CSC 411
Design and Analysis of Algorithms

Chapter 3 Brute Force and Exhaustive Search - Part 2

Instructor: Minhee Jun

junm@cua.edu

Brute Force algorithms we have seen

- Selection Sort
- Bubble Sort
- Sequential Search
- String Matching
- Closest-Pair Problem
- Convex Hall Problem

What we will see next

- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem

- Graph Traversal
  - *Depth First Search*
  - *Breadth First Search*

3.4 Exhaustive Search

- Exhaustive search: simply 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.

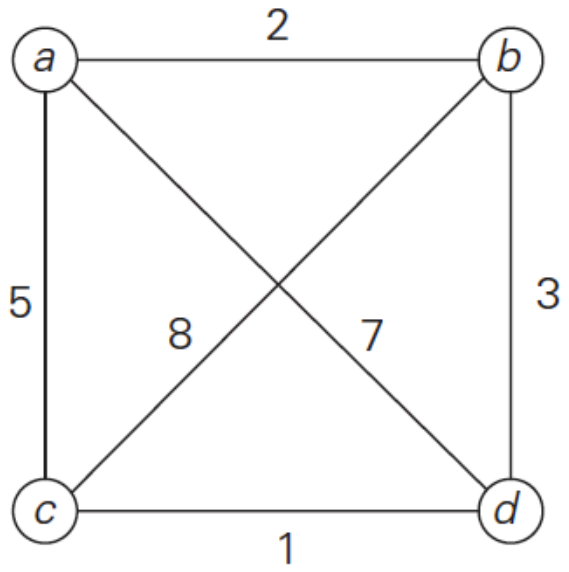
  1. Generate each and every element of the problem domain
  2. Selecting those of them that satisfy all the constraints, and then finding a desired element.

     e.g., the on that optimizes some objective function.

Example 1: Traveling Salesman Problem

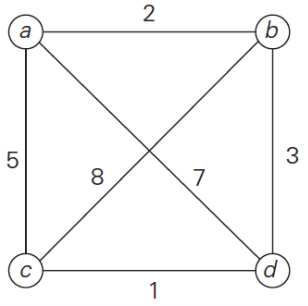
- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
  - Conveniently modeled by a weighted graph

- Example:



Traveling Salesman Problem (TSP)

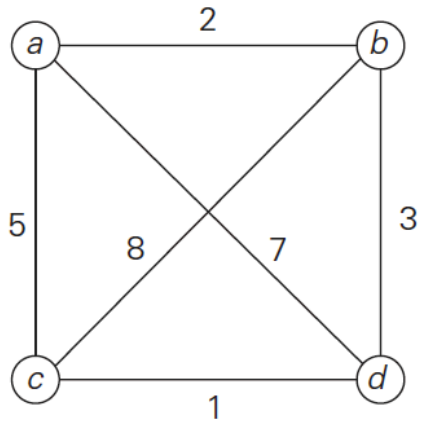
- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
  - Conveniently modeled by a weighted graph
  - i.e., Find the shortest Hamiltonian circuit of the graph (a cycle that passes through all the vertices of the graph exactly once) of the graph
  - i.e., a sequence of n+1 adjacent vertices $v_{i_0}, v_{i_1}, , \cdots, v_{i_{n-1}}, v_{i_0}$ where the first vertex of the sequence is the same as the last one and all the other n-1 vertices are distinct.
    - Assume that all circuits start and end at one particular vertex.



TSP by Exhaustive Search

Assume start from the verity ‘a’,

which one is the best?

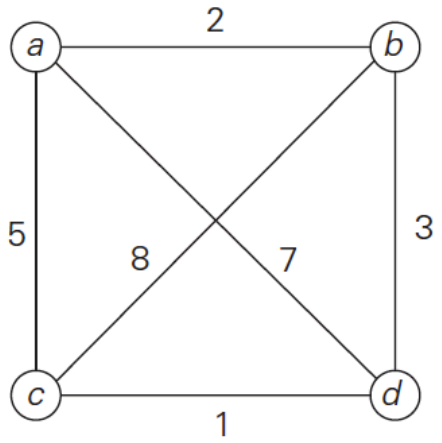


Tour	Length

TSP by Exhaustive Search

Assume start from the verity ‘a’,

which one is the best?



Efficiency: $\Theta(n!)$

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

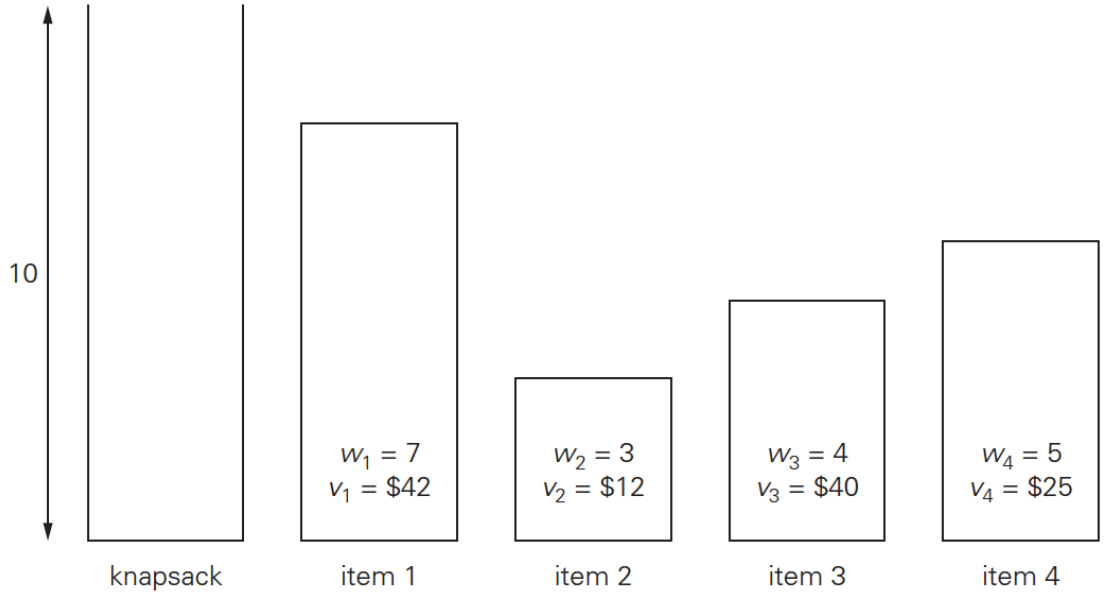
Traveling Salesman Problem (TSP)

- TSP is easy enough to explain, but it is very difficult to solve.
  - Efficiency: $O(n!)$
  - Exhaustive-search approach impractical for all but very small values of n

- Practical (real world) application for TSP
  - Vehicle routing problem
  - Genome sequencing
  - Manufacturing
  - Plane routing
  - Telephone routing
  - Job sequencing

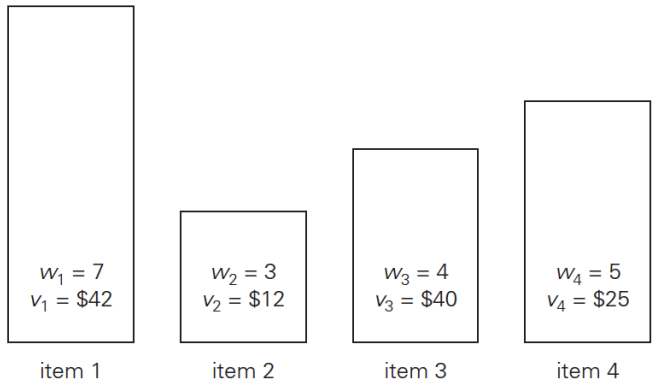
Example 2: Knapsack Problem

- Given *n* items (input):
  - weights: w_1 w_2 … w_n
  - values: v_1 v_2 … v_n
  - a knapsack of capacity *W*
- Find the most valuable subset of the items that fit into the knapsack
- Example: Knapsack capacity W=10



Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
$\varnothing$	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



Efficiency: $\Theta(2^n)$

What is the entire available space?

- Example: create a password
  - alphabetic lowercase password of 6 characters..
  - that's 26^6 = 308915776 combinations
  - brute forcing at 10,000 a second = 8.5 hours maximum

  - alphanumeric lowercase password of 6 characters..
  - that's 36^6 = 2176782336 combinations
  - brute forcing at 10,000 a second = 2.5 days maximum

  - alphabetic lowercase password of 8 characters..
  - that's 26^8 = 208827064576 combinations
  - brute forcing at 10,000 a second = 241.6 days maximum

NP-hard problem

- The exhaustive search are extremely inefficient for both the traveling sales man and knapsack problems
  - The best-known examples of NP-hard problem
- NP-hard problem
  - Non-deterministic polynomial-time hardness in computational complexity
  - A class of problems that are informally "at least as hard as the hardest problems in NP"
  - Most computer scientists believe that such algorithms do not exist, although this very important conjecture has never been proven

Example 3: 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

	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

- Algorithmic Plan:
  - Generate all legitimate assignments,
    compute their costs, and select the cheapest one.
  - The assignment problem can be specified by its cost matrix C

  1. Select one element / each row of C in different columns
     - We cannot select the smallest element in each row, because the smallest elements may happen to be in the same column
  2. Find the smallest total sum from sets of the selected elements in 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

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

Assignment Problem by Exhaustive Search

- Algorithmic Plan:
  1. The assignment problem can be specified by its cost matrix C
     - Select one element / each row of C in different columns
  2. Find the smallest total sum from sets of the selected elements in C

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

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

One person per job

Efficiency: $\Theta(n!)$

Exercise 3.4

4. Complete the application of exhaustive search to the instance of the assignment problem started in the text.

Exercise 3.4

9. *Eight-queens problem* Consider the classic puzzle of placing eight queens on an 8×8 chessboard so that no two queens are in the same row or in the same column or on the same diagonal. How many different positions are there so that

a. no two queens are on the same square?

b. no two queens are in the same row?

c. no two queens are in the same row or in the same column?

Also estimate how long it would take to find all the solutions to the problem by exhaustive search based on each of these approaches on a computer capable of checking 10 billion positions per second.

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 exact solution

Brute Force algorithms we have seen

- Selection Sort
- Bubble Sort
- Sequential Search
- String Matching
- Closest-Pair Problem
- Convex Hall Problem

What we will see next

- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem

- Graph Traversal
  - *Depth First Search*
  - *Breadth First Search*

3.4 Exhaustive Search

- Exhaustive search: simply 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.

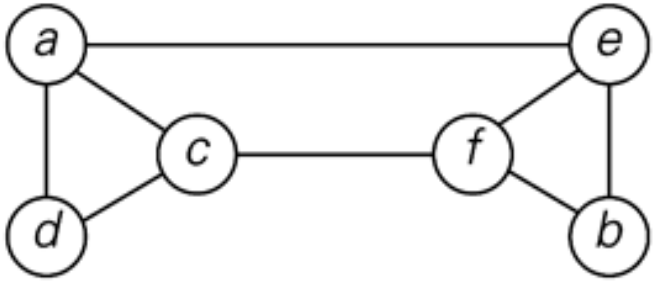
  1. Generate each and every element of the problem domain
  2. Selecting those of them that satisfy all the constraints, and then finding a desired element.

     e.g., the on that optimizes some objective function.

3.5 Graph Traversal: Depth-First Search and Breadth First Search

- Graph traversal algorithms:
  systematically process all vertices (and/or edges) of a graph

  - Depth-first search (DFS)
  - Breadth-first search (BFS)



  - Useful for many applications involving graphs in AI and operations research. For efficient investigation of fundamental properties of graphs such as connectivity and cycle presence.

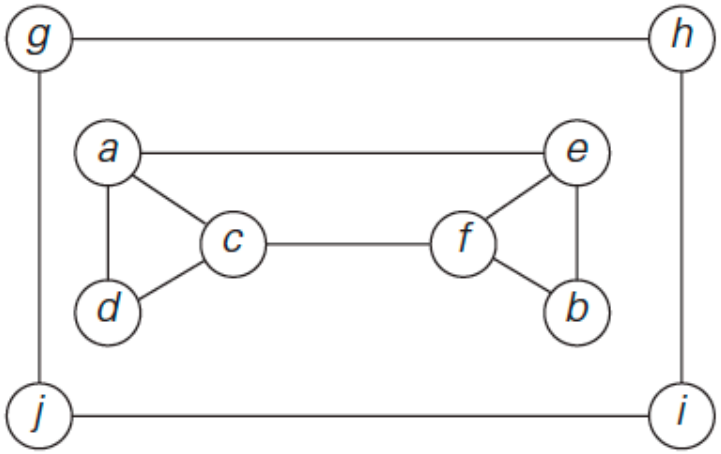
- Q: how we represent a graph? what kind of data structure?
  - Recall: adjacency matrix, adjacency list

Depth-First Search (DFS)

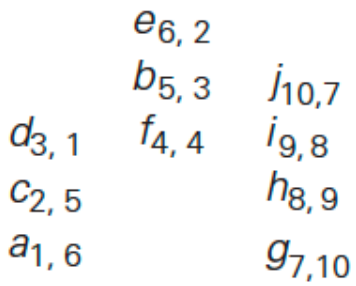
- Starts a graph's traversal at an arbitrary vertex by marking it as visited
- On each iteration, the algorithm process to an unvisited vertex that is adjacent to the one it is currently in.
  - If there are several such vertices, arbitrarily choose one.
  - This process continues until a dead end (no adjacent unvisited vertices)
  - At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.
- The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end.

Example: DFS traversal of undirected graph

- To trace the operation of DFS, use a <u>stack</u>
  - When a vertex is reached for the first time, push it onto the stack
  - When it becomes a dead end, pop a vertex off the stack



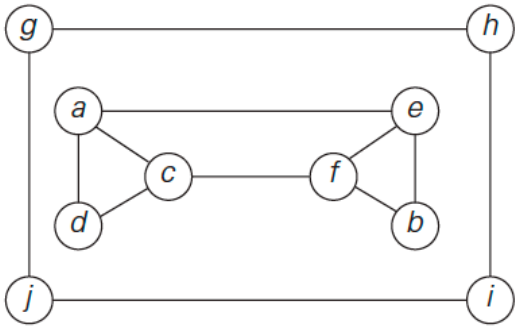
(a) Graph



(b) Traversal's stack

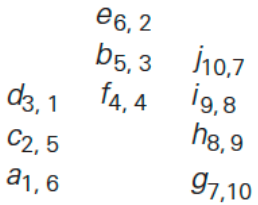
Example: DFS traversal of undirected graph

- Constructing DFS forest
  - If a vertex is reached for the first time, attach it as a child to the previous vertex with a tree edge
  - Otherwise, it is connected with a back edge



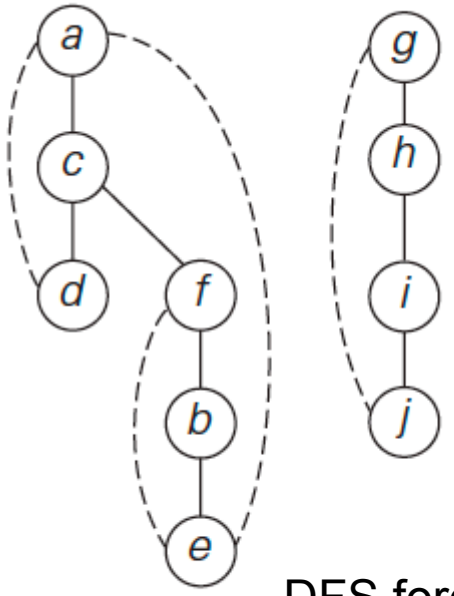
(a)

Graph



(b)

Traversal's stack



(c)

DFS forest
:Tree and back edges

Pseudocode of DFS

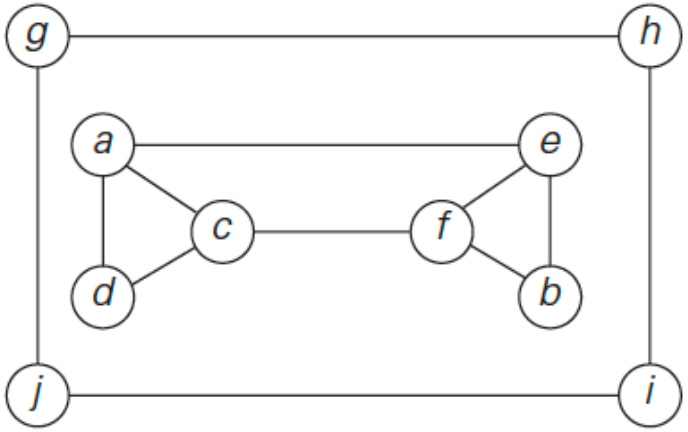
ALGORITHM $DFS(G)$

```
//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)
```

How efficient is DFS?

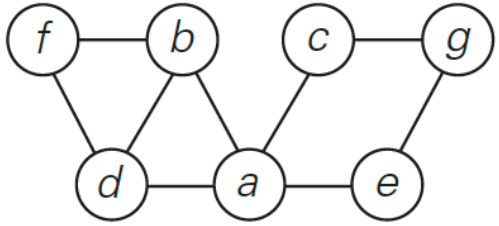
- Proportional to the size of the data structure used for representing the graph
- DFS can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V| + |E|)$



- Important elementary Applications of DFS:
  - checking connectivity, finding connected components
  - checking acyclicity
  - finding articulation points and biconnected components

Exercise 3.5

1. Consider the following graph.



a. Write down the adjacency matrix and adjacency lists specifying this graph. (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.)

b. Starting at vertex a and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).

Exercise 3.5

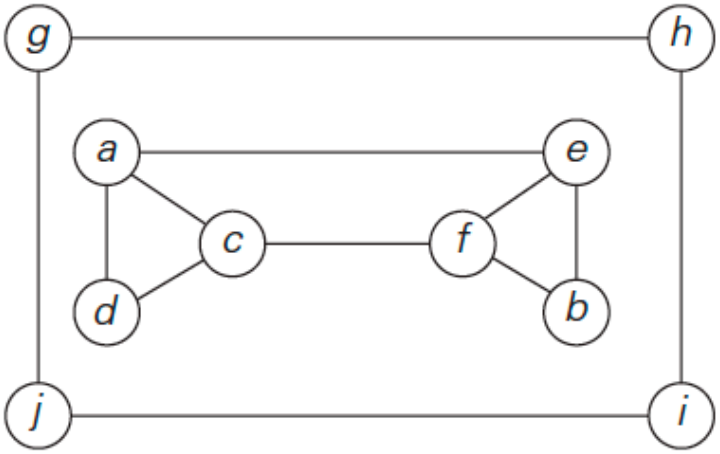
2. If we define sparse graphs as graphs for which $|E| \in O(|V|)$, which implementation of DFS will have a better time efficiency for such graphs, the one that uses the adjacency matrix or the one that uses the adjacency lists?

Breadth-first search (BFS)

- Visit first all the vertices that are adjacent to a starting vertex,
  - then all unvisited vertices two edges apart from it,
  - until all vertices in the same connected component.
  - If there still remain unvisited vertices,
    the algorithm has to be restarted at an arbitrary vertex
    for another connected component of the graph.

Example of BFS traversal of undirected graph

- to trace the operation of BFS, use a queue
  - Initialized with the starting vertex, which is marked as visited
  - On each iteration, identify all unvisited adjacent vertices, marks them as visited, and adds them to the queue
  - After that, the vertex is removed from the queue.

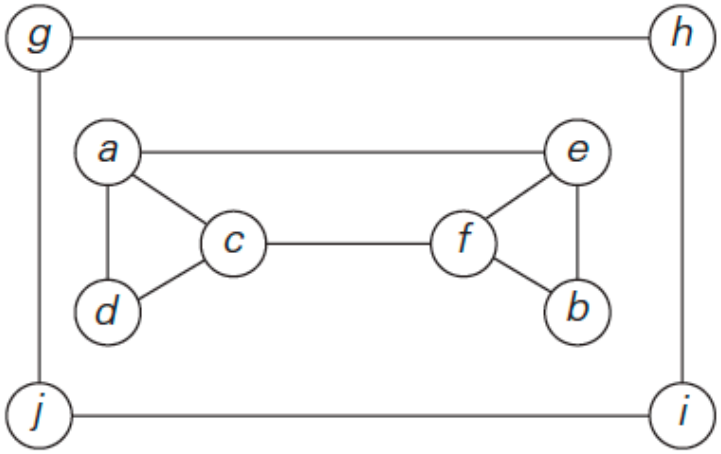


$a_1\ c_2\ d_3\ e_4\ f_5\ b_6$
$g_7\ h_8\ j_9\ i_{10}$

(a) Graph (b) Queue

Example of BFS traversal of undirected graph

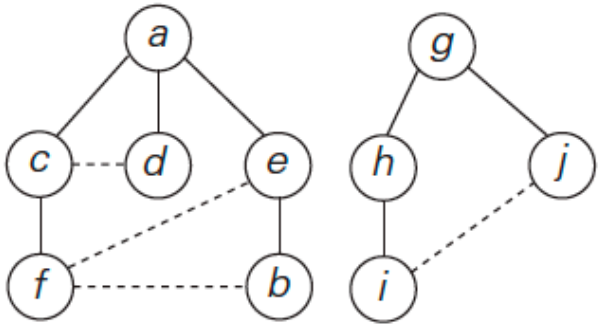
- Constructing BFS forest
  - A starting vertex as the foot of the first tree in such a forest
  - Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the previous vertex with tree edge.
  - If an edge leading to a previously visited vertex, the edge is a cross edge..



(a) Graph

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

(b) Queue



(c): BFS forest with tree/cross edges

Pseudocode of BFS

ALGORITHM *BFS*(G)

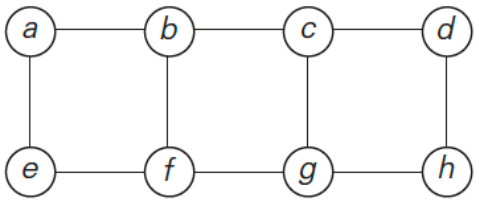
```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each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

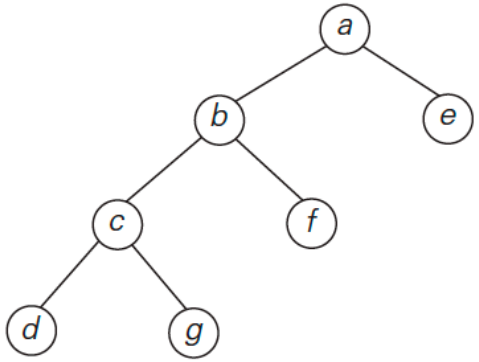
How efficient is BFS?

- BFS has same efficiency as DFS
- implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V| + |E|)$

- Important elementary applications of BFS:
  - Check connectivity and acyclicity of a graph.
  - Also find a path with the fewest number of edges between two given vertices



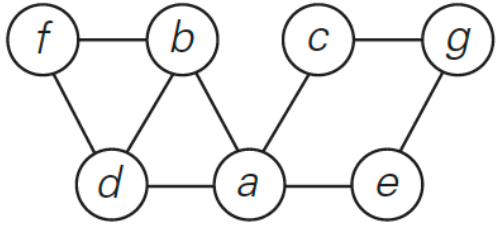
(a)



(b)

Exercise 3.5

1. Consider the following graph.



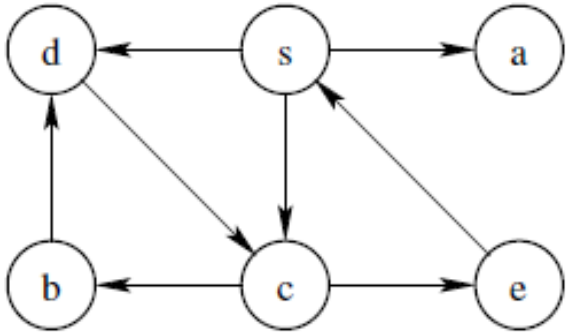
4. Traverse the graph of Problem 1 by breadth-first search and construct the corresponding breadth-first search tree. Start the traversal at vertex *a* and resolve ties by the vertex alphabetical order.

Example: DFS/BFS

- Find the visited node order for each type of graph search, starting with node s
  - Write the adjacency matrix for the graph
  - Write the adjacency linked list for the graph

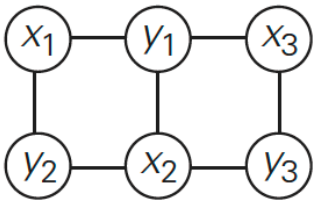
- Depth First Search
  - Solution: s a c b d e

- Breadth First Search
  - Solution: s a c d b e

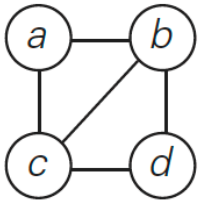


Exercise 3.5

8. A graph is said to be ***bipartite*** if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. (One can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called ***2-colorable***.) For example, graph (i) is bipartite while graph (ii) is not.



(i)



(ii)

a. Design a DFS-based algorithm for checking whether a graph is bipartite.

b. Design a BFS-based algorithm for checking whether a graph is bipartite.

Exercise 3.5

10. One can model a maze by having a vertex for a starting point, a finishing point, dead ends, and all the points in the maze where more than one path can be taken, and then connecting the vertices according to the paths in the maze.

a. Construct such a graph for the following maze.

b. Which traversal—DFS or BFS—would you use if you found yourself in a maze and why?

Exercise 3.5

11. *Three Jugs* Siméon Denis Poisson (1781–1840), a famous French mathematician and physicist, is said to have become interested in mathematics after encountering some version of the following old puzzle. Given an 8-pint jug full of water and two empty jugs of 5- and 3-pint capacity, get exactly 4 pints of water in one of the jugs by completely filling up and/or emptying jugs into others. Solve this puzzle by using breadth-first search.

Main Facts about DFS vs 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	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Brute Force algorithms we have learned in chapter 3

- Selection Sort
- String Matching
- Closest-Pair Problem
- Convex-Hull Problem
- Traveling Salesman Problem
- Knapsack Problem
- The Assignment Problem
- Graph Traversal
  - *Depth First Search*
  - *Breadth First Search*

Summary: Brute Force Algorithms

- Brute Force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved

- In many cases, Brute Force does not provide you a very efficient solution

- Brute Force may be enough for small or moderate size problems with current computers