Brute Force and Exhaustive Search

Madiba Hudson-Quansah

Contents

Chapter 1

Introduction

1.1 Brute Force

Definition 1.1.1: Brute Force

A straightforward problem-solving approach, directly based on the problem statement and definitions of the concepts involved. This technique involves enumerating all possible candidates for the solution and identifying the one that completely solves the problem statement.

An example of a brute force problem the exponentiation problem: Compute a^n for a nonzero integer a and a nonnegative integer n. By the definition of exponentiation:

$$a^n = a \times \ldots \times a \quad (n \text{ times})$$

This suggests computing a^n by multiplying a by itself n times. This is a brute force solution to the problem. The brute force approach is useful as it applies to a large class of problems and can be used as a starting point for more sophisticated algorithms. Also the expense of designing a more efficient algorithm may not be justified if the problem instances are small or infrequent and a brute force solution can solve the problem in a reasonable amount of time.

Chapter 2

Selection Sort and Bubble Sort

2.1 Selection Sort

Definition 2.1.1: Selection Sort

Start by scanning the entire list to identify the smallest element and exchange it with the first, putting the smallest element in its correct position in the sorted list. Then scan the remaining $n-1$ elements to identify the next smallest element and exchange it with the second element, putting the second smallest element in its correct position. Repeat until the entire list is sorted. Generally on the ith pass through the list, numbered from 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ items and swaps it with A_i, i.e.:

$$A_0 \leqslant A_1 \leqslant \ldots \leqslant A_{i-1} \mid A_i, \ldots, A_{\text{min}}, \ldots A_{n-1}$$

Which results in the list being sorted after $n-1$ passes.
The brute force approach in this is the continuous scanning and swapping of elements until the list is sorted.

Algorithm 1 SelectionSort (A, n)

▹ Sorts a given array by selection sort
▹ Input: An array $A[0 \ldots n-1]$ of orderable elements
▹ Output: An array $A[0 \ldots n-1]$ sorted in nondecreasing order

```
1:
2:  for i ← 0 to n − 2 do
3:      min < −i
4:      for j ← i + 1 to n − 1 do
5:          if A_j < A_min then
6:              min ← j
7:          end if
8:      end for
9:      swap A_i and A_min
10: end for
```

Analysed in terms of the size of the input list n, we have:

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{i=1}^{n-1} 2 \\ &= \sum_{i=0}^{n-2} 2(n-1) - (i+1) + 1 \\ &= \sum_{i=0}^{n-2} 2(n-i-1) \\ &= 2 \sum_{i=0}^{n-2} (n-1-i) \\ &= 2 \frac{(n-1)(n-2)}{2} \\ &= n^2 - 2n - n + 2 \\ &= n^2 - 3n + 2 \end{aligned}$$

$\therefore$ $O(n^2)$ and $\Theta(n^2)$

2.2 Bubble Sort

Definition 2.2.1: Bubble Sort

Compare adjacent elements of the list and exchange them if they are out of order, doing this repeatedly causes the largest element in the list to bubble up to the last position in the list. The next pass bubbles up the next largest element and so on, until after $n-1$ passes the list is sorted, i.e:

$$A_0, \ldots, A_j \leftrightarrow A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \leqslant \ldots \leqslant A_{n-1}$$

Algorithm 2 BubbleSort (A, n)

▹ Sorts a given array by bubble sort
▹ Input: An array $A[0 \ldots n-1]$ or orderable elements
▹ Output: An array $A[0 \ldots n-1]$ sorted in non-decreasing order

```
1: for i ← n − 1 to 0 do
2:     for j ← 0 to i − 1 do
3:         if A_{j+1} < A_j then
4:             swap A_j and A_{j+1}
5:         end if
6:     end for
7: end for
```

And the time complexity of the bubble sort algorithm is:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{i} 2 \\ &= \sum_{i=0}^{n-1} 2(i+1) \\ &= 2 \sum_{i=0}^{n-1} (i+1) \\ &= 2\left(\frac{n(n+1)}{2}\right) \\ &= n^2 + n \end{aligned}$$

Bubble sort works because on each pass through the list, the next largest element is moved to the end of the list. This done for the length of the list will result in all the elements eventually in their correct positions.

Chapter 3

Sequential Search and Brute-Force String Matching

3.1 Sequential Search

Definition 3.1.1: Sequential Search

A brute force search algorithm that scans the list of elements one by one until the desired element is found. The algorithm starts at the beginning of the list and compares each element with the target element until the target element is found or the end of the list is reached. The end of list check can be circumvented by appending the search key to the end of the list, which guarantees that the search will terminate when the search key is found.

Algorithm 3 SequentialSearch (A, n, K)

▷ ▷

▷ Implements sequential search with a search key as a sequential
▷ Input: An array A of n elements and a search key K
▷ Output: The index of the first element in A whose value is equal to K or -1 if no such element is found

```
1: A_n ← K
2: i ← 0
3: while A_i ≠ K do
4:     i ← i + 1
5: end while
6: if i = n then
7:     return -1
8: else
9:     return i
10: end if
```

The time complexity of the sequential search algorithm is $O(n)$, where n is the number of elements in the list

3.2 Brute-Force String Matching

Definition 3.2.1: Brute-Force String Matching

A brute force algorithm that compares the pattern to be matched with the text to be searched one character at a time. The algorithm starts at the beginning of the text and compares the first character of the pattern with the first character of the text. If the characters match, the algorithm compares the second character of the pattern with the second character of the text and so on. If the characters do not match, the algorithm moves to the next character in the text and repeats the process until the entire pattern has been compared to the text. If the pattern is found in the text, the algorithm returns the index of the first character of the pattern in the text. If the pattern is not found in the text, the algorithm returns -1.

Algorithm 4 BruteForceStringMatch (T, P, n, m)

▷

▷ Implements brute-force string matching
▷ Input: An array T of n characters representing a text and and array P of m characters representing a pattern
▷ Output: The index of the firs character in the text that starts a matching substring or -1 if the search is unsuccessful

```
1:  for i ← 0 to n − m do
2:      j ← 0
3:      while j < m and P_j = T_{i+j} do
4:          j ← j + 1
5:      end while
6:      if j = m then
7:          return i
8:      end if
9:  end for
10: return -1
```

Chapter 4

Depth-First Search and Breadth-First Search

4.1 Depth-First Search

Definition 4.1.1: Depth-First Search

Starts at a given vertex, marking it as visited then explores each adjacent vertex in turn. If a tie is encountered, it can be broken arbitrarily. This process continues until a dead end - a vertex with no adjacent unvisited vertices - is reached.
At this point the search backtracks to the most recently visited vertex with unexplored neighbours and continues until it backtracks to the starting vertex. At this point it has visited all the vertices in the same connected component as the starting vertex and if any unvisited vertices remain, the search is restarted at one of them.

A depth-first search can be implemented using a stack to trace the operation of the search. We push a node onto the stack when it is reached for the first time and pop off a node when it becomes a dead end.
A depth-first search is often accompanied by a **depth-first search forest**. The starting node of the traversal serves as the root of the first tree in such a forest. Whenever a new unvisited node is reached for the first time it is attached as a child to the node from which it was reached .Such an edge is called a **tree edge**, because the set of all such edges forms a forest.
DFS is implemented recursilvely as follows:

Algorithm 5 Depth-First Search (G)

▷ Implements a depth-first traversal of a given graph
▷ Input: $G = \langle V, E \rangle$
▷ Output: Graph G with its nodes marked with consecuitve integers in the order they are first encountered by the DFS traversal

```
1:  count ← 0
2:  for each vertex v in V do
3:      if v is marked with 0 then
4:          DFS( v)
5:      end if
6:  end for
7:
8:  function DFS( v)
9:      for each adjacent vertex w to v do
10:         count ← count + 1
11:         mark v with count
12:         if w is marked with 0 then
13:             DFS( w)
14:         end if
15:     end for
16: end function
```

And using a stack:

Algorithm 6 Depth-First Search (G)

▷ Implements a depth-first traversal of a given graph
▷ Input: $G = \langle V, E \rangle$
▷ Output: Graph G with its nodes marked with consecuitve integers in the order they are first encountered by the DFS traversal

```
1:  mark each vertex in V with 0 as a mark of being "unvisited"
2:  count ← 0
3:  S ← empty stack
4:  for each vertex v in V do
5:      if v is marked with 0 then
6:          S ← push v
7:          while S is not empty do
8:              w ← pop from S
9:              if w is marked with 0 then
10:                 count ← count + 1
11:                 mark w with count
12:                 for each adjacent vertex u to w do
13:                     S ← push u
14:                 end for
15:             end if
16:         end while
17:     end if
18: end for
```

The time complexity of DFS depends on the way the graph is represented. If the graph is represented as an adjacency matrix, the time complexity is $O\left(|V|^2\right)$, where V is the set of vertices in the graph, as the algorithm must visit each

possible edge in the graph to determine whether it is adjacent to the current vertex . If the graph is represented as an adjacency list, the time complexity is $O(|V| + |E|)$, as the algorithm must visit each vertex and each edge in the graph.

4.2 Breadth-First Search

Definition 4.2.1: Breadth-First Search

Starts at a given vertex, marking it as visited then explores each vertex adjacent to it, then all unvisited vertices two edges apart from the starting vertex, then three edges apart and so on, until all the vertices in in the same connected component as the starting vertex have been visited.
If there remains unvisited vertices, the search is restarted at one of them.

A Breadth-First search is usually implemented using a queue to trace the operation of the search. The queue is intialised with traversal's staring node and on each iteration the algorithm identifies all the adjacent nodes to the node at the front of the queue, marks them as visited and enqueues them.
At this point the node at the front of the queue is dequeued and the process is repeated until the queue is empty.
Similar to the depth-first search, a breadth-first search is often accompanied by a **breadth-first search tree**. The traversal's stating node becomes the root of the first tree in the forest and whenever a new unvisited node is reached for the first time, the node is attached as a child to the node it is being reached from, with an edge called the **tree edge**. If an edge leading to a previously visited node is encountered, the edge is called a **cross edge**.
Implemented:

Algorithm 7 Breadth-First Search (G)

▷ Implements a breadth-first search traversal of a given graph
▷ Input: Graph $G\langle V, E\rangle$
▷ Output: Graph G with it's vertices marked with consecutive integers in the order they are visited by the BFS traversal

```
1:  mark each vertex in V with 0 as a mark of being "unvisited"
2:  count ← 0
3:  for each v in V do
4:      if v is marked with 0 then
5:          Q ← enqueue v                                   ▷ Where Q is a queue
6:          while Q is not empty do
7:              count ← count + 1
8:              w ← dequeue node from the front of Q
9:              mark w with count
10:             for each vertex u in V adjacent to w do
11:                 if u is marked with 0 then
12:                     Q ← enqueue u
13:                 end if
14:             end for
15:         end while
16:     end if
17: end for
```