We review Savitch's Theorem that any language decidable non-deterministically in $O(f(n))$ space is deterministically decidable in $O((f(n) + \log(n))^2)$ space. We focus on logarithmic space.

This optional section organizes — at an intuitive level — some proof tricks we have seen in recent weeks. Specifically, one trick is to cache intermediate values in order to save time at the expense of space. Conversely, we may uncache in order to save space at the expense of time. Examples of caching include dynamic programming, say for deciding CFLs in polynomial time, and repeated squaring, say for the pset problem asking us to decide whether $a^b \equiv c \pmod{p}$ in polynomial time. Examples of uncaching include the trick in Savitch’s proof and the tricks in lecture’s NL = coNL proof. The basic pattern is this:

(a) Two algorithm tricks. (b) A toy example of space-time trade-off: permuting $(a,b,c,d)$.

For example, recall how to compute $a^{2^k}$ by repeated squaring (say we do all arithmetic mod $p$):

```python
def power(2**k):
    if k==0: return a
    sqrt = pow(2**k-1)
    return sqrt * sqrt
```

The same strategy works when $A$ is an $n \times n$ matrix, for convenience notated here as a collection $\{(a, c) \mapsto A[a,c] \forall a,c\}$ of index-value pairs.

```python
def power(2**k):
    if k==0: return A
    sqrt = power(2**(k-1))
    return { (a, c) \mapsto \text{sum}( sqrt[a,b]*sqrt[b,c] \forall b) \forall a, c }
```

Let’s say we are interested in just the $(s, t)$th entry of $A^{2^k}$. We may easily change our program to:

```python
def pow_entry(2**k, s, t):
    if k==0: return A[s, t]
    sqrt = { (a, c) \mapsto pow_entry(2**(k-1), a, c) \forall a, c }
    return \text{sum}( sqrt[s,b]*sqrt[b,t] \forall b)
```

Now, suppose $A$ is huge — maybe $nN \sim 2^k$. How much space do we currently use? The above solution creates a stack $\Theta(k)$ frames deep, and each frame stores a huge matrix (the value of that stack frame’s copy of $\text{sqrt}$) of size $n \times n$. So it uses $\Theta(kn^2) = \Theta(k^4)$ space. This is bigger than we’d like. So let us uncache the $\text{sqrt}$:

```python
def pow_entry(2**k, s, t):
    if k==0: return A[s, t]
    return \text{sum}( pow_entry(2**(k-1), s, b)*pow_entry(2**(k-1), b, t) \forall b)
```

The solution still creates a stack $\Theta(k)$ frames deep. However, each frame now avoids storing a whole matrix! Instead, each frame stores a counter for $b$, which takes $\log n$ space. So the overall amount of space is $\Theta(k \log n) = \Theta(k^2)$. Much better! This trick turns out to be Savitch in disguise.
PATH is clearly in NL, since we may store a single pointer that begins by pointing to the source node and keeps non-deterministically traversing edges until it arrives at the target node. By Savitch, PATH is hence decidable with $O((\log n)^2)$ space. To see this concretely, we consider the following algorithm. It uses $\Theta((\log n)^2)$ space per stack frame to store a node, and it uses $\Theta(\log(n))$ many stack frames, for a total of $\Theta((\log n)^2)$ space:

![Algorithm](image.png)

If $A \leq L B$ and $B \in (N)L$, then so is $A$.

(a) How to use a log space reduction $A \leq L B$ to translate a compact solution to $B$ into a compact solution to $A$. We un-cache each bit of $b$, which saves space and wastes time.

(b) Recall how to use a p-time reduction $A \leq p B$ to translate a fast solution to $B$ into a fast solution to $A$.

**NL vs coNL**

From lecture, we know that NL = coNL. Still, when we want to show a language (say, BIP, the language of bipartite graphs) is in NL = coNL, it is easiest to show directly that BIP is in coNL before invoking that NL = coNL to finish the proof. Some languages are easier to directly place in NL, while other languages are easier to directly place in coNL. As with (co)CFLs, (co)recognizable languages, and (co)NP languages, there is a simple heuristic for which to try: languages characterized by $\exists$s are easier to show are in NL, while languages characterized by $\forall$s are easier to show are in coNL.

<table>
<thead>
<tr>
<th>NL</th>
<th>coNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATH = ${(G, s, t) : \exists \text{path s} \rightsquigarrow t}$</td>
<td>PATH = ${(G, s, t) : \forall \text{path from s: endpoint is not t}}$</td>
</tr>
<tr>
<td>SC = ${G : \exists \text{cycle through all nodes}}$</td>
<td>SC = ${G : \exists \text{DFA that accepts all w}}$</td>
</tr>
<tr>
<td>ALL$_{DFA}$</td>
<td>$\forall \text{cycles: cycle has even length}$</td>
</tr>
<tr>
<td>BIP = ${G : \forall \text{cycles: cycle has even length}}$</td>
<td>ALL$_{DFA}$ = ${M : \forall w : M \text{ accepts w}}$</td>
</tr>
</tbody>
</table>

The proof of NL = coNL contributes to this table, too. Instead of writing PATH = $\{(G, s, t) : \exists \text{correct BFS computation history that does not reach t from s}\}$, that proof writes

PATH = $\{(G, s, t) : \exists \text{correct BFS computation history that does not reach t from s}\}$

*If we replace or by $*$ and $*$ by and, we obtain our previous matrix exponentiation program! The correspondence is not a coincidence, but to explain why would take us too far afield.
2-SAT is NL complete
Each 2-SAT clause \((a \lor b)\) is equivalent to the implications \(\neg a \implies b\) and \(\neg b \implies a\). So we represent a 2-SAT instance as a directed graph with two nodes \(a, \neg a\) for each variable and two edges \(\neg a \rightarrow b, \neg b \rightarrow a\) for each clause \((a \lor b)\). If for each \(a\) there is no path between \(a\) and \(\neg a\), then the 2-SAT instance is satisfiable. Otherwise, it is unsatisfiable. So 2-SAT is in coNL.

In fact, \(\text{PATH} \leq_L \text{2-SAT}\). We translate a graph \(G\) to a formula by introducing a variable for each node and a clause \((\neg a \lor b)\) for each edge \(a \rightarrow b\). To check whether from node \(s\) we may reach node \(t\), we further insert the clauses \((s \lor s)\) and \((\neg t \lor \neg t)\). If \(G\) admits a path from \(s\) to \(t\), then the edge-clauses of the 2-SAT formula suffice to prove that \(s\) implies \(t\). But this contradicts the two special clauses, so the 2-SAT formula is unsatisfiable. Conversely, if \(G\) admits no path from \(s\) to \(t\), then it is well-defined to set all nodes reachable from \(s\) to True and all nodes from which \(t\) is reachable to False. This assignment evidently satisfies the 2-SAT formula, including its special clauses. So the 2-SAT formula is satisfiable. So 2-SAT is NL complete.

SC is NL complete
STRONGLY-CONNECTED is the language of directed graphs such that every node is reachable from every other node. We call such graphs strongly connected. SC is in NL because we may use two pointers to iterate over every source-target pair of nodes, and for each pair, solve the PATH problem.

In fact, SC is NL complete. We may show this by reducing \(\text{PATH} \leq_L \text{SC}\). Well, given an instance \((G, s, t)\) of \(\text{PATH}\), we modify \(G\) by introducing arrows \(t \rightarrow v\) and \(v \rightarrow s\) for each \(V\) in the graph. Actually, since we need to do this reduction with only logarithmically much space, we refrain from manipulating \(G\) explicitly; we instead represent the modified graph \(G'\) implicitly by defining how to answer each query to “is there an edge from \(a\) to \(b\) in \(G'\)?”. The answer is “yes” if the the old graph \(G\) contains that edge or if \(a = t\) or if \(b = s\).

Then \(G'\) is strongly connected if \(G\) has a path from \(s\) to \(t\). For, to travel from \(u\) to \(v\) within \(G'\), we simply take the edge from \(u\) to \(s\), take the path from \(s\) to \(t\), then take the edge from \(t\) to \(v\).

Moreover, \(G'\) is strongly connected only if \(G\) has a path from \(s\) to \(t\), since the latter condition is a special case of the strong connectedness condition. We conclude that the reduction is correct.

This optional section builds intuition. We saw in discussion that L (and NL) involve languages decidable using finitely many pointers and counters. One way to package this intuition is that NL — for which PATH is a complete language — is the class of problems no harder than mazes. In analogy, P contains problems no harder than calculations in arithmetic or boolean logic, and NP contains problems no harder than puzzles, wherein one is given a simple criterion for success instead of a search procedure. Finally, as we saw just this week, PSPACE contains problems as hard as games, i.e. puzzles with adversaries, in which the criterion for success is itself quantified over adversary strategies.

Next week, we will show that mazes are strictly easier than games (i.e. there is a language in PSPACE that is not in NL), but, embarrassingly, we don’t know any other relations between mazes, calculations, puzzles, and games. More precisely, we may map out our knowledge of inclusions below. To aid comparison of space and time, we write P as PTIME and so forth:
(a) The above 13 language classes are organized by containment. If a language is shown strictly to the left of another language, then the former is contained in the latter. For example, this diagram says that LSPACE $\subseteq$ NLSPACE, but it does not claim that Recognizable $\subseteq$ coRecognizable. Some equalities and inequalities are also shown using the usual symbols $=$, $\neq$. For example, we learned that NLSPACE = coNLSPACE. Dashed lines indicate relationships wherein inequality is conjectured but so far neither proven nor disproven. For example, it is unknown whether PTIME = NPTIME.