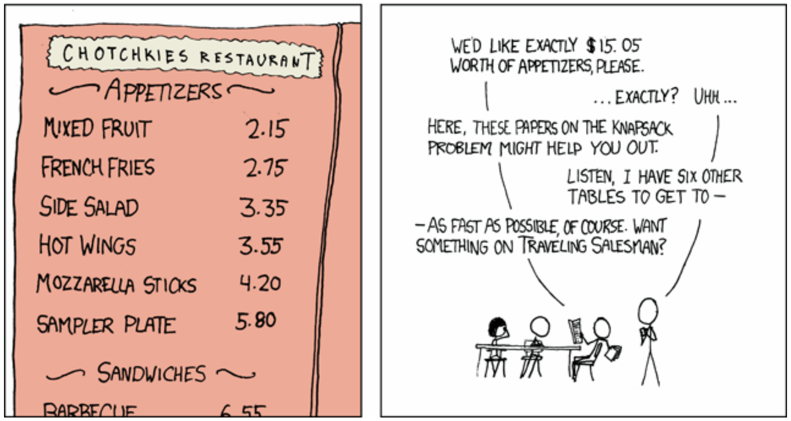
NP-Completeness!



Xkcd.com

Slides adapted from Ran Libeskind-Hadas,David Kauchak

Run-time analysis

We've spent a lot of time in this class putting algorithms into specific run-time categories:

- O(log n)
- O(n)
- O(n log n)
- O(n^2)
- O(n log log n)
- O($n^{1.67}$)
- …

When I say an algorithm is O(f(n)), what does that mean?

"Easy" and "Hard" problems…

(a first attempt at a definition)

- Easy: The problem can be solved in polynomial time: n^c

Some constant

problem size

- Hard: The problem *cannot* be solved in polynomial time, but it can be solved in "exponential time:" 2^{n^c}

There are also "very hard", "very very hard", "incredibly super hard" problems, etc.

---

n^2 and n^3 versus 2^n

The Mudd-O-Matic performs 10^9 operations/sec

	n = 10	n = 30	n = 50	n = 70
n^2	100 < 1 sec	900 < 1 sec	2500 < 1 sec	4900 < 1 sec
n^3	1000 < 1 sec	27000 < 1 sec	125K < 1 sec	343K < 1 sec
2^n	1024 < 1 sec	10^9 1 sec		

n^2 and n^3 versus 2^n

The Mudd-O-Matic performs 10^9 operations/sec

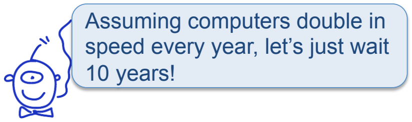
	n = 10	n = 30	n = 50	n = 70
n^2	100 < 1 sec	900 < 1 sec	2500 < 1 sec	4900 < 1 sec
n^3	1000 < 1 sec	27000 < 1 sec	125K < 1 sec	343K < 1 sec
2^n	1024 < 1 sec	10^9 1 sec	13 days	

---

n^2 and n^3 versus 2^n

The Mudd-O-Matic performs 10^9 operations/sec

	n = 10	n = 30	n = 50	n = 70
n^2	100 < 1 sec	900 < 1 sec	2500 < 1 sec	4900 < 1 sec
n^3	1000 < 1 sec	27000 < 1 sec	125K < 1 sec	343K < 1 sec
2^n	1024 < 1 sec	10^9 1 sec	13 days	37 thousand years



---

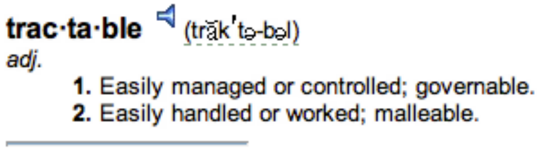
n^2 and n^3 versus 2^n

The Ran-O-Matic performs 10^9 operations/sec

	n = 10	n = 30	n = 50	n = 70
n^2	100 < 1 sec	900 < 1 sec	2500 < 1 sec	4900 < 1 sec
n^3	1000 < 1 sec	27000 < 1 sec	125K < 1 sec	343K < 1 sec
2^n	1024 < 1 sec	10^9 1 sec	13 days	37 thousand years

37 thousand years -> 37 years

Tractable vs. intractable problems



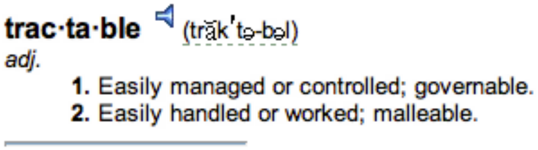
trac·ta·ble (trăk'tə-bəl)

adj.

1. Easily managed or controlled; governable.
2. Easily handled or worked; malleable.

What is a “tractable” problem?

Tractable vs. intractable problems

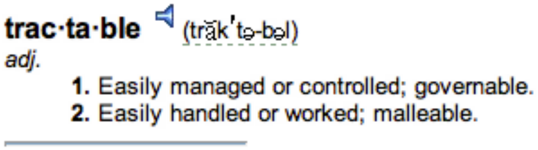


trac·ta·ble (trăk'tə-bəl)
adj.
1. Easily managed or controlled; governable.
2. Easily handled or worked; malleable.

Tractable problems can be solved in O(f(n)) where f(n) is a polynomial

---

Tractable vs. intractable problems



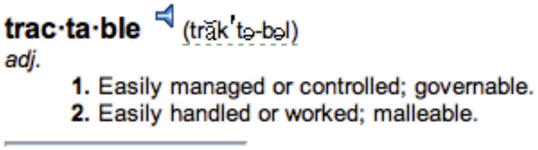
trac·ta·ble (trăk'tə-bəl)
adj.
1. Easily managed or controlled; governable.
2. Easily handled or worked; malleable.

What about…

$O(n^{100})$?

$O(n^{\log \log \log \log n})$?

Tractable vs. intractable problems

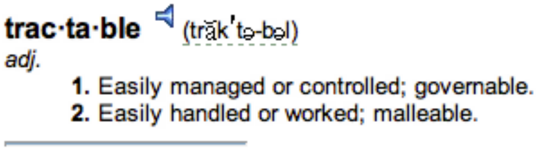


trac·ta·ble (trăk'tə-bəl)
adj.
1. Easily managed or controlled; governable.
2. Easily handled or worked; malleable.

Technically $O(n^{100})$ is tractable by our definition

Why don't we worry about problems like this?

Tractable vs. intractable problems

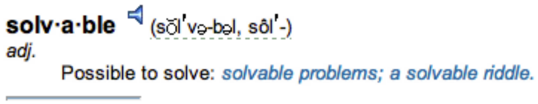


trac·ta·ble (trăk'tə-bəl)
adj.
1. Easily managed or controlled; governable.
2. Easily handled or worked; malleable.

Technically $O(n^{100})$ is tractable by our definition

- Few practical problems result in solutions like this
- Once a polynomial time algorithm exists, more efficient algorithms are usually found
- Polynomial algorithms are amenable to parallel computation

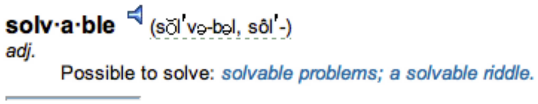
Solvable vs. unsolvable problems



solv·a·ble (sŏl′və-bəl, sôl′-)
adj.
Possible to solve: *solvable problems; a solvable riddle.*

What is a “solvable” problem?

Solvable vs. unsolvable problems



solv·a·ble (sŏl′və-bəl, sôl′-)
adj.
Possible to solve: *solvable problems; a solvable riddle.*

A problem is solvable if given enough (i.e. finite) time you could solve it

---

Sorting

Given n integers, sort them from smallest to largest.

Tractable/intractable?

Solvable/unsolvable?

Sorting

Given n integers, sort them from smallest to largest.

Solvable and tractable:
Mergesort: $\Theta(n \log n)$

Enumerating all subsets

Given a set of n items, enumerate all possible subsets.

Tractable/intractable?

Solvable/unsolvable?

Enumerating all subsets

Given a set of n items, enumerate all possible subsets.

Solvable, but intractable: $\Theta(2^n)$ subsets

For large n this will take a very, very long time

Halting problem

Given an arbitrary algorithm/program and a particular input, will the program terminate?

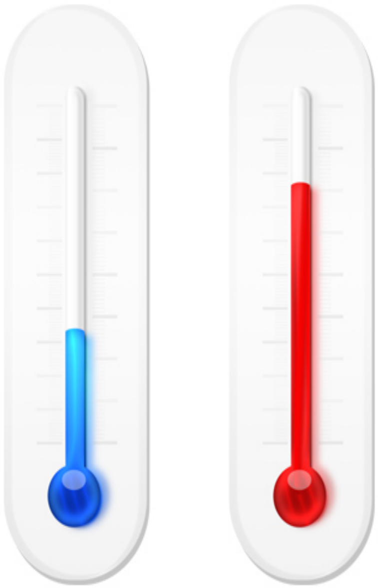
Tractable/intractable?

Solvable/unsolvable?

Halting problem

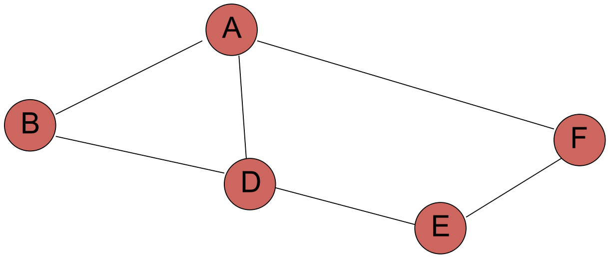
Given an arbitrary algorithm/program and a particular input, will the program terminate?

Unsolvable ☹



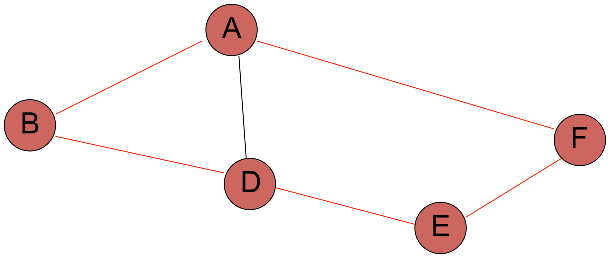
Hamiltonian cycle

Given an undirected graph G=(V, E), a hamiltonian cycle is a cycle that visits every vertex V exactly once



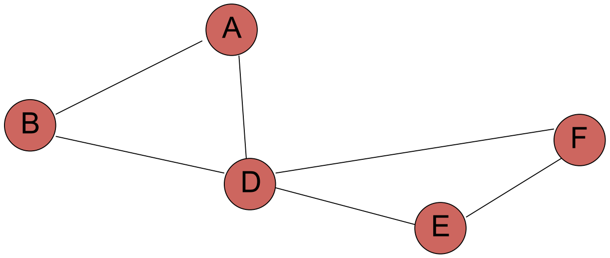
Hamiltonian cycle

Given an undirected graph G=(V, E), a hamiltonian cycle is a cycle that visits every vertex V exactly once



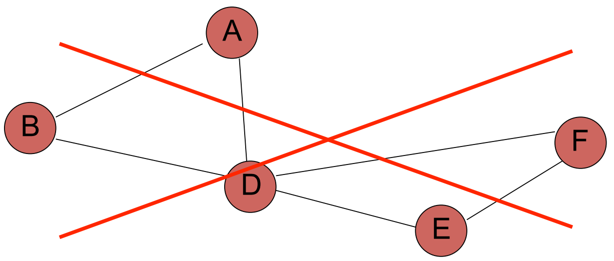
Hamiltonian cycle

Given an undirected graph G=(V, E), a hamiltonian cycle is a cycle that visits every vertex V exactly once



Hamiltonian cycle

Given an undirected graph G=(V, E), a hamiltonian cycle is a cycle that visits every vertex V exactly once



Hamiltonian cycle

Given an undirected graph, does it contain a hamiltonian cycle?

Tractable/intractable?

Solvable/unsolvable?

Write your guesses on your worksheet!

Hamiltonian cycle

Given an undirected graph, does it contain a hamiltonian cycle?

Solvable: Enumerate all possible paths (i.e. include an edge or don't) check if it's a Hamiltonian cycle

NP problems

NP is the set of problems that can be *verified* in polynomial time

A problem can be verified in polynomial time if you can check that a given solution is correct in polynomial time

(NP is an abbreviation for non-deterministic polynomial time)

Hamiltonian cycle

Given an undirected graph, does it contain a hamiltonian cycle?

Solvable: Enumerate all possible paths (i.e. include an edge or don't) check if it's a Hamiltonian cycle

How would we do this check exactly, specifically given a graph and a path?

Hamiltonian cycle

Given an undirected graph, does it contain a hamiltonian cycle?

Worksheet: Can a Hamiltonian cycle be verified in polynomial-time?

Checking hamiltonian cycles

```
HAM-CYCLE-VERIFY($G, p$)
for $i \leftarrow 1$ to $|V|$
    $visited[i] \leftarrow false$
$n \leftarrow length[p]$
if $p_1 \neq p_n$ or $n \neq |V| + 1$
    return $false$
$visited[p_1] \leftarrow true$
for $i \leftarrow 1$ to $n - 1$
    if $visited[p_i]$
        return $false$
    if $(p_i, p_{i+1}) \notin E$
        return $false$
    $visited[p_i] \leftarrow true$
for $i \leftarrow 1$ to $|V|$
    if $!visited[i]$
        return $false$
return $true$
```

Checking hamiltonian cycles

```
HAM-CYCLE-VERIFY($G, p$)
for $i \leftarrow 1$ to $|V|$
    $visited[i] \leftarrow false$
$n \leftarrow length[p]$
if $p_1 \neq p_n$ or $n \neq |V| + 1$
    return $false$
$visited[p_1] \leftarrow true$
for $i \leftarrow 1$ to $n - 1$
    if $visited[p_i]$
        return $false$
    if $(p_i, p_{i+1}) \notin E$
        return $false$
    $visited[p_i] \leftarrow true$
for $i \leftarrow 1$ to $|V|$
    if $!visited[i]$
        return $false$
return $true$
```

Make sure the path starts and ends at the same vertex and is the right length

Can't revisit a vertex

Edge has to be in the graph

Check if we visited all the vertices

Checking hamiltonian cycles

```
HAM-CYCLE-VERIFY($G, p$)
for $i \leftarrow 1$ to $|V|$
    $visited[i] \leftarrow false$
$n \leftarrow length[p]$
if $p_1 \neq p_n$ or $n \neq |V| + 1$
    return $false$
$visited[p_1] \leftarrow true$
for $i \leftarrow 1$ to $n - 1$
    if $visited[p_i]$
        return $false$
    if $(p_i, p_{i+1}) \notin E$
        return $false$
    $visited[p_i] \leftarrow true$
for $i \leftarrow 1$ to $|V|$
    if $!visited[i]$
        return $false$
return $true$
```

Running time?

O(V) adjacency matrix
O(V+E) adjacency list

What does that say about the hamilonian cycle problem?

It belongs to NP

NP problems

Why might we care about NP problems?

- If we can't verify the solution in polynomial time then an algorithm cannot exist that determines the solution in this time (why not?)
- All algorithms with polynomial time solutions are in NP

The NP problems that are currently not solvable in polynomial time *could in theory be solved in polynomial time*

So what is NP really?

- P: The set of decision problems that can be *solved* in time polynomial in the problem size
- NP: The set of decision problems whose solutions can be *verified* in time polynomial in the problem size

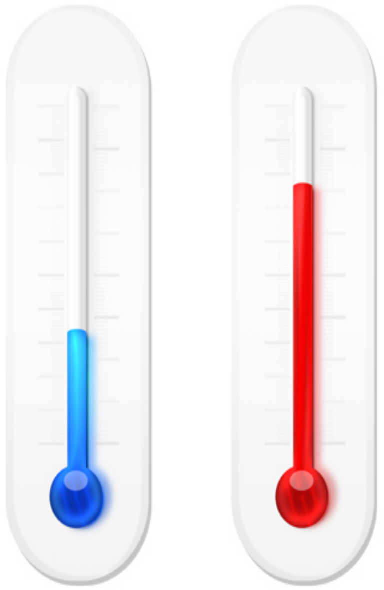
Examples of problems in P:

- Shortest path decision
- MST decision
- Network flow decision

Example of problems in NP:

- All of the above!
- 3SAT
- Vertex Cover
- Traveling salesperson

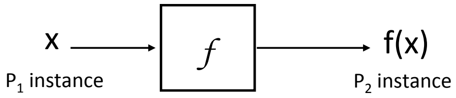
Notice that P is a subset of NP! But whether the two sets are equal is one of the most famous open problems in computer science and mathematics!



Reduction function

Given two problems P_1 and P_2 a *reduction function*, *f(x),* is a function that transforms a problem instance *x* of type P_1 to a problem instance of type P_2

such that: a solution to *x* exists for P_1 iff a solution for *f(x)* exists for P_2

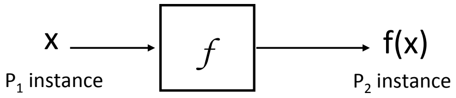


Reduction function

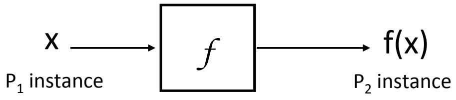
Where have we seen reductions before?

- Bipartite matching reduced to flow problem
- All pairs shortest path *through a particular vertex* reduced to single source shortest path

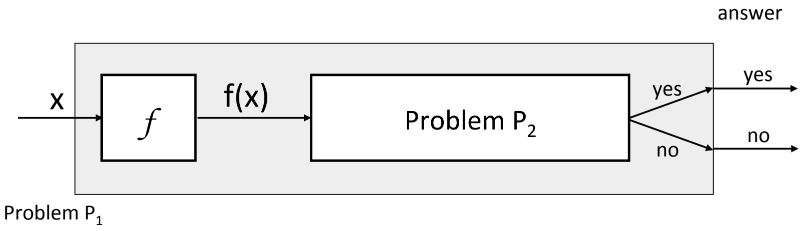
Why are they useful?



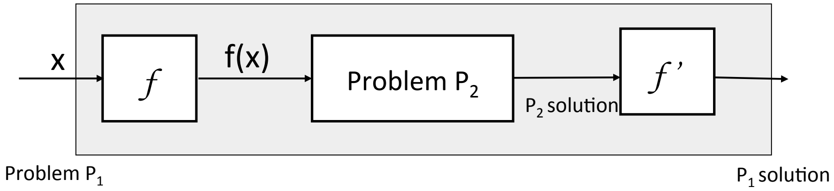
Reduction function



Allow us to solve P_1 problems if we have a solver for P_2

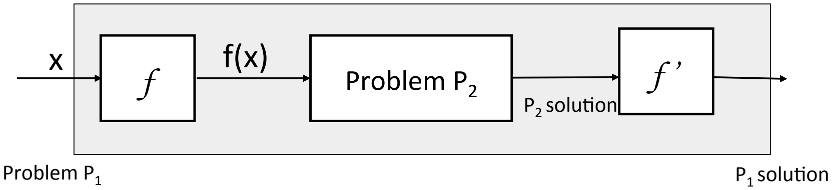


Reduction function



Most of the time we'll worry about yes no question, however, if we have more complicated answers we often just have to do a little work to the solution to the problem of P_2 to get the answer

Reduction function: Example

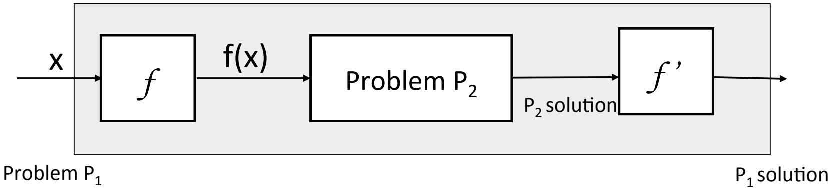


P1 = Bipartite matching
P2 = Network flow

Reduction function (f): Given *any* bipartite matching problem turn it into a network flow problem

What is *f* and what is *f'*?

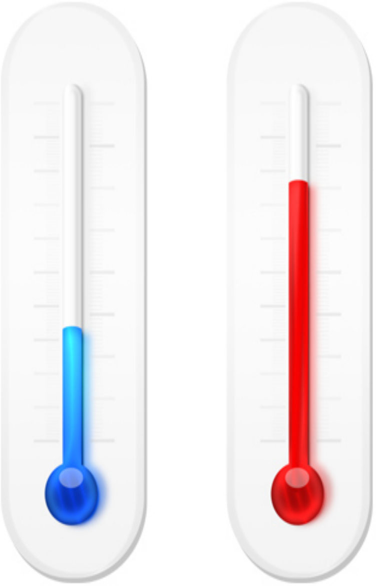
Reduction function: Example



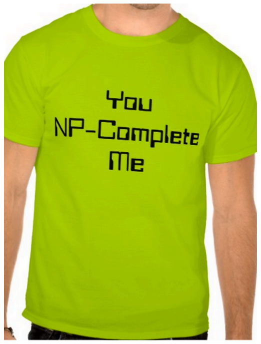
P1 = Bipartite matching
P2 = Network flow

Reduction function (f): Given *any* bipartite matching problem turn it into a network flow problem

A reduction function reduces problems instances



NP-Complete



A problem is *NP-complete* if:

1. it can be verified in polynomial time (i.e. in NP)
2. *any* NP-complete problem can be reduced to the problem in polynomial time (is NP-hard)

The hamiltonian cycle problem is NP-complete

What are the implications of this?
What does this say about how hard the hamiltonian cycle problem is compared to other NP-complete problems?

NP-Complete

A problem is *NP-complete* if:

1. it can be verified in polynomial time (i.e. in NP)
2. *any* NP-complete problem can be reduced to the problem in polynomial time (is NP-hard)

The hamiltonian cycle problem is NP-complete

It's *at least as hard* as *any* of the other NP-complete problems

---

NP-Complete

A problem is *NP-complete* if:

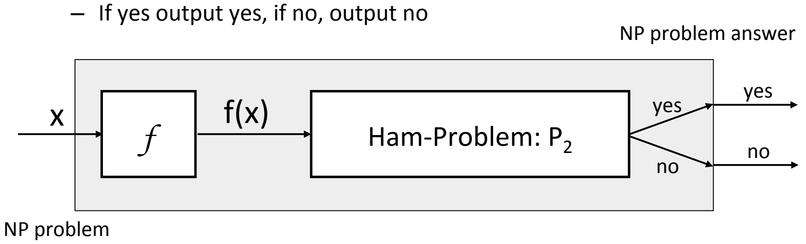
1. it can be verified in polynomial time (i.e. in NP)
2. *any* NP-complete problem can be reduced to the problem in polynomial time (is NP-hard)

If I found a polynomial-time solution to the hamiltonian cycle problem, what would this mean for the other NP-complete problems?

NP-complete

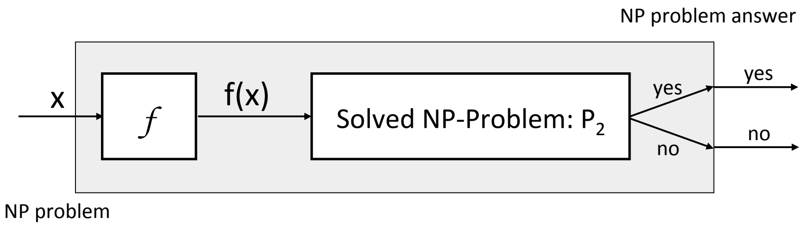
If a polynomial-time solution to the hamiltonian cycle problem is found, we would have a polynomial time solution to *any* NP-complete problem

- Take the input of the problem
- Convert it to the hamiltonian cycle problem (by definition, we know we can do this in polynomial time)
- Solve it
- If yes output yes, if no, output no



NP-complete

Similarly, if we found a polynomial time solution to *any* NP-complete problem we'd have a solution to *all* NP-complete problems



NP-complete problems

Longest path

Given a graph G with nonnegative edge weights does a simple path exist from *s* to *t* with weight at least *g*?

Integer linear programming

Linear programming with the constraint that the values must be integers

NP-complete problems

3D matching

Bipartite matching: given two sets of things and pair constraints, find a matching between the sets

3D matching: given three sets of things and triplet constraints, find a matching between the sets

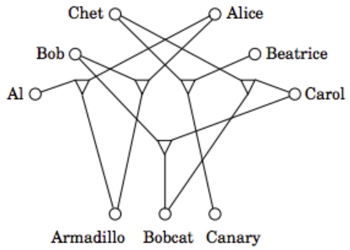
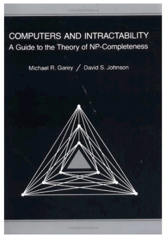


Figure from Dasgupta et. al 2008

P vs. NP

Polynomial time solutions exist	NP-complete (and no polynomial time solution currently exists)
Shortest path	Longest path
Bipartite matching	3D matching
Linear programming	Integer linear programming
Minimum cut	Balanced cut
…	…



I can't find an efficient algorithm, I guess I'm just too dumb.

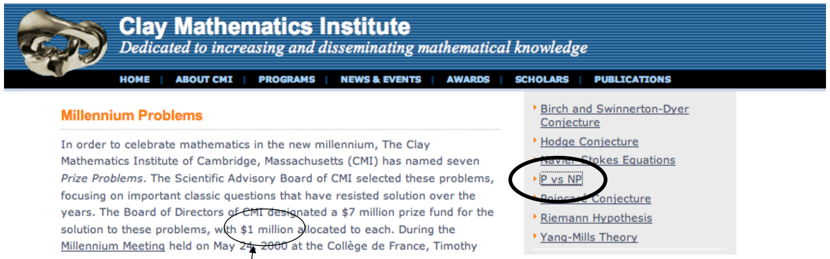
From Garey and Johnson, "Computers and Intractability: A Guide to the Theory of NP-completeness"

I can’t find an efficient algorithm, because no such algorithm is possible!

From Garey and Johnson, “Computers and Intractability: A Guide to the Theory of NP-completeness”

I can’t find an efficient algorithm, but neither can all these famous people!

From Garey and Johnson, “Computers and Intractability: A Guide to the Theory of NP-completeness”



Clay Mathematics Institute

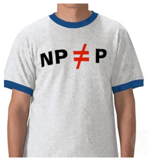
Dedicated to increasing and disseminating mathematical knowledge

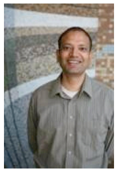
Millennium Problems

In order to celebrate mathematics in the new millennium, The Clay Mathematics Institute of Cambridge, Massachusetts (CMI) has named seven *Prize Problems*. The Scientific Advisory Board of CMI selected these problems, focusing on important classic questions that have resisted solution over the years. The Board of Directors of CMI designated a $7 million prize fund for the solution to these problems, with $1 million allocated to each. During the Millennium Meeting held on May 24, 2000 at the Collège de France, Timothy

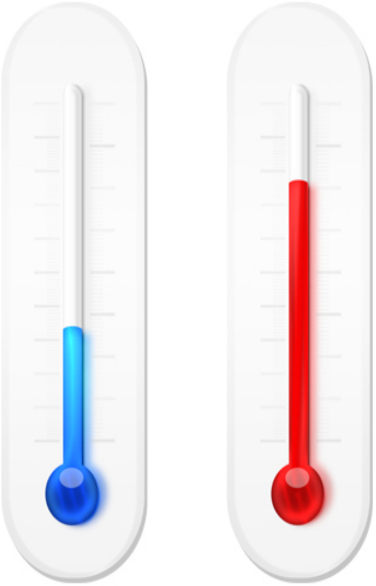
- Birch and Swinnerton-Dyer Conjecture
- Hodge Conjecture
- Navier-Stokes Equations
- P vs NP
- Poincaré Conjecture
- Riemann Hypothesis
- Yang-Mills Theory

$1 million





Vinay Deolalikar



Proving NP-completeness

A problem is *NP-complete* if:

1. it can be verified in polynomial time (i.e. in NP)
2. *any* NP-complete problem can be reduced to the problem in polynomial time (is NP-hard)

Ideas?

Proving NP-completeness

Given a problem NEW to show it is NP-Complete

1. Show that NEW is in NP
   a. Provide a verifier
   b. Show that the verifier runs in polynomial time
2. Show that all NP-complete problems are reducible to NEW in polynomial time
   a. Describe a reduction function *f* from a known NP-Complete problem to NEW
   b. Show that *f* runs in polynomial time
   c. Show that a solution exists to the NP-Complete problem IFF a solution exists *to the NEW problem generated by f*

Proving NP-completeness

Show that a solution exists to the NP-Complete problem IFF a solution exists *to the NEW problem generate by f*

- Assume we have an NP-Complete problem instance that has a solution, show that the NEW problem instance generated by *f* has a solution
- Assume we have a problem instance of NEW *generated by f* that has a solution, show that we can derive a solution to the NP-Complete problem instance

Other ways of proving the IFF, but this is often the easiest

Notice that just because…

$\text{FOO} \leq_p \text{BAR}$

does *not* mean that…

$\text{BAR} \leq_p \text{FOO}$

Proving NP-completeness

Show that all NP-complete problems are reducible to NEW in polynomial time

Why is it sufficient to show that one NP-complete problem reduces to the NEW problem?

Proving NP-completeness

Show that all NP-complete problems are reducible to NEW in polynomial time

All others can be reduced to NEW by first reducing to the one problem, then reducing to NEW. Two polynomial time reductions is still polynomial time!

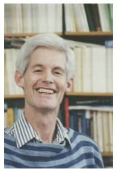
Proving NP-completeness

Show that all NP-complete problems are reducible to NEW in polynomial time

Show that *any* NP-complete problem is reducible to NEW in polynomial time

BE CAREFUL!

~~Show that NEW is reducible to any NP-complete problem in polynomial time~~

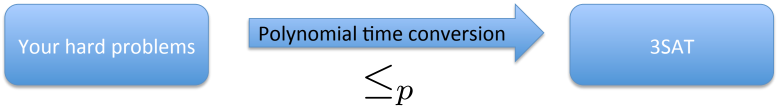


Stephen Cook

The Cook-Levin Theorem (1972)

Leonid Levin

"Every decision problem in NP can be (quickly) converted into a corresponding 3SAT decision problem" (i.e., 3SAT is "NP-complete")



Why 3SAT? Why not 2SAT or something completely different?

If 3SAT could be solved in poly time then…

NP-complete: 3-SAT

A boolean formula is in *n-conjunctive normal form* (*n*-CNF) if:

- it is expressed as an AND of clauses
- where each clause is an OR of no more than *n* variables

$$(a \vee \neg a \vee \neg b) \wedge (c \vee b \vee d) \wedge (\neg a \vee \neg c \vee \neg d)$$

3-SAT: Given a 3-CNF boolean formula, is it satisfiable?

3-SAT is an NP-complete problem

NP-Complete problems

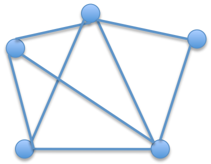
Why do we care about showing that a problem is NP-Complete?

- We know that the problem is hard (and we probably won't find a polynomial time exact solver)
- We may need to compromise:
  - reformulate the problem
  - settle for an approximate solution
- Down the road, if a solution is found for an NP-complete problem, then we'd have one too…

CLIQUE

A *clique* in an undirected graph G = (V, E) is a subset V' ⊆ V of vertices that are fully connected, i.e. every vertex in V' is connected to every other vertex in V'

CLIQUE problem: Does G contain a clique of size k?

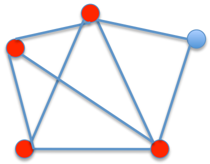


Is there a clique of size 4 in this graph?

CLIQUE

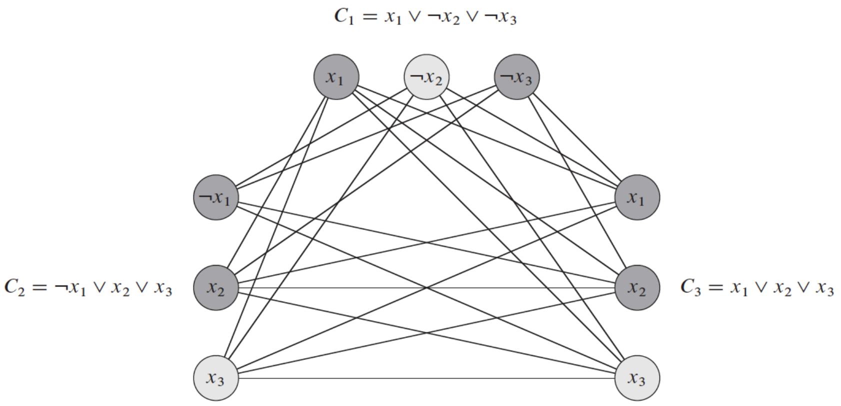
A *clique* in an undirected graph G = (V, E) is a subset V' ⊆ V of vertices that are fully connected, i.e. every vertex in V' is connected to every other vertex in V'

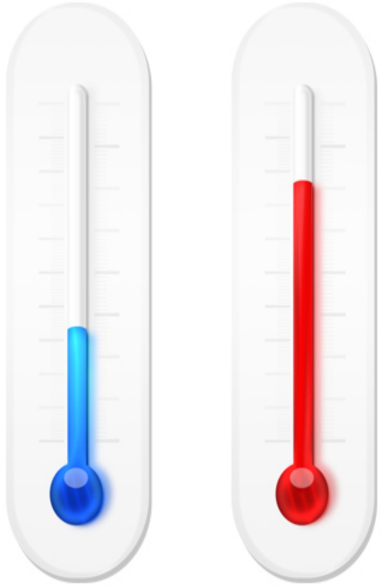
CLIQUE problem: Does G contain a clique of size k?



CLIQUE is an NP-Complete problem

The Clique Problem is NP-Complete





NP-complete: SAT

Given a boolean formula of *n* boolean variables joined by *m* connectives (AND, OR or NOT) is there a setting of the variables such that the boolean formula evaluate to true?

$$(a \wedge b) \vee (\neg a \wedge \neg b)$$

$$((\neg(b \vee \neg c) \wedge a) \vee (a \wedge b \wedge c)) \wedge c \wedge \neg b$$

Is SAT an NP-complete problem?

NP-complete: SAT

Given a boolean formula of *n* boolean variables joined by *m* connectives (AND, OR or NOT) is there a setting of the variables such that the boolean formula evaluate to true?

$$((\neg(b \vee \neg c) \wedge a) \vee (a \wedge b \wedge c)) \wedge c \wedge \neg b$$

1. Show that SAT is in NP
   a. Provide a verifier
   b. Show that the verifier runs in polynomial time
2. Show that all NP-complete problems are reducible to SAT in polynomial time
   a. Describe a reduction function *f* from a known NP-Complete problem to SAT
   b. Show that *f* runs in polynomial time
   c. Show that a solution exists to the NP-Complete problem IFF a solution exists *to the SAT problem generate by f*

NP-Complete: SAT

1. Show that SAT is in NP
   a. Provide a verifier
   b. Show that the verifier runs in polynomial time

Verifier: A solution consists of an assignment of the variables

- If clause is a single variable:
  - return the value of the variable
- otherwise
  - for each clause:
    - call the verifier recursively
    - compute a running solution

polynomial run-time?

NP-Complete: SAT

Verifier: A solution consists of an assignment of the variables

- If clause is a single variable:
  - return the value of the variable
- otherwise
  - for each clause:
    - call the verifier recursively
    - compute a running solution

linear time

- at most a linear number of recursive calls (each call makes the problem smaller and no overlap)
- overall polynomial time

NP-Complete: SAT

1.
2. Show that all NP-complete problems are reducible to SAT in polynomial time
   a. Describe a reduction function *f* from a known NP-Complete problem to SAT
   b. Show that *f* runs in polynomial time
   c. Show that a solution exists to the NP-Complete problem IFF a solution exists *to the SAT problem generate by f*

Reduce 3-SAT to SAT:

- Given an instance of 3-SAT, turn it into an instance of SAT

Reduction function:

- DONE ☺

- Runs in constant time! (or linear if you have to copy the problem)

NP-Complete: SAT

Show that a solution exists to the NP-Complete problem IFF a solution exists *to the NEW problem generate by f*

- Assume we have an NP-Complete problem instance that has a solution, show that the NEW problem instance generated by *f* has a solution
- Assume we have a problem instance of NEW *generated by f* that has a solution, show that we can derive a solution to the NP-Complete problem instance

---

- Assume we have a 3-SAT problem with a solution:
  - Because 3-SAT problems are a subset of SAT problems, then the SAT problem will also have a solution
- Assume we have a problem instance generated by our reduction with a solution:
  - Our reduction function simply does a copy, so it is already a 3-SAT problem
  - Therefore the variable assignment found by our SAT-solver will also be a solution to the original 3-SAT problem