Space Complexity

Last Updated April 11th, 2024

1 Introduction

Definition 1.1. Let Turing machine M be given.

> If M is deterministic, then $\mathbf{space}_M(x)$ denotes the number of different tape cells visited by M's head during the computation of M on input x.
>
> If M is nondeterministic, then $\mathbf{space}_M(x)$ denotes the maximum number of different tape cells visited by M's head along any branch of the computation of M on input x.

Note that $\text{space}_M(x)$ is undefined if M does not halt on input x along any of its branches.

Definition 1.2. Given deterministic Turing machine M that halts on all inputs, its **space complexity** is the function $f : \mathcal{N} \to \mathcal{N}$ for which

$$f(n) = \max_{|x|=n} \text{space}_M(x).$$

In other words, for given input size n, $f(n)$ gives the worst-case number of tape cells used by M on any input of size n.

Note that the space complexity of M is well defined since we assume M halts on all inputs.

1.1 Complexity Classes

Definition 1.3. Let $f : \mathcal{N} \to \mathcal{N}$ be a function. Then

$\texttt{SPACE}(f(n))$ is the set of all decision problems L that are decidable by a deterministic Turing machine whose space complexity is $\mathrm{O}(f(n))$.

$\texttt{NSPACE}(f(n))$ is the set of all decision problems L that are decidable by a nondeterministic Turing machine whose space complexity is $\mathrm{O}(f(n))$.

Example 1.4. The SAT decision problem is a member of SPACE(n), where $n = |F|$ is the size of the Boolean formula input $F(x_1, \ldots, x_n)$. This is true since we need O($|F|$) bits to hold a variable assignment α and O($|F|$) bits for a stack that is used to evaluate $F(\alpha)$.

The following results often prove useful when studying the relationship between space and time complexity.

Lemma 1.5. Let M be a deterministic Turing machine that halts on all inputs.

1. If M has time complexity $\texttt{DTIME}(t(n))$, then it also has space complexity $\mathrm{O}(\max(n, t(n)))$.
2. If deterministic Turing machine M halts on all inputs and has space complexity $\mathrm{O}(f(n))$, then its time complexity is $2^{\mathrm{O}(f(n))}$.

Proof. The first statement is true since the amount of space that a DTM uses for a given computation is always at least the input size n (unless the machine makes use of an additional read-only input tape, see next section) and at most one more than the number of computation steps (assuming that a single step moves from an existing configuration to the next).

For the second statement, observe that any deterministic Turing machine M that on some input reaches a non-terminating configuration more than once can never halt on that input (why?). Moreover, if $|M|$ has $\mathrm{O}(f(n))$ space complexity then there is a constant $c > 0$ such that, for sufficiently large n, M uses no more than $cf(n)$ tape cells when deciding any input of size n. This in turn implies that, during M's computation on some input x of size n, M can enter at most

$$(c \cdot f(n))|Q||\Gamma|^{cf(n)}$$

different configurations, where

1. $c \cdot f(n)$ bounds the number of possible head locations
2. $|Q|$ the number of states, and
3. $|\Gamma|^{cf(n)}$ bounds the number of different tape words that appear on the tape.

Therefore, if M is to halt on input x, M must finish its computation within

$$(c \cdot f(n))|Q||\Gamma|^{cf(n)} = 2^{\mathrm{O}(f(n))}$$

steps and therefore has time complexity $2^{\mathrm{O}(f(n))}$. □

1.2 Savitch's Theorem

Theorem 1.6. For any function $f : \mathcal{N} \to \mathcal{R}^+$, where $f(n) \geq n$,

$$\texttt{NSPACE}(f(n)) \subseteq \texttt{SPACE}(f^2(n)).$$

Proof. Before presenting the key idea behind the proof, we first consider the naive approach that, for NTM M, simulates the computation $M(x)$ by performing a depth-first traversal of the computation tree $T(M, x)$ of $M(x)$. Such a simulation would require the use of a configuration stack for keeping track of which option was exercised in each nondeterministic step. For example, if a configuration node of $T(M, x)$ has three children, then we need to keep track of the current child being explored. The problem with this approach is that, by Lemma 1.5, since M can take up to $2^{\mathrm{O}(f(n))}$ steps, all of which could possibly be nondeterministic, the stack could reach a size of $2^{\mathrm{O}(f(n))}$ configurations which requires more than $\omega(f(n))$ amount of space.

Now for the proof. We prove a slightly weaker version of the theorem by assuming that the function $f(n)$ outputs positive natural numbers and is computable in $\mathrm{O}(f(n))$ space. Most space functions that we will ever encounter satisfy this condition. Now let M be an NTM that decides problem L in $\mathrm{O}(f(n))$ space. For simplicity, we also assume that, whenever M accepts an input word, it ends its accepting computation by erasing the tape and terminating in the accepting state q_a. We denote this "universal" final configuration as c_f. Moreover, if we fix input x, then we let c_0 denote the initial configuration for which x is the input word. By Lemma 1.5, if M accepts x, then it will accept it within $2^{cf(n)}$ steps, for some constant $c > 0$. We may also assume that, for this same $c > 0$ the computation of M on input x requires no more than $cf(n)$ amount of space. The key idea is to devise a deterministic recursive divide-and-conquer algorithm that decides if c_f is reachable by c_0 in no more than $2^{cf(n)}$ steps. We provide a semiformal description of the algorithm

Name: `reachable`

Inputs: length-$cf(n)$ configurations c_1 and c_2 and positive integer t which is a power of 2.

Output: 1 iff c_2 is reachable from c_1 in the computation tree $T(M, x)$ using t or fewer steps.

If $c_1 = c_2$, then return 1. //Base Case 1

If $t = 1$, then //Base Case 2

> If c_2 is reachable from c_1 when applying a single step of M's program to c_1, then return 1.
> Else return 0.

//Recursive Case

For each length-$cf(n)$ configuration $\hat{c}$,

> If $\texttt{reachable}(c_1, \hat{c}, t/2) \wedge \texttt{reachable}(\hat{c}, c_2, t/2)$, then return 1.

Return 0.

A DTM $\hat{M}$ may then use this function via the call `reachable`$(c_0, c_f, 2^{cf(n)})$. Furthermore, the depth of the recursion is

$$\log(2^{cf(n)}) = cf(n),$$

and so the recursion stack holds at most $cf(n)$ configurations, each having length $cf(n)$ for a total of $c^2 f^2(n) = \mathrm{O}(f^2(n))$ amount of memory. Therefore, L is decidable by a DTM that uses at most $\mathrm{O}(f^2(n))$ space. □

2 Context-Sensitive Grammars and NSPACE(n)

In this section we prove the Myhill-Landweber-Kuroda Theorem which states that the set of context-sensitive languages is equal to the set of languages that belong to NSPACE(n). In this and the next section we will encounter examples of non-deterministic algorithms that are described in pseudocode. The following are a few tips to keep in mind when reading non-deterministic pseudocode.

1. When following the steps of a nondeterministic algorithm, imagine that you are residing on a single branch of the computation tree.
2. The branch on which you reside may split into one or more subbranches after processing a step for which a call is made to a nondeterministic algorithm.
3. The branch on which you reside may split into one or more subbranches after processing a `guess` step. For example, given finite set $S = \{s_1, \ldots, s_n\}$, the step

   ```
   x = guess(S)
   ```

   has the effect of splitting your branch into n subbranches for which x assumes the value s_i on the i th branch, for all $i = 1, \ldots, n$.

Before encountering nondeterministic algorithms, we first review context-free grammars and then slightly modify the CFG definition to obtain a definition for context-sensitive grammars.

2.1 Context-Free Grammars

A **Context-Free Grammar (CFG)** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set of variables
2. Σ is a finite set that is disjoint from V, called the **terminal set**
3. R is a finite set of rules where each **rule** has the form

   $$A \to s,$$

   where $A \in V$ and $s \in (V \cup \Sigma)^*$. Variable A is referred to as the **head** of the rule, while s is referred to its **body**.
4. $S \in V$ is the **start variable**

Example 2.1. Consider the set of rules

$$R = \{S \to SS, S \to aSb, S \to \varepsilon\}.$$

Then we may use this set of rules to define a CFG $G = (V, \Sigma, R, S)$, where

$$V = \{S\},$$

$$\Sigma = \{a, b\},$$

and variable S is the start variable.

For brevity we may list together rules having the same head as follows.

$$S \to SS \mid aSb \mid \varepsilon.$$

Here, each of the rule bodies is separated by a vertical bar. □

Example 2.2. One common use of CFG's is to provide grammatical formalism for natural languages. For example, consider the set of rules R:

$$\langle \text{SENTENCE} \rangle \to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle$$

$$\langle \text{NOUN-PHRASE} \rangle \to \langle \text{COMPLEX-NOUN} \rangle \mid \langle \text{COMPLEX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle$$

$$\langle \text{VERB-PHRASE} \rangle \to \langle \text{COMPLEX-VERB} \rangle \mid \langle \text{COMPLEX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle$$

$$\langle \text{PREP-PHRASE} \rangle \to \langle \text{PREP} \rangle \langle \text{COMPLEX-NOUN} \rangle$$

$$\langle \text{COMPLEX-NOUN} \rangle \to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle$$

$$\langle \text{COMPLEX-VERB} \rangle \to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle$$

$$\langle \text{ARTICLE} \rangle \to \text{a} \mid \text{the}$$

$$\langle \text{NOUN} \rangle \to \text{trainer} \mid \text{dog} \mid \text{whistle}$$

$$\langle \text{VERB} \rangle \to \text{calls} \mid \text{pets} \mid \text{sees}$$

$$\langle \text{PREP} \rangle \to \text{with} \mid \text{in}$$

Here, the variables are the ten parts of speech delimited by $\langle\ \rangle$, Σ is the lowercase English alphabet, including the space character, and $\langle \text{SENTENCE} \rangle$ is the start variable.

Example 2.3. A CFG may also be used to define the syntax of a programming language. One fundamental language component to any programming language is that of an *expression*. The following rules imply a CFG for defining expressions formed by a single terminal `a`, parentheses, and the two arithmetic operations + and ×. Here E stands for `expression`, T for `term`, and F for `factor`.

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid a$$

We have $V = \{E, T, F\}$, $\Sigma = \{+, \times, a, (,)\}$, and E is the start variable.

Grammar derivations

Let $G = (V, \Sigma, R, S)$ be a CFG, then the language $D(G) \in (V \cup \Sigma)^*$ of **derived words** is structurally defined as follows.

Atom $S \in D(G)$.

Compound Rule Suppose $s \in D(G)$, s is of the form uAv for some $u, v \in (V \cup \Sigma)^*$, $A \in V$, and $A \to \gamma$ is a rule of G, then

$$u\gamma v \in D(G).$$

In this case we write $s \Rightarrow u\gamma v$, and say that s **yields** $u\gamma v$. In words, to get a new derived word, take an existing derived word and replace one of its variables A with the body of a rule whose head is A.

The subset $L(G)$ of derived words $w \in D(G)$ for which $w \in \Sigma^*$ is called the **context-free language (CFL)** associated with G. Thus, the words of $L(G)$ consist only of terminal symbols.

The Derivation relation Let u and v be words in $(V \cup \Sigma)^*$. We say that u **derives** v, written $u \stackrel{*}{\Rightarrow} v$ if and only if either $u = v$ or there is a sequence of words $w_1, w_2, \ldots, w_n$ such that

$$u = w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n = v.$$

Such a sequence is called a **derivation sequence** from u to v.

$$L(G) = \{w \in \Sigma^* | S \stackrel{*}{\Rightarrow} w\}.$$

Example 2.4. Use the CFG from Example 2.1 to derive the word `aabbaababb`.

$$S \to SS \mid aSb \mid \varepsilon.$$

Solution.

2.2 Context Sensitive Languages

Informally, a grammar is "context free" when any of its production rules can be applied regardless of the symbols that surround its head. On the other hand, a **context-sensitive grammer (CSG)** is defined in almost the same manner as a CFG, except that now each rule has the form

$$\alpha B \gamma \to \alpha \beta \gamma,$$

where $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$, but $\beta \neq \varepsilon$. Note that $S \to \varepsilon$ is allowed so long as S does not occur in the body of any rule.

Example 2.5. Consider the language

$$\{a^n b^n c^n | n \geq 1\}.$$

It can be shown that it is not a CFL, but it is a CSL via the following set of rules.

$$S \to ABC$$
$$S \to ASB'C$$
$$CB' \to Z_1 B'$$
$$Z_1 B' \to Z_1 Z_2$$
$$Z_1 Z_2 \to B' Z_2$$
$$B' Z_2 \to B' C$$
$$BB' \to BB$$
$$A \to a$$
$$B \to b$$
$$C \to c$$

Use this grammar to derive $a^2 b^2 c^2$.

Solution.

2.3 Monotone Grammars

Definition 2.6. A **monotone grammar** is a grammar whose rules are of the form $\alpha \to \beta$, where $\alpha, \beta \in (V \cup \Sigma)^*$ and $|\alpha| \leq |\beta|$. In addition, it may contain $S \to \varepsilon$ if S does not occur in the body of any rule.

Monotone grammars often seem more convenient to work with since the rules seem more flexible than the more structured CSG rules. Also, notice that every CSG is a monotone grammar, since rules of the form $A \to \varepsilon$ are not allowed. It turns out the converse is also true!

Theorem 2.7. Every monotone language can be derived by a context-sensitive grammar.

Example 2.8. Again, consider the language

$$\{a^n b^n c^n | n \geq 1\}.$$

The following set of rules constitutes a monotone grammar for deriving this language.

$$S \to abc$$
$$S \to aSBc$$
$$cB \to Bc$$
$$bB \to bb$$

Use this grammar to derive $a^3b^3c^3$.

Solution.

Theorem 2.9. (Myhill-Landweber-Kuroda Theorem) The set of context-sensitive languages equals $\texttt{NSPACE}(n)$.

Proof. We use Theorem 2.7 and show the equivalence between the set of monotone languages and $\texttt{NSPACE}(n)$. First consder a monotone language L derived from the monotone grammar $G = (V, \Sigma, R, S)$. Consider the following nondeterministic program for an NTM.

> Name: `invert`
>
> Input $w \in (V \cup \Sigma)^*$ and monotone grammmar $G = (V, \Sigma, R, S)$.
>
> Output: 1 if w can be derived from G.
>
> If $w = S$, then return 1.
>
> If there is no rule $\alpha \to \beta$ of G for which $w = \rho\beta\gamma$ for some $\rho, \gamma \in (V \cup \Sigma)^*$, then return 0.
>
> Guess a rule $\alpha \to \beta$ of G for which $w = \rho\beta\gamma$ for some $\rho, \gamma \in (V \cup \Sigma)^*$.
>
> Return $\texttt{invert}(\rho\alpha\gamma)$.

Clearly G derives w iff the computation of `invert` on input w has at least one accepting branch. Moreover, since each rule is monotone, the recursive call made by `invert` on input $\rho\alpha\gamma$ requires no more space than w itself, and hence $L \in \texttt{NSPACE}(n)$.

Next, given an NTM $N = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$, we first make the following simplifying assumptions.

1. On an input word w of length n, N uses at most $n + 1$ tape cells.
2. N starts by moving to the final input symbol $x \in \Sigma$ and replacing it with x', a "primed" version of x. Note: technically, $x' \in \Gamma$, but below we define the set Γ' to denote all primed versions of members of Γ.
3. The only way an accepting computation occurs is when the head moves left from cell $n + 1$ to cell n and N enters the accept state.

Note that no generality is lost by assuming 1) since, for every Turing machine M that uses at most cn tape cells, for some integer constant $c \geq 1$, there is another machine M' that uses only $n + 1$ cells and for which $L(M') = L(M)$.

We now provide a monotone grammar that derives $L(N)$. Its start variable is S and terminal set is Σ. As for its "variables", we caution the reader that some of the variables look more like strings of characters. For this reason we list them in the following different groups.

G1. S, A

G2. (x, y), where $x \in \Gamma \cup \Gamma'$, and $y \in \Sigma$, where Γ' is a copy of Γ but with each member "primed". For example, if $a \in \Gamma$, then $a' \in \Gamma'$.

G3. (q, x, y), where $q \in Q$, $x \in \Gamma \cup \Gamma'$, and $y \in \Sigma$

G4. (T, x, y), where T is a fixed symbol, $x \in \Gamma$, and $y \in \Sigma$

We now move to defining the rules. The first group of rules shown below is used to create the initial configuration. Note that the purpose of the second component of each ordered pair is to permanently store the original input symbol, since, to complete the derivation, the grammar must derive the original input word. For each $a \in \Sigma$, we have the following.

$S \to (q_0, a, a)A$

$A \to (a, a)A$

$A \to (a', a)$

$S \to (q_0, a', a)$ //In case the input word has length 1

$S \to \lambda$ //If $q_0 = q_a$ then $\lambda \in L(N)$

The second group of rules supports moving the tape head left. Indeed, suppose $(p, b, \mathrm{L}) \in \delta(q, a)$, then for each $c, e \in \Sigma$ and $d \in \Gamma$,

$(d, e)(q, a, c) \to (p, d, e)(b, c)$

$(d, e)(q, a', c) \to (p, d, e)(b', c)$

The third group supports moving the tape head right, and we leave it as an exercise to write these rules since they are analogous to those of Group 2.

The fourth group, upon reaching the accepting state, converts the tape-cell pairs (x, y) to triples (T, x, y), where $x \in \Gamma$ and $y \in \Sigma$. Namely, if $(q_a, b, \mathrm{R}) \in \delta(q, a)$, then, for each $c \in \Sigma$ we have

$$(q, a', c) \to (T, a, c),$$

along with, for each $b, d \in \Sigma$ and $a, c \in \Gamma$,

$$(a, b)(T, c, d) \to (T, a, b)(T, c, d).$$

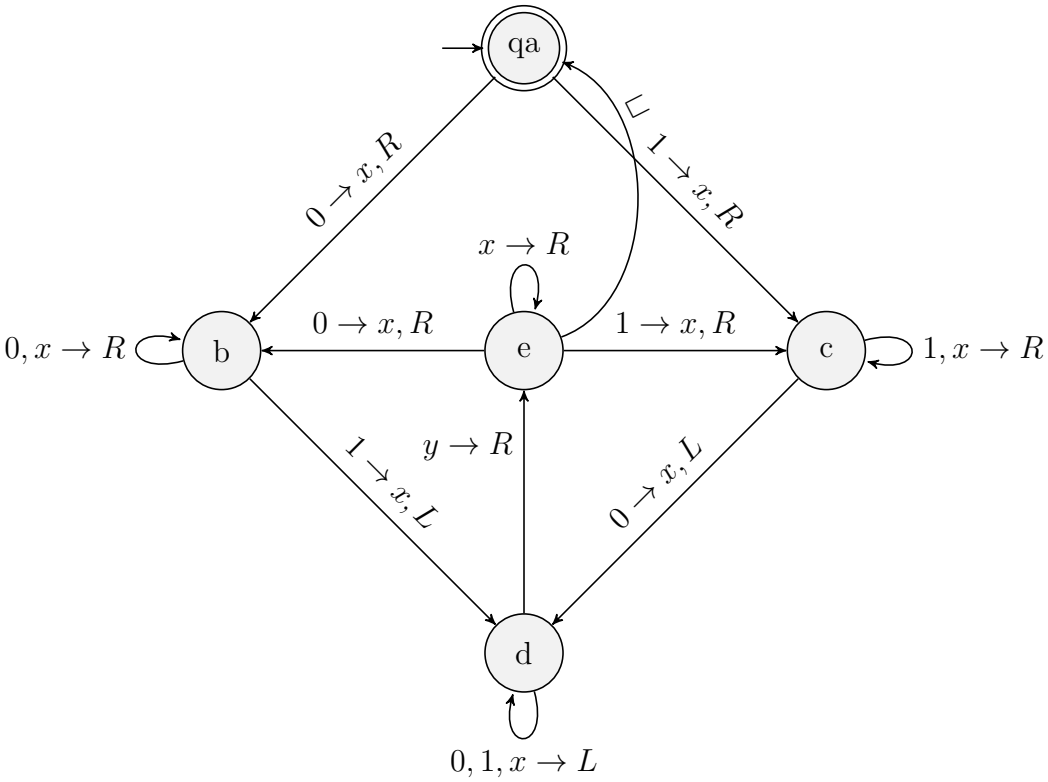
For the fifth and final group, up until now, assuming the derivation is mimicking an accepting computation branch, we have the derived word $(T, a_1, w_1) \cdots (T, a_n, w_n)$, where $w = w_1 \cdots w_n$ is the

input word. All that remains is to eliminate all symbols from each triple except for the input symbol. Thus, for all $a \in \Gamma$ and $b \in \Sigma$, we have

$$(T, a, b) \to b.$$

After some degree of reflection, it will hopefully seem clear that the defined monotone grammar will accept a word $w \in \Sigma^*$ iff $w \in L(N)$. $\square$

Example 2.10. Below is a state diagram for a Turing machine M that accepts all binary words having an equal number of zeros and ones and which satisfies the simplifying assumptions stated in the second half of the proof of Theorem 2.9. Use the monotone grammar provided in the second half of the proof of Theorem 2.9 to derive $w_1 = 1001$, and show that the grammar cannot derive $w_2 = 101$.



Solution.

3 NSPACE(n) = co-NSPACE(n)

In this section we prove the somewhat surprising result that NSPACE(n) = co-NSPACE(n). Whether or not these two space complexity classes were equal was an open question for nearly three decades before being independently resolved by both Neil Immerman and a Slovakian student named R. Szelepcsényi.

Theorem 3.1. (N. Immerman and R. Szelepcsényi) NSPACE(n) = co-NSPACE(n).

Solution Let language $A \in$ NSPACE(n) be given via NTM N. We must show that $\overline{A} \in$ NSPACE(n). For the moment, assume that there is a function $f : \mathcal{N} \to \mathcal{N}$ for which $f(n)$ equals the number of length-n words that belong to A, and that $f(n)$ may be nondeterministically computed in linear space. We use f to decide $\overline{A}$ in nondeterministic linear space using the following program.

Input w, where $n = |w|$.

Output: 1 if $w \in \overline{A}$.

Compute $f(n)$.

If $f(n)$ is undefined, then return 0.

$m = f(n)$.

count = 0.

For each $v \in \Sigma^n$,

Simulate N on input v.
If $N(v) = 1$, then
If $v = w$, then return 0. //$w \in A$
count = count + 1.
If count = m, then return 1. //$w \notin A$ and hence $w \in \overline{A}$

//The count never reached m, so this branch didn't acknowledge all words (possibly w) in A.

Return 0.

Notice that this algorithm computes in nondeterministic linear space, since both $f(n)$ and N can be nondeterministically simulated using linear space, and both m and the counter each require at most n bits.

3.1 Computing $f(n)$

What remains is to show how to compute $f(n)$ in nondeterministic linear space. By Theorems 2.7 and 2.9, $A \in \texttt{NSPACE}(n)$ implies that there is a monotone grammar $G = (V, \Sigma, R, S)$ that generates A. Define the set

$$D_i^n = \{w |\ w \in \Sigma^{\leq n} \text{ and } S \stackrel{i}{\Rightarrow}_G w\}$$

that represents all words w in $(V \cup \Sigma)^*$ having length at most n and for which w can be derived by G in at most i steps. Moreover, let

$$g(n) = \max_{i \geq 0} |D_i^n|.$$

Notice that $g(n)$ is total computable since, once there is an i_0 for which $|D_{i_0}^n| = |D_{i_0+1}^n|$, then it must be the case that $g(n) = |D_{i_0}^n|$ (why?). Also, $g(n)$ counts the number of words $w \in (V \cup \Sigma)^*$ for which $|w| \leq n$ and w can be derived from S. We leave it as (a relatively easy) exercise to show $f(n)$ can be computed in nondeterministic linear space, assuming the same is true for $g(n)$.

Now, notice that $g(n)$ may be computed in nondeterministic linear space with respect to size parameter n in case the same is true for computing $|D_i^n|$ for each $i \geq 0$. To complete the proof, the following is an algorithm for computing $|D_i^n|$. It relies on the algorithm `can_derive` which takes as input i) a word $w \in (V \cup \Sigma)^*$ and ii) a number of steps $i \geq 0$, and decides if w can be derived by G in no more than i steps. The algorithm is similar to the `invert` algorithm provided in the proof of Theorem 2.9, and so is left as an exercise.

Name: `num_derived`

Input: $i \geq 0$ and monotone grammmar $G = (V, \Sigma, R, S)$.

Output: $|D_i^n|$.

If $i = 0$, then return 1. //Only S can be derived in zero steps.

$m = |D_{i-1}^n| =$ `num_derived`$(i-1)$.

count $= 0$.

For each $w \in \Sigma^{\leq n}$,

count2 $= 0$.

For each $v \in \Sigma^{\leq n}$

`found` $= 0$.

If `can_derive`$(v, i-1)$,

count2 $=$ count2 $+ 1$.

If there is a rule $\alpha \to \beta \in R$ which, when applied to v, yields w,

`found` $= 1$

`break`.

If `found` $= 1$, then count $=$ count $+ 1$.

Else if count2 $\neq m$, //all possible parents of w not identified

Return UNDEFINED. //We may undercount the true value of $|D_i^n|$.

Return `count`.

Notice that, since `num_derived` is called immediately after checking the base case, there is really nothing to place on the call stack so long as i is globally stored (which is true for a Turing-machine implementation). Thus, the call stack requires zero space. Secondly, at any given moment the algorithm must store i along with two counters and two words w and v, all requiring O(n) space. Finally, notice that, similar to the `invert` algorithm from Theorem 2.9, the `can_derive` algorithm may be computed in nondeterministic linear space. Therefore, `num_derived` may be computed in nondeterministic linear space. □

4 Polynomial Space

Definition 4.1. PSPACE represents those decision problems that are decidable using a polynomial amount of space. In other words,

$$\texttt{PSPACE} = \bigcup_{k \geq 1} \texttt{SPACE}(n^k).$$

Proposition 4.2. The following inclusions hold.

$$\texttt{P} \subseteq \texttt{NP} \subseteq \texttt{PH} \subseteq \texttt{PSPACE} = \texttt{NPSPACE} \subseteq \texttt{EXPTIME}.$$

Proof. The first inclusion is Theorem 4.9 of the Complexity lecture. The second inclusion follows from the definition of PH. The equality PSPACE = NPSPACE follows from Savitch's theorem, and the final inclusion follows from Lemma 1.5. All that remains to show is that PH $\subseteq$ PSPACE. To this end, let $L \in \Sigma_k^p$ be given, and let $Q_1 x_1 \cdots Q_k x_k \; p(x_1, \ldots, x_k, y)$ be the predicate function associated with L. Consider the following recursive algorithm for evaluating L's predicate function.

Name: `eval`

Inputs:

1. instance y of decision problem L,
2. (possibly empty) list $L = [c_1, c_2, \ldots, c_l]$, $l \geq 1$, $c_j \in C_j = \text{dom}(x_j)$, $1 \leq j \leq l$.

Output: $Q_{l+1} x_{l+1} \cdots Q_k x_k \; p(c_1, \ldots, c_l, x_{l+1}, \ldots, x_k, y)$.

If $l = \text{length}(L) = k$, then return $p(c_1, \ldots, c_k, y)$.

If $l + 1$ is odd, then $//Q_{l+1} = \exists$

    For each $d \in C_{l+1}$,
        If `eval`$(y, L + d) = 1$, then return 1.
    Return 0.

Else $//Q_{l+1} = \forall$

    For each $d \in C_{l+1}$,
        If `eval`$(y, L + [d]) = 0$, then return 0.
    Return 1.

Since predicate function p is decidable in polynomial-time, it is also decidable using a polynomial amount of space. Therefore, the base is computable in a polynomial amount of space. As for the recursive cases, notice that the required memory consists of i) at most k counters, each having $\text{O}(q(|y|)$ bits and ii) list L whose size is also $\text{O}(q(|y|)$ which is a bound for each of the at most k certificates that are stored in L at any given time. Therefore, the algorithm requires a polynomial amount of space. □