CMPUT 675: Computational Complexity Theory **Winter 2019**

Lecture 7 (Jan 29): Alternating Turing Machines

Lecturer: Zachary Friggstad *Scribe: Zach Goldthorpe*

7.1 Read-Once Verifiers

In the previous lecture, we showed that the language

$$\textsc{Dirpath} := \{(G, s, t) : G \text{ is a directed graph with an } s\text{-}t \text{ path}\}$$

was **NL**-complete with respect to implicit log-space reductions. We can in fact show that $\textsc{Dirpath} \in$ co-**NL**, which by its **NL**-completeness would imply that **NL** = co-**NL**. In order to do so, we provide an equivalent reformulation of **NL** via a logarithmic-space analogue of **NP** verifiers.

Definition 1 *A* **read-once verifier** *is a verifier M (a Turing machine with two read-only input tapes) such that the head of the second input tape cannot traverse leftwards.*

We can view the second input tape as an interaction between the verifier and a "prover," who through this dialogue aims to convince the verifier that the input does lie in the language. Due to the memory constraints of the language class **NL**, the constraint that the second tape can only be read as a stream restricts a logarithmic-space verifier from memorising the entirety of any certificate.

Theorem 1 *A langauge is in* **NL** *iff it can be decided by a read-once logarithmic-space verifier.*

Proof. Suppose $L \in$ **NL**, then we have a non-deterministic Turing machine M that can decide L using logarithmic space. We can emulate the computation path of M accepting an input with a read-once verifier M', provided that the certificate describes which transition function of M to follow. Explicitly, M' given input (x, y) will simulate a specific computation path of M on x where at step i of the computation, M' will simulate M following transition function δ_b if the ith bit of y is b. M' will accept (x, y) iff this simulation of M on x leads to an accepting state, and will otherwise reject.

Since M used logarithmic space, its simulation by M' will also consume only logarithmic space. Further, as each bit of y is read at most once, and in the order provided, M' is indeed a read-once verifier. If $x \in L$, then there will be some computation path where $M(x)$ accepts, so there exists a string y of which transition function in M to follow which leads $M'(x, y)$ to accept; conversely, if $M'(x, y)$ accepts, then by following the transition functions of M according to y, the computation of M on x leads to an accepting state, meaning $x \in L$.

Conversely, suppose L can be decided with a read-once logarithmic-space verifier M', then we can non-deterministically check all certificates for an input on M' with a nondeterministic Turing machine M. Explicitly, M given input x will simulate M' on input x. Whenever M' checks the head of its second tape, the two transition functions of M will simulate this head pointing to a 0 and a 1, separately. Have M then accept the input x iff one of these simulations of M' leads M' to an accepting state.

As M' was a read-once verifier, the nondeterministic Turing machine M does not have to memorise previous bits of the second tape and can always make an arbitrary choice for the next queried bit. If $x \in L$, then for some certificate y, we have $M'(x, y) = \textsc{accept}$, meaning for some computation path of M on x do we see M

arrive at its accepting state. Conversely, if some computation path of M on x leads to an accepting state, then by collecting the bits used in simulating M' along this computation path to form the certificate y, we have that $M'(x, y) = \text{ACCEPT}$, meaning $x \in L$. ■

Using this equivalent view of **NL**, we can finally show the following:

Theorem 2 DIRPATH $\in$ co-**NL**.

Proof. Let (G, s, t) be a problem instance encoding a graph G with two nodes s and t. We implicitly define the read-once verifier witnessing $\overline{\text{DIRPATH}} \in \mathbf{NL}$ by describing the appropriate certificate that proves there is no s-t path in G.

Let $v_1, \ldots, v_n$ enumerate the nodes of G, then for $0 \le i \le n$ define R_i to be the set of all distinct vertices v_j in G reachable from s in at most i steps. The verifier will know that $R_0 = \{s\}$ by definition, so the certificate will gradually convince the verifier of the contents of each R_i up until R_n, where the verifier can then check $t \notin R_n$ to be convinced that there is no s-t path in G. We do so in two stages:

Step 1: Given that the verifier knows $|R_i|$, we can construct a read-once certificate for which the verifier can be convinced of an enumeration of the elements in R_i.

The certificate will present to the verifier each $v_j \in R_i$ in order of strictly increasing index j, following each node with a proof that $v_j \in R_i$. The verifier can check using logarithmic space that the indices are indeed strictly increasing, so this will convince the verifier that all nodes presented are distinct.

For each $v_j \in R_i$ presented, the certificate will be followed by an s-v_j path using at most i edges. The verifier can easily trace this path and count the number of edges in logarithmic space, and can also confirm that each edge is indeed an edge of G by querying the encoding of G.

Thus, the verifier can be convinced of an enumeration of R_i by confirming each presented v_j is in R_i and then counting the number of vertices to ensure that it is exactly $|R_i|$.

Step 2: Given that the verifier knows $|R_i|$ for $i < n$, we can construct a read-once certificate for which the verifier can be convinced of $|R_{i+1}|$.

The certificate will present to the verifier each node v_j in order by its index j, following each node with either a proof that $v_j \in R_{i+1}$, or a proof that $v_j \notin R_{i+1}$. The verifier can then count the nodes $v_j \in R_{i+1}$ and be convinced of $|R_{i+1}|$.

For each $v_j \in R_{i+1}$, the certificate need only present an s-v_j path using at most $(i+1)$ edges, as before. If instead $v_j \notin R_{i+1}$, then the certificate will present an enumeration of each $v_k \in R_i$ as per step 1, then the verifier can confirm that (v_k, v_j) is not an edge in G. This will convince the verifier that $v_j \notin R_{i+1}$.

Following these steps inductively given that $R_0 = \{s\}$ is known a priori, the verifier can be convinced of $|R_n|$ with a read-once certificate. After this, as per step 1, the certificate can just enumerate R_n and the verifier can ensure that $t \notin R_n$, convincing the verifier that there is no s-t path. ■

Corollary 1 **NL** $=$ co-**NL**.

For any language L decidable by a read-once verifier M in $O(f(n))$-space for some time-computable $f(n) \in \Omega(\log n)$, we can step through the same proof as above but considering the configuration graph for M on any input x to create a read-once verifier M' which uses $O(f(n))$-space to decide if $x \notin L$. This modification then proves:

Theorem 3 (Savitch, 1980) *For any space-constructible function $f(n) \in \Omega(\log n)$,*

$$\mathbf{NSPACE}(f(n)) = \text{co-}\mathbf{NSPACE}(f(n))$$

7.2 Polynomial-Time Hierarchy

Consider the language

$$\textsc{Exact-Ind-Set} := \{(G,k) : \text{graph } G \text{ has a maximum independent set of size exactly } k\}$$

We know that the larger language Ind-Set of all pairs (G,k) where G is a graph with an independent set of size at least k is **NP**-complete, but is Exact-Ind-Set $\in$ **NP**? Is it in co-**NP**? The language relies on two quantifiers:

$$\exists \mathcal{I} \text{ independent set} \, \forall \mathcal{I}' \text{ independent set} : |\mathcal{I}| = k, |\mathcal{I}'| \leq k$$

The universal quantifier makes a certificate to see Exact-Ind-Set $\in$ **NP** hard to create: how do we prove that there is no independent set of size strictly larger than k? Similarly, the existential quantifier makes a certificate to see Exact-Ind-Set $\in$ co-**NP** hard to create: how do we prove that there is no independent set of size at least k? That Exact-Ind-Set lies in **NP** or co-**NP** are both open problems, but theorists conjecture that neither is the case.

Nonetheless, if **P** = **NP**, then in fact Exact-Ind-Set $\in$ **P**: given the language Ind-Set would be polynomial-time solvable, we can determine in polynomial time if a graph G does not have an independent set of size at least $(k+1)$ while also determining that it does have an independent set of size at least k. These would imply that the maximum independent set of G has size precisely k.

Similarly, consider the language

$$\text{EQ-CNF} := \{(\phi,k) : \phi \text{ is a CNF with an equivalent CNF } \psi \text{ where } |\psi| \leq k\}$$

This language does not seem to be in **NP** since checking equivalence of two CNF's on n variables seems to require checking exponentially-many different variable assignments. The language also does not seem to be in co-**NP** because there is no clear way to certify that an equivalent CNF to ϕ of length at most k cannot exist. Again, both problems are open, and EQ-CNF relies on a similar sentence with two quantifiers:

$$\exists \psi \text{ CNF with } |\psi| \leq k \;\; \forall x : \phi(x) = \psi(x)$$

If **P** = **NP**, then again EQ-CNF $\in$ **P**: since the language SAT would become polynomial-time solvable, we can verify that two CNF's are equivalent by checking if $\overline{\phi \leftrightarrow \psi}$ is satisfiable (which is iff ϕ and ψ are inequivalent), and the existence of an equivalent CNF ψ for ϕ with $|\psi| \leq k$ becomes a problem in **NP**, which is then a problem in **P**.

We can even refine the above language and consider

$$\textsc{Exact-EQ-CNF} := \{(\phi,k) : \phi \text{ is a CNF where the smallest equivalent CNF } \psi \text{ has } |\psi| = k\}$$

which relies on a sentence with four quantifiers:

$$\exists \psi \text{ CNF with } |\psi| = k \;\; \forall \psi' \text{ CNF with } |\psi'| < k \;\; \exists x \, \forall y : \phi(y) = \psi(y) \text{ and } \phi(x) \neq \psi'(x)$$

This problem seems to lie in an even larger class than EQ-CNF, but if **P** = **NP**, then Exact-EQ-CNF $\in$ **P**: we have seen that we would have EQ-CNF $\in$ **P**, so we can then determine in polynomial time both that ϕ has an equivalent CNF ψ with $|\psi| \leq k$ and that ϕ does not have an equivalent CNF ψ with $|\psi| \leq k-1$. This would decide in polynomial time if the minimum equivalent CNF for ϕ has size exactly k.

Despite how these classes of languages are seemingly larger than both **NP** and co-**NP**, they all seem to collapse if **P** = **NP**. To explore the extent to which this is the case, we define a new family of complexity classes.

Definition 2 *For $k \geq 1$, let $\Sigma_k^{\mathbf{P}}$ denote the complexity class of languages L where there exists a polynomial-time Turing machine M taking $(k+1)$ inputs such that*

$$x \in L \iff \exists u_1 \, \forall u_2 \, \exists u_3 \, \ldots \mathsf{Q} u_k : M(x, u_1, \ldots, u_k) = \text{ACCEPT}$$

Q *will be $\forall$ if k is even and $\exists$ otherwise.*

For example, the complexity class $\Sigma_1^{\mathbf{P}}$ collects all languages L where there exists a polynomial-time Turing machine M taking 2 inputs such that $x \in L$ iff $\exists y : M(x, y) = \text{ACCEPT}$, which is precisely the class **NP**.

Dually, we also introduce the classes $\Pi_k^{\mathbf{P}} := \text{co-}\Sigma_k^{\mathbf{P}}$ for $k \geq 1$ of all languages L for which there exists a polynomial-time Turing machine M taking $(k+1)$ inputs so that $x \in L$ iff $\forall u_1 \, \exists u_2 \, \ldots \mathsf{Q} u_k : M(x, u_1, \ldots, u_k) = \text{ACCEPT}$. From these definitions, it is relatively straightforward to see that $\Sigma_k^{\mathbf{P}} \subseteq \Pi_{k+1}^{\mathbf{P}}$ and likewise that $\Pi_k^{\mathbf{P}} \subseteq \Sigma_{k+1}^{\mathbf{P}}$.

Indeed, suppose $L \in \Sigma_k^{\mathbf{P}}$ as witnessed by the Turing machine M that takes $(k+1)$ inputs. By modifying M to take $(k+2)$ inputs where it completely ignores the new input u_0, we have a polynomial-time Turing machine which accepts a string x iff $\forall u_0 \, \exists u_1 \, \ldots \mathsf{Q} u_k : M'(x, u_0, \ldots, u_k) = \text{ACCEPT}$, showing $L \in \Pi_{k+1}^{\mathbf{P}}$. The argument is analogous for seeing that $\Pi_k^{\mathbf{P}} \subseteq \Sigma_{k+1}^{\mathbf{P}}$.

Therefore, we can define the polynomial-time hierarchy complexity class using either of these families:

Definition 3 $\mathbf{PH} := \bigcup_{k \geq 1} \Sigma_k^{\mathbf{P}} = \bigcup_{k \geq 1} \Pi_k^{\mathbf{P}}$.

Note that $\mathbf{PH} \subseteq \mathbf{PSPACE}$: if $L \in \mathbf{PH}$, then $L \in \Sigma_k^{\mathbf{P}}$ for some k, which means we have a polynomial-time Turing machine M such that $x \in L$ iff $\exists u_1 \, \forall u_2 \ldots \mathsf{Q} u_k : M(x, u_1, \ldots, u_k) = \text{ACCEPT}$. We can then recursively try to populate each variable $u_1, \ldots, u_k$ with candidate strings—which are at most polynomial in $|x|$—and then simulate $M(x, u_1, \ldots, u_k)$. As k is fixed and M is polynomial-time computable, this will only use polynomial space in $|x|$. Therefore, $L \in \mathbf{PSPACE}$. However, the question of whether $\mathbf{PH} = \mathbf{PSPACE}$ remains open.

Theorem 4 *If $\mathbf{P} = \mathbf{NP}$, then $\mathbf{P} = \mathbf{PH}$.*

The proof of the above theorem follows from the theorem below by recalling that if $\mathbf{P} = \mathbf{NP}$, then $\mathbf{NP} = \text{co-}\mathbf{NP}$, and since $\mathbf{NP} = \Sigma_1^{\mathbf{P}}$, this would mean $\Sigma_1^{\mathbf{P}} = \Pi_1^{\mathbf{P}}$ and thus $\mathbf{PH} = \Sigma_1^{\mathbf{P}} = \mathbf{NP} = \mathbf{P}$.

Theorem 5 *If $\Sigma_k^{\mathbf{P}} = \Pi_k^{\mathbf{P}}$ for some $k \geq 1$, then $\mathbf{PH} = \Sigma_k^{\mathbf{P}}$.*

Proof. We proceed by induction on $i \geq k$ to show that $\Sigma_i^{\mathbf{P}} \cup \Pi_i^{\mathbf{P}} \subseteq \Sigma_k^{\mathbf{P}}$. Note that this is already true for $i < k$ since $\Sigma_i^{\mathbf{P}} \subseteq \Pi_{i+1}^{\mathbf{P}}$ and $\Pi_i^{\mathbf{P}} \subseteq \Sigma_{i+1}^{\mathbf{P}}$.

The claim when $i = k$ holds by assumption, so suppose $\Sigma_i^{\mathbf{P}} \cup \Pi_i^{\mathbf{P}} \subseteq \Sigma_k^{\mathbf{P}}$ for some $i \geq k$. Let $L \in \Sigma_{i+1}^{\mathbf{P}}$, then we have a polynomial-time Turing machine M such that $x \in L$ iff $\exists u_1 \, \forall u_2 \, \ldots \mathsf{Q} u_{i+1} : M(x, u_1, \ldots, u_{i+1}) = \text{ACCEPT}$. Consider now the language

$$L' := \{(x, u_1) : \forall u_2 \, \exists u_3 \, \ldots \mathsf{Q} u_{i+1}, M(x, u_1, \ldots, u_{i+1}) = \text{ACCEPT}\}$$

then $L' \in \Pi_i^{\mathbf{P}} \subseteq \Sigma_k^{\mathbf{P}}$. Thus, we have a polynomial-time Turing machine M' such that $(x, u_1) \in L'$ iff $\exists v_1 \, \forall v_2 \, \ldots \mathsf{Q} v_k : M'(x, u_1, v_1, \ldots, v_k) = \text{ACCEPT}$. Therefore,

$$\begin{aligned} x \in L &\iff \exists u_1 : (x, u_1) \in L' \iff \exists u_1 \, \exists v_1 \, \forall v_2 \, \ldots \mathsf{Q} v_k : M'(x, u_1, v_1, \ldots, v_k) = \text{ACCEPT} \\ &\iff \exists (u_1, v_1) \, \forall v_2 \, \exists v_3 \, \ldots \mathsf{Q} v_k : M'(x, u_1, v_1, \ldots, v_k) = \text{ACCEPT} \end{aligned}$$

showing that in fact $L \in \Sigma_k^{\mathbf{P}}$ since by merging u_1 and v_1 we can take M' to be a Turing machine with $(k+1)$ inputs rather than $(k+2)$. Therefore, $\Sigma_i^{\mathbf{P}} \subseteq \Sigma_k^{\mathbf{P}}$. Dually, if $L \in \Pi_{i+1}^{\mathbf{P}} = \text{co-}\Sigma_{i+1}^{\mathbf{P}}$, then $\bar{L} \in \Sigma_{i+1}^{\mathbf{P}} \subseteq \Sigma_k^{\mathbf{P}} = \Pi_k^{\mathbf{P}}$, meaning $L = \bar{\bar{L}} \in \text{co-}\Pi_k^{\mathbf{P}} = \Sigma_k^{\mathbf{P}}$. Therefore, $\Sigma_{i+1}^{\mathbf{P}} \cup \Pi_{i+1}^{\mathbf{P}} \subseteq \Sigma_k^{\mathbf{P}}$.

By induction, it follows that $\Sigma_i^{\mathbf{P}} \subseteq \Sigma_k^{\mathbf{P}}$ for every $i \geq 1$, and so $\Sigma_k^{\mathbf{P}} \subseteq \mathbf{PH} = \bigcup_{i \geq 1} \Sigma_i^{\mathbf{P}} \subseteq \Sigma_k^{\mathbf{P}}$. ■

The above theorems demonstrate in particular that if $\mathbf{P} = \mathbf{NP}$, then all of the layers of the polynomial hierarchy collapse to the same single complexity class $\mathbf{P}$.

7.2.1 PH-completeness

As usual, define a language L to be **PH**-complete if $L \in \mathbf{PH}$ and any language in **PH** is Karp-reducible in polynomial time to L. It turns out the very existence of **PH**-complete languages would also collapse part of the polynomial hierarchy:

Theorem 6 *If there is a* **PH***-complete language, then* $\mathbf{PH} = \Sigma_k^{\mathbf{P}}$ *for some* $k \geq 1$.

Proof. Suppose L is **PH**-complete, then since $L \in \mathbf{PH}$, then $L \in \Sigma_k^{\mathbf{P}}$ for some $k \geq 1$. Let M be a polynomial-time Turing machine sucvh that $x \in L$ iff $\exists u_1 \forall u_2 \ldots \mathsf{Q} u_k : M(x, u_1, \ldots, u_k) = \text{ACCEPT}$. Let $L' \in \mathbf{PH}$ be arbitrary now, then since $L' \leq_{\mathbf{P}} L$, we have a polynomial-time computible function $f(n)$ such that $x \in L' \iff f(x) \in L$. Therefore, $x \in L'$ iff $\exists u_1 \forall u_2 \ldots \mathsf{Q} u_k : M(f(x), u_1, \ldots, u_k) = \text{ACCEPT}$, showing that $L' \in \Sigma_k^{\mathbf{P}}$. ■

Therefore, it is unlikely that **PH**-complete languages exist. This also provides evidence suggesting $\mathbf{PH} \neq \mathbf{PSPACE}$, as otherwise the **PSPACE**-complete language TQBF would also be complete for **PH** and the **PH** then collapses.

However, for each layer $\Sigma_k^{\mathbf{P}}$, there do exist $\Sigma_k^{\mathbf{P}}$-complete languages with respect to polynomial-time Karp reductions:

Theorem 7 *The language*

$$\Sigma_k\text{-SAT} := \{\phi \text{ } \textit{CNF} : \exists u_1 \forall u_2 \exists u_3 \ldots \mathsf{Q} u_k, \phi(u_1, \ldots, u_k) = \text{TRUE}\}$$

is $\Sigma_k^{\mathbf{P}}$*-complete.*

Proof. The proof is analogous to the proof of the Cook-Levin theorem. ■

7.3 Alternating Turing Machines

Definition 4 *An* **alternating Turing machine** *is a Turing machine with two transition functions* δ_0, δ_1 *where each state* q *other than the halting states* $q_{\text{ACCEPT}}, q_{\text{REJECT}}$ *have a quantifier* $\mathsf{Q}_q \in \{\forall, \exists\}$.

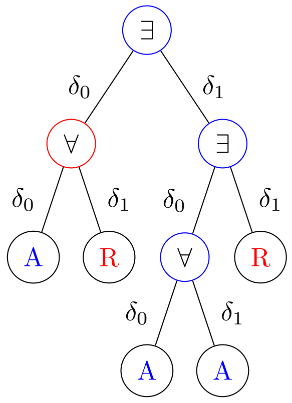
Given an input x into an alternating Turing machine, consider its computation tree of all possible combinations of transition functions that can be followed with the input x. To define when the Turing machine accepts x, we define temporarily the notion of "eventually accepting" inductively on this tree:

- the leaf nodes that are the accepting state q_{ACCEPT} are "eventually accepting"
- a non-leaf node q with $\mathsf{Q}_q = \forall$ is "eventually accepting" if every child node is "eventually accepting"

- a non-leaf node q with $\mathsf{Q}_q = \exists$ is "eventually accepting" if at least one child node is "eventually accepting"

The alternating Turing machine then accepts x iff the starting state q_{START} is "eventually accepting."

For example, in the computation tree below (where 'A' denotes ACCEPT and 'R' denotes REJECT), the blue nodes are the "eventually accepting" states, and the input is accepted because the topmost node is "eventually accepting."



Note that alternating Turing machines generalise nondeterministic Turing machines, as the latter can be considered as an alternating Turing machine where every non-halting state is quantified by $\exists$.

We can now construct a new family of complexity classes based on alternating Turing machines:

Definition 5 *For a function $f : \mathbb{N} \to \mathbb{N}$, let* $\mathbf{ATIME}(f(n))$ *be the class of languages L such that there exists an alternating Turing machine M such that M decides L, and when given input x, the machine M will halt in $O(f(|x|))$ many steps along any path in the computation tree for M on x.*

Example 1 *The class* $\mathbf{AP} := \bigcup_{c \geq 1} \mathbf{ATIME}(n^c)$ *is exactly* $\mathbf{PSPACE}$.

Proof. Just as in the argument that $\mathbf{PH} \subseteq \mathbf{PSPACE}$, we can simulate a polynomial-time alternating Turing machine with a polynomial-space Turing machine by recursing on each choice of transition function, and then can determine if a node is "eventually accepting" recursively as per the definition.

Conversely, $\mathbf{PSPACE} \subseteq \mathbf{AP}$ because the $\mathbf{PSPACE}$-complete language TQBF of true quantified Boolean formulae can be solved by an alternating Turing machine which for each quantified variable $\mathsf{Q}x$ in a TQBF instance will nondeterministically try both assignments of x, and will quantify the state which tries both assignments with Q. ■

7.4 Lower Bounds on Solving SAT

Definition 6 *For a pair of functions $t, s : \mathbb{N} \to \mathbb{N}$, define the complexity class* $\mathbf{TISP}(t(n), s(n))$ *to collect all languages that are decidable by an ordinary Turing machine which uses $O(s(|x|))$ space and runs in $O(t(|x|))$ time on any input x.*

Theorem 8 *For any $a \geq 1$ and $b > 0$ satisfying $a\left(\frac{a+b}{2}\right) < 1$, then* $\mathrm{SAT} \notin \mathbf{TISP}(n^a, n^b)$.

We do not present a proof in this lecture, but we will describe the main ideas of the proof here. To do so, we introduce yet another family of complexity classes:

Definition 7 *For $k \geq 1$ and a function $f : \mathbb{N} \to \mathbb{N}$, let $\Sigma_k\textbf{TIME}(f(n))$ be the class of all languages decidable by an alternating Turing machine M such that on any computation path for an input x:*

- *M runs in $O(f(|x|))$-time along this path*
- *the quantifiers on the nodes along this path swap at most $(k-1)$ times*
- *the quantifier of the starting node is $\exists$*

Also, we need a refinement of the Cook-Levin theorem, which provides a remarkably efficient reduction from any language in $\textbf{NTIME}(n)$ to SAT:

Theorem 9 *For a language $L \in \textbf{NTIME}(n)$, we have $L \leq_{\mathbf{P}}$ SAT via a reduction f satisfying for any input x*

1. *$|f(x)| \in O(|x| \log |x|)$*
2. *After $O(|x|)$ time and $O(\log |x|)$ space in preprocessing, any individual bit $f(x)_i$ can be determined in $O(\log |x|)$ time and space*

Proof. Omitted. ■

Here, "preprocessing" refers to checks such as finding the length of the input x, et cetera.

The proof of theorem 8, then proceeds in two steps:

1. First we will show $\textbf{NTIME}(n) \not\subseteq \textbf{TISP}(n^a, n^b)$.
2. Then, we use the efficient reductions from languages in $\textbf{NTIME}(n)$ to SAT to deduce SAT $\notin \textbf{TISP}(n^a, n^b)$. The crux of the argument is that if SAT can be solved by a Turing machine M which is constrained by $O(n^a)$ time and $O(n^b)$ space, then we can show $\textbf{NTIME}(n) \subseteq \textbf{TISP}(n^a, n^b)$ also. Indeed, for any $L \in \textbf{NTIME}(n)$, let $f : \mathbb{N} \to \mathbb{N}$ be an efficient reduction from L to SAT as per theorem 9. For any instance string x, we have after $O(|x|)$ time and $O(\log |x|)$ space in preprocessing—both which fit in the constraints of $\textbf{TISP}(n^a, n^b)$—we can run $M(f(x))$ in $O(|f(x)|^a)$ time and $O(|f(x)|^b)$ space by using an additional $O(\log |x|)$ time and space whenever a bit of $f(x)$ is queried. While this does not exactly fit in $\textbf{TISP}(n^a, n^b)$, we can adjust the constants a, b thanks to the slackness of the constraint $a\left(\frac{a+b}{2}\right) < 1$ to account for the logarithmic factors.