Analysis of Algorithms

Chapter 3.1, 3.2, 3.4, 3.5

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - Bubble Sort
  - Sequential Search
  - String Matching
  - Travelling Salesman Problem
  - Knapsack Problem
  - Assignment Problem
  - Depth-First Search
  - Breadth-First Search

Brute Force

- Applicable to wide variety of problems
- Easiest way solving problem
  - Do not think much
  - Just do it!..
- ***Brute force*** is a straightforward approach based on
  - the problem's statement
  - definitions of the concepts

Brute Force

- Example :

  Compute a^n for a given number a and a nonnegative integer n

  By the definition of exponentiation,

$$a^n = \underbrace{a \times \ldots \times a}_{n \quad \text{times}}$$

Brute Force

- Brute force approach: used for many elementary but important algorithmic tasks

  - compute sum of *n* numbers
  - find largest element in a list
    - yields reasonable algorithms of practical value

  - sorting
  - searching
  - string matching
    - inefficient but can be used to solve small-size instances of a problem

ROAD MAP

- **Brute Force and Exhaustive Search**
  - **Selection Sort**
  - Bubble Sort
  - Sequential Search
  - String Matching
  - Travelling Salesman Problem
  - Knapsack Problem
  - Assignment Problem
  - Depth-First Search
  - Breadth-First Search

Sorting Problem

- **Problem definition:**

  Given a list of *n* items, rearrange them in non-decreasing order.

Selection Sort

- **Approach :**

  1. scan the entire given list to find its smallest element

  2. exchange it with the first element, i.e., put the smallest element in its final position

  3. scan the list starting with second element, find smallest among last n-1 elements

  4. exchange it with second element i.e., put the second smallest element in its final position

Selection Sort

In general

On the i^{th} pass of the algorithm through the list

search for the smallest item among the last *n-i* elements

swaps it with A_i

$$A_0 \le A_1 \le \ldots \le A_{i-1} \quad | \quad A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final position *the last n-i elements*

after *n-1* passes, the list is sorted

Selection Sort

- <u>**Example**</u>

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Selection sort's operation on the list 89, 45, 68, 90, 29, 34, 17

Selection Sort

- **Algorithm :**

```
// The algorithm sorts a given array by selection sort
// Input  : An array A[0 .. n-1] of orderable elements
// Output : Array A[0 .. n-1] sorted in ascending order

SelectionSort (A[0 .. n-1])
for i ← 0 to n-2 do
    min ← i
    for j ← i+1 to n-1 do
         if A[j] < A[min]    min ← j
    swap A[i] and A[min]
```

Selection Sort

- **Analysis :** The number of comparisons

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$C(n) = \sum_{i=0}^{n-2} \left[(n-1) - (i+1) + 1 \right]$$

$$C(n) = \sum_{i=0}^{n-2} (n-1-i)$$

$$C(n) = \frac{n(n-1)}{2} = \theta(n^2)$$

Selection Sort

- **Discussion :**
  - The number of key swaps is only θ(n)
    - more precisely, *n-1*
    - one for each repetation of the loop
  - But selection sort is still a θ(n^2) algorithm

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - **Bubble Sort**
  - Sequential Search
  - String Matching
  - Travelling Salesman Problem
  - Knapsack Problem
  - Assignment Problem
  - Depth-First Search
  - Breadth-First Search

Bubble Sort

- **Approach :**

1. `Compare adjacent elements of the list`
2. `Exchange them if they are out of order`

After *i*th pass

$$A_0 , \ldots , A_j \overset{?}{\longleftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \leq \ldots \leq A_{n-1}$$

in their final position

Bubble Sort

- **Example**

89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17		$\mid$90

45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17		$\mid$90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17		$\mid$90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17		$\mid$90
45		68		29		34		17		$\mid$89		90

etc.

The first two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17

Bubble Sort

- **Algorithm :**

```
// The algorithm sorts a given array by buble sort
// Input : An array A[0 .. n-1] of orderable elements
// Output : Array A[0 .. n-1] sorted in ascending order

BubbleSort ( A[0..n-1])
for i←0 to n-2 do
   for j←0 to n-2-i
      if A[j+1] < A[j] swap A[j] and A[j+1]
```

Bubble Sort

- **Analysis :** The number of comparisons

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$C(n) = \sum_{i=0}^{n-2} \left[(n-2-i) - 0 + 1 \right]$$

$$C(n) = \sum_{i=0}^{n-2} (n-1-i)$$

$$C(n) = \frac{n(n-1)}{2} \in \theta(n^2)$$

Bubble Sort

An improvement:

- If a pass makes no swaps, the list is already sorted
  - Do not make any more passes
- Does not improve worst case
- Just runs faster for some inputs

Bubble Sort

- **Discussion :**
  - The number of key swaps depends on the input.
    - for the worst case of decreasing arrays, same as the number of comparisons
  - The improved algorithm works very well on already sorted lists
    - performs just one pass on them

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - Bubble Sort
  - **Sequential Search**
  - String Matching
  - Travelling Salesman Problem
  - Knapsack Problem
  - Assignment Problem
  - Depth-First Search
  - Breadth-First Search

Search Problem

- **Definition :**

  Search Problem is to find a given item (some search key *K*) in a list of *n* elements

  - A match with the search key is found in the list
  - The search key is not in the list

Sequential Search

- **Approach :**

  1. compare successive elements of a given list with a given search key

  2. exit if search key matches the element of the list

  3. exit if search key does not match any elements of the list

Sequential Search

- **Algorithm :**

```
// The algorithm implements sequential search with a
search key as a sentinel
// Input  : An array of n elements and a search key
// Output : The position of the first element in array
A[0 .. n-1] whose value is equal to K or -1 if no such
element is found

SequentialSearch(A[0..n], K)
   A[n] ← K
   i ← 0
   while A[i] ≠ K do
     i ← i+1
   if i < n return i
   else return -1
```

Sequential Search

- **Analysis :**

$$T(n) = \theta\ (n)$$

Sequential Search

- Any idea of improvement?..
- What if the list is sorted?..

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - Bubble Sort
  - Sequential Search
  - **String Matching**
  - Travelling Salesman Problem
  - Knapsack Problem
  - Assignment Problem
  - Depth-First Search
  - Breadth-First Search

String Matching

- **Problem definition:**

Given a string of **n** characters called ***text***

Given a string of **m** characters called ***pattern***

Find a substring of the text that matches the pattern

String Matching

- Brute-force Approach?..

String Matching

- **Approach :**

  1. align the pattern against the first *m* characters of the text

  2. start matching the corresponding pairs of characters from left to right

     - until either all m pairs of the characters match or
     - mismatching pair is encountered

  3. shift the pattern one position to the right

  4. resume character comparisons, starting with the first character of the pattern and its counterpart in the text

String Matching

- **Example**

N	O	B	O	D	Y	_	N	O	T	I	C	E	D	_	H	I	M
N	**O**	**T**															
	N	O	T														
		N	O	T													
			N	O	T												
				N	O	T											
					N	O	T										
						N	O	T									
							N	**O**	**T**								

The pattern’s characters that are compared with their text counterparts are in **bold** type

String Matching

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of n characters representing a text and
// an array $P[0..m-1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a
// matching substring or -1 if the search is unsuccessful
for $i \leftarrow 0$ **to** $n-m$ **do**
    $j \leftarrow 0$
    while $j < m$ **and** $P[j] = T[i+j]$ **do**
        $j \leftarrow j+1$
    if $j = m$ **return** i
return -1

String Matching

- **Analysis??**

String Matching

- In worst case, the algorithm make all *m* comparisons before shifting the pattern
- There are *n-m+1* tries.
- So in worst case the algorithm is $\Theta(nm)$
- The average case should be better than worst case
  - All tries cannot make m comparisons!..
- It has been shown that in random texts, algorithm is $\Theta(n+m) = \Theta(n)$

String Matching

- **Discussion :**

  There are several more sophisticated and more efficient algorithms for string matching.

  We will discuss them later !

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - Bubble Sort
  - Sequential Search
  - String Matching
  - **Travelling Salesman Problem**
  - Knapsack Problem
  - Assignment Problem
  - Depth-First Search
  - Breadth-First Search

Traveling Salesman Problem

- **Problem definition :**

  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

Traveling Salesman Problem

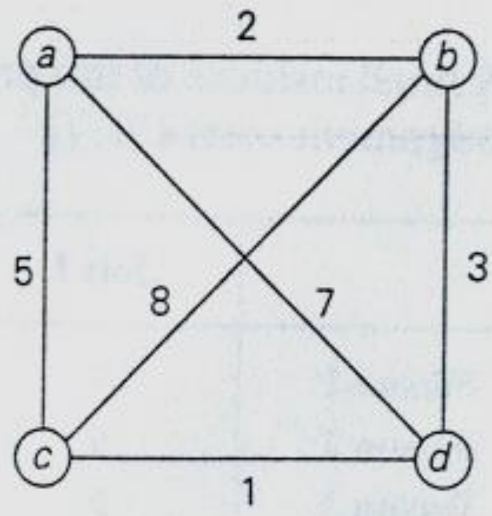
- The problem can be modeled by a weighted graph
- Vertices represents cities
- Edge weights represent the distances

- ***Exhaustive search*** can be used to solve the problem

Exhaustive Search

- Exhaustive search is a simple brute force approach to combinatorial problems
- Exhaustive search suggests
  - Generating each and every element of the problem’s domain
  - Selecting those of them that satisfy the problem constraints
  - Finding a desired element
    - Optimizes some objective function

Traveling Salesman Problem



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

A solution to a small instance of the traveling salesman problem by exhaustive search

Traveling Salesman Problem

- **Discussion :**
  - This problem has been intriguing for the last 150 years by
    - seemingly simple formulation,
    - important applications and
    - interesting connections to other combinatorial problems

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - Bubble Sort
  - Sequential Search
  - String Matching
  - Travelling Salesman Problem
  - **Knapsack Problem**
  - Assignment Problem
  - Depth-First Search
  - Breadth-First Search

Knapsack Problem

- **Problem Definition :**

  Given *n* items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity *W*.

  Find the most valuable subset of the items that fit into the knapsack

Knapsack Problem

- **Exhaustive Search Approach :**
  - Consider all the subsets of the set of n items given
  - Compute the total weight of each subset in order to identify feasible subsets
  - Find a subset of the largest value among them

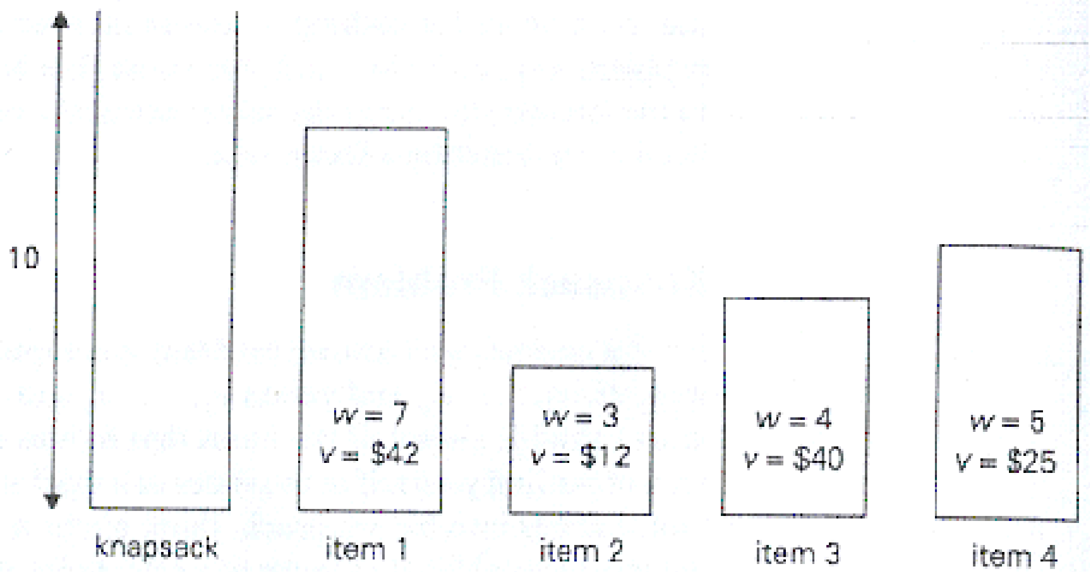
Knapsack Problem

- **Analysis:**

  # of subsets of an *n*-element set is 2^n

  So exhaustive search leads to a $\Omega(2^n)$ algorithm

Knapsack Problem



Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{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

Knapsack Problem

- **Discussion :**
  - Knapsack is a well known problem in algorithmics
  - Exhaustive search leads to algorithms that are extremely inefficient on every input
  - In fact knapsack is one of the examples of *NP-hard* problems
  - No polynomial-time algorithm is known for any NP-hard problem !

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - Bubble Sort
  - Sequential Search
  - String Matching
  - Travelling Salesman Problem
  - Knapsack Problem
  - **Assignment Problem**
  - Depth-First Search
  - Breadth-First Search

Assignment Problem

- **Problem Definition :**
  - There are *n* people who need to be assigned to execute *n* jobs, one person per job.
  - The cost of assigning the *i*th person to the *j*th job is a known quantity *C[i,j]* for each pair *i,j=1, … , n*.
  - Problem is to find an assignment with smallest total cost !

Assignment Problem

- **Exhaustive Search Approach :**
  - Assignment problem is completely specified by its cost matrix C [i,j]
  - Example

	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

Assignment Problem

- **Exhaustive Search Approach :**
  - Describe n-tuples $(j_1, \ldots, j_n)$ in which the ith component, $i=1, \ldots, n$, indicates the column of the element selected in the ith row
  - There is a one to one correspondence between feasible assignments and permutations of the first n integers

```
1. generate all permutations of integers
   1,2,…,n
2. compute the total cost of each assignment
   by summing up the corresponding elements of
   the cost matrix
3. select the one with smallest sum
```

Assignment Problem

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

$$\begin{array}{ll} \langle 1, 2, 3, 4 \rangle & \text{cost} = 9 + 4 + 1 + 4 = 18 \\ \langle 1, 2, 4, 3 \rangle & \text{cost} = 9 + 4 + 8 + 9 = 30 \\ \langle 1, 3, 2, 4 \rangle & \text{cost} = 9 + 3 + 8 + 4 = 24 \\ \langle 1, 3, 4, 2 \rangle & \text{cost} = 9 + 3 + 8 + 6 = 26 \\ \langle 1, 4, 2, 3 \rangle & \text{cost} = 9 + 7 + 8 + 9 = 33 \\ \langle 1, 4, 3, 2 \rangle & \text{cost} = 9 + 7 + 1 + 6 = 23 \end{array}$$

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

Assignment Problem

- **Discussion :**
  - The fact that the problem's domain grows exponentially (or faster) does not necessarily imply that there can be no efficient algorithm for solving it
  - We will discuss them later !

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - Bubble Sort
  - Sequential Search
  - String Matching
  - Travelling Salesman Problem
  - Knapsack Problem
  - Assignment Problem
  - **Depth-First Search**
  - Breadth-First Search

Depth-First Search (DFS)

- It is one of the principal algorithms used to make traversals on graphs
- It is useful in investigating several important properties of a graph
  - Connectivity
  - Acyclicity
- Based on decrease-by-one technique

Depth-First Search

- **Approach :**
  - Start from an arbitrary vertex, mark it as visited
  - On each iteration, proceed to an unvisited vertex that is adjacent to the current one
    - which of the adjacent unvisited candidates is choosen?
  - This process continues until a dead end
    - a vertex with no adjacent unvisited vertices
  - At a dead-end, the algorithm back up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there
  - Algorithm halts after backing up to the starting vertex (being a dead-end)
    - Then all vertices in the same connected component as the starting vertex have been visited
  - If unvisited vertices still remain, DFS must be restarted at any one of them

Depth-First Search

- It is convenient to use a ***stack*** for DFS
  - We push a vertex on to the stack when the vertex is reached for the first time
  - We pop a vertex off the stack when it becomes a dead end
- The following is recursive algorithm

Depth-First Search

ALGORITHM $DFS(G)$

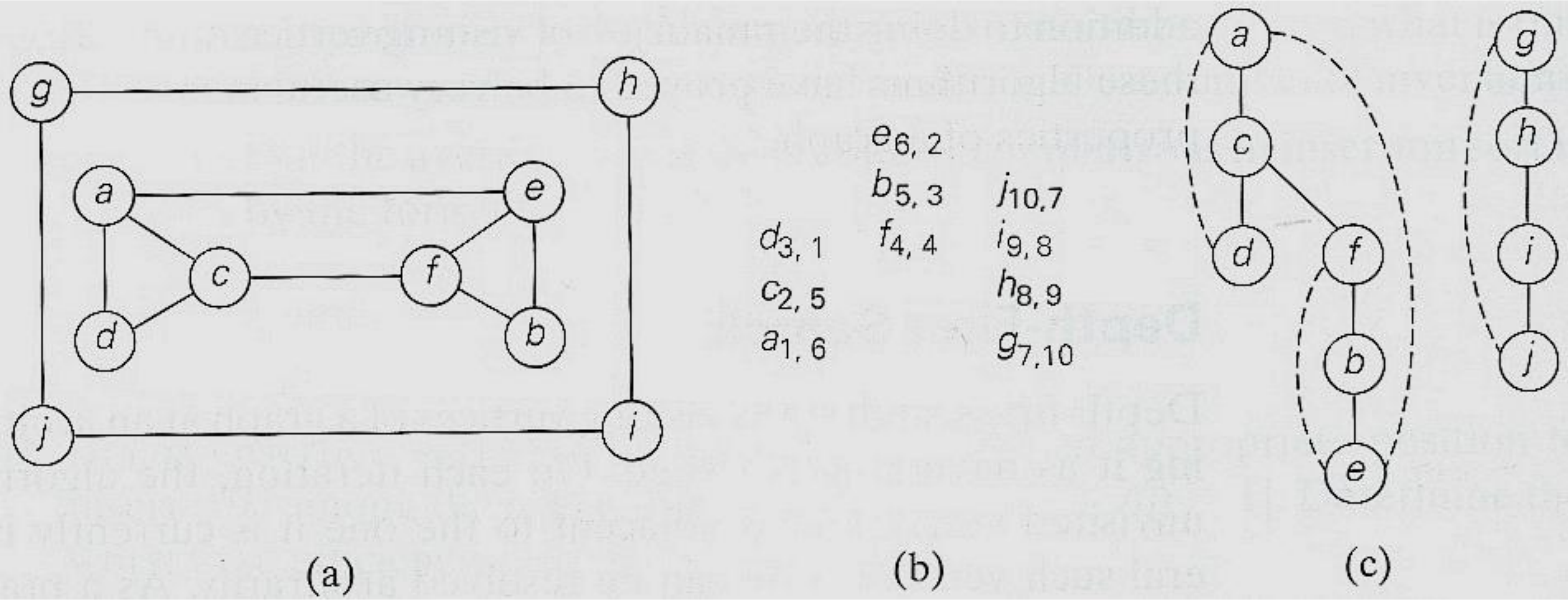
//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
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 and
//assigns them the numbers in the order they are encountered
//via global variable $count$
$count \leftarrow count + 1$; mark v with $count$
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        $dfs(w)$

Depth-First Search

- Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached
  - called *tree edge*
- The edge leading to a previously visited vertex other than its immediate predecessor
  - called *back edge*

Depth-First Search (DFS)



a – Example of DFS traversal
b – Traversal's stack
c – DFS forest
tree edges shown with solid lines and back edges shown with dashed lines

Depth-First Search (DFS)

- **Analysis :**

  How efficient is DFS ?

  Algorithm takes time proportional to the size of the data structure used for representing the graph

  - For adjacency matrix representation, time is in $\Theta(|V|^2)$
  - For adjacency list representation, time is in $\Theta(|V| + |E|)$

Depth-First Search (DFS)

- <u>**Discussion :**</u>
  - DFS is efficient to check important properties of graphs
  - Elementary applications of DFS
    - Checking connectivity
    - Checking acyclicity

ROAD MAP

- **Brute Force and Exhaustive Search**
  - Selection Sort
  - Bubble Sort
  - Sequential Search
  - String Matching
  - Travelling Salesman Problem
  - Knapsack Problem
  - Assignment Problem
  - Depth-First Search
  - **Breadth-First Search**

Breadth First Search (BFS)

- **Definition :**
  - It is the other principal algorithm used to make traversals on graphs
  - Again based on decrease-by-one method
  - If DFS is a traversal for the brave, BFS is a traversal for the cautious

Breadth First Search

- **<u>Approach :</u>**
  - Visit 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, algorithm has to be restarted at an arbitrary vertex of another connected component of the graph

Breadth First Search

- It is convenient to use a *queue*
  - different from DFS
- Queue is initialize with the traversal’s starting vertex which is marked as visited.
- On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex
- Marks them as visited and adds them to the queue
- After that the front vertex is removed fro the queue

ALGORITHM $BFS(G)$

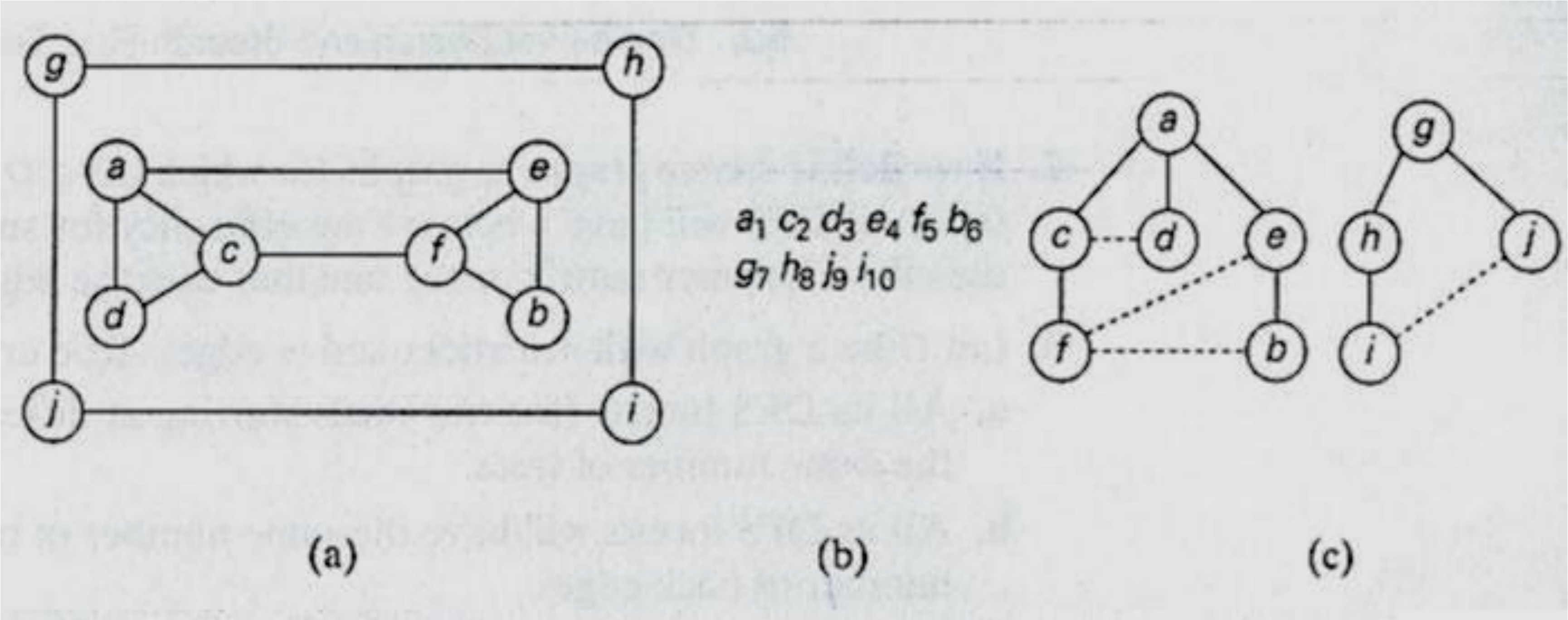
```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
      bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//and assigns them the numbers 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
```

Breadth First Search (BFS)

- Whenever a new unvisited vertex is reached for the first time, ithe vertex is attached as a child to the vertex is being reached from with an edge
  - called *tree edge*
- If an edge leading to a previously visited vertex other than its immediate predecessor is encountered, edge is
  - called *cross edge*

Breadth First Search



a – graph

b – traversal’s queue

c – BFS forest

Breadth First Search

- **Discussion :**
  - BFS can be used to check connectivity and acyclicity of a graph as DFS
  - It can be helpful in some situations where DFS can not
    - Finding a path with fewest number of edges between two given vertices

Main Facts About DFS and BFS

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 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 adjacent matrix	$\Theta(\lvert V^2\rvert)$	$\Theta(\lvert V^2\rvert)$
Efficiency for adjacent linked lists	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$