Search vs. verify

Is it significantly easier to check that an answer has been found or to find the answer to a problem?

Examples

- Is it easier to find a path that visits every room in a maze exactly once or to verify that a given path does exactly that?
- Is it easier to find a factor of
$$2^{67} - 1 = 147\,573\,952\,589\,676\,412\,927$$
or to verify that $2^{67} - 1 = 193707721 \times 761838257287$?
- Is it easier to find a proof of a mathematical theorem or to verify that a given proof is correct?

In each case it is clear that verification can't be harder than search, but is it significantly easier?

In each case we know we can verify quickly (i.e. in polynomial time).

But the best search algorithm we know takes exponential time.

Two possibilities:

- Every problem that can be quickly verified can be quickly solved: $\mathbf{P} = \mathbf{NP}$
- Many probems that can be quickly verified can not be quickly solved: $\mathbf{P} \neq \mathbf{NP}$

Currently we don't know which.

Intractability

[A very short and somewhat simplified overview.]

Recall our table of times. Assuming 1 operation takes $1\,\mathrm{ns}$.

	$n = 10$	$n = 50$	$n = 100$	$n = 1000$
$\log_2 n$	$3\,\mathrm{ns}$	$5\,\mathrm{ns}$	$6\,\mathrm{ns}$	$10\,\mathrm{ns}$
n	$10\,\mathrm{ns}$	$50\,\mathrm{ns}$	$100\,\mathrm{ns}$	$1\,\mu\mathrm{s}$
$n \log_2 n$	$33\,\mathrm{ns}$	$282\,\mathrm{ns}$	$664\,\mathrm{ns}$	$10\,\mu\mathrm{s}$
n^2	100 ns	$2.5\,\mu\mathrm{s}$	10 $\mu\mathrm{s}$	1 ms
n^3	1 $\mu\mathrm{s}$	$125\,\mu\mathrm{s}$	1 ms	1 s
2^n	$1\,\mu\mathrm{s}$	3.5×10^{24}y	4×10^{39}y	3×10^{310}y
$n!$	$3\,\mathrm{ms}$	10×10^{73}y	3×10^{167}y	1.3×10^{2577}y
2^{2^n}	6×10^{317}y	big	Bigger	HUGE

As a generalization (the ‘Cobham-Edmonds thesis’) we consider that

- Algorithms with polynomial (or better) time complexity are **feasible**.
- Algorithms with superpolynomial time complexities are **infeasible**.

What about problems?

Thus there are good problems and nasty problems.

Recall that a problem’s complexity is the time complexity of the best algorithm for that problem

If the best algorithm is polynomial (or lower), we say the problem is **tractable**.

If the best algorithm is superpolynomial, we say the problem is **intractable**.

There are a few problems that are *known to be intractable*

Seemingly intractable problems

There are a large number of problems such that

- No one knows they are intractable, but
- No one has found a polynomial time algorithm for them

An example is integer factorization

- In 1903 Frank Cole factored the Mersenne number $2^{67} - 1$
  - It took him a "three years of Sundays" to do all the calculations.
  - But it took only minutes to verify the result
- While factoring methods have improved, even the best known algorithms are still exponential with respect to the number of bits.
- Codes based on factoring 1000 bit numbers are still safe.
- In 2009 factoring a 768-bit number took the equivalent of 2000 CPU years at 2GHz

Decision problem

Decision problem are problems with "yes/no" answers.

To keep things simple, we focus on decision problems.

For example, a decision problem version of factoring asks whether x has a factor less than k.

This is no loss if we are trying to show problems are hard, since if the decision problem is intractable, then the search problem is also intractable.

Conversely if the decision problem is tractable, so is the search problem, as we can use binary search.

Easy to solve problems

The set of decision problems that can be solved in polynomial time (worst case) is called $\mathbf{P}$.

Does graph G have a path from s to t shorter than k

Does graph G have a spanning tree smaller than k

Does G have an Eulerian tour

Is x divisible by 3?

Is x composite?

Easy to check problems

It turns out that a lot of problems are easy to check, but seemingly hard to solve.

The set of decision problems whose “yes” answers can be checked in polynomial time, given a polynomial amount of evidence, is called $\mathbf{NP}$.

Examples:

Problem	**Evidence**
Every problem in P.	A trace of the computation
Does x have a factor smaller than k?	2 factors, one smaller than k
Does G have a Hamiltonian tour?	A Hamiltonian tour
Does a graph have a 3-colouring?	Such a colouring
Is a propositional formula Φ satisfiable?	An assignment to its variables that satisfies Φ.

(Equivalently, we can define $\mathbf{NP}$ as the set of decision problems we could solve in polynomial time if we had a “magic coin”.

The “magic coin” will answer any question we give it helpfully if the answer to the decision problem’s answer is “yes” and arbitrarily if the answer is “no”.

For example, to solve the Hamiltonian tour problem, we could ask the coin “Should I follow edge e next?”.

By the way, this model of computation is called ‘nondeterministic’ and $\mathbf{NP}$ stands for ‘nondeterministic polynomial time’

Exercise: Show the “magic coin” definition of $\mathbf{NP}$ is equivalent to the “polynomial time checking” definition of $\mathbf{NP}$.)

Polytime reductions

Suppose $F : X \to \mathbb{B}$ and $G : Y \to \mathbb{B}$ are two decision problems.

And that $H : X \to Y$ is a problem such that

- we know a polynomial time procedure h for H and
- $F(x) = G(H(x))$,for all x

We say h is a **polytime reduction** from F to G.

Lemma: If p and q are polynomials, so is their composition.

E.g. if $p(n) = n^2 + n$ and $q(n) = n^3 + 1$, then their composition is $\left(n^3 + 1\right)^2 + \left(n^3 + 1\right)$ which is $n^6 + 3n^3 + 2$

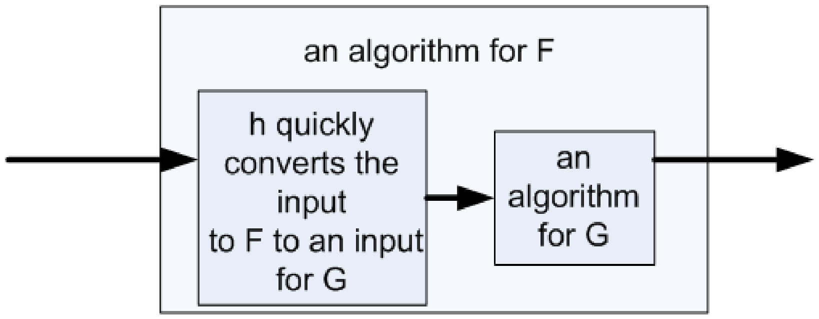
Theorem: Suppose there is a polytime reduction h from F to G, then

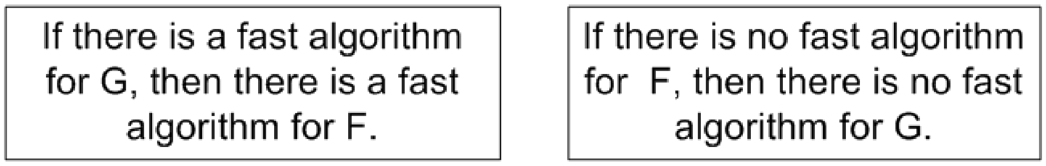
1. if G is tractable, so is F.
2. if F is intractable, so is G.

Proof

Assume there is a polytime reduction h from F to G

1. Suppose that G is tractable.
   a. Then there is a polytime procedure for G, call it g.
   b. We have the following polytime algorithm for F
      - proc $f(x)$ return $g(h(x))$ end f
   c. (This is polytime because h only has time to produce an output that is of polynomial size with respect to the size of x, so the total complexity can be no worse than the composition of the time functions for g and h, which, by the the lemma, is a polynomial)
   d. Therefore F is tractable.
2. By the contrapositive law: if F is intractable, then so is G.




If there is a fast algorithm for G, then there is a fast algorithm for F.

If there is no fast algorithm for F, then there is no fast algorithm for G.

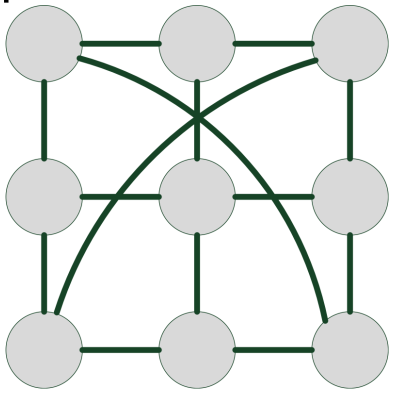
Example: 3-colouring graphs and PSAT

3-COLOURING: Given a graph G, is it 3 colourable.

A graph is 3-colourable iff we can colour each node red, blue, or green such that

- for every edge e, the endpoints of e are coloured differently.

Example: Is this graph 3-colourable?



PSAT: Given a propositional formula, is it satisfiable.

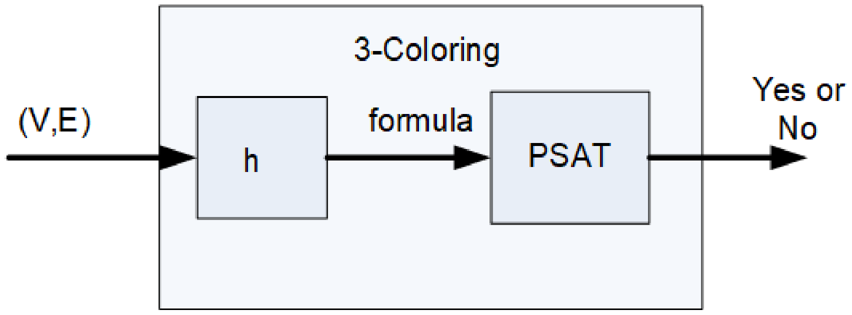
A propositional formula Φ is satisfiable iff we can assign each variable false or true so that the whole formula is satisfiable.

Examples:

- Satisfiable: $(\neg A \Rightarrow B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow \neg C)$
  - $*$ Evidence: Set B to true and A and C to false.
- Unsatisfiable: $(A \wedge B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow \neg C)$
  - $*$ Evidence: Try all 8 possible assignments

Now we can construct a reduction from 3-COLOURING to PSAT as follows

```
procedure $h(V, E)$
    for each node $u$ in $V$
        create 3 variables and generate the following formula
        $(R_u \vee G_u \vee B_u) \wedge \neg(R_u \wedge G_u) \wedge \neg(G_u \wedge B_u) \wedge \neg(B_u \wedge R_u)$
    for each edge $e$ in $E$
        let $u$ to $v$ be the endpoints of $e$
        generate the following formula
        $\neg(R_u \wedge R_v) \wedge \neg(G_u \wedge G_v) \wedge \neg(B_u \wedge B_v)$
    Conjoin all the generated conjuncts to make one big formula.
    return this formula
```



If there is a fast algorithm for PSAT, then there is a fast algorithm for 2-coloring.

If there is no fast algorithm for 3-coloring, then there is no fast algorithm for PSAT.

The fact that this reduction works and is polynomial time shows that

(a) If 3-COLOURING is intractable, so is PSAT

(b) If PSAT is tractable, so is 3-COLORING

Transitivity

If there is a reduction from F to G and a reduction from G to H, then there is a reduction from F to H.

Proof: Simply compose the reductions. The composed reduction is also poly time.

Patient zero: PSAT is a hardest $\mathbf{NP}$ problems

Above, we showed that one problem in $\mathbf{NP}$ reduces to PSAT.

In 1971 Steve Cook proved the following theorem

Theorem [Cook]. Every problem in $\mathbf{NP}$ can be reduced to PSAT.

Consequence:.If any problem in $\mathbf{NP}$ is intractable, then PSAT is intractable.

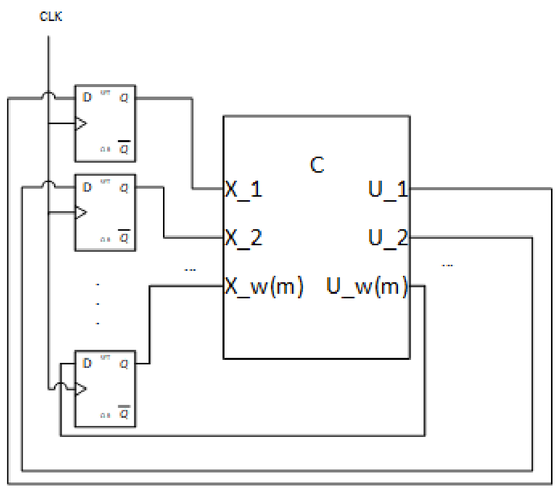
We call a problem that has that property $\mathbf{NP}$**-hard**. Thus

Consequence. PSAT is $\mathbf{NP}$-hard.

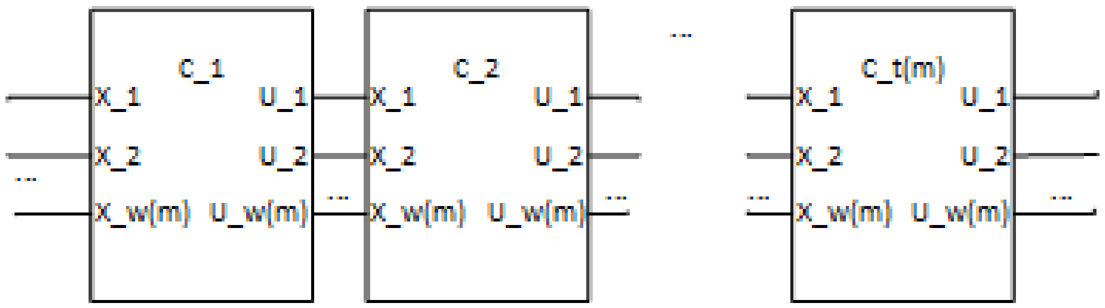
Proof sketch:

Before we get to the proof, let's think about algorithms and digital circuits. Suppose we have a deterministic algorithm f for a problem $F : Z \to \mathbb{B}$. And suppose we know that for an input of size m, the amount of state needed by z won't exceed $w(m)$ bits and z won't require more than $t(m)$ clock cycles.

For each input of size m, we can create a sequential circuit sc_m that uses $w(m)$ D-flip-flops. We'll use the convention that flip-flops 1 through m initially hold the input. Flip-flop 0 should hold the output after $t(m)$ clock cycles.



By unrolling the sequential circuit, we can create a combinational circuit that consists of $t(m)$ copies of the combinational part of the circuit.



Suppose f is a polynomial time algorithm: Then $w(m)$ is bounded by a polynomial in m as is $t(m)$. The size of the the sequential circuit will also be bounded be a polynomial. And so will be the size of the unrolled circuit.

Now on to the proof.

- Let $Q : X \to \mathbb{B}$ be a problem in $\mathbf{NP}$.
- Then there is a polynomial time algorithm $qc : X \times Y \to \mathbb{B}$ for checking if an instance X of Q is a yes given evidence Y. This follows from the definition of $\mathbf{NP}$.

  - $*$ There is a $y \in Y$ such that $qc(x, y) \Leftrightarrow Q(x)$.

- For each input size n (measured in bits) the size (in bits) of the evidence needed to show an item of X is a yes instance is bounded by some polynomial n. Call it $p(n)$. This also follows from the definition of $\mathbf{NP}$.
- For each input size n, we can construct a combinational circuit cqc_n with one output that computes the value of qc. The inputs of cqc_n are n bits to represent a member of X and $p(n)$ bits to represent Y. Since qc only takes a polynomial number of steps and the number of inputs is polynomial in n, the size of cqc_n will be polynomial in n.
- Given an n-bit input $x \in X$, we can produce a specialized version of cqc_n in which the first n inputs are set to the representation of x. Call this circuit cqc_x.
  - $*$ There is a way to set the $p(n)$ input bits of cqc_x so that cqc_x outputs true $\Leftrightarrow$ there is a $y \in Y$ such that $qc(x, y)$
- cqc_x can be represented by a propositional formula ϕ_x.
  - $*$ ϕ_x is satisfiable $\Leftrightarrow$ there is a way to set the $p(n)$ input bits of cqc_x so that cqc_x outputs true
- The function that maps $x \in X$ to ϕ_x can be implemented in polynomial time and so it is a polynomial time transformation.

Aside: A problem, such as PSAT, that is both in $\mathbf{NP}$ and is $\mathbf{NP}$-hard is called $\mathbf{NP}$**-complete**.

Consequence. PSAT is $\mathbf{NP}$-complete.

Given that PSAT is a hardest problem in $\mathbf{NP}$ a natural question is

is PSAT in $\mathbf{P}$?

or to put it another way is

$$\mathbf{P} = \mathbf{NP}\ ?$$

If PSAT or any other $\mathbf{NP}$-complete problem is tractable, then every problem in $\mathbf{NP}$ is tractable.

If PSAT or any other $\mathbf{NP}$-complete problem is intractable, then every $\mathbf{NP}$-hard problem is intractable.

In fact PSAT is only patient-zero in the $\mathbf{NP}$-hardness outbreak. It turns out that PSAT can be reduced to thousands of other important problems which would be useful to solve, which means that they too are $\mathbf{NP}$-hard.

If problem PSAT can be reduced to F, then every problem $G \in \mathbf{NP}$ can be reduced to F, and so F is $\mathbf{NP}$-hard. Proof: Since G can be reduced to PSAT and PSAT can be reduced to F, there is a composite reduction from G to F.

Reducing PSAT

If we can find a poly-time reduction from some $\mathbf{NP}$-complete problem (such as PSAT) to some other problem F in $\mathbf{NP}$, then

- every problem in $\mathbf{NP}$ can be reduced to F, so
- it too is one of the hardest problems in $\mathbf{NP}$, i.e.,
- F is also $\mathbf{NP}$-complete.

Example: Showing that 3-colouring is $\mathbf{NP}$-complete.

We already know that 3-colouring is in $\mathbf{NP}$. It remains to show that we can (poly time) reduce PSAT to 3-colouring.

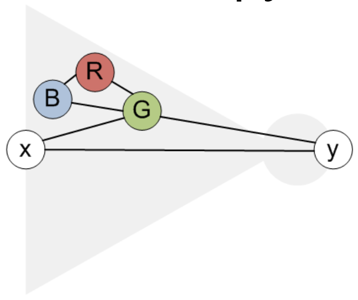
We need a polynomial time algorithm to transform any propositional formula Φ to a graph such that the graph can be 3-coloured iff the formula can be satisfied.

Here is the reduction.

Start with any propositional formula Φ.

Turn it into an equivalent circuit using only AND-gates and NOT-gates.

For each NOT-gate, make a copy of the following graph.

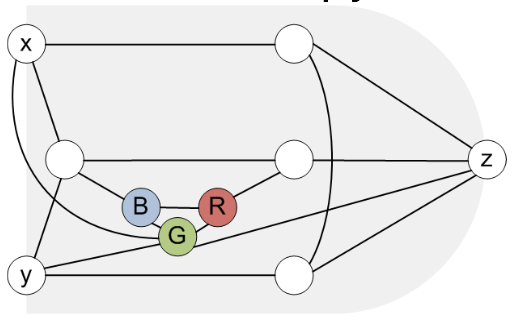


We can assume, w.l.o.g., that in any colouring B is blue, R is red and G is green.

Provided R is red, G is green, and B is blue:

- x and y are either blue or red; and
- x is red iff y is blue.

For each AND-gate, make a copy of the following graph.



Provided R is red, G is green, and B is blue:

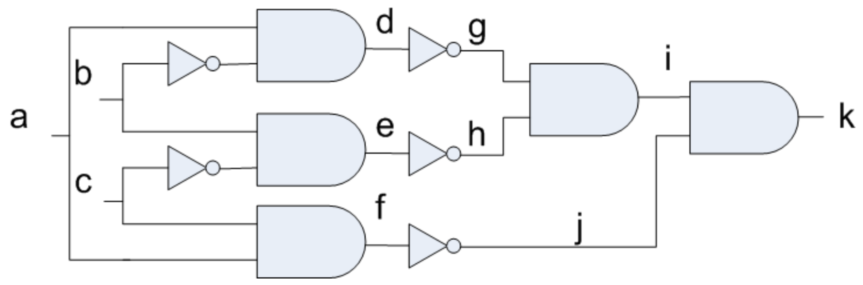
- x, y, and z are either blue or red;
- x and y are both blue iff z is blue.

Merge together

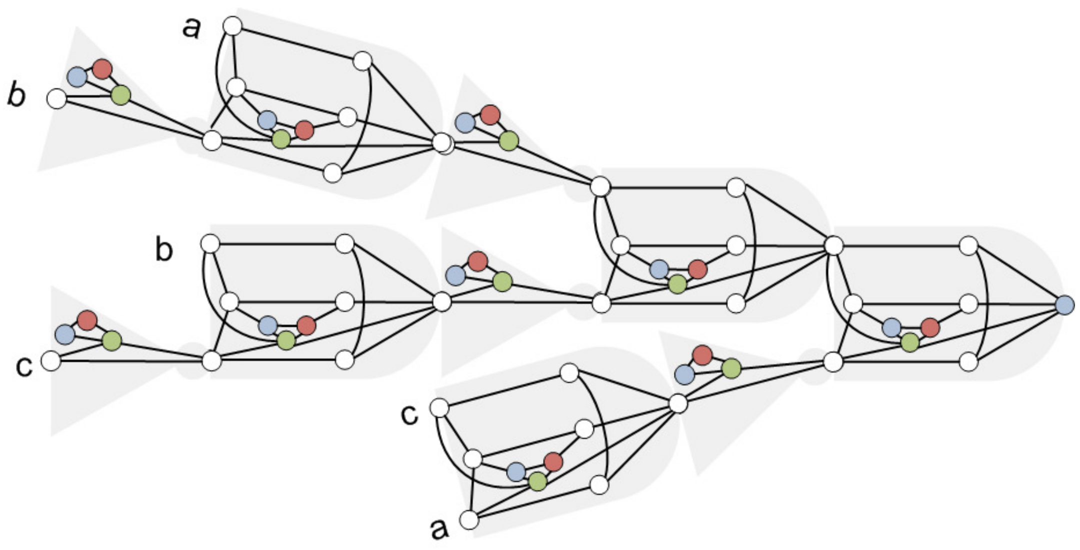
- Two nodes if one is an output port and the other is a corresponding input port
- All nodes that correspond to the same variable.
- All nodes labelled R., All nodes labelled G. All nodes labelled B.
- The final output and the B node.

Example: Φ is: $(a \Rightarrow b) \wedge (b \Rightarrow c) \wedge (c \Rightarrow \neg a)$

The circuit is this

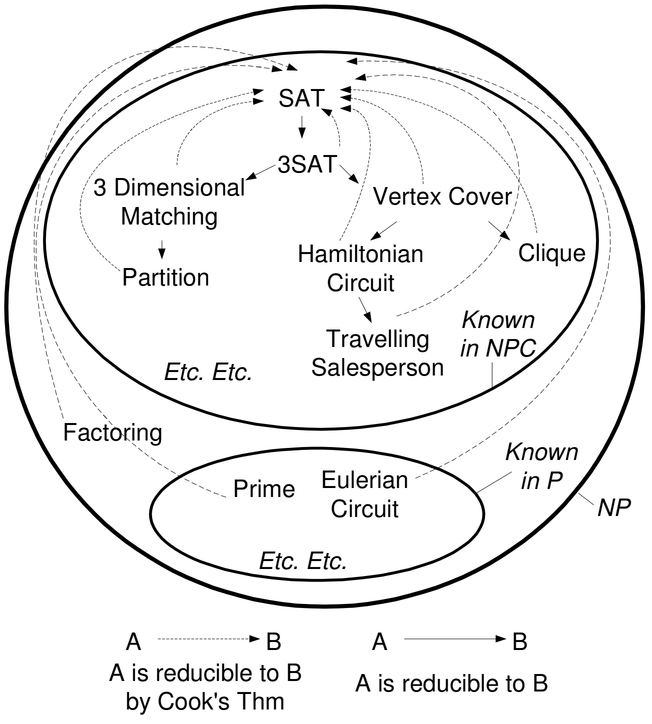


and a drawing of the final graph is this



where all nodes labelled a, b, and c are the same node and all nodes coloured red are the same node, etc.

Now the graph can be 3-coloured iff Φ can be satisfied.



Some problems in NP

1.

Note that:

- If just one $\mathbf{NP}$-complete problem can be shown to be tractable, then $\mathbf{P} = \mathbf{NP}$ meaning that all problems in $\mathbf{NP}$ are tractable.
- If just one $\mathbf{NP}$ problem can be shown to be intractable, then $\mathbf{P} \neq \mathbf{NP}$ and all $\mathbf{NP}$-complete problems are intractable.

The big picture

Two worlds

We live in one of two worlds:

A world where $\mathbf{P} = \mathbf{NP}$

In this world there is a polytime algorithm for PSAT and thus for every other problem in $\mathbf{NP}$.

A huge number of problems that are currently regarded as 'hard' will actually be 'easy'

- Scheduling problems
- Layout problems (e.g. VLSI layout)
- Planning problems
- Many artificial intelligence problems
- Automated theorem proving
  - Proving theorems will not be much harder than proof checking.
- Program verification is far easier.
- Digital circuit verification is far easier.

All methods of public-key cryptography are easily cracked.

- All your past HTTPS sessions can now be decrypted.
- e-commerce is impractical.
- All web "certificates" are easily faked. Trust no one.

A world where $\mathbf{P} \neq \mathbf{NP}$

Many problems that appear to be difficult

- actually are difficult.
- All known methods of public-key cryptography are *probably* secure.

Myths about intractability

Since computers are getting more powerful, time complexity will become irrelevant.

- If the best algorithm is $\Theta(2^n)$ and
- computer speed doubles each year, then
- every year we can increase n by 1 while keeping computation time the same.

Parallel computing will allow hard problems to be solved quickly

- If the best algorithm is $\Theta(2^n)$ and
- we throw twice as much hardware at it, then
- we can increase n by 1.

Quantum computers will solve all $\mathbf{NP}$ *problems.*

- There are fast quantum algorithms known for fast factoring.
- However not (yet) for any $\mathbf{NP}$-hard problem.
  - $*$ (Note factoring is not known to be $\mathbf{NP}$-hard.)
- Worst-case scenario:
  - $*$ $\mathbf{P} \neq \mathbf{NP}$ so $\mathbf{NP}$-hard problems really are intractable
  - $*$ but public-key cryptography is made insecure by QC.

Being $\mathbf{NP}$*-hard (or* $\mathbf{NP}$*-complete) is the 'kiss of death' for a problem*

1. Remember that $\mathbf{NP}$-hardness is all based on the worst case complexity.

  - $*$ In fact many $\mathbf{NP}$-hard problems can be solved quickly on many useful inputs.
  - $*$ For example PSAT can be solved quite quickly for large instances resulting from real-world problems (such as digital circuit verification).
  - $*$ Program verification systems such as Spec# rely on theorem provers and model-checkers that solve $\mathbf{NP}$-hard problems.

2. For many optimization problems (e.g., travelling salesman), near-optimal solutions can be found in polynomial time. Greedy algorithms are often reasonably good most of the time.

Further reading

David Harel

- *Algorithmics: The spirit of computing*
  - Easy to read. Not terribly technical.

Steve Cook

- Official Problem Definition
  - http://www.claymath.org/millennium/P_vs_NP/
  - Fairly technical, but self contained.

Scott Aaronson

- Quantum Computing and the Limits of the Efficiently Computable
  - http://www.youtube.com/watch?v=8bLXHvH9s1A