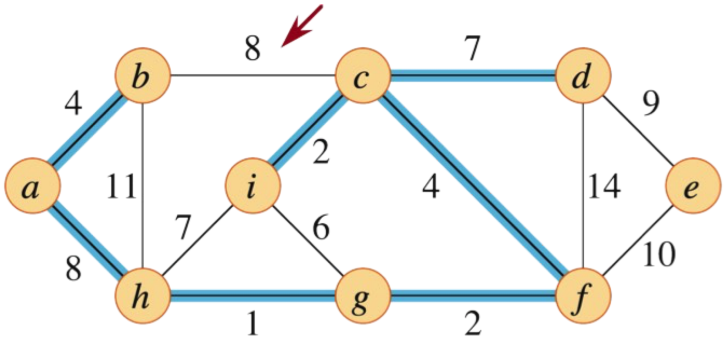
(h)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

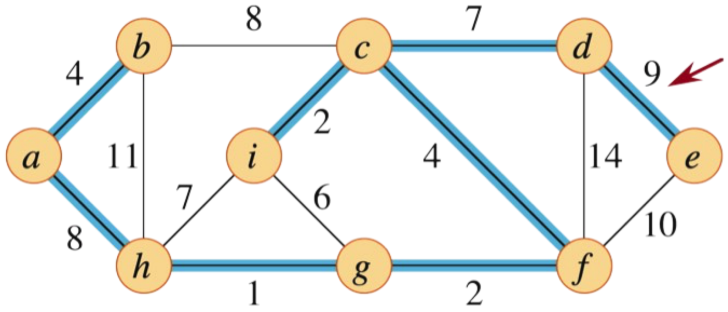
(i)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

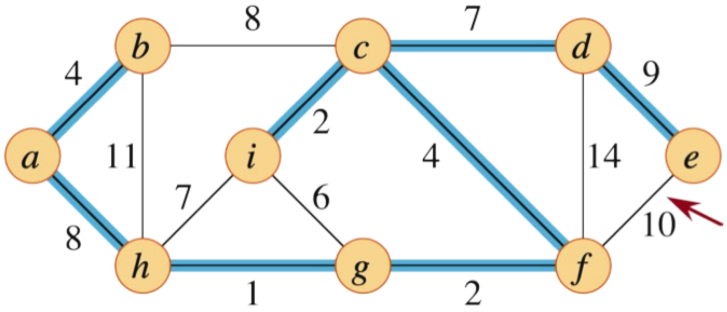
(j)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

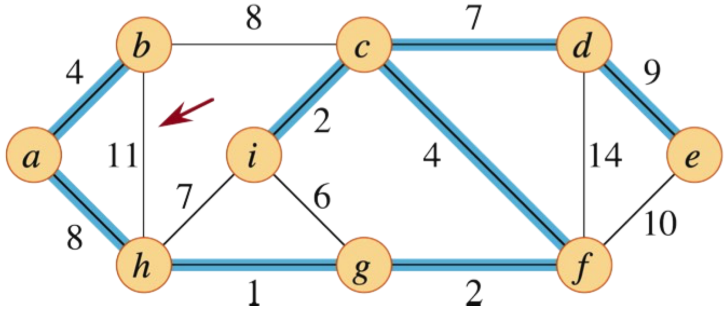
(k)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

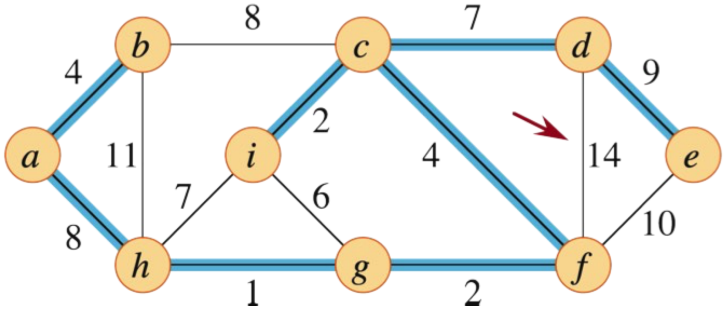
(l)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

(m)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

(n)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

Algorytm Prima

Algorytm Prima tworzy MST poczynając od dowolnego węzła r dodając do niego ten węzeł spoza drzewa, który łączy z drzewem łuk o najmniejszej wadze. Drzewo jest pamiętane niejawnie, w postaci atrybutów π wskazujących poprzednik węzła w drzewie. Atrybut key zawiera wagi łuków dla wszystkich węzłów sąsiednich z drzewem.

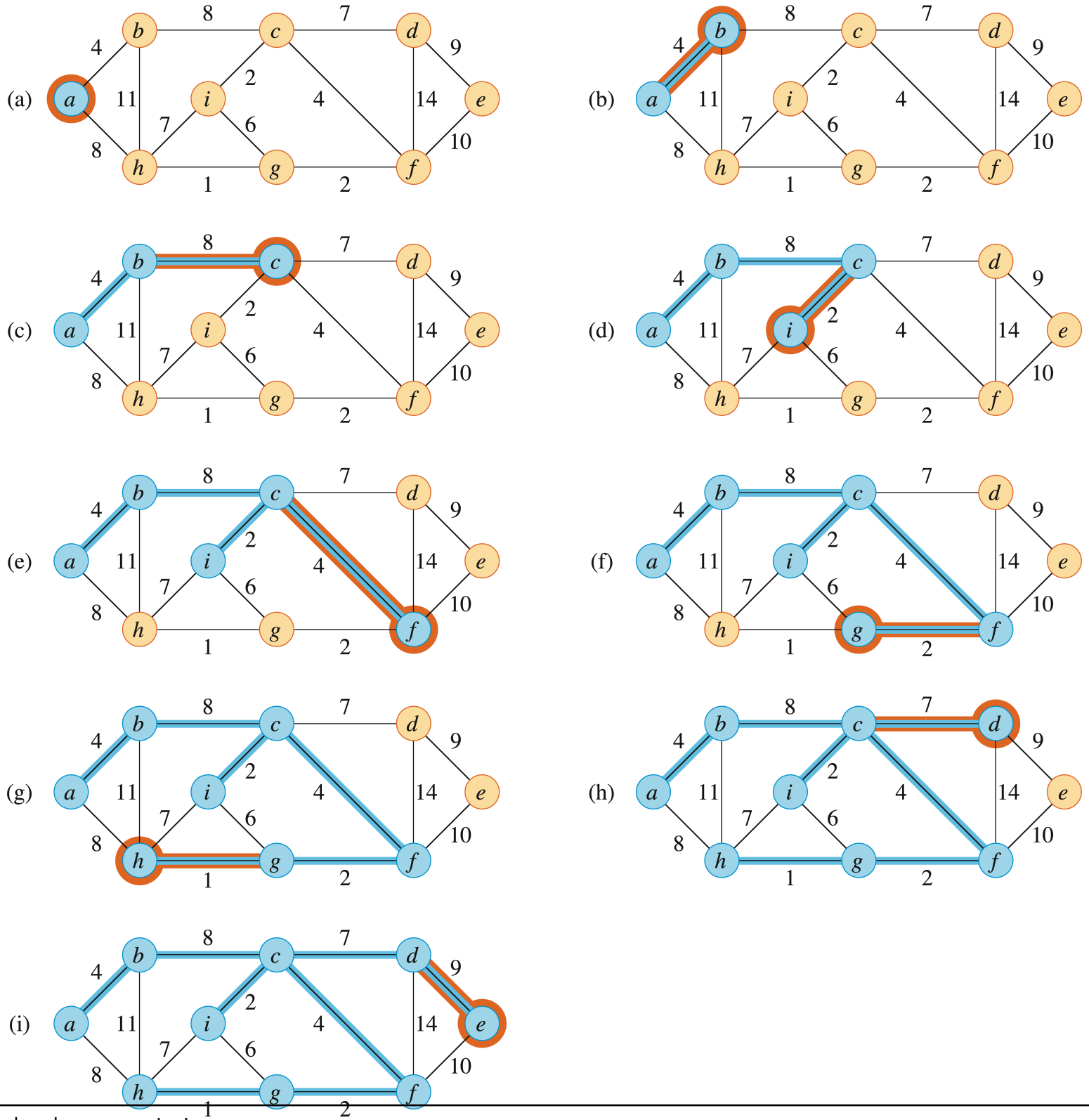
```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V
 2        u.key = ∞
 3        u.π = NIL
 4  r.key = 0
 5  Q = ∅
 6  for each vertex u ∈ G.V
 7        INSERT(Q, u)
 8  while Q ≠ ∅
 9        u = EXTRACT-MIN(Q)        // add u to the tree
10        for each vertex v in G.Adj[u] // update keys of u's non-tree neighbors
11              if v ∈ Q and w(u, v) < v.key
12                    v.π = u
13                    v.key = w(u, v)
14                    DECREASE-KEY(Q, v, w(u, v))
```

Kolejka priorytetowa Q zawiera wszystkie węzły jeszcze niedodane do drzewa. Kluczowa dla pracy algorytmu jest aktualizacja kluczy węzłów w Q (spoza drzewa), zawierających wagę „najlżejszego" łuku łączącego dany węzeł z drzewem.

W każdym przebiegu pętli **while** wybierany jest węzeł u połączony z dotychczas zbudowaną częścią drzewa łukiem o minimalnej wadze. Kolejka priorytetowa pozwala na efektywne wykonanie takiej operacji. Następnie analizowani są wszyscy sąsiedzi dodawanego węzła u aby zaktualizować informacje o najbliższych sąsiadach drzewa. Jest możliwe, że dany sąsiad węzła u o nazwie v jest nowo odkrytym sąsiadem drzewa, i w takim przypadku jego klucz ustawiany jest na wagę łuku (u, v) z pierwotnej wartości ∞. Ale jest też możliwe, że v był już znany jako sąsiad drzewa przez jakiś inny łuk, i w takim przypadku w kroku 11 wybierana jest waga lżejszego łuku łączącego v z drzewem.

Jak widać algorytm Prima jest również **zachłanny**, ponieważ w każdym kroku wybiera węzeł o minimalnej wadze łuku. Kluczowa dla poprawnej pracy tych zachłannych strategii w algorytmach Kruskala i Prima jest własność jaką muszą spełniać te minimalne wybierane elementy. W algorytmie Kruskala jest to wybór łuku o minimalnej wadze, łączącego niepołączone jeszcze drzewa lasu. W algorytmie Prima jest to wybór węzła nienależącego do drzewa, o minimalnej wadze łuku łączącego go z drzewem.

Czas działania algorytmu Prima jest $O(E \log V)$, podobnie jak algorytmu Kruskala.


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024, rozdział 21.