Turing Machines

Vermont Technical College
Peter C. Chapin

Hilbert's Problems

- David Hilbert gave a talk at the International Congress of Mathematicians in 1900 (Paris, France).
- Posed several problems, all unsolved at the time, to challenge the community.
  - “Can solutions be found during the 20th century?”
- Problems chosen because Hilbert felt their solutions would be enlightening.

Diophantine Equations

- Problem #10: *Find an "effective procedure" to determine if any given Diophantine equation with integer coefficients has an integer solution.*
- Diophantine equations are polynomials with arbitrary number of unknowns.
  - Do the equations below have integers x, y, z, and w that satisfy them? How can you tell?

$$6x + 3y^2 - 2z^4 = 0$$

$$19x^2 - 3y^4 + 4z^8 - 13w^5 = 0$$

“Effective Procedure?”

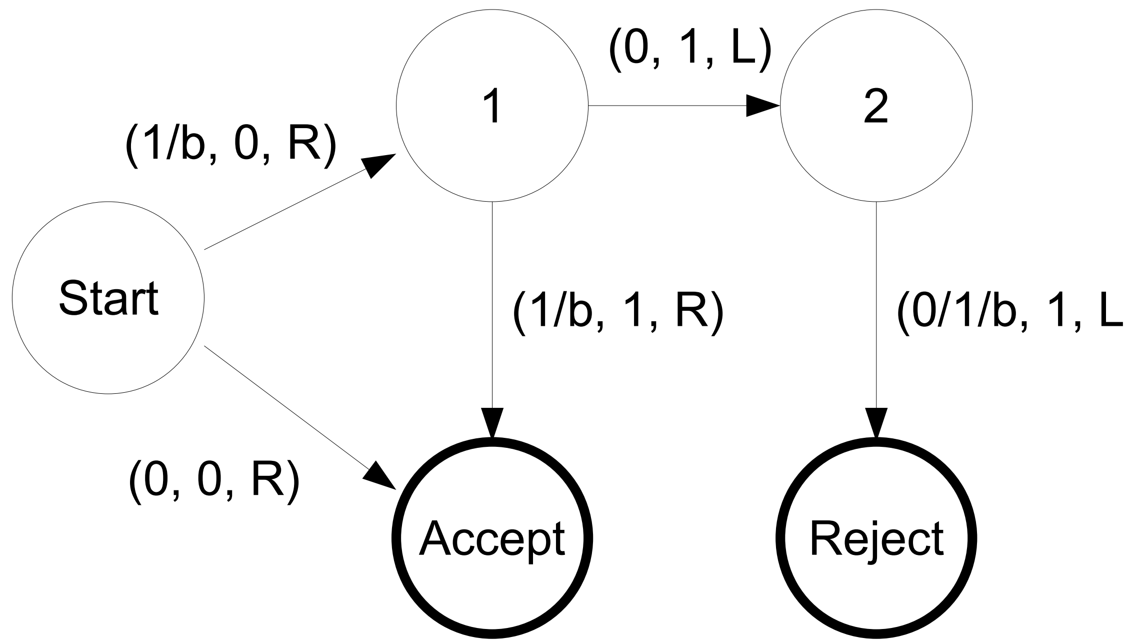
- Hilbert used the term “effective procedure”
  - Today we would call it an algorithm.
  - But what does that really mean?
- 1930s... much research done on the subject of computability.
  - What does it mean to say something is computable?
  - What are the limits of computability?
  - What, exactly, is an algorithm?
- Must consume finite resources!

Turing Machines

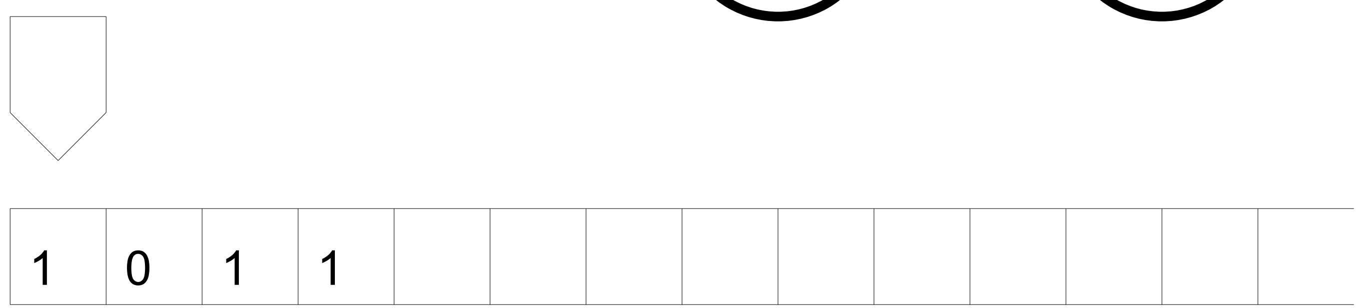
- Alan Turing devised a *model of computation* now called a “Turing Machine.”
  - Very simple theoretical device.
  - Not a real machine that you would build or use.
- Turing used his machines to reason about the nature of computation
  - ... and it's limits.

The Machine

State machine that forms the Turing machine's "program." Describes the action of the machine.



Head that can read or write symbols (or blanks) to the tape. Head can move left or right one position at a time.



"Tape" with each cell either blank for filled with a symbol from a finite "tape alphabet." The symbols 1 and 0 are sufficient, but other alphabets are also okay. The tape is *indefinitely* long to the right.

How It Works

- The input is put on the tape.
- The machine is initialized:
  - The head is put over the leftmost cell.
  - The machine is put into the start state.
- The machine makes “moves” as follows:
  - It reads the tape.
  - Based on the current state and the symbol read
    - It writes a symbol onto the tape.
    - Moves the head.
- Execution continues until accept or reject.

Model of Imperative Languages

- Turing Machines simulate imperative languages
  - Program reads/writes to tape (memory).
  - Thus TM programs use mutable data.
  - Memory (tape) contents control machine's action by directing it into different states.

Church-Turing Thesis

- *A Turing Machine can compute every computable function.*
  - Not provable because we don't have a good definition of "computable function." So...
- Defn: *An algorithm is that which can be computed on a Turning Machine.*
  - We use a Turning machine to provide that definition.
- Rationale: <u>No model of computation has ever been found that can compute more things than a Turing machine can compute!</u>

Amazing!

- Such a simple device...
  - Yet it can simulate all other models of computation.
- How?
  - Tape input is entirely general... any kind of data can be encoded.
  - Machine can read/write the tape, including arbitrary blank space at the end.
  - Has unlimited space and time available to it.
- Your laptop computer is no more powerful.
  - In fact, it is less powerful!

Can It Compute Everything?

- No!
  - Some problems can not be solved by a Turning machine!
  - Such problems are said to be “undecidable.” Their solutions are beyond the reach of computers to answer.

Post Correspondence Problem

Several infinitely high stacks of tiles. Every tile in a stack is the same type.

A	BB	C
ABB	B	AC

Draw tiles from the stacks and arrange to match the top string and bottom string.

A	BB	BB	= ABBBB
ABB	B	B	= ABBBB

Match found!

Is there an algorithm that, given a collection of tile types, can answer “yes” or “no” depending on if a match exists or not?

No such algorithm exists!

Halting Problem

- Given Turing machine M, encode it's program in some suitable way, <M>.
- Put <M>, together with an input string w, onto a Turing machine tape.
- Write a program for this other machine that answers: “Does M halt when given w as input?”
- **No such algorithm exists!**

Undecidability Everywhere!

- In fact, most interesting properties of software are undecidable.
  - Can a compiler know, in general, when it has fully optimized a piece of code?
    - NO!
  - Can you statically analyze a program to see if it has some useful security property?
    - Generally NO!... depending on the property.
  - Can you statically analyze a program to make sure it has no infinite loops?
    - NO! You are trying to solve the halting problem!

Hilbert's 10^{th} Problem Revisited

- We can now state Hilbert's 10^{th} problem more precisely.
  - Let <D> be a suitable encoding of a Diophantine equation. Can a Turning machine program be written that accepts <D> if the equation has integer solutions and rejects it otherwise?
  - RESOLVED: The Matiyasevich theorem, proved in the 1970, shows that the question is undecidable.
  - **No such algorithm exists.**
  - Hilbert was right: resolving this problem was very enlightening.

Why Do We Care?

- A Turing Machine is the theoretical basis of all imperative programming languages.
  - The steps taken by the program are like the states of the TM.
  - The memory read/written by the program is like the TM's tape.
- Any programming language that can simulate a Turing machine is *Turing Complete*.
  - It is thus capable of computing all things computable.
- **All useful PLs are Turing Complete**

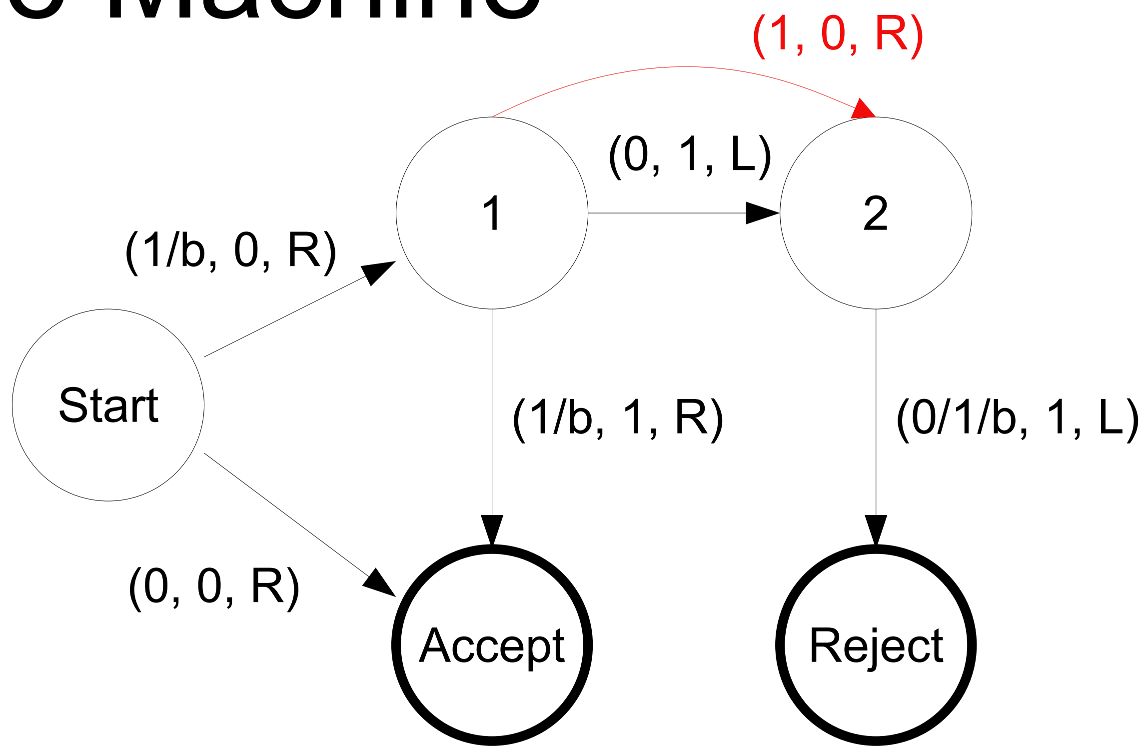
Non-Deterministic TMs

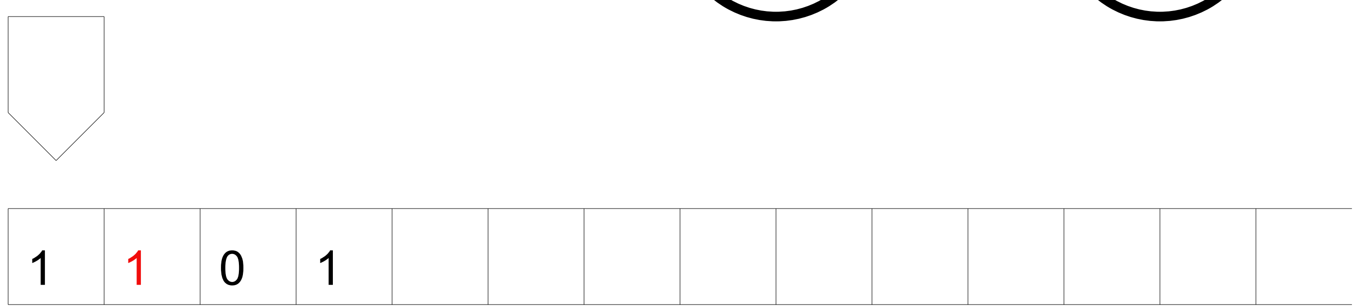
- So far we've covered *deterministic* TMs
  - Only one choice in state diagram for a given tape symbol. (Note: no choice is understood to mean a transition to REJECT).
- A *non-deterministic* TM allows multiple transitions from a state for the same tape symbol.

The Machine

When a choice is possible,
the machine splits with each
branch taking a different choice.
All branches make the next
move in parallel, etc.

Execution halts when one branch
reaches ACCEPT or **all** branches
reach REJECT.





More Powerful?

- Clearly a ND Turing Machine can do everything a deterministic one can do.
  - It doesn't even have to use its non-determinism
- Can a deterministic TM do everything a ND Turning Machine can do?
  - Yes!
  - The proof shows how a deterministic machine can simulate the action of the ND TM. It takes advantage of the indefinite tape size to simulate the states and tapes of all branches.

Running Time?

- A TM can execute in polynomial time any algorithm a "normal" computer can execute in polynomial time.
  - But typically with a higher degree polynomial since tape (memory) access is O(n).
- However, a ND TM can make an exponentially large number of branches
  - Consider a two-way choice at each step in all branches: 2, 4, 8, 16, etc.
  - All branches run in parallel

Running Time

- In n steps, a ND TM can create (e. g., 2^n) branches, each of which could run a polynomial time algorithm, all in parallel.
- A deterministic TM would need $O(2^n)$ time to simulate this.

Polynomial Time Checkers

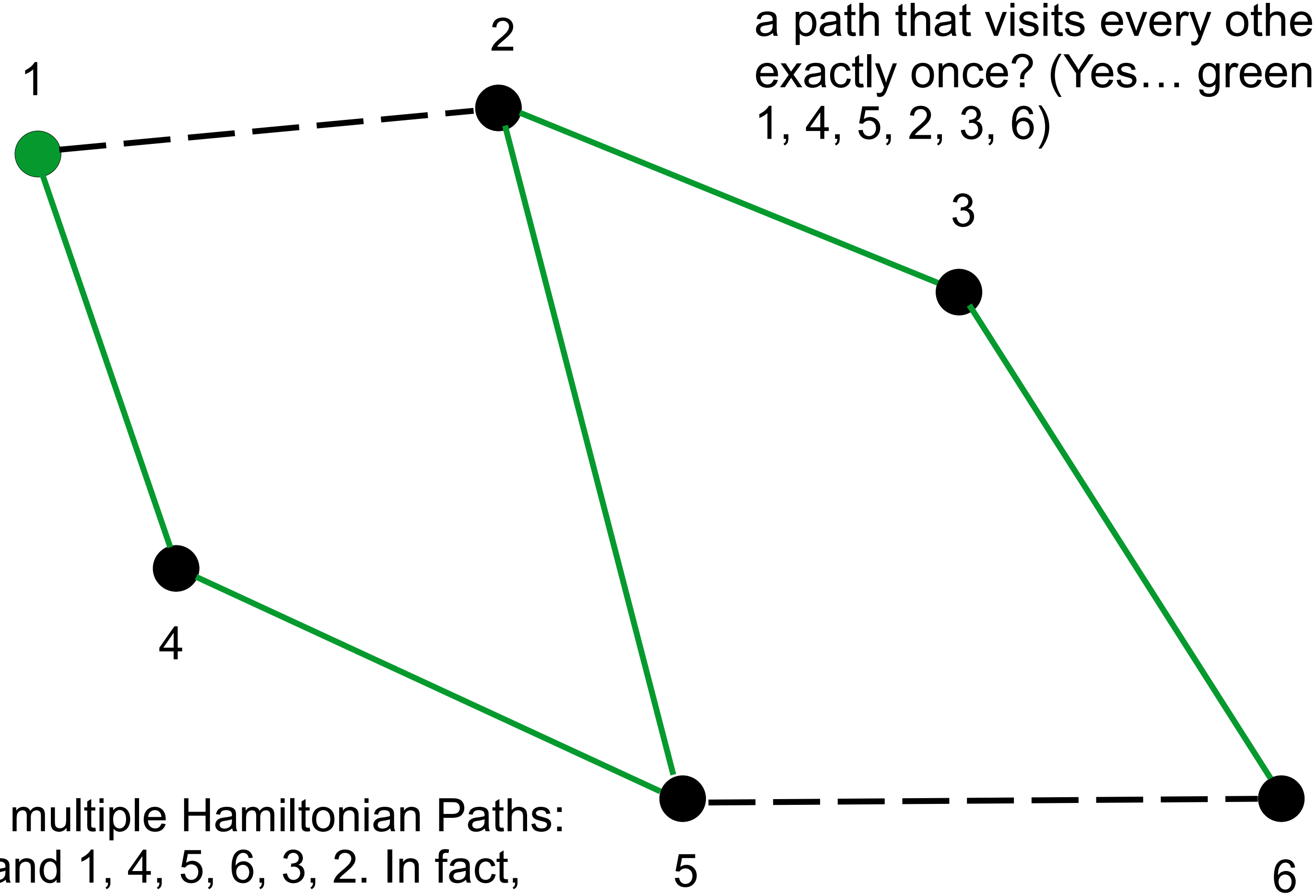
- A polynomial time checker is an algorithm that can verify a solution to a problem in polynomial time.
- Imagine enumerating all potential solutions and then using such a checker to find one that is an actual solution.
  - If there were exponentially many such potential solutions, a deterministic TM would require exponential time to do this.
  - A ND TM could do this in polynomial time.

P vs NP

- P
  - The class (set) of problems which a deterministic TM can solve in polynomial time.
- NP
  - The class (set) of problems which a ND TM can solve in polynomial time.
- Clearly $P \subseteq NP$
  - … since a ND TM can just use whatever algorithm works for the deterministic machine.

Hamiltonian Path

Starting at a given vertex is there a path that visits every other vertex exactly once? (Yes… green path: 1, 4, 5, 2, 3, 6)



This graph has multiple Hamiltonian Paths: 1, 2, 3, 6, 5, 4 and 1, 4, 5, 6, 3, 2. In fact, 1, 2, 3, 6, 5, 4, 1 is a *Hamiltonian Cycle*.

Hamiltonian Path

- There is a polynomial time checker:
  - Walk the path in *O(V)* time and…
    - Check off each vertex that is encountered
    - Verify that no vertex is encountered twice
- One algorithm for finding the Hamiltonian path:
  - Enumerate all permutations of vertexes: *O(n!)*
  - Execute the checker for each permutation
  - Halt when the a valid path is found.

ND TM Approach

- The non-deterministic TM can do this quickly:
  - Create a separate branch for each possible permutation of vertexes
  - Run the checker on all permutations at once
  - Halt if any of the branches accept
  - This is polynomial time!
- Thus *Hamiltonian Path is in NP*
  - But <u>is it in P also?</u> Can you think of a way to solve this problem on a deterministic machine that will run in polynomial time?

3SAT

- Another example: 3SAT
  - Consider a finite set of boolean variables, x_1, x_2, …, x_n.
  - Let E be a boolean expression which is a conjunction of disjunctions. Every disjoint involves at most 3 variables.
  - $E = (X_1 \lor (\neg X_3) \lor X_4) \land ((\neg X_2) \lor X_3 \lor (\neg X_4)) \land (X_1 \lor X_3 \lor (\neg X_4))$
  - The problem is to find values for the boolean variables that satisfy (“SAT”) the expression (i. e., make it True)

3SAT Verifier

- There is an obvious polynomial time verifier:
  - Given a proposed solution, substitute the values into the expression and evaluate it.
- Thus we have this algorithm for solving 3SAT:
  - Enumerate all possible sets of values for the variables: $O(2^n)$
  - Run the verifier on each potential solution until one is found that works (if any).
- *3SAT is in NP*

SAT

- 3SAT is a special case of the satisfiability problem (SAT).
  - In SAT, the number of disjoints in each disjunction can be any number, not just 3.
  - It is clear that *SAT is also in NP*.

Polynomial Time Reduction

- Let…
  - … P_1 and P_2 be two different problems (e. g., P_1 might be SAT and P_2 might be 3SAT).
  - … X be an *instance* of P_1 and Y be an instance of P_2.
- If there is an algorithm that runs in polynomial time that can convert X to Y…
  - … we say that P_1 can be *reduced* to P_2.

Polynomial Time Reduction

- **If** …
  - … P_1 can be reduced to P_2
  - … **and** there is a polynomial time solver for P_2
  - … **then** there is a polynomial time solver for P_1
- For each instance of P_1 ...
  - … reduce it (quickly) to an instance of P_2
  - … solve the P_2 instance (quickly)
- Note: *It is also important to transform the solution of the P_2 instance back to a solution of the P_1 instance "quickly."*

SAT vs 3SAT

- SAT can be reduced to 3SAT
  - Proof: *… elided …* (see any textbook on computational complexity)
  - This means you can transform a more general SAT instance into a more restricted 3SAT instance.
- So what??

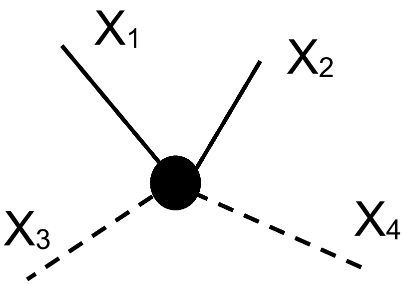
NP-Complete

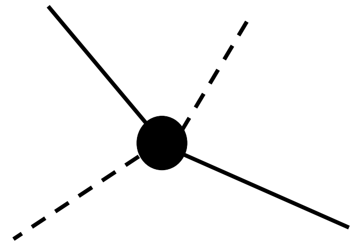
- It turns out that: *<u>any</u> problem in NP can be reduced to SAT!*
  - Wow, really?
  - Yes!
  - Proof: *… elided …* (Cook's Theorem)
- Thus
  - *3SAT is also NP-Complete*
  - Because any problem in NP can be reduced to 3SAT by first reducing it to SAT and then reducing the SAT instance to 3SAT.

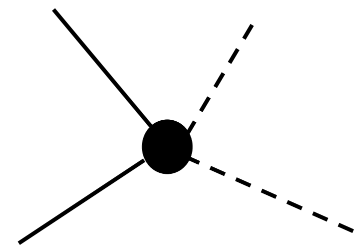
Reduction of Hamiltonian Cycle

- Let’s convert an instance of Hamiltonian Cycle to SAT…
  - Assign a boolean variable to every edge.
    - The variable is True if that edge is part of the cycle; False otherwise.
  - At each vertex there must be exactly two edges that are part of the cycle. No more no less.
    - The cycle must enter the vertex and exit it.
    - The cycle must only enter/exit once.

Reduction of Hamiltonian Cycle

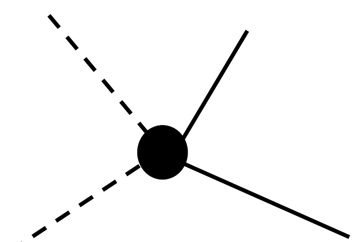


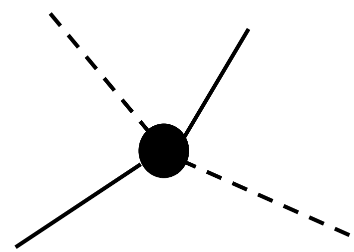




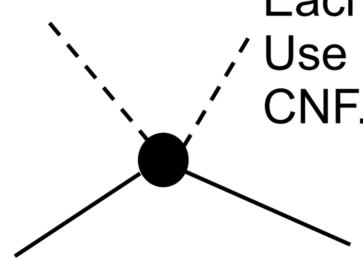
$$X_1 \wedge X_2 \wedge (\neg X_3) \wedge (\neg X_4)$$

etc...





Combine expressions for each configuration
Using OR. Form a similar expression for
Each vertex and combine all those with AND.
Use boolean algebra to reformulate into
CNF. *The result is a SAT instance*.



Reduction of Hamiltonian Cycle

- Suppose you had an efficient 3SAT solver…
  - You can now solve Hamiltonian Cycle efficiently
  - Reduce your Hamiltonian Cycle instance to SAT
  - Reduce your SAT instance to 3SAT
  - Use your efficient SAT solver

P = NP?

- Armed with an efficient 3SAT solver you can…
  - … efficiently solve EVERY problem in NP
  - In that case, P = NP. The two complexity classes are the same.
- The same is true for any NP-complete problem.
  - If you can find an efficient solver for even one of them (and there are many)…
  - … *you can efficiently solve them all!*

Hamiltonian Cycle

- Hamiltonian Cycle is in NP
  - It is also NP-complete
- To prove this you must find a way of reducing instances of a known NP-complete problem to instances of Hamiltonian Cycle.
  - SAT to 3SAT
  - 3SAT to VC (Vertex Cover)
  - VC to HC (Hamiltonian Cycle)

Is Every Problem NP-Complete?

- No!
  - It can be shown that if P != NP there must be some problems that are in NP but that are ***not*** NP-complete
  - Those problems can be reduced to any NP-complete problem, of course, but no NP-complete problem can be reduced to them (since that would make them NP-complete also)

P != NP

- Although not known for sure, this is the expected reality
  - Thus there are some problems (the NP-complete ones) for which no efficient solution is possible.
  - There are also problems in NP that are not NP-complete but for which no efficient solution is possible, but it’s less clear which they are.
  - For any given problem that isn’t NP-complete, maybe we are not smart enough to find a fast way to solve it. That is, maybe the problem is really in P.

Unfortunately...

- Many problems of interest are NP-complete
  - There are dozens of known NP-compete problems
  - Mostly all have been proved NP-complete by finding a reduction from a previously known NP-complete problem
  - *Except for SAT*. The proof of SAT's NP-completeness was done from first principles (and is based on Turning Machines)

Software Engineering?

- What's a software engineer to make of this?
  - Give up? Of course not!
- Note…
  - In some applications *n* is small so exponential time isn't actually a problem.
  - Often there are approximation algorithms that solve these problems correctly in most cases or to a good approximation in most cases.
  - When up against an NP-complete problem, don't expect to find a fast way to always solve it! Know your limits.