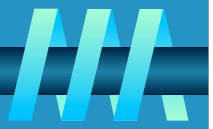
Chapter 3

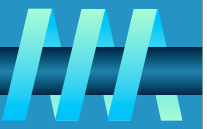
Brute Force and Exhaustive Search

Brute-Force Strengths and Weaknesses



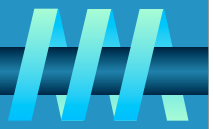
- **Strengths**
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)
- **Weaknesses**
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

Chapter 3:



- 3.1 selection sort and bubble sort
- 3.2 sequential search and brute-force string matching
- 3.3 closest-pair by brute force
- 3.4 exhaustive search
- 3.5 depth-first search and breadth-first search

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)

   $a^n = a*a*a*...*a$ {n times)

2. Computing $n!$

   $n! = 1*2*3*4*5*...*n$ {from 1to n}

3. Multiplying two matrices

   two nested loops...

4. Searching for a key of a given value in a list

   sequential search through list (ordered or unordered list)

Brute-Force Sorting Algorithm: Selection Sort

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 $A[i..n-1]$ and swap it with $A[i]$:

$$A[0] \le \ldots \le A[i-1] \mid A[i], \ldots, A[min], \ldots, A[n-1]$$

in their final positions

Example:

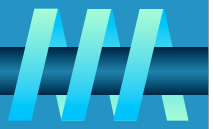
7	3	2	5
2	3	7	5
2	3	7	5
2	3	5	7
2	3	5	7

start with first element, find smallest to right

swap, start with 1 smaller list, do again...

(each time the working list is shorter by one)

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 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, comparison

Time efficiency: $\Theta(n^2)$ (Θ → grows at same rate as n^2)

Space efficiency: (considered an "in-place" sort algorithm)

Brute-Force Sorting Algorithm: Bubble Sort

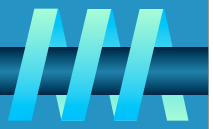
Bubble Sort Compare adjacent elements of the list and exchange them if they are out of order. Do this repeatedly and will "bubble up" the largest element to the last position on the list. Next pass does the second largest element, etc..., until after n-1 passes the list is sorted.

Pass i (0<=i<=n-2) of bubble sort can be represented by the following:

$$A_0,\ldots,A_j \leftarrow^{?}\rightarrow A_{j+i},\ldots,A_{n-i-1} \mid A_{n-i}<=\ldots<=A_{n-1}$$

as the comparison takes place | in their final positions

Analysis of Bubble Sort (1)



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 $A[j+1]$ ← basic operation, comparison

Time efficiency: $\Theta(n^2)$ (Θ → grows at same rate as n^2)

Space efficiency: (considered an "in-space" sort algorithm)

Analysis of Bubble Sort (2)

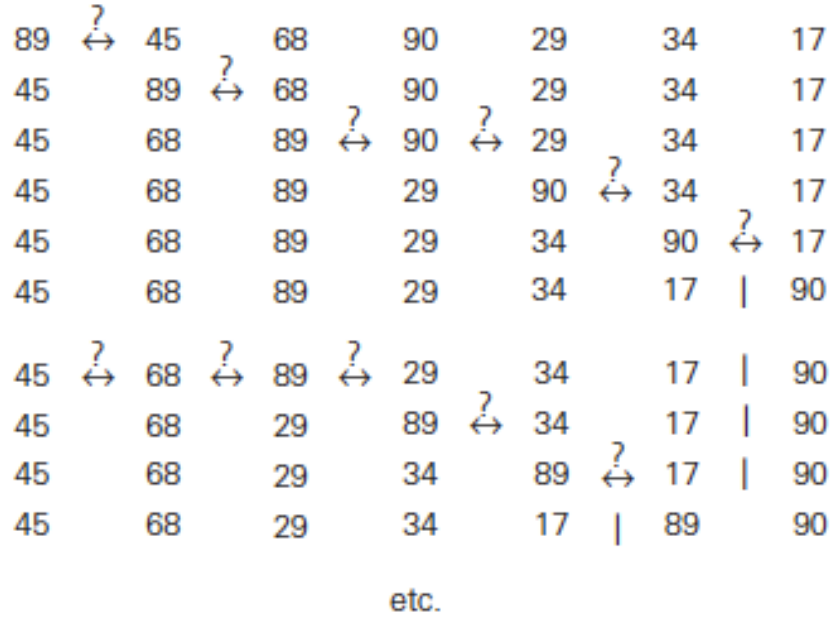
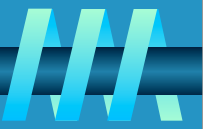


FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

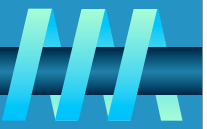
- ***NOTE: Simple improvement to standard bubble sort*: If no exchanges made during a pass through the list -- list is sorted -- so stop.**
- **While this may improve on "best" case performance, "average" and "worst" case performance remains the same as original.**

Chapter 3:



- 3.1 selection sort and bubble sort
- 3.2 sequential search and brute-force string matching
- 3.3 closest-pair by brute force
- 3.4 exhaustive search
- 3.5 depth-first search and breadth-first search

Brute Force: Sequential Search

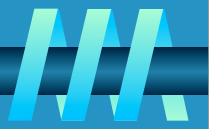


Sequential **Search Compare successive elements of a given list against a search key until match is found or run out of elements.**

Complexity: Θ(*n*) (Θ → grows at same rate as n)

This algorithm is simple and may be easily improved...

Brute Force: Sequential Search



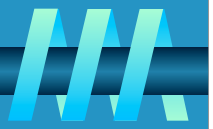
Trick/Tip: Append search key at end of list so that will always "match" at end and no longer need to test for "end of list" condition.

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     ← put search key at end of list (sentinel)
i ← 0
while A[i] ≠ K do
        i ← i + 1     ← only exits loop when finds K match
if i < n return i     ← returns index into array where match is found
else return −1        ← returns -1 to indicate match NOT found
```

Note: if finds K in last element of list, that is not really a match – so indicate failure.

Brute Force: Sequential Search



Trick/Tip: If know that list is sorted (increasing order here), may stop search if match or if list element is greater than search key.

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

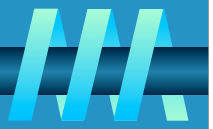
```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i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

Assuming that A[] is in increasing order

← if A[i] > K return -1 as will not be on list...

Best case will be better, no difference for average or worst case performance.

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

Pattern: 001011

Text: 10010101101001100101111010

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

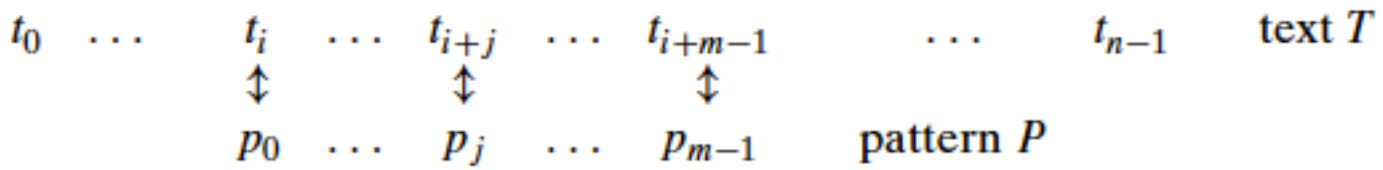
$$\begin{array}{ccccccccccc} t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & \ldots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & \\ & & p_0 & \ldots & p_j & \ldots & p_{m-1} & \text{pattern } P & & \end{array}$$

Examples of Brute-Force String Matching

Pattern: 001011

Text: 10010101101001100101111010

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:



Examples of Brute-Force String Matching

Pattern: 001011

Text: 10010101101001100101111010

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

t_0	$\ldots$	t_i	$\ldots$	t_{i+j}	$\ldots$	t_{i+m-1}	$\ldots$	t_{n-1}	text T
		$\updownarrow$		$\updownarrow$		$\updownarrow$			
		p_0	$\ldots$	p_j	$\ldots$	p_{m-1}	pattern P		

Examples of Brute-Force String Matching

Pattern: 001011

Text: 10010101101001100101111010

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$$\begin{array}{ccccccccccc} t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & \ldots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & \\ & & p_0 & \ldots & p_j & \ldots & p_{m-1} & \text{pattern } P & & \end{array}$$

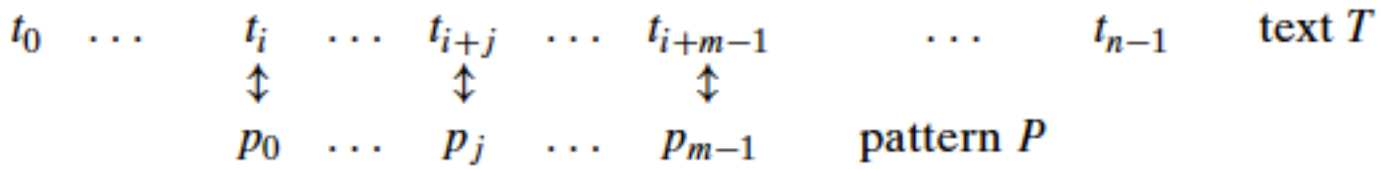
Examples of Brute-Force String Matching

Pattern: 001011

Text: 10010101101001100101111010

continues until find match...

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:



Examples of Brute-Force String Matching

Pattern: happy

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

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

t_0	$\ldots$	t_i	$\ldots$	t_{i+j}	$\ldots$	t_{i+m-1}	$\ldots$	t_{n-1}	text T
		$\updownarrow$		$\updownarrow$		$\updownarrow$			
		p_0	$\ldots$	p_j	$\ldots$	p_{m-1}	pattern P		

Examples of Brute-Force String Matching

Pattern: happy

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

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

t_0	$\ldots$	t_i	$\ldots$	t_{i+j}	$\ldots$	t_{i+m-1}	$\ldots$	t_{n-1}	text T
		$\updownarrow$		$\updownarrow$		$\updownarrow$			
		p_0	$\ldots$	p_j	$\ldots$	p_{m-1}	pattern P		

Examples of Brute-Force String Matching

Pattern: happy

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

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

t_0	$\ldots$	t_i	$\ldots$	t_{i+j}	$\ldots$	t_{i+m-1}	$\ldots$	t_{n-1}	text T
		$\updownarrow$		$\updownarrow$		$\updownarrow$			
		p_0	$\ldots$	p_j	$\ldots$	p_{m-1}	pattern P		

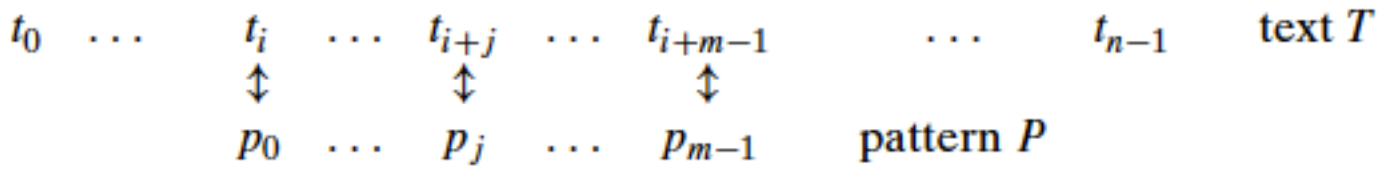
Examples of Brute-Force String Matching

Pattern: happy

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

continues until find match...

want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:



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
```

← walks up starting point in T[] for comparison string

← loop walks through pattern comparing to subset of text

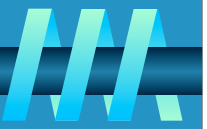
← if make it through pattern, we have match, return index i

Efficiency:

worst case: M(n-m+1) character comparisons O(nm)

average case: Θ(*n*) (Θ → grows at same rate as n)

Chapter 3:



- 3.1 selection sort and bubble sort
- 3.2 sequential search and brute-force string matching
- 3.3 closest-pair by brute force
- 3.4 exhaustive search
- 3.5 depth-first search and breadth-first search

Closest-Pair Problem

Find the two closest points in a set of *n* points (in the two-dimensional Cartesian plane).

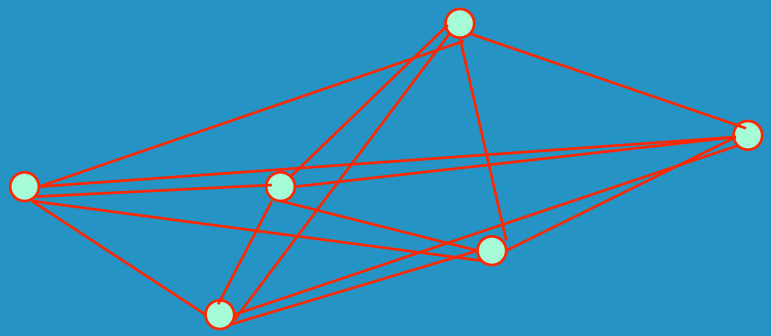
<u>Brute-force algorithm</u>

Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.

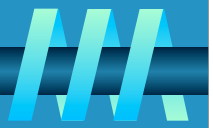
related to networking...

how many links in a fully connected network

n(n-1)/2



Closest-Pair Brute-Force Algorithm (cont.)



ALGORITHM *BruteForceClosestPair*(P)

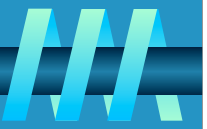
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n $(n \geq 2)$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i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

Efficiency: $\Theta(n^2)$ ($\Theta \rightarrow$ grows at same rate as n^2)

How to make it faster?

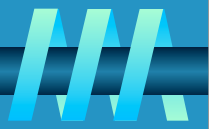
- **Simply compare the $((x_i-x_j)^2+(y_i-y_j)^2)$ values directly without square root as: smaller number you take square root of, the smaller the square root... (smaller produces smaller so direct comparison is possible).**
- **Note that this however does not change the efficiency class of the algorithm (but it is still faster...)**

Chapter 3:



- 3.1 selection sort and bubble sort
- 3.2 sequential search and brute-force string matching
- 3.3 closest-pair by brute force
- 3.4 exhaustive search
- 3.5 depth-first search and breadth-first search

Exhaustive Search



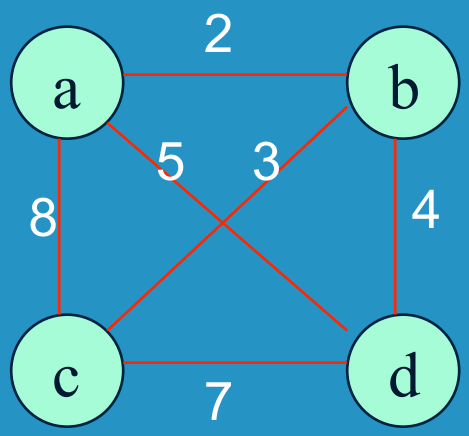
A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

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

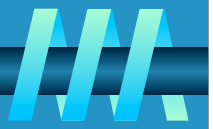
Example 1: Traveling Salesman Problem

- **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:**



Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

TSP by Exhaustive Search



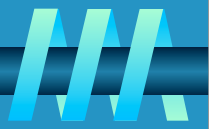
Tour		Cost	
a→b→c→d→a	(1)	2+3+7+5 = 17	optimal
a→b→d→c→a	(2)	2+4+7+8 = 21	
a→c→b→d→a	(3)	8+3+4+5 = 20	
a→c→d→b→a	(2)	8+7+4+2 = 21	
a→d→b→c→a	(3)	5+4+3+8 = 20	
a→d→c→b→a	(1)	5+7+3+2 = 17	optimal

Total tours? (n-1)! permutations

Less tours? half are simply reversed order (cuts search space in half...)

Efficiency: ½(n-1)! not very efficient...
(thus exhaustive search not a good solution for large n values)

Example 2: Knapsack Problem



Given n items:

- weights: w_1 w_2 … w_n
- values: v_1 v_2 … 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
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{}	0	$ 0
{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

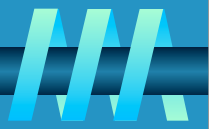
Efficiency: $\Omega(2^n)$

(grows at least as fast as 2^n)

as number of subsets in an n-element set is 2^n

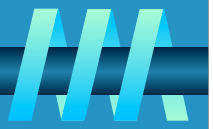
Knapsack capacity
Weight = 16

NP-hard problems



- No Polynomial-time algorithm is know for any NP-hard problem.
- It is generally accepted that there is no polynomial-time algorithm possible for the NP-hard class problems...
  - prove one wrong and become famous...
- Can beat exponential time for some using techniques discussed later in class – backtracking, branch-and-bound, approximation algorithms.
- For more on NP-hard (and other NP's) – check out wikipedia or take and advanced algorithm course

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? n! 4! = 24

Pose the problem as the one about a cost matrix:

Assignment Problem by Exhaustive Search

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

(person by row, job by column)

Assignment (col.#s)	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.

There will be 3 more sets of 6 similarly constructed cost matrix to complete the solution set.

Assignment Problem by Exhaustive Search

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

(person by row, job by column)

Assignment (col.#s)	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.

There will be 3 more sets of 6 similarly constructed cost matrix to complete the solution set.

Assignment Problem by Exhaustive Search

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

(person by row, job by column)

Assignment (col.#s)	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.

There will be 3 more sets of 6 similarly constructed cost matrix to complete the solution set.

Assignment Problem by Exhaustive Search

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

(person by row, job by column)

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

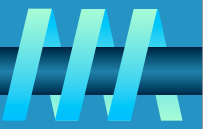
and so on…

There will be 3 more sets of 6 similarly constructed cost matrix to complete the solution set.

Final Comments on Exhaustive Search

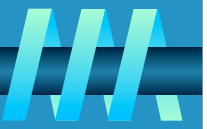
- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem
- In many cases, exhaustive search or its variation is the only known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

Chapter 3:



- 3.1 selection sort and bubble sort
- 3.2 sequential search and brute-force string matching
- 3.3 closest-pair by brute force
- 3.4 exhaustive search
- 3.5 depth-first search and breadth-first search

Graph Traversal Algorithms

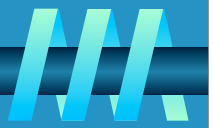


Many problems require processing all graph vertices (and edges) in systematic fashion

Graph traversal algorithms:

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

Depth-First Search (DFS)

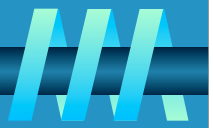


- Visits graph's vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.

- 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

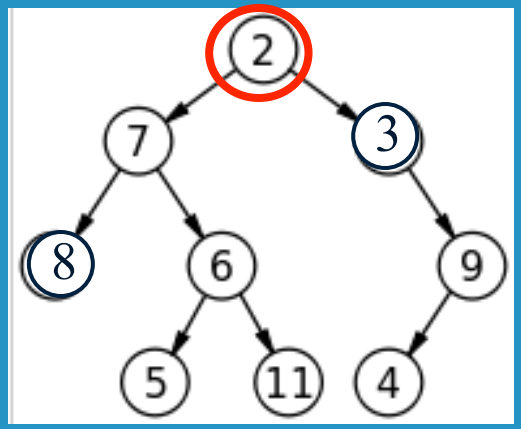


ALGORITHM *DFS*(G)

```
//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'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)
```

Example: DFS traversal of (binary tree) graph



DFS traversal stack:

$5_{5,2}$ $11_{6,3}$ $4_{9,6}$
$8_{3,1}$ $6_{4,4}$ $9_{8,7}$
$7_{2,5}$ $3_{7,8}$
$2_{1,9}$ 1st pushed on stack

vertex $_{\text{push, pop}}$

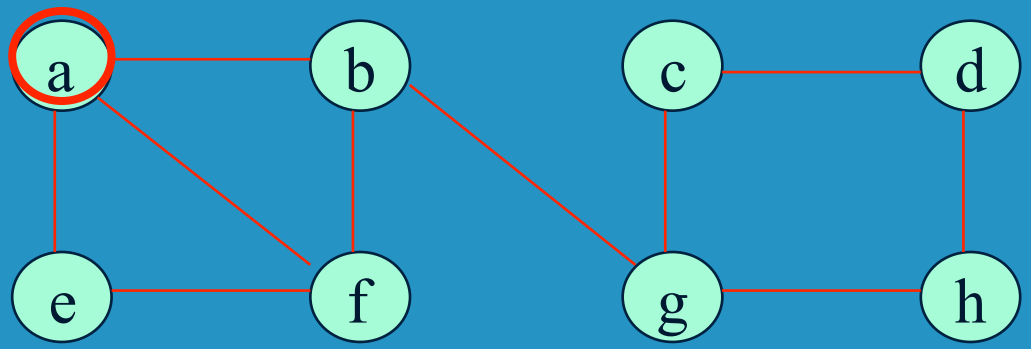
DFS tree:

Same as above

DFS traverse nodes left to right

Example: DFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



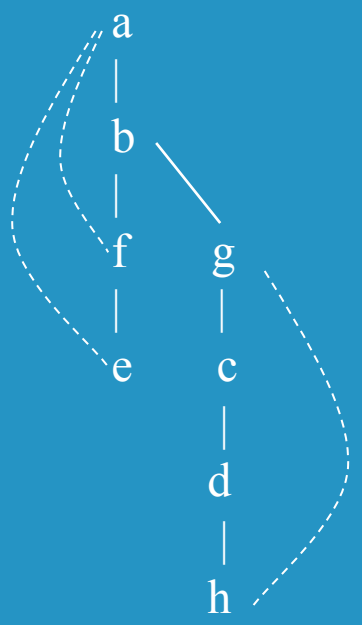
order pushed onto stack: a b f e g c d h
order popped from stack: e f h d c g b a

DFS traversal stack:

$h_{8,3}$ last pushed on stack
$d_{7,4}$
$c_{6,5}$
$g_{5,6}$
$e_{4,1}$
$f_{3,2}$
$b_{2,7}$
$a_{1,8}$ 1st pushed on stack

vertex push, pop

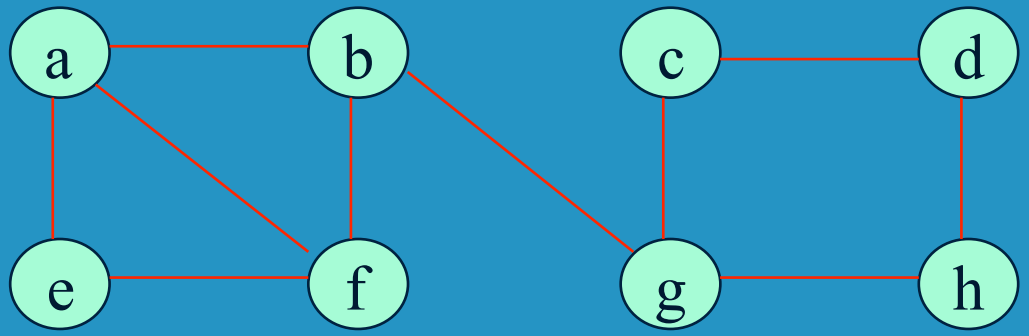
DFS tree:



edges

back edges

Example: DFS traversal of undirected graph



you can reproduce the graph from an adjacency matrix or list

Adjacency Matrix

	a	b	c	d	e	f	g	h
a	0	1	0	0	1	1	0	0
b	1	0	0	0	0	1	1	0
c	0	0	0	1	0	0	1	0
d	0	0	1	0	0	0	0	1
e	1	0	0	0	0	1	0	0
f	1	1	0	0	1	0	0	0
g	0	1	1	0	0	0	0	1
h	0	0	0	1	0	0	1	0

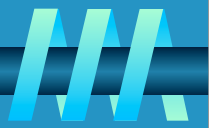
Notice that adjacency matrix is symmetric.

This is because links in undirected graph are considered bidirectional.

Adjacency List

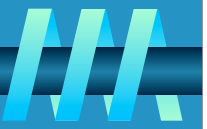
a	b	e	f
b	a	f	g
c	d	g	
d	c	h	
e	a	f	
f	a	b	e
g	b	c	h
h	d	g	

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
    - some vertices not used in DFS algorithm, graph not connected
  - checking acyclicity
    - no back edges, acyclic graph
  - finding articulation points and biconnected components
  - searching state-space of problems for solution (AI)

Breadth-first search (BFS)



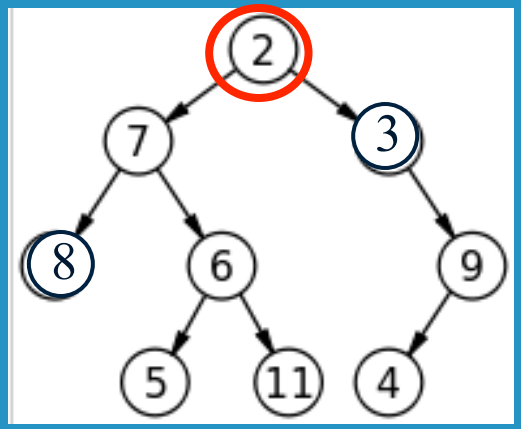
- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- "Redraws" graph in tree-like fashion (with tree edges and cross edges for undirected graph)

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 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
```

Example: BFS traversal of (binary tree) graph



BFS traversal queue:

2_1 7_2 3_3 8_4 6_5 9_6 5_7 11_8 4_9

BFS tree:

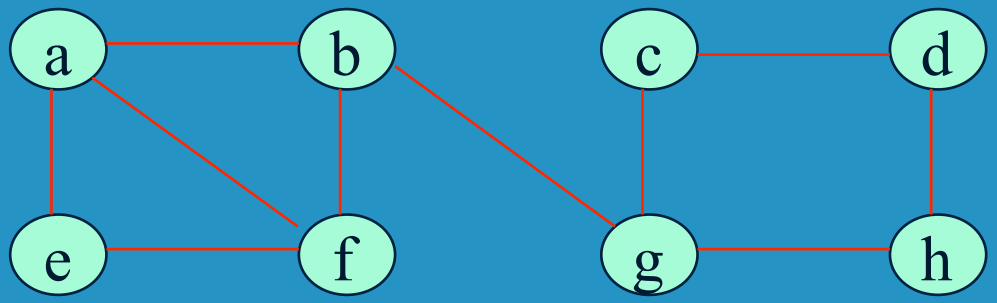
Same as above

BFS traverse nodes left to right

vertex queue-order

Example of BFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



BFS traversal queue:

a_1 b_2 e_3 f_4 g_5 c_6 h_7 d_8

BFS tree:

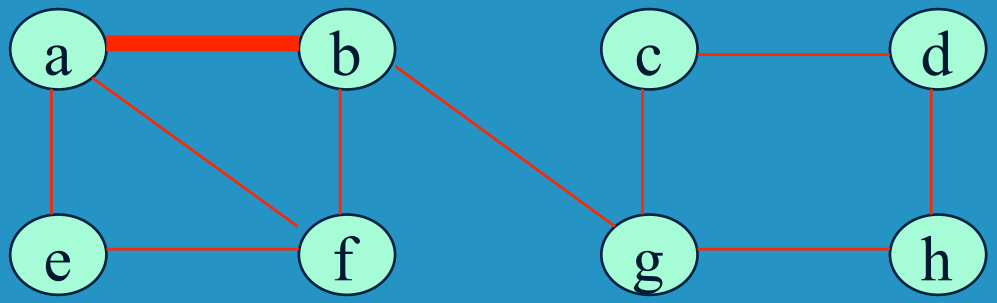
edges

cross edges

cross edges (same or adjacent levels in BFS)

Example of BFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



BFS traversal queue:

a_1 b_2 e_3 f_4 g_5 c_6 h_7 d_8

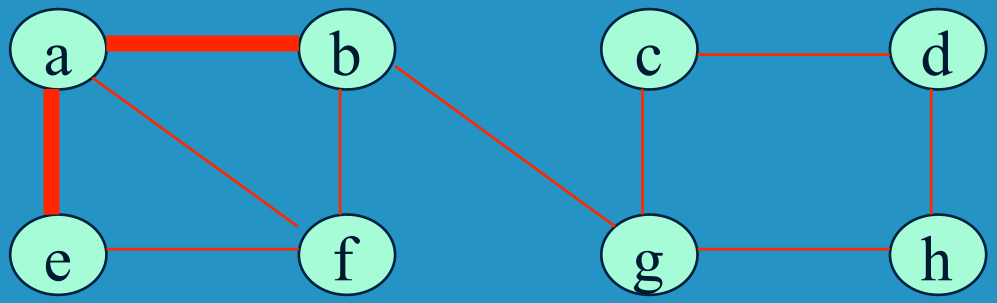
BFS tree:

edges

cross edges

Example of BFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



BFS traversal queue:

a_1 b_2 e_3 f_4 g_5 c_6 h_7 d_8

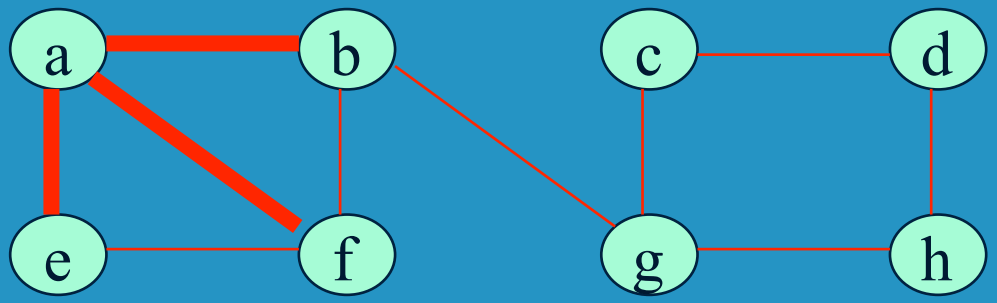
BFS tree:

edges

cross edges

Example of BFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



BFS traversal queue:

a_1 b_2 e_3 f_4 g_5 c_6 h_7 d_8

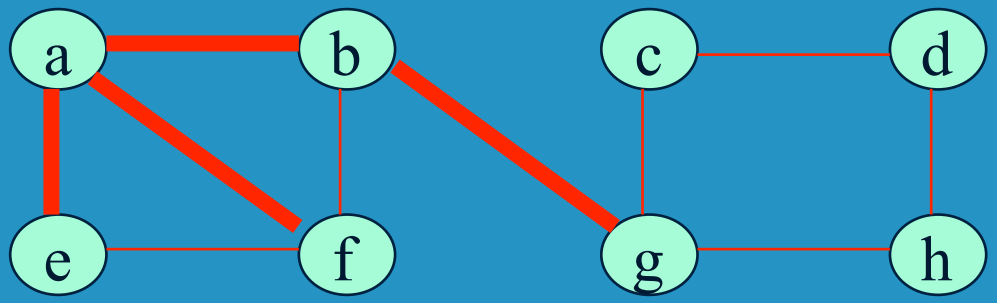
BFS tree:

edges

cross edges

Example of BFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



BFS traversal queue:

a_1 b_2 e_3 f_4 g_5 c_6 h_7 d_8

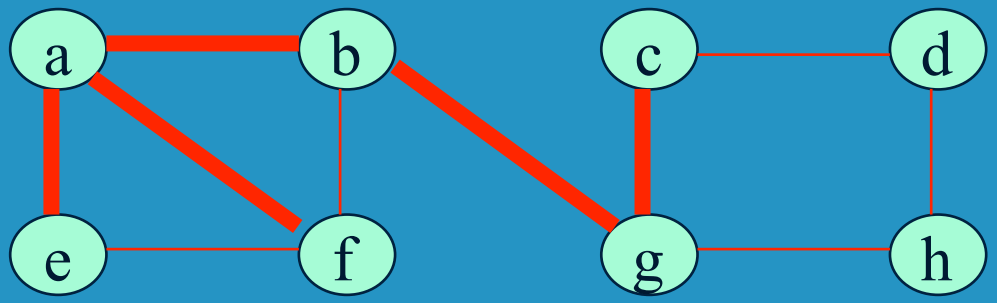
BFS tree:

edges

cross edges

Example of BFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



BFS traversal queue:

a_1 b_2 e_3 f_4 g_5 c_6 h_7 d_8

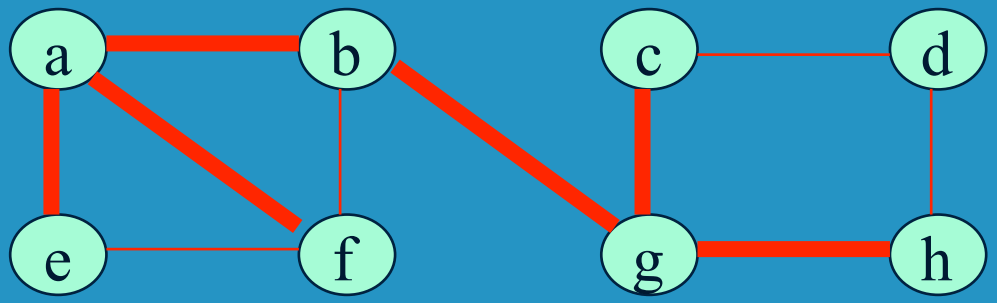
BFS tree:

edges

cross edges

Example of BFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



BFS traversal queue:

a_1 b_2 e_3 f_4 g_5 c_6 h_7 d_8

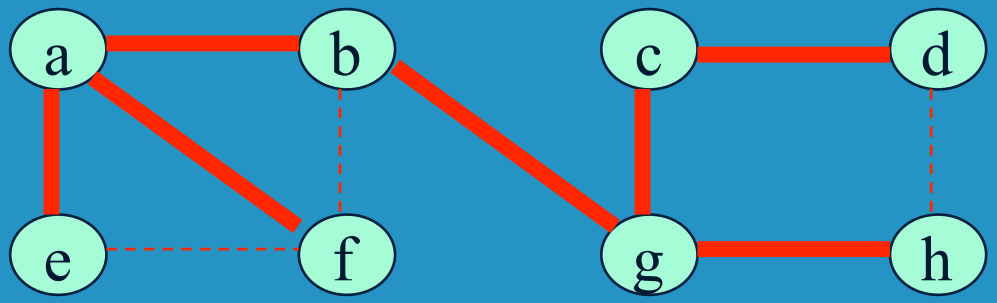
BFS tree:

edges

cross edges

Example of BFS traversal of undirected graph

Traverse using alphabetical order of vertices (as within reach)



BFS traversal queue:

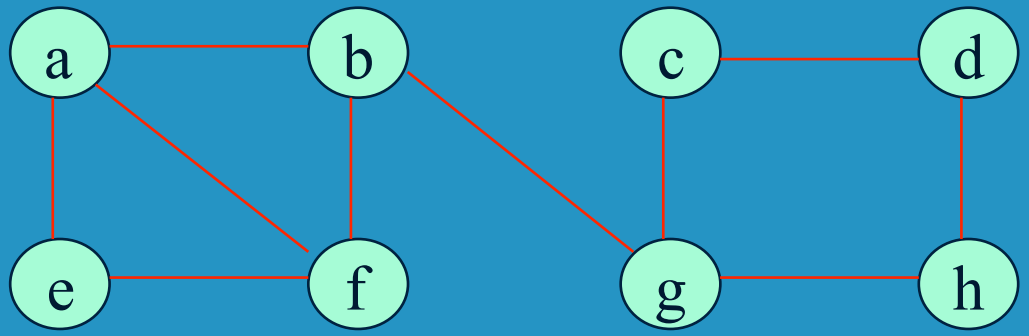
a_1 b_2 e_3 f_4 g_5 c_6 h_7 d_8

BFS tree:

edges

cross edges

Example: BFS traversal of undirected graph



Note that adjacency matrix and adjacency list are the same as that of DFS.

Why would it change, the adjacencies are still the same...

Adjacency Matrix

	a	b	c	d	e	f	g	h
a	0	1	0	0	1	1	0	0
b	1	0	0	0	0	1	1	0
c	0	0	0	1	0	0	1	0
d	0	0	1	0	0	0	0	1
e	1	0	0	0	0	1	0	0
f	1	1	0	0	1	0	0	0
g	0	1	1	0	0	0	0	1
h	0	0	0	1	0	0	1	0

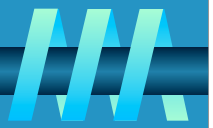
Adjacency List

a	b	e	f
b	a	f	g
c	d	g	
d	c	h	
e	a	f	
f	a	b	e
g	b	c	h
h	d	g	

Notes on BFS

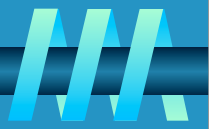
- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$
- Yields single ordering of vertices (order added/deleted from queue is the same)
- Applications: same as DFS, but can also find paths from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

DFS and BFS facts:



	DFS	BFS
Data Structure	stack	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 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