

# UCT

## Polynomial Space

KLAUS SUTNER

CARNEGIE MELLON UNIVERSITY

SPRING 2024



**1 Context Sensitive Languages**

**2 Linear Bounded Automata**

**3 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  $\text{NSPACE}(n)$  (and, by Savitch, in  $\text{SPACE}(n^2)$ ).
- Somewhat surprisingly, it's just a small step from there to all of PSPACE.

## Definition (CSG)

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

$$\alpha A \beta \rightarrow \alpha \gamma \beta \quad \text{where } \gamma \neq \varepsilon$$

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

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

Hence, no CSG can generate the empty word. To avoid this nuisance, some authors also allow  $S \rightarrow \varepsilon$  in which case  $S$  may not appear on the right hand side of any production.

## 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  $\mathcal{D}$ .

□

... shows that one can build context-sensitive grammars for very complicated languages. In fact, as long as a language is decidable, it looks like one can build a corresponding CSG.

**Question:**

Is every decidable language context-sensitive?

## Lemma

*Not all decidable languages are context-sensitive.*

*Proof.* Here is a cute diagonalization proof that looks like Russel's paradox.

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

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

By the last lemma,  $L$  is decidable, uniformly in  $i$ .

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

□

It is well-known that the language

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

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

$$S \rightarrow aSBc \mid abc$$

$$cB \rightarrow Bc$$

$$bB \rightarrow bb$$

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  $L \subseteq \mathcal{L}(G)$ .



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 perfectly allowed:

$$S \Rightarrow aaSBcBc \Rightarrow aaSBBcc$$

In the end, though, we need a string of terminals, and that will force us back into the language.

### Exercise

*Figure out the details.*

The production

$$cB \rightarrow Bc$$

is **not** context-sensitive.

Rather, it is a so-called **monotonic production**. We will show shortly that this is not a problem, we can translate monotonic productions into context-sensitive ones.

The given form is just much, much easier to read.

It is also known that the language

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

is not context free. But, again, it is easily context-sensitive: let  $V = \{S, A, B, C\}$  and set

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

Note that almost all productions are actually context free. The only exception are the critical commutation productions for  $A, B, C$ .

## Theorem

*Context-sensitive languages are closed under union, concatenation, Kleene star and reversal.*

*They are also closed under  $\epsilon$ -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. This corresponds directly to the length condition on the productions of a CSG.

- 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).

## Theorem (Kuroda)

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

$$A \rightarrow BC \quad AB \rightarrow CD \quad A \rightarrow a$$

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

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.

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

$$\pi : \alpha \rightarrow \beta \quad \text{where } \alpha \in \Gamma^* V \Gamma^*, \beta \in \Gamma^*, |\alpha| \leq |\beta|$$

The condition that  $\alpha$  must contain a syntactic variable is somewhat weird: it precludes replacements of strings made up of terminals only. One can also study general **rewrite systems** without any restrictions.

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. Clearly, every CSG is monotonic.

## 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

$$AB \rightarrow BA$$

Here are equivalent context-sensitive rules using a new variable  $X = X_{AB}$ :

$$AB \rightarrow AX$$

$$AX \rightarrow BX$$

$$BX \rightarrow BA$$

The green variable is the one that is being replaced. Note how the left/right context is duly preserved.



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

### Lemma

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

To see how, call a variable *productive* if it can derive a string of terminals,  $A \Rightarrow x \in \Sigma^*$ . Clearly all variables with productions  $A \rightarrow x$  are productive, say, at level 0. From those we can inductively define variables productive at level 1: productions  $A \rightarrow \alpha$  where  $\alpha$  contains only terminals and variables productive at 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  $V$ .

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

$$\mathfrak{C}(M) = \{ \# C_x^{\text{init}} \# C_2 \# \dots \# C_{n-1} \# C_Y^{\text{halt}} \# \mid \dots \}$$

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).

### Theorem (CSL Emptiness)

*It is undecidable whether a CSG generates the empty language.*

*Proof.*  $\mathfrak{C}(M) = \emptyset$  iff  $M$  halts on no input, a  $\Pi_1$ -complete problem. □

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

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

### Trick One:

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.

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

This may seem contrived, but nobody says we have to use the “canonical” representation<sup>†</sup>.

---

<sup>†</sup>In fact, exactly what is canonical lies in the eye of the beholder.

Two consecutive blocks  $C_i \# C_{i+1}^{\text{op}}$  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  $\mathfrak{C}_{\text{op}}(M)$ , 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.

### Trick Two:

Consider the complement

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

all strings that do not represent an accepting computation.

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

## Theorem

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

*Proof.*

We can focus on syntactically correct strings of the form  $(\# \Sigma^* Q \Sigma^*)^* \#$ , all others form a regular language that we can simply add on to the part generated by a context-free grammar (we are 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.

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

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

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

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  $G_{\text{even}}$  could produce the error

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

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

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  $G_{\text{even}}$  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.



### Exercise

*Determine in greater detail what the even/odd grammars need to do.*

*Then find another way of organizing this proof by using yet another representation.*

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  $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$ ?



## Theorem

*The language  $\mathfrak{C}_{\text{op}}(M)$  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  $(\# \Sigma^* Q \Sigma^*)^* \#$ ,

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

□

	$x \in L$	$L = \emptyset$	$L = \Sigma^*$	$L = K$	$L \cap K = \emptyset$
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).

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

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

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

By projection, we get back the original language, but padding by the appropriate number of  $b$ s renders things decidable.

1 Context Sensitive Languages

2 **Linear Bounded Automata**

3 Polynomial Space

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.

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  $\beta$ , where  $\alpha \rightarrow \beta$  is a production in  $G$ . Erase  $\beta$  and replace it by  $\alpha$ . 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.

## 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

$$\#q_0x_1x_2 \dots x_n\#$$

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 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.*



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

Initially, some terminal string  $a_1, a_2, \dots, a_n$  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  $a_i, \dots, a_j = \beta$  where  $\alpha \rightarrow \beta$  is a production.
- Replace by left-hand side: the block  $a_i, \dots, a_j$  is replaced by  $\alpha$ , 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.

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 cannot expand or shrink the tape, it just changes the inscription between the endmarkers.

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.

## 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).

□

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.

1 **Context Sensitive Languages**

2 **Linear Bounded Automata**

3 **Polynomial Space**

Context-sensitive recognition is naturally<sup>†</sup> in  $\text{NSPACE}(n)$ , and by Savitch, in  $\text{SPACE}(n^2)$ .

Two natural questions:

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

---

<sup>†</sup>We can simply copy the input to the work tape.

## 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

$$\exists x \varphi(x) \mapsto \varphi(0) \vee \varphi(1)$$

$$\forall x \varphi(x) \mapsto \varphi(0) \wedge \varphi(1)$$

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.



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

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

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

Likewise, the formula is a tautology iff

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

## Lemma

*Validity of QBF can be checked by a deterministic LBA.*

*Proof.* As a case in point, consider the formula

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

The argument easily extends to any other formula.

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

```
foreach  $x = 0, \dots, 2^n - 1$  do  
   $b = 0$   
  foreach  $y = 0, \dots, 2^m - 1$  do  
    if  $\varphi(x, y)$  is true  
      then  $b = 1$   
  if  $b = 0$   
    then return No  
return Yes
```

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



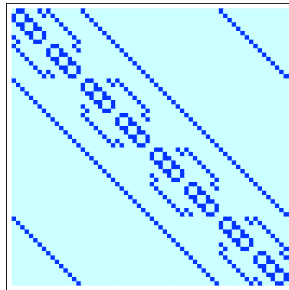
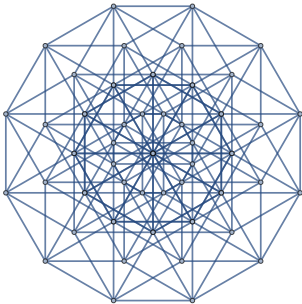
<http://satisfiability.org/>

<http://www.qbflib.org/>

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

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:



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

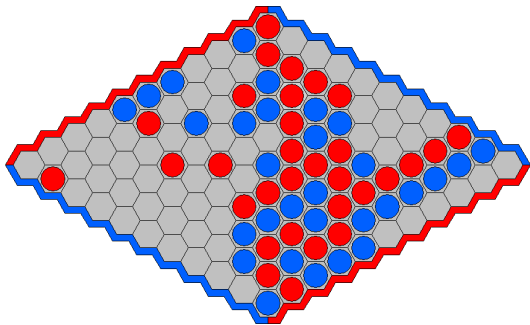
Here  $\text{dist}(x, y)$  refers to the Hamming distance:  $|\{i \mid x_i \neq y_i\}|$ .

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

And, there are clever ways to represent a Boolean formula such as circuits or ROBDDs.

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).

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.



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?

$A \cup B$  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.*

*Sketch of 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.

□

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.

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

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

Because of the padding, we can easily check membership in  $K$  in linear space. And the standard reduction shows that the problem is hard.

Clearly correct, but not too interesting. Here is a much better problem.

## 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.

The argument is exactly the same in the general case as the  $II_2$  case a few slides ago.

This method is exponential time but linear space.

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  $\Phi_k$  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

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

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

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

This quantified formula is only slightly larger than  $\Phi_{k-1}$  and can be built in polynomial time. On the other hand, it is logically equivalent to the original one from the last slide.

So we have a polynomial time reduction, as required.

□

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  $x \in \Sigma^*$ .  
Question: Is  $x$  in  $\mathcal{L}(G)$ ?

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



## Theorem

*Context Sensitive Recognition is PSPACE-complete.*

*Proof.*

For hardness use the following well-known fact from language theory:  
 $\varepsilon \notin L \in \text{NSPACE}(n)$  implies  $L$  is context-sensitive (via monotonic grammars).

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

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

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  $f(x) = (G, 1^{p(|x|)} \# x)$  is a polynomial time reduction from  $L$  to CSR.

□

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

Problem: **DFA Intersection**  
Instance: A list  $\mathcal{A}_1, \dots, \mathcal{A}_m$  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 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; otherwise, 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 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.

□

Many of the generalized games from above turn out to be PSPACE-complete. The proofs rely heavily on clever gadgets and can be quite difficult.

ICE is PSPACE-complete. So arithmetic circuits are difficult.

Testing whether two regular expressions are equivalent is PSPACE-complete (even if one of them is  $\Sigma^*$ ). The same holds true for nondeterministic finite state machines. But **not** for DFAs, those can be checked in near linear time (using Union-Find).