L and NL

Up until now, all time and space complexity bounds are at least linear.

$$f(n) \geq n$$

Sublinear time bounds

- UNINTERESTING in TM model,
- there is no time to read the input.

Sublinear space bounds

- are interesting
- TM can read input
- but does not have enough space to store it.

Needs some changes to the model.

L and NL

Consider a TM with two tapes:

- a read-only input tape
- a working tape

The read-only tape

- the TM can detect symbols but not change them
- head cannot move off portion containing input
- can detect when head is at left and right ends of tape

The working tape

- can be read and written normally
- only cells scanned on working tape contribute to space complexity

L and NL

The read-only tape is like the CD-ROM on a PC.

- input only
- may hold more data than can be stored in memory

L and NL

L is the class of languages decidable in logarithmic space on a deterministic Turing machine.

$$L = \text{SPACE}(\log n)$$

NL is the class of languages decidable in logarithmic space on a non-deterministic Turing machine.

$$NL = \text{NSPACE}(\log n)$$

L and NL

Claim: $A = \left\{0^k 1^k | k \geq 0\right\} \in L$

Remember our machine for deciding A:

M_1: On input w where w is a string,

- Scan across tape and *reject* if 0 is found to the right of a 1.
- Repeat while both 0s and 1s on tape:
  - scan across tape, crossing off single 0 and 1
- If 0s remain but no 1s, or 1s but no 0s, *reject*
- Otherwise, tape is empty, so *accept*.

This uses linear space!

L and NL

Claim: $A = \left\{0^k 1^k | k \geq 0\right\} \in L$

On working tape, keep track of

- number of 0s matched with 1s
- position of input head ($0 =$ leftmost position)

Algorithm

- position input head over first unmatched 0
- find first unmatched 1
- lather, rinse, repeat ...

Space required: two counters, each $\log n$.

L and NL

PATH $=$

$\{\langle G, s, t\rangle | G$ is a digraph with a path from s to $t\}$

We know that PATH $\in P$:

M: on input $\langle G, s, t\rangle$:

- Place mark on s
- Repeat until no additional nodes marked:
  - scan edges of G.
  - if edge (a, b) found from marked a to unmarked b,
  - then mark b.
- If t marked, *accept*, otherwise *reject*.

This algorithm uses linear space.

L and NL

Question: Does PATH have a deterministic logarithmic-space algorithm?

L and NL

Question: Does PATH have a deterministic logarithmic-space algorithm?

Answer: No one knows.

L and NL

Claim: PATH $\in NL$.

- NTM starts at s
- guesses a path to t
- records only position of current node at each step
- does *not* record entire path (too long)
- if t reached, *accept*.
- *reject* after m steps ($m =$ number of vertexes).

Savitch's Theorem Redux

Recall that Savitch's Theorem states that

- any non-deterministic TM can be converted to a deterministic TM
- $f(n)$-space converted to $f^2(n)$-space
- provided $f(n) \geq n$.

We can now improve the last condition.

Savitch's Theorem Redux

Recall that Savitch's Theorem states that

- any non-deterministic TM can be converted to a deterministic TM
- $f(n)$-space converted to $f^2(n)$-space
- provided $f(n) \geq \log n$.

Savitch's Theorem Redux

Proof is almost identical to original.

Need a more refined notion of configuration:

- state
- contents of work tape
- position of two tape heads
- but *not* the contents of the input tape.

Proof nearly identical. Space complexity is

$$\log n + \log f(n) + f(n)$$

where TM runs in $f(n)$ space.

If $f(n) \geq \log n$ then this expression is $O(\log n)$.

L and NL

As noted:

- PATH $\in NL$.
- It is not known if PATH $\in L$. (Good term project!)
- Moreover, no one knows if $L = NL$.

Sound familiar?

Naturally, we need a completeness theory.

L and NL

Informally, a language is NL-complete if it is one of the "hardest" languages in NL.

If $N \neq NL$, then no NL-complete language belongs to L.

As before, we define a language A to be NL-complete if

- $A \in NL$
- any language in NL is "easily" reduced to A.

Thus, any "efficient" algorithm for deciding A is an "efficient" algorithm for any language in NL.

L and NL

Any language in NL is "easily" reduced to A.

Question: So what is an "easy" reduction?

Answer: Polynomial-time reducibility won't do.

- all languages in NL (except $\emptyset$ and Σ^*) are polynomial-time reducible to one another!
- poly-time reducibility too crude to be useful.

Introduce notion of **log-space reducibility**.

Log-Space Reducibility

Definition: A **log-space transducer** is a TM with

- a read-only input tape
- a write-only output tape
- and a read/write work tape which holds only $O(\log n)$ symbols.

A log-space transducer M computes a function

$$f : \Sigma^* \longrightarrow \Sigma^*$$

where $f(w)$ is string on output tape when M halts with input w.

Log-Space Reducibility

We say that

- f is **log-space computable**.
- A is **log-space reducible** to B,
- written $A \leq_L B$,
- if A is mapping-reducible to B using a log-space computable f.

L and NL

A language B is *NL-complete* if

- $B \in NL$, and
- for every A in NL, $A \leq_L B$.

L and NL

Theorem: If $A \leq_L B$ and $B \in L$, then $A \in L$.

L and NL

Theorem: If $A \leq_L B$ and $B \in L$, then $A \in L$.

Let's try to follow the poly-time reducibility proof.

- A log-space algorithm for A maps input w to $f(w)$
- using log-space reduction f, and then
- applies log-space algorithm for B.

Question: What's wrong with this approach?

L and NL

Theorem: If $A \leq_L B$ and $B \in L$, then $A \in L$.

Let's try to follow the poly-time reducibility proof.

- A log-space algorithm for A maps input w to $f(w)$
- using log-space reduction f, and then
- applies log-space algorithm for B.

Problem: Storage needed to hold $f(n)$ may exceed polynomial bounds!

L and NL

Theorem: If $A \leq_L B$ and $B \in L$, then $A \in L$.

Instead, A's machine M_A computes symbols of $f(w)$ as needed by B's machine M_B.

- M_A keeps track of where M_B's input head would be on $f(w)$.
- Every time M_B moves,
  - M_A restarts computation of f on w from the beginning,
  - discarding all output except desired symbol of $f(w)$.
- result is time-inefficient, but
- only needs space for one symbol at a time.

Space-time trade-off.

L and NL

Theorem: PATH is NL-complete.

Idea: Construct a directed graph that simulates the non-deterministic log-space TM for A.

- nodes correspond to configurations
- there is an edge from one node to another if first configuration can yield the second in a single step.
- Machine accepts w whenever there is a path from the start configuration node to the accept configuration node.

L and NL

Theorem: PATH is NL-complete.

Suppose NTM M decides A in space $O(\log n)$.

Assume that M has a unique accepting configuration.

Given input w, we construct $\langle G, s, t\rangle$

- construction requires log space
- G is a directed graph
- s and t vertexes in G
- there is a path from s to t iff M accepts w.

L and NL

Theorem: PATH is NL-complete.

Given input w, we construct $\langle G, s, t \rangle$

- nodes of G are configurations of M on w
- for configurations c_1 and c_2, there is an edge (c_1, c_2) if c_2 is a possible next configuration from c_1.
- More precisely, (c_1, c_2) is an edge if
  - M's transition function indicates that
  - c_1's state and the symbols under the two heads
  - can yield the next state and tape movements
  - to turn c_1 into c_2
- s is the start configuration
- t is the unique accepting configuration.

L and NL

Why does the construction of $\langle G, s, t\rangle$ require log space?

Construct a transducer which maps w to a description of G.

Description consists of

- list of nodes
- list of edges

Space complexity:

- listing nodes is easy
- listing edges can be done in log space . . .

L and NL

Claim: Listing edges can be done in log space.

Can check in log space whether configuration c_1 of M can yield c_2 in one step.

- transducer must examine tape under head locations in c_1
- to see if c_2 is possible result.
- transducer checks all pairs (c_1, c_2) in turn.
- those that work are added to output tape.

This completes the proof.

L and NL

Corollary: If PATH $\in L$ then $L = NL$.

L and NL

P and NP

- it is not known whether $P \neq NP$
- it is not known whether coNP $\neq$ NP

L and NL

- it is not known whether $L \neq NL$
- what about whether $NL \neq$ coNL?

NL and coNL

Theorem: $NL =$ coNL

- surprising and counterintuitive . . .
- are there other surprises in store?

NL and coNL

Theorem: $NL =$ coNL

Idea:

- show that $\overline{\text{PATH}} \in NL$.
- result follows because PATH is NL-complete.
- construct an algorithm M with an accepting computation whenever input G does *not* contain a path from s to t.

NL and coNL: Idea

First, an easier problem.

Let c be the number of nodes in G reachable from s.

- assume c is provided as input
- show how to use c to compute $\overline{\text{PATH}}$.
- later show how to compute c.

NL and coNL: Idea

Given s, t, and c, the machine M operates as follows.

- One by one, M goes through nodes of G,
- and guesses whether each one is reachable from s.
- If it guesses u is reachable,
- M verifies by guessing a path from s to u.
- If branch does not verify in m steps, *reject*.
- If branch guesses t is reachable, *reject*.
- Keeps count of reachable nodes.
- When all nodes checked, *accept* if count is c, otherwise *reject*.

NL and coNL: Idea

In other words, M

- non-deterministically selects exactly c nodes reachable from s
- not including t
- and proves each is reachable by guessing the path.
- Because c is count of reachable nodes, remaining nodes are not reachable.

NL and coNL

Now we show how to calculate c, the number of reachable nodes from s.

- non-deterministically log-space procedure
- at least one branch has correct value for c
- remaining branches reject.

NL and coNL

For each i, $0 \leq i \leq m$,

- let A_i be the set of nodes of G of distance $\leq i$ from s
- $A_0 = \{s\}$
- $A_{i-1} \subset A_i$
- let $c_i = |A_i|$
- we will show how to calculate c_i from c_{i-1}
- repeated application yields c

NL and coNL

To calculate c_{i+1} from c_i, go through all nodes of G and accumulate A_{i+1}.

To determine whether $v \in A_{i+1}$,

- Go through all nodes of G,
- guess which nodes are in A_i,
- verify by guessing a path of length $\leq i$ from s.
- For each $u \in A_i$, check whether (u, v) is an edge in G.
- If so, add v to A_{i+1} and move on to next vertex.
- Keep track of nodes known to be in A_i.
- When c_i such nodes identified, we know $v \notin A_{i+1}$, so move on to next vertex.

NL and coNL

First: compute c.

On input G, s, t:

```
c_0 := 0
for i ∈ 1..m do
    c_i := 0; d := 0
    for each node v in G do
        for each node u in G do
            if (guess u is reachable from s)
              then if (guess path of length ≤ i)
                        then d := d + 1
                             if (u, v) is an edge in G
                               then c_i := c_i + 1
                               end if
                        else  reject
                        end if
              end if
            if d ≠ c_{i−1}
              then reject
              end if
```

NL and coNL

Now, given c, accept iff there is no path from s to t:

$c := c_m$
for each node u in G **do**
  if (guess u is reachable from s)
    then if (guess path of length $\leq m$)
      then if $(u = t)$
        then *reject*
        else d := d+1
        end if
      end if
    end if
  end for
if $d \neq c$
  then *reject*
  else *accept*

Extra Credit

The Japanase game GO-MOKU is

- played by two players: X and Y
- on a 19×19 board
- the first player to acheive 5 consecutive markers
- on a row, column, or diagonal wins.

Generalize the game to an $n \times n$ board.

Show that the problem of detemining whether X has a winning strategy from any position in generalized GO-MOKU is in PSPACE.