Where Are We?

We have a somewhat elaborate zoo of low complexity classes.

For the flagship $P$ and $NP$, we have lots of examples of natural problems that live in these classes.

So how about classes below $P$ or above $NP$? As always, we are looking for natural examples, not artificial languages obtained by exploiting universality.

Current Zoo

- $SPACE(1)$, constant space,
- $L = SPACE(log)$, logarithmic space,
- $NL = NSPACE(log)$, nondeterministic logarithmic space,
- $P = TIME(poly)$, polynomial time,
- $NP = NTIME(poly)$, nondeterministic polynomial time,
- $SPACE(n)$, linear space,
- $NSPACE(n)$, nondeterministic linear space,
- $PSPACE = SPACE(poly)$, polynomial space.
- All the old stuff: regular, CFL, CSL, primitive recursive, decidable, semidecidable.

Immerman’s Picture

Basics

Lemma

$L \subseteq NL \not\subseteq P$, but no separation known.

Proof. Suppose $M$ is a nondeterministic TM that runs in space $\log^4 n$ and $x$ some input of length $n$.

Build the computation graph of $M$ on $x$, $\Gamma(M, x)$, as follows.

Vertices are all configurations of $M$ of size $2^{-\log^4 n}$.

There is an edge $C \rightarrow C'$ if $C \vdash_{M} C'$.

$\Gamma(M, x)$ has size polynomial in $n$ and $M$ accepts $x$ if there is a path from $C_x$ to $C_H$.  

$\square$
So this is really our old friend

**Problem: Graph Reachability**

*Instance:* A digraph $G$, two nodes $s$ and $t$.

*Question:* Is there a path from $s$ to $t$ in $G$?

Clearly, Reachability is in deterministic linear time and space, or, alternatively, in nondeterministic logarithmic space (in NL).

To produce a NL-complete problem, one needs to be a bit careful to choose the right kind of reduction, something more fine-grained than just polynomial time.

As it turns out, logarithmic space is a good choice: it entails polynomial time, but there is no reason why a polynomial time computation should require only logarithmic space in general (just think about depth-first-search).

As always, the underlying machine model separates input/output tapes from the work tape.

Language $A$ is **log-space reducible** to $B$ if there is a function $f : \Sigma^* \rightarrow \Sigma^*$ that is computable by a log-space Turing machine such that

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

We will write $A \leq_{\text{log}} B$.

As it turns out, many of the polynomial time reductions showing $\text{NP}$-hardness are in fact log-space reductions. For example, SAT is $\text{NP}$-complete with respect to log-space reductions.

This can be seen by careful inspection of the proof for Cook-Levin: in order to construct the large Boolean formula that codes the computation of the nondeterministic Turing machine, one never needs more than logarithmic memory.

The following lemma shows that space complexity classes are closed under log-space reductions (under some mild technical conditions).

**Lemma**

- $A \leq_{\text{log}} B$ implies $A \leq_{\text{p}} B$.
- $\leq_{\text{log}}$ is a pre-order (reflexive and transitive).

**Proof.** To see part (1) note that the reduction can be computed in time $O(2^{\text{log } n}) = O(n^c)$.

Transitivity in part (2) is not trivial: one cannot simply combine two log-space transducers by identifying the output tape of the first machine with the input tape of the second machine: this space would count towards the work space and may be too large.

Therefore we do not compute the whole output, rather we keep a pointer to the current position of the input head of the second machine.

Whenever the head moves compute the corresponding symbol from scratch using the first machine. The pointer needs only $\log(n') = O(\log n)$ bits. $\square$
Therefore we use the same trick as in the transitivity lemma.

Let $M$ be an acceptor for $B$ in $s(n)$ space. Here is an algorithm to test whether $x \in A$.

Compute $y = f(x)$ symbol by symbol and input the symbols directly to $M$ to test whether $y \in A$.

If $M$ requires the same symbol twice, restart the whole computation. The first part takes space $\log |x|$, regardless of the number of recomputations.

Now $|y| \leq \ell(|x|)$, hence $M$ will use no more than $s(\ell(|x|))$ space on its worktape.

Thus the total space requirement is $s(\ell(n)) + \log n$.

Lemma

- If $B$ is in $L$ and $A \leq_{\log} B$ then $A$ is also in $L$.
- If $B$ is in $NL$ and $A \leq_{\log} B$ then $A$ is also in $NL$.

Proof.

Note that we cannot simply compute $y = f(x)$ in log-space and then use the log-space algorithm for $B$: $|y|$ is bounded only by $|x|^k + c$, so we cannot store $y$. Instead we recompute every symbol $y_i$ on demand, just like above. We are essentially trading time for space.

The nondeterministic case is entirely similar.

Definition

$B$ is NL-hard if for all $A$ in NL: $A \leq_{\log} B$.

$B$ is NL-complete if $B$ is in NL and is also NL-hard.

As before with NP and PSPACE, the key is to produce natural examples of NL-complete problems.

Ideally these problems should have been studied before anyone even thought about NL.

Recall that we have already seen the somewhat surprising result by Immerman and Szelepsényi that Reachability also lies in co-NL (though, at first glance, nondeterminism seems to be utterly useless when it comes to checking that there is no path from $s$ to $t$).

In terms of space complexity classes the result shows

Theorem (Immerman-Szelepsényi 1987/88)

For any (space-constructible) function $f(n) \geq \log n$, $\text{NSPACE}(f(n)) = \text{co-NSPACE}(f(n))$.

In particular at the low end we have NL = co-NL.

Note that the analogous statement for time, $\text{NP} = \text{co-NP}$, is generally believed to be false.
We have mentioned in the past that finite state machines are, in a sense, the most trivial algorithms, at least if one requires an algorithm to have infinitely many inputs and to read every bit in the input.

True, but there clearly are interesting computations that can be carried out with just a fixed, finite number of bits. E.g., multiplying two 64-bit numbers (given in some format or other) is hugely important.

And it can go wrong, ask INTEL.

Circuits and Languages

A single circuit represents a Boolean function $f : 2^n \rightarrow 2$ (or several such functions). We don’t allow unbounded fan-in in our circuits, so they are fairly close to physical reality.

Since there $2^{2^n}$ Boolean functions, some of the circuits will have to be large for simple counting reasons. Shannon already knew about a typical size of $\Theta(2^n/n)$

for a circuit implementing a Boolean function with $n$ inputs.

Of course, interesting functions like parity, majority, counting might well have much smaller circuits.

Languages

In order to use circuits to recognize a language $L$ over $\Sigma$ we need one circuit $C_n$ with $n$ inputs for each $n \in N$:

$$\forall n \exists C_n (C_n \text{ recognizes } L \cap \Sigma^*)$$

We can then define the language of this family of circuits as the set of words $x \in \Sigma^*$ such that $C_{|x|}$ on input $x$ evaluates to true.

Alas, this causes another problem: we really want the existential quantifier to be constructive, the circuit $C_n$ should be easily constructible from $n$. This leads to the distinction between uniform versus non-uniform families of circuits.

For example, in the non-uniform world, clearly every unary language is acceptable, even a highly undecidable one. And each of the circuits individually is trivial. Ouch.

Basics

Unconstrained circuits are not too interesting, but if we add a few bounds we obtain interesting classes.

Definition (NC)

A language $L$ is in $\text{NC}^d$ if there is a circuit family $(C_n)$ that decides $L$ and such that the size of $C_n$ is polynomial in $n$ and the depth of $C_n$ is $O(\log^d n)$.

$\text{NC}$ is the union of all $\text{NC}^d$.

The key idea is that, given enough processors, we can evaluate $C_n$ in $O(\log^d n)$ steps. It is reasonable to assume that a problem admits an efficient parallel algorithm iff it lies in $\text{NC}$. For example, it is easy to see that parity testing is in $\text{NC}^1$.

Lemma

$\text{NC}^1 \subseteq L \subseteq \text{NL} \subseteq \text{NC}^2$

We only indicate the first inclusion.

Given a log-depth circuit $C$ and input $x \in \Sigma^n$, we can evaluate the circuit by recursion, essentially performing DFS from the root.

The recursion stack has depth $\log n$, and each stack frame contains only a constant number of bits.
**Branching Programs**

25

Suppose we have a collection of functions \([w] \to [w]\) for some width \(w\) and Boolean variables \(x_1, \ldots, x_n\).

We can construct bounded width branching programs (BPs) by fixing a sequence of instructions of the form

\[(j_i, f_i, g_i)\]

where \(1 \leq i \leq \ell\), \(1 \leq j_i \leq n\) and \(f_i, g_i : [w] \to [w]\). Here \(\ell\) is the length of the program.

Given a BP \(P\) and Boolean values for the variables \(x_j\), we define the semantics of \(P(x)\) to be the function \([w] \to [w]\) obtained by composing the \(\ell\) functions

\[\text{if } x_{j_i} \text{ then } f_i \text{ else } g_i\]

**A Characterization**

29

Theorem (Barrington 1989)

The class of languages recognized by BWBP is exactly NC¹.

Converting a BWBP into an NC¹ circuit is fairly straightforward.

The opposite direction requires some group theory and linear algebra over finite fields, we’ll skip.

**Qui?**

26

Write \(I\) for the identity function on \([2]\), and \(\sigma\) for the transposition.

\[P : \langle x, \sigma, I \rangle, \langle y, I, \sigma \rangle, \langle z, \sigma, I \rangle\]

Then \(P(x, y, z) = \sigma\) iff \(x \oplus y \oplus z = 1\).

Given a family of function \(F \subseteq [w] \to [w]\) we can say that \(P\) recognizes \(w \in 2^n\)

if \(P(a) \in F\).

Given a program \(P_n\) with \(n\) variables for each \(n\) allows us to recognize a whole language over \(2\) (family of BPs).

**Déjà Vu All Over**

27

This may sound rather bizarre, but think about a DFA \(A\) over the alphabet \(2\).

The automaton consists essentially of two functions \(\delta_s : [w] \to [w]\) where \(w\) is the state complexity of \(A\), \(s \in 2\).

Each binary word determines a composition \(f\) of these functions, and acceptance depends on whether \(f(q_0) \in F\).

So any DFA is an example of a BWBP; in fact, we get a whole family of programs, one for each input length.

This is an example of a uniform family: there is one description that works for all \(n\). We could easily build a Turing machine that constructs \(C_n\) from \(0^n\).

**Bounds**

28

Families of BPs are only really interesting if we impose a few bounds:

- the width of all the programs should be bounded, and
- the length should be polynomial in \(n\).

These are called bounded width poly length branching programs (BWBPs).

Why?

It is known that every language \(L \subseteq 2^n\) can be recognized by a family of BPs of width 4 but exponential length; alternatively, we can achieve linear length at the cost of exponential width.

**Structuring Polynomial Time**

30

Here is another aspect of log-space reductions: they can also be used to study the fine-structure of \(P\).

Definition (\(P\)-Completeness)

A language \(B\) is \(P\)-hard if for every language \(A\) in \(P\): \(A \leq_{\log} B\).

It is \(P\)-complete if it is \(P\)-hard and in \(P\).

Theorem

Suppose \(B\) is \(P\)-complete. Then

- \(B \in \text{NC}\) iff \(\text{NC} = P\).
- \(B \in L\) iff \(L = P\).
Circuit Value Problem

<table>
<thead>
<tr>
<th>Problem: Circuit Value Problem (CVP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: A Boolean circuit ( C ), input values ( x ).</td>
</tr>
<tr>
<td>Question: Check if ( C ) evaluates to true with input ( x ).</td>
</tr>
</tbody>
</table>

Obviously \( \text{CVP} \) is solvable in polynomial time (even linear time).

There are several versions of \( \text{CVP} \), here is a particularly simple one: compute the value of \( X_m \) where

\[
X_0 = 0, \quad X_1 = 1 \\
X_i = X_L \cdot \overline{X_i}, \quad i = 2, \ldots, m
\]

Here \( \cdot \) = \& and \( L_i, R_i < i \).

Theorem (Ladner 1975)
The Circuit Value Problem is \( \mathsf{P} \)-complete.

Proof.
For hardness consider any language \( L \) accepted by a polynomial time Turing machine \( M \).
We can use ideas similar to Cook-Levin to encode the computation of the Turing machine \( M \) on \( x \) as a polynomial size circuit (polynomial in \( n = |x| \)); use lots of Boolean variables to express tape contents, head position and state.
Constructing this circuit only requires “local” memory; for example we need \( O(\log n) \) bits to store a position on the tape.
The circuit evaluates to true iff the machine accepts.

Proof Cont’d
Note that \( x \in L \) iff reach\( (C_x, 0, O(f(n))) \), so to test membership in \( L \) we only need to compute reach.

Claim
reach\( (C_1, 0) \) \iff \( C_1 = C_2 \lor C_1 \downarrow M C_2 \).
reach\( (C_1, k + 1) \) \iff \exists C (\text{reach}(C_1, C, k) \land \text{reach}(C, C_2, k)) \).

The claim is obvious from the definition, but note that it provides a recursive definition of reach.
Also observe that the recursion involves a search for the intermediate configuration \( C \).
Implementing a recursion requires memory, typically a recursion stack: we have to keep track of pending calls and the required information.

In this case the recursion stack will have depth $O(f(n))$: the length of a computation is bounded by $2^{O(f(n))}$.

Each stack frame requires space $O(f(n))$ for the configurations.

The search for $C$ can also be handled in $O(f(n))$ space.

Thus the total space complexity of the algorithm is $O(f^2(n))$: stack size times entry size.

\[ \square \]

Corollary

$L \subseteq NL \subseteq NC \subseteq P \subseteq NP \subseteq \text{PSPACE} = \text{NPSPACE}.$

Actual feasible computation in the RealWorld$^\text{TM}$ lives somewhere around NC, heading towards P.

Amazing progress has been towards tackling some NP-complete problems like SAT, but in general the light go out here.

\text{PSPACE} is really way too big.

---

Some PSPACE-Problems

We already mentioned quantified Boolean formulae: one can check if such a formula is valid in polynomial space.

Given a context sensitive grammar $G$ and a word $x \in \Sigma^*$, we can check in nondeterministic linear space if $G$ accepts $x$.

Given a nondeterministic finite state machine $M$, we can check we can check in nondeterministic linear space whether $M$ is non-universal (i.e., $L(M) \neq \Sigma^*$).

Various (generalized) board games such as Go and Hex are in PSPACE: one can check in polynomial space whether a given position is winning.

As before with NP, it is not too hard to come up with a totally artificial PSPACE-complete problem:

\[ K = \{ \epsilon \# x \# 1^t | x \text{ accepted by } M_t \in \text{time } |x|^t + t \text{ space} \} \]

Because of the padding, we can easily check membership in $K$ in linear space.

And the standard reduction shows that the problem is hard.

Not too interesting, here is a much better problem.

---

Quantified Boolean Formulae

Theorem

Validity testing for quantified Boolean formulae is PSPACE-complete.

Proof.

Membership in PSPACE is easy to see: we can check validity by brute-force using just linear space.

To see this, note that every block of existential and universal quantifiers can be considered as a binary counter: for $k$ Boolean variables we have to check values $0, \ldots, 2^k - 1$.

This is exponential time but linear space.

---

Quantified Boolean Formulae

Proof Cont’d

For hardness one uses the fact that a configuration $C^*$ of a Turing machine can be expressed by a (large) collection of Boolean variables much as in Cook-Levin, plus ideas from Savitch’s theorem.

We will construct a quantified Boolean formula $\Phi_k$ of polynomial size such that $\Phi_k(C_1, C_2)$ iff $C_1^* \in L \subseteq C_2^*$.

This is straightforward for $k = 0, 1$: copy the appropriate parts of the Cook-Levin argument.

For $k > 1$ note that $\Phi_k(C_1, C_2)$ iff $\exists C (\Phi_{k-1}(C_1, C) \land \Phi_{k-1}(C, C_2))$.

Unfortunately, this direct approach causes an exponential blow-up in size.

Therefore we use a trick:

\[ \Phi_k(C_1, C_2) \iff \exists C \forall D_1, D_2 ((C_1, C) = (D_1, D_2) \lor (C, C_2) = (D_1, D_2) \Rightarrow \Phi_{k-1}(D_1, D_2)) \]
**Quoi?**

So $\Phi_k(C_1, C_2)$ means that for some witness $C$ and all choices of $D_1$ and $D_2$ we have

$$(C_1, C) = (D_1, D_2) \lor (C, C_2) = (D_1, D_2) \Rightarrow \Phi_{k-1}(D_1, D_2)$$

This just gets us around having to write down $\Phi_{k-1}$ twice. \hfill $\square$

Depending on your temperament, you will recognize this as a brilliant technical maneuver, or dismiss it as a slimy trick.

---

**Context Sensitive Recognition**

We have seen that for context-sensitive grammars it is decidable whether a word is generated by the grammar.

**Problem:** Context Sensitive Recognition (CSR)

**Instance:** A CSG $G = (V, \Sigma, P, S)$ and a word $x \in \Sigma^*$.

**Question:** Is $x$ in $L(G)$?

In fact, we can solve this problem in $\text{NSPACE}(n)$ by guessing where in the current string to apply a production of $G$ backwards.

Ultimately we will wind up in $S$ iff the given string is in the language.

---

**CSR Completeness**

**Theorem**

Context Sensitive Recognition is PSPACE-complete.

**Proof.**

For hardness use the following well-known fact from language theory:

$\varepsilon \notin L \in \text{NSPACE}(n)$ implies $L$ is context-sensitive (via monotonic grammars).

Now let $L \subseteq \Sigma^*$ be in PSPACE, say, $L \in \text{SPACE}(p(n))$. Define

$L' = \{ x \# 1^{p(|x|)} | x \in L \}$

By the previous remark $L'$ is context-sensitive. In fact, a grammar $G$ for $L'$ can be constructed in polynomial time from a Turing machine for $L$.

Hence the map $f$ defined by $f(x) = (G, x \# 1^{p(|x|)})$ is a polynomial time reduction from $L$ to CSR. \hfill $\square$

---

**SAT to QBF**

The jump from SAT to QBF, corresponding to $\text{NP}$ versus PSPACE, may seem a bit abrupt.

Recall the standard classification of Boolean formulae in prenex normal form according to leading quantifiers:

- $\Sigma_k$: $k$ alternating blocks of quantifiers, starting with existential
- $\Pi_k$: $k$ alternating blocks of quantifiers, starting with universal

So e.g. a $\Pi_2$ formula looks like

$$\forall x_1 \forall x_2 \ldots \forall x_n \exists y_1 \exists y_2 \ldots \exists y_m \varphi(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

where $\varphi$ is quantifier-free.

---

**A Hierarchy**

It is clear that SAT corresponds to validity of $\Sigma_1$ formulae.

Likewise, TAUT corresponds to validity of $\Pi_1$ formulae.

But we would expect validity of $\Sigma_2$ formulae to be strictly more complicated. Ditto for $\Pi_2$.

And, of course, $\Sigma_3$ and $\Pi_3$ should be more complicated again.

Everything together is QBF.

---

**Back To Oracles**

We have seen that polynomial time Turing reducibility is a useful concept to unify different versions of a problem. In the standard setting, the Turing machine is deterministic (a real algorithm with an extra data base).

We can generalize this quite naturally by allowing the Turing machine itself to be nondeterministic. So suppose $C$ is some complexity class and set

$$\text{P}^C = \{ L(M) | M \text{ poly. time det. OTM with oracle in } C \}$$

$$\text{NP}^C = \{ L(M) | L \text{ poly. time non-det. OTM with oracle in } C \}$$

As usual, we do not charge for the activity of the oracle.
In the world of classical computability, oracles are benign in the sense that all the basic results just carry over. There is a kind of meta-theorem:

> Take any proof in classical computability, and add oracles everywhere. Then the proof is still valid.

Making this technically precise is probably difficult, but everyone working in the fields knows exactly what this means.

### Theorem (Baker, Gill, Solovay 1975)

There is an oracle \( A \) for which \( \mathbb{P}^A = \mathbb{NP}^A \).

There is an oracle \( B \) for which \( \mathbb{P}^B \neq \mathbb{NP}^B \).

Sadly, this means that standard techniques from computability theory are not going to help to resolve the \( \mathbb{P} = \mathbb{NP} \) question: all these arguments are invariant under oracles.

New tools are needed, and to-date no one knows what they are.

### Polynomial Hierarchy

Now define

\[
\Sigma^p_0 = \Pi^p_0 = \mathbb{P} \\
\Delta^p_{n+1} = \mathbb{P}^{\Sigma^p_n} \\
\Sigma^p_{n+1} = \mathbb{NP}^{\Sigma^p_n} \\
\Pi^p_n = \text{co-}\Sigma^p_n \\
\text{PH} = \bigcup \Sigma^p_n
\]

Then \( \Sigma^p_1 = \mathbb{NP} \) and \( \Pi^p_1 = \text{co-NP} \).

### In Pictures

Actually, the last picture shows the arithmetical hierarchy, the counterpart in classical computability theory to \( \text{PH} \). It's exactly the same structure, just replace \( \mathbb{P} \) by decidable, and \( \mathbb{NP} \) by semidecidable.

We have

\[
\Sigma^p_n \cup \Pi^p_n \subseteq \Delta^p_{n+1} 
\subseteq \Sigma^p_n \cap \Pi^p_{n+1}
\]

and \( \text{PH} \subseteq \text{PSPACE} \).

In the classical computability setting we have separation results galore. Sadly, in the complexity realm we don’t know whether \( \mathbb{P} \neq \text{PH} \).