Space Complexity and LOGSPACE

Michal Hanzlík*

Abstract

This paper deal with the computational complexity theory, with emphasis on classes of the space complexity. One of the most important class in this field is LOGSPACE. There are several interesting results associated with this class, namely theorems of Savitch and Immerman – Szelepcsényi. Techniques that are used when working with LOGSPACE are different from those known from the time complexity because space, compared to time, may be reused. There are many open problems in this area, and one will be mentioned at the end of the paper.

1 Introduction

Computational complexity focuses mainly on two closely related topics. The first of them is the notion of complexity of a "well defined" problem and the second is the ensuing hierarchy of such problems. By "well defined" problem we mean that all the data required for the computation are part of the input. If the input is of such a form we can talk about the complexity of a problem which is a measure of how much resources we need to do the computation.

What do we mean by relationship between problems? An important technique in the computational complexity is a reduction of one problem to another. Such a reduction establish that the first problem is at least as difficult to solve as the second one. Thus, these reductions form a hierarchy of problems.

*Department of Mathematics, Faculty of Applied Sciences, University of West Bohemia in Pilsen, Univerzitní 22, Plzeň, mikeus@kma.zcu.cz

1.1 Representation

In mathematics one usually works with mathematical objects as with abstract entities. This is not the case of computational complexity. For our purposes we have to define how data will be represented to be able to find a feasible computational model that will handle them.

Definition 1.1. A string is a finite binary sequence. For $n \in \mathbb{N}$, we denote by $\{0,1\}^n$ the set of all strings of length n and call its elements the $n-$bit strings. The set of all strings is denoted by

$$\{0,1\}^* := \bigcup_{n\in\mathbb{N}} \{0,1\}^n.$$

For $x \in \{0,1\}^*$, we denote the length of x by $|x|$.

Definition 1.2. Let $S \subseteq \{0,1\}^*$. A function $g : \{0,1\}^* \longrightarrow \{0,1\}$ solves the decision problem of S if for every x it holds that $g(x) = 1$ if and only if $x \in S$.

From the definition of the decision problem it is clear that the function g is actually the characteristic function of the set S.

Closely related to a decision problem is a notion of a language.

Definition 1.3. A language is a set (possibly infinite) of strings.

1.2 Turing Machine

Before we give a detailed definition it might be useful to take a look at a brief overview. For that we will consider only deterministic Turing machines (TM). Our TM will act as *acceptor*, which means that it will accept, reject or loop on every input. In general TMs may have several read/write work tapes or dedicated input read-only and output write-only tapes but it can be proven that they all have equivalent computational capabilities as one-tape TM.

We will define a one-tape deterministic TM as a 9-tuple

$$M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r),$$

where

- Q is a finite set of *states*;

- Σ is a finite *input alphabet* (in our case $\Sigma = \{0, 1\}$);
- Γ is a finite *tape alphabet* and $\Sigma \subset \Gamma$;
- $\vdash \in \Gamma - \Sigma$ is the *left end-marker*;
- $\sqcup \in \Gamma - \Sigma$ is the *blank symbol*;
- $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 0, 1\}$ is the *transition function*;
- $s \in Q$ is the *start state*;
- $t \in Q$ is the *accept state*;
- $r \in Q$ is the *reject state* ($r \neq t$).

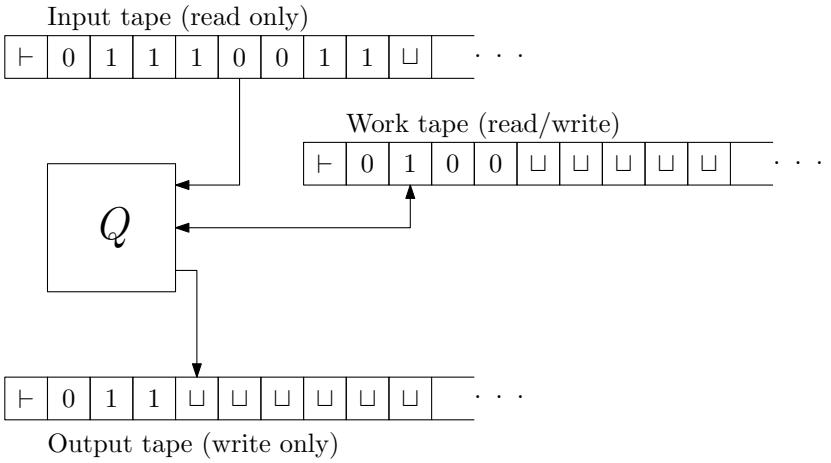

Figure 1.1: Schema of a three-tape Turing machine.

The most important of these is the transition function which describes how the TM processes the input. Formally it is defined as follows. For $p, q \in Q$, $a, b \in \Sigma$ and $d \in \{-1, 0, 1\}$ we have $\delta(p, a) = (q, b, d)$. This says that if the state is p and head is scanning symbol a then TM writes b, moves the head in direction d, and enters state q. There are two more criteria we require from the transition function. First the TM should never leave its work tape so for all $p \in Q$ there exist $q \in Q$ such that $\delta(p, \vdash) = (q, \vdash, 1)$. A second requirement is that whenever a TM enters an accept or a reject state it never leaves it so for all $b \in \Gamma$ there exist $c, c' \in \Gamma$ and $d, d' \in \{-1, 0, 1\}$ such that $\delta(t, b) = (t, c, d)$ and $\delta(r, b) = (r, c', d')$. At the beginning of every computation a TM is in state s (a start state). Then it may either enter an

infinite loop or end in one of the states t or r that stand for accept and reject, respectively. By entering these two states a TM halts the computation.

We are now going to describe what are deterministic and non-deterministic computational models. The difference between the former and the latter is in the transition function. The description above is of a deterministic TM. For a non-deterministic model the transition function δ is no longer a function but a relation $\Delta \subseteq (Q \backslash \{t, r\} \times \Gamma) \times (Q \times \Gamma \times \{-1, 0, 1\})$. This means that there is not a uniquely determined consecutive state but a finite set of possible next states. These choices create a rooted directed tree structure where root is a start state s (compare with an oriented path structure for a deterministic case). An accepting (resp. rejecting) computation of a non-deterministic TM is a path in the tree structure that starts in the root s and ends in one of the leafs of the tree that corresponds to the state t (resp. r). An input x (a finite string) is accepted by TM if there exists an accepting computation (TM halts when in state t).

1.3 Time Complexity

When using Turing machines as a model of computation we are able to measure the number of steps taken by the algorithm on each possible input. Such a function, denote it $t_A : \{0,1\}^* \longrightarrow \mathbb{N}$, is called the time complexity of algorithm A. To make this definition reasonable we will focus only on algorithms that halt on every input (i.e. for every input $x \in \{0,1\}^*$, $t_A(x)$ is finite number).

For our purposes we will be mainly interested in a dependence between the size of the input and number of steps of algorithm A. Thus, we will consider $T_A : \mathbb{N} \longrightarrow \mathbb{N}$ defined by $T_A(n) := \max_{x \in \{0,1\}^n} \{t_A(x)\}$. Notice that T_A represents time complexity of worst case input of length $n \in \mathbb{N}$ for the algorithm A.

So far we defined the time complexity of an algorithm. The *time complexity of a problem* is time complexity of "the fastest" algorithm that solves it. For two algorithms A and B, the functions T_A and T_B does not have to be comparable so we actually compare $O(T_A)$ and $O(T_B)$. It is obvious that the time complexity of a problem may depend on a model of computation (TM, RAM, ...).

1.4 Space Complexity

Another measure of efficiency is the use of space (or memory). A natural lower bound for time complexity is a linear function in size of the input (to process each element of the input at least once) but space can be reused during the computation so some of the most interesting space complexity classes are actually those using only a logarithmic amount of work space. Time and space efficiency measure is in conflict so one usually has to sacrifice[1] time when enhancing space complexity and the other way around.

The importance of space complexity is especially in the theoretical realm but we should note that there are several results which show a close connection between both criteria of complexity.

In subsection 1.2 we defined a TM as a model of computation and used it to measure the time complexity of an algorithm. To measure space complexity we will have to use a slightly more complicated computational model. This is mainly because we want to separate the sizes of input and output data from intermediate storage required by the computation. To do so we will use a 3-tape TM with one input tape which is read only, one output tape which is write only and a work tape which is read/write (see Figure Figure 1.1). We define the space complexity of a machine M on input $\{0,1\}^n$, denoted as $s_M(n)$, as a maximum of number of cells on a work tape used during a computation. Similarly the space complexity of an algorithm A will be defined as $S_A(n) := \max_{x\in\{0,1\}^n} \{s_A(n)\}$ and the space complexity of a problem will be again defined as a space complexity of the most space-efficient algorithm that solves it. Notice that as in case of time complexity a function in n which can be even a constant function.

We will further assume, when considering space complexity, that a TM in Figure Figure 1.1 never scans the input tape beyond the given input (i.e. there is a special symbol at the end of every input) and it also writes into each output tape cell at most once (this can be assured for a small additive penalty to the space complexity of a TM).

1.5 Complexity Classes

A complexity class is defined by three main criteria: model of computation (uniform or non-uniform), type of computational problem (decision, search,

[1] In the sense of substituting one resource for the other.

promise[2], etc.) and resource bound (function of input length). Most of the time we will consider Turing machines as a model of computation and we will work with decision problems. The resource bound criterion will be more interesting.

We recognize four general complexity classes. Denote by $T : \mathbb{N} \longrightarrow \mathbb{N}$ and $S : \mathbb{N} \longrightarrow \mathbb{N}$ two integer functions that we will use as a time and space complexity bound, respectively, and let $L(M)$ be the language accepted by TM M. We can now define

$$\begin{aligned}
\mathsf{DTime}(T(n)) &:= \{L(M) | M \text{ is a deterministic TM running in time } T(n)\}, \\
\mathsf{NTime}(T(n)) &:= \{L(M) | M \text{ is a non-deterministic TM running in time } T(n)\}, \\
\mathsf{DSpace}(S(n)) &:= \{L(M) | M \text{ is a deterministic TM running in space } S(n)\}, \\
\mathsf{NSpace}(S(n)) &:= \{L(M) | M \text{ is a non-deterministic TM running in space } S(n)\}.
\end{aligned}$$

1.5.1 Properties

The inclusions $\mathsf{DTime}(T(n)) \subseteq \mathsf{NTime}(T(n))$ and $\mathsf{DSpace}(S(n)) \subseteq \mathsf{NSpace}(S(n))$ are trivial and follow from the fact that a deterministic TM is a special case of a non-deterministic one.

The following theorem points out some basic relations between time and space complexity classes.

Theorem 1.4. *Let $T(n) \geq n$ and $S(n) \geq \log n$. Then*

$$\begin{aligned}
\mathsf{DTime}(T(n)) &\subseteq \mathsf{DSpace}(T(n)), \\
\mathsf{NTime}(T(n)) &\subseteq \mathsf{NSpace}(T(n)), \\
\mathsf{DSpace}(S(n)) &\subseteq \mathsf{DTime}(2^{O(S(n))}), \\
\mathsf{NSpace}(S(n)) &\subseteq \mathsf{NTime}(2^{O(S(n))}).
\end{aligned}$$

In the following text we will mainly focus on the space complexity classes as they are the main topic of the paper. One of the most interesting space complexity classes is the class $\mathsf{DSpace}(O(\log n))$, usually denoted as $\mathcal{LOGSPACE}$ (or $\mathcal{L}$). It contains a variety of natural computational problems that we will mention in the following chapter. Its non-deterministic equivalent is denoted as $\mathcal{NL}$. When considering space complexity, one should also mention class $\mathcal{PSPACE} = \cup_{c \in \mathbb{N}} \mathsf{DSpace}(n^c)$ and its non-deterministic variant

[2] A promise problem is a decision problem where the input is promised to belong to a subset of all possible inputs.

is called $\mathcal{NPSPACE}$. From the previously mentioned results follows that $\mathcal{NP} \subseteq \mathcal{PSPACE}$.

In 1970, a breakthrough in the study of space complexity was achieved by Walter Savitch [Sav70].

Theorem 1.5 (Savitch, 1970)**.** *Let* $S(n) \geq \log n$*. Then*

$$\mathsf{NSpace}(S(n)) \subseteq \mathsf{DSpace}(S(n)^2).$$

From Savitch's result it immediately follows that $\mathcal{PSPACE} = \mathcal{NPSPACE}$. Proving the similar equivalence for time complexity classes seems to be significantly more difficult.

Before we continue with the next result we define classes of complement problems.

Definition 1.6. Let $\mathcal{C}$ be a class of decision problems. The complement class denoted by *co*-$\mathcal{C}$ is a class of decision problems such that $S \in \mathcal{C}$ if and only if $\{0,1\}^* \backslash S \in$ *co*-$\mathcal{C}$.

Another important and more recent result was proven independently by Immerman and Szelepcsényi in 1987 [Imm88, Sze88]. It states the following.

Theorem 1.7 (Immerman, Szelepcsényi, 1987)**.** *For* $S(n) \geq \log n$*,*

$$\mathsf{NSpace}(S(n)) = \mathit{co}\text{-}\mathsf{NSpace}(S(n)).$$

Immerman and Szelepcsényi actually proved that $\mathcal{NL} = \mathit{co}\text{-}\mathcal{NL}$ but by using a "padding argument"[3] the result can be extended to any other space complexity non-deterministic class above $\mathcal{NL}$.

2 Logarithmic Space

Complexity classes using only logarithmic work space to process an input became a subject of interest because the space of size $\log n$ is just enough to maintain a counter that may store a number from 0 to n and hence it is possible to remember position of head on input and output tape. We actually allow usage of $O(\log n)$ space which means that the computational model may

[3]Padding is a technique for showing that if some complexity classes are equal, then some other classes possessing more computational resource are also equal by extending accepting language with new symbols. For details see [AB09].

have a constant number of counters counting from 0 to a polynomial in n. This is enough to solve a variety of natural problems such as adding and multiplying natural numbers. It also gives rise to a widely used reduction called a log-space reduction that is helpful when working with classes below $\mathcal{P}$.

Are there any problems that might be solved using sub-logarithmic amount of work space? A simple problem that requires only constant space is deciding if an integer in binary encoding is even or odd. It is quite surprising that there are problems that require sub-logarithmic space but will not do with constant space. The hierarchy of classes is as follows:

$$\mathsf{DSpace}(O(1)) \subsetneq \mathsf{DSpace}(O(\log\log n)) \subsetneq \mathcal{L} \subseteq \mathcal{P}.$$

The last inclusion follows from the fact that from Theorem 1.4 it holds that

$$\mathsf{DSpace}(S(n)) \subseteq \mathsf{DTime}(2^{O(n)})$$

and thus we have

$$\mathcal{L} = \mathsf{DSpace}(O(\log n)) \subseteq \mathsf{DTime}(2^{O(\log n)}) = \mathsf{DTime}(n^{O(1)}) = \mathcal{P}.$$

A similar relation holds also for the non-deterministic case (i.e., $\mathcal{NL} \subseteq \mathcal{NP}$). The question whether $\mathcal{L}$ is a proper subset of $\mathcal{P}$ or the equality holds is one of the most important open questions in complexity theory[4].

2.1 Composition Lemmas

There are two important composition lemmas that show how to compose Turing machines preserving the space bound restriction. For now assume we compose only two TMs where the second one uses the original input together with the output of the first TM as its input. An easy case is when output of first TM has length at most $\log n$, because we can emulate its output tape by additional work tape and this will still fulfill the space bound requirement. We call such composition a naive composition.

Lemma 2.1 (Naive composition)**.** *Let* $f_1 : \{0,1\}^* \longrightarrow \{0,1\}^*$ *and* $f_2 : \{0,1\}^* \times \{0,1\}^* \longrightarrow \{0,1\}^*$ *be computable in space* s_1 *and* s_2*, respectively. Then the function* f *defined by* $f(x) := f_2(x, f_1(x))$ *is computable in space* s *such that*

$$s(n) = s_1(n) + s_2(n + l(n)) + l(n), \tag{2.1}$$

where $l(n) = \max_{x\in\{0,1\}^n} \{|f_1(x)|\}$.

[4]The same question for non-deterministic classes (i.e., $\mathcal{NL} \subsetneq \mathcal{NP}$) is also unresolved.

Lemma 2.1 defines composition where no space optimization is used. The first term in (Equation 2.1) stands for the space required by the first TM that on input of size n uses $s_1(n)$ work space and produces output of size at most $l(n)$. The second TM has access to the original input as well as to the output produced by the first TM (which is also included into the space complexity of the composition) so the whole input for f_2 has size $n + l(n)$, thus the overall space required to compute f_2 is $s_2(n+l(n))$. One can achieve a simple improvement by reusing the work space of f_1. The space complexity of such a composition would be $s(n) = \max(s_1(n), s_2(n+l(n)))+l(n)$. For our purposes such an improvement would only decrease a constant in (Equation 2.1) and so we do not distinguish between these two approaches.

The second composition lemma is of major importance for us because it is one of the main tools for designing space bounded algorithms.

Lemma 2.2 (Emulative composition)**.** *Let f_1, f_2, s_1, s_2, l and f be as in Lemma 2.1. Then f is computable in space s such that*

$$s(n) = s_1(n) + s_2(n + l(n)) + O(\log(n + l(n))). \tag{2.2}$$

As an illustration of emulative composition consider a simple example. Let $f(e)$ be a function that assigns weights to the edges of a graph G that is acyclic with bounded degree and let s, t be vertices of G. Our task is to find a path of minimum weight between s and t. We obviously cannot remember values of function f for each edge of G because that would require space at least linear in the size of the input. So whenever an algorithm for finding a minimal path needs the weight of an edge e, the function $f(e)$ has to be invoked to provide such information. The function f is computed by P in Figure Figure 2.1 and the weights of the edges are exactly the information "stored" on the virtual device that together with the original input forms the input for Q.

Lemma 2.2 is described by Figure Figure 2.1. The terms $s_1(n)$ and $s_2(n)$ in (Equation 2.2) have exactly the same meaning as in Lemma 2.1. The last term is more interesting. It is the size of a constant number of auxiliary counters that store positions on the virtual tape. Each of the counters store a number from 0 to $n+l(n)$ so in the binary encoding one requires $\log(n+l(n))$ bits to store such a number and that is where the last term in (Equation 2.2) comes from. If Q requests a bit in a cell at position $\leq n$, it reads it from original input; otherwise, it asks P to recompute the requested bit. Hence we do not have to remember the output of P but every time Q asks for a part of its output, it is recomputed and we only need an auxiliary counter (of logarithmic size) to remember a position on the output tape of P. Such

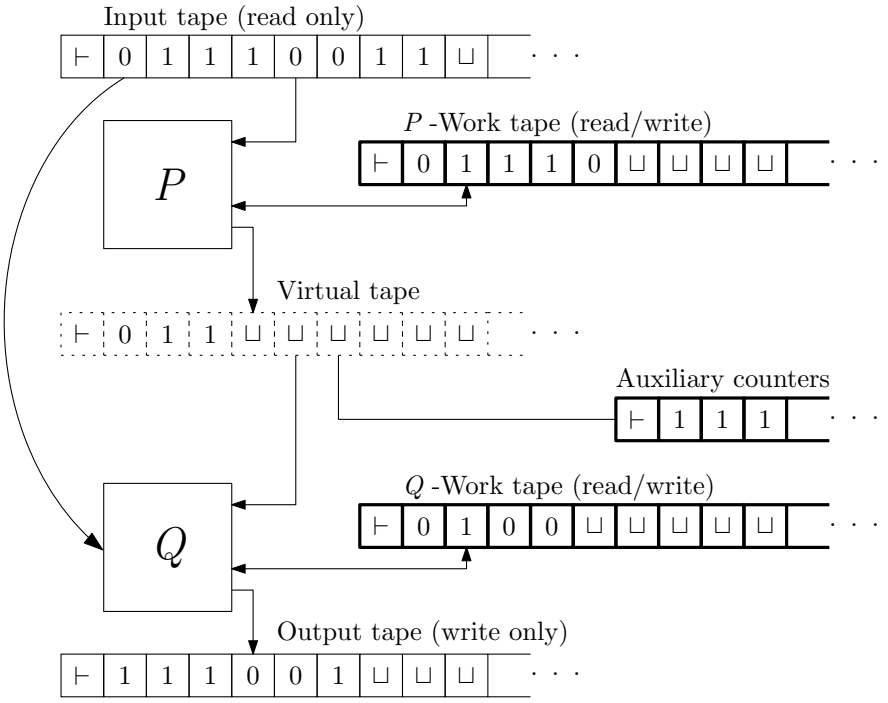

Figure 2.1: Emulative composition. Tapes shown in bold are considered when determining the space complexity of the composition. The virtual tape does not store any data, thus it does not occur in (Equation 2.2).

a virtual tape does not contain any data on its own so it is not included in the space complexity of the composition.

The main difference when applying such an approach is that we first call Q that itself makes calls to P whenever it requires additional information other than that in original input. It is an essence of most algorithms using logarithmic space because it removes the need to remember anything that can be computed from the original input. Even though it provides a very useful tool for log-space computations it has its limits. It is possible to do only a constant number of such compositions. For every emulative composition one has to add a constant number of counters and by performing, say, $\log n$ emulative compositions we end up with space complexity $\log^2 n$.

2.2 $\mathcal{NL}$ vs. $\mathcal{UL}$

$\mathcal{NL}$ is a class of problems accepted by non-deterministic logarithmic space bounded TMs. Non-determinism in space complexity has several features.

First, the class $\mathcal{NL}$ is closed under complement as states Theorem 1.7. It is widely believed that no such property holds for time complexity classes (especially $\mathcal{NP}$). Second, it is known that the whole $\mathcal{NL}$ is contained in $\mathsf{DSpace}(O(\log^2(n)))$ by Theorem 1.5.

The unambiguous logarithmic space complexity class (denoted $\mathcal{UL}$) is a class of decision problems solvable by a non-deterministic TM M such that

1. if an input $x \in L(M)$, exactly one computation path accepts,
2. if an input $x \notin L(M)$, all computation paths reject.

Unambiguity is a natural restriction of non-deterministic power. By the definition, $\mathcal{UL}$ is a restricted version of $\mathcal{NL}$ so a trivial inclusion is $\mathcal{UL} \subseteq \mathcal{NL}$. It is widely believed that whole non-determinism in logarithmic space is captured in $\mathcal{UL}$ and hence $\mathcal{UL} = \mathcal{NL}$.

Conjecture 2.3. $\mathcal{NL} = \mathcal{UL}$.

A fundamental problem contained in $\mathcal{NL}$ is the connectivity problem in general directed graphs denoted

$$\mathsf{STCONN} = \{G, s, t \mid \text{there is a directed path from } s \text{ to } t \text{ in } G\}.$$

Theorem 2.4. STCONN *is complete for* $\mathcal{NL}$ *under many-to-one log-space reductions.*

To show that $\mathcal{NL}$ is contained in another complexity class, it is enough to show that STCONN can be solved in it. Hence to prove Conjecture 2.3 one could show that $\mathsf{STCONN} \in \mathcal{UL}$.

3 Conclusion

In the first part of this paper we gave an overview of some basic facts from computational complexity theory. The next part was devoted to the logarithmic space complexity classes and to an open problem whether $\mathcal{NL} = \mathcal{UL}$. Many open problems remain in this field and it would be interesting to investigate them further.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[Gol08] Oded Goldreich. *Computational complexity: a conceptual perspective*. Cambridge University Press, 2008.

[Han12] Michal Hanzlík. *Spatial complexity of graph problems*. Diploma thesis, University of West Bohemia in Pilsen, 2012.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, October 1988.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, September 2008.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.