Lecture 8: Space Complexity

Anup Rao

April 20, 2022

Next, we turn out attention to 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.

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 **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 **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.

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 $2^{O(s(n))}$.

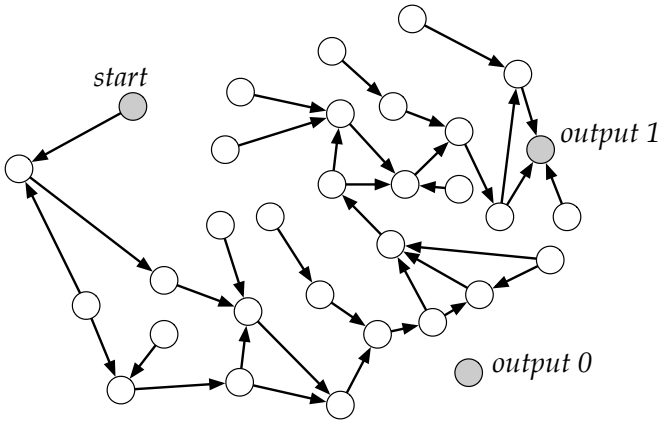

Figure 1: An example of a configuration graph.

IN THIS LECTURE, WE CONTINUE our discussion of space complexity classes. We first introduce a new definition. Given any set of boolean functions S, we write coS to denote the set

$$\{f : 1 - f \in S\}.$$

Thus $co\mathbf{NP}$ is the set of functions for which there is an efficiently verifiable proof that $f(x) = 0$.

Fact 4. $\mathbf{P} = co\mathbf{P}$

Fact 5. $\mathbf{L} = co\mathbf{L}$

Fact 6. $\mathbf{EXP} = co\mathbf{EXP}$

We do not know if $\mathbf{NP} = co\mathbf{NP}$. To show that $co\mathbf{NP} \subseteq \mathbf{NP}$, it would be enough to a polynomial time algorithm that can certify that a boolean formula is *unsatisfiable*.

Fact 7. *If* $\mathbf{P} = \mathbf{NP}$*, then* $\mathbf{NP} = co\mathbf{NP}$.

On the other hand, we can show:

Theorem 8. *For space constructible* $s(n)$, $\mathsf{NSPACE}(s(n)) = co\mathsf{NSPACE}(s(n))$.

Proof As usual we focus on the configuration graph. To prove the theorem, it will be enough to be able to verify that there is *no* path from two vertices u, v in the graph, in $s(n)$ space. This would show that if $f(x) = 1$ can be certified in space $s(n)$, then $f(x) = 0$ can also be certified in space $s(n)$. The other direction is completely symmetric.

We shall prove how to do this by designing a sequence of algorithms. Let C_i denote the set of vertices that are reachable from u in i steps. Suppose the graph is of size at most 2^s.

Claim 9. *Given any vertex* v *and a number* $i \leq 2^s$*, there is a non-deterministic space* $s(n)$ *algorithm such that:*

- *If $v \in C_i$, then some computational path outputs* 1
- *If $v \notin C_i$, then every computational path outputs* 0.

The algorithm simply guesses a path from u to v and checks that the path is a valid path of the graph by checking each edge in order.

Claim 10. *Given the size of $|C_{i-1}| = c$, and a vertex v, there is a non-deterministic space $s(n)$ algorithm such that*

- *If $v \notin C_i$, there is some computational path that outputs* 1.
- *If $v \in C_i$, then every computational path outputs* 0.

Since the algorithm is given the size of $C_{i-1}i$, the algorithm guesses each of the vertices of C_{i-1} in increasing order, and for each one, it checks that the vertex is different from the last vertex that was guessed, and then uses Claim 9 to verify that the vertex is indeed a member of C_{i-1}. It also makes sure that the given vertex is not v and not a neighbor of v. It maintains a count of all the number of vertices guessed and checks that $|C_{i-1}|$ vertices are given. If any of the checks fail, the algorithm outputs 0.

Finally, we argue that given the size of C_{i-1}, we can certify the size of $|C_i|$.

Claim 11. *Given the size of $|C_{i-1}| = c'$, there is a non-deterministic space $s(n)$ algorithm such that the algorithm either aborts or outputs $|C_i|$ on every computational path, and there is some computational path on which the algorithm outputs $|C_i|$.*

For each vertex v of the graph (in increasing order), the algorithm uses Claims 9 and 10 to check whether $v \in C_i$ or $v \notin C_i$, and it maintains a count of the number of vertices in C_i.

Thus, we obtain an algorithm that can verify that $v \notin C_n$ in $O(s(n))$ space. We first compute C_n by repeatedly using Claim 11 and then we apply Claim 10 to check whether $v \notin C_n$. ■