4 Space

In the previous chapter our focus was primarily on time complexity classes. In this chapter we will look more closely at space complexity. One might think that a similar pattern would emerge to that for time, but in fact space seems to behave very differently and, perhaps surprisingly, is rather easier to understand.

4.1 Determinism and Space

For example, the following theorem gives a close relationship between deterministic and non-deterministic space complexity classes.

**Savitch’s Theorem (1970).** Let $S : \mathbb{N} \to \mathbb{N}$ be a space constructible function. Then $\text{NSpace}(S(n)) \subseteq \text{Space}(S(n)^2)$.

**Proof.** Let $L \subseteq \mathcal{A}^*$ be accepted by an NDTM $N$ running in space $S(n)$. Then on input $\sigma \in \mathcal{A}^*$ any computation of $N$ on $\sigma$ never moves outside the region initially containing

$$\ldots, B^{S(n)} \sigma B^{S(n)} \ldots,$$

where $n = |\sigma|$ so we may (as usual) visualize a computation of $N$ on $\sigma$ as a sequence of truncated configurations $C_1, C_2, \ldots, C_h$ on just these squares. Furthermore if we let $k$ be such that $2^k$ is an upper bound on the number of possible tape symbols and state-tape symbol pairs then there are $\leq 2^{k(2S(n)+n)}$ ($= 2^{g(n)}$ say) possible choices for these $C_i$. Clearly no two $C_i$ can be the same, or we could fabricate a non-halting computation of $N$ on $\sigma$, so we must have $h \leq 2^{g(n)}$.

Fix some ordering $D_1, D_2, \ldots, D_{f(n)}$ of all possible (truncated) configurations of this length $2S(n) + n$ using these tape symbols and states from $N$, so $f(n) \leq 2^{g(n)}$. We assume we have a system of coding for (truncated) configurations $D_i$ with the following properties. Firstly, the code for each $D_i$ uses space linear in $S(n)$. Secondly, there is a TM which, given (the code for) $D_i$, works out the state of configuration $D_i$. Thirdly, there is a TM which, given $D_i$ and $D_j$, checks if $D_j$ can be obtained from $D_i$ by applying a rule of $N$. Fourthly, there is a TM which given input $\sigma$ computes the initial configuration of $N$ on $\sigma$ (which we will call $D_\sigma$). Fifthly, there is a TM which given $D_j$ (where $j < f(n)$) computes $D_{j+1}$. And finally, all of the above posited TM’s can be chosen to run in space $\alpha S(n) + \beta$ for some constants $\alpha$ and $\beta$. The details of such a coding are left as an exercise for the sceptics amongst you; it is straightforward but requires the space constructibility assumption on $S(n)$.

We will build a deterministic TM $M$ which, given $\sigma \in \mathcal{A}^*$, decides whether $N$ has an accepting computation on $\sigma$. To do this, it will exploit the following observation:

For $n \geq 1$, there is a computation sequence from $D_i$ to $D_j$ of length $\leq 2^n$ if and only if there is a configuration $D_k$ such that there are computation sequences from $D_i$ to $D_k$ and from $D_k$ to $D_j$, both of length $\leq 2^{n-1}$.

The machine $M$ will start out by asking, for each $j$ such that the state in configuration $D_j$ is the accept state, whether there is a computation from $D_1$ to $D_j$ of length $\leq 2^{g(n)}$. In order to answer this, it will check for each $k$, first whether there is a computation from $D_1$ to $D_k$ in $\leq 2^{g(n)-1}$ steps, and if so, whether there is a computation from $D_k$ to $D_j$ in $\leq 2^{g(n)-1}$ steps. It approaches each of these problems by dividing them up in the same way, this time looking for paths of length $\leq 2^{g(n)-2}$ steps, and so on. It carries on subdividing like this until it is asking whether there is a computation from $D_p$ to $D_q$ (say) in $\leq 2^0 = 1$ step, at which point it can answer the question just by checking if $D_p = D_q$ or $D_q$ can be obtained from $D_p$ by applying a single rule of $N$.

More formally, our TM $M$ acts as follows. During the computation, its tape will store a sequence of (codes for) 5-tuples of the form $(D_i, D_j, p, r, x)$ where $D_i$ and $D_j$ are (truncated) configurations of $N$, $p$ is a natural number, $r$ is $?$, $+$ or $\times$ and $x$ is $C$ (for complete), $F$ (for first) or $S$ (for second).

The machine first writes $(D_\sigma, D_1, g(n), ?, C)$ on its tape. This indicates that it is looking for a path from $D_\sigma$ to $D_1$ of length $\leq 2^{g(n)}$; the $?$ indicates that it doesn’t yet know whether there is such a path, and the $C$ indicates that this is the original complete path it is seeking, and not a subdivision.

Now at each stage in its computation, $M$ examines the final (rightmost) 5-tuple on the tape and acts as follows:
1. If the final tuple is \( \langle D_1, D_j, p, ?, x \rangle \) where \( p \geq 1 \) then append the tuple \( \langle D_1, D_k, p-1, ?, F \rangle \) at the (right-hand) end of the tape. [Start looking for a path from \( D_k \) of length \( \leq 2^{p-1} \), considering first the case \( k = 1 \).

2. If \( \langle D_i, D_j, 0, ?, x \rangle \) then check whether \( D_i = D_j \) and whether \( D_j \) can be obtained from \( D_i \) by applying a rule of \( N \). If either is true, replace the ? with \( \checkmark \); otherwise replace it with \( \times \). [Here we are asking for a computation of \( \leq 2^0 = 1 \) step, so we just check for it directly.]

3. If \( \langle D_i, D_j, p, x, x \rangle \) where \( x = C \) or \( x = F \) and \( j < f(n) \) then replace \( D_j \) with \( D_{j+1} \) and \( \times \) with ?. [Here we can’t find a suitable path from \( D_i \) to \( D_j \); so we move on to looking for a path from \( D_i \) to \( D_{j+1} \).]

4. If \( \langle D_i, D_j, p, x, C \rangle \) where \( j = f(n) \) then halt without accepting. [Here we have tried every possible way of building a path from \( D_i \) (which must be \( D_\sigma \) because of the “C”) to every accept configuration \( D_j \) without success.]

5. If \( \langle D_i, D_j, p, x, F \rangle \) where \( j = f(n) \) then erase this tuple, and replace the ? in the previous tuple with a \( \times \). [Here we have tried every possible way of building a path from \( D_i \) to wherever the previous tuple indicated it show go, without success.]

6. If \( \langle D_i, D_j, p, x, S \rangle \) then delete this tuple, and replace the ? in the previous one with a \( \times \). [We are looking for the (s)econd half of a path and it doesn’t exist, so the first half we have found is now useless and we must proceed without it.]

7. If \( \langle D_i, D_j, p, \checkmark, C \rangle \) then check if the state in configuration \( D_j \) is an accept state. If so accept; otherwise replace the \( \checkmark \) with a \( \times \). [There is a path from \( D_i \) (which must be \( D_\sigma \)) to \( D_j \). If \( D_j \) is an accepting configuration then we have found an accepting computation for \( \sigma \); otherwise the path we have found is irrelevant, so we may as well pretend it doesn’t exist!]

8. If \( \langle D_i, D_j, p, \checkmark, F \rangle \) then append a new tuple \( \langle D_j, D_k, p, ?, S \rangle \) where \( D_k \) is the second component of the previous tuple. [There is a path from \( D_i \) to \( D_j \), which we hope is the first half of a path from \( D_i \) to \( D_k \), so look for the second half.]

9. If \( \langle D_i, D_j, p, \checkmark, S \rangle \) then delete this tuple and the previous one, and change the ? in the one before that to a \( \checkmark \). [We have found the second half of a path for which we had already found the first half.]

It should be clear that if \( N \) does have an accepting computation on \( \sigma \) then \( M \) will eventually find it, so that \( M \) really does accept \( L \). \( M \) is clearly also deterministic so it only remains to check that \( M \) uses space quadratic in \( S(n) \). Our first assumption on the coding for the \( D_i \)'s guarantees that each 5-tuple on the tape takes only space which is a linear in \( S(n) \). The maximum number of tuples on the tape is clearly \( \leq 2g(n) + 2 \), because the third component of the tuples starts out at \( g(n) \) and decreases for at least every pair of tuples added (the worst case being a sequence of the form “\( F, S, F, S, \ldots \)”).

Since \( 2g(n) + 2 \) is also linear in \( S(n) \), the total space required to store the sequence of tuples is bounded above by a quadratic in \( S(n) \). Finally, the other assumptions on the coding ensure that the required constructions and manipulations of tuples can be performed without using a significant amount of extra working space. Hence, the total space requirement for \( M \) is quadratic in \( S(n) \), and so \( L \in \text{Space}(S(n)^2) \) as required.

4.2 Space and Complements

Recall that we defined \( NP^c \) to be the class of all languages whose complements are in \( NP \). More generally:

**Definition.** If \( \mathcal{C} \) is a complexity class then the complement of \( \mathcal{C} \) is the class of all languages \( L \subseteq \mathcal{A}^* \) such that \( \mathcal{A}^* \setminus L \in \mathcal{C} \). It is denoted \( \mathcal{C}^c \).

**The Immerman-Szelepcsényi Theorem (1987).** Let \( S : \mathbb{N} \rightarrow \mathbb{N} \) be a space constructible function. Then \( \text{NSpace}(S(n)) = \text{NSpace}(S(n))^c \).

---

55
Outline Proof. Let $M$ be an NDTM accepting $L \subseteq A^*$ and running in space $S(n)$. Then as in the proof of Savitch’s Theorem, on input $\sigma \in A^*$ with $|\sigma| = n$, $M$’s head never moves outside the region initially containing $B^S(n) \sigma B^S(n)$ so we only need to consider “truncated” configurations of length $2S(n) + n$.

Our first objective is to non-deterministically compute the exact number of different configurations which can be reached by computations of $M$ on input $\sigma$. Call this number $m_i$, and for $i \geq 0$, let $m_i$ be the number of different configurations which can be reached by computations of $M$ on input $\sigma$ in $\leq i$ steps. Notice that since $M$ is halting, there is a bound on the length of computations of $M$ on $\sigma$, so there must be some $i$ such that $m = m_i$. In fact, since there cannot be a computation of length $i + 2$ unless there is one of length $i + 1$, this will be the first $i$ such that $m_i = m_{i+1}$.

We compute the sequence $m_i$ inductively as follows. First, set $m_0 = 1$ (which is correct by definition). Now suppose that we know $m_i$ and want to compute $m_{i+1}$. We do this by the following procedure:

First set a counter $s$ to $0$. Then cycle through every (truncated) configuration $C$ in turn. For each configuration $C$, set a counter $r$ to $0$ and a boolean variable $b$ to $FALSE$, and then cycle through every configuration $D$ in turn. For each combination of $C$ and $D$, guess a computation from the initial configuration of length $\leq i$, and see if the computation makes sense and results in the configuration $D$. If not, then move onto the next $D$. If so, then increment the counter $r$, and check if $C = D$ or $C$ can be obtained from $D$ by the application of a rule. If it can, then set $b$ to $TRUE$. When all $D$’s have been considered; check if $r = m_i$. If not then halt and reject. Otherwise, if $b = TRUE$ then increment $s$. Now go on to the next $C$ (resetting $r$ to $0$ and $b$ to $FALSE$), and continue like this until all $C$’s have been considered.

Notice that for each $C$, the counter $r$ will count the number of $D$’s for which paths have been found from the initial configuration to $D$ of length $\leq i$. This number will end up being $m_i$ if and only if such a path was found for every $D$ such that a path exists. So the computations continuing at the end are exactly those in which a path is correctly guessed whenever such a path exists. Clearly, some computation will guess the correct paths whenever they exist, so some computation will continue to the end. In such a computation, the counter $s$ is incremented once for every configuration $C$ such that there exists a computation in $\leq i$ steps from the initial configuration to some configuration $D$, and a computation in $\leq i$ step from $D$ to $C$. But these are exactly the $C$’s for which there exists a computation from the initial configuration to $C$ in $\leq i + 1$ steps, so the final value of $s$ is $m_{i+1}$.

We continue computing the $m_i$’s in sequence until we find an $i$ such that $m_{i+1} = m_i$, whereupon we know that $m = m_i$. Clearly none of our counters will ever contain values greater than the number of possible (truncated) configurations, which means that each can be stored in space linear in $S(n)$. Since there is no reason to keep track of $m_{i-1}$ once we have computed $m_{i+1}$, the number of such counters we must store is bounded, so the total space required is also linear in $S(n)$.

Now we (or rather, all computations which have not rejected) know the number $m$ of configurations reachable by computations on $\sigma$, and also the maximum length $i$ of path that is required to reach them all. Our machine proceeds as follows.

Set a counter $t$ to $0$. Now for every non-accepting configuration $C$ of $M$, we try to guess a computation from the initial configuration of $M$ on $\sigma$ to $C$ in $i$ steps. At each stage, if we find one, then we increment $t$. Finally, we accept exactly if $t = m$. At the end of this process, some computation will have correctly guessed paths whenever they exist, and so will have $t$ equal to the number of non-accepting configurations reached by computations on $\sigma$; other computations may have smaller values of $t$. Now if $\sigma$ is accepted by $M$ then there must be fewer non-accepting configurations reachable than configurations reachable in total, so all computations will have $t < m$ and our machine rejects. On the other hand, if $\sigma$ is not accepted by $M$ then every reachable configuration is non-accepting so our “correctly guessing” computation will have $t = m$ and will accept.

Remark. The non-deterministic method used in the proof to find the value of $m$ is called inductive counting; it has many applications.

---

24 We have not actually made precise what it means for a non-deterministic machine to “compute” something — after all, which computation is supposed to compute it? What we mean here is that at least one computation will compute the correct value, and that any computation which does not compute the correct value will halt without accepting. This means that we can (just as in the deterministic computation) continue afterwards under the assumption that we have computed the correct value, since those computations which do not compute the correct value are irrelevant.

25 As usual, you can think of the counter as an extra tape which just stores a number in binary.

26 By which we mean, start with the initial configuration, and try to apply “guessed” rules. We can’t first guess a sequence of rules and then try to apply it, since this will take too much space!