CS 121: Introduction to Theoretical Computer Science Fall 2019

Section 11 — November 21, 20190

Prof. Boaz Barak, Prof. Madhu Sudan *William Burke, Pratap Singh*

1 Space complexity

So far in this course we've only classified functions by their time complexity. But memory is another very important resource for computation, so we also want to study the space used by algorithms and programs.

1.1 Modeling space-limited computation

We want to model the amount of memory that is essential for a particular computation, but not worry about the space used to store the input and output. (Intuitively, we want to measure the additional space associated with the computation). To do this, we modify our models of computation such that memory usage is easily measured. In this unit, our Turing machines have three tapes:

- *Input tape:* a read-only tape initialized with the input to the machine. We can move both left and right.
- *Output tape:* a write-only tape on which we write the output of our machine. For simplicity, assume we can only move right.
- *Work tape:* a read/write tape which we use for our "intermediate results". The number of cells used on the work tape is the memory or space used by the machine.

We saw some examples of this in class: addition and multiplication of two n-bit integers is in $O(\log n)$ space, bubble sort is in $O(\log n)$ space while merge sort is in $O(n)$ space.

Two more important problems for this unit are $CIRC-EVAL$ and $PATH$. $CIRC-EVAL$ is the problem of evaluating a circuit C on input x, and takes space linear in the size of the circuit - in the worst case we need to store some information for every gate in the circuit. $PATH(G,u,v)$ is the decision problem of finding whether a path exists from vertex u to vertex v in directed graph G. Later, we will see the space complexity of $PATH$.

Additionally, we showed in class that SAT, quantified SAT, and many two-player games like chess and Go are also solvable in $O(n)$ space.

1.2 Space classes

Similar to how we defined sets of functions $F : \{0,1\}^* \to \{0,1\}$ based on their time complexity, we can define sets of functions based on space complexity.

The class $SPACE(s(n))$ is the class of functions $F : \{0,1\}^* \to \{0,1\}$ that can be computed by a Turing machine using space $s(n)$, where s is a nice function. Note that this definition doesn't make any restriction on the amount of time taken by the Turing machine.

Importantly, $REG \in SPACE(O(1))$, i.e. regular expression matching can be computed in constant space.

We define several larger classes based on these $SPACE$ classes:

- The class **L** is defined as $\mathbf{L} = \bigcup_{c\in\mathbb{N}} SPACE(c\log n)$, that is the class of problems that can be solved using space logarithmic in the size of the input. We consider this class to be feasible.
- The class **NL** relates to **L** in the same way that **NP** relates to **P**. In particular, for every function $F \in \mathbf{NL}$, there exists some function $V \in \mathbf{L}$ s.t. $F(x) = 1 \iff V(x,w) = 1$ for some $w \in \{0,1\}^{a|x|^b}$.
- The class **BPL** relates to **L** in the same way that **BPP** relates to **P**. For every function $F \in \mathbf{BPL}$, there exists some function $A \in \mathbf{L}$ s.t. $\Pr(F(x) = A(x,r)) \geq \frac{2}{3}$ where the probability is taken over all random strings r whose length is polynomial in the length of x.
- The class **PSPACE** is defined as $\mathbf{L} = \bigcup_{c\in\mathbb{N}} SPACE(n^c)$, that is the class of problems that can be solved using space polynomial in the size of the input. In general, this class is not considered to be feasible, and in fact it contains NP-complete problems.

- We can define **NPSPACE**, **BPPSPACE** in the same way.

Recall the universal algorithm for simulating a Turing machine. In previous chapters, we saw that the universal algorithm can simulate a machine with at most polynomial overhead in time complexity. Similarly, it can be shown that there is a universal algorithm for simulating a Turing machine with logarithmic space overhead - that is, a Turing machine $U(M, x)$ which, on input a machine M and input x, outputs the value computed by $M(x)$, and if M has space complexity $s(|x|)$, $U(M, x)$ has space complexity $O(s(|x|) + \log |x|)$.

Similar to the time hierarchy theorem, we also have a *space hierarchy theorem.* This states that for every pair of "nice" functions $s_1(n), s_2(n) : \mathbb{N} \to \mathbb{N}$ greater than $\log n$ such that $s_1(n) = o(s_2(n))$ (i.e. s_2 is asymptotically bigger than s_1), then $SPACE(s_1(n)) \subset SPACE(s_2(n))$ and $SPACE(s_1(n)) \neq SPACE(s_2(n))$.

In class, we discussed the theorem that $TIME(f(n)) \subseteq SPACE(f(n)) \subseteq TIME(2^{f(n)})$. Intuitively, the first inclusion is because any TM that takes $f(n)$ time steps can only write to its work tape at most $f(n)$ times; the second inclusion follows because any TM that uses $f(n)$ memory can only write $2^{f(n)}$ unique configurations to that memory, and any program that writes the same state to memory multiple times can be sped up to write it only once. This theorem also implies that $L \subseteq P \subseteq PSPACE \subseteq EXP$, based on the definitions of those classes.

1.3 Reductions and space completeness

Similarly to how we defined polynomial-time reductions, we have a notion of reduction in space complexity too. This allows us to understand relationships between problems and define a notion of completeness for any of our classes - a function F is complete in class **X**, or **X**-complete, if and only if F is in **X** and every function in **X** can be reduced to F.

Formally, we say that $F \leq_{s(n)} G$ if and only if there exists some algorithm R that uses space $s(n)$ and $\forall x \in \{0,1\}^*$, $F(x) = G(R(x))$. That is, an input to F can be transformed to an input to G in $s(n)$ space.

Now, supposing G uses space $s_2(m)$ and the reduction algorithm R uses space $s_1(n)$, what can we say about the space usage of F? A naïve way to compute F using G would be to just compute $y = R(x)$ and then compute $G(y)$. How much space does this use? We need $|y| = m$ bits to write y, plus $s_1(n)$ bits for R and $s_2(m)$ bits for G. Since we can reuse the memory used for R for G, the total is $m + \max(s_1(n), s_2(m))$. For problems in $\mathbb{L}$, we might have something like $s_1(n) = 5 \log n$ and $s_2(m) = 10 \log m$, but then computing F would take $O(m)$, not $O(\log m)$ bits. Since we don't consider $O(m)$ to always be feasible, we want to do better than this.

It turns out that it is possible to do better. Instead of computing y first, we start by simply computing G using $s_2(m)$ space. Whenever our algorithm for G requires a bit y_i from y, we suspend the computation of G, saving its state using $O(s_2(m))$ bits, then run R using $s_1(n)$ additional memory. Importantly, the only bit of y that we actually write down is y_i, so we don't use $O(m)$ space in writing down y. We then continue computing G using that bit as normal. This reduction allows us to compute F using space $s_1(n) + s_2(m) + O(\log m)$, which will still be logarithmic if both s_1 and s_2 are logarithmic. (However, note that this is a very time-inefficient algorithm. In general, algorithms optimized for parsimonious memory usage may not be time-efficient, and vice versa.)

Now, with this notion of reduction, we can define completeness such as **NL**-completeness. A problem is **NL**-complete if and only if it is in **NL** and every problem in **NL** reduces to it in logarithmic space. We claim that $PATH$ is **NL**-complete. It is easy to see that $PATH$ is in **NL** - we can verify whether a path is correct by simply walking along it from the source vertex and checking if we reach the destination vertex; this takes $O(\log n)$ bits to store the number of the current vertex. The proof that $PATH$ is **NL**-hard is omitted here for brevity, but the general idea is to encode the configurations of any Turing machine in **NL** as a graph, and then show that executing the Turing machine is equivalent to solving $PATH$ for some pair of vertices in that graph. (See Arora and Barak, theorem 4.18, for the full proof).

The most important proof for this unit is that $PATH \in \mathbf{L^2}$, i.e. the class of problems that can be solved in $O(\log^2 n)$ time. This shows that $\mathbf{NL} = \mathbf{L^2}$. This result follows directly from a theorem that we discussed in class known as Savitch's theorem, which states that for any space class $NSPACE(f(n)) \subseteq SPACE(f(n)^2)$, but we will show this constructively. We define a recursive algorithm that tests graph G for connectivity between vertices $s, t \in V$ in at most k steps, STCON(s,t,k). PATH(s,t) reduces to calling STCON(s,t,n) where n is the total number of vertices in the graph. The algorithm runs as follows,

Definition 1 (STCON)**.** Let $G = \{V, E\}$ be the directed or undirected graph provided to the algorithm and let n be the number of vertices in V. Let s and t be vertices. The algorithm returns true if there is a path of length at most k between the vertices.

if $k == 0$ OR $k == 1$ **then**
  return $s == t$ OR (s is a neighbor of t)
end if

if $k > 1$ **then**
  for each vertex $u \in V, u \notin \{s, t\}$ **do**
    if $STCON(s, u, \lfloor k/2 \rfloor)$ AND $STCON(u, t, \lceil k/2 \rceil)$ **then**
      return TRUE
    else
    end if
  end for
end if
return FALSE

If we call STCON(s,t,n), we can see that this will return TRUE if there is a path between the vertices s and t. It will reach a recursion depth of at most log(n) and at each step, it will use at most log(n) memory. The total space complexity is then just $log^2(n)$

Exercise: Prove that $NP \subseteq PSPACE$. Hint: We did this in lecture. Think about 3SAT.

Exercise: Prove that $PSPACE = NPSPACE$. Hint: Use the verifier. Alternatively, apply Savitch's theorem.

Exercise: Prove that $SPACE(f(n)) \subseteq TIME(2^{f(n)})$. Hint: Use the pigeonhole principle. Solution is in 1.2. Also in lecture.

Exercise: A *strongly connected graph* is a directed graph $G = (V, E)$ such that for every pair of vertices $u, v \in V$, there exists a path from u to v and from v to u. Let $STRONGLY - CONNECTED : \{0,1\}^* \to \{0,1\}$ be the function that, on input a graph $G = (V, E)$, returns 1 if and only if G is strongly connected. Prove that $STRONGLY - CONNECTED$ is **NL**-complete.

Solution sketch:

Show that $STRONGLY - CONNECTED \in$ **NL**: the certificate w can be a list of paths between every pair of vertices in the graph. Then we can just use the same checking procedure which we used for $PATH$ on the $O(n^2)$ pairs of vertices, reusing the space, so the verifier runs in logspace.

Show that $STRONGLY - CONNECTED$ is **NL**-hard: reduce from $PATH$ to $STRONGLY - CONNECTED$. On input a graph G and source and target vertices s, t, do the following:

for each vertex $u \in V, u \notin \{s, t\}$ **do**
  add an edge from u to s
  add an edge from t to u
end for

If the original graph has a path from s to t, then this graph is strongly connected since for any pair of vertices, you can just follow the path from s to t. If the original graph did not have a path from s to t, then this graph is not strongly connected since the new edges don't add any paths from s to t, so there are at least two vertices (s and t) which don't have a path between them.

See also `http://cs.brown.edu/courses/gs019/asgn/hw6.sol.pdf`.

2 Cryptography

2.1 Definition of valid encryption scheme

Let $L : \mathbb{N} \to \mathbb{N}$ be some function. A pair of polynomial-time computable functions (E, D) mapping strings to strings is a *valid private key encryption scheme* (or *encryption scheme* for short) with plaintext length $L(\cdot)$ if for every $k \in \{0,1\}^n$ and $x \in \{0,1\}^{L(n)}$,

$$D(k, E(k, x)) = x \,. \tag{1}$$

We also require that our encryption schemes are *length regular* in the sense that all ciphertexts corresponding to keys of the same length are of the same length: there is some function $C : \mathbb{N} \to \mathbb{N}$ such that for every $k \in \{0,1\}^n$ and $x \in \{0,1\}^{L(n)}$, $|E(k, x)| = C(n)$.[1]

2.2 Security

Let's go back to the definition of valid encryption scheme. Remark that the definition does not say anything about security. We could choose $E(k, x) = x$ and still be able to make the equation above work. Although this is definitely correct, it doesn't seem useful in practice because it's easy to decrypt.

Exercise: Think about what conditions you might want to put such that the encryption is secure.

[1] The "length regularity" condition is added for technical convenience and is not at all important. You can ignore it in a first reading.

Solutions:

A valid encryption scheme (E, D) with length $L(\cdot)$ is *perfectly secrect* if for every $n \in \mathbb{N}$ and plaintexts $x, x' \in \{0,1\}^{L(n)}$, the following two distributions Y and Y' over $\{0,1\}^*$ are identical:

- Y is obtained by sampling a random $k \sim \{0,1\}^n$ and outputting $E_k(x)$.
- Y' is obtained by sampling a random $k \sim \{0,1\}^n$ and outputting $E_k(x')$.

It is not clear from the definition that a perfect encryption system exists. However:

One Time Pad (Vernam 1917, Shannon 1949)

There is a perfectly secret valid encryption scheme (E, D) with $L(n) = n$.

Proof idea

This idea is called the "Vernam Cipher", exceedingly simple: to encrypt a message $x \in \{0,1\}^n$ with a key $k \in \{0,1\}^n$ we simply output $x \oplus k$ where $\oplus$ is the bitwise XOR operation that outputs the string corresponding to XORing each coordinate of x and k.

Try proving this as an **exercise**.

2.3 Perfect secrecy requires long keys

The problem with the above encryption scheme is that to communicate n bits, a key of length n needs to be stored. We can actually prove the following result

For every perfectly secret encryption scheme (E, D) the length function L satisfies $L(n) \leq n$.

Proof idea: If the number of keys is smaller than the number of messages then the neighborhoods of all vertices in the corresponding graphs cannot be identical.

2.4 Computational secrecy

We defined computational secrecy as

Let (E, D) be a valid encryption scheme where for keys of length n, the plaintexts are of length $L(n)$ and the ciphertexts are of length $m(n)$. We say that (E, D) is *computationally secret* if for every polynomial $p : \mathbb{N} \to \mathbb{N}$, and large enough n, if P is an $m(n)$-input and single output NAND program of at most $p(n)$ lines, and $x_0, x_1 \in \{0,1\}^{L(n)}$ then

$$\left|\mathbb{E}_{k \sim \{0,1\}^n}[P(E_k(x_0))] - \mathbb{E}_{k \sim \{0,1\}^n}[P(E_k(x_1))]\right| < \frac{1}{p(n)} \tag{2}$$

Remember from class we proved the following

Suppose that the optimal PRG conjecture is true. Then for every constant $a \in \mathbb{N}$ there is a computationally secret encryption scheme (E, D) with plaintext length $L(n)$ at least n^a.

Proof idea

We simply take the one-time pad on L bit plaintexts, but replace the key with $G(k)$ where k is a string in $\{0,1\}^n$ and $G : \{0,1\}^n \to \{0,1\}^L$ is a pseudorandom generator.