Logarithmic Space and *NL*-Completeness

CSC 463

March 25, 2020

Motivation

- Many things that people care about in real life can use much memory: genomes, the web graph etc.
- Main memory in a computer is typically much smaller than memory available on disk.
- We want to see if there are algorithms for certain problems that use small amounts of main memory, so that large amounts of data can be manipulated on a computer without storing all of it at once in main memory.

The Computational Model

- Input with n bits already takes linear space to store, so we must precisely define what we mean when we say that an algorithm takes sublinear space.
- We consider a two-tape Turing machine where one tape is a read-only tape containing the input, and another tape is a "work" tape that can be freely used.
- Only the space used on the work tape counts towards the space complexity.
- Define $\mathbf{L} = SPACE(\log n)$, and $\mathbf{NL} = NSPACE(\log n)$.

The Computational Model: Examples

- Intuitively, an algorithm using $O(\log n)$ space in this model stores a fixed number of pointers, independent of n, and manipulates them in some way.
- **Example:** Given an n-bit string s, deciding if s has more ones than zeros is a problem in L. Keep two counters $count_0, count_1$ for the number of zeros and ones in s and test if $count_1 > count_0$. These take $O(\log n)$ space in total.

PATH is in NL

- Let **PATH** be the problem to checking if a **directed** graph has a directed path from starting vertex s to end vertex t.
- We know that **PATH** $\in$ **P** using algorithms such as depth-first search or breadth-first search. However, while these algorithms are efficient, they also use $O(n)$ space.
- We can reduce the space complexity to non-deterministic logarithmic space.

Path is in NL

- The algorithm stores up to three variables v_{cur}, v_{next}, l.
  1. Start with $v_{cur} = s, v_{next} = \emptyset, l = 0$.
  2. Choose v_{next} nondeterministically from a vertex pointed to from v_{cur} and let $l := l + 1$.
  3. If $v_{next} = t$, accept. Otherwise, if $l < n$, set $v_{cur} = v_{next}$ and repeat Step 2.
  4. If $l == n$, reject since a shortest $s - t$ path uses less than n additional vertices.
- A branch of this algorithm is guaranteed to find an $s - t$ path if one exists.
- This uses $O(\log n)$ space on a nondeterministic machine.
- Savitch's theorem implies that **PATH** $\in DSPACE(\log^2 n)$. However, this saving in space comes at the expense of much increased time.

NL-Completeness

- It is believed that *PATH* for **directed** graphs cannot be done in deterministic log-space. We define the notion of *NL*-Completeness using log-space reducibility.
- We say a function $f : \Sigma^* \mapsto \Sigma^*$ is **logspace computable** if there is a three-tape Turing machine M with
  1. One input read-only tape that can move left or right.
  2. One work tape of size $O(\log n)$ that can move left or right.
  3. A write-only output tape that can only move right.

  such that given an input w, M halts with $f(w)$ on its output tape.
- Equivalently, given inputs (x, i), there is a two-tape Turing machine using $O(\log n)$ space that computes the i^{th} bit of $f(x)$.

NL-Completeness

- We say that A is logspace reducible to B ($A \leq_L B$) if there is a logspace computable function f such that

$$w \in A \leftrightarrow f(w) \in B.$$

- If $A \leq_L B$ and $B \in L$, so is $A \in L$.
- If $A \leq_L B$ and $B \leq_L C$, then $A \leq_L C$.
- **Proof Sketch:** Given two logspace computable functions f, g, their composition $h = g(f(x))$ is logspace computable since a logspace Turing machine can store single bits of $f(x)$ on its work tape.
- A language B is **NL**-Complete if $B \in NL$ and $A \leq_L B$ for all $A \in NL$.

PATH is NL-Complete

- We have already shown that $PATH \in NL$. So now we need to show that given any $A \in NL$, there is a logspace computable function showing $A \leq_L PATH$.
- We will use the ideas of Savitch's theorem to help us prove this.
- A configuration of a log-space Turing machine M that decides A can be specified by:
  - A cell position on its reading tape and the symbol that is being read
  - The contents of the work tape

  All together this takes $O(\log n)$ space if input has size n.

PATH is NL-Complete

- Recall that configuration graph G_M of M is a graph where the vertices are its configurations, and there is a directed edge (c_1, c_2) if c_2 can be obtained from c_1 by a transition of M.
- We will assume that M has starting configuration c_0 and a unique accepting configuration c_{accept}. M accepts its input if and only if G_M has a path from c_0 to c_{accept}
- To complete the argument, we need to argue that G_M can be computed from a description of M in logspace.

PATH is NL-Complete

- We create the graph G_M by first listing its vertices, then its edges.
- The vertices can be listed in logspace since every potential configuration has size $O(\log n)$ and can tested if it is a legal configuration for M.
- Each edge can be listed in log space since given two configurations (c_1, c_2), one can test if c_2 can follow from c_1 in $O(\log n)$ space.
- All together, this shows that G_M can be created in logspace for a machine M deciding $A \in \mathbf{NL}$, and hence there is a reduction $A \leq_L \mathbf{PATH}$.
- Corollary: $\mathbf{NL} \subseteq \mathbf{P}$.

NL = coNL

- Define **coNL** as the set of languages where the complement $\overline{A} \in$ **NL**.
- We do not expect **NP** = **coNP**, so it is perhaps surprising that **NL** = **coNL**.
- To prove this, we need to show that $\overline{\textbf{PATH}} \in$ **NL** : checking if there is no $s - t$ path in a directed graph is in **NL**.
- Since $\overline{\textbf{PATH}}$ is **coNL**-Complete, showing $\overline{\textbf{PATH}} \in$ **NL** implies **NL** = **coNL**.

NL = coNL: Proof Part 1

- To show $\overline{\textbf{PATH}} \in \textbf{NL}$, we firstly consider a problem where we are given more information.
- Suppose we have a directed graph G, vertices s, t, and a number c, where c is the number of vertices reachable from s, and we want to check if there is no $s - t$ path.
- Let $R \subseteq V$ be the set of reachable vertices from s.
- A non-deterministic algorithm can guess R in log-space by checking if each vertex v lies in R or not, and verify that the guess was correct by checking $|R| = c$.
- Once R is obtained and we have verified $t \notin R$, we know for sure that there is no $s - t$ path.
- Note that $check\_path(s, u, l)$: checking if there is an $s - u$ path of length $\leq l$ for any $l \leq |V|$ can be done in NL.

NL = coNL: Pseudocode Part 1

```
test_no_path(G = (V,E), s, t, c):
    d = 0
    for u in V:
        guess_u = T or F nondeterministically
        if guess_u = T:
            check_path(s, u, |V|)
            if u = t: reject
            else: d = d + 1
    ## at this point, we have guessed a subset
    ## of vertices reachable from s
    if d != c: reject
    else: accept
```

- Hence with the additional variable c, we can certify if there is no $s - t$ path in **NL**.

NL = coNL: Proof Part 2

- Now we need to show that we can compute c, the number of reachable vertices in logspace. We do this using a technique called **inductive counting**.
- Let R_i be the set of vertices in G reachable from s with a path of length $\leq i$. Define $R_0 = \{s\}$ and $c_i = |R_i|$. We want to compute $c = c_{|V|}$.
- **Observation:** $v \in R_{i+1}$ iff there is an edge (u, v) for some $u \in R_i$.
- We can use this observation to compute c_{i+1} from c_i.

NL = coNL: Pseudocode Part 2

```
compute_c(G, s, t):
    old_c = 1
    for i = 0 to (|V|-1):
        new_c = 1 ## new_c = c_{i+1}, old_c = c_i
        for each v != s in V:
            d = 0
            for u in V:
                guess_u = T or F nondeterministically
                if guess_u = T:
                    check_path(s,u,i)
                    d = d + 1
                    if (u, v) is an edge:
                        new_c = new_c + 1
                        break
            if d != old_c: reject
        old_c = new_c
    return new_c
```

NL = coNL: Completing the proof

- In the i^{th} iteration of the outer for loop, if $v \in R_{i+1}$, some branch finds $u \in R_i$ where $(u, v) \in E(G)$, so v is counted in c_{i+1}.
- Otherwise, if $v \notin R_{i+1}$, then some branch certifies there is no edge between any vertex in R_i and v since we know c_i, so that branch ensures v is not counted in c_{i+1}.
- Since there is a branch where each iteration correctly computes c_{i+1}, then *compute_c* correctly returns $c_{|V|}$, and it can be done in **NL**.

NL = coNL: Completing the Proof

- So to design an algorithm for $\overline{\textbf{PATH}}$ given inputs G, s, t, we run $compute\_c(G, s, t)$ to obtain $c_{|V|}$ and then $test\_no\_path(G, s, t, c_{|V|})$.
- Both parts can be done in NL, so overall $\overline{\textbf{PATH}} \in \textbf{NL}$.
- Therefore, by **coNL**-completeness of $\overline{\textbf{PATH}}$, we conclude $\textbf{NL} = \textbf{coNL}$.
- This means that we can simplify proofs for showing problems are in **NL** or complete for **NL** by showing that their complements are in **NL** or complete for **NL**. Eg. $\textbf{2SAT} \in \textbf{NL}$ iff $\overline{\textbf{2SAT}} \in \textbf{NL}$.
- In general, $\textbf{NL} = \textbf{coNL}$ implies that $\mathrm{NSPACE}(s(n)) = \mathrm{coNSPACE}(s(n))$ for space-constructible $s(n) \geq \log n$.