Dijkstras Algorithm

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)

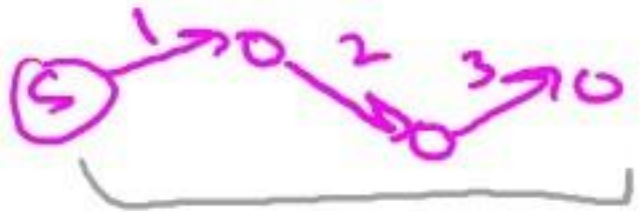
d_v is the length of the shortest path form source to v $w(v,u)$ is the length (weight) of edge from v to u

Single--Source Shortest paths

Input: directed graph G=(V, E). (m=|E|, n=|V|)

- each edge has non negative length l_e
- source vertex s

Length of path
= sum of edge lengths



Path length = 6

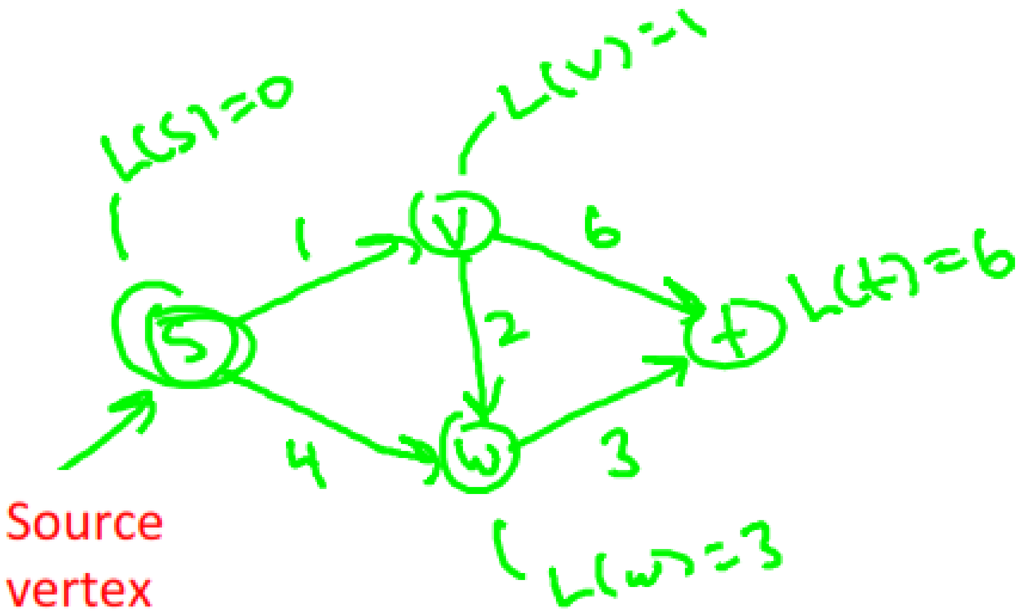
Output: for each $v \in V$, compute

L(v) := length of a shortest s-v path in G

Assumption:

1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $l_e \geq 0 \ \forall e \in E$

Example



Dijkstra's Algorithm

This array only to help explanation!

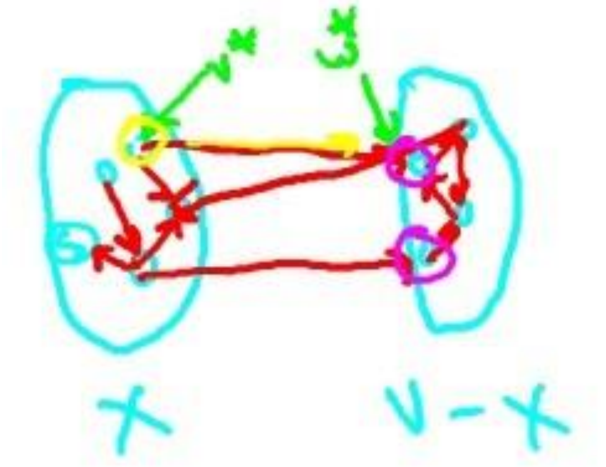
<u>Ini*alize:</u>

- X = [s] [ver*ces processed so far]
- A[s] = 0 [computed shortest path distances]
- B[s] = empty path [computed shortest paths]

<u>Main Loop</u>

- while X≠V:

-need to grow x by one node



<u>Main Loop cont'd:</u>

- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$

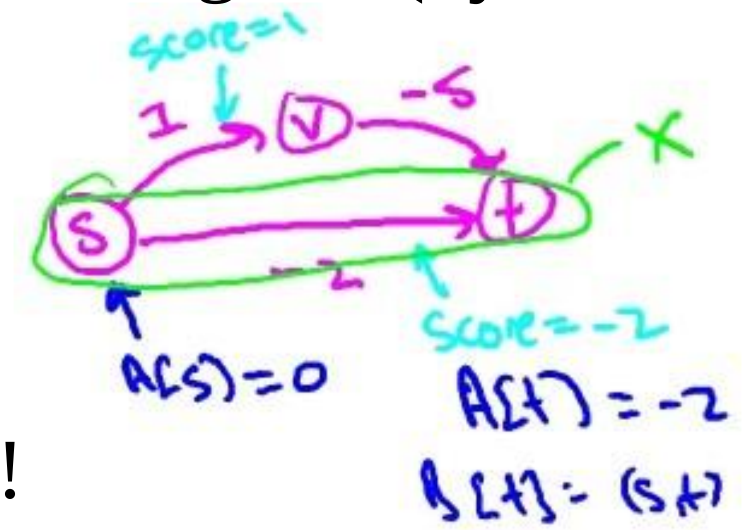
[call it (v*, w*)]

Already computed in earlier iteration

- add w* to X
- set $A[w^*] := A[v^*] + l_{v^*w^*}$
- set $B[w^*] := B[v^*]u(v^*, w^*)$

Non-Example

Question: why not reduce computing shortest paths with negative edge lengths to the same problem with non negative lengths? (by adding large constant to edge lengths)



Problem: doesn't preserve shortest paths !

Also: Dijkstra's algorithm incorrect on this graph !
(computes shortest s-t distance to be -2 rather than -4)

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)
```

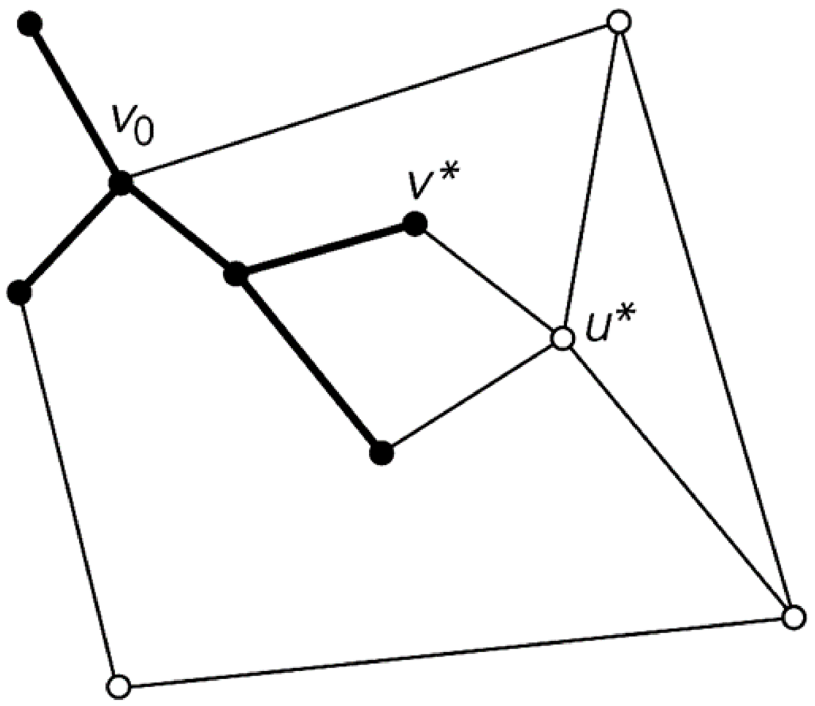
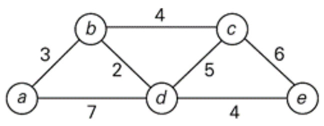
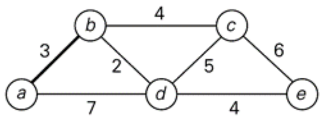
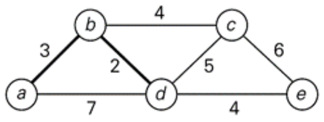
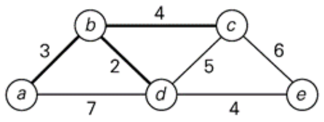
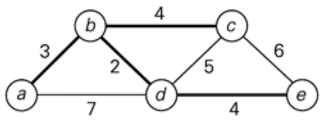


FIGURE 9.9 Idea of Dijkstra's algorithm. The subtree of the shortest paths already found is shown in bold. The next nearest to the source v_0 vertex, u^*, is selected by comparing the lengths of the subtree's paths increased by the distances to vertices adjacent to the subtree's vertices.



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)		

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

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

FIGURE 9.10 Application of Dijkstra's algorithm. The next closest vertex is shown in bold.

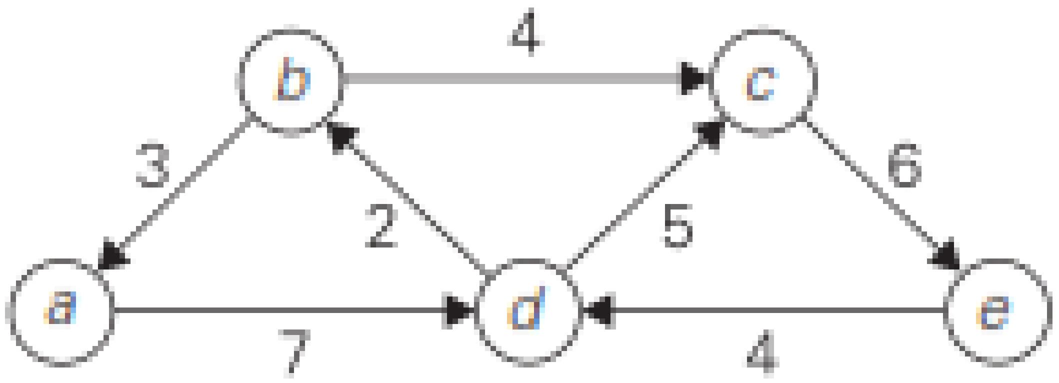
Efficiency

- 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

Example



Solution

Tree vertices	Remaining vertices
a(-,0)	b(-,∞) c(−,∞) **d(a,7)** e(-,∞)
d(a,7)	**b(d,7+2)** c(d,7+5) e(-,∞)
b(d,9)	**c(d,12)** e(-,∞)
c(d,12)	**e(c,12+6)**
e(c,18)	

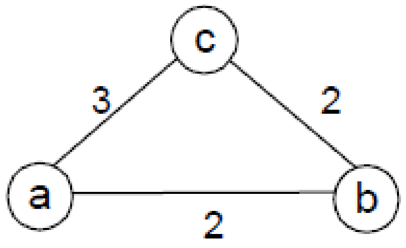
The shortest paths (identified by following nonnumeric labels backwards from a destination vertex to the source) and their lengths are:

from a to d:	$a-d$	of length 7
from a to b:	$a-d-b$	of length 9
from a to c:	$a-d-c$	of length 12
from a to e:	$a-d-c-e$	of length 18

. Let T be a tree constructed by Dijkstra's algorithm in the process of solving the single-source shortest-paths problem for a weighted connected graph G.

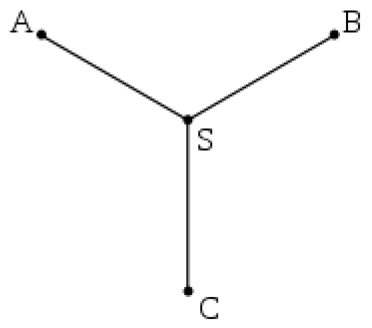
a. True or false: T is a spanning tree of G?

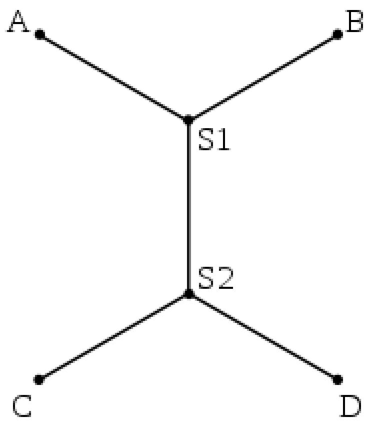
b. True or false: T is a minimum spanning tree of G?


c
3
2
a
b
2

Steiner Tree

- minimum-weight connected subgraph of G that includes all the vertices. It is always a tree. Steiner trees have practical applications, for example, in the determination of the shortest total length of wires needed to join some number of points


A
B
S
C


A
B
S1
S2
C
D

Huffman Code

- For example, we can use a ***fixed-length encoding that assigns to*** each symbol a bit string of the same length m (m = log n). This is exactly what the standard ASCII code does.
- One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter codewords to more frequent symbols and longer codewords to less frequent symbols.
- This idea was used, in particular, in the telegraph code invented in the mid-19th century by Samuel Morse. In that code, frequent letters such as e(.) are assigned short sequences of dots and dashes while infrequent letters such as q(--.-) and

  z(--..) have longer ones

- Variable-length encoding, which assigns codewords of different lengths to different symbols, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the ith) symbol? To avoid this complication, we can limit ourselves to the so-called prefix-free (or simply prefix) codes. In a prefix code, no codeword is a prefix of a codeword of another symbol.

- David Huffman MIT student

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***. (More 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 (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). 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—in the manner described above—a ***Huffman code***.

Example

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

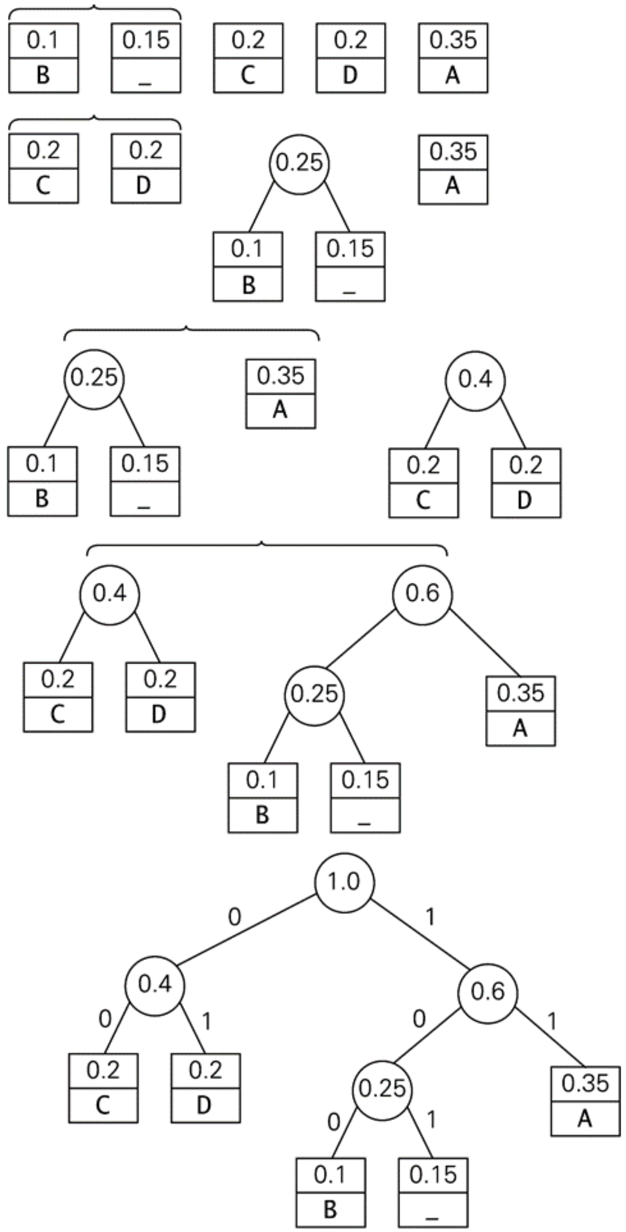
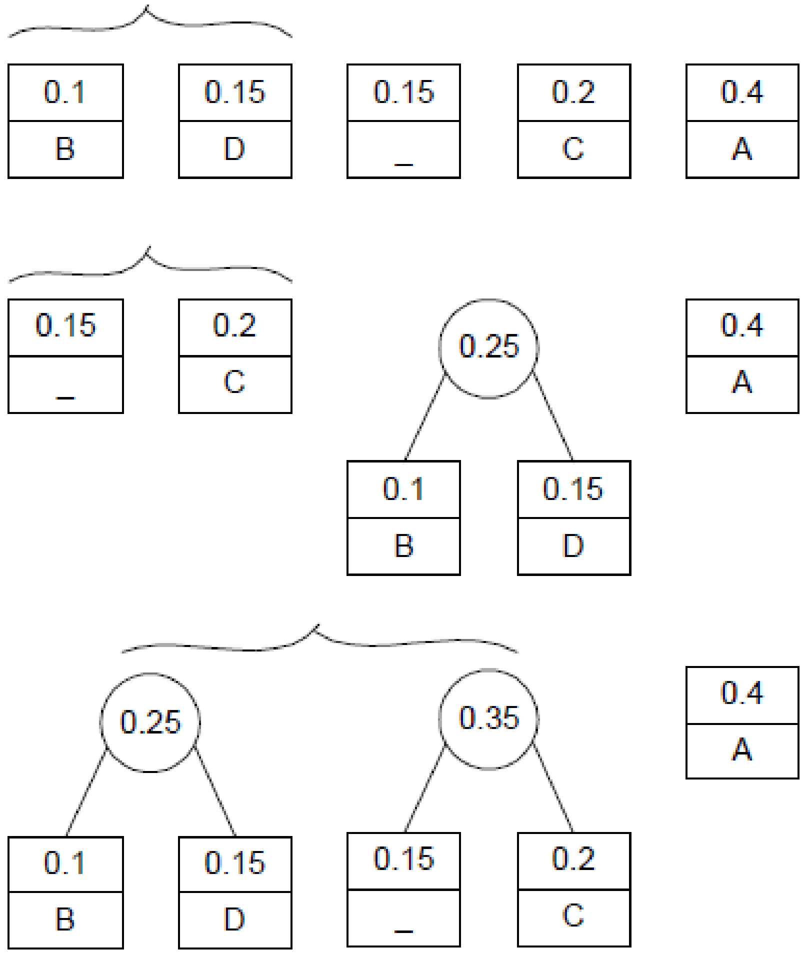


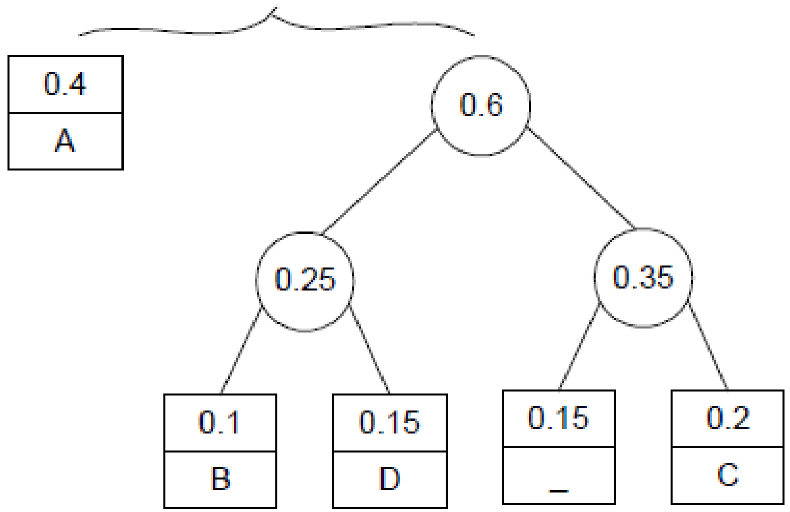
FIGURE 9.11 Example of constructing a Huffman coding tree

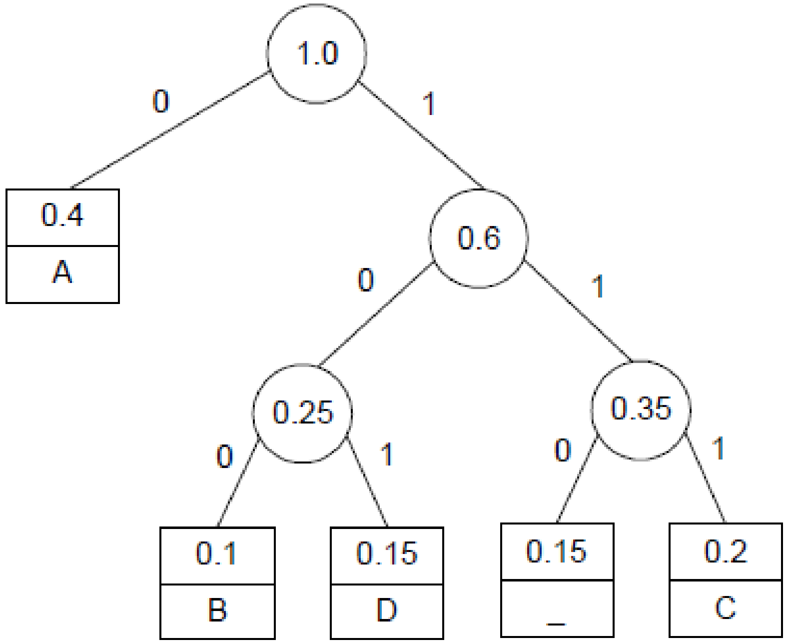
- Encode DAD
- 10011011011101 Decode this string

symbol	A	B	C	D	_
frequency	0.4	0.1	0.2	0.15	0.15

- ABACABAD encode
- 100010111001010

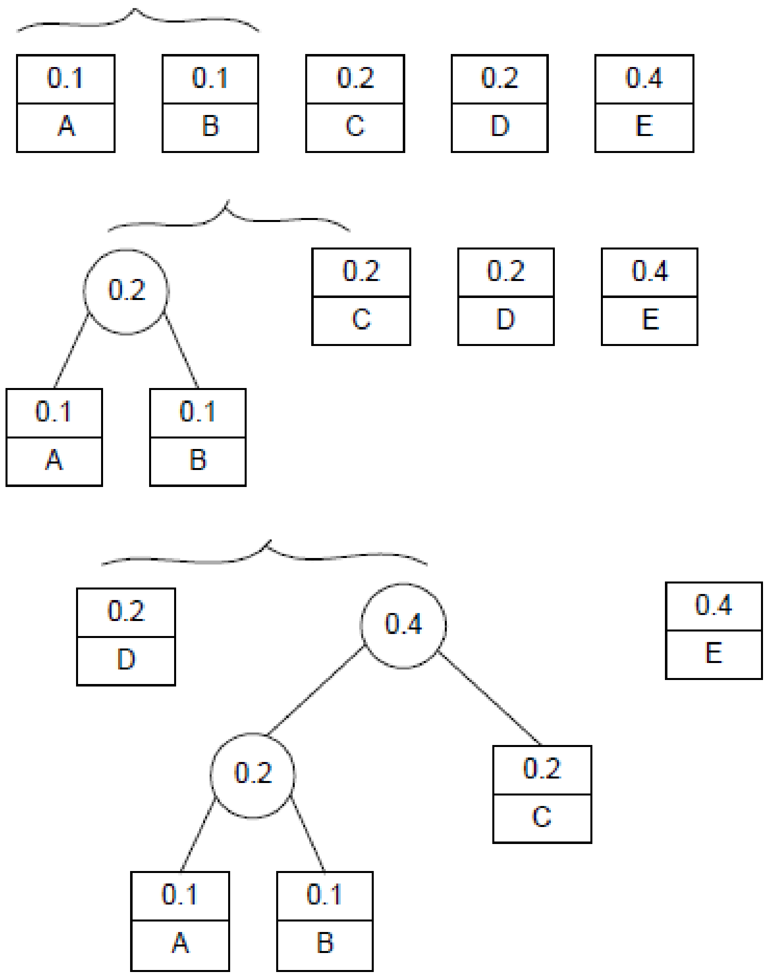

0.1
B
0.15
D
0.15
_
0.2
C
0.4
A
0.15
_
0.2
C
0.25
0.1
B
0.15
D
0.4
A
0.25
0.35
0.1
B
0.15
D
0.15
_
0.2
C
0.4
A

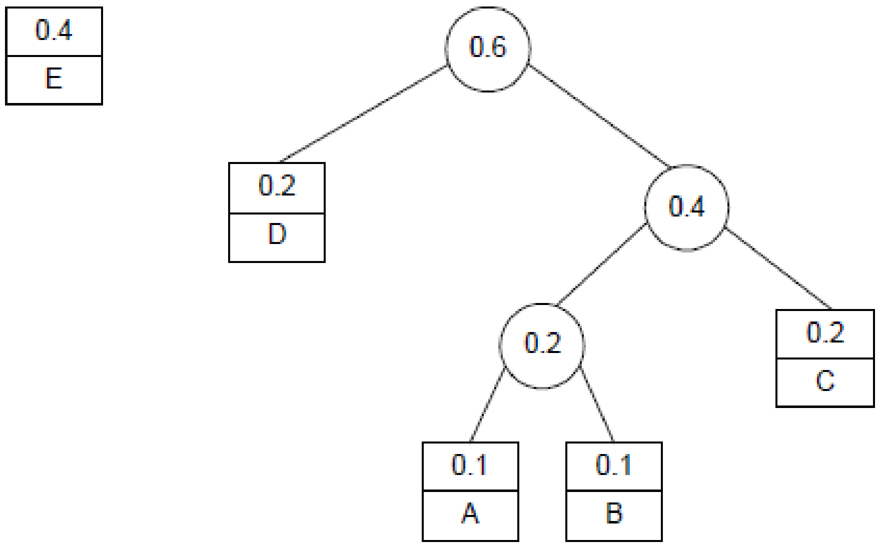

0.4
A
0.6
0.25
0.35
0.1
B
0.15
D
0.15
_
0.2
C

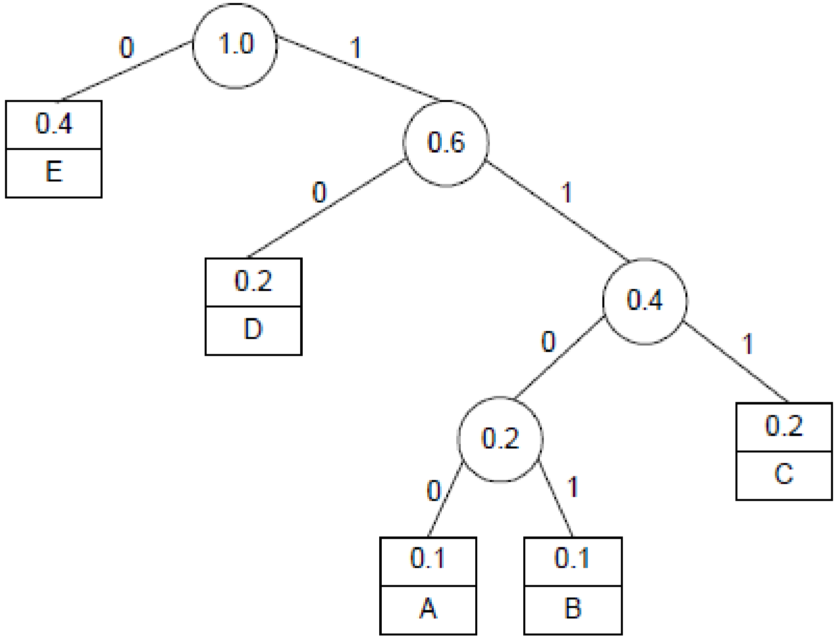

1.0
0
1
0.4
A
0.6
0
1
0.25
0.35
0
1
0
1
0.1
B
0.15
D
0.15
_
0.2
C

Example

symbol	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4


0.1
A
0.1
B
0.2
C
0.2
D
0.4
E
0.2
0.1
A
0.1
B
0.2
C
0.2
D
0.4
E
0.2
D
0.4
0.2
0.1
A
0.1
B
0.2
C
0.4
E


0.4
E
0.6
0.2
D
0.4
0.2
0.2
C
0.1
A
0.1
B


1.0
0
1
0.4
E
0.6
0
1
0.2
D
0.4
0
1
0.2
0.2
C
0
1
0.1
A
0.1
B

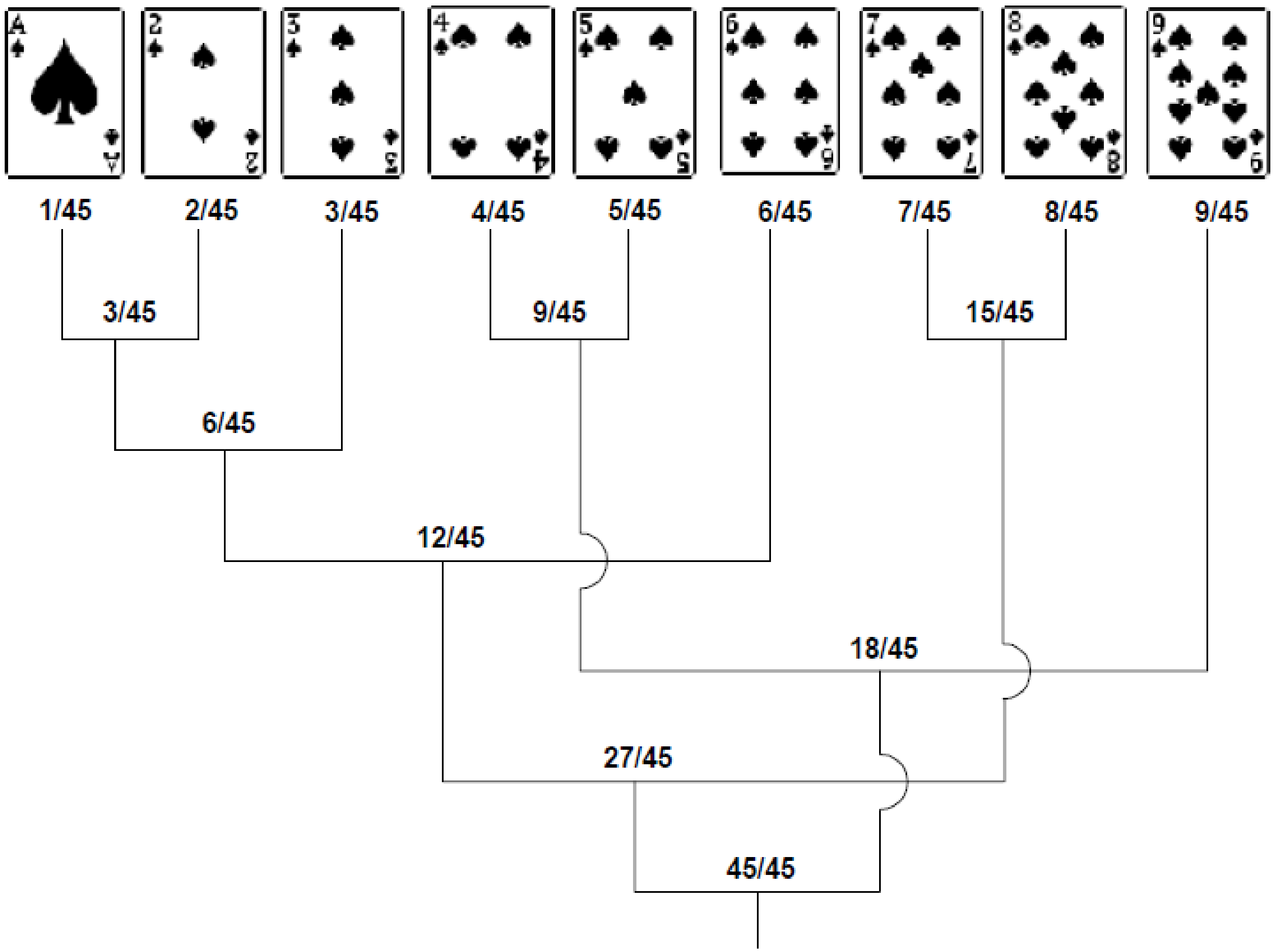
A *Huffman code* is an optimal prefix-free variable-length encoding scheme that assigns bit strings to symbols based on their frequencies in a given text. This is accomplished by a greedy construction of a binary tree whose leaves represent the alphabet symbols and whose edges are labeled with 0's and 1's.

Example

10. *Card guessing* Design a strategy that minimizes the expected number of questions asked in the following game [Gar94]. You have a deck of cards that consists of one ace of spades, two deuces of spades, three threes, and on up to nine nines, making 45 cards in all. Someone draws a card from the shuffled deck, which you have to identify by asking questions answerable with yes or no.

card	ace	deuce	three	four	five	six	seven	eight	nine
probability	1/45	2/45	3/45	4/45	5/45	6/45	7/45	8/45	9/45

Huffman's tree for this data looks as follows:



$$\bar{l} = \sum_{i=1}^{9} l_i p_i = \frac{5 \cdot 1}{45} + \frac{5 \cdot 2}{45} + \frac{4 \cdot 3}{45} + \frac{3 \cdot 5}{45} + \frac{3 \cdot 6}{45} + \frac{3 \cdot 7}{45} + \frac{3 \cdot 8}{45} + \frac{2 \cdot 9}{45} = \frac{135}{45} = 3.$$