Design and Analysis of Algorithms (UE17CS251)

Unit V - Greedy Technique

Mr. Channa Bankapur
channabankapur@pes.edu

Change-making problem:

How can a given amount of money be made with the least number of coins of given denominations?

Example:

A person buys things worth Rs. 72 and gives a Rs. 100 bill. How does the cashier give change for Rs. 28?

Change-making problem:

How can a given amount of money be made with the least number of coins of given denominations?

Example:
Change for Rs. **28**

10 + **18**
10 + 10 + **8**
10 + 10 + 5 + **3**
10 + 10 + 5 + 2 + **1**
10 + 10 + 5 + 2 + 1

Suppose the available coin denominations are:
Re. 1, Rs. 2, **Rs. 4**, Rs. 5, Rs. 10.

Example:
Change for Rs. **28**

10 + **18**
10 + 10 + **8**
10 + 10 + 5 + **3**
10 + 10 + 5 + 2 + **1**
10 + 10 + 5 + 2 + 1 **(5 coins)**

Suppose the available coin denominations are:
Re. 1, Rs. 2, **Rs. 4**, Rs. 5, Rs. 10.

Example:
Change for Rs. **28**

10 + **18**
10 + 10 + **8**
10 + 10 + 5 + **3**
10 + 10 + 5 + 2 + **1**
10 + 10 + 5 + 2 + 1 **(5 coins)**

10 + **18**
10 + 10 + **8**
10 + 10 + 4 + **4**
10 + 10 + 4 + 4 **(4 coins)**

Eg: Change-making problem

Consider the general instance of the following well-known problem. Give change for amount **n** using the minimum number of coins of denominations $\mathbf{d_1 < d_2 < \ldots < d_m}$.

Eg: Change-making problem

Consider the general instance of the following well-known problem. Give change for amount **n** using the minimum number of coins of denominations $\mathbf{d_1 < d_2 < \ldots < d_m}$.

Dynamic Programming solution!

$F(n) = \min \text{ for } j=n \geq d_j \{F(n - d_j)\}+1 \text{ for } n>0$
and $F(0) = 0$

Greedy Technique

- Locally optimal choices sometimes leads to globally optimal solution.
- Greedy choice: Always make a choice that look best at the moment.
- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- Dynamic Programming solution is an overkill if there is a solution which makes locally optimal choices that leads to globally optimal solution.

Solving an **optimization problem** using Greedy Technique:

- Construct a solution through a sequence of steps.
- Each step to expands a partially constructed solution so far, until a complete solution to the problem is reached.

On each step, the choice made must be

- **Feasible:** it has to satisfy the problem's constraints.
- **Locally optimal:** it has to be the best local choice among all feasible choices available on that step.
- **Irrevocable:** Once made, it cannot be changed on subsequent steps of the algorithm.

A **globally-optimal solution** can be found by a series of **local improvements** from a starting configuration.

Q: Find the minimum number of moves needed for a chess knight to go from one corner of a 10 × 10 checkerboard to the diagonally opposite corner.

Let the cells of the checkerboard are represented as (i, j) where i and j are in the range [1, 10].

Source cell: (1, 1)
Destination cell: (10, 10)

Q: Find the minimum number of moves needed for a chess knight to go from one corner of a 10 × 10 checkerboard to the diagonally opposite corner.

From (1, 1) to (10, 10)

(3, 2), (4, 4),
(6, 5), (7, 7),
(9, 8), (10, 10)
6 moves (3 sets of moves)

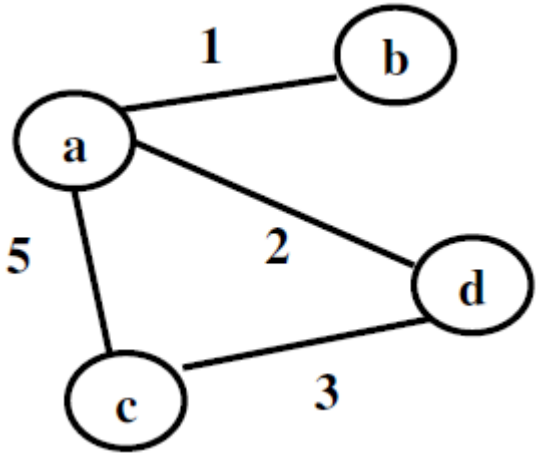
What if it's a 16 x 16 checkerboard?
What if it's a 8 x 8 checkerboard?

Examples of Greedy Algorithms:

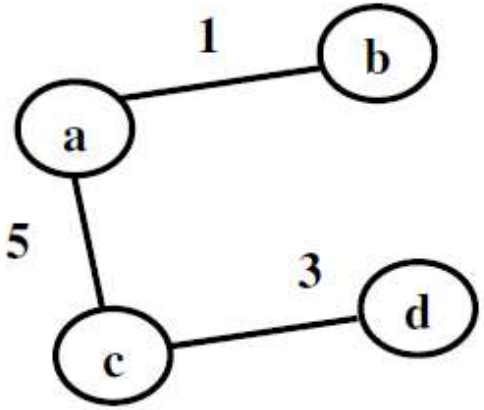
- Coin-change problem
- Minimum Spanning Tree (MST)
  - Prim’s Algorithm
  - Kruskal’s Algorithm
- Single-source shortest paths
  - Dijkstra’s Algorithm
- Huffman codes

Spanning Tree of a connected graph G is a connected acyclic sub-graph (tree) that includes all vertices of G.

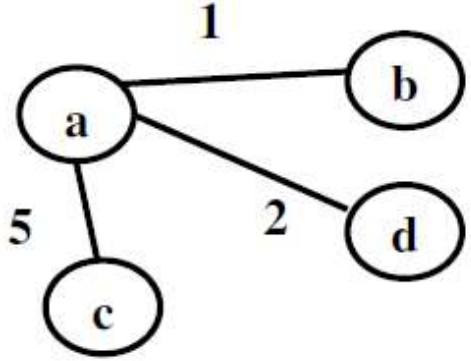
Minimum Spanning Tree (MST) of a weighted, connected graph G is a spanning tree of G with minimum total weight.



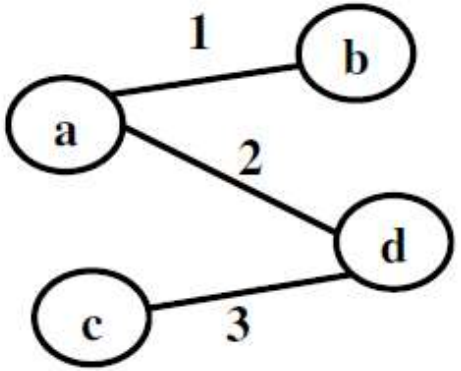
The spanning trees for the above graph are as follows:



Weight (T_1) = 9

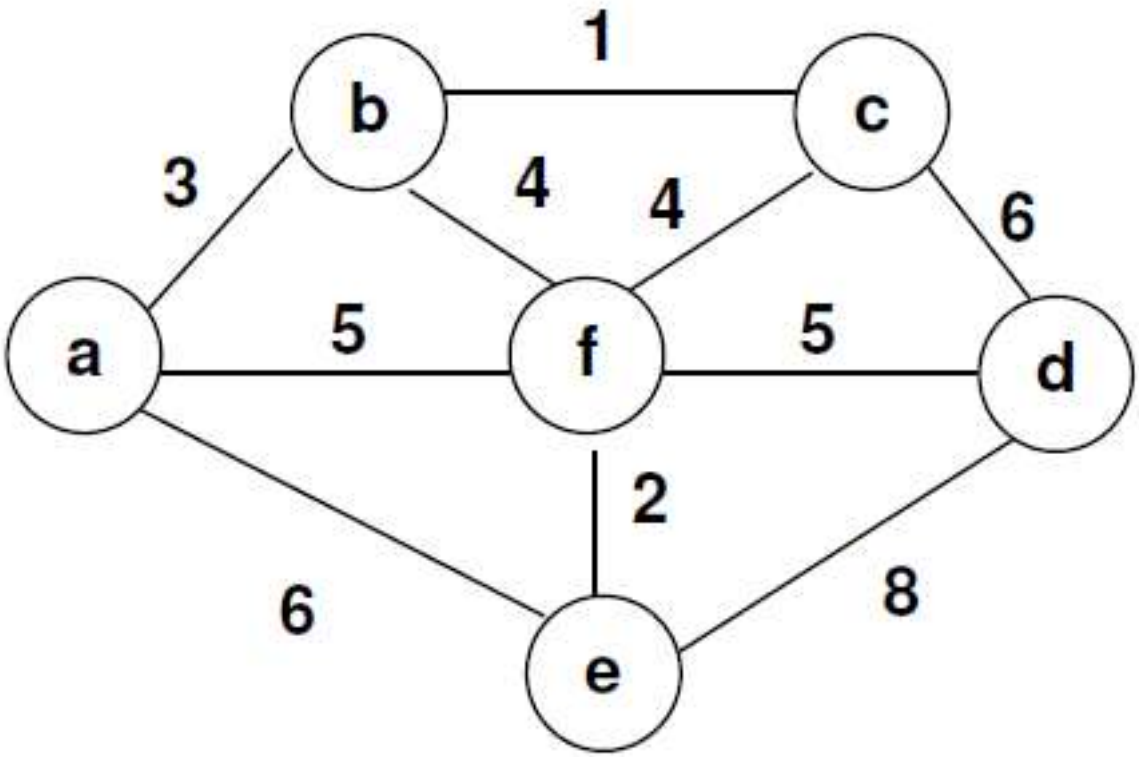


Weight (T_2) = 8



Weight (T_3) = 6

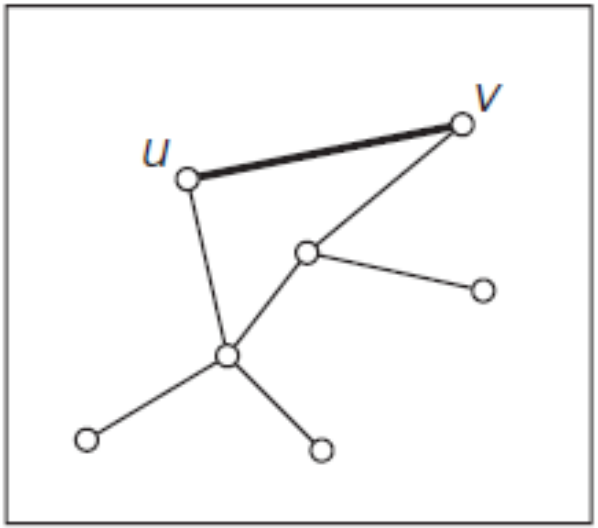
Find **Minimum Spanning Tree** of the following graph.

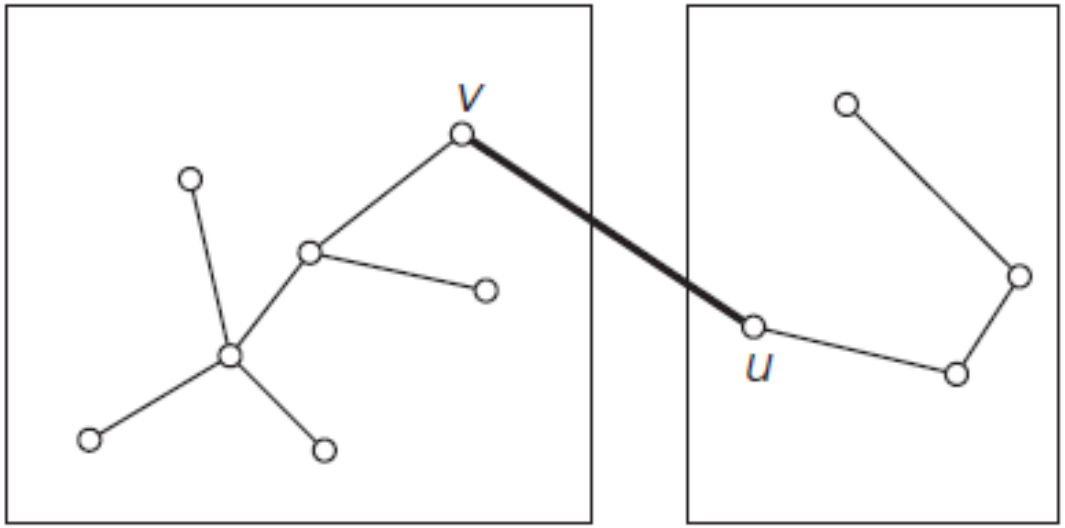


Don’t bother about Prim’s and Kruskal’s algorithm at the moment!

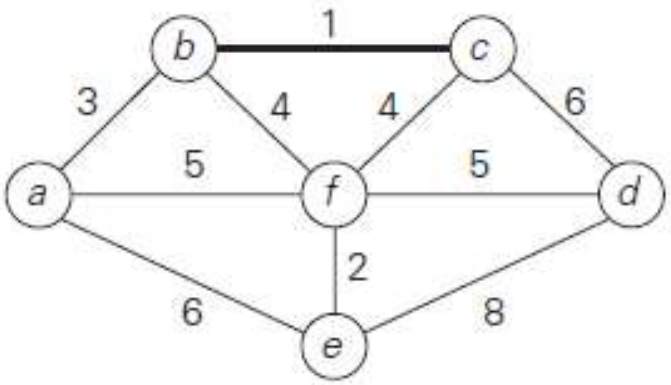
Kruskal's Algorithm:

- Sort the edges of the graph by weight in nondecreasing order.
- Start with a forest of **n** trees, where each tree is single node of the graph.
- For i = 1 to n-1 do
  - Among the edges which are not yet included, select the one with minimum weight that does not form a cycle. This operation merges two trees and hence reduces the number of trees by one.
- Return the only tree, which is a minimum spanning tree.

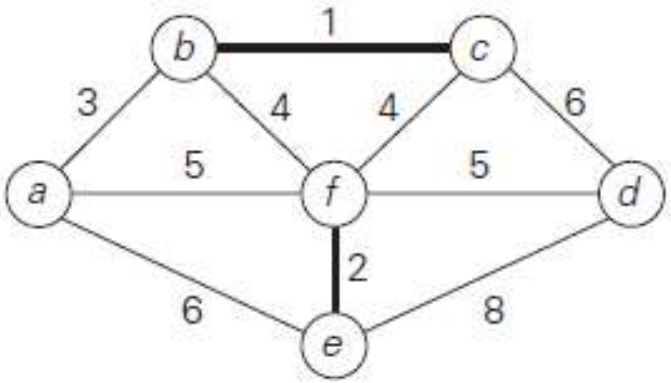




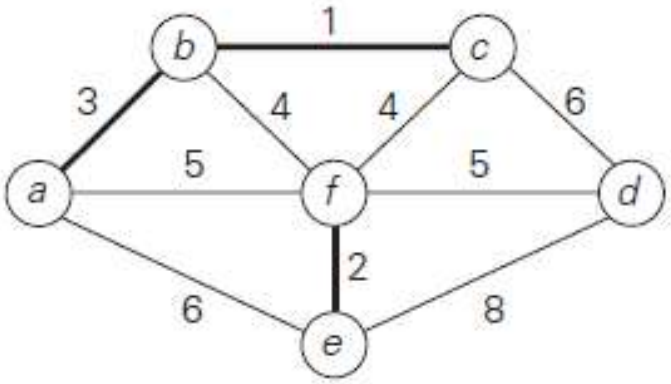
bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



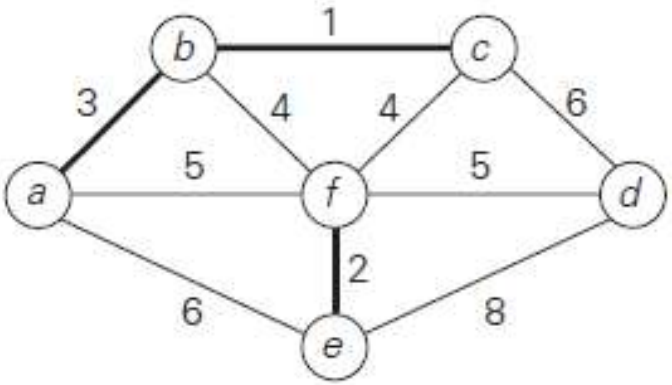
bc	**ef**	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



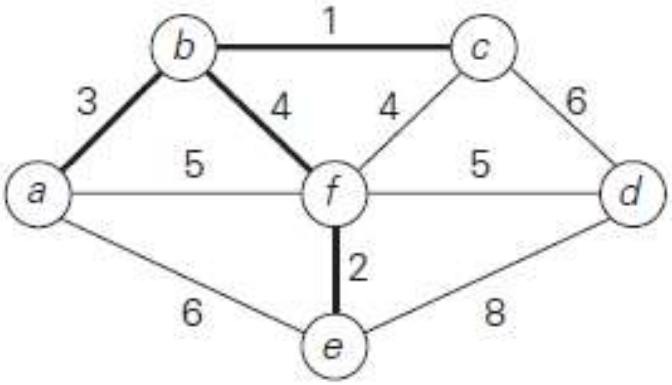
bc	ef	**ab**	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



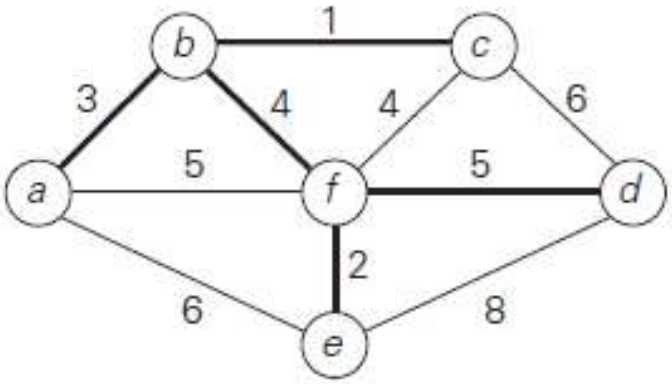
bc	ef	**ab**	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



bc	ef	ab	**bf**	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



bc	ef	ab	bf	cf	af	**df**	ae	cd	de
1	2	3	4	4	5	5	6	6	8

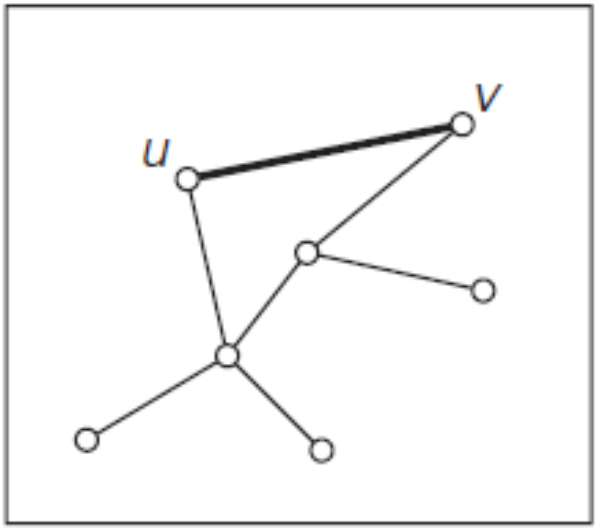


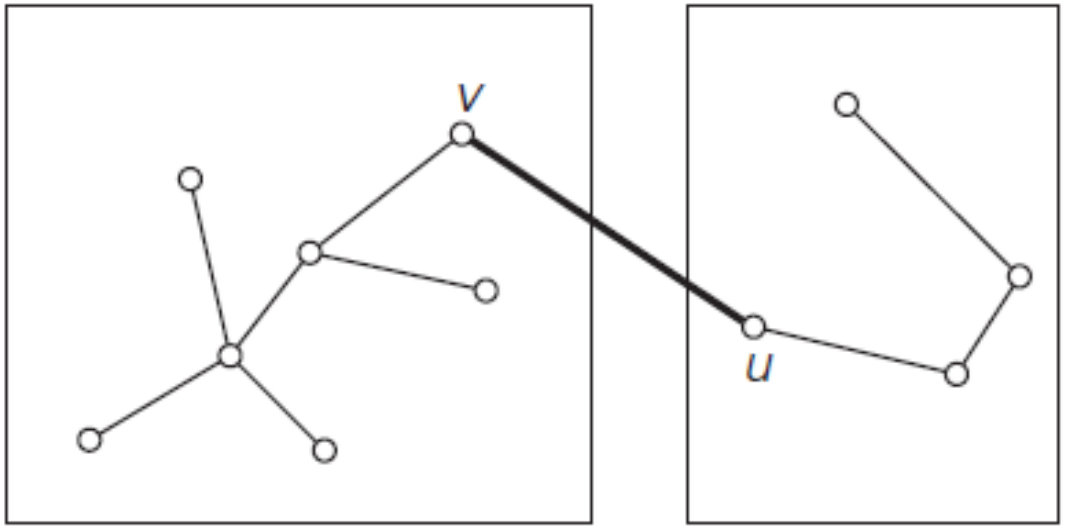
Kruskal's Algorithm:

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 $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //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

Kruskal's Algorithm:



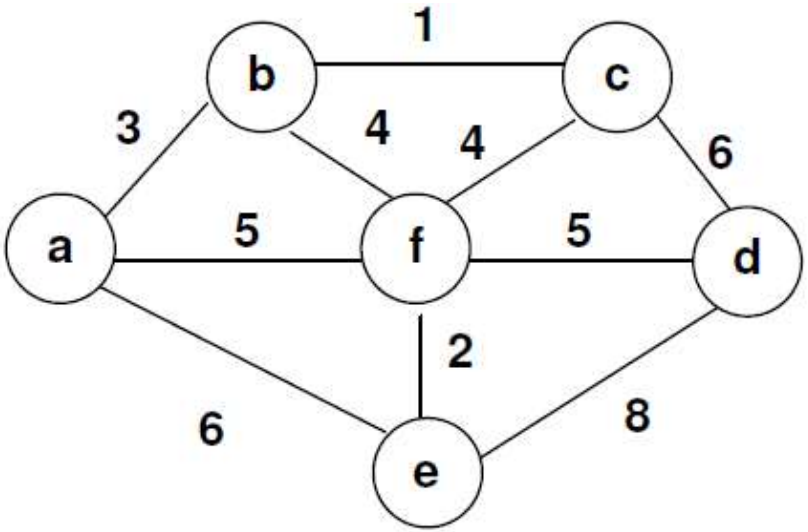


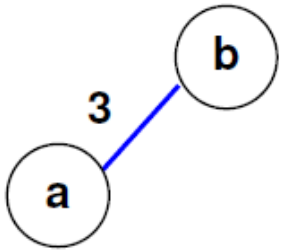
With an efficient Sorting algorithm and an Union-Find algorithm.

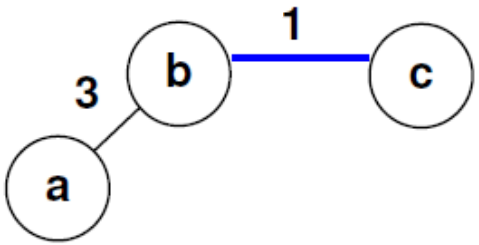
Efficiency: $\Theta(|E| \log |E|)$

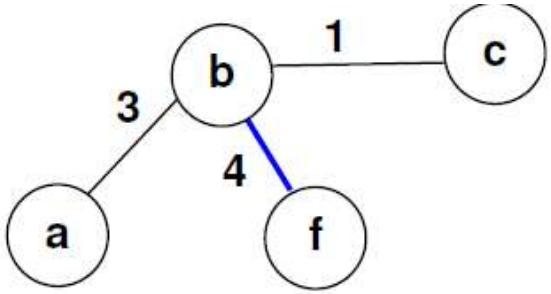
- **Prim's Algorithm:**
- Start with a tree, $\mathbf{T_1}$, consisting of one vertex.
- Adjacent vertices of the vertex in $\mathbf{T_1}$ are "fringe" vertices of $\mathbf{T_1}$.
- For i = 2 to n do
  - Construct $\mathbf{T_i}$ from $\mathbf{T_{i-1}}$ by adding the fringe vertex with the minimum weight edge from the set. The vertex is removed from the set of fringe vertices.
  - Add the adjacent vertices of the vertex to the set of fringe vertices which are not in $\mathbf{T_i}$.
  - Remove vertices from the set of fringe vertices where the new vertex is one of the terminal vertex of the edge.
- Return $\mathbf{T_n}$ which is a minimum spanning tree.

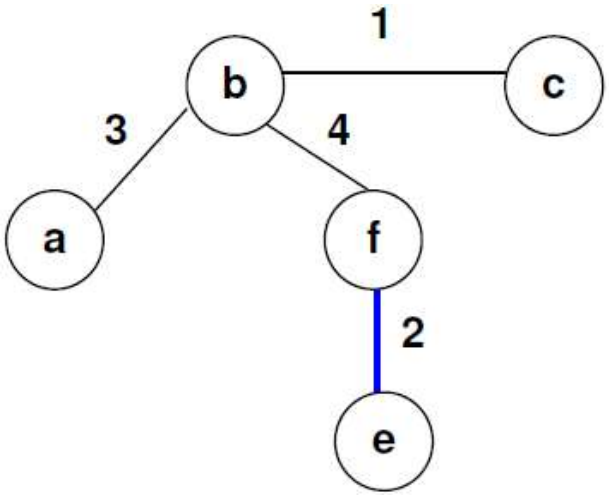
Prim's Algorithm:

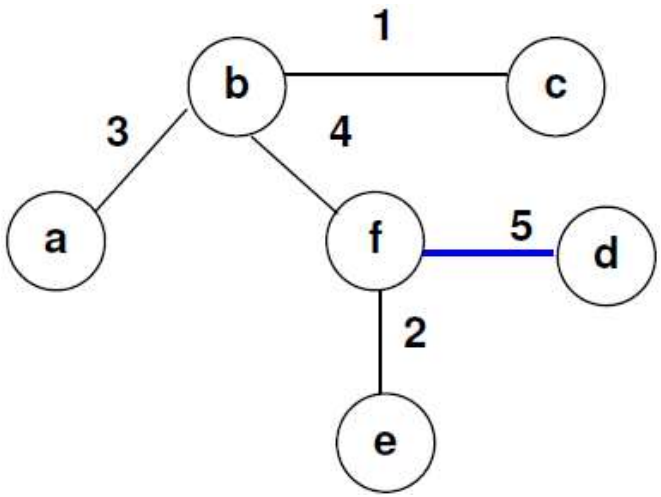




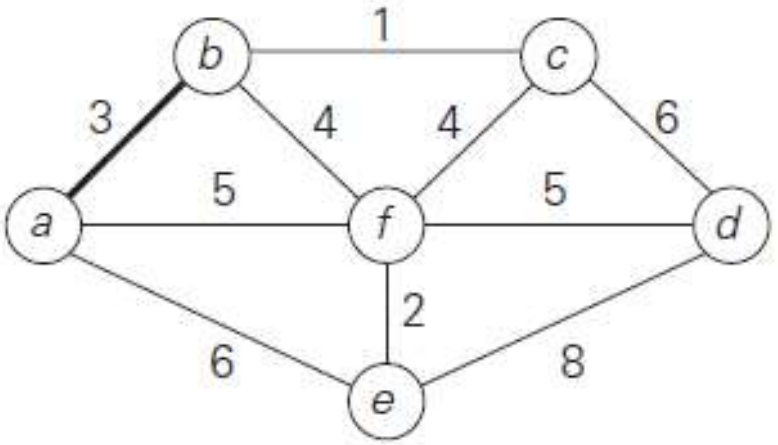
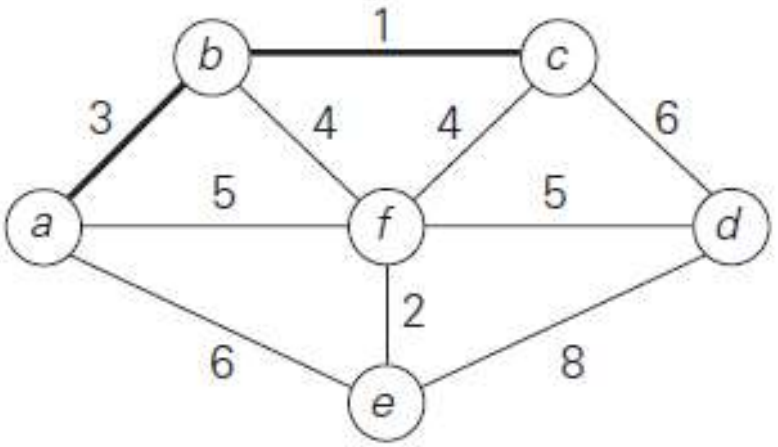


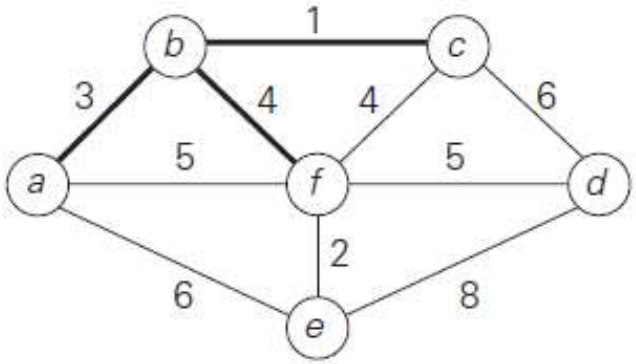
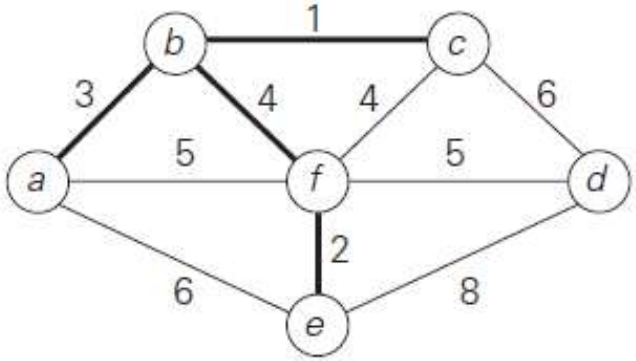
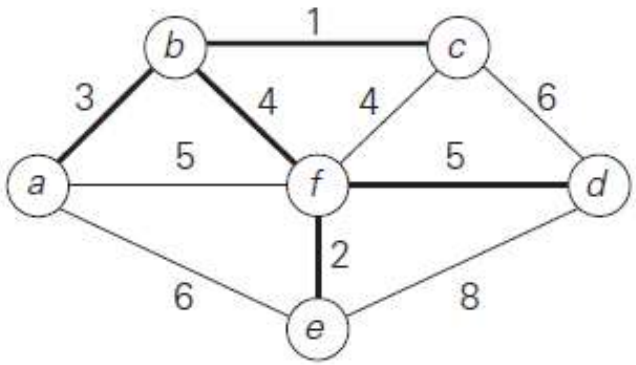






Prim's Algorithm:

Tree vertices	Remaining vertices	Illustration
a(–, –)	**b(a, 3)** c(–, ∞) d(–, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(–, ∞) e(a, 6) f(b, 4)	

Tree vertices	Remaining vertices	Illustration
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:

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
$E_T \leftarrow \varnothing$
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

Prim's Algorithm:

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 $V - V_T$

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

Move u^* from the set $V - V_T$ to the set of tree vertices V_T.

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.

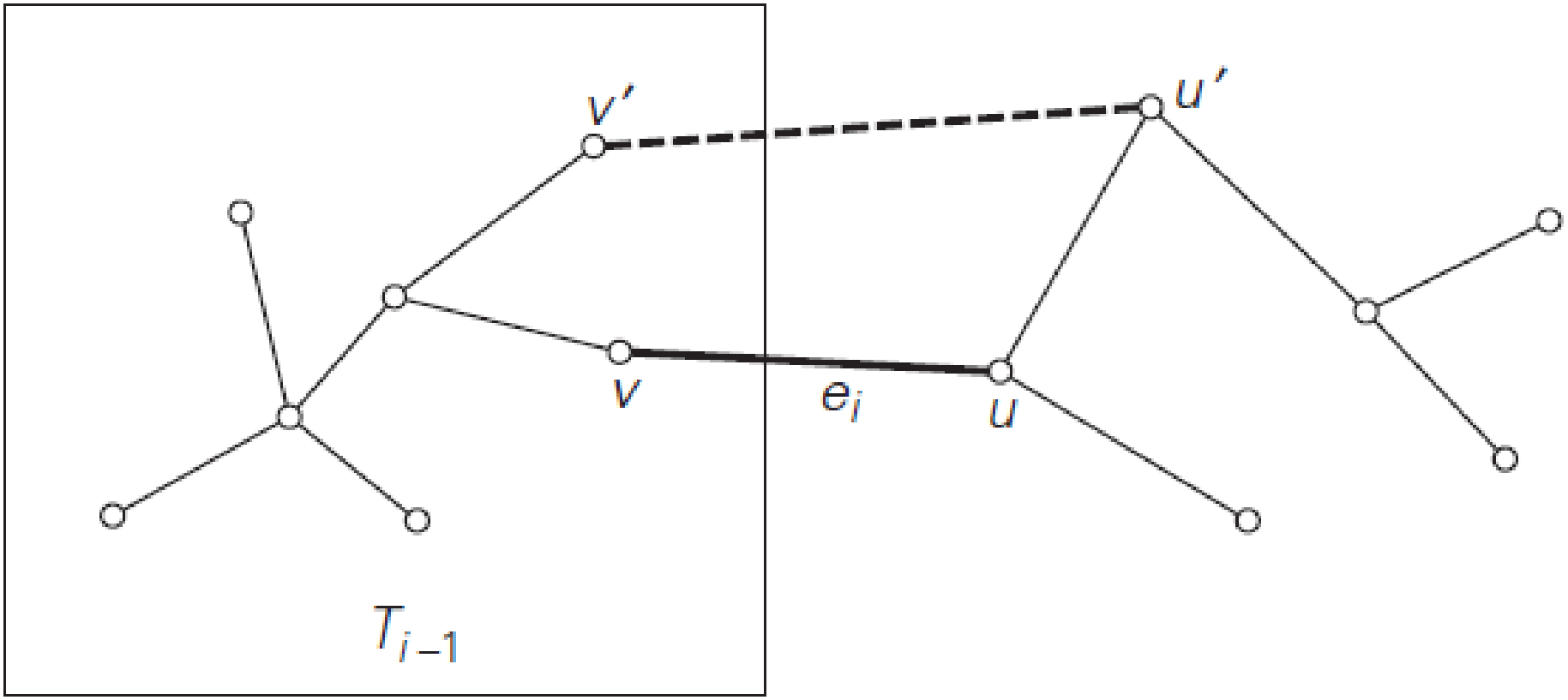
The set **V - V^T** is a priority queue and could be represented as a min-heap.

First operation is, extracting min valued element from the heap and hence takes O(**log|V|**) time. As it is executed for **|V|-1** times, the operation overall takes O(**|V| log|V|**) time.

Second operation is, for each adjacent vertex of **u***, which is in **V-V^T**. It may try to decrease the value of the key in the heap, which takes O(**log|V|**) time. The outer loop runs for **|V|-1** times (that is, for each vertex **u***) and the inner loop runs for each adjacent vertex of **u***. Put together the "Decrease Key" operation runs once for each edge of the graph. That's why it is O(**|E|log|V|**) time.

Running time = O(**|V| log|V|**) **+** O(**|E| log|V|**)
= O(**(|E|)log|V|**) since, **|E| >= |V|-1** for a connected graph.

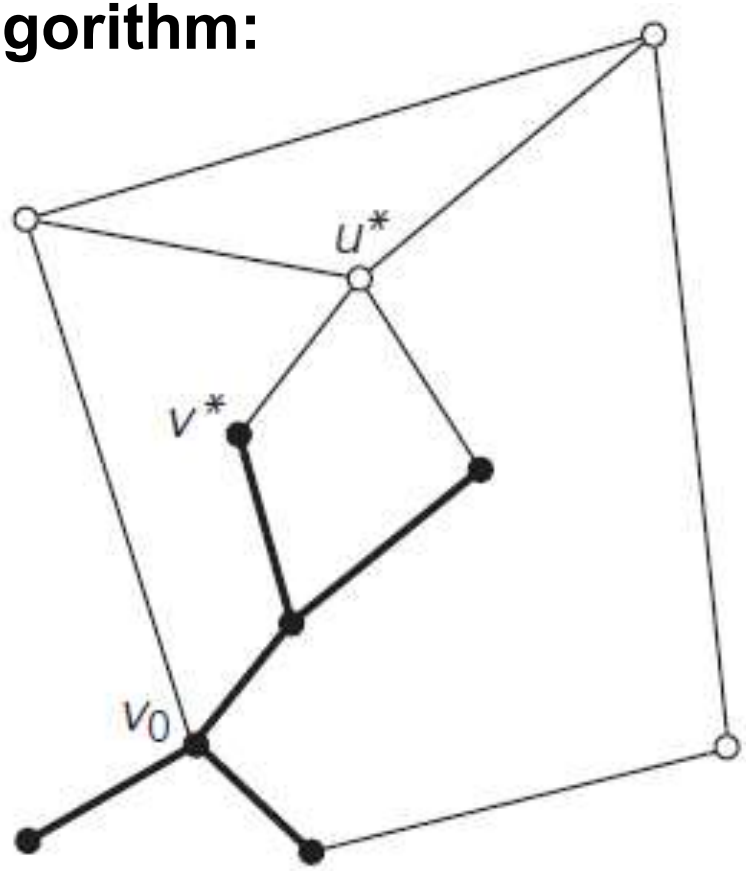
Does Prim's algorithm always yield a minimum spanning tree?



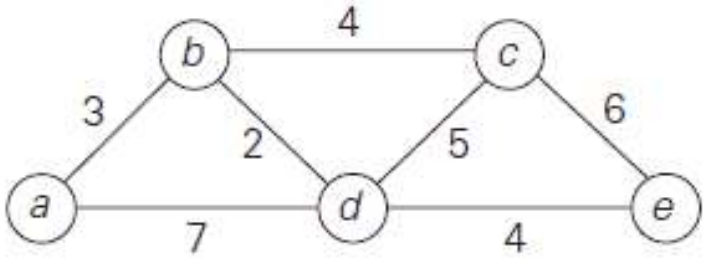
Dijkstra's Algorithm:

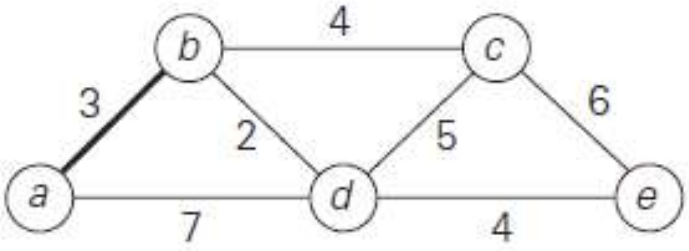
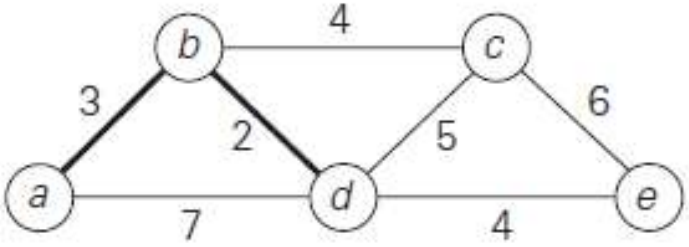
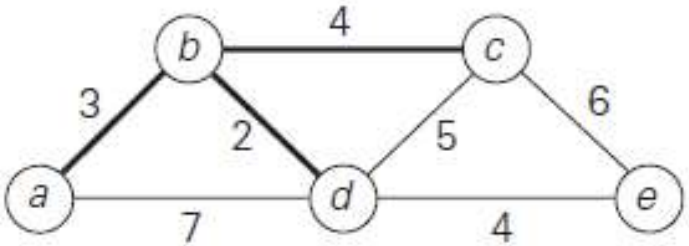
- Solves **single-source shortest-paths** problem.
- Problem: For a given vertex called the **source** in a weighted connected graph, find the shortest paths to all the other vertices.
- A **family of paths** each leading from the source to a different vertex in the graph. Some paths may have a common prefix. If a shortest path from the source to a vertex 'b' goes through a vertex 'a', then the shortest path from the source to the vertex 'a' is the same path until the vertex 'a'.

Dijkstra's Algorithm:

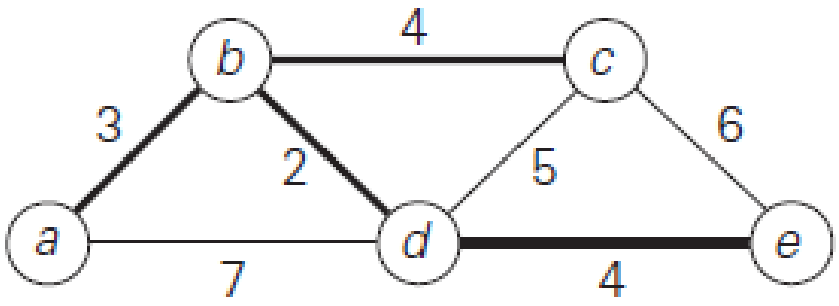


Idea of Dijkstra's algorithm. The subtree of the shortest paths already found is shown in bold. The next nearest to the source v_0 vertex, u^*, is selected by comparing the lengths of the subtree's paths increased by the distances to vertices adjacent to the subtree's vertices.



Tree vertices	Remaining vertices	Illustration
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)

from a to b : $a - b$ of length 3

from a to d : $a - b - d$ of length 5

from a to c : $a - b - c$ of length 7

from a to e : $a - b - d - e$ of length 9

Dijkstra's Algorithm:

- First, it finds the shortest path from the source to a vertex nearest to it, then to the second nearest, and so on.
- The algorithm finds the shortest paths to the graph's vertices in order of their shortest distance from the source vertex.
- All the vertices including the source, and the edges of the shortest paths leading to them from the source form a tree, which is a subgraph of the given graph.
- The algorithm is applicable to undirected and directed graphs with **nonnegative weights** only.

Dijkstra's Algorithm:

- The set of vertices not in ***Ti***, which are adjacent to the vertices in ***Ti*** can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.
- To identify the **i^{th}** nearest vertex, the algorithm computes, for every fringe vertex u, the shortest distance from the adjacent vertices in ***Ti*** and then selects the vertex with the smallest such distances. (This selection is the **greedy** step!)
- Why is this **locally optimal** choice is part of a **globally optimal** solution?

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 priority queue to empty
for every vertex v in V
    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)
```

Dijkstra's Algorithm:

- After we have identified a vertex **u^*** to be added to the tree, we need to perform two operations:
  - Move **u^*** from the fringe to the set of tree vertices.
  - For each remaining fringe vertex **u** that is connected to **u^*** by an edge of weight **$w(u^*, u)$** such that **$d_{u^*} + w(u^*,u) < d_u$**, update the labels of **u** by **u^*** and **$d_{u^*} + w(u^*, u)$**, respectively.

Dijkstra's Algorithm:

Time Complexity:

- $O(n^2)$
  - where n is the number of vertices
  - Graph is represented as **weight matrix**
  - **Priority Queue** is implemented as an **unordered array**

- O(m log n)
  - where m and n are the number of edges and vertices, respectively.
  - Graph is represented as **adjacency lists**
  - **Priority Queue** is implemented as a **min-heap**

Huffman Trees and Codes:

- **Codeword**: Encoding a text that comprises n characters from some alphabet by assigning to each of the text's characters some sequence of bits. This bit-sequence is called codeword.
- **Fixed length encoding**: Assigns to each character a bit string of the same length.
- **Variable length encoding**: Assigns codewords of different lengths to different characters.
- **Prefix free code**: In Prefix free code, no codeword is a prefix of a codeword of another character.

Huffman Trees and Codes:

Binary prefix code :

- The characters are associated with the leaves of a binary tree.
- All left edges are labeled as 0, and right edges as 1.
- Codeword of a character is obtained by recording the labels on the simple path from the root to the character’s leaf.
- Since, there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword.

About **Huffman Algorithm**:

- Invented by David A Huffman in 1951.
- Constructs binary prefix code tree.
- Huffman's algorithm achieves data compression by finding the best variable length binary encoding scheme for the symbols that occur in the file to be compressed. Huffman coding uses frequencies of the symbols in the string to build a variable rate prefix code
  - Each symbol is mapped to a binary string
  - More frequent symbols have shorter codes
  - No code is a prefix of another code (prefix free code)
- Huffman Codes for data compression achieves 20-90% Compression.

Huffman Algorithm:

Input: Alphabet and frequency of each symbol in the text.

Step 1: Initialize n one-node trees (forest) and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's weight. (Generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2: Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

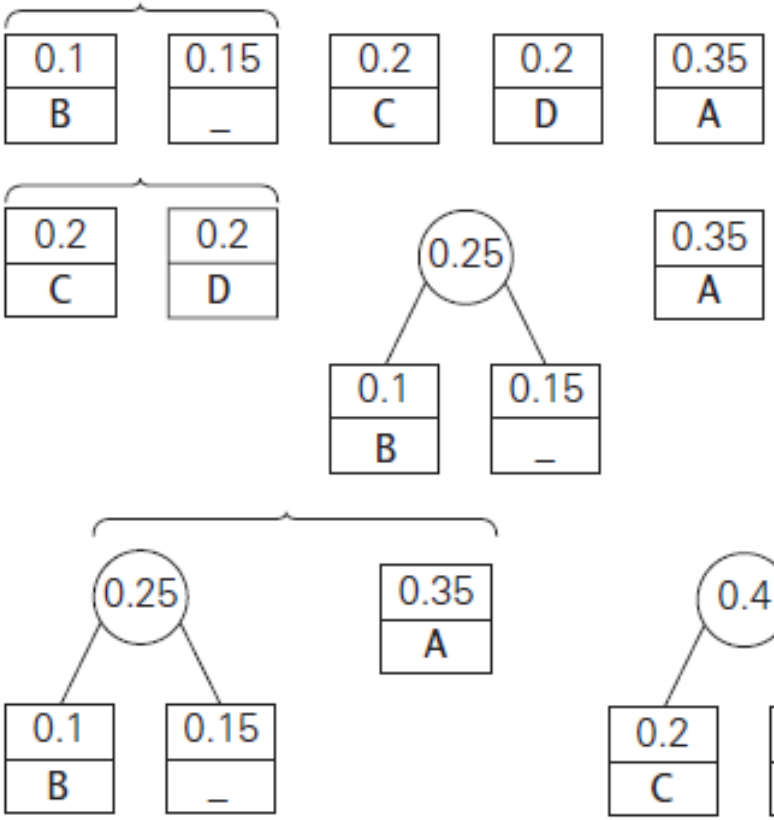
Huffman Algorithm:

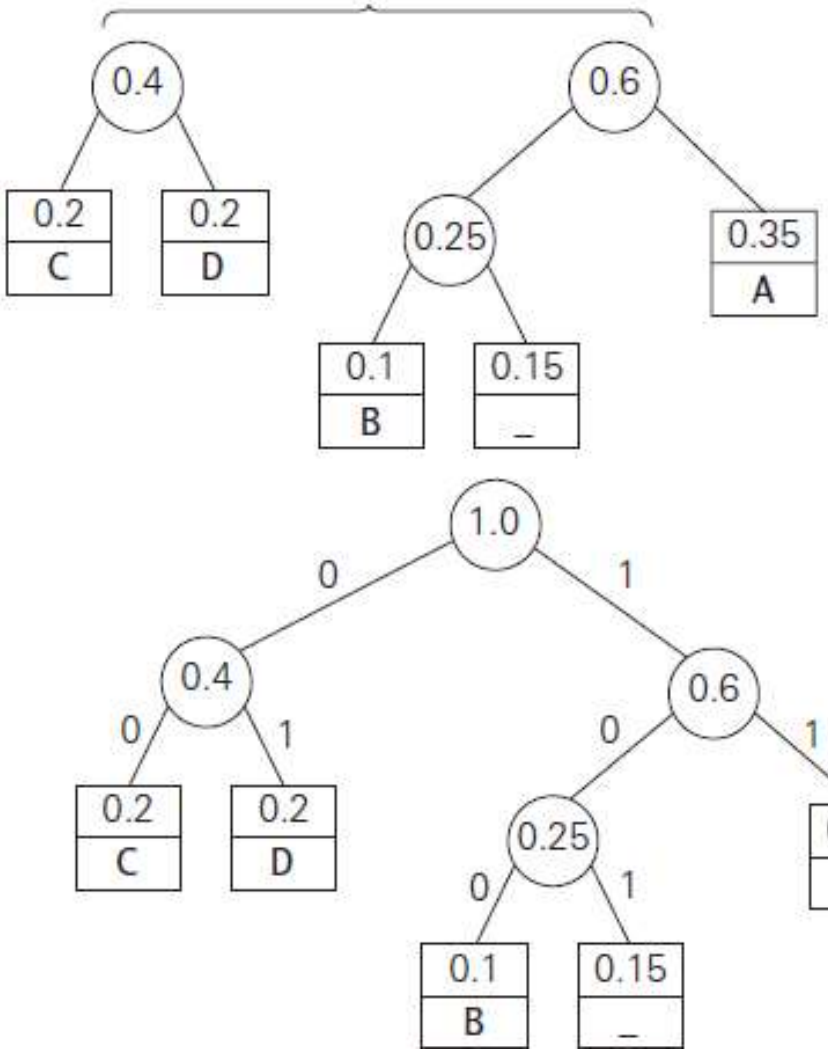
EXAMPLE

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

Huffman Algorithm:

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





Huffman Algorithm:

EXAMPLE

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

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

Huffman Algorithm: EXAMPLE

Character	A	B	C	D	-
probability	0.4	0.1	0.2	0.15	0.15

Character	A	B	C	D	-
probability	0.4	0.1	0.2	0.15	0.15
Code word	0	100	111	101	110

Compute compression ratio:

Bits per character = Codeword length * Frequency

= (1 * 0.4) + (3 * 0.1) + (3 * 0.2) + (3 * 0.15) + (3 * 0.15)

= 2.20

Compression ratio is = (3 – 2.20)/ 3 . 100% = 26.6%

Huffman Coding:

Optimal Encoding: When the frequencies of symbol occurrences are independent and known in advance, the Huffman Algorithm yields an optimal, i.e., minimal-length, encoding.

Drawback: It is necessary to include the coding table into the encoded text to make its decoding possible.

Lempel-Ziv algorithm: assign codewords not to individual symbols but to strings of symbols, allowing them to achieve better and more robust compressions in many applications.

Q: For the given alphabet and their frequencies in a text, write

- Huffman tree
- Huffman codes
- Encode the text: A_CAB_AB_AD
- Decode the code string: 10001011111010100100

Alphabet:	A	B	C	D	_
Frequencies:	45	17	2	1	35

Q: For the given alphabet and their frequencies in a text, write

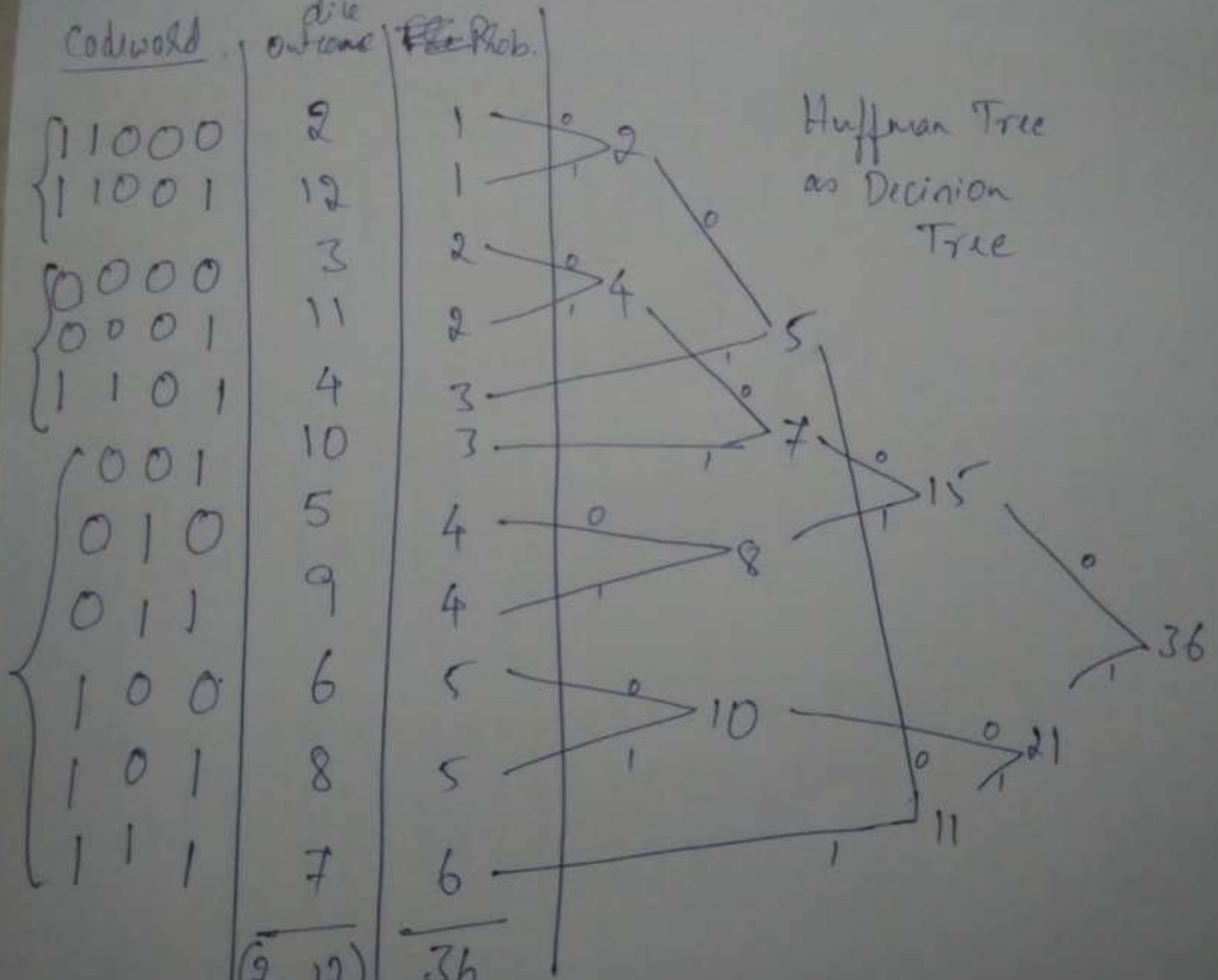
- Huffman tree
- Huffman codes
- Encode the text: A_CAB_AB_AD
- Decode the code string: 10001011111010010 0

Alphabet:	A	B	C	D	_
Frequencies:	45	17	2	1	35
Codes:	0	100	1010	1011	11

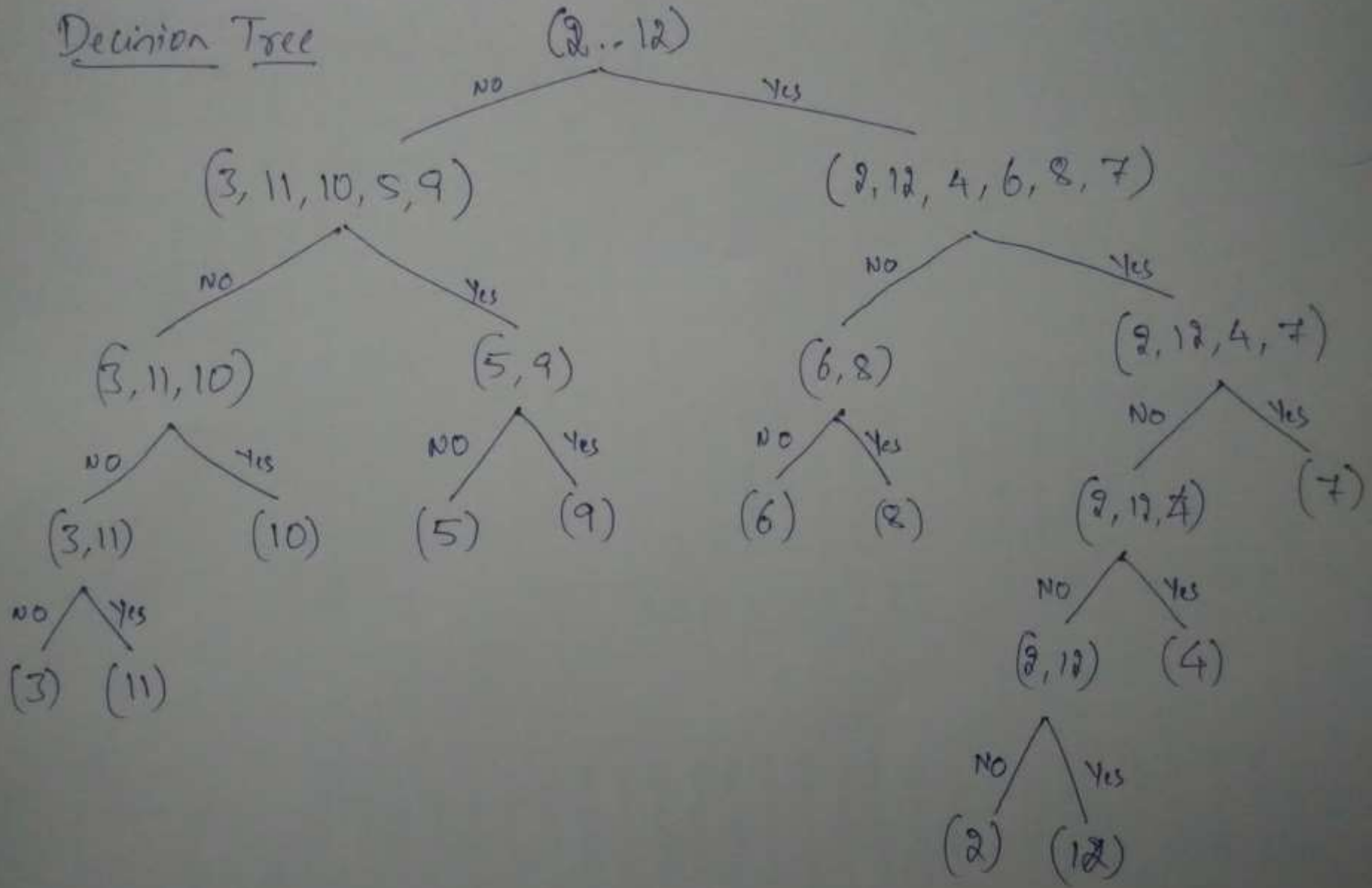
A_CAB_AB_AD: 0111010010011010011010 11

10001011111010010 0: BAD_CAB

Codeword	die outcome	Prob.
11000	2	1
11001	12	1
0000	3	2
00001	11	2
11011	4	3
001	10	3
0010	5	4
0011	9	4
100	6	5
101	8	5
111	7	6
	(2..12)	36



Decision Tree



Greed is good when it’s a good greed!

</ End of Greedy Technique >