Group A Lecture 1: The Self-Reducibility Technique

Adam Scrivener, Haofu Liao, Nabil Hossain, Shupeng Gui, Thomas Lindstorm-Vautrin

Department of Computer Science
University of Rochester

November 2, 2015
Table of Contents

1 Tree Pruning Technique
   ■ Theorem 1.2
   ■ Theorem 1.4
Theorem 1.2
Tally Set

A set $T$ is a tally set exactly if $T \subseteq 1^*$

---

**Theorem 1.2**

If there is a tally set that is $\leq^p_m$-hard for NP, then $P=NP$.

---

**Corollary 1.3**

If there is a tally set that is NP-complete, then $P = NP$.

---

- Let $T$ be a tally set that is $\leq^p_m$-hard. Then the NP-complete set $SAT \leq^p_m T$.

- Goal: We want to use $SAT \leq^p_m T$ to proof that $SAT$ can be decided in polynomial time. Thus, $SAT \in P$, then $P = NP$
Tree Pruning For SAT Problem

- $F[v_i=\text{True}]$ denotes the resulting boolean formula when we assign True to variable $v_i$.

- Boolean formula $F$ is satisfiable if and only if $F[v_1=\text{True}]$ is satisfiable or $F[v_1=\text{False}]$ is satisfiable.

- Find the satisfiable assignment by traversing the tree. If the traverse can be done in polynomial time, then $SAT \in P$. 

$F[v_1=\text{True}]$ for SAT Problem
Tree Pruning For SAT Problem

- Traverse is done layer by layer. The number of nodes in $i^{th}$ layer is $2^i$.
- If during the traverse we can ignore some redundant nodes (tree pruning) so that for each layer we only traverse polynomial number of nodes, then the entire traverse is polynomial.
Example: Tree Pruning For SAT Problem

- (Rabbit says) What nodes/formulas are redundant?
- If a formula is not satisfiable, then all of its descendants are not satisfiable. Thus, this formula is redundant.
- If a formula is "identical" to another formula, then it is redundant.
- If $f_1$ is satisfiable if and only if $f_2$ is satisfiable, then $f_1$ and $f_2$ is identical.
- (Rabbit says) How do we identify the redundancy?
Tree Pruning For SAT Problem

The Self-Reducibility Technique

Group A

Theorem 1.2

Theorem 1.4

Layer 1

Layer 2

Layer m

F\[v_1=\text{True}\]

F\[v_1=\text{False}\]

F\[v_1=\text{True}, \ v_2=\text{False}\]

F\[v_1=\text{False}, \ v_2=\text{True}\]

F\[v_1=\text{False}, \ v_2=\text{False}\]

\ldots

\ldots

\ldots

\ldots

F\[v_1=\text{True}, \ v_2=\text{True}, \ \ldots, \ v_m=\text{True}\]

F\[v_1=\text{True}, \ v_2=\text{True}, \ \ldots, \ v_m=\text{False}\]

F\[v_1=\text{False}, \ v_2=\text{False}, \ \ldots, \ v_m=\text{True}\]

F\[v_1=\text{False}, \ v_2=\text{False}, \ \ldots, \ v_m=\text{False}\]
Identify Redundancy

- Let $g$ be the deterministic polynomial-time function such that $\forall f \in SAT$ if and only if $g(f) \in T$, where $T$ is the $\leq^p_m$-hard Tally set.

- Recall that $T \subseteq 1^*$. If $g(f) \notin 1^*$, then $f$ is not satisfiable.

- For any two boolean formula $f \neq h$, and $g(f) = g(h)$, $f \in SAT \iff h \in SAT$.

$$f \in SAT \iff g(f) \in T$$

$\parallel$

$$h \in SAT \iff g(h) \in T$$

- (Rabbit says) How do we make sure the number of remaining nodes/formulas in each layer is polynomial?
The length of the output of a polynomial-time function is bounded by some polynomial.

Let $g(x)$ be a polynomial-time function, there exists an integer $k$ such that $\forall x, |g(x)| \leq |x|^k + k$.

If $g(x) \in 1^*$, then the longest possible output is $1^{|x|^k+k}$. Thus, the total number of possible outputs of $g(x)$ is $|x|^k + k + 1$.

**Example**

Given that $|g(x)| \leq |x|^k + k$ and $g(x) \in 1^*$, what are the possible outputs of $g(x)$?

$\epsilon, 1, 11, 111, 1111, 11111, \ldots, 11\ldots111_{|x|^k+k}$
Recall that for any two boolean formula $f$, $h$, if $g(f) = g(h)$, then $f$ and $g$ are “identical”. Similarly, if $g(f) \neq g(h)$, we say $f$ and $g$ are “distinct”.

Recall that the total number of possible outputs of $g(x)$ is $|x|^k + k + 1$.

Let $n$ be the size of formulas on the $i^{th}$ layer. Thus, among the $2^i$ formulas in this layer, at most $n^k + k + 1$ of them are “distinct.”
Proof Sketch

Layer 1

Layer 2

Layer m

The Self-Reducibility Technique
Group A
Tree Pruning Technique
Theorem 1.2
Theorem 1.4
Proof Sketch

- The input of layer $i$ are the output formulas from layer $i - 1$.
- Expand each formula by assigning $True$ and $False$ value to $v_i$ (Get the corresponding formulas in layer $i$).
- For each expanded formula $f$ in layer $i$, calculate $g(f)$. If $g(f) \notin 1^*$, remove $f$. If $f \in 1^*$ but exists expanded formula $h \neq f$ such that $g(f) = g(h)$, remove $f$.
- Output the resulting formulas in layer $i$. 
Proof

Stage 0
Outputs $C = F$ where $F$ is the original formula

Stage $i$
Input $C = \{F_1, \ldots, F_l\}$

Step 1: Replace $v_i$ by True or False to get

$$C = \{F_1[v_i = True], F_2[v_i = True], \ldots, F_l[v_i = True],$$
$$F_1[v_i = False], F_2[v_i = False], \ldots, F_l[v_i = False]\}$$

Step 2: $C' = \emptyset$
Step 3: For each $f$ in $C$ do

1. Compute $g(f)$
2. If $g(f) \in 1^*$ and for no formula $h \in C'$ does $g(f) = g(h)$, then add $f$ to $C'$.

Output of stage $i : C = C'$

Stage $m+1$
Input is $C$ which is now a variable-free formula collection. $F$ is satisfiable if an element in $C$ is true.
Questions?
Theorem 1.4
Theorem 1.4
If there is a sparse set that is $\leq_{m}^{P}$-hard for coNP, then $P=NP$.

Corollary 1.5
If there is a sparse coNP-complete set, then $P=NP$. 
Observation

**Theorem 1.4**

If there is a sparse set that is $\leq^p_m$-hard for coNP, then $P=NP$.

**Definition**

A set $S$ is **sparse** if it contains at most polynomially many elements at each length, i.e.,

$$(\exists \text{ polynomial } p)(\forall n)[||\{x|x \in S \wedge |x| = n\}|| \leq p(n)].$$

**Definition**

A language $A$ is **coNP-hard**, if $\forall L \in \text{coNP}, L \leq^p_m A$. 
Observation

Idea

Utilize Tree-pruning trick and the definition of coNP-hard to construct a polynomial-time algorithm for SAT. (SAT is NP-complete)

Explanation

- $\forall L \in NP, L \leq^p_m SAT$
- SAT solved in polynomial-time by deterministic Turing machine (DTM).
- $\iff$ All NP problems solved in polynomial-time by DTM.
- $\iff P = NP$. 
Let $S$ be a sparse set and also coNP-hard.

**Definition**

$\forall \ell, p_\ell(n)$ denotes the polynomial $n^\ell + \ell$.

**Definition**

$\|S^{\leq n}\|$ denotes the number of strings with length less than $n$ in $S$. 
Observation

Corollary

\[ \forall n, \| S^{\leq n} \| \leq p_d(n). \]

Proof.

- **S** is sparse \( \Rightarrow \| \{ x \mid x \in S \land |x| = n \} \| \leq p(n) \)
- We can obtain the upper bound \( p_{\text{max}} = \max_n p(n) \), where \( p_{\text{max}} \) is bounded by polynomial.
- \( \| S^{\leq n} \| = \sum_{i=0}^{n} p(i) \leq \sum_{i=0}^{n} p_{\text{max}} = (n + 1)p_{\text{max}} \), which is bounded by polynomial.
Observation

Recall

\[ SAT \in NP \Rightarrow \overline{SAT} \in coNP \text{ and } \overline{SAT} \in coNP \Rightarrow \overline{SAT} \leq^p S, \]

since \( S \in coNP\text{-hard}. \)

Let \( g \) denote the reduction function \( \overline{SAT} \leq^p S. \)

Corollary

\[ \forall x, |g(x)| \leq p_k(|x|). \]

Proof.

- Function \( g \) is computed by a DTM
- a DTM outputs at most 1 symbol in one step

\[ \Rightarrow |g(x)| \text{ is bounded by polynomial length, named } p_k(|x|). \]
Observation

Since \( \forall n, \| S \leq^n \| \leq p_d(n) \) and \( \forall x, |g(x)| \leq p_k(|x|) \), given \( g \) and \( S \),

\[ \| S \leq |g(x)| \| \leq p_d(p_k(|x|)). \]

Rabbit: Interesting! \( S \leq |g(x)| \) is a set with a polynomial number of elements.
Deterministic polynomial-time algorithm for SAT

**Input**

Boolean formula $F[v_1, v_2, \ldots, v_m]$, w.l.o.g, $m \geq 1$.

**Stage 0**

- Collection of boolean formulas, $C' = \{F\}$
- Pass $C'$ to Stage 1.

Rabbit: That’s pretty easy. I can do it.
Deterministic polynomial-time algorithm for SAT

Rabbit: If we keep this procedure to Stage $m$, the number of strings in each level will grow larger and larger!!!
A collection of formulas: Hi, we are from Stage i-1.

**Stage i**

**Step 1:** \( C = \{ F_1[v_i = True], F_2[v_i = True], ..., F_\ell[v_i = True], F_1[v_i = False], F_2[v_i = False], ..., F_\ell[v_i = False] \} \).

**Step 2:** Set \( C' = \emptyset \).

Rabbit: lol, I can do it but where is my carrot?
Stage i

Step 3: For each formula $f$ in $C$ do:

1. Compute $g(f)$.
2. If for no formula $h \in C'$ does $g(f) = g(h)$, then add $f$ to $C'$.
Deterministic polynomial-time algorithm for SAT

Stage i

**Step 4:** If $C'$ contains at least $p_d(p_k(|F|)) + 1$ elements, stop and immediately declare that $F \in SAT$.

**Figure:** Reduction Mapping
Stage i

Step 4: If $C'$ contains at least $p_d(p_k(|F|)) + 1$ elements, stop and immediately declare that $F \in SAT$.

Explanation

- Only $p_d(p_k(|F|))$ strings are in $S_{\leq p_k(|F|)}$.
- There is at least one formula named $H$ maps to a string in $\overline{S}$, i.e., $g(H) \notin S$.
- Since $g$ is the reduction function from $\overline{SAT}$ to $S$, $H$ is satisfiable. It imply that $F$ is satisfiable.
Stage i

**End Stage i:** $C'$ is the collection that gets passed on to Stage $i + 1$. 

Deterministic polynomial-time algorithm for SAT
Deterministic polynomial-time algorithm for SAT

Stage $m+1$

If some member of the formula collection output by Stage $m$ evaluates to being true, $F \in SAT$, and otherwise $F \notin SAT$.

Rabbit: Oh, my carrot! The proof is done here. Wait, rabbit!
Discuss

Comment

Obviously, this algorithm is computed by deterministic Turing machine.

- **Step 4 never met**
  Upper bound number of strings $p_{max} = \max p_d(p_k(|F|))$.
  \[ \Rightarrow \text{time for whole algorithm } t \leq mp_{max} \]

- **Step 4 invoked**
  This algorithm stops early before Stage $m+1$.
  \[ \Rightarrow \text{The algorithm is polynomial-time} \]

We construct a deterministic polynomial-time algorithm for SAT.

Rabbit: If I find a carrot like this set $S$, I will buy a million carrots (plus 9 millions).
Thank You!
Group A Lecture 2: Mahaney’s Theorem

Adam Scrivener, Haofu Liao, Nabil Hossain, Shupeng Gui, Thomas Lindstorm-Vautrin
Why Study Sparseness?

- Closely related to the p-isomorphism conjecture:

  \begin{quote}
  \textbf{Lemma 5.1} Let $L_1$ and $L_2$ be two p-isomorphic languages. Then two polynomials $p_1$ and $p_2$ exist such that, for any $n$,
  \[ c_{L_1}(n) \leq c_{L_2}(p_1(n)) \quad \text{and} \quad c_{L_2}(n) \leq c_{L_1}(p_2(n)). \]
  \end{quote}

- In other words, the census functions are polynomially related.
- Since the census function for SAT is known to be exponential (Bov-Cre p. 83), if a sparse NP-complete language $L$ is found, then the census function for $L$ cannot be polynomially related to the census function for SAT. So, the p-isomorphism conjecture falls.
Difficulty of Finding a Sparse NP-C Language

- Due to a result by Mahaney:

**Theorem 5.7** If a sparse NP-complete language exists, then $P = NP$.

- And thus, finding such an NP-complete language means proving that $P = NP$, which is widely believed to be false.
- But why does Theorem 5.7 hold?
Table of Contents

- Theorem 5.6
  - Complement of S is in NP
  - Satisfiability tree pruning
  - SAT is in P

- Theorem 5.7 (Mahaney’s Theorem)
  - PC(S)
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in P
Theorem 5.6 If an NP-complete sparse language exists such that its census function is computable in polynomial time, then $P = NP$. (A census function $c_L$ is said to be computable in polynomial time if a Turing transducer $T$ exists which reads $0^n$ as input and computes $c_L(n)$ in polynomial time.)
Table of Contents

- **Theorem 5.6**
  - Complement of $S$ is in NP
  - Satisfiability tree pruning
  - SAT is in $P$

- **Theorem 5.7 (Mahaney’s Theorem)**
  - $PC(S)$
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in $P$
Complement of S is in NP

begin {input: x}
    n := |x|;
k := c_S(n);
guess y_1, \ldots, y_k in set of k-tuples of distinct words
    each of which has length, at most, n;
    {check whether the guessed k-tuple coincides with S_{\leq n}}
    for i = 1 to k do
        if NT(y_i) rejects then reject;
    {check if x \in S_{\leq n}}
    for i = 1 to k do
        if y_i = x then reject;
accept;
end.

Further, the complement of SAT reduces to S.
Table of Contents

- **Theorem 5.6**
  - Complement of S is in NP
  - **Satisfiability tree pruning**
  - SAT is in P

- **Theorem 5.7 (Mahaney’s Theorem)**
  - PC(S)
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in P
Satisfiability Trees and Tree Pruning
Satisfiability Trees and Tree Pruning

- Since the complement of SAT reduces to $S$ by some function $f$, we can use $f$ as a pruning function.
- If $y$ is a node in the satisfiability tree and $f(y)$ is in $S$, then we know that $y$ is not satisfiable, and we can ignore $y$’s children in the tree.
- It is difficult to determine if $f(y)$ is in $S$ directly since $S$ is NP-C.
- To solve this, we build a list of elements in $S$ as our tree pruning algorithm runs, which we can query.
Table of Contents

- **Theorem 5.6**
  - Complement of S is in NP
  - Satisfiability tree pruning
  - SAT is in P

- **Theorem 5.7 (Mahaney’s Theorem)**
  - PC(S)
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in P
SAT is in P

begin {input: x} {main program}
  list := \{f(false)\};
  if sat(x) then accept else reject;
end.

function sat(y): Boolean;
begin
  if y = true then sat := true;
  if f(y) \in list then sat := false
  else
    begin
      derive y_0 and y_1 from y;
      if \neg sat(y_0) \land \neg sat(y_1) then
        begin
          if both y_0 and y_1 are not satisfiable then y is not satisfiable
          list := list \cup \{f(y)\};
          sat := false;
        end
      else sat := true;
    end;
end;
Mahaney’s Theorem

**Theorem 5.7** If a sparse NP-complete language exists, then $P = NP$. 
Table of Contents

- **Theorem 5.6**
  - Complement of S is in NP
  - Satisfiability tree pruning
  - SAT is in P

- **Theorem 5.7 (Mahaney’s Theorem)**
  - PC(S)
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in P
Mahaney’s Theorem

- We have shown that we can prove $P = NP$ if the sparse NP-C set has a poly-time computable census function.
- Can we still do it if we don’t have this assumption?
- Yes, and it will require “guessing” the correct value of the census function.
Define PC(S) as the set of triples \(<x,k,0^n>\) accepted by this machine:

```
begin {input: x, k, 0^n}
  if |x| > n \lor k > p(n) then reject;
  guess y_1, \ldots, y_k in set of k-tuples of distinct words
  each of which is of length, at most, n;
  for i = 1 to k do
    if NT(y_i) rejects then reject;
  for i = 1 to k do
    if y_i = x then reject;
  accept;
end.
```

**Note** that if \(|x| \leq n\) and \(k = c_s(n)\), then \(<x,k,0^n>\) is in the set if and only if \(x\) is in \(S\) complement.
Table of Contents

- **Theorem 5.6**
  - Complement of S is in NP
  - Satisfiability tree pruning
  - SAT is in P
- **Theorem 5.7 (Mahaney’s Theorem)**
  - PC(S)
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in P
Constructing a good pruning function

- Before, we used a reduction from the complement of SAT to $S$. This time we do the exact same thing, with a new reduction.
- First, let $h$ be a reduction from SAT to $S$, and $g$ be a reduction from $PC(S)$ to $S$.
- Let $p_h$ and $p_g$ limit the length of $h$ and $g$.
- Now, suppose that $x$, the string we want to determine is satisfiable or not, is of length $n$.
- Define $F^*(y) = g(h(y), c_s(p_h(n)), 0^{p_h(n)})$
- Claim: $F^*(y)$ is a reduction from the complement of SAT to $S$. 
Table of Contents

- **Theorem 5.6**
  - Complement of S is in NP
  - Satisfiability tree pruning
  - SAT is in P
- **Theorem 5.7 (Mahaney’s Theorem)**
  - PC(S)
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in P
A set of pruning functions

- For each $n$ and for each $k \leq p(p_h(n))$, where $p$ is a polynomial bounding the census function for $S$:
- Define $f_{n,k}(y) = g(h(y), k, 0^{p_h(n)})$.
- These are all poly-time computable when $|y| \leq n$
- Note that $F^*(y) = g(h(y), c_s(p_h(n)), 0^{p_h(n)}) = f_{n,c_s(p_h(n))}(y)$
- So, $F^*$ is among this set of functions, and it is a poly-time reduction from the complement of SAT to $S$, thus it is a valid pruning function.
- The problem: we do not know the census function, so we cannot compute $F^*(y)$. 
A set of pruning functions

- Observation: $c_s(p_h(n)) \leq p(p_h(n))$
- Solution idea: On input $x$ of size $n$, for all values of $k$ from 0 to $p(p_h(n))$, try the tree-pruning algorithm with pruning function $f_{n,k}$
- One of these $k$’s will work.
- Problem: we are not guaranteed that the tree-pruning algorithm will run in poly time with every choice of pruning function.
Table of Contents

- **Theorem 5.6**
  - Complement of S is in NP
  - Satisfiability tree pruning
  - SAT is in P
- **Theorem 5.7 (Mahaney’s Theorem)**
  - PC(S)
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in P
Properties of the correct pruning function $F^*$

- Observation: if $k = c_s(p_h(n))$, we have that
  \[ |\langle h(y), k, 0^{p_h(n)} \rangle| = 2p_h(n) + O[\log(p(p_h(n)))] \]

- Thus, there is an $n_0$ such that for all $n \geq n_0$,
  \[ |\langle h(y), k, 0^{p_h(n)} \rangle| \leq 2p_h(n) + p(p_h(n)) \]

- This means that the list created by the tree-pruning algorithm will be at most
  \[ p(p_g[2p_h(n) + p(p_h(n))]) \]
Properties of the correct pruning function $F^*$

- So, we know from the poly-bound proof of the tree-pruning function, that the algorithm visits at most

$$2[|x|p(p_g[2p_h(n) + p(p_h(n))]) + |x| - 1]$$

Nodes, which is polynomial in $|x|$.

- Further, if the amount of nodes explored exceeds this value, we know we have chosen the wrong $k$.  

Table of Contents

- **Theorem 5.6**
  - Complement of $S$ is in NP
  - Satisfiability tree pruning
  - SAT is in P

- **Theorem 5.7 (Mahaney’s Theorem)**
  - $PC(S)$
  - Constructing a good pruning function
  - A set of pruning functions
  - Properties of the correct pruning function
  - SAT is in P
SAT is in P

begin {input: \( x \)}
   \text{for } k = 0 \text{ to } p(p_h(|x|)) \text{ do}
   \begin{align*}
   \text{begin} \\
   \text{execute the tree-visiting algorithm described in the} \\
   \text{proof of Theorem 5.6 using } f_{|x|,k} \text{ as a} \\
   \text{pruning function and visiting, at most,} \\
   \text{2}||x|p(p_g[2p_h(n)+p(p_h(n))])+|x|-1| \text{ inner nodes; if} \\
   \text{the algorithm accepts then accept;}
   \text{end;}
   \text{reject;}
   \text{end.}
\end{align*}
end.
Group A Lecture 3: Sparse Sets and Turing Reductions

Adam Scrivener, Haofu Liao, Nabil Hossain, Shupeng Gui, Thomas Lindstrom-Vautrin

Department of Computer Science
University of Rochester

November 9, 2015
Introduction

- We’ve seen: if a sparse set is NP-hard or NP-complete w.r.t many-one reductions, then $P = NP$

- Today we investigate whether any sparse set can be NP-hard or NP-complete w.r.t. Turing reductions
  - weaker assumption (compared to many-one reductions)

- **Open Question:** do these Turing reduction based hypotheses imply that $P = NP$?

- We can show that the Polynomial Hierarchy **collapses**, given these assumptions hold

- Intuitively, PH **collapses** when all polynomial classes above a certain order are shown to be equal
The Polynomial Hierarchy

\[ \Sigma_0^P = \Pi_0^P = \Delta_0^P = \Delta_1^P = \mathbb{P} \]

\[ \Sigma_i^P = \text{NP} \quad \text{and} \quad \Pi_i^P = \text{co-NP} \quad \text{for} \quad i \geq 0 \]

E.g., \( \Sigma_1^P = \text{NP} = \text{NP} \quad \text{and} \quad \Pi_1^P = \text{co-NP} \)

Theorem 1.14

Restricted Query Classes

Proof

Step 1

Step 2

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_3^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_2^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_1^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \mathbb{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \cap \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{P} \)

np-complete

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_2^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_1^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \mathbb{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \cap \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_2^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_1^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \mathbb{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \cap \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_2^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_1^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \mathbb{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \cap \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_2^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_1^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \mathbb{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \cap \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_2^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \Sigma_1^P \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \mathbb{P} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{NP} \cap \text{co-NP} \)

\( \exists \mathbb{P} \forall \mathbb{P} \epsilon \text{P} \)
Polynomial Hierarchy Collapse

$$PH = \bigcup \Sigma_i^p$$

$$= P \cup NP \cup NP^{NP} \cup NP^{NP^{NP}} \cup NP^{NP^{NP^{NP}}} \cup \ldots$$

**Theorem (Gold)**

If $$\Sigma_i^p = \Pi_i^p$$, then $$PH = \Sigma_i^p$$

**Example**

if $$NP = coNP$$, then $$PH = NP$$

- An implication: any problem, that can be solved using an NP machine with access to an NP oracle, can also be solved with a non-det poly-time TM with no access to oracle

---

Can sparse sets be NP-complete or NP-hard w.r.t Turing reductions? What are the implications?

**Theorem 1.14 [Hemaspaandra-Ogihara]**

If NP has sparse Turing-complete sets, then the Polynomial Hierarchy collapses to $P^{NP}[\log n]$

**Theorem 1.15 [Hem-Ogi] (also called Karp-Lipton Thm)**

If NP has sparse Turing-hard sets, then the Polynomial Hierarchy collapses to $NP^{NP}$
Bounded/restricted query classes

- **Motivation:** add new classes to PH to capture problems that can be solved by restricting the number of queries made to oracle to $O(\log n)$ (instead of polynomial)

E.g. Odd colorability in graphs

- it is in $P^{NP}$ and is NP-hard, but not known whether it is in NP
- but it can be solved in $P^{NP}[\log n]$ by using a $P^{NP}$ machine making only $O(\log n)$ (rather than polynomially many) queries to an NP oracle.

---

Theorem 1.14

If NP has sparse $NP \leq_p^T$-complete sets, then

$$PH = P^{NP}[\log n]$$

Proof Strategy

- proof uses “census” approach

- For a sparse set, the census approach is to first obtain the exact number of elements in the set up to some given length, and then exploit that information.
Proof of Theorem 1.14

1) Let $S$ be a sparse set s.t. $S$ is $\leq^p_T$-complete for NP
2) For any $\ell$, let $p_\ell(n) = n^\ell + \ell$
3) Let $j$ be s.t. $(\forall n)[|S^{\leq n}| \leq p_j(n)]$
4) Let $M$ be deterministic poly-time TM s.t. $SAT = \mathcal{L}(M^S)$
   - $M$ exists since $S$ is Turing-hard for NP
5) Let $k$ be s.t. $p_k(n)$ bounds runtime of $M$ regardless of $M$’s oracle
6) Let $L$ be an arbitrary set in $\Sigma_2^p$ (i.e. $NP^{NP}$)
   - Since SAT is NP-complete, we have $\Sigma_2^p = NP^{SAT}$
7) $\exists$ non-det poly-time TM $N$ s.t. $L = \mathcal{L}(N^{SAT})$
8) Let $\ell$ be s.t. $p_\ell(n)$ bounds non-det runtime of $N$ for all oracles
9) Note that $L = \mathcal{L}(N^{\mathcal{L}(M^S)})$
Proof of Theorem 1.14 (Continued)

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

- \( V \in NP \) since \( S \in NP \)
- 1st set (census): will be exploited to compute \( |S^{\leq n}| \)
- 2nd set: will be used to check whether \( x \in L \)
Proof of Theorem 1.14 (Continued)

The following $P^{NP}[\log n]$ algorithm accepts $L$ by making $O(\log n)$ calls to the NP oracle $V$, for each input string $y$

**Step 1**

- In $O(\log |y|)$ sequential queries to $V$, compute $|S \leq p_k(p_\ell(|y|))|$
  - queries of the form $0 \# 1^{p_k(p_\ell(|y|))} \# 1^z$
  - vary $z$ as in binary search until we find $|S \leq p_k(p_\ell(|y|))|$
- Since $|S \leq p_k(p_\ell(|y|))|$ is bounded by $p_j(p_k(p_\ell(|y|)))$, thus $O(\log |y|)$ queries are sufficient to find census value
- Let the census value obtained be $r
Proof of Theorem 1.14 (Continued)

Step 2

Ask \( V \) the query \( 1\#y\#1^{p_k(p_\ell(|y|))}\#1^r \), and accept if and only if this query \( \in V \)

- Clearly this is a \( P^{NP}[\log n] \) algorithm
- Algorithm accepts \( L \)
- Since \( L \in \Sigma_2^P \) \( (= NP^{NP}) \), we have \( \Sigma_2^P = P^{NP}[\log n] \).
- Since \( P^{NP}[\log n] \) is closed under complementation, we have \( \Sigma_2^P = \Pi_2^P \)
- Therefore by Theorem (Gold), \( PH = \Sigma_2^P = P^{NP}[\log n] \)
THE KARP-LIPTON THEOREM

If there is a sparse NP-Turing-Hard set (\( \exists S \in \text{SPARSE} \forall T \in \text{NP} \exists S \))

the polynomial hierarchy collapses to \( \text{NP}^{\text{NP}} \) \( \text{(PH=NP}^{\text{NP}}) \)

(also congratulations if you did the reading, it is almost impossible)
A LITTLE REVIEW

What the heck does this mean?

SPARSE SETS:

\[ p(n) = n^2 + 1 \]

alphabet size \( |\Sigma| = 3 \)

It's easy to see why they're called sparse.

As a proportion of possible elements, sparse sets contain almost nothing.

\[ |\Sigma^*|^n = |\Sigma|^n = 3^n \]
A LITTLE REVIEW

A sparse NP-Turing-Hard set:

We have a sparse set $S$

Every set in NP can be solved by a machine in polynomial time with an oracle for $S$.

(not necessarily the same machine $M$)
A LITTLE REVIEW

The polynomial hierarchy (PH):

\[ \text{PH} = \text{P} \cup \text{NP} \cup \text{NP}^\text{NP} \cup \text{NP}^\text{NP} \cup \text{NP}^\text{NP} \cup \text{NP}^\text{NP} \cup \text{NP}^\text{NP} \cup \text{NP}^\text{NP} \cup \ldots \]

Note: \( \text{NP}^\text{P} = \text{NP} \)

and

\[ \text{P} \subseteq \text{NP} \subseteq \text{NP}^\text{NP} \subseteq \text{NP}^\text{NP} \subseteq \text{NP}^\text{NP} \subseteq \text{NP}^\text{NP} \subseteq \ldots \]
A LITTLE REVIEW

Polynomial time hierarchy collapse:

essentially when all subset-equals signs become
equals signs above a certain order.

OR in other words

All polynomial classes above a certain order are
shown to be equal

\[ P \leq NP = \text{NP}^\text{NP} \leq \text{NP}^\text{NP} \leq \text{NP}^\text{NP} \leq \text{NP}^\text{NP} \leq \ldots \]

(a collapse of PH to \( \text{NP}^\text{NP} \))

\[ \text{PH} = \text{P} \cup \text{NP} \cup \text{P} \cup \text{NP} \cup \text{NP} \cup \text{NP} \cup \ldots \]
How do we show a polynomial hierarchy collapse to $\text{NP}^{\text{NP}}$?

Notice if we show $\text{NP}^{\text{NP}} = \text{NP}^{\text{NP}^{\text{NP}}}$, all other higher order polynomial classes will collapse recursively:

$$
\ldots \leq \text{NP}^{\text{NP}} = \text{NP}^{\text{NP}^{\text{NP}}} = \text{NP}^{\text{NP}^{\text{NP}^{\text{NP}}}} = \ldots
$$

So $\text{NP}^{\text{NP}} = \text{NP}^{\text{NP}^{\text{NP}}} = \ldots$ is all we need to show.
AVAILBLE TOOLS

Remember we need to use our sparse set $S$ and the fact that any language in NP Turing reduces to $S$.

Since SAT is in NP,

$\text{SAT} \leq_T S$

So we have a machine $M$ such that

$\text{SAT} = L(M^S)$

IMPORTANT: $M$ is a deterministic Turing machine that runs in polynomial time.

In other words this machine $M$ given an oracle for $S$ is a machine for SAT.
LINES OF INQUIRY

We want to show that an arbitrary element of \( NP^{NP^{NP}} \) can be solved by an element of \( NP^{NP} \).

First we must see how to represent these:

\[ L \leq NP^{NP^{NP}} \]

\[ L = L(N_1, L(N_2^L(N_3))) \quad \text{for machines } N_1, N_2, N_3 \]

(nondeterministic poly time)
LINES OF INQUIRY

What the heck does this mean?

\[ L = L(N_1, L(N_2, L(N_3))) \]

Note: since SAT is NP complete,

we can make \( N_2 \) do the polynomial work of \( f \) and replace \( N_1 \) with a machine for SAT
POLYNOMIAL BOUNDS

Here are the polynomial bounds we will use and keep track of:

First, define $p_k(n)$ for natural numbers $k$ and $n$

$\overline{p_k(n)} = n^2 + k$.

For prefix of our sparse set:

$\forall n \left[ \exists S^{\leq n} \subseteq P_j(n) \right]$ (we can polynomially bound any prefix of our sparse set)

For all oracles $A$:

$\forall k \left[ \text{runtime} \left( M^A(x) \right) \leq p_k(1w) \right]$ (M is our deterministic polynomial TM such that $L(M^A)$ = SAT)

For all oracles $A$:

$\forall k \left[ \max \left( \text{runtime} \left( N_1^A(x) \right), \text{runtime} \left( N_2^A(x) \right) \right) \leq p_k(1w) \right]$ (N_1 and N_2 are nondeterministic polynomial TMs, we will be investigating in our proof)
Pause to Ponder 1.15  Show why this "without loss of generality claim" holds.

(Answer sketch for Pause to Ponder 1.15: Given a machine $M$, let the machines $M_1, M_2, \ldots$, be as follows. $M_i^A(x)$ will simulate the action of exactly $p_i(|x|)$ steps of the action of $M^A(x)$, and then will halt in an accepting state if $M^A(x)$ halted and accepted within $p_i(|x|)$ steps, and otherwise will reject. Note that since the overhead involved in simulating one step of machine is at most polynomial, for each $i$, there will exist an $\hat{i}$ such that for every $A$ it holds that $M_i^A$ runs in time at most $p_i(n)$. Furthermore, in each relativized world $A$ in which $M^A$ runs in time at most $p_i$, it will hold that $L(M^A) = L(M_i^A)$. Relatedly, in our proof, given the machine $M$ such that $\text{SAT} = L(M^S)$, we will in light of whatever polynomial-time bound $M^S$ obeys similarly replace $M$ with an appropriate $M_j$ from the list of machines just described.)
HOW DO POLYNOMIAL BOUNDS WORK?

Consider $M^A(x)$. Its runtime is bounded by $P_k(W)$. At each step $M^A$ can print at most one character/letter. So the largest query $M$ can make to $A$ is one in which it uses all its steps to form the query and submits it in the last step.

So query size is bounded by $P_k(W)$. 
MAIN INSIGHTS

Can we simulate $N_2^{SAT}$ which works in NP$^P$
with a machine that works in NP?

$N_2^{SAT \leq L_1}$ works in “NP” since SAT$\leq L_1$ is finite
and in "P" and "NP" = "NP".

(x is the input to SAT, not to $N_2$. It's the query.)

(since we can't actually easily find SAT$\leq L_2$
this is assuming we can)
MAIN INSIGHTS

Simulating SAT:
Suppose you know the length of the input to \( M, |x| \).
\( M \) can make a query of length at most \( p(|x|) \).

On inputs of these lengths,
\( M \leq^* p(|x|) \) simulates SAT

Takeaway:
A finite prefix of \( S \) is enough to simulate SAT given the input length.
Since $M$ is deterministic and polynomial, and $S \leq P^{\text{pol}}$, being finite is in $P$

$M \in P^{P} = P$
Now that we have the main insights, I will walk you through the meat of the formal proof.
Define

\[ V_0 = \{ 0\#1^n\#S' \mid (\exists z \in (\Sigma^*)^{\leq n})[(a) \ z \text{ is not a well-formed formula} \]
\[ \text{and } M^{S'}(z) \text{ accepts; or (b) } z \text{ is a well-formed formula free} \]
\[ \text{variables and either (b1) } M^{S'}(z) \text{ accepts and } z \notin \text{ SAT} \text{ or} \]
\[ \text{(b2) } M^{S'}(z) \text{ rejects and } z \in \text{ SAT; or (c) } z \text{ is a well-formed} \]
\[ \text{formula variables } z_1, z_2, \ldots \text{ and it is } \text{not} \text{ the case that: } M^{S'}(z) \]
\[ \text{accepts if and only if} \]
\[ (M^{S'}(z[z_1 = \text{True}]) \text{ accepts } \lor M^{S'}(z[z_1 = \text{False}]) \text{ accepts}) \}, \]

where, as defined earlier in this chapter, \( z[\ldots] \) denotes \( z \) with the indicated variables assigned as noted.

\[ V_1 = \{ 1\#S'\#z \mid z \in L(N^L_2(M^{S'})) \}. \]

\[ V = V_0 \cup V_1. \]
Step 1  Nondeterministically guess a set $S' \subseteq (\Sigma^*)^{\leq pk(p_k(p_k(|y|)))}$ satisfying $\|S'\| \leq p_j(p_k(p_k(|y|))))$. If $0#1^{pk(p_k(p_k(|y|)))} # S' \in V$ then reject. Otherwise, go to Step 2.

Step 2  Simulate the action of $N_1(y)$ except that, each time $N_1(y)$ makes a query $z$ to its $L(N_2^{SAT})$ oracle, ask instead the query $1#S' # z$ to $V$. 