Computer Algorithms

Lecture 3: Brute Force &Exhaustive Search – Ch 3

Dr. Manal Helal, Spring 2014.

http://moodle.manalhelal.com

Lecture Learning Objectives

1. Use a Brute Force algorithm design strategy to solve an appropriate problem such as sorting, searching, string matching, closest-pair, and/or convex-hull.

Brute Force

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) $a^n = \underbrace{a * \cdots * a}_{n \text{ times}}$
2. Computing $n!$
3. Multiplying two matrices
4. Searching for a key of a given value in a list

Brute-Force Sorting Algorithm

<u>*Selection Sort*</u> 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 $A[i..n-1]$ and swap it with $A[i]$:

$$A[0] \le \; . \; . \; . \le A[i-1] \;\; | \;\; A[i], \; . \; . \; . \; , A[min], \; . \; . \; ., A[n-1]$$

in their final positions

Example: 7 3 2 5

Analysis of Selection Sort

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 nondecreasing 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]
```

Time efficiency:

Space efficiency:

Stability:

Bubble Sort

Self-Study

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

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Sequential Search Improvement

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//          or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//          equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Brute-Force 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

Find i—the index of the leftmost character of the first matching substring in the text, such that:

$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$.

Pattern: 001011
Text: 10010101101001100101111010

Pattern: happy
Text: It is never too late to have a happy childhood.

Pseudocode and Efficiency

```
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:

Self-Study

Brute-Force Polynomial Evaluation

Problem: Find the value of polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$$

at a point $x = x_0$

Brute-force algorithm

```
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
```

Efficiency:

Polynomial Evaluation: Improvement

Self-Study

We can do better by evaluating from right to left:

Better brute-force algorithm

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

Efficiency:

Closest-Pair Problem

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.

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPair*(P)

//Finds distance between two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
return d

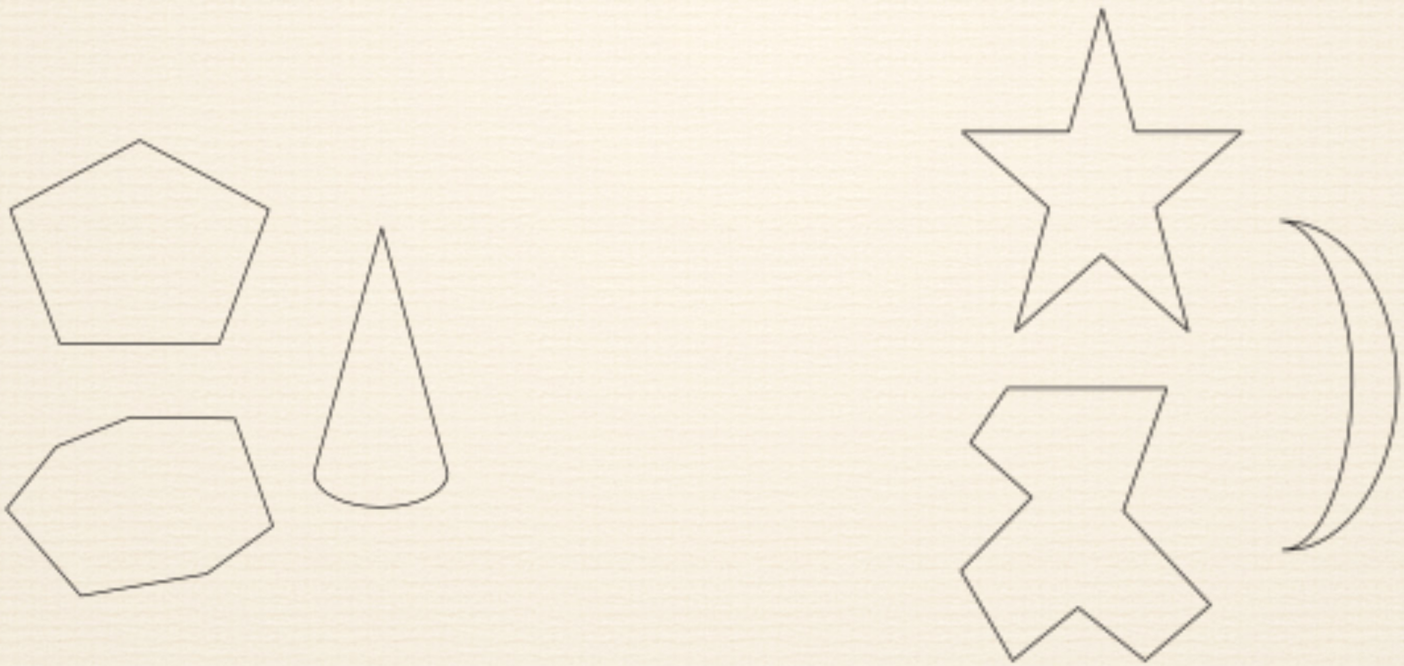
$$(x_i - x_j)^2 + (y_i - y_j)^2$$

Efficiency:

How to make it faster?

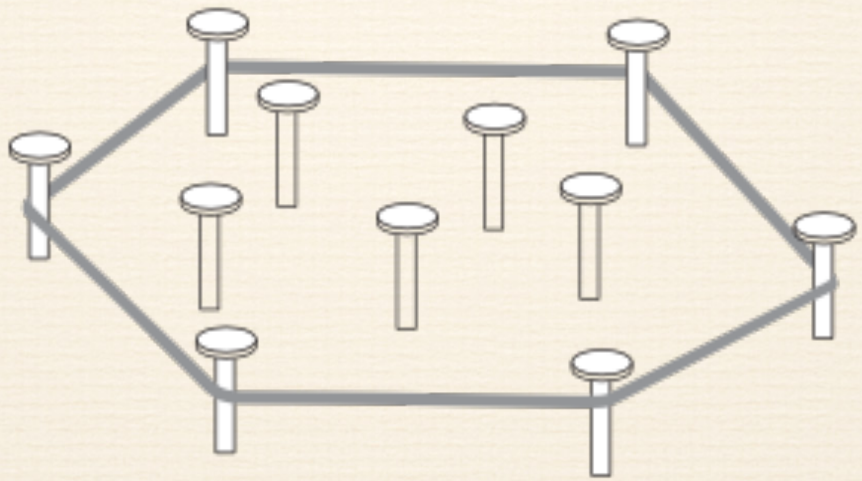
Convex Hull Problem

DEFINITION A set of points (finite or infinite) in the plane is called ***convex*** if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.



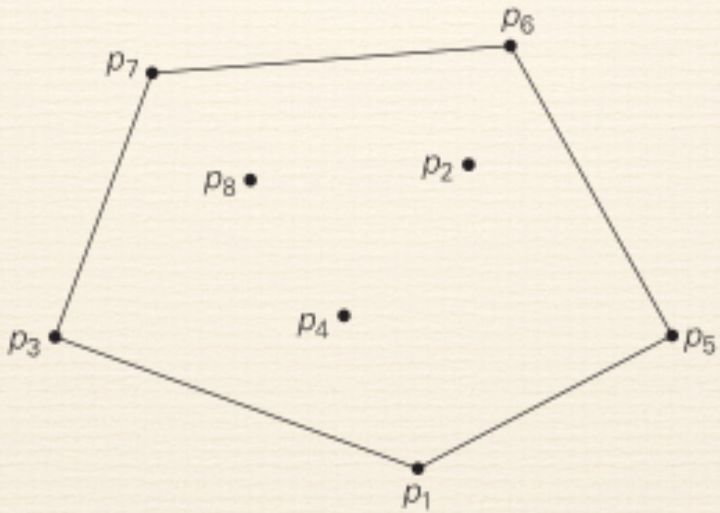
Convex Hull Problem – Cont'd

DEFINITION The ***convex hull*** of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.)



Convex Hull Problem – Cont'd

DEFINITION The ***convex hull*** of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.)



Convex Hull Brute Force Solution Plan

- Notice that a line segment connecting two points p_i and p_j of a set of n points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points:

1. a straight line through two points (x_1, y_1), (x_2, y_2) in the coordinate plane can be defined by the equation : $ax + by = c$, where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - y_1x_2$.
2. a lines cuts the plane in 2 halves: $ax + by > c$ for all pionts in one half, and $ax + by < c$ for the other. Choose the points with line equation of the same sign

Brute-Force Strengths and Weaknesses

<u>Strengths</u>

- wide applicability
- simplicity
- yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

<u>Weaknesses</u>

- 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.

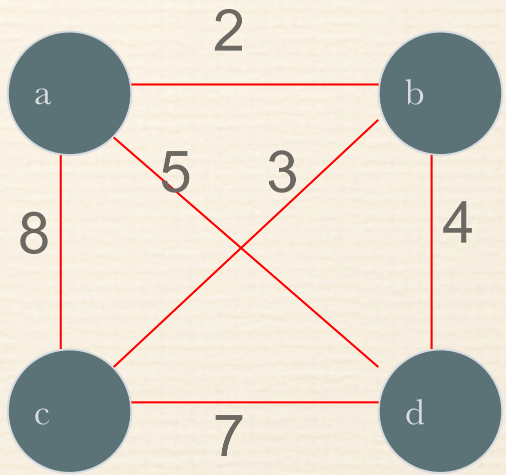
Method:

- generate a list of all potential solutions to the problem in a systematic manner (see algorithms in Sec. 5.4)

- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far when search ends, announce the solution(s) found

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
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph
- Example:



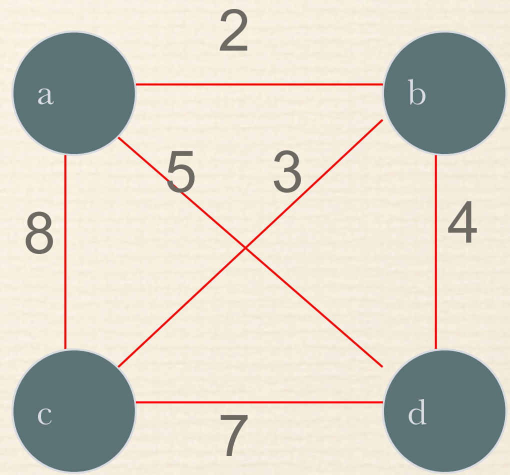
TSP by Exhaustive Search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

More tours?

Less tours?

Efficiency:



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: Knapsack capacity W=16

item	weight	value
•	2	$20
•	5	$30
•	10	$50
•	5	$10

Knapsack Problem by Exhaustive Search

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:

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

Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.
How many assignments are there?
Pose the problem as the one about a cost matrix:

Assignment Problem by Exhaustive Search

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

Assignment (col.#s)	Total Cost
<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

etc.

etc.

(For this particular instance, the optimal assignment can be found by exploiting the specific features of the number given. It is:)

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 exact solution

Graph Traversal Algorithms

Many problems require processing all graph vertices (and edges) in systematic fashion

Graph traversal algorithms:

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

Depth-First Search (DFS)

Visits graph's vertices by always moving away from last visited vertex to unvisited one, backtracks 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 "Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph)

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)
```

Pseudocode of DFS

Pre-Order/Depth-first

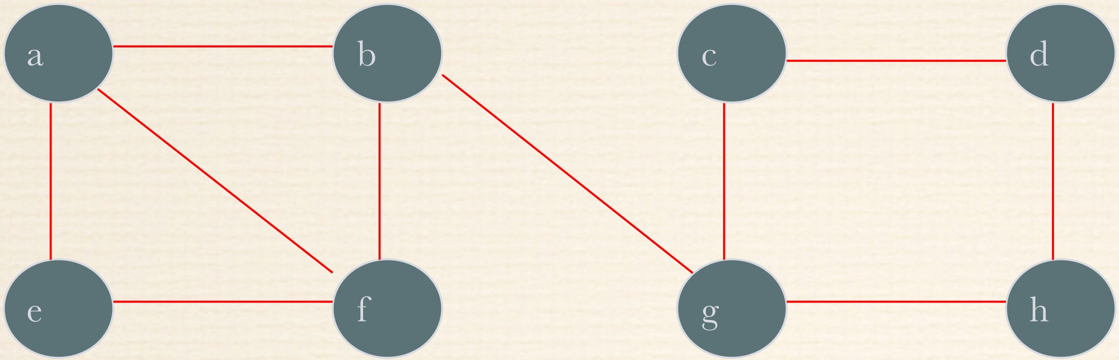
Algorithm 3.9: Pre-order traversal.

Input : An ordered tree T on $n > 0$ vertices.

Output: A list of the vertices of T in pre-order.

```
L ← []
S ← empty stack
r ← root of T
push(S, r)
while length(S) > 0 do
    v ← pop(S)
    append(L, v)
    [u_1, u_2, ..., u_k] ← ordering of children of v
    for i ← k, k − 1, ..., 1 do
        push(S, u_i)
return L
```

Example: DFS traversal of undirected graph



DFS traversal stack:

DFS tree:

Notes on DFS

- DFS can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$
- 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 finding articulation points and biconnected components searching state-space of problems for solution (AI)

Breadth-first search (BFS)

- Visits 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
- "Redraws" graph in tree-like fashion (with tree edges and cross edges for undirected graph)

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
```

Pseudocode of BFS

Level-Order/breadth-first / top- down traversal

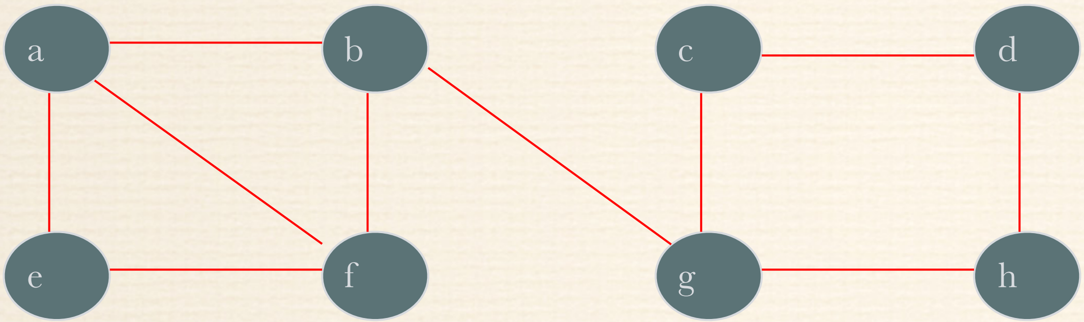
Algorithm 3.8: Level-order traversal.

Input : An ordered tree T on $n > 0$ vertices.
Output: A list of the vertices of T in level-order.

$L \leftarrow [\,]$
$Q \leftarrow$ empty queue
$r \leftarrow$ root of T
enqueue(Q, r)
while length$(Q) > 0$ **do**
    $v \leftarrow$ dequeue(Q)
    append(L, v)
    $[u_1, u_2, \dots, u_k] \leftarrow$ ordering of children of v
    for $i \leftarrow 1, 2, \dots, k$ **do**
        enqueue(Q, u_i)
return L

Example of BFS traversal of 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: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$
- Yields single ordering of vertices (order added/deleted from queue is the same)
- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges