The Design and Analysis of Algorithm

Exhaustive Search

Instructor: Sadullah Karimi
M.Sc. in CSE

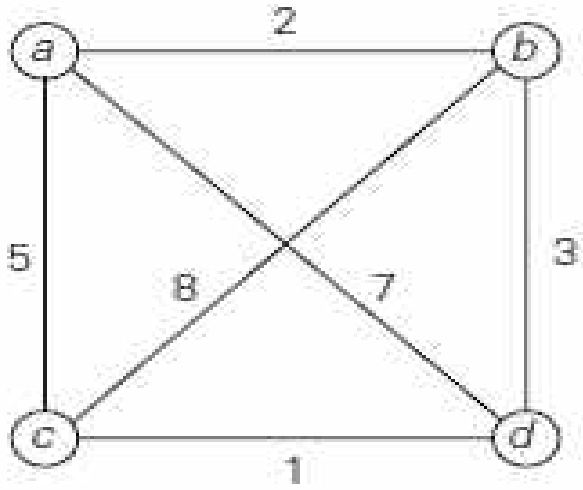
Exhaustive Search

- Exhaustive search is simply a brute-force approach to combinatorial problems.
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function).

- Three important problems:
  - The traveling salesman problem.
  - The knapsack problem.
  - The assignment problem.

Traveling Salesman Problem

- The traveling salesman problem (TSP) has been intriguing (Interesting) researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems.
- In layman's terms, the problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.
- The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.
- Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.
- It is named after the Irish mathematician Sir William Rowan Hamilton (1805–1865), who became interested in such cycles as an application of his algebraic discoveries.

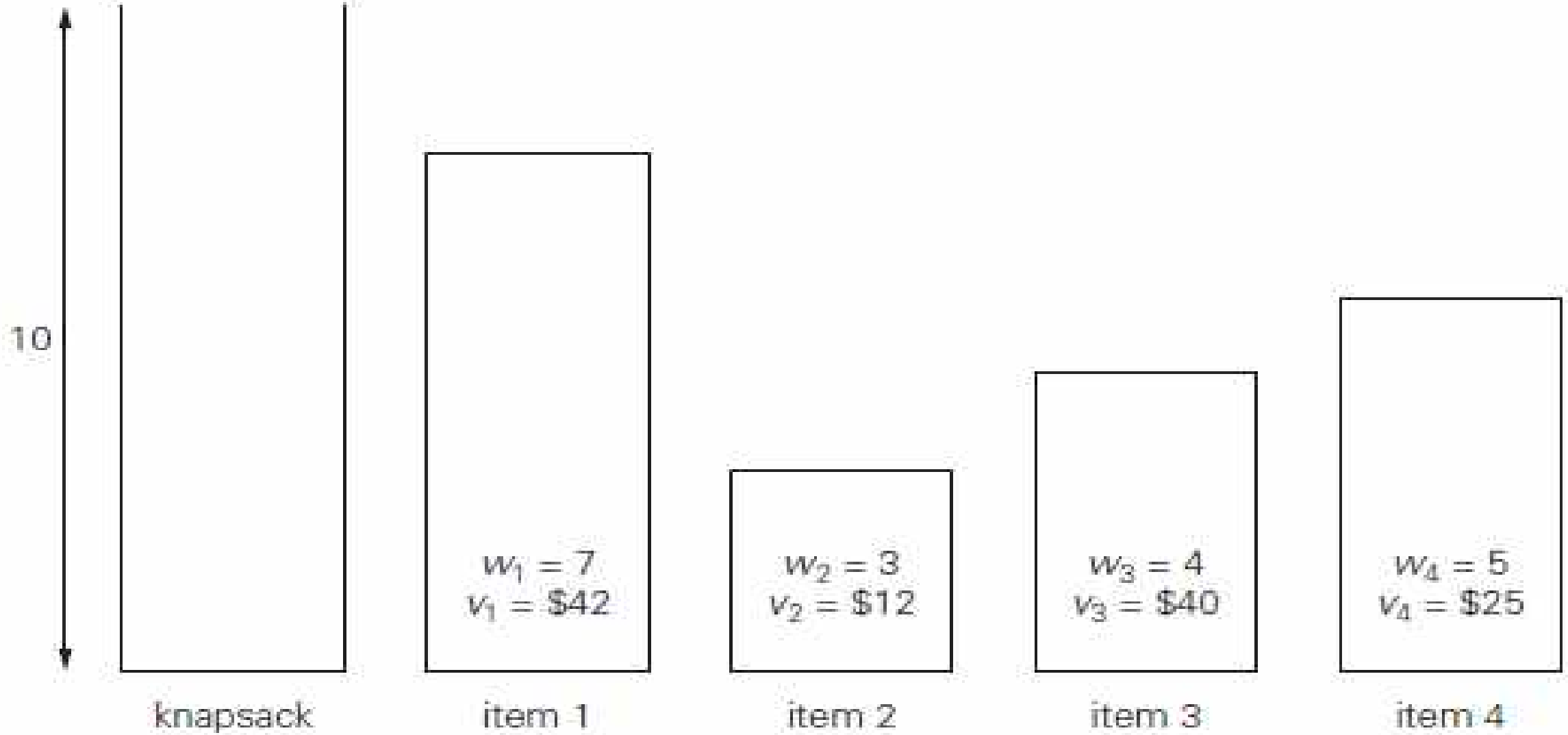


Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search.

Knapsack Problem

- Here is another well-known problem in algorithmics. Given n items of known weights w1, w2, . . . , wn and values v1, v2, . . . , vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. If you do not like the idea of putting yourself in the shoes of a thief who wants to steal the most valuable loot that fits into his knapsack, think about a transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.
- Figure 3.8a presents a small instance of the knapsack problem.
- an example, the solution to the instance of Figure 3.8a is given in Figure 3.8b.
- Since the number of subsets of an n-element set is , the exhaustive search leads to a () algorithm, no matter how efficiently individual subsets are generated.


10
$w_1 = 7$
$v_1 = \$42$
$w_2 = 3$
$v_2 = \$12$
$w_3 = 4$
$v_3 = \$40$
$w_4 = 5$
$v_4 = \$25$
knapsack
item 1
item 2
item 3
item 4

(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

FIGURE 3.8 (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. The information about the optimal selection is in bold.

Assignment Problem

In our third example of a problem that can be solved by exhaustive search, there are n people who need to be assigned to execute n jobs, one person per job.

(That is, each person is assigned to exactly one job and each job is assigned to exactly one person.)

The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i, j]$ for each pair

$i, j = 1, 2, \ldots, n$. The problem is to find an assignment with the minimum total cost.

A small instance of this problem follows, with the matrix representing the assignment costs C:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

$\langle 1, 2, 3, 4 \rangle$ cost = $9 + 4 + 1 + 4 = 18$

$\langle 1, 2, 4, 3 \rangle$ cost = $9 + 4 + 8 + 9 = 30$

$\langle 1, 3, 2, 4 \rangle$ cost = $9 + 3 + 8 + 4 = 24$

$\langle 1, 3, 4, 2 \rangle$ cost = $9 + 3 + 8 + 6 = 26$

$\langle 1, 4, 2, 3 \rangle$ cost = $9 + 7 + 8 + 9 = 33$

$\langle 1, 4, 3, 2 \rangle$ cost = $9 + 7 + 1 + 6 = 23$

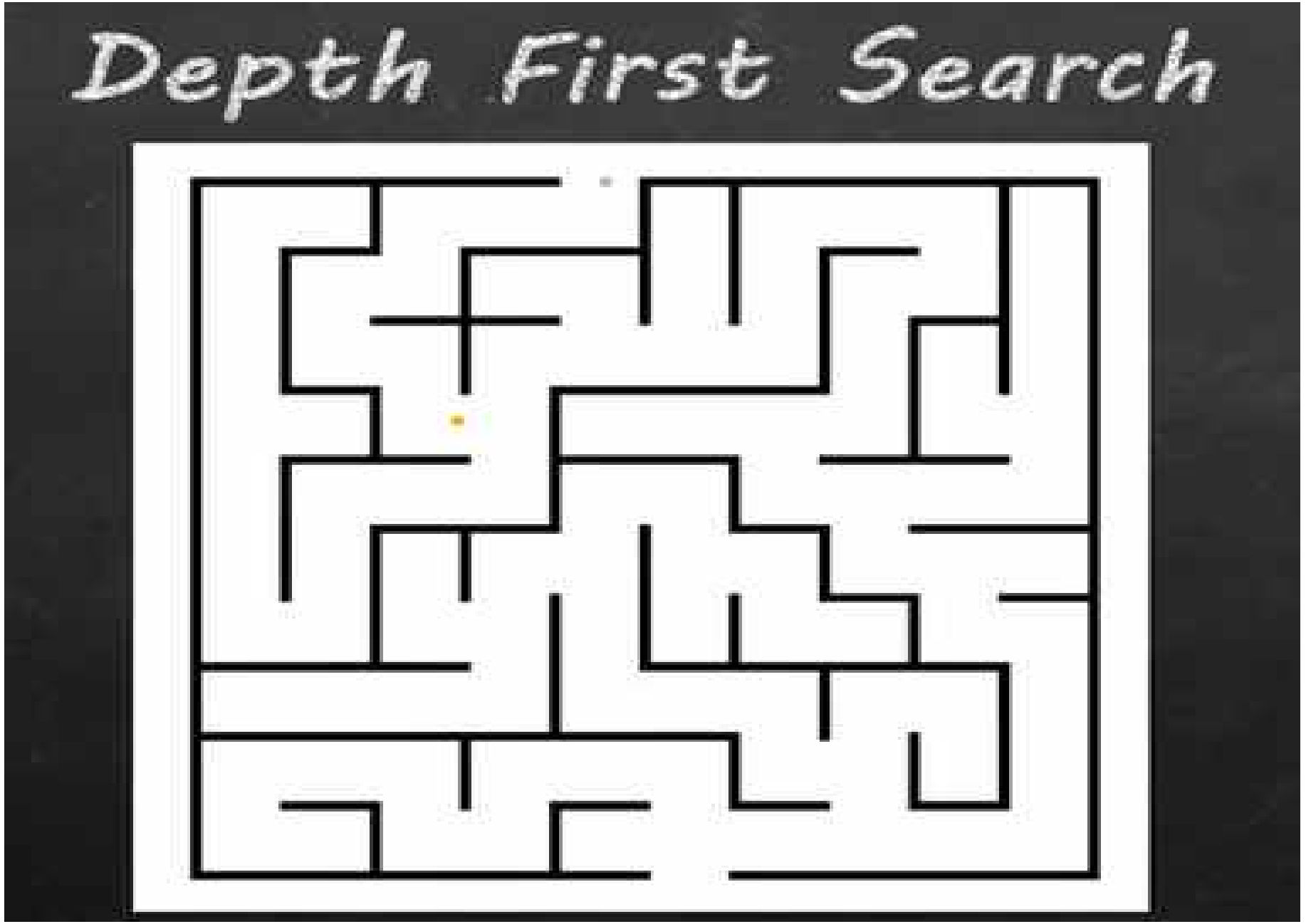
etc.

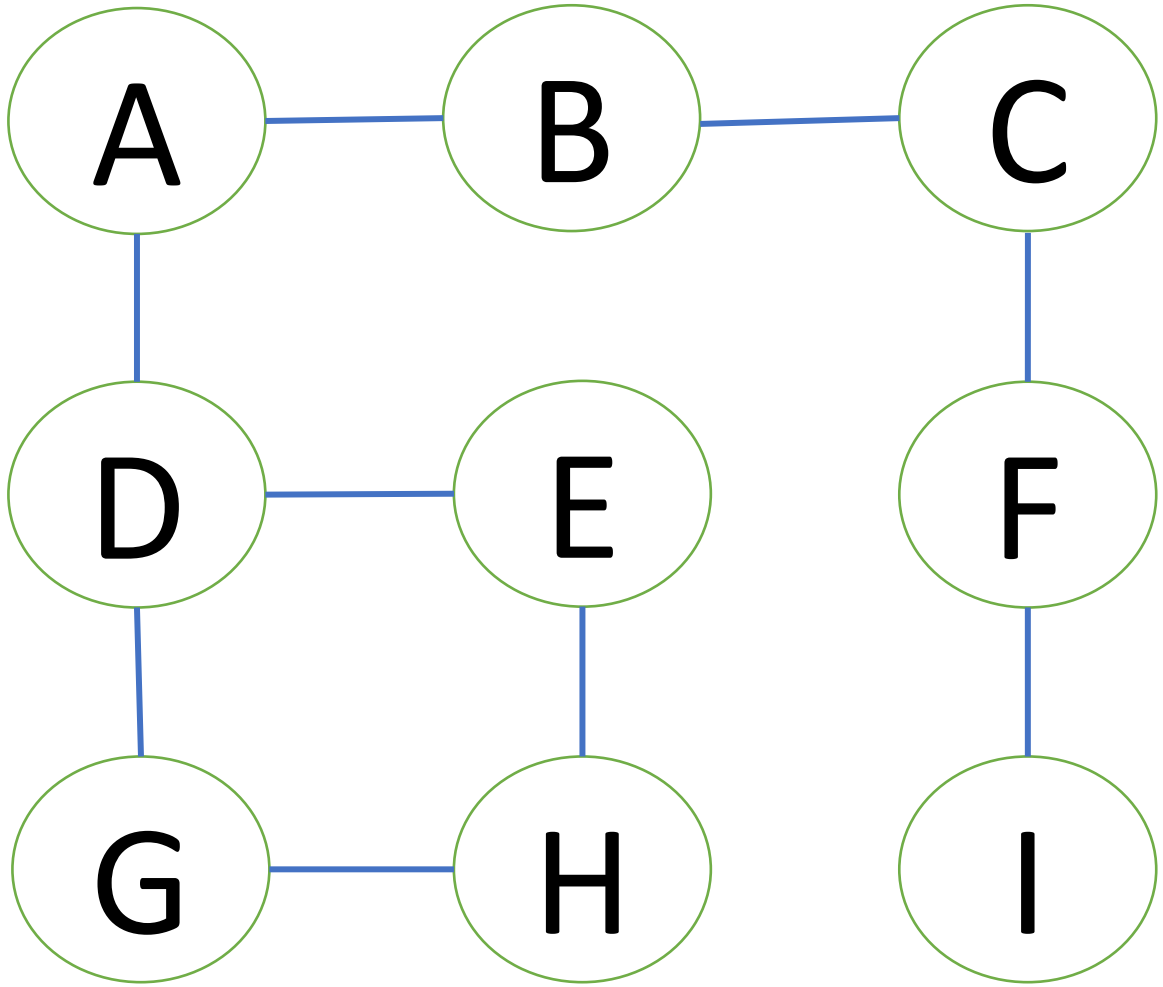
FIGURE 3.9 First few iterations of solving a small instance of the assignment problem by exhaustive search.

Depth-First Search

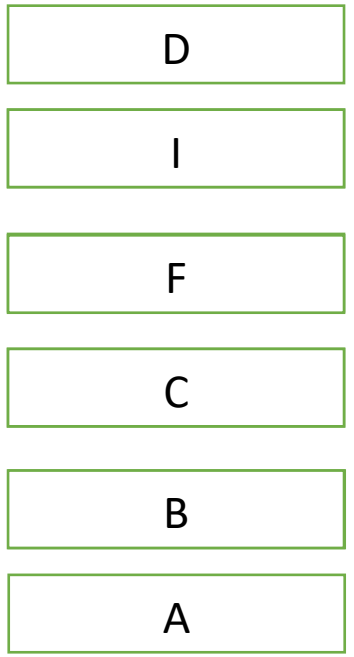
- ✓Depth-first search starts a graph's traversal (تقاطع گراف) at an arbitrary vertex by marking it as visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in.
- ✓ If there are several such vertices, a tie can be resolved arbitrarily.
- ✓ As a practical matter, which of the adjacent unvisited candidates is chosen is dictated by the data structure representing the graph.
- ✓In our examples, we always break ties by the alphabetical order of the vertices.
- ✓ This process continues until a dead end—a vertex with no adjacent unvisited vertices—is encountered.
- ✓Backtrack to last node that has unvisited adjacent neighbors.

Depth First Search





STACK



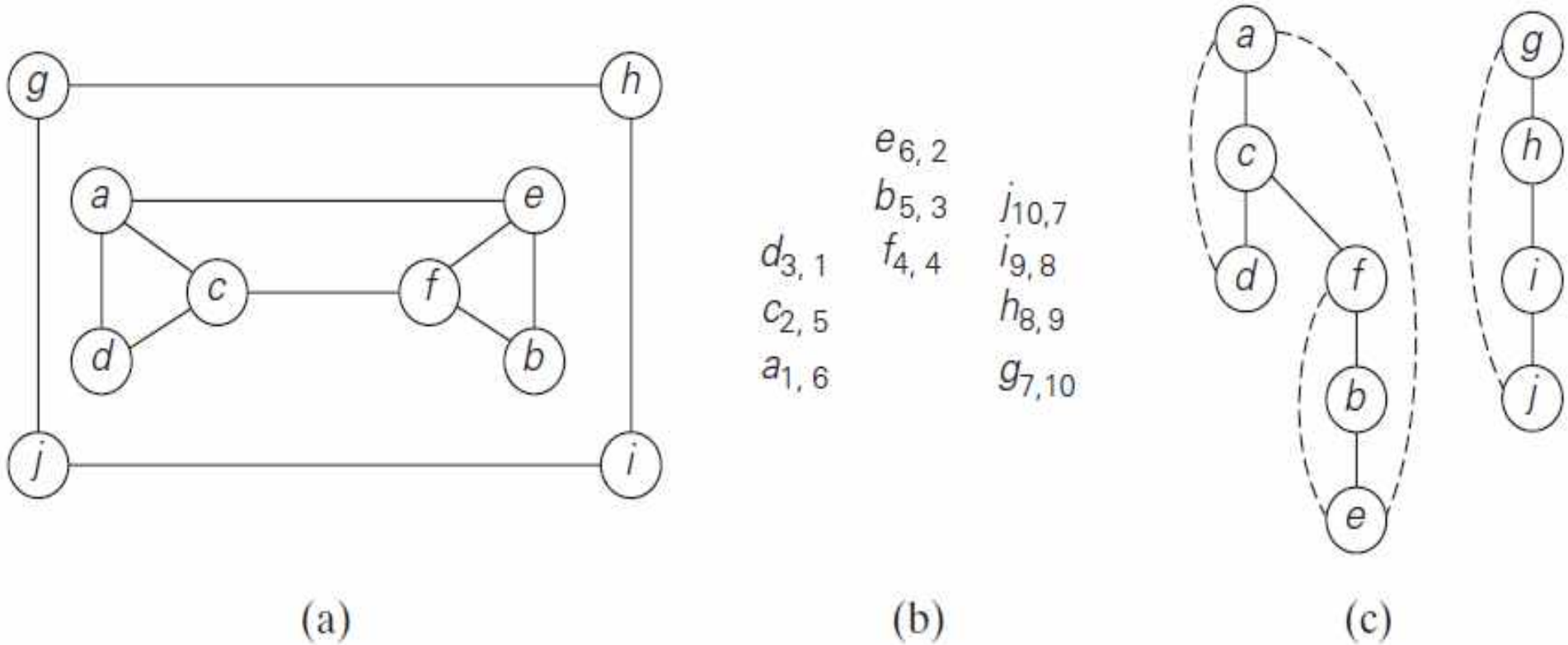


FIGURE 3.10 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

Pseudocode of the Depth-first Search

- **ALGORITHM** *DFS(G)*
- //Implements a depth-first search traversal of a given graph
- //Input: Graph $G = V, E$
- //Output: Graph G with its vertices marked with consecutive integers
- // in the order they are first encountered by the DFS traversal
- mark each vertex in V with 0 as a mark of being “unvisited”
- *count* ←0
- **for** each vertex v in V **do**
- **if** v is marked with 0
- *dfs(v)*
- *dfs(v)*
- //visits recursively all the unvisited vertices connected to vertex v
- //by a path and numbers them in the order they are encountered
- //via global variable *count*
- *count* ←*count* + 1; mark v with *count*
- **for** each vertex w in V adjacent to v **do**
- **if** w is marked with 0
- *dfs(w)*

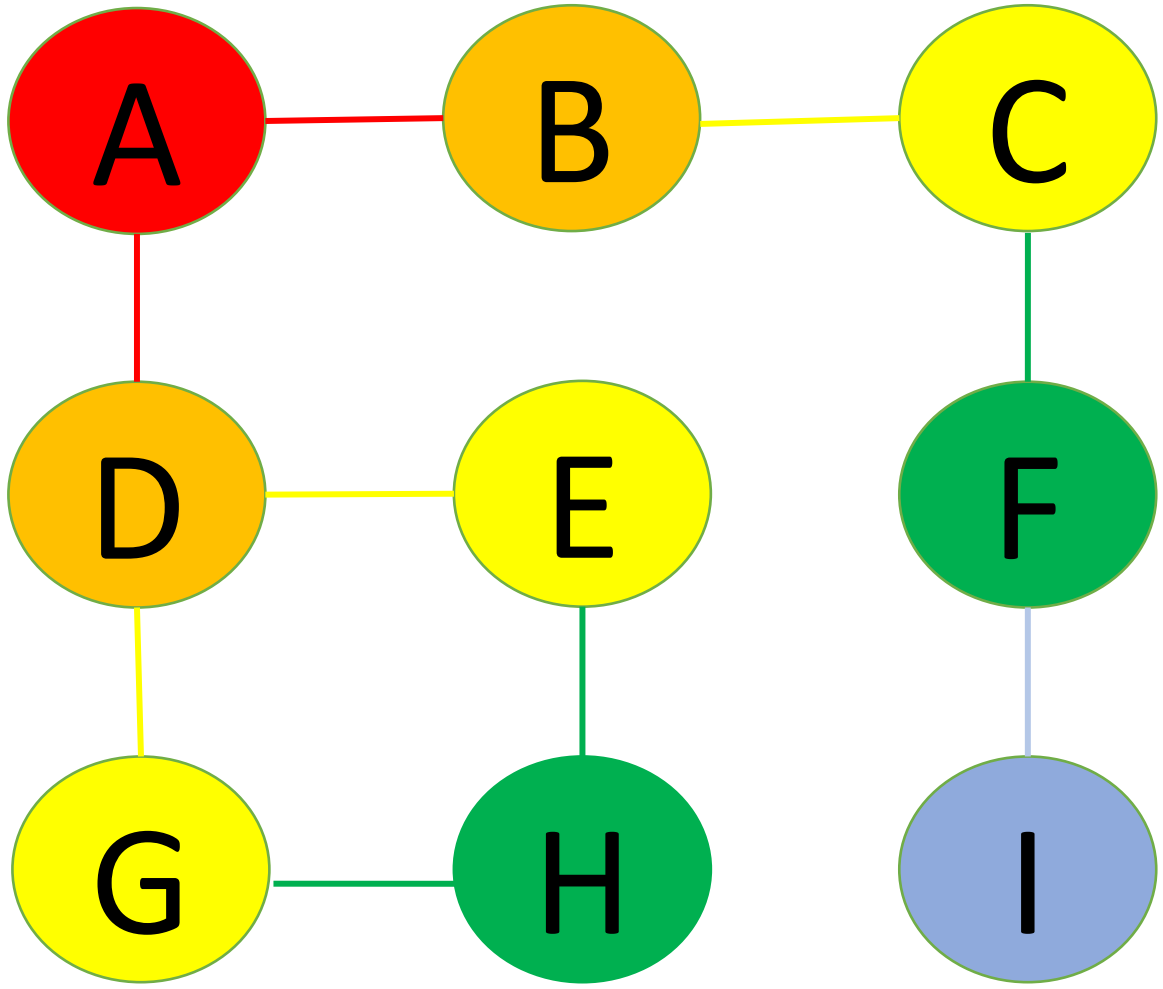
```
def DFS(G):
    # G is the graph with vertices V and edges E
    # Initialize all vertices as unvisited (0 means unvisited)
    for v in G.V:
        v.mark = 0  # 0 denotes unvisited

    count = 0  # To track the order of visits

    # For every vertex in G, if it's unvisited, perform DFS from it
    for v in G.V:
        if v.mark == 0:
            dfs(v, count)  # Start DFS traversal from vertex v

def dfs(v, count):
    # This recursive function performs the depth-first search
    count += 1
    v.mark = count  # Mark the vertex with the current count

    # For each vertex w adjacent to v, call DFS recursively if unvisited
    for w in v.adjacent:
        if w.mark == 0:
            dfs(w, count)
```

Explanation

- ✓Explanation:
- ✓Initialization:
  - First, we initialize all the vertices with a "0" mark (unvisited).
  - The global count is initialized to 0.
- ✓DFS Function:
  - This function is a recursive depth-first search that, for each vertex v, updates its mark with a consecutive integer, count.
  - Then, it explores all adjacent vertices (w) recursively.
- ✓Main DFS Loop:
  - For each vertex v,
  - if it has not been visited (marked as 0), we start the DFS from v.
- ✓Potential improvements:
  - The graph could be represented as a dictionary or an adjacency list (depending on your language or structure).
  - Depending on the use case, you could choose to either pass count as an argument to the DFS function or use a global variable for it.

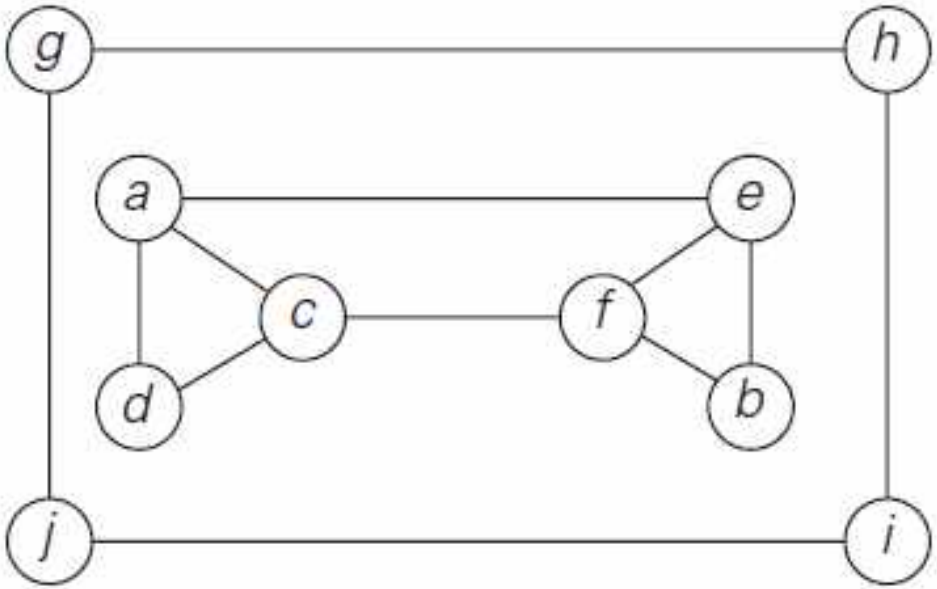
Breadth-First Search

- If depth-first search is a traversal for the brave (the algorithm goes as far from “home” as it can), breadth-first search is a traversal for the cautious.
- It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited.
- If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.
- This is done one level at time, rather one branch at time.

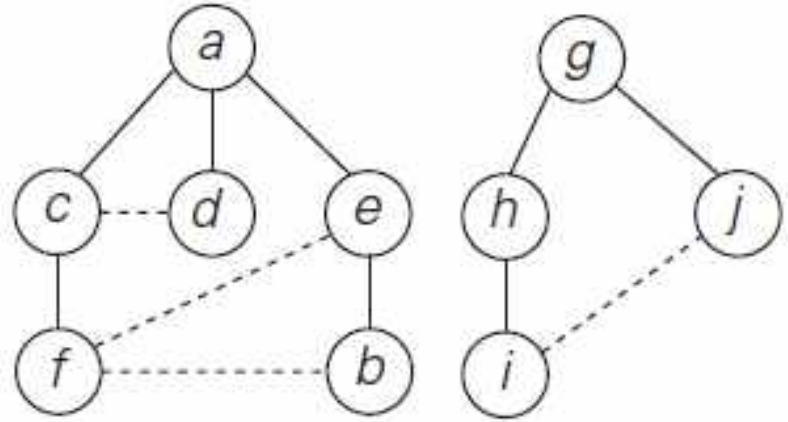


Queue

A B D C E G F H I



$a_1\ c_2\ d_3\ e_4\ f_5\ b_6$
$g_7\ h_8\ j_9\ i_{10}$



(a) (b) (c)

FIGURE 3.11 Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

Here is pseudocode of the breadth-first search.

- **ALGORITHM** *BFS(G)*
- //Implements a breadth-first search traversal of a given graph
- //Input: Graph $G = V, E$
- //Output: Graph G with its vertices marked with consecutive integers
- // in the order they are visited by the BFS traversal
- mark each vertex in V with 0 as a mark of being “unvisited”
- *count* ←0
- **for** each vertex v in V **do**
- **if** v is marked with 0
- *bfs(v)*
- *bfs(v)*
- //visits all the unvisited vertices connected to vertex v
- //by a path and numbers them in the order they are visited
- //via global variable *count*
- *count* ←*count* + 1; mark v with *count* and initialize a queue with v
- **while** the queue is not empty **do**
- **for** each vertex w in V adjacent to the front vertex **do**
- **if** w is marked with 0
- *count* ←*count* + 1; mark w with *count*
- add w to the queue
- remove the front vertex from the queue

```
def BFS(G):
    # G is the graph with vertices V and edges E
    # Initialize all vertices as unvisited (0 means unvisited)
    for v in G.V:
        v.mark = 0  # 0 denotes unvisited

    count = 0  # To track the order of visits

    # For each vertex v in G, if it's unvisited, perform BFS starting from it
    for v in G.V:
        if v.mark == 0:
            bfs(v, count)  # Start BFS traversal from vertex v
```

```
def bfs(start_vertex, count):
    # This function performs breadth-first search from the start_vertex
    count += 1
    start_vertex.mark = count  # Mark the start vertex with the current count

    # Initialize the queue and enqueue the start vertex
    queue = [start_vertex]

    while queue:
        current_vertex = queue.pop(0)  # Dequeue the front vertex

        # For each vertex w adjacent to the current_vertex, visit it if unvisited
        for w in current_vertex.adjacent:
            if w.mark == 0:  # If w is unvisited
                count += 1
                w.mark = count  # Mark w with the current count
                queue.append(w)  # Enqueue w
```

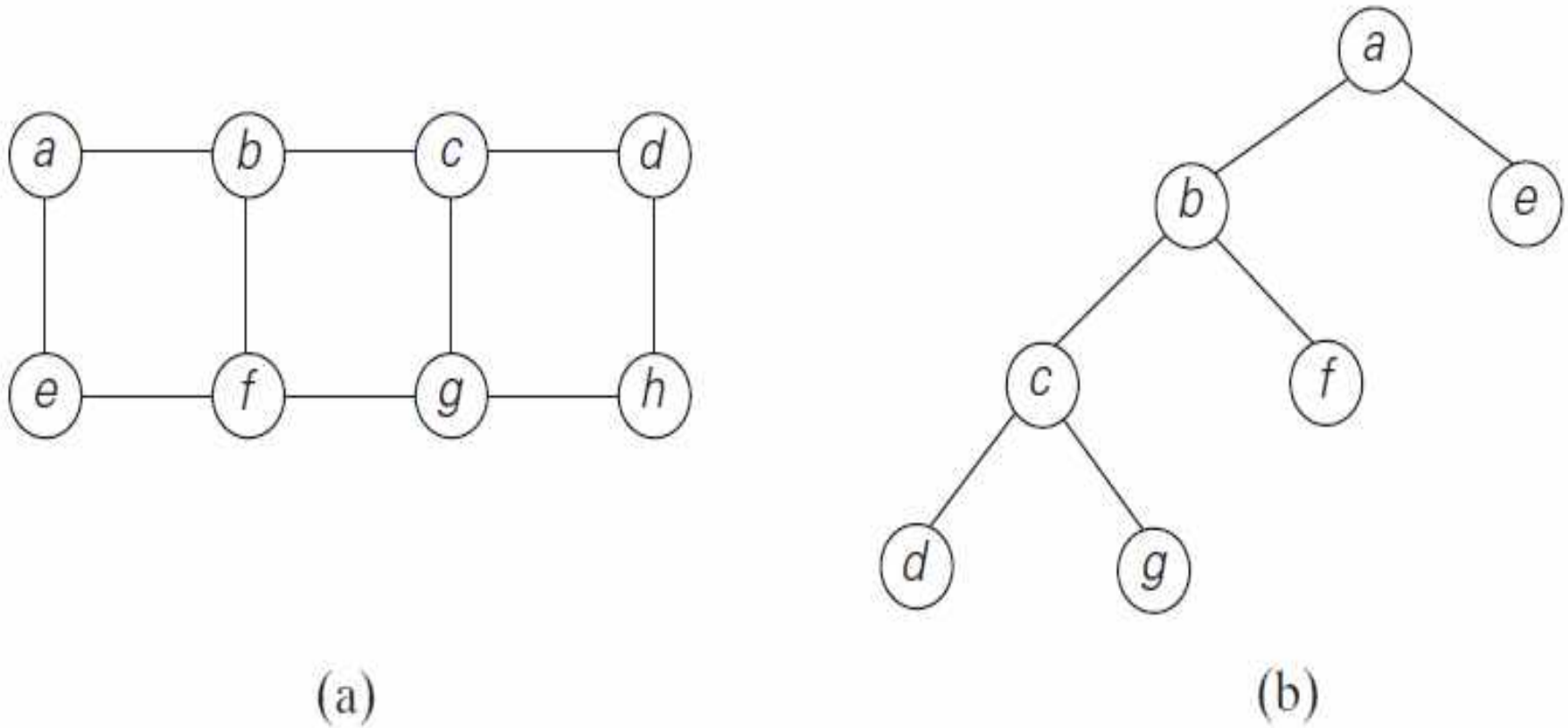


FIGURE 3.12 Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from a to g.

TABLE 3.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	**BFS**
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Questions: