Algorithms and Their Applications
- Brute Force and Exhaustive Search -

Won-Yong Shin

April 6th, 2020

연세대학교
YONSEI UNIVERSITY

- **Brute force** search
  - A *straightforward* approach, usually based directly on the problem's statement and definitions of the concepts involved

- Examples:
  1. Computing a^n ($a > 0$, n a nonnegative integer)
  2. Computing $n!$
  3. Multiplying two matrices
  4. Searching for a key of a given value in a list

- Selection sort
  - Scan the array to find its *smallest element* and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i ($0 \le i \le n$-2), find the smallest element in $A[i..n\text{-}1]$ and swap it with $A[i]$:

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \;\Big|\; \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

  - Example:

$\mid$ 89	45	68	90	29	34	**17**
17 $\mid$	45	68	90	**29**	34	89
17	29 $\mid$	68	90	45	**34**	89
17	29	34 $\mid$	90	**45**	68	89
17	29	34	45 $\mid$	90	**68**	89
17	29	34	45	68 $\mid$	90	**89**
17	29	34	45	68	89 $\mid$	90

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//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
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]
```

- Basic operation: the key comparison $A[j] < A[min]$
- **Time** efficiency (the # of key comparisons):

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

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

$$= \boxed{\Theta(n^2)}$$

- Bubble sort

$$\boxed{A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}}$$

  - Example:

$$\begin{array}{ccccccccccccc}
89 & \overset{?}{\leftrightarrow} & 45 & & 68 & & 90 & & 29 & & 34 & & 17 \\
45 & & 89 & \overset{?}{\leftrightarrow} & 68 & & 90 & & 29 & & 34 & & 17 \\
45 & & 68 & & 89 & \overset{?}{\leftrightarrow} & 90 & \overset{?}{\leftrightarrow} & 29 & & 34 & & 17 \\
45 & & 68 & & 89 & & 29 & & 90 & \overset{?}{\leftrightarrow} & 34 & & 17 \\
45 & & 68 & & 89 & & 29 & & 34 & & 90 & \overset{?}{\leftrightarrow} & 17 \\
45 & & 68 & & 89 & & 29 & & 34 & & 17 & | & 90 \\
 & & & & & & & & & & & & \\
45 & \overset{?}{\leftrightarrow} & 68 & \overset{?}{\leftrightarrow} & 89 & \overset{?}{\leftrightarrow} & 29 & & 34 & & 17 & | & 90 \\
45 & & 68 & & 29 & & 89 & \overset{?}{\leftrightarrow} & 34 & & 17 & | & 90 \\
45 & & 68 & & 29 & & 34 & & 89 & \overset{?}{\leftrightarrow} & 17 & | & 90 \\
45 & & 68 & & 29 & & 34 & & 17 & | & 89 & & 90
\end{array}$$

etc.

  - Time efficiency:

$$C(n) = \boxed{\Theta(n^2)}$$

- String matching
  - *pattern*: a string of m characters to search for
  - *text*: a (longer) string of n characters to search in
  - Problem: find a substring in the text that matches the pattern

- Brute-force algorithm
  - **Step 1:** Align pattern at beginning of text
  - **Step 2:** Moving *from left to right*, compare each character of pattern to the corresponding character in text until
    - All characters are found to match (successful search); or
    - A mismatch is detected
  - **Step 3:** While pattern is not found and the text is not yet exhausted, *realign pattern one position to the right* and repeat Step 2

Examples of Brute-Force String Matching

- Example 1
  - Pattern: 001011
  - Text: 10010101101001100101111010

- Example 2

```
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
```

```
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 ← 0 to n − m do
        j ← 0
        while j < m and P[j] = T[i + j] do
            j ← j + 1
        if j = m return i
    return −1
```

- Efficiency:
  - $m(n\text{-}m\text{+}1)$ character comparisons → $O(nm)$ class
  - Linear in n → $\boxed{\Theta(n)}$

- Problem:
  - Find the value of polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 \quad \text{at a point } x = x_0$$

- <u>Brute-force algorithm</u>

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
        for j ← 1 to i do              //compute x^i
            power ← power * x
        p ← p + a[i] * power
return p
```

- Basic operation: the # of multiplications
- Efficiency: $M(n) = \sum_{i=0}^{n} \sum_{j=1}^{i} 1 \in \boxed{\Theta(n^2)}$

- Improvement
  - We can do better by evaluating *from right to left*

- <u>Better brute-force algorithm</u>

```
p ← a[0]
power ← 1
for i ← 1 to n do
        power ← power * x
        p ← p + a[i] * power
return p
```

- Efficiency:

$$M(n) = \sum_{i=1}^{n} 2 = 2n$$

- Closest-pair problem by brute force
  - Find the two closest points in a set of n points (in the two-dimensional Cartesian plane)

- Brute-force algorithm
  - Compute the distance between *every pair* of distinct points and return the indexes of the points for which the distance is the smallest

Analysis of Closest-Pair Brute-Force Algorithm

ALGORITHM *BruteForceClosestPoints*(P)

//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices *index*1 and *index*2 of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //*sqrt* is the square root function
    if $d < dmin$
      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return *index*1, *index*2

- Basic operation: *squaring* a number
- Efficiency:

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

$$= (n-1)n \in \boxed{\Theta(n^2)}$$

- How to make it faster?

Brute Force Strengths and Weaknesses

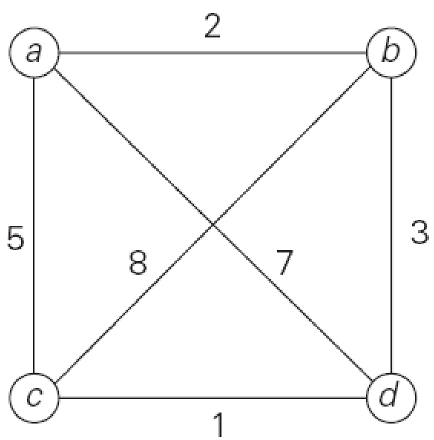
- Strengths
  - Wide applicability
  - Simplicity
  - Yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)
- Weaknesses
  - Rarely yields efficient algorithms
  - Some brute-force algorithms are unacceptably slow
  - Not as constructive as some other design techniques

- Exhaustive search
  - A brute force solution to a problem involving search for an element with a special property, usually among *combinatorial* objects such as permutations, combinations, or subsets of a set

Example 1: Traveling Salesman Problem

- Traveling salesman problem (TSP)
  - Given n cities with known distances between each pair, find the shortest tour that passes through all the cities *exactly once* before returning to the starting city

- Example:



Tour	**Cost**
a→b→c→d→a	2+8+1+7 = 18
a→b→d→c→a	2+3+1+5 = 11
a→c→b→d→a	5+8+3+7 = 23
a→c→d→b→a	5+1+3+2 = 11
a→d→b→c→a	7+3+8+5 = 23
a→d→c→b→a	7+1+8+2 = 18

  - Efficiency

$$\frac{1}{2}(n-1)! = \boxed{\Theta((n-1)!)}$$

Example 2: Knapsack Problem

- Given n items:
  - **weights**: $w_1 \ w_2 \ \ldots \ w_n$
  - **values**: $v_1 \ v_2 \ \ldots \ v_n$
  - A knapsack of **capacity** W

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

- Example: $W = 16$

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

- Solution

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

- Efficiency

$\boxed{\Theta(2^n)}$

Example 3: Assignment Problem

- The assignment problem
  - There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is **C[i,j]**
  - Find an assignment that minimizes the total cost

- Algorithmic plan: generate all legitimate assignments, compute their costs, and select the cheapest one
- How many assignments are there?

- 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 by Exhaustive Search

- Solution

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

Assignment (column indice)	**Total Cost**
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
	etc.

- Efficiency

$\boxed{\Theta(n!)}$

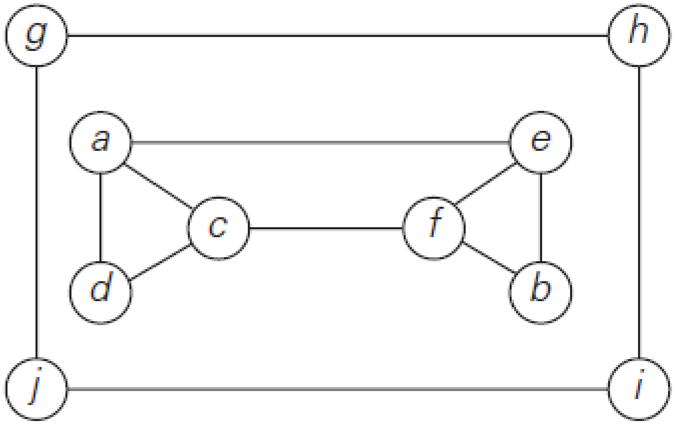
Final Comments on Exhaustive Search

- Exhaustive-search algorithms
  - run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  - Euler circuits
  - Shortest paths
  - Minimum spanning tree
  - Assignment problem
- In many cases,
  - Exhaustive search or its variation is the only known way to get an exact solution

Depth-First Search (DFS)

- DFS
  - Visit graph's vertices by always *moving away from last visited vertex to unvisited one*
  - Backtrack if no adjacent unvisited vertex is available
- Uses a **stack**
  - A vertex is pushed onto the stack when it's reached for the first time
  - A vertex is popped off the stack when it becomes a *dead end*, i.e., when there is no adjacent unvisited vertex
- "Redraw" graph in tree-like fashion
  - With **tree edges** and **back edges** for *undirected* graph

Example: DFS Traversal of Undirected Graph



Graph

$e_{6,2}$

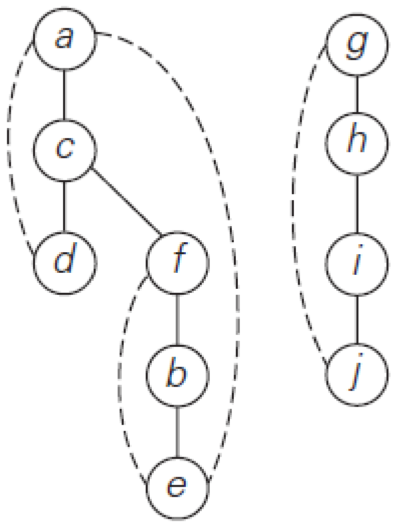
$b_{5,3}$ $j_{10,7}$

$d_{3,1}$ $f_{4,4}$ $i_{9,8}$

$c_{2,5}$ $h_{8,9}$

$a_{1,6}$ $g_{7,10}$

Traversal's stack

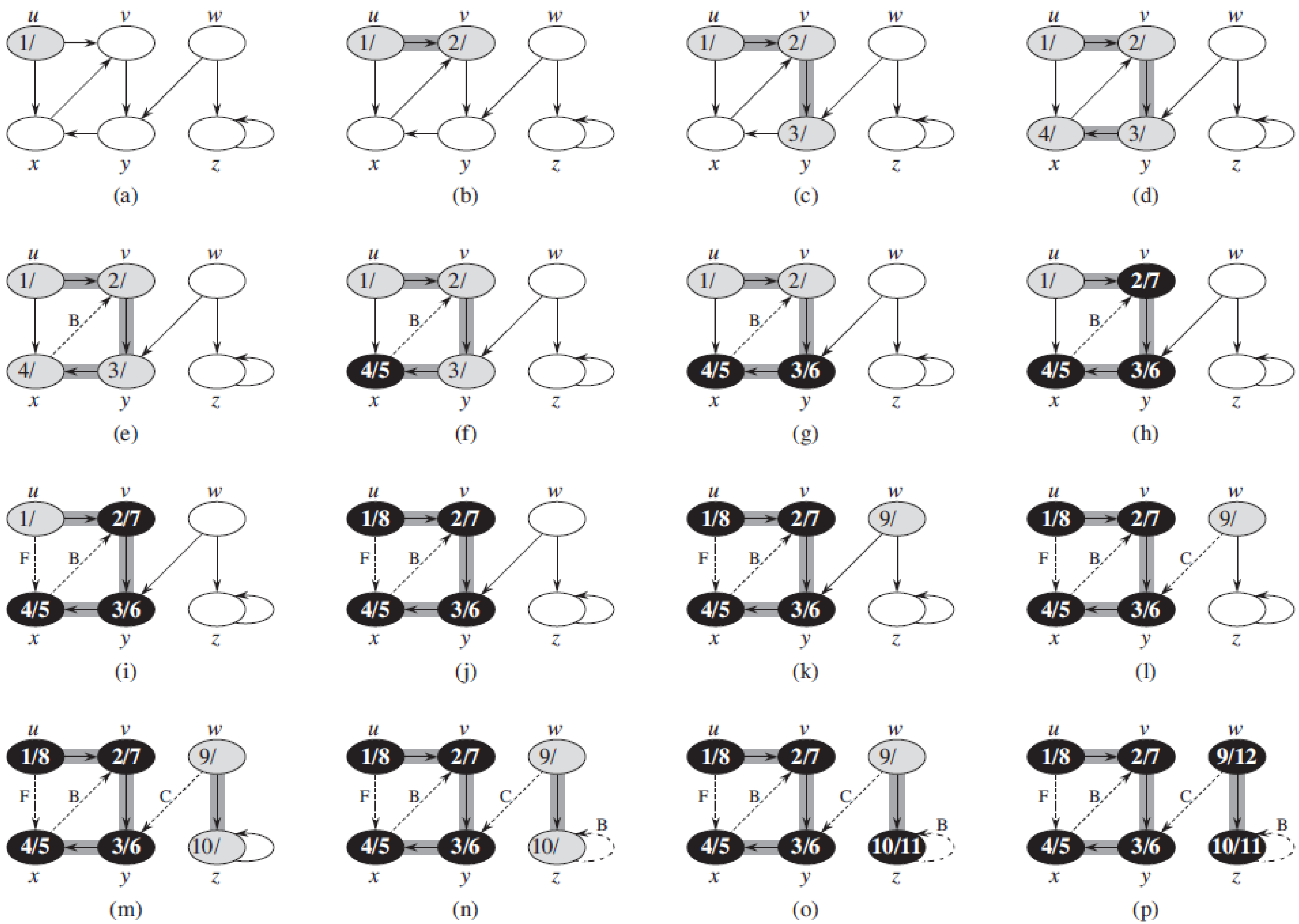


DFS forest with the tree and back edges

Classification of Edges (in Directed Graphs)

- Tree edges
  - Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v)
- Back edges
  - Edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree
- Forward edges
  - Nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree
- Cross edges
  - All other edges

Progress of the DFS Algorithm on a Directed Graph

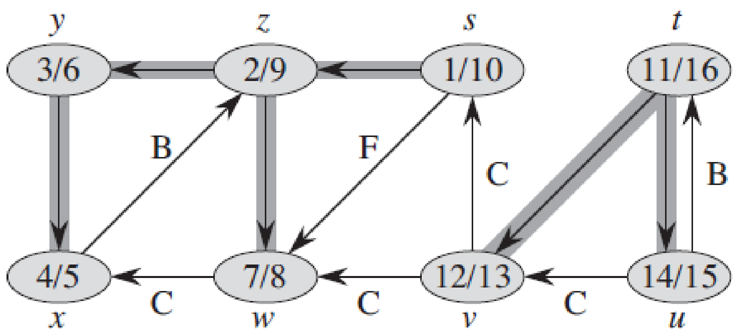


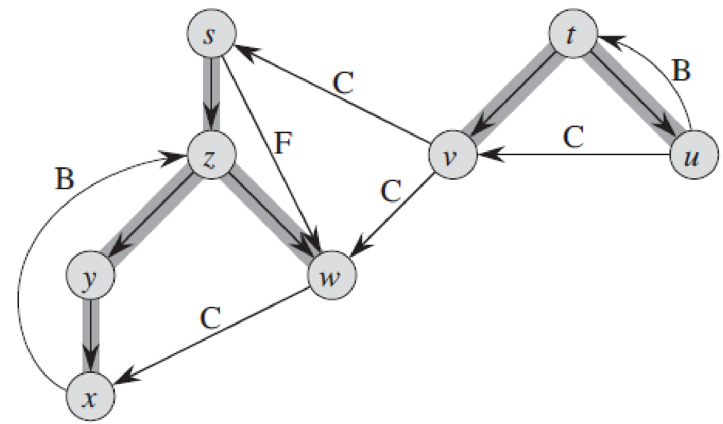
ALGORITHM $DFS(G)$

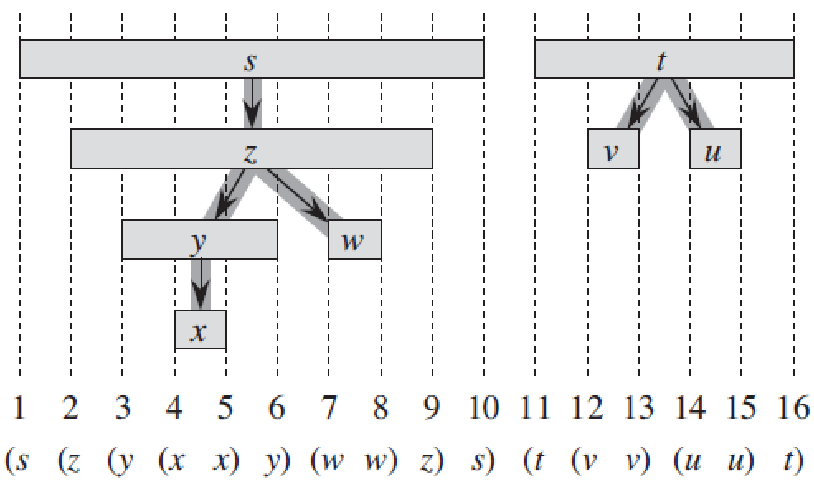
```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for 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)
```

Properties of Depth-First Search



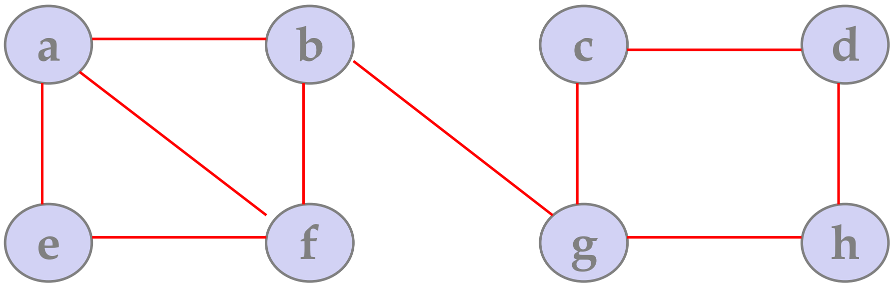




Refer to Parenthesis theorem

Another Example

- Undirected graph



  - DFS traversal stack
  - DFS tree

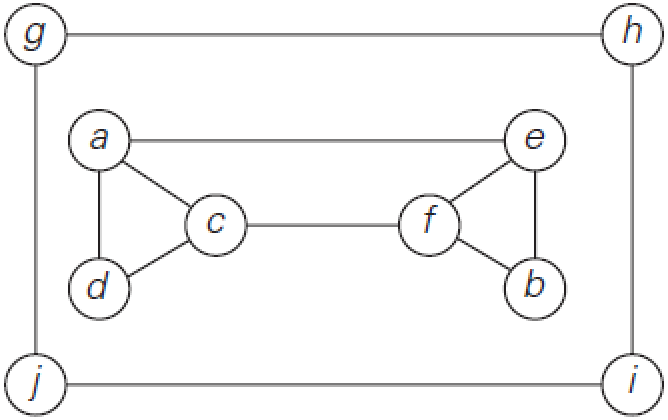
- DFS can be implemented with graphs represented as:
  - Adjacency matrices
  - Adjacency lists

- Yields two distinct ordering of vertices
  - Order in which vertices are first encountered (pushed onto stack)
  - Order in which vertices become *dead-ends* (popped off stack)

- Applications
  - Checking connectivity, finding connected components
  - Checking acyclicity
    - Whether there is a *back edge* from some vertex to its ancestor

- BFS
  - Visit graph vertices by *moving across to all the neighbors of last visited vertex*
- Instead of a stack, BFS uses a **queue**
- Similar to level-by-level tree traversal
- “Redraw” graph in tree-like fashion
  - With **tree edges** and **cross edges** for *undirected* graph

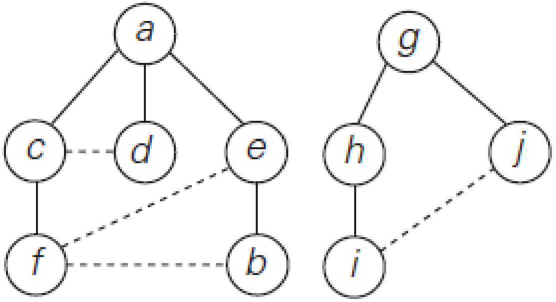
Example: BFS Traversal of Undirected Graph



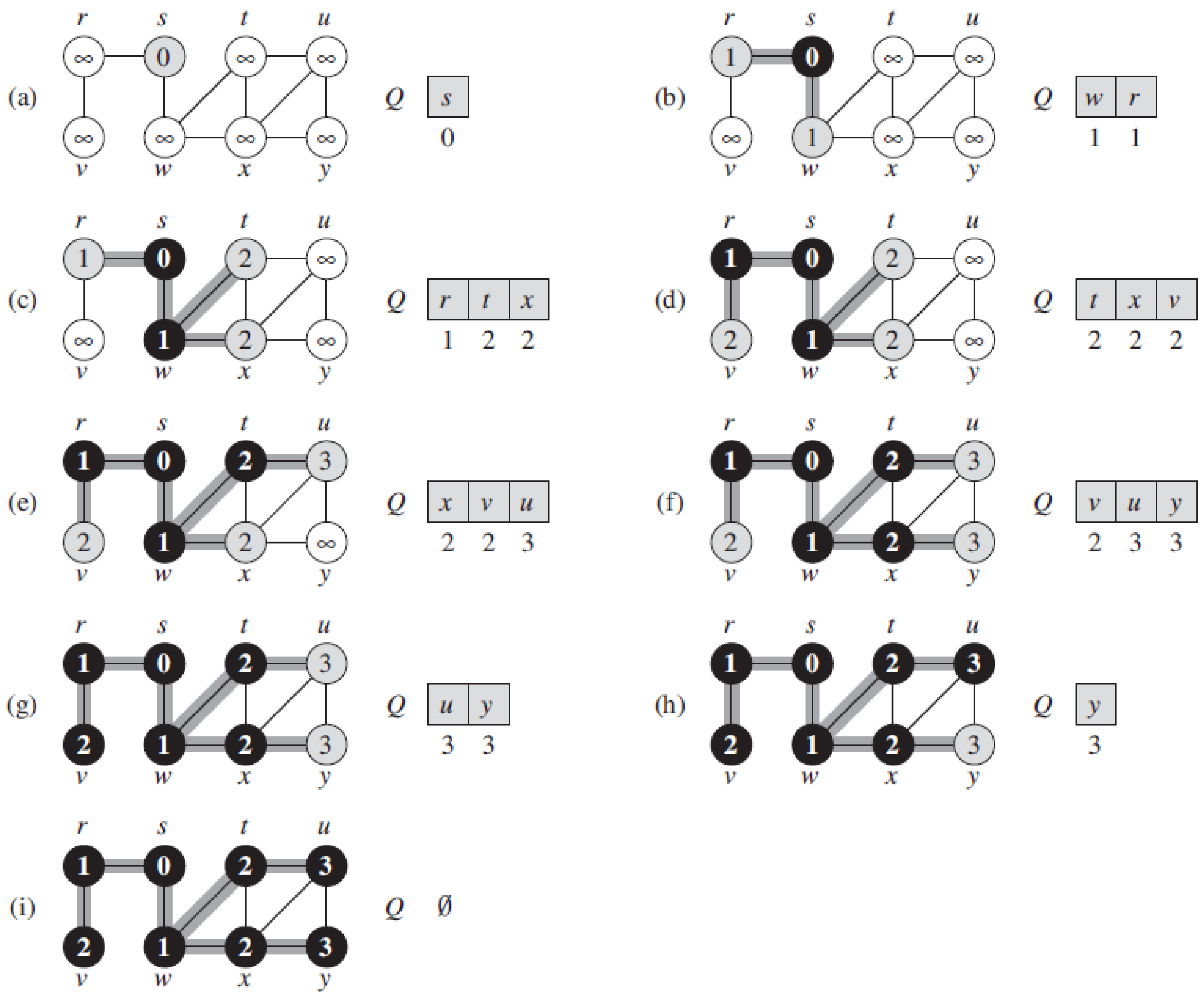
Graph

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

Traversal's queue



BFS forest with the
tree and cross edges



(a)
r s t u
v w x y
Q s
0
(b)
Q w r
1 1
(c)
Q r t x
1 2 2
(d)
Q t x v
2 2 2
(e)
Q x v u
2 2 3
(f)
Q v u y
2 3 3
(g)
Q u y
3 3
(h)
Q y
3
(i)
Q ∅

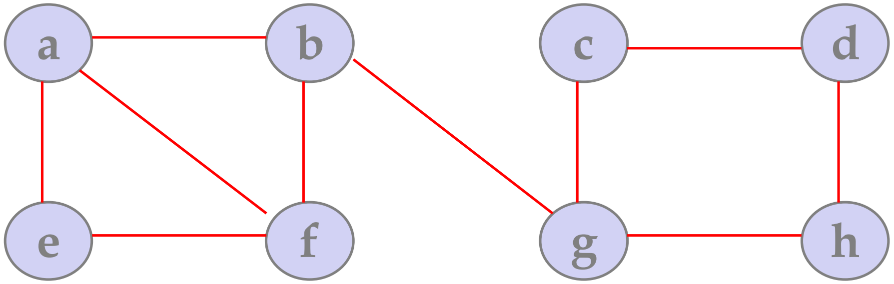
Pseudocode of BFS

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 by a path
//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
```

- Undirected graph



  - BFS traversal queue
  - BFS tree

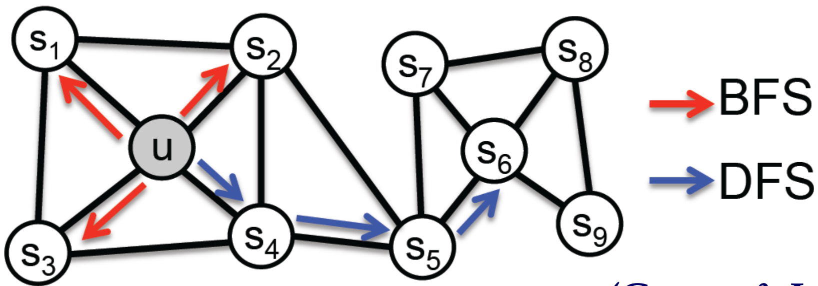
Notes on BFS

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - Adjacency matrices
  - Adjacency lists

- Yields *single* ordering of vertices
  - (order added to/deleted from queue is the same)

- Applications
  - Same as DFS

- Application of BFS and DFS on social media



(Grover & Leskovec KDD'16)

- BFS: sampling immediate neighbors of the source *u*
- DFS: sampling nodes sequentially at increasing distances from *u*