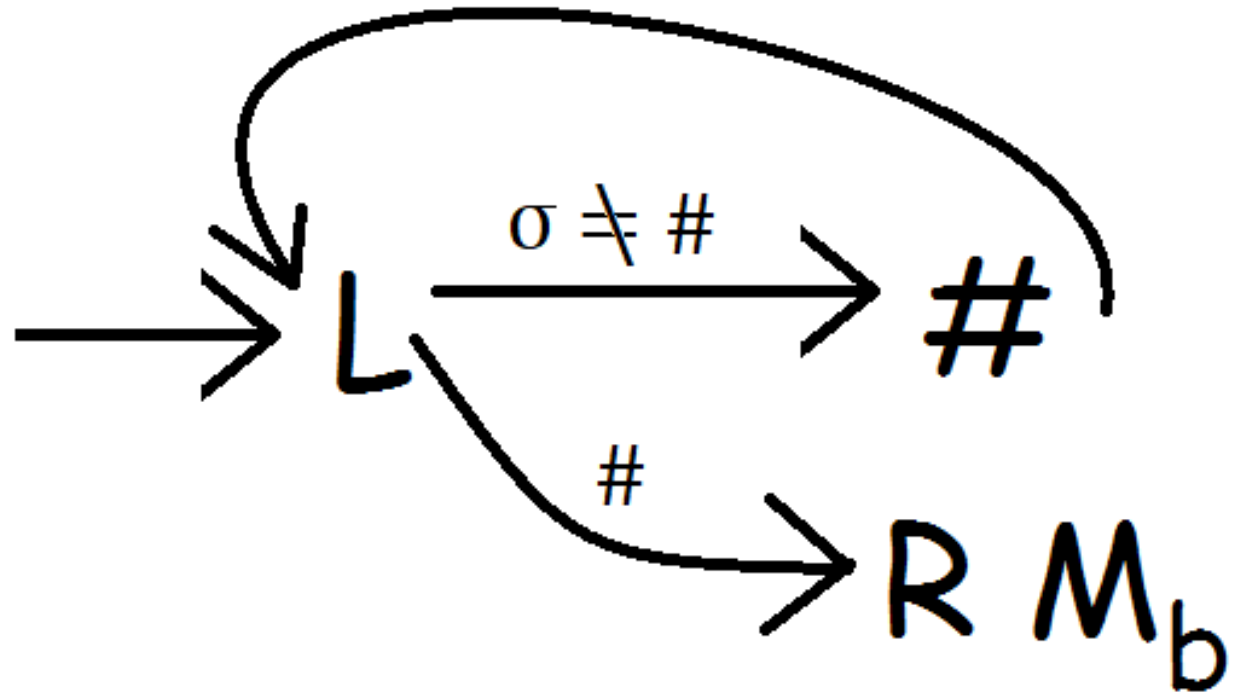
1. What does M_x do on input 01011?
2. What does M_x do on input 111?
3. What does M_x do on an arbitrary input w?
4. When will M_x halt on a given input?
5. What language does M_x accept?

M_x:

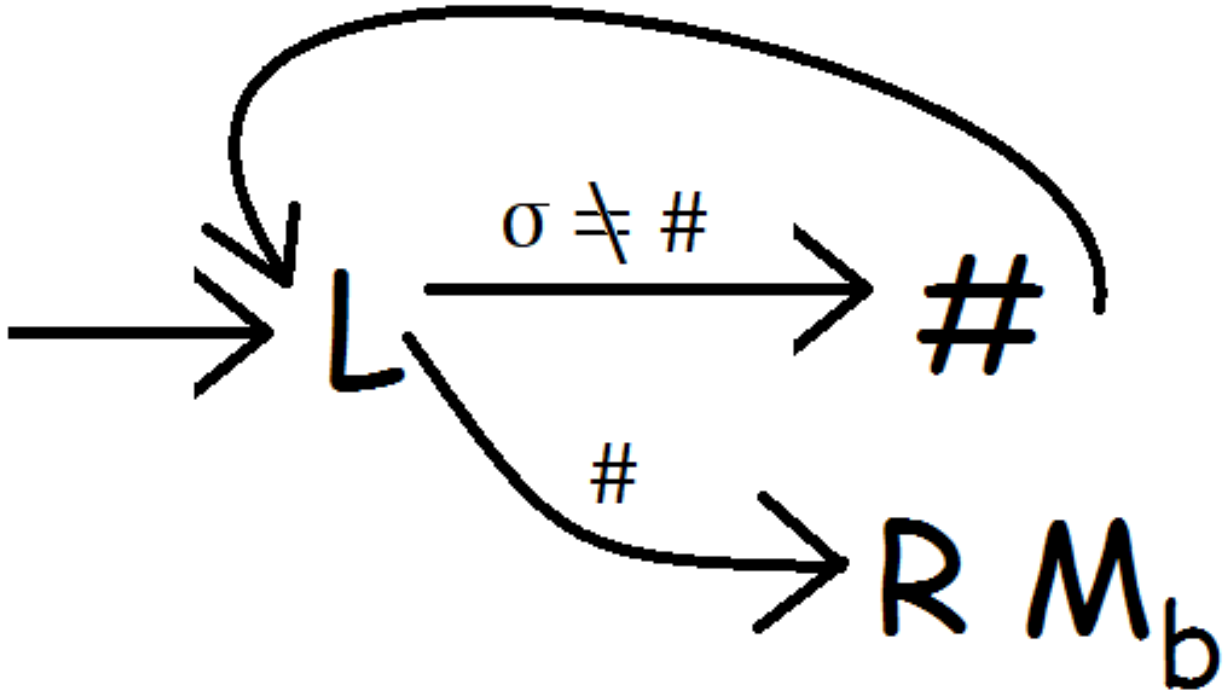


On any input string w, M_x erases the tape and then runs M_b on a blank tape.

If M_b halts when started on a blank tape, M_x halts on every input.

If M_b does not halt when started on a blank tape, M_x does not halt on any inputs.

M_x:



So the language M_x accepts is either Σ^* or Φ.

Suppose we have an algorithm for this question:

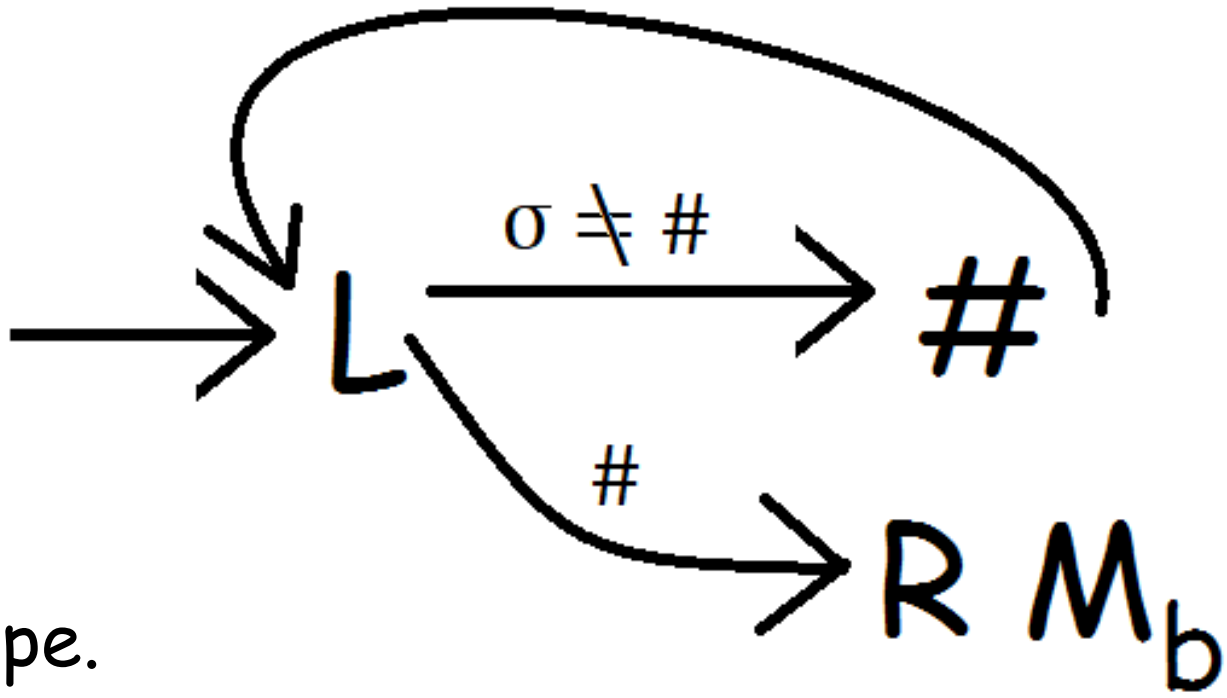
(c) Given M_c: Is there any string on which M_c halts?

Use M_x as input.

If the answer is "yes" then M_b halts when started on a blank tape.

If the answer is "no" then M_b does not halt when started on a blank tape.

M_x:



Suppose we have an algorithm for this question:

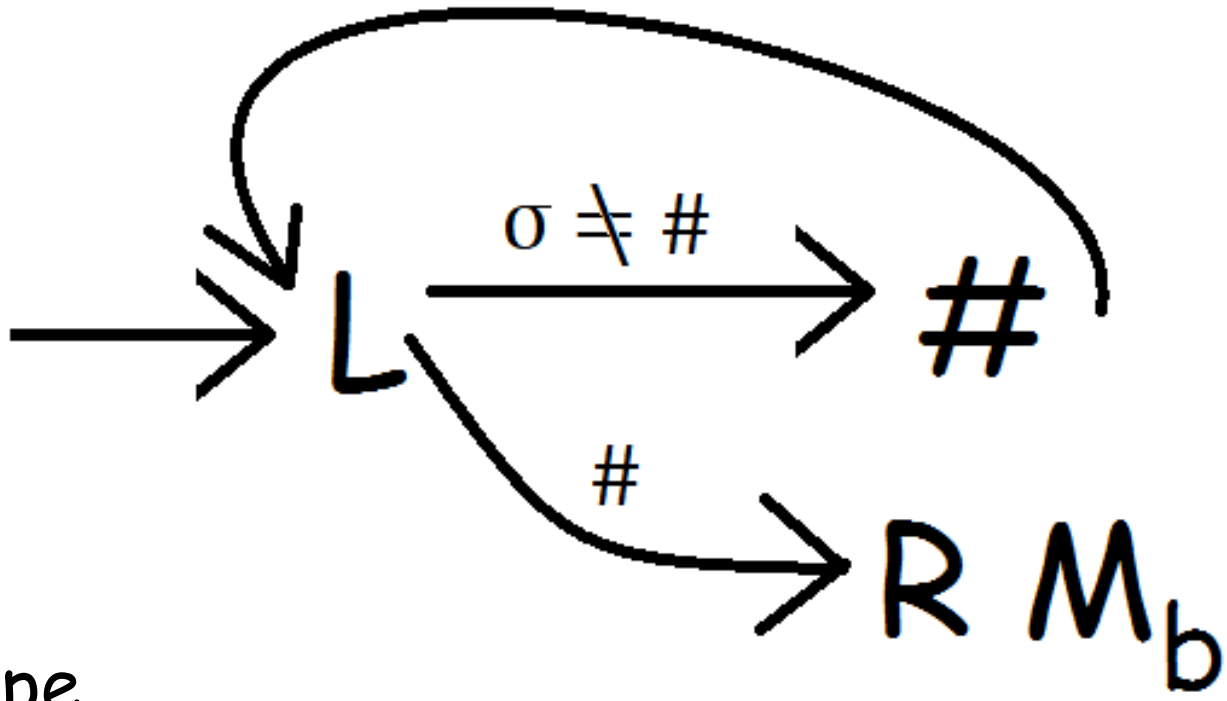
(d) Given M_d: Does M_d halt on every string?

Use M_x as input.

If the answer is "yes" then M_b halts when started on a blank tape.

If the answer is "no" then M_b does not halt when started on a blank tape.

M_x:



But last class we showed problem (b) was not decidable. Therefore (c) and (d) are also not decidable. Try to prove (e) is not decidable.

Theorem 5.4.2 (p. 255): The following problems are not Turing-decidable:

(a) Given M_a, w: Does M_a halt on input w?

(b) Given M_b: Does M_b halt on input ε?

(c) Given M_c: Is there any string on which M_c halts?

(d) Given M_d: Does M_d halt on every string?

(e) Given two TM's M_1 and M_2: Do they halt on the same input strings?

1. Which of these sequences correspond to Hamilton cycles in the graph?

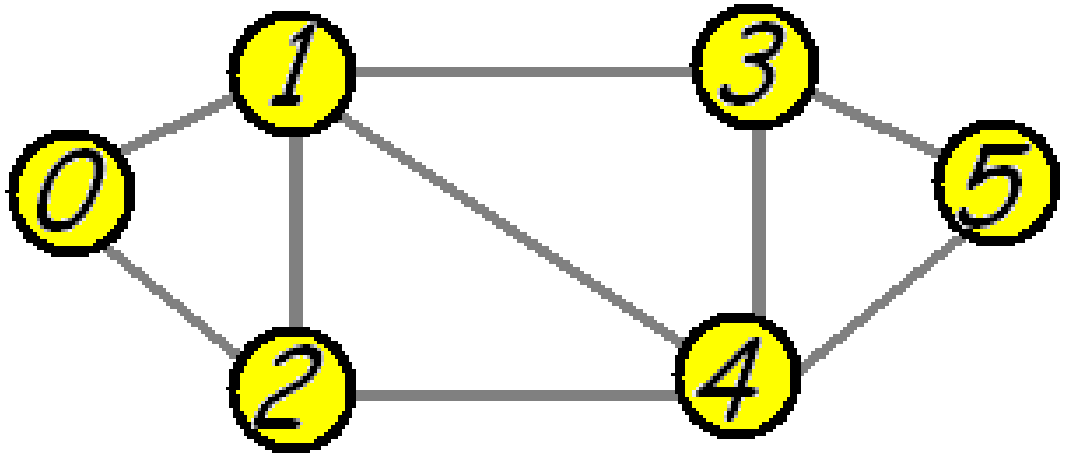
(a) 0 1 3 5 4 2

(b) 0 1 2 4 3 5

(c) 0 1 4 3 1 2

(d) 0 2 3 5 4 1

(e) 0 1 3 5 6 2



2. Give pseudocode for an algorithm to check if a seqence S[0..(n-1)] for a graph G stored in an adjacency matrix A[0..(n-1)][0..(n-1)] gives a Hamilton cycle.

Announcements:

There is a tutorial today.
Assignment #5 is due on Friday at the beginning of class.

Please do course evaluations.

Office hours for final exam:

Tuesday August 1: 12:30pm
Thursday August 3: 12:30pm
Friday August 4: 12:30pm

Sunday August 13: 3pm-6pm, Room TBA

Introduction to NP-completeness.

The class of NP-complete problems is defined.

Satisfiability, the most famous NP-complete problem is introduced.

To prove new problems are undecidable, we use halting problem reductions.

To prove new problems are NP-complete, reductions are also used but the transformation must preserve polynomial running time complexity.

Table 1: Comparing polynomial and exponential time complexity.
Assume a problem of size one takes 0.000001 seconds (1 microsecond).

	Size *n*					
	10	**20**	**30**	**40**	**50**	**60**
n	0.00001 second	0.00002 second	0.00003 second	0.00004 second	0.00005 second	0.00006 second
n^2	0.0001 second	0.0004 second	0.0009 second	0.0016 second	0.0025 second	0.0036 second
n^3	0.001 second	0.008 second	0.027 second	0.064 second	0.125 second	0.216 second
n^5	0.1 second	3.2 second	24.3 second	1.7 minutes	5.2 minutes	13.2 minutes
2^n	0.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	0.059 second	58 minutes	6.5 years	3855 centuries	$2*10^8$ centuries	$1.3*10^{13}$ centuries

(from M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, New York, 1979.)

Table 2: Effect of improved technology on several polynomial and exponential time algorithms. The following table represents the size of the largest problem instance solvable in 1 hour.

Time Complexity function	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N1	100 N1	1000 N1
n^2	N2	10 N2	31.6 N2
n^3	N3	4.46 N3	10 N3
n^5	N4	2.5 N4	3.98 N4
2^n	N5	N5+6.64	N5+9.97
3^n	N6	N6+4.19	N6+6.29

(from M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, New York, 1979.)

Class P

A decision problem (yes/no question) is *in the class P* if there is a polynomial time algorithm for solving it.

Polynomial time: $O(n^c)$ for some constant c.

If a problem is solvable in polynomial time for

- some sensible encoding of the input
- some reasonable machine (TM/RAM/PC)

it can be solved in polynomial time for all other sensible encodings/reasonable machines.

A program that runs on a Random Access Machine (such as our modern day computers) in time T(n) can be simulated on a single tape Turing machine in time $O(T^3(n))$.

This distinction is made between polynomial time and anything else (sometimes referred to as exponential time) because of the blowup in computation times which appear for exponential algorithms.

Class NP

A decision problem (yes/no question) is in the *class NP* if it has a nondeterministic polynomial time algorithm. Informally, such an algorithm:

1.Guesses a solution (nondeterministically).

2. Checks deterministically in polynomial time that the answer is correct.

Or equivalently, when the answer is "yes", there is a certificate (a solution meeting the criteria) that can be verified in polynomial time (deterministically).

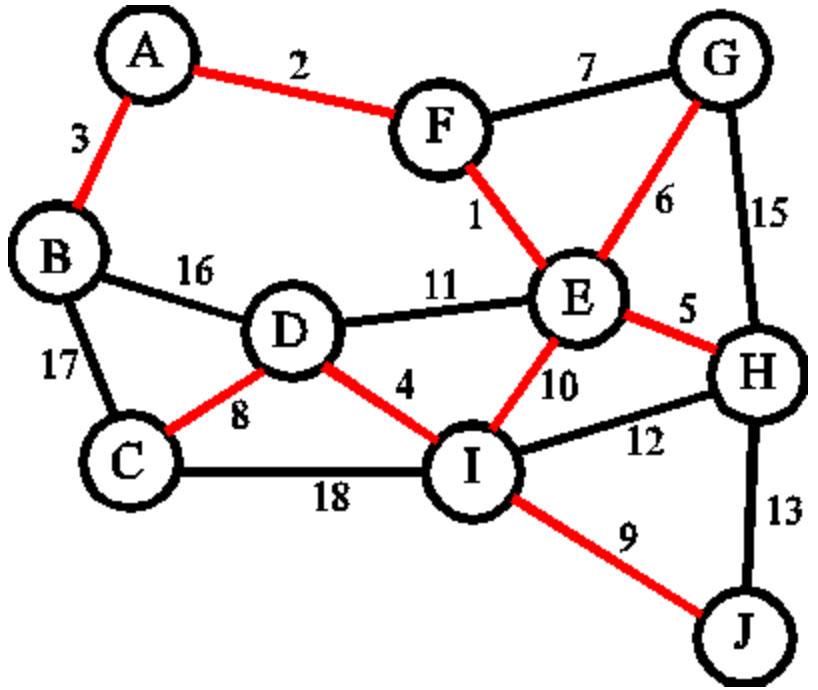
Example problem which is in P and NP:

Minimum Weight Spanning Tree (CSC 225).

Input: Graph G, integer k.

Question: Does G have a spanning tree of weight at most *k*?

If you are provided with a tree with weight at most k as part of the solution, the answer can be verified in O(n) time.



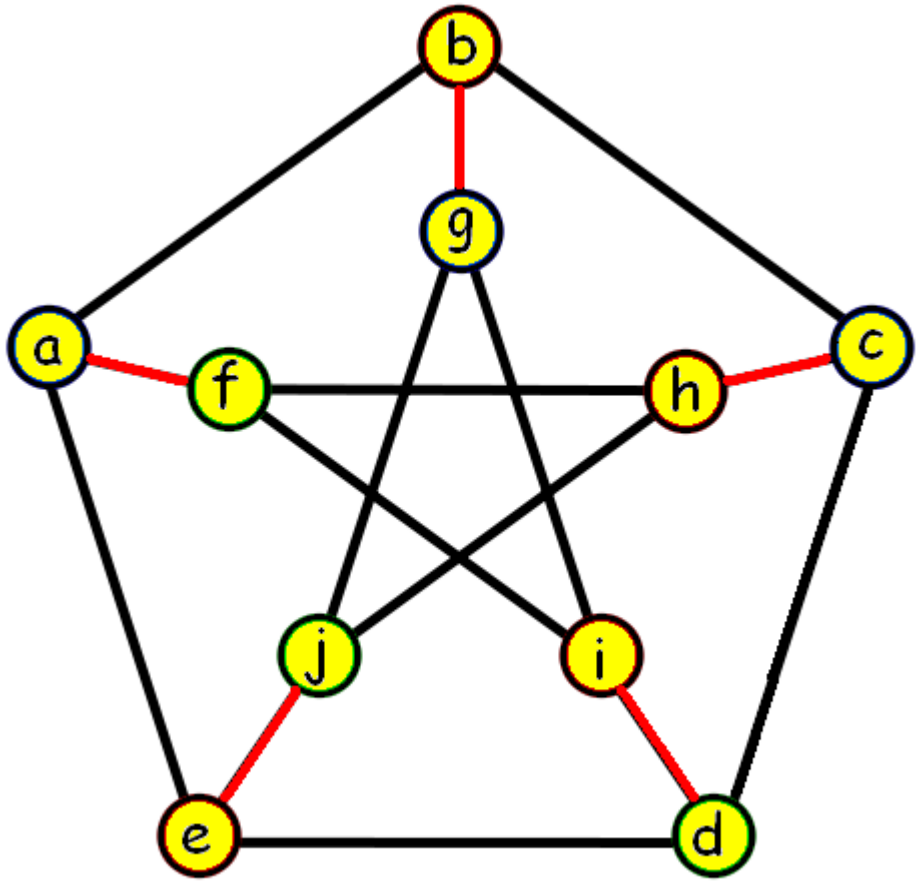
Matching is in NP: Given a graph G and integer k, does G have a k-edge matching?

Matching: disjoint edges.

Certificate, k=5:

(a,f) (b,g) (c,h)

(d,i)(e,j)



Hamilton Cycle is in NP: Input graph G.

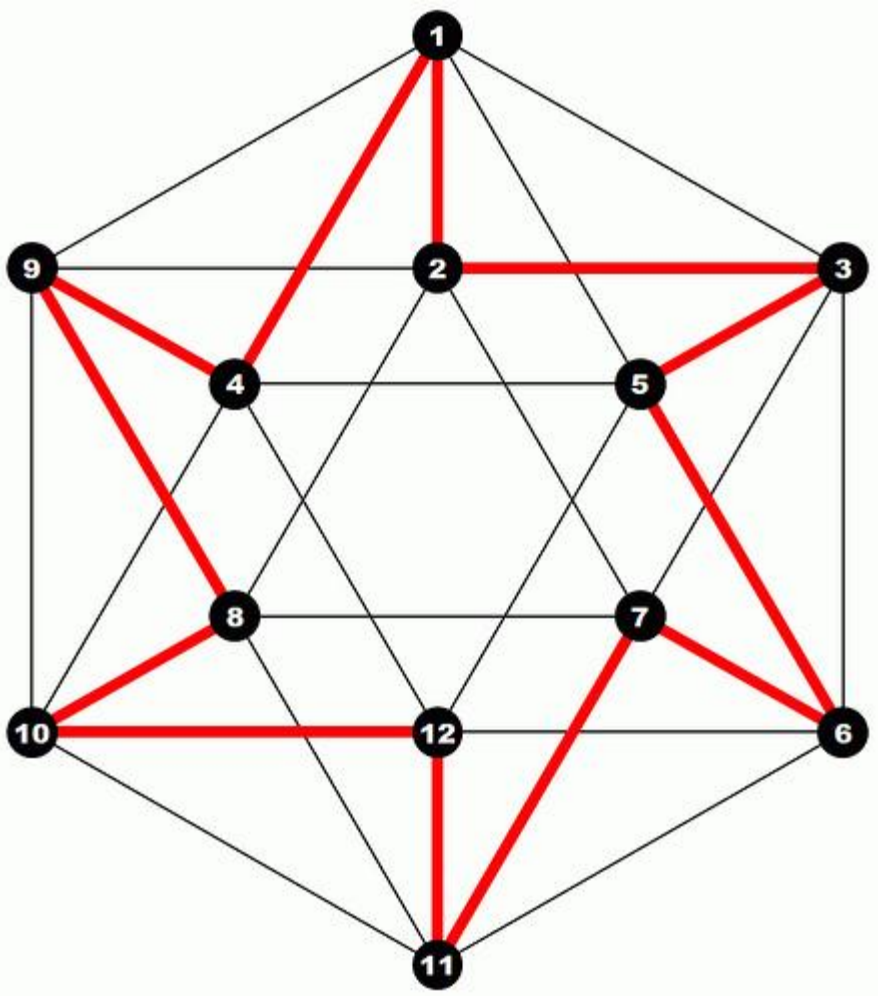
Does G have a Ham. cycle?

Certificate:

1, 2, 3, 5,

6, 7, 11, 12,

10, 8, 9, 4



Picture from: http://mathoverflow.net/faq

Does P= NP? the Clay Mathematics Institute has offered a $1 million US prize for the first correct proof.

Some problems in NP not known to be in P:

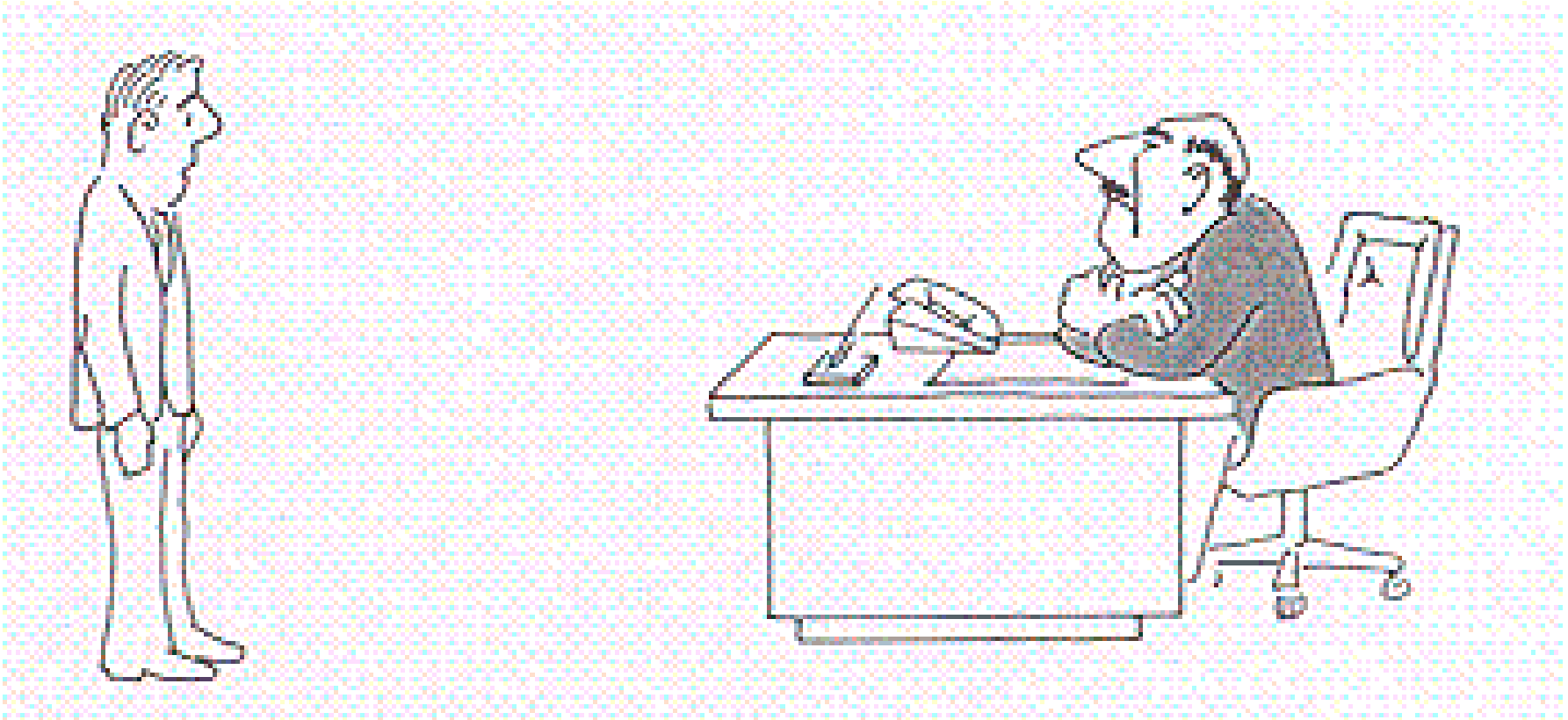
Hamilton Path/Cycle

Independent Set

Satisfiability

Note: Matching is in P. Learn more in a graph algorithms class.

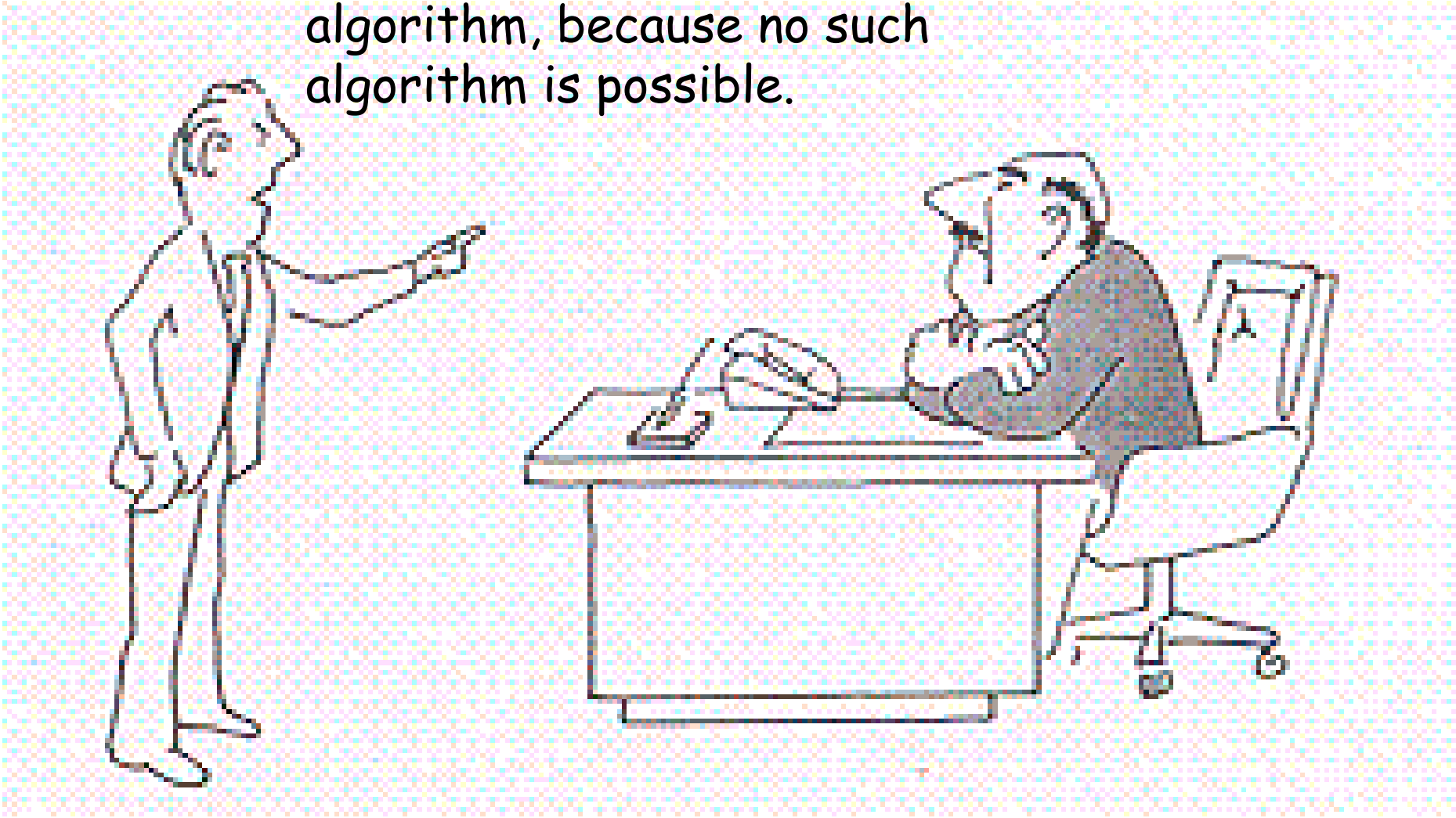
NP-completeness

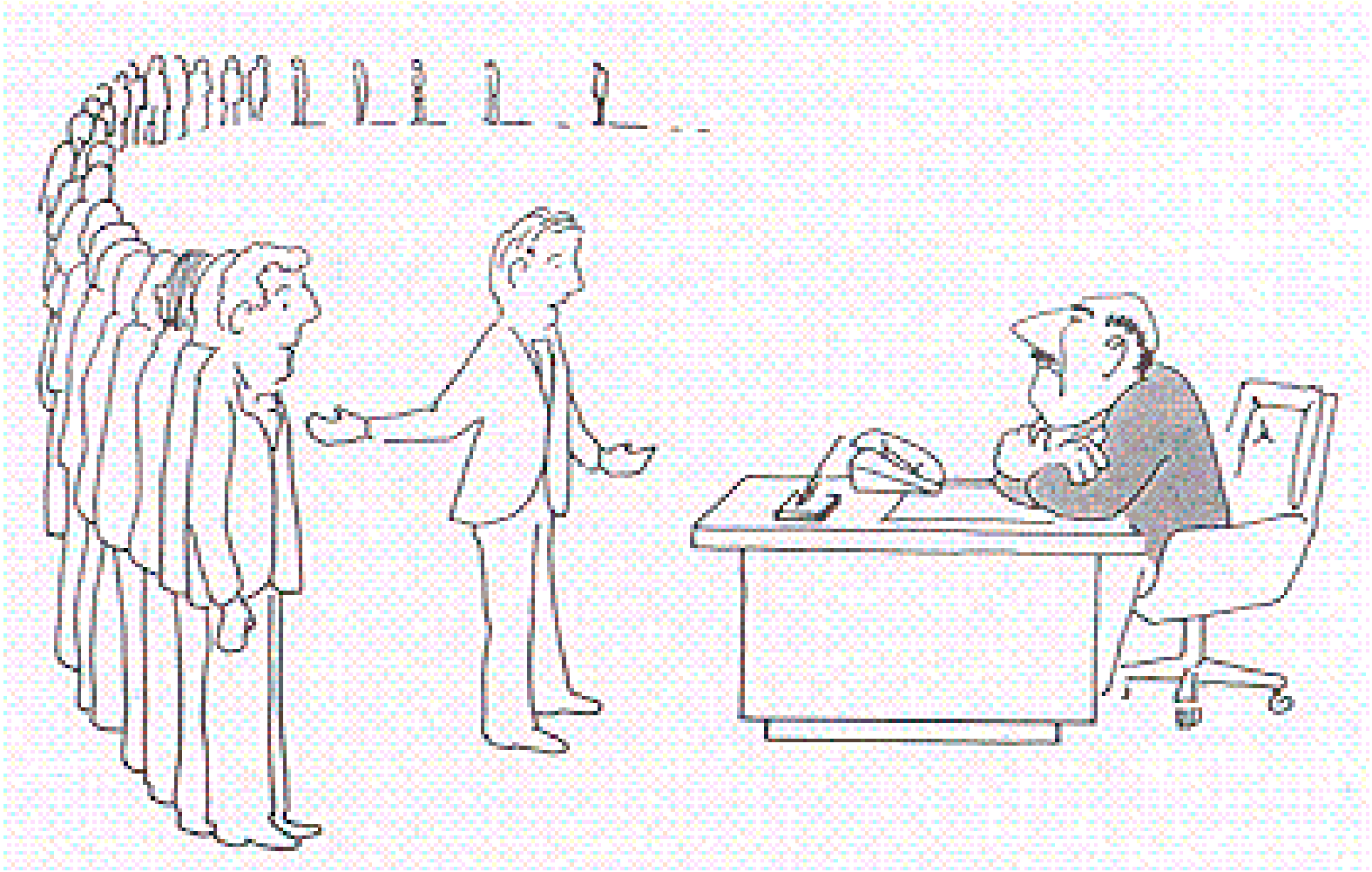


I can't find an efficient algorithm,

I guess I'm just too dumb.

I can't find an efficient algorithm, because no such algorithm is possible.





I can't find an efficient algorithm, but neither can all these famous people.

NP-complete Problems

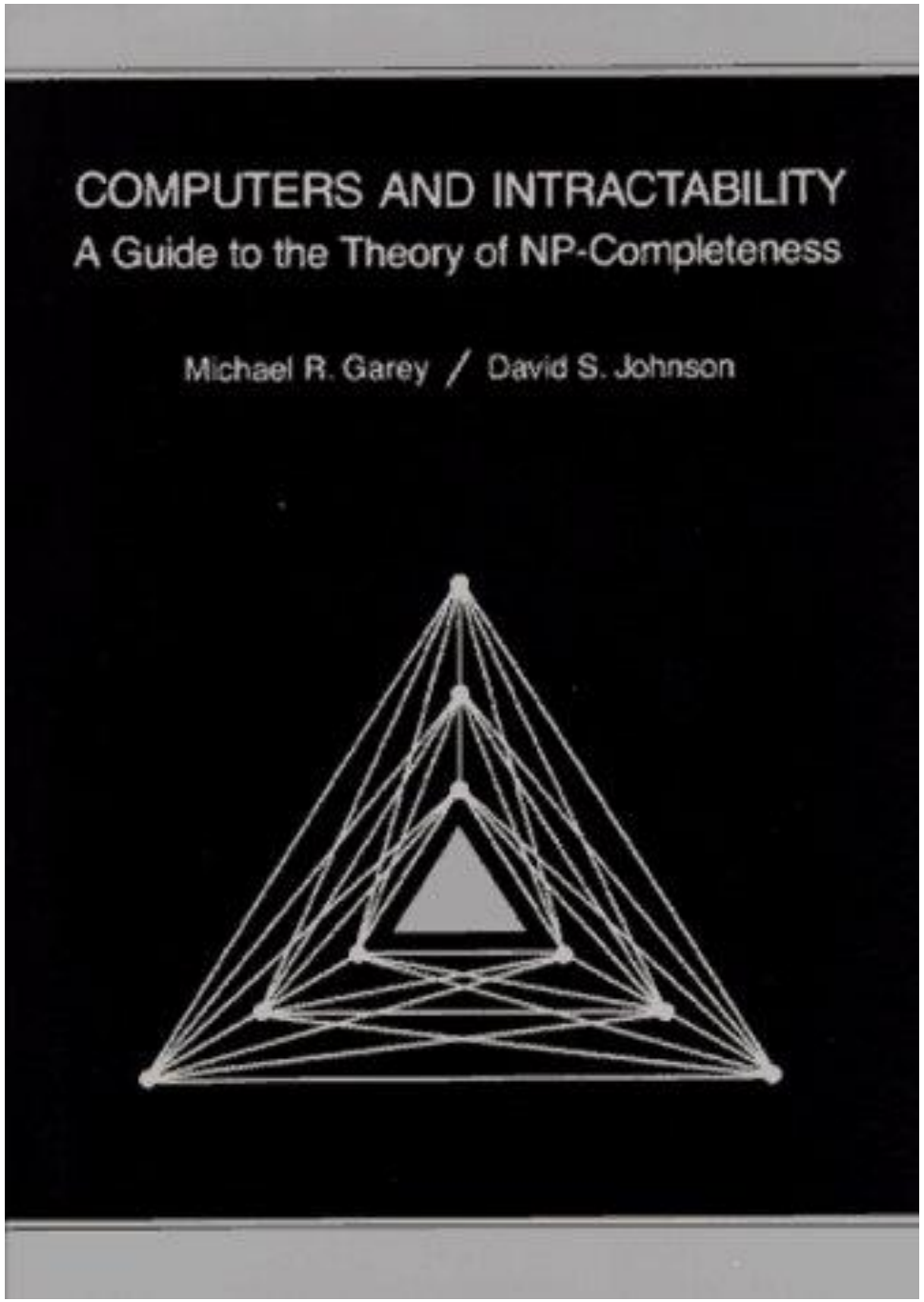
The class of problems in NP which are the "hardest" are called the *NP-complete* problems.

A problem Q in NP is NP-complete if the existence of a polynomial time algorithm for Q implies the existence of a polynomial time algorithm for all problems in NP.

Steve Cook in 1971 proved that SAT is NP-complete. Proof: will be given in our last class. Other problems: use reductions.

Bible for NP-completeness:

M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completness*, W. H. Freeman, 1st ed. (1979).



SAT (Satisfiability)

Variables: $u_1, u_2, u_3, \ldots u_k$.

A **literal** is a variable u_i or the negation of a variable $\neg u_i$.

If u is set to *true* then ¬ u is *false* and if u is set to *false* then ¬ u is *true*.

A **clause** is a set of literals. A clause is *true* if at least one of the literals in the clause is *true*.

The input to SAT is a collection of clauses.

This SAT problem has solution
u_1=T, u_2=F, u_3= T, u_4=F

(u_1 OR u_2 OR u_4) AND (¬ u_2 OR u_4) AND
(¬ u_1 OR u_3) AND (¬ u_4 OR ¬ u_1)

Does this SAT problem have a solution?

(u_1 OR u_2) AND (¬ u_2 OR u_3) AND

(¬ u_3 OR ¬ u_1) AND (¬ u_2 OR ¬ u_3) AND

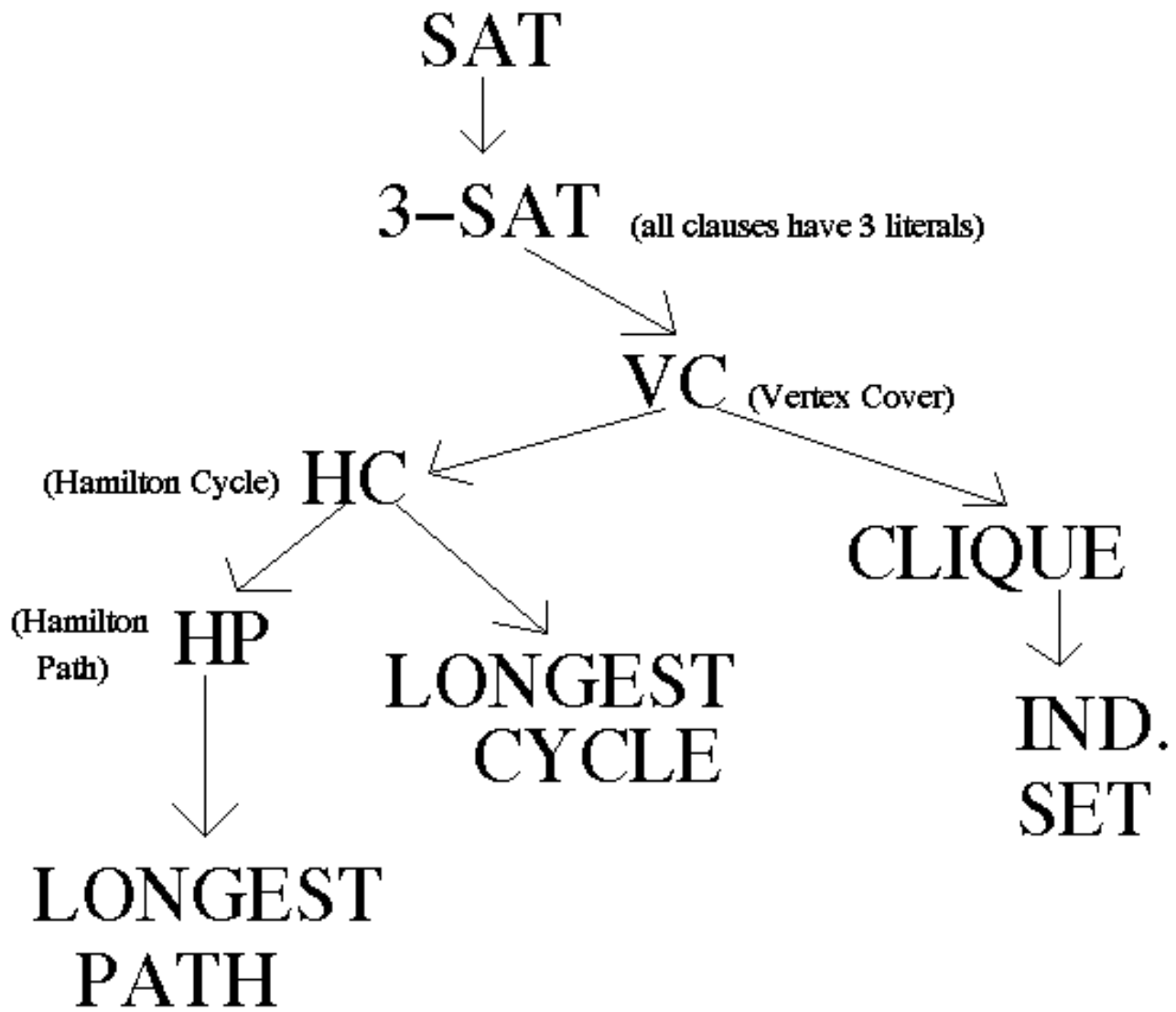
(u_3 OR ¬ u_1)

SAT (Satisfiability)

The output is the answer to: Is there an assignment of *true*/*false* to the variables so that every clause is satisfied (**satisfied** means the clause is *true*)?

If the answer is yes, such an assignment of the variables is called a **truth assignment**.

SAT is in NP: Certificate is true/false value for each variable in satisfying assignment.



History of NP–completeness Reductions

3-SAT- each clause must contain exactly 3 variables (assignment- at most 3).

Given: SAT is NP-complete (proof later)

Theorem: **3-SAT is NP-Complete.**

The first step in any NP-completeness proof is to argue that the problem is in NP.

The problem 3-SAT is a yes/no question.

Certificate: truth assignment, can be checked in polynomial time.

Next, we show that a polynomial time algorithm for 3-SAT implies the existence of one for SAT.

To convert a SAT problem to 3-SAT:

1. Clauses of size 1.

SAT: $\{z\}$

3-SAT:

$$\{z, y_1, y_2\},$$
$$\{z, \neg y_1, y_2\},$$
$$\{z, y_1, \neg y_2\},$$
$$\{z, \neg y_1, \neg y_2\}$$

y_1 and y_2 are new variables.

2. Clauses of size 2.

SAT: $\{z_1, z_2\}$

3-SAT: $\{z_1, z_2, y\}$,
$\{z_1, z_2, \neg y\}$

y is a new variable.

3. Clauses of size 3.

Leave these as they are since they are already acceptable for 3-SAT.

4.Clauses of size 4 or more.

SAT: $\{z_1, z_2, z_3, \ldots z_k\}$, $k>3$

3-SAT:

$\{ z_1, z_2, y_1\}$,
$\{\neg y_1, z_3, y_2\}$,
$\{\neg y_2, z_4, y_3\}$,
...
$\{\neg y_{k-4}, z_{k-2}, y_{k-3}\}$,
$\{\neg y_{k-3}, z_{k-1}, z_k\}$

$y_1, y_2, \ldots y_{k-3}$, are new variables.

This does not constitute a proof of NP-completeness unless we can argue that the size of the new 3-SAT problem problem is polynomially bounded by the size of the old SAT problem. Consider each case:

Size of clause	# new literals	size before	size after
1	2	1	12
2	1	2	6
3	0	3	3
$k \geq 4$	k-3	k	k + 2(k-3)

In all cases, the size after is at most 12 times the original problem size.

2-SAT: All clauses have at most 2 literals.

There is a linear time algorithm for 2-SAT

so 2-SAT is in P.

The 3-SAT problem is as hard as SAT but unless P=NP, 2-SAT is easier than 3-SAT or SAT.