“Search-to-Decision” Reductions

Suppose that $P = NP$. That would mean that all $NP$ languages can be decided in deterministic polytime. For example, given a graph, we could decide in deterministic polytime whether that graph is 3-colorable. But could we find an actual 3-coloring? It turns out that yes, we can.

In general, we can define an $NP$ search problem: Given a polytime relation $R$, a constant $c$, and a string $x$, find a string $y$, $|y| \leq |x|^c$, such that $R(x, y)$ is true, if such a $y$ exists. As the following theorem shows, if $P = NP$, then every $NP$ search problem can also be solved in deterministic polytime.

**Theorem 1.** If $NP = P$, then there is a deterministic polytime algorithm that, given a formula $\phi(y_1, \ldots, y_n)$, finds a satisfying assignment to $\phi$, if such an assignment exists.

**Proof.** We use a kind of binary search to look for a satisfying assignment to $\phi$. First, we check if $\phi(x_1, \ldots, x_n) \in SAT$. Since we assumed that $P = NP$, this can be done in deterministic polytime. Then we check if $\phi(0, x_2, \ldots, x_n) \in SAT$, i.e., if $\phi$ with $x_1$ set to False is still satisfiable. If it is, then we set $a_1$ to be 0; otherwise, we make $a_1 = 1$. In the next step, we check if $\phi(a_1, 0, x_3, \ldots, x_n) \in SAT$. If it is, we set $a_2 = 0$; otherwise, we set $a_2 = 1$. We continue this way for $n$ steps. By the end, we have a complete assignment $a_1, \ldots, a_n$ to variables $x_1, \ldots, x_n$, and by construction, this assignment must be satisfying.

The amount of time our algorithm takes is polynomial in the size of $\phi$: we have $n$ steps, where at each step we must answer a SAT question. Since, by our assumption, $P = NP$, each step takes polytime.

Theorem 1 shows the true importance of proving that $NP = P$. If $NP = P$, we could efficiently generate a correct solution for any problem with an efficient recognition algorithm for correct solution. For instance, if $P = NP$, then we could efficiently find a login password of any user of a network, since checking if a password matches a login name can be done efficiently. Thus, if $P = NP$, essentially any secret could be found out efficiently.
Remark 2. Consider the language

$$\text{Composite} = \{N \mid \text{some prime } p < N \text{ divides } N\}.$$  

This language is clearly in $\text{NP}$. Moreover, there is a known deterministic polytime algorithm for this problem (as Primality Testing is in $\text{P}$.)

The corresponding Search-version is basically the Factoring problem: Given $N$, find its nontrivial prime factor.

If there were a polytime “search-to-decision” reduction for this problem, we would get a polytime algorithm for factoring integers! However, no such algorithm is currently known (and conjectured not to exist).

2 Levin’s Universal Search

Suppose we are told that $\text{SAT} \in \text{P}$, yet we are not given an actual polytime algorithm for $\text{SAT}$. Can we still solve $\text{SAT}$ in polytime, without knowing the actual algorithm, but just knowing that it exists? Surprisingly, the answer is Yes! We can use a certain universal $\text{SAT}$ algorithm, based on Levin’s Universal Search algorithm (for inverting one-way functions).

First, suppose that $\text{SAT} \in \text{TIME}(n^c)$. Also suppose that we know $c$. (If we don’t know $c$, the actual polytime bound on solving $\text{SAT}$, we can still solve $\text{SAT}$ ourselves, but it will take slightly more than polynomial time.)

Theorem 3. Suppose $\text{SAT} \in \text{TIME}(n^c)$ for a known constant $c > 0$, but an unknown algorithm (solving $\text{SAT}$ in time $O(n^c)$). Then one can find an explicit polytime algorithm that solves $\text{SAT}$ in time $O(n^{c+2} \cdot \log^2 n + t_0(n) \cdot n)$, where $t_0(n) \in \text{poly}(n)$ is the time it takes to check if a given assignment satisfies a given $\text{SAT}$ instance of size $n$.

Proof. By the “Search-to-Decision” reduction for $\text{SAT}$, we know that $\text{SAT}$-Search is solved in time $O(n^{c+1})$. That is, there is a Turing machine, running in time $O(n^{c+1})$, that on a given $\text{SAT}$ instance $\phi$ of size $n$, either finds a satisfying assignment for $\phi$, or decides that $\phi$ is unsatisfiable. (Note that we only know that such a TM exists. We don’t know the actual TM, as it relies on the decision algorithm for $\text{SAT}$ that is not given to us.)

Here’s an algorithm to solve $\text{SAT}$:

“On input $\phi$ of size $n$,
1. for $i = 1$ to $n$
2. simulate TM $M_i$ on $\phi$ for $O(n^{c+1} \cdot \log^2 n)$ steps;
3. if $M_i$ produces a satisfying assignment for $\phi$, then Accept (and halt)
4. endfor
5. Reject”

For the analysis, first observe that the described algorithm never accepts an unsatisfiable formula. On the other hand, if given a satisfiable formula $\phi$ of size $n$, the algorithm will eventually simulate the TM for $\text{SAT}$-Search (which we know exists) that runs in time $n^{c+1}$.  

2
Let $M_d$ be the TM solving SAT-Search in time $n^{c+1}$. The simulation time of this $M_d$ on a universal TM (used by our algorithm) requires time $d \cdot n^{c+1} \cdot (c+1) \log n$. When $i = d$, this TM $M_d$ will be simulated on $\phi$ for the amount of time which is bigger than $d \cdot n^{c+1} \cdot (c+1) \log n$ for $n$ large enough (for $n > d$). Thus, for large enough $n$, our algorithm will discover a satisfying assignment for $\phi$ and correctly accept. This shows that the described algorithm is correct.

For the running time analysis, we simulate $n$ TMs for $O(n^{c+1} \cdot \log^2 n)$ time each, so the total time our algorithm takes is $O(n^{c+2} \cdot \log^2 n) + O(n \cdot t_0(n))$ for checking if any of the $n$ possible strings (produced by the TMs $M_i$) is a satisfying assignment.

**Remark 4.** The described universal algorithm for SAT is good theoretically: it runs only slightly slower than the assumed fastest algorithm for SAT. However, this algorithm is not very practical as it starts to work only for very large inputs sizes $n \gg d$, where $d$ is the index of a correct SAT algorithm. Presumably, a fast algorithm for SAT (if it exists at all!) would be quite complex and long, and so its index $d$ may be a huge constant (exponential in the description size of the program for $M_d$)!

## 3 Motivation: Lower bounds for SAT

Even though it is widely believed that $\text{NP} \neq \text{P}$, and so that SAT is not in $\text{P}$, we are so far unable to prove that SAT requires time $n^2$, or even that SAT requires time $n^{1.1}$.

What if we impose an additional requirement of small space? For proper functions $t, s : \mathbb{N} \rightarrow \mathbb{N}$, define the class $\text{TISP}(t, s)$ (for simultaneous Time and Space) to contain exactly those languages $L$ such that some TM $M$ decides $L$ in time at most $t$ and space at most $s$.

With the extra restriction, we are able to prove the following time-space tradeoff for SAT:

**Theorem 5** (Fortnow). $\text{SAT} \not\in \text{TISP}(n^{1.1}, n^{0.1})$.

That is, if we restrict our attention to algorithms using space at most $n^{0.1}$, we get that any such algorithm solving SAT would need to use strictly more time than $n^{1.1}$. (Equivalently, if we consider algorithms running in time at most $n^{1.1}$, we get that any such algorithm solving SAT would have to use more than $n^{0.1}$ space.)

The proof of this result requires the concept of alternating Turing machines, which generalize $\text{NP}$-machines and $\text{coNP}$-machines by allowing alternating “existential” and “universal” guesses. We explain this next.

## 4 Polynomial-Time Hierarchy

Recall that a language $L \in \text{NP}$ can be described by the formula: $x \in L$ iff $\exists$ (short $y$) $R(x, y)$, where $y$ is of length polynomial in the length of $x$, and $R$ is a polytime predicate.

Similarly, a language $L \in \text{coNP}$ can be described by the formula: $x \in L$ iff $\forall$ (short $y$) $R(x, y)$, where $y$ is of length polynomial in the length of $x$, and $R$ is a polytime predicate.
What happens if we allow some $k$ alternating quantifiers over short strings? We get the $k$th level of the polynomial-time hierarchy!

We call a $k$-ary relation $R$ *polynomially balanced* if, for every tuple $(a_1, \ldots, a_k) \in R$, the lengths of all $a_i$’s are polynomially related to each other.

**Definition 6.** For any $i \geq 1$, a language $L \in \Sigma_i^p$ iff there is a polynomially balanced $(i + 1)$-ary relation $R$ such that

$$L = \{ x \mid \exists y_1 \forall y_2 \exists y_3 \ldots Q_i y_i R(x, y_1, \ldots, y_i) \}.$$  

Here, $Q_i$ is $\exists$ if $i$ is odd, and $\forall$ if $i$ is even.

For example, $\Sigma_1^p = \text{NP}$.

**Definition 7.** For any $i \geq 1$, a language $L \in \Pi_i^p$ iff there is a polynomially balanced $(i + 1)$-ary relation $R$ such that

$$L = \{ x \mid \forall y_1 \exists y_2 \forall y_3 \ldots Q_i y_i R(x, y_1, \ldots, y_i) \}.$$  

For example, $\Pi_1^p = \text{coNP}$.

Note that, in general, for every $i$, $\Pi_i^p = \text{co}\Sigma_i^p$.

**Definition 8.** $PH = \bigcup_{i \geq 0} \Sigma_i^p$.

**Theorem 9.** $PH \subseteq \text{PSPACE}$

*Proof.* Recall that $\text{NP} \subseteq \text{PSPACE}$ since we can just enumerate (re-using space) over all candidate witnesses, and check if any one of them is valid. The case of $PH \subseteq \text{PSPACE}$ is a generalization of this idea. (Exercise!)

4.1 **Examples of problems in PH**

Unique-SAT = \{ $\phi$ | $\phi$ is a formula with exactly one satisfying assignment \}

**Theorem 10.** Unique-SAT is in $\Sigma_2^p$.

*Proof.* Note that $\phi \in \text{Unique} – \text{SAT}$ iff there is $y$ such that for all $z$, $z \neq y$, we have $\phi(y)$ is True and $\phi(z)$ is False.

Min-Circuit = \{ $C$ | $C$ is a Boolean circuit s.t. no smaller equivalent circuit exists \}

Here, the size of a Boolean circuit is the number of logical operations (ANDs, ORs, and NOTs), or gates, used in the circuit.

**Theorem 11.** Min-Circuit is in $\Pi_2^p$.

*Proof.* Note that $C$ is in Min-Circuit iff for every smaller circuit $C'$ there is an input $x$ such that $C(x) \neq C'(x)$.
4.2 Alternative definition of \( \text{PH} \)

**Definition 12.** An oracle TM is a TM \( M \) with special tape, called oracle tape, and special states \( q_f, q_{\text{yes}}, q_{\text{no}} \). When run with some oracle \( O \) (where \( O \) is just some language), \( M \) can query \( O \) on some strings \( x \) by writing these \( x \) onto its oracle tape, and then entering the state \( q_f \). In the next step, TM \( M \) (miraculously) finds itself in the state \( q_{\text{yes}} \) if \( x \in O \), or the state \( q_{\text{no}} \) if \( x \notin O \).

This definition of an oracle TM captures the notion of “having access to an efficient algorithm deciding \( O \)”.  

For complexity classes \( C_1 \) and \( C_2 \), we say that a language \( L \in C_1^{C_2} \) if there is an oracle TM \( M \) from class \( C_1 \) that, given oracle access to some language \( O \in C_2 \), decides \( L \).

For example, \( \text{Unique} - \text{SAT} \in \text{NP}^{\text{NP}} \). Given a formula \( \phi \), nondeterministically guess an assignment \( \alpha \). Check that \( \phi(\alpha) \) is True. If not, then Reject; otherwise, construct a new formula \( \phi'(x_1, \ldots, x_n) \equiv \phi(x_1, \ldots, x_n) \land [x_1 \ldots x_n \neq a_1 \ldots a_n] \). Ask the SAT oracle whether \( \phi' \) is satisfiable. If it is, then Reject; otherwise, Accept.

**Alternative definition of \( \text{PH} \).** Define \( \Sigma_i^p = \Pi_i^p = \text{P} \). For all \( i \geq 0 \), define \( \Sigma_{i+1}^p = \text{NP}^{\Sigma_i^p} \) and \( \Pi_{i+1}^p = \text{coNP}^{\Sigma_i^p} \). Finally, set \( \text{PH} = \cup_{i \geq 0} \Sigma_i^p \).

**Theorem 13.** The original definition and the alternative definition of \( \text{PH} \) are equivalent.

**Proof.** The base case of \( i = 0 \) is immediate: in both definitions, the 0th level is just the class \( \text{P} \).

Just for the sake of this proof, let us denote by \( \Sigma_i^1 \) and by \( \Sigma_i^2 \) the \( i \)th level of polytime hierarchy according to definitions 1 and 2, respectively. (The first definition is in terms of logical formulas; the second definition is in terms of oracle TMs.)

We need to show that \( \Sigma_i^1 = \Sigma_i^2 \), for all \( i \). The case of \( i = 0 \) is already argued. Let us assume the equivalence of the two definitions for \( i \), and prove it for \( i + 1 \).

Let us start by proving that \( \Sigma_{i+1}^1 \subseteq \Sigma_{i+1}^2 \). By definition, \( L \in \Sigma_{i+1}^1 \) iff there is a polybalanced relation \( R \) such that \( x \in L \iff \exists y_1 \forall y_2 \ldots R(x, y_1, y_2, \ldots, y_{i+1}) \). Consider the language \( L' = \{ (x, y) \mid \forall y_2 \ldots R(x, y, y_2, \ldots, y_{i+1}) \} \). It is easy to see that \( L' \in \Pi_i^1 \), and hence, by the induction hypothesis, \( L' \in \Pi_i^2 \). Now, to test if \( x \in L \) we can do the following: Nondeterministically guess a \( y \), then check if \( (x, y) \in L' \) by querying the \( \Pi_i^2 \) oracle. This algorithm shows that \( L \in \Sigma_{i+1}^2 \).

Let us now prove the other direction, i.e., that \( \Sigma_{i+1}^2 \subseteq \Sigma_{i+1}^1 \). Consider an arbitrary language \( L \in \Sigma_{i+1}^2 \). By definition, there is an \( \text{NP}^{\Sigma_i^2} \) TM \( M \) that decides \( L \). Also, we have that \( x \in L \) iff there is an accepting computation of \( M \) on \( x \).

For any input \( x \), consider a run of the TM \( M \) on \( x \). During that computation, the TM \( M \) may ask (up to a polynomial number of) oracle queries to the \( \Sigma_i^2 \) oracle. Some of these oracle queries have the answer Yes, and the others No. Note that the Yes answers can be verified in \( \Sigma_i^2 \), which is equal to \( \Sigma_i^1 \), by the inductive hypothesis. The No answers can be verified in \( \Pi_i^2 \), which is equal to \( \Pi_i^1 \), by the inductive hypothesis.
Thus, to test if \( x \in L \), we can guess (using the \( \exists \) quantifier) an accepting computation path of \( M \) on \( x \) together with all answers to the oracle queries, and check the correctness of our path, including all the answers to the oracle queries, in \( (\Sigma_i^1 \cup \Pi_i^1) \). Put together, this gives us a way to check whether \( x \in L \) by a \( \Sigma_i^{1+1} \) formula. Hence, we get \( L \in \Sigma_i^{1+1} \).

Finally, since \( \Pi_i = \text{co}\Sigma_i \) for each of the two definitions, we immediately obtain the equality \( \Pi_i^{1+1} = \Pi_i^{2+1} \).