COP 3530: Computer Science III
Summer 2005

Brute Force Algorithms

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
CSB 242, (407)823-2790

Course Webpage:
http://www.cs.ucf.edu/courses/cop3530/sum2005

School of Computer Science
University of Central Florida

What Is A Brute Force Algorithm?

- Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved.
- The "force" implied by the strategy's definitions is that of a computer and not that of one's intellect. "Just do it!" would be another way to describe the brute-force approach.
- As an example, consider computing a^n for a given number *a* and a nonnegative integer *n*.
  - By the definition of exponentiation, $a^n = \underbrace{a \times a \times \ldots \times a}_{n \text{ times}}$

Brute Force Algorithms (cont.)

- Can you think of other algorithms that you are familiar with that use the brute force strategy?
  - Bubble Sort
  - Selection Sort
  - Matrix Multiplication (using algorithm based strictly on definition of matrix multiplication)

```
Algorithm MatrixMult( A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])
Input: Two n X n matrices A and B
Output: Matrix C = A × B
for i = 0 to n-1 do
     for j = 0 to n-1 do
         C[i,j]= 0.0
         for k = 0 to n-1 do
              C[i,j] = C[i,j] + A[i,k] × B[k,j]
return C
```


Brute Force Algorithms (cont.)

- Though rarely a source of clever or efficient algorithms, the brute force approach should not be overlooked as an important algorithm design strategy for five main reasons.

1. Unlike some of the other strategies that we will see over the course of this semester, brute force is applicable to a very wide variety problems.
   - In fact, it seems to be the only general approach for which it is more difficult to point out problems that it cannot tackle!
2. For some important problems, such as sorting and searching, brute force yields reasonable algorithms with some practical value that have no limitation on instance size.
3. The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute force algorithm can solve those instances with acceptable speed.
4. Even if it is too inefficient in general, a brute force algorithm can still be useful for solving small-size instances of a problem.
5. A brute force algorithm can serve as an important theoretical or education purpose, e.g., as a benchmark to judge more efficient alternatives for solving a problem.

Exhaustive Search

- Many important problems require finding an element with a special property in a domain that grows exponentially (or faster) with instance size.
- Typically, such problems arise in situations that involve – explicitly or implicitly – combinatorial objects such as permutations, combinations, and subsets of a given set. Many such problems are optimization problems: they ask to find an element that maximizes or minimizes some desired characteristic such as a path's length or an assignment's cost.
- Exhaustive search is simply a brute force approach to a combinatorial problem.

Exhaustive Search (cont.)

- Exhaustive search suggests generating each and every element of the problem's domain, selecting those of them that satisfy the problem's constraints, and then finding a desired element (e.g., the one that optimizes some objective function).
- The assignment problem is one problem to which an exhaustive search can be illustratively applied.

The assignment problem is one of *n* people who need to be assigned to execute *n* tasks, one person per task (and one task per person). The cost that would accrue if the *i* th person is assigned to the *j* th task is the quantity C[*i,j*] for each pair *i, j* = 1…n. The problem is to find the assignment with the smallest total cost.

An Assignment Problem Example

	Task 1	Task 2	Task 3	Task 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

An Assignment Problem Example (cont.)

- It is easy to see that an instance of the assignment problem is completely specified by its cost matrix *C*.
- In terms of this matrix, the problem calls for a selection of one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.
- Note that no obvious strategy for finding a solution works here. For example, you cannot select the smallest element in each row because the smallest elements may happen to be in the same column. In fact, the smallest element in the matrix may not be part of the optimal solution.

An Assignment Problem Example (cont.)

- We can describe a feasible solution 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., the task number assigned to the *i* th person).
- For example, using the cost matrix on page 7, the assignment <2, 3, 4, 1> indicates a feasible assignment of person 1 to task 2, person 2 to task 3, person 3 to task 4, and person 4 to task 1.
- The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first *n* integers.

An Assignment Problem Example (cont.)

- Therefore, the exhaustive search approach to the assignment problem would require generating all the permutations of integers 1, 2,…, n, computing the total cost of each assignment by summing up the corresponding elements in the cost matrix, and finally selecting the one with the smallest sum.
- The solution to our problem instance is shown on the next page.

An Assignment Problem Example (cont.)

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18	<2, 1, 3, 4> cost = 2 + 6 + 1 + 4 = 13 (optimal)
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30	<2, 1, 4, 3> cost = 2 + 6 + 8 + 9 = 25
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24	<2, 3, 1, 4> cost =2 + 3 + 5 + 4 = 14
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26	<2, 3, 4, 1> cost = 2 + 3 + 8 + 7 = 20
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33	<2, 4, 1, 3> cost = 2 + 7 + 5 + 9 = 23
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23	<2, 4, 3, 1> cost = 2 + 7 + 1 + 7 = 17
<3, 1, 2, 4> cost = 7 + 6 + 8 + 4 = 25	<4, 1, 2, 3> cost = 8 + 6 + 8 + 9 = 31
<3, 1, 4, 2> cost = 7 + 6 + 8 + 6 = 27	<4, 1, 3, 2> cost = 8 + 6 + 1 + 6 = 21
<3, 2, 1, 4> cost = 7 + 4 + 5 + 4 = 20	<4, 2, 1, 3> cost = 8 + 4 + 5 + 9 = 26
<3, 2, 4, 1> cost = 7 + 4 + 8 + 7 = 26	<4, 2, 3, 1> cost = 8 + 4 + 1 + 7 = 20
<3, 4, 1, 2> cost = 7 + 7 + 5 + 6 = 25	<4, 3, 1, 2> cost = 8 + 3 + 5 + 6 = 22
<3, 4, 2, 1> cost = 7 + 7 + 8 + 7 = 29	<4, 3, 2, 1> cost = 8 + 3 + 8 + 7 = 26

Summary of Exhaustive Search

- Since the number of permutations to be considered for the general case of the assignment problem is n!, exhaustive search is impractical for all be very small instances of the problem.
- Fortunately, there is a much more efficient algorithm for this problem called the Hungarian method after the Hungarian mathematicians König and Egerváry whose work underlies the method.
- The good news is that just because a problem's domain grows exponentially (or faster) does not necessarily imply that there can be no efficient algorithm for solving it.
- The bad news is that this problem is more of an exception from the rule. More often than not, there are no known polynomial time algorithms for problems whose domains grow exponentially (assuming an exact solution is required). It is possible that such algorithms do not exist.