Introduction to Complexity Theory*
Lecture 8: Non-Uniform Polynomial Time - $\mathcal{P}$/Poly

Notes taken by Moshe Lewenstein, Yehuda Lindell and Tamar Seeman

January 14, 1999

Summary: In this lecture we introduce the notion of non-uniform polynomial time and the corresponding complexity class $\mathcal{P}$/poly. In this computational model, Turing machines are provided an external advice string to aid them in their computation. The *non-uniformity* is expressed in the fact that a different advice string may be defined for every different length of input. We show that $\mathcal{P}$/poly upper bounds efficient computation (as $\mathcal{BPP} \subseteq \mathcal{P}$/poly), yet even contains some non-recursive languages. The effect of introducing *uniformity* is discussed (as an attempt to rid $\mathcal{P}$/poly of its absurd intractable languages) and shown to reduce the class to be exactly $\mathcal{P}$. Finally, we show that, among other things, $\mathcal{P}$/poly may help us separate $\mathcal{P}$ from $\mathcal{NP}$. We do this by showing that trivially $\mathcal{P} \subset \mathcal{P}$/poly, and that under a reasonable conjecture $\mathcal{NP} \not\subseteq \mathcal{P}$/poly.

1 Introduction

The class of $\mathcal{P}$/poly, or non-uniform polynomial time, is the class of Turing machines which receive external advice to aid computation. More specifically for all inputs of length n a Turing machine is supplemented with a single advice string a_n of polynomial length . Alternatively we may view a non-uniform machine as an infinite series of Turing machines $\{M_n\}$, where M_i computes for inputs of length i. In this case the advice is "hardwired" into the machine.

The class of $\mathcal{P}$/poly provides an upper bound on what is considered to be efficient computation. This upper bound is not tight; for example, as we shall show later, $\mathcal{P}$/poly contains non-recursive languages. However, the upper bound ensures that every efficiently computable language is contained in $\mathcal{P}$/poly.

An additional motivation in creating the class of $\mathcal{P}$/poly is to help separate the classes of $\mathcal{P}$ and $\mathcal{NP}$. This idea is explained in further detail below.

1.1 The Actual Definition

We now define the class of $\mathcal{P}$/poly according to two different definitions, and then show that these two definitions are in fact equivalent. Recall that:

$$\chi_L(x) = \begin{cases} 1, & \text{if } x \in L; \\ 0, & \text{otherwise.} \end{cases}$$

* Lecture Notes for a course given by Oded Goldreich at the Weizmann Institute of Science, Israel.

Definition 1 (standard): *$L \in \mathcal{P}/poly$ if there exists a sequence of circuits $\{C_n\}$, where for each n, C_n has n inputs and one output, and there exists a polynomial $p(.)$ such that for all n, size(C_n) $\leq p(n)$ and $C_n(x) = \chi_L(x)$ for all $x \in \{0,1\}^n$.*

A series of polynomial circuits $\{C_n\}$ as defined above is called a *non-uniform family of circuits.* The non-uniformity is expressed in the fact that there is not necessarily any connection between a circuit of size n and $n+1$. In fact for every n we may define a completely different "algorithm".

Note that the circuits in the above definition can be simulated in time linear to their size. Thus although time is not explicitly mentioned in the definition, it is implicit.

Definition 2 (alternative): *$L \in \mathcal{P}/poly$ if there exists a polynomial-time two-input machine M, a polynomial $p(.)$, and a sequence $\{a_n\}$ of advice strings, where length(a_n) $\leq p(n)$, such that for all n and for all $x \in \{0,1\}^n, M(a_n, x) = \chi_L(x)$.*

If exponentially long advice were allowed in the above definition, then a_n could be a look-up table containing $\chi_L(x)$ for any language L and every input x of length n. Thus every language would trivially be in such a class. However, this is not the case as a_n is polynomially bounded. Restricting the length of the advice defines a more meaningful class, but as we have mentioned, some intractable problems still remain "solvable".

Proposition 1.1 *The two definitions of $\mathcal{P}/poly$ are equivalent.*

Proof:
($\Rightarrow$): Assume $L \in \mathcal{P}/$poly by Definition 1, i.e. there exists a family $\{C_n\}$ of circuits deciding L, such that size(C_n) is polynomial in n. Let desc(C_n) be the description of C_n according to a standard encoding of circuits. Consider the universal Turing machine M such that for all n, and all x of length n, M(desc(C_n),x) simulates $C_n(x)$. Then define the sequence $\{a_n\}$ of advice strings such that for every n, a_n = desc(C_n). Thus $L \in \mathcal{P}/$poly by Definition 2.

($\Leftarrow$): Assume L is in $\mathcal{P}/$poly by Definition 2, i.e. there exist a Turing machine M and a sequence of advice $\{a_n\}$ deciding L. We look at all possible computations of $M(a_n, \cdot)$ for n-bit inputs. $M(a_n, \cdot)$ is a polynomial time-bounded deterministic Turing machine working on n-length inputs. In the proof of Cook's Theorem, in Lecture 2, we showed that Bounded Halting is Levin-reducible to Circuit Satisfiability. Given an instance of Bounded Halting ($< M(\cdot,\cdot) >, x, 1^t$) the reduction is comprised of constructing a circuit C which on input y outputs $M(x,y)$. The situation here is identical since for $M(a_n, \cdot)$ a circuit may be constructed which on input x outputs $M(a_n, x)$. In other words we build a sequence $\{C_n\}$ of circuits, where for each n, C_n is an encoding of $M(a_n, \cdot)$. Thus L is in $\mathcal{P}/$poly by Definition 1. ■

It should be noted that in Definition 2, M is a finite object, whereas $\{a_n\}$ may be an infinite sequence (as is the sequence $\{C_n\}$ of circuits according to Definition 1). Thus $\mathcal{P}/$poly is an unrealistic mode of computation, since such machines cannot actually be constructed.

1.2 $\mathcal{P}/$poly and the $\mathcal{P} = \mathcal{NP}$ Question:

As mentioned above, one of the motivations in defining the class of $\mathcal{P}/$poly is to separate $\mathcal{P}$ from $\mathcal{NP}$. The idea is to show that there is a language which is in $\mathcal{NP}$ but is not in $\mathcal{P}/$poly, and thus not in $\mathcal{P}$. In this way, we would like to show that $\mathcal{P} \neq \mathcal{NP}$. To do so, though, we must first understand the relationship of $\mathcal{P}$/poly to the classes $\mathcal{P}$ and $\mathcal{NP}$. Trivially, $\mathcal{P} \subseteq \mathcal{P}$/poly because the class $\mathcal{P}$ may be viewed as the set of $\mathcal{P}$/poly machines with empty advice, i.e. $a_n = \lambda$ for all n.

At first glance, Definition 2 of $\mathcal{P}$/poly appears to resemble that of $\mathcal{NP}$. In $\mathcal{NP}$, $x \in L$ iff there exists a witness w_x such that $M(x, w_x) = 1$. The witness is somewhat analogous to the advice in $\mathcal{P}$/poly. However, the definition of $\mathcal{P}$/poly differs from that of $\mathcal{NP}$ in two ways:

1. For a given n, $\mathcal{P}$/poly has a universal witness a_n as opposed to $\mathcal{NP}$ where every x of length n may have a different witness.
2. In the definition of $\mathcal{NP}$, for every $x \notin L$, for every witness w, $M(x, w) = 0$. In other words, there do not exist false witnesses. However, this is not true for $\mathcal{P}$/poly. We do not claim that there are no bad advice strings for Definition 2 of $\mathcal{P}$/poly; we merely claim that there exists a good advice string.

We therefore see that the definitions of $\mathcal{NP}$ and $\mathcal{P}$/poly differ from each other; this raises the possibility that there may be a language which is in $\mathcal{NP}$ but not in $\mathcal{P}$/poly. As we shall show later this seems to be likely since a sufficient condition for the existence of such a language is based upon a reasonable conjecture. Since $\mathcal{P}$ is contained in $\mathcal{P}$/poly, finding such a language is sufficient to fulfill our goal. In fact, the original motivation for $\mathcal{P}$/poly was the belief that one may be able to prove lower bounds on sizes of circuits computing certain functions (e.g., the characteriztic function of an NP-complete language). So far, no such bounds are known (except if one restricts the circuits in various ways; as we'll discuss in next semester).

2 The Power of $\mathcal{P}$/poly

As we have mentioned, $\mathcal{P}$/poly is not a realistic mode of computation. Rather, it provides an *upper bound* on what we consider efficient computation (that is, any language not in $\mathcal{P}$/poly should definitely not be efficiently computable). In the last lecture we defined probabilistic computation and reevaluated our view of efficient computation to be $\mathcal{BPP}$, rather than $\mathcal{P}$. We now show that $\mathcal{BPP} \subseteq \mathcal{P}$/poly and therefore that $\mathcal{P}$/poly also upper bounds our "new" view of efficient computation.

However, we will also show that $\mathcal{P}$/poly contains far more than $\mathcal{BPP}$. This actually yields a very high upper bound. In fact $\mathcal{P}$/poly even contains non-recursive languages. This containment should convince anyone that $\mathcal{P}$/poly does not reflect any level of realistic computation.

Theorem 3 : *$\mathcal{P}$/poly contains non-recursive languages.*

Proof: This theorem is clearly implied from the following two facts:

1. There exist unary languages which are non-recursive, and
2. For every unary language L, $L \in \mathcal{P}$/poly.

We remind the reader that L is a unary language if $L \subseteq \{1\}^*$.

Proof of Claim 1:

Let L be any non-recursive language. Define $L' = \{1^{index(x)} \mid x \in L\}$ where index(x) is the position of x in the standard enumeration of binary strings (i.e. we view the string as a binary number). Clearly L' is unary and non-recursive (any Turing machine recognizing L' can trivially be used to recognize L).

Proof of Claim 2:

For every unary language L, define

$$a_n = \begin{cases} 1, & \text{if } 1^n \in L; \\ 0, & \text{otherwise.} \end{cases}$$

A Turing machine can trivially decide L in polynomial (even linear) time given x and $a_{|x|}$, by simply accepting iff x is unary and $a_{|x|} = 1$. Therefore, $L \in \mathcal{P}/\text{poly}$. ■

The ability to decide intractable languages is a result of the non-uniformity inherent in $\mathcal{P}/\text{poly}$. There is no requirement that the series $\{a_n\}$ is even computable.

Note that this method of reducing a language to its unary equivalent cannot help us with polynomial classes as the reduction itself is exponential. However, for recursive languages we are interested in computability only.

Theorem 4 : $\mathcal{BPP} \subseteq \mathcal{P}/poly$.

Proof: Let $L \in \mathcal{BPP}$. By means of amplification, there exists a probabilistic Turing machine M such that for every $x \in \{0,1\}^n$: $Prob_{r \in \{0,1\}^{poly(n)}}[M(x,r) = \chi_L(x)] > 1 - 2^{-n}$, (the probabilities are taken over all possible choices of random strings).
Equivalently, M is such that $Prob_r[M(x,r) \neq \chi_L(x)] < 2^{-n}$. We therefore have:

$$Prob_r[\exists x \in \{0,1\}^n : M(x,r) \neq \chi_L(x)] \leq \sum_{x \in \{0,1\}^n} Prob_r[M(x,r) \neq \chi_L(x)] < 2^n \cdot 2^{-n} = 1.$$

The first inequality comes from the Union Bound, that is, for every series of sets $\{A_i\}$ and every random variable X:

$$Prob(X \in \bigcup_{i=1}^{n} A_i) \leq \sum_{i=1}^{n} Prob(X \in A_i).$$

and the second inequality is based on the error probability of the machine.

Note that if for every random string r, there is at least one x such that $M(x,r) \neq \chi_L(x)$, then the above probability would equal 1. We can therefore conclude that there is *at least* one string r such that for every x, $M(x,r) = \chi_L(x)$. We therefore set $a_n = r$ (note that r is different for different lengths of input n, but this is fine according to the definition of $\mathcal{P}/\text{poly}$). Our $\mathcal{P}/\text{poly}$ machine simulates M, using a_n as its random choices. ■

This method of proof is called *the probabilistic method.* We do not know how to find these advice strings and the proof of their existence is implicit. We merely argue that the probability that a random string is not an adequate advice is strictly smaller than 1. This is enough to obtain the theorem.

3 Uniform Families of Circuits

As we have mentioned earlier, circuits of different sizes belonging to a non-uniform family may have no relation to each other. This results in the absurd situation of having families of circuits deciding non-recursive languages.

This leads us to the following definition which attempts to define families of circuits which *do* match our expectations of realistic computation.

Definition 5 (uniform circuits): *A family of circuits $\{C_n\}$ is called* uniform *if there exists a deterministic polynomial time Turing machine M such that for every n, $M(1^n) = desc(C_n)$, where $desc(C_n)$ is a standard encoding of circuits.*

Thus a uniform family of circuits has a succinct (finite) description (or equivalently for a series of advice strings). Clearly, a uniform family of circuits cannot recognize non-recursive languages. Actually, the restriction of uniformity is far greater than just this.

Theorem 6 : *A language L has a uniform family of circuits $\{C_n\}$ such that for all n and for all $x \in \{0,1\}^n$ $C_n(x) = \chi_L(x)$ if and only if $L \in \mathcal{P}$.*

Proof:
($\Rightarrow$) Let $\{C_n\}$ be a uniform family of circuits deciding L, and M the polynomial time Turing machine which generates the family. The following is a polynomial time algorithm for deciding L:

On input x:

- $C_{|x|} \leftarrow M(1^{|x|})$
- Simulate $C_{|x|}(x)$ and return the result.

Since M is polynomial-time bounded and the circuits are of polynomial size, the algorithm clearly runs in polynomial time. Therefore $L \in \mathcal{P}$.

($\Leftarrow$) $L \in \mathcal{P}$. Therefore, there exists a polynomial time Turing machine M deciding L. As in the proof of Cook's Theorem, a polynomial size circuit deciding L on strings of length n may be built from M in time polynomial in n. The Turing machine M' that constructs the circuits may then be taken as M in the definition of uniform circuits. That is, given x, M' calculates $|x|$ and builds the appropriate circuit.

Alternatively, by Definition 2, no advice is necessary here and we may therefore take $a_n = \lambda$ for every n. ■

4 Sparse Languages and the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ Question

In this section we will see why $\mathcal{P}$/poly may help us separate between $\mathcal{P}$ and $\mathcal{NP}$. We will first define sparse languages.

Definition 7 (sparse languages): *A language S is* sparse *if there exists a polynomial $p(\cdot)$ such that for every n $|S \cap \{0,1\}^n| \leq p(n)$.*

Example: Trivially, every unary language is sparse (take $p(n) = 1$).

Theorem 8 : *$\mathcal{NP} \subseteq \mathcal{P}$/poly if and only if for every $L \in \mathcal{NP}$, the language L is Cook-reducible to a sparse language.*

As we conjecture that no $\mathcal{NP}$-Complete language can be sparse, we have that $\mathcal{NP}$ contains languages not found in $\mathcal{P}$/poly.

Proof: It is enough for us to prove that SAT $\in \mathcal{P}$/poly if and only if SAT is Cook-reducible to some sparse language.

($\Rightarrow$) Suppose that SAT $\in \mathcal{P}$/poly. Therefore there exists a series of advice strings $\{a_n\}$ and a Turing machine M as in Definition 2, where $\forall n\ |a_n| \leq q(n)$ for some polynomial $q(\cdot)$.
Define $s_i^n = 0^{i-1}10^{q(n)-i}$ and define $S = \{1^n 0 s_i^n$: for $n \geq 0$ where bit i of a_n is 1 $\}$.
Clearly S is sparse since for every n $|S \cap \{0,1\}^{n+q(n)+1}| \leq |a_n| \leq q(n)$.

We now show a Cook-reduction of SAT to S:
Input: φ of length n

1. Reconstruct a_n by $q(n)$ queries to S. Specifically, the queries are: $1^n 0 s_1^n, 1^n 0 s_2^n, ..., 1^n 0 s_{q(n)}^n$.
2. Run $M(a_n, \varphi)$ thereby solving SAT in (standard) polynomial time.

We therefore solve SAT with a polynomial number of queries to an S-oracle, i.e. SAT Cook-reduces to S.

($\Leftarrow$) Suppose that SAT Cook-reduces to some sparse language S. Therefore, there exists a polynomial time bounded oracle machine M^S which solves SAT. Let $t(\cdot)$ be M's (polynomial) time-bound. Then, on input x, machine M makes queries of length at most $t(|x|)$.
Construct a_n in the following way: concatenate all strings of length *at most* $t(n)$ in S. Since S is sparse, there exists some polynomial $p(\cdot)$ such that $\forall n\ |S \cap \{0,1\}^n| \leq p(n)$. The length of the list of strings of lengths exactly i in a_n is then less than or equal to $i \cdot p(i)$ (i.e. at most $p(i)$ different strings of length i each). Therefore:

$$|a_n| \leq \sum_{i=1}^{t(n)} i \cdot p(i) < t(n)^2 \cdot p(t(n))$$

So, a_n is polynomial in length. Now, given a_n, every oracle query to S can be "answered" in polynomial time. For a given string x, we check if $x \in S$ by simply scanning a_n and seeing if x appears or not. Therefore, M^S may by simulated by a deterministic machine with access to a_n. This machine takes at most $t(n) \cdot |a_n|$ time (each lookup may take as long as scanning the advice). Therefore SAT $\in \mathcal{P}$/poly. ■

As we have mentioned, we conjecture that there are no sparse $\mathcal{NP}$-Complete languages. This conjecture holds for both Karp and Cook reductions. However for Karp-reductions, the ramifications of the existence of a sparse $\mathcal{NP}$-Complete language would be extreme, and would show that $\mathcal{P} = \mathcal{NP}$. This is formally stated and proved in the next theorem. It is interesting to note that our belief that $\mathcal{NP} \not\subseteq \mathcal{P}$/poly is somewhat parallel to our belief that $\mathcal{P} \neq \mathcal{NP}$ when looked at in the context of sparse languages.

Theorem 9 *$\mathcal{P} = \mathcal{NP}$ if and only if for every language $L \in \mathcal{NP}$, the language L is Karp-reducible to a sparse language.*

Proof:
($\Rightarrow$): Let $L \in \mathcal{NP}$. We define the following trivial function as a Karp-reduction of L to $\{1\}$:

$$f(x) = \begin{cases} 1, & \text{if } x \in L; \\ 0, & \text{otherwise.} \end{cases}$$

If $\mathcal{P} = \mathcal{NP}$ then L is polynomial-time decidable and it follows that f is polynomial-time computable. Therefore, L Karp-reduces to the language $\{1\}$, which is obviously sparse.

($\Leftarrow$): For sake of simplicity we prove a weaker result for this direction. However the claim is true as stated. Beforehand we need the following definition:

Definition 10 (guarded sparse languages): *A sparse language S is called* guarded *if there exists a sparse language G in $\mathcal{P}$ such that $S \subseteq G$.*

The language that we considered in the proof of theorem 8: $S = \{1^n 0 s_i^n : \text{for } n \geq 0, \text{ where bit } i \text{ of } a_n \text{ is } 1\}$ is an example of a sparse guarded language. It is obviously sparse and it is guarded by $G = \{1^n 0 s_i^n : \forall n \geq 0 \text{ and } 1 \leq i \leq q(n)\}$. Note that any unary language is also a guarded sparse language since $\{1^n : n \geq 0\}$ is sparse and trivially in $\mathcal{P}$.

The slightly weaker result that we prove for this direction is as follows.

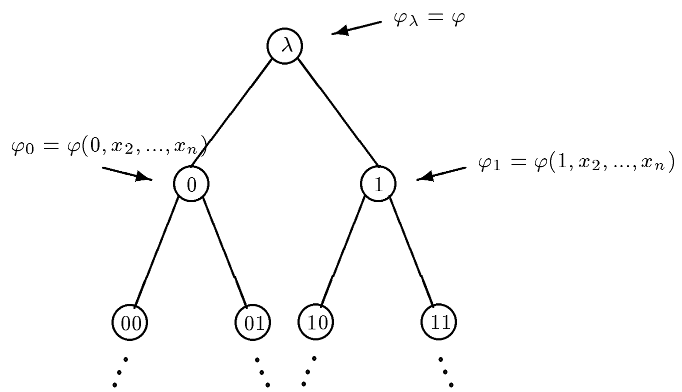
Proposition 4.1 *If SAT is Karp-reducible to a guarded sparse language then SAT $\in \mathcal{P}$.*

Proof: Assume that SAT is Karp-reducible to a sparse language S that is guarded by G. Let f be the Karp-reduction of SAT to S. We will show a polynomial-time algorithm for SAT.

Input: A Boolean formula $\varphi = \varphi(x_1, ..., x_n)$.

Envision the binary tree of all possible assignments. Each node is labelled $\alpha = \alpha_1\alpha_2...\alpha_i \in \{0,1\}^i$ which corresponds to an assignment of φ's first i variables. Let $\varphi_\alpha(x_{i+1}, ..., x_n) = \varphi(\alpha_1, ..., \alpha_i, x_{i+1}, ..., x_n)$ be the CNF formula corresponding to α. We denote $x_\alpha = f(\varphi_\alpha)$ (recall that $\varphi_\alpha \in \text{SAT} \Leftrightarrow x_\alpha \in S$).

The root is labelled λ, the empty string, where $\varphi_\lambda = \varphi$. Each node labelled α has two sons, one labelled $\alpha 0$ and the other labelled $\alpha 1$ (note that the sons have one variable less in their corresponding formulae). The leaves are labelled with n-bit strings corresponding to full assignments, and therefore to a Boolean constant.



The tree of assignments.

The strategy we will employ to compute φ will be a DFS search on this tree from root to leaves using a branch and bound technique. We backtrack from a node *only* if there is no satisfying assignment in its entire subtree. As soon as we find a leaf satisfying φ, we halt returning the assignment.

At a node α we consider x_α. If $x_\alpha \notin G$ (implying that $x_\alpha \notin S$), then φ_α is not satisfiable. This implies that the subtree of α contains no satisfying assignments and we can stop the search on this subtree. If $x_\alpha \in G$, then we continue searching in α's subtree.

At a leaf α we check if the assignment α is satisfiable (note that it is not sufficient to check that $x_\alpha \in G$ since f reduces to S and not to G). This is easy as we merely need to evaluate a Boolean expression in given values.

The key to the polynomial time-bound of the algorithm lies in the sparseness of G. If we visit a number of nodes equal to the number of strings in G of appropriate length, then the algorithm will clearly be polynomial. However, for two different nodes α and β, it may be that $x_\alpha = x_\beta \in G$ and we search both their subtrees resulting in visiting too many nodes. We therefore maintain a set B such that $B \subseteq G - S$ remains invariant throughout. Upon backtracking from a node α (where $x_\alpha \in G$), we place x_α in B. We then check for every node α, that $x_\alpha \notin B$ before searching its subtree, thus preventing a multiple search.

Algorithm: On input $\varphi = \varphi(x_1, ..., x_n)$.

1. $B \leftarrow \emptyset$
2. Tree-Search(λ)
3. In case the above call was not halted, reject φ as non-satisfiable.

In the following procedure, returning from a recursive call on α indicates that the subtree rooted in α contains no satisfying assignment (or, in other words, φ_α is not satisfiable). In case we reach a leaf associated with a satisfying assignment, the procedure halts outputting this assignment.

Procedure Tree-Search(α)

1. determine $\varphi_\alpha(x_{i+1}, ..., x_n) = \varphi(\alpha_1, ..., \alpha_i, x_{i+1}, ..., x_n)$
2. if $|\alpha| = n$: /* at a leaf - φ_α is a constant */
   if $\varphi_\alpha \equiv T$ then output the assignment α and halt
   else return
3. if $|\alpha| < n$:
   (a) compute $x_\alpha = f(\varphi_\alpha)$
   (b) if $x_\alpha \notin G$ /* checkable in poly-time, because $G \in \mathcal{P}$ */
   then return /* $x_\alpha \notin G \Rightarrow x_\alpha \notin S \Rightarrow \varphi_\alpha \notin$ SAT */
   (c) if $x_\alpha \in B$ then return
   (d) Tree-Search($\alpha 0$)
   Tree-Search($\alpha 1$)
   (e) /* We reach here only if both calls in the previous step fail. */
   if $x_\alpha \in G$ then add x_α to B
   (f) return

End Algorithm.

Correctness: During the algorithm B maintains the invariant $B \subseteq G - S$. To see this note that x_α is added to B only if $x_\alpha \in G$ and we are backtracking. Since we are backtracking there are no satisfying assignments in α's subtree, so $x_\alpha \notin S$.

Note that if $x_\alpha \in S$ then $x_\alpha \in G$ ($S \subseteq G$) and $x_\alpha \notin B$ (because B maintains $B \subseteq G - S$). Therefore, if φ is satisfiable then we will find some satisfying assignment since for all nodes α on the path from the root to the appropriate leaf, $x_\alpha \in S$, and its sons are developed.

Complexity: To show that the time complexity is polynomial it is sufficient to show that only a polynomial portion of the tree is "developed". The following claim will yield the desired result.

Claim 4.2 *Let α and β be two nodes in the tree such that (1) neither is a prefix/ancestor of the other and (2) $x_\alpha = x_\beta$. Then it is not possible that the sons of both nodes were developed (in Step 3d).*

Proof: Assume we arrived at α first. Since α is not an ancestor of β we arrive at β after backtracking from α. If $x_\alpha \notin G$ then $x_\beta \notin G$ since $x_\beta = x_\alpha$ and we will not develop either. Otherwise, it must be that $x_\alpha \in B$ after backtracking from α. Therefore $x_\beta \in B$ and its sons will not be developed (see Step 3c). ■

Corollary 4.3 *Only a polynomial portion of the tree is "developed".*

Proof: There exists a polynomial $q(.)$ that time-bounds the Karp-reduction f. Since every x_α is obtained by an application of f, $x_\alpha \in \cup_{i \leq q(n)}\{0,1\}^i$. Yet G is sparse so $|G \cap (\cup_{i \leq q(n)}\{0,1\}^i)| \leq p(n)$ for some polynomial $p(\cdot)$.

Consider a certain level of the tree. Every two nodes α and β on this level are not ancestors of each other. Moreover on this level of the tree there are at most $p(n)$ different α's such that $x_\alpha \in G$. Therefore by the previous claim the number of x_α's developed forward on this level is bounded by $p(n)$. Therefore the overall number of nodes developed is bounded by $n \cdot p(n)$. ■

SAT $\in \mathcal{P}$ and the proof is complete. ■