More Complexity

Klaus Sutner

Carnegie Mellon University

25-space-complex   2017/12/15 23:17

1. Below $\mathbb{NP}$

- Parallelism

- Above $\mathbb{NP}$

Where Are We?

We have a somewhat elaborate zoo of low complexity classes.

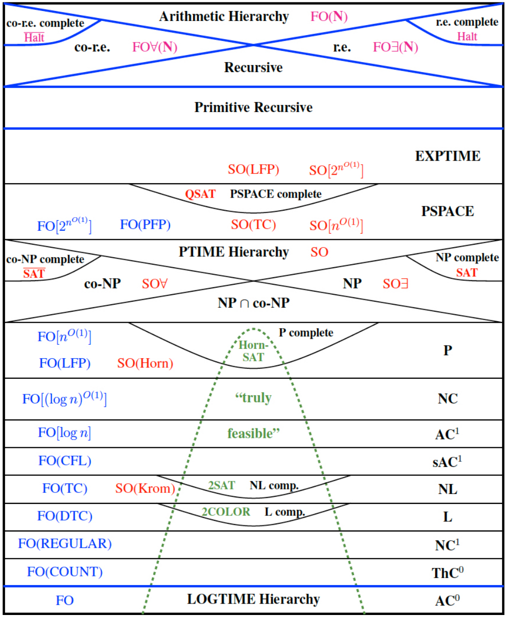
For the flagship $\mathbb{P}$ and $\mathbb{NP}$, we have lots of examples of natural problems that live in these classes.

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

Current Zoo

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

Immerman's Picture



Lemma

$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathbb{P}$, *but no separation known.*

Proof. Suppose M is a nondeterministic TM that runs in space $\log^k n$ 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 $2^{c \log^k n}$.

There is an edge $C \to C'$ iff $C \vdash_M^1 C'$.

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

$\square$

Same Old

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).

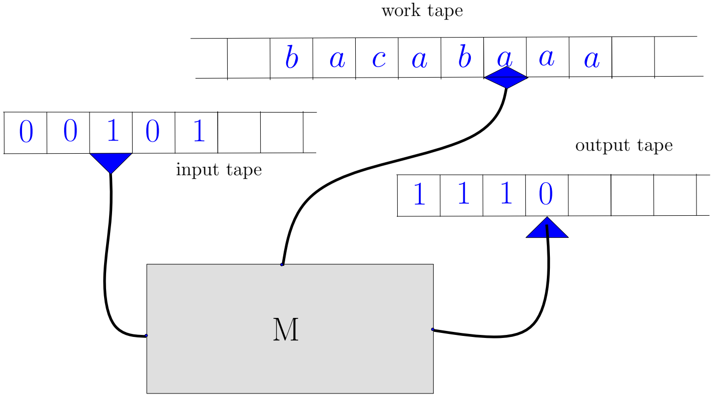
Appropriate Reductions

To produce a NL-complete problem, one needs to a 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.

Input/Output Tapes



Log-Space Reductions

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

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

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

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

Key Properties

Lemma

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

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

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 $\log(n^c) = O(\log n)$ bits.

□

Closure under Log-Space Reductions

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

Lemma

Let $A \leq_{\log} B$ *via a log-space reduction* $f : \Sigma^\star \to \Sigma^\star$ *. Suppose we have a polynomial bound* $|f(x)| \leq \ell(|x|)$ *for all* $x \in \Sigma^\star$*. Then* $B \in \mathrm{SPACE}(s(n))$ *implies* $A \in \mathrm{SPACE}(s(\ell(n)) + \log n)$*.*

The same holds for non-deterministic space.

Proof.

Note that one cannot use the same argument as for $\leq_m^p$ and time classes: ℓ may not be logarithmic (we only have $\ell(n) = O(n^c)$).

Same Old

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 $s(\ell(n)) + \log n$.

□

Closure Properties

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.

□

Hardness and Completeness

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 $\mathbb{NP}$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.

Reachability

Theorem

Graph Reachability NL*-complete with respect to log-space reductions.*

Proof.

As above, construct the computation graph $\Gamma(M, x)$ and note that the construction can be carried out in logarithmic space.

□

Immerman-Szelepsényi

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 $f(n) \geq \log n$,
$\mathrm{NSPACE}(f(n)) =$ *co*-$\mathrm{NSPACE}(f(n))$.

In particular at the low end we have $\mathrm{NL} =$ co-NL.

Note that the analogous statement for time, $\mathbb{NP} =$ co-$\mathbb{NP}$, is generally believed to be false.

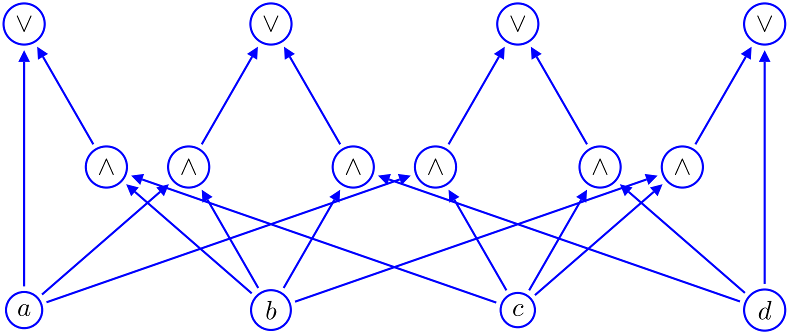
Rock Bottom

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.

Boolean Circuits



Boolean circuits provide a natural framework to certain computational problems like addition and multiplication (of integers or matrices).

Note that the evaluation of a circuit is inherently parallel, one can propagate values upward in parallel.

Circuits and Languages

A single circuit represents a Boolean function $f : \mathbf{2}^n \to \mathbf{2}$ (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 $\mathbf{2}$ we need one circuit C_n with n inputs for each $n \in \mathbb{N}$:

$$\forall\, n\, \exists\, C_n\, (C_n \text{ recognizes } L \cap 2^n)$$

We can then define the language of this family of circuits as the set of words $x \in \mathbf{2}^\star$ 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.

Nick's Class

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

Definition (NC)

A language L is in 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 $O(\log^d n)$.

NC is the union of all 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 in lies in NC.

For example, it is easy to see that parity testing is in NC^1.

Lemma

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

We only indicate the first inclusion.

Given a log-depth circuit C and input $\boldsymbol{x} \in \mathbf{2}^n$, 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

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

$$\langle\, j_i, f_i, g_i \,\rangle$$

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

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

if x_{j_i} **then** f_i **else** g_i

Quoi?

Write I for the identity function on $[2]$, and σ for the transposition.

$$P\colon \quad \langle\, x, \sigma, I \,\rangle\,, \langle\, y, I, \sigma \,\rangle\,, \langle\, z, \sigma, I \,\rangle$$

Then $P(x, y, z) = \sigma$ iff $x \oplus \overline{y} \oplus z = 1$.

Given a family of function $\mathcal{F} \subseteq [w] \to [w]$ we can say that P recognizes $\boldsymbol{x} \in \mathbf{2}^n$ if $P(\boldsymbol{x}) \in \mathcal{F}$.

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

Déjà Vu All Over

This may sound rather bizarre, but think about a DFA $\mathcal{A}$ over the alphabet $\mathbf{2}$.

The automaton consists essentially of two functions $\delta_s : [w] \to [w]$ where w is the state complexity of $\mathcal{A}$, $s \in \mathbf{2}$.

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

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 $L \subseteq \mathbf{2}^\star$ 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.

A Characterization

Theorem (Barrington 1989)

The class of languages recognized by BWBP is exactly NC^1.

Converting a BWBP into an NC^1 circuit is fairly straightforward.

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

Structuring Polynomial Time

Here is another aspect of log-space reductions: they can also be used to study the fine-structure of $\mathbb{P}$.

Definition ($\mathbb{P}$-Completeness)

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

It is $\mathbb{P}$-complete if it is $\mathbb{P}$-hard and in $\mathbb{P}$.

Theorem

Suppose B is $\mathbb{P}$-complete. Then

- $B \in \mathrm{NC}$ *iff* $\mathrm{NC} = \mathbb{P}$.
- $B \in \mathrm{L}$ *iff* $\mathrm{L} = \mathbb{P}$.

Circuit Value Problem

Problem:	**Circuit Value Problem (CVP)**
Instance:	A Boolean circuit C, input values x.
Question:	Check if C evaluates to true with input x.

Obviously CVP is solvable in polynomial time (even linear time).

There are several versions of CVP, here is a particularly simple one: compute the value of X_m where

$$X_0 = 0, \quad X_1 = 1$$

$$X_i = X_{L_i} \diamond_i X_{R_i}, \quad i = 2, \ldots, m$$

Here $\diamond_i = \wedge, \vee$ and $L_i, R_i < i$.

CVP is Hard

Theorem (Ladner 1975)

The Circuit Value Problem is $\mathbb{P}$-complete.

Proof.

For hardness consider any language A 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.

$\square$

Alternating Breadth First Search

A sizable list of $\mathbb{P}$-complete problems in known, though they tend to be a bit strange compare to $\mathbb{NP}$-complete problems.

For example, consider an undirected graph $G = \langle V, E \rangle$ where E is partitioned as $E = E_0 \cup E_1$.

Given a start vertex s and a target t, does t get visited along an edge in E_0 if the breadth-first search can only use edges in E_0 at even and E_1 at odd levels?

Theorem

ABFS is $\mathbb{P}$-complete.

- Below $\mathbb{NP}$

- Parallelism

3. Above $\mathbb{NP}$

Savitch's Theorem

Theorem (Savitch 1970)

Let M be a non-deterministic Turing machine that accepts language L with space complexity $f(n) \geq n$. Then there exists a deterministic Turing machine M' that accepts L and has space complexity $O(f(n)^2)$.

Hence $\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}(f^2(n))$.

Proof.

Let L be in $\mathrm{NSPACE}(f(n))$. Suppose M is a non-deterministic Turing machine that accepts L and has space complexity $S_M(n) \leq f(n)$.

Define a reachability predicate (uniformly in n). Here C_1, C_2 are configurations of M of size at most $f(n)$.

$$\text{reach}(C_1, C_2, k) \iff \exists\, s \leq 2^k \ (C_1 \vdash_M^s C_2)$$

Proof Cont'd

Note that $x \in L$ iff reach$(C_x, C_Y, O(f(n)))$, so to test membership in L we only need to compute reach.

Claim

reach$(C_1, C_2, 0)$ *iff* $C_1 = C_2$ *or* $C_1 \vdash^1_M C_2$.

reach$(C_1, C_2, k+1)$ *iff* $\exists C$ (reach$(C_1, C, k) \wedge$ 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 the Recursion

Implementing a recursion requires memory, typically a recursion stack: we have to keep track a 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.

$\square$

The Feasibility Region

Corollary

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

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

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

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 $x \in \Sigma^\star$, 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., $\mathcal{L}(M) \neq \Sigma^\star$).

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

Boring Completeness

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

$$K = \{\, e \# x \# 1^t \mid x \text{ accepted by } M_e \text{ in } t = |x|^e + e \text{ space} \,\}$$

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.

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 Φ_k of polynomial size such that $\Phi_k(C_1, C_2)$ iff $C'_1 \vdash_M^{\leq 2^k} C'_2$.

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 $\exists\, C\, (\Phi_{k-1}(C_1, C) \wedge \Phi_{k-1}(C, C_2))$.

Unfortunately, this direct approach causes an exponential blow-up in size. Therefore we use a trick:

$$\begin{aligned}\Phi_k(C_1, C_2) \iff \exists\, C\, \forall\, D_1, D_2\, ((C_1, C) = (D_1, D_2) \vee \\ (C, C_2) = (D_1, D_2) \Rightarrow \Phi_{k-1}(D_1, D_2))\end{aligned}$$

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

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

This just gets us around having to write down Φ_{k-1} twice.

$\square$

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 $G = \langle V, \Sigma, P, S \rangle$ and a word $x \in \Sigma^\star$.
Question:	Is x in $\mathcal{L}(G)$?

In fact, we can solve this problem in $\mathrm{NSPACE}(n)$ 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:
$\varepsilon \notin L \in \mathrm{NSPACE}(n)$ implies L is context-sensitive (via monotonic grammars).

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

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

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.

□

SAT to QBF

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

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

- Σ_k: k alternating blocks of quantifiers, starting with existential
- Π_k: k alternating blocks of quantifiers, starting with universal

So e.g. a Π_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 φ is quantifier-free.

A Hierarchy

It is clear that SAT corresponds to validity of Σ_1 formulae.

Likewise, TAUT corresponds to validity of Π_1 formulae.

But we would expect validity of Σ_2 formulae to be strictly more complicated. Ditto for Π_2.

And, of course, Σ_3 and Π_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 $\mathcal{C}$ is some complexity class and set

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

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

As usual, the we do not charge for the activity of the oracle.

Oracles Are Harmless

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.

Oracles Are Monsters

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.

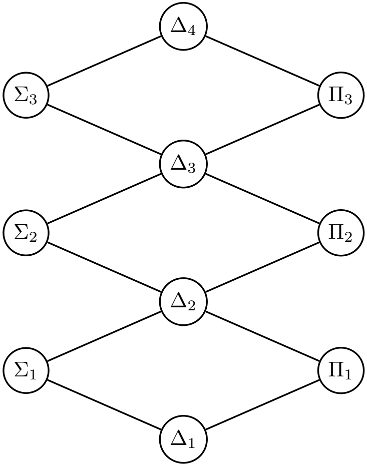
Polynomial Hierarchy

Now define

$$\begin{aligned}
\Sigma_0^p = \Pi_0^p &= \mathbb{P} \\
\Delta_{n+1}^p &= \mathbb{P}^{\Sigma_n^p} \\
\Sigma_{n+1}^p &= \mathbb{NP}^{\Sigma_n^p} \\
\Pi_n^p &= \text{co-}\Sigma_n^p \\
\mathrm{PH} &= \bigcup \Sigma_n^p
\end{aligned}$$

Then $\Sigma_1^p = \mathbb{NP}$ and $\Pi_1^p = \text{co-}\mathbb{NP}$.

In Pictures


Δ_4
Σ_3
Π_3
Δ_3
Σ_2
Π_2
Δ_2
Σ_1
Π_1
Δ_1

Truth in Advertising

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

We have

$$\Sigma_n^p \cup \Pi_n^p \subseteq \Delta_{n+1}^p \subseteq \Sigma_{n+1}^p \cap \Pi_{n+1}^p$$

and $\mathrm{PH} \subseteq \mathrm{PSPACE}$.

In the classical computability setting we have separation results galore. Sadly, in the complexity realm we don't know whether $\mathbb{P} \neq \mathrm{PH}$.