Chapter 4

Non-uniform Complexity

Chapter outline: Polynomial circuits, P/poly, Sparse sets. Equivalence among the three. Karp-Lipton Theorem. Mahaney's Theorem.

4.1 Polynomial circuits, P/poly, Sparse sets

As discussed in Lecture 1, Computational Complexity Theory has its genesis in Computability Theory. This theory is based on Turing's notion of a universal computing machine which is specified by a strictly finite number of rules. A major accomplishment of this theory is to demonstrate the existence of undecidable problems. While this was important both mathematically and philosophically, once we absorbed the notion of a strictly finitary device attempting to compute infinitely many instances of a problem as formulated by Turing Machines, the fact that there are some computational problems for which no TM can compute correctly for all instances becomes not so surprising.

If we carry out a "delayed" diagonalization proof, one can easily produce undecidable problems much like the Halting Problem, which nevertheless have very simple combinatorial structures. For example, we can have undecidable problems for which there are at most one string per length n. We can even make it much much sparser; all we need to do is to reject all strings at length n except for the string 1^n, and only for a very sparsely distributed set of values of n's (say, $n = 2^{2^k}$), we carry out a "delayed" diagonalization (say, against the machine M_k). Combinatorially, this set is rather simple: we can, for example, represent this set at length n by $x_1 \wedge \overline{x_1}$ (when it is the empty set $\emptyset$) or $x_1 \wedge x_2 \wedge \ldots \wedge x_n$ (when it consists of just 1^n). Of course which case it is for each n is undecidable computationally.

However, by a simple counting argument, most of 2^{2^n} Boolean functions from $\{0,1\}^n$ to $\{0,1\}$ do not have such simple representations. So it appears that there is a differently kind of complexity which is not accounted for by the consideration of computability in the TM model, a tiny box attempting to decide for all possible length input. Moreover, in

today's complexity community, most researchers believe that the crux of the matter as to why such problems as NP-complete problems seem to resist any polynomial time solution is not due to the lack of a single strictly finitary device attempting to compute infinitely many instances, *but rather* simply it is too "complicated" at a large length n. This notion of "complicatedness" refers to a finite number (2^n) of instances for any fixed large n. The intuitive feeling is that for these apparently hard problems such as NP-complete problems, for large n, the number of logical operations required to solve all instances of length n simply grows faster than any fixed polynomial in n, perhaps more likely an exponential function in n. Of course these are still conjectures with no proofs.

To formulate this intuition, one needs to define a different notion of "combinatorial" complexity, one which focuses attention on a finite length n, and ignores the issue of whether one can "string" them all together by a "uniform" finitary device such as a TM. This leads to the notion of non-uniform complexity.

We consider Boolean circuits. A Boolean circuit on n input bits is a directed acyclic graph (DAG) where each node is labeled by either an input variable x_i or a logical gate. We may, without loss of generality, assume these gates are $\wedge, \vee, \neg$ as they already form a complete logical basis. (Actually $\wedge, \neg$, or $\vee, \neg$ already do, but usually we include all three.) The default assumption is that nodes labeled by either $\wedge$ or $\vee$ have indegree 2, and nodes labeled by $\neg$ have indegree 1. Sometimes we allow $\wedge, \vee$ to have arbitrary indegrees, in which case we say the circuit has unbounded fan-in. A Boolean circuit may have one or more node labelled as output gate(s). The default is a single output node. Then a Boolean circuit computes a Boolean function in the obvious sense. The size of a Boolean circuit is the number of edges (wires) in the graph. If the outdegree is 1 for every node other than the input and output node (a.k.a. an internal node) then the Boolean circuit is called a Formula. In this case the Boolean circuit is essentially a tree, except at the input level. By De Morgan's Law, one can push the $\neg$ gates to the input level of the Formula, without increasing its size (or at most by $+n$ if not all negated variables are present). In a circuit or a formula, when all negation signs appear directly on the variables, sometimes the edges connecting $\neg$ and its respective variables are not counted in size, i.e., we couunt size in terms of inputs in the literals $x_1, x_2, \ldots, x_n \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}$. This process of pushing $\neg$ gates to the input level can be carried out for Boolean circuits in general, with at most an increase of size by a factor of 2. (Replicate each node v with a copy for $\overline{v}$.) The depth of a Boolean circuit is the longest path from input nodes to output(s).

A Boolean function $f_n : \{0,1\}^n \to \{0,1\}$ is always computable by a Boolean circuit as well as a Boolean formula; A Disjunctive Normal Form (DNF) or a Conjunctive Normal Form (CNF) will do. These are of depth 2 (in constant depth circuits, we do not count for depth the wires connecting the $\neg$ gates to the input level variables, i.e., we count the depth in terms of inputs by literals.) However in terms of size, it is easy to show that many Boolean functions have DNF and CNF size exponetial in n.

Exercise: The Parity function on n bits

$$\oplus : \{0,1\}^n \to \{0,1\}$$

where $\oplus(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i \bmod 2$ requires DNF and CNF size 2^{n-1}.

The circuit size $C(f_n)$ of a Boolean function f_n is the size of the smallest Boolean circuit which computes f_n. Similarly one can define formula size of f_n (and such definitions can be carried out for a variety of other models of computation: e.g., for any depth k, we can consider the depth k size of f_n, or if we consider only monotone circuit in which $\neg$ gates does not appear, we can define the monotone circuit size for any monotone function f_n.)

The class of problems we will focus on consists of Boolean function families $\{f_n\}_{n=1,2,\ldots}$ with $C(f_n) = n^{O(1)}$, i.e., function families which can be computed by circuit families with size bounded by a polynomial in n. A language L is said to have polynomial size circuits, if its family of characteristic functions at length n $\{L^{=n}\}$ does. A complexity class is said to have polynomial size circuits, if every language in the class does.

THEOREM 4.1 *P has polynomial size circuits.*

Proof. The proof is simple. Take any TM with time bound $T(n)$, and input x of length n. Then one can write down a square array of $T(n) \times T(n)$ cells, where each cell (i, j) refers to the computation of $M(x)$ at time step i and memory location j. It takes $O(1)$ gates to describe the following: the state M is in at time i, whether the tapehead is at location j at time i, and what content is in tape square j at time i. For $i = 1$ this is determined by the initial configuration. For $i > 1$, each cell (i, j) is determined by cells $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$. For example if the tapehead is not at these three locations at time $i-1$, then the cell at (i, j) should remain the same as $(i-1, j)$; if the tapehead is among the three locations at time $i-1$, then the cell at (i, j) should be decided according to the finitary transition function of M. In any case locally it is computed by a Boolean function of size $O(1)$. Thus the circuit which simulates the whole computation of $M(x)$ has size at most $O(T^2(n))$.

Note also that this circuit can be constructed by a logspace computation. ♣

A corollary of this theorem is that the Circuit Value Problem (CVP) is P-complete under logspace reductions.

Circuit Value Problem: Given a Boolean circuit C_n on n inputs and an input of n bits $(b_1, b_2, \ldots, b_n)$, is $C_n(b_1, b_2, \ldots, b_n) = 1$?

COROLLARY 4.2 *CVP is P-complete.*

The robustness of a complexity measure is demonstrated by its invariance under small changes. We noted that, e.g., the class P is unchanged whether we use multitape or onetape TMs, or a variety of other models. This notion of having polynomial size circuits is also quite robust. It does not depend on the particular logical basis functions $\wedge, \vee, \neg$ we chose. Nor does it change if we were to count size in terms of nodes, or allow unbounded fan-in, or push

all negations $\neg$ to the bottom level, etc. Moreover, there are several equivalent definitions for this class.

An advice function is $l : \mathbf{N} \to \Sigma^*$. Thus, for every length n, the function gives a string $l(n)$. The advice function is polynomially bounded if the length $|l(n)| = n^{O(1)}$.

DEFINITION 4.3 *Let $\langle \cdot, \cdot \rangle$ be a standard pairing function, let $\mathcal{C}$ be a complexity class, then $L \in \mathcal{C}/poly$ if there exists some set $L' \in \mathcal{C}$ and a polynomially bounded advice function l, such that for all $x \in \Sigma^{=n}$,*

$$x \in L \Leftrightarrow \langle x, l(|x|) \rangle \in L'.$$

It is important to note that the "advice" $l(|x|)$ only depends on the length of x, but not x itself. Thus, for $L \in \mathrm{P/poly}$ the advice function l can only give polynomially many bits of advice to the P computation of L' for all 2^n inputs $x \in \Sigma^{=n}$.

THEOREM 4.4 *$L \in \mathrm{P/poly}$ iff L has polynomial size circuits.*

Proof. If $L \in \mathrm{P/poly}$ we can adapt the proof above for Theorem 4.1 to show that L has polynomial size circuits.

If L has polynomial size circuits, then we can define an advice function l, such that $l(n)$ is a binary description of the circuit at length n, and the P language L' on input $\langle x, s \rangle$ simply evaluates the circuit encoded in s on x. ♣

This equivalence of P/poly and having polynomial size circuits is so well entrenched that people frequently speak of one term and literally mean the other. Often P/poly is a shart hand for having polynomial size circuits.

Another well-known equivalent condition of P/poly refers to sparse sets.

DEFINITION 4.5 *A set $S \subset \Sigma^*$ is called a sparse set if $|S^{=n}| = n^{O(1)}$, i.e., there are at most a polynomial number of strings at length n in S.*

Clearly S is sparse iff $|S^{\leq n}| = n^{O(1)}$, since $|S^{=n}| \leq |S^{\leq n}| \leq (n+1) \max_{0 \leq m \leq n} |S^{=m}|$. Sparse sets are considered as sets with low information content. Note that generally a set at length n may have up to 2^n strings.

THEOREM 4.6 *$L \in \mathrm{P}^S$ for some sparse set S iff L has polynomial size circuits.*

Proof. Suppose $L \in \mathrm{P}^S$ via a P-time oracle TM M and a sparse set S. On length n inputs, M can only query strings with length up to some polynomial of n in the oracle S. Up to that length there are only polynomially many strings in S. If we explicitly listed all these strings, then we can adapt the proof that P has polynomial size circuits to show that L also has polynomial size circuits.

Conversely, If $L \in \mathrm{P}/poly$ via advice function l, we can encode the advice string $l(n)$ in a sparse oracle. E.g.,

$$S = \{\langle 1^n, p_n\rangle | p_n \text{ is a prefix of } l(n), n \geq 0\}.$$

Given S, we can "suck" out the bits of $l(n)$ as follows: By querying $\langle 1^n, 0\rangle$ and $\langle 1^n, 1\rangle$ one can find the first bit of $l(n)$. Having found a prefix p of $l(n)$, we query $\langle 1^n, p0\rangle$ and $\langle 1^n, p1\rangle$, and we can extend the prefix p by one more bit (or we found that $p = l(n)$ is already fully extended.) ♣

The class P/poly is the non-uniform analog of P. It contains the class P, as well as non-recursive sets. Since it also includes non-recursive sets, it is not possible to include the class P/poly in any uniform complexity classes, such as EXP. However, the intuitive feeling is that sets in P/poly are "easy", in a combinatiorial sense. The belief is that NP $\neq$ P because NP $\not\subset$ P/poly. This remains the most concrete approach to a proof of NP $\neq$ P.

However is it possible to include higher uniform complexity classes than P in the class P/poly? What if NP $\subset$ P/poly? An unlikely complexity theoretic consequence would provide evidence that indeed NP $\not\subset$ P/poly. This is the content of the famous Karp-Lipton Theorem.

4.2 Karp–Lipton Theorem

In this section, we prove the classical Karp-Lipton theorem. It deals with the question of whether NP $\subset$ P/poly? It is believed that NP $\not\subset$ P/poly. Karp-Lipton theorem gives some evidence in this direction. It says that if NP $\subset$ P/poly, then PH $= \Sigma_2^P \cap \Pi_2^P$.

By NP-completeness, NP $\subset$ P/poly is equivalent to SAT $\in$ P/poly. Also by the equivalent formulations of P/poly, this is equivalent to the assertion that SAT(or any other NP-complete language, or every NP-complete language, or every language in NP) has polynomial size circuits. It is also equivalent to the assertion that SAT is polynomial time Turing reducible to a sparse set S: SAT $\in \mathrm{P}^S$.

Self Reducibility: The main idea is self-reducibility of SAT. Suppose we are given a circuit C that is suppopsed to compute SAT for all Boolean formulae of length at most n. It may or may not compute SAT correctly. It may make errors in two ways. It may reject a $\varphi \in$ SAT, and/or it may also accept a $\varphi \notin$ SAT. We can convert C into a circuit C', that does not make errors of the second type. C' works by using C as a black box. Upon input φ over the variables, $x_1, \ldots, x_m$, it feeds φ to C. If C rejects, C' rejects. If C accepts, C' tries to find a satisfying truth assignment σ to φ. To do that, C' converts φ to φ_1, by setting all the occurrences of x_1 in φ to be TRUE. And asks C if φ_1 is satisfiable. If so set $\sigma(x_1)$=TRUE, else set $\sigma(x_1)$=FALSE. Now repeat this process, with φ_1 in place of φ and x_2 in place of x_1, to get the value for $\sigma(x_2)$. Continue this process for the other variables we find a truth assignment σ. Then we check whether σ really satisfies φ, if so accept, else

reject φ. If C is indeed a correct circuit for SAT, C' would also be a correct one. But, no matter whether C is a correct circuit for SAT, C' may make errors only on one side. It may reject a satisfiable formula. But, it would never accept an unsatisfiable formula, because it only accepts when it has verified a satisfying truth assignment. This process of reducing an instance of a problem (in this case SAT) to another instance of the same problem of a smaller "size" (in this case the number of variables left unassigned) is called self-reducibility.

THEOREM 4.7 (KARP-LIPTON) *If* NP $\subset$ P/poly*, then* PH $= \Sigma_2^P \cap \Pi_2^P$.

Remark: Originally Karp and Lipton proved the collapse to the third level of the Polynomial Hierarchy. They noted in their paper that Sipser improved it to the second level of PH. The proof we give uses an idea first noted by Hopcroft.

Proof. The assumption implies that SAT has polynomial circuits. Then, we show that $\Pi_2^P \subseteq \Sigma_2^P$. This would imply that PH collapses to $\Sigma_2^P \cap \Pi_2^P$.

Let $L \in \Pi_2^P$. So, for some P-time predicate D, we have

$$x \in L \iff \forall^p y \exists^p z[D(x, y, z) = 0]$$

where $|y|$ and $|z|$ are polynomially bounded in $|x|$. We can convert the question of

$$\exists^p z[D(x, y, z) = 0]$$

into a SAT question via Cook's Theorem. Meaning, we have a polynomial time computable function $\varphi(.,.)$ such that

$$x \in L \iff \forall^p y[\varphi(x, y) \in \text{SAT}]$$

Let x be an input. We need to check whether $x \in L$. We simulate the Π_2^P computation of L as follows. Let the length of the formulas $|\varphi(x, y)|$ be at most $p(n)$, where $n = |x|$ and $p(.)$ is some polynomial. By our assumption, there is a (poly-sized) circuit C^* that would work correctly for SAT for all formulas of length at most $p(n)$. Our Σ_2^P computation, in its "there-exists" phase, guesses a circuit C. Then uses self-reducibility to convert it into a one-sided error circuit C'. Now it enters the "for all" phase and checks whether

$$\forall^p y[C'(\varphi(x, y)) = 1]$$

Suppose $x \in L$. One of the paths in the "there-exists" phase would guess C^*, a correct circuit for SAT. The corresponding C' will also be a correct circuit for SAT. As $x \in L$, $\forall^p y$, $\varphi(x, y)$ will be satisfiable and our C' would accept all of them for these polynomially bounded y's. So we see that,

$$(\exists^p C)(\forall^p y)[C'(\varphi(x, y)) = 1]$$

Suppose $x \notin L$. Then, there is some y_0, polynomially bounded in length, for which $\varphi(x, y_0)$ is not satisfiable. So, for all the guesses C, the corresponding C' would reject $\varphi(x, y_0)$, because C' never accepts any unsatisfiable formula. Thus, we see that,

$$(\forall^p C)(\exists^p y)[C'(\varphi(x, y)) = 0]$$

More formally, define the following deterministic polynomial algorithm $A(x, C, y)$: it first computes C' from C using self-reducibility. Then, it computes the formula $\varphi(x, y)$ using Cook's theorem. Output $C'(\varphi(x, y))$. The claim is then

$$x \in L \iff (\exists^p C)(\forall^p y)[A(x, C, y) = 1]$$

♣

4.3 Mahaney's Theorem

In the last section, we discussed Karp–Lipton theorem, which says that if SAT has polynomial size circuits, then the polynomial hierarchy collapses to the second level. In other words, if SAT Cook reduces to some sparse set S, then the hierarchy collapses to the second level. A natural question at this point is, what happens if SAT Karp reduces to a sparse set. Notice that this is a stronger assumption. Mahaney's theorem says that if SAT Karp reduces to a sparse set, then NP=P. The theorem also has implications in the context of Berman-Hartmanis conjecture, which we will discuss later. In this section, we prove Mahaney's theorem. This proof follows the idea of Ogihara and Watanabe.

DEFINITION 4.8 *Let $\sigma = \sigma_1\sigma_2 \ldots \sigma_m$ and $\tau = \tau_1\tau_2 \ldots \tau_k$ be binary strings. We say σ is to the left of τ (denoted $\sigma \preceq_l \tau$) if either σ is an extension of τ, or the first bit from the left where they differ σ has bit 0 and τ has bit 1. Formally, let $i = \max\{j \mid 0 \leq j \leq \min\{k, m\}, (\forall 1 \leq j' \leq j)[\sigma_{j'} = \tau_{j'}]\}$. Then $\sigma \preceq_l \tau$ if either $i = k$ or $i < k$ with $\sigma_{i+1} < \tau_{i+1}$.*

We can view the above definition pictorially as follows. Consider the binary tree of partial assignments to the n variables. Let σ and τ be two partial assignments, $\tau = \tau_1\tau_2 \ldots \tau_k$ and $\sigma = \sigma_1\sigma_2 \ldots \sigma_m$. σ is to the *left* of τ, if σ is below τ (i.e. τ is a prefix of σ) or σ is in a branch to the left of τ. The order $\preceq_l$ is the tree traversal order of Left-Right-Root.

We next define a language called *left cut* of SAT. Let φ be a formula and τ be a partial assignment. The pair $\langle \varphi, \tau \rangle$ is in L_{SAT}, if some assignment σ to the left of τ satisfies φ. Formally,

DEFINITION 4.9 *The leftcut set of* SAT *is the set* $\text{L}_{\text{SAT}} = \{\langle \varphi, \tau \rangle \mid \varphi$ *is a formula on* n *variables and* $\tau \in \{0,1\}^k, 0 \leq k \leq n$, *such that* $\exists\ \sigma \in \{0,1\}^n$ *with* $\sigma \preceq_l \tau$ *and* $\varphi|_\sigma =$ *True.*$\}$

Clearly $L_{\text{SAT}} \in \text{NP}$. The crucial property of L_{SAT} is the following. Let φ be a formula and σ and τ be two partial assignments, such that $\sigma \preceq_l \tau$. Then, if $\langle \varphi, \sigma \rangle$ is in L_{SAT}, then $\langle \varphi, \tau \rangle$ is also in L_{SAT}. In particular if $\varphi \in \text{SAT}$, then $\langle \varphi, \epsilon \rangle \in L_{\text{SAT}}$.

Suppose we are given $\langle \varphi, \sigma \rangle$ and $\langle \varphi, \tau \rangle$ and we have to bet on one of these to be in L_{SAT}. Clearly, we should place our bet on $\langle \varphi, \tau \rangle$, if $\sigma \preceq_l \tau$. In its contra-positive, if at least one of them is out of L_{SAT}, then we know $\langle \varphi, \sigma \rangle$ is out. The proof hinges on this property of L_{SAT}.

THEOREM 4.10 (MAHANEY) *For any sparse set* $S \neq \emptyset$, $\text{SAT} \leq_m^p S \iff \text{P} = \text{NP}$.

Proof. Let S be any non-empty sparse set. The implication $\text{P} = \text{NP} \Longrightarrow \text{SAT} \leq_m^p S$ is trivial, so we need only prove that $\text{SAT} \leq_m^p S \Longrightarrow \text{P} = \text{NP}$.

Suppose $\text{SAT} \leq_m^p S$. We shall design a polynomial time algorithm to solve SAT. First of all, notice that L_{SAT} is in NP. (Given $\langle \varphi, \tau \rangle$, we guess a (full) truth assignment σ, then verify that $\sigma \preceq_l \tau$ and σ satisfies φ; if so accept, else reject.) From the assumption $\text{SAT} \leq_m^p S$, it follows that $\text{L}_{\text{SAT}} \leq_m^p S$, via some polynomial time computable function f.

We will design a polynomial time algorithm for SAT. Let φ be the input formula over n variables and we need to check if it is satisfiable or not. If φ is satisfiable, the algorithm would, in fact, output the (lexicographically) left-most satisfying assignment.

Let us consider the binary tree formed by assignments on φ. The root of this tree corresponds to the empty assignment (denoted by the empty string ϵ, where no variable is assigned a value). Nodes of the tree correspond to partial assignments. We will do a breadth-first search on this tree, starting with the root. The tree has exponentially many nodes and we cannot hope to do a full search. Instead, as we go along, we will prune the tree and explore only parts of the tree. At any point of time, we will maintain only polynomially many nodes. We will ensure that, if φ is satisfiable, the left-most satisfying assignment is a descendant of a node not pruned away.

Consider strings of the form $\langle \varphi, \sigma \rangle$, where $\sigma = \sigma_1 \sigma_2 \ldots \sigma_m$ is some partial assignment for the input formula φ. Length of these strings is at most $|\varphi| + n$. As the reduction f runs in polynomial time, its output on these strings can be at most polynomial in n. Let this number be l. The sparse set can have at most N strings of length $\leq l$, where N is polynomial in l, and thus also a polynomial in n. As we explore the binary tree, level by level, we will maintain at most N nodes, at any point of time.

First, we explore the tree until we reach a level k, where the number of nodes $2^k > N$. At this point, we start pruning. We run the reduction f on all these 2^k partial assignments (with φ as the formula) and obtain 2^k output strings. We first look for duplicates. Let σ_1 and σ_2 be two partial assignments at level k such that $f(\langle \varphi, \sigma_1 \rangle) = f(\langle \varphi, \sigma_2 \rangle)$. If $\sigma_1 \preceq_l \sigma_2$, then we throw away σ_2, i.e. we will not explore that subtree any further. If $\sigma_2 \preceq_l \sigma_1$, then we throw away σ_1. Note that as $f(\langle \varphi, \sigma_1 \rangle) = f(\langle \varphi, \sigma_2 \rangle)$, either both (φ, σ_1) and (φ, σ_1) are in L_{SAT} or both are out. After removing all duplicates, if more than N nodes survive, we throw away the left-most partial assignment at this level. We keep removing the left-most

nodes until we have only N nodes. Clearly, after all is said and done at this level, we will have at most N nodes that survive.

Then we continue at the next level, expanding only these subtrees. In general, suppose $r \leq N$ nodes survive at level m. In level $m+1$, we consider their $2r$ children. If $2r \leq N$, we can move on to level $m+2$. If not, we run the reduction f on these $2r$ nodes, and obtain $2r$ output strings. We first eliminate duplicates and then, if more than N nodes still remain, we eliminate the the left-most nodes, until we have at most N nodes. Continuing this way, finally we will reach the leaf level of the tree. Here, all the nodes correspond to full truth assignments of φ. We will have at most N surviving full assignments. For each of these assignments, we check if any of them satisfy φ. If so φ is satisfiable. Otherwise, we claim it is not satisfiable.

Clearly, we maintain at most N nodes at any level of the tree and N is polynomial in n. Depth of the tree is n. Thus the algorithm runs in polynomial time. Suppose φ is satisfiable. We argue that the left-most satisfying assignment will remain a descedant of some node kept at any level, and thus survive at the leaf-level. We do this by induction on the level number m. This is clearly so at the root level. Consider level m and let the number of nodes surviving at this level be r. By induction, the left-most satisfying assignment is a descendant of one of the r surviving nodes. We consider the $2r$ children at level $m+1$. If $2r > N$. Suppose $f(\langle\varphi, \sigma_1\rangle) = f(\langle\varphi, \sigma_2\rangle)$ and $\sigma_1 \preceq_l \sigma_2$. So we remove a duplicate. Let $\sigma_1 \preceq_l \sigma_2$. In this case, we throw away σ_2. Clearly, the left-most satisfying assignment is not a descendant of σ_2, for otherwise we would have the contradiction that $\langle\varphi, \sigma_2\rangle \in \mathrm{L_{SAT}}$ and $\langle\varphi, \sigma_1\rangle \notin \mathrm{L_{SAT}}$, as σ_1 is strictly to the left of σ_2. Thus, throwing away σ_2 did not violate the inductive hypothesis.

After removing duplicates, if still some $d > N$ nodes exist, let these nodes be $\sigma_1, \sigma_2, \ldots, \sigma_d$. Then we removed the left-most node σ_1. We argue that this is also a correct procedure. By contradiction, suppose the left-most satisfying assignment σ is a descendant of σ_1. Then, σ is to the left of all the d surviving nodes. Thus, all of $\langle\varphi, \sigma_1\rangle, \langle\varphi, \sigma_2\rangle, \ldots \langle\varphi, \sigma_d\rangle$ are in $\mathrm{L_{SAT}}$. So, $f(\langle\varphi, \sigma_1\rangle), f(\langle\varphi, \sigma_2\rangle), \ldots f(\langle\varphi, \sigma_d\rangle)$ are all in S. Note that, all these strings are distinct and there are $d > N$ of them. But, by our assumption of sparseness of S, there can be at most N strings in S (at this length). This contradiction shows that throwing away the left-most node also maintains the inductive hypothesis. This completes the induction proof. ♣