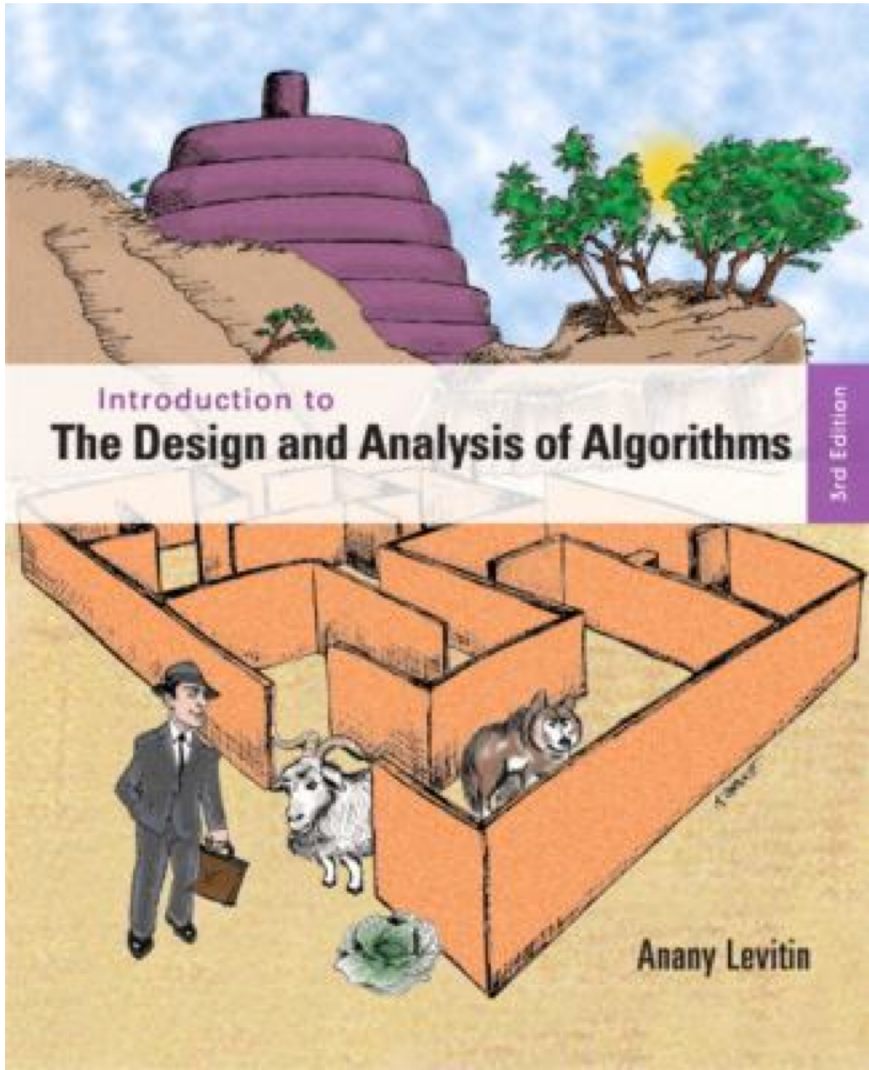
Chapter 9

Greedy Technique



Greedy Technique

- Constructs a solution to an *optimization problem* by making locally optimal choice at each stage with the hope of finding a global optimum

- *Example:* Prim's algorithm for finding a minimum spanning tree for a weighted undirected graph.

- For some problems, yields an optimal solution for every instance. For most, does not but can be useful for fast approximations.

Applications of the Greedy Strategy

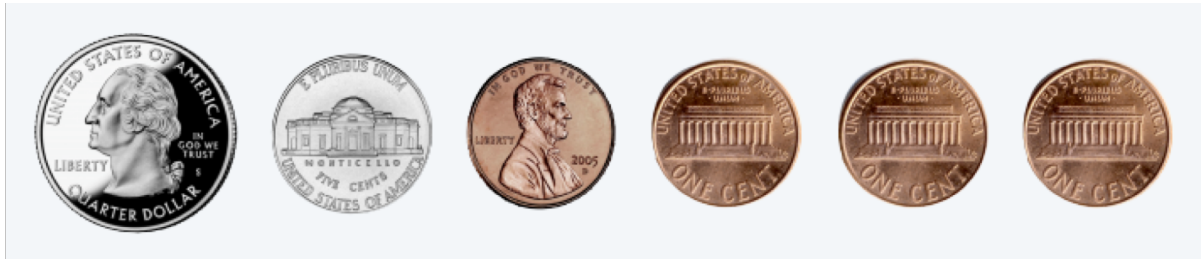
- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Change-Making Problem

Given unlimited amounts of coins of denominations $d_1 < \ldots < d_m$, give change for amount n with the least number of coins

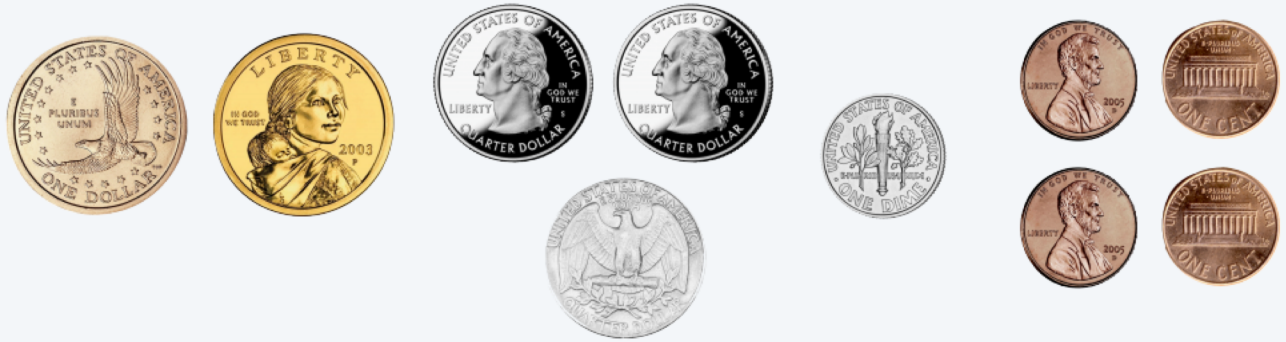
Example: $d_1 = 1c$, $d_2 = 5c$, $d_3 = 10c$, $d_4 = 25c$ and $n = 34c$



Cashier's algorithm

- At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex. $2.89.



Cashier's algorithm

- At each iteration, add coin of the largest value that does not take us past the amount to be paid.

CASHIERS-ALGORITHM $(x, c_1, c_2, \ldots, c_n)$

SORT n coin denominations so that $c_1 < c_2 < \ldots < c_n$

$S \leftarrow \phi$ ⟵ set of coins selected

WHILE $x > 0$

  $k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$

  IF no such k, RETURN "no solution"

  ELSE

    $x \leftarrow x - c_k$

    $S \leftarrow S \cup \{k\}$

RETURN S

- Question: Is cashier's algorithm optimal?

Cashier's algorithm for other denominations

- Is cashier's algorithm for any set of denominations?
- Answer: ?
- Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350.
  - Cashier's algorithm: 140¢.
  - Optimal: 140¢

Cashier's algorithm for other denominations

- Is cashier's algorithm for any set of denominations?
- Answer: No.
- Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350.
  - Cashier's algorithm: 140¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1.
  - Optimal: 140¢ = 70 + 70

Algorithm Comparison

- Time / Space
  - Greedy: O(1) space, O(n) time
  - Dynamic programming: O(n) space and O(m x n) time
- Applicability
  - Greedy: special sets of denominations, e.g. US coins
  - Dynamic programming: any denominations

Minimum Spanning Tree (MST)

- *Spanning tree* of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices

- *Minimum spanning tree* of a weighted, connected graph *G*: a spanning tree of *G* of minimum total weight

Example:

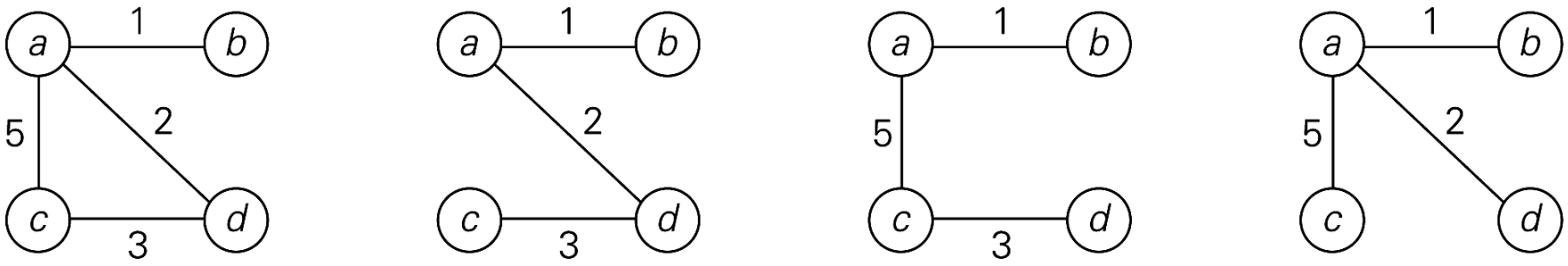


FIGURE 9.1 Graph and its spanning trees; T_1 is the minimum spanning tree

Prim's MST Algorithm

- Start with tree T_1 consisting of one (any) vertex and "grow" tree one vertex at a time to produce MST through a series of expanding subtrees $T_1, T_2, \ldots, T_n$

- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a "greedy" step!)

- Stop when all vertices are included

Refining Prim‘s Algorithm

1. All vertices are marked as not visited

2. Any vertex *v* you like is chosen as starting vertex and is marked as visited (define a cluster *C*)

3. The smallest-weighted edge $e = (v, u)$, which connects one vertex *v* inside the cluster *C* with another vertex *u* outside of *C*, is chosen and is added to the MST.

4. The process is repeated until a spanning tree is formed

Illustration of Prim's Algorithm

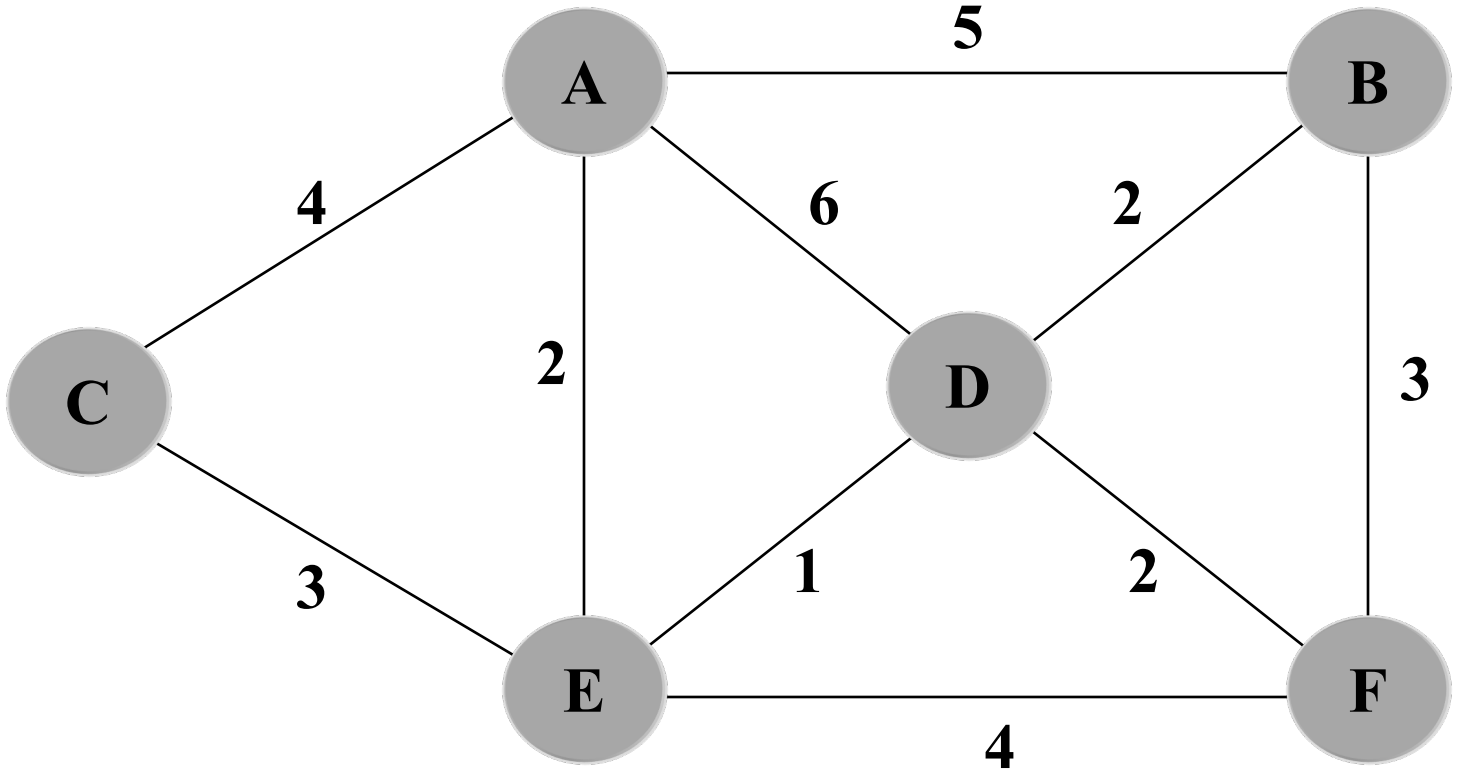


Illustration of Prim's Algorithm

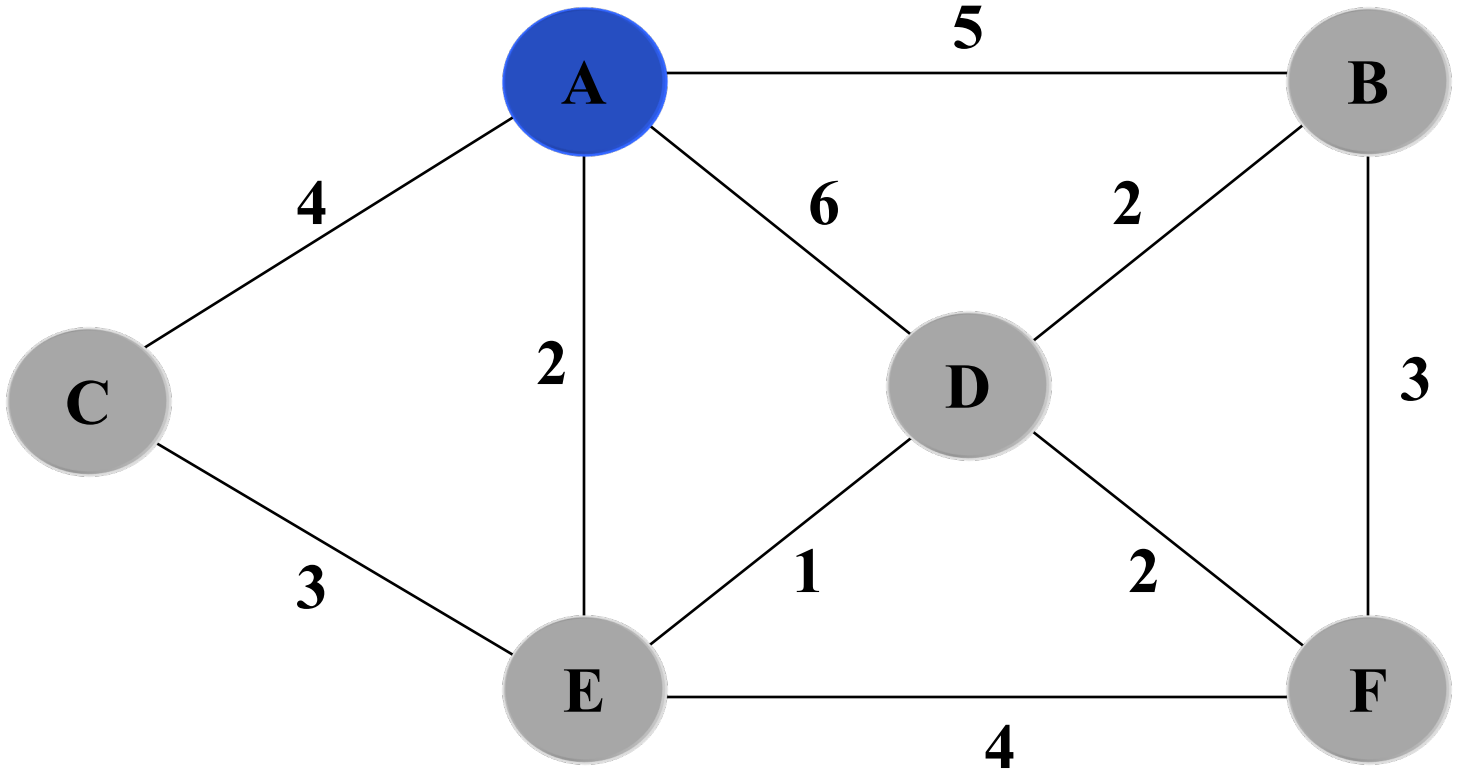


Illustration of Prim's Algorithm

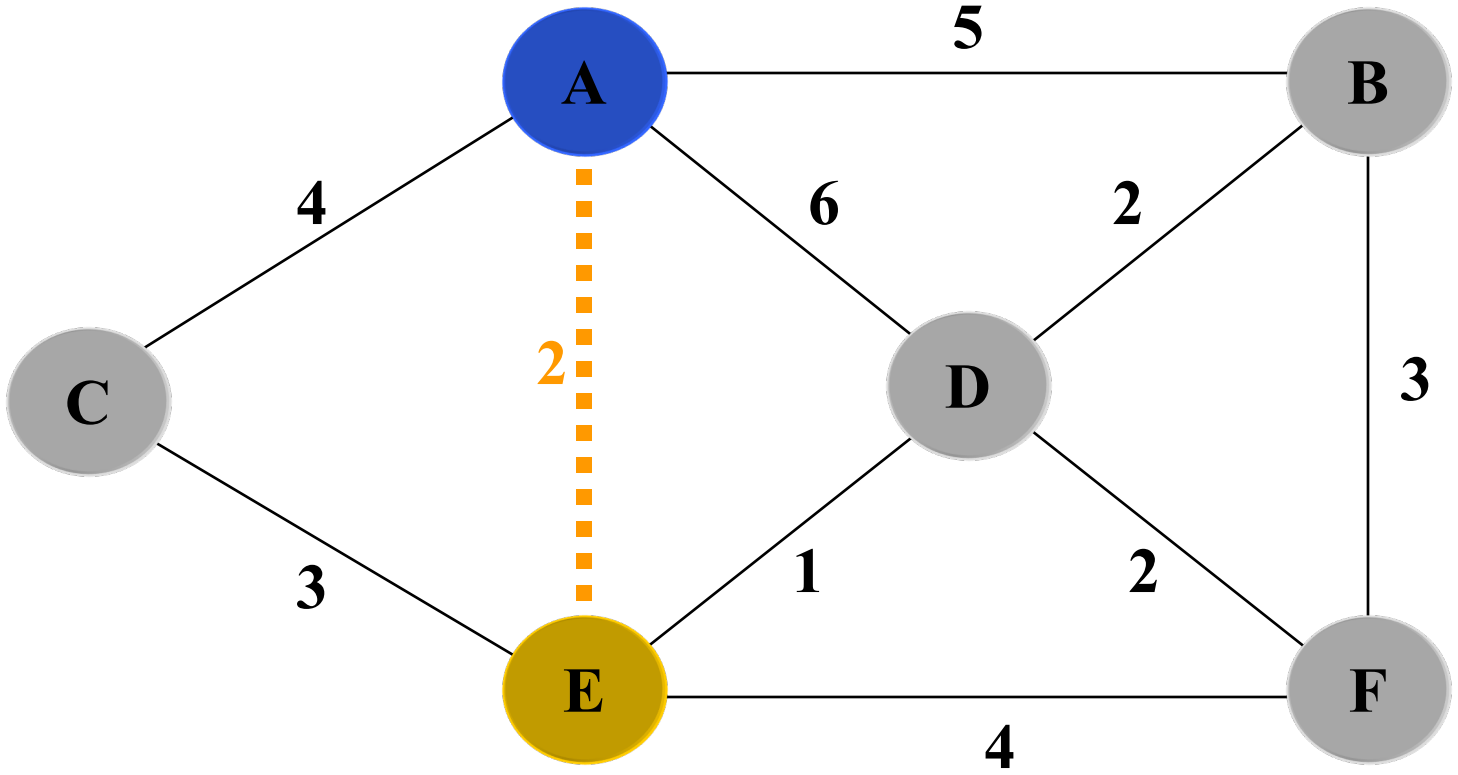


Illustration of Prim's Algorithm

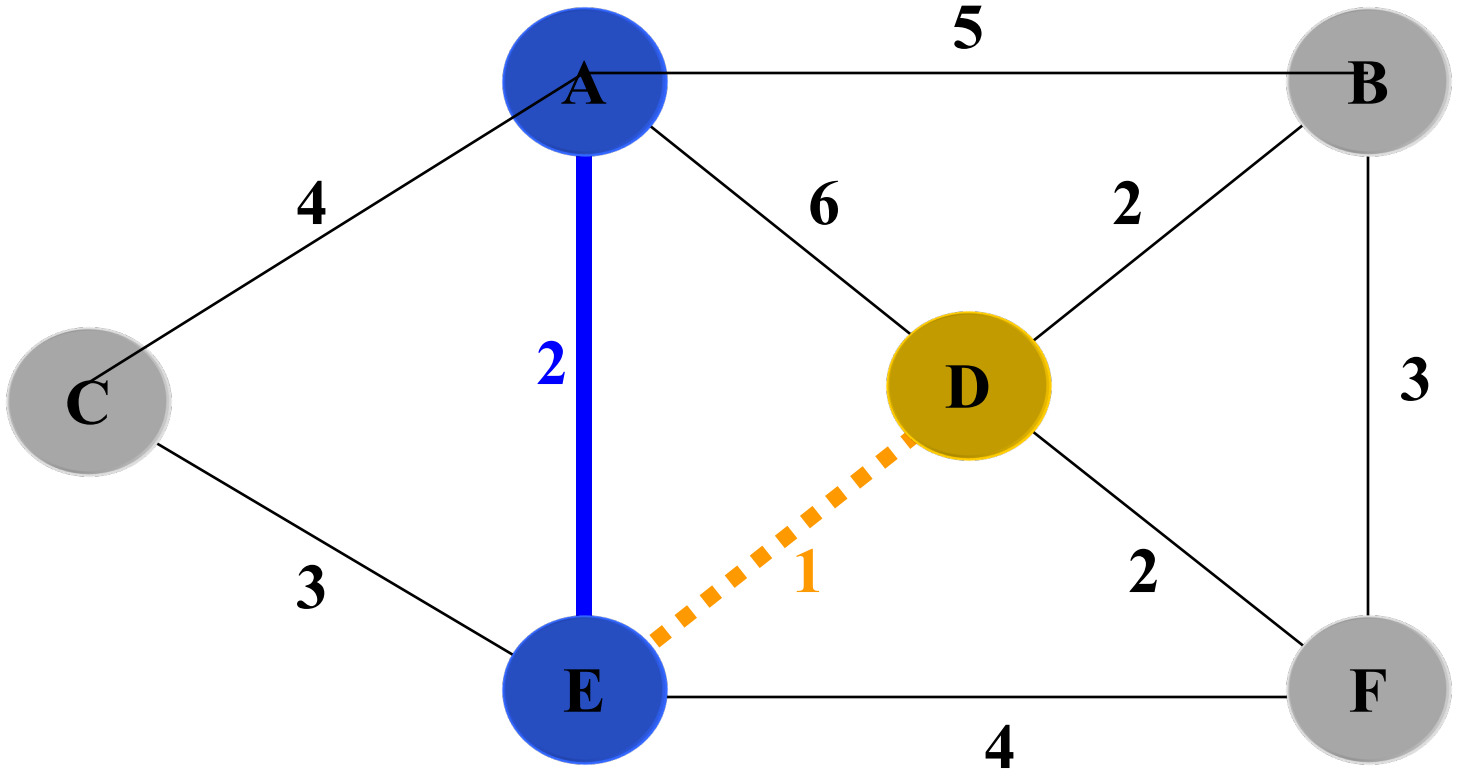


Illustration of Prim's Algorithm

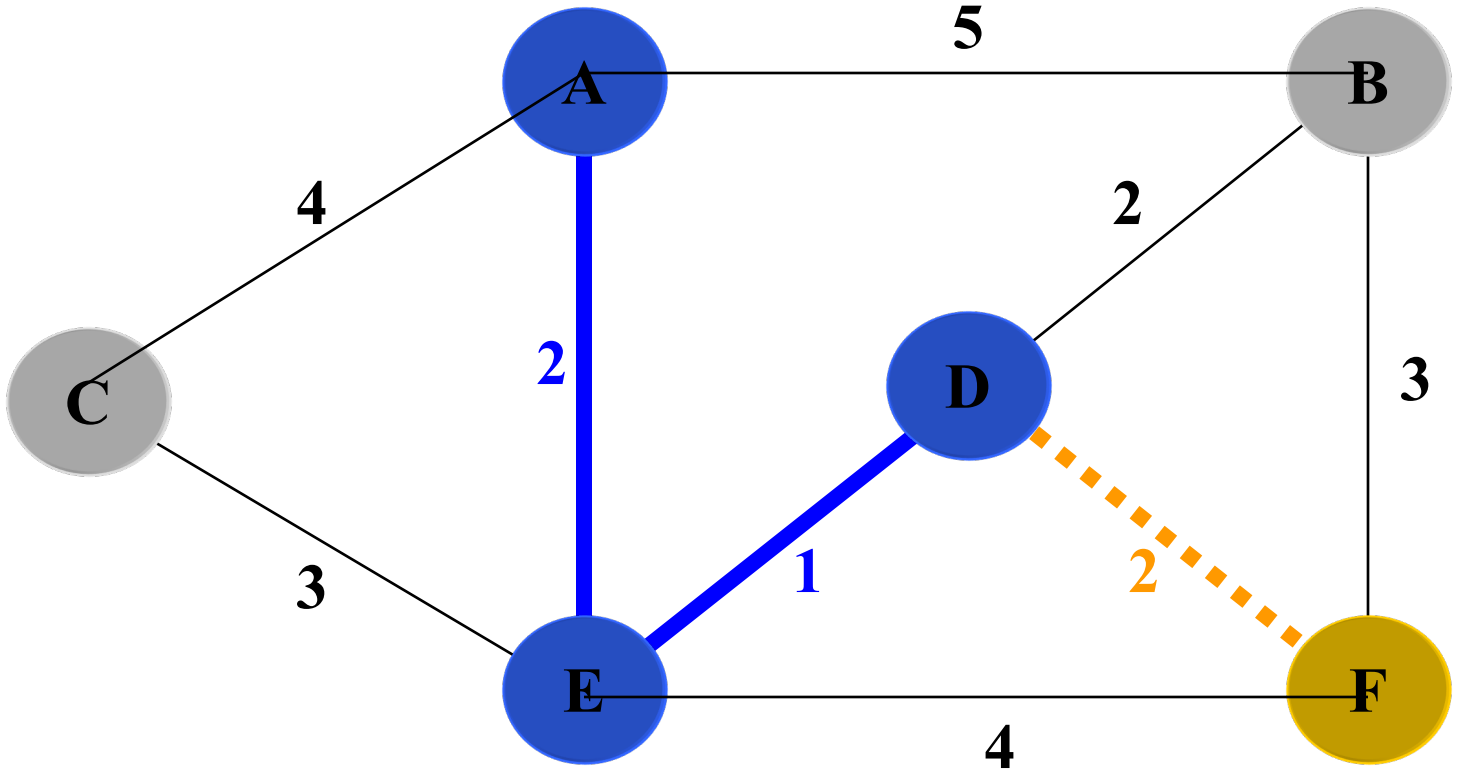


Illustration of Prim's Algorithm

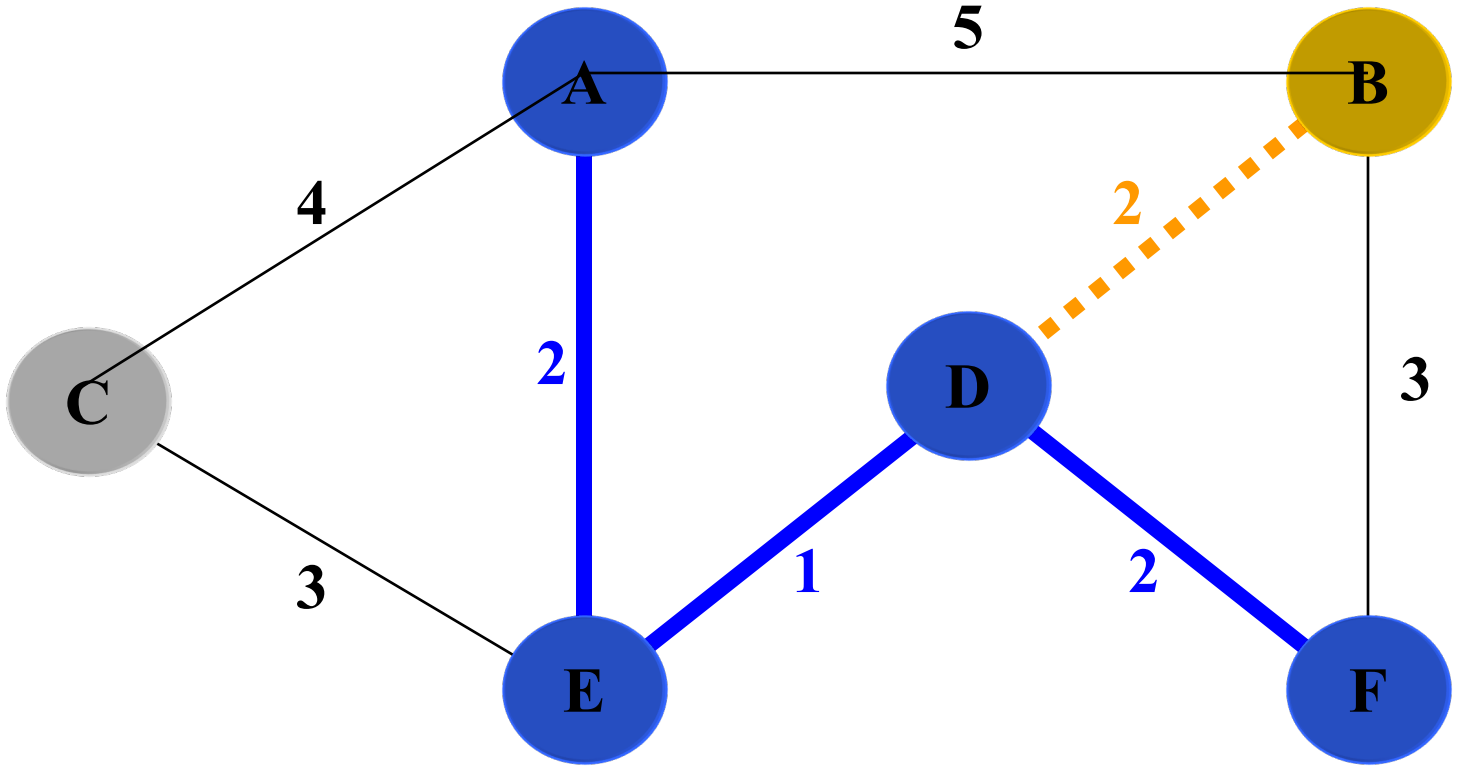


Illustration of Prim's Algorithm

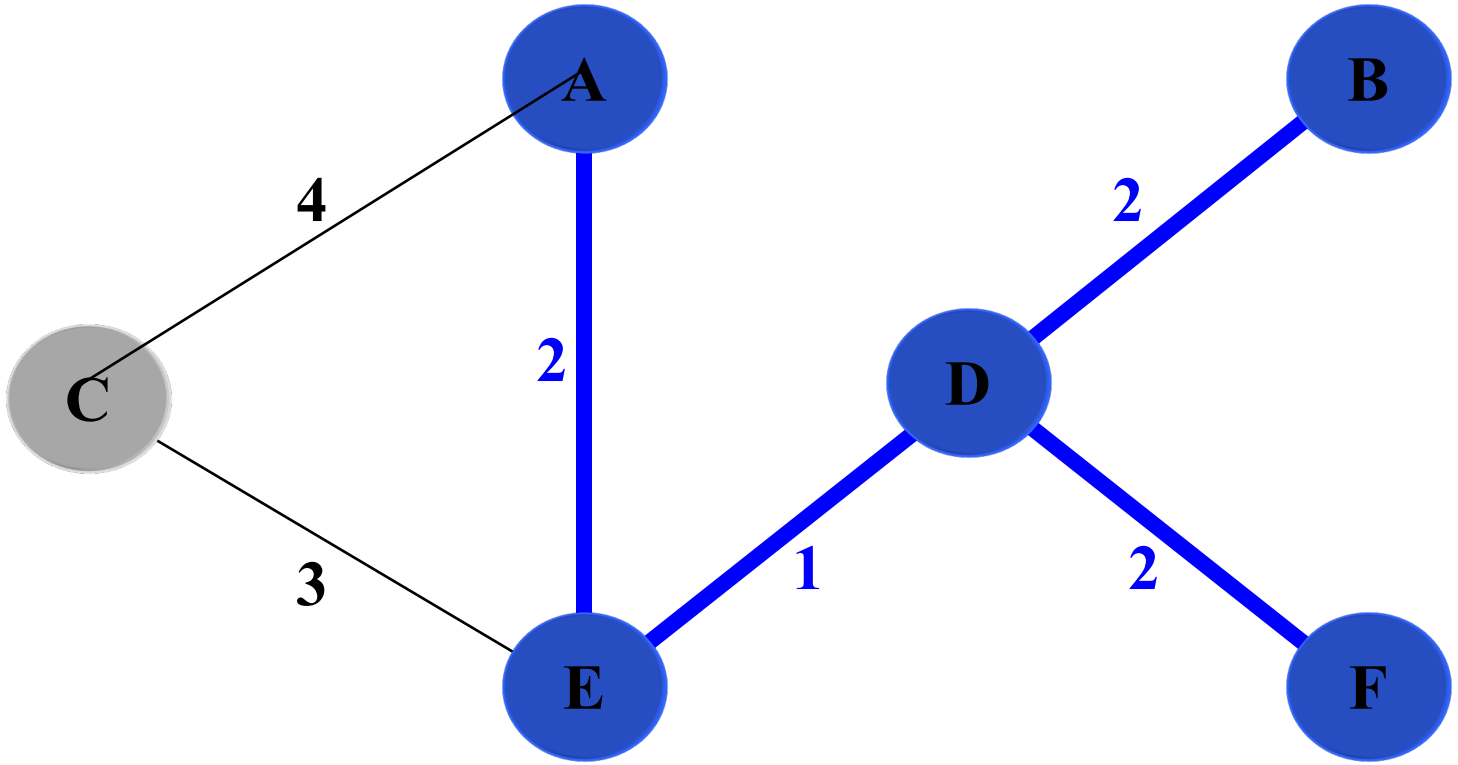


Illustration of Prim's Algorithm

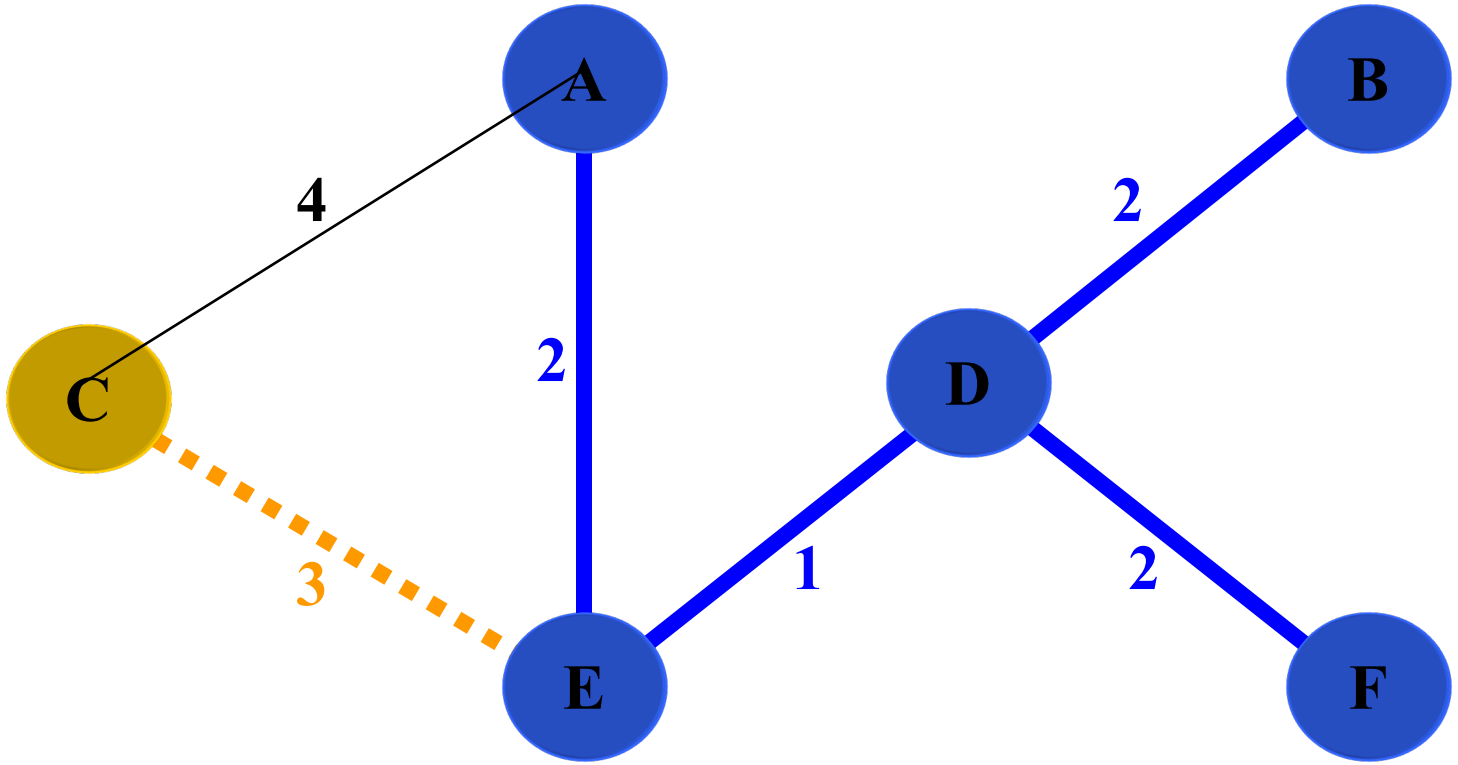
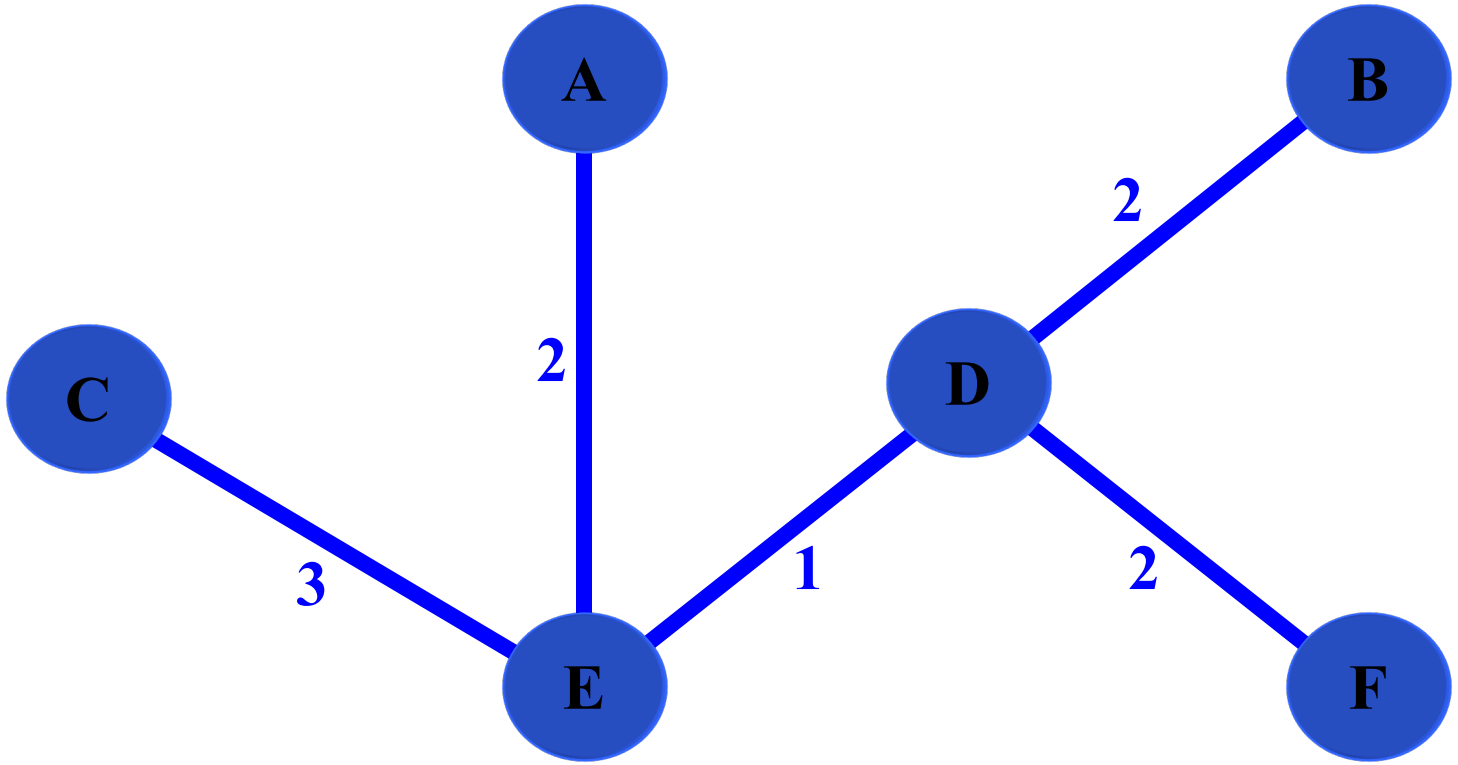


Illustration of Prim's Algorithm

Minimum- Spanning Tree



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

Complexity???

Complexity

- If implemented using adjacency lists and min-heap
  - Time: O(|E| log|V|)

Proof of Correctness of Prim's MST Algorithm

Claim: For $1 \le i \le |V|$, let T_i be the partially built tree in Prim's algorithm on input G = (V, E) after the i^{th} iteration (T_i has i vertices.). T_i is contained in some minimum spanning tree of G.

Proof by induction on i

- **Basis step**: i = 1. T_1 contains one vertex and no edges. Since every minimum spanning tree of G contains every vertex in V, T_1 is clearly contained in some minimum spanning tree of G.

- **Induction hypothesis**: Suppose that T_i is contained in some minimum spanning tree of G, for some fixed $1 \le i \le |V| - 1$.

Proof of Prim's MST Algorithm (cont.)

- **Induction step**: We need to show that T_{i+1} is contained in some minimum spanning tree of G.
  - Let the edge (v, w) be the one that Prim's algorithm chooses and adds to its partially built tree. Note that v is in the known set and w is not in the known set and that T_{i+1} is T_i with the vertex w and the edge (v, w) added.
  - By the induction hypothesis, T_i is contained in some minimum spanning tree. Call this minimum spanning tree T_i^*. If T_i^* contains the edge (v, w), then T_{i+1} is contained in the minimum spanning tree T_i^* and we are done with the induction step.

Proof of Prim's MST Algorithm (cont.)

- So suppose (v, w) is not in T_i*. We now show how to construct a minimum spanning tree that contains T_{i+1}. T_i* contains the edges in T_i. Since T_i * is a tree that spans G, then there is some edge not equal to (v, w), call this edge e, that connects the vertices of T_i to the rest of T_i *. Note that e has one endpoint in the known set after iteration i and one endpoint not in the known set. The following figure illustrates the situation.

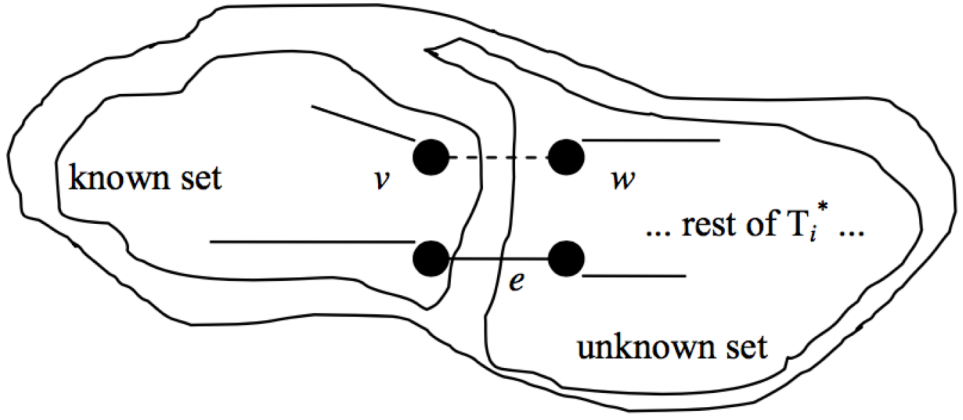


Figure 1. The situation after the *i*th iteration in Prim's algorithm. The known set comprises T_i. The solid black edges are in T_i^*. Prim's algorithm chooses edge (v, w) to add to T_i. There may be other edges from the known set to the unknown set.

Proof of Prim's MST Algorithm (cont.)

- The cost of e cannot be less than that of (v, w) because it would contradict the fact that Prim's algorithm chooses an edge that connects a known vertex to an unknown vertex with least cost. That is, if e had cost less than (v, w), then Prim's algorithm would have chosen e rather than (v, w) to add to its partially built tree.
- The cost of e cannot be greater than that of (v, w) because in T_i^* if we replace e with (v, w), then we have a spanning tree of G with a cost less than that of T_i^*(we have replaced e with an edge with less cost), which contradicts the fact that T_i^* is a minimum spanning tree.

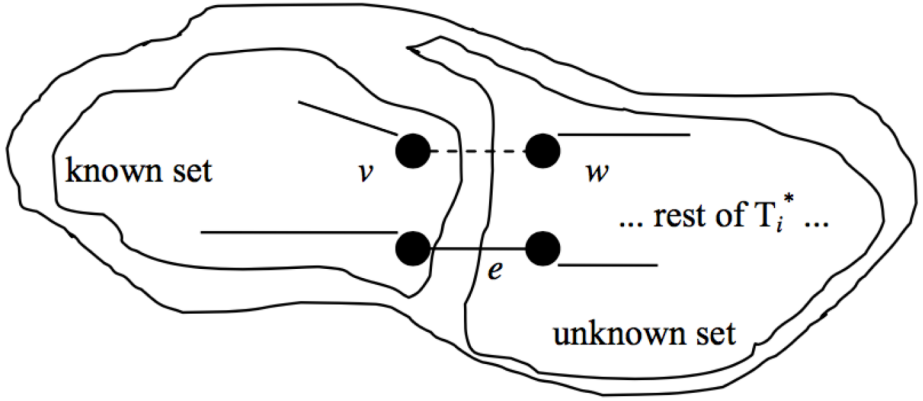


Figure 1. The situation after the *i*th iteration in Prim's algorithm. The known set comprises T_i. The solid black edges are in T_i^*. Prim's algorithm chooses edge (v, w) to add to T_i. There may be other edges from the known set to the unknown set.

Proof of Prim's MST Algorithm (cont.)

- So, e's cost must be the same as that of (v, w). If we take T_i^* and replace e with (v, w), we have a spanning tree of G with cost the same as that of T_i^*. Hence, this new tree is a minimum spanning tree and it contains T_i and (v, w). That is, this minimum spanning tree contains T_{i+1}.

- This concludes the proof by induction.

Another greedy algorithm for MST: Kruskal's

- Sort the edges in increasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

Illustration of Kruskal's Algorithm

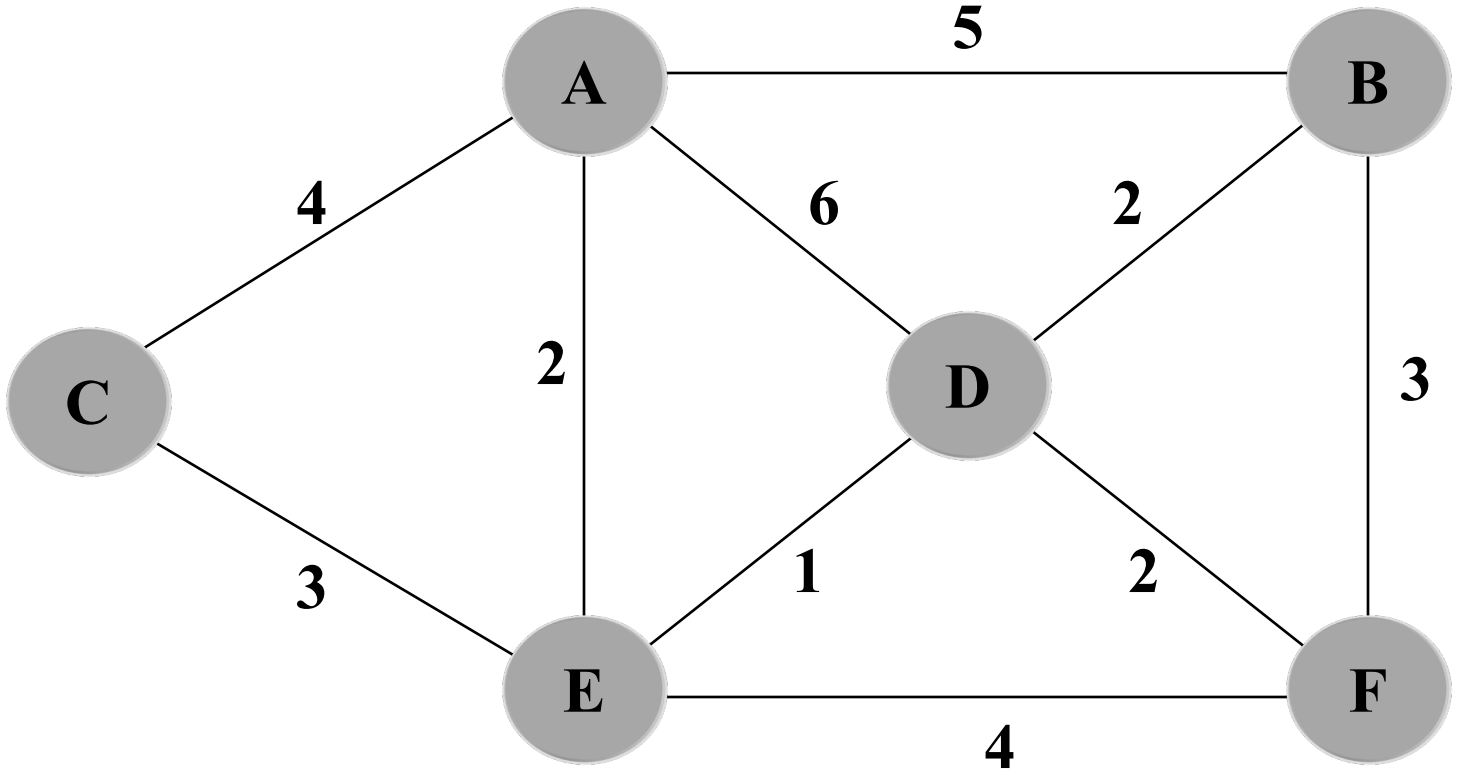


Illustration of Kruskal's Algorithm

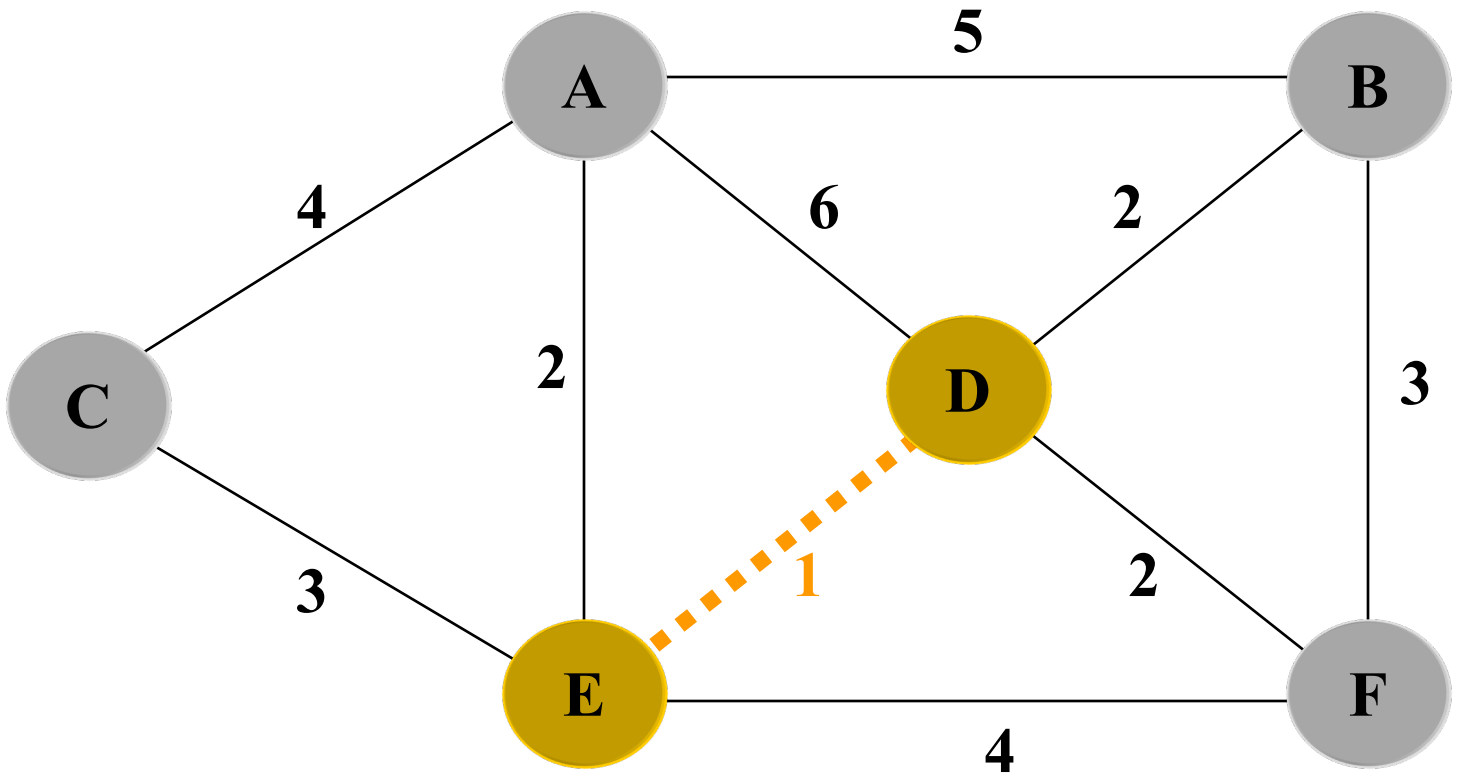


Illustration of Kruskal's Algorithm

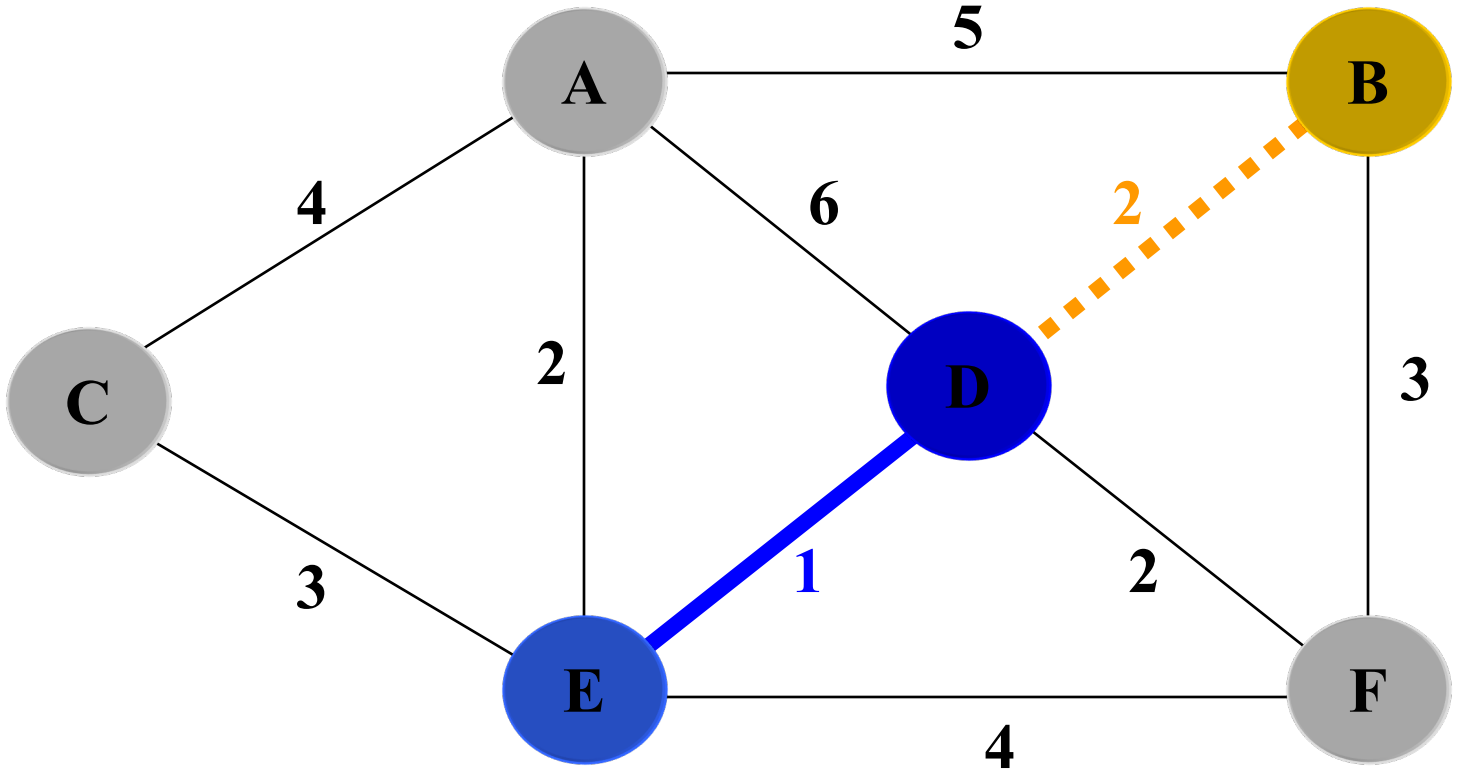


Illustration of Kruskal's Algorithm

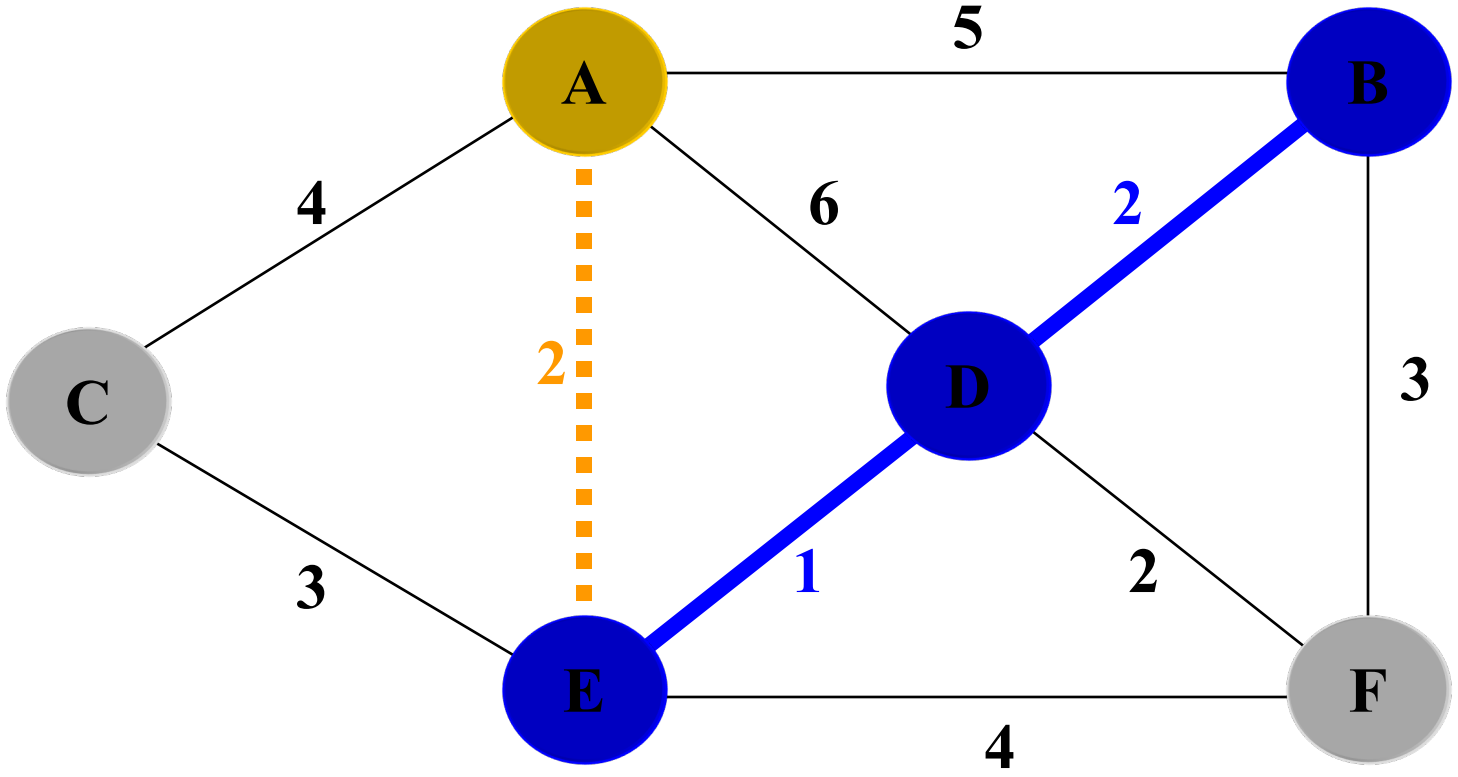


Illustration of Kruskal's Algorithm

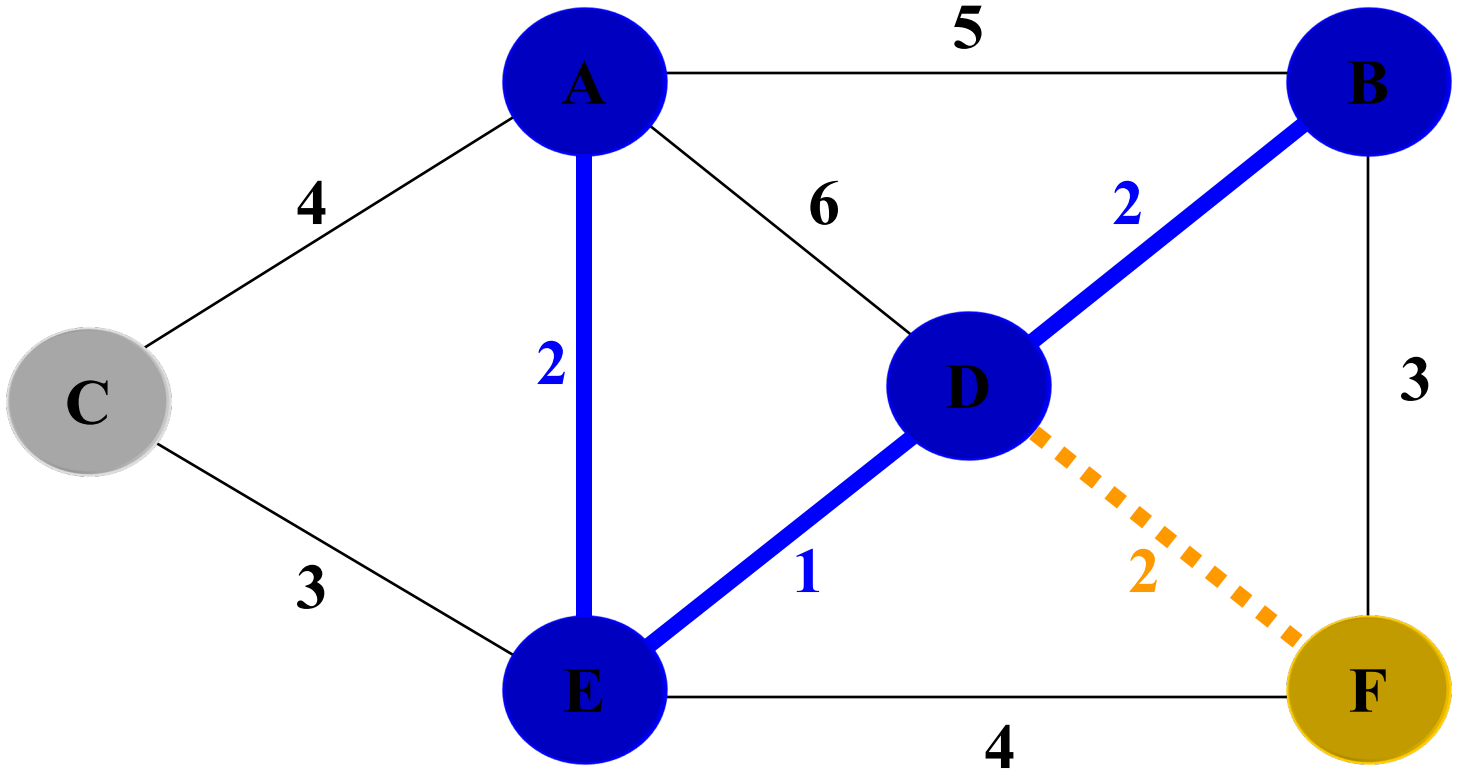


Illustration of Kruskal's Algorithm

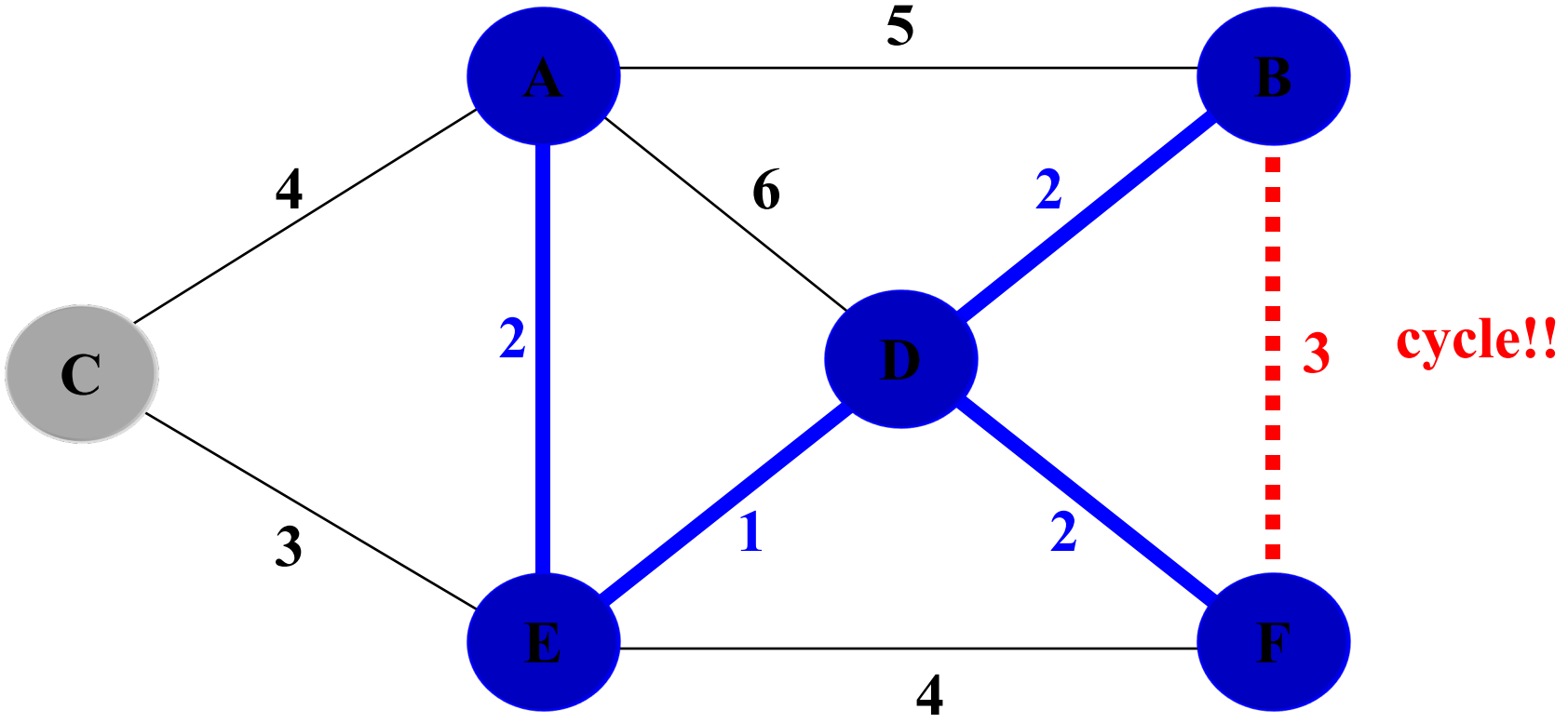


Illustration of Kruskal's Algorithm

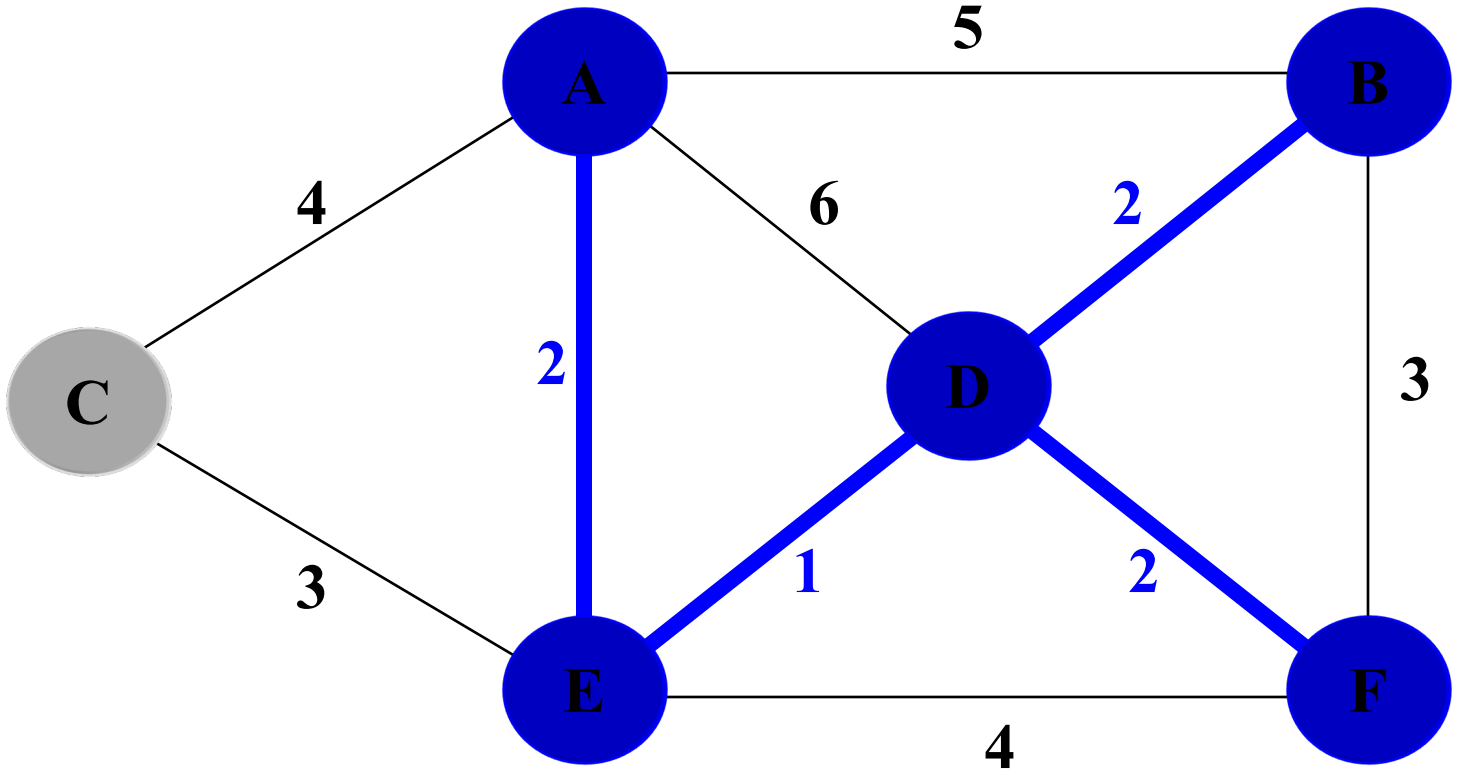


Illustration of Kruskal's Algorithm

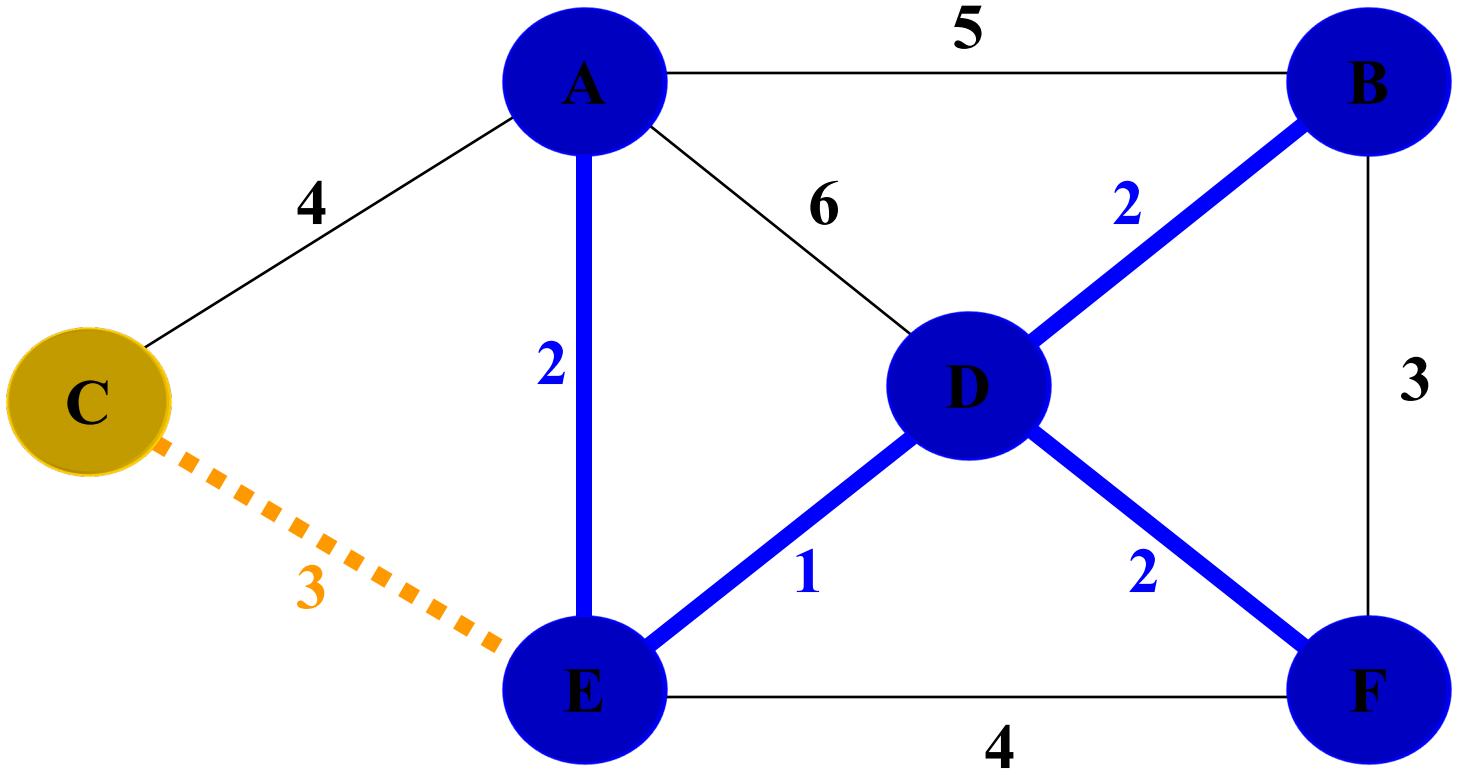
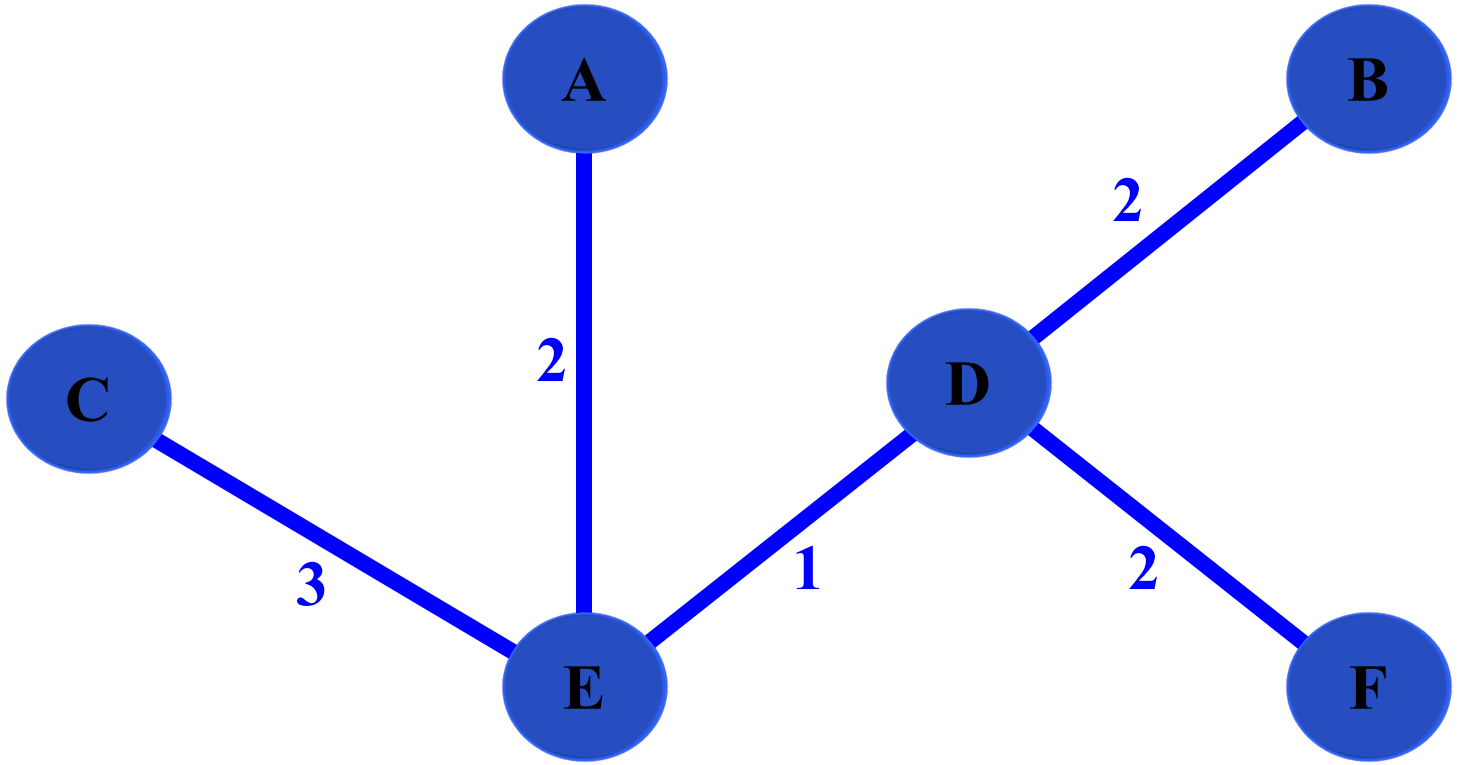


Illustration of Kruskal's Algorithm

Minimum- Spanning Tree



Kruskal's Algorithm

ALGORITHM *Kruskal*(G)

//Kruskal'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
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \ldots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \emptyset$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

Notes about Kruskal's algorithm

- Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)

- Cycle checking: a cycle is created iff added edge connects vertices in the same connected component

Disjoint-set Data Structure

- More efficient approach:
  - Employ *Disjoint Subsets and Union-Find* Algorithms

- First studied by Bernerd Galler and Michael J. Fischer in 1964

Bernie Galler, 2004

Professor, University of Michigan

Michael J. Fischer
Professor, Yale Univ.

Disjoint-set Data Structure

- Two sets are said to be **disjoint** if they have no elements in common
- A set of elements partitioned into a number of disjoint subsets
- Initially, each element *e* is a set in itself:
  - e.g., { $\{e_1\}$, $\{e_2\}$, $\{e_3\}$, $\{e_4\}$, $\{e_5\}$, $\{e_6\}$, $\{e_7\}$}

Operations: Union

- Union(*x*, *y*) – Combine or merge two sets *x* and *y* into a single set
  - Before:

    {{e_3, e_5, e_7} , {e_4, e_2, e_8}, {e_9}, {e_1, e_6}}

  - After Union(e_5, e_1):

    {{e_3, e_5, e_7, e_1, e_6} , {e_4, e_2, e_8}, {e_9}}

Operations: Find

- Determine which set a particular element is in
  - Useful for determining if two elements are in the same set

- Each set has a unique name
  - name is arbitrary; what matters is that find(*a*) == find(*b*) is true only if *a* and *b* in the same set
  - one of the members of the set is the "representative" (i.e. name) of the set: {{e_3, e_5, e_7, e_1, e_6} , {e_4, e_2, e_8}, {e_9}}

Operations: Find (cont.)

- Find(*x*) – return the name of the set containing *x*.
- Examples
  - {{e_3, e_5, e_7, e_1, e_6} , {e_4, e_2, e_8}, {e_9}}
  - Find(e_1) = e_5
  - Find(e_4) = e_8

Kruskal's Algorithm Implementation (Revisited)

Kruskals():

sort edges in increasing order of length (e_1, e_2, e_3, ..., e_m).

initialize disjoint sets.

T := {}.

for i = 1 to m

let e_i = (u, v).

if find(u) != find(v)

union(find(u), find(v)).

add e_i to T.

return T.

What does the disjoint set initialize to?
How many times do we do a union?
How many time do we do a find?
What is the total running time if we have *n* nodes and *m* edges?

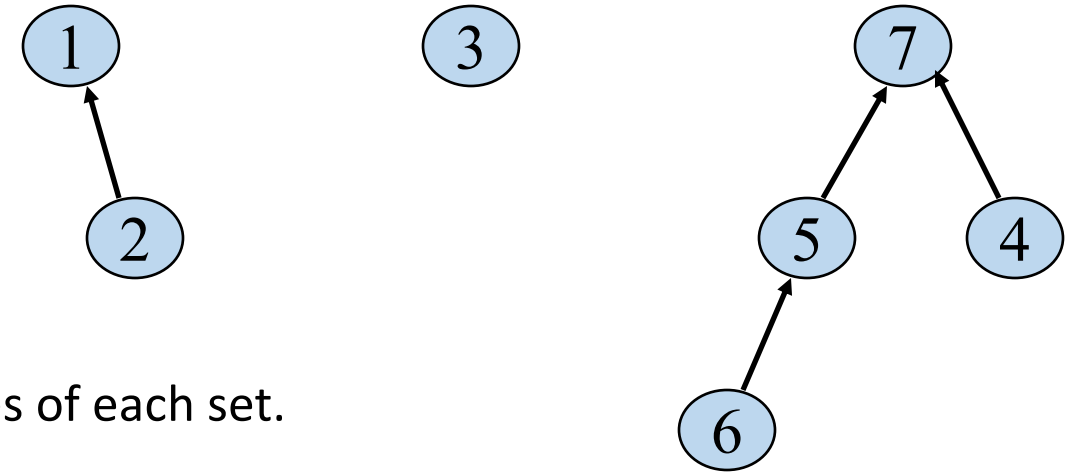
Disjoint Sets with Trees

- Observation: *trees* let us find many elements given one root (i.e. representative)…
- Idea: if we *reverse* the pointers (make them point up from child to parent), we can find a single root from many elements…
- Idea: Use one tree for each subset. The name of the class is the tree root.

Up-Tree for Disjoint Sets

Initial state

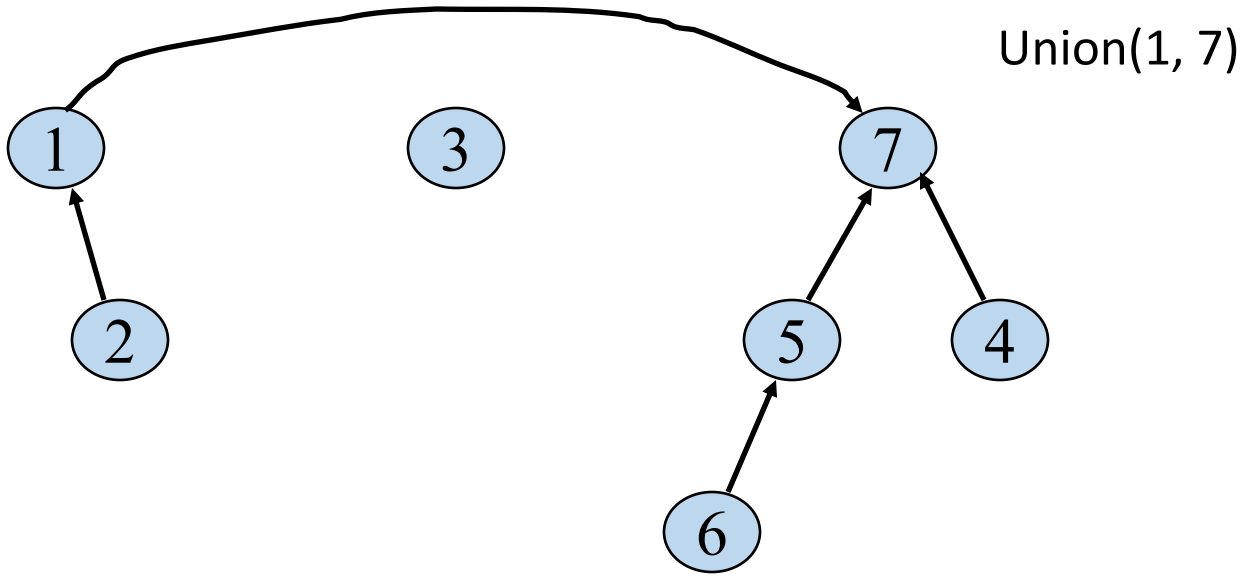
Intermediate state



Roots are the names of each set.

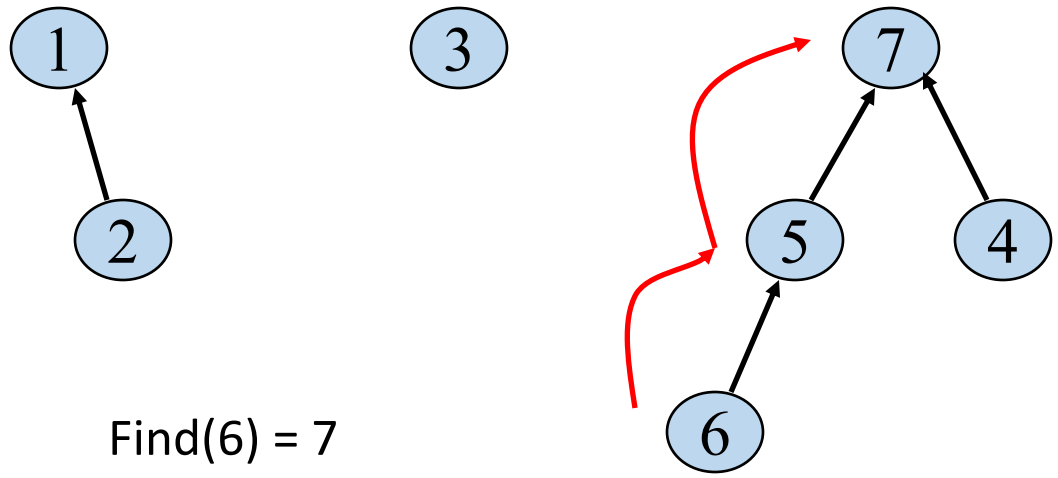
Union Operation

- Union(x, y) – assuming x and y roots, point x to y.



Find Operation

- Find(*x*): follow *x* to root and return root



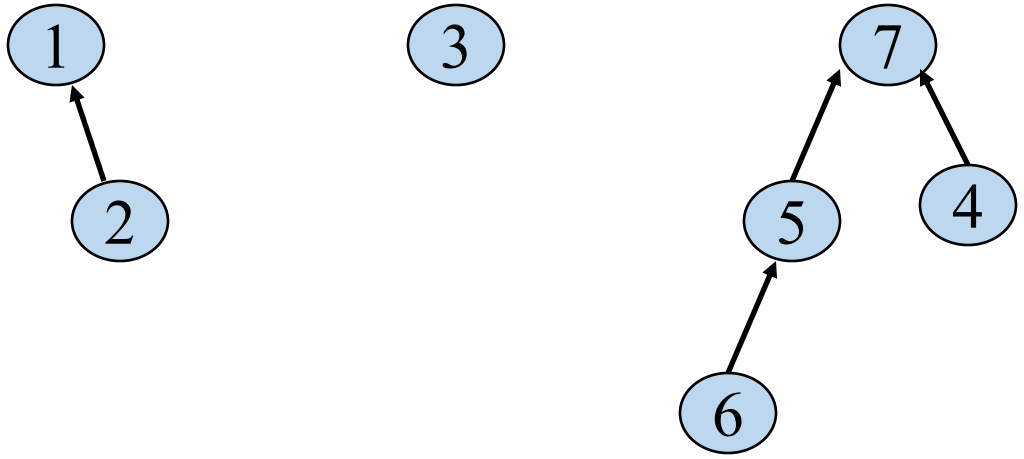
Find(6) = 7

Simple Implementation

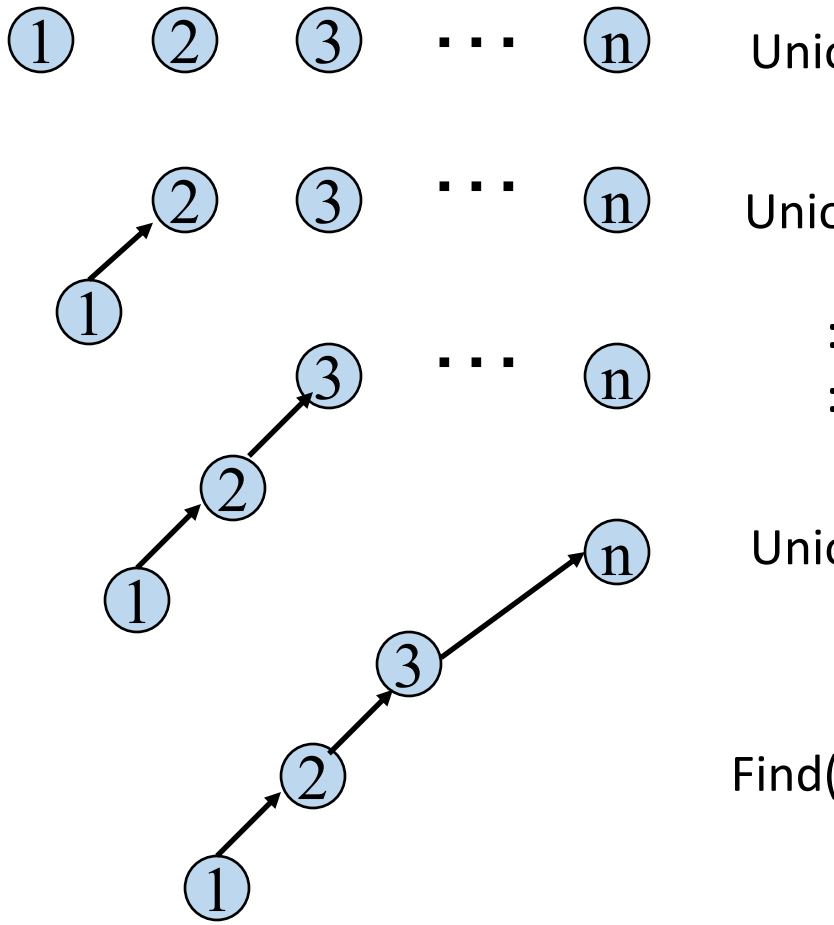
- Array of indices

	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0

Up[x] = 0 means
x is a root.



A Bad Case



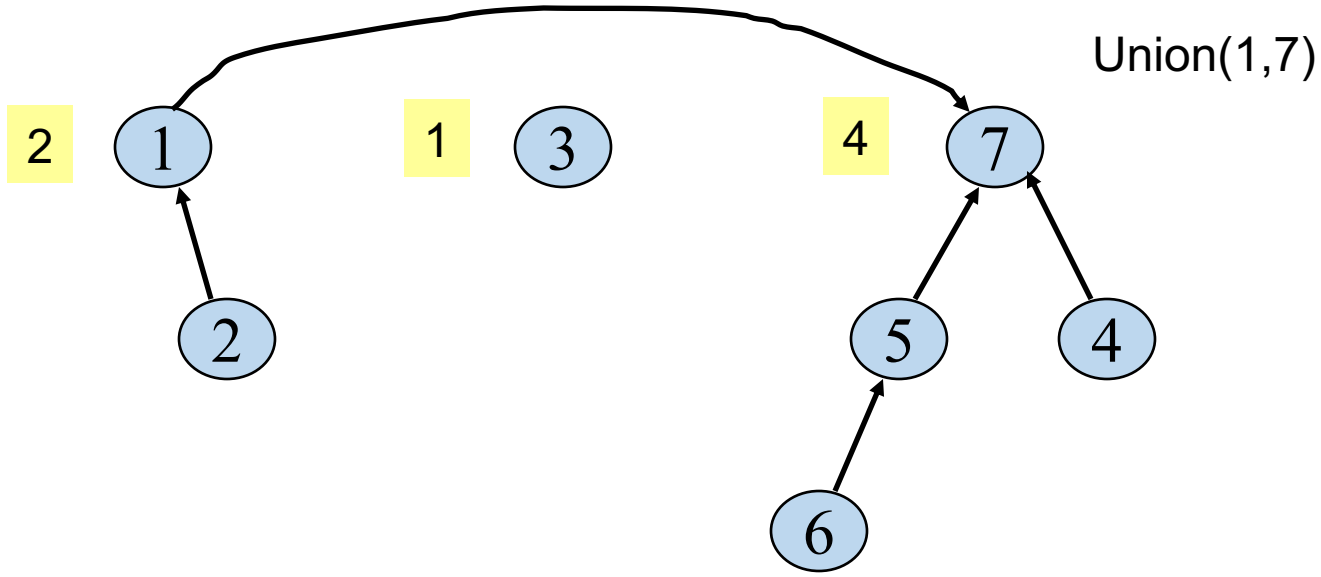
Improving Find

1. Improve union so that *find* only takes $\Theta(\log n)$
   - Union-by-size
   - Reduces complexity to $\Theta(m \log n + n)$

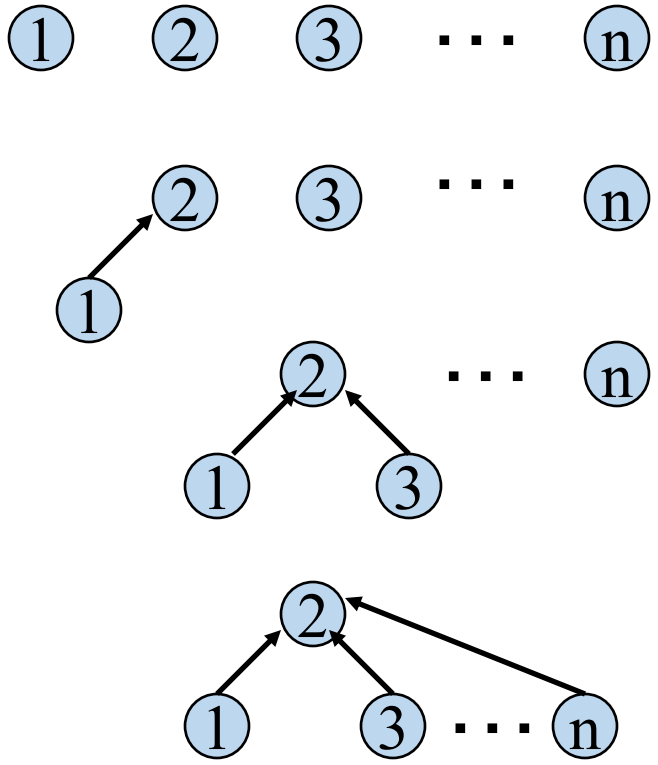
2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to <u>almost</u> $\Theta(m + n)$

Union by Rank

- Union by Rank (also called Union by Size)
  - Always point the smaller tree to the root of the larger tree



Example Again



Union(1,2)

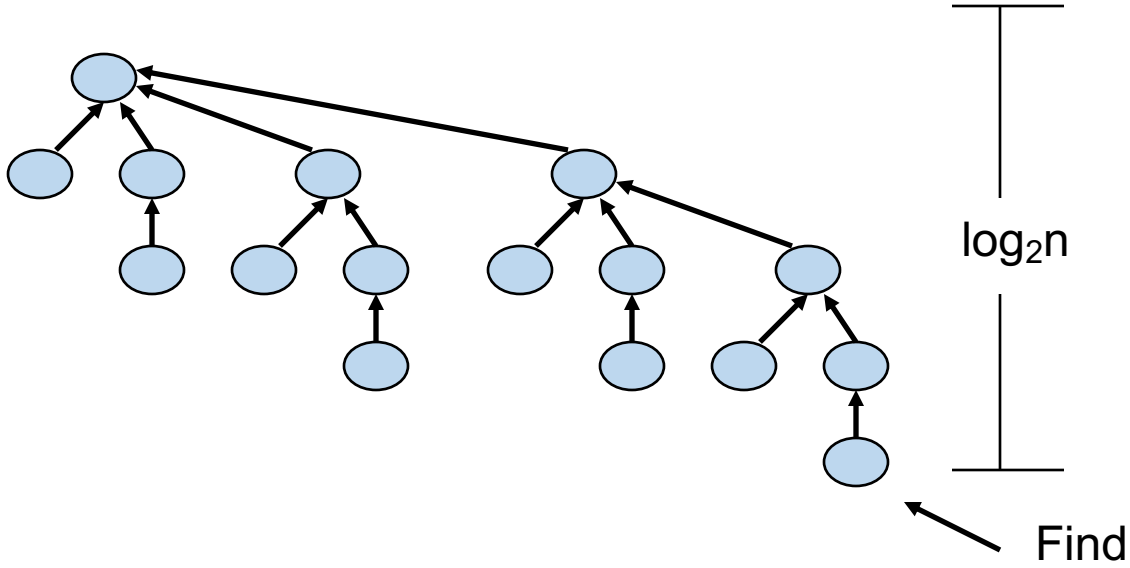
Union(2,3)

Union(n-1,n)

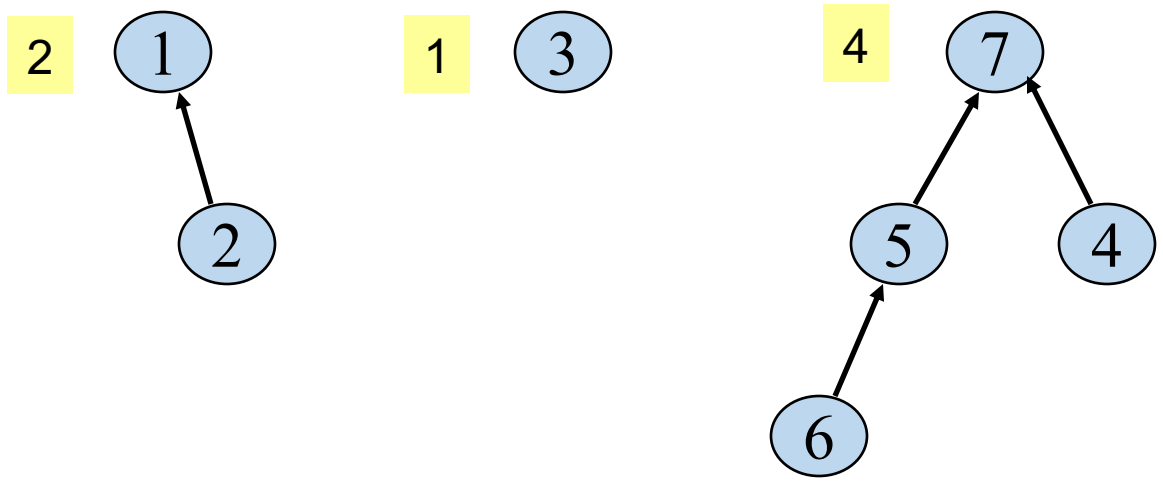
Find(1) constant time

Improved Runtime for Find via Union by Rank

- Depth of tree affects running time of Find
- Union by rank only increases tree depth if depth were equal
- Results in O(log *n*) for Find



Elegant Array Implementation



	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0
weight	2		1				4

Union by Rank

```
Algorithm Union(index i,j)
  //i and j are roots
  w_i <- weight[i]
  w_j <- weight[j]
  if w_i < w_j then
    up[i] <- j
    weight[j] <- w_i + w_j
  else
    up[j] <- i
    weight[i] <- w_i + w_j
```

Kruskal's Algorithm Implementation (Revisited)

Kruskals():
  sort edges in increasing order of length (e_1, e_2, e_3, ..., e_m).

  initialize disjoint sets.

  T := {}.

  for i = 1 to m
    let e_i = (u, v).
    if find(u) != find(v)
      union(find(u), find(v)).
      add e_i to T.

  return T.

How many times do we do a union?
How many time do we do a find?
What is the total running time if we have *n* nodes and *m* edges?

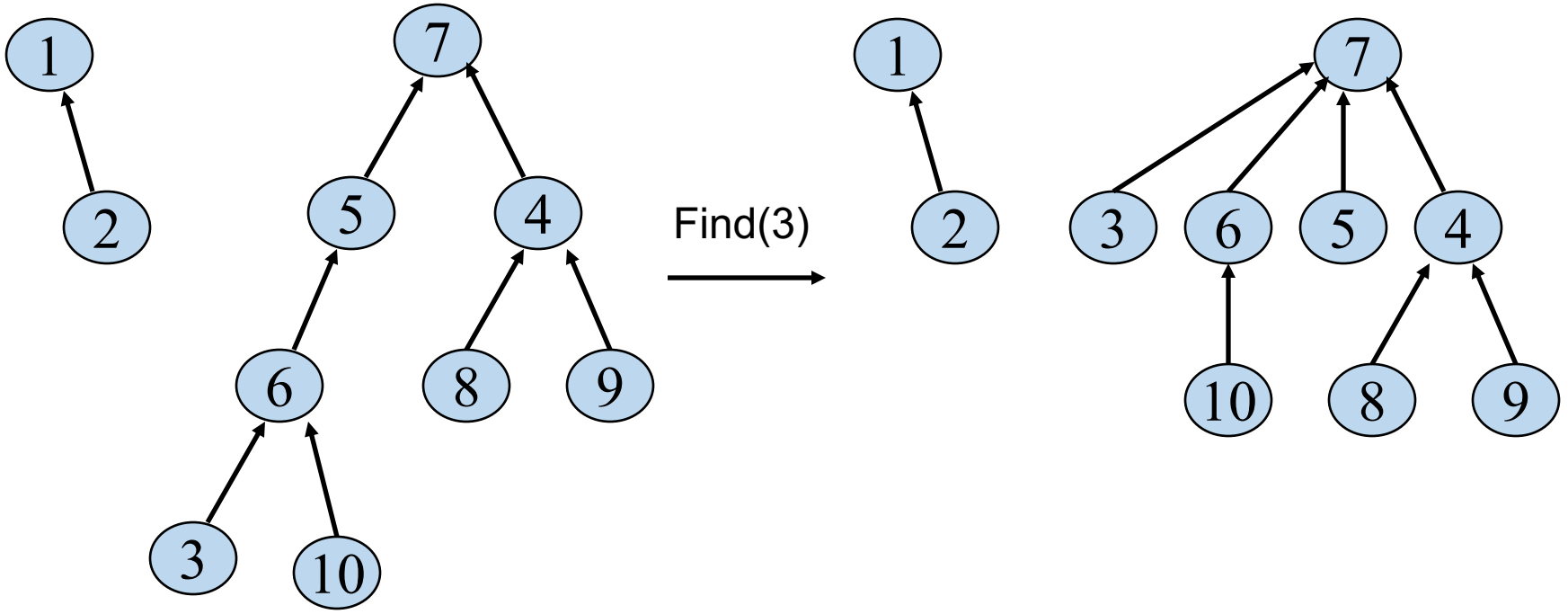
Kruskal's Algorithm Running Time

- Assuming |E| = m edges and |V| = n nodes
- Sort edges: O(m log m)
- Initialization: O(n)
- Finds: O(2 * m * log n) = O(m log n)
- Unions: O(m)

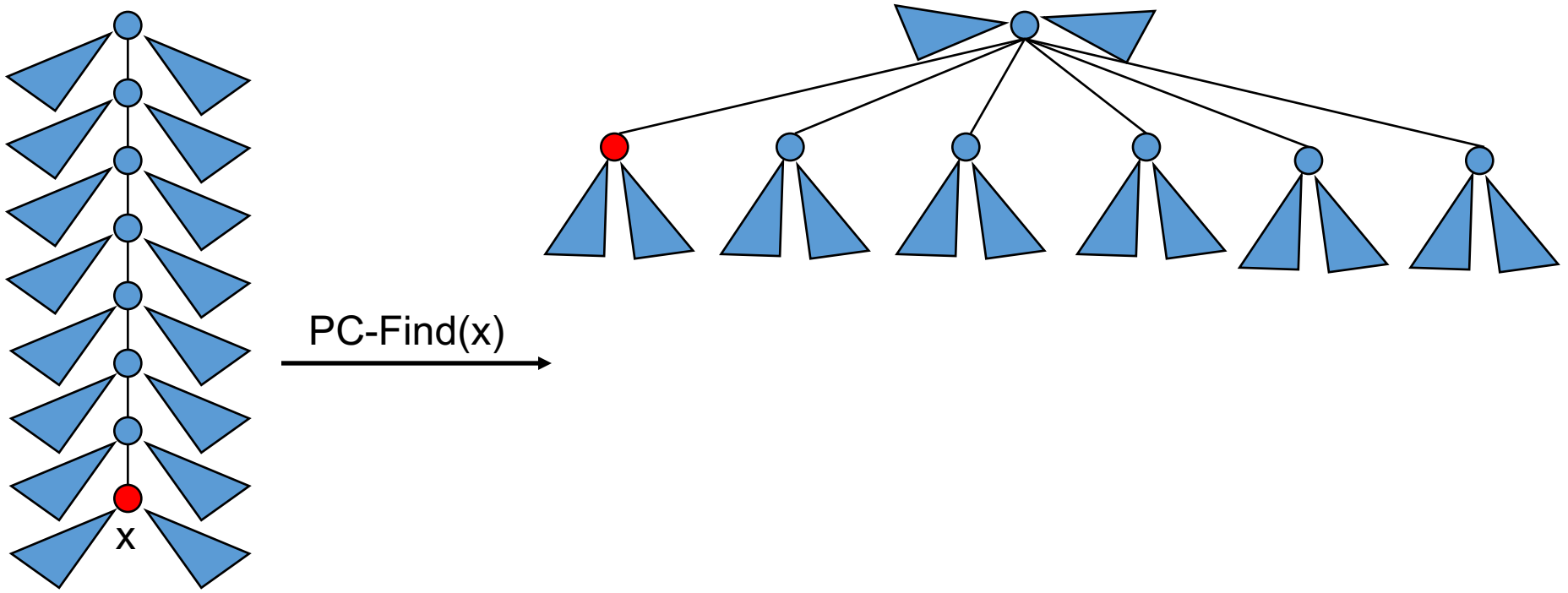
- Total running time: O (m log n + n + m log n + m) = O(m log n)
  - note: log n and log m are within a constant factor of one another

Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



Self-Adjustment Works



Disjoint Union / Find with Union By Rank and Path Compression

- Worst case time complexity
  - Union (using Union by Rank): $\Theta(1)$
  - Find (using Path Compression): $\Theta(\log n)$.

- Time complexity for $m \geq n$ operations on n elements is $\Theta(m \log^* n)$
  - log * is the number of times you need to apply the log function before you get to a number <= 1
  - log * n < 5 for all reasonable n. Essentially constant time per operation!

n	$\lg^* n$
1	0
2	1
$(3, 4]$	2
$[5, 16]$	3
$[17, 65536]$	4
$[65537, 2^{65536}]$	5

Amortized Complexity

- For disjoint union / find with union by rank and path compression
  - average time per operation is essentially a constant
  - worst case time for a Find is $\Theta(\log n)$

- An individual operation can be costly, but over time the average cost per operation is not

- This means the bottleneck of Kruskal's actually becomes the sorting of the edges

Outline

- Dijkstra's algorithm
- Huffman's algorithm

Shortest paths – Dijkstra's algorithm

Single Source Shortest Paths Problem: Given a weighted connected graph G, find shortest paths from source vertex *s* to each of the other vertices

Dijkstra's algorithm was published in 1959.

Implementation of Dijkstra's algorithm:

- $O(m + n^2)$ Dijkstra'59

Edsger Wybe Dijkstra (1930 – 2002), PhD, Professor Emeritus and Chair of Computer Sciences at University of Texas at Austin, Turing Award recipient (1972)

http://acm.math.spbu.ru/~sk1/download/papers/Thorup-1.pdf
https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Shortest paths – Dijkstra's algorithm

Single Source Shortest Paths Problem: Given a weighted connected graph G, find shortest paths from source vertex *s* to each of the other vertices

Dijkstra's algorithm was published in 1959.

Implementation of Dijkstra's algorithm:

- $O(m + n^2)$ Dijkstra'59
- O(m log n) William'64
- O(m + n log n) Fredman and Tarjan'87
- ……
- linear time and linear space algorithm (undirected graph), Mikkel Thorup'99

Edsger Wybe Dijkstra (1930 – 2002), PhD, Professor Emeritus and Chair of Computer Sciences at University of Texas at Austin, Turing Award recipient (1972)

http://acm.math.spbu.ru/~sk1/download/papers/Thorup-1.pdf
https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Shortest paths – Dijkstra's algorithm

Dijkstra's algorithm: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex u with the smallest sum

$$d_v + w(v,u)$$

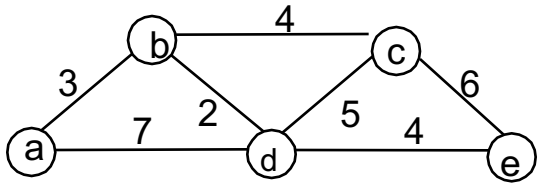
where

v is a vertex for which shortest path has already been found on preceding iterations (such vertices form a tree)

d_v is the length of the shortest path from source to v

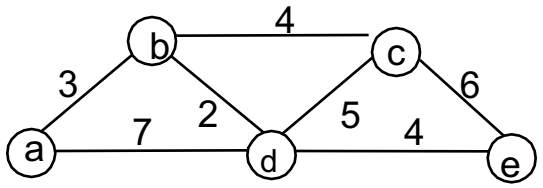
$w(v,u)$ is the length (weight) of edge from v to u

Example



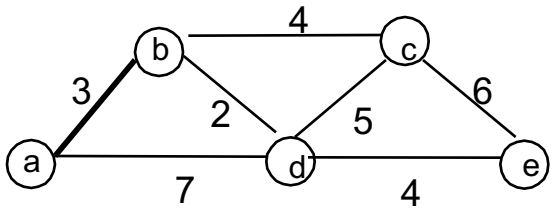
Find shortest path from *s* to nearest *v*, then to 2nd nearest, etc.

Example

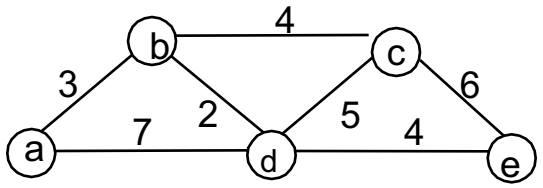


Find shortest path from *s* to nearest *v*, then to 2nd nearest, etc.

Tree vertices	Remaining vertices
a(-,0)	**b(a,3)** c(-,∞) d(a,7) e(-,∞)

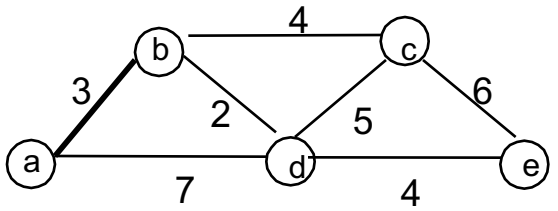


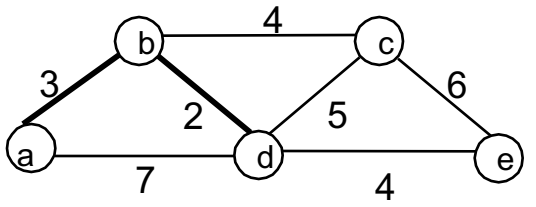
Example



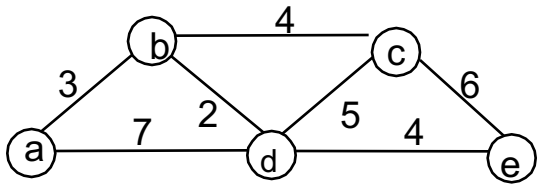
Find shortest path from *s* to nearest *v*, then to 2nd nearest, etc.

Tree vertices	Remaining vertices
a(-,0)	**b(a,3)** c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) **d(b,3+2)** e(-,∞)



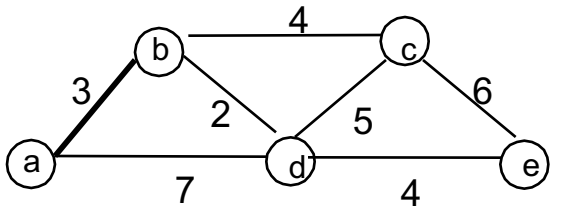


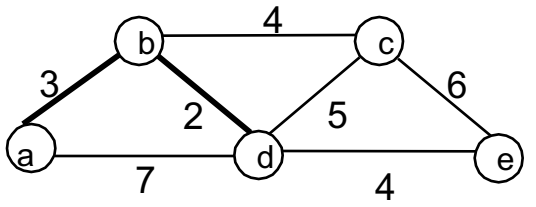
Example

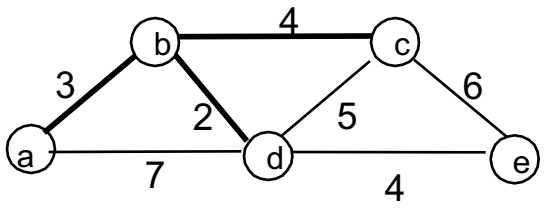


Find shortest path from *s* to nearest *v*, then to 2nd nearest, etc.

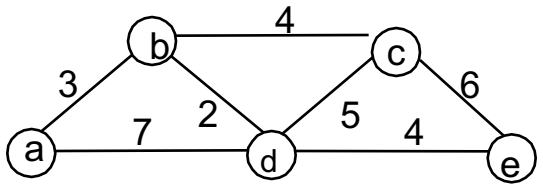
Tree vertices	Remaining vertices
a(-,0)	**b(a,3)** c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) **d(b,3+2)** e(-,∞)
d(b,5)	**c(b,7)** e(d,5+4)





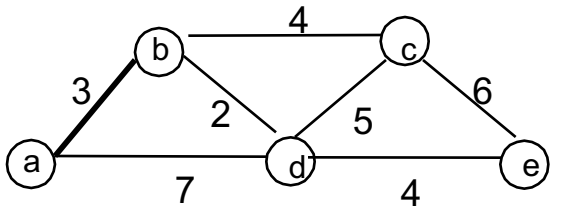


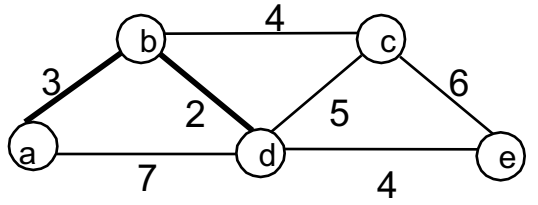
Example

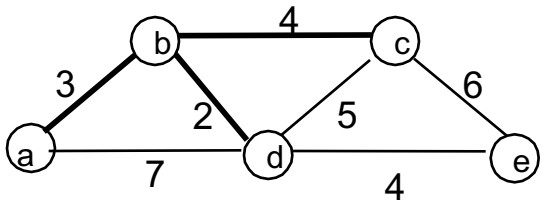


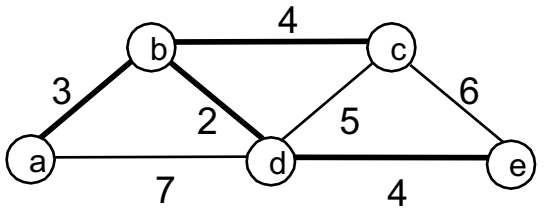
Find shortest path from *s* to nearest *v*, then to 2nd nearest, etc.

Tree vertices	Remaining vertices
a(-,0)	**b(a,3)** c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) **d(b,3+2)** e(-,∞)
d(b,5)	**c(b,7)** e(d,5+4)
c(b,7)	**e(d,9)**

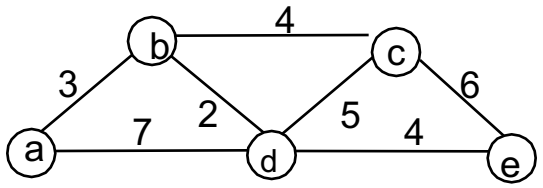






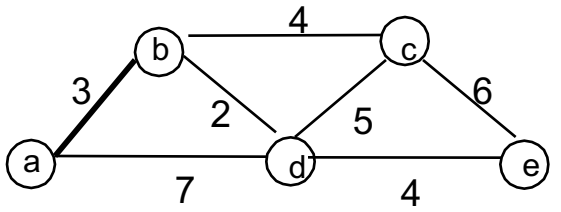


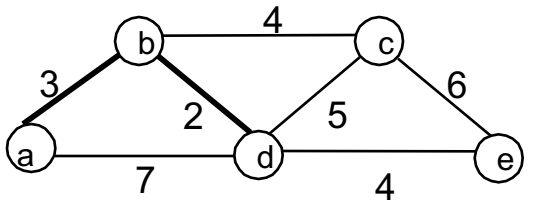
Example

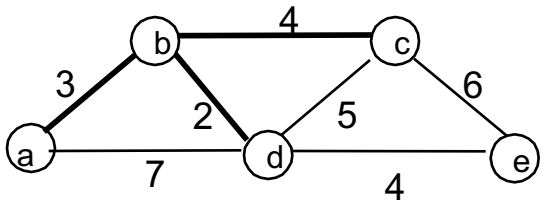


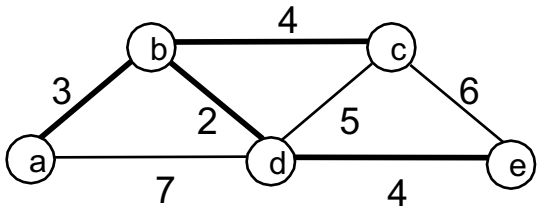
Find shortest path from *s* to nearest *v*, then to 2nd nearest, etc.

Tree vertices	Remaining vertices
a(-,0)	**b(a,3)** c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) **d(b,3+2)** e(-,∞)
d(b,5)	**c(b,7)** e(d,5+4)
c(b,7)	**e(d,9)**
e(d,9)	









ALGORITHM *Dijkstra*(G, s)

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//       and its vertex s
//Output: The length d_v of a shortest path from s to v
//        and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize vertex priority queue to empty
for every vertex v in V do
    d_v ← ∞;  p_v ← null
    Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)  //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);  p_u ← u*
            Decrease(Q, u, d_u)
```

Notes on Dijkstra's algorithm

- Doesn't work for graphs with negative weights
- Applicable to both undirected and directed graphs
- Efficiency
  - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(|E|\log|V|)$ for graphs represented by adj. lists and min-heap implementation of priority queue
- Don't mix up Dijkstra's algorithm with Prim's algorithm!

Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$ (here we let $S=V_T$), d(u) is the length of the shortest s-u path.

Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$ (here we let $S=V_T$), d(u) is the length of the shortest s-u path.

Proof by induction on |S|
Base case: |S| = 1 is trivial.

Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$ (here we let $S=V_T$), d(u) is the length of the shortest s-u path.

Proof by induction on |S|
Base case: |S| = 1 is trivial.
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

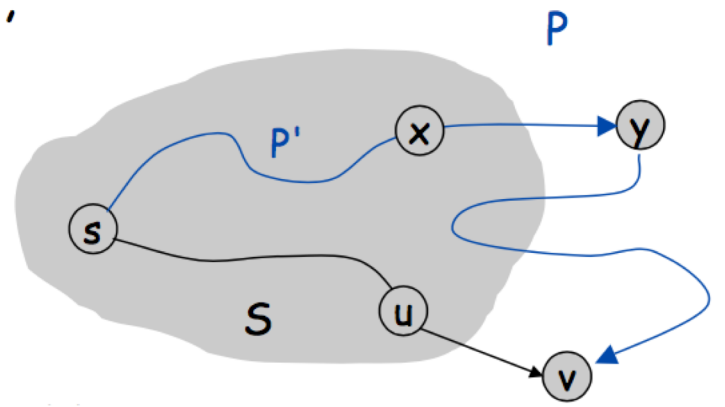
- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length d(v).

Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$ (here we let $S=V_T$), d(u) is the length of the shortest s-u path.

Proof by induction on |S|
Base case: |S| = 1 is trivial.
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length d(v).
- Consider any s-v path P. We'll see that it's no shorter than d(v).
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$d(P) \geq d(P') + w(x,y) \geq d(u) + w(u,y) \geq d(v)$$

Coding Problem

Coding: assignment of bit strings to alphabet characters

Codewords: bit strings assigned for characters of alphabet

Two types of codes:

- *fixed-length encoding* (e.g., ASCII)
- *variable-length encoding* (e,g., Morse code)

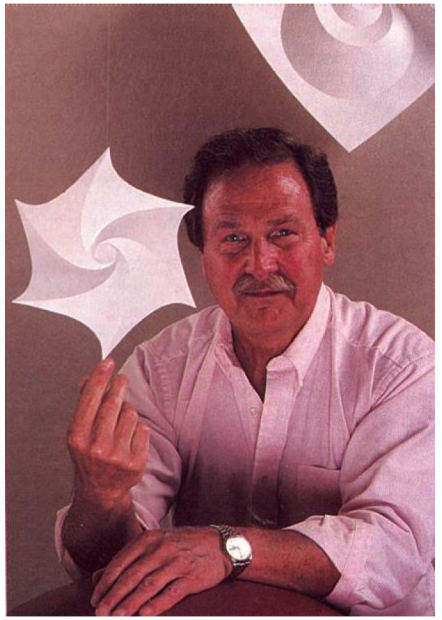
Prefix-free codes: no codeword is a prefix of another codeword

Problem: If frequencies of the character occurrences are known, what is the best binary prefix-free code?

Huffman Codes

- Proposed by Dr. David A. Huffman in 1952.

- Used in nearly every application involving compression and transmission of digital data, e.g., fax machines, modems, computer networks, and high-definition television.



David A. Huffman, PhD
(1925 - 1999)

Video Compression and Huffman Codes

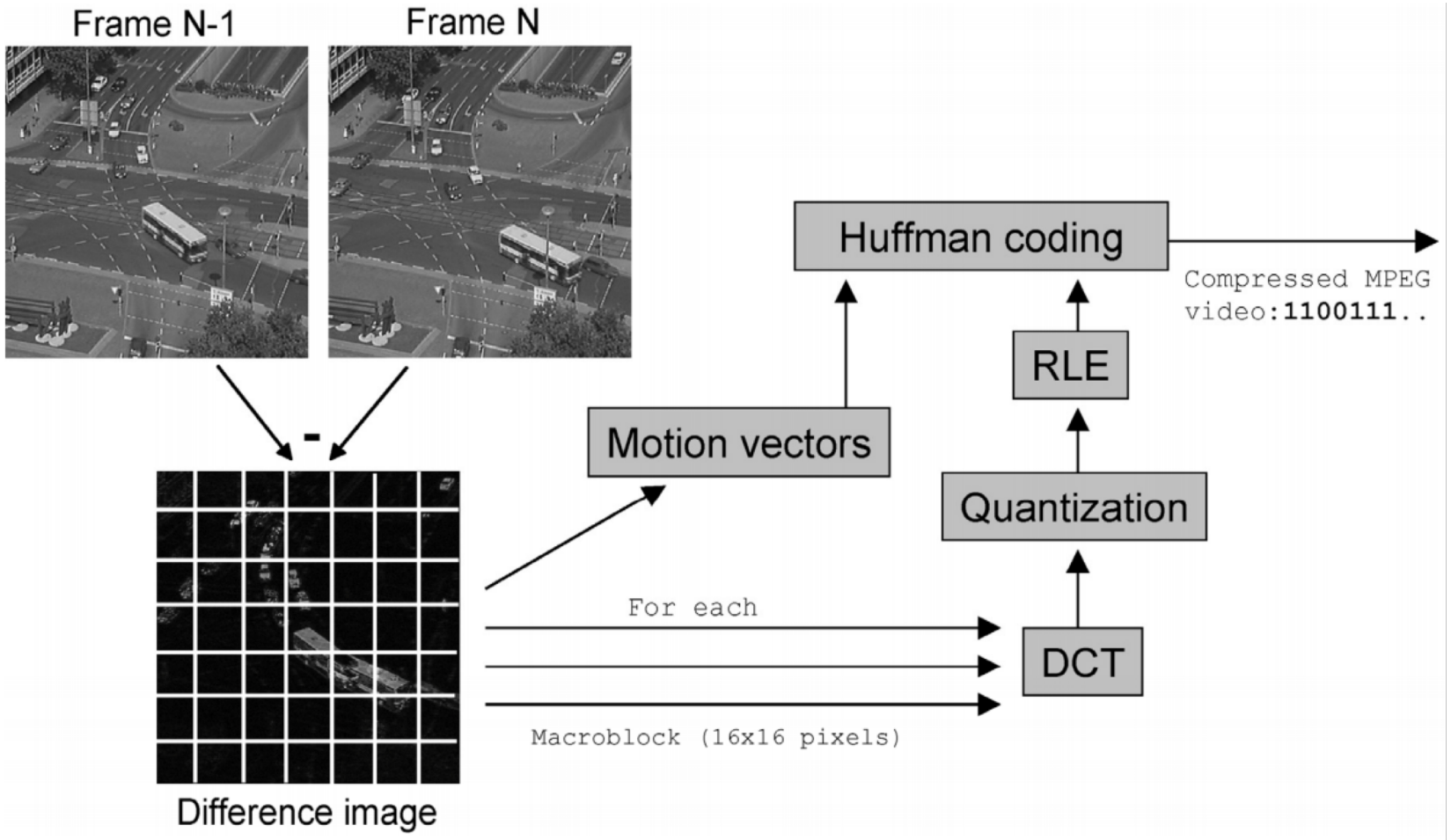


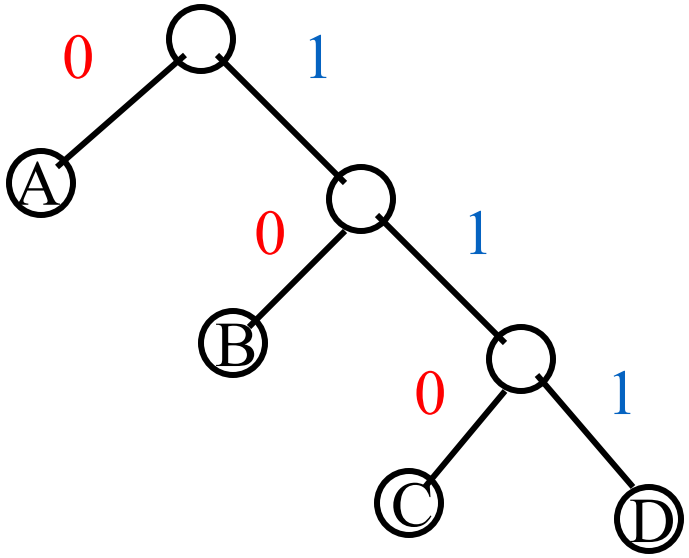
Figure 9: Illustration of the discussed 5 steps for a standard MPEG encoding.

Idea

Consider a binary tree, with:

0 meaning a left turn

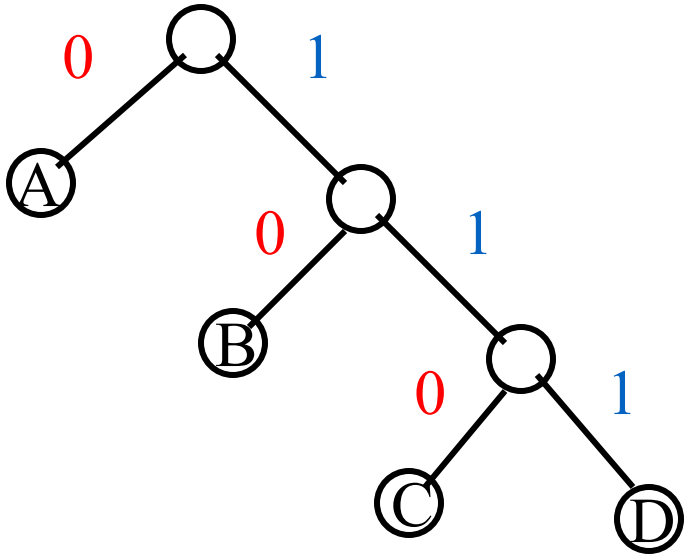
1 meaning a right turn.



Idea (cont.)

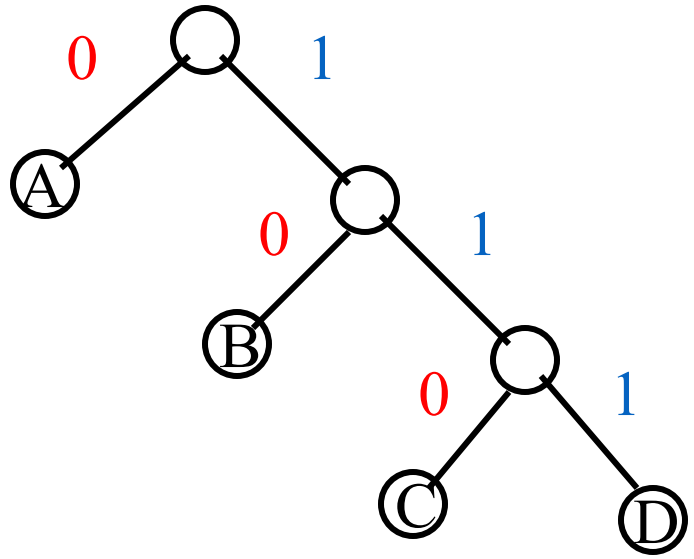
Consider the paths from the root to each of the leaves A, B, C, D:

A : 0
B : 10
C : 110
D : 111



Observe:

1. This is a prefix code, since each of the leaves has a path ending in it, without continuation.
2. How to make sure that the more frequent symbols are closer to the root so that they have a smaller code?



Greedy Algorithm:

1. Consider all pairs: <frequency, symbol>.

2. Choose the two lowest frequencies, and make them brothers, with the root having the combined frequency.

3. Iterate.

Huffman algorithm

Initialize trees of a single node each.

Keep the roots of all subtrees in a priority queue.

Iterate until only one tree left:
Merge the two smallest frequency subtrees into a single subtree with two children, and insert into priority queue.

Greedy Algorithm Example:

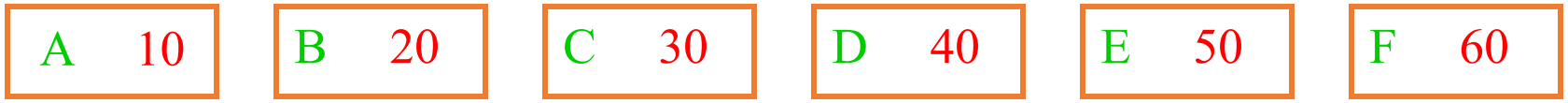
Alphabet: A, B, C, D, E, F

Frequency table:

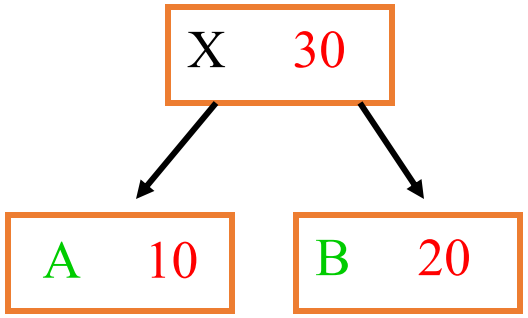
A	B	C	D	E	F
10	20	30	40	50	60

Total File Length: 210

Algorithm Run:

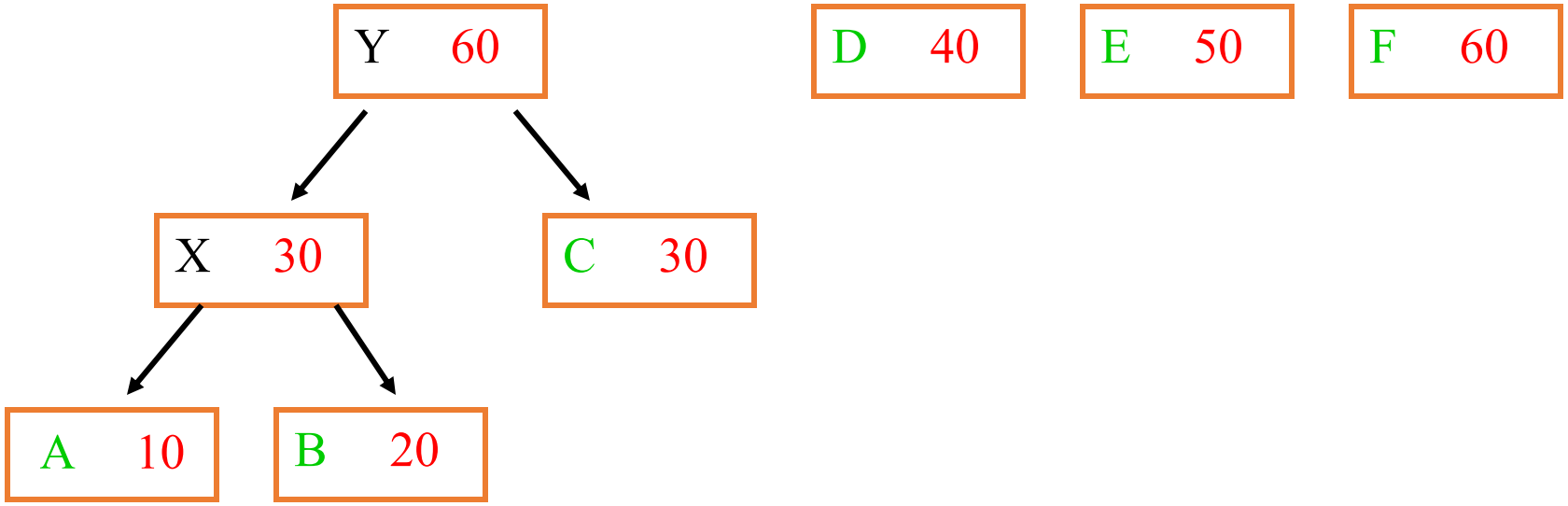


Algorithm Run:

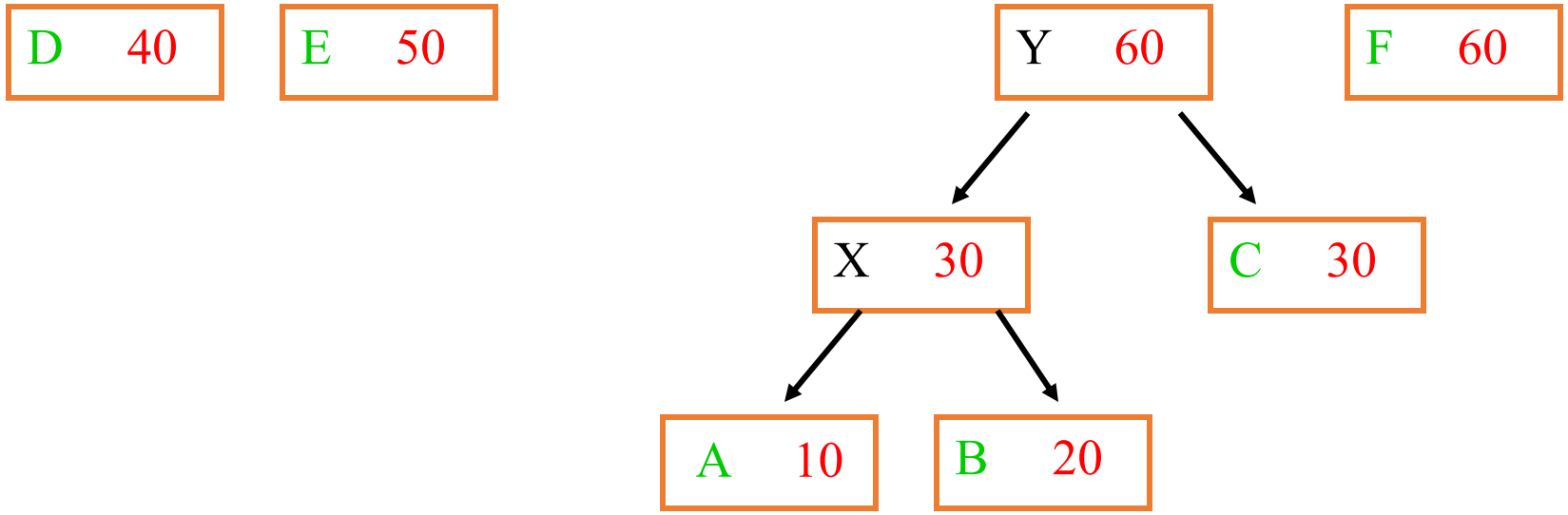


C 30 D 40 E 50 F 60

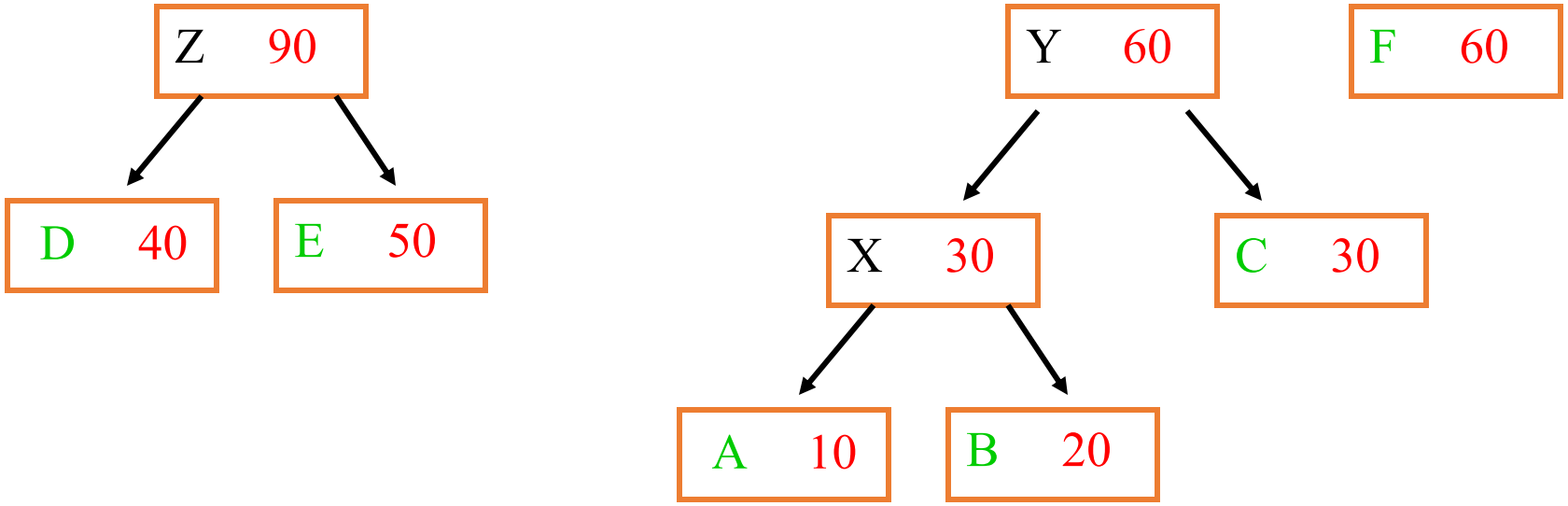
Algorithm Run:



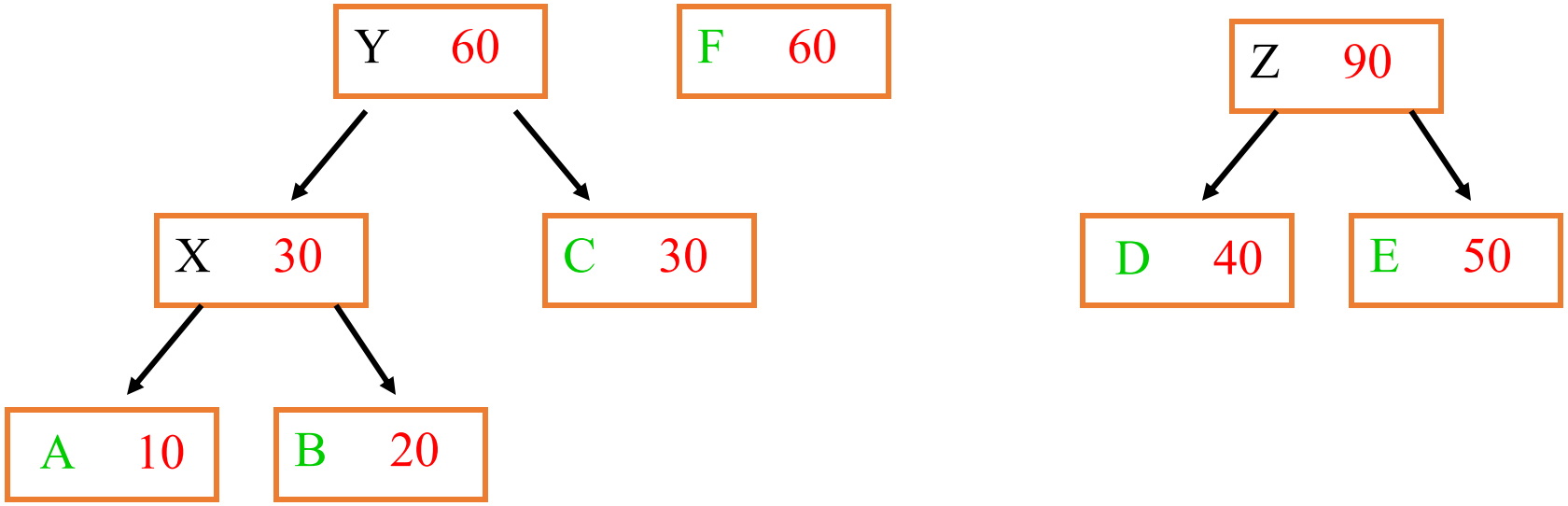
Algorithm Run:



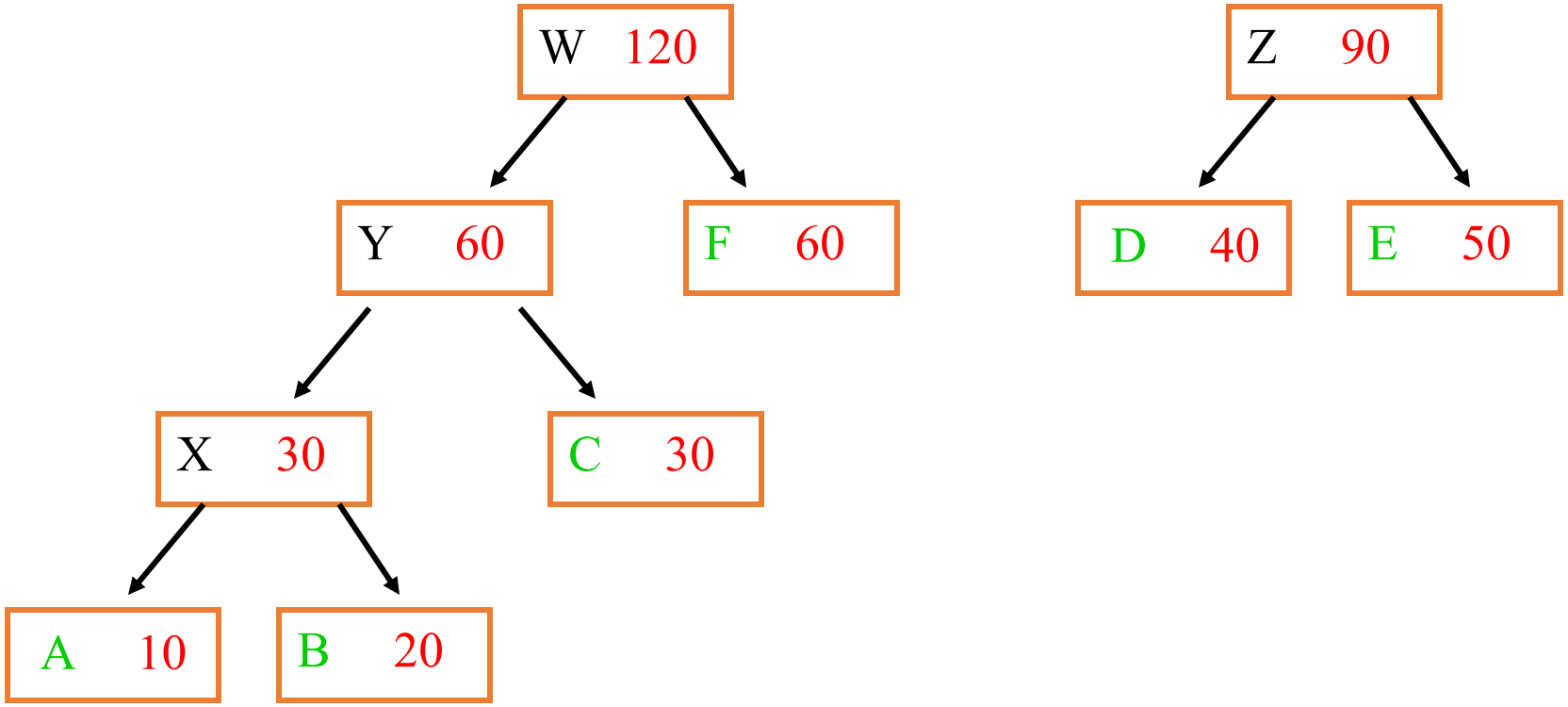
Algorithm Run:



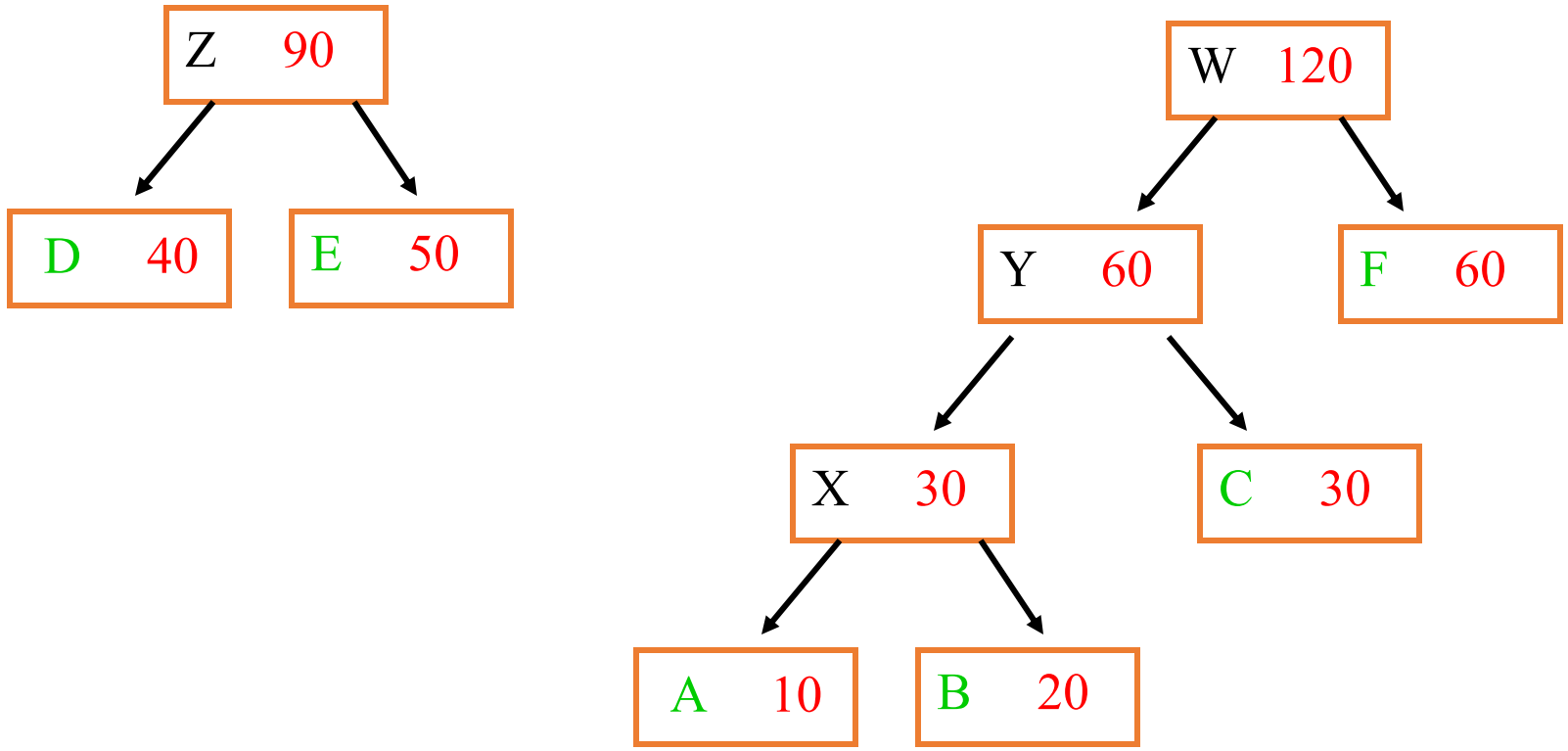
Algorithm Run:



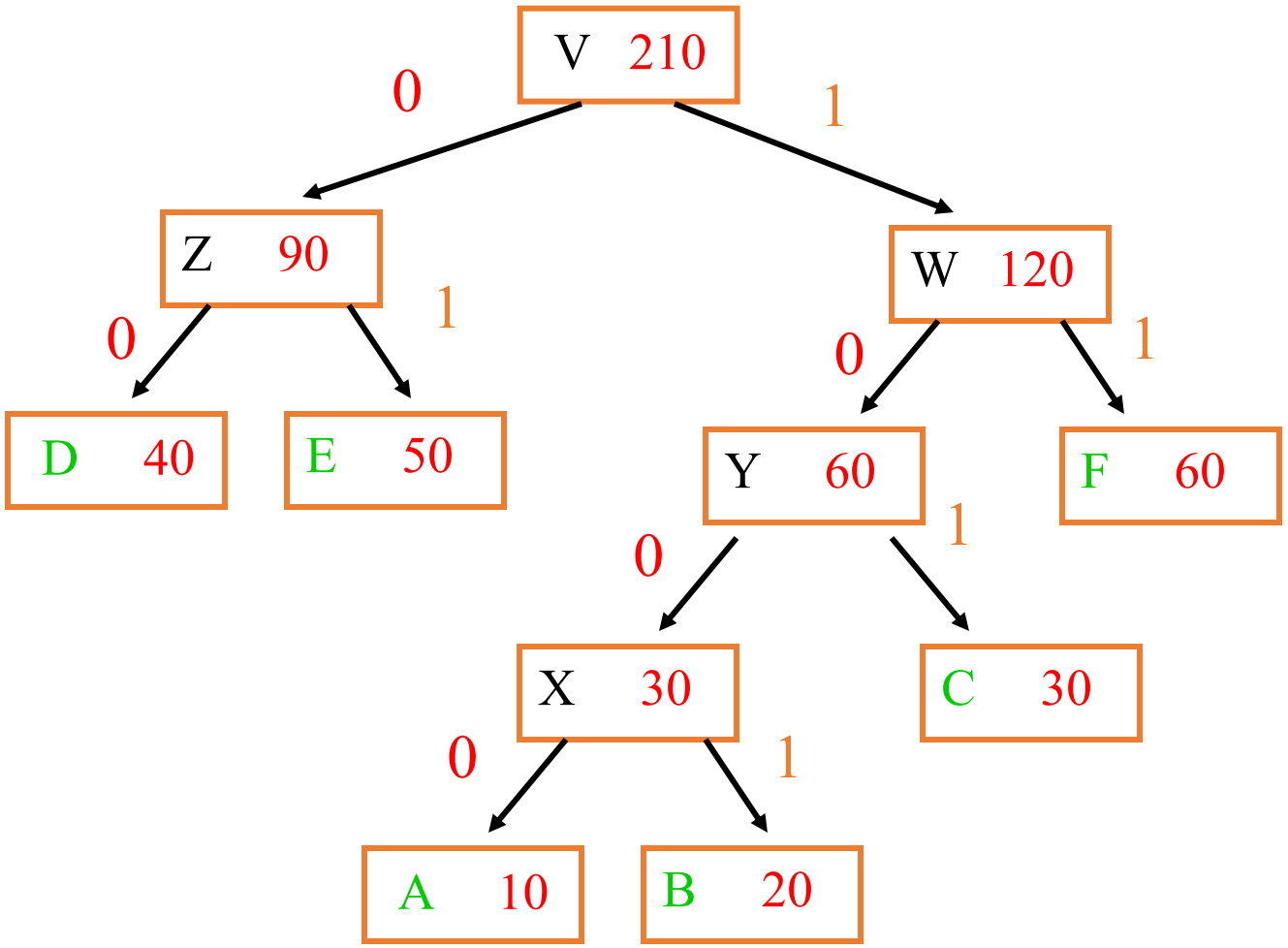
Algorithm Run:



Algorithm Run:

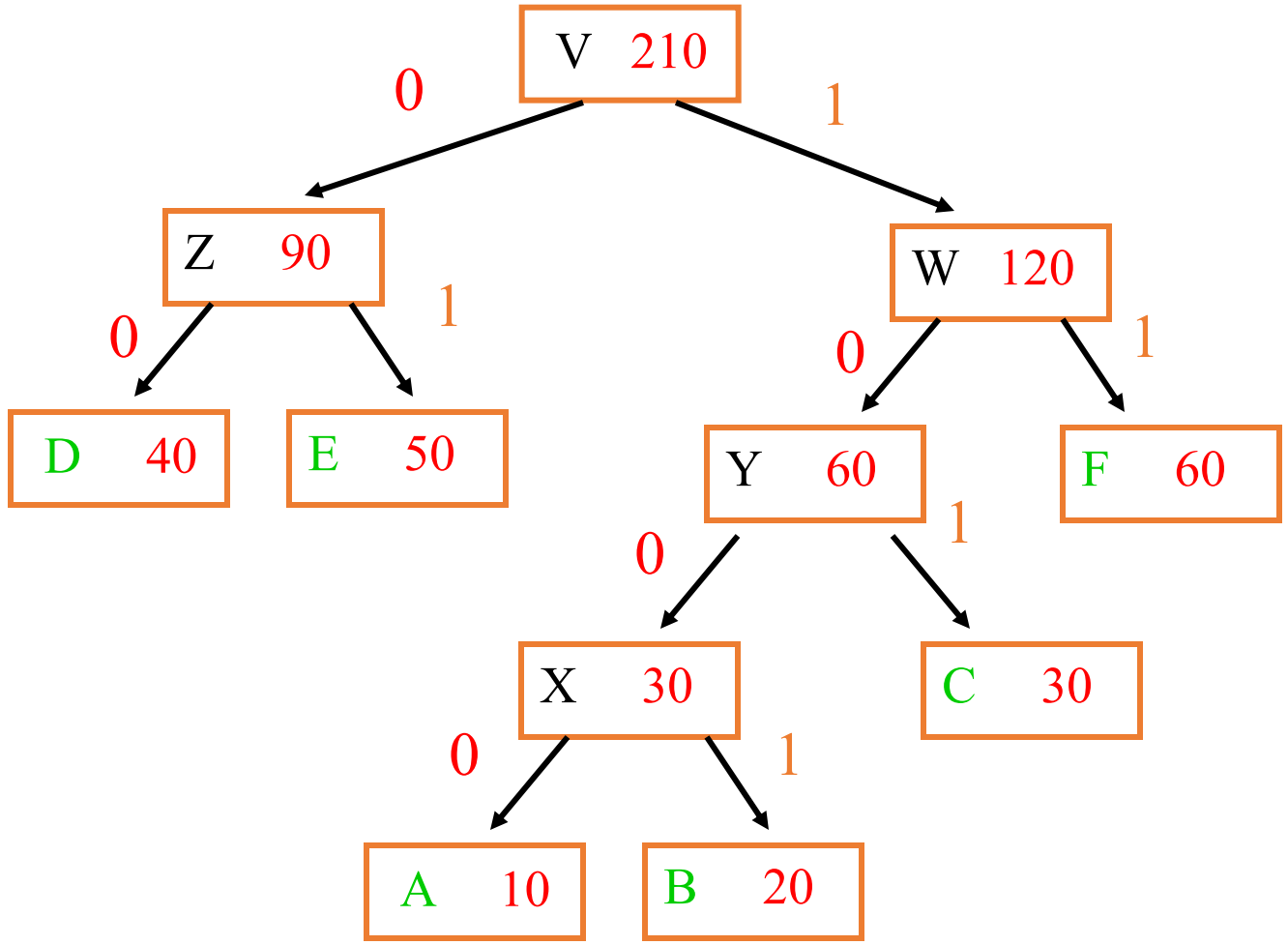


Algorithm Run:



The Huffman encoding:

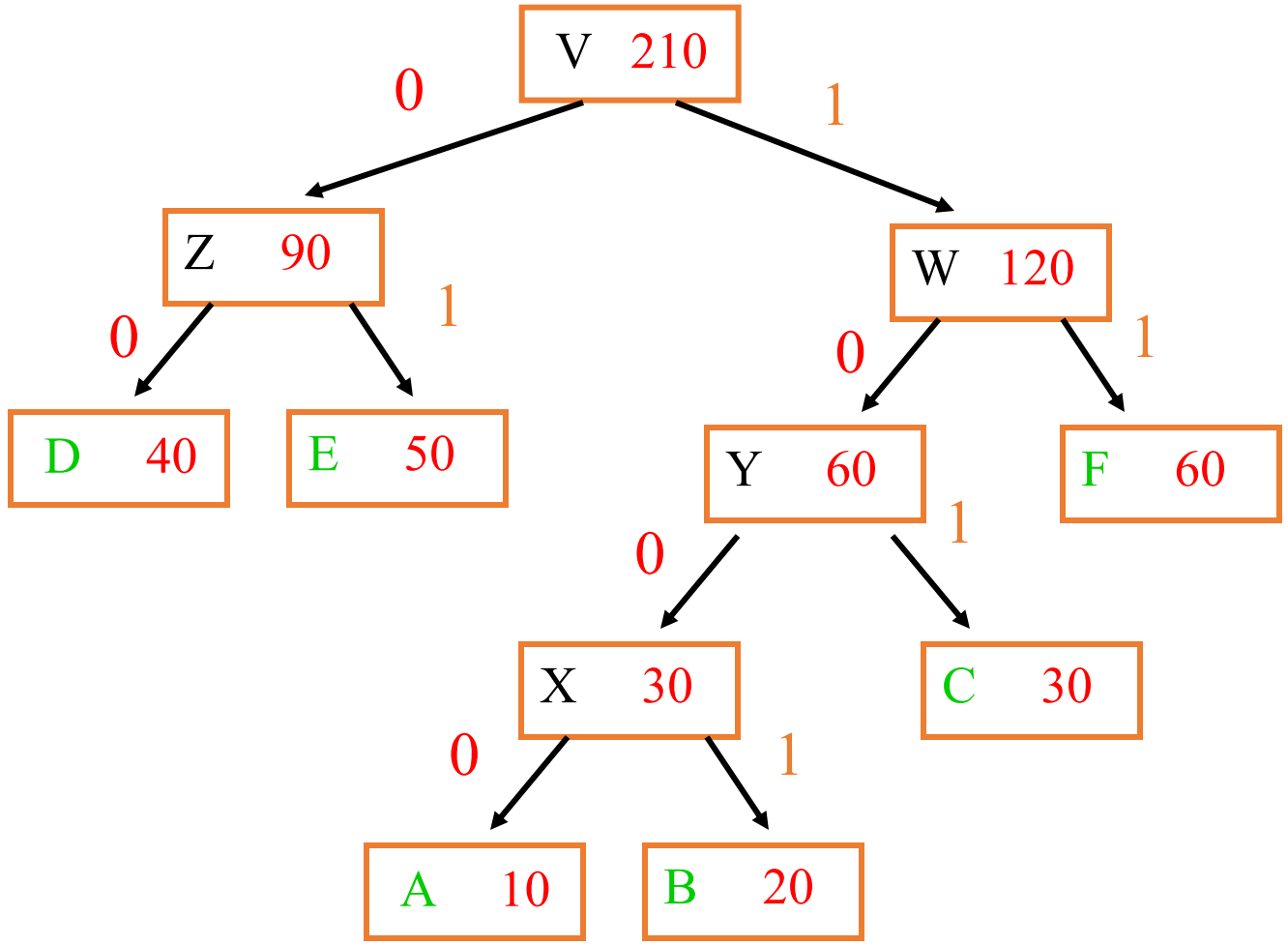
A: 1000
B: 1001
C: 101
D: 00
E: 01
F: 11



File Size ?

The Huffman encoding:

A: 1000
B: 1001
C: 101
D: 00
E: 01
F: 11



File Size:

10x4 + 20x4 + 30x3 + 40x2 + 50x2 + 60x2 =40 + 80 + 90 + 80 + 100 +120 = 510 bits

Note the savings:

The Huffman code:
Required 510 bits for the file.

Fixed length code:
Need 3 bits for 6 characters.
File has 210 characters.
Total: 630 bits for the file.

Compression ratio = (630-510)/630 = 20%

Algorithm time:

Each priority queue operation (e.g. heap):
$O(\log n)$

In each iteration: one less subtree.

Initially: n subtrees.

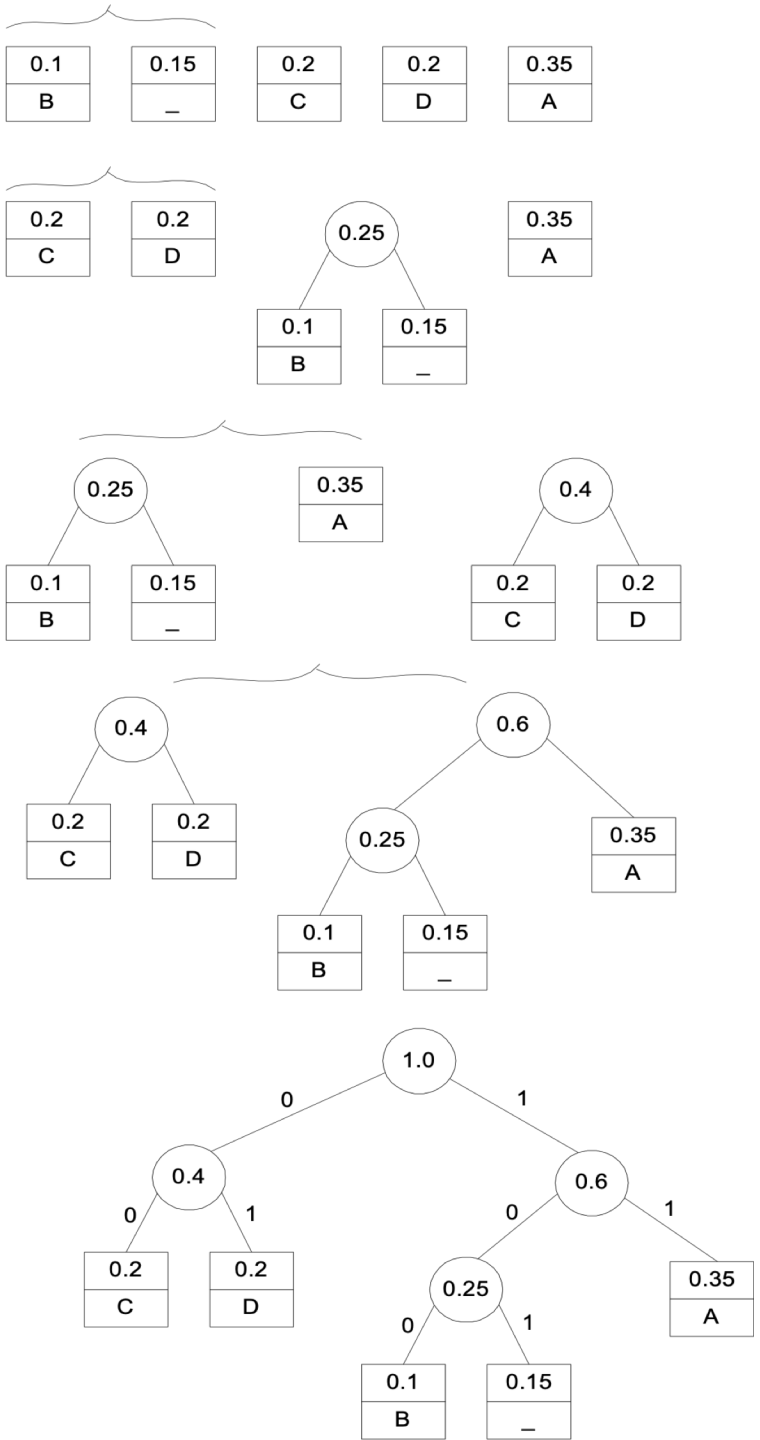
Total: $O(n \log n)$ time.

Example

character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

Example

character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101



Summary

- Huffman tree is **optimal**

- Drawback: It requires the coding table into the encoded text to make its decoding possible

- The drawback can be overcome by ***dynamic Huffman encoding,*** which updates coding tree each time a new symbol is read from the source text