Foundations of Computer Science
Logic, models, and computations

Chapter: Space complexity

Course CSC_41012_EP

of l'Ecole Polytechnique

Olivier Bournez

bournez@lix.polytechnique.fr

Version of July 11, 2025

Space complexity

In this chapter, we will focus on another critical resource of algorithms: Memory. In complexity theory, when talking about memory as a resource, it is more commonly called *memory space* or simply *space*.

We will start by showing how one can measure the memory used by an algorithm. We will then introduce the main complexity classes considered in complexity theory.

1 Polynomial space

In this section, we will state a set of definitions and results without proofs. The proofs will be given in next section.

We introduce the analog of TIME(t(n)) for memory:

Definition 1 (SPACE(t(n))**)** *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define* SPACE(t(n)) *as the class of problems (languages) that are decided by a Turing machine using $\mathcal{O}(t(n))$ cells of the tape, where n is the size of the input.*

1.1 Class PSPACE

We first consider the class of problems decided using polynomial space.

Definition 2 PSPACE *is the class of problems (languages) decided in polynomial space. In other words,*

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k).$$

Remark 3 *As in Chapter 12, we observe that the notion of space is independent of the computational model we use. Consequently the use of the Turing machine model as the basis model for measuring space complexity is rather arbitrary in what follows.*

We can also introduce the non-deterministic analog:

Definition 4 (NSPACE(t(n))) *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define* NSPACE(t(n)) *as the class of problems (languages) that are accepted by a non-deterministic Turing machine using $\mathcal{O}(t(n))$ cells of the tape on every branch of the computation tree, where n is the size of the input.*

It would then natural to define:

$$\text{NPSPACE} = \bigcup_{k \in \mathbb{N}} \text{NSPACE}(\text{n}^k),$$

but it turns out that the complexity class NPSPACE is nothing but PSPACE.

Theorem 5 (Savitch theorem) NPSPACE = PSPACE.

The proof of this result can be found in Section 23.

1.2 PSPACE-complete problems

The class PSPACE has some complete problems: The problem QBF (sometimes also called QSAT) consists, given some propositional calculus formula in conjunctive normal form ϕ with the variables $x_1, x_2, \cdots, x_n$ (that is to say given an instance similar to an instance of SAT), to determine whether $\exists x_1 \forall x_2 \exists x_3 \cdots \phi(x_1, \cdots, x_n)$?

Theorem 6 *The* QBF *problem is* PSPACE*-complete.*

We will not prove this result in this document.

Strategic games on graphs lead natural birth to PSPACE-complete problems.

For example, the game GEOGRAPHY consists in taking as input a finite oriented graph $G = (V, E)$. Player 1 selects a node u_1 of the graph. The player 2 must then select a node v_1 such that there is some arc from u_1 to v_1. This is then the turn of player 1 to select another node u_2 such that there is an arc from v_1 to u_2, and so one. One does not have the right and so on. We don't have the right to come back twice to the same node. The first player that cannot continue the path $u_1 v_1 u_2 v_2 \cdots$ loses. The problem GEOGRAPHY consists, given some graph G and a node for player 1, in determining if there exists some winning strategy for player 1.

Theorem 7 *The problem* GEOGRAPHY *is* PSPACE*-complete.*

2 Logarithmic space

It turns out that the class PSPACE is huge and contains all of P and also all NP: Imposing a polynomial memory space is practice often little restrictive.

This is why one often considers more restictive space bounds, in particular logarithmic space. But this introduces some difficulties which show a problem in the definitions. Indeed, a Turing machine uses at least the cells that contain its input, and hence Definition 1 is not able to talk about functions $t(n) < n$.

This is why one changes this definition with the following convention: When one measures the memory space, by convention one does not count the cells containing the input.

To do so, properly, one must replace Definition 1 by the following.

Definition 8 (SPACE(t(n))) *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define* SPACE(t(n)) *as the class of problems (languages) that are decided by a two tapes Turing machine:*

- *the first tape contains the input and is read-only: it can be read, but it cannot be written;*
- *the second is initially empty and read-write: it can be read and written, i.e., it is a usual tape;*

using $\mathcal{O}(t(n))$ cells of the second tape, where n is the size of the input.
We define NSPACE(t(n)) *with the analogous convention.*

Remark 9 *This new definition does not change anything to all for the previously introduced complexity classes. However, it the the following definitions meaningful.*

Definition 10 (LOGSPACE) *The class* LOGSPACE *is the class of languages (problems) decided by a Turing machine in logarithmic space. In other words,*

$$\text{LOGSPACE} = \text{SPACE}(\log(n)).$$

Definition 11 (NLOGSPACE) *The class* NLOGSPACE *is the class of languages (problems) decided by a non-deterministic Turing machine in logarithmic space. In other words.*

$$\text{NLOGSPACE} = \text{NSPACE}(\log(n)).$$

It turns out that.

Theorem 12 $\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$.

We furthermore know that $\text{NLOGSPACE} \subsetneq \text{PSPACE}$ but we do not know which of the intermediary inclusions are strict.

3 Some results and their proof

This section is devoted to proving some basic results of complexity theory, in particular to the relations between time and space. Observe that Theorem 12 follows from the results below.

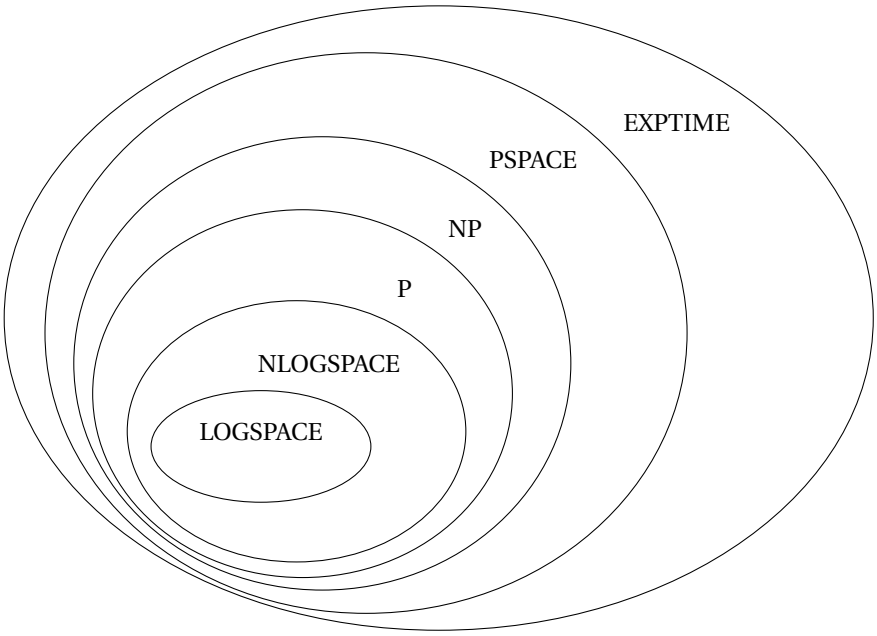

Figure 1: Inclusions between complexity classes

3.1 Preliminaries

In order not to uselessly complicate some of the proofs, we will restrict to functions $f(n)$ *of proper complexity*: We assume that the function $f(n)$ is non-decreasing, that is to say $f(n+1) \geq f(n)$, and such that there exists a Turing machine that takes as input w and that outputs $\mathbf{1}^{f(n)}$ in time $\mathcal{O}\left(n+f(n)\right)$ and in space $\mathcal{O}\left(f(n)\right)$, where $n = \text{length}(w)$.

Remark 13 *This is not really restrictive, since all the non-decreasing usual functions, such as* $\log(n)$, n, $n^2, \cdots, n\log n$, $n!$ *satisfy these properties.*

Remark 14 *We need this hypothesis, since function* $f(n)$ *could be not computable, and it could be impossible for example to write a word of length* $f(n)$ *in the coming proofs and algorithms.*

Remark 15 *In most of the following statements, one can avoid this hypothesis, but at the price of complications in the proofs that we will not discuss.*

3.2 Trivial relations

Since a deterministic Turing machine is a particular non-deterministic Turing machine, we have:

Theorem 16 SPACE(f(n)) ⊆ NSPACE(f(n)).

Furthermore.

Theorem 17 TIME(f(n)) ⊆ SPACE(f(n)).

Proof: A Turing machine writes at most one new cell at every step. The used memory space hence remains linear in the used time. Remember that the space taken by the input is not taken into account in the memory space. □

3.3 Non deterministic vs deterministic time

The following result if more interesting.

Theorem 18 *For every language in* NTIME(f(n)), *there exists an integer c such that this language is in* TIME($c^{f(n)}$). *If one prefers:*

$$\text{NTIME(f(n))} \subseteq \bigcup_{c\in\mathbb{N}} \text{TIME}(c^{f(n)}).$$

Proof: Let L be a problem in NTIME(f(n)). By using the principle that we used in previous chapter, we know that there exists a problem A such that to determine if a word w of length n is in L, it is sufficient to determine whether there exists a word $u \in \Sigma^*$ with $\langle w,u\rangle \in A$. This last test can be done in time $f(n)$, where $n = \text{length}(w)$. Since in time $f(n)$ one cannot read more than $f(n)$ letters from u, we can restrict to words u of length $f(n)$. Testing if $\langle w,u\rangle \in A$ for all the words $u \in \Sigma^*$ of length $f(n)$ is easily done in time $\mathcal{O}\left(c^{f(n)}\right) * \mathcal{O}\left(f(n)\right) = \mathcal{O}\left(c^{f(n)}\right)$, where $c > 1$ is the size of the alphabet Σ of the machine: Generating all words u of a given length, here $f(n)$), can be done for example by counting in base c. □

Remark 19 *To write the first u to be tested of length $f(n)$, we implicitly use the fact that this is feasible: This is the case if we assume $f(n)$ to be of proper complexity. We see here the interest of this (implicit) hypothesis. We will avoid discussing these type of problems in what follows, because they do not arise for usual functions $f(n)$.*

3.4 Non-deterministic time vs space

Theorem 20 NTIME(f(n)) ⊆ SPACE(f(n)).

Proof: We use exactly the same principle as in the previous proof, with the only difference that we are talking about space. Let L be a problem in NTIME(f(n)). By using the same idea as before, we know that a problem A such that to determine whether a word w of length n is in L, it is sufficient to know whether there exists $u \in \Sigma^*$ of length $f(n)$ with $\langle w,u\rangle \in A$: We use space $\mathcal{O}\left(f(n)\right)$ to generate one after

the other the words $u \in \Sigma^*$ of length $f(n)$ (for example by counting in base c) and then test for each of them if $\langle w, u\rangle \in A$. This last test can be done in time $f(n)$, hence space $f(n)$. The same space can be used for each of the words u. The space use is $\mathcal{O}\left(f(n)\right)$ for writing the u's plus $\mathcal{O}\left(f(n)\right)$ for the tests. So this takes space $\mathcal{O}\left(f(n)\right)$ in total. □

3.5 Non-deterministic space vs time

The decision problem REACH will play an important role: Given a directed graph $G = (V, E)$, two vertices u and v, one wants to decide if there exists a path between u and v in G. It is easy to see that REACH is in P.

To every (deterministic or not) Turing machine we associate a directed graph, its *configuration graph,* where the vertices correspond to configurations and whose arcs correspond to the one-step evolution function of the machine M, that is to say to relation $\vdash$ between configurations.

Every configuration X can be described by a word $[X]$ on the alphabet of the machine M: If the input w of length n is fixed, for a computation in space $f(n)$, there are less than $\mathcal{O}\left(c^{f(n)}\right)$ vertices in this graph G_w, where $c > 1$ is the size of the alphabet of the machine.

A word w is accepted by the machine M if and only if there is a path in this graph G_w between the initial configuration $X[w]$ encoding the input w, and an accepting configuration. We may assume without loss of generality that there is a unique accepting configuration X^*. Deciding the containment of a word w in the language recognized by M is consequently solving the problem REACH on $\langle G_w, X[w], X^*\rangle$.

We will translate in various forms all that is done on problem REACH. First, it is clear that the problem REACH can be solved in time and space $\mathcal{O}\left(n^2\right)$, where n is the number of vertices, by for example a depth-first search traversal.

We deduce:

Theorem 21 *If $f(n) \geq \log n$, then*

$$\mathrm{NSPACE(f(n))} \subseteq \bigcup_{c\in\mathbb{N}} \mathrm{TIME}(\mathrm{c}^{\mathrm{f(n)}}).$$

Proof: Let L be a problem of NSPACE(f(n)) recognized by a non-deterministic Turing machine M. By the previous discussion, one can determine whether $w \in L$ by solving decision problem REACH on $\langle G_w, X[w], X^*\rangle$: We said this can be done in time polynomial (quadratic) time on the number of vertices, hence in time $\mathcal{O}\left(c^{2\mathcal{O}(f(n))}\right)$, where $c > 1$ is the size of alphabet of the machine.

□

3.6 Non-deterministic space vs deterministic space

We will now show that REACH can be solved in space $\log^2(n)$.

Proposition 22 REACH $\in$ SPACE($\log^2$n).

Proof: Let $G = (V, E)$ be the directed graph given as input. Given two vertices x and y of this graph and an integer i, we write $PATH(x, y, i)$ if and only if there is a path of length than 2^i between x and y. We have $\langle G, u, v\rangle \in$ REACH if and only if $PATH(u, v, \log(n))$, where n is the number of vertices. It is hence sufficient to know how to decide the relation $PATH$ in order to decide REACH.

The trick is to compute $PATH(x, y, i)$ recursively by observing that we have $PATH(x, y, i)$ if and only if there is an intermediate vertex z such that $PATH(x, z, i-1)$ and $PATH(z, y, i-1)$. One tests then at each level of the recursion every possible vertex z.

To represent every vertex, $\mathcal{O}\left(\log(n)\right)$ bits are sufficient. To represent x, y, and i, one hence uses $\mathcal{O}\left(\log(n)\right)$ bits. The algorithm has a recursion of depth $\log(n)$, every level of the recursion requiring only to store a triple x, y, i and to test every z of length $\mathcal{O}\left(\log(n)\right)$. In total, we hence use space $\mathcal{O}\left(\log(n)\right) * \mathcal{O}\left(\log(n)\right) = \mathcal{O}\left(\log^2(n)\right)$. □

Theorem 23 (Savitch) *If $f(n) \geq \log(n)$, then*

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

Proof: We use the previous algorithm to determine if there is a path in graph G_w between $X[w]$ and X^*.

Observe that there is no need to explicitly construct the graph G_w but that one can use the previous algorithm *on-line*: Instead of writing down the graph G_w completely, and then reading in this encoding of the graph if there is an arc between a vertex X and a vertex X', one can in a lazy way, by recompute this information, determine every time that a test is needed whether $X \vdash X'$. □

Corollary 24 PSPACE = NPSPACE.

Proof: We have $\bigcup_{c\in\mathbb{N}}$ SPACE(n^c) $\subseteq \bigcup_{c\in\mathbb{N}}$ NSPACE(n^c) by Theorem 16, and $\bigcup_{c\in\mathbb{N}}$ NSPACE(n^c) $\subseteq$ $\bigcup_{c\in\mathbb{N}}$ SPACE(n^{2c}) $\subseteq \bigcup_{c\in\mathbb{N}}$ SPACE(n^c) by the previous theorem. □

4 Separation results

4.1 Hierarchy theorems

We say that a function $f(n) \geq \log(n)$ is *space constructible*, if the function that maps $\mathbf{1}^n$ to $\mathbf{1}^{f(n)}$ is computable in space $\mathcal{O}\left(f(n)\right)$.

Most of the usual functions are space constructible. For example, n^2 is space constructible since a Turing machine can obtain n in binary by counting the number of **1**s, and writing n^2 in binary by using any method to multiply n with itself. The space used for this is certainly at most $\mathcal{O}\left(n^2\right)$.

Theorem 25 (Space Hierarchy theorem) *For every space constructible function $f : \mathbb{N} \to \mathbb{N}$, there exists a language L that is decided in space $\mathcal{O}\left(f(n)\right)$ but not in space $o(f(n))$.*

Remark 26 *We will prove only a version weaker than the statement above. The precise above theorem is a generalization of the following idea. The factor* log *comes from the construction of a univeral Turing machine really more efficience that the one considered in this document, introducing only a logarithmic time speeddown.*

Proof:

One considers the (very artificial) language L that is decided by the following Turing machine B:

- on an input w of size n, B computes $f(n)$ and reserves (marks) a space $f(n)$ for the coming simulation;
- If w is not of the form $\langle A \rangle 10^*$, for a Turing machine A, then the Turing machine B rejects.
- Otherwise, B simulates A on the word w for $c^{f(n)}$ steps to determine whether A accepts in space at most $f(n)$:
  - If A accepts in this time, then B rejects;
  - otherwise B accepts.

In other words, B simulates A on w, step by step, and decrements a counter c at each step. If this counter reaches 0 or if A rejects, then B accepts. Otherwise, B rejects.

By the existence of a universal Turing machine, there exist integers k and d such that L is decided in space $d \times f(n)^k$.

Suppose that L is decided by a Turing machine A in space $g(n)$ with $g(n)^k = o(f(n))$. There must exists an integer n_0 such that for $n \geq n_0$, we have $d \times g(n)^k < f(n)$.

As a consequence, the simulation of A by B will indeed be complete on every input of size n_0 or more.

Consider what happens when B is run on the input $\langle A \rangle 10^{n_0}$. Since this input is of size greater than n_0, B answers the opposite of Turing machine A on the same input. Hence B and A do not decide the same language, and hence Turing machine A does not decide L, which is a contradiction.

As a consequence L is not decidable in space $g(n)$ for any function $g(n)$ with $g(n)^k = o(f(n))$.

□

In other words:

Theorem 27 (Space Hierarchy theorem) *Let $f, f' : \mathbb{N} \to \mathbb{N}$ be two space constructible functions such that $f(n) = o(f'(n))$.*
Then the inclusion SPACE(f) $\subsetneq$ SPACE(f') *is strict.*

Using the same principle, one can prove.

Theorem 28 (Nondeterminstic Hierarchy theorem) *Let $f, f' : \mathbb{N} \to \mathbb{N}$ be two space constructible functions such that $f(n) = o(f'(n))$.*
Then the inclusion NSPACE(f) $\subsetneq$ NSPACE(f') *is strict.*

4.2 Applications

We deduce.

Theorem 29 NLOGSPACE $\subsetneq$ PSPACE.

Proof: The class NLOGSPACE is completely included in SPACE($\log^2$ n) by Savitch's theorem. But the latter is a strict subclass of SPACE(n), which is included in PSPACE. □

Analogously, we obtain.

Definition 30 *Let*
$$\text{EXPSPACE} = \bigcup_{c \in \mathbb{N}} \text{SPACE}(2^{n^c}).$$

Theorem 31 PSPACE $\subsetneq$ EXPSPACE.

Proof: The class PSPACE is completely included in, say, SPACE($n^{\log(n)}$). The latter is a strict subset of SPACE(2^n), that is in turn included in EXPSPACE. □

5 Exercices

Exercise 1 *(solution on page 242) Prove that if every* NP*-hard language is* PSPACE*-hard, then* PSPACE = NP.

6 Bibliographic notes

Suggested readings To go further on the notions in this chapter, we suggest [Sipser, 1997], [Papadimitriou, 1994] and [Lassaigne & de Rougemont, 2004].

A reference book on the last results of the field is [Arora & Barak, 2009].

Bibliography This chapter contains some standard results in complexity theory.
We mostly used their presentation in the books [Sipser, 1997] and [Papadimitriou, 1994].

Index

Bibliography

[Arora & Barak, 2009] Arora, S. & Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press. https://doi.org/10.1017/cbo9780511804090

[Lassaigne & de Rougemont, 2004] Lassaigne, R. & de Rougemont, M. (2004). *Logic and complexity*. Discrete Mathematics and Theoretical Computer Science. Springer. https://doi.org/10.1007/978-0-85729-392-3

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.