Notes on Space-Bounded Complexity

These are notes for CS278, Computational Complexity, scribed by students in January 2001, revised in August 2002, April 2004, and then April 2007.) Sections 4 and 5 are "bonus" material, not part of the syllabus. These notes do not include the proof of PSPACE-completeness of the TQBF problem, which is well explained in Sipser's book. The problem called PATH in Siper's book is called STCONN here. L.T.

1 Space-Bounded Complexity Classes

A machine solves a problem using space $s(\cdot)$ if, for every input x, the machine outputs the correct answer and uses only the first $s(|x|)$ cells of the tape. For a standard Turing machine, we can't do better than linear space since x itself must be on the tape. So we will often consider a machine with multiple tapes: a read-only "input" tape, a read/write "work" or "memory" tape, and possibly a write-once "output" tape. Then we can say the machine uses space s if for input x, it uses only the first $s(|x|)$ cells of the work tape.

We denote by **L** the set of decision problems solvable in $O(\log n)$ space. We denote by **PSPACE** the set of decision problems solvable in polynomial space. A first observation is that a space-efficient machine is, to a certain extent, also a time-efficient one. In general we denote by **SPACE**$(s(n))$ the set of decision problems that can be solved using space at most $s(n)$ on inputs of length n.

Theorem 1 *If a machine always halts, and uses $s(\cdot)$ space, with $s(n) \geq \log n$, then it runs in time $2^{O(s(n))}$.*

PROOF: Call the "configuration" of a machine M on input x a description of the state of M, the position of the input tape, and the contents of the work tape at a given time. Write down $c_1, c_2, \ldots, c_t$ where c_i is the configuration at time i and t is the running time of $M(x)$. No two c_i can be equal, or else the machine would be in a loop, since the c_i completely describes the present, and therefore the future, of the computation. Now, the number of *possible* configurations is simply the product of the number of states, the number of positions on the input tape, and the number of possible contents of the work tape (which itself depends on the number of allowable positions on the input tape). This is

$$O(1) \cdot n \cdot |\Sigma|^{s(n)} = 2^{O(s(n))+\log n} = 2^{O(s(n))}$$

Since we cannot visit a configuration twice during the computation, the computation must therefore finish in $2^{O(s(n))}$ steps. □

NL is the set of decision problems solvable by a non-deterministic machine using $O(\log n)$ space. **NPSPACE** is the set of decision problems solvable by a non-deterministic machine using polynomial space. In general we denote by **NSPACE**$(s(n))$ the set of decision problems that can be solved by non-deterministic machines that use at most $s(n)$ bits of space on inputs of length n.

Analogously with time-bounded complexity classes, we could think that **NL** is exactly the set of decision problems that have "solutions" that can verified in log-space. If so, **NL** would be equal to **NP**, since there is a log-space algorithm V that verifies solutions to SAT. However, this is unlikely to be true, because **NL** is contained in **P**. An intuitive reason why not all problems with a log-space "verifier" can be simulated in **NL** is that an **NL** machine does not have enough memory to keep track of all the non-deterministic choices that it makes.

Theorem 2 $\mathbf{NL} \subseteq \mathbf{P}$.

PROOF: Let L be a language in **NL** and let M be a non-deterministic log-space machine for L. Consider a computation of $M(x)$. As before, there are $2^{O(s(n))} = n^{O(1)}$ possible configurations. Consider a directed graph in which vertices are configurations and edges indicate transitions from one state to another which the machine is allowed to make in a single step (as determined by its δ). This graph has polynomially many vertices, so in polynomial time we can do a depth-first search to see whether there is a path from the initial configuration that eventually leads to acceptance. This describes a polynomial-time algorithm for deciding L, so we're done. □

2 Reductions in NL

We would like to introduce a notion of completeness in **NL** analogous to the notion of completeness that we know for the class **NP**. A first observation is that, in order to have a meaningful notion of completeness in **NL**, we cannot use polynomial-time reductions, otherwise any **NL** problem having at least a YES instance and at least a NO instance would be trivially **NL**-complete. To get a more interesting notion of **NL**-completeness we need to turn to weaker reductions. In particular, we define *log space* reductions as follows:

Definition 3 *Let A and B be decision problems. We say A is log space reducible to B, $A \leq_{\log} B$, if $\exists$ a function f computable in log space such that $x \in A$ iff $f(x) \in B$, and $B \in L$.*

Theorem 4 *If $B \in \mathbf{L}$, and $A \leq_{\log} B$, then $A \in \mathbf{L}$.*

PROOF: We consider the concatenation of two machines: M_f to compute f, and M_B to solve B. If our resource bound was polynomial time, then we would use $M_f(x)$ to compute $f(x)$, and then run M_B on $f(x)$. The composition of the two procedures would given an algorithm for A, and if both procedures run in polynomial time then their composition is also polynomial time. To prove the theorem, however, we have to show that if M_f and M_B are log space machines, then their composition can also be computed in log space.

Recall the definition of a Turing machine M that has a log space complexity bound: M has one read-only input tape, one write-only output tape, and uses a log space work tape. A naive implementation of the composition of M_f and M_B would be to compute $f(x)$, and then run M_B on input $f(x)$; however $f(x)$ needs to be stored on the work tape, and this implementation does not produce a log space machine. Instead we modify M_f so that on input x and i it returns the i-th bit of $f(x)$ (this computation can still be carried out in logarithmic space). Then we run a simulation of the computation of $M_B(f(x))$ by using the modified M_f as an "oracle" to tell us the value of specified positions of $f(x)$. In order to simulate $M_B(f(x))$ we only need to know the content of one position of $f(x)$ at a time, so the simulation can be carried with a total of $O(\log|x|)$ bits of work space. □

Using the same proof technique, we can show the following:

Theorem 5 *if $A \leq_{\log} B, B \leq_{\log} C$, then $A \leq_{\log} C$.*

3 NL Completeness

Armed with a definition of log space reducibility, we can define **NL**-completeness.

Definition 6 *A is* **NL***-hard if* $\forall B \in$ **NL**$, B \leq_{\log} A$. *A is* **NL***-complete if $A \in$* **NL** *and A is* **NL***-hard.*

We now introduce a problem STCONN (s,t-connectivity) that we will show is **NL**-complete. In STCONN, given in input a directed graph $G(V, E)$ and two vertices $s, t \in V$, we want to determine if there is a directed path from s to t.

Theorem 7 *STCONN is* **NL***-complete.*

Proof:

1. STCONN $\in$ **NL**.

   On input *G(V,E), s,t*, set p to s. For $i = 1$ to $|V|$, nondeterminsitically, choose a neighboring vertex v of p. Set $p = v$. If $p = t$, accept and halt. Reject and halt if the end of the *for* loop is reached. The algorithm only requires $O(\log n)$ space.

2. STCONN is **NL**-hard.

   Let $A \in$ **NL**, and let M_A be a non-deterministic logarithmic space Turing Machine for A. On input x, construct a directed graph G with one vertex for each configuration of $M(x)$, and an additional vertex t. Add edges (c_i, c_j) if $M(x)$ can move in one step from c_i to c_j. Add edges (c, t) from every configuration that is accepting, and let s be the start configuration. M accepts x iff some path from s to t exists in G. The above graph can be constructed from x in log space, because listing all nodes requires $O(\log n)$ space, and testing valid edges is also easy.

□

4 Savitch's Theorem

What kinds of tradeoffs are there between memory and time? STCONN can be solved deterministically in linear time and linear space, using depth-first-search. Is there some sense in which this is optimal? Nondeterministically, we can search using less than linear space. Can searching be done deterministically in less than linear space?

We will use Savitch's Theorem to show that STCONN can be solved deterministically in $O(\log^2 n)$, and that every **NL** problem can be solved deterministically in $O(\log^2 n)$ space. In general, if A is a problem that can be solved nondeterministically with space $s(n) \geq \log n$, then it can be solved deterministically with $O(s^2(n))$space.

Theorem 8 *STCONN can be solved deterministically in $O(\log^2 n)$ space.*

Proof: Consider a graph $G(V, E)$, and vertices s, t. We define a recursive function REACH(u, v, k) that accepts and halts iff v can be reached from u in $\leq k$ steps. If $k = 1$, then REACH accepts iff (u, v) is an edge. If $k \geq 2, \forall w \in V - \{u, v\}$, compute REACH$(u, w, \lfloor \mathrm{k}/2 \rfloor)$ and REACH$(w, v, \lceil \mathrm{k}/2 \rceil)$. If both accept and halt, accept. Else, reject.

Let $S(k)$ be the worst-case space use of REACH$(\cdot, \cdot, k)$. The space required for the base case $S(1)$ is a counter for tracking the edge, so $S(1) = O(\log n)$. In general, $S(k) = O(\log n) + S(k/2)$ for calls to REACH and for tracking w. So, $S(k) = O(\log k * \log n)$. Since $k \leq n$, the worst-case space use of REACH is $O(\log^2 n)$. □

Essentially the same proof applies to arbitrary non-deterministic space-bounded computations. This result was proved in [Sav70]

Theorem 9 (Savitch's Theorem) *For every function $s(n)$ computable in space $O(s(n))$,* $\mathbf{NSPACE}(s) = \mathbf{SPACE}(O(s^2))$

PROOF: We begin with a nondeterministic machine M, which on input x uses $s(|x|)$ space. We define $REACH(c_i, c_j, k)$, as in the proof of Theorem 8, which accepts and halts iff $M(x)$ can go from c_i to c_j in $\leq k$ steps. We compute $REACH(c_0, c_{\text{acc}}, 2^O(s|x|))$ for all accepting configurations c_{acc}. If there is a call of REACH which accepts and halts, then M accepts. Else, M rejects. If REACH accepts and halts, it will do so in $\leq 2^{O(|x|)}$ steps.

Let $S_R(k)$ be the worst-case space used by REACH$(\cdot, \cdot, k)$: $S_R(1) = O(s(n)), S_R(k) = O(s(n)) + S_R(k/2)$. This solves $S_R = s(n) * \log k$, and, since $k = 2^O(s(n))$, we have $S_R = O(s^2(n))$. □

Comparing Theorem 8 to depth-first-search, we find that we are exponentially better in space requirements, but we are no longer polynomial in time.

Examining the time required, if we let $t(k)$ be the worst-case time used by REACH$(\cdot, \cdot, k)$, we see $t(1) = O(n + m)$, and $t(k) = n(2 * T(k/2))$, which solves to $t(k) = n^{O(\log k)} = O(n^{O(\log n)})$, which is super-polynomial. Savitch's algorithm is still the one with the best known space bound. No known algorithm achieves polynomial log space and polynomial time simultaneously, although such an algorithm is known for *undirected* connectivity.

5 Undirected Connectivity

In the undirected $s - t$ connectivity problem (abbreviated ST-UCONN) we are given an undirected graph $G = (V, E)$ and two vertices $s, t \in V$, and the question is whether that is a path between s and t in G.

While this problem is not known to be complete for **NL**, and it probably is not, ST-UCONN is complete for the class **SL** of decision problems that are solvable by *symmetric* non-deterministic machines that use $O(\log n)$ space. A non-deterministic machine is symmetric if whenever it can make a transition from a configuration s to a transition s' then the transition from s' to s is also possible. The proof of **SL**-completeness of ST-UCONN is identical to the proof of **NL**-completeness of ST-CONN except for the additional observation that the transition graph of a symmetric machine is undirected.

6 Randomized Log-space

We now wish to introduce *randomized* space-bounded Turing machine. For simplicity, we will only introduce randomized machines for solving decision problems. In addition to a read-only input tape and a read/write work tape, such machines also have a read-only *random* tape to which they have *one-way* access, meaning that the head on that tape can only more, say, left-to-right. For every fixed input and fixed content of the random tape, the machine is completely deterministic, and either accepts or rejects. For a Turing machine M, an input x and a content r of the random tape, we denote by $M(r, x)$ the outcome of the computation.

We say that a decision problem L belongs to the class **RL** (for *randomized* log-space) if there is a probabilistic Turing machine M that uses $O(\log n)$ space on inputs of length n and such that

- For every $x \in L$, $\mathbf{Pr}_r[M(r, x) \text{ accepts }] \geq 1/2$
- For every $x \notin L$, $\mathbf{Pr}_r[M(r, x) \text{ accepts }] = 0$.

It is easy to observe that any constant bigger than 0 and smaller than 1 could be equivalently used instead of 1/2 in the definition above. It also follows from the definition that $\mathbf{L} \subseteq \mathbf{RL} \subseteq \mathbf{NL}$.

The following result shows that, indeed, $\mathbf{L} \subseteq \mathbf{SL} \subseteq \mathbf{RL} \subseteq \mathbf{NL}$.

Theorem 10 *The problem ST-UCONN is in* $\mathbf{RL}$.

We will not give a proof of the above theorem, but just describe the algorithm. Given an undirected graph $G = (V, E)$ and two vertices s, t, the algorithm performs a random walk of length $100 \cdot n^3$ starting from s. If t is never reached, the algorithm rejects.

input: $G = (V, E)$, s, t

$v \leftarrow s$
for $i \leftarrow 1$ to $100 \cdot n^3$
    pick at random a neighbor w of v
    if $w = t$ then halt and accept
    $v \leftarrow w$ reject

The analysis of the algorithm is based on the fact that if we start a random walk from a vertex s of an undirected vertex G, then all vertices in the connected component of s are likely to be visited at least once after $\Theta(n^3)$ steps.

By Savitch's theorem, every $\mathbf{RL}$ algorithm can be simulated deterministically using $O((\log n)^2)$ space. An algorithm due to Saks and Zhou [SZ95], however, does much better and shows a deterministic simulation of $\mathbf{RL}$ that requires only $O((\log n)^{3/2})$ space. There has been no improvement to this general result in the past 12 years.

For the particular problem ST-UCONN, instead, the story is much more interesting. Nisan [Nis94] proved that there exists an algorithm running in polynomial time and $O(\log^2 n)$ space (but the polynomial has very high degree). This is better than Savitch's algorithm that uses $O(\log^2 n)$ space and $n^{O(\log n)}$ time. Nisan's result actually applies to all of $\mathbf{RL}$. Nisan, Szemeredy and Wigderson [NSW92] then applied some of Nisan's techniques to prove that ST-UCONN can be solved in space $O((\log n)^{3/2})$ and superpolynomial time. (As mentioned before, the same bound was then extended to all of $\mathbf{RL}$ by Saks and Zhou.) This was improved to $O(\log^{4/3} n)$ space and superpolynomial time complexity by Armoni, Ta-Shma, Nisan and Wigderson [ATSWZ00]. Finally, Omer Reingold [Rei05] proved that ST-UCONN can be solved in space $O(\log n)$ and polynomial time. In terms of complexity classes, this shows that $\mathbf{SL} = \mathbf{L}$.

7 NL = coNL

In order to prove that these two classes are the same, we will show that there is an $\mathbf{NL}$ Turing machine which solves $\overline{\text{STCONN}}$. $\overline{\text{STCONN}}$ is the problem of deciding, given a directed graph G, together with special vertices s and t, whether t is *not* reachable from s. Note that $\overline{\text{STCONN}}$ is **coNL**-complete.

Once we have the machine, we know that $\mathbf{coNL} \subseteq \mathbf{NL}$, since any language A in **coNL** can be reduced to $\overline{\text{STCONN}}$, and since $\overline{\text{STCONN}}$ has been shown to be in $\mathbf{NL}$ (by the existence of our machine), so is A. Also, $\mathbf{NL} \subseteq \mathbf{coNL}$, since if $\overline{\text{STCONN}} \in \mathbf{NL}$, by definition STCONN $\in \mathbf{coNL}$, and since STCONN is $\mathbf{NL}$-complete, this means that any problem in $\mathbf{NL}$ can be reduced to it and so is also in **coNL**. Hence $\mathbf{NL} = \mathbf{coNL}$. This result was proved independently in [Imm88] and [Sze88].

7.1 A simpler problem first

Now all that remains to be shown is that this Turing machine exists. First we will solve a simpler problem than $\overline{\text{STCONN}}$. We will assume that in addition to the usual inputs G, s and t, we also have an input r, which we will assume is equal to the number of vertices reachable from s in G, including s.

Given these inputs, we will construct a non-deterministic Turing machine which decides whether t is reachable from s by looking at all subsets of r vertices in G, halting with YES if it sees a subset of vertices which are all reachable from s but do not include t, and halting with NO otherwise. Here is the algorithm:

input: $G = (V, E)$, s, t, r
output: YES if it discovers that t is not reachable from s, and NO otherwise
assumption: there are exactly r distinct vertices reachable from s

$c \leftarrow 0$
for all $v \in (V - \{t\})$ **do**
    non-deterministically guess if v is reachable from s
    if guess = YES **then**
        non-deterministically guess the distance k from s to v
        $p \leftarrow s$
        for $i \leftarrow 1$ to k **do**
            non-deterministically pick a neighbor q of p
            $p \leftarrow q$
        if $p \neq v$, reject
        $c \leftarrow c + 1$
if $c = r$ **then** return YES, otherwise return NO

It is easy to verify that this algorithm is indeed in **NL**. The algorithm only needs to maintain the five variables c, k, p, q, v, and each of these variables can be represented with $\log |V|$ bits.

Regarding correctness, notice that, in the algorithm, c can only be incremented for a vertex v that is actually reachable from s. Since there are assumed to be exactly r such vertices, c can be at most r at the end of the algorithm, and if it is exactly r, that means that there are r vertices other than t which are reachable from s, meaning that t by assumption cannot be reachable form s. Hence the algorithm accepts if and only if it discovers that t is not reachable from s.

7.2 Finding r

Now we need to provide an **NL**-algorithm that finds r. Let's first try this algorithm:

input: $G = (V, E)$, s
output: the number of vertices reachable from s (including s in this count)

$c \leftarrow 0$
for all $v \in V$ **do**
    non-deterministically guess if v is reachable from s in k steps
    if guess = YES **then**
        $p \leftarrow s$

```
        for i ← 1 to k do
            non-deterministically guess a neighbor q of p (possibly not moving at all)
            p ← q
        if p ≠ v reject
        c ← c + 1
return c
```

This algorithm has a problem. It will only return a number c which is at most r, but we need it to return *exactly* r. We need a way to force it to find all vertices which are reachable from s. Towards this goal, let's define r_k to be the set of vertices reachable from s in at most k steps. Then $r = r_{n-1}$, where n is the number of vertices in G. The idea is to try to compute r_k from r_{k-1} and repeat the procedure $n-1$ times, starting from $r_0 = 1$. Now here is another try at an algorithm:

input: $G = (V, E)$, s, k, r_{k-1}
output: the number of vertices reachable from s in at most k steps (including s in this count)
assumption: r_{k-1} is the exact number of vertices reachable from s in at most $k-1$ steps

```
c ← 0
for all v ∈ V do
    d ← 0
    flag ← FALSE
    for all w ∈ V do
        p ← s
        for i ← 1 to k − 1 do
            non-deterministically pick a neighbor q of p (possibly not moving at all)
            p ← q
        if p = w then
            d ← d + 1
            if v is a neighbor of w, or if v = w then
                flag ← TRUE
    if d < r_{k−1} reject
    if flag then c ← c + 1
return c
```

Here is the idea behind the algorithm: for each vertex v, we need to determine if it is reachable from s in at most k steps. To do this, we can loop over all vertices which are a distance at most $k-1$ from s, checking to see if v is either equal to one of these vertices or is a neighbor of one of them (in which case it would be reachable in exactly k steps). The algorithm is able to force all vertices of distance at most $k-1$ to be considered because it is given r_{k-1} as an input.

Now, putting this algorithm together with the first one listed above, we have shown that $\overline{\text{STCONN}} \in \mathbf{NL}$, implying that $\mathbf{NL} = \mathbf{coNL}$. In fact, the proof can be generalized to show that if a decision problem A is solvable in non-deterministic space $s(n) = \Omega(\log n)$, then $\overline{A}$ is solvable in non-deterministic space $O(s(n))$.

References

[ATSWZ00] Roy Armoni, Amnon Ta-Shma, Avi Wigderson, and Shiyu Zhou. An $o(\log(n)^{4/3})$ space algorithm for (s,t) connectivity in undirected graphs. *Journal of the ACM*, 47(2):294–311, 2000.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17:935–938, 1988.

[Nis94] N. Nisan. RL $\subseteq$ SC. *Computational Complexity*, 4(1), 1994.

[NSW92] N. Nisan, E. Szemeredi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 24–29, 1992.

[Rei05] Omer Reingold. Undirected ST-connectivity in log-space. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, pages 376–385, 2005.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[SZ95] M. Saks and S. Zhou. RSPACE(S) $\subseteq$ DSPACE($S^{3/2}$). In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*, pages 344–353, 1995.

[Sze88] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.