9. L and NL

- We defined the robust complexity classes
  to capture interesting computational phenomena.
- Our goal in the following weeks
  is to study each of these classes in more detail.
- The computational phenomenon captured by a class
  will be made precise in terms of <u>computational problems</u>
  that are <u>complete</u> for the class:
  - ↳ they lie <u>in</u> the class (can be solved with the given resources)
  - ↳ they are <u>hard</u> for the class,
    i.e., every problem in the class can be reduced to them.

<u>Goal of this section:</u>

- ↳ Understand the computational problems that can be solved
  using logarithmic space (and potentially non-determinism).
- ↳ Understand the models of computation
  that can be used to solve these problems

  (a second requirement on the robust complexity classes was
  that they should be insensitive to the choice
  of the model of computation).

<u>Roughly:</u>

- NL is all about paths (existence and absence).
  We already know this since Savitch and Immerman & Szelepcsényi.
- L is about basic arithmetic.

9.1 Reductions and Completeness in Logarithmic Space

Goal: • Recall the (general) notions of many-one reductions, hardness, and completeness.
• Introduce logspace-computable reductions.

Definition:
Let $\mathcal{R}$ be some set of functions from $\Sigma_1^* \to \Sigma_2^*$.
A language $A \subseteq \Sigma_1^*$ is *$\mathcal{R}$-many-one reducible* to $B \subseteq \Sigma_2^*$,
if there is a function $f \in \mathcal{R}$, called the *reduction*,
so that $x \in A$ iff $f(x) \in B$.

We also write $A \leq_m^{\log} B$.

- Intuitively, $A \leq_m^{\log} B$ means that membership in A can be decided by deciding membership in B.
- "many-one" refers to the fact that the reduction need not be injective (injective reductions we called *one-one*).

Definition:
Let C be a complexity class and $\mathcal{R}$ be a set of functions.
- A language B is *C-hard wrt. $\mathcal{R}$-many-one reductions*, if *for all* $A \in C$ we have $A \leq_m^{\mathcal{R}} B$.
Intuitively, B is at least as hard as any problem in C.
- Language B is *C-complete wrt. $\mathcal{R}$-many-one reductions*, if • $B \in C$ and • B is C-hard wrt. $\mathcal{R}$-many-one reductions.

- Intuitively, B is a hardest element in C.
- We also phrase $B \in C$ as C is an upper bound and B is C-hard as C is a lower bound.

To be useful, reductions should fulfill two properties.

(1) The reduction should be weaker than the presumably harder one of the two complexity classes we are comparing.
Otherwise, (a part of) the computation can be carried out when computing the reduction.
In particular, if A is R-many-one reducible to B and $B \in C$, then $A \in C$ should follow.
In short: C should be closed under R-many-one reductions.

(2) Reducibility should be transitive.
In particular, if A is C-hard and $A \leq_m^R B$, then also B is C-hard.

- Popular classes R are
  - ↳ the polynomial-time computable functions ($\leq_m^{poly}$) and
  - ↳ the logarithmic-space computable functions ($\leq_m^{log}$).
- Many-one reductions are also called Karp reductions in the literature.
- There are also Turing reductions that may invoke the harder problem multiple times (like an oracle, we return to this later on).
  Turing reductions are useful for proving undecidability results.

Definition (Logspace reducible):

- Recall that a function $f: \Sigma_1^* \to \Sigma_2^*$ is *logspace computable* if there is a DTM with
  - ↳ read-only *input* tape over Σ_1
  - ↳ write-only (write and move) *output* tape over Σ_2
  - ↳ read-write *work* tape over Γ

  that
  - ↳ is *total*, i.e., halts on all inputs $x \in \Sigma_1^*$,
  - ↳ upon halting has written $f(x) \in \Sigma_2^*$ onto the output tape, and
  - ↳ is $O(\log n)$-space bounded (work tape).

  The DTM is also called *logspace transducer* in the literature.

- Now $A \subseteq \Sigma_1^*$ is *logspace-many-one reducible* to $B \subseteq \Sigma_2^*$, denoted by $A \leq_m^{\log} B$, if there is a logspace computable $f: \Sigma_1^* \to \Sigma_2^*$ so that for all $x \in \Sigma_1^*$ we have

$$x \in A \iff f(x) \in B.$$

Lemma (Transitivity):

- If $f: \Sigma_1^* \to \Sigma_2^*$ and $g: \Sigma_2^* \to \Sigma_3^*$ are logspace computable, then so is $g \circ f: \Sigma_1^* \to \Sigma_3^*$.
- In particular, if $A_1 \leq_m^{\log} A_2$ and $A_2 \leq_m^{\log} A_3$, then $A_1 \leq_m^{\log} A_3$.

Proof:

- If function f is computable by a logspace-bounded DTM, then $|f(x)|$ is polynomial in $|x|$.
  This is because the DTM can run for at most polynomial time
  ↳ before repeating a configuration.

- Suppose f and g are computable by the logspace-bounded DTMs M and N, respectively.
  To compute $g(f(x))$, the idea is to simulate N on input $f(x)$.
  The problem is that we cannot just compute $f(x)$ in advance – it would not fit onto the $O(\log |x|)$ work tape.
- Instead the new machine will pretend that it has a hypothetical input tape with $f(x)$ written on it.
  Technically, during the simulation of N, we will provide the symbols of $f(x)$ on demand.
  Whenever N wishes to read the ith symbol of $f(x)$, we compute this symbol as follows:
  - ↳ The number i is given to a subroutine that simulates M on input x from scratch.
  - ↳ The M-simulator counts and throws away all symbols up to the ith.
  - ↳ The ith symbol is returned to the caller.
- We need space for the work tapes of M and N, and two counters that count up to $|f(x)|$.
  The first counter is the head position of N on the hypothetical input $f(x)$.
  The second counter is for the M-simulator. □

Lemma: If $A \leq_m^{\log} B$ then $A \leq_m^{\text{poly}} B$.

It is not known whether logspace reductions are strictly stronger than polynomial time reductions.

Lemma:

- For all $A \subseteq \Sigma^*$ we have $A \in L$ iff $A \leq_m^{\log} \{0,1\}$.
- All languages $A \in L$, $A \neq \Sigma^*$, $A \neq \emptyset$, are L-complete.

This means a reduction is only meaningful within a class that is computationally stronger than the reduction itself.

Lemma: Let $A \leq_m^{\log} B$.

If $B \in \overset{L}{\underset{P}{NL}}$ then $A \in \overset{L}{\underset{P}{NL}}$.

Lemma:

- If A is NL-hard under logspace many-one reductions and $A \in L$, then $L = NL$.
- If A is P-hard under logspace many-one reductions and $A \in NL$, then $NL = P$.

9.2 Problems Complete for NL

Let PATH be the problem:

Given: A directed graph $G = (V, \to)$ and $s, t \in V$.

Question: Is there a path from s to t?

We use ACYCPATH to refer to the same problem but where G in the input is an acyclic graph.

Goal: Show that PATH is NL-complete.

Problem: · We do not have an NL-hard problem.
· Have to show that every problem in NL admits a reduction to PATH.
(This is similar to the hardness result for SAT by Cook & Levin)
· Once we have this, NL-hardness of a problem A can be shown by reducing PATH to A.

Approach: Give a logspace-computable reduction f_M for every NTM M that solves a problem in NL.

Theorem: PATH is NL-complete.

Proof: We have to show
(1) PATH $\in$ NL and (2) PATH is NL-hard.

↳ We already know (1): Non-deterministically guess a path and enforce termination with a binary counter that counts up to $|V|$.

↳ To show hardness, let A be a problem in NL.
Let M be an NTM that solves A in $O(\log n)$ space.
We show a logspace-computable function f_M that satisfies the following.
Given an input x (to be checked for membership in A)
the result value $f_M(x) = (G, s, t)$ satisfies
the directed graph G contains a path from s to t
iff M accepts x.

- The nodes of G are the configurations of M on input x.
  For configurations c_1 and c_2 of M on x,
  the pair (c_1, c_2) is an edge of G
  if c_2 is a possible next configuration of M from c_1.
  Node s is the initial configuration $c_0(x)$ of M on input x.
  Machine M can be assumed to have
  a single accepting configuration c_f (clear tape).
  This configuration is node t.
- Clearly, whenever M accepts x, some computation accepts,
  which corresponds to a path from s to t in G.
  Vice versa, if there is a path from s to t in G,
  some computation of M when started on input x accepts.
- It remains to show that f_M is computable in logspace.
  The transducer that outputs (G, s, t) on input x
  works as follows:
  We describe G by listing its nodes and edges.

Listing the nodes: Easy, because each node is a configuration of M on input x,
it can be represented by $d \cdot \log |x|$ space for some constant d.
The transducer enumerates all strings of length $\leq d \cdot \log |x|$
and tests each for being a valid configuration.
It outputs those that pass the test.

Listing the edges: Similar, but we enumerate pairs of strings,
check them for being configurations c_1 and c_2,
and check whether M's transition relation contains $c_1 \to_M c_2$.

□

Note: • Configuration graphs of dominating TMs are acyclic.
So it seems that also ACYCPATH should be NL-complete.
This is indeed true.
• NL-hardness, however, does not immediately follow
from the above reduction.
The problem is that we cannot check
the constructed graph for being acyclic.
• Instead, we give another reduction from PATH to ACYCPATH.

Theorem: PATH $\leq_m^{\log}$ ACYCPATH, so ACYCPATH is NL-complete.

Proof: Homework.

A famous result due to Cook & Levin (SAT) and Karp (3SAT)
is that 3SAT is NP-complete.
The restriction to two literals per clause is (presumably) easier to solve.

Theorem: 2SAT is NL-complete.

• We first show that
$\overline{\text{2SAT}} \in$ NL, which means that 2SAT $\in$ co-NL.
(unsatisfiability)
By Immerman & Szelepcsényi, co-NL = NL.
• To establish NL-hardness, we then
reduce $\overline{\text{ACYCPATH}}$ to 2SAT.
(unreachability)
Like ACYCPATH, $\overline{\text{ACYCPATH}}$ is also NL-hard.

Lemma: $\overline{\text{2SAT}} \in$ NL.

- To establish the NL-upper bound, we encode a 2CNF formula F into a graph $G(F)$ as follows:
  - the vertices are the variables of F and their negations,
  - there are edges $\alpha \to \beta$ and $\neg\beta \to \neg\alpha$ iff $\neg\alpha \vee \beta$ is a clause in F,
  - there is an edge $\neg\alpha \to \alpha$ iff α is a clause in F.
- Edges correspond to <u>implications</u>:

  $\neg\alpha \vee \beta \equiv \alpha \to \beta \equiv \neg\beta \to \neg\alpha$.

  A clause α is equivalent to $\alpha \vee \alpha$, and

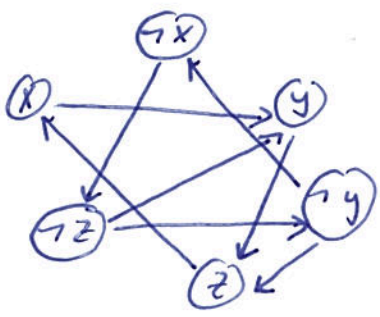
  $\alpha \vee \alpha \equiv \neg\neg\alpha \vee \alpha \equiv \neg\alpha \to \alpha$.

  Also paths in $G(F)$ correspond to implications, because implication is transitive.
- Note that $G(F)$ has a <u>symmetry</u>:

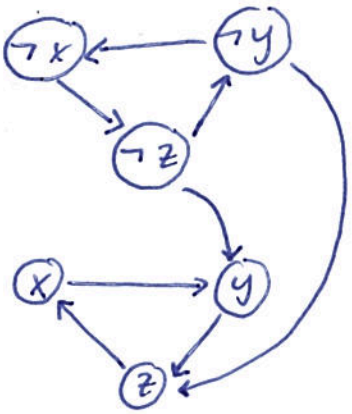
  $\alpha \to \beta$ iff $\neg\beta \to \neg\alpha$.

<u>Example</u>:
Let $F \equiv (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$.

$G(F)$:



=



Lemma: F is unsatisfiable iff there are paths from x to $\neg x$ and back from $\neg x$ to x.

Proof:

- "if" ($\Leftarrow$): • Towards a contradiction, assume the paths exist but still assignment ℓ satisfies F.
  Assume $\ell(x) = 1$ and thus $\ell(\neg x) = 0$.
  The case $\ell(x) = 0$ is symmetric.
  • Since there is a path from x to $\neg x$, there is edge $\alpha \to \beta$ on the path with $\ell(\alpha) = 1$ and $\ell(\beta) = 0$.
  • But the edge $\alpha \to \beta$ corresponds to the clause $\neg\alpha \vee \beta$ in F, which evaluates to $\ell(\neg\alpha \vee \beta) = 0$ and thus $\ell(F) = 0$. ↯

- only if" ($\Rightarrow$): • We proceed by contraposition and show that if there is no variable with such paths, then ℓ is satisfiable.

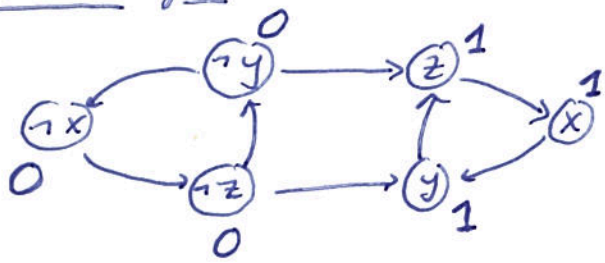
Repeat the following to assign truth values to nodes, until all nodes have a value:
(1) Pick a node α with no path from α to $\neg\alpha$ (α a literal).
(2) Assign true to all nodes reachable from α.
(3) Assign false to all nodes that reach $\neg\alpha$.

Note that:
- For all β, β and $\neg\beta$ are assigned values in the same round (by symmetry).
- There can be no clash: There is no path from α to β and to $\neg\beta$.

By symmetry of $G(F)$, there would be paths
from β and from $\neg\beta$ to $\neg\alpha$.
Hence, there is a path from α to $\neg\alpha$. ↯ Choice of α.

- There is no path from α to a node assigned false
in an earlier round.
Otherwise, α would have been assigned false.
- If the graph is not fully labelled,
there is a node that can be chosen.
Indeed, α is unlabelled iff $\neg\alpha$ is unlabelled.
If there is a path from α to $\neg\alpha$,
we are sure there is no path from $\neg\alpha$ to α
and we can select that node.
- After all nodes have been assigned,
no edge leads from true to false.
Hence, the assignment satisfies F. □

On the example:



□

Proof ($\overline{2SAT} \in NL$): Guess the two nodes and the two paths. □

Lemma: $\overline{ACYCPATH} \leq_m^{log} 2SAT$.
As $\overline{ACYCPATH}$ is NL-hard (homework), so is 2SAT.

Proof: • Given (G, s, t), we introduce a clause

$$\neg x \vee y$$

for every edge $x \to y$ in G.

Moreover, we add clauses s and $\neg t$ for the start and the target vertex.

- This instance of 2SAT is satisfiable iff there is no path from s to t in G. □

9.3 Problems in L

Goal: Show that the board game NIM is in L.

History:
- Computers for NIM since 1940.
- Ludwig Erhard (later chancellor, Wirtschaftswunder) lost against a NIM computer in 1951 (at Berliner Industrieausstellung).

Rules:
- Given is a collection of piles of sticks.
- In a move, a player may remove a non-zero number of sticks from a single pile.
- The players alternately take turns.
- The player who removes the very last stick wins.

Example:

$2\,2\,1 \xrightarrow{P_1} 2\,2\,0 \xrightarrow{P_2} 1\,2\,0 \xrightarrow{P_1} 1\,1\,0 \xrightarrow{P_2} 0\,1\,0 \xrightarrow{P_1} 0\,0\,0$

Formally, we define the problem NIM as follows:

Given: $\langle s_1, \ldots, s_k \rangle$ encoded in binary.

Question: Does Player I have a winning strategy from this position?

Theorem (Bouton 1901): $NIM \in L$.

Like in the case of 2SAT, proving membership in L requires a deeper understanding of the "semantics" of the problem.

Definition:

We assume configurations are arranged in a matrix

with
- rows = $S_1, \dots, S_k$
- least significant bit first.

A configuration is balanced,
if every column contains an even number of 1s.

Example: $2\ 2\ 1 =$

0 1
0 1
1 0
↑ not balanced.

Lemma:

(1) Given an unbalanced position,
there is a move that leads to a balanced position.

(2) Given a balanced position,
every move leads to an unbalanced position.

On the example: $u \xrightarrow{P_1} b \xrightarrow{P_2} u \xrightarrow{P_1} b \xrightarrow{P_2} u \xrightarrow{P_1} b = 000$

Lemma: Player I has a winning strategy
iff the initial position is unbalanced.

Proof:

- If the initial position is unbalanced,
the player chooses a turn that makes it balanced.
In a balanced position, the opponent cannot take the last stick.
- If the initial position is balanced,
the opponent has a winning strategy,
namely to always produce a balanced position.

□

In short:
- Balanced positions are losing for the player who has the turn.
- Unbalanced positions are winning for the player who has the turn.

Checking whether the initial position is balanced can be solved in deterministic logspace (flipping a bit for each column, going through the columns).

There are further useful problems in L:

Lemma:
- $\{(x,y,z) \mid x,y,z \text{ encoded in binary and } x+y=z\} \in L$.
- $\{(x,y,z) \mid x,y,z \text{ encoded in binary and } xy=z\} \in L$.