Greedy Technique

Prepared by
Dr. Rashmi S

Greedy Technique

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be:

Greedy Technique ..contd

- **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 decision was made, it cannot be changed on subsequent steps of the algorithm

Principle of Greedy Technique

Principle: A sequence of locally optimal choices will yield a globally optimal solution to the entire problem

However, a sequence of locally optimal choices does not always yield a globally optimal solution

Minimum Spanning Tree (MST) Problem

- Given n points, connect them in the cheapest possible way so that there will be a path between every pair of points.
- The solution is called the minimum spanning tree (MST).
- The MST is a tree connecting all points and has the minimum cost (also called length, or weight).

Minimum Spanning Tree (MST) Problem

- The minimum spanning tree problem is the problem of finding a minimum spanning tree (MST) for a given weighted connected undirected graph.
- The total number of possible spanning trees for n vertices (nodes, points, or cities) is n n-2 (it grows exponentially).
- Thus, it is impossible to employ the brute-force strategy to solve the MST problem. It is solved by the greedy method.

Example

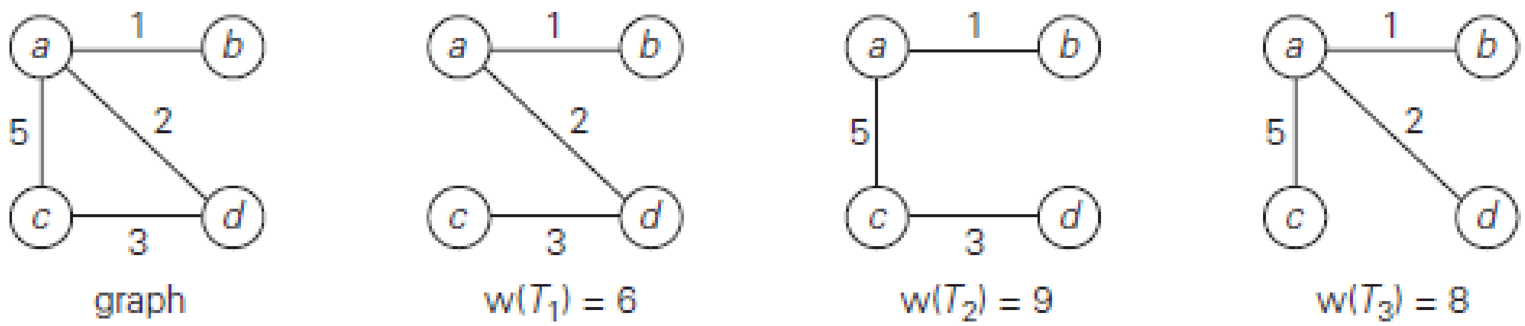


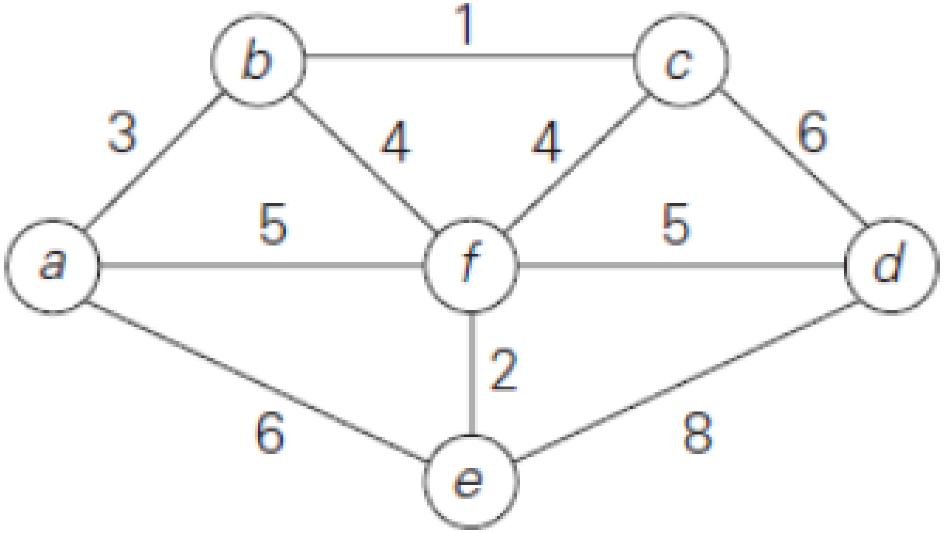
FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

Algorithms for obtaining MST

1) Prim's Algorithm
2) Kruskal's Algorithm

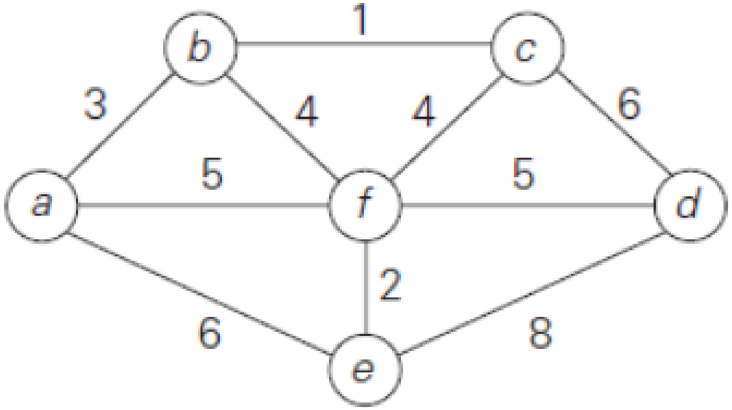
Prim’s Algorithm

• Consider the following weighted, connected, undirected graph

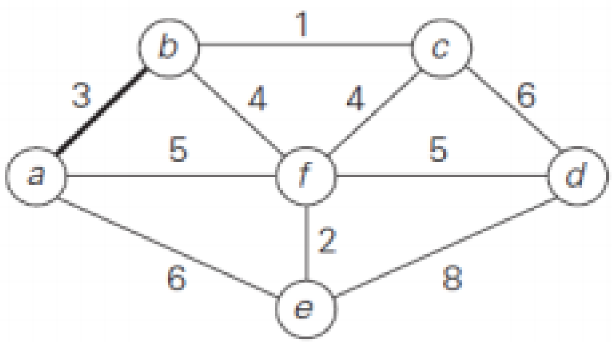
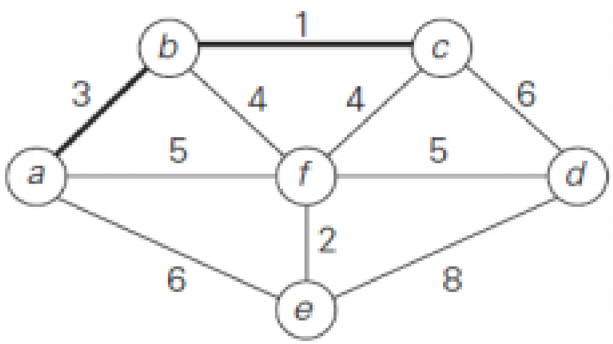


Prim's Algorithm

- Attach two labels to a vertex:
  - the name of the nearest tree vertex, and
  - the weight of the corresponding edge.
- Example: select a as starting vertex
  - a(-, -)
  - b(a, 3)
  - c(-, ∞)

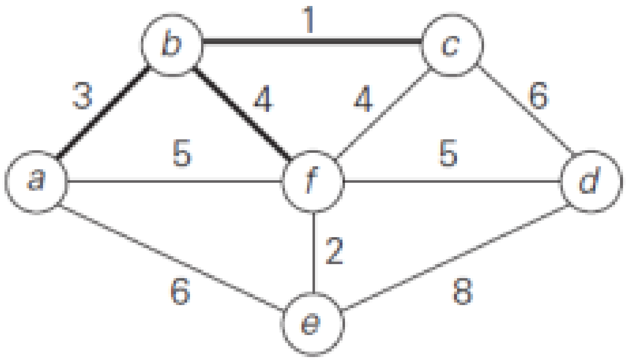


Prim’s Algorithm

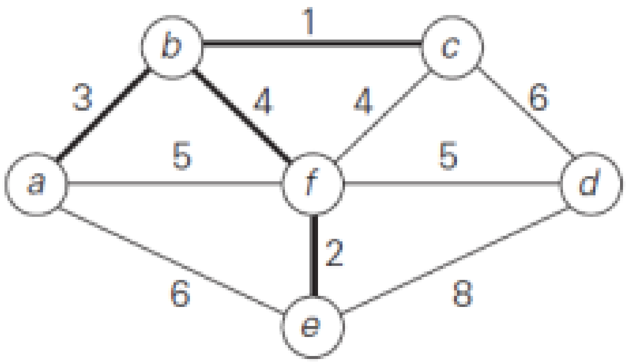
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	

Prim’s Algorithm

c(b, 1) d(c, 6) e(a, 6) **f(b, 4)**



f(b, 4) d(f, 5) **e(f, 2)**

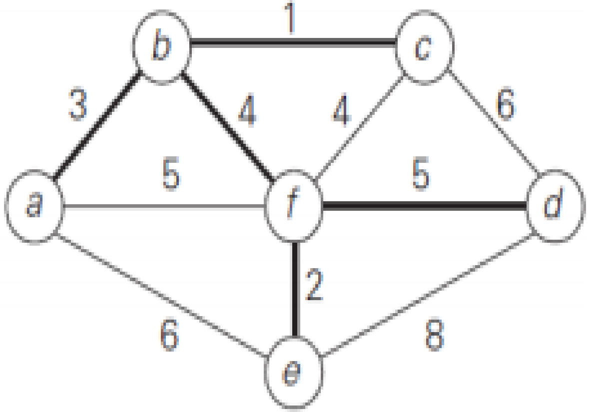


Prim's Algorithm

l

.

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

Time Complexity

- In particular, if a graph is represented by its weight matrix and the priority queue is implemented as an unordered array, the algorithm's running time will be in $\Theta(|V|^2)$

- If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $O(|E| \log |V|)$.

Thank You

Dijkstra’s Algorithm
for Single Source Shortest Path Problem
using Greedy Technique

Prepared by
Dr. Rashmi S

Single Source Shortest Path Problem

Single Source Shortest-Paths Problem:

for a given vertex called the *source in a weighted connected graph, find shortest paths to all* its other vertices.

- Best-known algorithm for the single-source shortest-paths problem, called *Dijkstra's algorithm*
- *This algorithm is applicable* to undirected and directed graphs with nonnegative weights only

Dijkstra’s algorithm

- Dijkstra’s algorithm finds the shortest paths to a graph’s vertices in order of their distance from a given source.
- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.
- In general, before its ith iteration commences, the algorithm has already identified the shortest paths to i – 1 other vertices nearest to the source.
- These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree Ti of the given graph

Dijkstra's algorithm

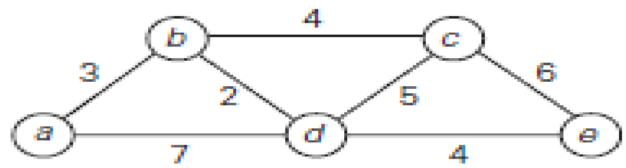
Among vertices not already in the tree, it finds vertex u with the smallest <u>sum</u>

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

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

Example



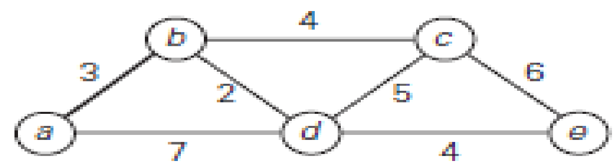
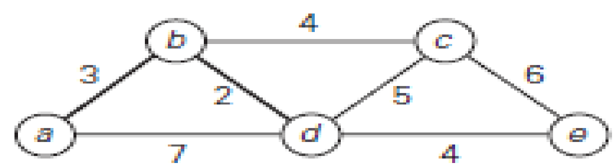
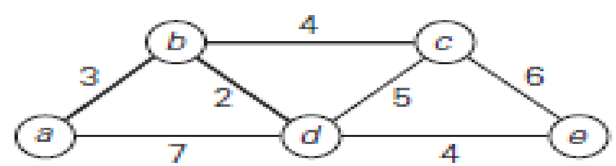
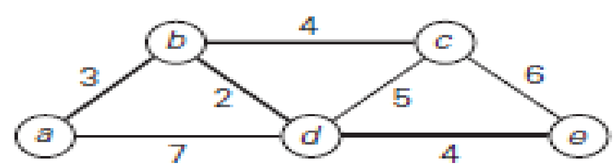
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

FIGURE 9.11 Application of Dijkstra's algorithm. The next closest vertex is shown in bold.

Dijkstra’s algorithm

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

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

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)
```

Time Complexity

- The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself
- It is $(\Theta|V|^2)$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array.
- For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in $O(|E| \log |V|)$.

Dijkstra's Algorithm

- Doesn't work for graphs with negative weights
- Applicable to both undirected and directed graphs

- Thank You

Huffman Trees and Codes

Prepared by
Dr. Rashmi S

Huffman Trees and codes

- A Huffman tree is a binary tree that minimizes the weighted path length from the root to the leaves of predefined weights.
- The most important application of Huffman trees is Huffman codes.
- A Huffman code is an optimal prefix-free variable-length encoding scheme that assigns bit strings to symbols based on their frequencies in a given text.
- This is accomplished by a greedy construction of a binary tree whose leaves represent the alphabet symbols and whose edges are labeled with 0's and 1's.

Huffman Trees and codes

- Huffman’s encoding is one of the most important file-compression methods.
- In addition to its simplicity and versatility, it yields an optimal, i.e., minimal-length, encoding

Encoding Methods

- Fixed Length Encoding
- Variable Length Encoding

Fixed –Length encoding

- Suppose we have to encode a text that comprises symbols from some n-symbol alphabet by assigning to each of the text's symbols some sequence of bits called the codeword.
- For example, we can use a fixed-length encoding that assigns to each symbol a bit string of the same length m
  ($m \geq \log2\ n$).
- This is exactly what the standard ASCII code does

Fixed Length Codes

Represent data as a sequence of 0's and 1's

Sequence: BACADAEAFABBAAAGAH

A fixed length code:

A 000 B 001 C 010 D 011

E 100 F 101 G 110 H 111

Encoding of sequence:

001000010000011000100000101000001001000000000110000111

The Encoding is 18x3=54 bits long.
Can we make the encoding shorter?

Variable-length encoding

- ***Variable-length encoding, which assigns codewords of different lengths to*** different symbols
- In a prefix code, no codeword is a prefix of a codeword of another symbol.
- Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some symbol, replace these bits by this symbol, and repeat this operation until the bit string's end is reached.

Variable Length Code-Huffman

- Make use of frequencies. Frequency of A=8, B=3, others 1
- A 0 B 100 C 1010 D 1011
- E 1100 F 1101 G 1110 H 1111
- Example: BACADAEAFABBAAAGH
- $\underbrace{\text{100010100101101100011010100100000111001111}}_{\text{42 bits (20\% shorter)}}$

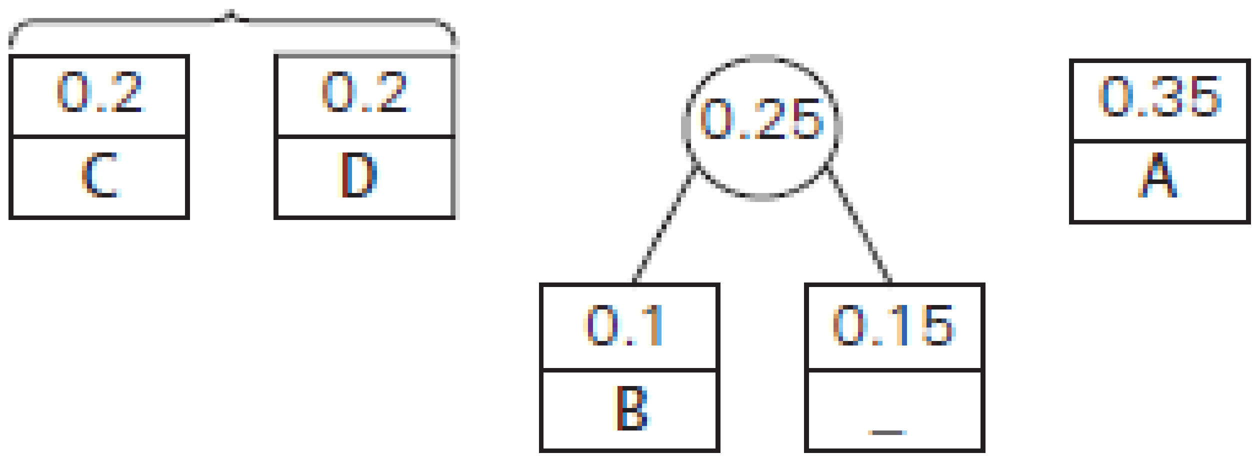
Huffman's algorithm

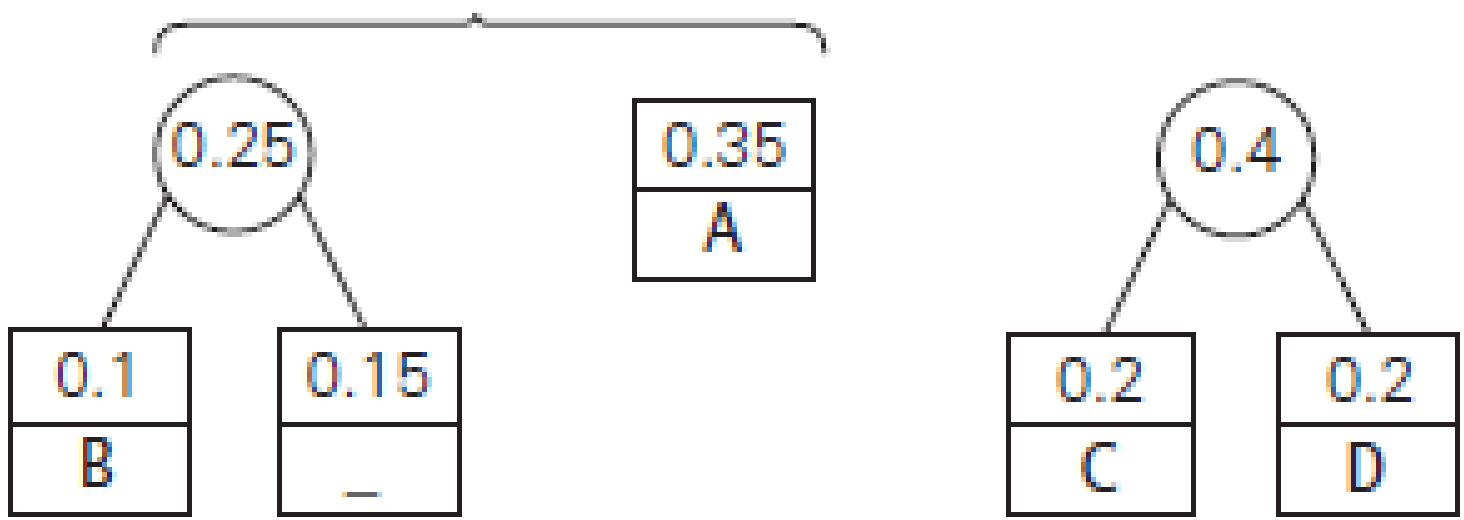
- **Step 1**: Initialize *n one-node trees and label them with the symbols of the* alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's *weight*
- **Step 2**: Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight .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**.**

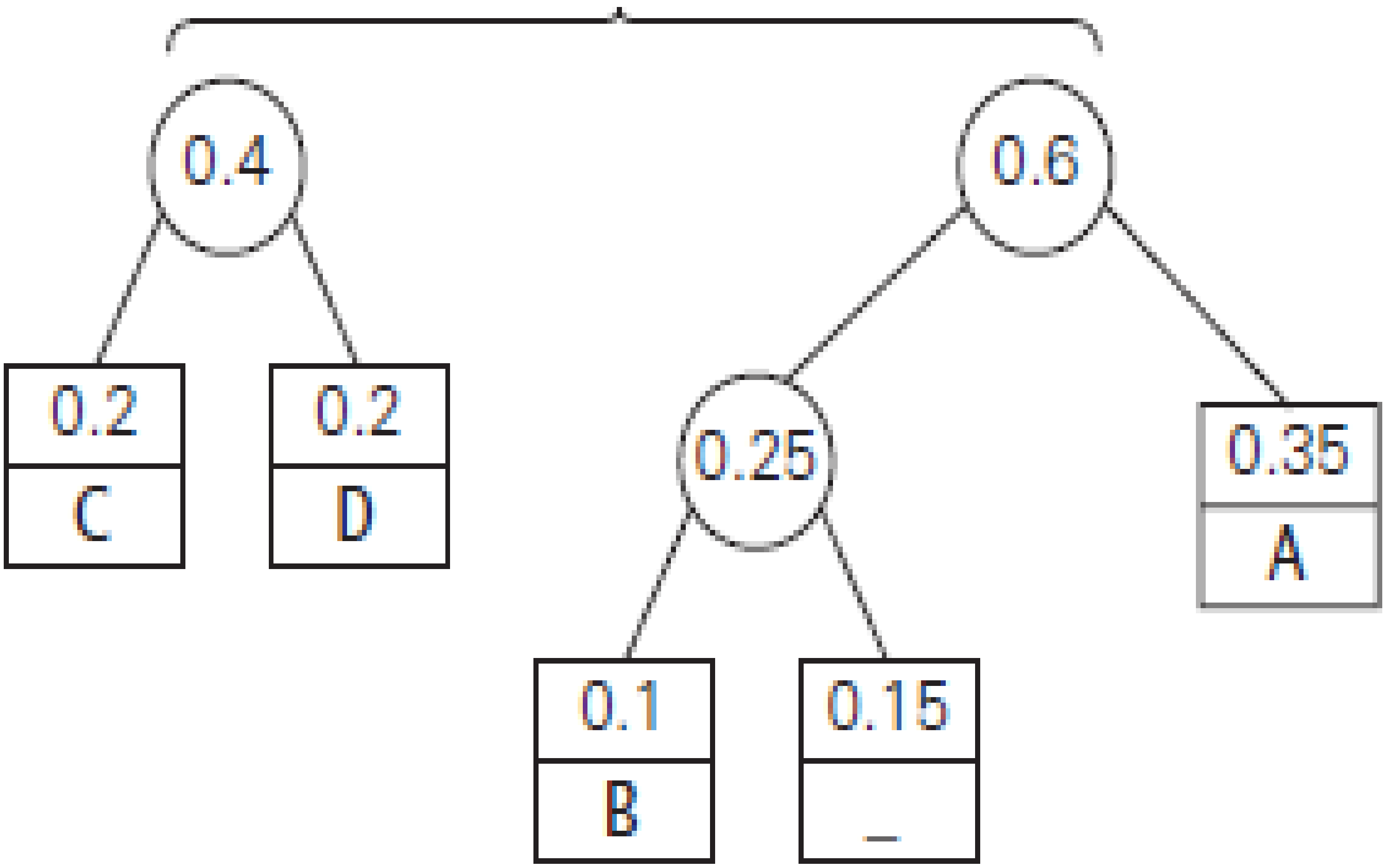
A tree constructed by the above algorithm is called a *Huffman tree. It defines a Huffman code*

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

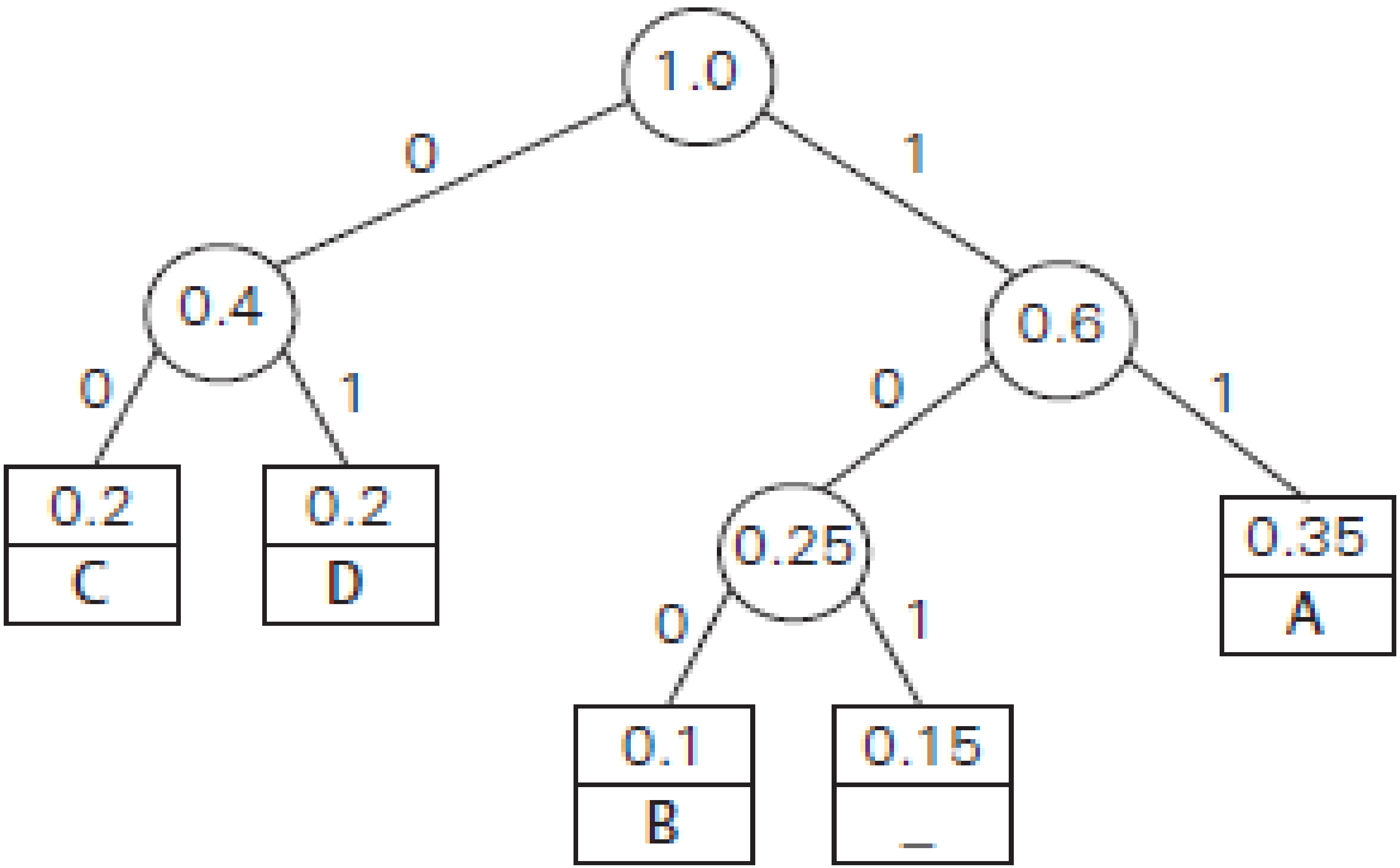
symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15


0.2
C
0.2
D
0.25
0.1
B
0.15
_
0.35
A


0.25
0.1
B
0.15
_
0.35
A
0.4
0.2
C
0.2
D


0.4
0.6
0.2
C
0.2
D
0.25
0.35
A
0.1
B
0.15
_

Example of constructing a Huffman coding tree



Example of constructing a Huffman coding tree

The resulting codewords are as follows:

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

- DAD is encoded as 011101
- 10011011011101 is decoded as BAD_AD

Cost of Tree Corresponding to Prefix Code

- Given a tree T corresponding to a prefix code. For each character c in the alphabet C,
  - let $f(c)$ denote the frequency of c in the file and
  - let $d_T(c)$ denote the depth of c's leaf in the tree.
  - $d_T(c)$ is also the length of the codeword for character c.
  - The number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

  - which we define as the ***cost*** of the tree T.

Compression Ratio

- With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25.$$

- Had we used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol

- Huffman codes have shown that the compression ratio for this scheme typically falls between 20% and 80%, depending on the characteristics of the text being compressed (generally about 25%)

Thank You