

Dynamic Programming

Steps in dynamic programming

- ▶ Every problem is divided into stages
- ▶ Each stage requires a decision
- ▶ Decisions are made to determine the state of the next stage
- ▶ The solution procedure is to find an optimal solution at each stage for every possible state
- ▶ This solution procedure often starts at the last stage and works its way forward

0/1 Knapsack Problem

- We are given a knapsack of capacity c and a set of n objects numbered $1, 2, \dots, n$. Each object i has weight w_i and profit p_i .
- Let $v = [v_1, v_2, \dots, v_n]$ be a solution vector in which $v_i = 0$ if object i is not in the knapsack, and $v_i = 1$ if it is in the knapsack.
- The goal is to find a subset of objects to put into the knapsack so that

$$\sum_{i=1}^n w_i v_i \leq c$$

(that is, the objects fit into the knapsack) and

$$\sum_{i=1}^n p_i v_i$$

is maximized (that is, the profit is maximized).

- Let us consider an instance defined by the first i items, $1 \leq i \leq n$, with weights w_1, \dots, w_i , values v_1, \dots, v_i , and knapsack capacity j , $1 \leq j \leq W$.
- Let $F(i, j)$ be the **value of an optimal solution to this instance**, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j
- We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the i th item and those that do.

1. Among the subsets that do not include the i th item, the value of an optimal subset is, by definition, $F(i - 1, j)$.
2. Among the subsets that do include the i th item (hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$.

Recurrence Relation

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

Let us consider the instance given by the following data:

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

		capacity j						
		i	0	1	2	3	4	5
	0		0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1		0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2		0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3		0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4		0	10	15	25	30	37

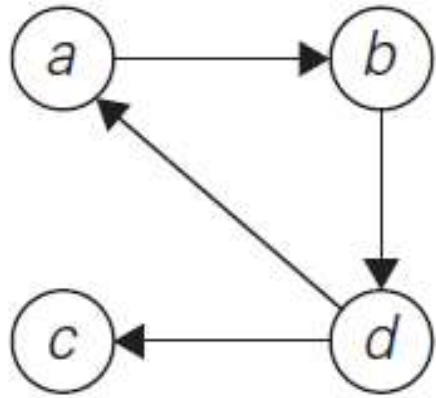
Example of solving an instance of the knapsack problem by the dynamic programming algorithm.

Algorithm

```
for i  $\leftarrow$  0 to n do
  for j  $\leftarrow$  0 to m do
    if (i=0 OR j=0)
       $V[i,j] \leftarrow 0$ 
    else
      if ( $W[i] > j$ )
         $V[i,j] \leftarrow V[i-1,j]$ 
      else
         $V[i,j] \leftarrow \max(V[i-1,j], V[i-1,j-w_i] + p[i])$ 
      endif
    endif
  end for
end for
```

Warshall's Algorithm

- Warshall's algorithm for **computing the transitive closure** of a directed graph
- The transitive closure of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the i th row and the j th column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the i th vertex to the j th vertex; otherwise, t_{ij} is 0.



(a)

$$A = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

(b)

$$T = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

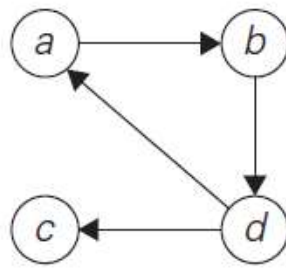
(c)

(a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

$$\begin{array}{c}
 \begin{array}{cc} & j & k \\ & \downarrow & \downarrow \\ R^{(k-1)} = & k & \left[\begin{array}{cc} & \\ & \\ & 1 & \\ & \uparrow & \\ i & 0 & \rightarrow 1 \end{array} \right] \\ & & \end{array}
 \end{array}
 \Rightarrow
 \begin{array}{c}
 \begin{array}{cc} & j & k \\ & \downarrow & \downarrow \\ R^{(k)} = & k & \left[\begin{array}{cc} & \\ & 1 & \\ & 1 & 1 \end{array} \right] \\ & & \end{array}
 \end{array}$$

Rule for changing zeros in Warshall's algorithm.

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix);
boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex a (note a new path from d to b);
boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., a and b (note two new paths);
boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., a , b , and c (no new paths);
boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., a , b , c , and d (note five new paths).

ALGORITHM *Warshall*($A[1..n, 1..n]$)

//Implements Warshall's algorithm for computing the transitive closure

//Input: The adjacency matrix A of a digraph with n vertices

//Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$

for $k \leftarrow 1$ **to** n **do**

for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** n **do**

$R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j] \text{ or } (R^{(k-1)}[i, k] \text{ and } R^{(k-1)}[k, j])$

return $R^{(n)}$

Exercise Problem

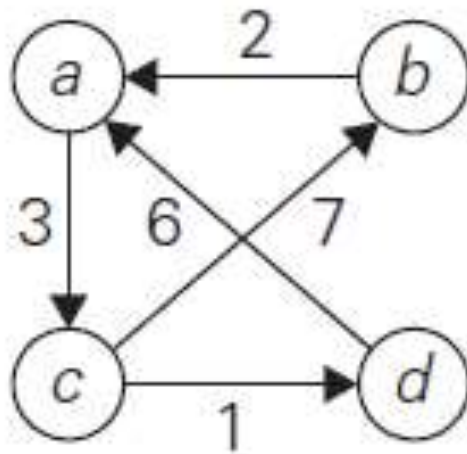
- Apply the Warshall's Algorithm to compute Transitive closure for the adjacency matrix.

	a	b	c	d
A	0	1	0	0
B	0	0	0	1
C	0	0	0	0
d	1	0	1	0

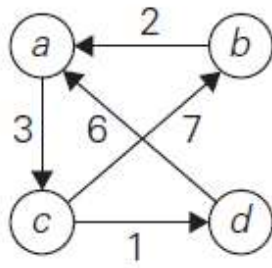
Floyd's Algorithm for the All-Pairs Shortest-Paths

- Given a weighted connected graph (undirected or directed), the ***all-pairs shortestpaths problem*** asks to find the distances—i.e., the lengths of the shortest paths— from each vertex to all other vertices.

- It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element d_{ij} in the i th row and the j th column of this matrix indicates the length of the shortest path from the i th vertex to the j th vertex.



$$D^{(0)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$



$$D^{(0)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{matrix}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just a (note two new shortest paths from b to c and from d to c).

$$D^{(2)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{matrix}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and b (note a new shortest path from c to a).

$$D^{(3)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{matrix}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a , b , and c (note four new shortest paths from a to b , from a to d , from b to d , and from d to b).

$$D^{(4)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{matrix}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a , b , c , and d (note a new shortest path from c to a).

ALGORITHM *Floyd*($W[1..n, 1..n]$)

//Implements Floyd's algorithm for the all-pairs shortest-paths problem

//Input: The weight matrix W of a graph with no negative-length cycle

//Output: The distance matrix of the shortest paths' lengths

$D \leftarrow W$ //is not necessary if W can be overwritten

for $k \leftarrow 1$ **to** n **do**

for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** n **do**

$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$

return D

Exercise Problem

- Write an algorithm to compute shortest distance from all nodes to all other nodes using dynamic programming. Write the Digraph and solve the problem for the cost adjacency matrix.

	a	b	c	d
A	0	∞	3	∞
B	2	0	∞	∞
C	∞	7	0	1

Fractional Knapsack

The fractional knapsack problem is a problem that can be solved using greedy algorithm to maximize the total value of a knapsack. The problem involves breaking items into smaller units or taking them as a whole to fill a knapsack of a given capacity.

The basic idea is to calculate the ratio $\text{profit}/\text{weight}$ for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it) without crossing the capacity of the knapsack.

Fractional Knapsack

Object	1	2	3	4	5	6	7
Profit	10	5	15	7	6	18	3
Weight	2	3	5	7	1	4	1
P/w	5	1.3	3	1	6	4.5	3

Capacity of the Knapsack is 15

Take object with highest Profit/Weight.

5th object

$$15 - 1 = 14$$

$$1^{st} \rightarrow 14 - 2 = 12$$

$$6^{th} \rightarrow 12 - 4 = 8$$

$$3^{rd} \Rightarrow 8 - 5 = 3$$

$$7^{th} \rightarrow 3 - 1 = 2$$

$$2^{nd} \Rightarrow 2 - \frac{2}{3} = 0$$

items selected

$$\begin{pmatrix} 1 & 2/3 & 1 & 0 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{pmatrix}$$

$$\sum x_i w_i = 1 \times 2 + 3 \times \frac{2}{3} + 1 \times 5 + 1 \times 1 + 1 \times 4 + 1 \times 1$$

$$2 + 2 + 5 + 1 + 4 + 1 = 15$$

$$\max \{ \text{Profit } x_i \} = 1 \times 10 + \frac{2}{3} \times 5 + 1 \times 15 + 1 \times 6 + 1 \times 18 + 1 \times 3$$

$$10 + \left(\frac{10}{3} \right) + 15 + 6 + 18 + 3$$

$$= 54 + 3.3 = 57.3$$

Greedy Technique

Greedy Technique

- It is a general design technique despite the fact that it is applicable to optimization problems only.
- 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.

Greedy Technique

On each step—and this is the central point of this technique—the choice made must be:

- *feasible*, i.e., it has to satisfy the problem's constraints
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm

These requirements explain the technique's name: on each step, it suggests a “greedy” grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.

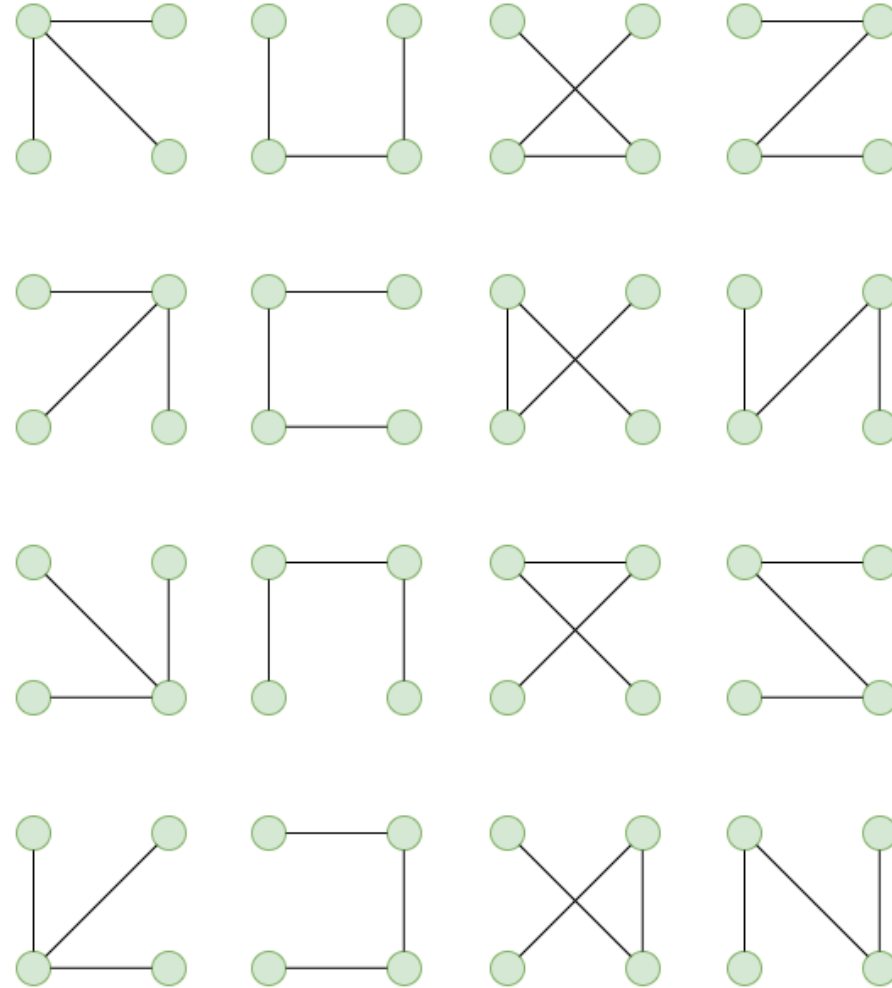
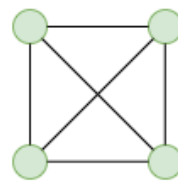
Prim's Algorithm

- Prim's Algorithm is a greedy algorithm that is used to **find the minimum spanning tree** from a graph
- Prim's algorithm finds the **subset of edges** that **includes every vertex** of the graph such that the **sum of the weights** of the edges can be **minimized**.

Spanning Tree

- Given a graph $G=(V,E)$, a subgraph of G that is connects all of the vertices and is a tree is called a spanning tree

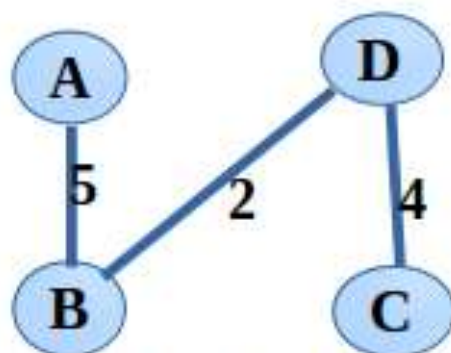
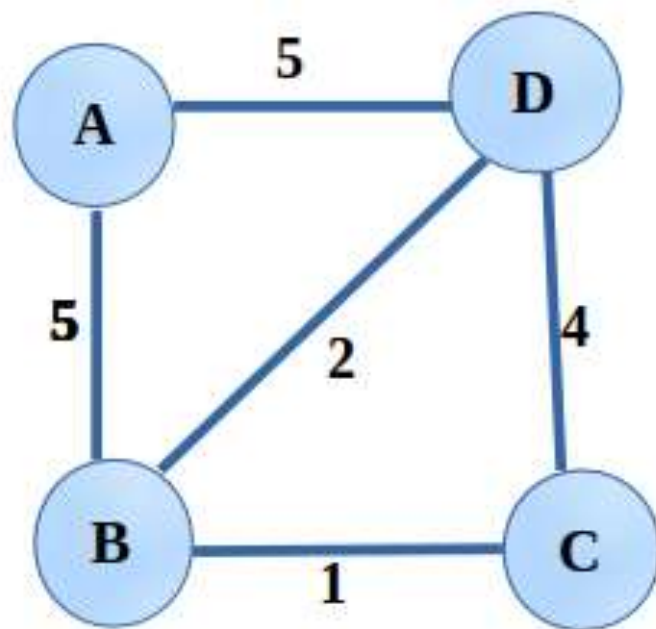
Graph =



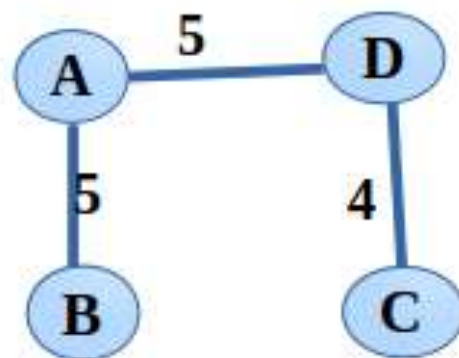
All Possible Spanning Trees of the Graph

Minimum Spanning Tree (MST)

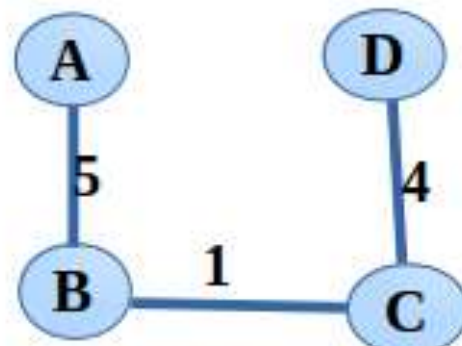
- A ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges.



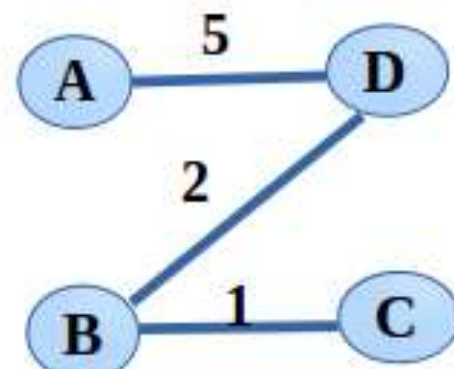
$$5+2+4 = 11$$



$$5+5+4 = 14$$

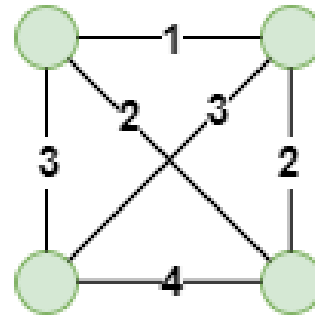


$$5+1+4 = 10$$

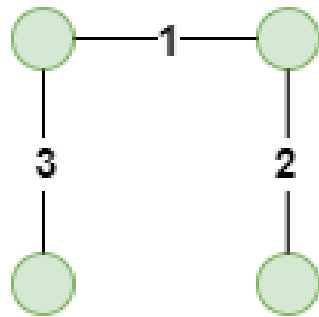


$$2+5+1 = 8$$

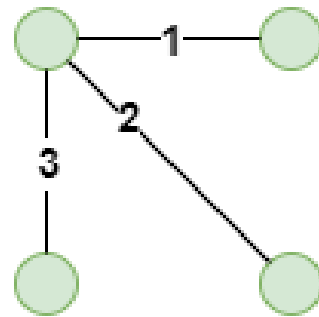
Graph(V,E) =



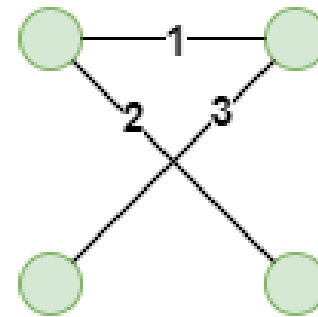
All Possible MST's of the above Graph



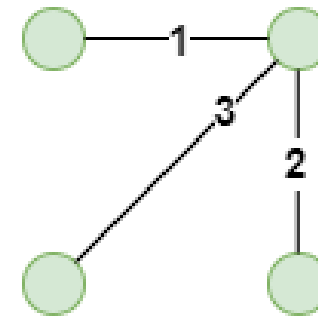
MST Cost =6



MST Cost =6



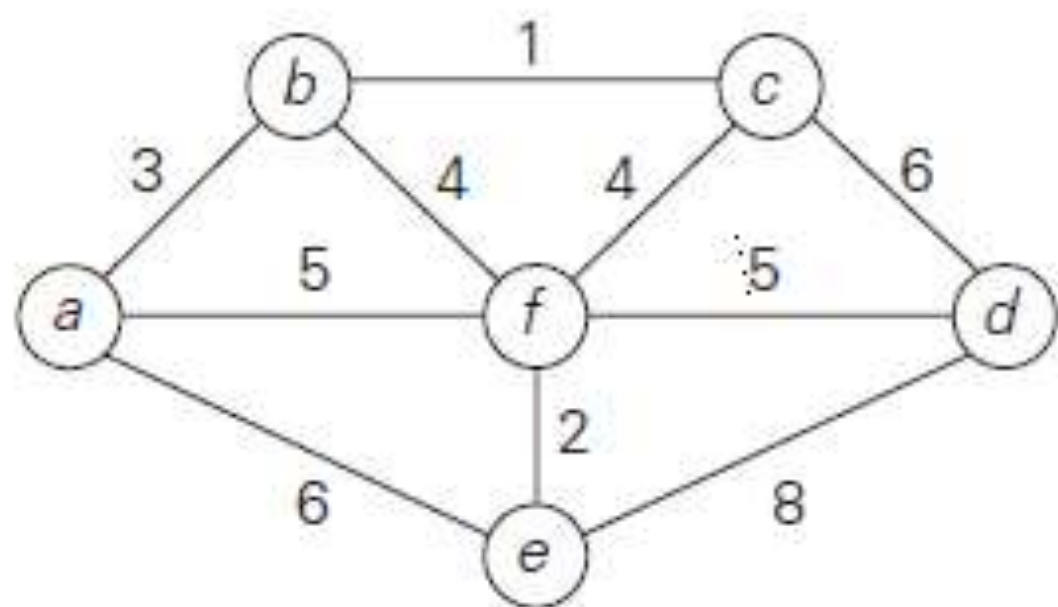
MST Cost =6

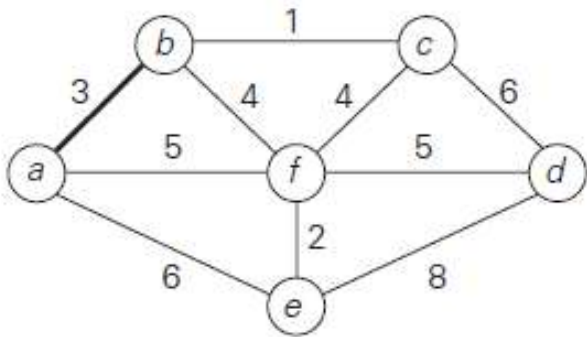
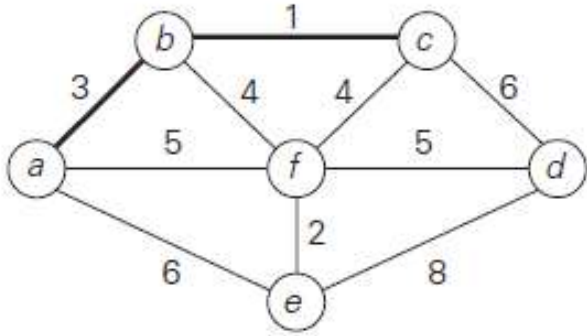
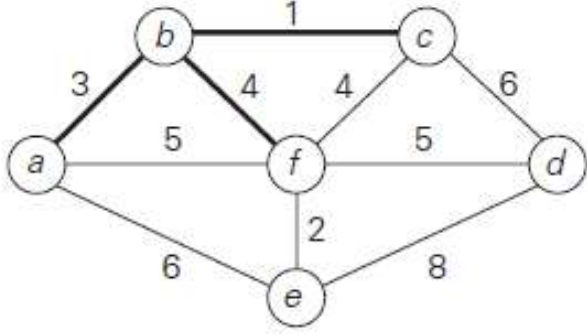


MST Cost =6

Applications of MST

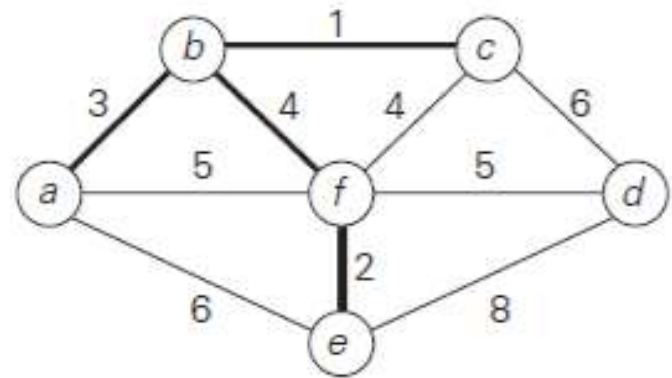
- The following problem arises naturally in many practical situations: given n points, connect them in the cheapest possible way so that there will be a path between every pair of points.
- It has direct applications to the design of all kinds of networks—including communication, computer, transportation, and electrical—by providing the cheapest way to achieve connectivity. It identifies clusters of points in data sets.



Tree vertices	Remaining vertices	Illustration
$a(-, -)$	$\mathbf{b(a, 3)}$ $c(-, \infty)$ $d(-, \infty)$ $e(a, 6)$ $f(a, 5)$	
$\mathbf{b(a, 3)}$	$\mathbf{c(b, 1)}$ $d(-, \infty)$ $e(a, 6)$ $f(b, 4)$	
$\mathbf{c(b, 1)}$	$d(c, 6)$ $e(a, 6)$ $\mathbf{f(b, 4)}$	

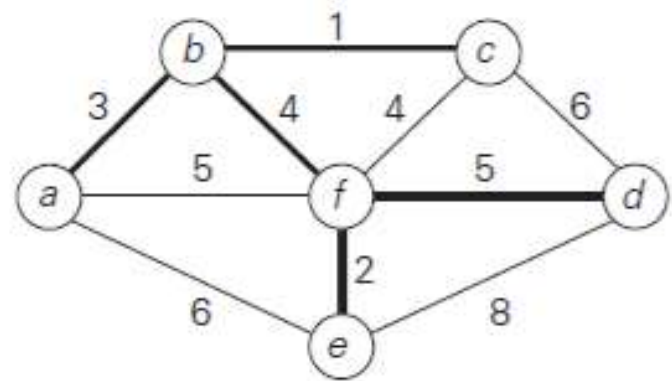
$f(b, 4)$

$d(f, 5)$ **$e(f, 2)$**



$e(f, 2)$

$d(f, 5)$



$d(f, 5)$

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 $V - V_T$

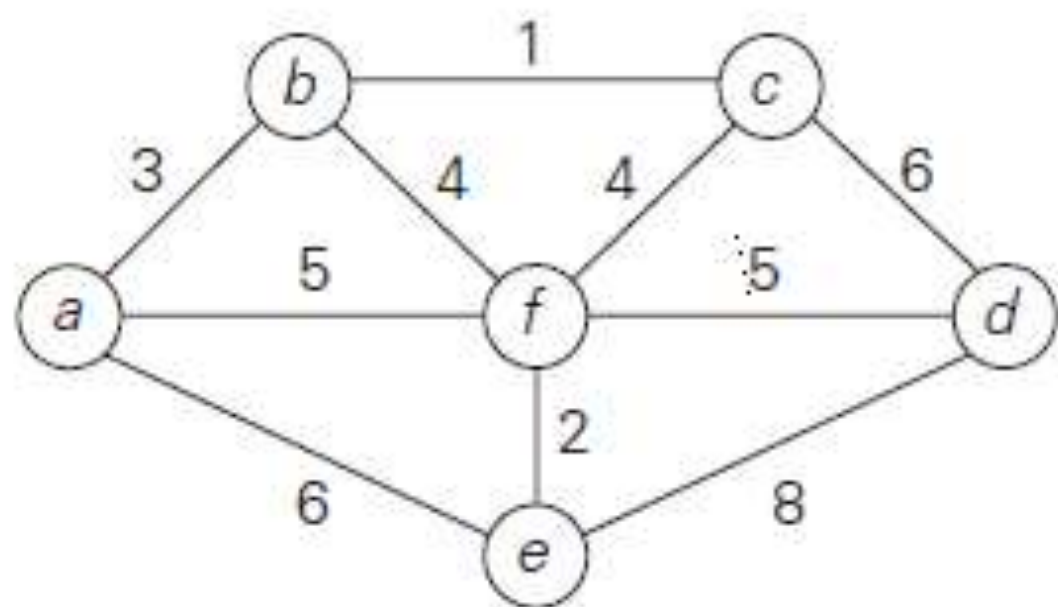
$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

Kruskal's algorithm

- The algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.
- The algorithm begins by **sorting the graph's edges in nondecreasing order of their weights**. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.



Tree edges	Sorted list of edges	Illustration
	bc 1 ef 2 ab 3 bf 4 cf 4 af 5 df 5 ae 6 cd 6 de 8	
bc 1	bc 1 ef 2 ab 3 bf 4 cf 4 af 5 df 5 ae 6 cd 6 de 8	
ef 2	bc 1 ef 2 ab 3 bf 4 cf 4 af 5 df 5 ae 6 cd 6 de 8	

ab
3

bc
1

ef
2

ab
3

bf
4

cf
4

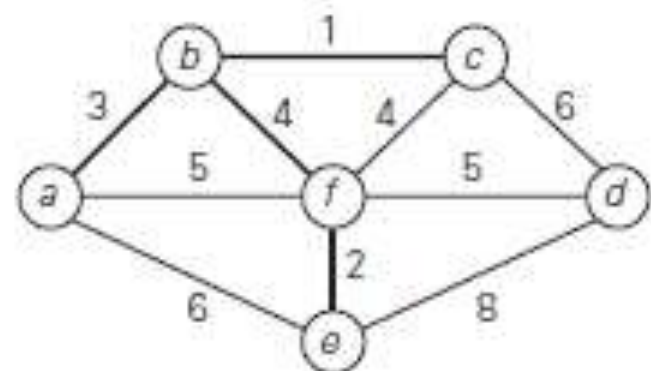
af
5

df
5

ae
6

cd
6

de
8



bf
4

bc
1

ef
2

ab
3

bf
4

cf
4

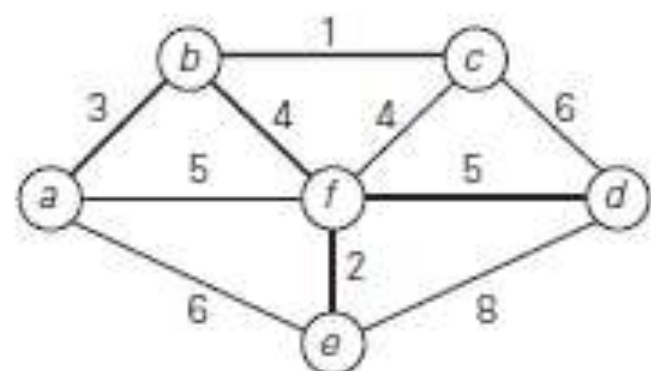
af
5

df
5

ae
6

cd
6

de
8



df
5

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 $w(e_{i_1}) \leq \dots \leq w(e_{i_{|E|}})$

$E_T \leftarrow \emptyset$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size

$k \leftarrow 0$ //initialize the number of processed edges

while $ecounter < |V| - 1$ **do**

$k \leftarrow k + 1$

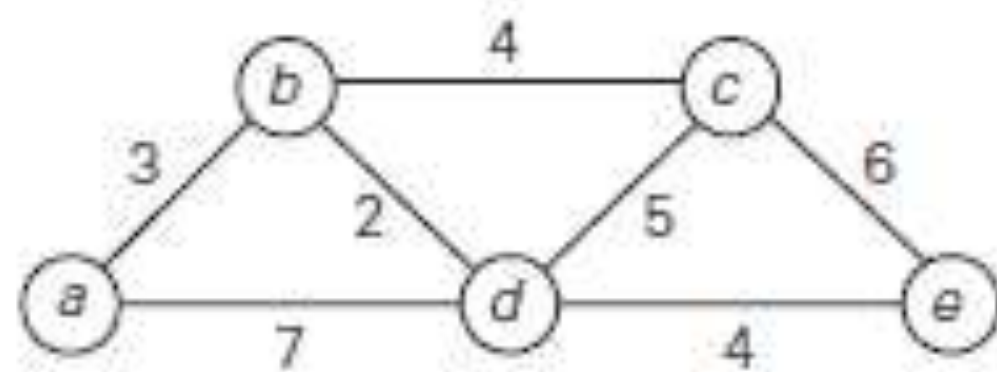
if $E_T \cup \{e_{i_k}\}$ is acyclic

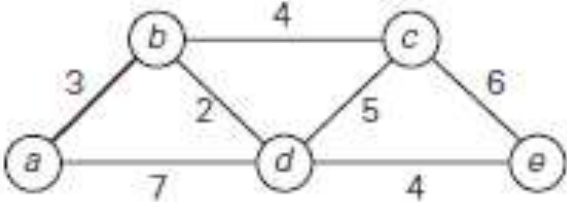
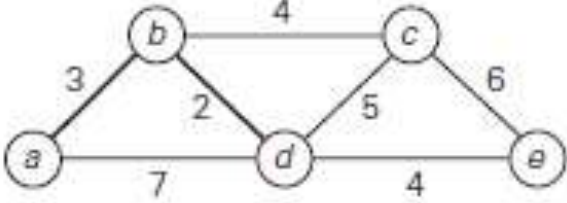
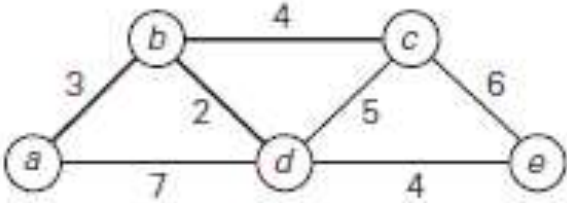
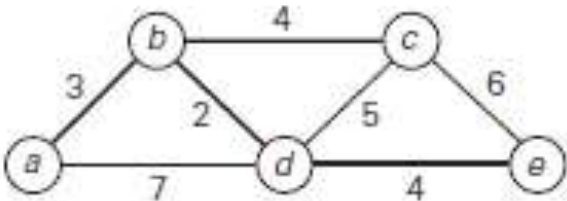
$E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$

return E_T

Dijkstra's Algorithm

- ***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.
- This algorithm is applicable to undirected and directed graphs with nonnegative weights only



Tree vertices	Remaining vertices	Illustration
$a(-, 0)$	$b(a, 3) \quad c(-, \infty) \quad d(a, 7) \quad e(-, \infty)$	
$b(a, 3)$	$c(b, 3 + 4) \quad d(b, 3 + 2) \quad e(-, \infty)$	
$d(b, 5)$	$c(b, 7) \quad e(d, 5 + 4)$	
$c(b, 7)$	$e(d, 9)$	
$e(d, 9)$		

ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths

//Input: A weighted connected graph $G = \langle V, E \rangle$ 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

$d_v \leftarrow \infty$; $p_v \leftarrow \mathbf{null}$

Insert(Q, v, d_v) //initialize vertex priority in the priority queue

$d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s

$V_T \leftarrow \emptyset$

for $i \leftarrow 0$ **to** $|V| - 1$ **do**

$u^* \leftarrow \text{DeleteMin}(Q)$ //delete the minimum priority element

$V_T \leftarrow V_T \cup \{u^*\}$

for every vertex u in $V - V_T$ that is adjacent to u^* **do**

if $d_{u^*} + w(u^*, u) < d_u$

$d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$

Decrease(Q, u, d_u)

Huffman Codes

Encoding messages

- ◆ Encode a message composed of a string of characters
- ◆ Codes used by computer systems
 - ASCII
 - uses 8 bits per character
 - can encode 256 characters
- ◆ ASCII and Unicode are *fixed-length codes*
 - all characters represented by same number of bits

Fixed Length code - Problems

- ◆ Suppose that we want to encode a message constructed from the symbols **A**, **B**, **C**, **D**, and **E** using a fixed-length code
 - How many bits are required to encode each symbol?
 - ◆ at least **3 bits** are required
 - ◆ 2 bits are not enough (can only encode four symbols)
 - ◆ How many bits are required to encode the message **DEAACAAAAABA**?
 - ◆ there are twelve symbols, each requires 3 bits
 - ◆ $12 \times 3 =$ **36 bits** are required

Drawbacks of fixed-length codes

- ◆ Same number of bits used to represent all characters
 - 'a' and 'e' occur more frequently than 'q' and 'z'
- ◆ **Potential solution:** use variable-length codes
 - variable number of bits to represent characters when frequency of occurrence is known
 - short codes for characters that occur frequently

Advantages of variable-length codes

- ◆ The advantage of variable-length codes over fixed-length is short codes can be given to characters that occur frequently
 - on average, the length of the encoded message is less than fixed-length encoding
- ◆ **Potential problem:** how do we know where one character ends and another begins?
 - not a problem if number of bits is fixed!

A = 00

B = 01

C = 10

D = 11

0010110111001111111111

A C D B A D D D D D

Prefix property

- ◆ A code has the **prefix property** if no character code is the prefix (start of the code) for another character
- ◆ Example:

Symbol	Code
P	000
Q	11
R	01
S	001
T	10

01001101100010

R S T Q P T

- ◆ 000 is not a prefix of 11, 01, 001, or 10
- ◆ 11 is not a prefix of 000, 01, 001, or 10 ...

Code without prefix property

- ◆ The following code does **not** have prefix property

Symbol	Code
P	0
Q	1
R	01
S	10
T	11

- ◆ The pattern **1110** can be decoded as **QQQP**, **QTP**, **QQS**, or **TS**

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.

Huffman coding tree

- ◆ Binary tree
 - each leaf contains symbol (character)
 - label edge from node to left child with 0
 - label edge from node to right child with 1
- ◆ Code for any symbol obtained by following path from root to the leaf containing symbol
- ◆ Code has prefix property
 - leaf node cannot appear on path to another leaf

Building a Huffman tree

- ◆ Find frequencies of each symbol occurring in message
- ◆ Begin with a forest of single node trees
 - each contain symbol and its frequency
- ◆ Do recursively
 - select two trees with smallest frequency at the root
 - produce a new binary tree with the selected trees as children and store the sum of their frequencies in the root
- ◆ Recursion ends when there is one tree
 - this is the Huffman coding tree

Example

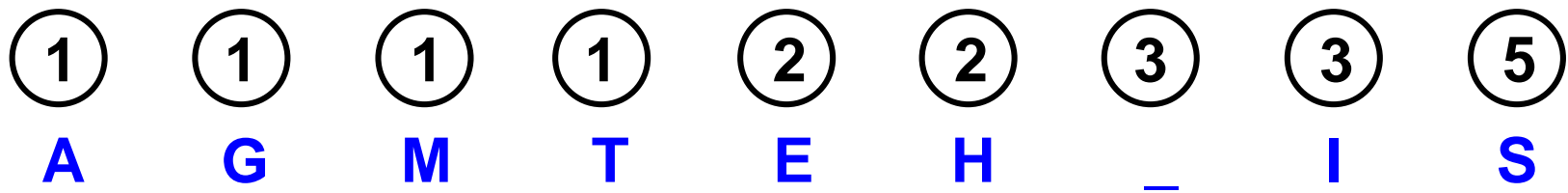
- ◆ Build the Huffman coding tree for the message

This is his message

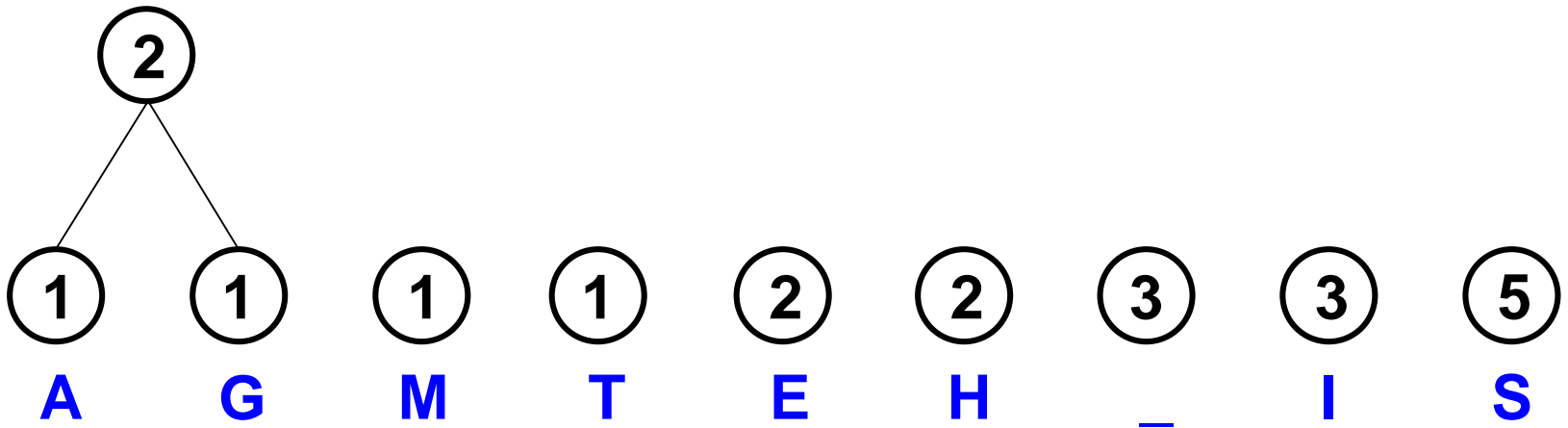
- ◆ Character frequencies

A	G	M	T	E	H	_	I	S
1	1	1	1	2	2	3	3	5

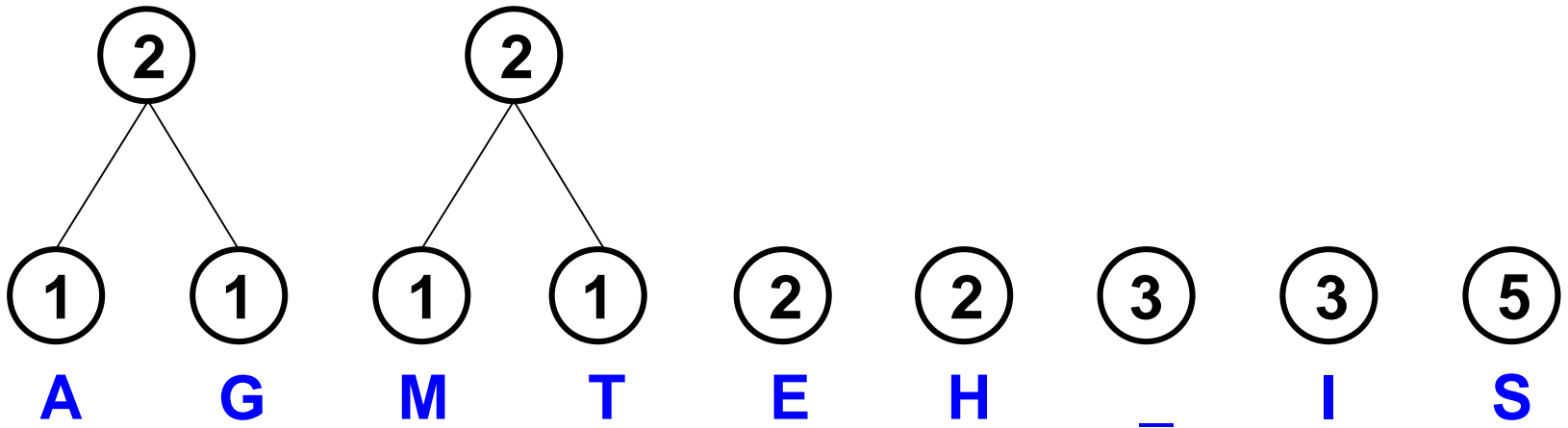
- ◆ Begin with forest of single trees



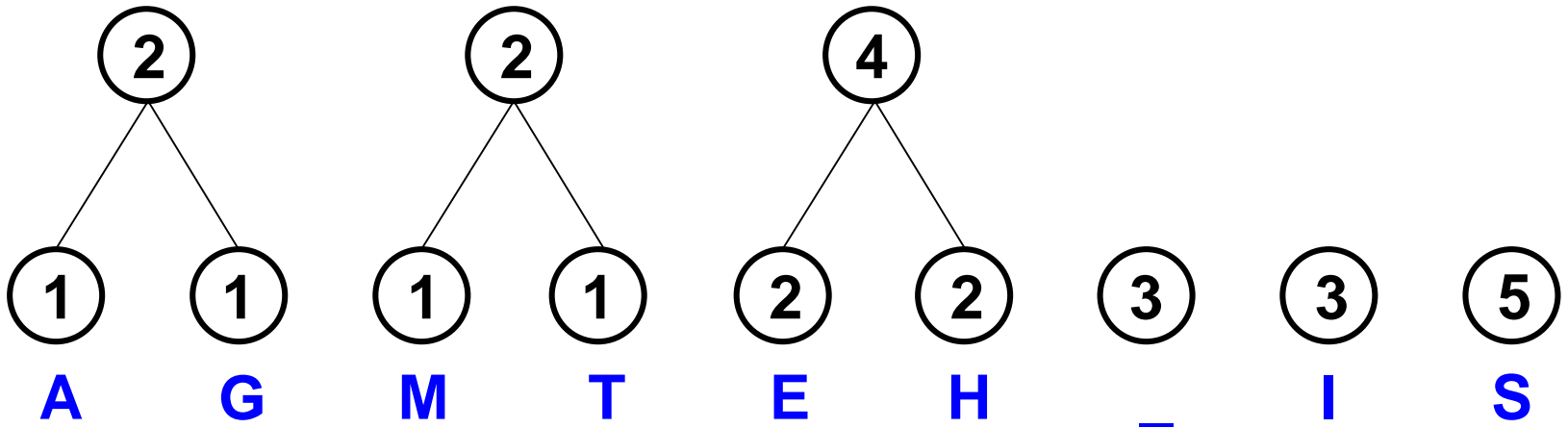
Step 1



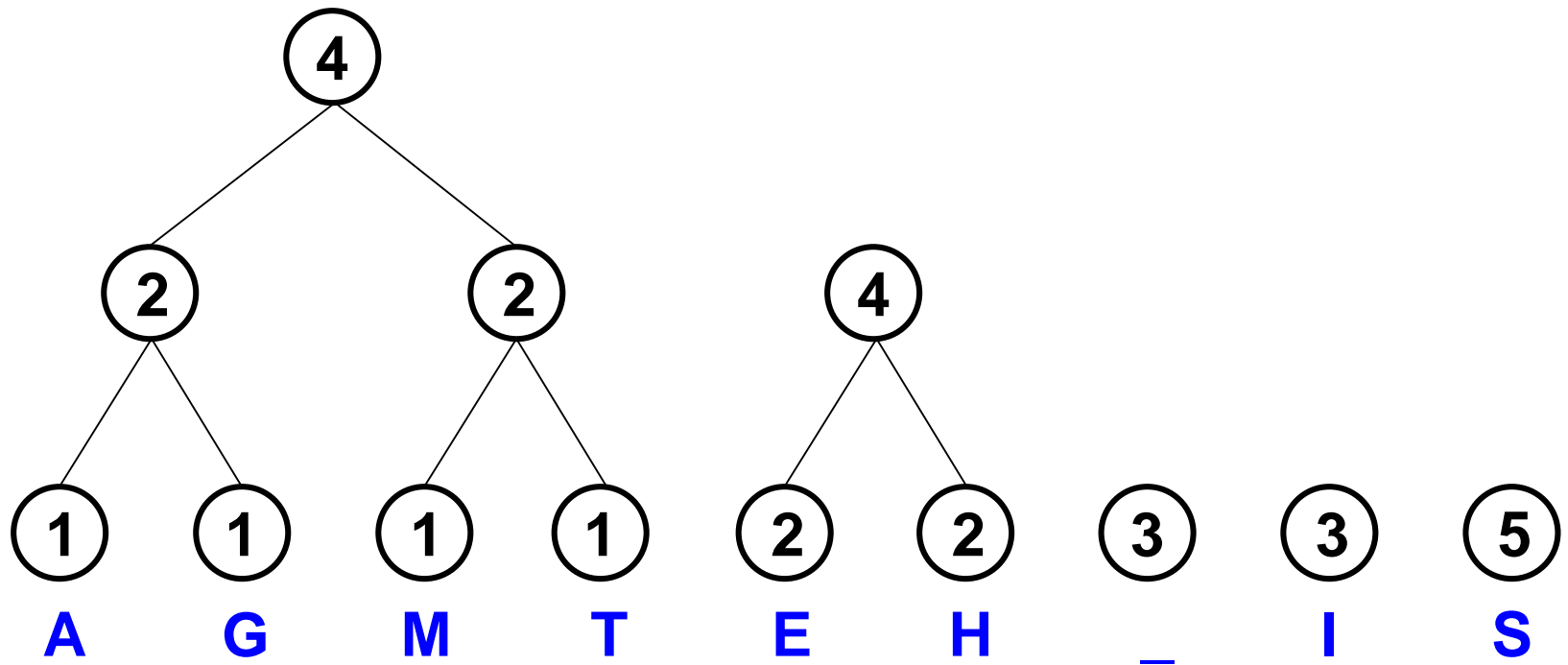
Step 2



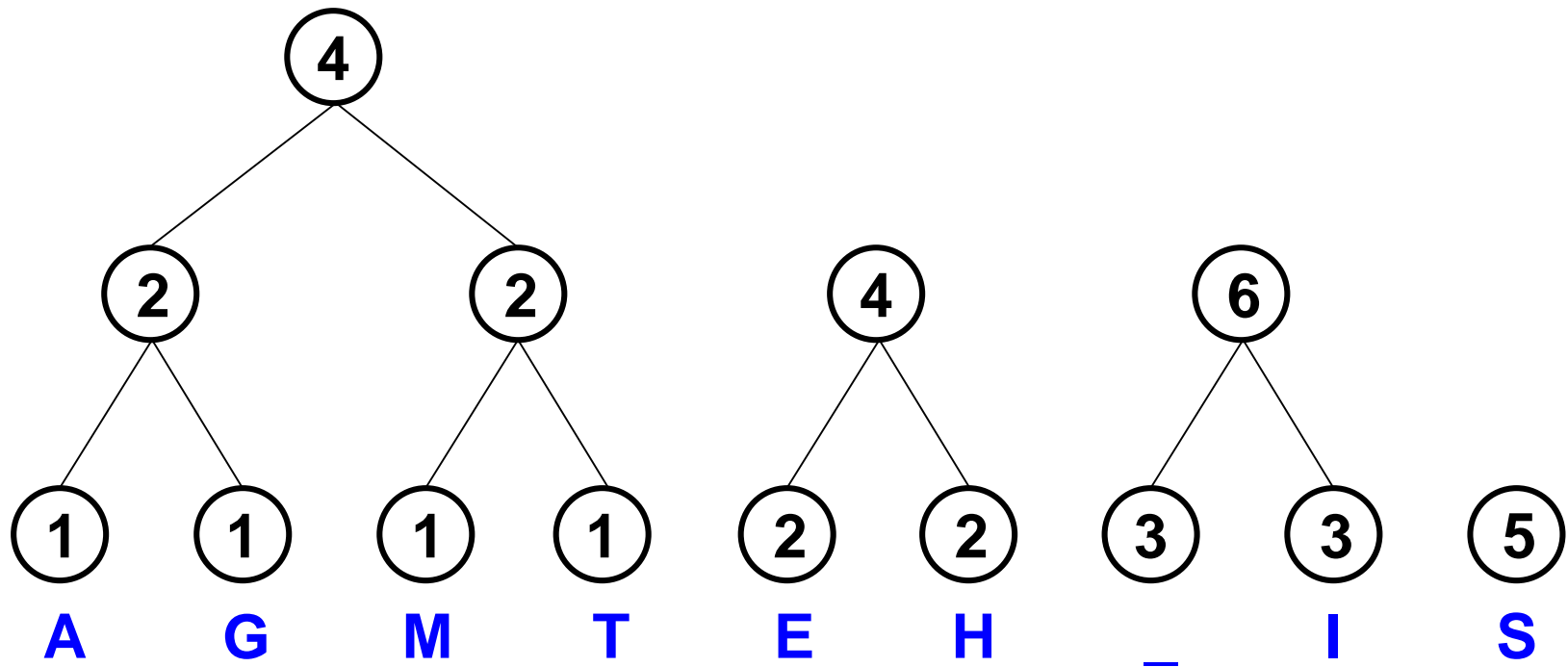
Step 3



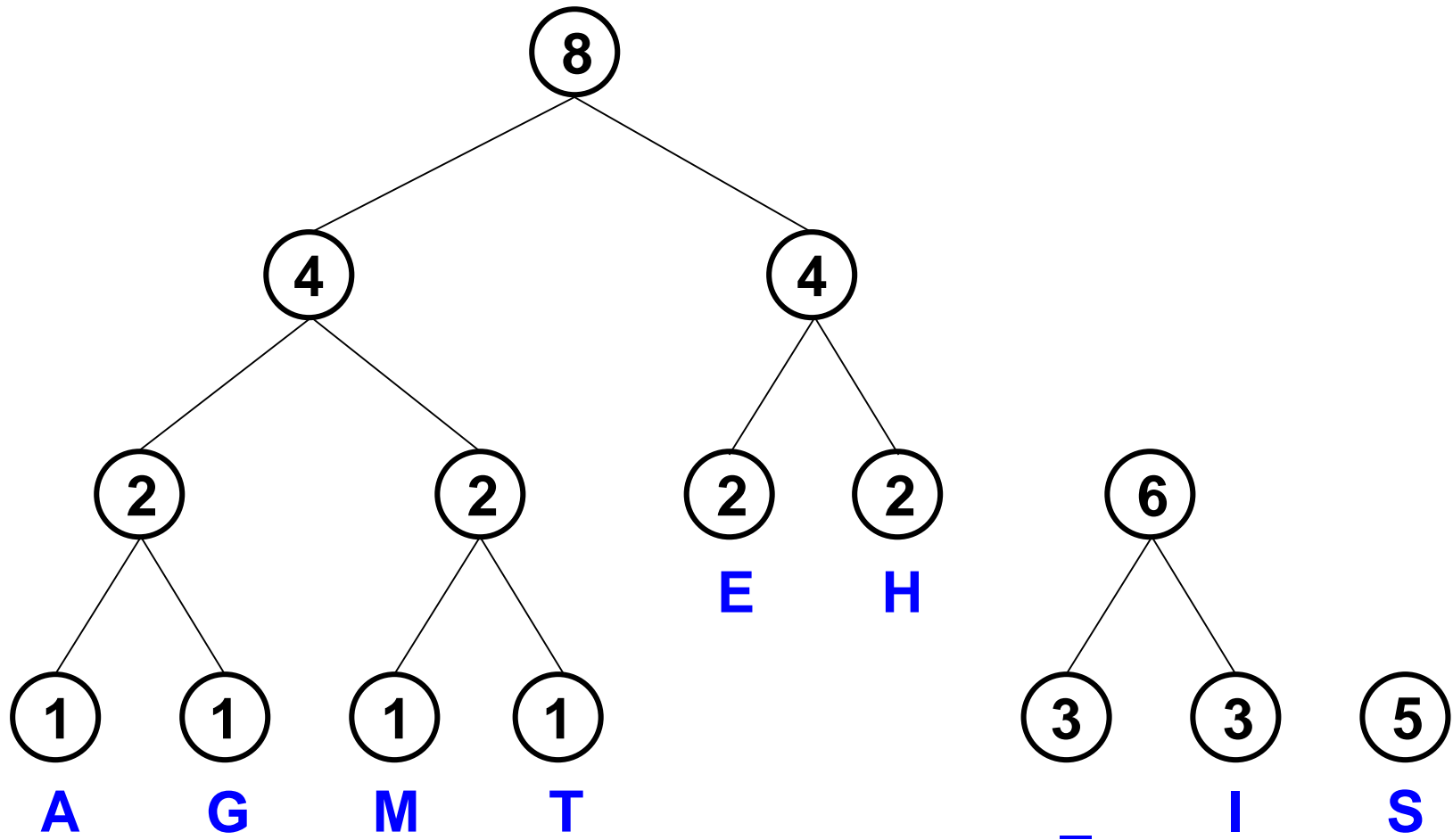
Step 4



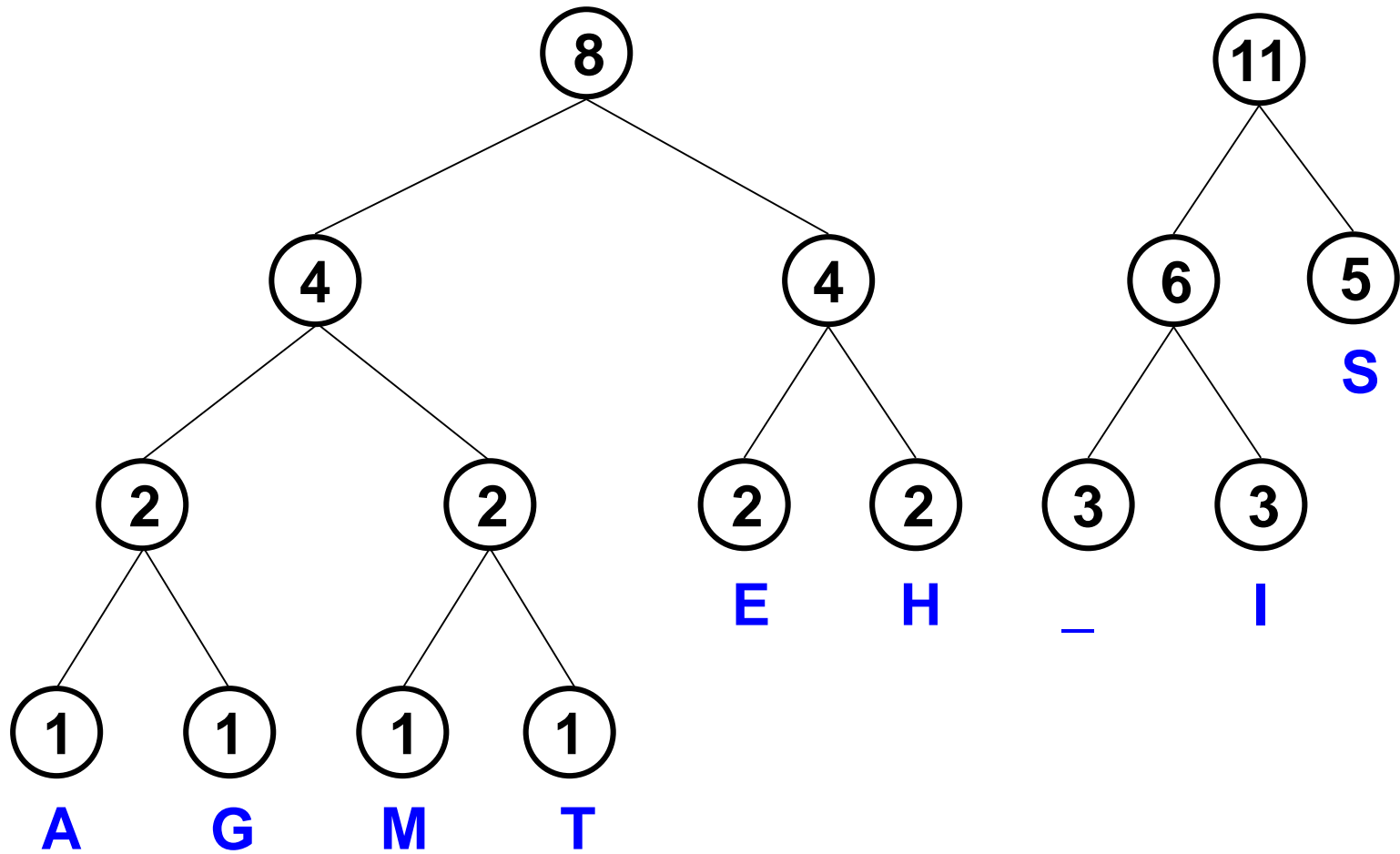
Step 5



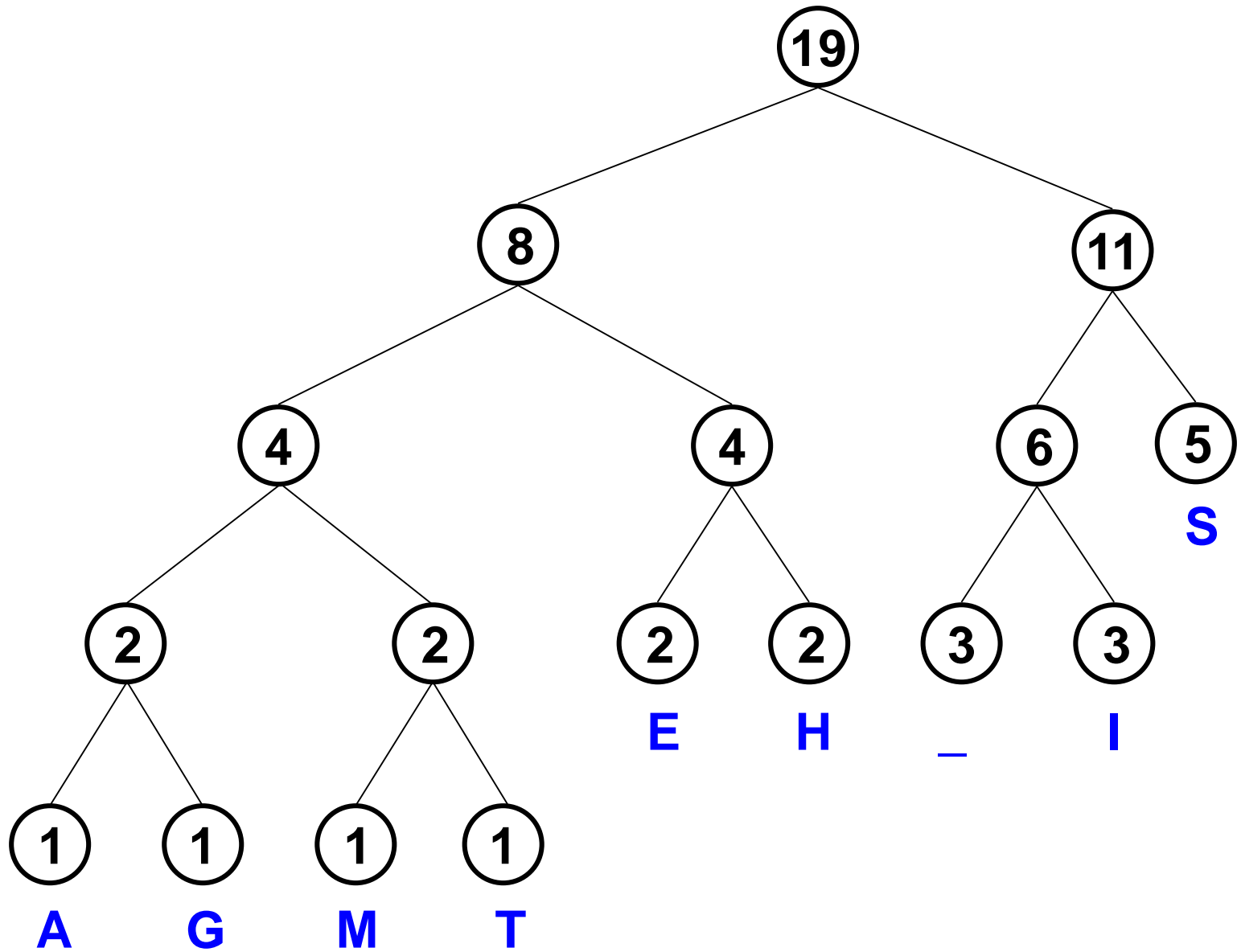
Step 6



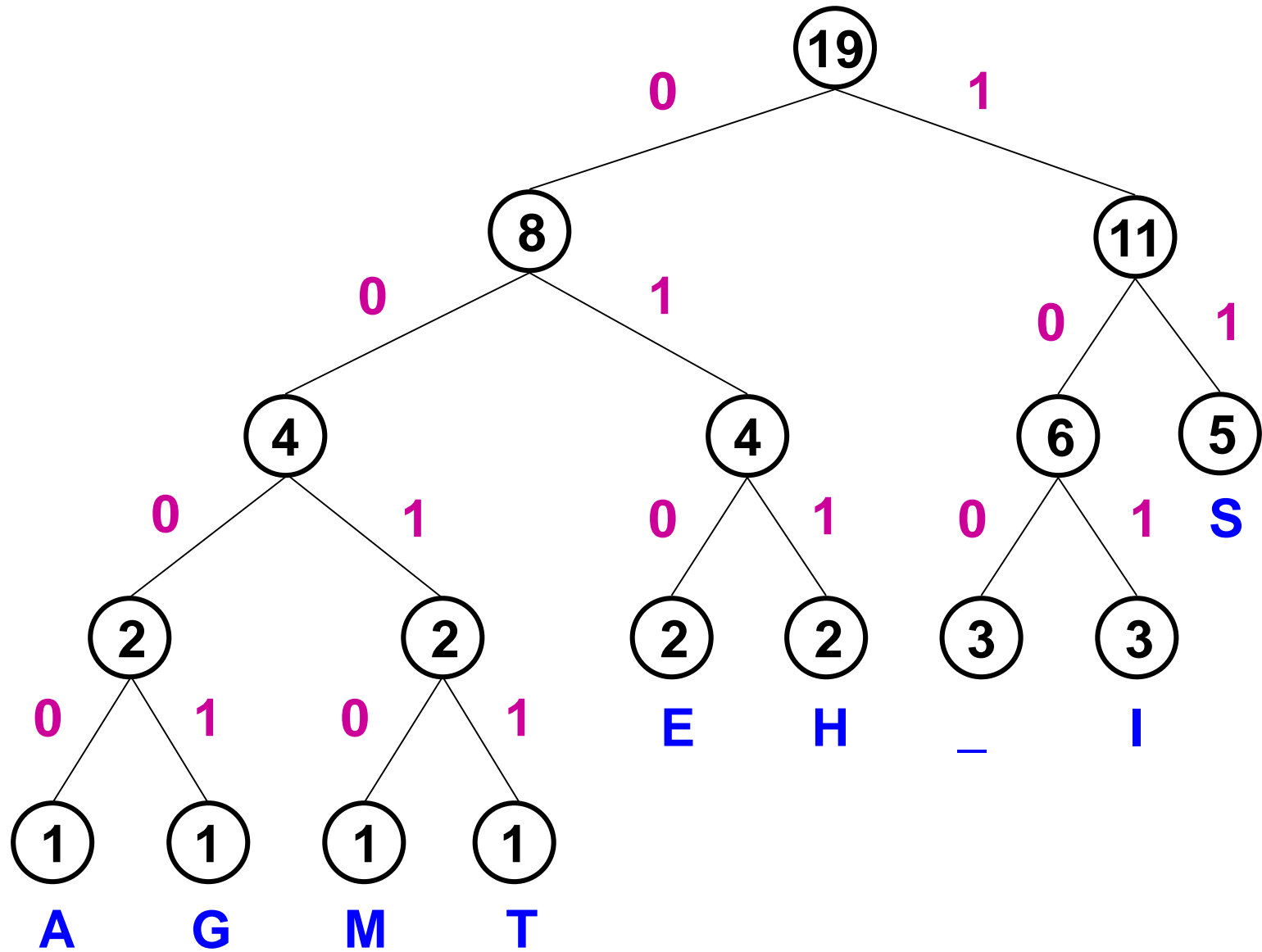
Step 7



Step 8



Label edges



Huffman code & encoded message

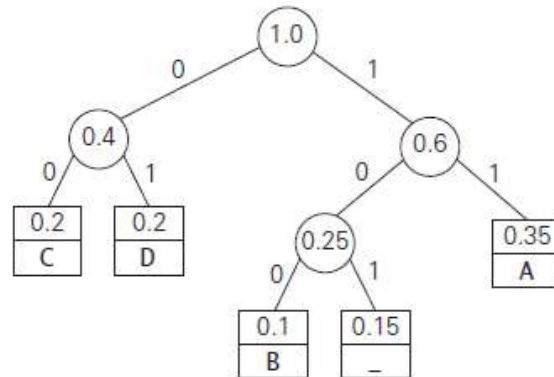
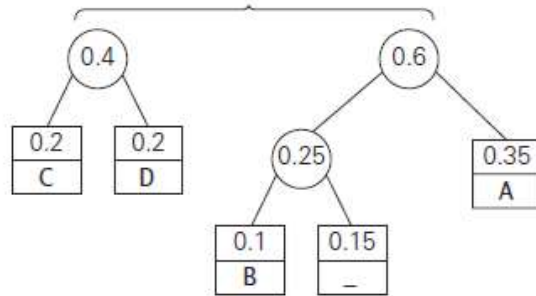
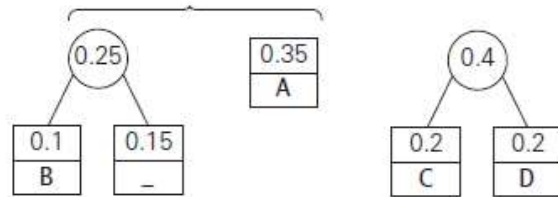
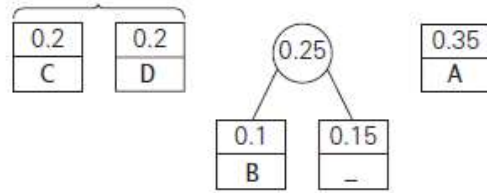
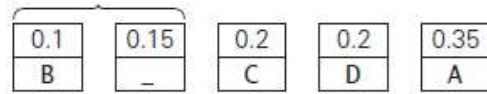
This is his message

S	11
E	010
H	011
_	100
I	101
A	0000
G	0001
M	0010
T	0011

00110111011110010111100011101111000010010111100000001010

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



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