

Algorithm strategies:

Greedy algorithms

Optimization problems

- An **optimization problem** is one in which you want to find, not just a solution, but the best solution
- A “**greedy algorithm**” sometimes works well for optimization problems
- A **greedy algorithm** works in phases. At each phase:
 - You take the **best you can get right now**, without regard for future consequences
 - You hope that by **choosing a local optimum at each step**, you will end up at a global optimum

Greedy algorithms

- The **greedy approach** suggests **constructing a solution** through a sequence of steps, **each expanding a partially constructed solution** obtained so far, **until a complete solution** to the problem is reached.

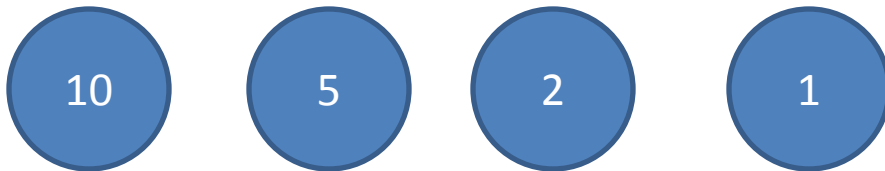
Greedy algorithms

- The **central point** of this technique—the choice made must be:
- **Feasible**, i.e., it has to satisfy the problem's constraints
- **Locally optimal**, i.e., it has to be the best local choice among all feasible choices available on that step
- **Irrevocable**, i.e., once made, it cannot be changed on subsequent steps of the algorithm

Greedy algorithms

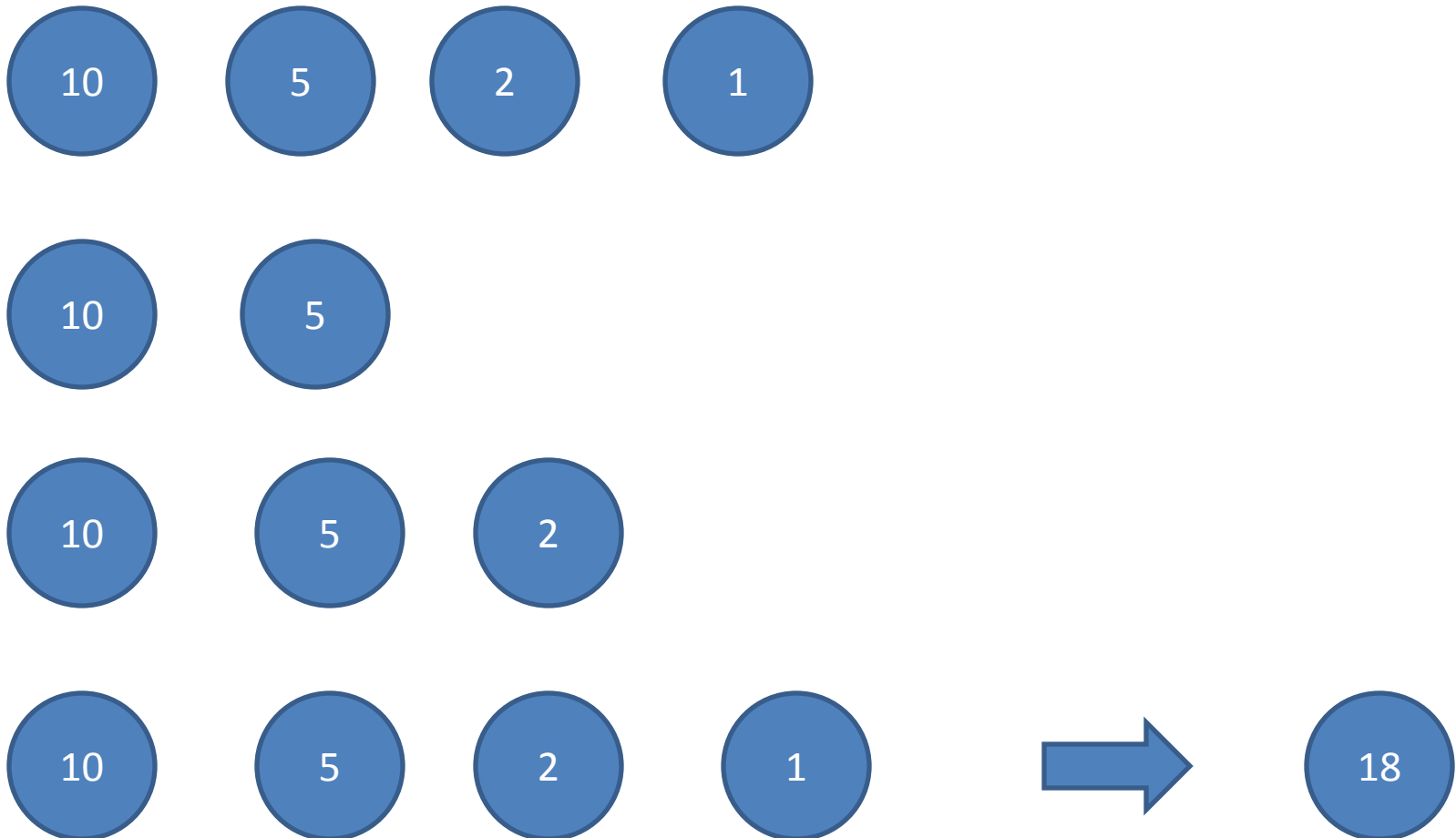
Change-making problem

- Give change for a specific amount **n** with the **least number of coins** of the denominations

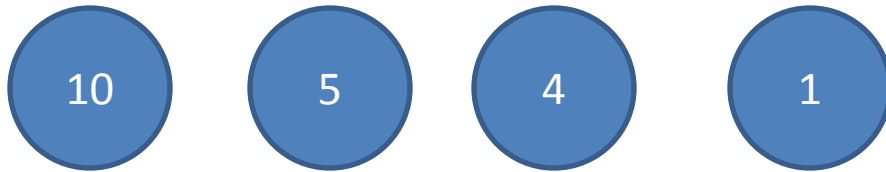


- Get 18 with least number of coins

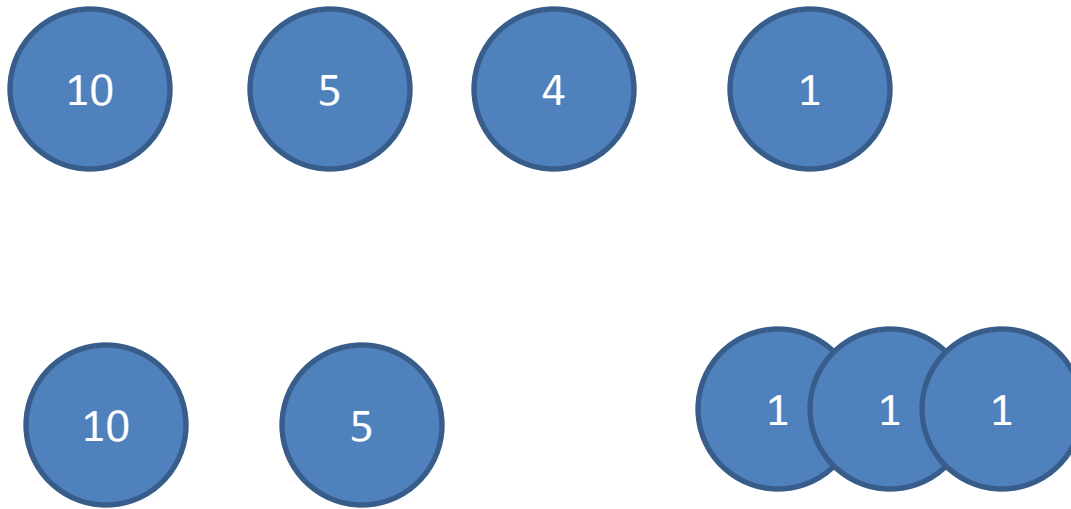
Greedy algorithms



Greedy algorithms

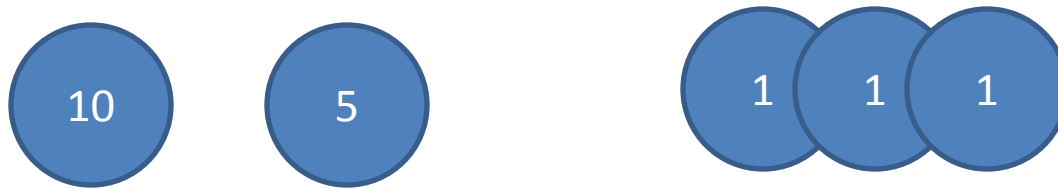
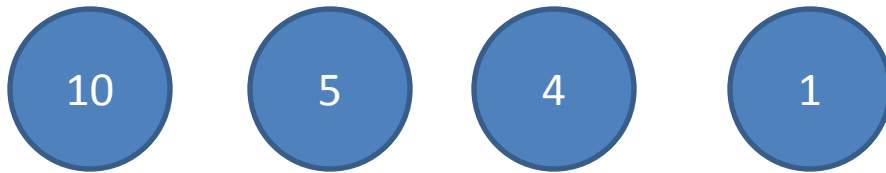


Greedy algorithms



Is this optimal ?

Greedy algorithms



Not optimal



Knapsack Problem

- Given n items of known **weights w_1, \dots, w_n** and **values v_1, \dots, v_n** and a knapsack of capacity **W** ,
- find the most valuable subset of the items that fit into the knapsack.
- Given **a set of items**, each with a **weight and a value**, determine a **subset of items** to include in a collection so that the **total weight is less than or equal** to a given limit and the **total value is as large as possible**.
- The knapsack problem is in combinatorial optimization problem.

Problem Scenario

- A thief is robbing a store and can carry a maximal weight of **W** into his knapsack.
- There are **n** items available in the store and weight of **i^{th}** item is **w_i** and its profit is **p_i** . What items should the thief take?
- In this context, the items should be selected in such a way that the thief will carry those items for which he will **gain maximum profit**.
- Based on the nature of the items, Knapsack problems are categorized as
 - **Fractional Knapsack**
 - **0-1 Knapsack**

0-1 Knapsack

Let us consider that the capacity of the knapsack **$W = 60$** and the list of provided items are shown in the following table.

Item	A	B	C	D
Profit (p_i)	280	100	120	50
Weight(w_i)	40	10	20	10
Ratio (p_i/w_i)	7	10	6	5

0-1 Knapsack

After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit (p_i)	100	280	120	50
Weight(w_i)	10	40	20	10
Ratio (p_i/w_i)	10	7	6	5

0-1 Knapsack

After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit (p_i)	100	280	120	50
Weight(w_i)	10	40	20	10
Ratio (p_i/w_i)	10	7	6	5

The total weight of the selected items is $10 + 40 + 10 = 60$

And the total profit is $100 + 280 + 50 = 380 + 50 = 430$

Fractional Knapsack

After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit (p_i)	100	280	120	50
Weight(w_i)	10	40	20	10
Ratio (p_i/w_i)	10	7	6	5

Fractional Knapsack

After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit (pi)	100	280	120	50
Weight(wi)	10	40	20	10
Ratio (pi/wi)	10	7	6	5

The total weight of the selected items is $10 + 40 + 20 * (10/20) = 60$

And the total profit is $100 + 280 + 120 * (10/20) = 380 + 60 = 440$

Fractional Knapsack

Algorithm: Greedy-Fractional-Knapsack ($w[1..n]$, $p[1..n]$, W)

for $i = 1$ to n do

$x[i] = 0$

weight = 0

for $i = 1$ to n

 if weight + $w[i] \leq W$ then

$x[i] = 1$

 weight = weight + $w[i]$

 else

$x[i] = (W - \text{weight}) / w[i]$

 weight = W

 break

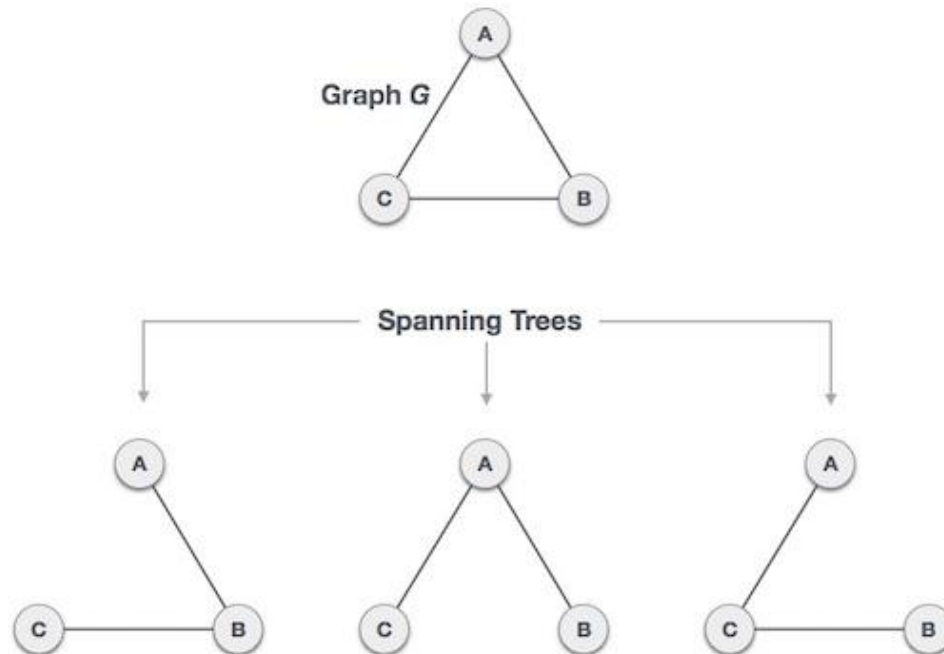
return x

$O(n)$

$O(n \log n)$

spanning tree

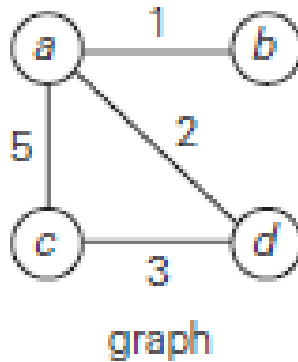
- A **spanning tree** of an **undirected connected graph** is its **connected acyclic sub graph** (i.e., a tree) that **contains all the vertices** of the graph.



minimum spanning tree problem

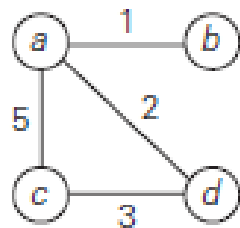
- If a **undirected connected graph** has **weights assigned to its edges**, a minimum spanning tree is its **spanning tree of the smallest weight**, where the **weight of a tree** is defined as the **sum of the weights on all its edges**.
- The **minimum spanning tree problem** is the problem of **finding a minimum spanning tree** for a **given weighted connected graph**.

minimum spanning tree problem

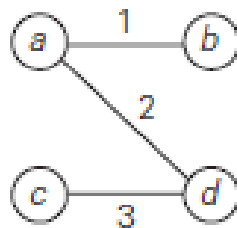


Find minimum spanning tree using brute force approach.

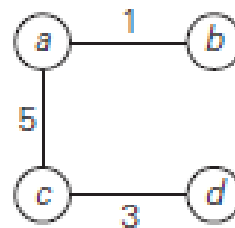
minimum spanning tree problem



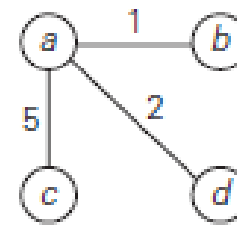
graph



$w(T_1) = 6$



$w(T_2) = 9$



$w(T_3) = 8$

T_1 is the minimum spanning tree.

Prim's Algorithm

- Prim's algorithm constructs a minimum spanning tree **through a sequence of expanding sub-trees**.
- The **initial** sub-tree in such a sequence **consists of a single vertex** selected arbitrarily from the set V of the graph's vertices.
- On **each iteration**, the algorithm expands the current tree in the greedy manner by **simply attaching to it the nearest vertex** not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight.)
- The algorithm **stops** after all the graph's vertices have been included in the tree being constructed.

Prim's Algorithm

ALGORITHM *Prim*(G)

//Prim's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph $G = (V, E)$

//Output: E_T , the set of edges composing a minimum spanning tree of G

$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex

$E_T \leftarrow \emptyset$

for $i \leftarrow 1$ to $|V| - 1$ do

 find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
 such that v is in V_T and u is in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

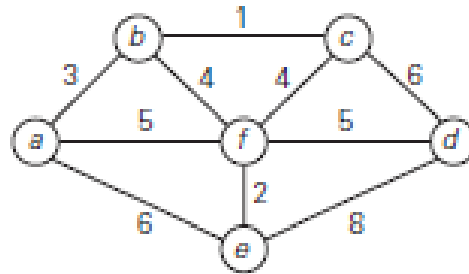
$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

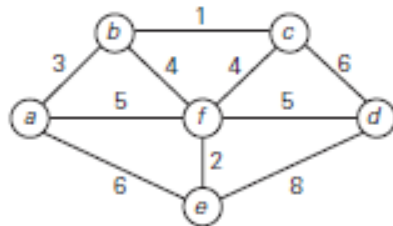
After we have identified a vertex u^* to be added to the tree, we need to perform two operations:

1. Move u^* from the set $V - V_T$ to the set of tree vertices V_T .
2. For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u 's current distance label, update its labels by u^* and the weight of the edge between u^* and u , respectively.

Prim's Algorithm

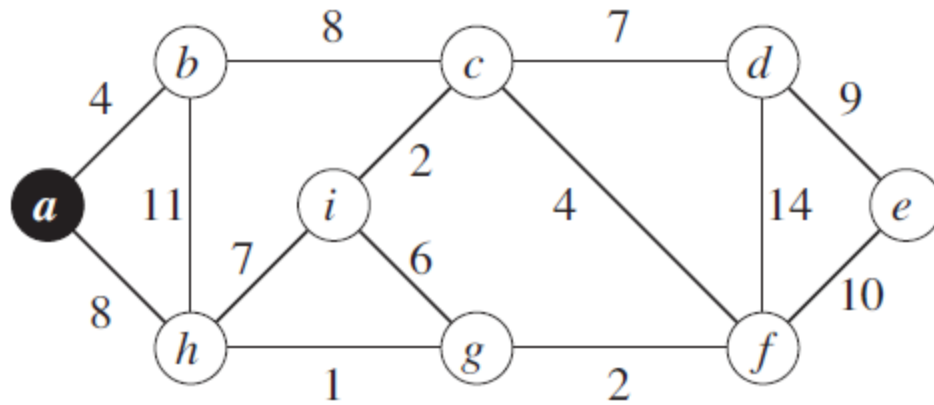


Prim's Algorithm

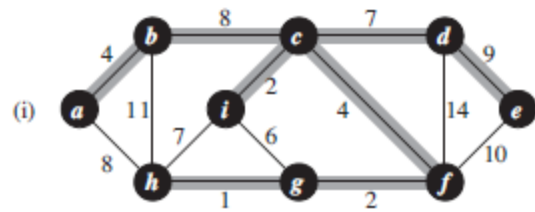
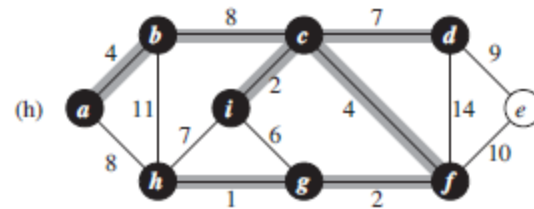
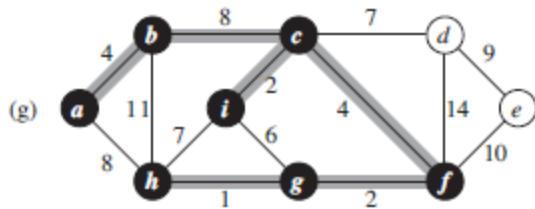
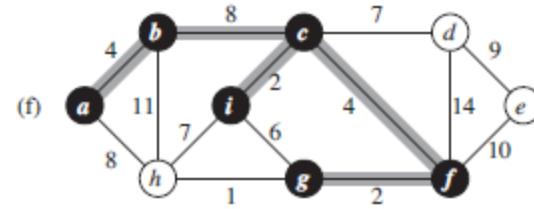
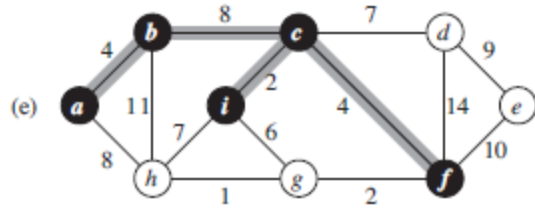
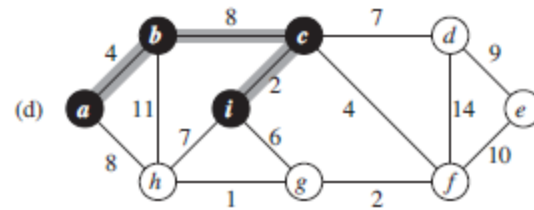
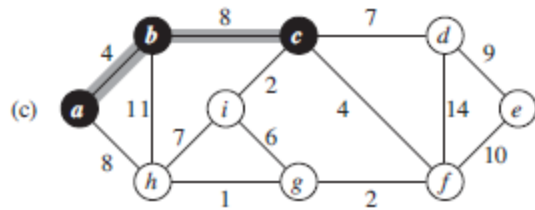
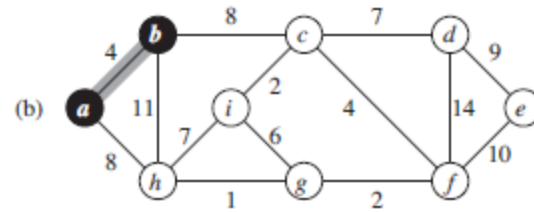
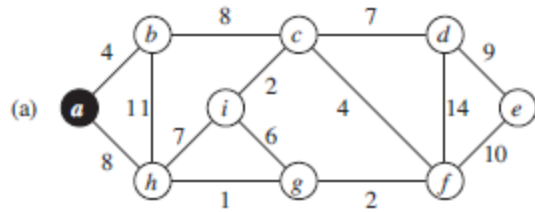


Tree vertices	Remaining vertices	Illustration
$a(-, -)$	$b(a, 3)$ $c(-, \infty)$ $d(-, \infty)$ $e(a, 6)$ $f(a, 5)$	
$b(a, 3)$	$c(b, 1)$ $d(-, \infty)$ $e(a, 6)$ $f(b, 4)$	
$c(b, 1)$	$d(c, 6)$ $e(a, 6)$ $f(b, 4)$	
$f(b, 4)$	$d(f, 5)$ $e(f, 2)$	
$e(f, 2)$	$d(f, 5)$	
$d(f, 5)$		

Prim's Algorithm



Prim's Algorithm



Proof of Prim's Algorithm

- Does Prim's algorithm always yield a minimum spanning tree?

Proof of Prim's Algorithm

- Does Prim's algorithm always yield a minimum spanning tree?

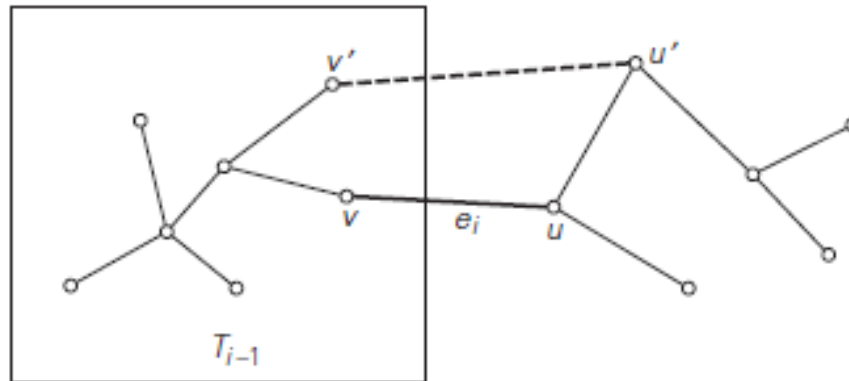
The answer to this question is yes.

Proof of Prim's Algorithm

- Let us prove by **induction** that each of the **sub-trees** T_i , $i = 0, \dots, n - 1$, generated by Prim's algorithm is a part (i.e., a sub-graph) of some minimum spanning tree.
- T_0 consists of a **single vertex** and hence must be a part of any minimum spanning tree.
- For the **inductive step**, let us assume that T_{i-1} is part of some minimum spanning tree T .
- We need to prove that T_i , generated from T_{i-1} by Prim's algorithm, is **also a part of a minimum spanning tree**

Proof of Prim's Algorithm

- By **contradiction** by assuming that no minimum spanning tree of the graph can contain T_i .



Prim's Algorithm

ALGORITHM *Prim*(G)

//Prim's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph $G = \langle V, E \rangle$

//Output: E_T , the set of edges composing a minimum spanning tree of G

$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex

$E_T \leftarrow \emptyset$

for $i \leftarrow 1$ **to** $|V| - 1$ **do**

 find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
 such that v is in V_T and u is in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

Compute the efficiency of the algorithm

MST-PRIM(G, w, r)

1 **for** each $u \in G.V$

2 $u.key = \infty$

3 $u.\pi = \text{NIL}$

4 $r.key = 0$

5 $Q = G.V$

6 **while** $Q \neq \emptyset$

7 $u = \text{EXTRACT-MIN}(Q)$

8 **for** each $v \in G.Adj[u]$

9 **if** $v \in Q$ and $w(u, v) < v.key$

10 $v.\pi = u$

11 $v.key = w(u, v)$

Prim's Algorithm

ALGORITHM *Prim*(G)

//Prim's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph $G = \langle V, E \rangle$

//Output: E_T , the set of edges composing a minimum spanning tree of G

$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex

$E_T \leftarrow \emptyset$

for $i \leftarrow 1$ **to** $|V| - 1$ **do**

 find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
 such that v is in V_T and u is in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

Compute the efficiency of the algorithm

The answer depends on the **data structures** chosen for the **graph** itself and for the **priority queue** of the set $V - V_T$ whose **vertex priorities** are the distances to the nearest tree vertices.

Prim's Algorithm

ALGORITHM *Prim(G)*

```
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G = (V, E)$ 
//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$ 
 $V_T \leftarrow \{v_0\}$  //the set of tree vertices can be initialized with any vertex
 $E_T \leftarrow \emptyset$ 
for  $i \leftarrow 1$  to  $|V| - 1$  do
    find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$ 
    such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$ 
     $V_T \leftarrow V_T \cup \{u^*\}$ 
     $E_T \leftarrow E_T \cup \{e^*\}$ 
return  $E_T$ 
```

If a **graph** is represented by its **weight matrix** and the **priority queue** is implemented as an **unordered array**

The algorithm's running time will be in $O(|V|^2)$

On each of the $|V| - 1$ iterations, the array implementing the **priority queue is traversed** to **find and delete** the minimum and then to update, if necessary, the priorities of the remaining vertices.

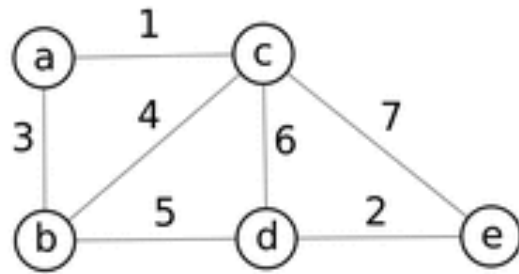
Kruskal's algorithm

MST-KRUSKAL(G, w)

```
1   $A = \emptyset$ 
2  for each vertex  $v \in G.V$ 
3      MAKE-SET( $v$ )
4  sort the edges of  $G.E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u, v) \in G.E$ , taken in nondecreasing order by weight
6      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7           $A = A \cup \{(u, v)\}$ 
8          UNION( $u, v$ )
9  return  $A$ 
```

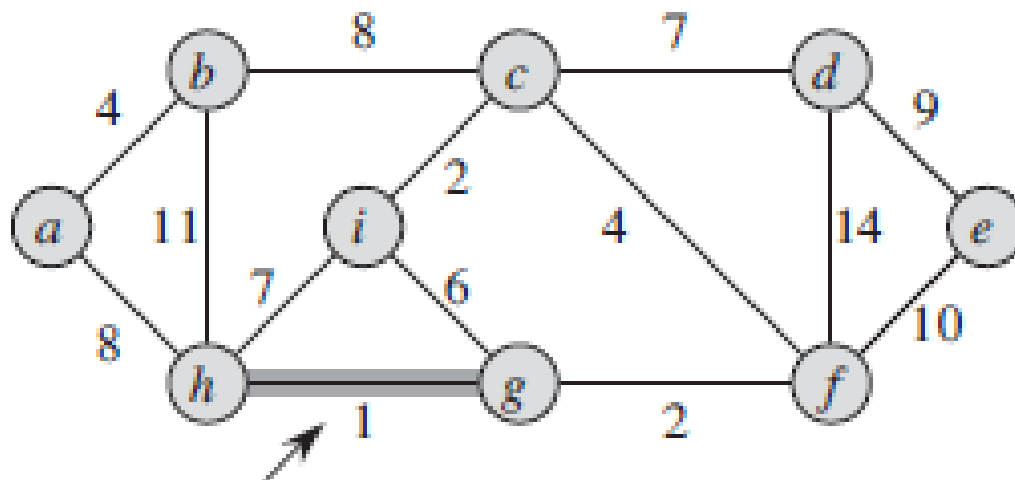
$O(E \lg E)$.

Kruskal's algorithm

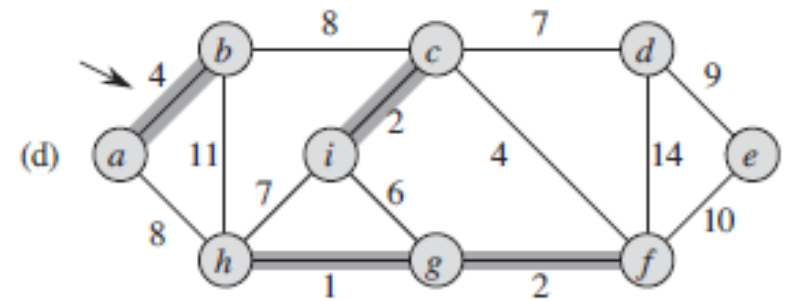
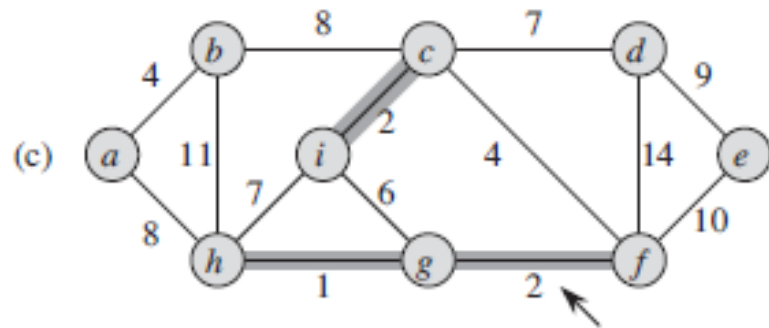
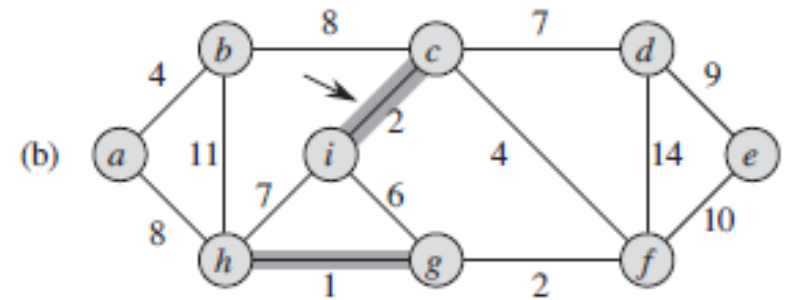
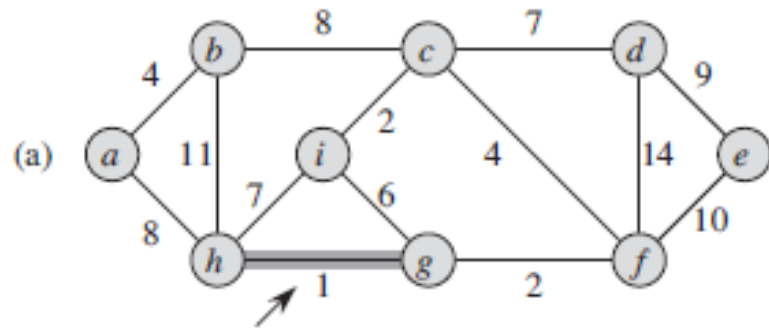


a b c d e

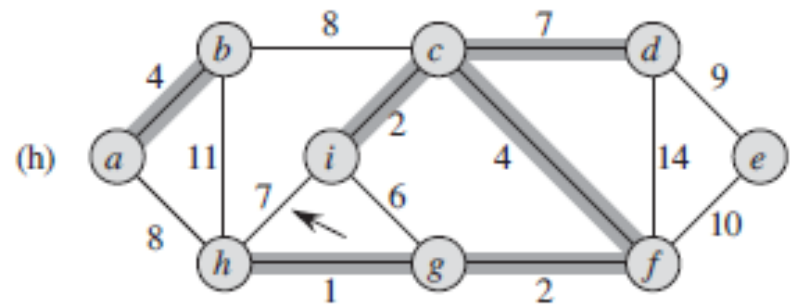
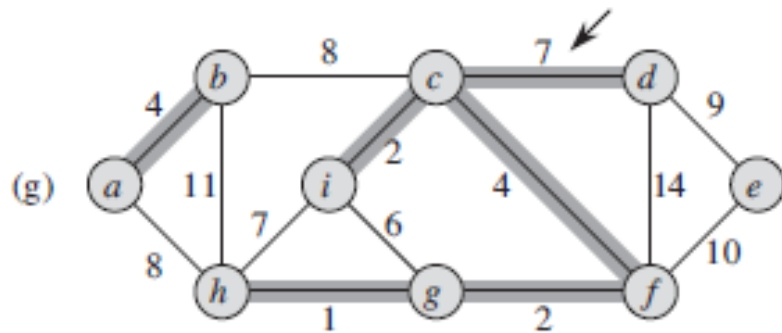
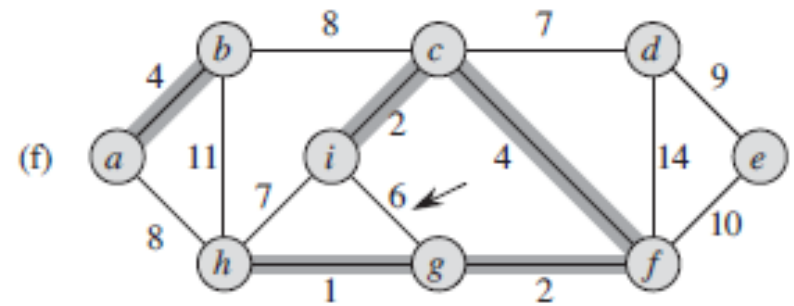
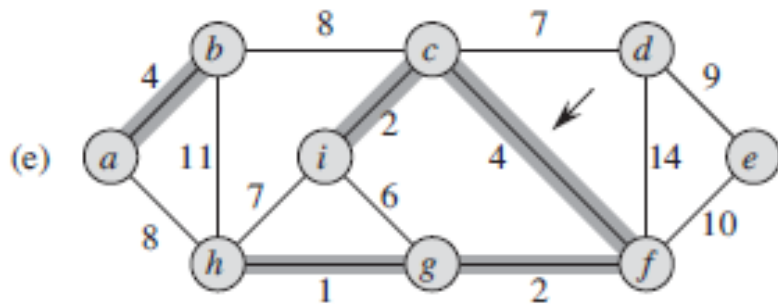
Kruskal's algorithm



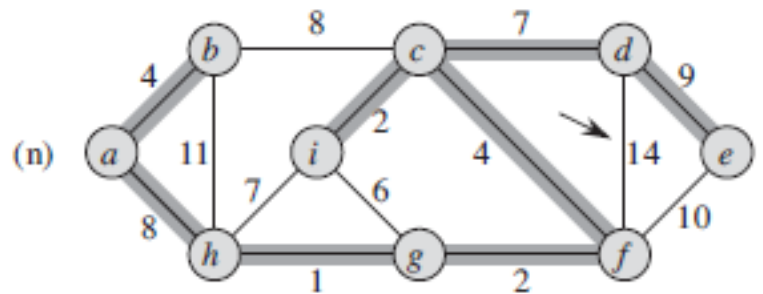
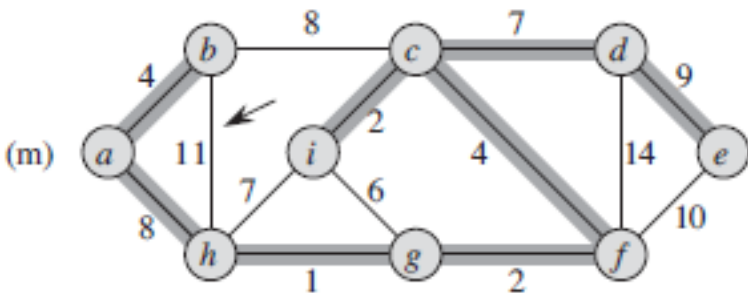
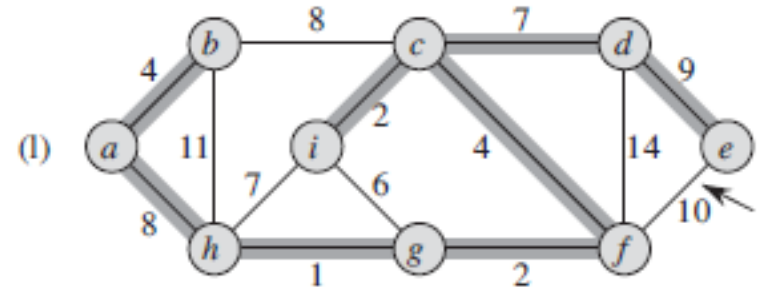
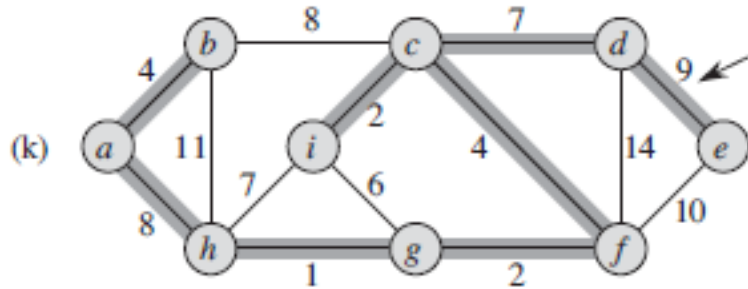
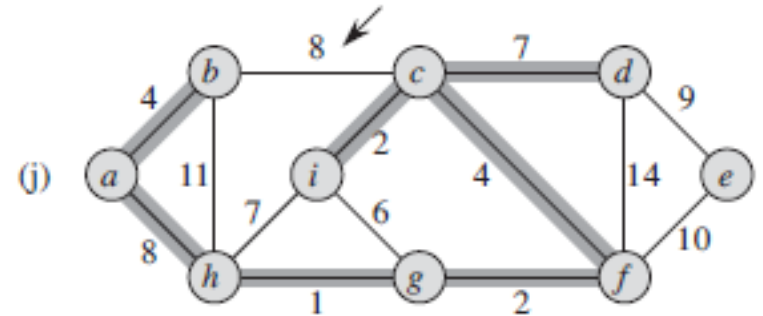
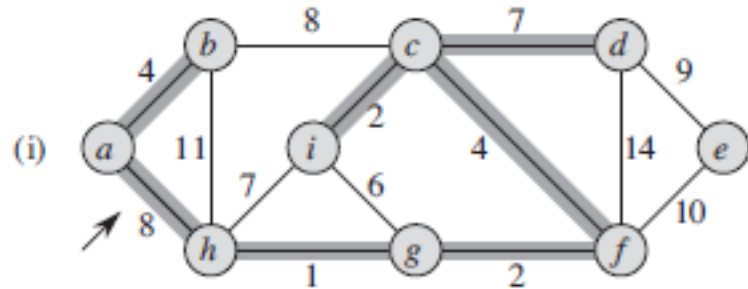
Kruskal's algorithm



Kruskal's algorithm



Kruskal's algorithm



Thank you