

Finite-State Optimality Theory: A Computational Package for Contender Generation

James Waugh

Department of Linguistics, The University of Chicago

May 2, 2018

Abstract

Optimality Theory (Prince and Smolensky 1993) is a fashionable typological theory for phonology, and OT accounts of phonological grammars are popular in the literature. Riggle (2011) provides an computational model of OT, borrowing finite-state machines from the earlier literature and connecting the problem to the computation of Pareto frontiers. I introduce a brief history of the problems OT is intended to address, of the use of finite-state models in computational phonology, and present a mathematical treatment from first principles of Riggle’s model as well as a Python package and some examples of its use treating problematic tonal data.

1 Introduction

1.1 Overview

It is the aspiration of many to bring robust mathematical models to the field of linguistics, and Optimality Theory (henceforth, “OT;” Prince and Smolensky 1993) has, despite some theoretical hurdles and substantial criticism, been quite fashionable for the last twenty-five years. By treating the phonological knowledge of a speaker as a mathematical optimization procedure, they have opened up a computationally interesting problem: because the theory, following Chomsky’s insistence that “it is not incumbent on a grammar to compute,” does not prescribe how to compute the segments it rules grammatical, the question of computational models and their efficiency naturally presents itself.

As it turns out, Prince and Smolensky’s model of constraint satisfaction goes beyond the bounds of linguistic theory. In the field of multi-criteria decision-making, one speaks of the *Pareto frontier* formed by a set of constraints: this frontier is in fact a more general case of the collection of what Riggle (2004) calls *contenders*, these being the possibly-optimal candidates in an OT model. By following questions that arise (even if they have become somewhat vulgar) in linguistic models, we can produce mathematically interesting results that escape the theoretical origins.

This thesis delves into the work of Riggle (2011) in employing the mathematical machinery of finite-state machines, which have a robust legacy in both linguistics and computer science, to model OT constraint satisfaction. I present both an abstract mathematical treatment of the problem as it’s presented in the linguistics literature and an implementation in Python as well as a short example of real computations.

1.2 Generative Origins and Rewrite Rules

Chomsky’s generative linguistics sets out to account for the linguistic competence of idealized natural language speakers. The program does this by building *generative grammars* that give maximally parsimonious, fully symbolic descriptions of a set of underlying forms as

well as transformational rules to account for the surface forms. This remains the dominant paradigm. His work, “Aspects of the Theory of Syntax,” proposes to do this for phonology, syntax, and semantics separately. In the case of phonology, we refer to the 1968 work “The Sound Patterns of English,” which formulated the transformational rules of phonology to take the following form:

$$A \rightarrow B / C_D$$

Here the rule specifies that an underlying form A corresponds to a surface form B when it occurs before D and after C .

Chomsky also first stipulated the principle of Economy: the grammar deemed to be explanatorily accurate is the one that is most parsimonious, that is, the one that is easiest for the language learner to acquire. This principle, analogous to a loss function for modern statistical models, remains deeply theoretically influential. In particular, it’s used contemporarily to scaffold the theory of learnability, and this thesis among countless other works will count the algorithmic efficiency of OT as a measure of its quality as a model.

Rewrite rules have, of course, seen mathematical investigation. Kaplan and Kay (1994) claims that C. Douglas Johnson proved in 1972 that SPE-style rewrite rules, that is, those for which C and D describe regular languages, are themselves regular relations, and give a formal proof of this fact in their own paper. Because finite systems of SPE-style rewrite rules have been empirically demonstrated to give full characterizations of natural language phonologies (and indeed, were for years the standard account given in descriptive grammars), linguists today are comfortable declaring all of phonology to be a regular or subregular module of human language faculty.

Applying entire systems of rewrite rules is computationally tractable and doing so established a complexity bound on natural phonological grammars. Whereas rewrite rules can’t on their face be “composed” in any meaningful sense, hence systems of them must be applied by hand, it is known that the class of regular expressions corresponds to a subclass of the category of mathematical objects called *finite-state transducers*, cousins of the Turing machine, and this subclass consists of objects that can be canonically composed to compute

multiple rules simultaneously. Kaplan and Kay independently discovered algorithms for constructing transducers from individual SPE-style rewrite rules. As for composing the rules, two computational models were proposed: sequential, “vertical” application of the rules, which seems to predict that phonology is allowed to have multiple “layers;” and simultaneous, “horizontal” application as proposed by Koskeniemi (1983). Koskeniemi constructed a “two-layer” model for Finnish morphology (not phonology), but constructed the automata “painstakingly” by hand. Nonetheless, he demonstrated that morphology needs only two layers, and Karttunen, Koskeniemi, and Kaplan (1987) present a full-fledged computational model of two-layer *phonology* built entirely upon regular relations on strings. (Karttunen laid the issue to rest and directly proved that the models are computationally equivalent.) The formalism underlying the software became known as the “Xerox calculus,” a formal system that establishes the complexity bound of “regular” on phonological grammars.

Naturally, the Xerox calculus inspires one to think in terms of *possible* phonological grammars. Because as mathematical objects they simply constitute regular relations, it’s entirely sensible to speak of the formal language whose elements are whole phonological grammars. (Unsurprisingly, we expect that it should be regular.) This allows us to ask computational questions about phonological *typology* rather than about individual phonological relations themselves, and thereby further investigation of linguistic universals. For instance, which model of phonological typology has the most efficient generation algorithm? Whereas the SPE accounts of phonological grammars are descriptively complete, the vulgar question has been repeated for the last fifty years, notably by Kiparsky and others: “what are the possible rewrite rules?” Thus a real desideratum in phonological theory is a restrictive yet explanatorily adequate typological model with clear learnability.

1.3 Optimality Theory

There exists an alternative approach to producing explanatorily accurate grammars. Chomsky famously insists that the goal of a grammar is not to provide an algorithm for selecting a candidate, and from a certain perspective it is naïve to take the systems of

rewrite rules as actual instructions for a machine to run. After all, the brain is not a Turing machine, and we cannot evaluate the efficiency of a grammar by counting the symbols used to represent it in a certain way. Instead, grammars can be built around constraints, which can instead of positively identifying grammatical items, merely assert which candidates are invalid, and, with adequate complexity, implicitly give rise to grammatical structures. Enter Optimality Theory.

Prince and Smolensky (1993) assert that there in fact exists a small universal set of rules called *constraints* that are present in every language, but which are ranked and obey the property of *violability*. It proposes a function GEN that generates all possible surface representations corresponding to a given underlying representation, and a function CON that assigns each candidate a ranked list of constraint violations. OT deems a surface representation *optimal* with respect to a given underlying representation if, among all outputs of GEN at the un, it violates the highest-ranked constraint the fewest times. According to OT, what differs between languages is only the ordering of constraints, which implies naïvely that the number of possible phonologies is factorial in the number of universal phonological constraints. This is powerful: with only ten constraints, simple permutation predicts over three million possible grammars.

Indeed, OT is too powerful. The theory seems to demand that constraints “count” violations (for instance, if a constraint bans epenthesis, adding two segments should *ceteris paribus* be less optimal than epenthesizing one segment) and that they distinguish between arbitrary numbers of violations. Such “gradient constraints,” as they’re called, provably generate non-regular grammars if they’re permitted within OT. Gerdemann and van Noord (2000) provide a proof of this fact, attributed to Hiller. Frank and Satta (1998) declare that the computational power of a system that does allow these constraints is an open question and conclude that OT is “too rich in generative capacity,” and must be “localized” (rather than predicting a “global” solution) to a bounded number of violation distinctions for it to fit “structural domains of bounded complexity” such as human phonology. Thus, the present task is to axiomatize some minimally complex subsystem of Optimality Theory that

correctly models the empirical diversity of natural phonologies.

Ellison (1995) presents an important formalization of OT. He calls it “primitive Optimality Theory,” or “OTP,” and predicates it upon “three idealizing assumptions.” OTP constraints must be binary, rather than weighted, and must be a regular relation. This has become accepted in the literature: Karttunen (1998) proposes that, if the number of possible violations is bounded, “counting” constraints can be unwrapped into sets of binary constraints, each of which tests for a specific number of violations, and Frank and Satta (1998) claim that “nearly all” of the constraints proposed in OT literature seem to be representable by regular relations. Additionally, in OTP, GEN must also define a regular language at each underlying representation. Ellison, in the same paper, provides a mathematical object he calls an *optimality system* along with algorithms for actually evaluating optimality. Eisner (1997) improved Ellison’s algorithm and applied it to predict surface representations taking the form of a generalization of Goldsmith’s (1976, 1990) theory of autosegmental phonology that Eisner calls “timelines.” (Bird and Ellison (1992, 1994) earlier proved that finite-state machines can account for autosegmental phonology.) He observes that his improved algorithm is NP-hard in the size of the grammar. (As will be seen later, this strongly and somewhat problematically motivates OT models to stick to string-based representations.) Karttunen (1998) also credits the paper with developing a “typology” of OT constraints.

The issue of counting persists. Frank and Satta (1998) conclude that a finite co-domain for the phonological relation, that is, the direct consequence of imposing a finite bound on the number of violations a constraint can incur, is in fact the *weakest* hypothesis on the phonological relation that allows it to remain regular. This seems to suggest that localization of OT of some variety is necessary in any typological model. Going forward, questions of minimizing computational and mathematical complexity may suggest that localization is better understood as an algorithm independent of phonological optimization.

Such questions have indeed emerged. Samek-Lodovici and Prince (2002) use the term “harmonically bounded” to refer to candidate surface representations that cannot be optimal

under any ranking of constraints, and Riggle (2004) introduces the term “contender” to refer to candidates that can win under some ranking of the constraints. Riggle (2004) also presents the phonological optimization problem in terms of weighted transducers and solves it with a shortest-weighted-path algorithm adapted from Dijkstra (1959) but notes that its computational complexity is unknown. Riggle (2011) presents an improved algorithm for computing contenders along with an analysis of its complexity.

1.4 The Shortcomings of OT

One of the great shortcomings of OT is that it cannot capture *opacity*, a salient feature of rewrite-rule grammars. The success of SPE-style rewrite rules relies in a large capacity upon the fact that rule *A* can capture a linguistically significant generalization, rule *B* can capture a linguistically significant generalization, and that the serial application of *A* to *B* can produce a generalization found in the same language. The compositionality of rewrite rules, of course, is totally absent in a strict two-level system such as OT. Moreover, there seem to be cases where the rule-intermediate representation is crucial to any sensible explanation of a phonological derivation. McCarthy (1997) presents compelling evidence from Tiberian Hebrew where epenthesis seems inexplicable from OT constraints alone, or at least from constraints that “seem natural,” which is the purported conceptual strength of the theory.

It’s worth bearing in mind that this shortcoming is actually only one to which models restricted to faithfulness and markedness constraints succumb. Because of the wild popularity of such models, I will use one of this genus in the mathematical treatment that follows. However, so-called PARSE/FILL models prevalent in the earlier OT literature can account for true alternation (Riggle, PC). The examples I’ll pursue in the following will somewhat resemble this precise issue.

1.5 The Present Task

Having said this, we understand a natural language to have two phonological alphabets: one, of underlying symbols, which represent all the phonological information that

morphology and syntax can specify; and another, of surface symbols, which specify all the phonological information necessary for phonetic processes to produce an unambiguous articulation. Mathematically speaking, then, given such alphabets, a natural language seems to have a formal language UR of underlying representations and a formal language SR of surface representations, and phonology is precisely modeled by the relation $\sim \subseteq UR \times SR$.

I will take this and run with it: in a few chapters, I'll formalize the notion of a phonological grammar and prove that the finite-state transducers in Riggle (2011) are equipped, if adequately localized, to compute phonological grammars. The mathematical exposition will bear out similarities to general multi-constraint decision making theory, and to other areas of mathematics, putting an otherwise linguistics-bound model “out in the open.” I'll proceed to document a Python package I have written to implement Riggle's contenders algorithm and, finally, narrate an investigation of problematic tonal data that will both demonstrate the utility of the package and bear out the representational and learnability issues with OT and tonal phenomena.

2 Phonology is Regular

2.1 Overview

The first task, if we're to justify an OT algorithm, is to prove a bound on the formal complexity of phonological grammar itself. That is, we need to make mathematically rigorous the behavior of phonology as a mental module. This will bear out the triumph of structuralism: because we have solid evidence for discrete, symbolic categories actually being treated as such by some sort of unconscious process, we can make structural statements about this process by analogizing it to other symbol-processing machines. This, as discussed, has its history. The model we will subscribe to is the “two-layer” idealization of phonology, whereby we posit that the phonological language module in the mind takes in an “input tape” of morphophonemes, runs some sort of black-box process, and produces an “output tape” of proper phonemes. An alternative model is the “multi-layer” approach, whereby there are multiple intermediate tapes, each resulting from the application of a different process, however Karttunen proved the two to be equivalent.

The first concept we need to employ is the complexity of a formal language. Specifically, we want to define and prove the following proposition, as this will justify the entire computational approach presented in this paper:

Proposition 2.1. *The relation defined by any phonological grammar is regular.*

This is an essential constraint on any typological theory of phonological grammar, and Optimality Theory will adapt comfortably enough to accommodate it.

2.2 Preliminary Definitions

Mathematics now becomes necessary to rigorously define the object of a regular relation. Recall the set-theoretic characterization of abstract relations and functions:

Definition 2.2. A *relation* \sim between sets X and Y is simply a subset of their Cartesian product: $\sim \subseteq X \times Y$. A *function* is simply a relation $f \subseteq X \times Y$ that associates each

element $x \in X$ to one unique value, denoted $f(x) \in Y$.

Additionally, to understand anything about the “tapes” we’ve discussed so far, it’s necessary to employ terminology from formal language theory. In particular, the following objects will be indispensable:

Definition 2.3. We will say the following:

- An *alphabet* Σ is a set;
- the elements of an alphabet are called *symbols*;
- a *string* s in Σ is a function $s : [1..n] \rightarrow \Sigma$, where $[1..n]$ denotes the subset consisting of the first n natural numbers;
- if s is a string, denote $s(k)$ by s_k ;
- the *closure*, or *Kleene star* of Σ , denoted Σ^* , is the collection of all strings in Σ ;
- a *formal language*, or, where unambiguous, *language*, is some subset of Σ^* for some alphabet Σ .

With this terminology, we can more precisely describe our idealization of phonology. We imagine that Universal Grammar consists of a syntactic/morphological module and a phonological module. An individual language L will have an idiosyncratic grammar, which comprises a syntactic/morphological component and a phonological component. We view the syntactic/morphological grammar as a black box that produces strings in the alphabet Σ of morphophonemes, which is also idiosyncratic to L . Call this language UR , the collection of *underlying representations* of L . The phonological grammar then reads elements of UR and “does phonology” to them, identifying them with strings in the alphabet Δ of phonemes in L . Call the language of such strings SR , the collection of *surface representations* in L . We don’t necessarily imagine phonological grammar as a function, for computational reasons, but rather as a relation that identifies whether a given $s \in UR$ phonologically corresponds to a given $t \in SR$.

2.3 Finite-State Transducers and Regular Relations

The study of formal languages and the study of linguistics are of course intimately intertwined - one important interdisciplinary result is Gold's (1967) mathematical treatment of learnability with respect to different methods of information reception. Importantly, he proves that the only class of languages that could be learned from a text – that is, classified in finite time from only a finite set of positive examples of sentences – are the regular languages.

Regular languages are the ones defined by regular expressions, which are a commonplace formalism in most word processors or spreadsheet management applications. Though it's common practice to define regularity in terms of the Chomsky hierarchy, which is done in a later section, for the sake of philosophically justifying the claim that phonology is regular, and for the sake of using algebraic language that will resound elsewhere in the treatment of OT, I will follow Kaplan and Kay (1994) and show that the use of finite-state transducers follows from the algebraic definition of regular languages. I deviate by offering a monoid-oriented definition of regular languages.

To do this, define the following:

Definition 2.4. A *monoid* $M = (X, \cdot, \varepsilon)$ is a set X together with an associative binary operator $\cdot : X \times X \rightarrow X$ and for which $\varepsilon \in X$ is an identity, i.e. for every $x \in X$, $x \cdot \varepsilon = \varepsilon \cdot x = x$.

Let M^* denote the Kleene star of M , that is, the minimal collection containing M and that is closed under \cdot .

Monoids axiomatize a notion of concatenation or ordered combination, and this generalizes the behavior of strings within formal language theory. The following characterization is the canonical example of a monoid:

Example 2.5. Let Σ be an alphabet and let $::$ be the string concatenation operation where,

for $s = s_1 \dots s_n, t = t_1 \dots t_n \in \Sigma^*$, we have

$$(s :: t)_k = \begin{cases} s_k, & k < n \\ t_{k-n}, & k \geq n \end{cases}$$

For ε denoting the empty string, the triple $(\Sigma^*, ::, \varepsilon)$ is a monoid called the *free monoid on Σ^** .

Semirings, another structure that will be instrumental in the optimization algorithm presented later, are very easy to define if monoids are on the table. Two-level phonology is also nicely captured in terms of the product of monoids. Let us introduce the following:

Lemma 2.6. *Let $M = (X, \cdot, \varepsilon)$ and $N = (Y, *, \epsilon)$ be monoids. Then $M \times N = (X \times Y, \otimes, \emptyset)$ is a monoid, where $(m, n) \otimes (\tilde{m}, \tilde{n}) = (m, \cdot \tilde{m}, n * \tilde{n})$ and $\emptyset = (\varepsilon, \epsilon)$.*

Proof. Routine use of definitions. See appendix. □

This highlights one of the features of two-level phonology: it's a product monoid. This is a substantial claim that deserves to be argued in full: not only do we believe that phonology is a process that assigns grammatical correspondence between underlying forms and surface forms, but we believe in the integrity of a phonological phrase. In other words, the way we characterize the two-level model supposes that a surface form is “fully processed.” That is, if we have two underlying representations s, s' and corresponding surface representations t, t' such that $(s, t), (s', t') \in (\sim) \subseteq UR \times SR$, we posit that $(s :: s', t :: t') \in (\sim) \subseteq UR \times SR$. This means that no further derivation is allowed to occur across phonological boundaries. Because we have not specified what the content of UR or SR actually is yet, this makes it clear: an underlying form is one delimited by phonological boundaries. Intuitively, one would expect this to correspond to the vernacular notion of a sentence.

This said, we can posit the following propositions:

Proposition 2.7. *Let $M = (UR, ::, \varepsilon)$ and $N = (SR, ::, \varepsilon)$. Then $M \times N$ is a monoid.*

We can thus refer to $M \times N$ defined above as $UR \times SR$, since we know the collection to have a canonical monoid structure.

Proposition 2.8. *Let \sim be the relation induced by a phonological grammar. Then $P = (\sim, *, \epsilon)$ where $(s, t) * (s', t') = (s :: s', t :: t')$ and $\epsilon = (\epsilon, \epsilon)$ is a monoid.*

Having formalized the phonology-induced relation as a monoid, we're now ready to define the structure of regularity that we want to give it. Using monoids, we can define the class of regular languages in a way that bears out their concrete algebraic properties with relation to phonology:

Definition 2.9. Let $M = (X, \cdot, \epsilon)$ be a monoid. For some collection S of subsets of M , and elements $A, B \in S$, define the following notation:

- $A \cdot B = \{a \cdot b \mid a \in A, b \in B\};$
- $A^0 = \{\epsilon\};$
- $A^{n+1} = A^n \cdot A;$
- $A^* = \bigcup_{k=0}^{\infty} A^k.$

We say that the collection S is the *rational set* for M if the following hold:

- For all $A, B \in S$, $A \cup B \in S$;
- For all $A, B \in S$, $A \cdot B \in S$;
- For all $A \in S$, $A^* \in S$;
- For any collection \tilde{S} satisfying the above three conditions, $S \subseteq \tilde{S}$.

A formal language is called *regular* if it is an element of the rational set for some monoid.

The idea, then, is that the relation \sim induced by a phonological grammar is an element of the rational set for the monoid $UR \times SR$. This is not a mere assertion though *prima facie* it might sound somewhat nonsensical to think about entire natural-language phonologies as behaving according to these general algebraic relations. The literature by and large agrees that SPE-style rewrite rules, which are regular expressions and treated below, are adequate

| Language | Machine |
|------------------------|--------------------------|
| Recursively Enumerable | Turing Machine |
| Context-Sensitive | Linear Bounded Automaton |
| Context-Free | Pushdown Automaton |
| Regular | Finite-State Automaton |

Table 1: The Chomsky Hierarchy

to describe all natural-language phonologies and that this is only a partial characterization. Thus it suffices to restrict the space of possible relations induced by phonological grammars to a class that will be computationally tractable. Even if many grammars in this class are nonsensical, the presence of finite-state machines will not preclude further restriction.

To make this point clearer, we can introduce the Chomsky hierarchy, which is ordered such that higher classes contain the lower ones. It's visible in Table 1.

We can now introduce the machine, specifically, the finite-state transducer (hereafter, “FST”), that has “become ubiquitous” in the world of computational linguistics. (Mohri 1997) As a complete treatment of the hierarchy is beyond the scope of this paper, we will present only the proof of correspondence between the FST and the rational set of the UR - SR monoid. First, we define the machine:

Definition 2.10. A *finite-state transducer* $T = (Q, q_0, F, \Sigma, \Delta, \sigma, \delta)$ is a 7-tuple consisting of the following data:

- Q is a set whose elements are called *states*;
- $q_0 \in Q$ is the *initial state*;
- $F \subseteq Q$ is the set of *terminal states*;
- Σ is the *input alphabet*, whose elements are designated as *input symbols*;
- Δ is the *output alphabet*, whose elements are designated as *output symbols*;
- $\sigma : Q \times \Sigma \rightarrow \Delta^*$ is the *output function*;
- $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*.

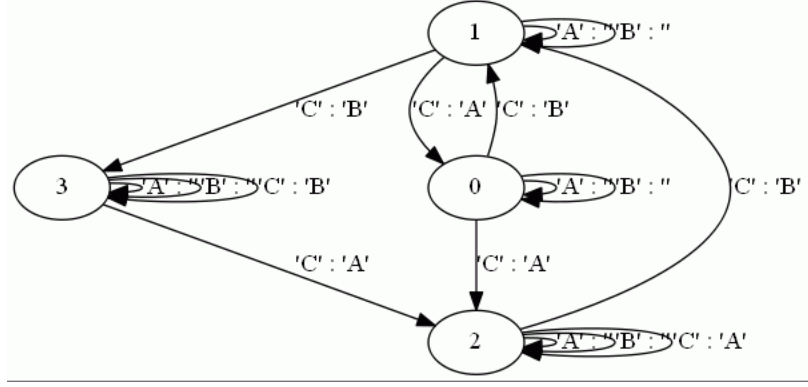


Figure 1: A simple FST

The functions σ, δ extend to maps $\sigma : Q \times \Sigma^* \rightarrow \Delta^*$ and $\delta : Q \times \Sigma^* \rightarrow Q$ by the following recursive relations:

$$\begin{aligned}\sigma(s, \epsilon) &= s, & \sigma(s, xy) &= \sigma(s, x)\sigma(\delta(s, x), y) \\ \delta(s, \epsilon) &= s, & \delta(s, xy) &= \delta(\delta(s, x), y)\end{aligned}$$

Given a string $s \in \Sigma^*$, T is said to *accept s with output t* , where $t = \sigma(q_0, s)$, if $\delta(q_0, s) \in F$.

An FST can be visualized with a state-and-arrow diagram, such as those presented in Riggle (2011). For example, Figure 1 displays a simple FST.

The machine takes an input string s and treats it as a “tape,” stepping through it symbol-by-symbol and at each step changing to a new state determined by its current state and current input symbol. The map δ represents this, taking (q_0, s) to some state - if the state $\delta(q_0, s)$ is a terminal state, then the “tape” s has been accepted by the machine. Additionally, at each step, it “prints” a symbol to its output string as determined by the current state and current input symbol. The map σ represents this, taking (q_0, s) and “prints” an output string t .

Definition 2.11. An *FST-computable function* is a function $f : X \rightarrow Y$ such that there exists an FST T with the following properties:

1. For each $x \in X$, if $f(x)$ is defined, then T accepts x with output $f(x)$;

2. For each $x \in X$, if $f(x)$ is not defined, then T does not accept x .

The function that we want to compute is a specific one, namely the *characteristic function* of the phonological relation we’ve thus far identified.

Definition 2.12. Given some domain X , an *FST-computable set* is a subset $S \subseteq X$ such that the characteristic function $K_S : X \rightarrow \{0, 1\}$ (taking value 1 for all $x \in S$ and 0 for all $x \notin S$, that is, yielding the binary truth value of the statement “ $x \in S$ ”) is FST-computable.

An *FST-computable relation* $\sim \subseteq X \times Y$ is a relation that is also an FST-computable set. That is, the characteristic function $K : X \times Y \rightarrow \{0, 1\}$ determining the truth value of the statement “ $x \sim y$ ” is FST-computable.

Finally, we can appeal to a quite tedious proof from the literature to validate the use of finite-state machines to compute regular relations. In its full rigor, it spans many pages, and is largely of technical interest, painstakingly building and checking a machine to fit a given regular expression:

Lemma 2.13. *A relation is FST-computable if and only if it is regular.*

Proof. For the adventurous reader. See Kaplan and Kay (1994). □

Once we argue that the relations induced by phonological grammars are regular, it will now immediately follow that we can attack them with the awesome machinery of the finite-state machine. FSTs are very computationally tractable objects, and therefore pervasive in the literature, because σ and δ are easy to implement in any imperative programming language.

2.4 Rewrite Rules and the Final Proof

To conclude the proof, we formalize the properties of SPE-style rewrite rules. These appear in the literature, as noted above, in the following form:

$$A \rightarrow B / C_D$$

Mathematically speaking, a rewrite rule of this form can be described, paraphrasing Chomsky and Halle (1968), as follows:

Definition 2.14. A *rewrite rule* R over an alphabet Σ is a relation defined by a 4-tuple (A, B, C, D) where C and D define regular languages, and $A, B \in \Sigma$. Let any $(s, t) \in R$ if and only if, when applied left-to-right, every instance of A occurring between elements of C and D in s corresponds to an instance of B in t in the same context.

The following result is known from the literature, and involves rather cumbersome and unenlightening manipulation of FSTs, but nonetheless demonstrates, in short, that a machine can be built to recognize and validate the alternation $A \rightarrow B$ when it occurs between expressions in C and D , respectively:

Lemma 2.15. *Rewrite rules as defined above induce regular relations.*

Proof. See Johnson (1972) or Kaplan and Kay (1994). □

Additionally, the pervasiveness of rewrite-rule-based analyses of natural language phonologies should bear out the validity of the following assumption:

Proposition 2.16. *Any phonological grammar can be adequately represented as a collection of rewrite rules.*

Ergo, the phonological relation has at least one representation that takes the form of a regular relation and the proposition from the previous section is proven.

Corollary 2.17. *The relation defined by any phonological grammar is regular.*

Corollary 2.18. *Any phonological grammar can be computed by a finite state machine.*

This completes the preliminary task of computational phonology by bounding the complexity of individual phonological grammars. This result was reached over 25 years ago: the Xerox calculus (Karttunen, Koskenniemi, and Kaplan 1987) embraces the rewrite rule formalism, and a complete software package for building regular grammars in the style of SPE is available.

3 Optimality Theory

3.1 Overview: Learnability as Economy

Though rewrite rules have sufficed as scaffolding for the machinery of finite-state automata, we now move the treatment to the (once) fashionable Optimality Theory. As hinted at above, a large concern espoused in Tesar and Smolensky (1996) is typological and regards the *grammar space*. A theory of phonology as such must concern itself not only with the correctness of a given account of a particular natural language process, but also with the space of all possible accounts within the formalism provided. Because of the nature of the field, of course, not only is it necessary to weed out nonsense grammars that result from overgeneration; it's also necessary to choose between many different accounts of the same phenomenon that might arise out of the same formalism.

Recall the principle of Economy (Chomsky and Halle 1968): the best grammar is the most parsimonious. For the SPE, this was a matter of the number of symbols used in the account: the fewer rewrite rules, the better the theory. However, one must consider that algorithms could be used to write these rules, and that the fewest number of symbols seems to be something more like the Kolmogorov complexity of the program used to generate the grammar. However, this presents its own issues, as the recent histories of quantum computing and cognitive science have borne out: to analogize the activity of the unconscious mind to the operation of a Turing machine and to then report the “economy” of a grammar in terms of the language of the model seems rather ridiculous. As Prince and Smolensky (1993) argue, “real-world efficiency is strongly tied to architecture and to specific algorithms, so that estimates of what can be efficiently handled have changed radically as new discoveries have been made, and will continue to do so. Consequently, there are neither grounds of principle nor grounds of practicality for assuming that computational complexity considerations, applied directly to grammatical formalisms, will be informative.”

An interpretation of this line of thinking is that *learnability*, that is, the minimum number of symbols the mind has to figure out, is the measure of economy. Chomsky, Halle, Prince,

and Smolensky all agree that “it is not incumbent on a grammar to compute.” (Chomsky and Halle 1968; Prince and Smolensky 1993) Instead, Prince and Smolensky posit a view of phonological grammar as a total ordering of violable constraints on the well-formedness of surface forms, where the constraints themselves are part of Universal Grammar and the *only* content of an idiosyncratic grammar is its ordering. In this way, explicit reference to the computational properties of the formalism is avoided, and the Economy principle is recast, simply put, as “banned options are available only to avoid violations of higher-ranked constraints and can only be used minimally.” (Prince and Smolensky 1993)

Tesar and Smolensky (1996) thus argue that the triumph of Optimality Theory over contemporary so-called “Principles and Parameters” accounts of grammar lies in the learnability structure of the grammar space. Although, they argue, OT gives rise to a prediction of “factorial typology,” (i.e. the number of possible orderings of n universal constraints is $n!$) the structure induced by the total ordering is sufficient to make it triumph economically on account of superior “search speed.” This seems to contradict Smolensky’s earlier statement, and the juxtaposition should bear out what’s really going on: it’s foolish, agrees the literature, to measure the activity of the learner against the specific symbols offered in a formalism, but it is meaningful to establish mathematical bounds on the efficiency of a possible learning algorithm that operates outside of the specific formalism of the theory.

Though production is different from learnability, the same doctrine seems to apply to Prince and Smolensky’s account of OT implementation. The method of judgment is not specified, and therefore the theory does not have to account explicitly for the computation of optimal forms. Instead, it falls upon the researcher to establish bounds on the efficiency of doing so. This will be precisely the task we take on in the next section.

3.2 OT Preliminaries and the Algebra of Multisets

The structure that Smolensky claims is so desirable is simply the fact of total ordering on the constraints. Although it remains up to the Optimality Theorist to posit what the constraints are, the demand for total ordering allows access to a wealth of literature on

combinatorial optimization techniques. To begin the treatment of OT, I will highlight the two mathematical objects central to the investigation here:

Definition 3.1. An *total order* on a set X is a relation $(<) \subseteq X \times X$ where the following axioms hold:

- For every $x \in X$, $x < x$;
- For every $x, y \in X$, exactly one of the three following propositions holds: $x < y$, $y < x$, or $x = y$;
- For every $x, y, z \in X$, if $x < y$ and $y < z$, then $x < z$.

Write $x \leq y$ if either $x < y$ or $x = y$ is true.

Definition 3.2. A *multiset* is a pair (C, m_C) where C is any set and $m_C : C \rightarrow \mathbb{N}$ is called the *multiplicity function*.

The formal set of axioms that will characterize OT in the model we propose is the following list:

- There exists a universal set of phonological constraints, common to all natural languages, which take the form of named relations $\rho \subseteq \Sigma^* \times \Delta^*$, and each language imposes a total order on these constraints. This ordering is the only idiosyncratic component of a natural language’s phonological grammar. (Σ and Δ can be, in the spirit of Chomsky and Halle (1968), assumed to be subsets of a universal alphabet of all possible feature matrices.) Moreover, every phonological relation is a regular one;
- There exists a function GEN that maps each string in Σ^* to the full set of possible surface representations, i.e. presents every $t \in \Delta^*$ as a possible corresponding surface form;
- There exists a function CON that assigns to each output t of GEN on a given input s a multiset consisting of one count of the name of each constraint ρ that (s, t) violates for each time it is violated;


| Candidates from /input/ | | \mathbb{C}_1 | \mathbb{C}_2 |
|--|--|----------------|----------------|
|  ω | | | * |
| z | | * ! | |

Figure 2: Generic OT tableau from Prince and Smolensky (1993)

- If \sim is the relation induced by a given phonological grammar, then for every $(s, t) \in (\sim)$, the representation t shall be an *optimal candidate*, that is, a surface representation that, under CON, incurs the fewest violations of the highest-ranked constraint among constraints not equally violated by the output of GEN at s .

In the literature, an optimality judgment takes the form of a tableau that looks something like the following:

Considering all output strings, we see the tableau picking the optimal candidate following the method defined above: going down the list of “tied” constraints, it picks a “winner” who violates the most important constraint the least. Because this looks a lot like a matrix, and because mathematicians and computer scientists have some sort of predilection for seeing problems in terms of matrices, the model I propose will largely remain loyal to the effort to present the machinery in this form.

It becomes again pertinent to mention the *factorial typology* property: that the language of OT grammars is simply the language consisting of permutations of the constraint hierarchy $c_1 \dots c_n$ and thus contains at maximum $n!$ elements. This ensures at the very least that any algorithm computing the full class of OT grammars terminates in finite time, which is not a guarantee available to the unrestricted SPE-style theory of typology. Indeed, it makes additional and quite desirable guarantees that will be analyzed in the appropriate sections.

To flesh out the formalization, we need to build the objects. The plan is stated in Table 2. To repeat what’s stated in the above axioms:

Proposition 3.3. *A phonological grammar is a total order on CON.*

| OT Object | Model Object |
|-----------|---|
| GEN | Δ^* |
| CON | (Multiset-weighted) FST |
| EVAL | Optimization in the semiring of multisets |

Table 2: Computational OT according to Riggle (2011)

As another easy definition, since the Kleene star gives rise to a regular expression that will satisfy the semantics dictated by GEN, that is, will generate every possible surface form:

Proposition 3.4. *The function GEN maps an underlying form s to the set Δ^* .*

Riggle (2011) maintains that Ellison’s (1995) idealizing assumptions remain necessary to keep formalizations of OT tractable. In particular, as proven in the last chapter, we maintain the assumption that GEN and all the relations are regular or else overpower the theory. To formalize the phonological constraints, we have to depart from Ellison’s assumption that they must be binary relations and instead allow constraints to count the number of violations incurred by a given underlying-surface pair. If we want to collect multisets that can actually tally the number of violations of constraints, this becomes necessary. This decision will “overpower” our mathematical model of OT, i.e. allow the factorial typology to generate phonological grammars that don’t induce regular relations, but fortunately this can be prevented with localization.

Since we want to keep track of the number of violations incurred with respect to each constraint, the constraints will have to bear distinct names:

Definition 3.5. A *constraint symbol* is an element c of a set C . Call C the *constraint alphabet* in this context. A *violation profile* is a multiset with values in some constraint set C . For a given constraint alphabet C , let V_C denote the collection of all violation profiles with constraints in C , in addition to the element ∞ .

We now define what will constitute a syntax for the behavior of constraint violations. Hereafter I’ll use Python-style notation to indicate multisets, i.e. $S = \{c_1 : n_1, \dots, c_N : n_N\}$ indicates a multiset $S = (C, m_C)$ where $N \in \mathbb{N}$, $C = \{c_1, \dots, c_N\}$ and $m_C(c_k) = n_k$ for all $k \in [1..N]$.

Prince and Smolensky (1993) speak of *harmony maximization*, a notion borrowed from connectionism in cognitive science, as the principle guiding a computation that's otherwise only determined implicitly. To pursue harmony, the algorithm implementing EVAL will seek the “lightest” multiset, thereby necessitating that we define a total order on the collection of violation profiles. Moreover, because we've established that $UR \times SR$ is a monoid, we'll need to take the union of violation profiles when we concatenate UR-SR tuples, and we'll need this union to make sets heavier with respect to the order. The two operations of taking the minimum of two multisets with respect to the total order and of taking the union will induce an algebraic structure called a semiring, which permits the use of an efficient optimization algorithm. This method, introduced in Riggle (2004), allows a parsimonious and somewhat natural characterization of optimality.

Definition 3.6. A *semiring* is a 5-tuple $(R, \oplus, \otimes, 0, 1)$ where $(R, \oplus, \otimes, 1, 0)$ where $(R, \oplus, 0)$ and $(R, \otimes, 1)$ are monoids, \oplus is commutative, and the following propositions hold:

- $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$;
- $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$;
- $a \otimes 0 = 0 \otimes a = 0$

An *ordered semiring* is a semiring for which R has a partial order \prec .

A semiring is a generalization of the algebraic structure known as a ring, and it can be thought of as an axiomatization of the behavior of addition and multiplication on the natural numbers. The lack of inverse elements on either operator and the presence of an order on the elements naturally gives rise to an optimization problem: given an ordered semiring, determine in finite time the minimal nontrivial element.

Definition 3.7. Given an ordered semiring R , we say $x \in R$ is *optimal* if:

- $0 < x$;
- For every $y \in R$, $x \leq y$

Let X denote the set of optimal elements of R . A *semiring optimization procedure* is the FST-computable characteristic function K_X for this set (it will be the constant function 0 if the set does not exist).

Ordered-semiring-optimality will model phonological optimality. Semiring optimization is precisely what we're going to do with the collection of violation profiles: to each element $s \in \Sigma^*$, our (forthcoming) definition of the function CON will associate a single violation profile, so optimizing over the collection of violation profiles, once we deem it possible, will yield a class of phonologically optimal strings. The fact that we can't guarantee a single output string seems to present an issue, but any ambiguity arising from this should vanish with adequately many constraints, and moreover the fact that we've defined phonological optimality as a *relation* actually mathematically accounts for this possibility.

Finally, since OT needs constraints to induce an order on their violation profiles, we define the following:

Definition 3.8. Let $<$ be an arbitrary total order on a constraint alphabet C . For two elements $x, y \in V_C$, define the partial order \prec as follows: consider the set I of constraints c for which $m_x(c) \neq m_y(c)$ under the total order $<$. Let c_0 be the maximum element of I . Then $x \prec y$ if $m_x(c_0) < m_y(c_0)$, $y \prec x$ if $m_y(c_0) < m_x(c_0)$, and the two are incomparable under \prec otherwise. Finally, for all $x \in V_C$, define $x \prec \infty$ and observe that necessarily $\emptyset \prec x$. Call \prec the *harmonic inequality* induced on V_C by $(C, <)$.

The collection of violation profile thus inherits an ordered semiring structure as follows:

Definition 3.9. Let $\min_{\prec} : V_C \times V_C \rightarrow V_C$ be defined as the minimum operator under \prec . That is, $Z = X \min_{\prec} Y$ is X if $X \prec Y$ and Y otherwise.

Definition 3.10. Let $\uplus : V_C \times V_C \rightarrow V_C$ be defined by multiset union. That is, $(X, m_X) \uplus (Y, m_Y) = (Z, m_Z)$ where $Z = X \cup Y$ and $m_Z(\alpha) = m_X(\alpha) + m_Y(\alpha)$. In the case of ∞ , define $X \uplus \infty = \infty$.

These are the two primitive operators on violation profiles.

Lemma 3.11. *Let C be a constraint alphabet. Then $\mathcal{V} = (V_C, \min_{\prec}, \oplus, \emptyset, \infty)$ is a semiring. Moreover, under any total order $<$ on C , \mathcal{V} together with the harmonic inequality induced by $<$ is an ordered semiring.*

Proof. Routine algebra to fulfill the definition. See appendix. \square

This proves that our formalization of phonological constraints permits the problem of implementing the semiring optimization procedure. The remainder of this paper will be essentially dedicated to solving this problem and establishing a bound on the computational efficiency of doing so. However, the structure \mathcal{V} is the general semiring of *all* possible violations of an arbitrary alphabet of constraint symbols. In some sense, it is the syntax for the language in which we will pose the problem - what is needed now is a semantics for the symbolic behavior of constraints and their multisets. This will arise by formalizing the function CON, which will determine how to assign violation profiles to strings in Σ^* .

3.3 The Optimality System

Although the relations necessary for the model will make binary judgments, it's necessary to generalize their definition to allow for the violations assigned to retain distinct names so that they can be tallied in profiles. This is true to how the tableau presentations work, and it allows the profiles to be tallied without remembering the hierarchy of the constraints.

Definition 3.12. Let Σ be an alphabet. A *weighted relation* in a set X over $\Sigma \times \Delta$ for a constraint symbol $c \in C$ is any function $\rho_c : \Sigma^* \times \Delta^* \rightarrow X$. If S is a collection of pairs (c, ρ_c) for which each c permits a unique weighted relation, call S a *constraint set* for V_C over Σ .

With this, finally, we can borrow the abstract algebraic notion of a generator to describe a violation semiring corresponding to a given input string. This, then, is an algebraic object representing the same collection that OT tableaux in the literature portray. In other terms, the violation semiring will determine the outputs of both GEN and CON at once. Explicitly defined, we have:

Definition 3.13. Let Σ be an alphabet, C a constraint alphabet, and S a constraint set for C over Σ . For a chosen input string $s \in \Sigma^*$, define

$$V_S(s) = \{\emptyset, \infty\} \cup \{\rho_c(s, \tilde{s}) \mid \rho_c \in S, \tilde{s} \in \Delta^*\}$$

Let $\mathcal{V}(s)$ be the set generated by $V_S(s)$ under the operations \min_{\prec} and \uplus defined above. Call $\mathcal{V}(s)$ the *violation profile semiring* for S generated by s . Call $\mathcal{T} : s \mapsto \mathcal{V}(s)$ the *tableau operator* for S on Σ^* .

This is the last piece of the proposition beginning this section: we can see that, because it assigns to each string a violation profile, and each profile encodes precisely what's found in a single row of an OT tableau, the operator provides a countable amount of entries that a finite picture of a tableau can only hint at. It bears mention here, then, the possible utility of a computational approach to the theory: rather than ask that the theorist rely on anecdotal diagrams, an optimization algorithm can return the full story in seconds or less.

Lemma 3.14. *Given Σ, C, S, s as above, it holds that $\mathcal{V}(s) \subseteq \mathcal{V}$ and $(\mathcal{V}(s), \min_{\prec}, \uplus, \emptyset, \infty)$ is indeed a semiring that inherits a partial order from any total order on C .*

Proof. Clear. See appendix. □

Imitating Ellison (1995), we can wrap this all up into one concise object:

Definition 3.15. Let Σ be an alphabet, C a constraint alphabet, and S a constraint set for C over Σ . Let $\mathcal{V} = (V_C, \min_{\prec}, \uplus, \emptyset, \infty)$, let \mathcal{T} denote the tableau operator for \mathcal{V} , as above, and let \mathcal{O} be a semiring optimization procedure for \mathcal{V} . An *optimality system* for S over Σ is the structure $(\mathcal{V}, \mathcal{T}, \mathcal{O})$.

One can think of \mathcal{T} as producing the tableau for which \mathcal{O} is the little hand that points to the optimal candidate. The OT-optimal candidate for a given underlying representation $s \in \Sigma^*$ is then conveniently denoted $\mathcal{OT}(s)$.

Recall that this model is still too powerful. Fortunately, our formulation makes localization simple enough of a task. If we bound the number of violations a constraint can incur, then T naturally maps to a finite codomain:

Definition 3.16. A *localized multiset with precision n* is a set $(C, \min\{m_C, n\})$. Let \mathcal{V}_n be defined the same as \mathcal{V} , but where the constraint violations are localized with precision n and call it the *localized violation semiring with precision n* . Define $\mathcal{T}_n(s)$ analogously. Then a *localization* (with precision n) for an optimality system $(\mathcal{V}, \mathcal{T}, \mathcal{O})$ is simply $(\mathcal{V}, \mathcal{T}_n, \mathcal{O})$.

We can tack on this restriction after making all of our original definitions, so it’s not a huge hurdle to clear. Additionally, it seems that setting $n \geq 10$ would make any resulting judgment failures extremely marginal. This avoids the following pitfall”

Lemma 3.17. A *localized optimality system induces a regular relation*.

Proof. See Frank and Satta (1998). □

This leaves us to attack the remaining questions: how do we implement the semiring optimization procedure, and how does the class of relations that we consider as constraints affect its the computational properties? The optimality system has provided a straightforward formalization of OT as we understand it, so we’re ready to ask about the computational properties, that is, the Economy, of \mathcal{O} .

3.4 Weighted Transducers

Recall now what was proved in the previous chapter: phonology is regular. Because of this, each weighted constraint within the optimality system can be computed by a finite-state transducer. We can go further and extend the definition of our FSTs to actually hold on to the violation profiles accrued in the process of computation. This will allow us to take the weighted FST as the target of \mathcal{O} , and we’ll show that this target is a very tractable one.

Definition 3.18. Let W be set. A *weighted finite-state transducer* with weights in W (hereafter “W-weighted FST”) is a 6-tuple $T = (Q, q_0, F, \Sigma, \Delta, A)$ consisting of Q, q_0, F, Σ , and Δ defined as in an unweighted FST, and where A takes the following form:

$$A \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times W \times Q$$

Call the elements of A the *arcs* of T . For an arc $a = (q, x, y, z, q') \in A$, define the following:

- $s(a) = q$ is called the *start state* of a ;
- $t(a) = q'$ is called the *end state* or *terminal state* of a ;
- $i(a) = x$ is called the *input segment* of a ;
- $o(a) = y$ is called the *output segment* of a ;
- $w(a) = z$ is called the *weight* of a .

Weighted FSTs have exactly the same computational power as unweighted FSTs, and differ only by returning a violation profile for each input-output pair instead of a binary truth value. Because binary relations are just sets, they obey union and intersection. For weighted relations $\rho_c, \rho_d : \Sigma^* \times \Delta^* \rightarrow X$, we can define their intersection as $\rho_{c \cap d}(s) = \rho_c(s) \cup \rho_d(s)$ whenever X “permits a notion of union.” For a precise case, take the following:

Definition 3.19. Let c, d be constraint symbols in some constraint set S for C over $\Sigma \times \Delta$. Then define the *aggregate constraint relation* $\rho_{c,d}$ by $\rho_{c,d}(s) = \rho_c(s) \uplus \rho_d(s)$. Because \uplus is commutative, for any finite set K of constraint symbols, $\rho_K(s) = \biguplus_{k \in K} \rho_k(s)$ is well-defined.

Since any binary relation r is isomorphic to a weighted relation ρ_c that returns \emptyset instead of “false” and $\{c : 1\}$ (for some constraint symbol c) instead of “true,” we can make the following construction:

Definition 3.20. Let R be a finite collection of regular binary relations indexed over some constraint alphabet K . Construct a set \tilde{R} as follows: for each $r_k \in R$, let ρ_k be a weighted relation returning, for each input, $\{k : 1\}$ if r_k holds and \emptyset otherwise. Call $\rho_{\tilde{R}} = \rho_R^K$ the *aggregate weighted relation* for R indexed over K .

Note that each ρ_k is isomorphic to a regular relation, hence regular, and that the aggregate weighted relation must then also be a regular relation. Hence it permits computation by a weighted FST.

As mentioned above, our definition of constraint set is more general than what our computational model will use. Instead, we want to give names to a bunch of binary relations, “lift” them so that they can count, and then build a weighted FST that computes their aggregate weighted relation. This FST is indeed guaranteed to exist, and this constitutes a central result in the exposition.

Theorem 3.21 (Equivalence of multiset-weighted FSTs to systems of OT constraints). *Let Σ be an alphabet and let C be a constraint alphabet. Let R be a finite set of binary constraints indexed over C . Then the following hold:*

- *R together with any total order $<$ naturally induces a constraint set S ;*
- *For every $s \in UR$, there exists an FST T with weights in $V_C(s)$ that computes ρ_R^C ;*
- *$\mathcal{V}(s)$ is the semiring formed by $\{\rho_R^C(s, \tilde{s}) \mid \tilde{s} \in \Delta^*\}$ under \min_{\prec} and \uplus , that is, T computes single entries in $\mathcal{T}(s)$*

Proof. See appendix. □

Definition 3.22. Call T as constructed above the *aggregate transducer* over s with weights in V_C .

The duality of regularity and FST-computability is crucial here: it allows a non-constructive existence proof that is mathematically parsimonious, but the construction will be necessary in the software implementation.

We see that T is the computational object that enumerates our countably infinite tableau object, $\mathcal{T}(s)$, in a countable number of steps. This infiniteness is unavoidable, of course, which implies that we have to “lazily” evaluate the function, that is, compute the composed function \mathcal{OT} as one routine, if we want to pick out an optimal candidate in finitely many steps. In other words, we have to compute GEN and CON simultaneously or at least alternatingly.

Riggle (2011) presents a dynamic-programming solution to do exactly this. Taking our machine to be the composition of all constraints in some constraint set C with weights in

$\mathcal{V}_n(s)$ allows localization and ranking to remain properties of the semiring and be considered as extrinsic to the implementation of GEN and CON. In the remainder of this section, we'll present the algorithm and state its computational properties.

3.5 Paths and Dynamic Programming

We can now delve into some of the convenient properties of the finite-state machine characterization that permit Riggle's (2011) dynamic programming solution. Note that henceforth we're interested only in automata with weights in semirings, as these bestow the necessary structure for optimization.

Definition 3.23. Let $T = (Q, q_0, F, \Sigma, \Delta, A)$ be a transducer with weights in a semiring R . A *path* π is an element $a_1 \dots a_n \in A^*$ for which $t(a_k) = s(a_{k+1})$ for all $k \in [1..n]$. If $t(a_n) = s(a_1)$, then we call π a *loop*. If $s(\pi) = q_0$ and $t(\pi) \in F$, then we call π a *complete path*.

Define the weight of a path as follows:

$$w(\pi) = \biguplus_{k=1}^n w(a_k)$$

Conveniently, paths are naturally identified with surface form candidates.

Definition 3.24. Let $\pi = a_1 \dots a_n \in A^*$ be a path. The *underlying representation* corresponding to π , or its *input*, is the string $i(\pi) = i(a_1) \dots i(a_n) \in \Sigma^*$. The *surface representation* corresponding to π , or its *output*, is the string $o(\pi) = o(a_1) \dots o(a_n) \in \Delta^*$.

Therefore, we can naturally pose the problem of phonological optimality in the following terms:

Proposition 3.25. *The class of optimal surface representations for a given underlying representation s and with respect to a constraint set C is precisely the collection of outputs corresponding to the minimal-weight complete paths in the aggregate transducer over s with weights in V_C .*

Let \mathcal{P} denote the collection of complete paths in a given transducer. Due to the nature of the harmonic inequality, we're guaranteed that \prec induces a total order on V_C , hence \mathcal{P} is guaranteed to have a minimal weight and hence a class of minimally-weighted, i.e. most harmonic, paths.

Finally, because \emptyset acts as an annihilator for the \min_{\prec} (i.e. harmonic minimum) operator, the semiring \mathcal{V} is bounded. This means that \min_{\prec} is a *monotonic* operator, which ensures that dynamic programming will yield a valid solution.

3.6 Directed Graphs and Adjacency Matrices

A common way to visualize finite-state machines is as directed graphs with labels associated to their connections. The definition above, taken from Riggle (2011), makes this an easy conceptual leap: the states are nodes and the arcs are labels. Ultimately this will allow \mathcal{O} to take the form of a shortest-path computation using adjacency matrices, which is a robust linear-algebraic approach and one that will generalize nicely.

Recall first the definition of a directed graph:

Definition 3.26. A *directed graph* $G = (V, E)$ is a set V of n many *vertices* together with a collection $E \subseteq V \times V$ of m many *edges*. A *directed graph with labels* is a 4-tuple $G = (V, E, L, \ell)$ with V, E as above and where L is a set of *labels* and $\ell : E \rightarrow L$ is an injection.

Think of ℓ as the function that assigns labels for the sake of bookkeeping. Objects of this type can be visualized like the following:

Definition 3.27. Let $T = (Q, q_0, F, \Sigma, \Delta, A)$ be a weighted FST. Define:

$$E = \{(s(a), t(a)) \mid a \in A\}, \quad L = \{(i(a), o(a), w(a)) \mid a \in A\}$$

Let $\ell : (s(a), t(a)) \mapsto (i(a), o(a), w(a))$. Call $\Gamma(T) = (Q, E, L, \ell)$ the *graph* of T .

A common way to represent a directed graph (V, E) is with an adjacency matrix:

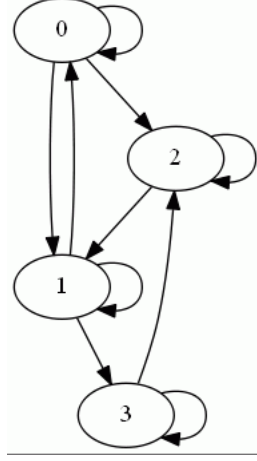


Figure 3: A simple directed graph

Definition 3.28. Let $G = (V, E)$ be a directed graph without labels. The *adjacency matrix* $M(G)$ is the $n \times n$ matrix defined as follows:

$$M(G)_{i,j} = \begin{cases} 1, & (i, j) \in E \\ 0, & (i, j) \notin E \end{cases}$$

Adjacency matrices encode, as the name would suggest, whether or not nodes are “adjacent,” that is, directly connected with arcs. Multiplying an adjacency matrix by itself yields a matrix encoding accessibility by length-2 paths. Multiplying n times, for $n = |V|$, yields a matrix encoding all possible accessibilities, since a path reaching every node and some more than once must be redundantly long.

This concept naturally extends to directed graphs with labels, insofar as we can produce the adjacency matrices, though we will have to redefine matrix multiplication to make this precise. For handling graphs of our variety of weighted transducers, the adjacency matrix would have entries in $L = \Sigma \times \Delta \times \mathcal{V}(s)$. However, this set will not permit a useful definition of multiplication, therefore we wouldn’t be able to multiply matrices containing them. Instead, we present an analogous object better suited to the task at hand.

Recall that running an input tape through a finite-state transducer T over $s \in \Sigma^*$ is equivalent to traversing $\Gamma(T)$ in the following manner: let $t = \varepsilon$, start at the initial state,

move along any number of arcs with empty input, concatenating their output segments to t , then for $k \in [1..n]$ repeat the process of moving along one arc with input segment s_k , concatenating its output segment to t , and then along any number of empty arcs and concatenating their output segments to t . At the end, if every move has been valid and the current state is terminal, then t is a valid output string for s .

Because we will only want to traverse in this pattern, we can forget keeping track of the input segments. And, because we want to keep track of all possible surface forms constructible at each step, we can deal with sets of fragmentary surface representations.

Definition 3.29. Let $\overline{\Delta}$ denote the power set 2^{Δ^*} , i.e. the set of sets of strings in Δ .

This gives significantly more structure. Finally, since we traverse exactly one segment at a time, we want a matrix with entries in $\mathcal{V} \times \overline{\Delta}$ where the $(i, j)^{th}$ entry represents the minimum weight as well as the set of possible outputs received when traveling from node i to node j . With that in mind, we can define the following:

Definition 3.30. Let $T = (Q, q_0, F, \Sigma, \Delta, A)$ be a transducer with weights in \mathcal{V} and under the harmonic inequality \prec . For states i, j , define the following:

$$W(i, j) = \min_{\prec} \{w(a) \mid a \in A, s(a) = i, t(a) = j\}, \quad O(i, j) = \{o(a) \mid a \in A, w(a) = W(i, j)\}$$

Let $\Gamma(T) = (Q, E, L, \ell)$ as above. For a given $c \in \Sigma$, define the *output matrix* for c as the matrix $M(c)$ where

$$M(c)_{i,j} = \begin{cases} (W(i, j), O(i, j)), & (i, j) \in E \\ (\infty, \emptyset), & (i, j) \notin E \end{cases}$$

Moreover, denote $\lambda = M(\varepsilon)^{|Q|}$, which is indeed a valid choice, and call this the *transition matrix* for T .

If we can define matrix multiplication on the output matrices for a given input string in a way that preserves the property where the $(i, j)^{th}$ entry in the product matrix keeps the minimum weight incurred for traveling along the path specified by the string consisting

of the corresponding symbols in Σ , then we can solve optimality for any $s \in \Sigma^*$ simply by computing the following:

$$M(s) = \lambda \times M(s_1) \times \lambda \times \dots \times M(s_n) \times \lambda$$

At this point, the semiring structure reappears. Recall that matrix multiplication for real-valued matrices consists of multiplication and addition. The real numbers form a semiring, and much of the behavior of real-valued matrix multiplication is retained when the definition is generalized to semiring-valued matrices. Explicitly, we define:

Definition 3.31. Let M, N be $n \times n$ matrices with values in some semiring $(R, \oplus, \otimes, 1, 0)$. Then define their product $P = M \cdot N$ as follows:

$$P_{i,j} = \bigoplus_{k=0}^n M_{i,k} \otimes N_{k,j}$$

Define their sum $S = M + N$ by $S_{i,j} = M_{i,j} \oplus N_{i,j}$.

For bonus points, we assert the following:

Lemma 3.32. Let $\mathcal{M}(R)$ be the collection of R -valued matrices, let $\bar{0}$ denote the matrix with entries all $0 \in R$, and let Id denote the identity matrix, i.e. that with entries $1 \in R$ along the diagonal and $0 \in R$ everywhere else. Then $(\mathcal{M}(R), +, \cdot, \bar{0}, \text{Id})$ is itself a semiring.

Proof. That the monoids form is clear. Then, distributivity and commutativity of the underlying operators allows an interchange of finite sums that allows associativity to hold. Finally, standard algebraic manipulations demonstrate distributivity. See appendix. \square

Thus any collection of entries in a generalized adjacency matrix that forms a semiring structure can “keep track of,” that is, meaningfully perform arithmetic on, path labels.

As it turns out, we can build $\mathcal{V} \times \bar{\Delta}$ itself into a semiring that keeps track of the outputs of the paths constructed through dynamic programming. Consider the set $\mathcal{V} \times \bar{\Delta}$ and define

the following operations:

$$(v, A) \sqcup (w, B) = \begin{cases} (v, A \cup B), & v = w \\ (v, A), & v \prec w \\ (w, B), & w \prec v \end{cases}$$

$$(v, A) \times (w, B) = (v \uplus w, \{a :: b \mid a \in A, b \in B\})$$

The former allows the “unification” of tuples, ensuring the uniqueness of weights. This preserves the total ordering property and allows that each violation profile in the semiring corresponds to the full list of representation fragments incurring it. The latter operation is simply the tupling of the two canonical notions of union on the factor semirings. Note that we deal with sets of strings, rather than just strings, so the “union” is in fact all possible concatenations of strings in the two sets.

Definition 3.33. Let D denote $\mathcal{V} \times \overline{\Delta}$, let $0_{\mathcal{D}}$ denote (∞, \emptyset) and let $1_{\mathcal{D}}$ denote $(\emptyset, \{\varepsilon\})$. In the case of a violation semiring over a single string, denote $D(s) = \mathcal{V}(s) \times \overline{\Delta}$.

Now we demonstrate the following:

Lemma 3.34. *The tuple $\mathcal{D} = (D, \sqcup, \times, 0_{\mathcal{D}}, 1_{\mathcal{D}})$ is a semiring.*

Proof. Associativity is trivial. First, an easy check that the operators form monoids. Second, check case-by-case that \sqcup distributes \times . See appendix. \square

Hence our output matrices have a well-behaved multiplication operation that suffices to compute $\mathcal{OT}(s)$. Thanks to dynamic programming, minimization gets performed only $(|\Sigma| + 1)|Q|^2$ many times, rather than at every step in multiplication. This optimization immediately excludes co-finitely many candidates that could never be phonologically optimal and ensures that the matrix method produces only a finite tableau. That is, it only *partially* computes $\mathcal{T}(s)$ during optimization, which is exactly what we want. To be formal, we have the following result:

Lemma 3.35. *Let $s = s_1 s_2 \dots s_n \in \Sigma^*$ and let \prec order \mathcal{V} . Then the $(i, j)^{th}$ entry in $M(s)$ is the tuple (w, S) where w is the \prec -minimum weight of any path from i to j with input s , and where S is the collection of strings corresponding to paths with this weight.*

Proof. Follows cleanly from the definition of \boxtimes . See appendix. \square

As a corollary, then, the strings corresponding to minimal complete paths will be found in one row of $M(s)$, hence the procedure computes \mathcal{OT} . Having succeeded in reducing the problem of optimality to the computation of λ and $M(s_k)$ as above, we have successfully split the problem into subproblems, producing a proper dynamic programming approach. We thus have an efficient ranking-specific (since \min_{\prec} is ranking-specific) method for computing the composite \mathcal{OT} .

3.7 Analysis and Remarks

Naïvely, computing each matrix means computing the minimum under \prec for the set of arcs between nodes i and j , hence this must be performed $(|\Sigma| + 1)|Q|^2$ times. On top of this, matrix multiplication must be performed $2|s|$ times, and itself consists of multiplying $n \times n$ matrices, thereby taking $|Q|^3$ operations of \min_{\prec} and \boxplus . Thus the complexity of optimization under each ranking \prec is bounded by $2|s||Q|^3$, which is of course linear in input length.

However, this is true only for a single ranking of S . The naïve computation of surface forms corresponding to *every* possible ranking is still factorial in the number of constraints - that is - bounded by $2|s||Q|^3(|S|!)$, and Riggle (2011) proposes a method for reducing this to polynomial time.

4 Computing Contenders

As mentioned previously, Prince and Smolensky’s factorial typology property is a neat component of OT. Once constraints are specified, simple permutation gives a prediction about the full grammar space - for the theorist, the presence of nonsensical grammars corresponding to a given hypothetical CON set should bear on the quality of a given OT analysis. Riggle (2011) presents a modification to the semiring optimization presented in the previous chapter that imports the power of dynamic programming to accomplish precisely this: modifying the semiring to include all possible rankings of CON is indeed feasible and presents a coherent model of phonological typology with clear learnability. It will also give rise to a software implementation that gives expedient feedback to the theorist attempting to build OT grammars.

4.1 Harmonic Bounding and Elementary Ranking Conditions

It’s simple enough to state the mathematical task at hand:

Definition 4.1. Given an optimality system $(\mathcal{V}, \mathcal{O}, \mathcal{T})$ and some $s \in UR$, a string $t \in SR$ is a *contender* if there exists some ordering \prec under which $t \in \mathcal{OT}(s)$.

Definition 4.2. Let $\mathfrak{K} : UR \rightarrow 2^{SR}$ denote the function taking each underlying representation $s \in UR$ to the set of surface representations that are contenders with respect to s .

Following Riggle (2011), computing \mathfrak{K} can be accomplished by keeping track of how constraints “split” elements of \mathcal{T} . Note also that, if we compute \mathfrak{K} , then we have all of the information from \mathcal{O} except for the constraint hierarchy. Recalling the initial OT axioms to which we’re committed, this means that \mathfrak{K} ought to be part of Universal Grammar.

Definition 4.3. For two elements $a, b \in D$ over a constraint set S , where $a = (v, A)$ and

| /VC/ | ONS | NOC | DEP | MAX | |
|---------------|-----|-----|-----|-----|--|
| a. VC. | * | * | | | $erc(b, a) = (\{ons, noc\}, \{dep, max\})$ |
| b. CV. | | | * | * | $erc(b, b) = (\{\}, \{\})$ |
| c. CV.CV. | | | ** | | $erc(b, c) = (\{dep\}, \{max\})$ |
| d. ϵ | | | | ** | $erc(b, d) = (\{max\}, \{dep\})$ |

Figure 4: As appears in Riggle (2011), the ERC set for a with respect to $\{a, b, c, d\}$ for $S = \{ONS, NOC, MAX, DEP\}$

$b = (w, B)$, define the following functions:

$$W(a, b) = \{c \in S \mid m_v(c) < m_w(c)\}$$

$$L(a, b) = \{c \in S \mid m_w(c) < m_v(c)\}$$

Here W is a mnemonic for “win” and L for “lose,” referring to how a fares under the harmonic inequality. We call the tuple $(W(a, b), L(a, b))$ the *elementary ranking condition* between a and b . We can extend the definition as follows:

Definition 4.4. Given $A \subseteq D$, define the *ERC set* for $a \in A$ as follows:

$$\mathfrak{E}(a, A) = \{(W(a, b), L(a, b)) \mid b \in A\}$$

This function produces all the ERCs for a given set of candidates. This will look something like Figure 1. ERCs give a lightweight characterization of potential contenderhood. As can be seen, many of them will result in “all-loser” profiles, that is, will correspond to entries in \mathcal{T} that are more numerous in every constraint than other entries. Riggle (2011) calls such entries *harmonically bounded*. By discarding the harmonically bounded elements, contender generation can be improved in efficiency.

In multi-constraint decision making, the Pareto frontier is the collection of candidates that are not simply bounded, that is to say, those that are not bounded by any single other candidate. The contenders function restricts us further to the candidates that have to win under a specific ranking, eliminating candidates that are compositely bounded by multiple

other candidates.

We want to “complete” \mathfrak{E} to one that determines the contender status of its first argument with respect to the second argument. Given a set of ERCs, we define a function that determines whether or not all can be satisfied by a single ranking.

Definition 4.5. Let E be an ERC set. Define $w(E)$ and $l(E)$ as its left and right projections, respectively, i.e.

$$w(E) = \bigcup \{w \mid (w, l) \in E\}, \quad l(E) = \bigcup \{l \mid (w, l) \in E\}$$

Next, define $d(E) = \{e \in E \mid w(\{e\}) \subseteq l(E)\}$.

Define the *consistency* of E as the following binary function:

$$\mathfrak{S}(E) = \begin{cases} \text{TRUE}, & l(E) = \emptyset \\ \text{FALSE}, & w(E) \subseteq l(E) \\ \mathfrak{S}(d(E)), & \text{else} \end{cases}$$

Here the function d discards any ERC that could be satisfied by ranking one of its “winning” constraints above all constraints that cause inconsistencies in E . Repeating this process will result either in some inconsistency that can’t be escaped in this manner or will exhaust all the constraints and come up empty, in which case consistency is reached. The process is called *recursive constraint demotion* (Riggle 2011; Tesar 1995; Tesar and Smolensky 1996).

Lemma 4.6. *Given a constraint set S , a string $s \in UR$, and a set $A \subseteq D(s)$, where $D(s)$ denotes the violation semiring in S over s , it holds that an element $a \in A$ is a contender if $\mathfrak{E}(a, A)$ is consistent.*

Proof. If $E = \mathfrak{E}(a, A)$ is consistent, then there exists a ranking of S such that every $e \in E$ can be satisfied, that is, under which a must be optimal in A . By definition, then, a is a

contender in A . □

Since we've proven contenderhood of a to be equivalent to $\mathfrak{S}(\mathfrak{E}(a, A))$, this allows an explicit definition of the “contenders” function.

Definition 4.7. Let $s \in UR$ and let $A \subseteq D(s)$. Let S be a constraint set. Define the following:

$$\mathfrak{K}(A) = \{a \in A \mid \mathfrak{S}(\mathfrak{E}(a, A))\}$$

Corollary 4.8. For a constraint set S , a string $s \in UR$, and a set $A \subseteq D(s)$, it holds that $\mathfrak{K}(A) = \{a \in A \mid \mathfrak{S}(\mathfrak{E}(a, A))\}$ is the set of surface representations that are contenders with respect to S .

Thus we have a method of computing the contenders.

4.2 The Contender Semiring

Using the function \mathfrak{K} , we can improve the efficiency of contender generation by pruning harmonically-bounded candidates. As it turns out, we can generalize the procedure from the previous section to keep track not only of violation profiles, but also of all of the surface representation fragments that accru through the traversal of the graph of the FST. Riggle (2011) remarks that, by pruning the fragments from the arcs as soon as they cease to be possible contenders, we can produce an efficient algorithm.

Definition 4.9. Let K denote the power set $2^{\mathcal{V} \times 2^{\Delta^*}}$. Define the following:

$$U(A, B) = \bigcup_{(w, T) \in A \cup B} \{T \mid v = w\}$$

For $A, B \in K$, define the *unification* of A and B as follows:

$$A \sqcup B = \{(v, U(A, B)) \mid (v, S) \in A \cup B\}$$

Define the following operations on K :

$$\begin{aligned} A \sqcup_S B &= \mathfrak{K}(A \sqcup B) \\ A \times_S B &= \mathfrak{K}\left(\bigsqcup(A \ltimes B)\right) \end{aligned}$$

Furthermore, define the following constants:

$$0_{\mathcal{K}} = \{(\infty, \emptyset)\}, \quad 1_{\mathcal{K}} = \{(\emptyset, \{\varepsilon\})\}$$

In short, the union operator here collects all candidate fragments corresponding to unique violation profiles and returns all the ones that could be optimal under at least one ranking. The multiplication operator takes all possible combinations of (weight, fragment set) tuples and returns the ones that could be optimal under at least one ranking. We can now almost prove that we have a semiring, but first, a supporting lemma:

Lemma 4.10. *\mathfrak{K} is idempotent. That is, for a constraint set S , a string $s \in UR$, and a set $A \subseteq D(s)$, it holds that $\mathfrak{K}(\mathfrak{K}(A)) = \mathfrak{K}(A)$.*

Proof. By definition we know that $\mathfrak{K}(\mathfrak{K}(A)) \subseteq \mathfrak{K}(A) \subseteq A$. Then it suffices to show that $\mathfrak{K}(A) \subseteq \mathfrak{K}(\mathfrak{K}(A))$. Suppose $x \in \mathfrak{K}(\mathfrak{K}(A))$. Then under some ranking \prec of S , it's the least element in $\mathfrak{K}(A) \subseteq D(s)$. Therefore any $y \in \mathfrak{K}(A)$ obeys $x \prec y$. For some $y \in A \setminus \mathfrak{K}(A)$, that is, the set of non-contenders in A , then it must be the case that $x \prec y$. Therefore x is the least element in A under \prec , therefore x is a contender within A in addition to being a contender in $\mathfrak{K}(A)$. Thus it's the case that every $x \in \mathfrak{K}(\mathfrak{K}(A))$ is also in $\mathfrak{K}(A)$, thus $\mathfrak{K}(\mathfrak{K}(A)) = \mathfrak{K}(A)$. \square

Lemma 4.11. *With respect to a given constraint set S , the tuple $\mathcal{K}_S = (\mathfrak{K}(K), \sqcup_S, \times_S, 0_{\mathcal{K}}, 1_{\mathcal{K}})$ is a semiring.*

Proof. See appendix. \square

Hence, by previous results, we can fill matrices with values in \mathcal{K} and use the finite-state optimization procedure from the previous section to produce contenders efficiently.

4.3 Analysis and Remarks

Using the semiring optimization method described in the previous section, we obtain $2|s|$ instances of matrix multiplication, which means running $|Q|^3$ operations of \sqcup_S and \times_S . Each of these, of course, requires running \mathfrak{K} , which is bounded by $|s|^3$ comparisons under the computational equivalent of \min_{\prec} . (Insofar as both w and l and \min_{\prec} involve doing $|S|$ comparisons, one for each constraint.) This is obtained as follows: each instance of \mathfrak{S} involves the computation of w and l on each of $|E|$ ERCS. Because \mathfrak{S} must remove at least one ERC each time it recurs, the total number of semiring operations is bounded by $2|E|^2$. Thus the number of comparisons run by $\mathfrak{K}(A)$ is bounded by $2|A|^3|s|$. Now A is an arbitrary subset of the countably-infinite semiring D , and it's a bit difficult to bound. This gives, for the full routine, a grand total of $4|A|^3|S|^3$ operations of \min_{\prec} , \uplus , and \times .

5 A Python Implementation

After having explicated the mathematical task and exploring some of its properties, the corresponding software package should be easily navigable.

5.1 Preliminary Notes

The code contains a couple of auxiliary components, which could be discussed as formal language topics in their own right. One is the convenient construction of the finite-state machines: As can be seen in the code, both MAX and DEP are hard-coded, whereas I present a routine called `fst__from__prohibited__string` that generates FSTs from the strings they're meant to penalize. This, I believe, is faithful to a certain prediction of Prince and Smolensky that there are two types of constraints: faithfulness and markedness. Because faithfulness is for my purposes a closed class of constraints, MAX and DEP are left clumsily and directly constructed.

Second, I present an algorithm for intersecting FSTs that corresponds to a theorem proved in chapter four. The class of FSTs that can actually be intersected this way is nontrivial and presents interesting questions beyond the scope of the present project. The algorithm is used prolifically to generate example finite-state machines. (Perhaps I can run through an example later.)

Note that a substantial amount of the code is largely for book-keeping and for having convenient macros. I make relevant commentary on the important algorithms.

5.2 `arc.py`

This file is structural code for keeping track of the tuples that represent arcs. In contrast to the 5-tuples presented in chapter 3, I wrap the input, output, and weights into an additional object called a *label* for purposes of computational expedience. The arc and label objects can be seen to compute the functions $s(a)$, $t(a)$, $i(a)$, $o(a)$, and $w(a)$.

```
class Label():
```

```

def __init__(self, _input, _output, _violation):
    self.input = _input
    self.output = _output
    self.violation = _violation

def __repr__(self):
    return "Label(" + self.input.__repr__() + ", " + self.output.__repr__() +
        ", " + self.violation.__repr__() + ")"

class Arc():

    def __init__(self, _start, _end, _label):
        self.start = _start
        self.end = _end
        self.label = _label

    def __repr__(self):
        return "Arc(" + self.start.__repr__() + ", " + self.end.__repr__() + ", " +
            self.label.__repr__() + ")"

```

5.3 build__fst.py

This file is a library of functions for the construction of FSTs from banned strings and through intersection of existing FSTs. The workings of them are unfortunately beyond the scope of the project, and the relevant outputs of these functions, viewed in the GraphViz package, will verify that they are indeed the correct implementations of the FSTs treated in the former chapters. These images are included above.

```

from harmony import otimes
from unionfind import UnionFind
from arc import Label, Arc
from collections import Counter
from fst import FST, dummy_fst

def add_elision_arc(fst, node, in_seg):
    fst.arcs.add(Arc(node, node, Label(in_seg, '', Counter())))

def add_epenthesis_arc(fst, start, end, out_seg):
    fst.arcs.add(Arc(start, end, Label('', out_seg, Counter())))

def add_alteration_arcs(fst, start, end, in_seg, out_seg):

```

```

    if in_seg != '_':
        add_epenthesis_arc(fst, start, end, out_seg)
    fst.arcs.add(Arc(start, end, Label(in_seg, out_seg, Counter()))))

def violates_hard_constraint(input_string, arc, layer, length):
    if layer == length:
        return True
    if (arc.label.output == ''):
        if ((arc.label.input == '.') or (arc.label.input == '#') or
            (arc.label.input == '|') or (arc.label.input == '>') or
            (arc.label.input == '<')):
            return True

def states_set_product(m, n):
    return [(p, q) for p in m.states for q in n.states]

def states_mega_product(m, n):
    return [(p, q) for p in states_set_product(m, m) for q in
            states_set_product(n, n)]

def identical_labels_between(fst_1, start_1, end_1, fst_2, start_2, end_2):
    return [(label_1, label_2) for label_1 in fst_1.labels_between(start_1, end_1)
            for label_2 in fst_2.labels_between(start_2, end_2) if (label_1.input ==
            label_2.input) and (label_1.output == label_2.output)]

def similar_labels_between(fst_1, start_1, end_1, fst_2, start_2, end_2):
    # Return all label pairs with the same output, indexed by which output
    labels_1 = fst_1.labels_between(start_1, end_1)
    labels_2 = fst_2.labels_between(start_2, end_2)
    symbols = set([label_1.output for label_1 in labels_1] + [label_2.output for
        label_2 in labels_2])
    labels_lists = []
    for symbol in symbols:
        labels_list = [(label_1, label_2) for label_1 in labels_1 for label_2 in
            labels_2 if (label_1.output == label_2.output) and label_1.output ==
            symbol]
        if labels_list:
            labels_lists.append(labels_list)
    return labels_lists

def product_state(a, b):
    return a.__str__() + "x" + b.__str__()

def layered_state(a, b):
    return a.__str__() + "y" + b.__str__()

```

```

def longest_suffix(s, t):
    # Returns the element in t that is the longest proper suffix of s
    return max(filter(lambda x : (s.endswith(x) and s != x), t), key=(lambda x:
        len(x)))

def fst_from_prohibited_string(input_alphabet, output_alphabet, banned_string,
    violation_name):
    length = len(banned_string)
    fst = FST(set([""]), "", set(), set(), "")
    # Add arcs
    if length > 1:
        for i in range(1, length):
            fst.states.add(banned_string[0:i])
            add_alteration_arcs(fst, banned_string[0:i-1], banned_string[0:i], '_',
                banned_string[i-1])
    # Send a penalty arc to the longest valid suffix
    fst.arcs.add(Arc(banned_string[0:-1], longest_suffix(banned_string,
        fst.states), Label('_', banned_string[-1], Counter({violation_name: 1}))))
    # Add loopback arcs and return
    for state in fst.states:
        for char in input_alphabet:
            add_elision_arc(fst, state, char)
        for char in fst.chars_not_leaving(state, output_alphabet):
            add_alteration_arcs(fst, state, longest_suffix(state + char,
                fst.states), '_', char)
    return fst

def traverse_states(state_lookup, start):
    # Crude cycle-finding algorithm returning all states accessible from start
    arc_set = state_lookup[start]
    fst_states = set([arc.end for arc in arc_set])
    arc_set = set.union(*[state_lookup[state] for state in fst_states])
    new_states = set([arc.end for arc in arc_set])
    while not new_states.issubset(fst_states):
        fst_states = set.union(fst_states, new_states)
        arc_set = set.union(*[state_lookup[state] for state in fst_states])
        new_states = set([arc.end for arc in arc_set])
    return fst_states

def fst_intersect(m, n):
    arcs = set()
    state_lookup = dict((product_state(p, q), set()) for p, q in
        states_set_product(m, n))
    start = product_state(m.start, n.start)

```

```

# Compute arcs for each state pair
for ((x, y), (z, w)) in states_mega_product(m, n):
    labels_lists = similar_labels_between(m, x, y, n, z, w)
    elision_arcs = set()
    for labels_list in labels_lists:
        arcs_by_input = set()
        for (k, l) in labels_list:
            add_arc = False
            seg = ''
            if k.output == '':
                # Faithfulness constraint; cares about input
                if k.input == l.input:
                    if k.input not in elision_arcs:
                        add_arc = True
                        seg = k.input
                        elision_arcs.add(seg)
            elif ((k.input == '_' ) or (k.input == l.input)) and (l.input not in
                arcs_by_input):
                # Markedness constraint
                add_arc = True
                seg = l.input
                arcs_by_input.add(seg)
            elif (l.input == '_' ) and (k.input not in arcs_by_input):
                # Markedness constraint
                add_arc = True
                seg = k.input
                arcs_by_input.add(seg)
            if add_arc:
                intersection_arc = Arc(product_state(x, z), product_state(y, w),
                    Label(seg, k.output, otimes(k.violation, l.violation)))
                arcs.add(intersection_arc)
                state_lookup[intersection_arc.start].add(intersection_arc)
# Figure out the states reachable from the start
fst_states = traverse_states(state_lookup, start)
fst = FST(fst_states, start, fst_states, filter((lambda arc : arc.start in
    fst_states), arcs), 1)
return fst

```

The first three routines are simple macros, and the fourth exists to enforce certain structural constraints that ensure that the FSTs behave correctly, for instance, that they only have as many “layers” as there are characters in the input string. The next two are again macros for taking the Cartesian product of state sets, which is necessary for the intersection

algorithm. `identical__labels__between` collects all the identical arcs between two pairs of states, as is necessary for intersecting FSTs.

The first big routine, `fst__from__input`, takes an input string and builds the corresponding FST in the way described above. The first loop adds a copy of the base FST for each character in the input string and adjusts the arcs accordingly. It then adds states to the last layer, sets up the arcs leading to these states, and defines the set of end states.

The next big routine, `fst__from__prohibited__string`, adds a “chain” of states where the name of each is the first n letters of the banned string s , and adds arcs to take the n^{th} state to the $(n+1)^{th}$ with the alternation $s_{n+1} : s_{n+1}$. From the state q whose name contains all but the last character of s , it adds an arc to the state whose name is the longest suffix of q that takes the alternation $\sigma : \sigma$ for σ the last character of s and that assigns a violation profile $\{c : 1\}$ for c the constraint given as an argument. It then runs through all the states and adds the standard alternation arcs, sending them from each state to its longest proper suffix. Doing this allows cases like **AAA** to count **NOAA** twice correctly, as looping back to the original state would only allow the machine to count disjoint instances of **AA** rather than all substring occurrences.

Finally, the routine `fst__intersect` takes two FSTs and produces a new one that computes both of their constraints simultaneously. It is not the case that all finite-state transducers can be intersected in this manner - indeed, the task is only even easy to state for FSTs that correspond to systems of penalized regular expressions. A treatment of the full class of intersectable FSTs is beyond the scope of this project, as mentioned above, and Riggle (2011) and others restrict themselves to the class of regex-crunching machines (and therefore demand that finite-state OT constraints be regular expressions) for this reason. The routine takes the Cartesian product of the two sets of states and unifies arcs according to the following rule:

“There exists an arc from state (x, y) to state (z, w) with alternation L and violation $v \uplus w$ if and only if there exist arcs from both state x to state z with alternation L and violation v and from state y to state w with alternation L and violation w .”

5.4 contenders.py

This file contains the core algorithm that reduces optimization of the contenders semiring to matrix multiplication.

```
from harmony import succ, oplus, otimes, m_otimes, m_oplus, t_unit, t_ann,
    h_unbounded, unify_list, m_exponentiate
from fst import FST
import time

def char_adjacency_matrix(fst, input_segment, state_array):
    size = len(state_array)
    matrix = []
    counter = 0
    # Build the matrix
    for i in state_array:
        matrix.append([])
        for j in state_array:
            vec = []
            for arc in fst.arcs_between(i, j):
                # Keep only unique entries
                # Check first for the specific character and then for the catchall
                if (arc.label.input == input_segment) and not (arc.label.violation in
                    vec):
                    t = (arc.label.violation, [arc.label.output])
                    vec.append(t)
                elif (arc.label.input == '_') and not (arc.label.violation in vec):
                    t = (arc.label.violation, [arc.label.output])
                    vec.append(t)
                # If we can't move, assign infinity as the weight
                if len(vec) == 0:
                    vec = t_ann()
            matrix[counter].append(vec)
            counter += 1
    return matrix

def transition_matrix(fst, state_array, ranking):
    lambda_matrix = char_adjacency_matrix(fst, '', state_array)
    # Not moving is free
    for i in range(len(state_array)):
        lambda_matrix[i][i] = t_unit()
    return m_exponentiate(len(state_array), lambda_matrix, len(state_array),
        ranking)

def string_adjacency_matrix(fst, state_array, input_string, input_alphabet,
```

```

    ranking):
lambda_matrix = transition_matrix(fst, state_array, ranking)
# Index matrices by symbols
symbol_matrix = dict([(input_segment, m_otimes(char_adjacency_matrix(fst,
    input_segment, state_array), lambda_matrix, len(state_array), ranking))
    for input_segment in input_alphabet])
# Multiply them and return
input_matrix = lambda_matrix
for symbol in input_string[0:]:
    input_matrix = m_otimes(input_matrix, symbol_matrix[symbol],
        len(state_array), ranking)
return input_matrix

def contenders(fst, input_string, input_alphabet, ranking):
    state_array = []
    end_columns = []
    start_rows = []
    # Cache terminal states
    k = 0
    for i in fst.states:
        state_array.append(i)
        if i in fst.end:
            end_columns.append(k)
        if i == fst.start:
            start_rows.append(k)
        k += 1
    # Build the matrix
    M = string_adjacency_matrix(fst, state_array, input_string, input_alphabet,
        ranking)
    # Fetch contenders from the cells
    contenders = []
    for j in end_columns:
        for i in start_rows:
            contenders += M[i][j]
    # Unify, RCD again, and return
    contenders = unify_list(contenders, ranking)
    return h_unbounded(contenders, ranking)

```

The first routine computes the adjacency matrix of the subgraph of a given FST containing only arcs a for which $i(a) = \sigma$ for some character σ . To do this, it simply considers every tuple of states and adds a tuple $(w(a), \{o(a)\})$ to a list if a suitable arc exists between the two states. If no arcs exist, it assigns infinity as the weight. The list becomes the $(i, j)^{th}$

cell in the adjacency matrix. The second routine calls the first, but with an empty input segment, and replaces the diagonal cells with 0_K . This obeys the fact that transitioning from a state to itself and consuming no input segment is “free.”

The third routine simply multiplies all the adjacency matrices as described in lemma 3.35 and in Riggle (2011). The fourth routine is a wrapper for this that takes in the states in a Python list, caches the start and end states, computes the matrix multiplication, and then retrieves the values from the resulting matrix in the cells corresponding to optimal paths from start to finish. It adds them to a list and runs the contenders algorithm on this resulting union. As proven in previous chapters, this routine encapsulates the computation of \mathcal{OT} .

5.5 fsot.py

This file contains the main routine, which takes in an underlying representation from the user and runs the contender semiring optimization procedure.

```

from contenders import contenders
from arc import Arc, Label
from collections import Counter
from fst import FST
from build_fst import fst_from_prohibited_string, fst_intersect
from harmony import filter_for_hard_constraints
import re

ot_inf = 'inf'
ot_max = 'max'
ot_dep = 'dep'
ot_noc = 'noc'
ot_ons = 'ons'
ot_tf = 'tf'
test = 'test'
default = 'def'

# Debug stuff

debug_alphabet = {'A', 'B'}
debug_ranking = [ ot_dep, ot_max, 'no_aa', 'no_abab' ]

```

```

debug_max = FST( {1}, 1, {1}, { Arc(1, 1, Label('A', '', Counter({ot_max :
1}))), Arc(1, 1, Label('B', '', Counter({ot_max : 1}))), Arc(1, 1, Label('A',
'A', Counter())), Arc(1, 1, Label('B', 'B', Counter())), Arc(1, 1, Label('',
'A', Counter())), Arc(1, 1, Label('', 'B', Counter())) }, 1 )
debug_dep = FST( {1}, 1, {1}, { Arc(1, 1, Label('A', '', Counter())), Arc(1, 1,
Label('B', '', Counter())), Arc(1, 1, Label('A', 'A', Counter())), Arc(1, 1,
Label('B', 'B', Counter())), Arc(1, 1, Label('', 'A', Counter({ot_dep :
1}))), Arc(1, 1, Label('', 'B', Counter({ot_dep : 1}))) }, 1 )
debug_no_aa = fst_from_prohibited_string(debug_alphabet, debug_alphabet, "AA",
'no_aa')
debug_no_abab = fst_from_prohibited_string(debug_alphabet, debug_alphabet,
"ABAB", 'no_abab')

fst_debug = fst_intersect(debug_no_aa, fst_intersect(fst_intersect(debug_max,
debug_dep), debug_no_abab))

# Kinyarwanda stuff

tone_ranking = [ ot_max, ot_dep, ot_tf, 'pfl', 'ar' ]

tone_input_alphabet = {'H', 'L', '.', '#', '>', '<', '|'}
tone_output_alphabet = {'H', 'M', 'L', '.', '#', '>', '<', '|'}

# Hardcoded faithfulness - tone height alternations and max and dep
fst_tone_faith = FST( {1}, 1, {1}, { Arc(1, 1, Label('', 'L', Counter({ot_dep :
1}))), Arc(1, 1, Label('', 'M', Counter({ot_dep : 1}))), Arc(1, 1, Label('',
'H', Counter({ot_dep : 1}))), Arc(1, 1, Label('H', '', Counter({ot_max :
1}))), Arc(1, 1, Label('L', '', Counter({ot_max : 1}))), Arc(1, 1, Label('.',
'', Counter({ot_max : 1}))), Arc(1, 1, Label('H', 'L', Counter({ot_tf: 2}))),
Arc(1, 1, Label('L', 'H', Counter({ot_tf: 2}))), Arc(1, 1, Label('H', 'M',
Counter({ot_tf: 1}))), Arc(1, 1, Label('M', 'H', Counter({ot_tf: 1}))),
Arc(1, 1, Label('M', 'L', Counter({ot_tf: 1}))), Arc(1, 1, Label('L', 'M',
Counter({ot_tf: 1}))), Arc(1, 1, Label('.', '.', Counter())), Arc(1, 1,
Label('L', 'L', Counter())), Arc(1, 1, Label('H', 'H', Counter())), Arc(1, 1,
Label('#', '#', Counter())), Arc(1, 1, Label('>', '>', Counter())), Arc(1, 1,
Label('<', '<', Counter())), Arc(1, 1, Label('|', '|', Counter())) }, 1 )

fst_phrase_final_lowering1 = fst_from_prohibited_string(tone_input_alphabet,
tone_output_alphabet, "H<", 'pfl')
fst_phrase_final_lowering2 = fst_from_prohibited_string(tone_input_alphabet,
tone_output_alphabet, "M<", 'pfl')
fst_phrase_final_lowering = fst_intersect(fst_phrase_final_lowering1,
fst_phrase_final_lowering2)

fst_anticipatory_raising1 = fst_from_prohibited_string(tone_input_alphabet,

```

```

    tone_output_alphabet, "LH", 'ar')
fst_anticipatory_raising2 = fst_from_prohibited_string(tone_input_alphabet,
    tone_output_alphabet, "L.H", 'ar')
fst_anticipatory_raising3 = fst_from_prohibited_string(tone_input_alphabet,
    tone_output_alphabet, "L#H", 'ar')
fst_anticipatory_raising =
    fst_intersect(fst_intersect(fst_anticipatory_raising1,
    fst_anticipatory_raising2), fst_anticipatory_raising3)

# Assemble the FST
fst_kinyarwanda = fst_intersect(fst_tone_faith,
    fst_intersect(fst_anticipatory_raising, fst_phrase_final_lowering))

# Main routine

while True:
    input_form = raw_input("Enter input tonal pattern.\n\n> ")
    set_contenders = contenders(fst_kinyarwanda, input_form, tone_input_alphabet,
        tone_ranking)
    print ("Contenders: \n" + set_contenders.__str__())

```

5.6 fst.py

This file contains the representation of the FST, importing the arc and label objects from `arc.py`. All of the macros should be rather self-explanatory.

```

from harmony import succ, oplus, otimes, m_otimes, m_oplus, t_unit, t_ann,
    h_unbounded
from collections import Counter, deque
from arc import Arc, Label
from unionfind import UnionFind
import re

class FST():

    def __init__(self, _states, _start, _end, _arcs, _input_length):
        self.states = _states
        self.start = _start
        self.end = _end
        self.arcs = _arcs
        self.input_length = _input_length

```

```

def __repr__(self):
    return "FST(" + self.states.__repr__() + ", " + self.start.__repr__() + ",
        " + self.end.__repr__() + ", " + self.arcs.__repr__() + ", " +
        self.input_length.__repr__() + ")"

def to_dot(self):
    dot = "digraph {\n"

    for arc in self.arcs:
        dot += "_" + arc.start.__str__() + " -> " + "_" + arc.end.__str__() +
            "[label=\"" + arc.label.input.__repr__() + " : " +
            arc.label.output.__repr__() + " , " +
            arc.label.violation.__repr__()[8:-1] + "\"]; \n"
    dot += "}\n"

    return dot

def arcs_leaving(self, node):
    return filter(lambda a : (a.start == node), self.arcs)

def nodes_from(self, node):
    return [arc.end for arc in self.arcs_leaving(node)]

def arcs_arriving(self, node):
    return filter(lambda a : (a.end == node), self.arcs)

def external_arcs_arriving(self, node):
    return filter(lambda a : (a.start != node), self.arcs_arriving(node))

def arcs_between(self, node_from, node_to):
    return filter(lambda a : a.end == node_to, self.arcs_leaving(node_from))

def labels_between(self, start, end):
    return set(q.label for q in self.arcs_between(start, end))

def output_char_leaves(self, char, node):
    return filter(lambda a : a.label.output == char, self.arcs_leaving(node))

def chars_not_leaving(self, node, sigma):
    return filter(lambda char: len(self.output_char_leaves(char, node)) == 0,
        sigma)

def states_by_layer(self):
    queue = deque()
    if self.input_length > 0:

```

```

        for i in range(0, self.input_length + 1):
            states_in_layer = filter(lambda state : state.__str__().split("y")[1]
                                    == i.__str__(), self.states.copy())
            queue.append(states_in_layer)
        else:
            queue.append(self.states.copy())
    return queue

def dummy_fst():
    return FST(set([0]), 0, set(), set(), 0)

```

5.7 harmony.py

This file contains preliminary algebraic definitions that inducesemiring structure.

```

from collections import Counter
import itertools
import time
import re

def succ(v, w, ranking):
    for i in range(len(ranking)):
        if (v['inf'] == 0 and w['inf'] > 0):
            return 1
        if (v['inf'] > 0 and w['inf'] == 0):
            return -1
        if (v[ranking[i]] < w[ranking[i]]):
            return 1
        if (v[ranking[i]] > w[ranking[i]]):
            return -1
    return 0

def otimes(v, w):
    if (v['inf'] == 0 and w['inf'] == 0):
        return v + w
    else:
        return Counter({'inf': 1})

def oplus(v, w, ranking):
    if (succ(v, w, ranking) == 1):
        return v
    else:

```

```

    return w

def erc(a, b, ranking):
    (aweight, astrset) = a
    (bweight, bstrset) = b
    (a_wins, b_wins) = (0, 0)
    # Sanity check
    if (aweight['inf'] > 0):
        if (bweight['inf'] > 0):
            return (a_wins, b_wins)
        else:
            return (a_wins, 2**(len(ranking)))
    elif (bweight['inf'] > 0):
        return (2**(len(ranking)), b_wins)
    for i in range(len(ranking)):
        if (aweight[ranking[i]] < bweight[ranking[i]]):
            a_wins |= (2**i)
        if (bweight[ranking[i]] < aweight[ranking[i]]):
            b_wins |= (2**i)
    return (a_wins, b_wins)

def consistent(erc_list, ranking):
    if not erc_list:
        return True
    # If there exists any violation that simply beats everything
    if any([all([w & 2**i for (w, l) in erc_list]) for i in range(len(ranking))]):
        return True
    (w_set, l_set) = (0, 0)
    for (w, l) in erc_list:
        w_set |= w
        l_set |= l
    if l_set == 0:
        return True
    if (w_set == 0) or ((~w_set | l_set) == -1):
        return False
    # Evaluate ERC sets - keep e if w(e) \sge l(e_list)
    new_erc_list = [(w, l) for (w, l) in erc_list if (~w | l_set) == -1]
    return consistent(new_erc_list, ranking)

def simply_unbounded(k, klist, ranking):
    (v, w) = k
    for (a, b) in klist:
        # Check if a[r] \geq v[r] in all cases
        bounded = True
        for r in ranking:

```



```

        if v[r] < a[r]:
            bounded = False
        if bounded:
            return True
    return False

def filter_for_hard_constraints(mylist):
    constraints =
        re.compile(r'((L|M|H|h){3})|((\.\.)(\>\.)(\.\<)|(\#\.)|(\.\#)|(\#<)|(\>\#))')
    valid = lambda x : not constraints.search(x)
    return [(v, filter(valid, S)) for (v, S) in mylist]

def h_unbounded(klist, ranking):
    # Filter in a couple passes
    klist = filter((lambda (v, w) : v['inf'] == 0), klist)
    klist = filter(lambda a : simply_unbounded(a, klist, ranking), klist)
    klist = filter_for_hard_constraints(klist)
    return filter((lambda k : consistent([erc(k, k_, ranking) for k_ in klist],
        ranking)), klist)

def v_to_str(profile):
    if profile['inf'] > 0:
        return "\infty"
    if profile == Counter():
        return "\epsilon"
    # Return an alphabetical string of key-count pairs
    keys = sorted(profile.keys())
    mystring = ""
    for key in keys:
        mystring += key + ":" + profile[key].__str__() + ";"
    return mystring

def str_to_v(mystring):
    # Sanity check
    if (mystring == "\epsilon"):
        return Counter()
    if (mystring == "\infty"):
        return Counter({'inf' : 1})
    kvs = mystring.split(';')
    v = Counter()
    for kv in kvs:
        if (kv != ""):
            k_v = kv.split(':')
            v[k_v[0]] = int(k_v[1])
    return v

```

```

def unify_list(klist, ranking):
    # Cast to strings so we can use a set
    strs = set([v_to_str(v) for (v, s) in klist])
    union = []
    # Collect, rebuild the tuples, and return
    for string in strs:
        stringset = set([])
        for (v, s) in klist:
            if(v_to_str(v) == string):
                stringset |= set(s)
        union.append((str_to_v(string), list(stringset)))
    return union

def s_otimes(s1, s2):
    return [t1 + t2 for t1 in s1 for t2 in s2]

def t_ann():
    return [(Counter({'inf' : 1}), [])]

def t_unit():
    return [(Counter([]), [""])]

def t_oplus(a, b, ranking):
    contenders = h_unbounded(unify_list(a + b, ranking), ranking)
    # Sanity check
    if not contenders:
        return t_ann()
    return contenders

def t_otimes(a, b, ranking):
    # Sanity check
    if (not a) or (not b):
        return t_ann()
    contenders = h_unbounded([(otimes(v1, v2), s_otimes(s1, s2)) for (v1, s1) in a
        for (v2, s2) in b], ranking)
    # Again
    if not contenders:
        return t_ann()
    return contenders

def m_oplus(m, n, size):
    return [[t_oplus(m[row][column], n[row][column]) for column in range(size)]
        for row in range(size)]

```

```

def m_dot(m, n, row, column, size, ranking):
    m_sum = t_ann()
    for k in range(size):
        m_sum = t_oplus(m_sum, t_otimes(m[row][k], n[k][column], ranking), ranking)
    m_sum = h_unbounded(unify_list(m_sum, ranking), ranking)
    # Sanity check
    if not m_sum:
        m_sum = t_ann()
    return m_sum

def m_rowdot(a, b, size, ranking):
    m_sum = t_ann()
    for k in range(size):
        m_sum = t_oplus(m_sum, t_otimes(a[k], b[k], ranking), ranking)
    return m_sum

def m_otimes(m, n, size, ranking):
    return [[m_dot(m, n, row, column, size, ranking) for column in range(size)]
            for row in range(size)]

def m_exponentiate(n, matrix, size, ranking):
    if n == 1:
        return matrix
    # Do this binarily to save cycles
    half = int(n/2)
    other_half = n - n/2
    new_matrix = m_otimes(m_exponentiate(half, matrix, size, ranking),
                          m_exponentiate(half, matrix, size, ranking), size, ranking)
    return new_matrix

def m_equals(m, n, size):
    for row in range(size):
        for column in range(size):
            if m[row][column] != n[row][column]:
                return False
    return True

```

The first routine implements the harmonic inequality \succ . It runs through the ranking of constraints and returns 1 if the first multiset has fewer violations at the highest-ranked constraint and -1 if it has more. It returns 0 if it terminates without deciding - that is, if the multisets are equal. The next two routines are implementations of \uplus , and \min_{\succ} , respectively. The following two routines are sort algorithms using \succ for sets of arcs and of

multisets, respectively.

The next collection of routines has to do with ERC computation. The function `erc` produces the tuple $(W(a, b), L(a, b))$ considered as a pair of binary strings representing wins and losses, respectively. It then uses binary operations to efficiently run the comparisons described in chapter four. The routine `consistent` quite directly implements \mathfrak{S} as recursively defined in chapter four, although the binary operations used for computational efficiency are a bit obtuse. Finally, `h_unbounded` computes \mathfrak{K} through multiples filters, again for the sake of efficiency.

The next set of routines handle the “unification” of a list of tuples. The task is to take a Python list object containing tuples (v, S) where v is a Counter representing a violation profile and S is a Python list containing strings, and to return a new list containing values (w, T) where all values w are unique and T is the union of all S where (v, S) was an entry in the original list and $v = w$. This is difficult in Python because the Counter object cannot be hashed or easily compared, and therefore I have opted to write routines to represent violation profiles in canonical string form for easy comparison. The routine `unify_list` can thus use the Python set object for easy collection of unique entries and thereby perform the unification rather efficiently.

Next, the functions `t_ann` and `t_unit` return representations of the values $0_{\mathcal{K}}$ and $1_{\mathcal{K}}$, respectively. Then `t_oplus` and `t_otimes` implement \sqcup_S and \times_S , respectively, with some sanity checks on the values.

Finally, the last four routines implement matrix-semiring algebra for \mathcal{K}_S . This unfortunately forecloses on directly using numpy or other libraries optimized for matrix computations, and conceivably puts real constraints on run time.

5.8 A Simple Example: $\{A, B\}$

In the simplest case, the program is set to compute contenders over the alphabet $\Sigma = \{A, B\}$ with constraints $S = \{\text{DEP}, \text{MAX}, \text{NOAA}, \text{NOAB}\}$, where DEP and MAX are the usual faithfulness constraints and NOAA and NOABAB penalize the strings AA and $ABAB$,

respectively. Below are the machines representing these constraints:

The final aggregate product takes the following form:

Running `fsot.py` will expand the above machine by following the procedure `fst__from__input`: adding a new copy M_σ for each symbol σ in the input string s and, for each arc with an empty input segment, keeping it in place, but for each arc that originates in M_σ and has σ as an input segment, shifts the target of said arc to M_τ for τ the next input segment. This results in substantially larger machines that do not well fit in L^AT_EX.

As can be seen, the code correctly counts the number of violations of the `noAA` and `noABAB` constraints even in the cases of overlapping substrings, and it excludes bounded options such as `AAB` for the case of $s = \text{ABAB}$. Unfortunately, a lack of runtime optimization makes computation on strings longer than six characters practically infeasible at the present state of the code and examples on less contrived sets must await improvements in speed.

5.9 A Complicated Example: Kinyarwanda

The software package can also be used to demonstrate a case of the inadequacy of OT analyses. For this, I bring in data from Kinyarwanda (Mpiranya forthcoming) and try to build constraints to explain even a simple case. Tonal phenomena are notoriously difficult territory for phonological theory, having lead to the development of autosegmental phonology (Goldsmith 1976) to account for spreading phenomena in Eastern Bantu languages, including Kinyarwanda. The following example should demonstrate the value of autosegmental representations as an enrichment of the string representation that the present model has adopted.

In short, Kinyarwanda data permit the following generalizations:

- Tones are underlying binary, being either high (H) or low (L). Mpiranya also reports a surface tone that is “raised” anticipatorily before high tones;
- Only one tone occupies each mora;

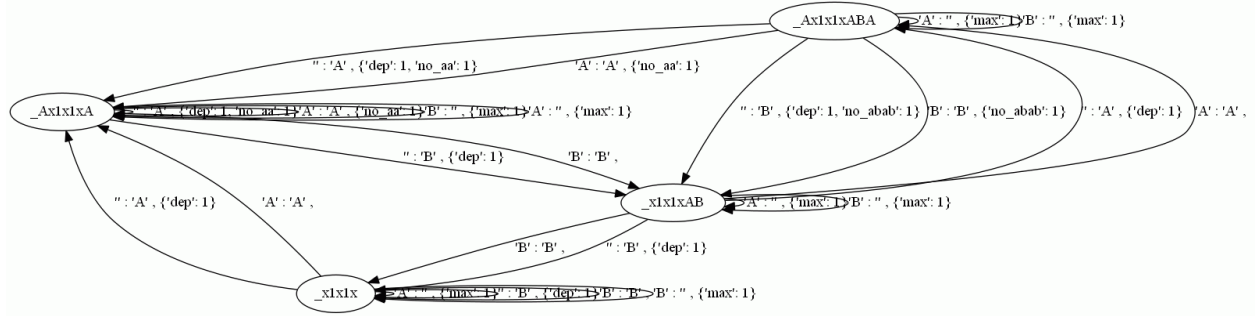


Figure 8: Machine encoding the constraint set S

| Input | Output |
|--------|---|
| AA | [(Counter({'max': 1}), ['A']), (Counter({'no_aa': 1}), ['AA']), (Counter({'dep': 1}), ['ABA'])] |
| AAA | [(Counter({'no_aa': 2}), ['AAA']), (Counter({'dep': 3}), ['ABBABA']), (Counter({'max': 2}), ['A'])] |
| AB | [(Counter(), ['AB'])] |
| ABAB | [(Counter({'max': 1}), ['ABA', 'ABB', 'BAB']), (Counter({'no_abab': 1}), ['ABAB']), (Counter({'dep': 1}), ['ABBAB'])] |
| AABAB | [(Counter({'dep': 1, 'no_aa': 1}), ['AABBAB']), (Counter({'max': 1, 'no_aa': 1}), ['AABA', 'AABB']), (Counter({'max': 2}), ['ABA', 'ABB', 'BAB']), (Counter({'no_abab': 1, 'no_aa': 1}), ['AABAB']), (Counter({'max': 1, 'no_abab': 1}), ['ABAB']), (Counter({'no_abab': 2, 'dep': 1}), ['ABABAB']), (Counter({'dep': 3}), ['ABBABBAB'])] |
| ABABAB | [(Counter({'no_abab': 2}), ['ABABAB']), (Counter({'max': 1}), ['ABBAB']), (Counter({'dep': 2}), ['ABBABBAB'])] |

Figure 9: Output of `fsot.py` for various short inputs

| Datum | Input | Output |
|----------------------|----------------|----------------|
| <i>umugoré</i> | >L.L.L.H< | >L.L.M.L< |
| <i>umugoré mwīzá</i> | >L.L.L.H#LL.H< | >L.L.M.H#LM.L< |

Figure 10: Representation of a simple tonal phenomenon

- Vowels can be mono- or bi-moraic and length does not vary phonologically, only morphologically;
- Tones are lexical and only one occurs per morpheme;
- Tones are not affected by the phonemic qualities of the moras they occupy.

One of the simplest examples we want to consider is the case of words like *umugoré* (“woman”), which bear high tones but which surface in isolation with the first tone raised and the second lowered. However, if the word occurs before another, e.g. in *umugoré mwīzá* (“beautiful woman”), the high tone should stay in place along with the anticipatory raising. Of course, it should be the case that the second word also undergoes anticipatory raising and phrase-final lowering.

To account for this, we choose the following alphabets:

$$\Sigma = \{>, <, \#, ., L, H\}$$

$$\Delta = \{>, <, \#, ., L, M, H\}$$

The first two characters represent the start and end of a phrase, respectively. Then we include a word boundary and a syllable boundary. Because we’re only concerned with tonal phenomena, having low, mid, and high should suffice. From this, we simply need to get the alternations in figure 7.

To do this, we can define some intuitive constraints. First, we demand a certain faithfulness constraint. This is hard-coded in the code above but is not inscrutable because it’s a single-state machine. I consider three different faithfulness constraints: the usual MAX and DEP, as well as a constraint TF (“tonal faithfulness”) that allows tones to alternate without having to incur both of the former. It receives one violation for moving between H and L or

between L and M, and two violations for moving between L and H. (This might immediately tip one off that something is amiss: that we have to actually allow for alternation arcs rather than strict input-only or output-only violations seems fishy, but if we imagine that the alternations actually refer to the loss or addition of features, i.e. $[+\text{tone}]$, then this still holds some theoretical water.) Moreover, constraints must be added to prevent the epenthesis or elision of the structural elements. This is reflected in the code, where strings that involve empty syllables and arcs that epenthesize or elide boundaries are expressly removed. Then, as can be seen in the code, I define the following:

| Constraint | Abbreviation | Banned Expression |
|-----------------------|--------------|-------------------------------|
| Phrase-final lowering | PFL | $(H<) \mid (M<)$ |
| Anticipatory raising | AR | $(L.H) \mid (L\#H) \mid (LH)$ |

We can then declare the line:

```
fst__kinyarwanda = fst__intersect(fst__tone__faith,
                                   fst__intersect(fst__anticipatory__raising,
                                                  fst__phrase__final__lowering))
```

Running the program with the inputs in figure 7 now yields the results in figure 8. (The GraphViz package does make the machine renderable, but it takes some care to replace the characters ‘.’ and ‘#’ with alphanumeric substitutes and the image that results has dimensions 5000×200 pixels and is difficult to display effectively in L^AT_EX.) As we can see, the desired forms don’t appear because the two-layer model cannot rule optimal an alternation that simultaneously both lowers the final tone and that raises the previous tone. This would incur two violations of TF and there’s nothing especially wrong, when considering only the output, with the string $>L.L.L.L<$. Additionally, notice the prevalence of epenthesis. Kinyarwanda bans this, but this does not preclude it typologically. It does imply, however, that DEP will have to be the highest-ranked constraint for Kinyarwanda tones if the phenomena

| Input | Output |
|----------------|---|
| >L.L.L.H< | <pre>[(Counter({'dep': 1, 'ar': 1}), ['>L.L.L.HL<']), (Counter({'tf': 1, 'pfl': 1}), ['>L.L.M.H<', '>L.L.L.M<']), (Counter({'dep': 1, 'pfl': 1}), ['>L.L.LM.H<', '>L.L.L.MH<']), (Counter({'ar': 1, 'pfl': 1}), ['>L.L.L.H<']), (Counter({'dep': 2}), ['>L.L.LM.HL<']), (Counter({'tf': 2}), ['>L.L.L.L<'])]</pre> |
| >L.L.L.H#LL.H< | <pre>(Counter({'tf': 2, 'pfl': 1}), ['>L.L.L.M#LM.H<', '>L.L.M.H#LM.H<', '>L.L.M.H#LL.M<', '>L.L.L.M#LL.M<']), (Counter({'ar': 2, 'dep': 1}), ['>L.L.L.H#LL.HL<']), (Counter({'tf': 3}), ['>L.L.L.M#LL.L<', '>L.L.M.H#LL.L<']), (Counter({'dep': 2, 'pfl': 1}), ['>L.L.L.MH#LL.MH<', '>L.L.LM.H#LL.MH<']), (Counter({'tf': 2, 'ar': 1}), ['>L.L.L.H#LL.L<']), (Counter({'ar': 2, 'pfl': 1}), ['>L.L.L.H#LL.H<'])]</pre> |

Figure 11: Results for $S = \{\text{MAX}, \text{DEP}, \text{TF}, \text{PFL}, \text{AR}\}$

are to comport with this OT analysis.

To move forward is somewhat difficult. There are a number of ways to “cheat” the analysis, of course, in the tradition of OT, for instance adding a character **H** that refers to a “lowered high tone.” Allowing this to alternate with **H** for only one violation of **TF**, as well as penalizing $(Lh) | (L.h) | (L\#h)$ with one violation of **AR** yields the possibility of getting the following violation profile:

$$(\text{Counter}(\{'tf': 2\}), ['>L.L.L.L<', '>L.L.M.h<'])$$

However, not only have we effectively plugged the dam in our theory, but it would still remain to have to make it more expensive to lower twice than to raise and lower. This could in turn be fixed by making it more expensive to lower than to raise, but at this point the theory has effectively descended into parody.

Another answer is to punt and declare anticipatory raising to be an entirely prosodic phenomenon, but this begs the question as to where OT’s explanatory power terminates. Is

tone contouring beyond the scope of OT? Does it only handle segmental alternations?

Of course, a review of the literature bodes poorly for this analysis: after Goldsmith (1976), Bantu tones are almost universally treated in terms of autosegmental representations rather than simple strings. This becomes clearer after investigating more complex tonal patterns in Mpiranya (forthcoming), such as the infamous unbounded spreading under Meeussen’s rule.

Unbounded spreading proves quite problematic indeed for an OT approach. In short, Kinyarwanda is a prefixing language that appears to contain a special morpheme boundary (hereafter denoted |) that occurs before verb stems. Many verbs bear lexical tones and these can give rise to remarkable alternations. Mpiranya offers the following data:

| Phrase | Translation |
|-------------------------|-----------------------------------|
| <i>kubona</i> | “to see” |
| <i>kunahakubona</i> | “to even see you there” |
| <i>kunabihakubonana</i> | “to even see you there with that” |

For our analysis, since we (like Mpiranya) take the segmental qualities to be irrelevant, this presents as

| UR | SR |
|-----------------|-----------------|
| L H.L | L H.L |
| L.L.L.L H.L | L.H.L H.L.L |
| L.L.L.L.L H.L.L | L.H.L.H L.H.L.L |

In light of the already-pressing difficulty of constraining the alternations of H and L, it seems particularly difficult to motivate the every-other-syllable spreading, especially when considering that no such spreading occurs in roots with no lexical tone. It seems intractable to avoid direct reference to this, which, if we consider tones as strings, is a global property of the representation. Because OT’s pride lies in strict locality, it seems that the present model is inadequate.

However, simply enriching the representation bodes poorly for the Economy principle: Eisner (1997) implements an FST approach to autosegmental representations, but concludes that the computational complexity of even generating outputs is NP-hard in the size of the grammar. Adding constraints and trying to run an optimization algorithm could be far too complex for theoretical consideration. This suggests that the learnability of an OT model on autosegmental representations makes it rather implausible.

6 Appendix: Proofs

Here I give full proofs omitted in the main body of the paper.

Lemma (2.6). *Let $M = (X, \cdot, \varepsilon)$ and $N = (Y, *, \epsilon)$ be monoids. Then $M \times N = (X \times Y, \otimes, \emptyset)$ is a monoid, where $(m, n) \otimes (\tilde{m}, \tilde{n}) = (m \cdot \tilde{m}, n * \tilde{n})$ and $\emptyset = (\varepsilon, \epsilon)$.*

Proof. We show that the new operator is associative. Take $(x_0, y_0), (x_1, y_1), (x_2, y_2) \in M \times N$. Then we have

$$\begin{aligned} ((x_0, y_0) \otimes (x_1, y_1)) \otimes (x_2, y_2) &= (x_0 \cdot x_1, y_0 * y_1) \otimes (x_2, y_2) \\ &= ((x_0 \cdot x_1) \cdot x_2, (y_0 * y_1) * y_2) \\ &= (x_0 \cdot (x_1 \cdot x_2), y_0(y_1 * y_2)) \\ &= (x_0, y_0) \otimes (x_1 \cdot x_2, y_1 * y_2) \\ &= (x_0, y_0) \otimes ((x_1, y_1) \otimes (x_2, y_2)) \end{aligned}$$

The third step follows from the postulate that M and N are monoids, i.e. that \cdot and $*$ are associative binary operations. Then, we show that \emptyset is an identity element for \otimes :

$$\begin{aligned} (x, y) \otimes \emptyset &= (x, y) \otimes (\varepsilon, \epsilon) \\ &= (x \cdot \varepsilon, y * \epsilon) \\ &= (x, y) \\ &= (\varepsilon \cdot x, \epsilon \cdot y) \\ &= (\varepsilon, \epsilon) \otimes (x, y) \\ &= \emptyset \otimes (x, y) \end{aligned}$$

Therefore the product is a monoid. □

Lemma (3.11). *Let C be a constraint alphabet. Then $\mathcal{V} = (V_C, \min_{\prec}, \uplus, \emptyset, \infty)$ is a semiring. Moreover, under any total order $<$ on C , \mathcal{V} together with the harmonic inequality induced by $<$ is an ordered semiring.*

Proof. Clearly the operators form the desired monoids. Multiset union splits into naive set union and addition, both of which are commutative, hence it's commutative. Then we can check:

- Left distributivity holds:

$$\begin{aligned}
(a \min_{\prec} b) \uplus c &= \min_{\prec} \{a, b\} \uplus c \\
&= \min_{\prec} \{a \uplus c, b \uplus c\} \\
&= (a \uplus c) \min_{\prec} (b \uplus c)
\end{aligned}$$

- Right distributivity holds:

$$\begin{aligned}
a \uplus (b \min_{\prec} c) &= a \otimes \min_{\prec} \{b, c\} \\
&= \min_{\prec} \{a \uplus b, a \uplus c\} \\
&= (a \uplus b) \min_{\prec} (a \uplus c)
\end{aligned}$$

- The identities hold:

$$\begin{aligned}
a \min_{\prec} \infty &= a \\
a \uplus \emptyset &= \emptyset \uplus a = a \\
a \uplus \infty &= \infty \uplus a = \infty
\end{aligned}$$

Finally, the harmonic inequality as extended to V_C by the definition above is a partial order, completing the proof. \square

Lemma (3.14). *Given Σ, C, S, s as above, $\mathcal{V}(s) \subseteq \mathcal{V}$ and $(\mathcal{V}(s), \min_{\prec}, \uplus, \infty, \emptyset)$ is indeed a semiring that inherits a partial order from any total order on C .*

Proof. The subset inclusion is clear, since \mathcal{V} is the collection of all possible multisets. It then

inherits the semiring structure. Because it's defined to be closed under these operators, $\mathcal{V}(s)$ is therefore a semiring. \square

Lemma (3.21). *Let Σ be an alphabet and let C be a constraint alphabet. Let R be a finite set of binary constraints indexed over C . Then the following hold:*

- *R together with any total order $<$ naturally induces a constraint set S ;*
- *For every $s \in UR$, there exists an FST T with weights in $V_C(s)$ that computes ρ_R^C ;*
- *$\mathcal{V}(s)$ is the semiring formed by $\{\rho_R^C(s, \tilde{s}) \mid \tilde{s} \in \Delta^*\}$ under \min_{\prec} and \uplus , that is, T computes single entries in $\mathcal{T}(s)$.*

Proof. Unfortunately, a full proof of correctness would get lost in the weeds quite quickly, and the topic of FST composition goes well beyond the scope of the present thesis. I present a sketch of the proof; a better discussion of the issue can be found in Mohri (2002) and elsewhere.

- This follows immediately from Definition 3.12;
- Consider first each relation ρ_r for $r \in C$. Because each is equivalent to a binary relation, it follows from Lemma 2.13 that each corresponds to an FST T_r . because each computes a regular relation, there exists a class of arcs that are traversed upon the acceptance of the final character of a string in the regular relation ρ_r . Consider T_r to bear a weight of $\{r : 1\}$ on each of these arcs and a weight of \emptyset on each other arc. Let T be the intersection of the collection $\{T_r \mid r \in C\}$, which is guaranteed to exist because of the regularity of the relations. Then T computes ρ_R^C .
- Invoking the previous result, we take it to be the case that for each substring of s that matches each relation ρ_r , a weight $\{r : 1\}$ is incurred. Therefore the machine T assigns the profile

\square

Lemma (3.32). *Let $\mathcal{M}(R)$ be the collection of R -valued matrices, let $\bar{0}$ denote the matrix with entries all $0 \in R$, and let Id denote the identity matrix, i.e. that with entries $1 \in R$ along the diagonal and $0 \in R$ everywhere else. Then $(\mathcal{M}(R), +, \cdot, \bar{0}, \text{Id})$ is itself a semiring.*

Proof. First, see that $(\mathcal{M}(R), +, \bar{0})$ is a commutative monoid, since pointwise \oplus is associative and commutative by assumption and takes 0 as an identity by definition. Similarly, $(\mathcal{M}(R), \cdot, \text{Id})$ is a monoid - we can see that semiring-matrix multiplication is associative:

$$\begin{aligned} ((L \cdot M) \cdot N)_{i,j} &= \bigoplus_{\ell=0}^n ((L \cdot M)_{i,\ell} \otimes N_{\ell,j}) \\ &= \bigoplus_{\ell=0}^n \left(\left(\bigoplus_{k=0}^n (L_{i,k} \otimes M_{k,\ell}) \right) \otimes N_{\ell,j} \right) \end{aligned}$$

Then, by the right-distributive property, we get

$$((L \cdot M) \cdot N)_{i,j} = \bigoplus_{\ell=0}^n \left(\bigoplus_{k=0}^n (L_{i,k} \otimes M_{k,\ell} \otimes N_{\ell,j}) \right)$$

Commutativity then allows us to interchange the finite sums, resulting in

$$((L \cdot M) \cdot N)_{i,j} = \bigoplus_{k=0}^n \left(\bigoplus_{\ell=0}^n (L_{i,k} \otimes M_{k,\ell} \otimes N_{\ell,j}) \right)$$

And then, by left-distribution,

$$\begin{aligned} ((L \cdot M) \cdot N)_{i,j} &= \bigoplus_{k=0}^n \left(L_{i,k} \otimes \left(\bigoplus_{\ell=0}^n (M_{k,\ell} \otimes N_{\ell,j}) \right) \right) \\ &= \bigoplus_{k=0}^n (L_{i,k} \otimes (M \cdot N)_{k,j}) \\ &= (L \cdot (M \cdot N))_{i,j} \end{aligned}$$

Additionally, we show that Id is the identity for semiring-matrix multiplication:

$$\begin{aligned}
(M \cdot \text{Id})_{i,j} &= \bigoplus_{k=0}^n M_{i,k} \otimes \text{Id}_{k,j} \\
&= M_{i,j} \\
(\text{Id} \cdot M)_{i,j} &= \bigoplus_{k=0}^n \text{Id}_{i,k} \otimes M_{k,j} \\
&= M_{i,j}
\end{aligned}$$

Thus it's a monoid.

Finally, we demonstrate distributivity. On the left, we have

$$\begin{aligned}
((L + M) \cdot N)_{i,j} &= \bigoplus_{k=0}^n ((L_{i,k} \oplus M_{i,k}) \otimes N_{k,j}) \\
&= \bigoplus_{k=0}^n ((L_{i,k} \otimes N_{k,j}) \oplus (M_{i,k} \otimes N_{k,j})) \\
&= \bigoplus_{k=0}^n (L_{i,k} \otimes N_{k,j}) \oplus \bigoplus_{k=0}^n (M_{i,k} \otimes N_{k,j}) \\
&= (L \cdot N)_{i,j} \oplus (M \cdot N)_{i,j}
\end{aligned}$$

By our definition of $+$, we conclude that $((L + M) \cdot N) = (L \cdot N) + (M \cdot N)$. And on the

right, we similarly have

$$\begin{aligned}
(L \cdot (M + N))_{i,j} &= \bigoplus_{k=0}^n (L_{i,k} \otimes (M_{i,k} \oplus N_{k,j})) \\
&= \bigoplus_{k=0}^n ((L_{i,k} \otimes M_{i,k}) \oplus (L_{i,k} \otimes N_{k,j})) \\
&= \bigoplus_{k=0}^n ((L_{i,k} \otimes M_{i,k})) \oplus \bigoplus_{k=0}^n ((L_{i,k} \otimes N_{k,j})) \\
&= (L \cdot M)_{i,j} \oplus (L \cdot N)_{i,j}
\end{aligned}$$

Again, this suffices to conclude that $(L \cdot (M + N)) = (L \cdot M) + (L \cdot N)$. This proves the distributive properties, and thus that semiring-matrices form a semiring. \square

Lemma (3.34). *The tuple $\mathcal{D} = (D, \sqcup, \bowtie, 0_{\mathcal{D}}, 1_{\mathcal{D}})$ is a semiring.*

Proof. First, $(D, \sqcup, 0_{\mathcal{D}})$ is a monoid, since minimization is associative and has ∞ as an identity, and by our definition of \sqcup , since \emptyset is an identity for set union, (∞, \emptyset) qualifies as an identity for the tupled operator.

Second, $(D, \bowtie, 1_{\mathcal{D}})$ is a monoid, since both operations in the tuples are associative, the empty set is an identity for multiset union, and product-set-concatenation of a set A by the empty string preserves every element in A .

Finally, observe that minimization and union are commutative, hence so is \sqcup , and so it remains only to prove distributivity. On the left, consider the expression $((u, A) \sqcup (v, B)) \bowtie (w, C)$. If $u = v$, then the expression reduces to

$$\begin{aligned}
((u, A) \sqcup (v, B)) \bowtie (w, C) &= (u, A \cup B) \bowtie (w, C) \\
&= (u \uplus w, \{a :: c \mid a \in A \cup B, c \in C\}) \\
&= (u \uplus w, \{a :: c \mid a \in A, c \in C\}) \uplus (v \uplus w, \{b :: c \mid b \in B, c \in C\}) \\
&= ((u, A) \bowtie (w, C)) \sqcup ((v, B) \bowtie (w, C))
\end{aligned}$$

Otherwise, if \sqcup returns, without loss of generality, (u, A) , we have

$$\begin{aligned}
((u, A) \sqcup (v, B)) \times (w, C) &= (u, A) \times (w, C) \\
&= (u \uplus w, \{a :: c \mid a \in A, c \in C\}) \\
&= (u \uplus w, \{a :: c \mid a \in A, c \in C\}) \sqcup (v \uplus w, \{b :: c \mid b \in B, c \in C\}) \\
&= ((u, A) \times (w, C)) \sqcup ((v, B) \times (w, C))
\end{aligned}$$

On the right, the same line of reasoning holds. If we have harmonic equivalence, then

$$\begin{aligned}
(u, A) \times ((v, B) \sqcup (w, C)) &= (u \uplus v, \{a :: b \mid a \in A, b \in B \cup C\}) \\
&= (u \uplus v, \{a :: b \mid a \in A, b \in B\}) \uplus (u \uplus w, \{a :: c \mid a \in A, c \in C\}) \\
&= ((u, A) \times (v, B)) \sqcup ((u, A) \times (w, C))
\end{aligned}$$

And otherwise, without loss of generality, we have

$$\begin{aligned}
(u, A) \times ((v, B) \sqcup (w, C)) &= (u, A) \times (v, B) \\
&= (u \uplus v, \{a :: b \mid a \in A, b \in B\}) \\
&= (u \uplus v, \{a :: b \mid a \in A, b \in B\}) \sqcup (u \uplus w, \{a :: c \mid a \in A, c \in C\}) \\
&= ((u, A) \times (v, B)) \sqcup ((u, A) \times (w, C))
\end{aligned}$$

This exhausts the criteria, hence we conclude that the tuple is indeed a semiring. \square

Lemma (3.35). *Let $s = s_1 s_2 \dots s_n \in \Sigma^*$ and let \prec order \mathcal{V} . Then the $(i, j)^{th}$ entry in $M(s)$ is the tuple (w, S) where w is the \prec -minimum weight of any path from i to j with input s , and where S is the collection of strings corresponding to paths with this weight.*

Proof. Because the primitive output matrices are defined to contain the shortest adjacencies, it suffices to demonstrate that their products contain minimum-weight paths. This follows

directly from the definition of \sqcup : for primitive matrices A and B , we have:

$$\begin{aligned} (AB)_{i,j} &= \bigoplus_{k=1}^{|Q|} A_{i,k} \otimes B_{k,j} \\ &= (A_{i,1} \uplus B_{1,j}) \sqcup (A_{i,2} \uplus B_{2,j}) \sqcup \dots \sqcup (A_{i,|Q|} \uplus B_{|Q|,j}) \end{aligned}$$

This is the \prec -minimum path from i to j through a single state k . Thus, by induction, any finite product of these matrices encodes the \prec -minimum path. \square

Lemma (4.11). *With respect to a given constraint set S , the tuple $\mathcal{K}_S = (\mathfrak{K}(K), \sqcup_S, \times_S, 0_{\mathcal{K}}, 1_{\mathcal{K}})$ is a semiring.*

Proof. First we have to prove the monoidal structure. First consider $(K, \sqcup_S, 0_{\mathcal{K}})$. For $A, B, C \in \mathfrak{K}(K)$, we have

$$(A \sqcup_S B) \sqcup_S C = \mathfrak{K}(\mathfrak{K}(A \sqcup B) \sqcup C)$$

By definition, $\mathfrak{K}(A \sqcup B) \sqcup C \subseteq A \sqcup B \sqcup C$. Moreover, we have by simple subset relations that $\mathfrak{K}(A \sqcup B) \sqcup C \subseteq \mathfrak{K}(A \sqcup B \sqcup C)$. Therefore we have

$$(A \sqcup_S B) \sqcup_S C \subseteq \mathfrak{K}(A \sqcup B \sqcup C)$$

We also know by the lemma from chapter four that $\mathfrak{K}(A \sqcup B \sqcup C) = \mathfrak{K}(\mathfrak{K}(A \sqcup B \sqcup C))$. Therefore we have

$$\mathfrak{K}(A \sqcup B \sqcup C) = \mathfrak{K}(\mathfrak{K}(A \sqcup B \sqcup C)) \subseteq \mathfrak{K}(\mathfrak{K}(A \sqcup B) \sqcup C) \subseteq \mathfrak{K}(A \sqcup B \sqcup C) = \mathfrak{K}(\mathfrak{K}(A \sqcup B \sqcup C))$$

Thus $\mathfrak{K}(\mathfrak{K}(A \sqcup B) \sqcup C) = \mathfrak{K}(A \sqcup B \sqcup C)$. We can apply exactly the same argument to $\mathfrak{K}(A \sqcup \mathfrak{K}(B \sqcup C))$, therefore

$$\mathfrak{K}(\mathfrak{K}(A \sqcup B) \sqcup C) = \mathfrak{K}(A \sqcup B \sqcup C) = \mathfrak{K}(A \sqcup \mathfrak{K}(B \sqcup C))$$

Ergo \sqcup_S is associative. Additionally, we can compute:

$$\begin{aligned}
A \sqcup_S 0_K &= A \sqcup_S \{(\infty, \emptyset)\} \\
&= \mathfrak{K}(A \sqcup \{(\infty, \emptyset)\}) \\
&= \mathfrak{K}(\{(v, U(A, B)) \mid (v, S) \in A \cup \{(\infty, \emptyset)\}\}) \\
&= \mathfrak{K}(A \cup \{(\infty, B)\})
\end{aligned}$$

Here B is some arbitrary set of string fragments, but this does not matter because the infinity-weight profile is unique by definition of U and cannot be a contender. Therefore we have

$$\begin{aligned}
A \sqcup_S 0_K &= \mathfrak{K}(A \cup \{(\infty, B)\}) \\
&= A
\end{aligned}$$

Thus the first monoidal structure is proven.

Now consider $A, B, C \in \mathfrak{K}(K)$. We have

$$(A \times_S B) \times_S C = \mathfrak{K}\left(\bigsqcup\left(\mathfrak{K}\left(\bigsqcup(A \times B)\right) \times C\right)\right)$$

Here we want to make a similar argument as the above. We know that \times , as defined, is associative because multiset union and concatenation are associative, and we know that \sqcup is associative because it's a limited application of set unions. Additionally, we have that

$$\mathfrak{K}\left(\bigsqcup(A \times B)\right) \times C \subseteq \bigsqcup(A \times B) \times C$$

Therefore we have

$$\mathfrak{K}\left(\bigsqcup\left(\mathfrak{K}\left(\bigsqcup(A \times B)\right) \times C\right)\right) \subseteq \mathfrak{K}\left(\bigsqcup\left(\bigsqcup(A \times B) \times C\right)\right)$$

Because \times is associative and \sqcup is idempotent (because all it does is take unions of string sets when tuples have identical violation profiles), this is even simpler:

$$\mathfrak{K} \left(\sqcup \left(\mathfrak{K} \left(\sqcup (A \times B) \right) \times C \right) \right) \subseteq \mathfrak{K} \left(\sqcup (A \times B \times C) \right)$$

Additionally, we know by simple subset relations that

$$\sqcup \left(\mathfrak{K} \left(\sqcup (A \times B \times C) \right) \right) \subseteq \sqcup \left(\mathfrak{K} \left(\sqcup (A \times B) \right) \times C \right)$$

Finally, because \sqcup only merges string sets across tuples with identical violations and because \mathfrak{K} only picks according to violations, the two commute. Thus we have, by our lemma,

$$\sqcup \left(\mathfrak{K} \left(\sqcup (A \times B \times C) \right) \right) = \mathfrak{K} \left(\sqcup \left(\mathfrak{K} \left(\sqcup (A \times B \times C) \right) \right) \right)$$

Thus we can set up the same inclusion made in the last argument, namely the following:

$$\sqcup \left(\mathfrak{K} \left(\sqcup (A \times B \times C) \right) \right) \subseteq \sqcup \left(\mathfrak{K} \left(\sqcup (A \times B) \right) \times C \right)$$

And

$$\sqcup \left(\mathfrak{K} \left(\sqcup (A \times B) \right) \times C \right) \subseteq \mathfrak{K} \left(\sqcup \left(\mathfrak{K} \left(\sqcup (A \times B \times C) \right) \right) \right)$$

Where all terms are therefore necessarily equal. Then we see that

$$\begin{aligned} A \times_S 1_{\mathcal{K}} &= A \times_S \{(\emptyset, \{\})\} \\ &= \sqcup (\mathfrak{K} (A \times \{(\emptyset, \{\})\})) \\ &= \sqcup \mathfrak{K} (A) \\ &= A \end{aligned}$$

Thus the second monoidal structure is preserved.

Now we prove that \times_S distributes over \sqcup_S . For $A, B, C \in \mathfrak{K}(K)$, consider

$$\begin{aligned}
(A \sqcup_S B) \times_S C &= \mathfrak{K}(A \sqcup B) \times_S C \\
&= \mathfrak{K}\left(\bigsqcup(\mathfrak{K}(A \sqcup B) \times C)\right) \\
&= \mathfrak{K}\left(\bigsqcup(A \sqcup B) \times C\right) \\
&= \mathfrak{K}\left(\left(\bigsqcup(A \times C)\right) \sqcup \left(\bigsqcup(B \times C)\right)\right) \\
&= \mathfrak{K}\left(\mathfrak{K}\left(\bigsqcup(A \times C)\right) \sqcup \mathfrak{K}\left(\bigsqcup(B \times C)\right)\right) \\
&= \mathfrak{K}\left(\bigsqcup(A \times C)\right) \sqcup_S \mathfrak{K}\left(\bigsqcup(B \times C)\right) \\
&= (A \times_S C) \sqcup_S (B \times_S C)
\end{aligned}$$

The fourth step is justified because \sqcup removes duplicates and \times behaves like a Cartesian product and thus distributes over set union. The fifth step is justified because of the idempotence of \mathfrak{K} .

At last, the semiring criteria are satisfied.

□

References

- [1] Chomsky, Noam, and Morris Halle. The Sound Patterns of English. MIT Press, 1968.
- [2] Dijkstra, Edsger W. “A Note on Two Problems in Connexion with Graphs.” *Numerische mathematik* 1.1 (1959): 269-271.
- [3] Eisner, Jason. “Efficient Generation in Primitive Optimality Theory.” Proceedings of the eighth conference on European chapter of the Association for Computational Linguistics. Association for Computational Linguistics, 1997.
- [4] Ellison, Mark T. “Phonological Derivation in Optimality Theory.” University of Edinburgh, 1995.
- [5] Frank, Robert, and Giorgio Satta. “Optimality Theory and the Generative Complexity of Constraint Violability.” *Computational Linguistics* 24.2 (1998): 307-315.
- [6] Gerdemann, Dale, and Gertjan Van Noord. “Approximation and Exactness in Finite State Optimality Theory.” arXiv preprint cs/0006038 (2000).
- [7] Gold, E. Mark. “Language Identification in the Limit.” *Information and Control* 10.5 (1967): 447-474.
- [8] Goldsmith, John A. *Autosegmental Phonology*. Vol. 159. Bloomington: Indiana University Linguistics Club, 1976.
- [9] Kaplan, Ronald M., and Martin Kay. “Regular Models of Phonological Rule Systems.” *Computational Linguistics* 20.3 (1994): 331-378.
- [10] Karttunen, Lauri. “The Proper Treatment of Optimality in Computational Phonology: Plenary Talk.” Proceedings of the International Workshop on Finite State Methods in Natural Language Processing. Association for Computational Linguistics, 1998.
- [11] Karttunen, Lauri, Kimmo Koskeniemi, and Ronald Kaplan. “A Compiler for Two-level Phonological Rules.” *Tools for Morphological Analysis* (1987).

- [12] Koskenniemi, Kimmo. “Two-Level Model for Morphological Analysis.” IJCAI. Vol. 83. 1983.
- [13] McCarthy, John J. “Sympathy and Phonological Opacity.” Talk given at Hopkins Optimality Theory Workshop/Maryland Mayfest 1997, accessed in *Phonology* 16.3 (1999): 331-399.
- [14] Mohri, Mehryar. “Finite-state Transducers in Language and Speech Processing.” *Computational Linguistics* 23.2 (1997): 269-311.
- [15] Mohri, Mehryar, Fernando Pereira, and Michael Riley. “Weighted Finite-State Transducers in Speech Recognition.” *Computer Speech & Language* 16.1 (2002): 69-88.
- [16] Mpiranya, Fidèle. *Kinyarwanda Language Companion*. University of Chicago, forthcoming.
- [17] Prince, Alan, and Paul Smolensky. *Optimality Theory: Constraint Interaction in Generative Grammar*. Rutgers University Center for Cognitive Science Technical Report 2, 1993.
- [18] Riggle, Jason. “Generation, Recognition, and Learning in Finite State Optimality Theory.” Diss. University of California, Los Angeles, 2004.
- [19] Riggle, Jason. “Efficiently Generating Contenders.” University of Chicago, 2011.
- [20] Samek-Lodovici, Vieri, and Alan Prince. “Fundamental Properties of Harmonic Bounding” Technical Reports of the Rutgers Center for Cognitive Science, 2002 .
- [21] Tesar, Bruce, and Paul Smolensky. “Learnability in Optimality Theory.” Baltimore: Department of Cognitive Science, Johns Hopkins University (1996).