Brute Force

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

Examples – based directly on definitions:

1. Computing a^n ($a > 0$, n a nonnegative integer)

2. Computing n!

3. Multiplying two matrices

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

Sorting by Brute Force

Use definition of sorted and obvious algorithm?

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 $A[i]$:**

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

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 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 $A[min]$

Time efficiency:

Space efficiency: ?

Stability:

String Matching by Brute Force

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find first substring in text that matches pattern

Brute-force: Scan text LR, compare chars, looking for pattern,

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

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:

(Basic op and dataset assumptions?)

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

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

Efficiency: A(n)= ?. M(n)=

Polynomial Evaluation: Improvement

Improve by evaluating from right to left (Horner's Method):

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

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

Efficiency: A(n)= ?. M(n)=

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.

Closest-Pair Brute-Force Algorithm (cont.)

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:
Advantage:
How to make it faster? Constant factor, n lg n

Complex Hull

Convex Hull:

BF Algorithm:

Complexity: n^?

Complex Hull

Convex Hull:

BF Algorithm: For each segment, are all points on one side

Complexity: n^3

Complex Hull

Convex Hull:

BF Algorithm: For each segment, are all points on one side

Complexity: n^3

Improved algorithm: Average case: n^1
Worst case: n^2

Exhaustive Search

Many Brute Force Algorithms use Exhaustive Search

- Example: Brute force Closest Pair

Approach:

1. Enumerate and evaluate all solutions, and
2. Choose solution that meets some criteria (eg smallest)

Frequently the obvious solution

But, slow (Why?)

Exhaustive Search – More Detail

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

Exhaustive Search – More Examples

Traveling Salesman Problem (TSP)

Knapsack Problem

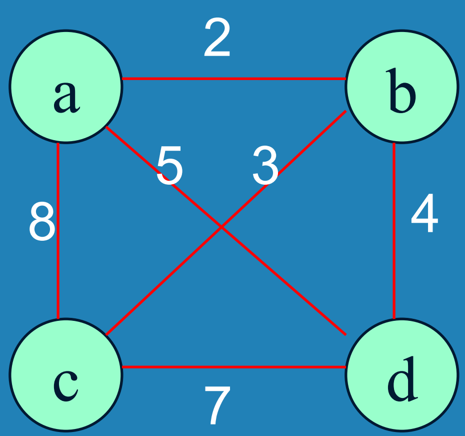
Assignment Problem

Graph algorithms:
- Depth First Search (DFS)
- Breadth First Search (BFS)

Better algorithms may exist

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
- More formally: 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

Have we considered all tours?
Do we need to consider more?
Any way to consider fewer?

Efficiency: Number of tours = number of …

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

Have we considered all tours? Start elsewhere: b-c-d-a-b
Do we need to consider more? No
Any way to consider fewer? Yes: Reverse

Efficiency: # tours = O(# permutations of b,c,d) = O(n!)

Example 2: Knapsack Problem

Given n items:

- weights: $w_1 \quad w_2 \; \ldots \; w_n$
- values: $v_1 \quad v_2 \; \ldots \; v_n$
- 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
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack: 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: how many subsets?

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 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4

Algorithmic Plan:

Generate all legitimate assignments

Compute costs

Select cheapest

Assignment Problem: 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
…	…

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

Assignment Problem: 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
…	…

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

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 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4

Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

How many assignments are there?

Describe sol'n using cost matrix:

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 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4

Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

How many assignments are there: permutations of 1..n = n!

Sol'n using cost matrix: select one from each row/col. Min sum.

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

GRAPHS

Many problem solutions use a graph to represent the data:

- TSP
- Cities and roads
- Network nodes and connections among them
- People and friends

What is a graph? A graph is defined by two sets:

- Set of Vertices
- Set of Edges that connect the vertices

We look at two aspects of graphs:

- Standard graph algorithms (eg DFS and BFS in this chapter)
- Solve some problems with graphs

Graph Traversal Algorithms

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

<u>Graph traversal algorithms</u>:

- Depth-first search (DFS): Visit children first
- Breadth-first search (BFS): Visit siblings first

GRAPHS – Definition Expanded

Graph consists of two sets:

- **Set of vertices (aka nodes)**
- **Set of edges that connect vertices**
  - **Edges may have weights**
  - **Edges may have directions:**
    - **Undirected graph: edges have no directions**
    - **Directed graph: edges have direction**
      - **AKA Digraph**

GRAPH TERMS (Chap 1)

Degree of a node: Number of edges from it (in deg and out deg)

Path: Sequence of vertices that are connected by edges

Cycle: Path that starts and ends at same node

Connected graph:

- **every pair of vertices has a *path* between them**

Complete graph:

- **every pair of vertices has an *edge* between them**

Tree: connected acyclic graph

- **Tree with n nodes has n-1 edges**
- **Root need not be specified**
- **Regular terms: Parent, child, ancestors, descendents, siblings, …**

Forest: set of trees (ie not-necessarily-connected acyclic graph)

Graph Implementation

Adjacency matrix:

- Row and column for each vertex
- 1 for edge or 0 for no edge
- Undirected graph: What is true of the matrix?

Adjacency lists

- Each node has a list of adjacent nodes

Each has pros and cons. Performance:

- Check if two nodes are adjacent: …
- List all adjacent nodes: …

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.
- Implements backtracking using a stack
  - a vertex is pushed when it's reached for the first time
  - a vertex is popped when it becomes a dead end, i.e., when there are no adjacent unvisited vertices
- Marks edges in tree-like fashion (mark edges as tree edges and back edges [goes back to already discovered *ancestor* vertex].
  - In a DFS of an undirected graph, each edge becomes either a tree edge or a back edge (back to ancestor in tree)

Pseudocode of DFS

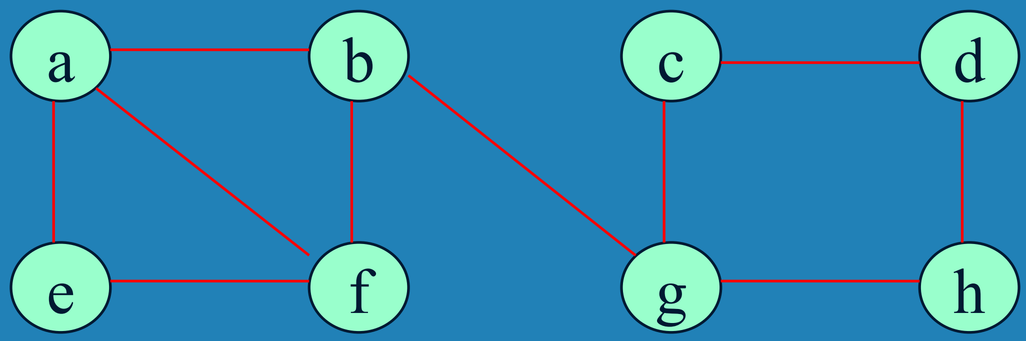
ALGORITHM *DFS(G)*

//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
    $dfs(w)$

Stack?

Example: DFS traversal of undirected graph

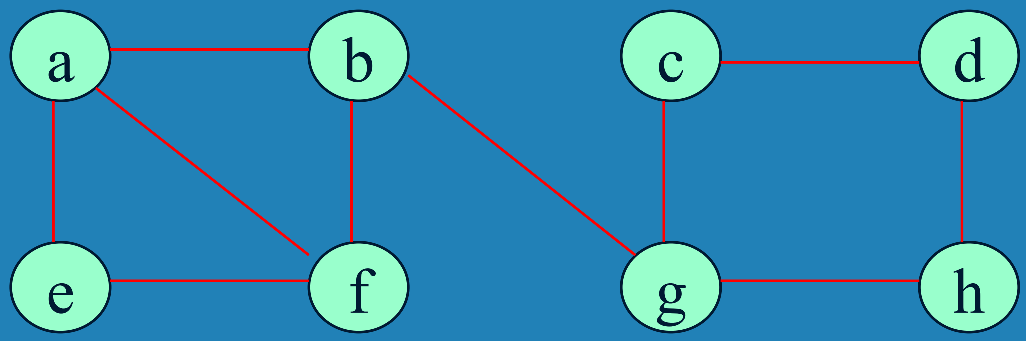


DFS traversal stack:

DFS tree:

$\mathbf{a}_{1,}$

Example: DFS traversal of undirected graph



Nodes pushed: a b f e g c d h

Nodes popped: e f h d c g b a

Tree Edges: each v in dfs(v) defines a tree edge

Back Edge: encountered edge to previously visited *ancestor*

What nodes are on the stack?

Complexity:

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)
  - Orderings and edges used by various algorithms
    - (eg scheduling / topological sort)
- 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

- BFS uses a queue (not a stack like DFS)

- Similar to level-by-level tree traversal

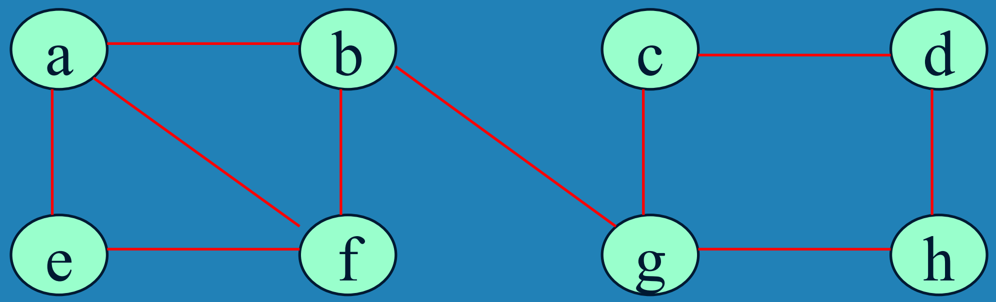
- Marks edges in tree-like fashion (mark tree edges and cross edges [goes across to an already discovered *sibling* vertex]
  - In a BFS of an undirected graph, each edge becomes either a tree edge or a cross edge (to neither ancestor nor descendant in tree-common ancestor or other tree)

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
```

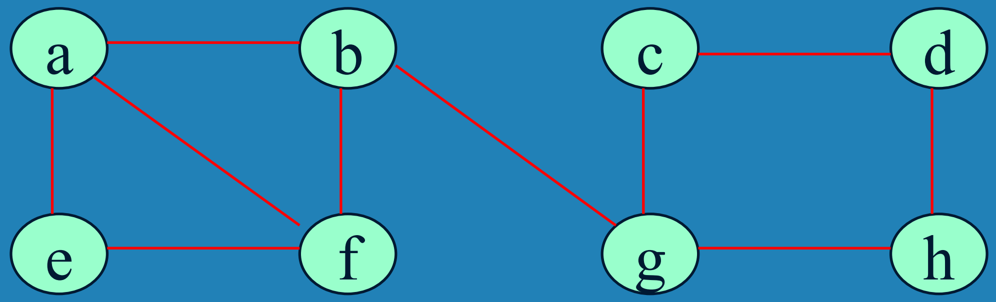
Example of BFS traversal of undirected graph



BFS traversal queue:

BFS tree:

Example of BFS traversal of undirected graph



BFS traversal queue: a, b e f, g, c h, d

Level: 1, 2 2 2, 3, 4 4, 5

Tree Edges: as dfs

Cross Edges: encountered edge to previously visited *sibling or sibling's descendent (eg hypothetical edge* eg)

What nodes are on the queue?

Performance:

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 [How: mark depth from root]**

Brute Force: Review

- Based on problem statement and definitions
- Typically slow, but may be only known algorithm
- Useful to consider first
  - better algorithm frequently known
- Examples:
  - Sorting and Searching
  - Exhaustive Search:
    - Pattern Match, TSP, Knapsack, Assignment,
  - Graph (DFS, BFS)

Brute-Force Strengths and Weaknesses

- **Strengths**
  - Wide applicability
  - Simplicity
  - Yields reasonable algorithms for some important problems (e.g., matrix multiply, sorting, searching, string matching)
  - Algorithm may be good enough for small problem
  - Improvement may be too hard
  - Provides yardstick for comparison

- **Weaknesses**
  - Rarely yields efficient algorithms
  - Some brute-force algorithms are unacceptably slow
  - Not as constructive as some other design techniques