Chapter 4

Brute Force

Brute Force

A straightforward approach usually based directly on the problem's statement and definitions of the concepts involved.

"Just do it!" would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)= a*a*….*a
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 $A[i..n\text{-}1]$ and swap it with $A[i]$:

$$A[0] \;\le\; .\;\;.\;\;.\;\; \le A[i\text{-}1] \;\mid\; A[i],\; .\;\;.\;\;.\;\;,\, A[min],\; .\;\;.\;\;.,\, A[n\text{-}1]$$

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 ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

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

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

Time efficiency: **$\Theta(n$^2)**

Space efficiency: **$\Theta(1)$, so in place**

Stability: **yes**

Brute Force - Bubble Sort

Another brute force application to the sorting problem is to

- compare adjacent elements of the list.
- and exchange them if they are out of order.

By doing it repeatedly, we end up “Bubbling up” the largest element to the last position on the list
The next pass bubbles up the second largest element, and so on until, after n-1 passes, the list is sorted.

Brute Force - Bubble Sort

Algorithm BubbleSort (A[0…n-1])

for i ← 0 to n-2 do

for j ← 0 to n-2-i do

if A[j]<A[j+1]

swap A[j] and A[j+1]

The number of times it is executed T(n) depends only on the array's size and is given by the following sum:

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

$$T(n) = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

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

1. Pattern: 001011
   Text: 10010101101001100101111010

2. 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
```

Time efficiency: **Θ(mn) comparisons (in the worst case)**

Why?

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 $x = x_0$

<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
```

Efficiency: **Θ(n^2) multiplications**

Polynomial Evaluation: Improvement

We can do better by evaluating from right to left:

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

$p \leftarrow a[0]$
$power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  $power \leftarrow power \ast x$
  $p \leftarrow p + a[i] \ast power$
return p

Efficiency: **$\Theta(n)$ multiplications**

Horner's Rule is another linear time method.

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.

Clos

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 $index1$ and $index2$ 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 $index1, index2$

Efficiency:

Θ(n^2) multiplications

How to make it faster?

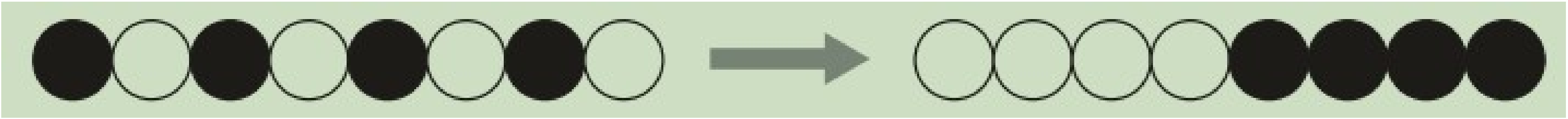
Using divide-and-conquer!

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

Write a brute force algorithm for Alternating disks puzzle.



Exhaustive Search

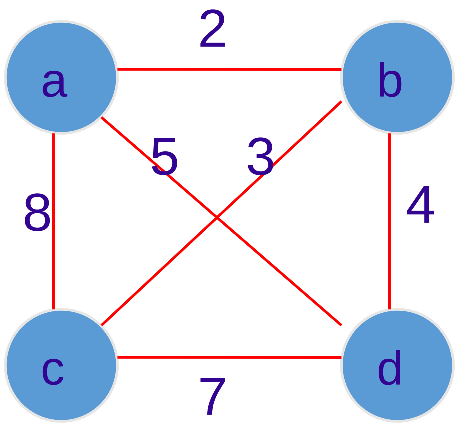
A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method:

- generate a list of all potential solutions to the problem in a systematic manner
- 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

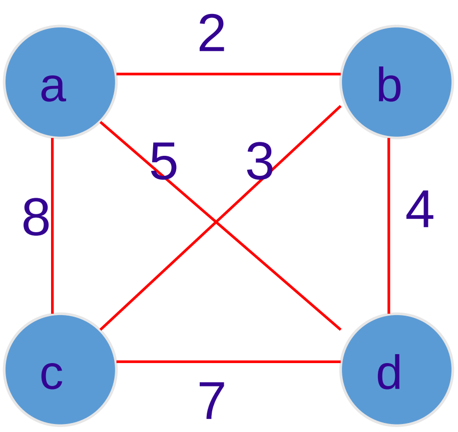
- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively: Find shortest ***Hamiltonian circuit*** in a weighted connected graph
- ***Example***:



How do we represent a solution (Hamiltonian circuit)?

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



Efficiency:

$\Theta((n-1)!)$

Chapter 5 discusses how to generate permutations fast.

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
{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

Each subset can be represented by a binary string (bit vector, Ch 5).

Efficiency: Θ(2^n)

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.

n!

How many assignments are there?

Assignment Problem by Exhaustive Search

C =

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

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.

(For this particular instance, the optimal assignment can be found by exploiting the specific features of the number given. It is: 2,1,3,4)

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time <u>only on very small instances</u>

- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - assignment problem

The Hungarian method runs in O(n^3) time.

- In many cases, exhaustive search or its variation is the only known way to get exact solution

Graph Traversal

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: (Brave Traversal)

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

- **Recursive or it uses a stack**

- **Using 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

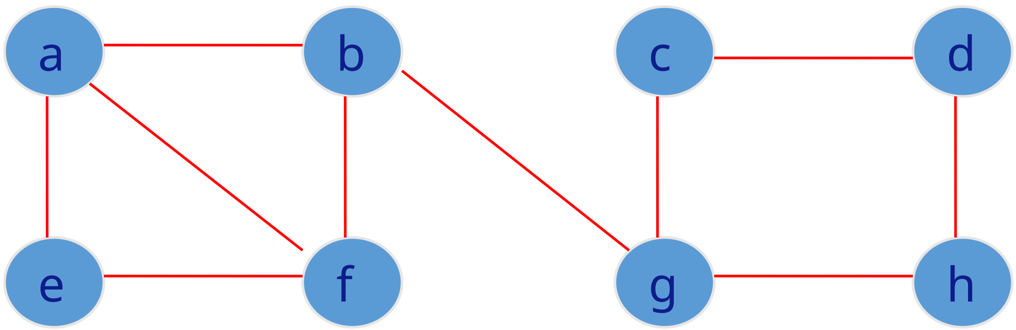
Algorithm

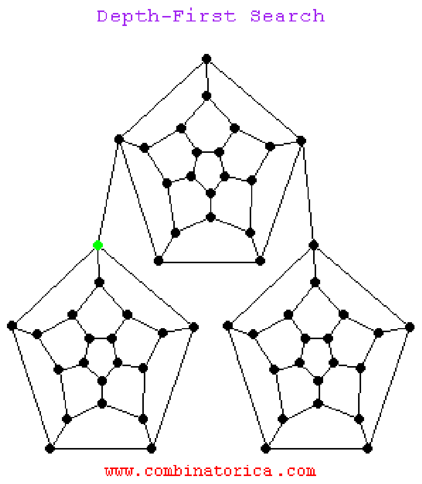
ALGORITHM $DFS(G)$

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

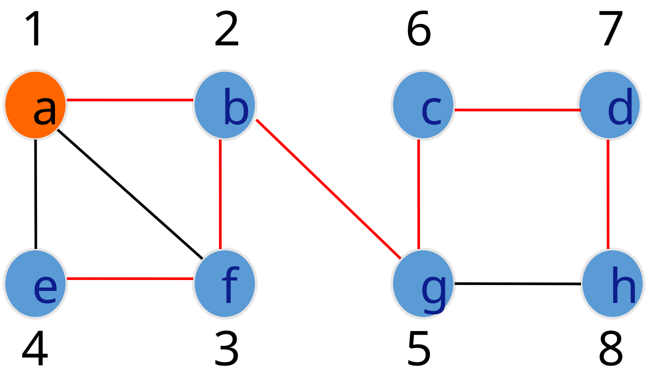
$dfs(v)$
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable $count$
$count \leftarrow count + 1$; mark v with $count$
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        $dfs(w)$

Example: DFS traversal of undirected graph





DFS tree:



Red edges are tree edges and other edges are back edges.

DFS traversal stack:

a
ab
abf
abfe
abf
ab
abg
abgc
abgcd
abgcdh
abgcd
…

Notes on DFS

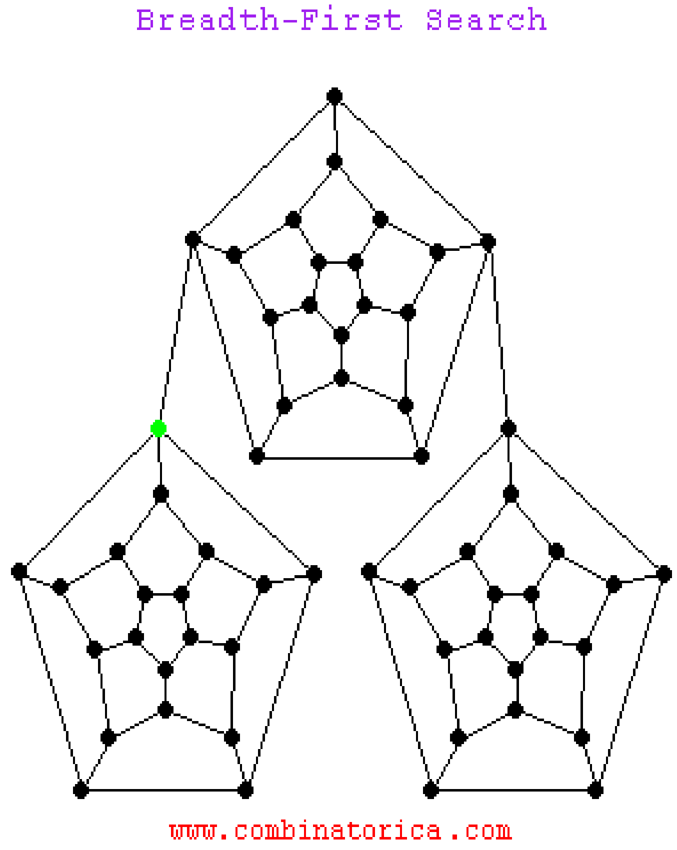
- **DFS can be implemented with graphs represented as:**
  - adjacency matrices: $\Theta(|V|^2)$. Why?
  - adjacency lists: $\Theta(|V|+|E|)$. Why?

- **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 a cyclicity (if no back edges)

The Pr'

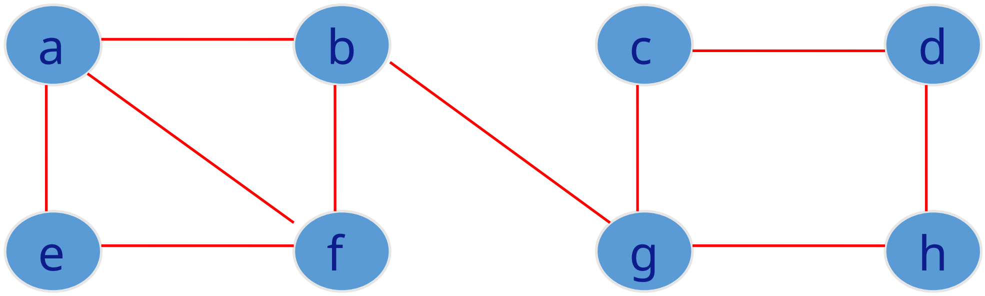
Finding paths from a vertex to all other vertices with the smallest number of edges



Breadth First Search

- Visits graph vertices by moving across to all the neighbors of the last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- “Redraws” graph in tree-like fashion.

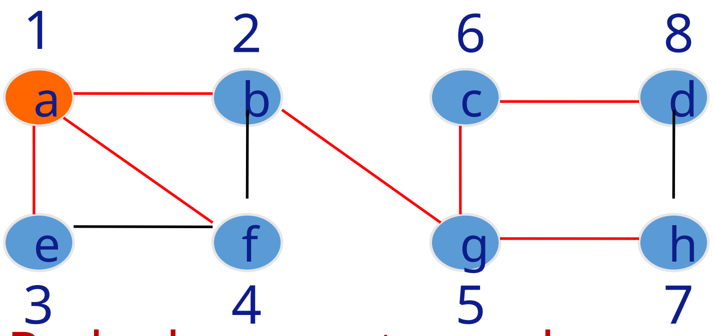
Example of BFS traversal of undirected graph



BFS traversal queue:

a
bef
efg
fg
g
ch
hd
d

BFS tree:



Red edges are tree edges and white edges are cross edges.

Write an Algorithm for BFS Using a queue?

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

$bfs(v)$
//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 $count$
$count \leftarrow count + 1$;   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
            $count \leftarrow count + 1$;   mark w with $count$
            add w to the queue
    remove the front vertex from the queue