Unit 3

CI43/CY43
Design and Analysis of Algorithms

By

Dr. Sini Anna Alex

Greedy Technique

Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:

- *feasible*
- *locally optimal*
- *irrevocable*

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: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$
Greedy solution:

Greedy solution:

- optimal for any amount and “typical’’ set of denominations
- not optimal for all coin denominations …

JOB SEQUENCING WITH DEADLINES

The problem is stated as below.

- There are n jobs to be processed on a machine.
- Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- Pi is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.

JOB SEQUENCING WITH DEADLINES (Contd..)

- A feasible solution is a subset of jobs J such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.

  Example : Let n = 4, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

JOB SEQUENCING WITH DEADLINES (Contd..)

Sr.No.	Feasible Solution	Processing Sequence	Profit value
(i)	(1,2)	(2,1)	110
(ii)	(1,3)	(1,3) or (3,1)	115
(iii)	(1,4)	(4,1)	127 ↑ is the optimal one
(iv)	(2,3)	(2,3)	25
(v)	(3,4)	(4,3)	42
(vi)	(1)	(1)	100
(vii)	(2)	(2)	10
(viii)	(3)	(3)	15
(ix)	(4)	(4)	27

GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION

- Consider the jobs in the non increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.
- In the example considered before, the non-increasing profit vector is

(100 27 15 10)	(2 1 2 1)
p_1 p_4 p_3 p_2	d_1 d_4 d_3 d_2

GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION (Contd..)

$J = \{ 1\}$ is a feasible one

$J = \{ 1, 4\}$ is a feasible one with processing sequence (4,1)

$J = \{ 1, 3, 4\}$ is not feasible

$J = \{ 1, 2, 4\}$ is not feasible

$J = \{ 1, 4\}$ is optimal

GREEDY ALGORITHM FOR JOB SEQUENSING WITH DEADLINE

```
Procedure greedy job (D, J, n)
// J is the set of n jobs to be completed//
// by their deadlines //
   J ←{1}
   for I ← 2 to n do
  If  all jobs in JU{i} can be completed
by their deadlines
  then J ← JU{I}
end if
   repeat
  end greedy-job
```

J may be represented by one dimensional array J (1: K)

The deadlines are

$D(J(1)) \leq D(J(2)) \leq .. \leq D(J(K))$

To test if JU {i} is feasible, we insert i into J and verify

$D(J®) \leq r \qquad 1 \leq r \leq k+1$

EXAMPLE: let n = 5, (p_1,--------p_5) = (20,15,10,5,1) and (d_1,--d_5) = (2,2,1,3,3). Using the above rule

J	assigned slot	jobs being considered	action or
Ø	none	1	assigned to [1, 2]
{1}	[1,2]	2	[0,1]
{1,2}	[0,1],[1,2]	3	cannot fit reject as [0,1] is not free
{1,2}	[0,1],[1,2]	4	assign to [2,3]
{1,2,4}	[0,1],[1,2],[2,3]	5	reject

The optimal solution is {1,2,4}

- As there are only n jobs and each job takes one unit of time, it is necessary to consider the time slots [i-1,i] $1 \leq i \leq b$ where $b = \min \{n, \max \{d_i\}\}$
- The time slots are partitioned into b sets .
- i represents the time slot [i-1,i]

  [0 1]is slot 1

  [1 2] is slot 2
- For any slot i, n_i represents the largest integers such that $n_i \leq i$ and slot n_i is free.

  If [1,2] is free

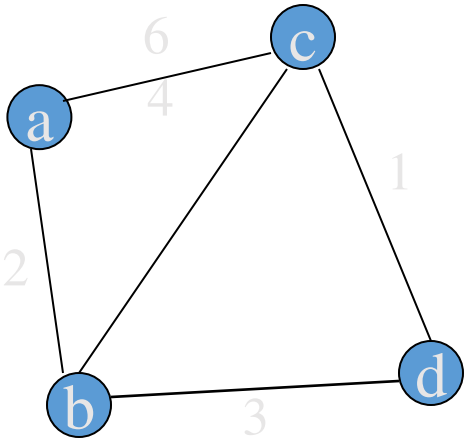
  n_2=2 otherwise

  n_2=1 if [0 1] is free

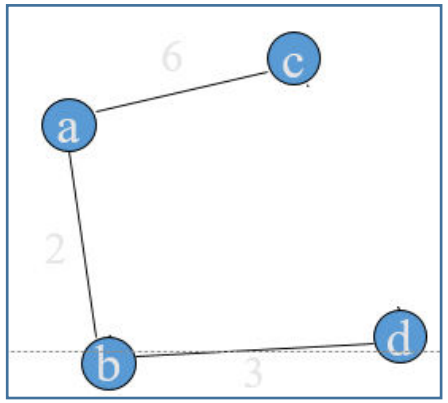
Minimum Spanning Tree (MST)

- *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

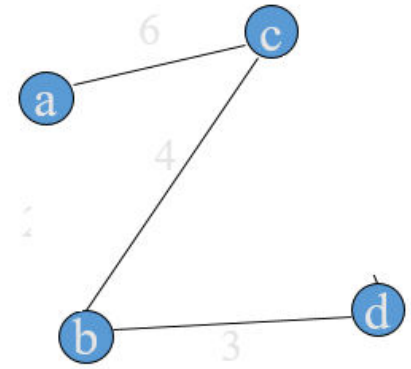
Example:



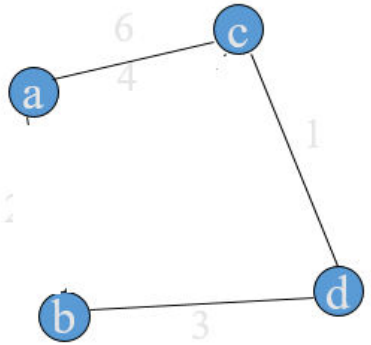
1



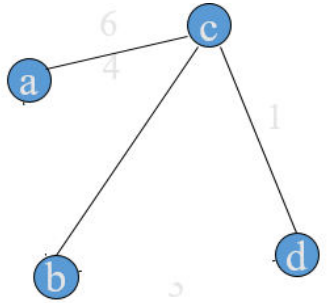
2



3



4



Prim's MST algorithm

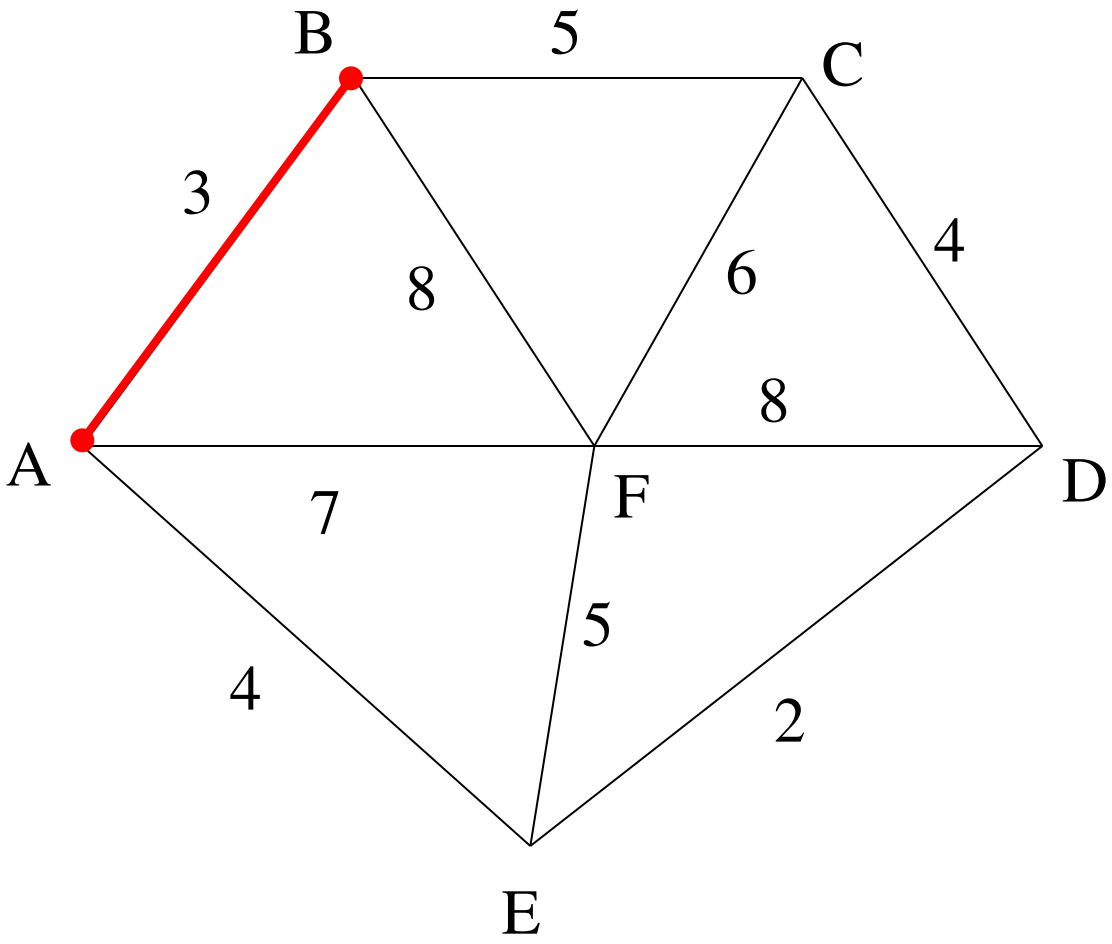
- Start with tree T_1 consisting of one (any) vertex and "grow" tree one vertex at a time to produce MST through a series of expanding subtrees T_1, T_2, …, T_n

- 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

Example

Prim's Algorithm



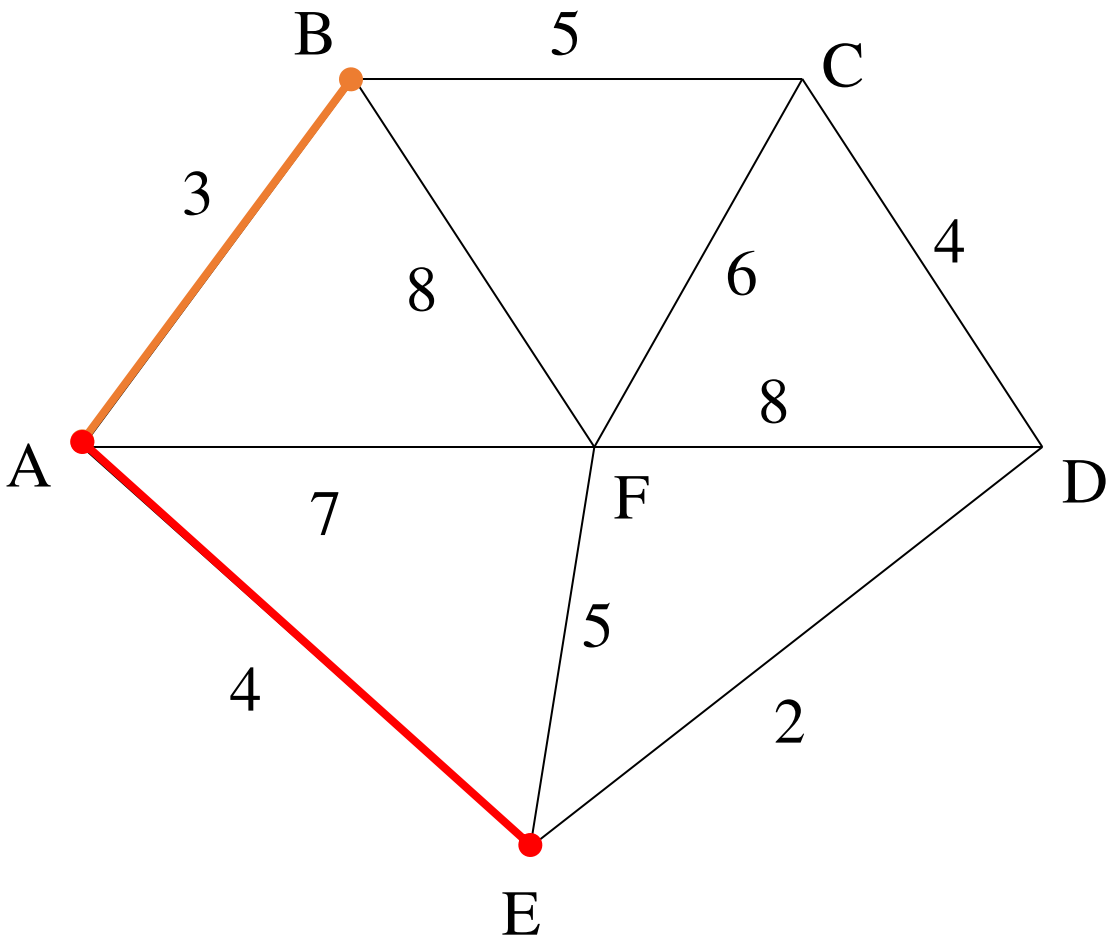
Select any vertex

A

Select the shortest edge connected to that vertex

AB 3

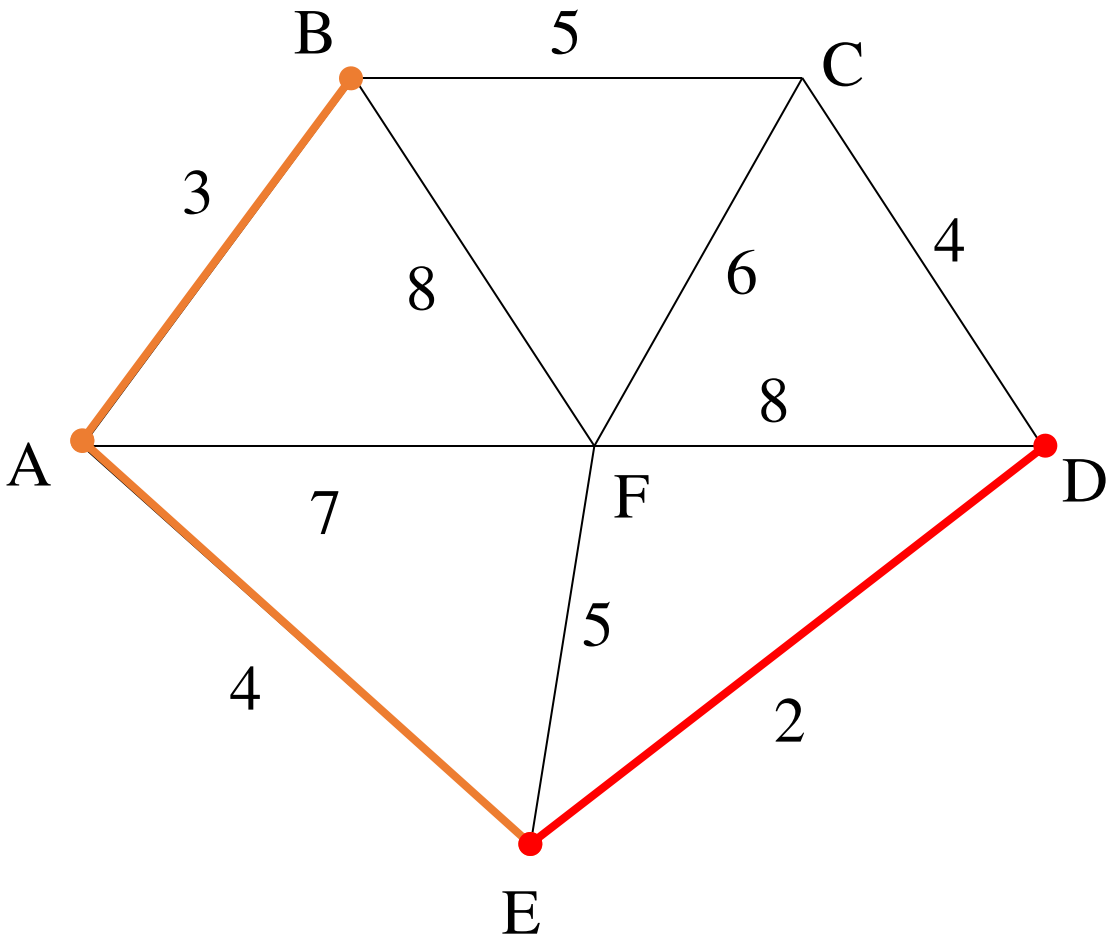
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

AE 4

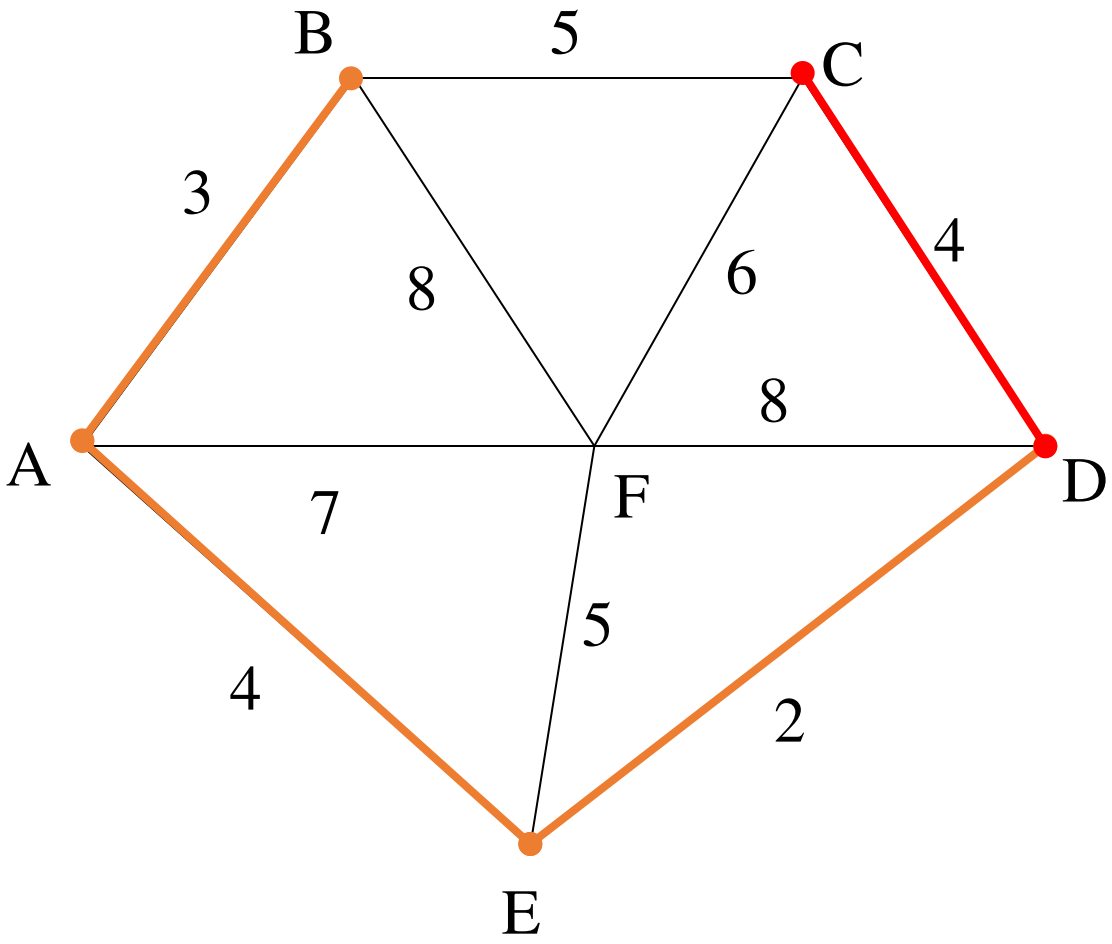
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

ED 2

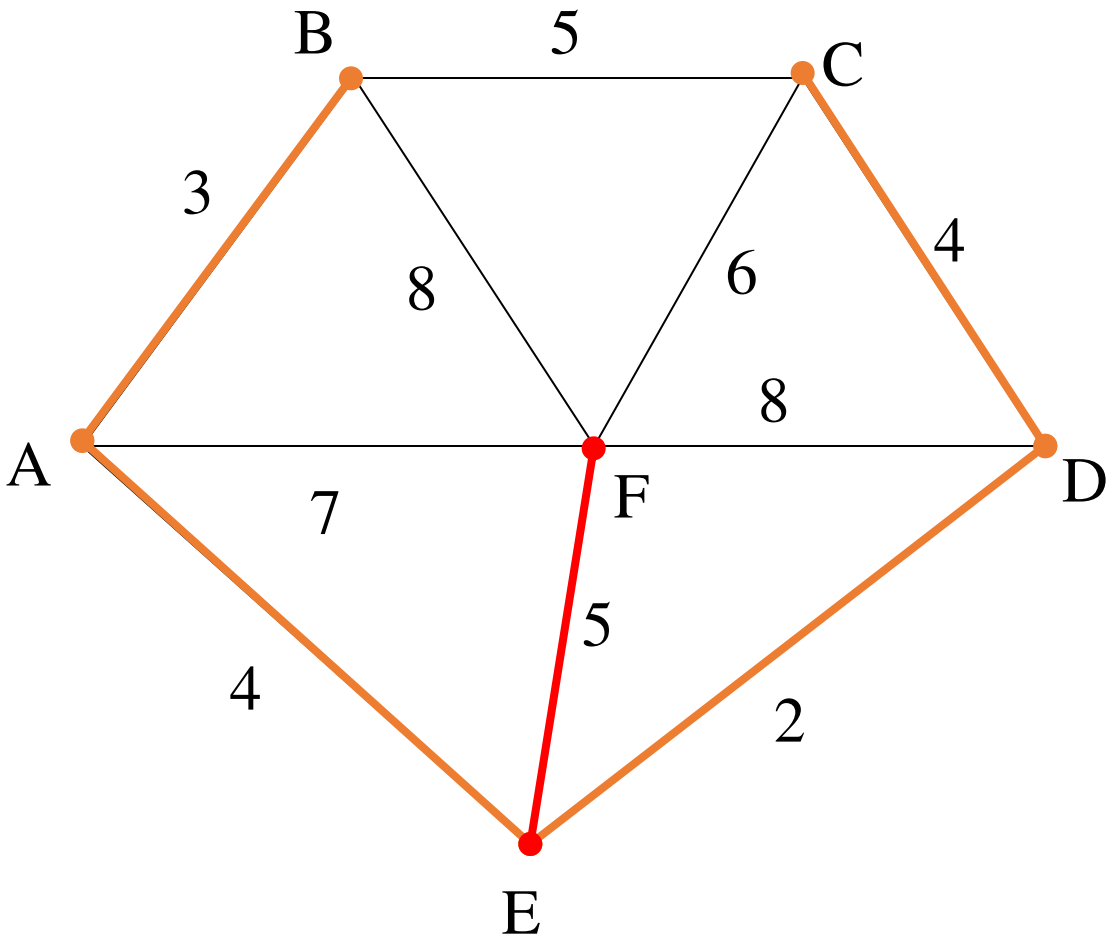
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

DC 4

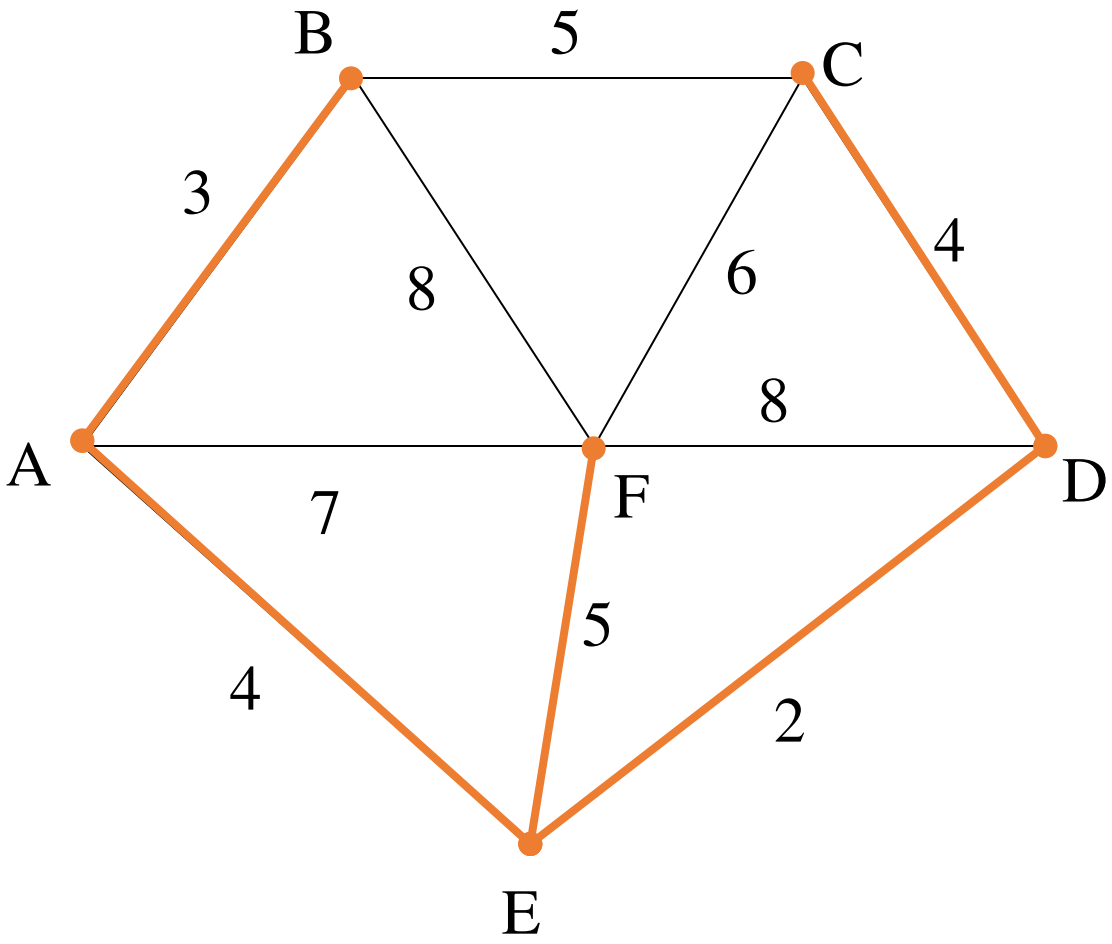
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

EF 5

Prim's Algorithm



All vertices have been connected.

The solution is

AB 3
AE 4
ED 2
DC 4
EF 5

Total weight of tree: 18

Notes about Prim's algorithm

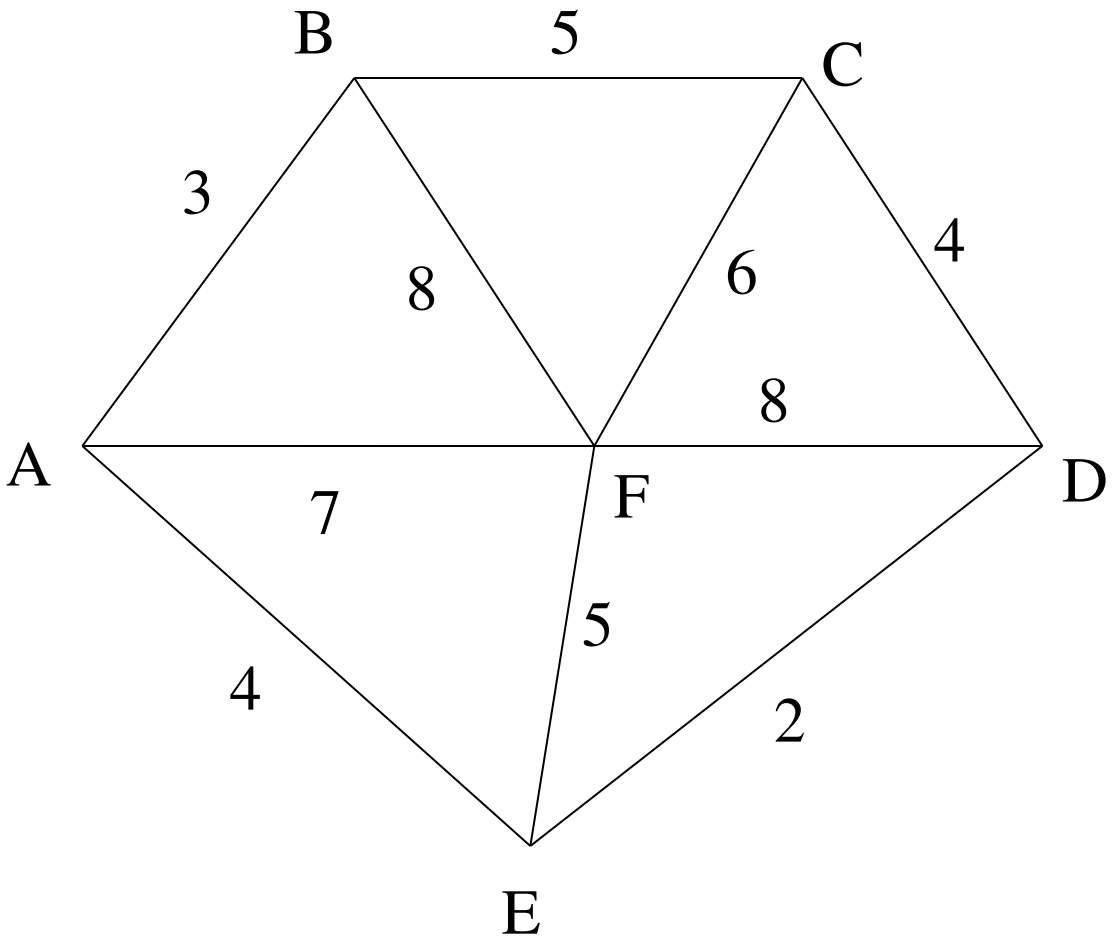
- Proof by induction that this construction actually yields MST
- Needs priority queue for locating closest fringe vertex
- Efficiency
  - $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue
  - $O(m \log n)$ for adjacency list representation of graph with n vertices and m edges and min-heap implementation of priority queue

Another greedy algorithm for MST: Kruskal's

- Sort the edges in nondecreasing 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.)

EXAMPLE

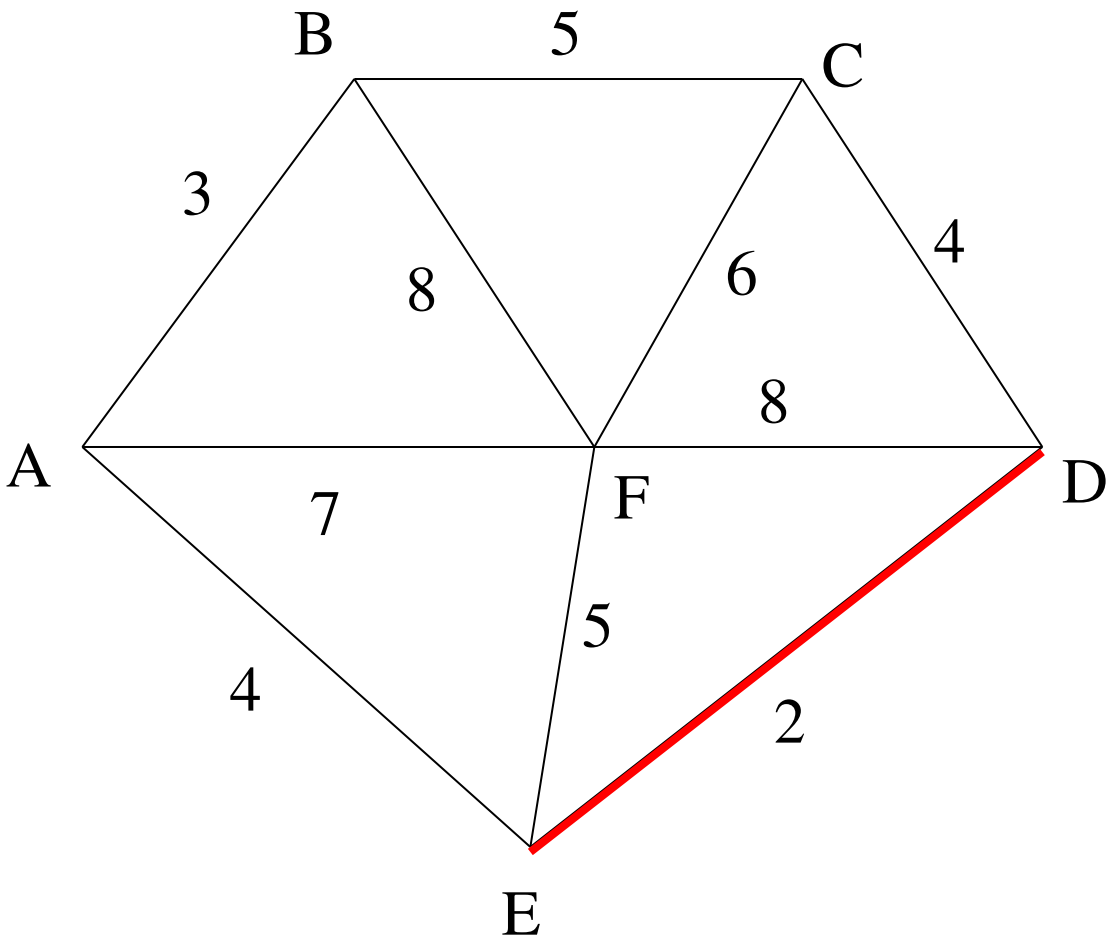
Kruskal's Algorithm



List the edges in order of size:

ED 2
AB 3
AE 4
CD 4
BC 5
EF 5
CF 6
AF 7
BF 8
CF 8

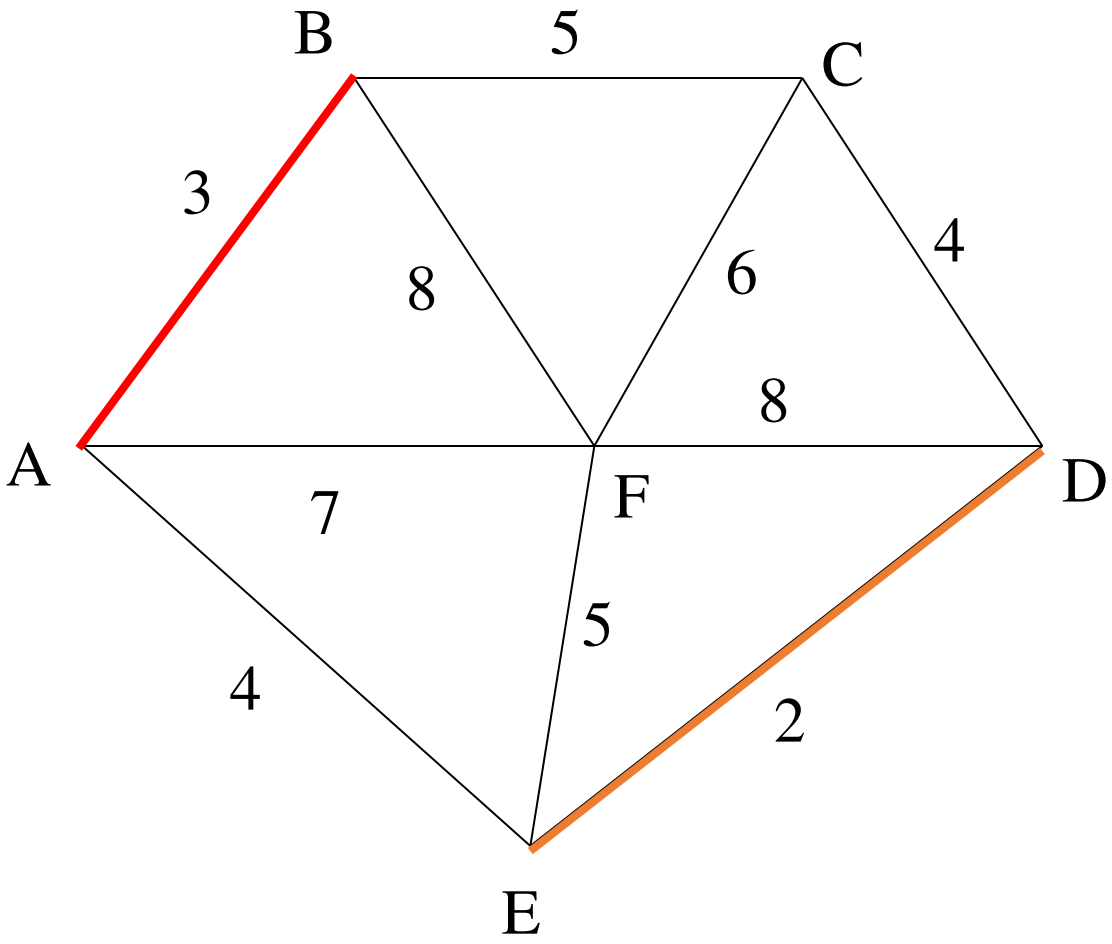
Kruskal's Algorithm



Select the shortest edge in the network

ED 2

Kruskal's Algorithm

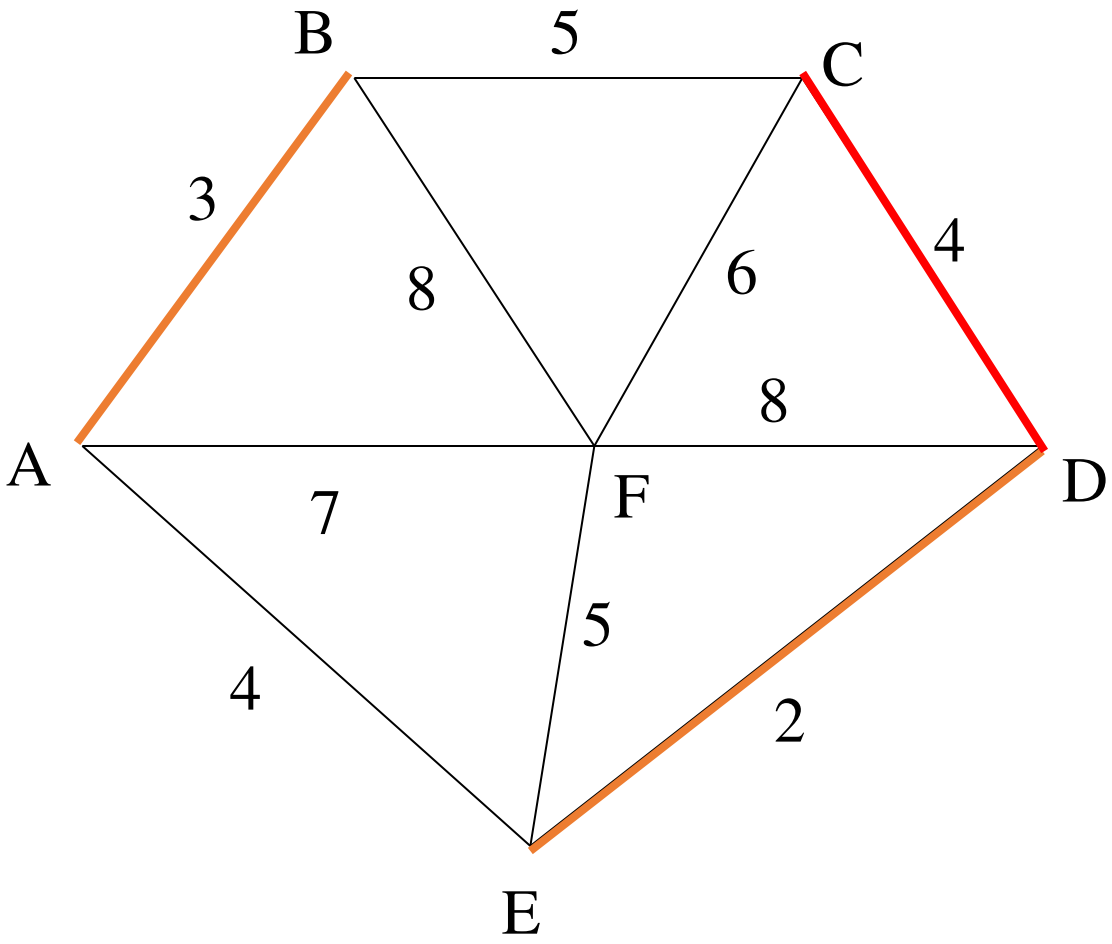


Select the next shortest edge which does not create a cycle

ED 2

AB 3

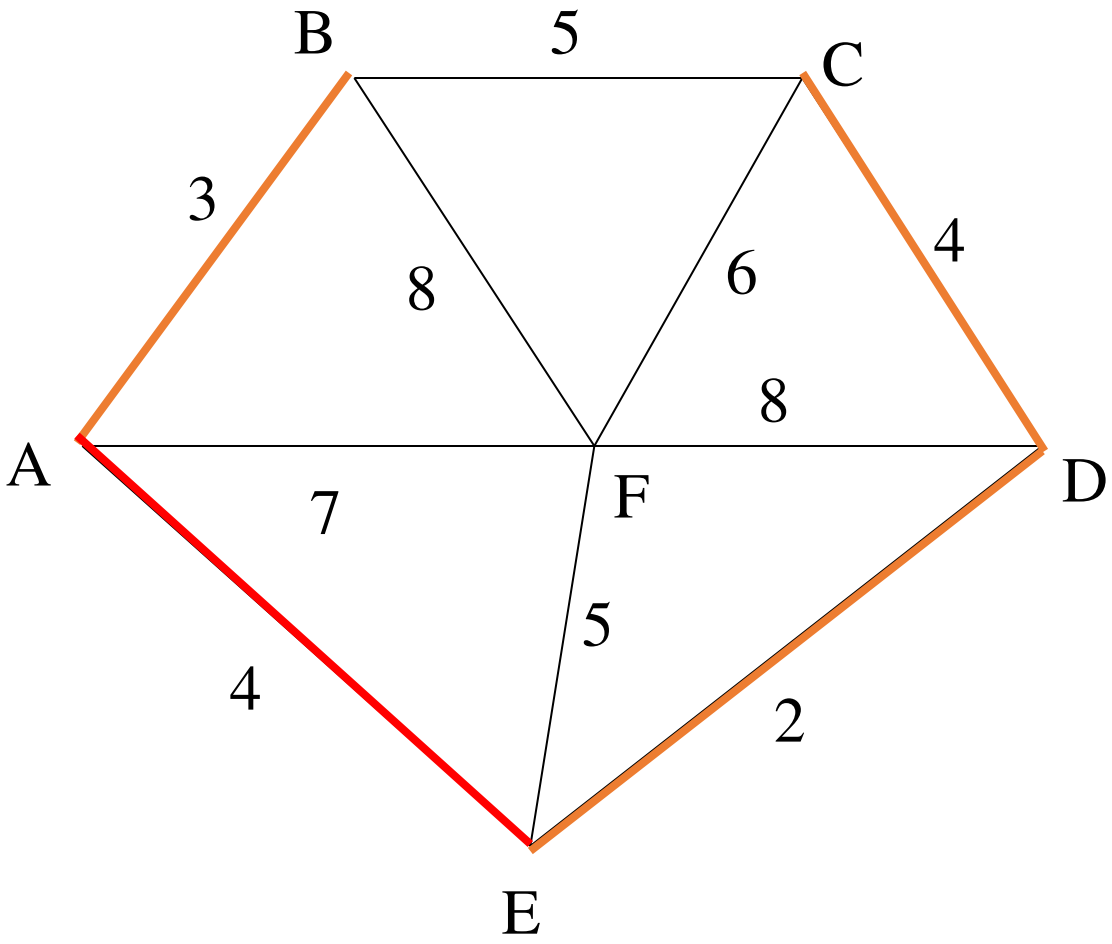
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4 (or AE 4)

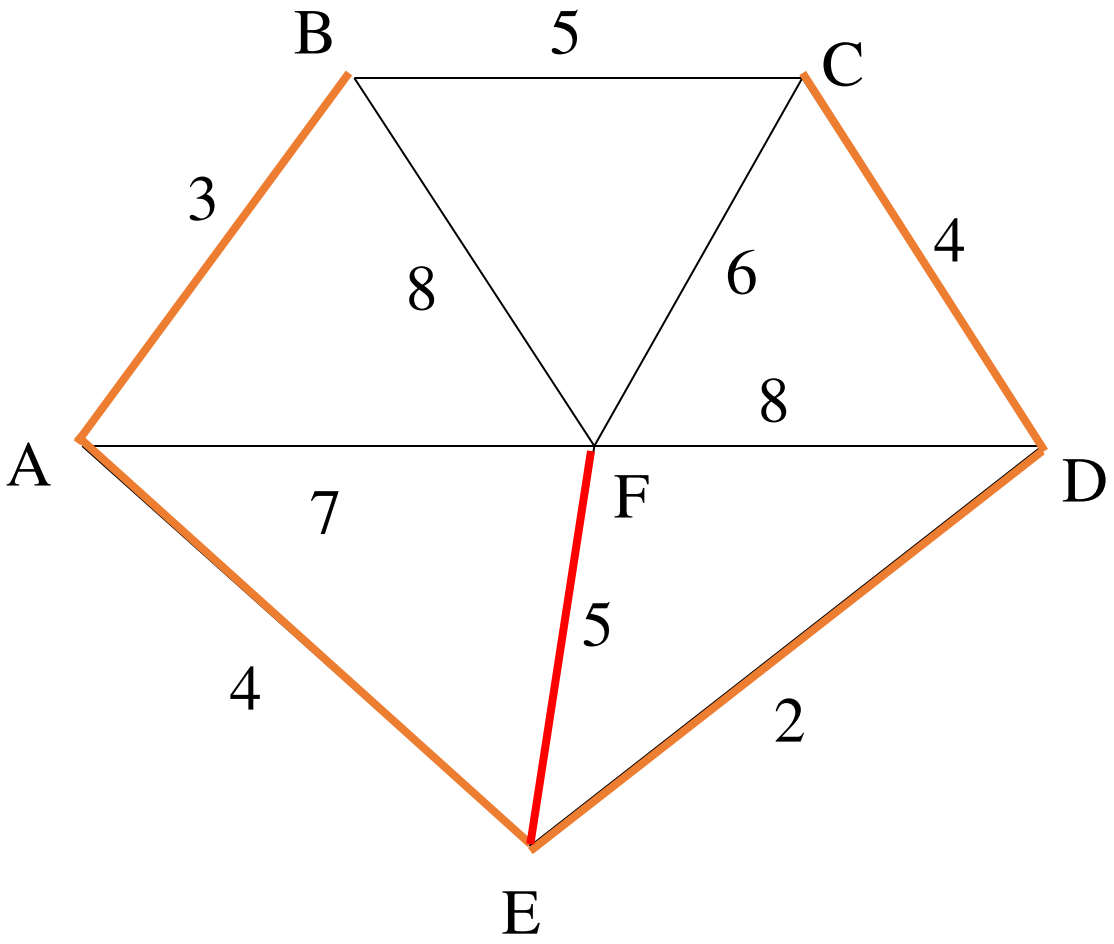
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4

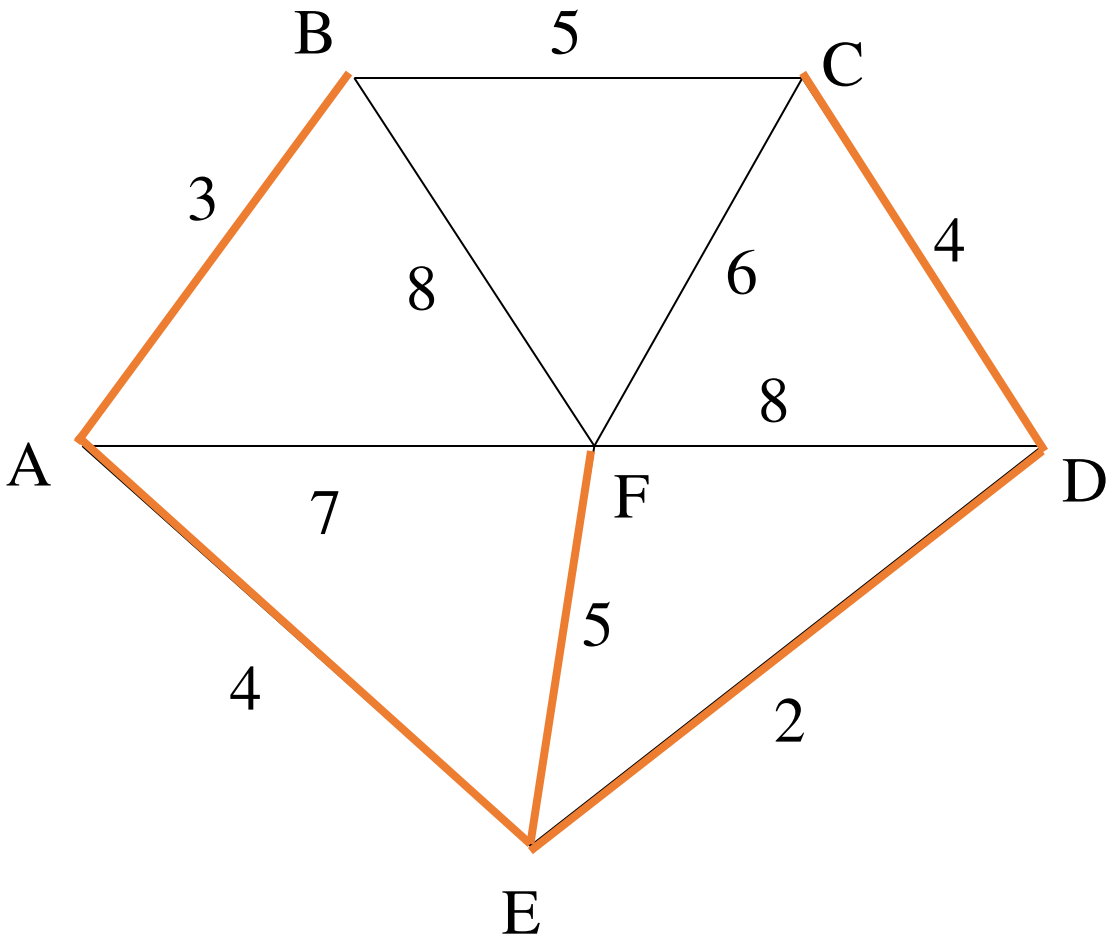
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4
BC 5 – forms a cycle
EF 5

Kruskal's Algorithm



All vertices have been connected.

The solution is

ED 2
AB 3
CD 4
AE 4
EF 5

Total weight of tree: 18

Notes about Kruskal's algorithm

- Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)
- Prim's Algorithm will work better for dense graphs.

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

Kruskal's algorithm

1. Select the shortest edge in a network
2. Select the next shortest edge which does not create a cycle
3. Repeat step 2 until all vertices have been connected

Prim's algorithm

1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

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: 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 been already found on preceding iterations (such vertices form a tree)

d_v is the length of the shortest path form source to *v*
$w(v,u)$ is the length (weight) of edge from *v* to *u*

Dijkstra's Pseudo Code

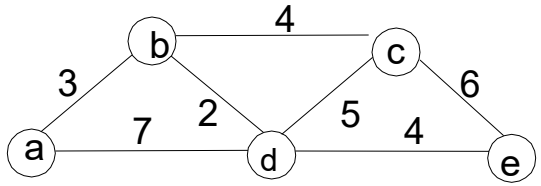
- Graph *G*, weight function *w*, root *s*

```
DIJKSTRA(G, w, s)
 1  for each v ∈ V
 2        do d[v] ← ∞
 3  d[s] ← 0
 4  S ← ∅   ▷ Set of discovered nodes
 5  Q ← V
 6  while Q ≠ ∅
 7        do u ← EXTRACT-MIN(Q)
 8           S ← S ∪ {u}
 9           for each v ∈ Adj[u]
10                do if d[v] > d[u] + w(u, v)
11                      then d[v] ← d[u] + w(u, v)
```

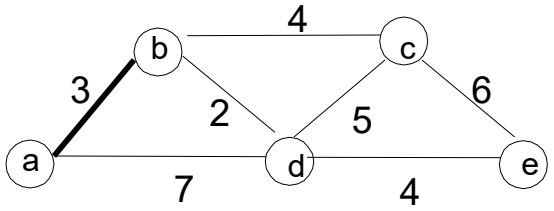
relaxing edges

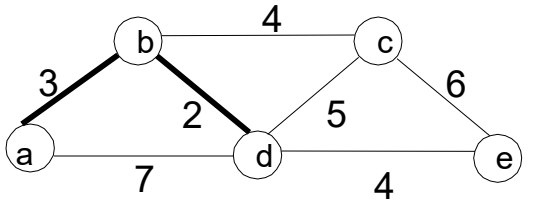
```
Dijkstra(v1, v2):
   for each vertex v:                          // Initialization
      v's distance := infinity.
      v's previous := none.
   v1's distance := 0.
   List := {all vertices}.

   while List is not empty:
      v := remove List vertex with minimum distance.
           mark v as known.
      for each unknown neighbor n of v:
         dist := v's distance + edge (v, n)'s weight.

         if dist is smaller than n's distance:
            n's distance := dist.
            n's previous := v.

reconstruct path from v2 back to v1,
following previous pointers.
```

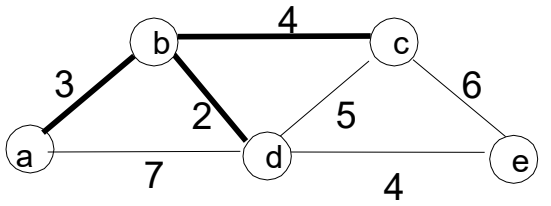
Example 1:

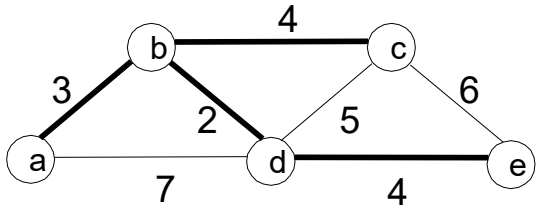


Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) d(b,3+2) e(-,∞)
d(b,5)	c(b,7) e(d,5+4)
c(b,7)	e(d,9)
e(d,9)	





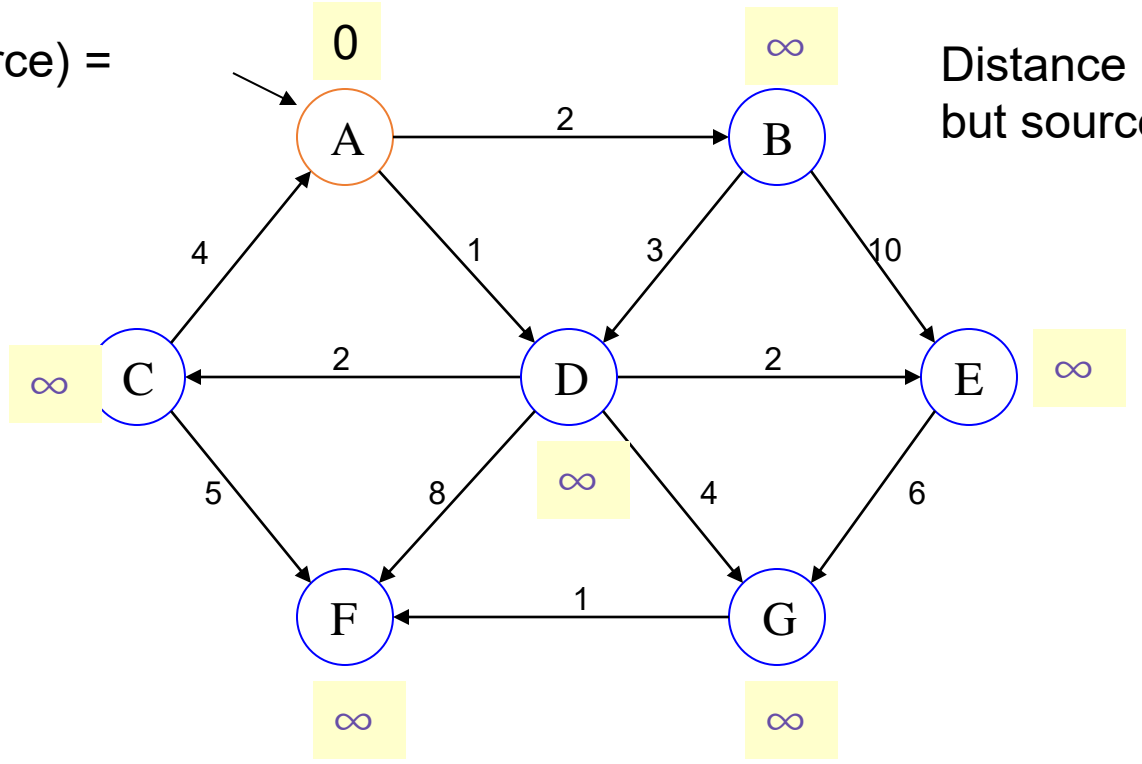




Example 2: Initialization

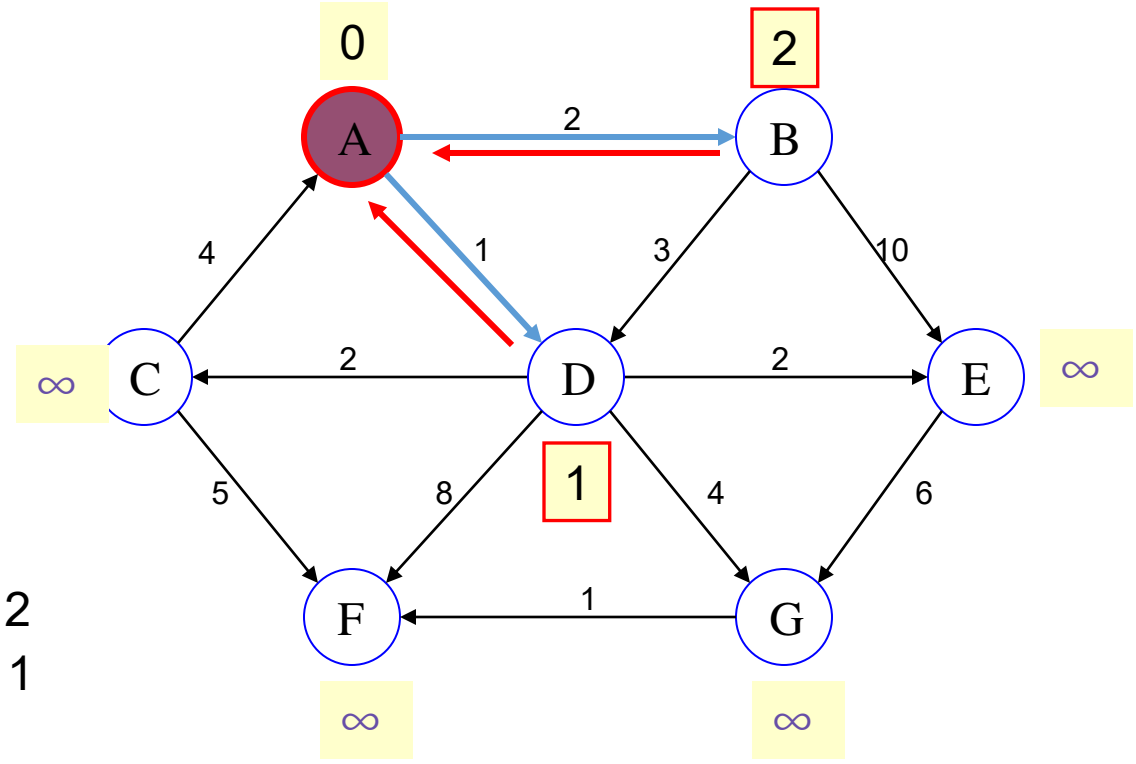
Distance(source) = 0

Distance (all vertices but source) = ∞



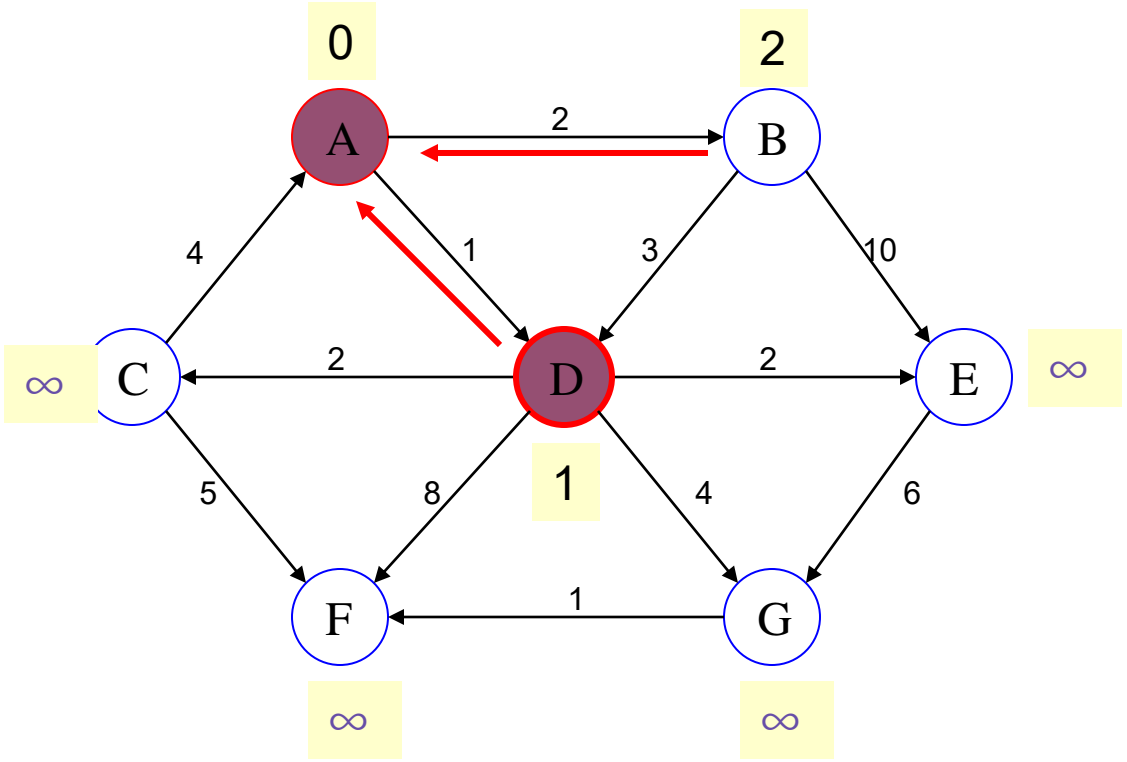
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



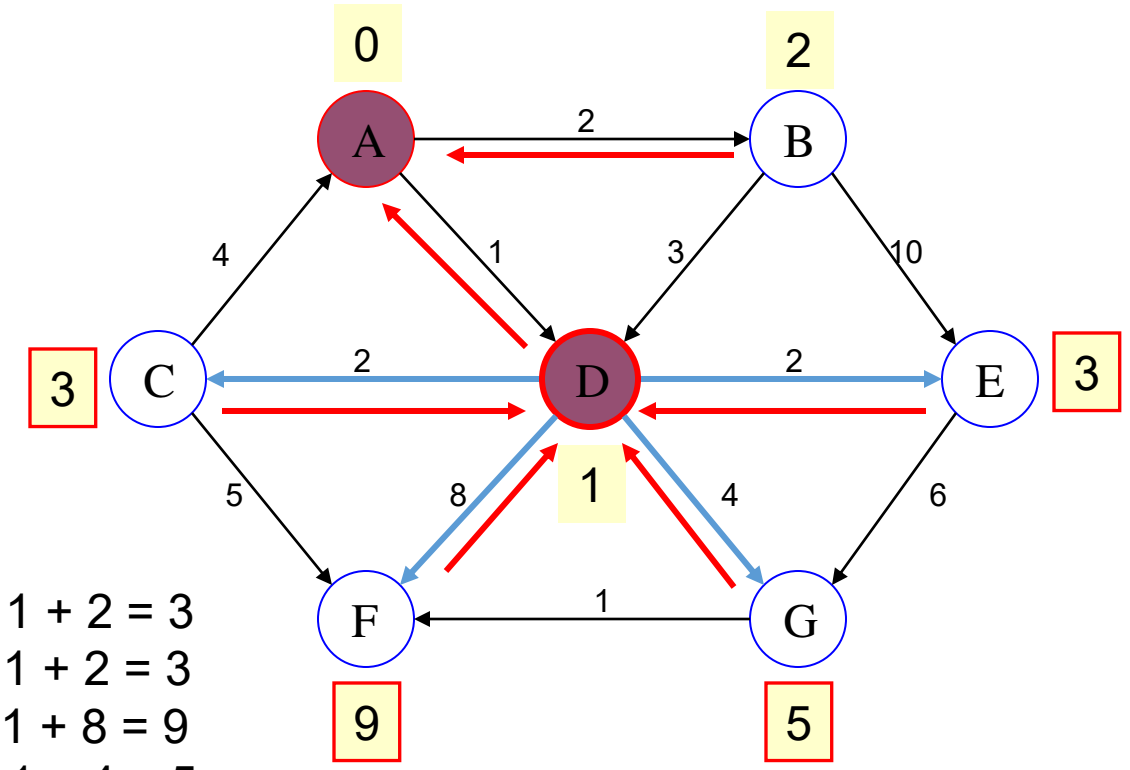
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

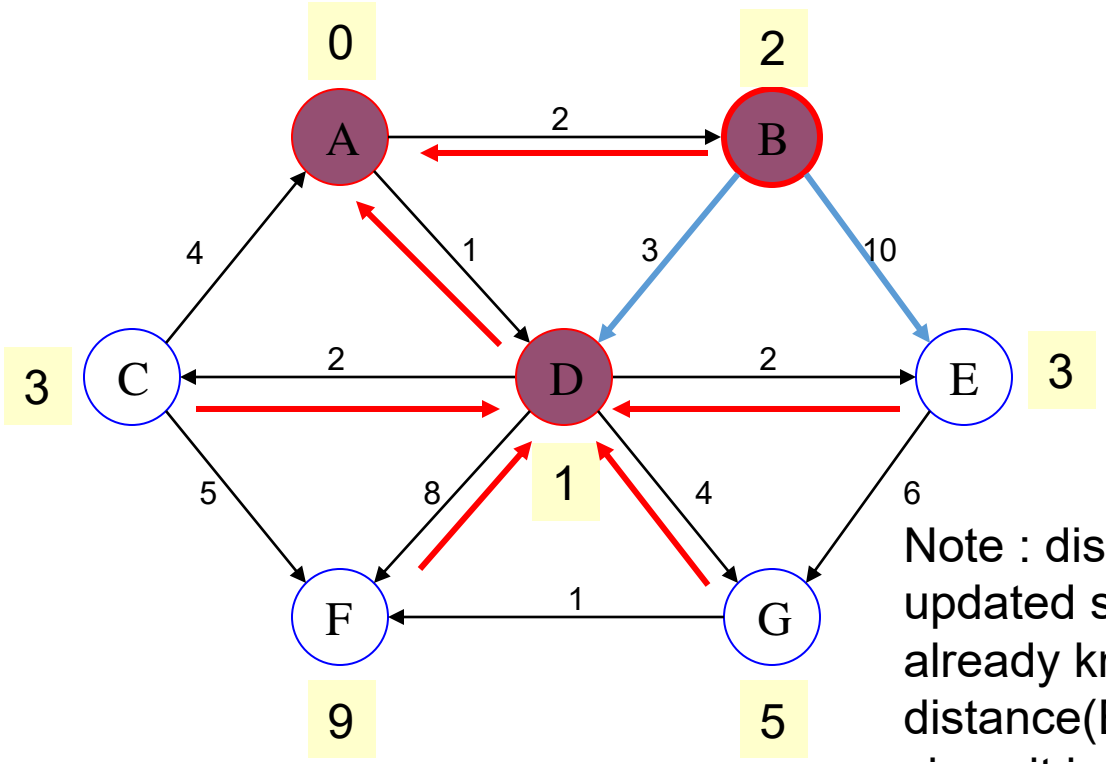
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

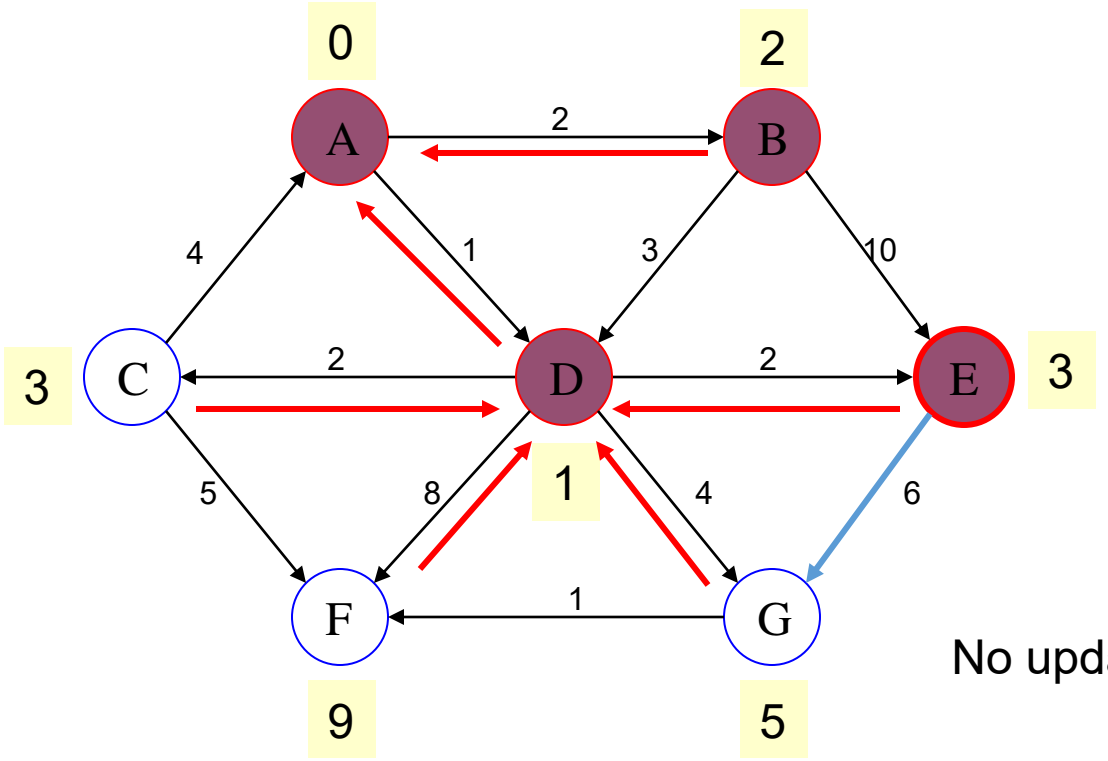
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

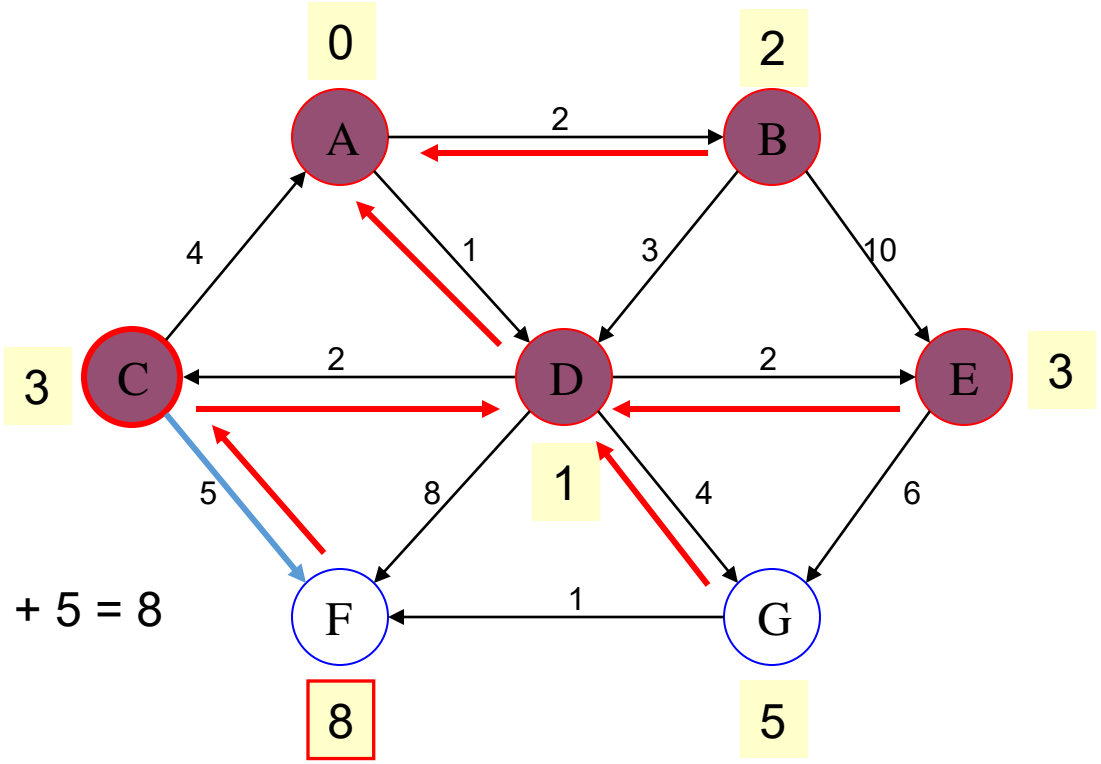
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

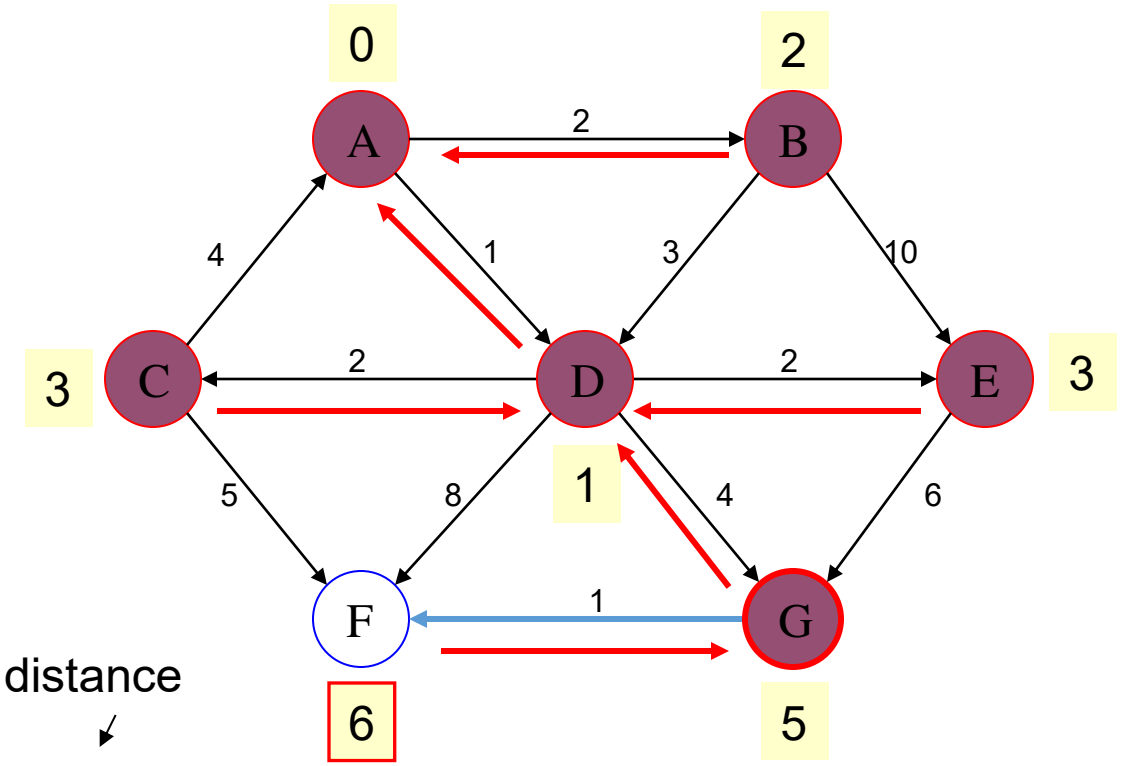
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

Pick vertex List with minimum distance (G) and update neighbors

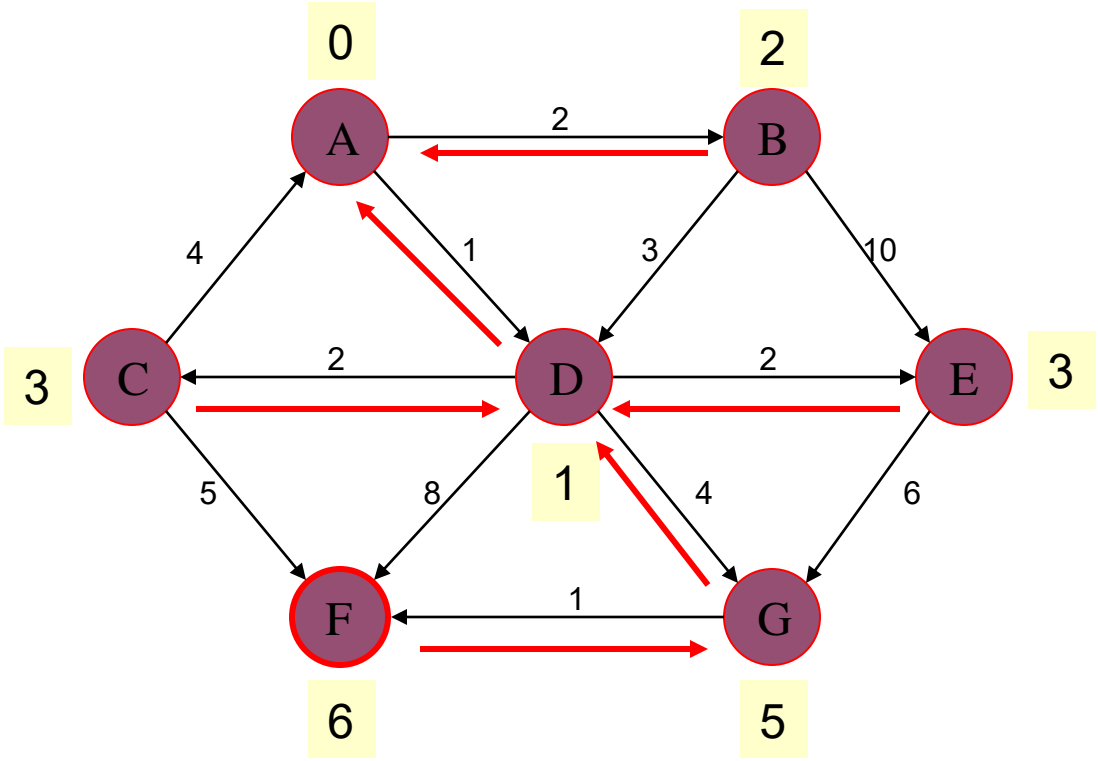


Previous distance

↓

Distance(F) = min (8, 5+1) = 6

Example (end)



Pick vertex not in S with lowest cost (F) and update neighbors

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!

Time Complexity: Using List

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

- Good for dense graphs (many edges)

- |V| vertices and |E| edges
- Initialization O(|V|)
- While loop O(|V|)
  - Find and remove min distance vertices O(|V|)
- Potentially |E| updates
  - Update costs O(1)

Total time $O(|V^2| + |E|) = O(|V^2|)$

Time Complexity: Priority Queue

For sparse graphs, (i.e. graphs with much less than $|V^2|$ edges) Dijkstra's implemented more efficiently by *priority queue*

- Initialization O(|V|) using O(|V|) buildHeap
- While loop O(|V|)
  - Find and remove min distance vertices O(log |V|) using O(log |V|) deleteMin

- Potentially |E| updates
  - Update costs O(log |V|) using decreaseKey

Total time O(|V|log|V| + |E|log|V|) = O(|E|log|V|)

- |V| = O(|E|) assuming a connected graph

Huffman Coding

1 Encoding and decoding messages

- Fixed-length coding
- Variable-length coding

2 Huffman coding

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?

Fixed-length coding

Problem

- Consider a message containing only the characters in the alphabet {'a', 'b', 'c', 'd'}.
    - The ASCII code (unsigned int) representation of the characters in the message is not appropriate: we have only 4 characters, each of which is coded using 8 bits.
    - A code that uses only 2 bits to represent each character would be enough to store any message that is a combination of only 4 characters.
- Fixed-length coding
    - The code of each character (or symbol) has the same number of bits.

Example

Message = aabccdaa...
Alphabet = {'a', 'b', 'c', 'd'}
Encoding = 0000011010110000...

Encoding scheme

Char	Code
a	00
b	01
c	10
d	11

Question

How many bits do we need to encode uniquely each character in a message made up of characters from an *n*-letter alphabet ?

Answer

$\lceil \log n \rceil$ bits at least.

Variable-length coding

Each character is assigned a different length.

Encoding scheme 1

Char	Code
a	10
b	1
c	0

Example

Message = aaabcabc. . .
Alphabet = {'a', 'b', 'c', 'd'}
Encoding = 101010101010. . .

Problem when decoding

When decoding, there is more than one possible message:
Message = aaabcabc. . .which is correct,
or, Message = bcbcbcbc. . .which is incorrect.

Consider now the following encoding scheme:

Encoding scheme 2

Char	Code
a	00
b	01
c	1

Example

Message = aaabcabc. . .
Alphabet = {'a', 'b', 'c', 'd'}
Encoding = 000000011000111. . .

Decoding

When decoding, only one possible message:
Message = aaabcabc. . .which is the correct message.

Huffman coding...

Objective

- Huffman coding is an algorithm used for lossless data compression.

- By lossless, it is meant that the exact original data can be recovered by decoding the compressed data.

Applications

- Several data compression softwares, WinZip, zip, gzip,. . . Use lossless data encoding.

• Consider the alphabet {'a', 'b', 'c', 'd', 'e', 'f', 'g'}

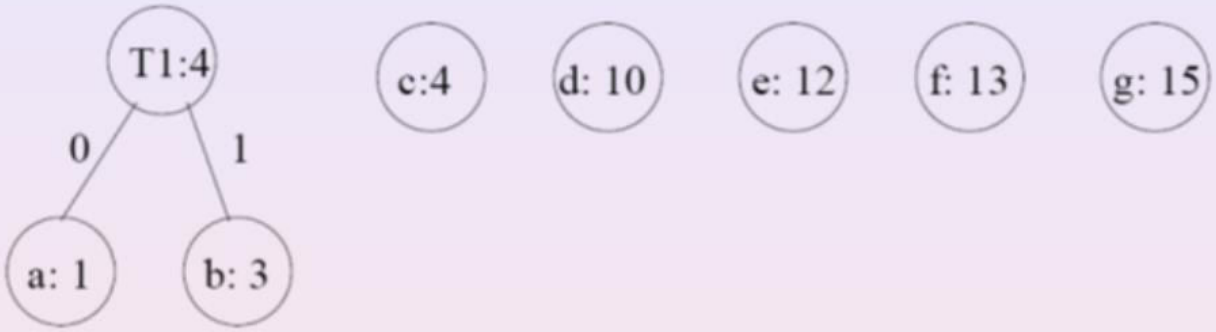
• Count the frequency of each character in the message (number of times the character appears). For example:

Char	Frequency
a	1
b	3
c	4
d	10
e	12
f	13
g	15

meaning that 'a' appears only once in the message, 'b' appears 3 times in the message. . .

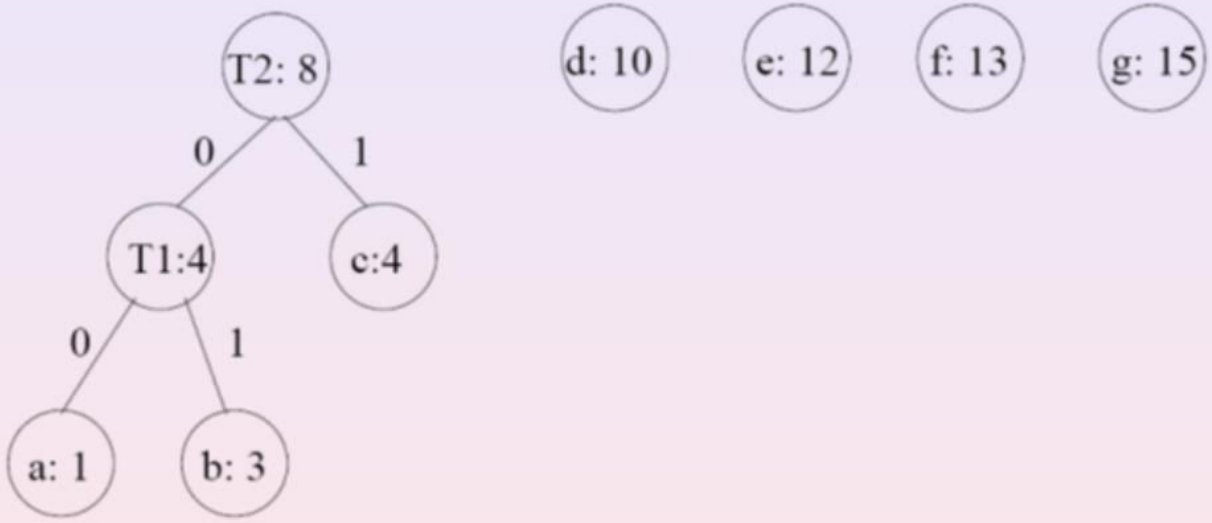
1

Merge a and b:



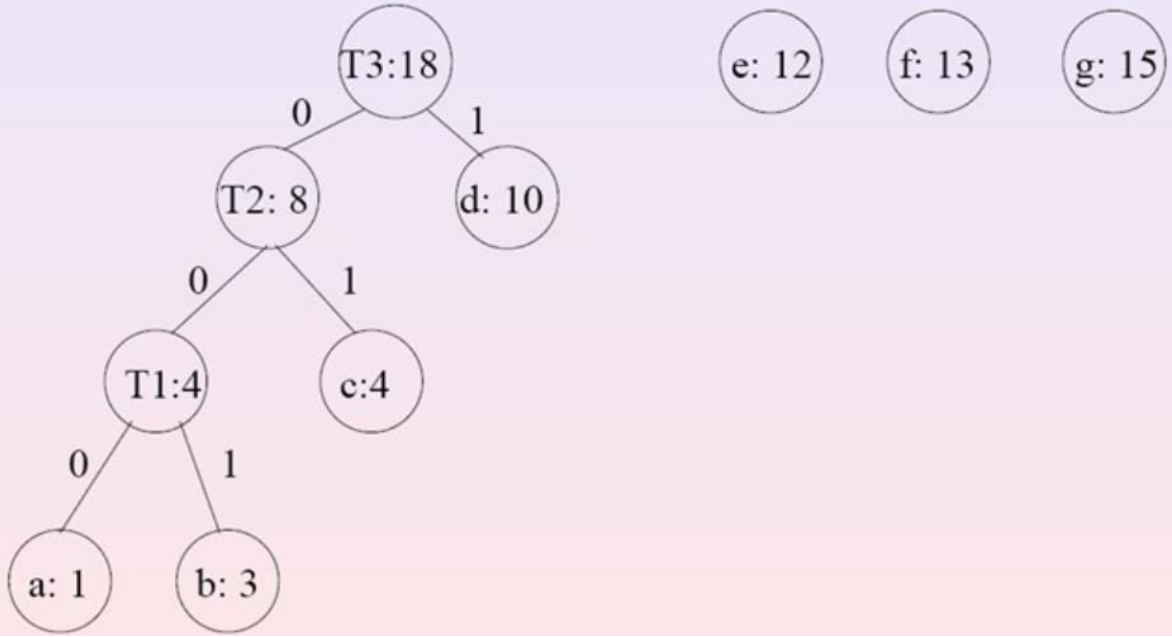
2

Merge T1 and c:



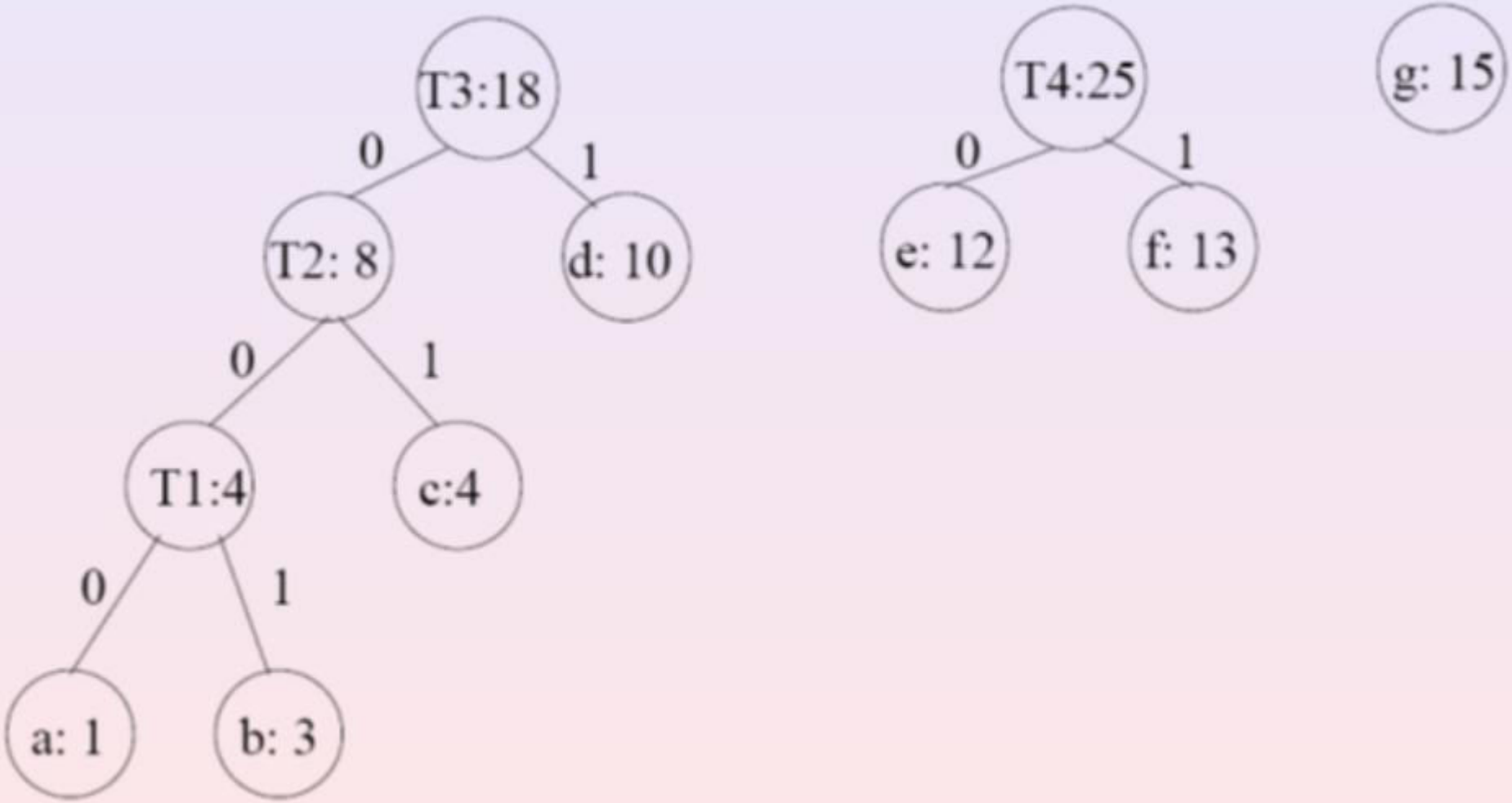
3

Merge T2 and d:



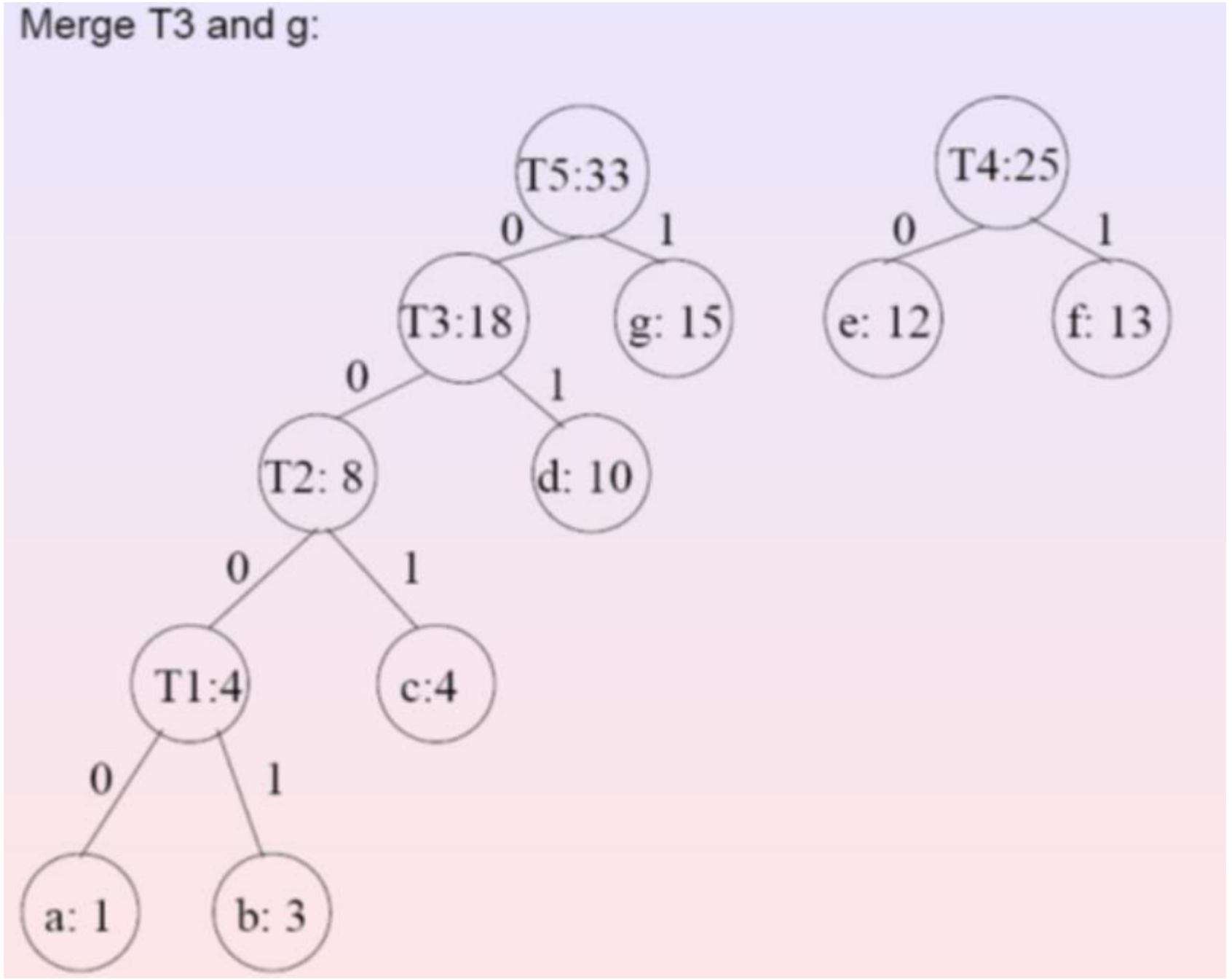
4

Merge f and e:



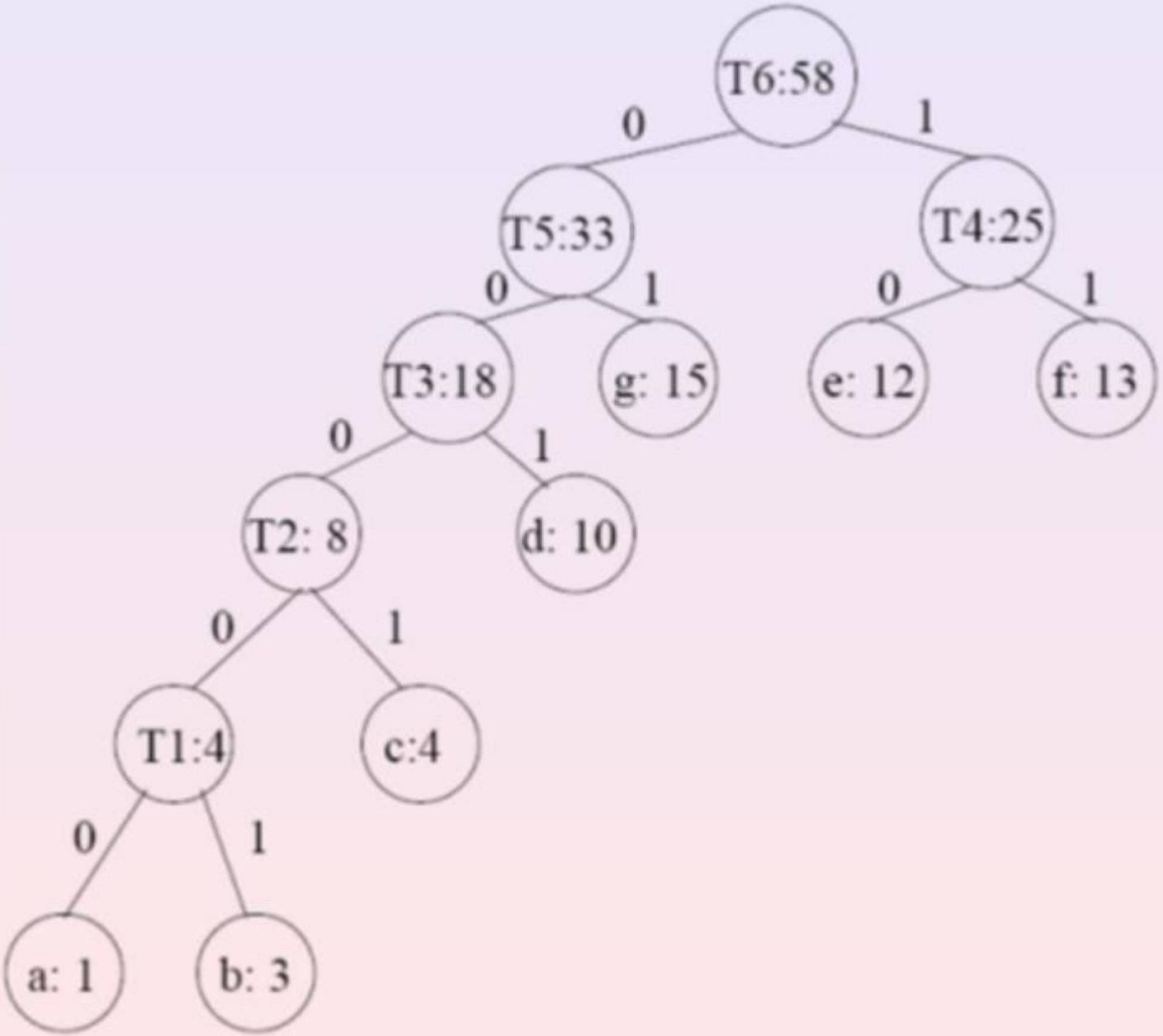
5

Merge T3 and g:



6

Merge T5 and T4:



The code of each character is obtained by concatenating the labels of the edges on the path from the root to the node representing the character:

Char	Frequency
a	1
b	3
c	4
d	10
e	12
f	13
g	15

Char	Code	# of bits
a	00000	5
b	00001	5
c	0001	4
d	001	3
e	10	2
f	11	2
g	01	2

Let f_i be the frequency of a character and d_i the number of bits in the code of that character:

- The total number of bits required to encode the message M = $\sum_{i=1}^{7} d_i f_i$
  = = 5·1+5·3+4·4+3·10+2·13+2·12+2·15 = 146 bits.

- We need 146 bits to encode the message with the given frequency table.

Huffman codes

- Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves
- Optimal binary tree minimizing the expected (weighted average) length of a codeword can be constructed as follows

Huffman's algorithm

Initialize *n* one-node trees with alphabet characters and the tree weights with their frequencies.

Repeat the following step *n*-1 times: join two binary trees with smallest weights into one (as left and right subtrees) and make its weight equal the sum of the weights of the two trees.

Mark edges leading to left and right subtrees with 0's and 1's, respectively.

Example

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

codeword 11 100 00 01 101

average bits per character: 2.25

for fixed-length encoding: 3

compression ratio: (3-2.25)/3*100% = 25%