

Complexity & Advanced Algorithms

Siddharth Bhat

Contents

1	NLogSpace-completeness	5
1.1	Co-NLogSpace	5
1.1.1	Solving $\overline{\text{PATH}}$ in NL	5
1.2	Oracles	6
1.2.1	P^{poly}	7
1.2.2	P^{poly} contains non-recursive languages	7
2	Advice & Time Hierarchies	9
2.1	P^{poly}	9
2.2	Unary language that is non-recursive	9
2.2.1	Sparse language	10
2.2.2	Cook reduction	10
3	Gaps in space and time	13
3.1	Space Hierarchy	13

Chapter 1

NLogSpace-completeness

1.1 Co-NLogSpace

$L \in \text{Co-NLogSpace} \equiv L^c \in \text{NLogSpace}$. That is, complement the language L . if L^c is in NLogSpace , then $L \in \text{Co-NLogSpace}$.

We intuitively believe that $\text{NP} \neq \text{Co-NP}$. However, we can show that $\text{NLogSpace} = \text{Co-NLogSpace}$.

$$\begin{aligned}\text{PATH} &= \{\langle G, u, v \rangle \mid \text{exists path between vertices } (u, v)\} \\ \overline{\text{PATH}} &= \{\langle G, u, v \rangle \mid \text{no path between vertices } (u, v)\}\end{aligned}$$

We assume that $\overline{\text{PATH}}$ is co-NL-Complete.

If we show that $\overline{\text{PATH}}$ is in NLogSpace , then every problem in co-NL will be in NL

1.1.1 Solving $\overline{\text{PATH}}$ in NL

$$\begin{aligned}V_R &\equiv \{\text{set of vertices reachable from } u\} \\ V_{NR} &\equiv \{\text{set of vertices } \mathbf{not} \text{ reachable from } u\}\end{aligned}$$

Sid confusion, why can't we use PATH as a subroutine: When we have an NDTM, we cannot *observe that the NDTM returns a 0*. We can *observe if an NDTM succeeds*, but there are weird paths and exponential number of paths where the NDTM does not return a 0? But if this is true, then how is PATH NL-complete? I am very confused.

To represent V_R and V_{NR} , we use 1 bit per vertex (since V_R and V_{NR} are disjoint), so total space is V .

Assume we know $|V_R|$. In this case, we can check whether v is unreachable from u — Enumerate all vertices. If they are reachable from u , bump up a counter. If we don't hit v till the counter gets to $|V_R|$, then what we know that is v is unreachable.

However, if v were reachable from u , then as we enumerate, we would find v as we were going through all vertices (we would not hit V_R unless we visit v).

This is important, because in an NDTM, if *any* of the paths accept, then we accept.

$$V_R = \cup_i V_R(i)$$

$$V_R(0) = \{u\}$$

to compute $cur \in? V_R(i+1)$, first **recompute** that $pred \in V_R(i)$, and then check that $(cur, pred) \in E(G)$. We cannot **store** $V_R(i)$, since we don't have enough space.

eventually we will reach $V_R(|V|)$, where we stop.

We can compute $|V_R| = \sum_i |V_R(i)|$. We compute $|V_R(i)|$ by checking over each vertex it's membership into $V_R(i)$. And if it does, we bump up our counter.

Reference: Read Sipser-Chapter 8

```
def belongs(G, i, startv, endv, curv):
    """Check if curv belongs to V_R(i)"""
    if i == 1:
        return startv == curv
    else:
        # log(V)
        for pred in G.vertices:
            # This can use a modified version of PATH that stores lengths?
            if small_belongs(G, i - 1, startv, endv, pred):
                if isneighbour(pred, curv):
                    return True

        return False

def countcard(G, startv, endv):
    """Count the cardinality of V_R"""
    card = 0
    # log(V)
    for i in len(G.vertices):
        # this is also log(V)
        for curv in G.vertices:
            if small_belongs(G, i, startv, endv, curv):
                card += 1
    return card
```

1.2 Oracles

For all inputs w of length $|w| = n$, there exists a **single** advice (a_n is allowed to be a single string that is polynomial in n). So, $a : \mathbb{N} \rightarrow \Sigma^*$, and the advice of a given input w is $a(|w|)$.

1.2.1 P^{poly}

$L \in \text{P}^{\text{poly}}$ if there is a polynomial time turing machine M which takes two inputs — a string $x \in \Sigma^*$, and an advice $a_n \in \Sigma^*$, such that for all inputs w such that $|w| = n$, then there exists a polynomial $p(n)$ with $|a_n| \leq p(|w|)$.

We force it to be polynomial in the word-length, because things like a lookup table take exponential space in the word-length (number of strings of length n is 2^n).

We can see that the advice is somewhat "hardwired" into the machine given the input length (since $a : \mathbb{N} \rightarrow \Sigma^*$). So, we have a sequence of machines $M_i : \mathbb{N} \rightarrow \{\text{Turing machines}\}$, and we instantiate the machine $M_{|w|}$ to check if $|w| \in L$.

NP is allowed to have a *varying witness*, while P^{poly} will have the *same* advice.

We don't even need to know if the advice string should be able to be found in polynomial time.

1.2.2 P^{poly} contains non-recursive languages

Chapter 2

Advice & Time Hierarchies

2.1 P^{poly}

This class could possibly be bigger than P.

In NP, witnesses are different for each string. In P^{poly} , witnesses are fixed for strings of a given length.

The advice string need to even be found in polynomial time!

Recursive language: Halts on all inputs with yes/no
Recursively enumerable: Halts and returns yes on inputs which belong to the language. On inputs that do not halt, undefined behavior.

2.2 Unary language that is non-recursive

L is a unary language $\equiv L \subseteq 1^*$

Theorem 1 *Every unary language is decidable by P^{poly}*

Proof. let L be a unary language.

Since the only characteristic of a string in a unary language is its length, for any given length, there is *at most one string of that length* in L . So, we can index the set L by the string lengths! Hence, the advice function allows us to build up a lookup table for *any* unary language.

We construct the advice function $a_L : \mathbb{N} \rightarrow \{0, 1\}$ be such that $a_L(n) = 1$ if 1^n belongs to L . Now, let M decide L as follows: $M(str) = a(|str|)$. Since we don't need to build a (it's an oracle we take for granted, the proof is done).

Theorem 2 P^{poly} contains non-recursive languages.

Proof. Let $L_{nr} \subset \{0, 1\}^*$ be a nonrecursive language. We define $L_w = \{1^{\#w} \mid w \in L_{nr}\}$, which is a unary language. A string $1^k \in L_w$ acts as a witness for the existence of some string $w \in L_{nr}$ as the lex-ordering-position of the string w .

Example of $\#$ evaluated on some strings

$\#0 \rightarrow 0$
 $\#1 \rightarrow 1$
 $\#00 \rightarrow 3$
 $\#01 \rightarrow 4$
 $\#100 \rightarrow 5$
 \dots

L_{nr} has now been reduced to L_u , since the mapping with $\#$ is a *bijection*. Also, L_u can be decided by P^{poly} . Hence, L_u can decide nonrecursive languages.

Question: Is the set $\{0, 1\}^*$ countable? It doesn't feel like it is!

2.2.1 Sparse language

A **sparse language** is one where the number of strings of length n is bounded by a polynomial. $|L \cap \{0, 1\}^n| \leq p(n)$.

Idle thought: Is there a classification theorem for sparse languages? "sparse-complete"

We study the relationship between NP and P^{poly} , using sparse languages.

2.2.2 Cook reduction

A language L_1 cook reduces to a language L_2 if there is a polynomial-time turing machine M_{L_1} that recognizes L_1 given oracle access to L_2 .

The machine M_{L_1} Can query membership to L_2 multiple times (polynomial) before deciding if a string $w \in L_1$.

Lemma 1 If L_1 Cook-reduces to L_2 and $L_2 \in P$, then $L_1 \in P$.

Proof. L_1 is decided by a polynomial-time turing machine M_{L_1} , so it can make at most polynomial queries to L_2 . Since $L_2 \in P$, There exists a polynomial-time turing machine M_{L_2} which solves the membership query.

The total running time for M_{L_1} is in P, so it can make at most polynomial queries to M_{L_2} . Hence, M_{L_1} can simulate M_{L_2} and solve the membership problem.

Theorem 3 Every language $L \in \text{NP}$ is Cook-reducible to a sparse language iff $\text{NP} \subseteq \text{P}^{\text{poly}}$.

This theorem is significant because we strongly believe that no NP -complete language is sparse! So, we believe that $\text{NP} \not\subseteq \text{P}^{\text{poly}}$.

Since SAT is NP -complete, we simply need to show that SAT is cook-reducible to a sparse language iff $\text{NP} \subseteq \text{P}^{\text{poly}}$.

We will exhibit polynomial-time advice string for all inputs of a given length, to use the power of P^{poly} .

Proof. (Forward) SAT Cook-reducible to a sparse language $L \implies \text{SAT} \in \text{P}^{\text{poly}}$

There is a polynomial-time machine M which can solve SAT given oracle access to sparse language L .

We want to show that SAT is in P^{poly} .

Let M run in time $p(n)$ on inputs of length n

The advice string $a(n)$ we want to give is the oracle behaviour on sparse language L . Since the machine M can ask for string of length at most $p(n)$.

Since the language is sparse, the set of all strings of a given length in L is polynomial. So, $a(n) = \text{concat}(\{w \in L \mid |w| \leq p(n)\})$ where *concat* concatenates all the strings. $a(n)$ will be polynomial in length since the length of each string w is bounded by $p(n)$. Let $\text{sparse}(n)$ be the polynomial that controls the sparsity of L for any string n . That is, for any length i , the language L contains at most $\text{sparse}(i)$ strings.

The total number of strings in $a(n)$ will be $N = \sum_{i=0}^{p(n)} \text{sparse}(i)$, which is a polynomial in n . Hence, $a(n)$ is a legal advice string.

We're done here, we converted oracle access to a sparse language into a polynomial advice string.

(Backward) $SAT \in P^{poly} \implies SAT$ Cook-reducible to a sparse language L

We are given a machine M_{sat} which seeks advice $a(n) : \mathbb{N} \rightarrow \{0, 1\}^*$. The machine M_{sat} runs for polynomial time $p_{sat}(n)$.

We need to construct a sparse language L_{sparse} , such that given oracle access to L_{sparse} , we can solve SAT using a new machine M' .

Consider all strings that are queried by M_{sat} to M_{poly} . For an input of length n , the machine M_{sat} can query a $p_{sat}(n)$ times at maximum. Hence, we the language consisting of the subset of a that is sampled by M_{sat} is a sparse language. Given access to this language, we can substitute the function a with the sparse language which contains all advice accessed from a .

Chapter 3

Gaps in space and time

We wish to study what is not computable given some resource. If there resource is time, we want to understand what can be solved in $t(n)$ but not in smaller than $t(n)$ — in the sense of $o(t(n))$.

We can try to construct a hierarchy of problems that can be solved given increasing time.

$$f(n) \in o(g(n)) \equiv \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$
$$f(n) \in O(g(n)) \equiv \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in O(1)$$

3.1 Space Hierarchy

A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is said to be **space constructible** if there exists a turing machine that on input 1^n , it computes $f(n)$ using space $O(f(n))$. So the output can be $1^{f(n)}$ say, since that uses space $O(f(n))$.

Most common functions such as polynomials, exponentials, and logarithms are all space constructible.

Theorem 4 *Let f be a space-constructible function. There exists a language L which can be decided in $O(f(n))$ space, but not in $o(f(n))$ space.*

Proof. The proof is to **construct** a language which can be decided on $O(f(n))$ space, but not in $o(f(n))$ space. Such a language tends to be artificial due to the construction having to work for all f .

We need two properties for this language L we create:

- It is **not decidable** in $o(f(n))$ space.
- It **is** decidable in $O(f(n))$ space.

We will use diagonalization to show an construct an L that **cannot be decided** in $o(f(n))$ space. List each TM that runs in $o(f(n))$ space. This collection of all TMs (viewed as strings) is written as:

$$ALLTM = \cup_{i=0}^{\infty} \{0, 1\}^i$$

We will define a language L which cannot be decided by **any** TM on the above list.

We will create a matrix of the form $DECIDE(i, j) = M_i(\langle M_j \rangle)$. That is, we feed M_i the string of M_j . ($\langle M_j \rangle$ interprets the machine M_j as a string).

Now, create a language L :

$$L \equiv \{M \mid M(\langle M \rangle) = 0\}$$

Note that L is **not decidable** in $o(f(n))$ space. Proof by contradiction: Assume such a machine M_{contra} exists. We now ask if $\langle M_{contra} \rangle \in L$?

If $\langle M_{contra} \rangle \in L$, then $M_{contra}(\langle M_{contra} \rangle) = 0$. But since M_{contra} **decides** L , $M_{contra}(\langle M_{contra} \rangle) = 0 \implies \langle M_{contra} \rangle \notin L$.

On the other hand, say that $\langle M_{contra} \rangle \notin L$, then $M_{contra}(\langle M_{contra} \rangle) = 1$. But since M_{contra} **decides** L , $M_{contra}(\langle M_{contra} \rangle) = 1 \implies \langle M_{contra} \rangle \in L$. This is also a contradiction.

We now move to show that L **can be decided** in $O(f(n))$ space. Consider a machine INTERPRET that does this:

```
def INTERPRET(w):
    Mw = convert_to_TM(w)

    # Naive solution: Try to run Mw, see what happens.
    # flag = Mw.run(w)

    # Problem 1: How do we know it runs in o(f(n)) space?
    # flag = Mw.run_with_bounded_space(w, space_bound=f(n))

    # Problem 2: How do we know that Mw halts?
    # Count the size of the config. space, and reject if Mw
    # takes more steps than the configuration space size.
    flag = Mw.run_wth_bounded_space_and_steps(w, space_bound=f(n),
                                              steps_bound=Mw.config_space_size())

    return !flag
```