UCT

Polynomial Hierarchy

Klaus Sutner

Carnegie Mellon University

Spring 2024

1 **The Polynomial Hierarchy**

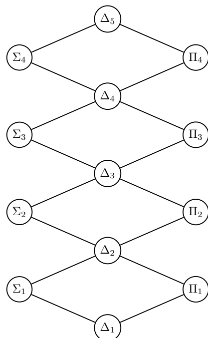
2 **Alternating Turing Machines**

We have seen a variety of time and space complexity classes; we also have a number of hierarchy theorems (sadly, most of them base on diagonalization and lacking in natural examples); lastly, there are lots of hardness/completeness results. This whole discussion is about decision problems.

The counting classes from last time indicate that corresponding counting problems can be significantly harder—just think about 2-SAT.

Here is yet another look at increased complexity, not by counting solutions, but by asking slightly more complicated questions than the ones that produce $\mathbb{NP}$ and co-$\mathbb{NP}$.

Recall that there is a natural hierarchy beyond the fundamental classes of decidable and semidecidable problems.



The hierarchy is proper, and there are fairly straightforward complete problems such as FIN, TOT, REC and so on (at least for the bottom levels). Higher up, at least we can get complete problems by discussing the truth of formulae of arithmetic.

My favorite quote from Stefan Banach:

> A mathematician is a person who can find analogies between theorems; a better mathematician is one who can see analogies between proofs and the best mathematician can notice analogies between theories. One can imagine that the ultimate mathematician is one who can see analogies between analogies.

Does this apply to our (relatively) low complexity classes? Is there any reasonable way to use a similar approch to shed light on, say, the $\mathbb{P}$ versus $\mathbb{NP}$ problem?

If we think of $\mathbb{P}$ and $\mathbb{NP}$ as an analogue to decidable versus semidecidable (the classes Δ_1 versus Σ_1 in the AH), it is tempting to try and construct a polynomial time hierarchy (PH) analogous to the AH.

We mentioned this PH briefly when we first introduced $\mathbb{NP}$, here is a more detailed discussion.

> **Warning:** Needless to say, we have no idea how to prove an analogue to Post's hierarchy theorem this time around.

Still, it may be interesting to study this putative hierarchy. For example, there might be interesting examples of problems that are complete for the first few levels of the hierarchy.

Recall: Definition PH

$$\Sigma^p_0 = \Pi^p_0 = \mathbb{P}$$

$$\Sigma^p_{n+1} = \mathsf{proj}_P(\Pi^p_n)$$

$$\Pi^p_n = \mathsf{comp}(\Sigma^p_n)$$

$$\Delta^p_n = \Sigma^p_n \cap \Pi^p_n$$

$$\mathrm{PH} = \bigcup \Sigma^p_n$$

Here proj_P refers to polynomially bounded projections: the witness is required to have length polynomial in the length of the witness. Operation comp is just plain complement (unchanged from AH).

Here is a version of the Independent Set problem that makes slightly more sense than the usual decision version.

Problem:	**Maximum Independent Set (**MIS**)**
Instance:	A ugraph G, a number k.
Question:	Does the largest independent set in G have size k?

Even more natural would be the counting or search version, but for a discussion of the polynomial hierarchy we have to stick with decision problems. Hence the annoying k parameter.

Intuitively, how difficult should we expect MIS to be?

We can easily guess a vertex set of size k and verify that it is independent.

Of course, that's nowhere near enough: we have to make sure no larger independent sets exist. In this case, it suffices to check that there are no such sets of size $k + 1$.

There are $\binom{n}{k+1}$ possible candidates, an exponential number since k is not fixed.

So, at least intuitively, MIS seems to be a there-exists-for-all problem, something in Σ_2^p rather than just $\mathbb{NP}$.

It is most natural to try to find the smallest Boolean formula equivalent to a given one. We express this search problem as a decision problem, say, for DNF formulae.

Problem:	**Minimum Equivalent DNF (**MEQDNF**)**
Instance:	A Boolean DNF formula Φ, a bound k.
Question:	Is Φ equivalent to a DNF formula with at most k literals?

Here we assume that our language has Boolean variables, constants, and the usual connectives. In particular we have constants $\top$ and $\bot$.

We could nondeterministically guess the small formula Φ', but then we need to verify that it is equivalent to Φ. In other words, we have to check that $\Phi \Leftrightarrow \Phi'$ is a tautology, which is co-$\mathbb{NP}$-complete.

Access to a MEQDNF oracle would immediately demolish SAT.

Lemma

SAT *is polynomial time Turing reducible to* MEQDNF*.*

Proof.

Let $\Psi = \bigwedge C_i$ be a CNF formula, an instance of SAT.

Define the DNF formula $\Phi = \neg\Psi = \bigvee \neg C_i$.

So Ψ is satisfiable iff Φ fails to be a tautology.

It is easy to check in polynomial time whether Φ is a contradiction. If not, Φ is a tautology iff it is equivalent to a formula with 0 literals: the constant $\top$ (constant $\bot$ is ruled out since Φ is not a contradiction).

$\square$

A pattern π is an expression formed from strings in $\mathbf{2}^\star$ and variables, using only concatenation: something like $1X1$. We can generate a language $\mathcal{L}(\pi)$ by replacing the variables by strings over $\mathbf{2}^\star$.

This is a bit like regular expressions, just think of a variable X as $\mathbf{2}$. Careful, though, it's actually quite a bit more complicated. For example

$$\mathcal{L}(XX) = \{\, ww \mid w \in \mathbf{2}^\star \,\}$$

is the copy language which fails to be context-free, never mind regular. The language is context-sensitive, though. On the other hand, we cannot produce all context-sensitive languages this way: $\{\, (00)^i(01)^i(10)^i \mid i \geq 0 \,\}$ cannot be described this way.

Given (finite) languages $P[N]$ of positive[negative] examples, we would like a pattern that is consistent with P and N.

Problem:	**Pattern Consistency**
Instance:	Two finite languages $P, N \subseteq \mathbf{2}^\star$.
Question:	Is there a pattern π such that $P \subseteq \mathcal{L}(\pi) \subseteq \overline{N}$?

Again, we could perform a simple nondeterministic guess, but it seems that the guess would have to be followed by an exponential brute-force verification: we have to find the proper substitutions for all the variables to show $P \subseteq \mathcal{L}(\pi)$ and we need to make sure that there is no substitution that produces $\overline{N} \cap N \neq \emptyset$.

Suppose $\pi = u_1 X u_2 Y u_3 X u_4 Z u_5$ and $m = |u_1 \ldots u_5|$.

Here is a problem about avoiding subgraphs by making a limited number of deletions in a given graph.

Problem:	**Node Deletion**
Instance:	Two graphs G and H, a number k.
Question:	Can one delete at most k vertices from G to obtain a graph that does not contain H as a subgraph?

We could nondeterministically guess the vertices that determine the subgraph G', and then verify that G' has no subgraph isomorphic to H. Again, the second part seems exponential.

Let's take a closer look at the class Σ_2^p in the polynomial hierarchy.

Definition

L is in Σ_2^p if there is a polynomial time decidable relation V, the verifier, and a polynomial p such that

$$x \in L \iff \exists\, u \in \mathbf{2}^{p(|x|)}\, \forall\, v \in \mathbf{2}^{p(|x|)}\; V(u, v, x).$$

So from the definition we have $\mathbb{NP}, \text{co-}\mathbb{NP} \subseteq \Sigma_2^p$.

All the examples from above are in Σ_2^p: use the existential quantifiers for guessing a witness and the universal ones for verification of the guess. The verification process can be handled by a deterministic polynomial time TM.

All of our examples of problems in Σ_2^p are complete for this class:

- Maximum Independent Set
- Minimum Equivalent DNF
- Pattern Consistency
- Node Deletion

So Σ_2^p seems to be a reasonable class that contains at least some RealWorld$^{\text{TM}}$ problems.

For Σ_2^p we have one block of existential quantifiers followed by one block of universal quantifiers. Of course, this generalizes:

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

So e.g. L is in Π_3^p iff

$$x \in L \iff \forall\, u \in \mathbf{2}^{p(|x|)}\, \exists\, v \in \mathbf{2}^{p(|x|)}\, \forall\, w \in \mathbf{2}^{p(|x|)}\, V(u, v, w, x).$$

Note that this description also lends itself nicely to finding a class in the PH that contains a particular given problem: write things down concisely, then count quantifiers. This is very similar to the arithmetical hierarchy in CRT.

It is clear from the definitions that Σ^p_k is closely connected to the problem of testing the validity of a Σ_k Boolean formula (and similarly Π_k for Π^p_k): we can express the workings of the Turing machine acceptor as a Boolean formula as in the Cook-Levin theorem.

Theorem

Validity of Σ_k Boolean formulae is Σ^p_k-complete.

Validity of Π_k Boolean formulae is Π^p_k-complete.

So just like for the AH, we can find reasonable complete problems for the PH. But in this case we don't know that they get more and more complicated.

Example: Integer Expressions

Recall the ICE problem which turns out to be PSPACE-complete.

Problem:	**Integer Circuit Evaluation (ICE)**
Instance:	A ICE circuit C, an integer a.
Question:	Is a in the output set?

We have a circuit computing sets of integers. At the leaves, there are singletons $\{x\}$ and the internal gates are

Union $A \cup B$

Sum $A \oplus B = \{\, a + b \mid a \in A, b \in B \,\}$

Product $A \otimes B = \{\, a \cdot b \mid a \in A, b \in B \,\}$

Sum Only

In a very similar scenario we have sum integer expressions, composed of numbers (written in binary), and binary operations $\cup$ and $\oplus$. There is no multiplication.

Write $\mathcal{L}(E) \subseteq \mathbb{N}$ for the finite set associated with the expression E. An interval in $\mathcal{L}(E)$ is a subset $[a, b] \subseteq \mathcal{L}(E)$. Here is a sligthly strange question about intervals being contained in the set generated by E.

Problem:	**Sum Integer Expression Intervals (SIEI)**
Instance:	An integer expression E, a number k.
Question:	Does $\mathcal{L}(E)$ have an interval of length k?

As before with ICE, the cardinality of $\mathcal{L}(E)$ is not polynomially bounded by $|E|$, so we cannot simply evaluate the expression in polynomial time.

Lemma

Sum Integer Expression Intervals is Σ_3^p-complete.

Assume that all inputs are at most ℓ-bit and the expression has depth d (think of E as a parse tree or a circuit). Then the largest number m in $\mathcal{L}(E)$ is at most $\ell + d$ bits and thus polynomial in $|E|$.

To check $a \in \mathcal{L}(E)$ for $a \leq m$, we can guess a subtree T of E. T has nodes labeled by $\mathbb{Z}$, contains the root with label a, and has the following properties. Let ν be a node in T, then:

- If ν is a union node labeled b, then T contains exactly one child node of ν, also labeled b.
- If ν is a sum node labeled b, then T contains both child nodes of ν, labeled b_1 and b_2, where $b = b_1 + b_2$.
- If ν is a leaf, it's label is the same as the number stored there.

Then there is an interval of length k iff

$$\exists\, a, b \leq m \,\forall\, x \leq m \,\big(b = a + k - 1 \wedge a \leq x \leq b \Rightarrow x \in \mathcal{L}(E)\big).$$

We have just seen $x \in \mathcal{L}(E)$ is in $\mathbb{NP}$, so SIEI is in Σ^p_3. □

Alas, completeness is much harder, we'll skip.

A similar argument shows that inequality of sum integer expressions is Σ^p_2.

We know how to attach an oracle $A \subseteq \Sigma^\star$ to a Turing machine $\mathcal{M}$. If we do this systematically for all machines in a certain class $\mathcal{C}$, we obtained a relativized class $\mathcal{C}^A$.

The oracle may vary over a class $\mathcal{D}$:

$$\mathcal{C}^{\mathcal{D}} = \bigcup_{A \in \mathcal{D}} \mathcal{C}^A$$

For example, $\mathbb{P}^{\mathbb{P}} = \mathbb{P}$ and $\mathbb{P}^{\mathbb{NP}} = \mathbb{P}^{\mathsf{SAT}}$.

Similar definitions apply to function classes, and function oracles.

We can now check that our original definition of PH in terms of projections and complements can also be expressed in terms of oracles like so:

$$\Sigma_0^p = \Pi_0^p \;=\; \mathbb{P}$$

$$\Sigma_{n+1}^p = \mathbb{NP}^{\Sigma_n^p}$$

$$\Pi_n^p = \mathsf{comp}(\Sigma_n^p)$$

$$\Delta_{n+1}^p = \mathbb{P}^{\Sigma_n^p}$$

$$\mathrm{PH} = \bigcup \Sigma_n^p$$

Exercise

Verify that our two definitions are really equivalent.

There is no analogue to the hierarchy theorem for the arithmetical hierarchy, but it is still true that

$$\Sigma_k^p \cup \Pi_k^p \subseteq \Delta_{k+1}^p \subseteq \Sigma_{k+1}^p \cap \Pi_{k+1}^p$$

Also, if there is a glitch at some level $k \geq 1$, then the rest of the hierarchy collapses:

$$\Sigma_k^p = \Pi_k^p \quad \text{implies} \quad \Sigma_k^p = \mathrm{PH}$$

$$\Sigma_k^p = \Sigma_{k+1}^p \quad \text{implies} \quad \Sigma_k^p = \mathrm{PH}$$

This leaves open the possibility that, say, the first 42 levels are proper, and the rest collapses. Perish the thought.

Lemma

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

$\mathrm{PH} = \mathrm{PSPACE}$ *implies the polynomial hierarchy collapses.*

Proof.

Containment in PSPACE is clear from the alternating quantifier description.

For the collapse, recall that QBF Validity is in PSPACE, and subsumes all the Σ_k Validity problems.

If we had equality, then QBF Validity would already be equivalent to Σ_k Validity for some level k, and everything would collapse down to that level.

□

We have Σ_k^p-complete problems for all levels $k \geq 1$, generic as well as concrete (at least for some levels).

But note that the last argument seems to rule out the existence of PH-complete problems: if L were PH-complete, then $L \in \Sigma_k^p$ for some k simply by the definition of the polynomial hierarchy.

But then the hierarchy collapses at level k—which sounds less than plausible. Famous last words.

1 The Polynomial Hierarchy

2 Alternating Turing Machines

We defined the polynomial hierarchy in analogy to the arithmetical hierarchy: by applying suitable projections and complements to polynomial time decidable sets. An alternative definition can be given in terms of oracles.

Question: Is there a machine model for PH?

In other words, we would like some class of Turing machines that defines precisely the languages in PH.

We will have to push the envelope a bit to make this happen, the resulting machines are a bit more complicated than ordinary Turing machines, or even plain nondeterministic ones.

- Plain, deterministic Turing machines are our goto model of computation; they have all kinds of great properties and some obnoxious ones. In particular, they are clearly physically realizable.

- Nondeterministic Turing machines add a significant level of abstraction: we are now dealing with a tree of computations; realizability fades away. Another substantial difference is that it becomes much more difficult to talk about computing functions, and even decision problems are somewhat awkward[†]

- Probabilistic Turing machines return to the realm of realizable computation and seem critical for a adequate definition of feasibility. But: there is a major difference between nondeterministic and probabilistic machines.

- Alternating Turing machines push the level of abstraction much higher and are quite removed from realizability.

[†] The big justification really is the enormous success of nondeterminism in the context of finite state machines.

A DFA is just the formalization of a perfectly practical algorithm: scan a string letter by letter, update your state via table lookup, decide acceptance on the basis of the last state. Runs in linear time and constant space with very good constants.

By contrast, an NFA is prima facie an abstraction: there may be exponentially many possible runs on a single input, and acceptance is determined by an existential quantification: is there a run . . . .

The reason this abstract model is still hugely important for practical algorithms is that acceptance testing for NFAs is still linear time, albeit with worse constants. On the other hand, other operations such as union are actually easier for NFAs.

Fast Acceptance Testing

Proposition

For any DFA $\mathcal{A}$ and any input string x we can test in time linear in $|x|$ whether $\mathcal{A}$ accepts x, with very small constants.

```
p = q0;                          // reset
while(  a = x.next() )           // next input symbol
      p = delta[p][a];           // table look-up

return  p in F;                  // table look-up
```

Of course, it might take some time to compute the lookup table δ in the first place, but once we have it acceptance testing is very fast.

The key insight is that testing for nondeterministic machines is very, very similar: instead of single states p, we have sets of states $P \subseteq Q$.

// nondeterministic acceptance testing

$P = I$
while $a = x$.next() **do**
$\quad P = \{\, q \mid \exists\, p \in P\, (p \xrightarrow{a} q) \,\}$
return $P \cap F \neq \emptyset$

Dealing with a set of states P rather than a single state p is slower, but only by a constant. And there are many hacks to make the computation reasonably fast in typical practical situations.

The total damage is still $O(|x|)$ and the constants are often quite reasonable.

So here is a wild idea:

> **Question:** Is there a useful notion of acceptance based on "for all runs such that such and such"?

One problem is whether these "universal" automata are more powerful than ordinary FSMs. As we will see, we still only get regular languages.

But this raises the question of how the state complexities compare: recall that nondeterministic FSMs can be exponentially smaller than their deterministic counterparts—one of the reasons they are attractive in practical pattern matching applications.

How would one formally define a type of FSM $\mathcal{A} = \langle Q, \Sigma, \delta; I, F \rangle$ where acceptance means all runs have a certain property?

The underlying transition system $\langle Q, \Sigma, \delta \rangle$ will be unaffected, it is still a labeled digraph.

The acceptance condition now reads:

> $\mathcal{A}$ accepts x if all runs of $\mathcal{A}$ on x starting at I end in F.

Let's call these machines ∀FA.

Read: for-all-FA. It's tempting to call them "universal FA", but that collides with the standard use where universal means "accepting all inputs."

By the same token, a NFA would be a ∃FA, a there-exists-FA.

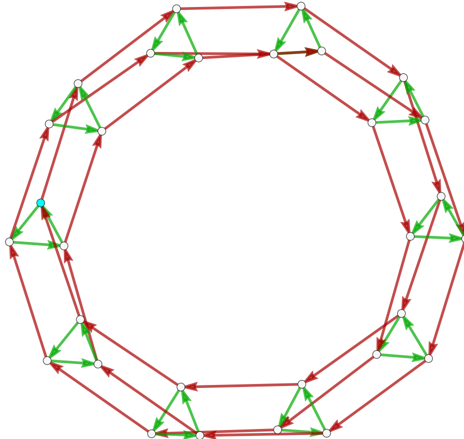
As an example consider the mod counter languages

$$K_{a,m} = \{\, x \in \mathbf{2}^\star \mid \#_a\, x = 0 \pmod{m} \,\}$$

with state complexity m. For the union $K_{0,m} \cup K_{1,n}$ we have a natural NFA of size $m + n$. However, for the intersection $K_{0,m} \cap K_{1,n}$ we only have a product machine that has size mn.

More importantly, note that nondeterminism does not seem to help with intersection: there is no obvious way to construct a smaller NFA for $K_{0,m} \cap K_{1,n}$.

This happens on a number of occasions: there are regular languages where nondeterminism seems utterly useless. The natural construction of the machine is automatically deterministic.



But we can build a $\forall$FA of size just $m + n$: take the disjoint union and declare the acceptance condition to be universal.

What is really going on here?

Let's assume that Q_1 and Q_2 are disjoint. Starting at $\{q_{01}, q_{02}\}$ we update both components. So after a while we are in state

$$\{p, q\} \qquad p \in Q_1, q \in Q_2$$

In the end we accept iff $p \in F_1$ and $q \in F_2$.

This is really no different from a product construction, we just don't spell out all the product states explicitly: a perfect example of a succinct representation.

Choosing clever representations is often critically important.

For example, acceptance testing for a ∀FA is basically the same as for an NFA: we keep track of the set of states $\delta(I, x) \subseteq Q$ reachable under some input and simply change the notion of acceptance: this time we want $\delta(I, x) \subseteq F$.

For example, if some word x crashes all possible computations so that $\delta(I, x) = \emptyset$, then x is accepted. This may sound weird, but it's perfectly fine.

Likewise we can modify the Rabin-Scott construction that builds an equivalent DFA: as before calculate the (reachable part of the full) powerset and adjust the notion of final state:

$$F' = \{ P \subseteq Q \mid P \subseteq F \}$$

There is almost no difference.

> A mathematician is a person who can find analogies between theorems; a better mathematician is one who can see analogies between proofs and the best mathematician can notice analogies between theories. One can imagine that the ultimate mathematician is one who can see analogies between analogies.
>
> S. Banach

We can think of the transitions in a NFA as being disjunctions:

$$\delta(p, a) = q_1 \lor q_2$$

We can arbitrarily pick q_1 or q_2 to continue. Similarly, in a $\forall$FA, we are dealing with conjunctions:

$$\delta(p, a) = q_1 \land q_2$$

meaning: We must continue both at q_1 and at q_2. So how about

$$\delta(p, a) = (q_1 \lor q_2) \land (q_3 \lor q_4)$$

Or perhaps

$$\delta(p, a) = (q_1 \lor \neg q_2) \land q_3$$

Does this make any sense?

Think of threads: both $\wedge$ and $\vee$ correspond to launching multiple threads. The difference is only in how we interpret the results returned from each of the threads.

For $\neg$ there is only one thread, and we flip the answer bit.

In other words, a "Boolean" automaton produces a computation tree very much like a plain NFA. But the acceptance condition is a bit more involved.

For historical reasons, these devices are called alternating automata.

In an alternating finite automaton (AFA) we admit transitions of the form

$$\delta(q, a) = \varphi(q_1, q_2, \ldots, q_n)$$

where φ is an arbitrary Boolean formula over $Q = \{q_1, q_2, \ldots, q_n\}$, even one containing negations.

How would such a machine compute? Initially we are in "state"

$$q_{01} \vee q_{02} \vee \ldots \vee q_{0k}$$

the disjunction of all the initial states.

One Step and Acceptance

Suppose we are in state Φ, some Boolean formula over Q. Then under input a the next state is defined by substituting formulae for the variables:

$$\Phi[\, q_1 \mapsto \delta(q_1, a), \ldots, q_n \mapsto \delta(q_n, a) \,]$$

The substitutions are supposed to be carried out in parallel, so each variable $q \in Q$ is replaced by $\delta(q, a)$, yielding a new Boolean formula. In the end we accept if

$$\Phi[F \mapsto 1, \overline{F} \mapsto 0] = 1$$

Meaning: replace all variables in F by `true`, and all variables in $\overline{F}$ by `false`.

Exercise

Verify that for both NFA and ∀FA this definition behaves as expected.

The name "alternating automaton" may sound a bit strange.

The original paper by Chandra, Kozen and Stockmeyer that introduced these machines in 1981 showed that one can eliminate negation without reducing the class of languages.

One can then think of alternating between existential states (at least one spawned process must succeed) and universal states (all spawned processes must succeed).

In a moment, we will use an analogous construction to Turing machines.

Theorem

Alternating automata accept only regular languages.

Proof.

Let $\text{Bool}(Q)$ be the collection of all Boolean formulae with variables in Q and $\text{Bool}_0(Q)$ a subset where one representative is chosen in each class of equivalent formulae (say, the length-lex first in DNF) and consider the corresponding normalization map $\nu : \text{Bool}(Q) \to \text{Bool}_0(Q)$.

We can build an equivalent DFA over the state set $\text{Bool}_0(Q)$ of state complexity at most 2^{2^n}.

- The initial state is $\nu(\bigvee_{q \in I} q)$.
- Transitions are $\Delta(\boldsymbol{p}, a) = \nu(\boldsymbol{p} \mapsto \delta(\boldsymbol{p}, a))$.
- The final states are $\{\, \varphi \in \mathsf{Bool}_0(Q) \mid \varphi[F \mapsto 1, \overline{F} \mapsto 0] = 1 \,\}$.

It is easy to see that the new, ordinary DFA is equivalent to the given, alternating one.

□

But note that the cost of eliminating alternation is potentially doubly exponential, significantly worse that for determinization (corresponding to logical-or only).

Because an AFA can be much, much smaller that the minimal DFA. In fact, the 2^{2^n} bound is tight: there are AFAs on n states where the minimal equivalent DFA is doubly exponential in n.

So we have a succinct representation for a regular language, but one that still behaves reasonably well under the usual algorithms. Avoiding the standard DFA representation is often critical for feasibility: in reality we cannot actually construct the full DFA in many cases. Laziness is a good idea in this context.

BTW, this is even true in pattern matching: DFAs should be avoided unless they are absolutely necessary (because the pattern contains a negation).

So we understand alternating finite state machines, but we really need to talk about alternating Turing machines.

As before, we assume there are two transition functions

$$\delta_i : Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$$

where $i \in \mathbf{2}$, we can choose either one at each step.

Here is the critical idea: every state is labeled by $\exists$ or $\forall$.

These labels are used in the definition of acceptance.

The notion of alternation comes from the fact that one could alternate between $\exists$ and $\forall$ states in a computation.

Suppose $\mathcal{M}$ is an ATM and let $\mathfrak{C}(\mathcal{M}, x)$ be the computation graph defined as usual.

Now define the following class of "accepting nodes" in $\mathfrak{C}(\mathcal{M}, x)$:

- Accepting configurations are accepting nodes.
- If state p in configuration C is labeled $\exists$, and there is a successor of C that is accepting, then C is also accepting.
- If state p in configuration C is labeled $\forall$, and all the successors of C are accepting, then C is also accepting.

We say that $\mathcal{M}$ accepts x if the initial configuration $q_0 x$ is accepting.

Just to be entirely clear about this: the computation graph $\mathfrak{C}(\mathcal{M}, x)$ has not changed at all, it's no different from any old nondeterministic Turing machine.

What has changed is our definition of acceptance, just having at least one path to an accepting configuration is no longer sufficient. In particular, acceptance cannot be simply translated into a graph reachability question.

Definition

Let $\mathcal{M}$ be an alternating Turing machine.
We say that $\mathcal{M}$ runs in alternating time $t(n)$ if every path in the computation graph $\mathfrak{C}(\mathcal{M}, x)$, $|x| = n$, has length at most $t(n)$.
Similarly we define alternating space.

In symbols: $\mathsf{ATIME}(t)$ and $\mathsf{ASPACE}(s)$

Lastly, define

$$\mathsf{AP} = \mathsf{ATIME}(\mathsf{poly})$$

$$\mathsf{ALOG} = \mathsf{ASPACE}(\log)$$

The big question is how these alternating polynomial time/space classes relate to ordinary complexity classes.

Lemma

ALOG $= \mathbb{P}$

Lemma

AP $= \mathrm{PSPACE}$

Both claims will follow easily from the more general results below.

Theorem

Suppose $f(n) \geq n$ *is reasonable. Then*

$$\mathsf{ATIME}(f) \subseteq \mathrm{SPACE}(f) \subseteq \mathsf{ATIME}(f^2)$$

Theorem

Suppose $f(n) \geq \log n$ *is reasonable. Then*

$$\mathsf{ASPACE}(f) = \mathrm{TIME}(2^{O(f)})$$

ATIME(f) $\subseteq \mathrm{SPACE}(f)$

Suppose $\mathcal{M}$ is alternating, $O(f)$ time. Construct a simulator $\mathcal{M}'$ that performs a DFS traversal of the computation graph $\mathfrak{C}_{\mathcal{M}}(x)$ of $\mathcal{M}$ on input x to determine acceptance.

We have to make sure that $\mathcal{M}'$ needs only $O(f)$ deterministic space. The naive approach would produce a recursion stack of depth $O(f)$, with stack frames of size $O(f)$, yielding space complexity $O(f^2)$.

To avoid this, simply record the nondeterministic choice at each step (a single bit), and recompute the actual configuration when needed (we are focused on space, not time).

$\mathrm{SPACE}(f) \subseteq \mathsf{ATIME}(f^2)$

This is similar to the approach in Savitch's theorem. Instead of the plain recursion used there, our alternating machine will branch existentially to find the "middle point", and then universally to verify both pieces.

We check for paths of length $2^{cf(n)}$ where c is large enough so this number bounds the total number of configurations. Hence the total damage is $O(f^2)$ alternating time.

ASPACE$(f) \subseteq \mathrm{TIME}(2^{O(f)})$

Suppose $\mathcal{M}$ is alternating, $O(f)$ space. Construct a simulator $\mathcal{M}'$ that builds the computation graph $\mathfrak{C}_{\mathcal{M}}(x)$ of $\mathcal{M}$ on input x, first ignoring alternation. Nodes have size at most $cf(n)$ for some constant c.

Then run a marking algorithm that labels nodes as accepting when appropriate, working backwards from the leaves. Accept x if the initial configuration is ultimately marked.

The size of the graph is $2^{O(f)}$, and one round of marking is linear in the size. There are at most $2^{O(f)}$ rounds, yielding our deterministic time bound.

$\mathrm{TIME}(2^{O(f)}) \subseteq \mathsf{ASPACE}(f)$

This time $\mathcal{M}$ is an ordinary deterministic machine, time $2^{O(f)}$. The alternating machine $\mathcal{M}'$ must obey a $O(f)$ space bound, so we cannot simply construct a computation graph. This is rather tricky, here is a sketch of the proof.

Think of the computation of $\mathcal{M}$ on input x as being laid out in a $N \times N$ square, $N = 2^{O(f)}$, much as in the homework problem on square tilings (except here we don't have to bother with tilings, each row will just be a configuration represented as a word in $\Sigma^\star Q \Sigma^\star$). The square itself is too large, but we can keep pointers to individual cells.

We guess the position of the accepting state in (r, c). Then, for all $s = r, r{-}1, \ldots, 1$, we guess existentially the three cells in the row below, check that they are good, and universally verify that the parents are also good. In the bottom row we verify against the initial configuration of $\mathcal{M}$.

It seems plausible that $\mathrm{PH} \subsetneq \mathrm{PSPACE}$, so our ATMs are a bit too powerful to characterize the polynomial hierarchy.

To fix this, we can impose more constraints: call an ATM Σ_k if the initial state is labeled $\exists$, and every path in $\mathfrak{C}(\mathcal{M}, x)$ alternates at most $k-1$ times between different labels. This produces a notion of $\Sigma_k\mathrm{TIME}$ that matches with the polynomial hierarchy:

$$\Sigma_k^p = \bigcup \Sigma_k\mathrm{TIME}(n^c).$$

So it is unbounded alternation that seems to push us a bit too far.

Kozen has shown that, for every k, one can construct an oracle A such that

$$(\Sigma_k^p)^A \neq (\Sigma_{k+1}^p)^A = \mathrm{PSPACE}^A$$

One could interpret this as a warning that one should not be too sure about the structure of the polynomial hierarchy.

Just to be clear, very little is known about collapsing/separating oracles in the spirit of Baker-Gill-Solovay, and the inferences one should draw from these results. For example, it is an open problem whether

$$\mathrm{Pr}_A[\mathbb{P}^A = \mathbb{NP}^A] = 1.$$

The BGS theorem just shows that there is some A for which equality holds, and another for which it fails.