COMPUTATIONAL COMPLEXITY

author

This Hypercard stack was prepared by:
Dennis L. Bricker,
Dept. of Industrial Engineering,
University of Iowa,
Iowa City, Iowa 52242
e-mail: dbricker@icaen.uiowa.edu

How can we compare two different algorithms for the same problem?

- QUALITY OF SOLUTION
- COMPUTATIONAL EFFICIENCY

A measure often used to measure computational efficiency is computer execution time (cpu time)

... but cpu time depends upon

- type of computer
- programming language
- programmer skills
- etc.

PRINCIPLE OF INVARIANCE

The principle of invariance says that two different implementations of the *same* algorithm will not differ in computational efficiency by more than a multiplicative constant

If two implementations of the same algorithm, which may differ in programming language &/or machine used, take $\mathbf{t}_1(\mathbf{n})$ and $\mathbf{t}_2(\mathbf{n})$ seconds for an instance of size n, then there exists a $\mathbf{c} > 0$ and integer $\mathbf{N}$ such that $\mathbf{t}_1(\mathbf{n}) \leq \mathbf{c}\, \mathbf{t}_2(\mathbf{n})$ for all $\mathbf{n} \geq \mathbf{N}$.

One appropriate way to measure the computational efficiency is to count the number of elementary operations that are required by the algorithm, i.e., additions, subtractions, multiplications, divisions, comparisons, etc.

Specifically, we compute the "worst-case" number of elementary operations, which may be quite different from the "typical-case" problem encountered in actual applications.

A more "macro" view would count the number of iterations that the algorithm must perform as a function $t(n)$ of the size n of the problem if the computational effort per iteration is stable, e.g., bounded by some function of n.

We say that an algorithm takes time *of the order* $t(n)$, where t is a given function, if there exists a $c > 0$ and an implementation of the algorithm capable of solving *every* instance of the problem of size n in a time bounded by $ct(n)$.

This is denoted $O(t(n))$ and is called the *time complexity* of the algorithm.

EXAMPLE Dykstra's Shortest Path Algorithm

Denote: n = # nodes

k = current stage (#permanent labels)

so (n-k) = # of temporary labels.

At stage k ($1 \leq k \leq n$),

3 operations are required for each temporary label:

1 addition & 1 comparison for updating

1 comparison for selecting label to be made permanent

Total:

$$t(n) = \sum_{k=1}^{n} 3(n-k) = 3\sum_{k=1}^{n} n - 3\sum_{k=1}^{n} k$$

$$= 3\ n^2 - 3\ n \times \frac{n}{2} = {}^{3}\!/_{2}\ n^2$$

That is, the algorithm is $O(n^2)$

Polynomial Time Algorithm

An algorithm for which the time (equivalently, the number of operations) is $O(p(n))$, i.e., proportional to $p(n)$, where $p(n)$ is a polynomial function and n is the "size" of the problem, is called a *polynomial time* algorithm

Exponential Time Algorithm

An algorithm which is not "polynomial time" is usually referred to as an *exponential time* algorithm.

Example: Balas' implicit enumeration algorithm
In the worst-case scenario, no node of the enumeration tree is fathomed, and all 2^n completions are *explicitly* enumerated, so that the algorithm is $O(2^n)$.

The importance of the distinction between polynomial time & exponential time algorithms is evident in the following table, which gives cpu times for various problem sizes

While the computational burden may not be significantly different for "small" n, as n increases the differences become dramatic!

Com – plexity	Size of problem (n)					
	10	20	30	40	50	60
$O(n)$	0.00001 sec.	0.00002 sec.	0.00003 sec.	0.00004 sec.	0.00005 sec.	0.00006 sec.
$O(n^2)$	0.0001 sec.	0.0004 sec.	0.0009 sec.	0.0016 sec.	0.0025 sec.	0.0036 sec.
$O(n^3)$	0.001 sec.	0.008 sec.	0.027 sec.	0.064 sec.	0.125 sec.	0.216 sec.
$O(n^5)$	0.1 sec.	3.2 sec.	24.3 sec.	1.7 min.	5.2 min.	13 min.
$O(2^n)$	0.001 sec.	1 sec.	17.9 min	12.7 days	35.7 yr	36.6 centuries
$O(3^n)$	0.059 sec.	58 min	6.5 yr	3855 centuries	2x10 centuries	1.3x10 centuries

Suppose computer can perform 10 operations/sec.

CLASSIFICATION OF PROBLEMS

P = set of all problems which are solvable by a polynomial time algorithm

NP = set of all problems which are solvable by a "nondeterministic" polynomial time algorithm

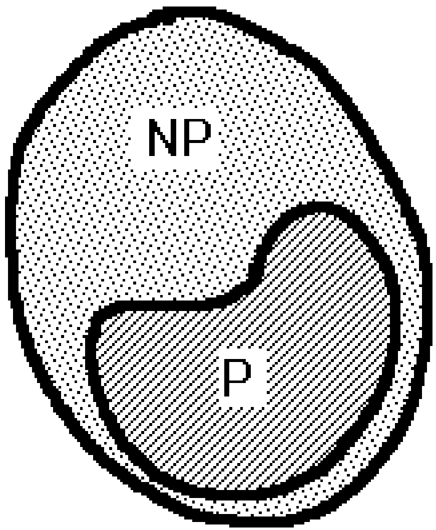
i.e., for which a "guess" can be evaluated in polynomial time (practically all problems of interest)

For example, the shortest route problem is a member of the set P.

Clearly, $\boxed{P \subseteq NP}$

That is, if a problem can be solved in polynomial time, then it is certainly possible to evaluate its objective function for a candidate solution in polynomial time.

$$P \subseteq NP$$



Is $P \neq NP$?

That is, does there exist a problem for which no polynomial time algorithm can never be found?

TSP (the traveling salesman problem) is certainly in NP, and at this time no one has been able to design a polynomial time algorithm for its solution. Conversely, no one has been able to prove the nonexistence of a polynomial time TSP algorithm!

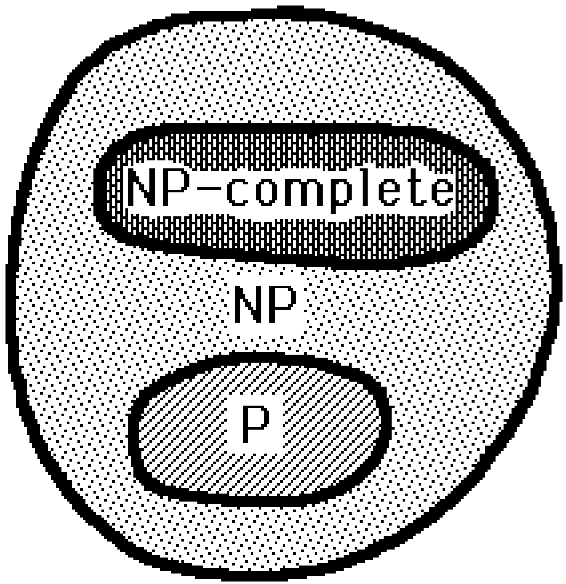
Conjecture: $\boxed{P \neq NP}$

This is still an open question, although most mathematicians/computer scientists believe it to be true.

It *has* been proved that

$$\boxed{\textit{If} \quad P \neq NP \;, \textit{ then } \; TSP \notin P}$$

that is, if there exist problems for which no polynomial time algorithms can be found, the traveling salesman problem is one such problem.



A problem is called NP-Complete if all problems in NP can be reduced in polynomial time to that problem.

It is known that the TSP is NP-complete....
if, therefore, it is ever shown that TSP ∈ P, then NP = P.

Caveat:

The fact that no polynomial time algorithm is known for the TSP does not imply that no such algorithm exists!

Until the publication in 1979 by L.G. Khachian of his "Ellipsoid" algorithm for linear programming, no polynomial time algorithm was known for LP!

The Simplex method for LP is NOT polynomial time in the worst case, in which every basic feasible solution is encountered!