Local Algorithms for Finding a Missing String

**Missing String:** Given a list of $M$ strings of length $N$ ($M < 2^N$), find a string not on the list.

Cantor’s Diagonal Argument shows:

If $M \leq N$, then we can find a missing string by taking the diagonal of the list.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>k</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$x_2$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_k$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Local Algorithms for Finding a Missing String**

**MISSING STRING:** Given a list of $M$ strings of length $N$ ($M < 2^N$), find a string not on the list.

If the list is truth tables of “easy” functions, we are asking you to find a “hard” function!

Cantor’s Diagonal Argument shows:

If $M \leq N$, then we can find a missing string by taking the diagonal of the list.

<table>
<thead>
<tr>
<th>$M$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>$k$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$x_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_k$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For all $i = 1, \ldots, N$, we can obtain the $i$-th bit of a missing string with **only one bit probe** into the input:

[probe the $i$-th bit of the $i$-th string, output the opposite bit]

That is, when $M \leq N$, there’s a “1-probe” algorithm for **MISSING STRING**

**Under what conditions can we use only $k$ probes? (Why?)**
Local Algorithms for Finding a Missing String

**MISSING STRING:** Given a list of $M$ strings of length $N$ ($M < 2^N$), find a string not on the list

If $M \leq N$, then we can find a missing string by taking the diagonal of the list.

For all $i = 1, \ldots, N$, we can obtain the $i$-th bit of a missing string with **only one bit probe**. 

*That is, there is a “1-probe” algorithm for **MISSING STRING**.*

**What can we do with $k$ probes?**

Theorem 1 (easy): For $M = kN$, there is no $k - 1$ probe algorithm for **MISSING STRING**.

**Proof:** Suppose for each $i = 1, \ldots, N$, your algorithm makes only $k - 1$ probes and computes a missing string $y$.

The total number of different strings you probed, over all $i$, is $\leq (k - 1)N < M$.

So there’s some string on the list you haven’t probed at all.

Your algorithm fails on every input that includes $y$ among those non-probed strings.
**Local Algorithms for Finding a Missing String**

**MISSING STRING:** Given a list of $M$ strings of length $N$ ($M < 2^N$), find a string not on the list.

**Theorem 2:** For $M \leq kN$, there is an $O(k \log k)$-probe algorithm for MISSING STRING.

**Idea:** Combine the diagonal argument and the algorithm that makes $O(M)$ probes!

Divide the $N$ bits of each string into $\sim N/b$ blocks of length $b$

Divide the $M$ strings into $\sim M/t$ blocks of length $t$, where $t \leq 2^b - 1$.

If $\frac{M}{t} \leq \frac{N}{b}$, then if we find a missing string for each purple block, their concatenation is a missing string for the entire set!

**Have:** $M = kN$, **Want:** $M \leq (2^b - 1) N/b$. We therefore set:

$$b = \log(k) + \log\log(k) + O(1)$$

Then, to get any particular bit of the missing string, number of probes is

$$O(t) \leq O(2^b) \leq O(k \log k)$$
What Good are Local Algorithms?

Theorem 2: For $M \leq kN$, there is an $O(k \log k)$-probe algorithm for MISSING STRING.

Corollary: New Time Hierarchy Theorems, against Non-Uniform Programs!

Time Hierarchy Theorem: For “reasonable” $g, h$ where $h(n) \gg g(n)$,

$\text{TIME}(h(n)) \not\subseteq \text{TIME}(g(n))$

The time hierarchy can be generalized to work for “small non-uniform advice”

Define $f \in \text{TIME}[g(n)]/a(n)$ if for every $n$, there is some program of length $a(n)$, running in time $g(n)$, that decides $f$.

Time Hierarchy Against Advice [Folklore]: For “reasonable” $g, h$ where $h(n) \gg g(n)$,

$\text{TIME}(h(n)) \not\subseteq \text{TIME}(g(n))/n$

Idea: The hard function running in $O(h(n))$ time treats its input of length $n$ as a program, and simulates the program on its own code, outputting the opposite answer.
What Good are Local Algorithms?

Theorem 2: For $M \leq kN$, there is an $O(k \log k)$-probe algorithm for MISSING STRING.

Corollary: New Time Hierarchy Theorems, against Non-Uniform Programs!

Time Hierarchy Theorem: For “reasonable” $g, h$ where $h(n) \gg g(n)$,

$$\text{TIME}(h(n)) \nsubseteq \text{TIME}(g(n))$$

The time hierarchy can be generalized to work for “small non-uniform advice”

Define $f \in \text{TIME}[g(n)]/a(n)$ if for every $n$, there is some program of length $a(n)$, running in time $g(n)$, that decides $f$.

Time Hierarchy Against Advice, Version 2 [Folklore]: For “reasonable” $g, h$

$$\text{TIME}(2^n + g(n) \cdot h(n)) \nsubseteq \text{TIME}(h(n))/g(n)$$

Idea: The hard function enumerates all $2^{g(n)}$ programs of length $g(n)$, and simulates each of them on every possible $n$-bit input. Then it computes a missing string from the list of $M = 2^{g(n)}$ strings of length $N = 2^n$
What Good are Local Algorithms?

Theorem 2: For \( M \leq kN \), there is an \( O(k \log k) \)-probe algorithm for Missing String.

Corollary: New Time Hierarchy Theorems, against Non-Uniform Programs!

Old Time Hierarchies Against Advice: For “reasonable” \( g, h \) where \( h(n) \gg g(n) \),
\[
\text{TIME}(h(n)) \not\subseteq \text{TIME}(g(n)) / (n) \quad \text{TIME}(2^{n+g(n) \cdot h(n)}) \not\subseteq \text{TIME}(h(n)) / g(n)
\]

New Hierarchy Against Advice [informally]: for \( g(n) \geq n, h(n) \geq n \),
\[
\text{TIME}(2^{g(n) \cdot g(n) \cdot h(n) / 2^n}) \not\subseteq \text{TIME}(h(n)) / (g(n))
\]

Finding a function not in \( \text{TIME}(h(n)) / (g(n)) \) amounts to finding a missing string
of length \( N = 2^n \) from a list of \( M = O(2^{g(n)}) \) strings [all programs of length \( g(n) \)]
where any particular bit of any particular string can be determined in \( O(h(n)) \) time.

For \( k = 2^{g(n)-n} \), we have \( M \leq kN \). Thus we can compute any bit of some missing string,
probing \( O(k \log k) \leq O(2^{g(n)-n} \cdot g(n)) \) bit positions, each probe taking \( O(h(n)) \) time.
What Good are Local Algorithms?

Theorem 2: For $M \leq kN$, there is an $O(k \log k)$-probe algorithm for Missing String.

Corollary: New Time Hierarchy Theorems, against Non-Uniform Programs!

Some New Hierarchies:

$$\text{TIME}(5^n) \not\subseteq \text{TIME}(2^n)/(2n)$$

$$\text{TIME}(n^{2c+1}) \not\subseteq \text{TIME}(n^c)/(n + c \log(n))$$

$$\text{TIME}(2^{cn} \cdot \text{poly}(n)) \not\subseteq \text{TIME}(O(n))/(cn + n)$$

Note: All the above hierarchies relativize, and there is an oracle $A$ such that $\text{TIME}^A[2^{cn}] \subseteq \text{TIME}^A[O(n)]/(cn + n)$!!
(Main) Open Problems

• Does \textsc{Missing String} have small uniform depth-3 circuits, or not? (it cannot have depth-2 circuits, and it does have depth-4 circuits)

• How many probes (as a function of $k$) are necessary and sufficient to find a missing string, when $M \leq k N$? The answer is somewhere between $k$ and $O(k \log k)$

• More connections between \textsc{Missing String} and lower bounds?
How to Prove Lower Bounds With Algorithms

Lecture 4: The Power of Constructing Bad Inputs

(based on work with Lijie Chen, Ce Jin, and Rahul Santhanam, FOCS 2022)
Lower Bounds are Hard to Prove

There are many barriers

• Relativization [Baker-Gill-Solovay, 70’s]
• Natural Properties [Razborov-Rudich, 90’s]
• Algebrization [Aaronson-Wigderson, 00’s]

Summary: The standard methods that we use to reason about generic computation cannot resolve $P \neq NP$ (or $P \neq PSPACE$, or $EXP \neq ZPP$, or $NEXP \neq BPP$, etc.)

We apparently know a lot about what strong lower bounds CANNOT look like. We know many limitations on how such proofs must proceed.
What *could* a proof of P ≠ NP look like?

We seem to know a lot about what a proof of P ≠ NP (and P ≠ PSPACE, EXP ≠ ZPP, NEXP ≠ BPP, etc.) *cannot* look like...

... can we identify properties that such lower bound proofs *must* possess?

What properties are missing from lower bounds we know how to prove, which we will *have* to include in a proof of NEXP ≠ BPP (or any of the above)?
What could a proof of $P \neq NP$ look like?

Let $f : \{0,1\}^* \rightarrow \{0,1\}$ and let $\mathcal{A}$ be a class of algorithms.

A lower bound "$f \notin \mathcal{A}$" is a claim of the form:

$$(\forall A \in \mathcal{A})(\exists \infty n)(\exists x_A \in \{0,1\}^n)[A(x_A) \neq f(x_A)]$$

Fix a lower bound problem $f \notin \mathcal{A}$, and fix an algorithm $A$. What is the complexity of constructing a “bad” $x_A$ of length $n$?

The literature on lower bounds gives roughly two types of answers:

1. “Random” or non-constructive ways of choosing $x_A$
   Proofs relying on counting/information-theoretic arguments

2. “Efficient” ways of choosing $x_A$
   Proofs based on diagonalization arguments
What could a proof of $P \neq NP$ look like?

Our starting inspiration is from [Gutfreund-Shaltiel-Ta Shma’05]

If $P \neq NP$, then
“bad inputs to SAT algorithms can be efficiently constructed”.

The following theorem can be derived from their paper:

For every $n^k$-time algorithm $A$, there is an algorithm $R_A$ such that for infinitely many $n$, $R_A(1^n)$ outputs a formula $F_n$ of length $n$ such that $F_n$ is SAT $\iff A(F_n) = 0$.

Furthermore, $R_A$ runs in $n^{O(k^2)}$ time.
Gutfreund-Shaltiel-Ta Shma ‘05

If $P \neq NP$, then for infinitely many input lengths, “bad inputs to SAT algorithms can be efficiently constructed”

Let’s start by getting a refuter for an $n^k$-time algorithm $A$ trying to print SAT assignments, when they exist. (It first attempts to print a SAT assignment, and outputs UNSAT if that assignment fails to satisfy.)

All bad inputs are satisfiable formulas on which $A$ prints UNSAT

$R_A(1^n)$: Construct a formula $F_n$ encoding the property:

$(\exists G. |G| = n \text{ and assignment } a)[G(a) = 1 \land A(G) \text{ outputs UNSAT}]$

Run $A$ on $F_n$. If $A$ prints some $G'$, then output $G'$ else output $F_n$

We are asking $A$ to print its own counterexamples $G'$. If $A$ does this for $\infty$ many $n$, then we are done. If $A$ does not do this for $\infty$ many $n$, then $\{F_n\}$ is printed for almost every $n$. This set contains $\infty$ many counterexamples, because we assumed $P \neq NP$. 
Gutfreund-Shaltiel-Ta Shma ’05

If $P \neq NP$, then for infinitely many input lengths, “bad inputs to SAT algorithms can be efficiently constructed”

Getting a refuter for an $n^k$-time algorithm $A$ trying to decide SAT:

$R_A(1^n)$: Construct a formula $F_n$ encoding the property:
$(\exists G, |G| = n$ and assignment $a)[G(a) = 1 \land A(G)$ outputs UNSAT$]$

Use search-to-decision on $A$ to search for a SAT assignment to $F_n$.
(Try setting a variable $x$ to 0, then as 1, seeing what $A$ reports...)

Cases: (1) $A(F_n) = “UNSAT”$. Then output $F_n$
(2) We find a SAT assignment $G’$. Then output $G’$
(3) We find a subformula $F’’$ such that $A(F’’)=“SAT$,$$
\text{but } A(F’’[x = 0]) = A(F’’[x = 1])=“UNSAT”$

In case (3), $A$ is wrong on at least one of the 3 subformulas!
Can report the set as “bad”. (To get an $R_A$ so that only one is reported, consider three different algs $R_A^1, R_A^2, R_A^3$ such that $R_A^i$ reports the $i$-th)
What could a proof of \( P \neq NP \) look like?

Our starting inspiration is from [Gutfreund-Shaltiel-Ta Shma’05]

If \( P \neq NP \), then
“bad inputs to SAT algorithms can be efficiently constructed”.

The following theorem can be derived from their paper:

For every \( n^k \)-time algorithm \( A \),
there is an algorithm \( R_A \) that for infinitely many \( n \), \( R_A(1^n) \) outputs a formula \( F_n \) of length \( n \) such that \( F_n \) is SAT ⇔ \( A(F_n) = 0 \).
Furthermore, \( R_A \) runs in \( n^{O(k^2)} \) time.

Say there is a \( P \)-constructive separation of \( f \notin A \)
if for all \( A \)-algorithms \( A \), there is a poly-time algorithm \( R_A \) which
(for \( \infty \) \( n \)) given \( 1^n \) can output \( x_A \) such that \( A(x_A) \neq f(x_A) \).

[GST’05] \( P \neq NP \) ⇒ There’s a \( P \)-constructive separation of SAT \( \notin P \)
Constructive Separations

There is a $P$-constructive separation of "$f \not\in A$"
if for all $A$-algorithms $A$, there is a poly-time algorithm $R_A$ which
(for $\infty$ $n$) given $1^n$ can output $x_A$ such that $A(x_A) \neq f(x_A)$.

[GST’05] $P \neq NP \implies$ There’s a $P$-constructive separation of SAT $\not\in P$

Which lower bound problems require constructive separations, and which do not?

This is an algorithmic question about the nature of the lower bound. What algorithms are implied by separations?
We Argue: Constructive Separations Are Key!

1. Essentially all open separation problems regarding polynomial time require constructive separations

   **Thm:** For all $C \in \{P, ZPP, BPP\}$ and $D \in \{NP, PSPACE, EXP, NEXP, \ldots\}$ if $C \neq D$ then there is a $C$-constructive separation of $C \neq D$.

2. Making many known lower bounds constructive, implies major lower bounds!

   **Example 1:** [Maass 84] $\text{PALINDROMES} \not\in NTIME^{1[o(n^2)]}$. 

   **Thm:** Making Maass’ lower bound $P$-constructive implies $TIME(2^{O(n)})$ contains functions of exponential circuit complexity. “Constructivizing” this lower bound implies universal derandomization! [IW97]

   Therefore, “constructivity” is a property we want of lower bounds!
2. Making many *known* lower bounds constructive, *requires* resolving other major lower bound problems

**Example 2: Randomized streaming algorithm lower bounds.**

Many problems (including simple ones like DISJOINTNESS) are well-known to require $\Omega(n)$-space randomized streaming algorithms.

**Thm:** For any $\Pi \in NP$, a $P^{NP}$-constructive separation of $\Pi$ from $(\log n)^{\omega(1)}$ space randomized streaming algs implies $EXP^{NP} \neq BPP$.

**Example 3: Randomized query lower bounds.**

Promise-MAJORITY: distinguish bit strings with $> \frac{1}{2} + \epsilon$ ones from strings with $< \frac{1}{2} - \epsilon$ ones. Well-known to require $\Theta\left(\frac{1}{\epsilon^2}\right)$ queries.

**Thm:** A “uniform $AC^0$”-constructive separation of Promise-MAJORITY from randomized query algorithms using $o\left(\frac{1}{\epsilon^2}\right)$ queries and $poly\left(\frac{1}{\epsilon}\right)$ time implies $P \neq NP$.

Many more results in the paper...
Constructive Separation of P ≠ PSPACE

**Thm:** If \( P \neq PSPACE \) then there is a \( P \)-constructive separation of \( P \neq PSPACE \)

**Proof Idea:** Instead of SAT, look at TQBF (true quant. Boolean formulas)

Getting a refuter for an \( n^k \)-time algorithm \( A \) trying to decide TQBF:

\( R_A(1^n) \): Construct a formula \( F_n \) encoding the property:

\[
(\exists QBF \text{ "}(Qx)G(x)\text{"}, |G| = n)
\]

[either \( A(\forall x)G(x) \neq A(G(0)) \land A(G(1)) \)

or \( A(\exists x)G(x) \neq A(G(0)) \lor A(G(1)) \)]

Cases: (1) \( A(F_n) = \text{"false"} \). Then output \( F_n \)

(2) \( A(F_n) = \text{"true"} \). Try using search-to-decision to find "\((Qx)G(x)\)". Either this fails (in which case you get 3 bad formulas) or it succeeds, but then one of \((Qx)G(x), G(1), G(0)\) must be wrong!

Can report set of 3 as "bad". (Can also modify \( R_A \) to only report one)
Making One-Tape TM Lower Bounds Constructive?

[Maass 84] **PALINDROMES ∉ NTIME1[o(n²)].**

**Thm:** Making Maass’ lower bound $P$-constructive implies that $E$ contains functions of circuit complexity $> 2^{\epsilon n}$ for some $\epsilon > 0$

**Proof Sketch:** We make a nondeterministic one-tape TM $M$ that takes $N^{1+O(\epsilon)}$ time and correctly decides all palindromes of length $N = 2^n$ with circuit complexity $< 2^{\epsilon n}$.

Therefore, any $P$ algorithm that (on $1^n$) prints a “bad” input for $M$ must print a **hard function**. (The consequence follows by padding.)

$M$ guesses a circuit $C$ of size $N^\epsilon$ encoding the input $x$, and guesses $|x|$. “Dragging along” the description of $C$ as it reads the bits of $x$, it verifies the truth table of $C$ equals $x$. It verifies $x$ is a palindrome by using $C$ to check that the bits of the 2nd half of $x$ match the 1st half, and it uses $|x|$ to determine when to start checking the 2nd half.
Some Open Questions

• Relativization tells us that “direct” diagonalization-based proof methods are limited...
  But any major separation against $P$ (or $BPP$, or $ZPP$) will require a key property of diagonalization-based proofs: 
  *the ability to efficiently produce “bad” inputs.*
  What other methods of proof could yield this consequence?

• What about separations against LOGSPACE? NC? ACC0? E.g. Does $NP \neq LOGSPACE$ imply a constructive separation?

• Are there *equivalences* between “derandomization” and constructive separations based on the probabilistic method?
Thank you!