The Proper Treatment of Optimality in Computational Phonology

Lauri Karttunen

Xerox Research Centre Europe
6, chemin de Maupertuis
38240 Meylan, France

Abstract. This paper presents a novel formalization of optimality theory. Unlike previous treatments of optimality in computational linguistics, starting with Ellison (1994), the new approach does not require any explicit marking and counting of constraint violations. It is based on the notion of "lenient composition", defined as the combination of *ordinary composition* and *priority union*. If an underlying form has outputs that can meet a given constraint, lenient composition enforces the constraint; if none of the output candidates meet the constraint, lenient composition allows all of them. For the sake of greater efficiency, we may "leniently compose" the GEN relation and all the constraints into a single finite-state transducer that maps each underlying form directly into its optimal surface realizations, and vice versa. Seen from this perspective, optimality theory is surprisingly similar to the two older strains of finite-state phonology: classical rewrite systems and two-level models. In particular, the ranking of optimality constraints corresponds to the ordering of rewrite rules.

1 Introduction

It has been recognized for some time that Optimality Theory (OT), introduced by Prince and Smolensky [24], is from a computational point of view closely related to classical phonological rewrite systems (Chomsky and Halle [1]) and to two-level descriptions (Koskenniemi [21]).

Ellison [6] observes that the GEN function of OT can be regarded as a regular relation and that OT constraints seem to be regular. Thus each constraint can be modeled as a transducer that maps a string to a sequence of marks indicating the presence or absence of a violation. The most optimal solution can then be found by sorting and comparing the marks. Frank and Satta [7] give a formal proof that OT models can be construed as regular relations provided that the number of violations is bounded. Eisner [3, 4, 5] develops a typology of OT constraints that corresponds to two types of rules in two-level descriptions: restrictions and prohibitions.

The practice of marking and counting constraint violations is closely related to the tableau method introduced in Prince and Smolensky for selecting the most optimal output candidate. Much of the current work in optimality theory consists of constructing tableaux that demonstrate the need for particular constraints and rankings that allow the favored candidate to emerge with the best score.

From a computational viewpoint, this evaluation method is suboptimal. Although the work of GEN and the assignment of violation marks can be carried out by finite-state transducers, the sorting and counting of the marks envisioned by Ellison and subsequent work (Walther [26]) is an off-line activity that is not a finite-state process. This kind of optimality computation cannot be straight-forwardly integrated with other types of linguistic processing (morphological analysis, text-to-speech generation etc.) that are commonly performed by means of finite-state transduction.

This paper demonstrates that the computation of the most optimal surface realizations of any input string can be carried out entirely within a finite-state calculus, subject to

the limitation (Frank and Satta [7]) that the maximal number of violations that need to be considered is bounded. We will show that optimality constraints can be treated computationally in a similar manner as two-level constraints and rewrite rules. For example, optimality constraints can be merged with one another, respecting their ranking, just as it is possible to merge rewrite rules and two-level constraints. A system of optimality constraints can be imposed on a finite-state lexicon creating a transducer that maps each member of a possibly infinite set of lexical forms into its most optimal surface realization, and vice versa.

For the sake of conciseness, we limit the discussion to optimality theory as originally presented in Prince and Smolensky [24]. The techniques described below can also be applied to the correspondence version of the theory (McCarthy and Prince [22]) that expands the model to encompass output/output constraints between reduplicant and base forms.

To set the stage for discussing the application and merging of optimality constraints it is useful to first look at the corresponding operations in the context of rewrite rules and two-level constraints. Thus we can see both the similarities and the differences between the three approaches.

2 Background: rewrite rules and two-level constraints

As is well-known, phonological rewrite rules and two-level constraints can be implemented as finite-state transducers (Johnson [9], Karttunen, Koskenniemi and Kaplan [14], Kaplan and Kay [10]).

The application of a system of rewrite rules to an input string can be modeled as a cascade of transductions, that is, a sequence of compositions that yields a relation mapping the input string to one or more surface realizations. The application of a set of two-level constraints is a combination of intersection and composition (Karttunen [18]).

To illustrate the idea of rule application as composition, let us take a concrete example, the well-known vowel alternations in Yokuts (Kisseberth [20], Cole and Kisseberth [2], McCarthy [23]). Yokuts vowels are subject to three types of alternations:

- Underspecified suffix vowels are rounded in the presence of a stem vowel of the same height: *dub+hIn* → *dubhun*, *bok'+Al* → *bok'ol*.
- Long high vowels are lowered: *?u:t+It* → *?o:tut*, *mi:k+It* → *me:kit*.
- Vowels are shortened in closed syllables: *sa:p* → *sap*, *go:b+hIn* → *gobhin*.

Because of examples such as *?u:t+hIn* → *?othun*, the rules must be applied in the given order. Rounding must precede lowering because the suffix vowel in *?u:t+hIn* emerges as *u*. Shortening must follow lowering because the stem vowel in *?u:t+hIn* would otherwise remain high giving *?uthun* rather than *?othun* as the final output.

These three rewrite rules can be formalized straight-forwardly as regular replace expressions (Karttunen [19]) and compiled into finite-state transducers. The derivation *?u:t+hIn* → *?othun* can thus be modeled as a cascade of three compositions that yield a transducer that relates the input directly to the final output.

The first step, the composition of the initial network (an identity transducer on the string *?u:t+hIn*) with the rounding transducer, produces the network that maps between *?u:t+hIn* and *?u:t+hun*. The symbol `.o.` in Figure 1 denotes the composition operation. It is important to realize that the result of each rule application in Figure 1 is not an output string but a relation. The first application produces a mapping from *?u:t+hIn* to *?u:t+hun*. In essence, it is the original Rounding transducer restricted to the specific input. The resulting network represents a relation between two languages (= sets of strings). In this case both languages contain just one string but if the Rounding rule were optional, the output language would contain two strings: one with, the other without rounding.

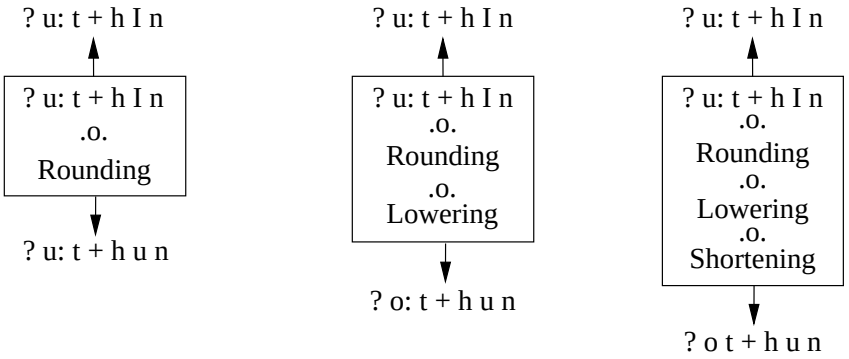

Figure 1. Cascade of rewrite rule applications.

At the next step in Figure 1, the intermediate output created by the Rounding transducer is eliminated as a result of the composition with the Lowering transducer. The final stage is a transducer that maps directly between the input string and its surface realization without any intermediate stages.

We could achieve this same result in a different way: by first composing the three rules to produce a transducer that maps any lexical form directly to its Yokuts surface realization (Figure 2) and then applying the resulting single transducer to the particular input.

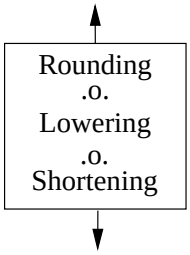

Figure 2. Yokuts vowel alternations.

The small network (21 states) pictured in Figure 2 merges the three rules and thus represents the complexity of Yokuts vowel alternations without any "serialism", that is, without any intermediate representations.

In the context of the two-level model, the Yokuts vowel alternations can be described quite simply. The two-level version of the rounding rule controls rounding by the lexical context. It ignores the surface realization of the trigger, the underlyingly high stem vowel. The joint effect of the lowering and shortening constraints is that a lexical *u:* in *?u:t+hIn* is realized as *o*. Thus a two-level description of the Yokuts alternations consists of three rule transducers operating in parallel (Figure 3).

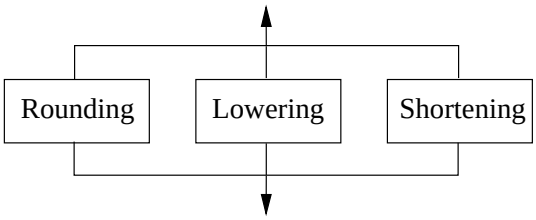

Figure 3. Parallel two-level constraints.

The application of a two-level system to an input can be formalized as *intersecting composition* (Karttunen [18]). It involves constructing a partial intersection of the constraint networks and composing it with the input. We can of course carry out the intersection of the rules independently of any particular input. This merging operation results in the very same 21-state transducer as the composition of the corresponding rewrite rules pictured in Figure 2.

Thus the two descriptions of Yokuts sketched above are completely equivalent in the sense that they yield the same mapping between lexical and surface forms. They decompose the same complex vowel alternation relation in different ways into a set of simpler relations that are easily understood and manipulated.[1] As we will see shortly, optimality theory can be characterized as yet another way of achieving this kind of decomposition.

The fundamental computational operation for rewrite rules is composition, as it is involved both in the application of rules to strings and in merging the rules themselves. For two-level rules, the corresponding operations are intersecting composition and intersection.

Turning now to optimality theory, our main interest will be in finding what the corresponding computations are in this new paradigm. What does applying a constraint mean in the context of optimality theory? Can optimality constraints be merged while taking into account their ranking?

3 Optimality theory

Optimality theory (Prince and Smolensky [24]) abandons rewrite rules. Rules are replaced by two new concepts: (1) a universal function called GEN and (2) a set of ranked universal constraints. GEN provides each input form with a (possibly infinite) set of output candidates. The constraints eliminate all but the best output candidate. Because many constraints are in conflict, it may be impossible for any candidate to satisfy all of them. The winner is determined by taking into consideration the language-specific ranking of the constraints. The winning candidate is the one with the least serious violations.

In order to explore the computational aspects of the theory it is useful to focus on a concrete example, even simpler than the Yokuts vowel alternation we just discussed.[2] We will take the familiar case of syllabification constraints discussed by Prince and Smolensky [24] and many subsequent authors (Ellison [6], Tesar [25], Hammond [8]).

3.1 GEN for syllabification

We assume that the input to GEN consists of strings of vowels `V` an consonants `C`. GEN allows each segment to play a role in the syllable or to remain "unparsed". A syllable contains at least a nucleus and possibly an onset and a coda.

Let us assume that GEN marks these roles by inserting labeled brackets around each input element. An input consonant such as *b* will have three outputs `O[b]` (onset), `D[b]` (coda), and `X[b]` (unparsed). Each vowel such as *a* will have two outputs, `N[a]` (nucleus) and `X[a]` (unparsed), In addition, GEN "overparses", that is, it freely inserts empty onset `O[ ]`, nucleus `N[ ]`, and coda `D[ ]` brackets.

For the sake of concreteness, we give here an explicit definition of GEN using the notation of Xerox regular expression calculus (Karttunen *et al* [15]). We define GEN as the composition of four simple components, `Input`, `Parse`, `OverParse`, and `SyllableStructure`. The definitions of the first three components are shown in Figure 4.

[1] For more discussion of these issues, see Karttunen [17].

[2] The Yokuts case is problematic for the OT theory (Cole and Kisseberth [2], McCarthy [23]) because rounding depends on the height of the stem vowel in the underlying representation. Cole and Kisseberth offer a baroque version of the two-level solution. McCarthy strives mightily to distinguish his "sympathy" candidates from the intermediate representations postulated by the rewrite approach.

```
define Input     [C | V]*                          ;

define Parse     C -> ["O[" | "D[" | "X["] ... "]"
                            .o.
                 V -> ["N[" | "X["] ... "]"        ;

define OverParse [. .] (->) ["O["|"N["|"D["] "]"   ;
```

Figure 4. Input, Parse, and OverParse

A replace expression of the type `A -> B ... C` in the Xerox calculus denotes a relation that wraps the prefix strings in `B` and the suffix strings in `C` around every string in `A`. Thus `Parse` is a transducer that inserts appropriate bracket pairs around input segments. Consonants can be onsets, codas, or be ignored. Vowels can be nuclei or be ignored. `OverParse` inserts optionally unfilled onsets, codas, and nuclei. The dotted brackets `[. .]` specify that only a single instance of a given bracket pair is inserted at any position. The role of the third GEN component, `SyllableStructure`, is to constrain the output of `Parse` and `OverParse`. A syllable needs a nucleus, onsets and codas are optional; they must be in the right order; unparsed elements may occur freely. For the sake of clarity, we define `SyllableStructure` with the help of four auxiliary terms (Figure 5).

```
define Onset             "O[" (C) "]"   ;
define Nucleus           "N[" (V) "]"   ;
define Coda              "D[" (C) "]"   ;
define Unparsed          "X[" [C|V] "]" ;

define SyllableStructure [[(Onset) Nucleus (Coda)]/Unparsed]* ;
```

Figure 5. SyllableStructure

Round parentheses in the Xerox regular expression notation indicate optionality. Thus `(C)` in the definition of `Onset` indicates that onsets may be empty or filled with a consonant. Similarly, `(Onset)` in the definition of `SyllableStructure` means that a syllable may have or not have an onset. The effect of the `/` operator is to allow unparsed consonants and vowels to occur freely within a syllable. The disjunction `[C|V]` in the definition of `Unparsed` allows consonants and vowels to remain unparsed.
With these preliminaries we can now define GEN as a simple composition of the four components (Figure 6).

```
define GEN        Input
                    .o.
                OverParse
                    .o.
                  Parse
                    .o.
            SyllableStructure ;
```

Figure 6. GEN for syllabification

With the appropriate definitions for `C` (consonants) and `V` (vowels), the expression in Figure 6 yields a transducer with 22 states and 229 arcs.
It is not necessary to include `Input` in the definition of GEN but it has technically a beneficial effect. The constraints have less work to do when it is made explicit that the auxiliary bracket alphabet is not included in the input.

Because GEN over- and underparses with wild abandon, it produces a large number of output candidates even for very short inputs. For example, the string *a* composed with GEN yields a relation with 14 strings on the output side (Figure 7).

```
   N[a]
   N[a]N[]
   N[a]D[]
N[]N[a]
N[]N[a]N[]
N[]N[a]D[]
N[]X[a]
N[]X[a]N[]
N[]X[a]D[]
O[]N[a]
O[]N[a]N[]
O[]N[a]D[]
O[]X[a]N[]
   X[a]N[]
```

Figure 7. GEN applied to *a*

The number of output candidates for *abracadabra* is nearly 1.7 million, although the network representing the mapping has only 193 states. It is evident that working with finite-state tools has a significant advantage over manual tableau methods.

3.2 Syllabification constraints

The syllabification constraints of Prince and Smolensky [24] can easily be expressed as regular expressions in the Xerox calculus. Figure 8 lists the five constraints with their translations.

Syllables must have onsets.	`define HaveOns N[" => "O[" (C) "]" _ ;`
Syllables must not have codas.	`define NoCoda  ~$"D[" ;`
Input segments must be parsed.	`define Parse  ~$"X[" ;`
A nucleus position must be filled.	`define FillNuc ~$"N[" "]" ;`
An onset position must be filled.	`define FillOns ~$"O[" "]" ;`

Figure 8. Syllabification constraints

The definition of the `HaveOns` constraint uses the *restriction* operator `=>`. It requires that any occurrence of the nucleus bracket, `[N`, must be immediately preceded by a filled `O[C]` or unfilled `O[ ]` onset. The definitions of the other four constraints are composed of the negation `~` and the contains operator `$`. For example, the `NoCoda` constraint, `~$"D["`, can be read as "does not contain `D[`". The `FillNuc` and `FillOns` constraints forbid empty nucleus `N[ ]` and onset `O[ ]` brackets.

These constraints compile into very small networks, the largest one, `HaveOns`, contains four states. Each constraint network encodes an infinite regular language. For example, the `HaveOns` language includes all strings of any length that contain no instances of `N[` at all and all strings of any length in which every instance of `N[` is immediately preceded by an onset.

The identity relations on these constraint languages can be thought of as filters. For example, the identity relation on `HaveOns` maps all `HaveOns` strings into themselves and blocks on all other strings. In the following section, we will in fact consistently treat the constraint networks as representing identity relations.

3.3 Constraint application

Having defined GEN and the five syllabification constraints we are now in a position to address the main issue: *how are optimality constraints applied?*
Given that GEN denotes a relation and that the constraints can be thought of as identity relations on sets, the simplest idea is to proceed in the same way as with rewrite rules in Figure 2. We could compose GEN with the constraints to yield a transducer that maps each input to its most optimal realization letting the ordering of the constraints in the cascade to implement their ranking (Figure 9).

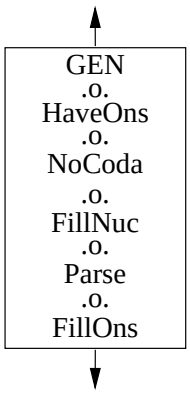

Figure 9. Merciless cascade.

But it is immediately obvious that composition does not work here as intended. The 6-state transducer illustrated in Figure 9 works fine on inputs such as *panama* yielding `O[p]N[a]O[n]N[a]O[m]N[a]` but it fails to produce any output on inputs like *america* that fail on some constraint. Only strings that have a perfect output candidate survive this merciless cascade. We need to replace composition with some new operation to make this schema work correctly.

4 Lenient composition

The necessary operation, let us call it *lenient composition*, is not difficult to construct but to our knowledge it has not been previously defined. Frank and Satta [7] come very close but do not take the final step to encapsulate the notion. Hammond [8] has the idea but lacks the means to spell it out in formal terms.
As the first step towards defining lenient composition, let us review an old notion called *priority union* (Kaplan [12]). This term was originally defined as an operation for unifying two feature structures in a way that eliminates any risk of failure by stipulating that one of the two has priority in case of a conflict.[3] A finite-state version of this notion has proved very useful in the management of transducer lexicons (Kaplan and Newman [11]).
Let us consider the relations `Q` and `R` depicted in Figure 10. The `Q` relation maps a to x and b to y. The `R` relation maps b to z and c to w. The priority union of `Q` and `R`, denoted

[3] The DPATR system at SRI (Karttunen [16]) had the same operation with a less respectable title. It was called "clobber".

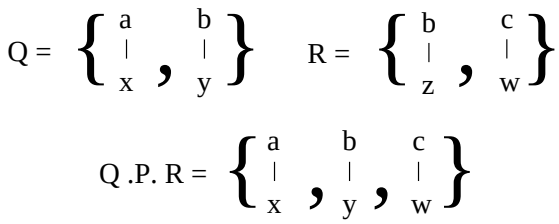

Figure 10. Example of priority union.

`Q .P. R`, maps *a* to *x*, *b* to *y*, and *c* to *w*. That is, it includes all the pairs from `Q` and every pair from R that has as its upper element a string that does not occur as the upper string of any pair in `Q`. If some string occurs as the upper element of some pair in both `Q` and `R`, the priority union of `Q` and `R` only includes the pair in `Q`. Consequently `Q .P. R` in Figure 10 maps *b* to *y* instead of *z*.

The priority union operator `.P.` can be defined in terms of other regular expression operators in the Xerox calculus. A straight-forward definition is given in Figure 11.

```
Q .P. R = Q | [~[Q.u] .o. R]
```

Figure 11. Definition of priority union

The `.u` operator in Figure 11 extracts the "upper" language from a regular relation. Thus the expression `~[Q.u]` denotes the set of strings that do not occur on the upper side of the `Q` relation. The effect of the composition in Figure 11 is to restrict R to mappings that concern strings that are not mapped to anything in `Q`. Only this subset of R is unioned with `Q`.

We define the desired operation, lenient composition, denoted `.O.`, as a combination of ordinary composition and priority union (Figure 12).

```
R .O. C  = [R .o. C] .P. R
```

Figure 12. Definition of lenient composition

To better visualize the effect of the operation defined in Figure 12 one may think of the relation `R` as a set of mappings induced by GEN and the relation `C` as one of the constraints defined in Figure 8. The left side of the definition, `[R .o. C]` restricts R to mappings that satisfy the constraint. That is, any pair whose lower side string is not in `C` will be eliminated. If some string in the upper language of `R` has no counterpart on the lower side that meets the constraint, then it is not present in `[R .o. C].u` but, for that very reason, it will be "rescued" by the priority union. In other words, if an underlying form has some output that can meet the given constraint, lenient composition enforces the constraint. If none of the output candidates meet the constraint, lenient composition allows all of them. The definition of lenient composition entails that the upper language of `R` is preserved in `R .O. C`.

Many people, including Hammond [8] and Frank and Satta [7], have independently had a similar idea without conceiving it as a finite-state operation.[4] If one already knows about priority union, lenient composition is an obvious idea.

Let us illustrate the effect of lenient composition starting with the example in Figure 7 The composition of the input *a* with GEN yields a relation that maps *a* to the 14 outputs in Figure 7. We will leniently compose this relation with each of the constraints in the order of their ranking, starting with the `HaveOns` constraint (Figure 13). The lower-case operator `.o.` stands for ordinary composition, the upper case `.O.` for lenient composition.

[4] Hammond implements a pruning operation that removes output candidates under the condition that "pruning cannot reduce the candidate set to null" (p 13). Frank and Satta (p. 7) describe a process of "conditional intersection" that enforces a constraint if it can be met and does nothing otherwise.

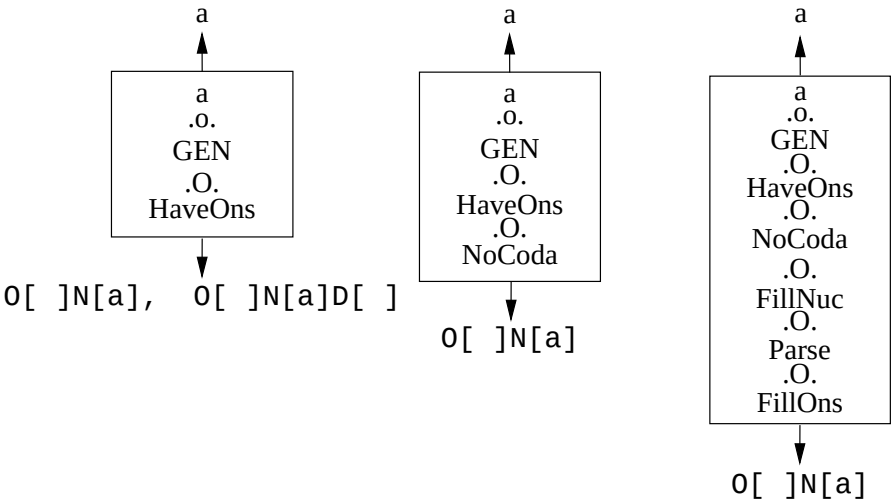

Figure 13. Cascade of constraint applications.

As Figure 13 illustrates, applying `HaveOns` by lenient composition removes most of the 14 output candidates produced by GEN. The resulting relation maps *a* to two outputs `O[ ]N[a]` and `O[ ]N[a]D[ ]`. The next highest-ranking constraint, `NoCoda`, removes the latter alternative. The twelve candidates that were eliminated by the first lenient composition are no longer under consideration.

The next two constraints in the sequence, `FillNuc` and `Parse`, obviously do not change the relation because the one remaining output candidate, `O[ ]N[a]`, satisfies them. Up to this point, the distinction between lenient and ordinary composition does not make any difference because we have not exhausted the set of output candidates. However, when we bring in the last constraint, `FillOns`, the right half of the definition in Figure 12 has to come to the rescue; otherwise the result would be the null relation, that is, no output for *a*.

This example demonstrates that the application of optimality constraints can be thought of as a cascade of lenient compositions that carry down an ever decreasing number of output candidates without allowing the set to become empty. Instead of *intermediate representations* (c.f. Figure 1) there are *intermediate candidate populations* corresponding to the columns in the left-to-right ordering of the constraint tableau.

Instead of applying the constraints one by one to the output provided by GEN for a particular input, we may also leniently compose the GEN relation itself with the constraints. Thus the suggestion made in Figure 9 is (nearly) correct after all, provided that we replace ordinary composition with lenient composition (Figure 14).

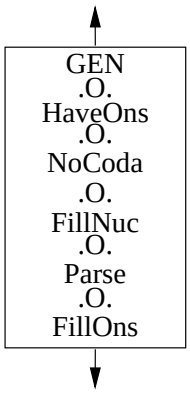

Figure 14. Lenient cascade

The composite single transducer shown in Figure 14 maps *a* and any other input directly into its viable outputs without ever producing any failing candidates.

5 Multiple violations

However, we have not yet addressed one very important issue. It is not sufficient to obey the ranking of the constraints. If two or more output candidates violate the same constraint multiple times we should prefer the candidate or candidates with the smallest number of violations. This does not come for free. The system that we have sketched so far does not make that distinction. If the input form has no perfect outputs, we may get a set of outputs that differ with respect to the number of constraint violations. For example, the transducer in Figure 14 gives three outputs for the string *bebop* (Figure 15).

```
O[b]N[e]X[b]X[o]X[p]
O[b]N[e]O[b]N[o]X[p]
X[b]X[e]O[b]N[o]X[p]
```

Figure 15. Two many outputs

Because *bebop* has no output that meets the `Parse` constraint, lenient composition allows all outputs that contain a `Parse` violation regardless of the number of violations. Here the second alternative with just one violation should win but it does not.
Instead of viewing `Parse` as a single constraint, we need to reconstruct it as a series of ever more relaxed parse constraints. The `^>` operator in Figure 16 means "more than".

```
define Parse    ~$["X["]     ;
define Parse1  ~[[$"X["]^>1] ;
define Parse2  ~[[$"X["]^>2] ;
        ...
define ParseN  ~[[$"X["]^>N] ;
```

Figure 16. A family of `Parse` constraints

Our original `Parse` constraint is violated by a single unparsed element. `Parse1` allows one unparsed element. `Parse2` allows up to two violations, and `ParseN` up to *N* violations.
The single `Parse` line in Figure 14 must be replaced by the sequence of lenient compositions in Figure 17 up to some chosen *N*.

```
Parse
  .O.
Parse1
  .O.
Parse2
  .O.
ParseN
```

Figure 17. Gradient `Parse` constraint

If an input string has at least one output form that meets the `Parse` constraint (no violations), all the competing output forms with `Parse` violations are eliminated. Failing that, if the input string has at least one output form with just one violation, all the outputs with more violations are eliminated. And so on.
The particular order in which the individual parse constraints apply actually has no effect here on the final outcome because the constraint languages are in a strict subset relation:

`Parse` ⊂ `Parse1` ⊂ `Parse2` ⊂ ... `ParseN`.[5] For example, if the best candidate incurs two violations, it is in `Parse2` and in all the weaker constraints. The ranking in Figure 17 determines only the order in which the losing candidates are eliminated. If we start with the strictest constraint, all the losers are eliminated at once when `Parse2` is applied; if we start with a weaker constraint, some output candidates will be eliminated earlier than others but the winner remains the same.

As the number of constraints goes up, so does the size of the combined constraint network in Figure 14, from 66 states (no `Parse` violations) to 248 (at most five violations). It maps *bebop* to `O[b]N[e]O[b]N[o]X[p]` and *abracadabra* to `O[]N[a]X[b]O[r]N[a]O[c]N[a]-O[d]N[a]X[b]O[r]N[a]` correctly and instantaneously.

It is immediately evident that while we can construct a cascade of constraints that prefer *n* violations to *n+1* violations up to any given *n*, there is no way in a finite-state system to express the general idea that fewer violations is better than more violations. As Frank and Satta [7] point out, finite-state constraints cannot make infinitely many distinctions of well-formedness. It is not likely that this limitation is a serious obstacle to practical optimality computations with finite-state systems as the number of constraint violations that need to be taken into account is generally small.

It is curious that violation counting should emerge as the crucial issue that potentially pushes optimality theory out of the finite-state domain thus making it formally more powerful than rewrite systems and two-level models. It has never been presented as an argument against the older models that they do not allow unlimited counting. Is the additional power an asset or an embarrassement?

6 Conclusion

This novel formalization of optimality theory has several technical advantages over the previous computational treatments:

- No marking, sorting, or counting of constraint violations.
- Application of optimality constraints is done within the finite-state calculus.
- A system of optimality constraints can be merged into a single constraint network.

This approach shows clearly that optimality theory is very similar to the two older strains of finite-state phonology: classical rewrite systems and two-level models. In optimality theory, lenient composition plays the same role as ordinary composition in rewrite systems. The top-down serialism of rule ordering is replaced by the left-to-right serialism of the constraint tableau.

The new lenient composition operator has other uses beyond phonology. In the area of syntax, Constraint Grammar (Karlsson *et al.* [13]) is from a formal point of view very similar to optimality theory. Although constraint grammars so far have not been implemented as pure finite-state systems it is evident that the lenient composition operator makes it possible.

References

1. Noam Chomsky and Morris Halle. 1968. *The Sound Pattern of English.* Harper and Row, New York.
2. Jennifer S. Cole and Charles W. Kisseberth. 1995. Restricting multi-level constraint evaluation: Opaque rule interaction in Yawelmani vowel harmony. (ROA-98-0000).
3. Jason Eisner. 1997a. Decomposing FootForm: Primitive constraints in OT. In *SCIL VIII.* (ROA-205-0797).

[5] Thanks to Jason Eisner (p.c.) for this observation.

4. Jason Eisner. 1997b. Efficient generation in primitive optimality theory. In *ACL'97*, Madrid, Spain. (ROA-206-0797).
5. Jason Eisner. 1997c. What constraints should OT allow? Handout (20p) for talk at the LSA Annual Meeting, Chicago, 1/4/97. (ROA-204-0797).
6. Mark T. Ellison. 1994. Phonological derivation in optimality theory. In *COLING'94 Vol II*, pages 1007–1013, Kyoto, Japan. (ROA-75-0000), (cmp-lg/9505031).
7. Robert Frank and Giorgio Satta. 1998. Optimality theory and the generative complexity of constraint violability. *Computational Linguistics (forthcoming)*. (ROA-228-1197).
8. Michael Hammond. 1997. Parsing syllables: Modeling OT computationally. (ROA-222-1097).
9. C. Douglas Johnson. 1972. *Formal Aspects of Phonological Description*. Mouton, The Hague.
10. Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.
11. Ronald M. Kaplan and Paula S. Newman. 1997. Lexical resource reconciliation in the Xerox Linguistic Environment. In *ACL/EACL'98 Workshop on Computational Environments for Grammar Development and Linguistic Engineering*, Madrid, Spain.
12. Ronald M. Kaplan. 1987. Three seductions of computational psycholinguistics. In P. Whitelock, M. M. Wood, H. L. Somers, R. Johnson, and P. Bennett, editors, *Linguistic Theory and Computer Applications*. Academic Press, New York. reprinted in *Formal Issues in Lexical-Functional Grammar*. University of Chicago Press, 1996.
13. Fred Karlsson, Atro Voutilainen, Juha Heikkilä, and Arto Anttila. 1995. *Constraint Grammar: A Language-Independent Framework for Parsing Unrestricted Text*. Mouton de Gruyter, Berlin/New York.
14. Lauri Karttunen, Kimmo Koskenniemi, and Ronald M. Kaplan. 1987. A compiler for two-level phonological rules. Technical report, Center for the Study of Language and Information, Stanford University, June 25.
15. Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Journal of Natural Language Engineering*, 2(4):305–328.
16. Lauri Karttunen. 1986. D-PATR: A development environment for unification-based grammars. In *COLING'86*, pages 74–80.
17. Lauri Karttunen. 1993. Finite-state constraints. In John Goldsmith, editor, *The Last Phonological Rule*, pages 173–194. Chicago University Press, Chicago.
18. Lauri Karttunen. 1994. Constructing lexical transducers. In *COLING'94*.
19. Lauri Karttunen. 1995. The replace operator. In *Proceedings of the 33rd Annual Meeting of the ACL*, Cambridge, MA. (cmp-lg/9504032).
20. Charles Kisseberth. 1969. On the abstractness of phonology. *Papers in Linguistics*, 1:248–282.
21. Kimmo Koskenniemi. 1983. Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.
22. John McCarthy and Alan Prince. 1998. Faithfulness and identity in prosodic morphology. In R. Kager, H. vand der Hulst, and W. Zonneveld, editors, *The prosody-morphology interface*. Cambridge University Press, Cambridge, UK. (ROA-216-0997).
23. John J. McCarthy. 1998. Sympathy & phonological opacity. (ROA-252-0398).
24. Alan Prince and Paul Smolensky. 1993. Optimality Theory: Constraint Interaction in Generative Grammar. Technical Report TR-2, Rutgers University Cognitive Science Center, New Brunswick, NJ. To appear, MIT Press.
25. Bruce Tesar. 1995. *Computational Optimality Theory*. Ph.D. thesis, University of Colorado, Boulder, CO.
26. Markus Walther. 1996. OT SIMPLE - A construction-kit approach to Optimality Theory implementation. (ROA-152-1096).