UCT

Polynomial Space

Klaus Sutner

Carnegie Mellon University

Spring 2022

Towards Polynomial Space

Logarithmic space and linear (deterministic) space clearly make algorithmic sense. But how about nondeterministic linear space?

Generalizing context-free grammars naturally leads to context-sensitive grammars and the obvious parsing “algorithm” for a context-sensitive language is in $\mathrm{NSPACE}(n)$ (and, by Savitch, in $\mathrm{SPACE}(n^2)$).

Somewhat surprisingly, it’s just a small step from there to all of PSPACE.

Context-Sensitive Grammars

Definition (CSG)

A context-sensitive grammar (CSG) is a grammar where all productions are of the form

$$\alpha A \beta \to \alpha \gamma \beta \qquad \text{where } \gamma \neq \varepsilon$$

For technical reasons, some authors also allow $S \to \varepsilon$ in which case S may not appear on the righthand side of any production. A language is context-sensitive if it can be generated by a context-sensitive grammar.

Note the constraint that the replacement string $\gamma \neq \varepsilon$; as a consequence we have

$$\alpha \Rightarrow \beta \qquad \text{implies} \qquad |\alpha| \leq |\beta|$$

Brute-Force Recognition

Lemma

Every context-sensitive language is decidable.

Proof.

Suppose $w \in \Sigma^n$. In any potential derivation $(\alpha_i)_{i<N}$ we have $|\alpha_i| \leq n$.

So consider the derivation graph $\mathcal{D}$ (recall the pictures from last time for CFLs):

- vertices are $\Gamma^{\leq n}$,
- edges are $\alpha \Rightarrow^1 \beta$.

Then w is in L if w is reachable from vertex S in $\mathcal{D}$.

□

Needless to say . . .

Lemma

Not all decidable languages are context-sensitive.

Proof. Here is a cute diagonalization argument for this.

Let $(x_i)_i$ be an effective enumeration of $\Sigma^\star$ and $(G_i)_i$ an effective enumeration of all CSG over Σ (say, both in length-lex order). Set

$$L = \{\, x_i \mid x_i \notin \mathcal{L}(G_i) \,\}$$

By the lemma, L is decidable.

But L cannot be context-sensitive by the usual diagonal mumbo-jumbo.

$\square$

Example: Counting

It is well-known that the language

$$L = \{\, a^n b^n c^n \mid n \geq 1 \,\}$$

is not context free (as opposed to $a^i b^i$). On the other hand, here is a context-sensitive grammar G for L: let $V = \{S, B\}$ and set

$$\begin{aligned} S &\to aSBc \mid abc \\ cB &\to Bc \\ bB &\to bb \end{aligned}$$

A typical derivation looks like

$$S \Rightarrow a^{n+1} bc(Bc)^n \Rightarrow a^{n+1} bB^n c^{n+1} \Rightarrow a^{n+1} b^{n+1} c^{n+1}$$

It follows by induction that $L \subseteq \mathcal{L}(G)$.

Not so fast ...

Alas, we also need to show that $\mathcal{L}(G) \subseteq L$.

This is a bit harder: we need to show that the productions cannot be abused in some unintended way to generate other strings: recall that there is no restriction on the order in which productions can be applied, they just have to match.

E.g., the following is allowed:

$$S \Rightarrow aaSBcBc \Rightarrow aaSBBcc$$

Exercise

Figure out the details.

Example: More Counting

It is also known that the language

$$L = \{\, x \in \{a,b,c\}^\star \mid \#_a x = \#_b x = \#_c x \,\}$$

is not context free. But, again, it is easily context-sensitive:

let $V = \{S, A, B, C\}$ and set

$$\begin{aligned}
S &\to S' \mid \varepsilon \\
S' &\to S'ABC \mid ABC \\
XY &\to YX && \text{for all } X, Y \in \{A, B, C\} \\
A &\to a \\
B &\to b \\
C &\to c
\end{aligned}$$

Note that most productions are actually context free. The critical part is the commutation productions for $\{A, B, C\}$.

Closure Properties

Theorem

Context-sensitive languages are closed under union, concatenation, Kleene star and reversal.
They are also closed under ε-free homomorphisms.

Proof. Straightforward by manipulating the grammar. □

Note that arbitrary homomorphisms do not work in this case: they erase too much information and can force too large a search.

Two Innocent Questions

- Are CSL closed under intersection?
- Are CSL closed under complement?

The answer is Yes in both cases (so this is quite different from context-free languages).

The proof for intersection can be based on a machine model, and is much easier than the proof for complement (which requires a special and very surprising counting technique; see next lecture).

Normal Form

Theorem (Kuroda)

Every context-sensitive grammar can be written with productions of the form

$$A \to BC \qquad AB \to CD \qquad A \to a$$

The proof is very similar to the argument for Chomsky normal form for CFG (using only productions $A \to BC$ and $A \to a$).

Note that the recognition algorithm becomes particularly simple when the CSG is given in Kuroda normal form: we first get rid of all terminals and then operate only on pairs of consecutive variables.

Monotonicity

Derivations in CSGs are length-non-decreasing. Correspondingly, define a grammar to be monotonic if all productions are of the form

$$\pi : \alpha \to \beta \qquad \text{where } \alpha \in \Gamma^\star V \Gamma^\star, \beta \in \Gamma^\star, |\alpha| \leq |\beta|$$

As we have seen already, a monotonic grammar can only generate a decidable language.

In fact, these look rather similar to context-sensitive grammars except that we are now allowed to manipulate the context (but see 2 slides down). In particular, every CSG is monotonic.

Monotonic is Context-Sensitive

Theorem

A language is context-sensitive iff it is generated by a monotonic grammar.

One way to prove this is to convert monotonic productions to context-sensitive ones. As a simple example, consider a commutativity rule

$$AB \to BA$$

Here are equivalent context-sensitive rules:

$$\begin{aligned} AB &\to AX \\ AX &\to BX \\ BX &\to BA \end{aligned}$$

Here X is a new variable and the green variable is the one that is being replaced. Note how the left/right context is duly preserved.

And the Difference?

Both CFGs and CSGs produce decidable languages, but the context-free ones are much, much less complicated (recall the CYK algorithm).

Note that the Emptiness problem (is $\mathcal{L}(G) = \emptyset$?) for CFLs is solvable in linear time.

To see how, call a variable *productive* if it derives a string of terminals, $A \Rightarrow x \in \Sigma^\star$. Clearly all variables with productions $A \to x$ are productive, say, at level 0. From those we can inductively define variables productive at level 1: productions $A \to \alpha$ where α contains only terminals and productive level 0, and so on. Then G has empty language iff S fails to be productive.

Could a similar approach work for CSLs? As Kuroda normal form shows, on the face of it, the productions seem only mildly more complicated. Try, but it is really crucial here that the LHSs of context-free productions are in V.

Undecidability of CSG Emptiness

Recall that we can easily code computations of a TM as strings, producing a language of all accepting computations:

$$\mathfrak{C}(M) = \{ \# q_0 x \# C_2 \# \ldots \# C_{n-1} \# q_Y \# \mid \ldots \}$$

It is not terribly hard to construct a context-sensitive grammar for this language $\mathfrak{C}(M)$. This requires a bit of work using a grammar directly, but is fairly easy to see from the machine model (see below). Hence we have the following result.

Theorem (CSL Emptiness)

It is undecidable whether a CSG generates the empty language.

And Context-Free Languages?

One might wonder whether this kind of argument could be reorganized to produce undecidability for CFLs.

At first glance, this seems highly problematic: the copy language $\{\, ww \mid w \in \Sigma^\star \,\}$ is not context-free, and $\mathfrak{C}(M)$ is clearly worse.

A Trick:

The representation from the last slide is arguably the most natural, but there are other options: we could use alternating mirror images of configurations. For simplicity, assume that the number of steps is odd.

$$\mathfrak{C}_{\mathsf{op}}(M) = \{\, \# C_1 \# C_2^{\mathsf{op}} \# C_3 \# C_4^{\mathsf{op}} \ldots \# C_{2n}^{\mathsf{op}} \# \mid \ldots \}$$

This may seem contrived, but nobody says we have to use the most obvious representation.

So What?

Two consecutive blocks $C_i \# C_{i+1}^{\mathsf{op}}$ are almost palindromes. Hence they should be manageable by a CFG.

Alas, there is another problem: we have to check not just one pair, but all of them: for a string to be in $\mathfrak{C}_{\mathsf{op}}(M)$ all consecutive pairs have to be correct (and the first and last must be initial and accepting). To get around this problem, we use the negation: unlike correctness, having an error is a local, existential property.

Another Trick:

Consider the complement

$$\overline{\mathfrak{C}}_{\mathsf{op}}(M) = \Sigma^\star - \mathfrak{C}_{\mathsf{op}}(M)$$

all strings that do not represent an accepting computation.

So $\overline{\mathfrak{C}}_{\mathsf{op}}(M) = \Sigma^\star$ iff $\mathcal{L}(M) = \emptyset$, an undecidable property.

CFL Universality

Theorem

The complement $\overline{\mathfrak{C}}_{\mathsf{op}}(M)$ of all accepting computations is context-free. Hence it is undecidable whether a context-free language is all of $\Sigma^\star$.

Proof.

We can focus on syntactically correct strings of the form $(\#\,\Sigma^\star Q \Sigma^\star)^\star \#$, all others form a regular language that we can simply add on to the part generated by a context-free grammer (dealing with strings that look right syntactically, but have a semantics error somewhere).

We want to generate strings that contain a bad part $\#C\#C'\#$: two near-palindromes that do not represent a step in a computation.

Proof Contd

The problem is that we need to know which of the two configurations is written backwards.

Therefore we set up two CFGs G_{even} and G_{odd} that generate bad strings with at least one error between consecutive configurations:

$$C_{2i+1} \# C_{2i+2}^{\mathsf{op}} \qquad \text{or} \qquad C_{2i}^{\mathsf{op}} \# C_{2i+1}$$

Each grammar "knows" whether the first or second configuration is written backwards, so it can generate a corresponding near-palindrome with at least one error. For example, given $\delta(p, a) = (q, b, +1)$, grammar G_{even} could produce

$$\# x_\ell \ldots x_1 p a y_1 \ldots y_r \# y_r \ldots y_1 b q x_1 \ldots x_\ell \#$$

Make sure you understand why this fragment is in fact incorrect.

Proof Contd

We can then pad out the bad pair with the appropriate even/odd number of configurations to get a string representing a computation with error.

For example, grammar G_{even} must pad on the left and right by an even number of configurations.

The union of these grammars, plus the regular language for the syntactically wrong strings produces the desired context-free language of all non-computations.

$\square$

Exercise

Find another way of organizing this proof.

More?

So we have

- Recognition and Emptiness for context-free languages are decidable (in fact polynomial time).
- Universality of context-free languages is undecidable.

Anything else that could go wrong? We know CFLs are not closed under intersections, so how about the following:

Problem:	**CFL Intersection**
Instance:	Two context-free grammars G_1, G_2.
Question:	Is $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$?

CFL Intersection

Theorem

The language $\mathfrak{C}_{\mathsf{op}}(M)$ of accepting computations is the intersection of context-free languages.
Hence it is undecidable whether the intersection of two CFLs is empty.

Sketch of proof.

Since CFLs are closed under intersection by a regular language, we can easily force all strings to be of the syntactically correct form $(\# \Sigma^\star Q \Sigma^\star)^\star \#$,

To ensure semantic correctness, we can adapt the even/odd approach: in L_1, all the even/odd pairs are correct, in L_2 the odd/even pairs. This works since we are dealing with repeated near-palindromes.

Then $\mathfrak{C}_{\mathsf{op}}(M) = L_1 \cap L_2$.

$\square$

Quotients

Recall that for two languages $K, L \subseteq \Sigma^\star$ their quotient is defined by

$$L^{-1}K = \{\, v \in \Sigma^\star \mid \exists\, u \in L\, (uv \in K) \,\}$$

Corollary

It is undecidable whether $L^{-1}K$ is empty for two CFLs L and K.

Sketch of proof.

This is another application of the even/odd method.

Consider only computations on the empty tape, so $C_1 = q_0$. L will check the even/odd pairs, and K the odd/even pairs.

Then $L^{-1}K \neq \emptyset$ iff M accepts the empty tape.

$\square$

Decidability Overview

	$x \in L$	$L = \emptyset$	$L = \Sigma^\star$	$L = K$	$L \cap K = \emptyset$
regular	Y	Y	Y	Y	Y
DCFL	Y	Y	Y	Y	N
CFL	Y	Y	N	N	N
CSL	Y	N	N	N	N
decidable	Y	N	N	N	N
semidec.	N	N	N	N	N

Standard decision problems for various language classes.

Needless to say, for the decidable ones we would like a more precise complexity classification (in which case it may matter how precisely the instance is given).

Formal Grammar vs. CSG

Note that one can “simulate” an arbitrary formal grammar G with a semidecidable language by a CSG G' as follows: let S' be the new start symbol, new terminals b and $\#$, and add productions

$$\begin{aligned} S' &\to \#S & & \\ \alpha &\to \beta & &\text{if } |\alpha| \le |\beta| \\ \alpha &\to \beta b^r & &\text{if } 0 < r = |\alpha| - |\beta| \\ xb &\to bx & & x \in \Gamma, x = \# \end{aligned}$$

Then G derives w iff G' derives $b^r\#w$ for some r.

By projection, we get back the original language, but padding by some bs renders things decidable.

A Machine Model?

In order to find a parser for CSL it seems natural to look for an associated machine models:

- semidecidable — Turing machines
- context free — pushdown automata
- regular — finite state machines

We need a machine model that is stronger than pushdown automata (FSM plus stack), but significantly weaker than full Turing machines.

Note that two stacks are equivalent to TMs, so we need a different approach.

Backwards Derivation

Suppose L is context-sensitive via G. The idea is to run a derivation of G backwards, starting at a string x of terminals.

To this end, nondeterministically guess a handle in the current string, a place where there is a substring of the form β, where $\alpha \to \beta$ is a production in G. Erase β and replace it by α. Rinse and repeat.

The original string x is in L iff we can ultimately reach S this way.

Of course, this is yet another path existence problem in a suitable digraph.

Linear Bounded Automata

Definition

A linear bounded automaton (LBA) is a type of one-tape, nondeterministic Turing machine acceptor where the input is written between special end-markers and the computation can never leave the space between these markers (nor overwrite them).

Thus the initial configuration looks like

$$\# q_0 x_1 x_2 \dots x_n \#$$

and the tape head can never leave this part of the tape.

It may seem that there is not enough space to perform any interesting computations on an LBA, but note that we can use a sufficiently large tape alphabet to “compress” the input to a fraction of its original size and make room.

The Myhill-Landweber-Kuroda Theorem

The development happened in stages:

- Myhill 1960 considered deterministic LBAs.
- Landweber 1963 showed that they produce only context-sensitive languages.
- Kuroda 1964 generalized to nondeterministic LBAs and showed that this produces precisely all the context-sensitive languages.

Theorem

A language is accepted by a (nondeterministic) LBA iff it is context-sensitive.

Proof 1

Here is the argument for a monotonic grammar, dealing with CSGs is entirely similar.

Initially, some terminal string $a_1, a_2, \ldots, a_n$ is written on the tape. Then repeat the following steps:

- Search handle: the head moves to the right a random number of places, say, to a_i.
- Check righthand side: the machine verifies that $a_i, \ldots, a_j \; = \; \beta$ where $\alpha \to \beta$ is a production.
- Replace by left-hand side: the block $a_i, \ldots, a_j$ is replaced by α, possibly leaving some blanks.
- Collect: remove possible blanks by shifting the rest of the tape left.

This loop repeats until the tape is reduced to S and we accept. If any of the guesses goes wrong we reject.

Proof 2

Also note that nondeterminism is critical here: grammars are naturally nondeterministic and a deterministic machine would have to search through all possible choices. That seems to require more than just linear space (open problem, see below).

For the opposite direction it is a labor of love to check in great gory detail that the workings of an LBA can be described by a context-sensitive grammar. The critical point here is that the LBA expand/shrink the tape, it just changes the inscription between the endmarkers.

Digression: Physical Realizability II

It was recognized already in the 1950s that Turing machines are, in many ways, too general to describe anything resembling the type of computation that was possible on emerging digital computers.

The Rabin-Scott finite state machine paper was one forceful attempt to impose a radical constraint on Turing machines that brings them into the realm of “feasible” computation.

Myhill’s introduction of LBA is another attempt at constructive restrictions. As we now know, LBAs are still a rather generous interpretation of the notion of feasible computation; real, practical algorithms need further constraints.

Still, it is a perfectly good model and there are many interesting problems that fit perfectly into this framework.

Intersection Closure

Theorem

CSL are closed under intersection.

Proof.

Given two LBAs M_i for L_i. we can construct a new LBA for $L_1 \cap L_2$ by using a 2-track tape alphabet $\Gamma = \Sigma \times \Sigma$.

The upper track is used to simulate M_1, the lower track is used to simulate M_2.

It is easy to check that the simulating machine is again a LBA (it will sweep back and forth over the whole tape, updating both tracks by one step on the way).

$\square$

Why Not CFG?

In essence, the argument says that we can combine two LBAs into a single one that checks for intersection. This is entirely similar to the argument for FSMs.

Burning Question:
Why can't we do the same for PDAs?

Because we cannot in general combine two stacks into a single one (though this works in some cases; the stack height differences need to be bounded). But in general, two stacks suffice to simulate a Turing machine.

More Problems

Context-sensitive recognition is naturally[†] in $\mathrm{NSPACE}(n)$, and by Savitch, in $\mathrm{SPACE}(n^2)$.

Two natural questions:

- Are there other natural decision problems of this kind?
- Is there any reason to consider larger space classes, say, $\mathrm{SPACE}(n^{42})$?

[†]We can simply copy the input to the work tape.

Quantified Boolean Formulae

Definition

A quantified Boolean formula (QBF) is a formula consisting of propositional connectives "and," "or" and "not," as well as existential and universal quantifiers.

If all variables in a QBF are bounded by a quantifier, then the formula has a truth value: we can simply expand it out as in

$$\exists\, x\, \varphi(x) \mapsto \varphi(0) \lor \varphi(1)$$

$$\forall\, x\, \varphi(x) \mapsto \varphi(0) \land \varphi(1)$$

In the end we are left with a propositional formula without variables which can simply be evaluated in time linear in its size. Of course, that size is exponential in the size of the original one.

The QBF is a succinct description of the expanded one.

Validity of QBF

Problem:	**Validity of Quantified Boolean Formulae (QBF)**
Instance:	A quantified Boolean sentence φ.
Question:	Is φ valid?

Note that many properties of propositional formulae can easily be expressed in terms of QBF. For example, $\varphi(x_1, x_2, \ldots, x_n)$ is satisfiable iff

$$\exists\, x_1, \ldots, x_n\, \varphi(x_1, x_2, \ldots, x_n) \text{ is valid}$$

Likewise, the formula is a tautology iff

$$\forall\, x_1, \ldots, x_n\, \varphi(x_1, x_2, \ldots, x_n) \text{ is valid}$$

QBF

Lemma

Validity of QBF can be checked by a deterministic LBA.

Proof. As a case in point, consider the formula

$$\forall\, x_1, \ldots, x_n\, \exists\, y_1, \ldots, y_m\, \varphi(\boldsymbol{x}, \boldsymbol{y})$$

The argument easily extends to any other formula.

To check validity we use two loops, one counting from 0 to $2^n - 1$ for $\boldsymbol{x}$ and another counting from 0 to $2^m - 1$ for $\boldsymbol{y}$.

Validity Tester

```
foreach $x = 0, \dots, 2^{n-1}$ do
    $v = 0$
    foreach $y = 0, \dots, 2^{m-1}$ do
        if $\varphi(x, y)$
        then $v = 1$; break
    if $v = 0$
    then return No
return Yes
```

Of course, the running time is exponential, but linear space is quite enough.

□

QBF Solvers

http://satisfiability.org/

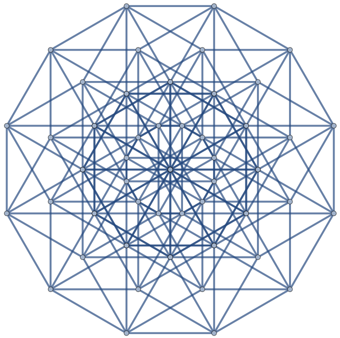
http://www.qbflib.org/

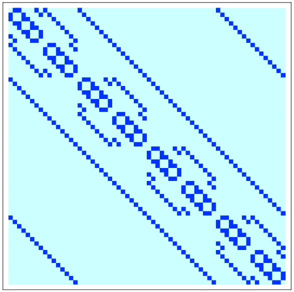
There is a lot of current research on building powerful QBF solvers.

Succinct Representations

A quantified Boolean formula can be construed as a succinct representation of a much larger ordinary Boolean formula: deciding validity becomes harder because the input is much shorter.

Succinct representations appear in many places:





Succinct Hypercube

- Vertex set: $\mathbf{2}^d$
- Edge set: $(x, y) \in \mathbf{2}^d \times \mathbf{2}^d$ such that $\text{dist}(x, y) = 1$

Here $\text{dist}(x, y)$ refers to the Hamming distance: $|\{\, i \mid x_i \neq y_i \,\}|$.

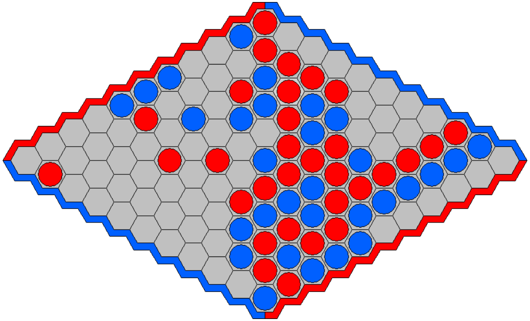
More generally, we could use a Boolean formula $\Phi(x, y)$ to determine the edges in a graph $G(\mathbf{2}^d, \Phi)$ on $\mathbf{2}^d$.

And the formula could be represented by a ROBDD.

Games

Many games such as Hex, Checkers, Chess, . . . are naturally finite (at least with the right set of rules). Thus, from the complexity perspective, they are trivial.

However, they often generalize to an infinite class of games. For example, Hex can obviously be played on any $n \times n$ board.



Incidentally, John Nash proved that there is no draw in Hex (in 1949).

Winning

There are several natural questions one can ask about these generalized games:

- Is a particular board position winning?
- Does the first player have a winning strategy?

If the total number of moves in a game is polynomial in the size of the board (Checkers, Hex, Reversi), then these questions are typically in PSPACE: we can systematically explore the game tree to check if a position is winning.

Integer Circuit Evaluation

Consider a circuit with n integer inputs; we think of $x \in \mathbb{Z}$ as the singleton $\{x\}$.

Internal nodes all have indegree 2 (but unbounded outdegree), and come in three types:

Union	$A \cup B$
Sum	$A \oplus B = \{\, a + b \mid a \in A, b \in B \,\}$
Product	$A \otimes B = \{\, a \cdot b \mid a \in A, b \in B \,\}$

We assume there is exactly one output node of outdegree 0, the subset of $\mathbb{Z}$ appearing there is the result of evaluating the circuit.

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

PSPACE

$A \cup B$ is linear; we might also admit arbitrary finite sets as input.

But: $A \oplus B$ and $A \otimes B$ are both potentially quadratic, so the sets defined by an ICE can be exponentially large.

Lemma

ICE is in PSPACE*.*

Proof. We cannot simply evaluate the circuit in polynomial space: the sets might be too large.

But we can nondeterministically guess the one element in each set that is actually needed to obtain the target value a in the end, and follow these witnesses throughout the circuit.

$\square$

Comment

This result may seem a bit contrived, but it actually is quite natural: there is a long-standing question of how circuit evaluation compares to formula evaluation (it should be slightly harder if the fan-out is higher than 1).

For example, Boolean circuit evaluation is $\mathbb{P}$-complete, but evaluating a Boolean formula is in NC^1 (presumably a smaller class based on circuits, more later).

More generally, there are interesting connections between various kinds of circuits (or straight-line programs) and small complexity classes.

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 Contd

For hardness one uses the fact that a configuration of a Turing machine can be expressed by a (large) collection C 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

$$\Phi_k(C_1, C_2) \iff \exists\, t \leq 2^k\, (C_1' \vdash_M^t C_2').$$

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

No Cigar

Unfortunately, this direct approach causes an exponential blow-up in size: Φ_k is about twice as large as Φ_{k-1}. Therefore we use a trick:

$$\Phi_k(C_1, C_2) \iff \exists\, C\, \forall\, D_1, D_2 \Big((C_1, C) = (D_1, D_2) \lor \\ (C, C_2) = (D_1, D_2) \Rightarrow \Phi_{k-1}(D_1, D_2) \Big)$$

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

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

This is logically equivalent to the obvious formula from the last slide, but it gets us around the problem of having to write down Φ_{k-1} twice.

$\square$

Context Sensitive Recognition

We have seen that for context-sensitive grammars it is decidable whether a word is generated by the grammar (and that's about all that is decidable). On the other hand, non-emptiness is already undecidable, so one might expect that recognition is difficult.

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 $\mathcal{L}(G)$?

Note that this problem is in $\mathrm{NSPACE}(n)$, near the apparent bottom of PSPACE.

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.

$\square$

PSPACE and FSMs

Here is a question regarding the intersection of a family of DFAs.

Problem:	**DFA Intersection**
Instance:	A list $\mathcal{A}_1, \ldots, \mathcal{A}_m$ of DFAs.
Question:	Is $\bigcap \mathcal{L}(\mathcal{A}_i)$ empty?

Note that m, the number of machines, is not fixed. We can check Emptiness in linear time on the accessible part of the product machine $\mathcal{A} = \prod \mathcal{A}_i$; alas, the latter has exponential size in general.

Theorem (Kozen 1977)

The DFA Intersection Problem is PSPACE*-complete.*

The Proof

The problem is in linear NSPACE as follows.

Let $n_i = |\mathcal{A}_i|$ and set $n = \prod n_i$.

```
set $p_i = \text{init}(\mathcal{A}_i)$
for $i = 1..n$ do
    guess $a \in \Sigma$
    compute $p_i = \delta_i(p_i, a)$
    if $p_i \in F_i$ for all $i$
    then return Yes
return No
```

As usual, this "algorithm" may produce false No's, but a Yes guarantees a yes-instance.

For hardness, let T be a Turing machine with polynomial space bound $p(n) \geq n$. As usual, encode configurations as strings

$$C = a_\ell \dots a_2 a_1 \, p \, b_1 b_2 \dots b_r$$

and a whole computation as a string

$$\#C_1\#C_2\dots\#C_m\$$$

$\#$ and $\$$ are new markers and we may safely assume that the length of each configuration is exactly $N = p(n) + 2$: pad with blank tape symbols and make sure there is at least one blank never used at either end. Also, assume that m is odd (modify the TM if necessary).

The idea is similar to the even/odd approach for CSLs: we construct a family of DFAs that make sure that $C_{2i+1} \mapsto C_{2i+2}$ is correct, and another family that checks $C_{2i} \mapsto C_{2i+1}$. We will only deal with the first case.

Here is machine $\mathcal{A}_k$, $0 \leq k \leq N-3$, that checks $\#C_{2i+1}\#C_{2i+2}$.

- It reads $\#$ and skips k letters.
- It remembers the next 3 letters x, y and z.
- It skips $N-k-3$ letters, reads a $\#$ and skips k more letters.
- Then $\mathcal{A}_k$ makes sure that the next three letters x', y', z' are compatible with the transition function of the Turing machine.
- It skips another $N-k-3$ letters. If it reads a $\#$ it starts all over; othe-wise, it reads $\$$ and accepts.

More precisely, if $xyz = apb$ and $\delta(p,b) = (c,q,+1)$, then $x'y'z' = acq$. For $\delta(p,b) = (c,q,-1)$ we have $x'y'z' = qac$.

If $y \notin Q$ we simply skip forward, without any checks.

Combining "odd" $\mathcal{A}_k$ machines with corresponding "even" machines insures that every accepted string is indeed a computation: the state has is in some position $k+2$, so machine $\mathcal{A}_k$ (or its counterpart) checks that the next configuration is proper. The other machines ignore this particular configuration, and the next.

Lastly, we add one more DFA that checks that the sequence of configurations starts and ends at the proper initial and final configuration.

All the DFAs together accept precisely one string, coding the accepting computation of M, iff the polynomial space machine M accepts its input.

$\square$

Determinization

We can think of a nondeterministic automaton $\mathcal{A}$ as a succinct representation of an equivalent deterministic automaton $\mathcal{B}$: the accessible part of the power automaton as detailed in the standard Rabin-Scott construction.

The size of $\mathcal{B}$ has no better than an exponential bound

$$|\mathcal{B}| \leq 2^{|\mathcal{A}|}$$

Unfortunately, this bound is tight in general. It remains tight when one considers only semiautomata where $Q = F = I$, even when they are nearly deterministic and co-deterministic.

Blow-Up Example 1

Consider the following DFA $\mathcal{A} = \langle [n], \Sigma, \delta, 1, \{1\} \rangle$ where $\Sigma = \{a, b, c\}$ and the transition function is given by

δ_a	a cyclic shift on $[n]$,
δ_b	the transposition that interchanges 1 and 2,
δ_c	sends 1 to 2, identity elsewhere.

δ_a and δ_b together generate the whole symmetric group on n symbols, adding δ_c produces all maps $[n] \to [n]$ (in other words, the transition semigroup of $\mathcal{A}$ has maximal size n^n).

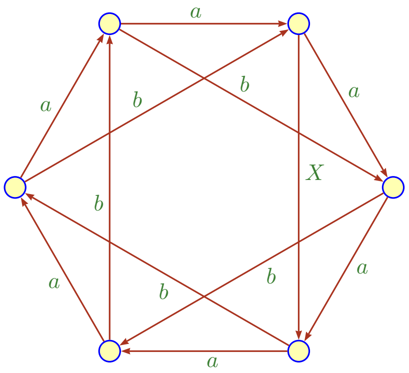
Furthermore, the reversal rev$(\mathcal{A})$ of $\mathcal{A}$ produces a power automaton of maximal size 2^n, and the power automaton turns out to be minimal.

Blow-Up Example 2

Here is a 6-state semiautomaton, based on a circulant with strides 1 and 2.

$X = b$: machine is deterministic and co-deterministic, so the power automaton has size 1.

$X = a$: the power automaton has maximal size 2^6.



Predicting Blowup

It would be nice to have a cheap test whether a nondeterministic finite state machine can be determinized without too many states. Alas ...

Problem:	**Power Automaton Size**
Instance:	A nondeterministic automaton $\mathcal{A}$, a bound β.
Question:	Is the size of the power automaton of $\mathcal{A}$ at least β?

This problem can be handled by an LBA:

- guess β-many subsets $P \subseteq Q$, say, in lexicographic order, and
- for each, guess a string $x \in \Sigma^\star$ and verify that $\delta(I, x) = P$.

No Luck

This is where the story ends: there is essentially no other way to handle this problem than to just rely on brute-force enumeration.

Theorem (KS 2003)

Power Automaton Size is PSPACE*-complete.*

It is also hard to determine whether a particular subset $P \subseteq Q$ appears in the power automaton.

And More

Many of the generalized games from above turn out to be PSPACE-complete.

ICE is PSPACE-complete.

Testing whether two regular expressions are equivalent is PSPACE-complete (even if one of them is $\Sigma^{\star}$). The same holds true for nondeterministic finite state machines.