Universal Generation for Optimality Theory Is PSPACE-Complete

Sophie Hao
Center for Data Science
New York University
sophie.hao@nyu.edu

This article shows that the universal generation problem for Optimality Theory (OT) is PSPACE-complete. While prior work has shown that universal generation is at least NP-hard and at most EXPSPACE-hard, our results place universal generation in between those two classes, assuming that NP ≠ PSPACE. We additionally show that when the number of constraints is bounded in advance, universal generation is at least NL-hard and at most NP^{NP}-hard. Our proofs rely on a close connection between OT and the intersection non-emptiness problem for finite automata, which is PSPACE-complete in general and NL-complete when the number of automata is bounded. Our analysis shows that constraint interaction is the main contributor to the complexity of OT: The ability to factor transformations into simple, interacting constraints allows OT to furnish compact descriptions of intricate phonological phenomena.

1. Introduction

Optimality Theory (OT; Prince and Smolensky 1993, 2004) is a constraint-based formalism for describing mappings between strings. Its primary application lies in theoretical phonology, where it has been used to explain the relationship between the underlying representations of linguistic utterances (URs) and their surface representations (SRs). According to OT phonology, SRs are the result of mutations applied to URs that remove patterns deemed undesirable, or **marked**, by the grammar. An OT grammar consists of a set of **markedness constraints** that identify the marked patterns to be removed, along with a set of **faithfulness constraints** that require SRs to resemble the original URs as much as possible. The constraints are **gradient** in the sense that some UR–SR pairs may violate a constraint more than others, and they are **ranked** in the sense that some constraints are more important than others. Each UR is mapped to the potential SRs that violate the constraints the least, with higher-ranking constraints taking priority over lower-ranking ones.

Like many approaches in social and behavioral sciences, OT casts the pronunciation of utterances as a constrained optimization problem. Unlike rule-based treatments of

phonological mappings (Chomsky and Halle 1968; Johnson 1970, 1972; Kaplan and Kay 1994), the OT framework does not provide any obvious algorithm for generating SRs from URs. For this reason, the computational complexity of optimization has been a topic of substantial interest in the formal analysis of OT. Prior work has shown that the **universal generation problem for OT** (Heinz, Kobele, and Riggle 2009), where an algorithm must generate an SR given a UR and a list of ranked constraints as input, is NP-hard in the total size of the constraints (Eisner 1997, 2000b; Wareham 1998; Idsardi 2006), making it at least as hard as typical combinatorial optimization problems such as the traveling salesperson problem. Assuming that $P \neq NP$, this result is commonly interpreted to show that any algorithm that solves the universal generation problem must require intensive resources, at least in the worst case. In practice, most implementations of OT (e.g., Ellison 1994; Eisner 1997, 2000a; Riggle 2004; Gerdemann and Hulden 2012) utilize exponential time and space, since they involve representing constraints as finite-state machines and intersecting them using the construction of Rabin and Scott (1959).

This paper establishes a tight characterization of the complexity of universal generation. We show that, using the most general formulation of OT in the literature (Riggle 2004), universal generation can be carried out using polynomial space, and that verifying the correctness of an SR is complete in the class of polynomial-space computable decision problems. To be precise, this article proves the following two theorems.

Theorem 1
The problem of deciding whether an SR y optimally satisfies a list of ranked, finite-state constraints $\langle C_1, C_2, \ldots, C_n \rangle$ for a UR x is PSPACE-complete.

Theorem 2
There is a polynomial-space algorithm that takes a UR x and a list of ranked, finite-state constraints $\langle C_1, C_2, \ldots, C_n \rangle$ and outputs an SR y that optimally satisfies $\langle C_1, C_2, \ldots, C_n \rangle$. (That is, the relation that associates URs and constraint lists with optimal SRs is in FPSPACE.)

Whereas prior work shows that universal generation is at least NP-hard and at most EXPSPACE-hard, our result places universal generation *in between* those two complexity classes, assuming that NP $\neq$ PSPACE. To establish inclusion in PSPACE, we show that the automaton-intersection-based techniques used in OT implementations can be executed without writing down the intersected automaton or the SR in memory. To establish PSPACE-hardness, we show that the intersection non-emptiness problem for finite-state automata, a PSPACE-complete problem (Kozen 1977), can be reduced to universal generation for an OT grammar with only markedness constraints. In addition to these main results, we also show that universal generation is at least NL-hard and at most NP^{NP}-hard when the number of constraints in the grammar is fixed a priori.

The techniques and algorithms featured in our proofs identify several features of OT that contribute to the complexity of universal generation. These features include the ability of OT to generate exponentially long SRs, the ability of constraints to assign exponentially large violation numbers, and the logical complexity of the concept of optimization. By far the most significant contributor to computational complexity, however, is the ability of OT to produce concise explanations of phonological phenomena, where intricate UR–SR transformations are factored into simple but conflicting requirements on well-formedness and communicative transparency. Our analyses show that PSPACE-complete complexity is the price paid by OT in exchange for this theoretical elegance.

2. Preliminaries

We begin by introducing notation and reviewing definitions from automata theory and complexity theory. Although we state definitions and theorems directly relevant to this paper, we assume familiarity with basic concepts such as finite-state machines, Turing machines, and time and space complexity. For readers less familiar with these concepts, an accessible introduction is provided by Sipser (2013).

Let Σ and Γ denote finite alphabets. For an alphabet Σ, Σ^* is the set of strings over Σ. The **length** of a string $x \in \Sigma^*$ is denoted by $|x|$, and the **empty string** ε the unique string of length 0. Σ_ε is the set $\Sigma \cup \{\varepsilon\}$. We refer to subsets of Σ^* as **languages**. If ϕ and ψ are symbols, strings, or languages, then $\phi\psi$ is the (elementwise) concatenation of ϕ with ψ, ϕ^k is the concatenation of k copies of ϕ, and ϕ^* is the closure of ϕ under concatenation. When appropriate, we identify alphabet symbols with strings of length 1, and individual strings with singleton languages.

We say that a set A is a **monoid under** $\star$ if $\star$ is a binary operation on A such that

- A is closed under $\star$ (i.e., for all $a, b \in A$, $a \star b \in A$);

- $\star$ is associative (i.e., for all $a, b, c \in A$, $(a \star b) \star c = a \star (b \star c)$); and

- $\star$ has an identity element $e \in A$ (i.e., there exists $e \in A$ such that $a \star e = e \star a = a$ for all $a \in A$).

We consider two kinds of monoids in this paper. For an alphabet Σ, the **free monoid over** Σ is the monoid Σ^* under concatenation; and the **natural numbers** are the monoid $\mathbb{N}$ under addition, where $\mathbb{N}$ is the set of non-negative integers.

2.1 Automata Theory

In this article, we deal with two kinds of finite-state machines: finite-state automata and finite-state transducers. We use the following definitions for these machines. We assume that all machines are deterministic.

Definition 1 (Automata)
A **deterministic finite-state automaton** (DFA) is a tuple $M = \langle Q, \Sigma, q_0, F, \rightarrow \rangle$, where

- Q is the finite set of **states**;

- Σ is the **input alphabet**;

- $q_0 \in Q$ is the **start state**;

- $F \subseteq Q$ is the set of **accept states**; and

- $\rightarrow : Q \times \Sigma \rightarrow Q$ is the **transition function**.

We write $q \xrightarrow{a} r$ to mean that $\rightarrow(q, a) = r$. For $x = x_1 x_2 \ldots x_n \in \Sigma^*$, we say that M **accepts** x and write $x \in M$ if there exist states $q_1, q_2, \ldots, q_n \in Q$, with $q_n \in F$, such that

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \xrightarrow{x_3} \ldots \xrightarrow{x_{n-1}} q_{n-1} \xrightarrow{x_n} q_n$$

Otherwise, we say that M **rejects** x, and write $x \notin M$. We identify M with the set of strings accepted by M. We say that a language is **regular** if it is accepted by a DFA. A set of numbers $A \subseteq \mathbb{N}$ is **regular** if the language $\{a^i \mid i \in A\} \subseteq a^*$ is regular.

We assume that finite-state transducers take strings as input, but may produce output from an arbitrary monoid. Although our transducers are deterministic, we assume that each input is padded with an implicit end-of-string marker, giving the transducer an opportunity to produce output after the entire input has been read. We do not allow our transducers to reject inputs: they must produce an output for *every* possible input.

Definition 2 (Transducers)
A **subsequential finite-state transducer** (SFST) is a tuple $T = \langle Q, A, B, q_0, \rightarrow, \# \rangle$, where

- Q is the finite set of **states**;

- A, the **input monoid**, is the free monoid over some alphabet Σ;

- B, the **output monoid**, is a monoid under some operation $\star$;

- $q_0 \in Q$ is the **start state**;

- $\rightarrow : Q \times \Sigma \rightarrow B \times Q$ is the **intermediate transition function**; and

- $\# : Q \rightarrow B$ is the **final transition function**.

We write $q \xrightarrow{a}_b r$ to mean that $\rightarrow(q, a) = \langle b, r \rangle$, and we may optionally write $q \rightarrow_b \#$ to mean that $\#(q) = b$. For $x = x_1 x_2 \ldots x_n \in A = \Sigma^*$, we say that T **outputs** y **on input** x and write $T(x) = y$ if there exist states $q_1, q_2, \ldots, q_n$ and elements $y_1, y_2, \ldots, y_{n+1} \in B$ such that

$$q_0 \xrightarrow[y_1]{x_1} q_1 \xrightarrow[y_2]{x_2} q_2 \xrightarrow[y_3]{x_3} \cdots \xrightarrow[y_{n-1}]{x_{n-1}} q_{n-1} \xrightarrow[y_n]{x_n} q_n \xrightarrow[y_{n+1}]{} \#$$

and $y = y_1 \star y_2 \star \cdots \star y_{n+1}$. We identify T with the function mapping strings $x \in \Sigma^*$ to outputs $T(x) \in B$. We say that a function is **subsequential** if it is computed by an SFST.

When $T = \langle T_1, T_2, \ldots, T_n \rangle$ is a tuple of SFSTs with the same input monoid, we use the notation $T(x)$ to denote $\langle T_1(x), T_2(x), \ldots, T_n(x) \rangle$.

2.2 Complexity Theory

As usual, we formalize algorithms as deterministic or nondeterministic Turing machines (DTMs and NTMs, respectively), but we abstract away from their exact definitions. We assume that all Turing machines have a read-only input tape, a read–write work tape, and a write-only output tape, each of which uses the tape alphabet $\{0, 1\}$. We assume that all DTMs and NTMs are **write-once**: Their output tape heads cannot move left, and must move to the right immediately after writing a bit. We assume that all mathematical objects ϕ are represented on the tapes as a bit string $[\![\phi]\!] \in \{0, 1\}^*$. A Turing machine **computes** a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ if, for every input $x \in \{0, 1\}^*$, the machine halts with $f(x)$ on the output tape after starting its computation with x on the input tape and ε on the work and output tapes. A Turing machine **decides** a language $L \subseteq \{0, 1\}^*$ if it computes the characteristic function $\mathbb{1}_L : \{0, 1\}^* \rightarrow \{0, 1\}$ of L.

We say that a Turing machine **runs in polynomial** (resp. **linear**, **exponential**) **time** if its total number of computation steps is always polynomial (resp. linear, exponential) in the length of its input. We say that a Turing machine **runs in polynomial** (resp. **logarithmic**, **linear**, **exponential**) **space** if the size of the contents of its work tape is always at most polynomial (resp. logarithmic, linear, exponential) in the length of its input.

In this article, we deal with the following complexity classes.

- NL is the class of languages decidable by an NTM in logarithmic space.

- P is the class of languages decidable by a DTM in polynomial time.

- NP is the class of languages decidable by an NTM in polynomial time.

- coNP is the class of languages whose complements are in NP.

- NP^{NP} is the class of languages decidable by an NTM in polynomial time with oracle access to a language in NP or coNP (i.e., there is some language $L \in NP \cup coNP$ such that the NTM is allowed to decide L in one computational step).

- PSPACE is the class of languages decidable by a DTM in polynomial space.

- NPSPACE is the class of languages decidable by an NTM in polynomial space.

- coPSPACE is the class of languages whose complements are in PSPACE.

- coNPSPACE is the class of languages whose complements are in NPSPACE.

- EXPSPACE is the class of languages that are decidable by a DTM in exponential space.

The following relationships between the above complexity classes are currently known.

- $NL \subseteq P \subseteq NP \subseteq NP^{NP} \subseteq PSPACE \subsetneq EXPSPACE$

- $P \subseteq coNP \subseteq NP^{NP}$

- $coNPSPACE = coPSPACE = PSPACE = NPSPACE$, by Savitch's (1970) theorem

- $NL \subsetneq PSPACE$, by the space hierarchy theorem

We say that a language L is **hard with respect to a complexity class** A (or alternatively, that L is A-**hard**) if every language in A is **reducible to** L, according to the following definition.

Definition 3 (Logspace Reduction)
A function $f : \{0,1\}^* \to \{0,1\}^*$ is **logspace reducible to** a function $g : \{0,1\}^* \to \{0,1\}^*$ if and only if there exists a function $h : \{0,1\}^* \to \{0,1\}^*$ computed by a DTM in logarithmic space such that $f = g \circ h$. A language L is **logspace reducible to** a language M

if and only if $\mathbb{1}_L$ is logspace reducible to $\mathbb{1}_M$. We refer to h as a **reduction** of f to g (or L to M).

We say that L is **complete with respect to** A, or A-**complete**, if $L \in A$ and L is A-hard. To show that a language L is A-hard, it suffices to show that an A-hard language is logspace reducible to L.

3. Background: Computational Analysis of Optimality Theory

This section surveys past work relevant to this article, providing background for our main contributions. We begin with a high-level introduction of OT as it is commonly used in phonology. We then turn to the formal treatment of OT, reviewing ways in which it has been conceptualized as a formal system and as a computational problem.

3.1 Introduction to OT Phonology

We introduce OT by way of example. Consider the English plural suffix -s. Although the UR for this suffix is /z/, it surfaces as [s] when preceded by a voiceless consonant (e.g., *cats* [kæts]) and as [əz] when preceded by a sibilant (e.g., *foxes* [fɑksəz]). A typical OT analysis would propose that the SR distribution of -s is caused by the following constraints, listed in order of rank.

- MAX: Assign one violation for every symbol deleted from the UR.

- OCP: Assign one violation for every two consecutive sibilants in the SR.[1]

- AGREE(voice): Assign one violation for every voiceless consonant that is adjacent to a voiced consonant in the SR.

- DEP: Assign one violation for every symbol inserted into the UR.

- IDENT(voice): Assign one violation for every z that is changed to s and vice versa.[1]

At least one of the three faithfulness constraints MAX, DEP, and IDENT(voice) is violated whenever the SR differs from the UR. Since MAX is the highest-ranking constraint, the plural suffix can never be deleted. Changing /z/ to [s] or epenthesizing ə to form [əz], which would violate IDENT(voice) and DEP, respectively, can only occur if [z] violates one of the two markedness constraints, OCP or AGREE(voice). AGREE(voice) is violated when -s is preceded by a voiceless consonant. Since IDENT(voice) is the lowest-ranking faithfulness constraint, the violation of AGREE(voice) is repaired by changing /z/ to [s]. On the other hand, OCP is violated when -s is preceded by a sibilant. However, changing /z/ to [s] does not repair the OCP violation, so [ə] is epenthesized instead.

OT analyses are typically visualized using a **tableau**—a table showing various potential SRs, or **candidates**, and the degree to which each constraint is violated. Figure 1 shows tableaux that analyze the SRs of *cats* and *foxes*. The UR is shown in the top-left corner; each row corresponds to a candidate SR, and each column corresponds to a

---

1 In practice, these constraints apply to broader classes of phonemes than what we have described here. We state these constraints here in a restricted form for simplicity of exposition.

/kætz/ *cats*	MAX	OCP	AGREE(voice)	DEP	IDENT(voice)
a. [kætz]	0	0	1	0	0
☞ b. [kæts]	0	0	0	0	1
c. [kætəz]	0	0	0	1	0
d. [kæt]	1	0	0	0	0
/fɑksz/ *foxes*	MAX	OCP	AGREE(voice)	DEP	IDENT(voice)
a. [fɑksz]	0	1	1	0	0
b. [fɑkss]	0	1	0	0	1
☞ c. [fɑksəz]	0	0	0	1	0
d. [fɑks]	1	0	0	0	0

Figure 1
Tableaux showing the violations incurred by various candidates for the English plural OT grammar. In each tableau, the SR is marked with the symbol ☞.

constraint. The numbers in the cells indicate the number of violations assigned by each constraint to each candidate.

Observe that the five constraints we have discussed here do not suffice for uniquely determining the pronunciation of -*s*. For instance, because the description of DEP given above makes no distinction between different vowels, there is no reason why the epenthesized vowel should be [ə] and not some other vowel. Furthermore, the constraints we have stated here do not distinguish between the plural suffix -*s* and other instances of the segment /z/; this means that, for example, our analysis predicts that the SR for *catnip* should be *[kætənɪp] instead of the true SR [kætʰnɪp]. Accounting for all the possible edge cases, however, would require the introduction of an unwieldy number of constraints into the analysis. For this reason, the constraints included in an OT analysis are typically limited to those that are directly relevant for explaining the phenomenon under examination. In this case, because we are only interested in explaining when -*s* surfaces as [z], [s], or [əz], it suffices for our discussion to only include constraints that distinguish between those three possible SRs for the suffix -*s*.

3.2 OT as a Formal System

Numerous formalizations of OT have been proposed in the literature, such as those of Ellison (1994), Eisner (1997), Frank and Satta (1998), Karttunen (1998), Chen-Main and Frank (2003), and Riggle (2004). These formalizations share several common characteristics: The URs and SRs are represented as strings; constraints are implemented using DFAs or SFSTs; and each candidate is associated with a vector of violation numbers, known as a **violation profile**, corresponding to a row in a tableau. Many of these treatments introduce restrictions, such as setting a maximum bound on the number of violations that can be assigned by a constraint, designed to facilitate compilation of OT grammars into SFSTs for fast runtime performance.

The most general formalization of OT is that of Riggle (2004), which Section 4 describes in detail. In this version of OT, candidates are represented as strings of paired symbols $\langle x_1, y_1 \rangle \langle x_2, y_2 \rangle \ldots \langle x_n, y_n \rangle$, where $x = x_1 x_2 \ldots x_n$ is the UR, $y = y_1 y_2 \ldots y_n$ is a potential SR, and each x_i and y_i has length 1 or 0. This representation, inspired by McCarthy and Prince's (1995) **Correspondence Theory**, reifies epenthesis, deletion, and

other segment-level operations by aligning each symbol of *y* with a symbol of *x*. Constraints are then implemented as SFSTs that read a candidate and output the number of violations incurred by that candidate. Because this formalism is not designed for compilation into SFSTs, there are no restrictions on the number of violations a constraint may assign. Riggle (2004) implements universal generation by first constructing a constraint requiring the UR to be *x*, and then intersecting this constraint with all the other constraints in the grammar. This operation results in an SFST that reads a candidate and outputs the full violation profile for that candidate. The optimal candidate is found by using Dijkstra's (1959) algorithm to find the shortest path, measured by violation profiles, through the state diagram of intersected SFST.

3.3 OT as a Computational Problem

Complexity results for OT crucially depend on how OT is formalized as a computational problem. Although prior work always assumes that an optimal SR *y* must be computed from a UR *x* given a list of constraints *C*, there is variation in the literature in terms of whether *C* is considered part of the problem instance, or whether it is treated as a constant. Heinz, Kobele, and Riggle (2009) categorize these problem formulations into three types:

- the **simple generation problem**, where *C* is treated as a constant;

- the **universal generation problem**, where *C* is part of the input; and

- the **quasi-universal generation problem**, where *C* is a constant, but the input includes a permutation π of *C*.

Synthesizing prior complexity results, Heinz, Kobele, and Riggle report that the simple and quasi-universal generation problems can be solved in linear time, while the universal generation problem is NP-hard. These results are a consequence of the fact that, in the simple and quasi-universal generation problems, the exponential space used by the constraint intersection step of Riggle's (2004) algorithm can be treated as a constant, since the constraints themselves are treated as constants.

The quasi-universal generation problem was originally proposed by Heinz, Kobele, and Riggle (2009) as part of a discussion of the implications of complexity results on OT phonology as a theory of human behavior. The quasi-universal generation problem reifies the typical assumption in OT phonology that all languages share the same *set* of constraints, but differ from one another in the *ranking* of those constraints. While Idsardi (2006) interprets the NP-hardness of universal generation to mean that "[OT] is computationally intractable," Heinz, Kobele, and Riggle use the quasi-universal generation problem to argue that the assumption of a universal constraint set suffices to make OT tractable.

In this article, we propose a fourth version of OT generation: the **bounded universal generation problem**, where *C* is treated as part of the input, but the number of constraints in *C* is bounded by a constant. In Section 7, we show that bounded universal generation lies between the complexity classes NL and NP^{NP}, making it easier than universal generation (assuming $\text{NP} \neq \text{PSPACE}$) but possibly harder than quasi-universal generation (assuming $\text{P} \neq \text{NP}$).

In prior work, OT generation problems are formulated as **function problems**, where an algorithm is expected to output an SR. This is the version of universal generation

Table 1
Computational problems associated with OT, formulated as decision problems, along with their computational complexity. Variables are interpreted as follows: x represents a UR, y represents an SR, $C = \langle C_1, C_2, \ldots, C_n \rangle$ is a list of ranked constraints, π is a permutation of C, and k is a constant integer. Entries marked with * are introduced in this article. All other terms and results are introduced in Heinz, Kobele, and Riggle (2009).

	Language	Complexity
Simple Generation	$\mathsf{SG}(C) = \left\{ [\![\langle x, y \rangle]\!] \,\middle|\, \begin{array}{l} y \text{ optimally satisfies } C \\ \text{for UR } x \end{array} \right\}$	DTIME(n)
Quasi-Universal Generation	$\mathsf{QUG}(C) = \left\{ [\![\langle \pi, x, y \rangle]\!] \,\middle|\, \begin{array}{l} y \text{ optimally satisfies} \\ \pi(C) \text{ for UR } x \end{array} \right\}$	DTIME(n)
Bounded Universal Generation*	$\mathsf{BUG}(k) = \left\{ [\![\langle C, x, y \rangle]\!] \,\middle|\, \begin{array}{l} y \text{ optimally satisfies} \\ C \text{ for UR } x, \text{ where} \\ \|C\| = k \end{array} \right\}$	$\mathrm{NP}^{\mathrm{NP}}$ and NL-Hard*
Universal Generation	$\mathsf{UG} = \left\{ [\![\langle C, x, y \rangle]\!] \,\middle|\, \begin{array}{l} y \text{ optimally satisfies } C \text{ for} \\ \text{UR } x \end{array} \right\}$	PSPACE-Complete*

considered in Theorem 2. However, classical complexity classes like P, NP, NL, PSPACE, and EXPSPACE only include **decision problems**, where the algorithm is expected to decide a language of bit strings. For this reason, in Table 1 we reformulate the four OT generation problems as decision problems where an algorithm must verify whether or not a string y is the correct SR for given a UR and a list of constraints.

4. Formal Definition of Optimality Theory

This section describes the version of OT proposed by Riggle (2004). We choose to use this version of OT because it is the most powerful: It allows arbitrary finite-state constraints that assign arbitrary numbers of violations. Because our goal is to establish PSPACE as an *upper* bound on the complexity of OT, using the most powerful version of OT available makes it likely that our results extend to other versions of OT.

We begin by describing the representation of candidates. As mentioned in Section 3.2, candidates are represented as pairs of strings in which each symbol of one string is optionally aligned with a symbol of the other. This allows candidates to record the operations (epentheses, deletions, and substitutions) used to derive the candidate SR from the UR, so that faithfulness constraints may be evaluated.

Definition 4 (Representation of Candidates)
A **candidate over** Σ **and** Γ is a string over the alphabet $\Sigma_\varepsilon \times \Gamma_\varepsilon$. For $\alpha = \langle x_1, y_1 \rangle \langle x_2, y_2 \rangle \ldots \langle x_n, y_n \rangle$, we define $\alpha^\triangleleft = x_1 x_2 \ldots x_n$ and $\alpha^\triangleright = y_1 y_2 \ldots y_n$.

Next, we define constraints as SFSTs that map candidates to numbers of violations. For the purposes of our analysis, it is not necessary to make a formal distinction between markedness and faithfulness constraints.

Definition 5 (Constraints)
A **constraint over** Σ **and** Γ is a SFST with input monoid $(\Sigma_\varepsilon \times \Gamma_\varepsilon)^*$ and output monoid $\mathbb{N}$. If C_i is a constraint, then for a candidate $\alpha \in (\Sigma_\varepsilon \times \Gamma_\varepsilon)^*$, the output of C_i on input α is denoted by $C_i(\alpha)$. If $C = \langle C_1, C_2, \ldots, C_n \rangle$ is a tuple of constraints over Σ and Γ, then

the **violation profile of** α **with respect to** C, denoted by $C(\alpha)$, is defined as the tuple $C(\alpha) = \langle C_1(\alpha), C_2(\alpha), \ldots, C_n(\alpha) \rangle$.

Finally, we define an ordering relation $<$ on violation profiles, such that a candidate α is "more optimal" than candidate β for a list of constraints C if and only if $C(\alpha) < C(\beta)$. Informally, more optimal candidates are those that incur fewer violations of higher-ranked constraints, where constraints are listed in order of decreasing rank.

Definition 6 (Ordering of Violation Profiles)
For each $k \in \mathbb{N}$, the **lexicographic ordering on** $\mathbb{N}^k$ is the ordering $<$ defined by $\langle a_1, a_2, \ldots, a_k \rangle < \langle b_1, b_2, \ldots, b_k \rangle$ if and only if there exists i such that $a_i < b_i$ and $a_j = b_j$ for all $j < i$. We write $a \leq b$ for $a, b \in \mathbb{N}^k$ to mean that $a < b$ or $a = b$.

We define optimal SRs as SRs that correspond to candidates that are minimal with respect to $<$.

Definition 7 (Optimality)
Let C be a list of constraints over Σ and Γ, and let $x \in \Sigma^*$ be a UR. We say that an SR $y \in \Gamma^*$ is **optimal with respect to** C **and** x if and only if there is a candidate $\alpha \in (\Sigma_\varepsilon \times \Gamma_\varepsilon)^*$ such that

- $\alpha^{\triangleleft} = x$,

- $\alpha^{\triangleright} = y$, and

- for all $\beta \in (\Sigma_\varepsilon \times \Gamma_\varepsilon)^*$ with $\beta^{\triangleleft} = x$, $C(\beta) \geq C(\alpha)$.

Example 1 (OCP and Faithfulness Constraints)
Recall the OCP markedness constraint and the faithfulness constraints MAX, DEP, and IDENT(voice) from Section 3.1. Assuming that $\Sigma = \Gamma$ is the alphabet of International Phonetic Alphabet symbols, Figure 2 illustrates how these constraints may be implemented using SFSTs. The markedness constraint AGREE(voice) is implemented using an SFST with a structure similar to that of OCP.
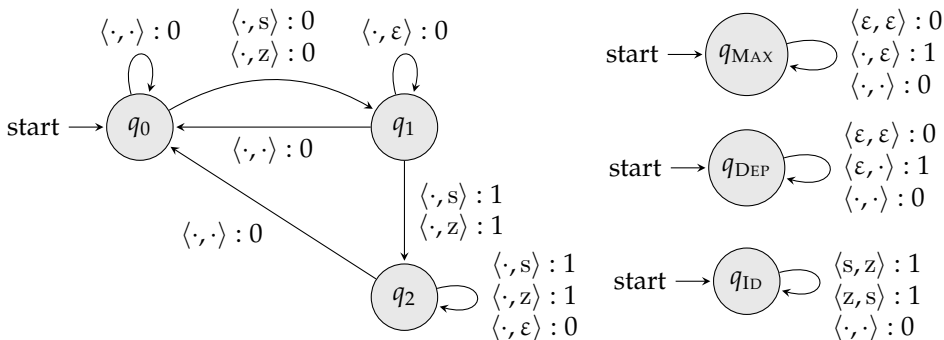

Figure 2
SFSTs for the constraints OCP (left), MAX (upper right), DEP (middle right), and IDENT(voice) (lower right) in the English plural OT grammar. The notation $\cdot$ refers to any symbol in $\Sigma_\varepsilon \cup \Gamma_\varepsilon$ that results in a valid transition.

The candidate [kæts] for *cats* is represented as $\langle k, k \rangle \langle æ, æ \rangle \langle t, t \rangle \langle z, s \rangle$, while the candidate [fɑksəz] for *foxes* is represented as $\langle f, f \rangle \langle ɑ, ɑ \rangle \langle k, k \rangle \langle s, s \rangle \langle ɛ, ə \rangle \langle z, z \rangle$. Identifying constraints with their SFSTs as shown in Figure 2, it is easy to see that [kæts] undergoes the following transitions when read by the SFST for IDENT(voice):

$$q_{\text{ID}} \xrightarrow[0]{\langle k,k \rangle} q_{\text{ID}} \xrightarrow[0]{\langle æ,æ \rangle} q_{\text{ID}} \xrightarrow[0]{\langle t,t \rangle} q_{\text{ID}} \xrightarrow[1]{\langle z,s \rangle} q_{\text{ID}} \xrightarrow[0]{} \#$$

while [fɑksəz] undergoes the following transitions when read by DEP:

$$q_{\text{DEP}} \xrightarrow[0]{\langle f,f \rangle} q_{\text{DEP}} \xrightarrow[0]{\langle ɑ,ɑ \rangle} q_{\text{DEP}} \xrightarrow[0]{\langle k,k \rangle} q_{\text{DEP}} \xrightarrow[0]{\langle s,s \rangle} q_{\text{DEP}} \xrightarrow[1]{\langle ɛ,ə \rangle} q_{\text{DEP}} \xrightarrow[0]{\langle z,z \rangle} q_{\text{DEP}} \xrightarrow[0]{} \#$$

The candidate [fɑksz] undergoes the following transitions when read by OCP:

$$q_0 \xrightarrow[0]{\langle f,f \rangle} q_0 \xrightarrow[0]{\langle ɑ,ɑ \rangle} q_0 \xrightarrow[0]{\langle k,k \rangle} q_0 \xrightarrow[0]{\langle s,s \rangle} q_1 \xrightarrow[1]{\langle z,z \rangle} q_2 \xrightarrow[0]{} \#$$

Consider the list of constraints $C = \langle \text{MAX}, \text{OCP}, \text{AGREE(voice)}, \text{DEP}, \text{IDENT(voice)} \rangle$. As discussed in Section 3.1, these five constraints do not actually suffice to predict the correct forms for English plurals. For example, observe that

$$C(\langle k, ɑ \rangle \langle æ, ɑ \rangle \langle t, ɑ \rangle \langle z, ɑ \rangle) = \langle 0, 0, 0, 0, 0 \rangle < \langle 0, 0, 0, 0, 1 \rangle = C(\langle k, k \rangle \langle æ, æ \rangle \langle t, t \rangle \langle z, s \rangle)$$

This means that ɑɑɑɑ is optimal with respect to C and kætz, but kæts is not. While the constraints that eliminate candidates like [ɑɑɑɑ] are typically abstracted away in phonological theory, in the formal setting we must assume that C contains *all* constraints necessary to achieve the desired mapping.

5. Universal Generation in Polynomial Space

In this section, we prove that universal generation can be done in polynomial space, deriving Theorem 2 as well as one half of Theorem 1. To do so, we will use the following formulation of the universal generation problem.

Lemma 1
The language

$$\text{ug} = \{[\![\langle C, x, v \rangle]\!] \mid \exists \alpha [\alpha^{\triangleleft} = x \wedge C(\alpha) \leq v]\}$$

is in PSPACE.

The language ug represents the problem of deciding whether, given an SR x, list of constraints C, and violation profile v, there exists a candidate for x that is more optimal than v. Proving a statement like Lemma 1 is a common approach to complexity analysis for combinatorial optimization problems. For analogy, consider the traveling salesperson problem, which asks an algorithm to find the shortest path that connects a set of points in Euclidean space. The complexity analysis of the traveling salesperson problem is typically stated as follows (see Arora and Barak 2009, p. 40, for an overview).

Proposition 1 (Traveling Salesperson Problem)
The language

$$\mathsf{tsp} = \{[\![\langle P, l\rangle]\!] \mid \text{There is a path of length at most } l \text{ connecting all points in } P\}$$

is NP-complete.

The reason the traveling salesperson problem is formulated in this way is because it is easy to extend an algorithm that decides tsp to one that finds the shortest path connecting all the points in P. Such an algorithm would iterate through possible values of l, while using an NTM that decides tsp to nondeterministically generate paths through the points of P with a length of at most l. When l is small enough such that $[\![\langle P, l\rangle]\!] \notin \mathsf{tsp}$, then the most recently generated path is returned.

In the remainder of this section, we apply this line of reasoning to universal generation for OT. We begin by proving Lemma 1, and then we use Lemma 1 to prove Theorem 2 and the first half of Theorem 1.

5.1 Proof of Lemma 1

To prove Lemma 1, we will adopt a strategy similar to the proof of Proposition 1, where an NTM decides tsp by nondeterministically generating a path through the points in P and verifying that it is a valid path with length at most l. In our case, we will use an NTM to nondeterministically generate a candidate α, and check that $\alpha^{\triangleleft} = x$ and $C(\alpha) \leq v$. Since PSPACE = NPSPACE (Savitch 1970), if our NTM uses only polynomial space, then Lemma 1 is proven.

The main challenge to this approach is that we cannot guarantee that the length of α is polynomial in $|[\![\langle C, x, v\rangle]\!]|$. Proposition 2, stated below, shows that an NTM that decides ug will occasionally need to generate a candidate that does not fit within polynomial space. While generating such a candidate is not a problem (since NPSPACE imposes no restriction on the running time of an NTM), our NTM will not be able to write down the candidate in memory, at least not in its entirety.

Proposition 2 (Existence of Long Optimal Candidates)
For every polynomial $f(n)$, there is a UR x and a list of constraints C such that

- there is a candidate α such that $\alpha^{\triangleleft} = x$ and $C(\alpha) = \langle 0, 0, \ldots, 0\rangle$, and

- for all candidates α with $\alpha^{\triangleleft} = x$, $C(\alpha) = \langle 0, 0, \ldots, 0\rangle$ only if
  $|\alpha| > f(|[\![\langle C, x, v\rangle]\!]|)$.

Proof. See Appendix A.   □

Thankfully, we do not need to write down α in order to verify that $C(\alpha) \leq v$. Instead, we simply present each symbol pair of α to the constraints as it is guessed. The previously guessed symbol pairs do not need to be remembered; it suffices to remember the most recent state of each constraint, as well as the number of violations that have been assigned by the constraints so far. While remembering the most recent states of the constraints only requires at most linear space, the representations of the violation numbers could potentially grow indefinitely as more and more symbol pairs are generated. Therefore, to ensure that the information required to decide ug fits within

polynomial space, we need to establish an upper bound on the number of violations that can be issued by a list of constraints.

Lemma 2 (Exponential Upper Bound on Violation Numbers)
Let x be a UR, let $C = \langle C_1, C_2, \ldots, C_n \rangle$ be a list of constraints, and let $l = |[\![\langle C, x \rangle]\!]|$. Then,

- there is a candidate of length at most le^l that is optimal for C and x, and

- for all candidates α, if $|\alpha| \leq le^l$, then for all i, $C_i(\alpha) \leq le^{2l}$.

Proof. See Appendix A.   □

By Lemma 2, in order to decide whether $[\![\langle C, x, v \rangle]\!] \in \text{ug}$, it suffices for our NTM to only consider candidates of up to exponential length, since these candidates are guaranteed to include at least one optimal candidate. As long as this length limit is observed, the violation numbers computed by the constraints will be exponential in value, and therefore their binary representations will be polynomial in size.

We are now ready to prove Lemma 1.

Proof of Lemma 1. Define the following nondeterministic procedure for deciding ug. On input $[\![\langle C, x, v \rangle]\!]$, where $C = \langle C_1, C_2, \ldots, C_n \rangle$ is a list of constraints over Σ_ε and Γ_ε:

1. Initialize the variables $x' = \varepsilon$ and $l' = 0$. Let $l = |[\![\langle C, x, v \rangle]\!]|$.

2. For each $i \in \{1, 2, \ldots, n\}$, initialize the variable $v_i = 0$, and initialize q_i to be the start state of C_i.

3. Repeat indefinitely:

    (a) Nondeterministically generate a pair $\langle a, b \rangle \in \Sigma_\varepsilon \times \Gamma_\varepsilon$ such that x begins with $x'a$.

    (b) For each $i \in \{1, 2, \ldots, n\}$, let u_i and r_i be such that

    $$q_i \xrightarrow[u_i]{\langle a,b \rangle} r_i,$$

    where $\rightarrow$ is the transition function for C_i.

    (c) Update $x' \leftarrow x'a$ and $l' \leftarrow l' + 1$, and for each $i \in \{1, 2, \ldots, n\}$, update $v_i \leftarrow v_i + u_i$ and $q_i \leftarrow r_i$.

    (d) If $l' = le^l$, then terminate this loop. Otherwise, nondeterministically decide whether or not to terminate this loop.

4. For each $i \in \{1, 2, \ldots, n\}$, update $v_i \leftarrow v_i + \#_i(q_i)$, where $\#_i$ is the final transition function for C_i.

5. If $x' = x$ and $\langle v_1, v_2, \ldots, v_n \rangle \leq v$, then return 1. Otherwise, return 0.

This algorithm generates a candidate α of length at most le^l, where $l = |[\![\langle C, x, v\rangle]\!]|$. While doing so, it keeps track of $\alpha^\triangleleft = x'$ and $C(\alpha) = \langle v_1, v_2, \ldots, v_n\rangle$, but does not remember α itself. It then checks whether $\alpha^\triangleleft = x$ and $C(\alpha) \leq v$, and returns 1 if these conditions are met. Recall that an NTM returns 1 as long as *some* set of nondeterministic choices leads to an output of 1. By Lemma 2, at least one such set of choices leads to the generation of an optimal candidate, causing 1 to be returned if and only if $[\![\langle C, x, v\rangle]\!] \in \mathsf{ug}$.

To verify that the NTM described above uses only polynomial space, we observe that the input $\langle C, x, v\rangle$ as well as the variables x', l', l, $q_1, q_2, \ldots, q_n$, and indices used for looping all fit within linear space, and that Lemma 2 guarantees that $v_1, v_2, \ldots, v_n$ are of polynomial size when represented in binary form. □

5.2 Proof of Theorem 2

We now prove that universal generation, formulated as a function problem, can be done in polynomial space. Below we restate Theorem 2 using the formalism we have defined in Section 4.

Theorem 2 (Restated)
There is a polynomial-space computable function

$$\mathsf{UGFunc}([\![\langle C, x\rangle]\!]) = [\![y]\!]$$

such that C is a list of constraints, x is a UR, and y is an SR that is optimal for C and x.

Recall that in Section 2, we have assumed that the measurement of space complexity does not include the output tape. Therefore, the mere fact that the output of UGFunc may be exponential in size thanks to Proposition 2 does not automatically disprove Theorem 2, as long as only polynomially many positions of the work tape are used.

Our strategy for implementing UGFunc is as follows. Since the SR might be exponentially long, we cannot write down the SR on the work tape. Instead, we generate the SR one symbol at a time, writing each symbol to the output tape before generating the next symbol. Because the output tape is write-once, we cannot go back and change a previously generated symbol. Therefore, we use the following lemma to verify that our generated symbols are correct before writing them to the output tape.

Lemma 3
The language

$$\mathsf{ugFirst} = \{[\![\langle C, x, \langle a, b\rangle, v\rangle]\!] \mid \exists\alpha[(\langle a, b\rangle\alpha)^\triangleleft = x \wedge C(\langle a, b\rangle\alpha) \leq v]\}$$

is in PSPACE.[2]

Proof. Write $C = \langle C_1, C_2, \ldots, C_n\rangle$, and assume that $x = ax'$ for some x' (if not, then $[\![\langle C, x, \langle a, b\rangle, v\rangle]\!] \notin \mathsf{ugFirst}$). For each $i \in \{1, 2, \ldots, n\}$, let $q_{i,0}$ be the start state of C_i, and let r_i and u_i be such that

$$q_{i,0} \xrightarrow[u_i]{\langle a, b\rangle} r_i$$

---

2 It is actually PSPACE-complete, but we will not prove this here.

where $\to$ is the transition function for C_i. Now, for each i, let C_i' be C_i, but with r_i as the start state instead of $q_{i,0}$. Let $C' = \langle C_1', C_2', \ldots, C_n' \rangle$, and let $v' = \langle v_1 - u_1, v_2 - u_2, \ldots, v_n - u_n \rangle$. The membership of $[\![\langle C, x, \langle a, b \rangle, v \rangle]\!]$ in ugFirst can then be decided in polynomial space by simply deciding whether $[\![\langle C, x', v' \rangle]\!] \in$ ug. $\square$

Lemma 3 allows us to verify (in polynomal space) whether a symbol pair $\langle a, b \rangle$ is the first valid pair of an optimal candidate by checking whether $[\![\langle C, x, \langle a, b \rangle, v \rangle]\!] \in$ ugFirst, where v is the violation profile of optimal candidates. To do this, we need to be able to compute the violation profile of optimal candidates in the first place. This can be done in polynomial space thanks to the following lemma.

Lemma 4
The function given by

$$\mathsf{OptViol}([\![\langle C, x \rangle]\!]) = [\![v]\!]$$

where C is a list of constraints, x is a UR, and $C(\alpha) = v$ whenever α is optimal for C and x, is polynomial-space computable.

Proof. Given input $[\![\langle C, x \rangle]\!]$, let $l = |[\![\langle C, x \rangle]\!]|$. By Lemma 2, if $\mathsf{OptViol}([\![\langle C, x \rangle]\!]) = [\![v]\!]$, then v is of the form $v = \langle v_1, v_2, \ldots, v_n \rangle$, where $v_i \leq le^{2l}$ for all i. Therefore, in order to compute $\mathsf{OptViol}$ in polynomial space, it suffices to loop over all possible $v \in \{0, 1, \ldots, le^{2l}\}^n$ in reverse lexicographic order and check whether or not $[\![\langle C, x, v \rangle]\!] \in$ ug. When a result of 0 is obtained (i.e., $[\![\langle C, x, v \rangle]\!] \notin$ ug), the previous value of v is the optimal violation profile for C and x. $\square$

We are now ready to prove Theorem 2.

Proof of Theorem 2. Consider the following deterministic algorithm. On input $[\![\langle C, x \rangle]\!]$:

1. Let v be the optimal violation profile for C and x; thus, $\mathsf{OptViol}([\![\langle C, x \rangle]\!]) = [\![v]\!]$. Write $v = \langle v_1, v_2, \ldots, v_n \rangle$.

2. Let $C = \langle C_1, C_2, \ldots, C_n \rangle$, where each C_i is a constraint over Σ and Γ.

3. Repeat at most le^l times, where $l = |[\![\langle C, x \rangle]\!]|$:

   (a) For each $\langle a, b \rangle \in \Sigma_\varepsilon \times \Gamma_\varepsilon$ in lexicographic order, where ε is the last symbol of both Σ_ε and Γ_ε:

       i.   If $[\![\langle C, x, \langle a, b \rangle, v \rangle]\!] \notin$ ugFirst, then skip the following steps and move on to the next iteration of this for-loop.

       ii. Write $[\![b]\!]$ to the output tape, and let x' be such that $x = ax'$. Set $x \leftarrow x'$.

       iii. For each $i \in \{1, 2, \ldots, n\}$, let $q_{0,i}$ be the start state of C_i, and let r_i be such that

$$q_{0,i} \xrightarrow[u_i]{\langle a, b \rangle} r_i$$

where $\rightarrow$ is the transition function for C_i.
Update $v_i \leftarrow v_i - u_i$, and change the start
state of C_i from $q_{0,i}$ to r_i.

iv.   Break out of this for-loop.

(b)   If the inner for-loop in Step 3(a) finishes without writing
anything to the output tape, then break out of this outer
loop.

4.   Return the current contents of the output tape.

The algorithm above is designed to implement UGFunc. To do so, it first computes
the optimal violation profile, which by Lemma 4 requires only polynomial space, and
then generates an optimal SR one symbol at a time, using Lemma 3 to verify that the
generated symbol is valid. To ensure that the algorithm terminates, we set a time limit
of le^l for the outer loop, which by Lemma 2 is enough time to generate an optimal
candidate. With this time limit, however, it is possible that the outer loop terminates
before the entire UR has been included in the generated candidate (i.e., Step 5 may be
reached while $x \neq \varepsilon$). In order to prevent this, we assume in Step 3(a) that symbol pairs
of the form $\langle \varepsilon, b \rangle$ are the last to be considered by the inner loop. This causes the UR to
be generated as early as possible, leaving superfluous epentheses and instances of $\langle \varepsilon, \varepsilon \rangle$
until the end of the computation.

Let us verify that the algorithm above uses polynomial space. By Lemma 4, Step
1 uses polynomial space, and by Lemma 3, Step 3(a)i uses polynomial space. Since
PSPACE = NPSPACE, we can assume that OptViol and ugFirst are decided determinis-
tically. By Lemma 2, the variable v only requires polynomial space, and it is clear that
the other variables only require polynomial space as well.   □

5.3 Partial Proof of Theorem 1

We conclude this section by showing that the decision version of the universal gener-
ation problem, as stated in Table 1, is in PSPACE. This proves one half of Theorem 1.

Theorem 3
The language

$$UG = \{ [\![\langle C, x, y \rangle]\!] \mid y \text{ is optimal for } C \text{ and } x \}$$

is in PSPACE.

Proof. Fix an input $[\![\langle C, x, y \rangle]\!]$, and let C_0 be the markedness constraint shown in
Figure 3. This constraint checks whether a candidate α corresponds to the SR y: we
have $C_0(\alpha) = 0$ if and only if $\alpha^\triangleright = y$. Now, observe that UG is decided in polynomial
space using the following algorithm. On input $[\![\langle C, x, y \rangle]\!]$:

1.   Construct the constraint C_0, described in Figure 3.
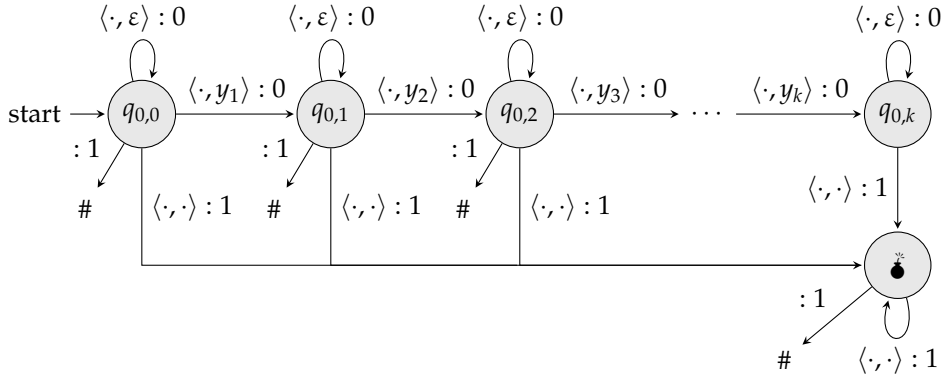
Figure 3
A constraint that assigns one or more violations to a candidate α when $\alpha^{\triangleleft} \neq y = y_1 y_2 \ldots y_k$. The notation $\cdot$ refers to any symbol in $\Sigma_{\varepsilon} \cup \Gamma_{\varepsilon}$ that results in a valid transition.

2.   Writing $C = \langle C_1, C_2, \ldots, C_n \rangle$, let $C' = \langle C_0, C_1, \ldots, C_n \rangle$.

3.   Writing $\mathsf{OptViol}([\![\langle C, x \rangle]\!]) = [\![\langle v_1, v_2, \ldots, v_n \rangle]\!]$, let $v' = \langle 0, v_1, v_2, \ldots, v_n \rangle$.

4.   Return 1 if $[\![\langle C', x, v' \rangle]\!] \in \mathsf{ug}$ and 0 otherwise.

This algorithm clearly runs in polynomial space, since $|[\![\langle C', x, v' \rangle]\!]|$ is linear in $|[\![\langle C, x, y \rangle]\!]|$. It decides whether $[\![\langle C, x, y \rangle]\!] \in \mathsf{UG}$ by deciding the existence of an optimal candidate α such that $\alpha^{\triangleright} = y$. The condition that $\alpha^{\triangleright} = y$ is enforced by requiring that $C_0(\alpha) = 0$, and the condition that α is optimal is enforced by requiring that $C(\alpha) \leq \mathsf{OptViol}([\![\langle C, x \rangle]\!])$. $\square$

6. PSPACE-Hardness of Universal Generation

We now complete the proof of Theorem 1 by showing that ug, UGFunc, and UG are PSPACE-hard. To do so, we reduce the following PSPACE-complete problem to ug, UGFunc, and UG in logarithmic space.

Definition 8 (A PSPACE-Complete Problem, Kozen 1977)
The **intersection non-emptiness problem for DFAs** is the language $\mathsf{INE\text{-}DFA} \subseteq \{0, 1\}^*$ defined by

$$\mathsf{INE\text{-}DFA} = \left\{ [\![\langle M_1, M_2, \ldots, M_n \rangle]\!] \;\middle|\; \text{Each } M_i \text{ is a DFA and } \bigcap_{i=1}^{n} M_i \neq \varnothing \right\}$$

The intersection non-emptiness problem for DFAs asks, given a list of DFAs, whether there are strings accepted by all DFAs in the list. Already, we can see a natural connection between universal generation and the intersection non-emptiness problem: Both deal with the intersection of finite-state machines. In order to reduce $\mathsf{INE\text{-}DFA}$ to OT universal generation, we convert each DFA M into the following OT constraint.

- ACCEPT(M): Assign one violation to candidate α if $\alpha^{\triangleright} \notin M$, unless $\alpha = \langle {\llcorner}, {\llcorner} \rangle$.

We then add a constraint called NOTBLANK, defined below, as the lowest-ranking constraint.

- NOTBLANK: Assign one violation to candidate α if $\alpha^{\triangleright} = {\llcorner}$.

In other words, we convert a set of DFAs $\{M_1, M_2, \ldots, M_n\}$ into the list of constraints $C = \langle \text{ACCEPT}(M_1), \text{ACCEPT}(M_2), \ldots, \text{ACCEPT}(M_n), \text{NOTBLANK} \rangle$, where ${\llcorner}$ is a special symbol not in any of the M_is' alphabets.

The basic idea behind our approach is as follows. If y is a string accepted by all the M_is, then y is always an optimal SR for C and UR ${\llcorner}$, since such a y would not violate any of the constraints in C. If no string is accepted by all the M_is (i.e., if $\bigcap_{i=1}^{n} M_i = \varnothing$), then the only optimal SR for C and ${\llcorner}$ is ${\llcorner}$. This is because ${\llcorner}$ only violates NOTBLANK, whereas any other SR would violate at least one of the ACCEPT(M_i) constraints, which are ranked higher than NOTBLANK. We can therefore reduce the problem of deciding whether $\bigcap_{i=1}^{n} M_i = \varnothing$ to the problem of deciding whether ${\llcorner}$ is an optimal SR for the constraint list C described above.

Example 2
Let $\Gamma = \{\text{a}, \text{b}\}$. Suppose $M_1 = \text{a}^*\text{b}^*$, $M_2 = (\Gamma\Gamma)^*$, and $M_3 = \text{b}\Gamma^*\text{a}$; and assume that these languages are identified with DFAs that accept them.

In the upper portion of Figure 4, we consider the constraint list $C = \langle \text{ACCEPT}(\text{a}^*\text{b}^*), \text{ACCEPT}((\Gamma\Gamma)^*), \text{NOTBLANK} \rangle$. Observe that $\text{a}^*\text{b}^* \cap (\Gamma\Gamma)^*$ is the set of even-length strings where no b precedes an a. Since the candidate $\alpha = \langle {\llcorner}, \text{a} \rangle \langle \varepsilon, \text{a} \rangle \langle \varepsilon, \text{a} \rangle \langle \varepsilon, \text{b} \rangle \langle \varepsilon, \text{b} \rangle \langle \varepsilon, \text{b} \rangle$ represents an SR satisfying these criteria (i.e., $\alpha^{\triangleright} = \text{aaabbb} \in \text{a}^*\text{b}^* \cap (\Gamma\Gamma)^*$), it is optimal for C and $x = {\llcorner}$.

In the lower portion of Figure 4, we consider the constraint list $C = \langle \text{ACCEPT}(\text{a}^*\text{b}^*), \text{ACCEPT}(\text{b}\Gamma^*\text{a}), \text{NOTBLANK} \rangle$. Now, observe that $\text{a}^*\text{b}^* \cap \text{b}\Gamma^*\text{a} = \varnothing$: strings in $\text{b}\Gamma^*\text{a}$ must

${\llcorner}$	ACCEPT(a*b*)	ACCEPT((ΓΓ)*)	NOTBLANK
☞ a. aaabbb	0	0	0
b. bbbaaa	1	0	0
c. aabbb	0	1	0
d. ${\llcorner}$	0	0	1

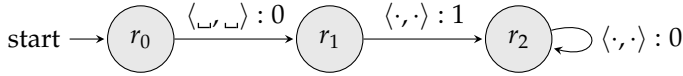| ${\llcorner}$	ACCEPT(a*b*)	ACCEPT(bΓ*a)	NOTBLANK
a. aaabbb	0	1	0
b. bbbaaa	1	0	0
c. aabbb	0	1	0
☞ d. ${\llcorner}$	0	0	1

Figure 4
Sample tableaux illustrating optimal SRs for the UR ${\llcorner}$ given a list of ACCEPT(M_i) constraints followed by NOTBLANK. When the intersection of the M_is is non-empty (upper tableau), the optimal SRs are precisely those that are accepted by all the M_is. When the intersection of the M_is is empty (lower tableau), ${\llcorner}$ is the only optimal SR.

begin with b and end with a, but a*b* does not allow any instance of b to precede an instance of a. Since the two ACCEPT(M_i) constraints contradict one another, any candidate other than $\langle \text{␣}, \text{␣} \rangle$ must violate at least one of them. Therefore, $\langle \text{␣}, \text{␣} \rangle$ is the only optimal candidate for this list of constraints.

6.1 Conversion of Automata to Constraints

We now spell out how exactly we can convert a DFA $M = \langle Q, \Gamma, q_0, F, \rightarrow \rangle$ into the constraint ACCEPT(M), implemented by the SFST $T = \langle R, A^*, \mathbb{N}, r_0, \Rightarrow, \# \rangle$, where $R = Q \cup \{r_0, r_1, r_2, \text{☃}\}$ and $A = \{\text{␣}, \varepsilon\} \times (\Gamma \cup \{\text{␣}, \varepsilon\})$. To do this, we propose a procedure in four steps.

The first step is to build the following SFST, which implements the "unless $\alpha = \langle \text{␣}, \text{␣} \rangle$" condition of ACCEPT($M$).

$$\text{start} \longrightarrow \boxed{r_0} \xrightarrow{\langle \text{␣}, \text{␣} \rangle : 0} \boxed{r_1} \xrightarrow{\langle \cdot, \cdot \rangle : 1} \boxed{r_2} \circlearrowright \langle \cdot, \cdot \rangle : 0$$

This SFST reads candidates of the form $\alpha \in \langle \text{␣}, \text{␣} \rangle A^*$ and assigns one violation if α is more than just $\langle \text{␣}, \text{␣} \rangle$.

The second step is to make a copy of M and convert it into an SFST that, upon reading a symbol pair $\langle a, b \rangle \in A$, simulates the behavior that M exhibits when reading b. To do this, for each transition of M of the form

$$q \xrightarrow{b} r$$

we add transitions of the form

$$q \xRightarrow[0]{\langle \varepsilon, b \rangle} r \quad \text{and} \quad q \xRightarrow[0]{\langle \text{␣}, b \rangle} r$$

to T. If $q \in Q$ is a state for which M does not have an ε-transition (i.e., there is no $r \in Q$ such that $q \xrightarrow{\varepsilon} r$), then we additionally add transitions of the form

$$q \xRightarrow[0]{\langle \varepsilon, \varepsilon \rangle} q \quad \text{and} \quad q \xRightarrow[0]{\langle \text{␣}, \varepsilon \rangle} q$$

The third step is to make T assign a violation when its simulation of M results in a rejection. There are two ways in which M can reject a string: Either it ends its computation on a non-accepting state, or it can reach a state for which there is no valid transition corresponding to the next input symbol. To handle the former case, we make the final transition function assign a violation when the computation ends on a non-accepting state (i.e., we set $\#(q) = 1$ whenever $q \in Q \backslash F$). To handle the latter case, we introduce a sink state ☃, and create transitions

$$q \xRightarrow[1]{\langle \varepsilon, b \rangle} \text{☃} \quad \text{and} \quad q \xRightarrow[1]{\langle \text{␣}, b \rangle} \text{☃}$$

whenever there is no transition of the form $q \xrightarrow{b} r$, as well as transitions

$$\ddot{\bullet} \xRightarrow[0]{\langle a,b \rangle} \ddot{\bullet}$$

for all $\langle a, b \rangle \in A$.

The fourth and final step is to connect the initial SFST consisting of the states r_0, r_1, and r_2 with the other states containing a copy of M. Since r_0 is the start state of T, we need it to exhibit the same behavior as q_0. Therefore, for every transition of the form

$$q_0 \xRightarrow[i]{\langle a,b \rangle} r$$

we add the transition

$$r_0 \xRightarrow[i]{\langle a,b \rangle} r$$

Example 3
Figure 5 shows a DFA M for the language $\mathrm{a^*b^*}$ over alphabet $\Gamma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, along with an SFST T for ACCEPT(M), constructed according to the procedure we have outlined.

The top row of T's states contains the states r_0, r_1, and r_2, which implement the "unless $\alpha = \langle \_, \_ \rangle$" condition of ACCEPT($M$). Below these three states is a copy of M's states: q_0, q_1, and q_2. Since q_2 is a non-accepting state in M, T has #(q_2) = 1. Below q_0, q_1, and q_3 is the sink state $\ddot{\bullet}$, which is reached whenever the symbol pair $\langle \varepsilon, \mathrm{c} \rangle$ or $\langle \_, \mathrm{c} \rangle$ is read. These transitions exist because M does not contain any valid transitions that involve reading a c.

There are only three ways in which T can emit a violation. The first is if a symbol pair is read after encountering $\langle \_, \_ \rangle$, causing T to transition from r_1 to r_2. Since $\_ \notin \Gamma$, no candidate beginning with $\langle \_, \_ \rangle$ can represent a string accepted by M (i.e., if $\alpha \in \langle \_, \_ \rangle A^*$, then $\alpha^\triangleright \notin M$); so if this transition is taken, then we know that a violation needs to be assigned. The second case is if the last state of T is a non-accepting state of M (viz., q_2), whereupon the final transition function assigns a violation. The third case is if T transitions to $\ddot{\bullet}$, causing exactly one violation to be assigned. Once $\ddot{\bullet}$ is reached, no further violations are assigned.

We now verify that the conversion procedure that we have described uses only logarithmic space.

Lemma 5
The function given by

$$f(\llbracket M \rrbracket) = \llbracket \mathrm{ACCEPT}(M) \rrbracket$$

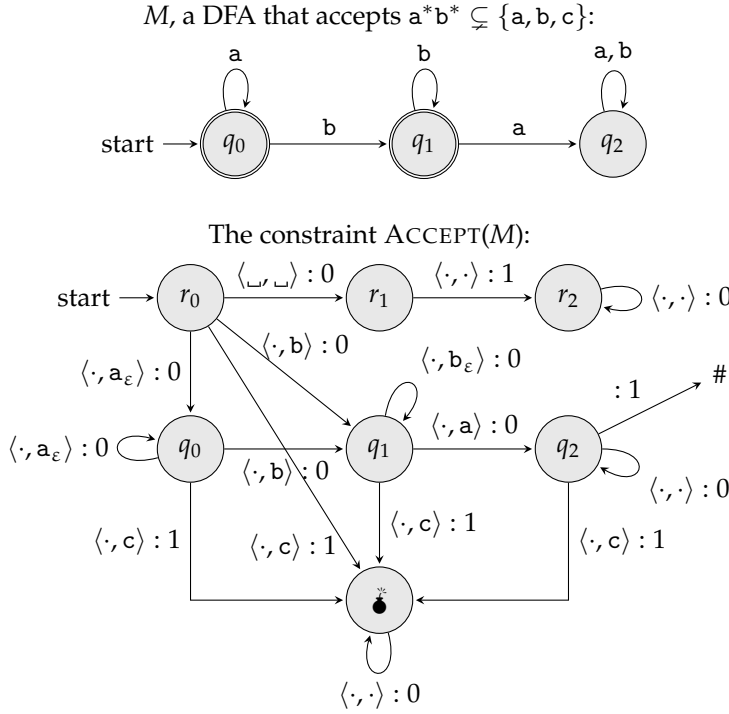where M is a DFA and ACCEPT(M) is implemented according to the procedure described above, is logspace computable.

M, a DFA that accepts $\mathtt{a}^*\mathtt{b}^* \subsetneq \{\mathtt{a},\mathtt{b},\mathtt{c}\}$:

The constraint ACCEPT(M):

Figure 5
A DFA M for the language $\mathtt{a}^*\mathtt{b}^* \subsetneq \{\mathtt{a},\mathtt{b},\mathtt{c}\}$ (above), and the constraint ACCEPT(M) over $\{\sqcup\}$ and $\{\mathtt{a},\mathtt{b},\mathtt{c},\sqcup\}$ (below). The notation $\mathtt{a}_\varepsilon$ (resp. $\mathtt{b}_\varepsilon$) refers to either $\mathtt{a}$ (resp. $\mathtt{b}$) or ε, and the notation $\cdot$ refers to any symbol in $\Sigma_\varepsilon \cup \Gamma_\varepsilon$ that results in a valid transition.

Proof. Write $M = \langle Q, \Gamma, q_0, F, \rightarrow \rangle$. Let $[\![T]\!] = f([\![M]\!])$, where $T = \langle R, A^*, \mathbb{N}, r_0, \Rightarrow, \# \rangle$, $R = Q \cup \{r_0, r_1, r_2, \overset{*}{\bullet}\}$, and $A = \{\sqcup, \varepsilon\} \times (\Gamma \cup \{\sqcup, \varepsilon\})$. We assume that a DTM implementing f writes $[\![T]\!]$ on its output tape by concatenating its six sub-components: $[\![R]\!]$, $[\![A^*]\!]$, $[\![\mathbb{N}]\!]$, $[\![r_0]\!]$, $[\![\Rightarrow]\!]$, and $[\![\#]\!]$. $[\![\mathbb{N}]\!]$ and $[\![r_0]\!]$ can be treated as constant values, while $[\![R]\!]$ and $[\![A^*]\!]$ are constructed by concatenating $[\![Q]\!]$ and $[\![\Gamma]\!]$, respectively, with constant values. Since $[\![Q]\!]$ and $[\![\Gamma]\!]$ can be copied verbatim from $[\![M]\!]$, the components $[\![R]\!]$, $[\![A^*]\!]$, $[\![\mathbb{N}]\!]$, and $[\![r_0]\!]$ can be written to the output tape without using the work tape. To show that f can be computed in logarithmic space, therefore, it suffices to show that $[\![\Rightarrow]\!]$ and $[\![\#]\!]$ can be written to the output tape using at most logarithmic space.

To that end, consider the following procedure for implementing f. We assume that the functions $\Rightarrow$ and $\#$ are represented as lists of input–output pairs. On input $[\![M]\!]$, where $M = \langle Q, \Gamma, q_0, F, \rightarrow \rangle$:

1. Write $[\![R]\!]$, $[\![A^*]\!]$, $[\![\mathbb{N}]\!]$, and $[\![r_0]\!]$ to the output tape, where $R = Q \cup \{r_0, r_1, r_2, \overset{*}{\bullet}\}$ and $A = (\{\sqcup, \varepsilon\} \times (\Gamma \times \{\sqcup, \varepsilon\}))$.

2. Write the transition $\left[\!\left[r_0 \xRightarrow[0]{\langle\sqcup,\sqcup\rangle} r_1 \right]\!\right]$ to the output tape.

3. For each $\langle a, b \rangle \in A$:

(a)   Write $\left[\!\!\left[r_1 \xrightarrow[1]{\langle a,b\rangle} r_2 \right]\!\!\right]$, $\left[\!\!\left[r_2 \xrightarrow[0]{\langle a,b\rangle} r_2 \right]\!\!\right]$, and $\left[\!\!\left[\overset{\circ}{\bullet} \xrightarrow[0]{\langle a,b\rangle} \overset{\circ}{\bullet} \right]\!\!\right]$ to the output tape.

4.   For each state $q \in Q$ and symbol $b \in \Gamma \cup \{\text{␣}, \varepsilon\}$:

    (a)   If M has a transition $q \xrightarrow{b} r$ for some r:

        i.   Write $\left[\!\!\left[q \xrightarrow[0]{\langle \varepsilon,b\rangle} r \right]\!\!\right]$ and $\left[\!\!\left[q \xrightarrow[0]{\langle \text{␣},b\rangle} r \right]\!\!\right]$ to the output tape.

        ii.   If $q = q_0$, then write $\left[\!\!\left[r_0 \xrightarrow[0]{\langle \varepsilon,b\rangle} r \right]\!\!\right]$ and $\left[\!\!\left[r_0 \xrightarrow[0]{\langle \text{␣},b\rangle} r \right]\!\!\right]$ to the output tape.

    (b)   Otherwise:

        i.   If $b = \varepsilon$, then write $\left[\!\!\left[q \xrightarrow[0]{\langle \varepsilon,\varepsilon\rangle} q \right]\!\!\right]$ and $\left[\!\!\left[q \xrightarrow[0]{\langle \text{␣},\varepsilon\rangle} q \right]\!\!\right]$ to the output tape.

        ii.   Otherwise, write $\left[\!\!\left[q \xrightarrow[1]{\langle \varepsilon,b\rangle} \overset{\circ}{\bullet} \right]\!\!\right]$ and $\left[\!\!\left[q \xrightarrow[1]{\langle \text{␣},b\rangle} \overset{\circ}{\bullet} \right]\!\!\right]$ to the output tape.

5.   For each state $q \in R$:

    (a)   If $q \in Q \backslash F$, then write $\left[\!\!\left[q \underset{1}{\Rightarrow} \# \right]\!\!\right]$ to the output tape.

    (b)   Otherwise, write $\left[\!\!\left[q \underset{0}{\Rightarrow} \# \right]\!\!\right]$ to the output tape.

The only information that needs to be stored on the work tape in this algorithm is the looping indices used in Steps 3, 4, 4(a), and 5, which range over $\{\text{␣}, \varepsilon\}$, $\Gamma \cup \{\text{␣}, \varepsilon\}$, Q, R, and $\rightarrow$. Since all the transitions of M are listed explicitly in $[\![M]\!]$, the work tape is not needed for the loop over transitions in Step 4(a), because the input tape head can be used as a looping index (i.e., it can point to the rightmost position of the transition under

consideration at each loop iteration). For the other loop counters, values in Γ and Q can be represented in logarithmic space by identifying each symbol or state with the binary representation of the leftmost position in $[\![M]\!]$ where the symbol or state is mentioned for the first time.   □

6.2 Reduction of INE-DFA to Universal Generation

We now formally present our reductions of INE-DFA to universal generation, which proves that universal generation is PSPACE-hard. We begin with a straightforward proof that INE-DFA is reducible to ug.

Proposition 3
INE-DFA is logspace-reducible to ug. (Thus, ug is PSPACE-hard.)

Proof. It suffices to simply convert a list of DFAs $[\![\langle M_1, M_2, \ldots, M_n\rangle]\!]$ into the tuple $[\![\langle C, \textvisiblespace, \langle 0, 0, \ldots, 0\rangle\rangle]\!]$, where

$$C = \langle \textsc{Accept}(M_1), \textsc{Accept}(M_2), \ldots, \textsc{Accept}(M_n), \textsc{NotBlank}\rangle$$

As we have established, a candidate α can only achieve a perfect violation profile of $C(\alpha) = \langle 0, 0, \ldots, 0\rangle$ if $\alpha^{\triangleright}$ is accepted by all the M_is. By Lemma 5, constructing the $\textsc{Accept}(M_i)$s only requires logarithmic space, assuming that the $\textsc{Accept}(M_i)$s are written to the output tape one at a time. The constraint $\textsc{NotBlank}$ and the UR $\textvisiblespace$ are constant values, and therefore do not require the work tape to generate. The tuple $\langle 0, 0, \ldots, 0\rangle$ representing the perfect violation profile is not constant, however, because it has a length of n. Nonetheless, it can be written using a logarithmically-sized counter on the work tape that counts the number of 0s written from 0 to n.   □

Reducing to UG is somewhat trickier. It is easy to check whether the intersection of DFAs *is* empty by checking whether $\textvisiblespace$ is an optimal SR for the $\textsc{Accept}(M_i)$ and $\textsc{NotBlank}$ constraints. In order to reduce intersection *non*-emptiness to UG, however, it seems prima facie that a logspace reduction algorithm would need to furnish an example of a string that is accepted by all the DFAs, in order to check whether that string is an optimal SR. Thankfully, we can avoid this complication by relying on the following two facts:

- PSPACE = coPSPACE (that is to say, a decision problem is in PSPACE if and only if its negation is in PSPACE), and

- a problem is PSPACE-hard if and only if its negation is coPSPACE-hard.

This implies that the **intersection emptiness problem for DFAs**, a coPSPACE-complete problem, is PSPACE-complete, so it suffices to reduce this problem to UG.

Lemma 6
The language

$$\mathsf{IE\text{-}DFA} = \left\{ [\![\langle M_1, M_2, \ldots, M_n \rangle]\!] \;\middle|\; \text{Each } M_i \text{ is a DFA and } \bigcap_{i=1}^{n} M_i = \varnothing \right\}$$

is logspace-reducible to UG. (Thus, UG is coPSPACE-hard.)

Proof. To reduce $\mathsf{IE\text{-}DFA}$ to UG, we convert a list of DFAs $[\![\langle M_1, M_2, \ldots, M_n \rangle]\!]$ into the tuple $[\![\langle C, \text{␣}, \text{␣} \rangle]\!]$, where

$$C = \langle \textsc{Accept}(M_1), \textsc{Accept}(M_2), \ldots, \textsc{Accept}(M_n), \textsc{NotBlank} \rangle$$

By construction, ␣ is optimal for C and ␣ if and only if $[\![\langle M_1, M_2, \ldots, M_n \rangle]\!] \in \mathsf{IE\text{-}DFA}$; and we have already seen that this conversion can be done using logarithmic space. □

Finally, we discuss the idea of reducing $\mathsf{INE\text{-}DFA}$ to UGFunc. Strictly speaking, the concept of a logspace reduction is only defined for decision problems; since UGFunc does not return binary outputs, it is impossible by definition to reduce $\mathsf{INE\text{-}DFA}$ to UGFunc. Informally, however, it is easy to see that $\mathsf{INE\text{-}DFA}$ can be solved efficiently with oracle access to UGFunc, since one can simply construct the $\textsc{Accept}(M_i)$ and $\textsc{NotBlank}$ constraints, use the oracle to generate an optimal SR for ␣, and check whether this SR is ␣. The time and space requirements of such an algorithm depend on details concerning how the oracle returns its output to the DTM, since by Proposition 2 it is possible that the oracle may return an output of super-polynomial length.

7. Bounded Universal Generation

In this section, we briefly discuss the **bounded universal generation problem for OT** defined in Section 3.3, where the number of constraints is bounded a priori. Since the intersection non-emptiness problem for DFAs is NL-complete when the number of DFAs is bounded a priori (Jones 1975), from the arguments in Section 6 it immediately follows that the bounded versions of ug and UG are NL-hard.

Intuitively speaking, the difference between the bounded and unbounded versions of $\mathsf{INE\text{-}DFA}$ is as follows. Both Kozen (1977) and Jones (1975) decide intersection non-emptiness using a strategy similar to the one presented in Section 5, where a string accepted by all the DFAs is nondeterministically generated. During this process, the NTM needs to keep track of the most recent state of the DFAs. The size of this information is $O(n \log(l))$, where n is the number of DFAs in the input and l is the length of the binary representation of the input. When the number of DFAs is bounded, n is treated as a constant, so $O(n \log(l)) = O(\log(l))$. When the number of DFAs is not bounded, n is approximately linear in l in the worst case, so $O(n \log(l))$ is approximated as $O(l \log(l))$.

This logic cannot be applied to universal generation, however, because unlike DFA states, violation numbers cannot be represented using logarithmically many bits in general. In the following lemma, we derive a polynomial bound on the length of the shortest optimal candidate, but leave open the possibility that the optimal violation bound may contain exponential values.

Lemma 7 (Analog of Lemma 2 for Bounded Universal Generation)
Let x be a UR, let $C = \langle C_1, C_2, \ldots, C_n \rangle$ be a list of constraints, and let $l = |[\![\langle C, x \rangle]\!]|$. Then,

- there is a candidate of length at most $(l/n)^n$ that is optimal for C and x, and

- for all candidates α, if $|\alpha| \leq (l/n)^n$, then for all i, $C_i(\alpha) \leq 2^l (l/n)^n$.

Proof. See Appendix A.   □

By modifying the algorithms in Section 5 according to Lemma 7, we can deduce that the bounded version of ug is in NP, since the generation of an optimal candidate only requires nondeterministic polynomial time. Additionally, we prove here that the bounded version of UG is in NP^{NP}.

Lemma 8
For n fixed, the language

$$\{[\![\langle C, x, v \rangle]\!] \mid |C| = n \wedge \exists \alpha [\alpha^{\lhd} = x \wedge C(\alpha) \leq v]\}$$

is in NP.

Proof. To decide this language in nondeterministic polynomial time, it suffices to check that $|C| = n$, guess a candidate α of length at most $(|[\![\langle C, x \rangle]\!]|/n)^n$, and return 1 if $C(\alpha) < v$ and 0 otherwise.   □

Proposition 4
The language

$$\text{BUG}(n) = \{[\![\langle C, x, y \rangle]\!] \mid |C| = n \text{ and } y \text{ is optimal for } C \text{ and } x\}$$

is in NP^{NP}.

Proof. Consider the following nondeterministic algorithm. On input $[\![\langle C, x, y \rangle]\!]$:

1. Return 0 if $|C| \neq n$. Write $C = \langle C_1, C_2, \ldots, C_n \rangle$.

2. Construct the constraint C_0 shown in Figure 3, which requires optimal candidates α to satisfy $\alpha^{\rhd} = y$.

3. Generate a candidate α of length at most $(|[\![\langle C', x \rangle]\!]|/(n+1))^{n+1}$, where $C' = \langle C_0, C_1, \ldots, C_n \rangle$.

4. Using an oracle for the language in Lemma 8 with $n+1$ constraints, decide whether $C'(\alpha)$ is an optimal violation profile for C' and x. Return 1 if so, and return 0 otherwise.

This algorithm clearly decides $\mathsf{BUG}(n)$, and in Step 4 an oracle for a language in NP is invoked. It therefore remains to show that this algorithm runs in polynomial time. To that end, note that $|[\![C_0]\!]| = O(|y|)$, since C_0 has as many states and transitions as the length of y, plus or minus a constant. Therefore, constructing C_0 only requires polynomial time, and the candidate length bound $(|[\![\langle C', x\rangle]\!]|/(n+1))^{n+1}$ remains polynomial in $|[\![\langle C, x, y\rangle]\!]|$. $\qquad\square$

Because violation numbers cannot be represented in logarithmic space in general, we conjecture here that the bounded versions of ug and UG are not in NL. We cannot prove this conjecture using current techniques, however, since it is still unknown whether $\mathrm{NL} \neq \mathrm{NP}$ or whether $\mathrm{NL} \neq \mathrm{NP}^{\mathrm{NP}}$.

8. Discussion

Our formal results establish universal generation for OT as a maximally difficult problem in the class of polynomial-space computable decision problems. In theory, this means that universal generation for OT is as difficult as solving quantified Boolean formulae (Stockmeyer and Meyer 1973) or playing games such as Othello (Iwata and Kasai 1994), Rush Hour (Flake and Baum 2002), and *The Legend of Zelda: Ocarina of Time* (Aloupis et al. 2015). In this section, we interpret our results by identifying and discussing four properties of OT universal generation that play an important role in our analyses:

- the expressive power of constraint intersection, which places a PSPACE-hard lower bound on OT and related systems;

- the ability of constraints to multiplicatively increase the length of the shortest SR and assign exponentially high violation numbers, which prevents universal generation from being done in nondeterministic polynomial time (assuming $\mathrm{NP} \neq \mathrm{PSPACE}$) or in nondeterministic logarithmic space (assuming $\mathrm{NP} \neq \mathrm{NL}$) in the bounded case;

- the logical structure of ug and UG, which makes the latter more complex than the former in the bounded setting; and

- representational assumptions we have made in this article, which affect our accounting of time and memory resources.

8.1 Expressivity of Constraint Intersection

Our analysis from Section 5 and Section 6 shows that the main contributor to the complexity of universal generation is the ability to intersect arbitrarily many finite-state constraints. Since the DFA intersection emptiness and non-emptiness problems are PSPACE-complete, the ability to intersect constraints gives OT a PSPACE-hard lower bound on the complexity of universal generation. The fact that OT universal generation does not require additional complexity beyond PSPACE implies that the complexity of constraint intersection dominates the complexity of other components of OT such as the constraint ranking mechanism or the ability to optimize violation profiles.

Given this insight, it is not difficult to see that other OT-like formalisms that involve constraint intersection are also PSPACE-complete. For instance, Frank and Satta's (1998) and Chen-Main and Frank's (2003) version of OT, which uses binary-valued constraints

that can assign at most one violation, is PSPACE-complete, since it is transparently reducible in both directions to DFA intersection. The version of Harmonic Grammar (HG) proposed by Pater (2009) and Potts et al. (2010), where constraint interaction is implemented by taking a weighted sum of constraint violations instead of using a constraint ranking mechanism, is also PSPACE-complete, since the analyses presented in Section 5 and Section 6 are just as applicable to HG as they are to OT.

One interpretation of our PSPACE-completeness result is that OT is "too powerful": A theory of phonology should not predict that computing the SR for a UR is computationally intractable when in reality, human speakers have little difficulty producing SRs on the fly. Another interpretation, which we propose here, is that our PSPACE-completeness result reflects the explanatory power that is offered by the method of factoring intricate phonological phenomena into simple constraints on markedness and faithfulness. A typical analysis in OT phonology, like the toy example we gave in Section 3.1, includes a plain-language description of the phenomenon under consideration, followed by a ranked list of proposed constraints that accounts for the phenomenon. We can understand this style of analysis to be a process in which the phonologist composes a compact description of a complex generalization by factoring it into a much simpler, formally clean list of ranked constraints. Under this view, our PSPACE-completeness result validates the explanatory effectiveness of this approach by showing that these compact descriptions have the potential to explain enormously complex phenomena.

8.2 Violation Numbers, Candidate Length, and Grammar Size

A recurring theme in the proofs we have presented is the need to control the maximum value of violation numbers as well as the maximum possible length of the shortest optimal candidate for a UR. In Section 5, we have seen that the reason ug cannot be decided in nondeterministic polynomial time (assuming that $NP \neq PSPACE$) is because the shortest optimal candidate may be longer than polynomial length. Similarly, in Section 7 we were unable to prove that the bounded version of ug could be decided in nondeterministic logarithmic space because remembering violation numbers requires linear space in the worst case. If the length and violation profile of an optimal SR were both subject to polynomial bounds, then the full version of ug would be NP-complete, and the bounded version of ug would be NL-complete.

The reason why long optimal candidates exist can be understood by examining the proofs in Appendix A. Roughly speaking, these arguments show that given a UR x and a list of constraints $C = \langle C_1, C_2, \ldots, C_n \rangle$ where each C_i has q_i states, the length of the shortest optimal candidate for C and x is at most

$$(2 + |x|) \prod_{i=1}^{n} q_i$$

This means that in the worst case, each constraint C_i multiplies the length of shortest optimal candidates by a factor of q_i, causing candidate length to grow exponentially in the number of constraints in the grammar.

The reason for the existence of large violation numbers, on the other hand, is attributed to the compactness of the bit-string representation of integers. Since a sequence of n bits can represent an integer of value up to 2^n, it is difficult to avoid the possibility of exponential violation numbers incurred by a candidate. One possible approach for doing so would be to use a unary representation of integers on the input tape, but

a binary representation on the work tape. The use of unary numbers is not without precedent in OT phonology, since the visualization of tableaux typically uses a unary representation of violation numbers (e.g., "∗ ∗ ∗" represents a violation number of 3). If such a representation is used, then violation numbers only require logarithmic space on the work tape, making bounded universal generation NL-complete.

8.3 Logical Structure of Optimization

Another recurring theme is the use of nondeterminism in our proofs. For example, our proof that ug $\in$ PSPACE actually shows that ug $\in$ NPSPACE by guessing a candidate that is at least as optimal as the violation bound given. We argue here that our use of nondeterminism reflects the logical structure of the universal generation problem. In complexity theory, nondeterministic complexity classes typically correspond to decision problems whose statements involve existential quantification. For instance, the Hamiltonian path problem, an NP-complete problem, asks whether *there exists* a path in a graph that visits all the vertices. Similarly, the statement of ug also involves existential quantification: ug asks whether *there exists* a candidate α such that $\alpha^{\triangleleft} = x$ and $C(\alpha) \leq v$. On the other hand, the complements of nondeterministic classes correspond to universal quanitification. For example, the complement of the Hamiltonian path problem, a coNP-complete problem, asks whether *for all* paths in a graph, at least one vertex is not visited. In Lemma 6, the problems IE-DFA and UG both involve universal quantification: IE-DFA asks whether *all* strings are rejected by at least one DFA, and UG asks whether *all* SRs are less optimal than the one given in the input.

Although ug and UG are both PSPACE-complete, the latter is logically more complex than the former, in the sense that the statement of UG involves an alternation of quantifiers. Whereas ug merely asks whether *there exists* a candidate α for x such that $C(\alpha) \leq v$, UG asks whether *there exists* a candidate α such that $\alpha^{\triangleleft} = x$, $\alpha^{\triangleright} = y$, and *for all* candidates β with $\beta^{\triangleleft} = x$, $C(\alpha) \leq C(\beta)$. This discrepancy in logical complexity is not captured by PSPACE, since PSPACE is closed under quantifier alternation in an appropriate sense (see Arora and Barak 2009, Chapter 5, for details); but it is reflected in the *bounded* versions of these problems. As we showed in Section 7, the bounded version of ug is in NP, which corresponds to existential quantification, while the bounded version of UG is in NP^{NP}, which corresponds to problems with a single $\exists\forall$ alternation.

8.4 Representations

Finally, we briefly discuss the impact of representation on our complexity analysis. Our representational assumptions for bit strings are based on convention in complexity theory: numbers, states, and alphabet symbols are represented as binary strings of logarithmic length; tuples are represented by concatenation of their elements; and finite functions are represented as lists of input–output pairs. Abstracting away from bit strings, our representation of phonological objects, particularly the Correspondence-Theoretic representation of candidates as strings of symbol pairs, largely follows the assumptions of prior literature such as Chen-Main and Frank (2003), Riggle (2004), and Hao (2019). These representational choices have a measurable impact on our complexity results: for instance, as discussed in Section 8.2, using a tableau-style unary representation of violation numbers would make the bounded universal generation problem NL-complete. More dramatic effects on complexity may be observed when using sophisticated representations designed to account for suprasegmental phenomena. Lamont (2023), for instance, shows that the undecidable Post Correspondence Problem

(Post 1946) is Turing-reducible to OT universal generation when candidates are represented as autosegmental structures (Goldsmith 1976).

9. Conclusion

In this article, we have obtained several theoretical results regarding the computational complexity of OT. Namely, we have shown that OT universal generation is PSPACE-complete, while bounded universal generation is at least NL-hard and at most NP$^{\text{NP}}$-hard. The close relationship between OT universal generation and the intersection non-emptiness problem for DFAs shows that our complexity lower bounds are almost entirely attributable to the expressive power of automaton intersection, which allows OT to produce concise, elegant explanations of sophisticated phonological phenomena. However, more careful inspection of our proof techniques as well as our results for bounded universal generation reveals that candidate length, violation numbers, and the logical structure of optimization problems also contribute to the time and memory requirements of OT algorithms.

Appendix A. Optimal Candidate Length and Violations

This appendix presents the proofs of Proposition 2 and Lemma 2 from Section 5.1. These results are restated below.

Proposition 2
For every polynomial $f(n)$, there is a UR x and a list of constraints C such that

- there is a candidate α such that $\alpha^{\lhd} = x$ and $C(\alpha) = \langle 0, 0, \dots, 0 \rangle$, and

- for all candidates α with $\alpha^{\lhd} = x$, $C(\alpha) = \langle 0, 0, \dots, 0 \rangle$ only if
  $|\alpha| > f(|[\![\langle C, x, v \rangle]\!]|)$.

Lemma 2
Let x be a UR, let $C = \langle C_1, C_2, \dots, C_n \rangle$ be a list of constraints, and let $l = |[\![\langle C, x \rangle]\!]|$. Then,

- there is a candidate of length at most le^l that is optimal for C and x, and

- for all candidates α, if $|\alpha| \leq le^l$, then for all i, $C_i(\alpha) \leq le^{2l}$.

The interpretation of Proposition 2 and Lemma 2 is that they set an exponential upper bound on the length and violation profile of the shortest SR for a list of constraints and a UR. We begin with a straightforward proof of Proposition 2.

Proof of Proposition 2. Let $x = \varepsilon$, and for each $k \geq 1$, let $v_k = \langle 0, 0, \dots, 0 \rangle$ be the zero vector of length k. For $i > 1$, let MOD(i) and NOTEMPTY be constraints over $\Sigma = \Gamma = \{\texttt{a}\}$ defined as follows.

- MOD(i): Assign one violation to candidate α if $|\alpha^{\rhd}|$ is not a multiple of i.

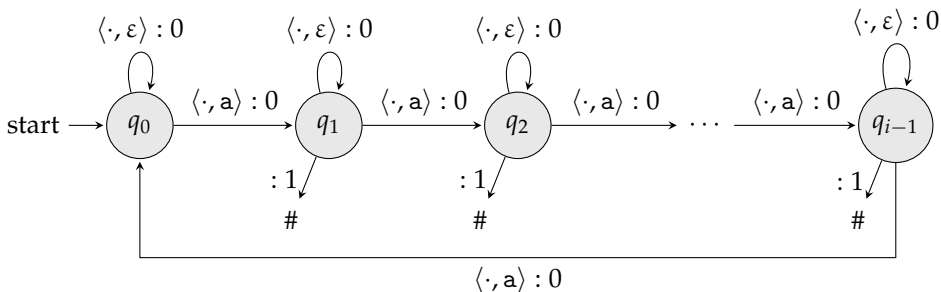- NOTEMPTY: Assign one violation to candidate α if $|\alpha^{\rhd}| = 0$.

Figure A.1
The constraint MOD(i), which assigns one violation when the SR's length is not a multiple of i.
The notation $\cdot$ refers to any symbol in $\Sigma_\varepsilon \cup \Gamma_\varepsilon$ that results in a valid transition.

Now, let p_i denote the ith prime number. (Thus, $p_1 = 2$, $p_2 = 3$, etc.) For $k \geq 1$, let

$$C_k = \langle \text{NOTEMPTY}, \text{MOD}(p_1), \text{MOD}(p_2), \ldots, \text{MOD}(p_k) \rangle.$$

Observe that $C_k(\alpha) \leq v_k = \langle 0, 0, \ldots, 0 \rangle$ only if $\alpha^{\triangleright} \in (a^{p_1 p_2 \cdots p_k})^*$. Since NOTEMPTY eliminates ε as a possible optimal candidate, the length of α must therefore satisfy

$$|\alpha| \geq p_1 p_2 \ldots p_k \geq f(|[\![\langle x, C_k, v_k \rangle]\!]|)$$

for k large enough.   □

To prove Lemma 2, we use a strategy based on Riggle's (2004) approach to universal generation, where optimal SRs are generated by finding the shortest path through the state diagram of an SFST that computes the violation profile for a candidate while ensuring that the candidate corresponds to the intended UR. The length of the shortest optimal candidate is then bounded above by the pumping length of this SFST. Furthermore, a bound on the optimal violation profile is obtained by observing that an SFST can only assign a linear number of violations to a candidate, since each transition is associated with a constant number of violations assigned.

To derive the specific mathematical formulae appearing in Lemma 2, we introduce a technical lemma that relates the total number of states in a collection of SFSTs with the pumping length of the intersection of the SFSTs.
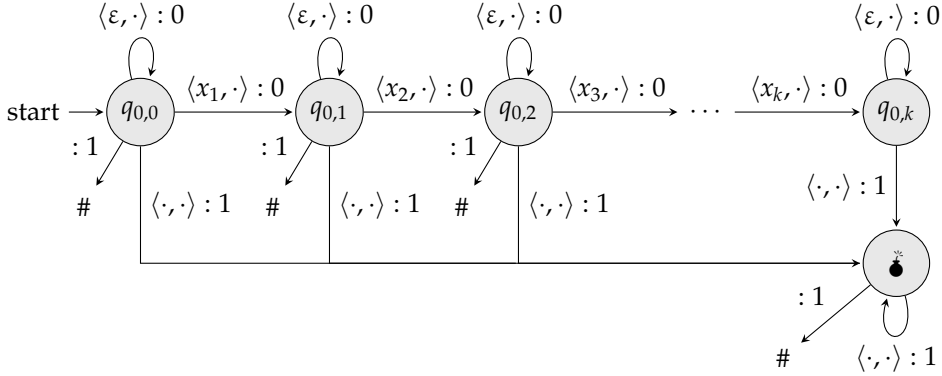
Lemma 9
Fix $l \in \mathbb{N}$. For $q_1, q_2, \ldots, q_n \in \mathbb{N} \backslash \{0\}$, if $q_1 + q_2 + \cdots + q_n \leq l$, then $q_1 q_2 \ldots q_n \leq e^{l/e}$.

Proof. We use strong induction on l. When $l = 1$, we have $n = 1$ and $q_1 = 1$; thus,

$$q_1 q_2 \ldots q_n = q_1 = 1 < e^{1/e} \approx 1.44$$

Now fix l, and assume that Lemma 9 holds for all $j < l$. If $q_1 + q_2 + \cdots + q_n \leq l$, then

$$q_1 q_2 \ldots q_n \leq e^{(l - q_n)/e} q_n$$

Figure A.2
A constraint that assigns one or more violations to a candidate α when $\alpha^\lhd \neq x = x_1 x_2 \ldots x_k$. The notation $\cdot$ refers to any symbol in $\Sigma_\varepsilon \cup \Gamma_\varepsilon$ that results in a valid transition.

The proof is complete if we can show that $q_n \leq e^{q_n/e}$. This follows easily, however, from the observations that $1 < e^{1/e}$ and $e^{l/e}$ is exponential in l. $\qquad\square$

Proof of Lemma 2. We begin by constructing an SFST $C_\cap$ takes a candidate α as input and outputs a tuple $C_\cap(\alpha) = \langle v_0, v_1, \ldots, v_n \rangle$, where $C(\alpha) = \langle v_1, v_2, \ldots, v_n \rangle$ and $v_0 = 0$ if and only if $\alpha^\lhd = x$. Let C_0 be the constraint illustrated in Figure A.2, which computes v_0. For each $i \in \{0, 1, \ldots, n\}$, write $C_i = \langle Q_i, \Sigma_\varepsilon \times \Gamma_\varepsilon, \mathbb{N}, q_{i,0}, \to_i, \#_i \rangle$. Let $C_\cap = \langle Q_\cap, \Sigma_\varepsilon \times \Gamma_\varepsilon, \mathbb{N}^k, q_0, \to, \# \rangle$ be defined as follows.

- $Q_\cap = Q_0 \times Q_1 \times \cdots \times Q_n$

- $q_0 = \langle q_{0,0}, q_{1,0}, \ldots, q_{n,0} \rangle$

- $\langle q_{0,j_0}, q_{1,j_1}, \ldots, q_{n,j_n} \rangle \xrightarrow[\langle v_0, v_1, \ldots, v_n \rangle]{\langle a,b \rangle} \langle q_{0,j_0'}, q_{1,j_1'}, \ldots, q_{n,j_n'} \rangle$ if and only if for all
  $i \in \{0, 1, \ldots, n\}$, $q_{i,j_i} \xrightarrow[v_i]{\langle a,b \rangle} q_{i,j_i'}$

- $\#(\langle q_{0,j_0}, q_{1,j_1}, \ldots, q_{n,j_n} \rangle) = \langle \#_0(q_{0,j_0}), \#_1(q_{1,j_1}), \ldots, \#_n(q_{n,j_n}) \rangle$

$C_\cap$ is simply the intersection of $C_0, C_1, \ldots, C_n$, where transition outputs of each C_i are concatenated together into tuples of violation numbers.

In order for a candidate α to be a valid candidate for x (i.e., $\alpha^\lhd = x$), it is necessary and sufficient for $C_\cap$ to end its computation on one of the states in $F = \{q_{0,k}\} \times Q_1 \times Q_2 \times \cdots \times Q_n$ on input α (i.e., C_0 must end on state $q_{0,k}$). Therefore, let $\alpha = \alpha_1 \alpha_2 \ldots \alpha_m$ be a candidate such that

$$q_0 \xrightarrow[c_1]{\alpha_1} q_1 \xrightarrow[c_2]{\alpha_2} \cdots \xrightarrow[c_m]{\alpha_m} q_m$$

where $q_m \in F$. We need to show that there is such an α that is optimal while satisfying $m \leq le^l$ and $c = c_1 + c_2 + \cdots + c_m \leq le^{2l}$.

To obtain the bound on m, first note that without loss of generality, we can assume that $q_i \neq q_j$ whenever $i \neq j$; that is to say, $C_\cap$ never enters any state more than once. This because if $q_i = q_j$ with $i \neq j$, then we would have

$$q_0 \xrightarrow[c_1]{\alpha_1} q_1 \xrightarrow[c_2]{\alpha_2} \cdots \xrightarrow[c_i]{\alpha_i} q_i \xrightarrow[c_{i+1}]{\alpha_{j+1}} q_{j+1} \xrightarrow[c_{j+2}]{\alpha_{j+2}} q_{j+2} \cdots \xrightarrow[c_m]{\alpha_m} q_m$$

In this case the shorter candidate $\alpha' = \alpha_1 \alpha_2 + \cdots + \alpha_i \alpha_{j+1} \alpha_{j+2} \ldots \alpha_m$ would be at least as optimal as α, since

$$C(\alpha') = C(\alpha) - (c_{i+1} + c_{i+2} + \cdots + c_j)$$
$$\leq C(\alpha)$$

Now, assuming that $C_\cap$ never enters any state more than once on input α, it follows that $m \leq |Q_\cap|$. Assuming that each state in each Q_i is represented by at least one bit of $[\![\langle C, x \rangle]\!]$, we have

$$|Q_1| + |Q_2| + \cdots + |Q_n| \leq l$$

so using Lemma 9 we can compute

$$|Q_\cap| = |Q_0| \times |Q_1| \times \cdots \times |Q_n|$$
$$\leq (|x| + 2)e^{l/e}$$
$$\leq le^{l/e}$$
$$\leq le^l$$

thus $m \leq le^l$.

Now, to obtain a bound on c, we observe that no constraint in C can assign more than 2^l violations in a single transition, assuming that numbers are represented in binary form. Therefore, writing $c = \langle v_0, v_1, \ldots, v_n \rangle$, for each i we have

$$v_i \leq 2^l m$$
$$\leq 2^l \cdot le^l$$
$$\leq e^l \cdot le^l$$
$$= le^{2l}$$

$\square$

In Section 7, we stated an alternate version of Lemma 2, reproduced below, in which a polynomial bound on the length of the shortest optimal candidate is obtained by treating the number of constraints as a constant.

Lemma 7
Let x be a UR, let $C = \langle C_1, C_2, \ldots, C_n \rangle$ be a list of constraints, and let $l = |[\![\langle C, x \rangle]\!]|$. Then,

- there is a candidate of length at most $(l/n)^n$ that is optimal for C and x, and

- for all candidates α, if $|\alpha| \leq (l/n)^n$, then for all i, $C_i(\alpha) \leq 2^l (l/n)^n$.

To obtain these bounds, it suffices to use the same proof as in Lemma 2, but with the following alternate version of Lemma 9.

Lemma 10
Fix $l, n \in \mathbb{N}$. For $q_1, q_2, \ldots, q_n \in \mathbb{N}\backslash\{0\}$, if $q_1 + q_2 + \cdots + q_n \leq l$, then $q_1 q_2 \ldots q_n \leq (l/n)^n$.

Proof. Without loss of generality, assume that $q_1 + q_2 + \cdots + q_n = l$. We prove a somewhat different statement: if $q_i \neq q_j$ for some i and j, then

$$q_i q_j < \left(\frac{q_i + q_j}{2}\right)^2 \tag{A.1}$$

This strict inequality implies that the maximum value of $q_1 q_2 \ldots q_n$ is attained when all q_is have the same value. If real values of q_i are allowed, then the maximum value of $q_1 q_2 \ldots q_n$ is $(l/n)^n$, hence the lemma.

To prove Inequality A.1, simply observe that

$$(q_i + q_j)^2 - (q_i - q_j)^2 = q_i^2 + 2q_i q_j + q_j^2 - q_i^2 + 2q_i q_j - q_j^2$$
$$= 4q_i q_j$$

thus

$$q_i q_j = \frac{(q_i + q_j)^2 - (q_i - q_j)^2}{4}$$
$$< \frac{(q_i + q_j)^2}{4} \qquad\qquad \text{assuming } q_i \neq q_j$$
$$= \left(\frac{q_i + q_j}{2}\right)^2$$

$\square$

References
Aloupis, Greg, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. 2015. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160. https://doi.org/10.1016/j.tcs.2015.02.037

Arora, Sanjeev and Boaz Barak. 2009. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge, United Kingdom. https://doi.org/10.1017/CBO9780511804090

Chen-Main, Joan and Robert Frank. 2003. Implementing faithfulness constraints in a finite state model of optimality theory. In *Proceedings of the 14th Irish Conference on*

Artificial Intelligence and Cognitive Science, pages 28–33.

Chomsky, Noam and Morris Halle. 1968. *The Sound Pattern of English*, first edition. Harper & Row, New York, NY, USA.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271. https://doi .org/10.1007/BF01386390

Eisner, Jason. 1997. Efficient generation in primitive optimality theory. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 313–320. https://doi.org/10 .3115/976909.979657

Eisner, Jason. 2000a. Directional constraint evaluation in optimality theory. In *Proceedings of the 18th Conference on Computational Linguistics*, volume 1, pages 257–263. https://doi.org/10 .3115/990820.990858

Eisner, Jason. 2000b. Easy and Hard Constraint ranking in OT: Algorithms and complexity. In *Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology*, pages 22–33.

Ellison, T. Mark. 1994. Phonological derivation in optimality theory. In *Proceedings of the 15th Conference on Computational Linguistics*, volume 2, pages 1007–1013. https://doi.org/10 .3115/991250.991312

Flake, Gary William and Eric B. Baum. 2002. Rush hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1):895–911. https://doi.org/10.1016/S0304 -3975(01)00173-6

Frank, Robert and Giorgio Satta. 1998. Optimality theory and the generative complexity of constraint violability. *Computational Linguistics*, 24(2):307–315.

Gerdemann, Dale and Mans Hulden. 2012. Practical finite state optimality theory. In *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing*, pages 10–19.

Goldsmith, John Anton. 1976. *Autosegmental Phonology*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

Hao, Yiding. 2019. Finite-state optimality theory: Non-rationality of harmonic serialism. *Journal of Language Modelling*, 7(2):49–99. https://doi.org/10.15398 /jlm.v7i2.210

Heinz, Jeffrey, Gregory M. Kobele, and Jason Riggle. 2009. Evaluating the complexity of

optimality theory. *Linguistic Inquiry*, 40(2):277–288. https://doi.org/10 .1162/ling.2009.40.2.277

Idsardi, William J. 2006. A simple proof that optimality theory is computationally intractable. *Linguistic Inquiry*, 37(2):271–275. https://doi.org/10 .1162/ling.2006.37.2.271

Iwata, Shigeki and Takumi Kasai. 1994. The Othello game on an $n \times n$ board is PSPACE-complete. *Theoretical Computer Science*, 123(2):329–340. https://doi.org /10.1016/0304-3975(94)90131-7

Johnson, C. Douglas. 1970. *Formal Aspects of Phonological Description*. Ph.D. thesis, University of California, Berkeley, Berkeley, CA, USA.

Johnson, C. Douglas. 1972. *Formal Aspects of Phonological Desription*. Mouton, The Hague, Netherlands. https://doi.org /10.1515/9783110876000

Jones, Neil D. 1975. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85. https://doi.org /10.1016/S0022-0000(75)80050-X

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Karttunen, Lauri. 1998. The proper treatment of optimality in computational phonology. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 1–12. https://doi.org /10.3115/1611533.1611534

Kozen, Dexter. 1977. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*, pages 254–266, IEEE, Providence, RI, USA. https://doi.org /10.1109/SFCS.1977.16

Lamont, Andrew. 2023. Optimality Theory is not computable. Colloquium talk at Atelier de phonologie, CNRS SFL Laboratory, Paris, France.

McCarthy, John J. and Alan Prince. 1995. Faithfulness and reduplicative identity. *University of Massachusetts Occasional Papers in Linguistics*, 18: Papers in Optimality Theory:249–384.

Pater, Joe. 2009. Weighted constraints in generative linguistics. *Cognitive Science*, 33(6):999–1035. https://doi.org/10 .1111/j.1551-6709.2009.01047.x, PubMed: 21585494

Post, Emil L. 1946. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*,

52(4):264–268. `https://doi.org/10.1090/S0002-9904-1946-08555-9`

Potts, Christopher, Joe Pater, Karen Jesney, Rajesh Bhatt, and Michael Becker. 2010. Harmonic Grammar with linear programming: From linear systems to linguistic typology. *Phonology*, 27(1):77–117. `https://doi.org/10.1017/S0952675710000047`

Prince, Alan and Paul Smolensky. 1993. Optimality theory: Constraint interaction in generative grammar. Technical Report 2, Rutgers University, New Brunswick, NJ, USA.

Prince, Alan and Paul Smolensky. 2004. *Optimality Theory: Constraint Interaction in Generative Grammar*. Blackwell Publishing, Malden, MA, USA. `https://doi.org/10.1002/9780470759400`

Rabin, M. O. and D. Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125. `https://doi.org/10.1147/rd.32.0114`

Riggle, Jason Alan. 2004. *Generation, Recognition, and Learning in Finite State Optimality Theory*. Ph.D. thesis, University of California, Los Angeles, Los Angeles, CA, USA.

Savitch, Walter J. 1970. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192. `https://doi.org/10.1016/S0022-0000(70)80006-X`

Sipser, Michael. 2013. *Introduction to the Theory of Computation*, third edition. Cengage Learning, Boston, MA, USA.

Stockmeyer, L. J. and A. R. Meyer. 1973. Word problems requiring exponential time: Preliminary report. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 1–9. `https://doi.org/10.1145/800125.804029`

Wareham, Harold Todd. 1998. *Systematic Parameterized Complexity Analysis in Computational Phonology*. Ph.D. thesis, University of Victoria, Victoria, Canada.