Chapter 3 Brute Force and Exhaustive Search

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

The exponentiation problem: compute a^n, $a \neq 0$, $n \geq 0$, $n \in \mathbb{Z}$.

$$a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ times}}$$

Brute force approach: compute a^n by multiplying 1 by a n times.

3.1 Selection Sort and Bubble Sort

The problem of sorting: given a list of n orderable items, rearrange them in nondescending order.

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]
```

3.1 Selection Sort and Bubble Sort cont.

What is the size of the input? n

What is the basic operation? comparison

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

$$= \frac{(n-1)n}{2}$$

Thus, selection sort is a $\Theta(n^2)$ algorith on all inputs.

The number of swaps is only $\Theta(n)$, or, more precisely $n-1$.

3.1 Selection sort and Bubble Sort cont.

Bubble Sort

```
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 <- 0 to n-2 do
    for j <- 0 to n-2-i do
        if A[j+1] < A[j] swap A[j] and A[j+1]
```

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

$$= \frac{(n-1)n}{2} \in \Theta(n^2).$$

The number of swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of comparisons: $S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2)$.

3.2 Sequential Search and Brute-Force String Matching

```
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
```

What if A was sorted?

SequentialSearch and SequentialSearch2 are both linear in the worst case.

3.2 Sequential Search and Brute-Force String Matching cont.

Brute-Force String Matching

Given a string of n characters called the text and a string of m characters ($m \leq n$) called the pattern, find a substring of the text that matches the pattern.

```
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
```

3.2 Sequential Search and Brute-Force String Matching cont.

~~The algorithm shifts the pattern almost always after a single character comparison~~. The worst case is ~~much worse~~: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n-m+1$ tries. Thus, in the worst case, the algorithm makes $m(n-m+1)$ character comparisons, which puts it in the $O(nm)$ class.

For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons. Therefore, the average-case efficiency should be considerably better that the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e., $\Theta(n)$.

There are more efficient algorithms for string searching (Section 7.2)

3.3 Closest-Pair and Convex-Hull Problems by Brute Force

Two well-known problems dealing with a finite set of points in the plane.

Closest-Pair Problem: Find the two closest points in a set of n points.

The distance between two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

We consider only the pairs of points (p_i, p_j) for which $i < j$.

```
ALGORITHM    BruteForceClosestPair(P)
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n (n ≥ 2) points p_1(x_1, y_1), ..., p_n(x_n, y_n)
//Output: The distance between the closest pair of points
d ← ∞
for i ← 1 to n-1 do
    for j ← i+1 to n do
        d ← min(d, sqrt((x_i - x_j)^2 - (y_i - y_j)^2))
return d
```

3.3 Closest-Pair and Convex-Hull Problems by Brute Force cont.

The basic operation? Computing the square root.

Can we ignore square root function in the BruteForceClosestPair algorithm? Yes.

The basic operation in the refined algorithm? Squaring a number.

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

$$= (n-1)n \in \Theta(n^2).$$

↑ asymptotic efficiency class this algorithm belongs to

3.3 Closest-Pair and Convex-Hull Problems by Brute Force cont.

Convex-Hull Problem

Def. 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.

Def. 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.)

3.3 Closest-Pair and Convex-Hull Problems by Brute Force cont.

Theorem. The convex hull of any set S of $n > 2$ points not all on the same line is a convex polygon with the vertices at some of the points of S.

The convex-hull problem is the problem of constructing the convex hull for a given set S of n points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. These vertices are called extreme points.

Def. An extreme point of a convex set is a point of this set that is not a middle point of any line segment with endpoints in the set

To solve the convex-hull problem we need to find:

- which of n points of a given set are extreme points of the set's convex hull
- which pairs of points need to be connected to form the boundary of the convex hull.

3.3 Closest-Pair and Convex-Hull Problems by Brute Force cont.

There is an algorithm based on the following observation about line segments making up the boundary of a convex hull: 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. Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

3.3 Closest-Pair and Convex-Hull Problems by Brute Force cont.

The 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 $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1 y_2 - y_1 x_2$.

Such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$.

Thus, to check whether certain points lie on the same side of the line, we can check whether the expression $ax + by - c$ has the same sign for each of these points.

What is the time efficiency of this algorithm? It is in $O(n^3)$: for each of $n(n-1)/2$ pairs of distinct points, we need to find the sign of $ax + by - c$ for each of the other $n-2$ points.

3.4 Exhaustive Search

Exhaustive search is a brute-force approach to combinatorial problems.

Traveling Salesman Problem

It asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

Consider $c_0, c_1, c_2, \ldots, c_{n-1}, c_0$.

We can get all the tours by generating all the permutations of n cities, compute the tour lengths, and find the shortest among them.

The total number of permutations needed is $n!$.

3.4 Exhaustive Search cont.

Knapsack Problem

Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n given items, computing the total weight of each subset in order to identify feasible subsets, and finding a subset of the largest value among them.

Since the number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

3.4 Exhaustive search cont.

Traveling-Salesman Problem and Knapsack Problem are the best-known examples of so called NP-hard problems.
No polynomial-time algorithm is known for any NP-hard problem.

see Millenium Prize Problems

3.5 Depth - First Search and Breadth - First Search

Depth - First Search

It starts a graph's traversal at an arbitrary vertex by marking it as visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. If there are several such vertices, a tie can be resolved arbitrarily. This process continues until a dead end - a vertex with no adjacent unvisited vertices - is encountered. 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

By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth - first search must be restarted at any one of them.

3.5 Depth-First Search and Breadth-First Search cont.

```
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 alg.
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)
```

```
ALGORITHM  dfs(v)
// visits recursively all the unvisited
// vertices connected to vertex v by a pa
// 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)
```

3.5 Depth-First Search and Breadth-First Search cont.

This algorithm takes the time proportional to the size of the data structure used for representing the graph in question. Thus, for the adjacency matrix representation, the traversal time is in $\Theta(|V|^2)$, and for the adjacency list representation, it is in $\Theta(|V|+|E|)$ where $|V|$ and $|E|$ are the number of the graph's vertices and edges, respectively.

Breadth-First Search

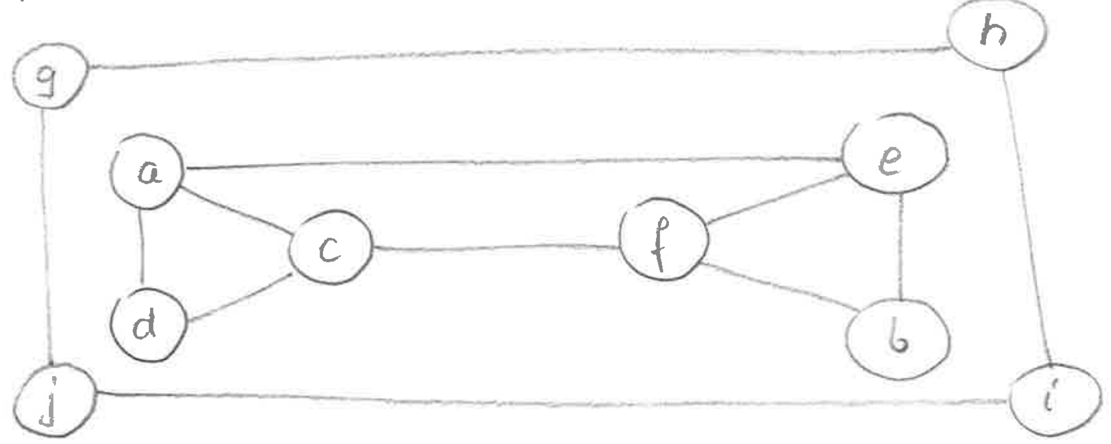
This algorithm proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, untill all the vertices in the same connected component as the starting vertex are visited. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

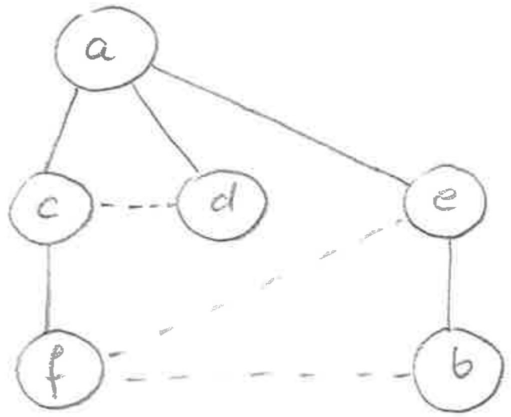
3.5 Depth-First Search and Breadth-First Search cont.

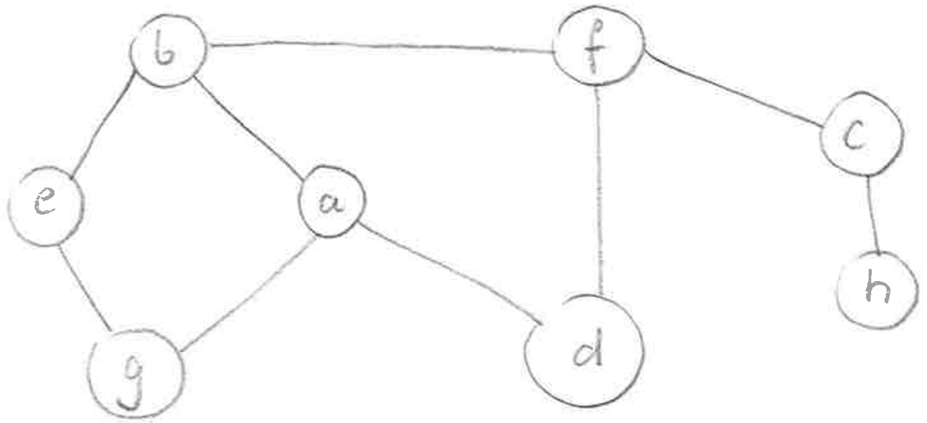
```
ALGORITHM  BFS(G)
// Implements a breadth-first search traversal of a given graph
// Input : Graph G = <V, E>
// Output : Graph G with its vertices marked with consecutive
// integers in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)
```

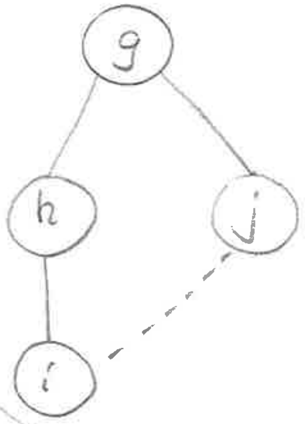
```
bfs(v)
// visits all the unvisited vertices connected to v
// by a path and numbers them in the
// order they are visited via global variable
// count
count ← count + 1; mark v with count and
initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front
    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
```

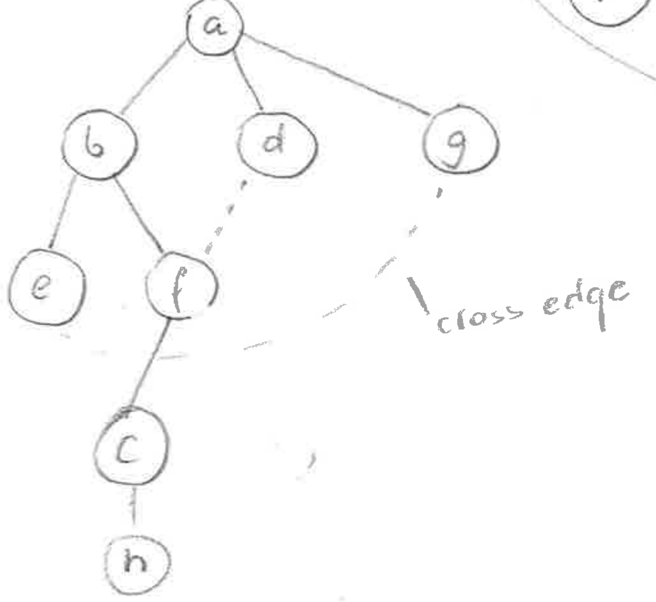
BFS









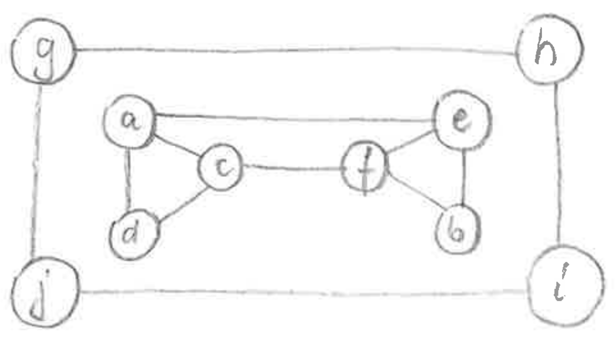


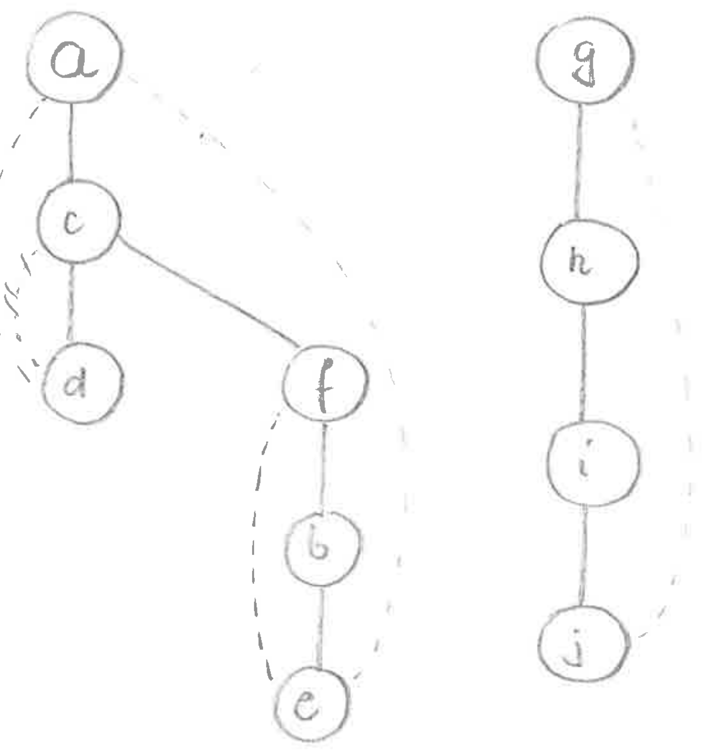
~~a~~ ~~b~~ ~~d~~ ~~g~~ ~~e~~ ~~f~~ ~~c~~ ~~d~~ ~~h~~

3.5 Depth - First Search and Breadth - First Search cont.

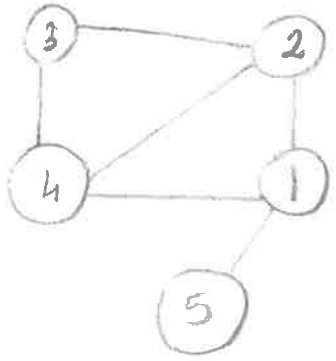
Breadth-first search has the same efficiency as depth-first search: it is in $\Theta(|V|^2)$ for the adjacency matrix representatio
and in $\Theta(|V|+|E|)$ for the adjacency list representation.

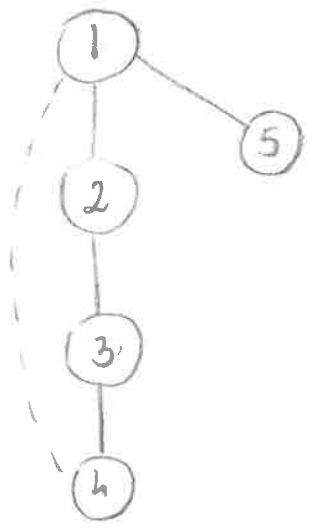
Example DFS





Example

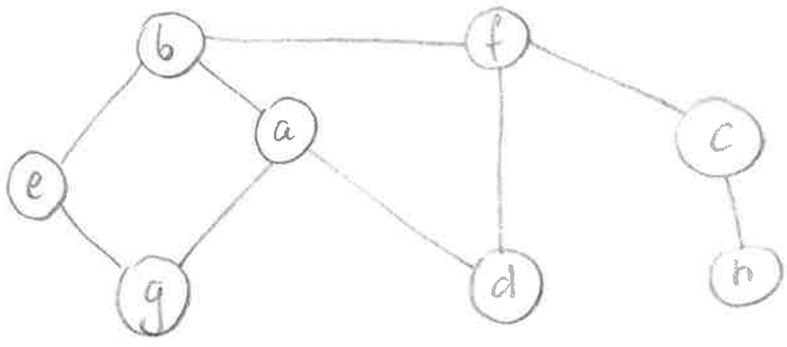


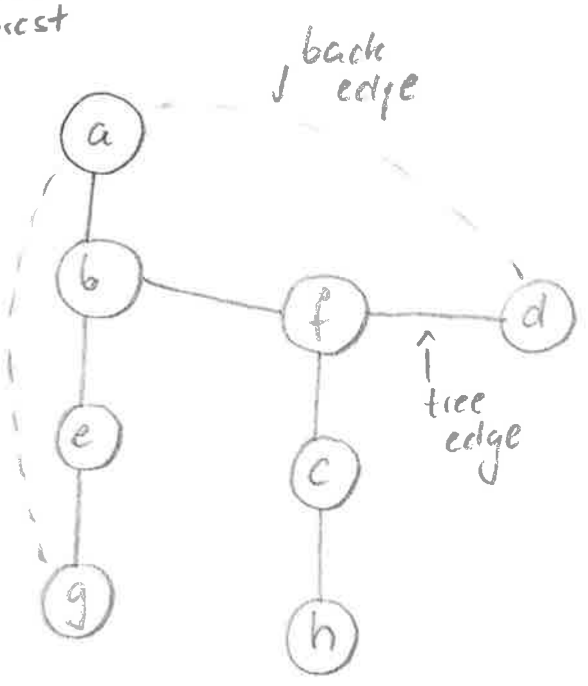


u max # of tree edges + back edges ≠ # of edges in the input graph

correct that!

DFS forest




back edge
tree edge