UCT

Space Bounds

Klaus Sutner

Carnegie Mellon University

Spring 2024

1 Space

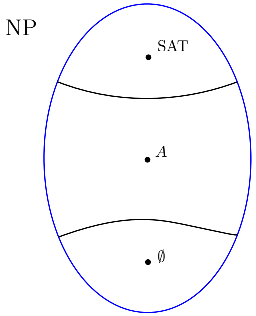
2 Space Classes

3 Nondeterministic Space

4 The Zoo

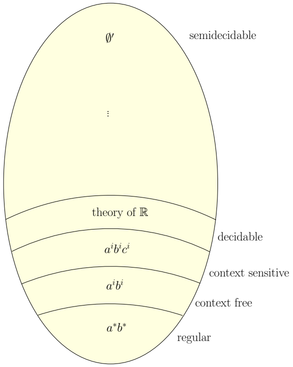
Where Are We?

Based on time complexity and the fundamental idea of nondeterminism, one of the key area in the vicinity of “easily decidable” looks like this:



Ladner’s theorem provides the intermediate A, assuming $\mathbb{P} \neq \mathbb{NP}$.

1950s



Before modern complexity theory, there was Chomsky's hierarchy, motivated by linguistics rather than effective computation.

A Nuisance

Take regular languages and finite state machines, one of the great success stories of computer science.

What is the complexity of a regular language?

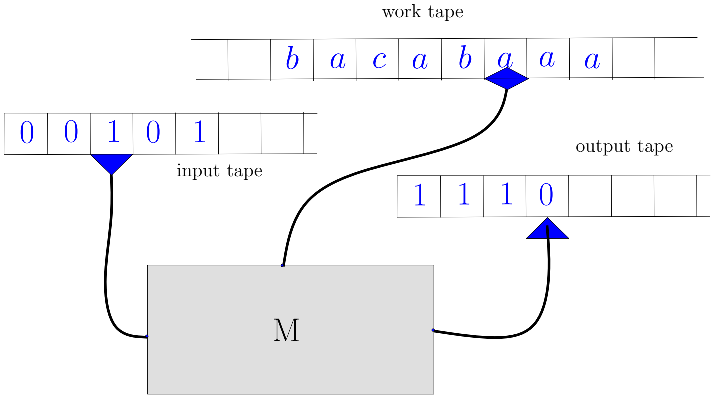
Certainly linear time, but that description misses the boat: the key here is that finite state machines are memory-less. We currently don't have a good way of describing this scenario.

How About Space?

Both from the theory perspective (Blum's idea of an abstract complexity measure) and practical considerations, arguably the second most important complexity measure besides time is space—the memory required to carry out a computation.

- Space is a tighter bound than Time.
- Space, unlike Time, can be reused.
- In the RealWorld$^{\text{TM}}$, space is much less forgiving than time.

Recall: Transducer



Measuring Work Space

- We are using an off-line machine: there is a separate, read-only, two-way input tape. So the input can be read repeatedly (but we cannot mark it any way).
- The machine also has a separate write-only output tape. For acceptors it is more convenient to use special states instead.
- The work tape is read/write in the usual manner. The space complexity is measured exclusively in terms of the work tape.

In other words, we do not hold the poor Turing machine responsible for input or output size.

Space Complexity

More precisely, let $(C_i)_{i<N}$ be the computation of TM $\mathcal{M}$ on input x. Write $\mathsf{spc}(C)$ for the number of tape cells used on the work tape in configuration C. Then the space complexity of $\mathcal{M}$ on x is defined as

$$S_{\mathcal{M}}(x) = \max\Big(\,\mathsf{spc}(C_i) \mid 0 \leq i < N\,\Big)$$

In other words, we consider the largest configuration that appears during the computation. This makes perfect sense, it does not help much if most of the computation requires little space, but there is a brief phase where memory goes way up.

As usual, quietly assume $S_{\mathcal{M}}(x) = \infty$ if the computation diverges and uses arbitrarily large configurations. But note: a computation can diverge and have limited space complexity.

White and Gray Blanks

One way of avoiding the search for the maximum tape use is to modify the definition of our Turing machines slightly: orginally the work tape is filled with *white blank* symbols (recall that the input is separate).

Once the machine gets going, it never writes a white blank, only other tape symbols and *gray blanks*. So in the end we can simply count the ordinary symbols plus the gray blanks on the tape (which necessarily form a contiguous block).

In other words, we cannot cheat by using a huge amount of memory at some point, and then just erasing everything with white blanks.

Worst Case Space Complexity

As before with time, we focus on all instances of size n rather than individual ones.

$$S_{\mathcal{M}}(n) = \max\Big(\, S_{\mathcal{M}}(x) \mid x \text{ has size } n \,\Big)$$

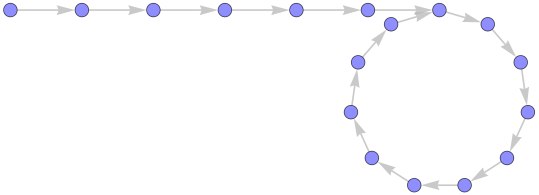
Again, this is worst case, average case space complexity can be defined similarly. As it turns out, average time complexity is very interesting, but average space complexity is less so (consider quick sort).

This is really a topic for an algorithms course.

Same Old?

Question: What are the similarities/differences between space and time complexity?

Here is a difference: a divergent computation obviously requires an unbounded amount of time. But a divergent computation may well only use bounded amount of space: the computation may be caught in a loop



Note that a loop cannot be too long: there are only exponentially many configurations (of a Turing machine) with a given space bound.

Loop Monitors

Unlike Halting, looping is decidable: given an arbitrary Turing Machine $\mathcal{M}$, we can construct a new machine $\mathcal{M}'$ that

- simulates $\mathcal{M}$, and
- keeps track of the history of the computation.

Say, we keep track of the history (all previous configurations) on an extra tape. Then we can easily check whether $\mathcal{M}$ has entered a loop.

Exercise

How much of a slow-down would this simulation cause?
How does the space complexity change?

1 Space

2 **Space Classes**

3 Nondeterministic Space

4 The Zoo

Space Complexity Classes

Definition

Let $f : \mathbb{N} \to \mathbb{N}$ be a function.

$$\mathrm{SPACE}(f) = \{\, \mathcal{L}(\mathcal{M}) \mid \mathcal{M} \text{ a TM}, S_{\mathcal{M}}(n) = O(f(n)) \,\}$$

A (deterministic) space complexity class is a class

$$\mathrm{SPACE}(\mathcal{F}) = \bigcup_{f \in \mathcal{F}} \mathrm{SPACE}(f).$$

for some collection of functions $\mathcal{F}$.

Again, as in the time complexity case, one needs to be a bit careful with technical details.

Space Constructibility

In this case, the critical condition is the following: a function $s : \mathbb{N} \to \mathbb{N}$ is space constructible if there is a Turing machine that runs in $O(s(n))$ space and, for any input of size n, writes $s(n)$ on the tape as output (in binary).

Any function one encounters in the wild is indeed space constructible.

Without this assumption one has to jump through extra hoops in certain proofs, or the arguments may fail altogether. For example, one may have to try out possible values for $s(n)$ until we find one that works.

Some Important Classes

Here are some typical examples for deterministic space complexity classes.

- $\mathrm{SPACE}(1)$, constant space.
- $\mathrm{SPACE}(\log n)$, logarithmic space.
- $\mathrm{SPACE}(n)$, linear space.
- $\mathrm{PSPACE} = \mathrm{SPACE}(\text{poly})$, polynomial space.

Note that unlike with polynomial time, polynomial space is really a step to too far, even if we assume that the polynomials involved are low degree for real problems: a cubic memory requirement would mean a megabyte of input turns into an exabyte.

Linear or slightly superlinear space is a much more reasonable constraint for feasible computation.

How Small Can It Get?

Recall that we assume that a Turing machine reads all its input, so for time complexity we have $n \leq T_{\mathcal{M}}(n)$.

Clearly, the analogous restriction $n \leq S_{\mathcal{M}}(n)$ would be totally inappropriate, it makes perfect sense to talk about sub-linear space complexities.

In fact, even constant space complexity makes sense.

Lemma

$\mathrm{SPACE}(1)$ *is the class of regular languages.*

Proof

Consider a Turing machine that uses space $\leq c$ and uses accept/reject states (rather than output).

We can remove the work tape by encoding its information in state: there are only $[c] \times \Sigma^c$ possible tape inscriptions and head positions and we think of them as being part of the machine state.

The resulting machine is finite state, but the input tape is two-way. We can invoke a classical theorem from the Rabin/Scott 1959 paper that shows that there is an equivalent one-way finite state machine (and the proof is constructive).

$\square$

A Little Exercise

Suppose a directed graph is represented by its adjacency list, written out as a string:

$$\# \, a_{11}|a_{12}|\ldots|a_{1d_1} \; \# \, a_{21}|a_{22}|\ldots|a_{2d_2} \; \# \; \ldots \; \# \, a_{n1}|a_{n2}|\ldots|a_{nd_n} \; \#$$

where, say, $\Sigma = \{0, 1, |, \#\}$ and the $a_{ij} \in \mathbf{2}^\star$ represent numbers in $[n]$ written in binary, padded to uniform length k.

Each adjacency list $\# \, a_{i1}|a_{i2}|\ldots|a_{id_i}$ contains up to $n-1$ k-bit blocks, separated by $|$. The size of the input is essentially $k \sum (d_i + 1) = O(\log n \, n^2)$.

Exercise

Find a "natural" graph problem that is in

- *linear space*
- *logarithmic space*
- *doubly logarithmic space*

Falling off a Cliff

The last example is already a bit contrived. Certainly it is not at all clear how to come up with a $\log\log\log n$ algorithm, never mind $\log^4 n$ or some such.

As mentioned, $\mathrm{SPACE}(1)$ is the class of regular languages.

Innocent question: is there anything between $\log\log n$ and 1?

Theorem (Hartmanis, Lewis, Stearns, 1965)

Let $f(n) = o(\log\log n)$. Then $\mathrm{SPACE}(f)$ is the same as constant space.

Note the little-oh, there are non-regular languages recognizable in space $\log\log n$.

Proof Sketch

Suppose $\mathcal{M}$ accepts some non-regular language in space f. For every $k \geq 0$, there must be some input that requires at least k space. Hence we can define an unbounded sequence of numbers

$$\ell_k = \min\big(\ell \mid \exists\, x \in \Sigma^{\ell} (\mathcal{M} \text{ uses} \geq k \text{ space on } x) \;\big)$$

Let $x_k \in \Sigma^{\star}$, $|x| = \ell_k$, be a corresponding witness.

We use crossing sequences:[†] when the input head moves between positions i and $i+1$, record the configuration of the machine in a crossing sequence $\mathcal{X}_i$.

[†]You may have seen these in the proof that 2-way DFA are no more powerful than 1-way DFA, or in the argument that a 1-tape TM requires quadratic time to recognize palindromes.

Contd

The number of all possible configurations of $\mathcal{M}$ in input x_k is bounded by $2^{cf(\ell_k)} + \log n$, where c is some constant: the state plus work tape part, plus the position of the read head. We'll get rid of the latter.

Since $f(n) \in o(\log\log n)$, for k sufficiently large, the number of configurations of $\mathcal{M}$ on input x_k is $o(\log \ell_k)$. The entries in a single crossing sequence must be distinct, otherwise we are stuck in a loop. So we have $o(\ell_k)$ distinct possible sequences (really, crossing sets).

On the other hand, the number of crossing sequences is ℓ_k and we must have $\mathcal{X}_i = \mathcal{X}_j$ for some $i \neq j$.

But then we can do surgery on the witness x_k to produce a shorter word that already requires $f(\ell_k)$ space, contradicting the minimality of the witness.

$\square$

Space vs. time

it is clear from the definitions that time is more constrained than space:

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

There is a better result due to Hopcroft et al., but that requires more work.

For a bound going in the opposite direction, we can count the number of instantaneous descriptions of a Turing machine of given space complexity to obtain a bound on the length of any accepting computation of such a machine.

Theorem

Let $f(n) \geq \log n$*. Then*

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

Computation Graphs

Here is a simple tool that is often useful: given a Turing machine $\mathcal{M}$ (deterministic or nondeterministic), define the computation graph $\mathfrak{C}(\mathcal{M})$ as follows:

- vertices are all configurations of $\mathcal{M}$,
- there is an edge (C, C') if the machine can go from C to C' in one step.

Usually we are only interested in the subgraph $\mathfrak{C}(\mathcal{M}, x)$, for some input x: the part of $\mathfrak{C}(\mathcal{M})$ accessible from the initial configuration C_x^{init}.

The size of the nodes in $\mathfrak{C}(\mathcal{M}, x)$ is $O(s(|x|))$ where $s(n) \geq \log n$ is the space complexity of $\mathcal{M}$ (we can ignore larger nodes as far as acceptance goes; we could also use strong complexity).

Computation Trees

For deterministic machines, $\mathfrak{C}(\mathcal{M}, x)$ is boring, but for nondeterministic ones we get branching.

On occasion it is preferable to think of the computations of a Turing machine as describing a tree: C' is a child of C if the machine can move from C to C' in one step. Technically, in order to ensure a tree structure, we need to distinguish between nodes on different branches. For example, we could use finite sequences of configurations

$$C_x, C_1, C_2, \ldots, C_{k-1}, C_k$$

as nodes, rather than just configurations: we keep track of the history of the computation.

This is really a general method in graph theory, we can unfold a digraph into a tree (which may well be infinite if there are cycles).

Proof of Theorem

We need to show that $\mathrm{SPACE}(f) \subseteq \mathrm{TIME}(2^{O(f(n))})$.

So let $\mathcal{M}$ be any space f Turing machine and x an input of size n.

The nodes in the computation graph $\mathfrak{C}(\mathcal{M}, x)$ have size $O(f(n))$ since $f(n) \geq \log n$, so the size of the graph is at most $O(2^{c\, f(n)})$ and it can be constructed in the this amount of time.

But then we can simply run a linear time reachability test like DFS to check for an eccepting configuration in the graph.

$\square$

Recall: Time Hierarchy Theorems

It is intuitively clear that $\mathrm{TIME}(f)$ will be larger than $\mathrm{TIME}(g)$ provided that f is sufficiently much "larger" than g. Since the simulation requires a bit of time, one has to be a bit careful with technical details in a time hierarchy theorem.

Theorem (Hartmanis, Stearns 1965)

Let f be time constructible, $g(n) = o(f(n))$.

Then $\mathrm{TIME}(g(n)) \subsetneq \mathrm{TIME}(f(n)\log f(n))$.

And Space?

As the example of log-log space bounds shows, we need to be careful.

In the following, we assume that time/space functions are always reasonable (say, time-constructible and space-constructible).

Remember Poincaré: don't insult the fathers (insulting mothers was presumably OK around 1900).

Also, on occasion it will be helpful to assume that the functions in question are not too small, something along the lines of $f(n) \geq \log n$.

We'll just call these functions reasonable.

Space Hierarchy

The analogous result for space is actually a bit easier to state.

Theorem (Hartmanis, Lewis, Stearns 1965)

Let f, g *be reasonable,* $g(n) = o(f(n))$.

Then $\mathrm{SPACE}(g(n)) \subsetneq \mathrm{SPACE}(f(n))$.

As with time complexity, these results are proved by diagonalization and do not produce natural examples of hard problems.

Given a concrete combinatorial problem it is usually very, very hard to find a lower bound for its concrete space complexity, we have to make do with hardness results, just as for time complexity.

Proof

There is a universal Turing machine $\mathcal{U}$ that has space complexity $f(n)$ and simulates all $g(n)$-space machines $\mathcal{M}_e$ in the sense that there is a constant c_e such that for $n = |x|$:

- $\mathcal{U}(e\#x)$ uses space $c_e \cdot g(n)$,
- $\mathcal{U}(e\#x) = \mathcal{M}_e(x) \in \mathbf{2}$ provided that $c_e \cdot g(n) \leq f(n)$,
- $\mathcal{U}(e\#x)$ aborts if $c_e \cdot g(n) > f(n)$.

Note that c_e cannot be avoided, the simulated machine may have a larger tape alphabet.

So $\mathcal{U}$ has space complexity f by brute force. This is completely analogous to using a clock to enforce time complexity; this time, we take scissors and cut off a length of tape.

Contd

Now consider a new machine $\mathcal{M}$ such that

$$\mathcal{M}(e\#x) = 1 - \mathcal{U}(e\#e\#x) = 1 - \mathcal{M}_e(e\#x)$$

So we flip the output bit if there is one, and abort otherwise. Note that $\mathcal{U}$ simulates $\mathcal{M}_e$ on input $e\#x$, the usual diagonalization mumbo-jumbo.

Then $\mathcal{M}$ also has space complexity $f(n)$, but $\mathcal{M}(e\#x)$ disagrees with $\mathcal{M}_e(e\#x)$ for all sufficiently large x by the very definition of $\mathcal{M}$.

Thus $\mathcal{M}$ is not in $\mathrm{SPACE}(g)$, as required.

$\square$

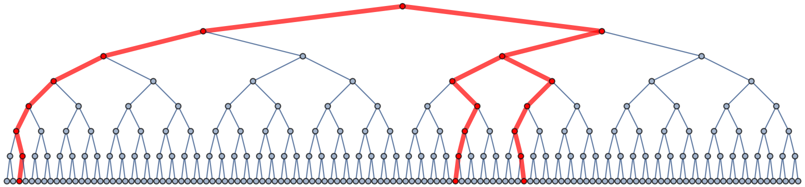
1 Space

2 Space Classes

3 **Nondeterministic Space**

4 The Zoo

Nondeterministic Machines



Space complexity also makes sense for nondeterministic machines: pick the (accepting) branch that requires the least memory.

This is directly analogous to our definition of nondeterministic time complexity.

Nondeterministic Space

We can handle space complexity $S_{\mathcal{M}}(x)$ in a similar manner:

$$S_{\mathcal{M}}(x) = \min\big(\,\mathsf{spc}(\beta) \mid \beta \text{ accepting branch in } \mathcal{T}_x\,\big)$$

Here $\mathcal{T}_x$ is the computation tree of the machine on input x, and spc(β) stands for the largest amount of memory used along branch β. Note that this does not necessarily mean the the branch shorter than all others.

Similarly define $S_{\mathcal{M}}(n)$ in the usual worst-case manner. We will not be concerned with average-case space complexity here.

Weak versus Strong

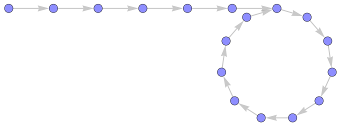
One might object that from an algorithms point of view our definition is a bit weak: arguably we would prefer a bound on all branches, not just on the accepting ones. One should not distinguish between Yes and No instances when it comes to resource bound.

This alternative notion is called strong space complexity. Similarly we could use strong time complexity.

All true, but most papers in complexity theory use the weak model rather than the strong: the difference only comes into play when dealing with low classes, otherwise we can clean up the machines to make them behave properly. And weak is easier to deal with than strong.

Trade-Offs

Note that one can sometimes trade space for time: a computation can be made to use less memory by re-computing values rather than storing them. Floyd's cycle detection algorithm is a nice example.



Memoizing is a trick that goes in the opposite direction: avoid recomputation at a cost of an additional hash table.

Or space measure ignores these issues, we go for the least memory-intensive branch, no matter how long the computation is. Obviously, it would be of interest to combine both measures.

Nondeterministic Complexity Classes

We can now lift the definitions of deterministic space complexity classes to nondeterministic ones.

Definition

Let $f : \mathbb{N} \to \mathbb{N}$ be a (reasonable) function.

$$\mathrm{NSPACE}(f) = \{\, \mathcal{L}(\mathcal{M}) \mid \mathcal{M} \text{ a nondeterministic TM}, S_{\mathcal{M}}(n) = O(f(n)) \,\}$$

As before, this generalizes easily to $\mathrm{NSPACE}(\mathcal{F})$ for some class of functions $\mathcal{F}$.

As usual, this is for acceptors (decision problems) only, we don't yet know how to handle nondeterministic transducers. In fact, it's difficult: think about computing a function on a nondeterministic machine. There is a major clash between nondeterminism and single-valuedness.

Some Results

From the definitions

$$\mathrm{TIME}(f) \subseteq \mathrm{NTIME}(f) \subseteq \mathrm{NSPACE}(f)$$

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

More interesting is the question how much deterministic time/space is required to capture a nondeterministic class. Using standard ideas from deterministic simulation, we can show the following.

Theorem

Assume that $\log n \leq g(n)$*. Then*

$$\mathrm{NTIME}(f) \subseteq \mathrm{SPACE}(f)$$
$$\mathrm{NSPACE}(g) \subseteq \mathrm{TIME}(2^{O(g)})$$

Savitch's Theorem

Theorem (Savitch 1970)

Assume that $\log n \leq g(n)$. *Then*

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

Sketch of proof.

This is again deterministic simulation, but keeping an eye on space requirements.

The trick is to use a divide-and-conquer approach: a computation of length t is broken up into two subcomputations of length $t/2$.

Since the target is a space class and space, unlike time, can be reused, one can show that the divide-and-conquer algorithm requires no more than $g(n)^2$ space.

Savitch Proof

Let L be in $\mathrm{NSPACE}(g(n))$. Suppose $\mathcal{M}$ is a non-deterministic Turing machine that accepts L and has space complexity $O(g(n))$. Let x be an input of length n.

Clearly we are dealing with yet another path-existence problem in the computation graph of $\mathfrak{C}(\mathcal{M}, x)$. This time we will use a recursive algorithm to check for paths.

Define a reachability predicate that bounds the number of steps to get from one configuration to the other (uniformly in n). Here C_1, C_2 are configurations of $\mathcal{M}$ of size at most $c\, g(n)$.

$$\mathsf{reach}(C_1, C_2, k) \iff \exists\, s \leq 2^k \left(C_1 \vdash^s_{\mathcal{M}} C_2\right)$$

Proof Contd

Note that $x \in L$ iff reach$(C_x^{\text{init}}, C_Y^{\text{halt}}, O(g(n)))$, so to test membership in L we only need to compute the reach relation.

Claim

reach$(C_1, C_2, 0)$ *iff* $C_1 = C_2$ *or* $C_1 \vdash^1_{\mathcal{M}} C_2$.

reach$(C_1, C_2, k+1)$ *iff* $\exists C \left(\text{reach}(C_1, C, k) \wedge \text{reach}(C, C_2, k)\right)$.

The claim is obvious from the definition, but note that it provides a recursive definition of reach.

Also observe that the recursion involves an exponential search for the intermediate configuration C.

Implementing the Recursion

Implementing a recursion requires memory, typically a recursion stack: we have to keep track of pending calls and the required memory information.

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

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

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

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

$\square$

1 Space

2 Space Classes

3 Nondeterministic Space

4 The Zoo

Some Classes

- $\mathrm{SPACE}(1)$, constant space, regular languages
- $\mathbb{L} = \mathrm{SPACE}(\text{log})$, logarithmic space
- $\mathbb{NL} = \mathrm{NSPACE}(\text{log})$, 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
- Classical stuff: CFL, CSL, primitive recursive, decidable, semidecidable.

The Chain

Theorem

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

Proof.

The sticky points are

- $\mathbb{NL} \subseteq \mathbb{P}$: see next slide
- $\mathbb{NP} \subseteq \mathrm{PSPACE}$: search for all witnesses
- $\mathrm{PSPACE} = \mathrm{NPSPACE}$: Savitch's theorem

Proof

To show that $\mathbb{NL} \subseteq \mathbb{P}$, suppose we have a logarithmic space acceptor $\mathcal{M}$.

The nodes in the computation graph $\mathfrak{C}(\mathcal{M}, x)$ can be specified in $O(\log n)$ bits: the input head position requires $\log n$, and the rest of the configuration is $O(\log n)$.

Hence the size of the graph is polynomial in n. Moreover, we can construct the graph in polynomial time (say, write down the adjacency lists).

But $\mathcal{M}$ accepts x iff there is a path in $\mathfrak{C}(\mathcal{M}, x)$ from C_x^{init} to C_Y^{halt}. We can use any standard graph reachability algorithm to check this.

$\square$

Separation

The time and space hierarchy results of Hartmanis, Lewis and Stearns (from 1965, no less) give us some rather weak separation results:

Theorem

$\mathbb{L} \neq \mathrm{PSPACE}$ *and* $\mathbb{P} \neq \mathrm{EXP}_1$.

Surprisingly, that's it for separation results: nothing else is known at this point.

At the core of all these results is old-fashioned diagonalization: a naysayer might kvetch that nothing has happened since Cantor (well, let's say, since Turing).

That would be grossly unfair; to name just one example, $\mathrm{MIP}^\star = \mathrm{RE}$ is truly mindboggling.

Oracles and Relativization

Alas, it is not so easy to come up with other promising approaches to separation. Annoyingly, it seems that entirely new ideas are needed, there is little hope to transfer known methods from CRT.

One reason is that CRT seems invariant under oracles: all the basic results of the classical theory carry over very nicely and with essentially zero effort to a situation where oracles are allowed (Turing's genius at work, again).

The classical theory really is nothing but the study of $\{e\}^{\emptyset}$.

This means that for every theorem in classical computability theory there is a relativized version that uses an oracle. In particular all arguments involving diagonalization carry over to the oracle world.

Oracle Classes

By fixing a particular oracle $A \subseteq \Sigma^\star$ and attaching it to all machines in a certain class we can relativize the class.

For example, $\mathbb{P}^A$ is the collection of all decision problems decidable by a polynomial time Turing machine with oracle A.

Clearly, $\mathbb{P}^A = \mathbb{P}$ whenever $A \in \mathbb{P}$, but for "smarter" oracles we get more interesting problems.

Exercise

What would $\mathbb{P}^{\mathsf{SAT}}$ *look like*[†]*?*

[†] This is perfectly practical and historically accurate, just think of the oracle as being given by a state-of-the-art SAT solver. What can you do with it?

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 $\mathbb{P}^B \neq \mathbb{NP}^B$.

In conjunction with the relativization claim from above, 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 it is not really clear which line of attack might ultimately succeed in separating the classes. Not to mention that a group of researchers think that the answer really is $\mathbb{P} = \mathbb{NP}$.

Proof Sketch

For the identity $\mathbb{P}^A = \mathbb{NP}^A$, we can use a padded version of halting:

$$A = \{\, e\#x\#0^n \mid \mathcal{M}_e \text{ accepts } x \text{ in at most } 2^n \text{ steps} \,\}$$

This is a padded version of the Halting set for plain exponential computations. With this oracle it is not too hard to check that

$$\mathbb{P}^A = \mathbb{NP}^A = \mathrm{EXP}_1$$

The oracle "absorbs" the potential advantages of nondeterministic computation.

Incidentally, we could also use a PSPACE-complete oracle (say, quantified Boolean formulae).

Proof Sketch II

To obtain $\mathbb{P}^B \neq \mathbb{NP}^B$, first define the length set of $B \subseteq \Sigma^\star$ as

$$L_B = \{\, 0^{|x|} \mid x \in B \,\} \subseteq 0^\star$$

and note that the tally language L_B is in $\mathbb{NP}^B$.

Hence it suffices to construct B such that $L_B \notin \mathbb{P}^B$.

This time, we won't be able to give an explicit definition of B, instead we resort to our standard trick: B is constructed in stages. We need the usual enumeration of Turing machines $(\mathcal{M}_e)$ that return 0/1 outputs (we will clock them later).

Initially, $B_0 = \emptyset$. At each stage σ, we will put some strings into B, or block them from ever entering B. As usual, B_σ always determines only finitely many strings, the whole construction is effective.

Stage σ

Let $n = \max\Big(|x| \mid x \text{ determined by } B_{<\sigma} \Big) + 1$.

Run $\mathcal{M}_\sigma^{B_{<\sigma}}(0^n)$ for $2^n/42$ steps[†]. This is not a typo, we are doing more than just destroy polynomial time computations.

Here we assume that oracle queries $z \in B_{<\sigma}$? are handled as follows:

- If the membership status of z has been settled, return the correct answer.
- Otherwise, answer No, and block z from entering B.

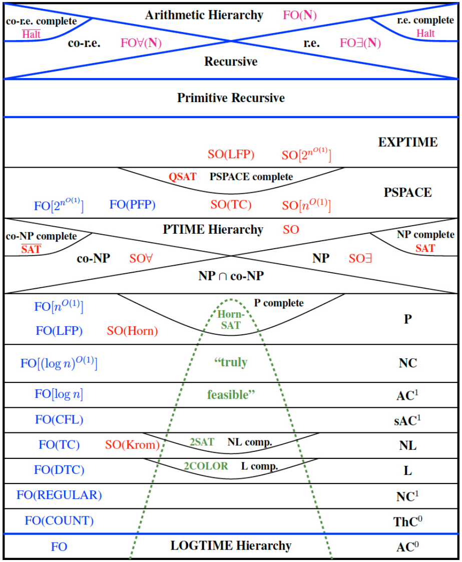
Now if $\mathcal{M}_\sigma^{B_{<\sigma}}(0^n)$ accepts, then block all strings in $\mathbf{2}^n$ from B.

Otherwise, pick a free string $z \in \mathbf{2}^n$ and place it in B_σ, block the rest. This string is guaranteed to exist, right?

$\square$

[†]Ponder deeply: why 42?

Immerman's Zoo



Arithmetic Hierarchy
FO(N)
co-r.e. complete
Halt
r.e. complete
Halt
co-r.e.
FO∀(N)
r.e.
FO∃(N)
Recursive
Primitive Recursive
EXPTIME
SO(LFP)
SO[2^{n^{O(1)}}]
QSAT
PSPACE complete
PSPACE
FO[2^{n^{O(1)}}]
FO(PFP)
SO(TC)
SO[n^{O(1)}]
PTIME Hierarchy
SO
co-NP complete
SAT
NP complete
SAT
co-NP
SO∀
NP
SO∃
NP ∩ co-NP
FO[n^{O(1)}]
P complete
Horn-SAT
P
FO(LFP)
SO(Horn)
FO[(log n)^{O(1)}]
"truly
NC
FO[log n]
feasible"
AC^1
FO(CFL)
sAC^1
FO(TC)
SO(Krom)
2SAT
NL comp.
NL
FO(DTC)
2COLOR
L comp.
L
FO(REGULAR)
NC^1
FO(COUNT)
ThC^0
FO
LOGTIME Hierarchy
AC^0