Analysis and Design of Algorithms

Chapter 3: Brute Force

School of Software Engineering © Ye Luo

Brute Force

Brute Force

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

- ***Example:***
  - *Computing a^n ($a > 0$, n and a are nonnegative integer)*
  - *Computing $n!$*
  - *Multiplying two matrices*
  - *Searching for a key of a given value in a list*
  - *Consecutive Integer Algorithm for gcd (m,n)*

Brute-Force Sorting Alg. — Selection Sort

Idea of Selection Sort

- *Problem*

  Given an array of *n* orderable items (e.g. numbers, characters from some alphabet, character strings), rearrange them in non-decreasing order

Selection Sort

- *Idea*
  - Scan the entire array to find its smallest element and swap it with the first element. — put the smallest element in its final position in the sorted array
  - Starting with the second element, to find the smallest among the next n-1 elements and swap it with the second element. — put the second smallest element in its final position in the sorted array
  - Generally, on pass i ($0 \leq i \leq n-2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:
  - After n-1 passes, the array is sorted

$$A[0] \leq \;.\;\;.\;\;.\; \leq A[i-1] \mid A[i], \;.\;\;.\;\;.\;, A[min], .\;\;.\;\;., A[n-1]$$

in their final positions

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 $A[min]$

Selection Sort

- ***Example:***

Selection Sort on the list {89, 45, 68, 90, 29, 34, 17 }

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

FIGURE 3.1 Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Selection Sort

Analysis of Selection Sort

- ***Basic operation: key comparison A[j] < A[min]***
- ***Input size: number of elements, n***
- ***Time efficiency*** $\Theta(n^2)$

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

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

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

- *Number of key swaps:* $\Theta(n)$

Brute-Force Sorting Alg. — Bubble Sort

Idea of Bubble Sort

Idea

- Compare adjacent elements of the list and exchange them if they are out of order
- By doing it repeatedly, we end up “bubbling” the largest element to the last position on the list
- The next past bubbles up the second largest element, and so on until, after n-1 passes, the list is sorted
- Pass i

$$A_0 \ldots\ldots A_j \overset{?}{\longleftrightarrow} A_{j+1} \ldots\ldots A_{n-i-1} \mid A_{n-i} \le \ldots \le A_{n-1}$$

in their final positons

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 ascending 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]
}
```

Bubble Sort

- ***Example:***

Bubble Sort on the list {89, 45, 68, 90, 29, 34, 17 }

89	$\overset{?}{\longleftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\longleftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\longleftrightarrow}$	90	$\overset{?}{\longleftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\longleftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\longleftrightarrow}$	17
45		68		89		29		34		17	$\mid$	90

45	$\overset{?}{\longleftrightarrow}$	68	$\overset{?}{\longleftrightarrow}$	89	$\overset{?}{\longleftrightarrow}$	29		34		17	$\mid$	90
45		68		29		89	$\overset{?}{\longleftrightarrow}$	34		17	$\mid$	90
45		68		29		34		89	$\overset{?}{\longleftrightarrow}$	17	$\mid$	90
45		68		29		34		17	$\mid$	89		

Bubble Sort

Analysis of Bubble Sort

- ***Basic operation: key comparison***
- ***Input size: number of elements, n***
- ***Time efficiency*** $\Theta(n^2)$

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

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

- *number of key swaps: depends on the input*

$$S_{worst}(n) = C(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Thinking: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm

Exhaustive Search

- ***Problem***

  Searching for an element with a special property, in a domain that grows exponentially (or faster) with an instance size,

  Usually involve combinatorial objects such as permutations, combinations, or subsets of a set.

  Many such problems are optimization problems, to find an element that maximizes or minimizes some desired characteristic

  such as a path's length or an assignment's cost

Exhaustive Search

- ***Exhaustive Search— Brute-Force for combinatorial***
  - Generate a list of **all potential solutions** to the problem in a systematic manner
  - Selecting those of them that satisfy all the constraints
  - Evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
  - Then search ends, announce the desired solution(s) found (e.g. the one that optimizes some objective function)
  - *typically requires for generating certain combinatorial objects*

Exhaustive Search: Traveling Salesman Problem

Idea

- *Problem*

  Given *n* cities with known distances between each pair, find the shortest tour that passes through <u>all</u> the cities <u>exactly once</u> before returning to the starting city

- *Idea*

  - weighted graph:

    vertices: cities

    edge weights: distances

  - Alternatively: To find shortest Hamiltonian circuit in a weighted connected graph

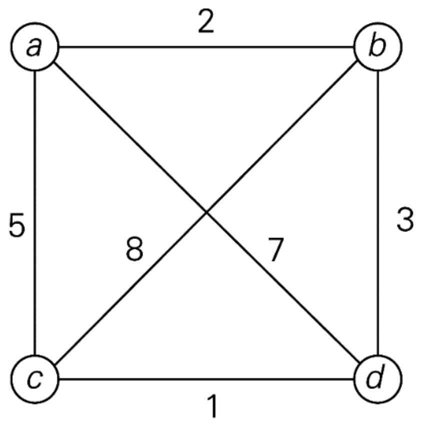
  Hamiltonian circuit: a cycle that passes through all the vertices of the graph exactly once

Exhaustive Search: Traveling Salesman Problem

- *Idea*
  - Hamiltonian circuit can be defined as a sequence of n+1 adjacent vertices $v_{i0}, v_{i1}, v_{i2}, \ldots\ldots, v_{in-1}, v_{i0}$
  - Generating all the permutations of n-1 intermediate cities
  - Computing the tour lengths
  - Find the shortest among them

Traveling Salesman Problem

- ***Example:***



Tour	Length	
$a \longrightarrow b \longrightarrow c \longrightarrow d \longrightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \longrightarrow b \longrightarrow d \longrightarrow c \longrightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \longrightarrow c \longrightarrow b \longrightarrow d \longrightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \longrightarrow c \longrightarrow d \longrightarrow b \longrightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \longrightarrow d \longrightarrow b \longrightarrow c \longrightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \longrightarrow d \longrightarrow c \longrightarrow b \longrightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Traveling Salesman Problem

Analysis of Exhaustive Search for TSP

- *number of permutations* $(n\text{-}1)!$

Exhaustive Search: Knapsack Problem

Idea

- *Problem*

Given

weights: $w_1 \quad w_2 \ \ldots \ w_n$
values: $v_1 \quad v_2 \ \ldots \ v_n$
a knapsack of capacity W

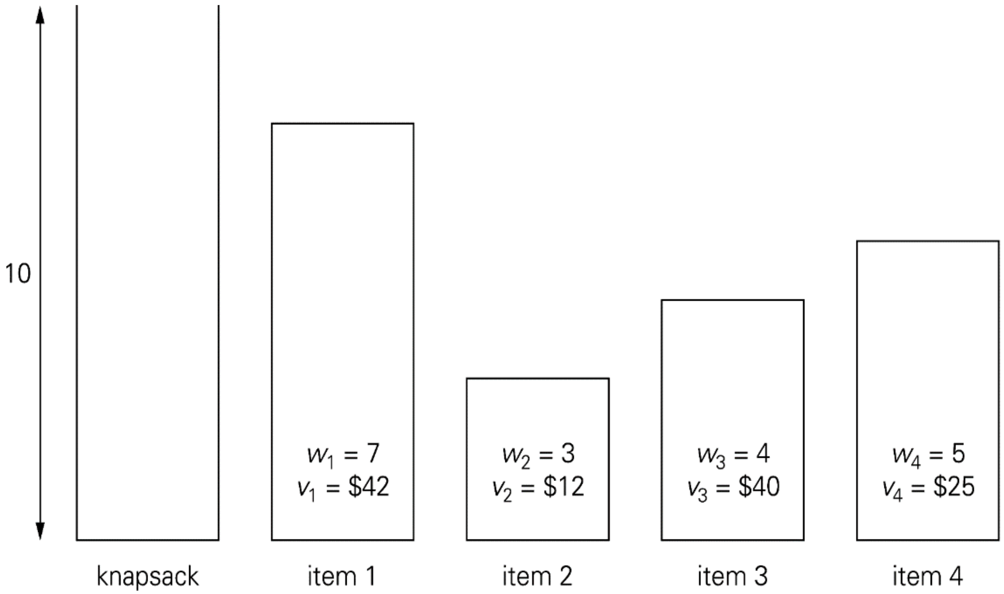
find the most valuable subset of the items that fit into the knapsack

Exhaustive Search: Knapsack Problem

- *Idea*
  - generating all subsets of the set of *n* items given
  - computing the total weight of each feasible subset (i.e. the ones with the total weight not exceeding the knapsack's capacity)
  - finding a subset of the largest value among them

Knapsack Problem

- ***Example:***



(a)

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)

Knapsack Problem

Analysis of Exhaustive Search for Knapsack

- *number of subsets for an n-element set* 2^n

For Exhaustive Search for Knapsack Problem and TSP problem,

- *examples of so-called NP-hard problem*
- *no polynomial-time algorithm is known for NP-hard problem*

Exhaustive Search: Assignment Problem

- ***Idea***
  - *Problem*

    There are n people who need to be assigned to n jobs, one person per job.

    Each person is assigned to exactly one job, and each job is assigned to exactly one person

    The cost of assigning person i to job j is $C[i, j]$

    Find an assignment that minimizes the total cost.

Exhaustive Search: Assignment Problem

- *Idea*

describe the feasible solutions to the Assignment Problem as n-tuples $<j_1, \ldots, j_n>$ *in which the i-th component indicates the column of the element selected in the i-th row (i.e. job number assigned to the i-th person)*

- Generating all legitimate assignments,
- Compute their costs
- Select the cheapest one

Assignment Problem

- ***Example:***

	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

Pose the problem as the one about a cost matrix:

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

$\langle 1, 2, 3, 4 \rangle$ cost = 9 + 4 + 1 + 4 = 18

$\langle 1, 2, 4, 3 \rangle$ cost = 9 + 4 + 8 + 9 = 30

$\langle 1, 3, 2, 4 \rangle$ cost = 9 + 3 + 8 + 4 = 24

$\langle 1, 3, 4, 2 \rangle$ cost = 9 + 3 + 8 + 6 = 26

$\langle 1, 4, 2, 3 \rangle$ cost = 9 + 7 + 8 + 9 = 33

$\langle 1, 4, 3, 2 \rangle$ cost = 9 + 7 + 1 + 6 = 23

etc.

Assignment Problem

Analysis of Exhaustive Search for Assignment

- *number of permutations* $n!$

  - *no known polynomial-time algorithms for problems whose domain grows exponentially with instance size*

Summary

- 蛮力法是一种简单直接地解决问题的方法，通常直接基于问题的描述和所涉及的概念定义
- 蛮力法的优点：广泛适用性和简单性；蛮力法的缺点：大多效率低
- 蛮力法一般是得到一个算法，为设计改进算法提供比较依据
- 穷举法是蛮力法之一，包括旅行商问题，背包问题和分配问题。