Lecture 9: Space

Anup Rao

April 27, 2021

In this lecture, we begin our exploration of space. Recall, that the space complexity of an execution of a Turing machine is defined to be the maximum value attained by the pointer to the work tape during the execution. So, it is just a count of the number of cells used on the work tape during the execution of the algorithm.

So far, we have only discussed time complexity.

There is a subtle issue that came up last time in lecture. I think the best way to resolve this issue is use a slightly different definitino of Turing machines: we add the constraint that the pointer to the input tape can never exceed n—it is not allowed to move past the last bit of the input. If the code tries to move the pointer to the right beyond this last input bit, the machine simply leaves the pointer where it is. This does not affect any of the theorems or facts we proved in past lectures.

This is to address an issue with the number of possible configurations of the machines before. I will discuss the issue again below.

The smallest space class that makes sense is $\mathbf{L} = \mathsf{DSPACE}(\log n)$. This is because even maintaining a pointer to the input takes $\log n$ work space. While we do not necessarily need to maintain such pointers in the work tape, if we want to be able to design algorithms that have the same complexity regardless of the specific choices made for the Turing machine, then we need to maintain such pointers in order to simulate one Turing machine by another.

For example, if we are designing an algorithm to add two n-bit integers a, b, then if a, b are written on two different tapes (or interleaved on one tape), the computation can be carried out with $O(1)$ space. If, on the other hand, the inputs are written on one tape a, b, then we need space $O(\log n)$ in order to correctly maintain counters to allow us to scan between the corresponding bits a_i and b_i.

As usual the non-deterministic version of this class is when the machine can make non-deterministic choices, and is called $\mathbf{NL} = \mathsf{NSPACE}(\log n)$. There is a subtle issue about the definition of $\mathbf{NL}$: if we allow the machine to remember the non-deterministic choices that it made for free (for example by giving it access to a guess tape that it can read from), then the power of the class changes significantly. Another interesting class is $\mathbf{PSPACE} = \bigcup_c \mathsf{DSPACE}(n^c)$. The corresponding non-deterministic class is actually equal to $\mathbf{PSPACE}$, as we shall prove below.

A very useful fact when composing space bounded computations is the following:

Claim 1. *If it takes space $s_1(n) \geq \log n$ to compute f and space $s_2(n) \geq \log n$ to compute g, then one can compute the composition $f(g(x))$ in space $O(s_1(n) + s_2(n))$.*

The idea is that in the computation of f, every time we need to lookup an output symbol of $g(x)$, we can recompute it. Thus, as long as $s_1(n), s_2(n)$ are enough to store pointers into the output locations, we actually only need to sum the spaces to compute the composition.

Savitch's Algorithm

One of the most interesting small space algorithms is Savitch's graph search algorithm.

Theorem 2 (Savitch)**.** *Given a directed graph G with two special vertices s, t, there is an algorithm that can compute whether or not there is a path from s to t in the graph, using space* $O(\log^2 n)$*.*

Proof We shall give a recursive algorithm that can compute the values $A(u, v, i)$ as defined below:

$$A(u, v, i) = \begin{cases} 1 & \text{if there is a path from } u \text{ to } v \text{ of length } 2^i, \\ 0 & \text{else.} \end{cases}$$

Note that $A(u, v, i) = 1$ if and only if $\exists z$ such that $A(u, z, i-1) = 1$ and $A(z, v, i-1) = 1$. Thus, to compute $A(u, v, i)$, do

1. For all z, recursively compute $A(u, z, i-1)$ and $A(z, v, i-1)$, and output 1 if both computations result in 1.
2. Otherwise output 0.

If the size of the graph is 2^s, there are $s + 1$ recursive calls, where $A(u, v, 0)$ can be computed trivially by looking up the corresponding bit in the input. In each recursive call, the algorithm needs to store only the vertices u, v, z, which takes $O(\log n)$ space. Thus the total space used is $O(\log^2 n)$. ■

One reason Savitch's algorithm is so important is because, in some sense, graph search is a complete problem for small space computation. Let us discuss this point next.

Configuration Graphs

Given an input x to a (possibly non-deterministic) Turing machine M, the configuration graph $G_{M,x}$ is the directed graph where there is a distinct vertex for every possible value of the pointers to the input and work tapes, the value of the string written in the work tape and the current line-number of the line of code that is about to be executed in the machine. There is an edge from u to v if and only if the configuration u could possibly become the configuration v after one step of the program is executed.

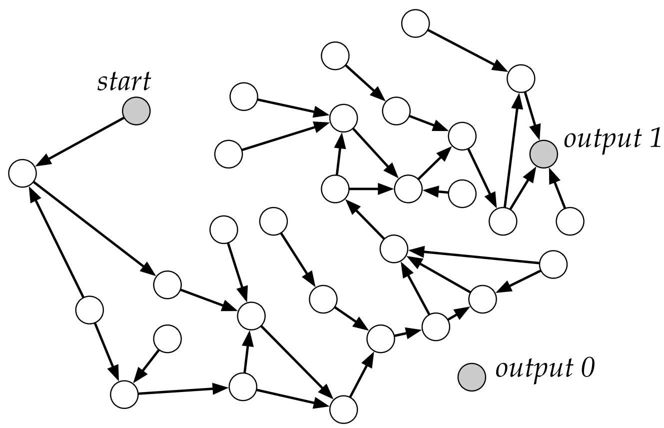

Figure 1: An example of a configuration graph.

Lemma 3. *If the machine uses space $s(n) \geq \Omega(\log n)$, then the number of vertices in the configuration graph is at most $2^{O(s(n))}$.*

Proof The number of options for locations of the pointers is at most $n \cdot s(n)$. The number of options for the contents of the work tape is at most $2^{O(s(n))}$. The number of options for the lines of code is $O(1)$. Thus, the number of different vertices in the graph is at most the product of these numbers, which is at most $2^{O(s(n))}$. ■

The number of options for the pointer that points to the input tape is at most n. This is because we do not allow the pointer on the input tape to move past the actual input. As we discussed in class, even if we did not place this restriction, we can prove that if the Turing machine moves the input pointer more than $2^{O(s(n))}$ steps beyond the input, then the machine does not halt. So, even without this restriction, the number of possible values for the input pointer is at most $2^{O(s(n))}$.

One can check that the configuration graph $G_{M,x}$ can be computed efficiently:

Theorem 4. *If M is a space $s(n)$ machine, then for every x, $G_{M,x}$ can be computed in space $O(s(n))$.*

The algorithm that computes $G_{M,x}$ is straightforward—on input M, x, the algorithm enumerates all possible configurations. For each one, it simulates a single execution step of the machine M to determine where the out edges from the corresponding configuration go.

Consequences to Space Complexity Classes

One consequence of the notion of configuration graphs is that small space machines must have bounded time complexity.

Theorem 5. *If $s(n) \geq \log n$, then $\mathsf{DSPACE}(s(n)) \subseteq \mathsf{DTIME}(2^{O(s(n))})$.*

Proof Consider any machine M computing a function $f \in \mathsf{DSPACE}(s(n))$. We claim that its running time must be at most $2^{O(s(n)}$. For any fixed input, the configuration graph has at most $2^{O(s(n))}$ vertices by Lemma 3. If the running time is more than the number of vertices , the turing machine must repeat a configuration, and then it will enter an infinite loop and will never halt. That contradicts the fact that M computes f. ■

Corollary 6. $\mathbf{L} \subseteq \mathbf{P}$.

Corollary 7. $\mathbf{PSPACE} \subseteq \mathbf{EXP}$.

Moreover, given any non-deterministic machine, one can simulate it deterministically by first computing its configuration graph, and then running Savitch's algorithm. This gives:

Theorem 8 (Savitch)**.** *For any space-constructible $s : \mathbb{N} \to \mathbb{N}$,*

$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DSPACE}(s(n)^2).$$

We obtain the following corollaries to Theorem 8.

Corollary 9. $\mathbf{NL} \subseteq \mathbf{L}^2 = \mathsf{DSPACE}(\log^2 n)$.

Corollary 10. $\mathbf{PSPACE} = \mathbf{NPSPACE}$.