Slides on **Theorems 1.2**, **1.4**, **1.7**, and **1.14** of *The Complexity Theory Companion* by Hemaspaandra and Ogihara

Slides by Group 1:

Jacob Balazer

Justin Moore

Lior Privman

Leila Seghatoleslami

Arrvindh Shriraman

Wenzhao Tan

Jumping right into the thick of things…[1]

Theorem 1.2

$$(\exists T \,.\, T \text{ is a tally set} \wedge T \text{ is } \mathcal{NP}\text{-hard}) \Rightarrow \mathcal{P} = \mathcal{NP}$$

Corollary 1.3

$$(\exists T \,.\, T \text{ is a tally set} \wedge T \text{ is } \mathcal{NP}\text{-complete}) \Leftrightarrow \mathcal{P} = \mathcal{NP}$$

Basic strategy for proving Theorem 1.2

(1) Assume $\exists T \,.\, T$ is a tally set $\wedge$ T is $\mathcal{NP}$-hard

(2) Construct a *deterministic* poly-time algorithm for some $\mathcal{NP}$-complete language.

[1] These slides contain many unattributed quotes from *The Complexity Theory Companion* by Hemaspaandra and Ogihara.

If using SAT was made illegal, then only criminals would use SAT…

SAT

SAT = $\{f \mid f$ is a satisfiable boolean formula$\}$

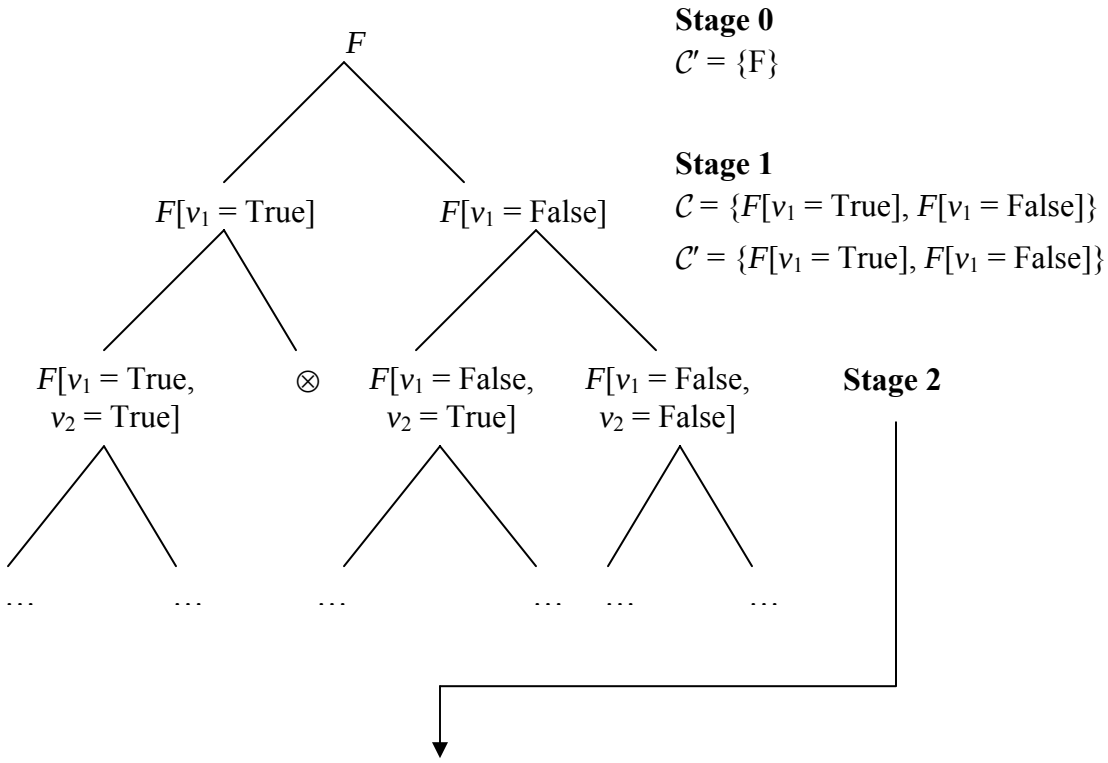
Examples of SAT

$v_1 \vee v_2 \vee v_3$ satisfiable with assignment:
[v_1 = True, v_2 = False, v_3 = False]

$v_1 \wedge \overline{v_1}$ unsatisfiable

w.l.o.g., f contains variables $v_1 \ldots v_m$, $m \geq 1$

Example Execution of the Algorithm…



$\mathcal{C}$ = {$F[v_1$ = True, v_2 = True], $F[v_1$ = True, v_2 = False],
$F[v_1$ = False, v_2 = True], $F[v_1$ = False, v_2 = False]}

$\mathcal{C}'$ = {$F[v_1$ = True, v_2 = True],
$F[v_1$ = False, v_2 = True], $F[v_1$ = False, v_2 = False]}

SAT trees grow too fast, so to prove Theorem 1.2, we will use pruning…

The Algorithm

Stage 0: $\mathcal{C}' \leftarrow \{F\}$

Stage *i*: $1 \le i \le m$, given that $\mathcal{C}'$ at the end of Stage $i-1$ is the collection of formulas: $\{F_1, \ldots, F_\ell\}$.

Step 1 Let $\mathcal{C}$ be the collection

$\{F_1[v_i = \text{True}], F_2[v_i = \text{True}], \ldots, F_\ell[v_i = \text{True}],$

$F_1[v_i = \text{False}], F_2[v_i = \text{False}], \ldots, F_\ell[v_i = \text{False}]\}$

Step 2 $\mathcal{C}' \leftarrow \varnothing$

Step 3 For each formula f in $\mathcal{C}$

If $g(f) \in 1^*$ and for no formula $h \in \mathcal{C}'$ does $g(f) = g(h)$

then add f to $\mathcal{C}'$

Stage $m+1$: return "yes", F is satisfiable, if some (variable-free) formula $f \in \mathcal{C}'$ is satisfiable, otherwise return "no".

"I find your lack of faith disturbing." –Darth Vader

The Proof of Theorem 1.2

Lemma 1 The algorithm returns "yes" $\Leftrightarrow$ input formula $F \in$ SAT

After **Stage 0**, $\mathcal{C}'$ contains a satisfiable formula $\Leftrightarrow$ input formula $F \in$ SAT.

After **Stage *i*, Step 1**, $\mathcal{C}$ contains a satisfiable formula $\Leftrightarrow$ $\mathcal{C}'$ contains a satisfiable formula, by the self-reducibility of SAT.

After **Stage *i*, Step 3**, each formula f from **Step 1** is kept unless either:

$g(f) \notin 1^*$

g many-one reduces SAT to T, so:

$g(f) \notin 1^* \Rightarrow g(f) \notin T \Rightarrow f \notin$ SAT

$g(f) \in 1^*$, but some $h \in C'$ has $g(f) = g(h)$

$[(f \in \text{SAT} \Leftrightarrow g(f) \in T)$
$\wedge (h \in \text{SAT} \Leftrightarrow g(h) \in T)$
$\wedge g(f) = g(h)]$
$\Rightarrow f \in \text{SAT} \Leftrightarrow g \in \text{SAT}$

…Proof Continued

Lemma 2 The algorithm runs in deterministic poly-time

THE COOL PART!

Let $p = |F|$ be the number of bits in the representation of F.

In Step 3, we are calling g on formulas of various lengths

–each of these formulas has length $\leq p$

–g runs for at most $p^k + k$ steps for some k

–g will never output a string of length $> p^k + k$

If $\mathcal{C}'$ contains $p^k + k + 1 + x$ formulas that under the action of g produce elements of 1^*, then by the pigeonhole principle, the $g(f) = g(h)$ test will eliminate at least x of those formulas.

Proof **of Theorem 1.2**

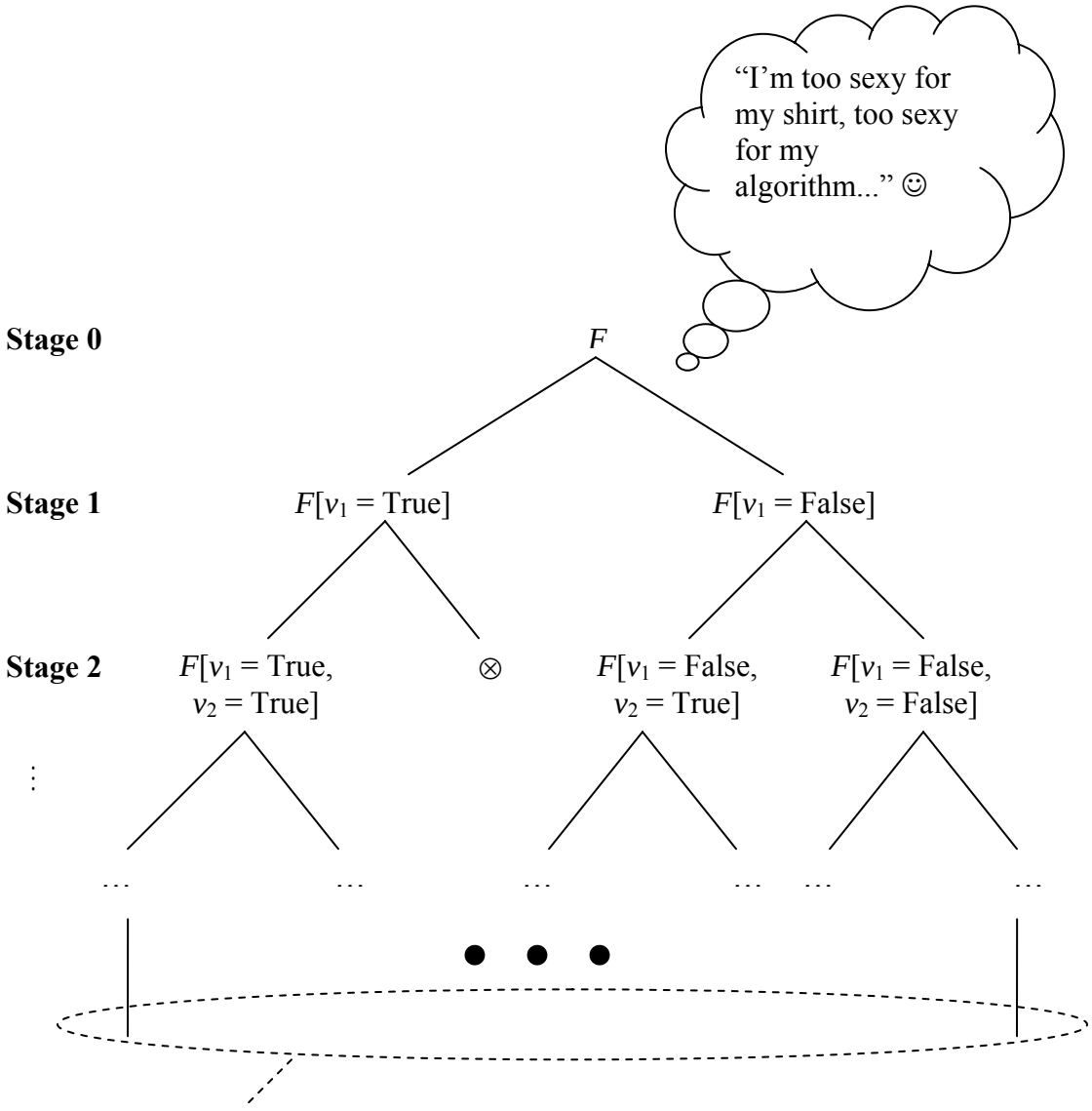
($\exists T$. T is a tally set $\wedge$ T is $\mathcal{NP}$-hard)

$\Rightarrow$ there is a deterministic poly-time algorithm for SAT
(by *Lemma 1* and *Lemma 2*)

$\Rightarrow \mathcal{P} = \mathcal{NP}$

Example Execution of the Algorithm…

…using the $h(f) = h(g)$ test in **Part 3**.



(note: this is the max width after pruning)

Again! Again!

Theorem 1.4

$(\exists S$. S is a sparse set $\wedge$ S is co$\mathcal{NP}$-hard$) \Rightarrow \mathcal{P} = \mathcal{NP}$

Basic strategy for proving Theorem 1.4

(1) Assume $\exists S$. S is a sparse set $\wedge$ S is co$\mathcal{NP}$-hard

(2) Construct a *deterministic* poly-time algorithm for some $\mathcal{NP}$-complete language.

Definition For any ℓ, let $p_\ell(x) = x^\ell + \ell$

We know by the definition of g, that

$(\exists k)(\forall x)[|g(x)| \leq p_k(x)]$

since $g(x)$ runs in poly-time, its output lengths are polynomially bounded

We also know by the definition of spare sets, that

$(\exists d)(\forall x)[\|S^{\leq x}\| \leq p_d(x)]$

i.e., number of strings in S of length x or less is polynomially bounded

SAT trees grow too fast, so to prove Theorem 1.2, we will use pruning…

The Algorithm

Stage 0: $\mathcal{C}' \leftarrow \{F\}$

Stage *i*: $1 \le i \le$ m, given that $\mathcal{C}'$ at the end of Stage $i-1$ is the collection of formulas: $\{F_1, \ldots, F_\ell\}$.

Step 1 Let $\mathcal{C}$ be the collection

$\{F_1[v_i = \text{True}], F_2[v_i = \text{True}], \ldots, F_\ell[v_i = \text{True}],$

$F_1[v_i = \text{False}], F_2[v_i = \text{False}], \ldots, F_\ell[v_i = \text{False}]\}$

Step 2 $\mathcal{C}' \leftarrow \varnothing$

Step 3 For each formula f in $\mathcal{C}$

If for no formula $h \in \mathcal{C}'$ does $g(f) = g(h)$

then add f to $\mathcal{C}'$

Step 4 If $\mathcal{C}'$ contains at least $p_d(p_k(|\text{F}|))+1$ elements, return "yes"

Stage m + 1: return "yes", F is satisfiable, if some (variable-free) formula $f \in \mathcal{C}'$ is satisfiable, otherwise return "no".

Are we there yet?

The Proof of Theorem 1.4

Lemma 3 The algorithm returns "yes" $\Leftrightarrow$ input formula $F \in$ SAT

The only difference from *Lemma 1* which we need to consider is the addition of **Step 4**…

THE OTHER COOL PART!

Let n represent $|F|$.

For any formula H in the algorithm, $|g(H)| \leq |g(F)|$

$|g(F)| \leq p_k(n)$

How many strings of length $p_k(n)$ or less in S?

$\left\| S^{p_k(n)} \right\| \leq p_d(p_k(n))$

By the pigeonhole principle,

If $\mathcal{C}'$ contains at least $p_d(p_k(n))+1$

$\Rightarrow$ some $g(h) \notin S \Rightarrow h \notin \overline{SAT}$

Lemma 4 The algorithm runs in deterministic poly-time

Clearly the size of $\mathcal{C}$' is always bounded by the polynomial $p_d(p_k(n))$

Theorem 1.4 follows from *Lemma 3* and *Lemma 4*.

Mahaney's Theorem

a.k.a. Hem/Ogi Theorem 1.7
a.k.a. Bov/Cre Theorem 5.7

If a sparse, NP-Complete language exists =>

P = NP

Definitions

Let S be a sparse NP-Complete language

Define $p_\ell(n) = n^\ell + \ell$

We know that $SAT \leq_m^p S$ since S is NP-Complete

The function that reduces, σ, is bounded by p_a

Define $C(n) = |S^{\leq n}|$ and $C_a(n) = |S^{\leq p_a(n)}|$

Since S is sparse, C(n) is bounded by p_d

What did the sparse set say to its complement?
"Why do you have to be so dense?"

What we would want to happen, or Why this proof isn't really easy

What if $\overline{S}$ were in NP?

Since S is NP-Complete, $\overline{S} \leq_m^p S$

Since many-one reductions are closed under complementation, $S \leq_m^p \overline{S}$

Thus, $\overline{S}$ is NP-Complete, S is co-NP-Complete and Hem/Ogi theorem 1.4 shows that P=NP.

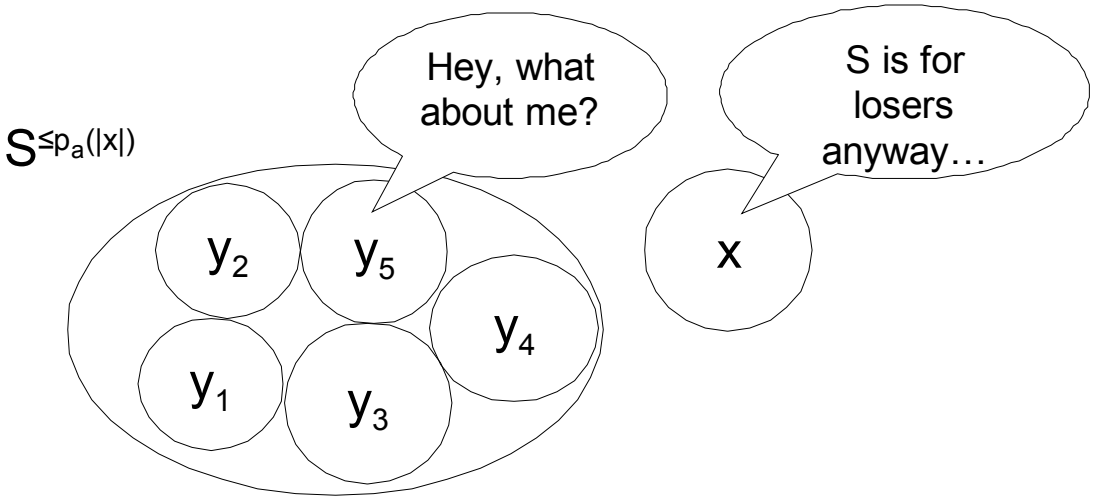
If only the proof were as easy as putting many-one reductions into a presentation…

Sorry, not quite so easy…

However, $\overline{S}$ is not necessarily in NP

Let's define $\overline{S}$ in terms of $C_a(n)$:

$$\overline{S}=\{x \mid \exists y_1, y_2,\ldots,y_{C_a(|x|)} [[(|y_1|\leq p_a(|x|) \wedge y_1\neq x \wedge y_1\in S]$$
$$\wedge [(|y_2|\leq p_a(|x|) \wedge y_2\neq x \wedge y_2\in S]$$
$$\wedge \ldots \ldots \ldots$$
$$\wedge [(|y_{C_a(|x|)}|\leq p_a(|x|) \wedge y_{C_a(|x|)}\neq x \wedge y_{C_a(|x|)}\in S]$$
$$\wedge \text{ all the y's are distinct }] \}$$



If only we had a way to have $\overline{S}$ be an NP language…

Unfortunately, we cannot find the value of $C_a(|x|)$

Fix this by parameterizing the number of y's:

$$\hat{S}=\{\langle x,m\rangle \mid \exists y_1, y_2,\ldots,y_m\, [\, [(|y_1|\leq p_a(|x|) \wedge y_1\neq x \wedge y_1\in S]$$
$$\wedge\, [(|y_2|\leq p_a(|x|) \wedge y_2\neq x \wedge y_2\in S]$$
$$\wedge \ldots\ldots\ldots$$
$$\wedge\, [(|y_m|\leq p_a(|x|) \wedge y_m\neq x \wedge y_m\in S]$$
$$\wedge \text{ all the y's are distinct }]\,\}$$

We will call this the pseudo-complement of S

Note that for any $\langle x,m\rangle$, $\langle x,m\rangle\in \hat{S}$ iff:

a) $m < C_a(|x|)$ or

b) $m = C_a(|x|)$ and $x\notin S$

How can this pseudo-complement help?

We can prove that $\hat{S}$ is in NP by constructing an algorithm that decides $\hat{S}$ in non-deterministic polynomial time.

Here's a modified version of Bov-Cre's algorithm:

```
begin {input: x, m}
   if m > p_d(p_a(|x|)) then reject;
   guess y_1, y_2, …, y_m in set of m-tuples of
      distinct words, each of which is of
      length, at most, p_a(|x|);
   for i = 1 to m do
      if y_i = x then reject;
   simulate M_S(y_i) along all Ms's paths starting
       at i = 1
      if M_s(y_i) is going to accept and i < m
         simulate M_s(y_{i+1}) along all M_s's paths;
      if M_s(y_i) is going to accept and i = m
         accept along that path;
   accept;
end.
```

Since $\hat{S}$ is in NP and S is NP-Complete,

$\hat{S} \leq_m^p S$ by some function ψ with bound p_g

Why is it called recap?
We never capped anything in the first place…

capitulate
\Ca*pit"u*late\, v. t. To surrender or transfer, as an army or a fortress, on certain conditions. [R.]

So far, we've figured out the following:

a) $\hat{S}$ many-one poly-time reduces to S by ψ with time bound p_g

b) SAT many-one poly-time reduces to S by σ with time bound p_a

c) The sparseness of S, C(n), is assured by p_d

d) Bov-Cre is way too algorithmic

e) It is probably going to snow today

--Hey, we all chose Rochester for some reason

Next:

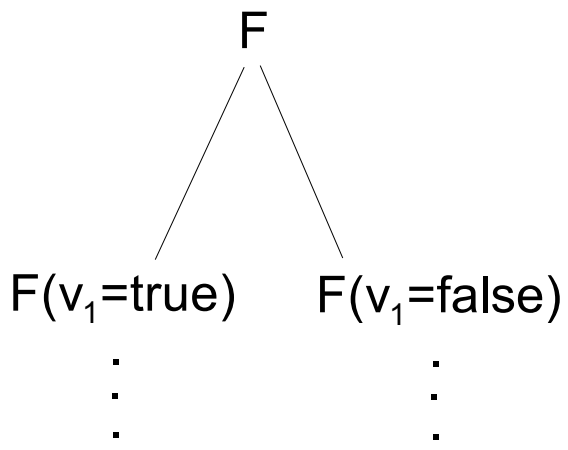
What's our favorite way to show P=NP?

What's our favorite way to show that SAT can be decided in polynomial time?

Get out the hedge trimmers…

We have some formula F

We want to know if it's in SAT



Look familiar?

This tree will get way too bushy for our purposes though, so we need to come up with a way to prune it

What's this? A polynomial number of hedge trimmers?
Only a theorist would think of something like that

Given a formula, for each m in [1, $p_d(p_a(|F|))$]

(this is every possible value of m for F)

Create and prune a tree of assignments to variables just as we did for theorem 1.4 using a new pruning algorithm. When we get to the end, check each assignment to see if it's satisfiable.

What we want to happen:

b) The number of leaves to be bounded by a polynomial

c) The pruning algorithm to be polynomial time

d) If F is satisfiable, then one of the leaves of the tree at the end is satisfiable

e) The snow to wait at least another 3-4 weeks so it wont instantly turn into slush and then ice

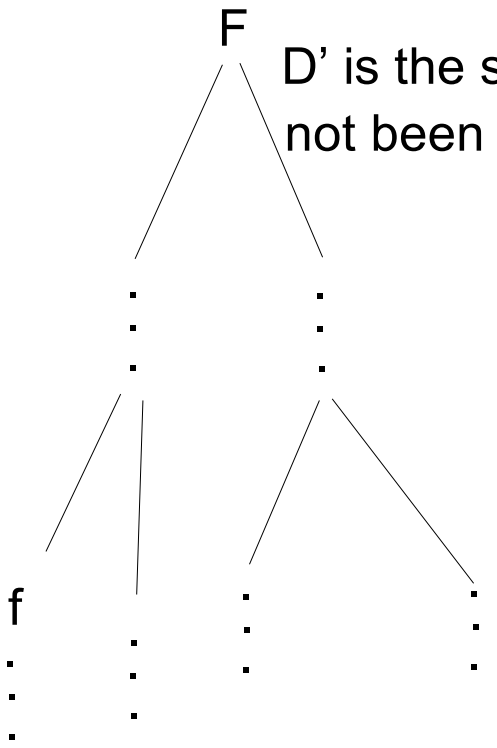
What that will get us:

b) A polynomial time algorithm that decides SAT

c) More time to put off getting snow tires for our cars

This slide is a great example of why I am not a digital art major

For each stage of the tree:
D is the set of all formulas generated by assigning true and false to the previous stage's result
D' is the set of all formulas that have not been pruned from D (i.e. D' $\subseteq$ D)



How do we get to D' from D?

```
for each f in D
   if |D'| ≤ p_d(p_g(p_a(|F|))) and
      for each f' in D'
         ψ(<σ(f), m>) ≠ ψ(<σ(f'), m>)
   then
      add f to D'
```

When we're done:

Check each (variable-free) formula in the bottom layer to see if it's satisfiable

There are only a polynomial number

If any is satisfiable, we're done

If for all m's, no formula in the bottom layer is satisfiable, F is not satisfiable

What's next?

Mappings…

A few comparisons…

Some polynomial bounds…

Tree pruning…

P=NP

Wait… I don't get it…

How is it so hard to draw nice trees when you are using presentation software with the "snap-to-grid" feature?

Demystification (why the pruning works):

It is important to note that when we have found the correct $m = C_a(p_a(|F|))$ that

f is not satisfiable iff $\psi(<\sigma(f), C_a(p_a(|F|))>) \in S$

Why, you ask?

Recall that SAT reduces to S

This $f \notin SAT$ iff $\sigma(f) \notin S$

Remember $\hat{S}$?

$m=C_a(p_a(|F|))$ and $\sigma(f) \notin S$ iff $<\sigma(f), C_a(p_a(|F|))> \in \hat{S}$

But $\hat{S}$ reduces to S too!

$<\sigma(f), C_a(p_a(|F|))> \in \hat{S}$ iff $\psi(<\sigma(f), C_a(p_a(|F|))>) \in S$

If $p_a(p_q(p_r(p_l(m+C_n(x))))) = p_j(p_n(p^4(p_a(n_m - |1|))))$, then 2 = 3

At least something is obvious in these slides…

How does this help?

There are a bounded number of unsatisfiable formulas that are mapped in S.

This is p_d (the sparsity of S) composed with p_g (the limit on mappings to S through ψ) composed with p_a (the limit on mappings to S through σ)*

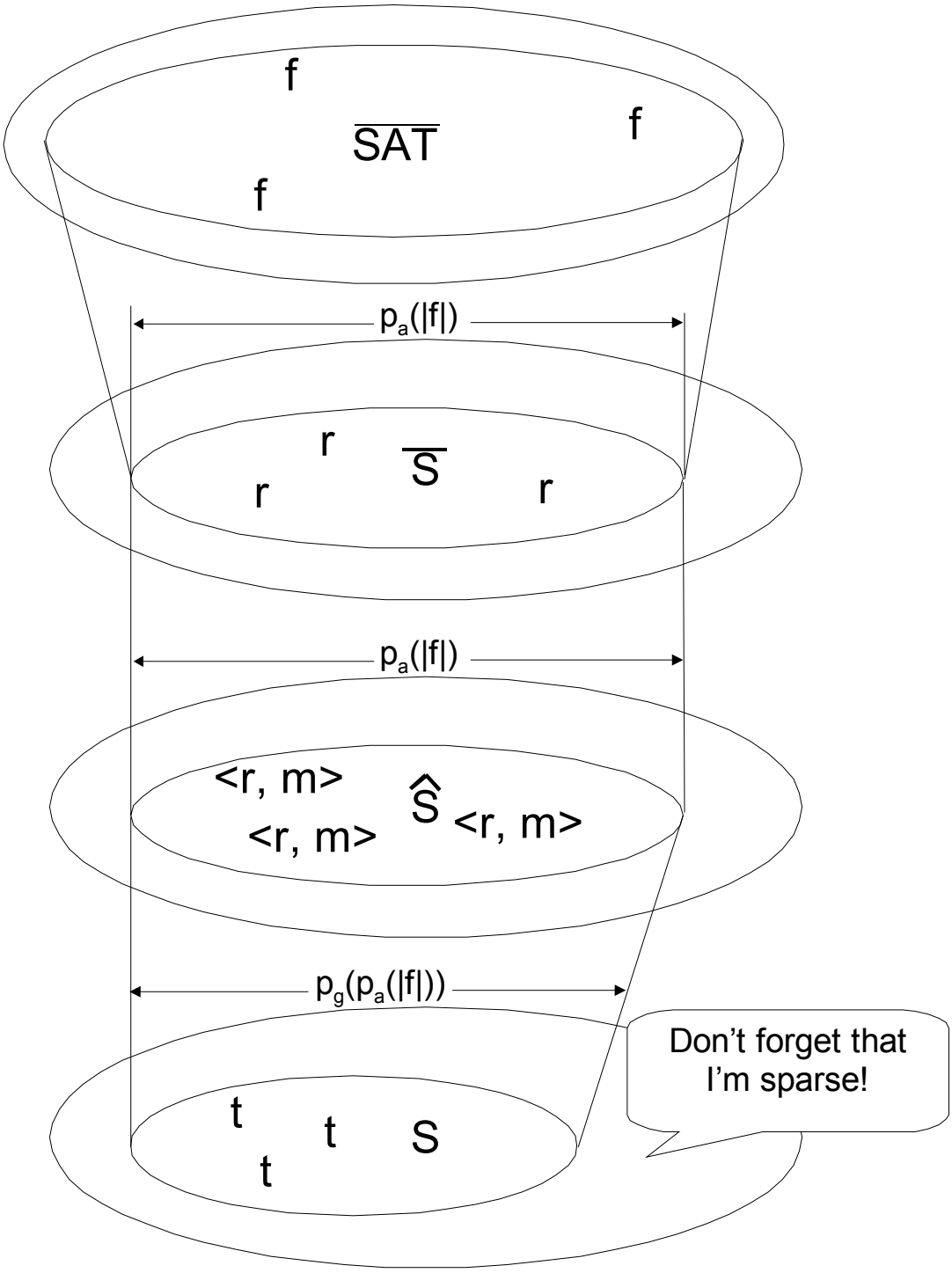
If we have chosen $m = C_a(p_a(|F|))$, and we have found more than $p_d(p_g(p_a(|F|)))$ values then:

Not all those $\psi(<\sigma(f), m>)$'s are in S so at least one of the f's is satisfiable

Thus, we can happily prune away all but one over the bound of these values, leaving a polynomial number while still guaranteeing one of them is sure to have a satisfying assignment.

*Since m is constant for each tree, pairing $\sigma(f)$ with m will not make the number of possible mappings in $\hat{S}$ bigger. Thus we don't need to worry about the pairing in $\hat{S}$ changing the bound.

This complicated diagram makes it much easier to see. Trust me.



Wait, if I prove P=NP, I win a million dollars…

In the universe that has a sparse NP-Complete set, I am rich!

Most of you are saying right now:

"Yes, that is true, but how do you know if you have an $m = C_a(p_a(|F|))$"

An interesting fact:

There are a polynomial number of m's.

Does it really matter what happens to the tree with $m \neq C_a(p_a(|F|))$?

As long as we're not wasting too much time pruning trees the wrong way, the other m's don't create too much overhead.

If F is not satisfiable, we'll never get a satisfying assignment; if F is satisfiable, maybe we'll randomly keep an assignment with $m \neq C_a(p_a(|F|))$ but when $m = C_a(p_a(|F|))$ each stage is guaranteed to have at least one satisfiable formula.

Wait… did we just do what I think we did?

Since for some value m, there is a tree that outputs a satisfiable formula iff the formula is satisfiable

There are at most a polynomial number of leaves

The pruning function runs in a polynomial amount of time

There are only a polynomial number of trees

We just decided if a formula is satisfiable in a polynomial amount of time

Thus an NP-Complete language is decidable by a deterministic polynomial algorithm and P = NP

…now what?

Theorem 1.14 (Hemaspaandra and Ogihara):

If there exists a sparse NP $\leq_T^p$ -complete set, then $NP^{NP} = P^{NP[O(\log n)]}$

Recall that $NP^{NP} = \sum_2^p$.

$P^{NP[O(\log n)]}$ is the class of languages recognizable by some deterministic polynomial-time machine that may make up to $O(\log n)$ queries to an NP oracle, where n is the length of the input.

Proof Outline:

1. Assume the existence of a sparse NP $\leq_T^p$ -complete set S.
2. Use this to show that an arbitrary NP^{NP} problem can be solved with a $P^{NP[O(\log n)]}$ machine.

Proof Part 1: Define an NP^{NP} language in terms of a sparse NP $\leq_T^p$ -complete set:

Let S be a sparse NP $\leq_T^p$ -complete set.

Because S is NP Turing-complete, all NP languages Turing reduce to S. Let M be a deterministic polynomial-time machine that solves SAT using S.

$$SAT = L(M^S)$$

Because M is a deterministic polynomial-time machine, its execution time is bounded by a polynomial function: for input of length n,

$p_k(n)$ for some k,
where we define $p_k(n) = n^k + k$

This effectively places an upper bound on the length of strings that M will ever query oracle S with, since M's execution time is bounded, and M can write at most one symbol to its oracle tape per state transition.

Let L be an arbitrary language in $\mathrm{NP}^{\mathrm{NP}}$.

This means that L is recognizable by some nondeterministic polynomial-time machine N which uses SAT as an oracle (since SAT is NP-complete).

$$L = \mathrm{L}(N^{\mathrm{SAT}})$$

Substituting our earlier solution that $\mathrm{SAT} = \mathrm{L}(M^S)$

$$L = \mathrm{L}(N^{\mathrm{L}(M^S)})$$

Since N is a polynomial-time nondeterministic machine, its execution will be bounded by a polynomial function: for input of length n,

$p_\ell(n)$ for some ℓ

Note that this effectively places an upper limit on the length of a string that N can query its SAT oracle with, since it can write at most one symbol to its oracle tape per state transition.

For $L = \mathrm{L}(N^{\mathrm{L}(M^S)})$, since N's queries to its SAT oracle are limited to length $p_\ell(|y|)$ for input y, here M can query S for strings of length at most $p_k(p_\ell(|y|))$. A solution to L will only ever need to query S with strings of length $\leq p_k(p_\ell(|y|))$ for input y. That is, only a subset of S need be considered for each query y:

$S^{\leq n}$, where $n = p_k(p_\ell(|y|))$

Because S is sparse, the number of strings that will be in this subset is bounded by a polynomial function of $|y|$.

How can we solve L with less than an NP^{NP} machine?

Observing that $L = L(N^{L(M^S)})$, normally we would expect that this language could only be recognized by an NP^{NP} machine. We will exploit the fact that for each string y for which we want to determine membership in L, oracle queries to S are only required for a subset of S that has size polynomial in $|y|$.

If the elements of $S^{\leq n}$ can somehow be enumerated, then oracle queries to S can be simulated by a deterministic polynomial-time subroutine.

*** *If* we can know the exact number of elements in $S^{\leq n}$, then we can in nondeterministic polynomial time enumerate all the elements in $S^{\leq n}$. ***

Define *V*: (this is the NP part of our $P^{NP[O(\log n)]}$ solution to *L*

$$V = \{ 0\#1^n\#1^q \mid ||S^{\leq n}|| \geq q \} \cup \{ 1\#x\#1^n\#1^q \mid (\exists Z \subseteq S^{\leq n})[||Z|| = q \wedge x \in \mathrm{L}(N^{\mathrm{L}(M^Z)})]\}$$

The P part of our solution is a deterministic polynomial-time algorithm that will make O(log *n*) oracle queries to *V*.

The first set in *V* is a mechanism by which we can determine $||S^{\leq n}||$ for any *n*. Note that for $||S^{\leq n}||=r$, the string $0\#1^n\#1^z$ will be in *V* for all $z \geq r$, and not for any $z < r$.

The second set in *V* is a mechanism that lets us test a string *x* for membership in *L*, but only if we tell the machine that accepts *V* what $||S^{\leq n}||$ is for a given *n* by setting *q* to $||S^{\leq n}||$. Observe that if $Z \subseteq S^{\leq n}$ and $||Z|| = q = ||S^{\leq n}||$, then $Z = S^{\leq n}$.

Algorithm: (this is the P part of our $P^{NP[O(\log n)]}$ solution)

1. For input y calculate n as $p_k(p_\ell(|y|))$.
2. Repeatedly query V with strings in the form $0\#1^n\#1^z$, varying z in a binary search fashion until the exact value of $||S^{\leq n}||$ is found. Call that value r. Because S is sparse, $||S^{\leq n}||$ is bounded by a polynomial function (remembering that n itself is also bounded by $p_k(p_\ell(|y|))$), and so the binary search will complete in $O(\log|y|)$ time.
3. Query V with a string in the form $1\#y\#1^n\#1^r$, and accept only if V returns 'yes'.

How can *V* be calculated in nondeterministic polynomial time?

V is the union of two sets, both of which we can show to be NP separately:

NP algorithm for $\{ 0\#1^n\#1^q \mid ||S^{\leq n}|| \geq q \}$:

Algorithm idea: Find a size q subset of $S^{\leq n}$. If one exists, then $||S^{\leq n}|| \geq q$.

1. If input is not in the form $0\#1^n\#1^q$, reject.
2. Nondeterministically guess a subset of $(\Sigma^*)^{\leq n}$ with size q.
3. Sequentially test each element in the subset for membership in *S*: simulate the machine for *S* on each element in sequence. If the current path of the simulation of *S* accepts, continue. If the current path rejects, reject. (Since *S* is NP and there are only q elements that need to be tested, the time required to test all the elements is polynomial in $q*n$.)

NP algorithm for

$\{ 1\#x\#1^n\#1^q \mid (\exists Z \subseteq S^{\leq n})[||Z|| = q \ \wedge \ x \in L(N^{L(M^Z)})]\}$:

Algorithm summary: enumerate the elements in Z. Once you have them, oracle calls to Z can be simulated by a deterministic polynomial-time subroutine that compares the query string against the elements of Z. Simulate N, and use polynomial deterministic subroutines to simulate M and Z.

1. If input is not in the form $1\#x\#1^n\#1^q$, reject.
2. Nondeterministically guess a size q subset of $(\Sigma^*)^{\leq n}$, call this Z.
3. Sequentially test each element in Z for membership in S. If the current path for the simulation of the machine for S rejects, reject; otherwise continue on to the next element.
4. Test whether $x \in L(N^{L(M^Z)})$: Simulate N on input x. Oracle calls to $L(M^Z)$ can be simulated by a deterministic polynomial-time subroutine that tests the query string against every element in our previously enumerated set Z.

Further results from $NP^{NP} = P^{NP[O(\log n)]}$

(equivalently $\sum_2^p = P^{NP[O(\log n)]}$)

$P^{NP[O(\log n)]}$ is closed under complementation, which implies that $\sum_2^p$ is also closed under complementation, i.e. $\sum_2^p = \text{co}\sum_2^p$ or $\sum_2^p = \prod_2^p$, which implies that PH = $\sum_2^p$. (Recall that PH is the polynomial hierarchy – the union of $\sum_i^p$ for all *i*.)