Notes on Computer Theory Last updated: September, 2015

Space complexity

Notes by Jonathan Katz, lightly edited by Dov Gordon

1 Space complexity

Recall that when defining the space complexity of a language, we look at the number of cells required on the *work tapes* of a turing machine that decides the language. In particular, we do not care about the number of cells required to represent the input or output of the computation: this will allow us to compute functions in sub-linear space. Just as in the case of time complexity, we will also consider the space complexity of non-deterministic turing machines. In direct analogy to time complexity, we consider the space of a non-deterministic machine to be the amount of space used in the worst-case over the choices of state transitions. We now define some of the important space-complexity classes we will study:

Definition 1

$$\begin{aligned} \mathsf{PSPACE} &\stackrel{\text{def}}{=} \bigcup_c \text{SPACE}(n^c) \\ \mathsf{NPSPACE} &\stackrel{\text{def}}{=} \bigcup_c \text{NSPACE}(n^c) \\ \mathsf{L} &\stackrel{\text{def}}{=} \text{SPACE}(\log n) \\ \mathsf{NL} &\stackrel{\text{def}}{=} \text{NSPACE}(\log n). \end{aligned}$$

We have seen that $\text{TIME}(t(n)) \subseteq \text{NTIME}(t(n)) \subseteq \text{SPACE}(t(n))$. What can we say in the other direction? To study this we look at *configurations* of a Turing machine, where a configuration consists of all the information necessary to describe the Turing machine at some instant in time. We have the following easy claim.

Claim 1 *Let M be a (deterministic or non-deterministic) machine using space $s(n)$. The number of configurations $\mathcal{C}_M(n)$ of M on any fixed input of length n is bounded by:*

$$\mathcal{C}_M(n) \leq |Q_M| \cdot n \cdot s(n) \cdot |\Sigma_M|^{s(n)}, \tag{1}$$

where Q_M are the states of M and Σ_M is the alphabet of M. In particular, when $s(n) \geq \log n$ we have $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$.

Proof The first term in Eq. (1) comes from the number of states, the second from the possible positions of the input head, the third from the possible positions of the work-tape head, and the last from the possible values stored on the work tape. (Note that since the input is fixed and the input tape is read-only, we do not need to consider all possible length-n strings that can be written on the input tape.) ■

Theorem 2 $\text{SPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$

Proof Let $L \in \text{SPACE}(s(n))$, and let M be a machine using space $O(s(n))$ and deciding L. Consider the computation of $M(x)$ for some input x of length n. There are at most $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$ configurations of M on x, but if $M(x)$ ever repeats a configuration then it would cycle and never halt. Thus, the computation of $M(x)$ must terminate in at most $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$ steps. ■

Definition 2 *A function $f : \mathbb{N} \to \mathbb{N}$ is* space constructible *if it is non-decreasing and there exists a Turing machine that on input 1^n outputs the binary representation of $f(n)$ using $O(f(n))$ space. Note that if f is space constructible, then there exists a Turing machine that on input 1^n marks off exactly $f(n)$ cells on its work tapes (say, using a special symbol) without ever exceeding $O(f(n))$ space.*

All functions you would "normally encounter" are space constructible; functions that aren't are specifically constructed counterexamples.
We can use this to obtain the following relationship between space and time:

Theorem 3 *Let $s(n)$ be space constructible with $s(n) \geq \log n$. Then* $\text{NSPACE}(s(n)) \subseteq \text{NTIME}(2^{O(s(n))})$.

Proof Let $L \in \text{NSPACE}(s(n))$. Then there is a non-deterministic Turing machine M deciding L and using space $O(s(n))$ on every computation path (i.e., regardless of the non-deterministic choices it makes). Consider a machine M' that runs M but only for at most $2^{O(s(n))} \geq \mathcal{C}_M(n)$ steps (and rejects if M has not halted by that point); this can be done using a counter of length $O(s(n))$ and so M' still uses $O(s(n))$ space. We claim that M' still decides L. Clearly if $M(x) = 0$ then $M'(x) = 0$. If $M(x) = 1$, consider the *shortest* computation path on which $M(x)$ accepts. If this computation path uses more than $\mathcal{C}_M(|x|)$ steps, then some configuration of M must repeat. But then there would be another sequence of non-deterministic choices that would result in a shorter accepting computation path, a contradiction. We conclude that $M(x)$ has an accepting computation path of length at most $\mathcal{C}_M(|x|)$, and so if $M(x)$ accepts then so does $M'(x)$. ■

The theorem above may seem to give a rather coarse bound for $\text{SPACE}(s(n))$, but intuitively it does appear that space is more powerful than time since space can be *re-used* while time cannot. In fact, it is known that $\text{TIME}(s(n))$ is a strict subset of $\text{SPACE}(s(n))$ (for space constructible $s(n) \geq n$), but we do not know much more than that. We conjecture that space is much more powerful than time; in particular, we believe:

Conjecture 4 $\mathcal{P} \neq \mathsf{PSPACE}$.

Note that $\mathcal{P} = \mathsf{PSPACE}$ would, in particular, imply $\mathcal{P} = \mathcal{NP}$.

2 Hierarchy Theorems

So far, we've been talking about how different classes of languages relate to one another. It is also natural to ask whether additional resources actually give additional power, within the same class of languages. For example, are all languages in $\mathcal{P}$ "equivalent," or are there some languages that require time n^2 while others require only n? We show that, with both space and time, more resources does mean more power (at least to a certain extent). We first show that more space gives more power.

Theorem 5 (Space hierarchy theorem) *Let $G(n) \geq \log n$ be space constructible, and $g(n) = o(G(n))$. Then* $\text{SPACE}(g(n))$ *is a proper subset of* $\text{SPACE}(G(n))$.

Proof We show the existence of a language L such that $L \in \text{SPACE}(G(n))$ but $L \notin \text{SPACE}(g(n))$. We define L by describing a Turing machine M_L, using space $O(G(n))$, that decides it. M_L does the following on input $w = (\langle M \rangle, x)$ of total length n:

1. Run $M(w)$ (i.e. run $M(\langle M \rangle, x)$) with at most $G(n)$ space and for at most $2^{2G(n)}$ steps (these bounds are imposed on M).
2. If $M(w)$ accepts within the given time and space bounds, then reject. Otherwise, accept.

In step 1, we can use the fact that G is space constructible to mark off exactly $G(n)$ tape cells for M to use. We can similarly mark off an additional $2G(n)$ cells to use as a counter for checking the number of steps M makes, and one last set of $G(n)$ cells to use for any remaining computation. By construction, M_L uses space $\tilde{G}(n) = 4 \cdot G(n)$.

We need to show that no machine using space $O(g(n))$ can decide L. Assume the contrary. Then there exists a machine M'_L deciding L and using space $\tilde{g}(n) = O(g(n))$. Choose k large enough so that 1) $\tilde{g}(k) < G(k)$, 2) M'_L makes fewer than $2^{G(k)}$ steps[1] on inputs of length k, and 3) the simulation of M'_L on inputs of length k can be performed in $G(k)$ space.[2] Consider the input $w = (M'_L, 1^k)$. If we run $M_L(w)$, recall that it runs $M'_L(M'_L, 1^k)$. Then (1) M_L has enough time and space to simulate the entire execution of $M'_L(w)$, and thus (2) $M_L(w)$ outputs the opposite of whatever $M'_L(w)$ outputs. We conclude that M_L and M'_L do not decide the same language. ■

We have a completely analogous time hierarchy theorem, though the result is quantitatively (slightly) weaker. We again need to define "nice" functions:

Definition 3 *A function $f : \mathbb{N} \to \mathbb{N}$ with $f(n) \geq n$ for all n is* time constructible *if it is non-decreasing and there exists a Turing machine that on input 1^n outputs the binary representation of $f(n)$ in $O(f(n))$ steps. Note that if f is time constructible, then there exists a Turing machine that on input 1^n runs for $O(f(n))$ steps and then halts.*

Theorem 6 (Time hierarchy theorem) *Let G be time constructible. If $g(n) \log g(n) = o(G(n))$, then* $\text{TIME}(g(n))$ *is a proper subset of* $\text{TIME}(G(n))$.

Proof The high-level structure of the proof is the same as in the proof of the previous theorem. We define L by giving a Turing machine M_L, using time $O(G(n))$, that decides it. M_L does the following on input $w = (M, y)$ of length $|w| = n$:

1. Run $M(w)$ using at most $c \cdot G(n)$ steps for some fixed constant c (see below).
2. If $M(w)$ accepts within the given time bound, then reject. Otherwise, accept.

We can implement step 1 using the fact that G is time constructible: in alternating steps, simulate $M(w)$ and run a Turing machine that is guaranteed to stop within $O(G(n))$ steps; halt the entire computation once the latter machine halts. We thus have that M_L runs in time $O(G(n))$.

We need to show that no machine using time $O(g(n))$ can decide L. Assume the contrary. Then there exists a machine M'_L deciding L in time $O(g(n))$. Consider an input of the form $w = (M'_L, 1^k)$. If we run $M_L(w)$ then, for k large enough, M_L has enough time to simulate the entire execution

[1] This condition is achievable because M'_L runs in time at most $O(n2^{O(g(n))})$, which is asymptotically smaller than $2^{2G(n)}$.

[2] This condition is achievable because universal simulation with constant space overhead is possible.

of $M'_L(w)$. (Here we use the fact that universal simulation is possible with logarithmic overhead.) But then, for k large enough, $M_L(w)$ outputs the opposite of whatever $M'_L(w)$ outputs. We conclude that M_L and M'_L do not decide the same language. ■

The barrier to getting a tighter time hierarchy theorem is the logarithmic time overhead in universal simulation. If a better simulation were possible, we would obtain a tighter separation.

3 Configuration Graphs and the Reachability Method

In this section, we will see several applications of the so-called *reachability method*. The basic idea is that we can view the computation of a non-deterministic machine M on input x as a directed graph (the *configuration graph* of $M(x)$) with vertices corresponding to configurations of $M(x)$ and an edge from vertex i to vertex j if there is a one-step transition from configuration i to configuration j. Each vertex in this graph has out-degree at most 2. (We can construct such a graph for deterministic machines as well. In that case the graph has out-degree 1 and is less interesting.) If M uses space $s(n) \geq \log n$, then each vertex in the configuration graph of $M(x)$ can be represented using $O(s(n))$ bits.[3] If we assume, without loss of generality, that M has only a single accepting state, then the question of whether $M(x)$ accepts is equivalent to the question of whether there is a path from the initial configuration of $M(x)$ to the accepting configuration. We refer to this as the *reachability* problem in the graph of interest.

3.1 NL and NL-Completeness

We further explore the connection between graphs and non-deterministic computation by looking at the class NL. Recall that a language is in NL if there is a non-deterministic machine that decides the language while never using more than $O(\log n)$ cells on its work-tapes, when starting with input of size n on its input tape. The input tape is read-only, and, because the machine decides the language, it does not require an output tape (i.e. a single accept state suffices).

As usual, we can try to understand NL by looking at the "hardest" problems in that class. Here, however, we need to use a more refined notion of reducibility. In defining a log-space reduction, as we do next, we will need to talk about functions that are computable in log space (rather than languages that are decidable in that space): in this case, the input tape remains read-only, the work tapes must never use more than $\log n$ storage, and the output tape is write-only. The output tape is not limited in storage size.

Definition 4 *L is* log-space reducible *to L' if there is a function f computable in space $O(\log n)$ such that $x \in L \Leftrightarrow f(x) \in L'$.*

Note that if L is log-space reducible to L' then L is Karp-reducible to L' (by Theorem 2); in general, however, we don't know whether the converse is true.

Definition 5 *L is NL-complete if (1) $L \in$ NL, and (2) for all $L' \in$ NL it holds that L' is log-space reducible to L.*

Log-space reducibility is needed[4] for the following result:

[3] Note that x is fixed, so need not be stored as part of a configuration. Whenever we construct an algorithm M' that operates on the configuration graph of $M(x)$, the input x itself will be written on the input tape of M' and so M' will not be "charged" for storing x.

[4] In general, to study completeness in some class $\mathcal{C}$ we need to use a notion of reducibility computable within $\mathcal{C}$.

Lemma 7 *If L is log-space reducible to L' and $L' \in \mathsf{L}$ (resp., $L' \in \mathsf{NL}$) then $L \in \mathsf{L}$ (resp., $L \in \mathsf{NL}$).*

Proof Let f be a function computable in log space such that $x \in L$ iff $f(x) \in L'$. The "trivial" way of trying to prove this lemma (namely, on input x computing $f(x)$ and then determining whether $f(x) \in L'$) does *not* work: the problem is that $|f(x)|$ may potentially have size $\omega(\log |x|)$ in which case this trivial algorithm uses superlogarithmic space. Instead, we need to be a bit more clever. The basic idea is as follows: instead of computing $f(x)$, we simply compute the i^{th} bit of $f(x)$ whenever we need it. In this way, although we are wasting time (in re-computing $f(x)$ multiple times), we never uses more than logarithmic space. ■

Consider the problem of *directed connectivity* (denoted CONN). Here we are given a directed graph on n-vertices (say, specified by an adjacency matrix) and two vertices s and t, and want to determine whether there is a directed path from s to t.

Theorem 8 CONN *is* NL*-complete.*

Proof To see that it is in NL, we need to show a non-deterministic algorithm using log-space that never accepts if there is no path from s to t, and that sometimes accepts if there is a path from s to t. The following simple algorithm achieves this:

if $s = t$ accept
set $v_{\text{current}} := s$
for $i = 1$ to n:
    guess a vertex v_{next}
    if there is no edge from v_{current} to v_{next}, reject
    if $v_{\text{next}} = t$, accept
    $v_{\text{current}} := v_{\text{next}}$
if $i = n$ and no decision has yet been made, reject

The above algorithm needs to store i (using $\log n$ bits), and at most the labels of two vertices v_{current} and v_{next} (using $O(\log n)$ bits).

To see that CONN is NL-complete, assume $L \in \mathsf{NL}$ and let M_L be a non-deterministic log-space machine deciding L. Our log-space reduction from L to CONN takes input $x \in \{0,1\}^n$ and outputs a graph (represented as an adjacency matrix) in which the vertices represent configurations of $M_L(x)$ and edges represent allowed transitions. (It also outputs $s = \mathsf{start}$ and $t = \mathsf{accept}$, where these are the starting and accepting configurations of $M(x)$, respectively.) Each configuration can be represented using $O(\log n)$ bits, and the adjacency matrix (which has size $O(n^2)$) can be generated in log-space as follows:

For each configuration i:
    for each configuration j:
        Output 1 if there is a legal transition from i to j, and 0 otherwise
        (if i or j is not a legal state, simply output 0)
Output start, accept

The algorithm requires $O(\log n)$ space for i and j, and to check for a legal transition. ■

We can now easily prove the following:

Theorem 9 *For $s(n) \geq \log n$ a space-constructible function,* $\text{NSPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$.

Proof We can solve CONN in linear time (in the number of vertices) using breadth-first search, and so $\text{CONN} \in \mathcal{P}$. By the previous theorem, this means $\mathsf{NL} \subseteq \mathcal{P}$ (a special case of the theorem).

In the general case, let $L \in \text{NSPACE}(s(n))$ and let M be a non-deterministic machine deciding L using $O(s(n))$ space. We construct a deterministic machine running in time $2^{O(s(n))}$ that decides L by solving the reachability problem on the configuration graph of $M(x)$, specifically, by determining whether the accepting state of $M(x)$ (which we may assume unique without loss of generality) is reachable from the start state of $M(x)$. This problem can be solved in time linear in the number of vertices in the configuration graph. ■

Corollary 10 $\mathsf{NL} \subseteq \mathcal{P}$.

Summarizing what we know,

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}.$$

By the hierarchy theorems (and Savitch's theorem, below) we know NL is a strict subset of PSPACE, and $\mathcal{P}$ is a strict subset of EXP. But we cannot prove that any of the inclusions above is strict.

3.2 Savitch's Theorem

In the case of time complexity, we believe that non-determinism provides a huge (exponential?) benefit. For space complexity, this is surprisingly not the case:

Theorem 11 (Savitch's Theorem) *Let $s(n) \geq \log n$ be a space-constructible function. Then* $\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2)$.

Proof This is another application of the reachability method. Let $L \in \text{NSPACE}(s(n))$. Then there is a non-deterministic machine M deciding L and using space $O(s(n))$. Consider the configuration graph G_M of $M(x)$ for some input x of length n, and recall that (1) vertices in G_M can be represented using $O(s(n))$ bits, and (2) existence of an edge in G_M from some vertex i to another vertex j can be determined using $O(s(n))$ space.

We may assume without loss of generality that M has a single accepting configuration (e.g., M erases its work tape and moves both heads to the left-most cell of their tapes before accepting). $M(x)$ accepts iff there is a directed path in G_M from the starting configuration of $M(x)$ (called start) to the accepting configuration of $M(x)$ (called accept). There are $V = 2^{O(s(n))}$ vertices in G_M, and the crux of the proof comes down to showing that reachability on a general V-node graph can be decided in deterministic space $O(\log^2 V)$.

Turning to that general problem, we define a (deterministic) recursive algorithm Path with the property that $\mathsf{Path}(a, b, i)$ outputs 1 iff there is a path of length at most 2^i from a to b in a given graph G; the algorithm only needs the ability to enumerate the vertices of G and to test for directed edges between any two vertices i, j in this graph. The algorithm proceeds as follows:

$\underline{\mathsf{Path}}(a, b, i)$:

- If $i = 0$, output "yes" if $a = b$ or if there is an edge from a to b. Otherwise, output "no".
- If $i > 0$ then for each vertex v:
  - If $\mathsf{Path}(a, v, i-1)$ and $\mathsf{Path}(v, b, i-1)$, return "yes" (and halt).
- Return "no".

Let $S(i)$ denote the space used by $\mathsf{Path}(a, b, i)$. We have $S(i) = O(\log V) + S(i-1)$ and $S(0) = O(\log V)$. This solves to $S(i) = O(i \cdot \log V)$.

We solve our original problem by calling $\mathsf{Path}(\mathsf{start}, \mathsf{accept}, \log V)$ using the graph G_M, where G_M has $V = 2^{O(s(n))}$ vertices. This uses space $O(\log^2 V) = O(s(n)^2)$, as claimed. ■

Corollary 12 $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

Is there a better algorithm for directed connectivity than what Savitch's theorem implies? Note that the algorithm implies by Savitch's theorem uses polylogarithmic space but superpolynomial time (specifically, time $2^{O(\log^2 n)}$). On the other hand, we have *linear*-time algorithms for solving directed connectivity but these require linear space. The conjecture is that $\mathsf{L} \neq \mathsf{NL}$, in which case directed connectivity does not have a log-space algorithm, though perhaps it would not be earth-shattering if this conjecture were proven to be false. Even if $\mathsf{L} \neq \mathsf{NL}$, we could still hope for an algorithm solving directed connectivity in $O(\log^2 n)$ space and polynomial time.

3.3 The Immerman-Szelepcsényi Theorem (As the proof appears in Sipser [1])

As yet another example of the reachability method, we will show the somewhat surprising result that non-deterministic space is closed under complementation. To prove this, we will demonstrate that $\overline{\textsc{Conn}} \in \mathsf{NL}$. Recall that

$$\overline{\textsc{Conn}} \stackrel{\text{def}}{=} \left\{ (G, s, t) : \begin{array}{c} G \text{ is a directed graph in which} \\ \text{there is } \textit{no} \text{ path from vertex } s \text{ to vertex } t \end{array} \right\}.$$

Recall, this means that there is a non-deterministic machine that uses $\log n$ space (where n is the size of the graph), and accepts on at least one computational path, iff *there is no path* from s to t. Note that there is no trivial way to show this by using a machine M for Conn, say by flipping the result of the computation. Consider what happens if we do flip the output of M: it is feasible that, on *any* input, even those that are in the language, M has many rejection paths. (We only know that M has at least 1 accepting path when x is in the language.) In this case, $\overline{M}$ would always have at least one accepting path, even if there *is* a path from s to t (i.e. even when $(G, s, t) \notin \overline{\textsc{Conn}}$).

Instead, we have to give a direct construction that computes the fact that there is no path. Suppose for a moment that we know there are exactly c_n nodes of distance at most n from the start point, s. Then we could non-deterministically choose c_n nodes from v, test whether we found all such nodes, and test whether t is among the set. If we found all nodes of distance n, and t is not among them, clearly there is no path to t. So the problem now reduces to calculating c_n, which we will show how to do if we know c_{n-1}: we can then use this procedure to iterate through $1 \leq i \leq n$, computing c_i, and eventually determining whether there is a path to t.

Let C_i denote the set of nodes with paths of distance at most i from s, and let $c_i = |C_i|$. To compute c_i from c_{i-1}, we guess the set C_{i-1} and verify that we found them all. Verifying membership of C_{i-1} is done by guessing a path. If we've guessed them all correctly, we then iterate

through $u \in C_{i-1}$, and through all members $v \in V$, testing whether $(u, v) \in E$. If so, we increment c_{i+1}, and if not, we move to the next item in V. The details follow.

Theorem 13 $\overline{\text{CONN}} \in \mathsf{NL}$.

Proof

$\underline{M(G, s, t):}$

1. Let $c_0 = 1$.
2. For $i = 0 \ldots n-1$
    3. Let $c_{i+1} = 0$.
    4. Let $d = 0$.
    5. Non-deterministically guess a set C_i. For each node $u \in C_i$:
        6. For each node $v \in V$:
            7. Nondeterministically choose a path of length at most i, starting at s. If u is not on this path, *halt and reject.*
            8. increment d
            9. If $(u, v) \in E$, increment c_{i+1}, fetch the next $u \in C_i$, and return to Step 6.
        10. If $d \neq c_i$, *halt and reject. //In this case, we guessed the set C_i incorrectly.*
11. Non-deterministically fix a a set C_n. *//We now use c_n to check whether t is within distance n of s.*
12. For each $u \in C_n$:
    13. Nondeterministically follow a path of length at most n from s. If u is not on this path, *halt and reject.*
    14. If $u = t$, *halt and reject.*
    15. increment d.
16. If $d \neq c_m$, *halt and reject.* Otherwise, *halt and accept.*

■

References

[1] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.