Theory of Computation: Log-space

Languages in L

- EVEN : The set of strings with an even number of 1s.
- EVEN is in L: We basically need to keep a counter of the number of 1s in the input string. And later check if this is even by checking the last bit. The space required is $O(\log n)$.

Languages in NL

- PATH: The set of all $< G, s, t >$ such that G is a directed graph which has a path from s to t.
- PATH $\in$ NL: First, if there is a path from s to t, then there is one of length at most n.
- Create a non-deterministic walk starting at s, making a non-deterministic choice of a neighbour from the current vertex and stopping after n steps. If the walk ends at t then this is a desired path.
- $O(\log n)$ space required: Only need to know the number of steps so far and the index of the current vertex.

NL = L?

- It is not known whether PATH belongs to L.This is an open question.
- It is quite possible that even 3-SAT could belong to L.
- Consequence of $3 - SAT \in L$: Recall that $NSPACE(f(n)) \in DTIME(2^{f(n)})$ for space constructible f.
  So, $L \subseteq NL \subseteq P$.
  $3 - SAT \in L \implies NP = P$. (Does not imply $L = NL$!)

NL-completeness

- Polynomial time reductions are too expensive!
- Logspace computable functions: A function $f : \{0,1\}^* \to \{0,1\}^*$ that is polynomially bounded (there is a c such that $f(x) \leq |x|^c$ for all x) and the languages $L_f = \{< x, i > | f(x)_i = 1\}$ and $L'_f = \{< x, i > | i \leq |f(x)|\}$ are in L. Eg. $f(x) = |x|$.

NL-completeness

- Logspace reducibility: A language B is logspace reducible to language C, denoted as $B \leq_l C$, if there is a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ that is logspace computable and $x \in B$ iff $f(x) \in C$ for every x.
- NL-completeness: C is NL-complete if it is in NL and for every B in NL, $B \leq_l C$.

NL-completeness

- Log space reductions: For logspace computable functions, it is possible to compute in $O(\log n)$ space whether the i^{th} bit of $f(x)$ is 1, and whether all of $f(x)$ has been computed or not.
- So, log space reductions can also be thought of as reductions where the output tape (whose space does not count towards space bound of the machine) is a write-only tape: you can write a bit or move to the right; you cannot move left to reread a previous bit.
- Actually, the two notions are equivalent.

Composition of logspace computable functions

- For logspace computable functions f, g, h such that $h(x) = g(f(x))$ is also logspace computable.
- Proof: Let M_f and M_g be logspace machines computing $f(x)_i$ and $g(y)_j$ respectively.
- We will compute M_h to output $g(f(x))_j$. Input tape of M_h has $< x, j >$ written.
- M_h has to simulate M_g on $f(x)$ and then read the j^{th} bit from the output. So it tries to maintain M_g's bit by bit simulation on $f(x)$ - cannot do the whole thing as it will require much more than logspace.

Composition of logspace computable functions

- Suppose M_g needs to know the bit at the i^{th} cell of $f(x)$ for its simulation.
- M_h stores the current worktape of M_g safely.
- It invokes M_f on input $< x, i >$ to get $f(x)_i$.
- Then it resumes simulation of M_g on this bit.
- Total space required $= (O(\log(|g(f(x))| + |x| + |f(x)|)))$. As $|f(x)| \leq \text{poly}(x)$ and similar properties for g, this becomes $O(\log(|x|))$.

Composition of logspace computable functions

- Similar argument to show that $L'_h = \{< x, i > | i \leq h(x)\}$ is in L: Again the machine for h has to "pretend" that it also has access to $f(x)$ on its input tape and not just $< x, j >$.
- This shows that h is logspace computable.

Transitivity of logspace reductions

- If $B \leq_l C$ and $C \leq_l D$ then $B \leq_l D$: B reduces to C by logspace computable function f, and C to D by logspace computable function g. We know that h such that $h(x) = g(f(x))$ is also logspace computable.
- If $B \leq_l C$ and $C \in L$ then $B \in L$: Let f be the reduction from B to C and g be the funtion such that $g(y) = 1$ iff $y \in C$. Then h such that $h(x) = g(f(x))$ is such that $h(y) = 1$ iff $y \in B$ and it requires deterministic computation taking logspace. So B is in L.
- In particular if an NL-complete language is in L iff NL $=$ L.

PATH is NL-complete

- Note: If PATH is in L then NL = L.
- We have seen that PATH is in NL.
- PATH is NL-hard: Take L to be in NL that is decided by an $O(\log n)$-space nondeterministic machine M.
- Need to define a logspace computable function f for the reduction $L \leq_l PATH$.
- For input x, $f(x)$ will be the configuration graph $G_{M,x}$: each configuration in a logspace machine can be described in $O(\log n)$ bits; $G_{M,x}$ has $2^{O(\log n)}$ vertices.

PATH is NL-complete

- Correctness of reduction: $G_{M,x}$ has a path from C_s to C_t iff M accepts x.
- How to compute $f(x)$: The graph can be represented as an adjacency matrix: contains 1 in position (C, C') if there is an edge from C to C' in $G_{M,x}$.
- We need to show that the adjacency matrix can be computed by a logspace reduction: need to describe a logspace machine that can compute any desired bit in it.
- Given a C and C', a deterministic machine can in space $O(|C| + |C'|) = O(\log(|x|))$ examine if the two configurations have valid form and if C can transition to C' according to the transition function of M.

Immerman-Szelepcsenyi Theorem

- Statement: For every space constructible $S(n) \geq \log n$, $NSPACE(S(n)) = coNSPACE(S(n))$.
- Corollary: NL = coNL.
- Comment: Space complexity classes behave very differently from time complexity classes:
  Savitch's Theorem has no analogue in time complexity.
  I-S Theorem has no analogue in time complexity.

Proof of I-S Theorem

- Take a problem Π in NL with a $O(S(n))$-space machine M.
- Configurations are of size $O(S(n))$,; Configuration graph has $2^{O(S(n))}$ vertices.
- An input x belongs to Π iff $G_{M,x}$ has a path from C_s to C_t.

Proof of I-S Theorem

- If $x \in L$ there is an algorithm to verify if $G_{M,x}$ has a path from C_s to C_t: Starting from C_s, guess a path of length at most $2^{O(S(n))}$ till C_t.
- If $x \notin L$ we need an algorithm to verify if $G_{M,x}$ does not have a path from C_s to C_t. (Then $\overline{L} \in NL$)
- Notation: $\mathcal{C}_i$ is the set of all vertices C in $G_{M,x}$ that are reachable from C_s in exactly i steps.
- Note that $\mathcal{C}_0$ only contains C_s.

Proof of I-S Theorem

- Primer: Suppose I know that the number of vertices in $\mathcal{C}_i$ is m_i, can I check if a given vertex C_v is in $\mathcal{C}_i$ or not?
- Each $m_i = 2^{O(S(n))}$, which can be stored in $O(S(n))$ space.
- If C_v belongs to $\mathcal{C}_i$ then again we can guess an i-length path. What if C_v does not belong?
- Design a new algorithm: For each C_u, $u \neq v$, check if C_u belongs to $\mathcal{C}_i$.
  If at the end the number of u for which it is verified that C_u belongs to $\mathcal{C}_i$ is m_i then it must be the case that $C_v \notin \mathcal{C}_i$.
  If the number is $< m_i$ then it must be the case that $C_v \in \mathcal{C}_i$.

Proof of I-S Theorem

- So how do we find m_i correctly: It must be correct in order for the previous algorithm to work.
- Algorithm to find m_i if m_{i-1} is correctly known: First, let's design an algorithm that can check if a C_v belongs or not to $\mathcal{C}_i$ if m_{i-1} is known.
- Take a C_v. Check for each vertex C_w that has an edge to C_v whether or not it belongs to $\mathcal{C}_{i-1}$ (Previous algorithm can be used each time). This can answer whether or not C_v belongs to $\mathcal{C}_i$.

Proof of I-S Theorem

- Algorithm to find m_i if m_{i-1} is correctly known:
- Initially $m_i = m_{i-1}$.
  Run through all C_u and check whether it belongs to C_i or not from previous algorithms.
  If there is an i-path from C_s, then increment m_i.

Proof of I-S Theorem

- Now we know $\mathcal{C}_0 = \{C_s\}$ and $m_0 = 1$.
- Iteratively, find $m_{2^{O(S(n))}}$, as the path from C_s to C_t can be of length at most $N = 2^{O(S(n))}$. Counter for $i \leq N$ can be stored in $O(S(n))$ space.
- Finally, check whether C_t belongs to $\mathcal{C}_N$ or not.