PRINCETON UNIVERSITY COS 522: COMPUTATIONAL COMPLEXITY

Lecture 2: Space Complexity

Lecturer: *Sanjeev Arora* Scribe:*scribename*

Today we discuss space-bounded computation. A space-bounded machine has a read-only input tape, and a read-write work tape. We say that it runs in $S(n)$ space if the work tape has $O(S(n))$ cells when the input has n bits. (See Figure 1.) We denote by $\mathbf{SPACE}(S(n))$ the class of languages that can be decided in $O(S(n))$ space. We will restrict attention to $S(n) \geq \log n$, so the machine has enough space to maintain a pointer into the input tape. Note that $\mathbf{DTIME}(t(n)) \subseteq \mathbf{SPACE}(t(n))$ since a TM running in time $t(n)$ can only use $t(n)$ cells in the work tape. Also, $\mathbf{SPACE}(s(n)) \subseteq \mathbf{DTIME}(2^{O(t(n))})$, since a machine with a work tape of size $s(n)$ only has $O(n \cdot 2^{O(s(n))}) = 2^{O(s(n))}$ different configurations, and it cannot enter the same configuration twice since that would mean it is in an infinite loop. (Recall that the machine is required to halt on every input.)

We can similarly define nondeterministic space-bounded machines, and the class $\mathbf{NSPACE}(s(n))$. The following definitions are similar to the definitions of **P** and **NP**.

DEFINITION 1 $\mathbf{PSPACE} = \cup_{c>0}\mathbf{SPACE}(n^c)$.
$\mathbf{NPSPACE} = \cup_{c>0}\mathbf{NSPACE}(n^c)$.

Note that $\mathbf{NP} \subseteq \mathbf{PSPACE}$, since polynomial space is enough decide 3SAT (just cycle through all 2^n assignments, where n is the number of variables).

Let TQBF be the set of quantified boolean formulae that are true.

EXAMPLE 1 The formula $\forall x \exists y\ (x \wedge y) \vee (\overline{x} \wedge \overline{y})$ is in TQBF but $\forall x \forall y\ (x \wedge y) \vee (\overline{x} \wedge \overline{y})$ is not in TQBF.

For a proof of the following theorem, see Sipser Chapter 8.

THEOREM 1
TQBF is complete for **PSPACE** *under polynomial-time reductions.*

1 Two suprising algorithms

Now we describe two surprising algorithms for space-bounded computation. Let PATH be the language

$$\text{PATH} = \{< G, s, t >:\ G \text{ is a directed graph in which there is a path from } s \text{ to } t\} \quad (1)$$

PATH has a trivial polynomial time algorithm that uses depth-first search. It is unclear how to implement decide PATH in sublinear space, though. Note that PATH $\in$ **NL**, since a nondeterministic machine can take a "nondeterministic walk" starting at s, always maintaining the index of the vertex it is at, and using nondeterminism to select an outgoing edge out of this vertex for the next vertex to go to. The machine accepts iff the walk ends

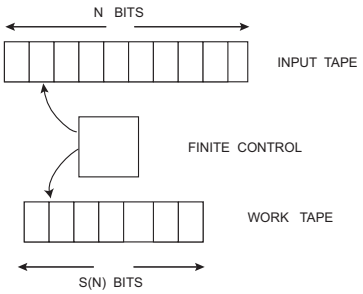

Figure 1: A TM running in space $O(S(n))$

at t in at most n steps, where n is the number of nodes. Thus it needs to keep track of the current vertex, which it can do using $O(\log n)$ bits.

The next theorem says that this computation can be made deterministic with quadratic blowup in space.

THEOREM 2 (SAVITCH)
PATH $\in$ **SPACE**$(\log^2 n)$.

PROOF: Let $< G, s, t >$ be the input. We describe a recursive procedure REACH?(u, v, l) that returns "YES" if there is a path from u to v of length at most l and "NO" otherwise. The procedure uses the observation that if there is a path from u to v of length at most l, there is a "middle" node z along this path such that there is a path of length $\lceil l/2 \rceil$ from u to z and $\lfloor l/2 \rfloor$ from z to v.

The procedure is as follows. *If $l = 1$, return YES iff (u, v) is an edge. Otherwise for each node z, run* REACH?*$(u, z, \lceil l/2 \rceil)$ and* REACH?*$(u, z, \lfloor l/2 \rfloor)$ and return YES if both return YES.*

The main observation is that all recursive calls in this description can reuse the same space. Also, keeping track of the current value of z takes only $O(\log n)$ space. Thus if $S(l)$ denotes the space requirement, we have

$$S(l) \leq O(\log n) + S(\lceil \frac{l}{2} \rceil). \tag{2}$$

This yields $S(l) = O(\log n \log l)$.

The final answer is REACH?(s, t, n), so the space required is $S(n) = O(\log^2 n)$. □

The next result concerns $\overline{\text{PATH}}$, the complement of PATH. A decision procedure for this language must accept when there is no path from s to t in the graph. The following result is quite surprising.

THEOREM 3 (IMMERMAN-SZLEPCSENYI)
$\overline{PATH} \in$ **NL**.

PROOF: How can a nondeterministic computation decide that there is no path from s to t? Let us first consider an easier problem. Suppose somebody gives the machine a number c, which is exactly the number of nodes reachable from s. Then the machine's task becomes

easier. It keeps aside t and for every other nodes, it sequentially tries to guess —using a nondeterministic walk —a path from s to that node. It accepts at the end iff it succeeds for c nodes, since that means that t is not one of the c nodes connected to s. Note that if e very branch of this nondeterministic computation fails (i.e., does not accept) then there do not exist c nodes different from t that are reachable from s, which means that then t must be reachable from s.

Now we can describe the nondeterministic computation for $\overline{PATH}$. We use an inductive counting technique to calculate c, whereby step i determines c_i, the number of nodes reachable from s in i steps. (Thus c_n is the same as c.) Clearly, $c_0 = 1$. To compute c_{i+1} from c_i we use a modification of the idea in the previous paragraph. For each node u we perform a nondeterministic computation which succeeds iff u has distance exactly $i+1$ from s. Our basic "nondeterministic walk" is used over an over. The following nondeterministic procedure omputes $c_{i+1} - c_i$:

Maintain a counter, initialized to 0. *Enumerate nodes one by one. For each node u, start a nondeterminstic computation that accepts iff the distance of u from s is exactly $i+1$ (namely, if you can enumerate exactly c_i nodes different from u whose distance to s is at most i, and one of them has an edge to u). If this nondeterministic computation succeeds, increment the counter and continue. If it rejects, HALT immediately and reject.*

□

2 Picture of Space-Bounded Complexity Classes

The following two are simple corollaries of the two algorithms, since the PATH problem is complete for **NL**.

Theorem 4 (Savitch)
$\mathbf{NSPACE}(s(n)) \subseteq \mathbf{SPACE}(s(n)^2)$.

(Thus in particular, $\mathbf{PSPACE} = \mathbf{NPSPACE}$, in contrast to the conjecture $\mathbf{P} \neq \mathbf{NP}$.)

Theorem 5 (Immerman-Szlepcsenyi)
$\mathbf{NSPACE}(s(n)) = \mathbf{coNSPACE}(s(n))$.

(This is in contrast to the conjecture that $\mathbf{NP} \neq \mathbf{coNP}$.)

Thus the following is our understanding of space-bounded complexity.

$$\mathbf{DTIME}(s(n)) \subseteq \mathbf{SPACE}(s(n)) \subseteq \mathbf{NSPACE}(s(n)) = \mathbf{coNSPACE}(s(n)) \subseteq \mathbf{DTIME}(2^{O(s(n))}).$$

Exercises

§1 In analogy with the characterization of **NP** in terms of certificates, show that we can define $\mathbf{NSPACE}(s(n))$ as the set of languages for which certificates can be checked deterministically using $O(S(n))$ space, where the certificate is provided on a read-only tape of size $2^{O(S(n))}$ and the verifier can scan it only once from left to right.