Chapter 3: Brute Force

Adequacy is sufficient. (Adam Osborne)

Introduction

Brute force is a straightforward approach to solving a problem without regard for efficiency. Example: an $O(n)$ algorithm for a^n:

algorithm $Power(a, n)$
  // Input: A real number a and an integer $n \geq 0$
  // Output: a^n
  $result \leftarrow 1$
  for $i \leftarrow 1$ **to** n **do**
    $result \leftarrow result * a$
  return $result$

Bubble Sort and Selection Sort 3

Bubble Sort

My bubble sort varies from the book.

algorithm $BubbleSort(A[0..n-1])$
  // Sorts a given array by bubble sort
  // Input: An array A of orderable elements
  // Output: Array $A[0..n-1]$ in ascending order
  $sorted \leftarrow$ **false**
  while $\neg sorted$ **do**
    $sorted \leftarrow$ **true**
    for $j \leftarrow 0$ **to** $n-2$ **do**
      if $A[j] > A[j+1]$ **then**
        swap $A[j]$ and $A[j+1]$
        $sorted \leftarrow$ **false**

Correctness of Bubble Sort

- If A is not sorted, $sorted$ is set to false, and loop continues.
- Once a pair of elements are swapped, they won't be swapped again.
- n elements have $n(n-1)/2$ different pairs, so at most $n(n-1)/2$ swaps, so loop must eventually exit.
- The number of comparisons is $\Theta(n^2)$. See book.

Selection Sort

algorithm $SelectionSort(A[0..n-1])$
  // Sorts a given array by selection sort
  // Input: An array A of orderable elements
  // Output: Array $A[0..n-1]$ in ascending order
  for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
      if $A[j] < A[min]$ **then** $min \leftarrow i$
    swap $A[i]$ and $A[min]$

Efficiency of Selection Sort

- Correct because $A[i]$ is minimum of $A[i..n-1]$.
- The $i = 0$ pass (outer loop iteration) performs $n-1$ comparisons.
- The $i = 1$ pass performs $n-2$ comparisons.
- The last $i = n-2$ pass performs 1 comparison.
- The number of comparisons $C(n)$ is $\Theta(n^2)$.

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

Brute-Force String Matching

algorithm $BruteForceStringMatch$
$(T[0..n-1], P[0..m-1])$
// Implements brute-force string matching
// Input: Text array T and pattern array P
// Output: The index where P is found or -1
for $i \leftarrow 0$ **to** $n-m$ **do**
$j \leftarrow 0$
while $j < m$ **and** $P[j] = T[i+j]$ **do**
$j \leftarrow j+1$
if $j = m$ **then return** i
return -1

Analysis of Brute-Force String Matching

- Correct because every possible index i is checked.
- $i = n - m$ is the highest possible index.
- At worst, m comparisons are made for a given value of i.
- There are $n - m + 1$ values for i.
- The number of comparisons is $\leq m(n-m+1) \leq mn \in O(mn)$.

Closest Pair Problem

- A point (2D case) is an ordered pair of values (x, y).
- The (Euclidean) distance between two points $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is:

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

- The closest-pair problem is finding the two closest points in a set of n points.
- The brute force algorithm checks every pair of points, which will make it $\Theta(n^2)$.
- We can avoid computing square roots by using squared distance.

Closest-Pair Brute-Force Algorithm

algorithm $BruteForceClosestPair(P)$
// Finds two closest points by brute force
// Input: A list P of $n \geq 2$ points
// Output: The indices of the closest pair
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n-1$ **do**
for $j \leftarrow i+1$ **to** n **do**
$d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$
if $d < dmin$ **then**
$dmin \leftarrow d$
$imin \leftarrow i$
$jmin \leftarrow j$
return $imin$, $jmin$

The Convex Hull Problem

- A region (set of points) in the plane is *convex* if every line segment between two points in the region is also in the region.
- The *convex hull* of a finite set of points P is the smallest convex region containing P.
- Theorem: The *convex hull* of a finite set of points P is a convex polygon whose vertices is a subset of P.
- The *convex hull problem* is finding the convex hull given P.

Examples of Convex Sets

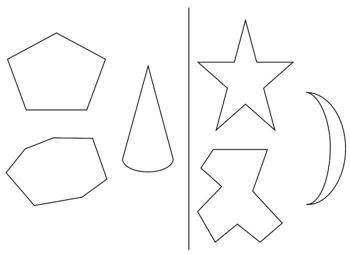
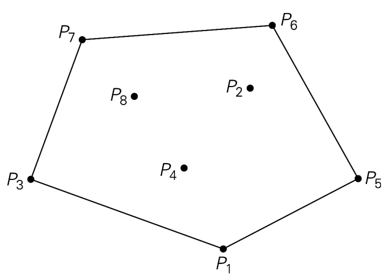


FIGURE 3.4 (a) Convex sets. (b) Sets that are not convex.

Example of Convex Hull



Idea for Solving Convex Hull

- Consider the straight line that goes through two points P_i and P_j.
- Suppose there are points in P on both sides of this line.
  - This implies that the line segment between P_i and P_j is not on the boundary of the convex hull.
- Suppose all the points in P are on one side of the line (or on the line).
  - This implies that the line segment between P_i and P_j is on the boundary of the convex hull.

Development of Idea for Convex Hull

- The straight line through $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ can be defined by a nonzero solution for:

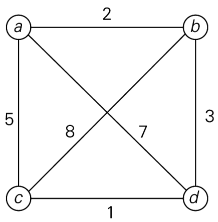
$$\begin{aligned} a\,x_i + b\,y_i &= c \\ a\,x_j + b\,y_j &= c \end{aligned}$$

- One solution is $a = y_j - y_i$, $b = x_i - x_j$, and $c = x_i y_j - y_i x_j$.
- The line segment from P_i to P_j is on the convex hull if either $a\,x + b\,y \geq c$ or $a\,x + b\,y \leq c$ is true for all the points.
- Brute force algorithm is $\Theta(n^3)$. See book.

Exhaustive Search

- *Exhaustive search* generates all combinatorial objects (e.g., permutations, combinations, subsets) for a problem.
- The *traveling salesman problem (TSP)* is finding the shortest tour through n cities.
  - Brute Force Approach: Calculate the distance of all cycles of n vertices in a weighted graph.
- The *knapsack problem* is finding the most valuable subset of items $\leq$ a given weight.
  - Brute Force Approach: Calculate the value and weight of all subsets.

TSP Example

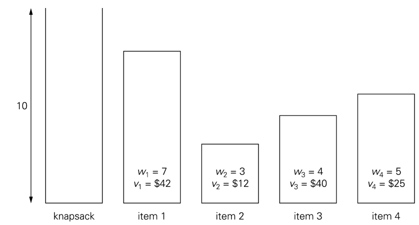


TSP Solution

Tour	Length	
a —> *b* —> *c* —> *d* —> *a*	*l* = 2 + 8 + 1 + 7 = 18	
a —> *b* —> *d* —> *c* —> *a*	*l* = 2 + 3 + 1 + 5 = 11	optimal
a —> *c* —> *b* —> *d* —> *a*	*l* = 5 + 8 + 3 + 7 = 23	
a —> *c* —> *d* —> *b* —> *a*	*l* = 5 + 1 + 3 + 2 = 11	optimal
a —> *d* —> *b* —> *c* —> *a*	*l* = 7 + 3 + 8 + 5 = 23	
a —> *d* —> *c* —> *b* —> *a*	*l* = 7 + 1 + 8 + 2 = 18	

FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search

Knapsack Problem Example



(a)

Knapsack Problem Solution

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

FIGURE 3.8 (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. (The information about the optimal selection is in bold.)