Copyrighted material - for classroom use only

Note Title 5/7/2013

NP-Complete Problems

Only for CSC501 & CSC401 at CSUDH

P and *NP*

NP-Complete Problems

Approximation Algorithms

Bin Packing

The Knapsack and Subset Sum Problems

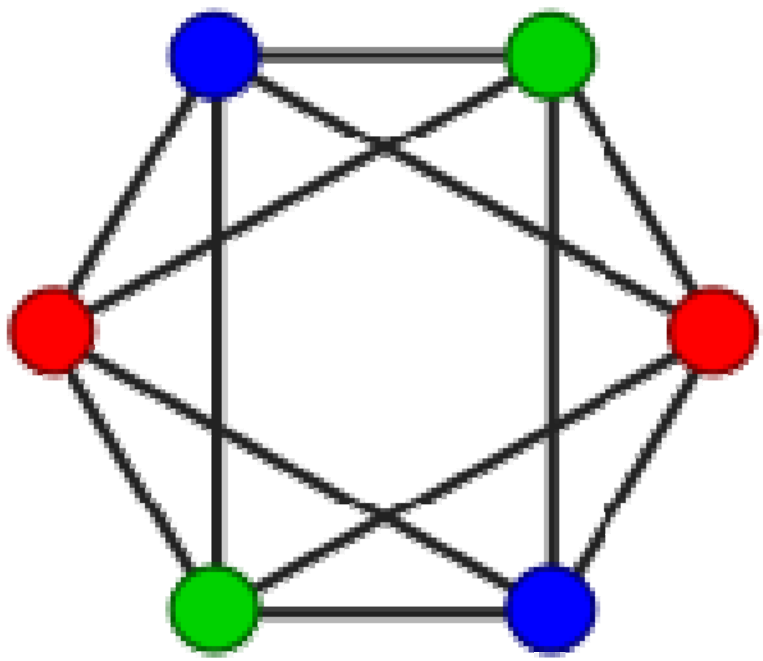
Graph Coloring

Graph Coloring

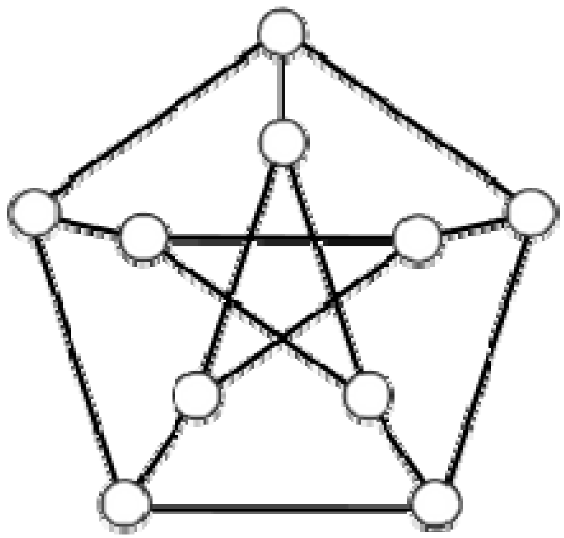
A *coloring* of a graph $G = (V, E)$ is a mapping $C:V \rightarrow S$, where S is a finite set (of "colors"), such that if $vw \in E$ then $C(v) \neq C(w)$; in other words, adjacent vertices are not assigned the same color. The *chromatic number* of G, denoted $\chi(G)$, is the smallest number of colors needed to color G, that is, the smallest k such that there exists a coloring C for G and $|C(V)| = k$.

Optimization problem: Given G, determine $\chi(G)$ (and produce an optimal coloring, i.e., one that uses only $\chi(G)$ colors).

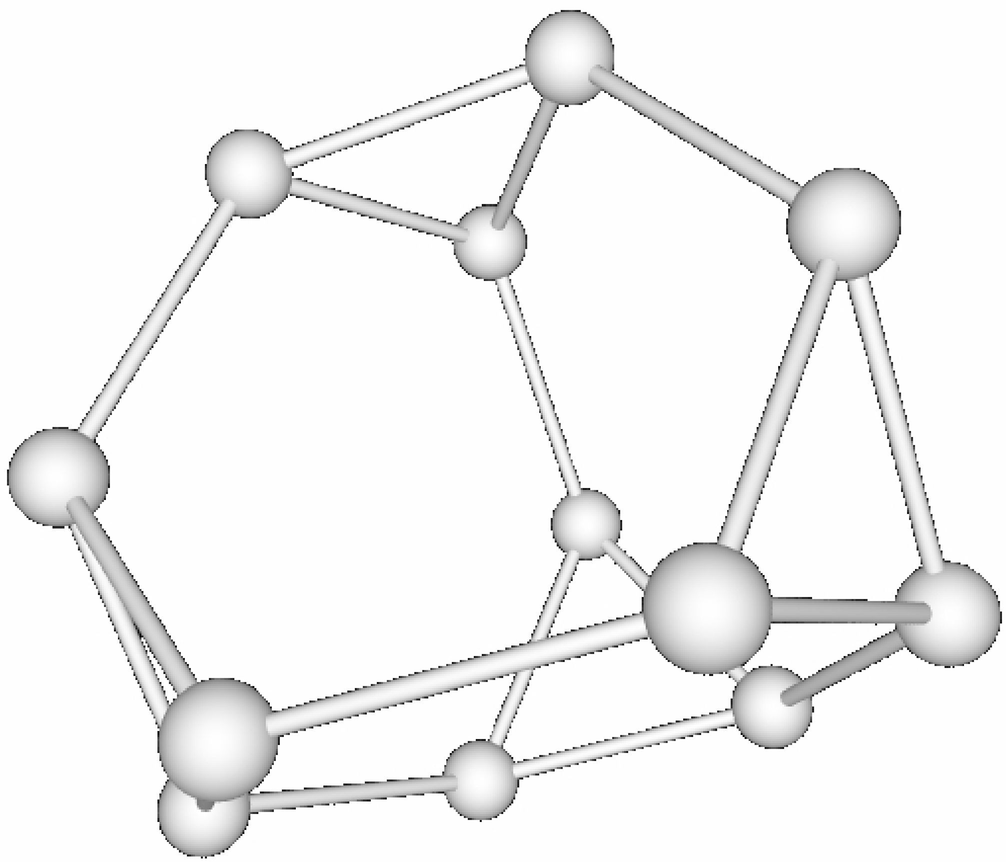
Decision problem: Given G and a positive integer k, is there is a coloring of G using at most k colors? (If so, G is said to be *k-colorable*.)



An example of graph that is 3-colorable but not 2-colorable



Another example



Job Scheduling with Penalties

Suppose that n jobs $J_1, \ldots, J_n$ are to be executed one at a time. We are given execution times $t_1, \ldots, t_n$, deadlines $d_1, \ldots, d_n$ (measured from the starting time for the first job executed), and penalties for missing the deadlines $p_1, \ldots, p_n$. Assume that the execution times, deadlines, and penalties are all positive integers. A schedule for the jobs is a permutation π of $\{1, 2, \ldots, n\}$, where $J_{\pi(1)}$ is the job done first, $J_{\pi(2)}$ is the job done next, and so forth. The total penalty for a particular schedule is

$$P_\pi = \sum_{j=1}^{n} \left[\textbf{if } t_{\pi(1)} + \cdots + t_{\pi(j)} > d_{\pi(j)} \textbf{ then } p_{\pi(j)} \textbf{ else } 0 \right].$$

Optimization problem: Determine the minimum possible penalty (and find an optimal schedule, i.e., one that minimizes the total penalty).

Decision problem: Given, in addition to the inputs described, a nonnegative integer k, is there a schedule with $P_\pi \leq k$?

Bin Packing

Suppose we have an unlimited number of bins each of capacity 1, and n objects with sizes $s_1, \ldots, s_n$, where $0 < s_i \leq 1$.

Optimization problem: Determine the smallest number of bins into which the objects can be packed (and find an optimal packing).

Decision problem: Given, in addition to the inputs described, an integer k, do the objects fit in k bins?

Knapsack

Suppose we have a knapsack of capacity C (a positive integer) and n objects with sizes $s_1, \ldots, s_n$ and "profits" $p_1, \ldots, p_n$ (where $s_1, \ldots, s_n$ and $p_1, \ldots, p_n$ are positive integers).

Optimization problem: Find the largest total profit of any subset of the objects that fits in the knapsack (and find a subset that achieves the maximum profit).

Decision problem: Given k, is there a subset of the objects that fits in the knapsack and has a total profit at least k?

Subset Sum

The input is a positive integer C and n objects whose sizes are positive integers $s_1, \ldots, s_n$.

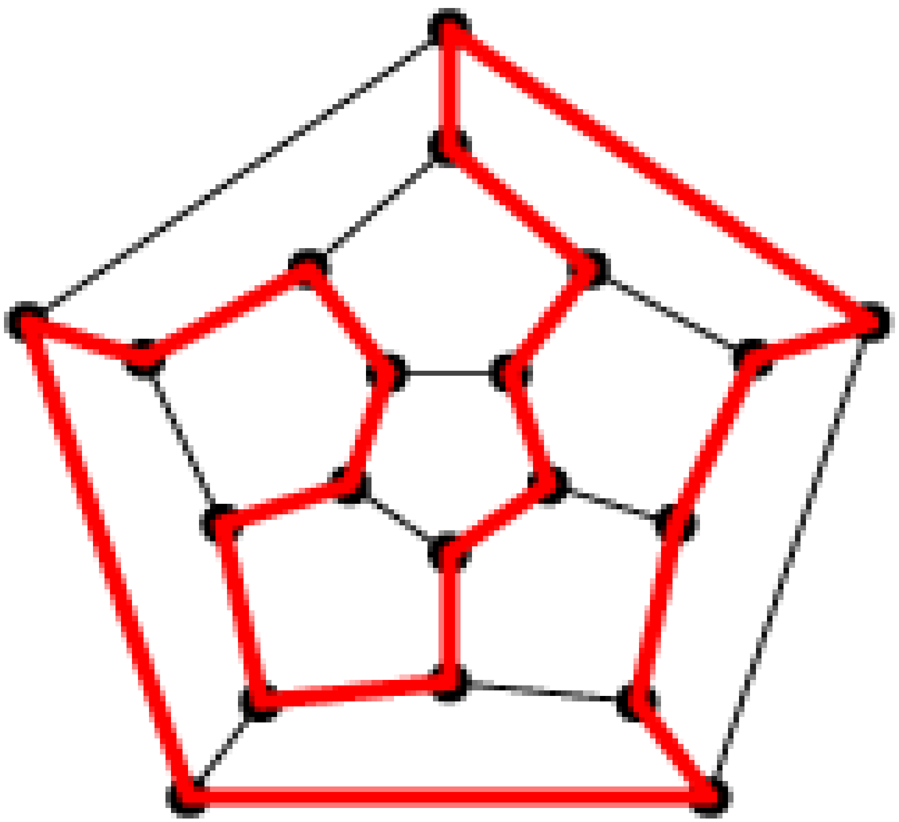
Optimization problem: Among subsets of the objects with sum at most C, what is the largest subset sum?

Decision problem: Is there a subset of the objects whose sizes add up to exactly C?

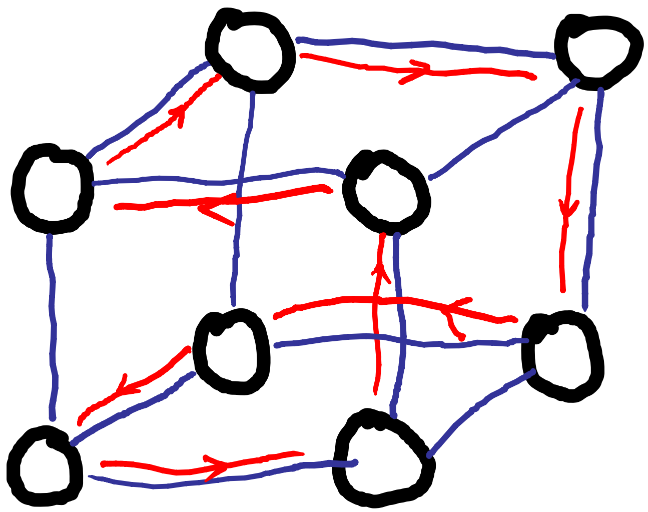
Hamilton Paths and Hamilton Circuits

A Hamilton path (Hamilton circuit, or cycle) in a graph or digraph is a path (cycle) that passes through every vertex exactly once. (*Circuit* is another term for *cycle*, and Hamilton cycles are most commonly called Hamilton circuits.)

Decision problem: Does a given graph or digraph have a Hamilton path (circuit)?



3 - cube



does have a Hamiltonian cycle.

And so does any n - cube

Minimum Tour (Traveling Salesman Problem)

Optimization Problem: Given a weighted graph, find a minimum weighted Hamilton circuit.

This problem is widely known as the traveling salesperson problem; the salesperson wants to minimize total traveling while visiting all the cities in a territory. Other applications include routing trucks for garbage pickup and package delivery.

Decision Problem: Given a weighted graph and an integer k, is there a Hamilton circuit with total weight at most k?

CNF-satisfiability

A logical (or Boolean) variable is a variable that may be assigned the value *true* or *false*. If v is a logical variable, then $\bar{v}$, the negation of v, has the value *true* if and only if v has the value *false*. A *literal* is a logical variable or the negation of a logical variable. A *clause* is a sequence of literals separated by the Boolean **or** operator ($\vee$). A logical expression in *conjunctive normal form* (*CNF*) is a sequence of clauses separated by the Boolean **and** operator ($\wedge$). An example of a logical expression in CNF is

$$(p \vee q \vee s) \wedge (\bar{q} \vee r) \wedge (\bar{p} \vee r) \wedge (\bar{r} \vee s) \wedge (\bar{p} \vee \bar{s} \vee \bar{q}),$$

where p, q, r, and s are logical variables.

Decision problem: Is there a truth assignment, i.e., a way to assign the values *true* and *false*, for the variables in the expression so that the expression has value *true*?

Excercise

Is this CNF formula satisfiable?

$$(p \vee q \vee s) \wedge (\bar{q} \vee r) \wedge (\bar{p} \vee r) \wedge (\bar{r} \vee s) \wedge (\bar{p} \vee \bar{s} \vee \bar{q}) \wedge (q \vee \bar{s})$$

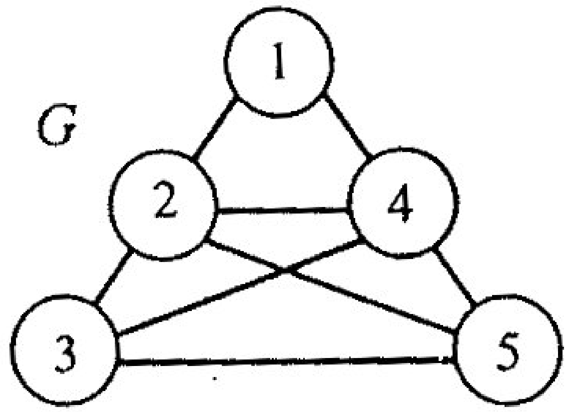
Answer this question in two ways.

1. By brute force – try all truth assignments
2. By logic + brute force – first, simplify the formula using logic and then try truth assignments.

You should see that way 2 is faster than way 1.

Example 1 Nondeterministic graph coloring

Let $k = 4$ and let the graph G in Fig. .1 be the input. For simplicity, we will denote colors by letters B (blue), R (red), G (green), O (orange), and Y (yellow). (This is not a satisfactory notation in general because large graphs may need more than 26 colors.) Here is a list of a few possible strings s and the output that would result.



Input: 4, 5, (1,2) (1,4) (2,4) (2,3) (3,5) (2,5) (3,4) (4,5)

k — number of vertices — edges of *G*

Input for nondeterministic graph coloring

s	Output	Reason
RGRBG	*no*	v_2 and v_5, both green, are adjacent
RGRB	*no*	Not all vertices are colored
RBYGO	*no*	More than k colors
RGRBY	*yes*	A valid 4-coloring
R%*,G@	*no*	Bad syntax

Since there is one possible computation of the algorithm that produces a *yes* output, the answer for the input G is *yes*.

The Class *P*

Definition *P* is the class of decision problems that are polynomial bounded.

$q \in P$ iff $\exists$ program s that solves (or answers) q, and
$\exists$ constant k, such that:
s solves problem q and the worst-case running time t of s satisfies
$t(n) \in O(n^k)$
where n is the size of an input to q (and s)

The Class *NP*

A *nondeterministic algorithm* has two phases:

1. The nondeterministic phase. Some completely arbitrary string of characters, *s*, is written beginning at some designated place in memory. Each time the algorithm is run, the string written may differ. (This string may be thought of as a guess at a solution for the problem, so this phase may be called the guessing phase, but *s* could just as well be gibberish.)
2. The deterministic phase. A deterministic (i.e., ordinary) algorithm begins execution. In addition to the decision problem's input, the algorithm may read *s*, or it may ignore *s*. Eventually it halts with an output of *yes* or *no* — or it may go into an infinite loop and never halt. (Think of this as the checking phase — the deterministic algorithm is checking *s* to see whether it is a solution for the decision problem's input.)

Non-deterministic algorithm (solution) A(I) for a decision problem "P(I)?"

1. Given an instance i of input I, "guess" string s.

2. Given an instance i of input I and s, determine if s is a witness of "The answer to P(I)? is YES".

3. If step 2 (above) ended with the positive determination, output "YES" and stop. Otherwise, stop.

Extra condition not mentioned in the text.

4. For every valid instance i of the input I for the problem P(I):

there exists a “guess” s for which A(i) outputs "YES"

if, and only if,

"YES" is the correct answer to the instance "P(i)?" of the problem "P(I)?".

Definition *NP* is the class of decision problems for which there is a polynomial-bounded nondeterministic algorithm. (The name *NP* comes from "*n*ondeterministic *p*olynomial bounded.")

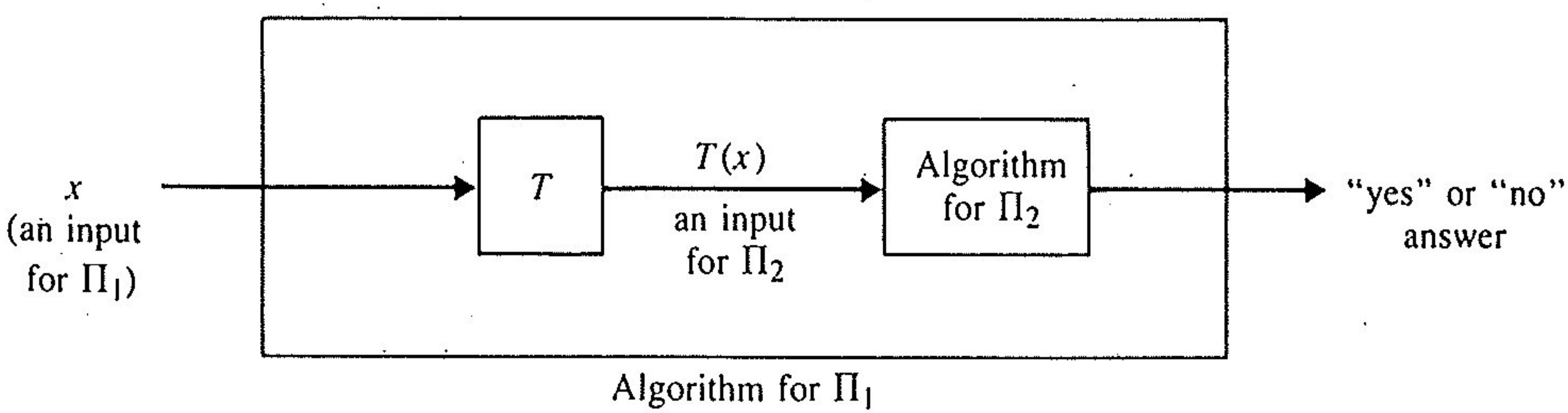
Theorem .1 Graph coloring, the Hamilton path and circuit problems, job scheduling with penalties, bin packing, the subset sum problem, the knapsack problem, CNF-satisfiability, and the traveling salesperson problem are all in *NP*.

Theorem .2 $P \subseteq NP$.

NP-Complete Problems

Notation: $\Pi_1 \le \Pi_2$

Polynomial Reductions



Reduction of problem Π_1 to problem Π_2. Π_2's answer for $T(x)$ must be the same as Π_1's answer for x.

Example A simple reduction

Let the problem Π_1 be: Given n Boolean variables, does at least one of them have the value *true*? (In other words, this is a decision-problem version of computing Boolean **or**.) Let Π_2 be: Given n integers, is the maximum of the integers positive? Let $T(x_1, x_2, \ldots, x_n) = y_1, y_2, \ldots, y_n$ where $y_i = 1$ if $x_i = true$, and $y_i = 0$ if $x_i = false$. Clearly an algorithm to solve Π_2, when applied to $y_1, y_2, \ldots, y_n$, solves Π_1 for $x_1, x_2, \ldots, x_n$. ■

Definition Let T be a function from the input set for a decision problem Π_1 into the input set for a decision problem Π_2. T is a *polynomial reduction* (also called a *polynomial transformation*) from Π_1 to Π_2 if

1. T can be computed in polynomial-bounded time, and
2. For every input x for Π_1, the correct answer for Π_2 on $T(x)$ is the same as the correct answer for Π_1 on x.

Notation: $\Pi_1 \leq_P \Pi_2$

Definition Π_1 is *polynomially reducible* (also called *polynomially transformable*) to Π_2 if there exists a polynomial transformation from Π_1 to Π_2. (We usually simply say that Π_1 is *reducible* to Π_2; the polynomial bound is understood.) The notation $\Pi_1 \leq_p \Pi_2$ is used to indicate that Π_1 is reducible to Π_2.

Theorem .3 If $\Pi_1 \leq_p \Pi_2$ and Π_2 is in P, then Π_1 is in P.

Definition A problem Π is *NP-complete* if it is in *NP* and for every other problem Π' in *NP*, $\Pi' \leq_p \Pi$.

Theorem .4 If any *NP*-complete problem is in *P*, then $P = NP$.

Theorem .5 (Cook's theorem) The CNF-satisfiability problem is *NP*-complete.

Theorem .6 Graph coloring, the Hamilton path and circuit problems, job scheduling with penalties, bin packing, the subset sum problem, the knapsack problem, and the traveling salesperson problem are all *NP*-complete.

A problem that is not known to be in NP or P

Integer factorization problem.

Optimization. Given an inter $n > 1$, find prime factorization of n.

($n = k_1^{\alpha_1} \times \ldots \times k_m^{\alpha_m}$, where k_i's are prime and α_i's are integer)

Decision problem. Given $k \geq 1$, does n have a factor larger than 1 and less than k?

The integer factorization problem is not known to be in NP <u>and</u> is not known to be in P.

However, the primality problem (Is n a prime number?) has been proven to be in P.

AKS primality test has a worst-case polynomial time (size $n = \lfloor \lg n \rfloor + 1$).

To show that the problem Π is *NP*-~~complete~~ hard, choose some known *NP*-complete problem Π' and reduce Π' to Π, not the other way around. The logic is as follows:

> Since Π' is *NP*-complete, all problems in *NP* $\le_p \Pi'$.
>
> Show $\Pi' \le_p \Pi$.
>
> Then all problems in *NP* $\le_p \Pi$.
>
> Therefore, Π is *NP*-~~complete~~ hard.

No one knows if $P = NP$.

Many top experts believe $P \neq NP$.

There are some results suggesting that it is extremely difficult to decide the question:

$$\text{Is } P = NP \text{ ?}$$

Example from our experience with math.

Some easy looking theorems (like Fermat's theorem saying that $x^n + y^n = z^n$ has integer solutions for $x > 0$, $y > 0$, $z > 0$ iff $n = 1$ or 2) Prooved really hard to prove. But once a proof has been published, everybody with enough math preparation can verify it.

If $P \neq NP$ then

There are solvable problems that are extremely difficult to solve, but once they are solved, checking their solution is <u>easy</u>.

And so is distribution of that solution.

It's a bit like with password-protected computer.
As long as the password is kept secret, it is very difficult for a hacker to login.
Once a password is guessed, it becomes easy to login (and to distribute it).

If $P \neq NP$ then ...

Job scheduling problem is practically unsolvable, except, perhaps, for some special (easy) cases.

This explains why free-market (distributed) economies outperform centrally-planned economies.

This also explains how knowledge workers can be deprived of the fruits of their work.

What Makes a Problem Hard?

The 3-CNF satisfiability problem is the CNF-satisfiability problem restricted to expressions with exactly three literals per clause. It is *NP*-complete. If there are at most two literals per clause, satisfiability can be checked in polynomial-bounded time.

A *k-clique* in a graph is a subgraph consisting of k mutually adjacent vertices (i.e., a complete graph on k vertices.)

The problem of determining whether a graph has a k-clique is *NP*-complete, but for planar graphs it is in *P* because a planar graph cannot have a clique with more than four vertices. (The clique problem is also in *P* for graphs with bounded degrees.)

Determining if a graph is 2-colorable is easy; determining if it is 3-colorable is *NP*-complete. It is still *NP*-complete if the graphs are planar and the maximum degree is 4.

These examples do not yield any nice generalizations about *why* a problem is *NP*-complete. There are still a great many open questions in this field, the main one being, of course, Does $P = NP$?

Determining an existence of Hamiltonian cycle in a graph G is hard. (But easy for: n-cube, complete graph, acyclic graph, etc.

Determining an existence of Euler's cycle (passes through every edge once and covers all vertices) in a graph G is easy.

Question: How to do that?