Approximation and Exactness in Finite State Optimality Theory

Dale Gerdemann
University of Tübingen
dg@sfs.nphil.uni-tuebingen.de

Gertjan van Noord
University of Groningen
vannoord@let.rug.nl

Abstract

Previous work (Frank and Satta, 1998; Karttunen, 1998) has shown that Optimality Theory with *gradient* constraints generally is not finite state. A new finite-state treatment of gradient constraints is presented which improves upon the approximation of Karttunen (1998). The method turns out to be exact, and very compact, for the syllabification analysis of Prince and Smolensky (1993).

1 Introduction

Finite state methods have proven quite successful for encoding rule-based generative phonology (Johnson, 1972; Kaplan and Kay, 1994). Recently, however, Optimality Theory (Prince and Smolensky, 1993) has emphasized phonological accounts with default constraints on surface forms. While Optimality Theory (OT) has been successful in explaining certain phonological phenomena such as *conspiracies* (Kisseberth, 1970), it has been less successful for computation. The negative result of Frank and Satta (1998) has shown that in the general case the method of counting constraint violations takes OT beyond the power of regular relations. To handle such constraints, Karttunen (1998) has proposed a finite-state approximation that counts constraint violations up to a predetermined bound. Unlike previous approaches (Ellison, 1994; Walther, 1996), Karttunen's approach is encoded entirely in the finite state calculus, with no extra-logical procedures for counting constraint violations.

In this paper, we will present a new approach that seeks to minimize constraint violations without counting. Rather than counting, our approach employs a filter based on matching constraint violations against violations in alternatively derivable strings. As in Karttunen's counting approach, our approach uses purely finite state methods without extra-logical procedures. We show that our *matching* approach is superior to the *counting* approach for both size of resulting automata and closeness of approximation. The matching approach can in fact exactly model many OT analyses where the counting approach yields only an approximation; yet, the size of the resulting automaton is typically much smaller.

In this paper we will illustrate the matching approach and compare it with the counting approach on the basis of the Prince & Smolensky syllable structure example (Prince and Smolensky, 1993; Ellison, 1994; Tesar, 1995), for each of the different constraint orderings identified in Prince & Smolensky.

2 Finite State Phonology

2.1 Finite State Calculus

Finite state approaches have proven to be very successful for efficient encoding of phonological rules. In particular, the work of Kaplan and Kay (1994) has provided a compiler from classical generative phonology rewriting rules to finite state transducers. This work has clearly shown how apparently procedural rules can be recast in a declarative, reversible framework.

In the process of developing their rule compiler, Kaplan & Kay also developed a high-level finite state calculus. They argue convincingly that this calculus provides an appropriate high-level approach for expressing regular languages and relations. The alternative conception in term of states and transitions can become unwieldy for all but the simplest cases.[1]

[1] Although in some cases such a direct implementation can be much more efficient (Mohri and Sproat, 1996; van Noord and Gerdemann, 1999).

`[]`	empty string
`[E1,E2,...,En]`	concatenation of `E1...En`
`{}`	empty language
`{E1,E2,...,En}`	union of `E1...En`
`(E)`	grouping for op. precedence
`E*`	Kleene closure
`E+`	Kleene plus
`E^`	optionality
`E1 - E2`	difference
`~E`	complement
`$ E`	containment
`E1 & E2`	intersection
`?`	any symbol
`E1 x E2`	cross-product
`A o B`	composition
`domain(E)`	domain of a transduction
`range(E)`	range of a transduction
`identity(E)`	identity transduction[2]
`inverse(E)`	inverse transduction

Table 1: Regular expression operators.

Kaplan & Kay's finite state calculus now exists in multiple implementations, the most well-known of which is that of Karttunen et al. (1996). In this paper, however, we will use the alternative implementation provided by the FSA Utilities (van Noord, 1997; van Noord, 1999; van Noord and Gerdemann, 1999). The FSA Utilities allows the programmer to introduce new regular expression operators of arbitrary complexity. This higher-level interface allows us to express our algorithm more easily. The syntax of the FSA Utilities calculus is summarized in Table 1.

The finite state calculus has proven to be a very useful tool for the development of higher-level finite state operators (Karttunen, 1995; Kempe and Karttunen, 1996; Karttunen, 1996; Gerdemann and van Noord, 1999). An interesting feature of most such operators is that they are implemented using a generate-and-test paradigm. Karttunen (1996), for example, introduces an algorithm for a leftmost-longest replacement operator. Somewhat simplified, we may view this algorithm as having two steps. First, the generator freely marks up possible replacement sites. Then the tester, which is an identity transducer, filters out those cases not conforming to the leftmost-longest strategy.

[2]If an expression for a recognizer occurs in a context where a transducer is required, the identity operation will be used implicitly for coercion.

Since the generator and tester are both implemented as transducers, they can be composed into a single transducer, which eliminates the inefficiency normally associated with generate-and-test algorithms.

2.2 Finite State Optimality Theory

The generate-and-test paradigm initially appears to be appropriate for optimality theory. If, as claimed in Ellison (1994), *Gen* is a regular relation and if each constraint can be implemented as an identity transducer, then optimality theory analyses could be implemented as in fig. 1.

```
     Gen
      o
 Constraint1
      o
     ...
      o
 ConstraintN
```

Figure 1: Optimality Theory as Generate and Test

The problem with this simple approach is that in OT, a constraint is allowed to be violated if none of the candidates satisfy that constraint. Karttunen (1998) treats this problem by providing a new operator for *lenient* composition, which is defined in terms of the auxiliary operation of priority union. In the FSA Utilities calculus, these operations can be defined as:[3]

```
macro(priority_union(Q,R),
      {Q, ~domain(Q) o R}).
macro(lenient_composition(S,C),
      priority_union(S o C, S)).
```

The effect here is that the lenient composition of `S` and `C` is the composition of `S` and `C`, except for those elements in the domain of `S` that are not mapped to anything by `S o C`. For these elements not in the domain of `S o C`, the effect is the same as the effect of `S` alone. We use the

[3]The notation `macro(Expr1,Expr2)` is used to indicate that the regular expression `Expr1` is an abbreviation for the expression `Expr2`. Because Prolog variables are allowed in both expressions this turns out to be an intuitive and powerful notation (van Noord and Gerdemann, 1999).

notation `S lc C` as a succinct notation for the lenient composition of S and C. Using lenient composition an OT analysis can be written as in fig. 2.

```
Gen
lc
Constraint1
lc
...
lc
ConstraintN
```

Figure 2: Optimality Theory as Generate and Test with Lenient Composition

The use of lenient composition, however, is not sufficient for implementing optimality theory. In general, a candidate string can violate a constraint multiple times and candidates that violate the constraint the least number of times need to be preferred. Lenient composition is sufficient to prefer a candidate that violates the constraint 0 times over a candidate that violates the constraint at least once. However, lenient composition cannot distinguish two candidates if the first contains one violation, and the second contains at least two violations.

The problem of implementing optimality theory becomes considerably harder when constraint violations need to be counted. As Frank and Satta (1998) have shown, an OT describes a regular relation under the assumptions that `Gen` is a regular relation, and each of the constraints is a regular relation which maps a candidate string to a natural number (indicating the number of constraint violations in that candidate), where the range of each constraint is finite. If constraints are defined in such a way that there is no bound to the number of constraint violations that can occur in a given string, then the resulting OT may describe a relation that is not regular. A simple example of such an OT (attributed to Markus Hiller) is the OT in which the inputs of interest are of the form `[a*,b*]`, *Gen* is defined as a transducer which maps all `a`'s to `b`'s and all `b`'s to `a`'s, or alternatively, it performs the identity map on each `a` and `b`:

```
{[(a x b)*,(b x a)*],
 [(a x a)*,(b x b)*]}
```

This OT contains only a single constraint, $*A$: a string should not contain `a`. As can easily be verified, this OT defines the relation $\{(a^nb^m, a^nb^m)|n \leq m\} \cup \{(a^nb^m, b^na^m)|m \leq n\}$, which can easily be shown to be non-regular.

Although the OT described above is highly unrealistic for natural language, one might nevertheless expect that a constraint on syllable structure in the analysis of Prince & Smolensky would require an unbounded amount of counting (since words are of unbounded length), and that therefore such analyses would not be describable as regular relations. An important conclusion of this paper is that, contrary to this potential expectation, such cases in fact *can* be shown to be regular.

2.3 Syllabification in Finite State OT

In order to illustrate our approach, we will start with a finite state implementation of the syllabification analysis as presented in chapter 6 of Prince and Smolensky (1993). This section is heavily based on Karttunen (1998), which the reader should consult for more explanation and examples.

The inputs to the syllabification OT are sequences of consonants and vowels. The input will be marked up with *onset*, *nucleus*, *coda* and *unparsed* brackets; where a syllable is a sequence of an optional onset, followed by a nucleus, followed by an optional coda. The input will be marked up as a sequence of such syllables, where at arbitrary places *unparsed* material can intervene. The assumption is that an unparsed vowel or consonant is not spelled out phonetically. Onsets, nuclei and codas are also allowed to be empty; the phonetic interpretation of such constituents is *epenthesis.*

First we give a number of simple abbreviations:

```
macro(cons,
      {b,c,d,f,g,h,j,k,l,m,n,
       p,q,r,s,t,v,w,x,y,z}  ).
macro(vowel,   {a,e,o,u,i}).

macro(o_br,     'O['). % onset
macro(n_br,     'N['). % nucleus
macro(d_br,     'D['). % coda
macro(x_br,     'X['). % unparsed
macro(r_br,     ']').
macro(bracket,
      {o_br,n_br,d_br,x_br,r_br}).
```

```
macro(onset,    [o_br,cons^  ,r_br]).
macro(nucleus, [n_br,vowel^ ,r_br]).
macro(coda,     [d_br,cons^  ,r_br]).
macro(unparsed,[x_br,letter ,r_br]).
```

Following Karttunen, *Gen* is formalized as in fig. 3. Here, `parse` introduces *onset*, *coda* or *unparsed* brackets around each consonant, and *nucleus* or *unparsed* brackets around each vowel. The `replace(T,Left,Right)` transducer applies transducer `T` obligatory within the contexts specified by `Left` and `Right` (Gerdemann and van Noord, 1999). The `replace(T)` transducer is an abbreviation for `replace(T,[],[])`, i.e. `T` is applied everywhere. The *overparse* transducer introduces optional 'empty' constituents in the input, using the *intro_each_pos* operator.[4]

In the definitions for the constraints, we will deviate somewhat from Karttunen. In his formalization, a constraint simply describes the set of strings which do not violate that constraint. It turns out to be easier for our extension of Karttunen's formalization below, as well as for our alternative approach, if we return to the concept of a constraint as introduced by Prince and Smolensky where a constraint adds *marks* in the candidate string at the position where the string violates the constraint. Here we use the symbol `@` to indicate a constraint violation. After checking each constraint the markers will be removed, so that markers for one constraint will not be confused with markers for the next.

```
macro(mark_violation(parse),
     replace(([] x @),x_br,[]).

macro(mark_violation(no_coda),
     replace(([] x @),d_br,[]).

macro(mark_violation(fill_nuc),
     replace(([] x @),[n_br,r_br],[])).

macro(mark_violation(fill_ons),
     replace(([] x @),[o_br,r_br],[])).

macro(mark_violation(have_ons),
     replace(([] x @),[],n_br)
                      o
     replace((@ x []),onset,[])).
```

The *parse* constraint simply states that a candidate must not contain an unparsed constituent. Thus, we add a mark after each unparsed bracket. The *no_coda* constraint is similar: each coda bracket will be marked. The *fill_nuc* constraint is only slightly more complicated: each sequence of a nucleus bracket immediately followed by a closing bracket is marked. The *fill_ons* constraint treats empty onsets in the same way. Finally, the *have_ons* constraint is somewhat more complex. The constraint requires that each nucleus is preceded by an onset. This is achieved by marking all nuclei first, and then removing those marks where in fact an onset is present.

This completes the building blocks we need for an implementation of Prince and Smolensky's analysis of syllabification. In the following sections, we present two alternative implementations which employ these building blocks. First, we discuss the approach of Karttunen (1998), based on the lenient composition operator. This approach uses a *counting* approach for multiple constraint violations. We will then present an alternative approach in which constraints eliminate candidates using *matching*.

3 The Counting Approach

In the approach of Karttunen (1998), a candidate set is leniently composed with the set of strings which satisfy a given constraint. Since we have defined a constraint as a transducer which marks candidate strings, we need to alter the definitions somewhat, but the resulting transducers are equivalent to the transducers produced by Karttunen (1998). We use the (left-associative) *optimality operator* `oo` for applying an OT constraint to a given set of candidates:[5]

[4] An alternative would be to define *overparse* with a Kleene star in place of the option operator. This would introduce unbounded sequences of empty segments. Even though it can be shown that, with the constraints assumed here, no optimal candidate ever contains two empty segments in a row (proposition 4 of Prince and Smolensky (1993)) it is perhaps interesting to note that defining *Gen* in this alternative way causes cases of infinite ambiguity for the counting approach but is unproblematic for the matching approach.

[5] The operators 'o' and 'lc' are assumed to be left associative and have equal precedence.

```
macro(gen,       {cons,vowel}*
                       o
                   overparse
                       o
                     parse
                       o
                syllable_structure ).

macro(parse, replace([[] x {o_br,d_br,x_br},cons, [] x r_br])
                                 o
             replace([[] x {n_br,x_br},     vowel,[] x r_br])).

macro(overparse,intro_each_pos([{o_br,d_br,n_br},r_br]^)).

macro(intro_each_pos(E), [[ [] x E, ?]*,[] x E]).

macro(syllable_structure,ignore([onset^,nucleus,coda^],unparsed)*).
```

Figure 3: The definition of *Gen*

```
macro(Cands oo Constraint,
          Cands
            o
  mark_violation(Constraint)
            lc
        ~ ($ @)
            o
     { @ x [], ? - @}*    ).
```

Here, the set of candidates is first composed with the transducer which marks constraint violations. We then leniently compose the resulting transducer with `~($ @)`[6], which encodes the requirement that no such marks should be contained in the string. Finally, the remaining marks (if any) are removed from the set of surviving candidates. Using the optimality operator, we can then combine *Gen* and the various constraints as in the following example (equivalent to figure 14 of Karttunen (1998)):

```
macro(syllabify, gen
                  oo
               have_ons
                  oo
               no_coda
                  oo
               fill_nuc
                  oo
                parse
                  oo
               fill_ons     ).
```

As mentioned above, a candidate string can violate a constraint multiple times and candidates that violate the constraint the least number of times need to be preferred. Lenient composition cannot distinguish two candidates if the first contains one violation, and the second contains at least two violations. For example, the above `syllabify` transducer will assign three outputs to the input `bebop`:

```
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]
```

In this case, the second output should have been preferred over the other two, because the second output violates 'Parse' only once, whereas the other outputs violate 'Parse' three times. Karttunen recognizes this problem and proposes to have a sequence of constraints Parse0, Parse1, Parse2 ... ParseN, where each ParseX constraint requires that candidates not contain more than X unparsed constituents.[7] In this case, the resulting transducer only *approximates*

[6] As explained in footnote 2, this will be coerced into an identity transducer.

[7] This construction is similar to the construction in Frank and Satta (1998), who used a suggestion in Ellison (1994).

the OT analysis, because it turns out that for any X there are candidate strings that this transducer fails to handle correctly (assuming that there is no bound on the length of candidate strings).

Our notation is somewhat different, but equivalent to the notation used by Karttunen. Instead of a sequence of constraints Cons0 ... ConsX, we will write `Cands oo Prec :: Cons`, which is read as: apply constraint `Cons` to the candidate set `Cands` with *precision* `Prec`, where "precision" means the predetermined bound on counting. For example, a variant of the `syllabify` constraint can be defined as:

```
macro(syllabify, gen
                    oo
                 have_ons
                    oo
                 no_coda
                    oo
                 1 :: fill_nuc
                    oo
                 8 :: parse
                    oo
                 fill_ons     ).
```

Using techniques described in §5, this variant can be shown to be *exact* for all strings of length ≤ 10. Note that if no precision is specified, then a precision of 0 is assumed.

This construct can be defined as follows (in the actual implementation the regular expression is computed dynamically based on the value of `Prec`):

```
macro(Cands oo 3 :: Constraint,
          Cands
            o
  mark_violation(Constraint)
            lc
  ~ ([($ @),($ @),($ @),($ @)])
            lc
    ~ ([($ @),($ @),($ @)])
            lc
      ~ ([($ @),($ @)])
            lc
        ~ ($ @)
            o
     { @ : [], ? - @}*  ).
```

4 The Matching Approach

4.1 Introduction

In order to illustrate the alternative approach, based on matching we return to the `bebop` example given earlier, repeated here:

```
c1:   O[ b ] N[ e ] X[ b ] X[ o ] X[ p ]
c2:   O[ b ] N[ e ] O[ b ] N[ o ] X[ p ]
c3:   X[ b ] X[ e ] O[ b ] N[ o ] X[ p ]
```

Here an instance of 'X[' is a constraint violation, so c2 is the best candidate. By counting, one can see that c2 has one violation, while c1 and c3 each have 3. By matching, one can see that all candidates have a violation in position 13, but c1 and c3 also have violations in positions not corresponding to violations in c2. As long the positions of violations line up in this manner, it is possible to construct a finite state filter to rule out candidates with a non-minimal number of violations. The filter will take the set of candidates, and subtract from that set all strings that are similar, except that they contain additional constraint violations.

Given the approach of marking up constraint violations introduced earlier, it is possible to construct such a matching filter. Consider again the 'bebop' example. If the violations are marked, the candidates of interest are:

```
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 ]
```

For the filter, we want to compare alternative mark-ups for the same input string. Any other differences between the candidates can be ignored. So the first step in constructing the filter is to eliminate everything except the markers and the original input. For the syllable structure example, finding the original input is easy since it never gets changed. For the "bebop" example, the filter first constructs:

```
   b   e @ b @ o   @   p
   b   e   b   o   @   p
 @ b @ e   b   o   @   p
```

Since we want to rule out candidates with at least one more constraint violation than necessary, we apply a transducer to this set which inserts at least one more marker. This will yield an infinite set of bad candidates each of which has at least two markers and with one of the markers coming directly before the final 'p'.

In order to use this set of bad candidates as a filter, brackets have to be reinserted. But since the filter does not care about positions of brackets, these can be inserted randomly. The result is the set of all strings with at least two markers, one of the markers coming directly before the final 'p', and arbitrary brackets anywhere. This set includes the two candidates c1 and c3 above. Therefore, after applying this filter only the optimal candidate survives. The three steps of deleting brackets, adding extra markers and randomly reinserting brackets are encoded in the `add_violation` macro given in fig. 4.

The application of an OT constraint can now be defined as follows, using an alternative definition of the optimality operator:

```
macro(Cands oo Constraint,
          Cands
            o
   mark_violation(Constraint)
            o
     ~ range(Cands
               o
         mark_violation(Constraint)
               o
          add_violation)
            o
      {(@ x []),(? -  @)}* ).
```

Note that this simple approach only works in cases where constraint violations line up neatly. It turns out that for the syllabification example discussed earlier that this is the case. Using the `syllabify` macro given above with this matching implementation of the optimality operator produces a transducer of only 22 states, and can be shown to be exact for all inputs!

4.2 Permutation

In the general case, however, constraint violations need not line up. For example, if the order of constraints is somewhat rearranged as in:

```
parse oo fill_ons oo have_ons
    oo fill_nuc oo no_coda
```

the matching approach is not exact: it will produce wrong results for an input such as 'arts':

```
N[a]D[r]O[t]N[]D[s]        %cf: art@s
N[a]O[r]N[]D[t]O[s]N[]     %cf: ar@ts@
```

Here, the second output should not be produced because it contains one more violation of the `fill_nuc` constraint. In such cases, a limited amount of permutation can be used in the filter to make the marker symbols line up. The `add_violation` filter of fig. 4 can be extended with the following transducer which permutes marker symbols:

```
macro(permute_marker,
 [{[? *,(@ x []),? *,([] x @)],
   [? *,([] x @),? *,(@ x [])]}*,? *]).
```

Greater degrees of permutation can be achieved by composing `permute_marker` several times. For example:[8]

```
macro(add_violation(3),
 {(bracket x []), (? - bracket)}*
               o
      [[? *,([] x @)]+, ? *]
               o
        permute_marker
               o
        permute_marker
               o
        permute_marker
               o
 {([] x bracket), (? - bracket)}*  ).
```

So we can incorporate a notion of 'precision' in the definition of the optimality operator for the matching approach as well, by defining:

```
macro(Cands oo Prec :: Constraint),
        Cands
          o
 mark_violation(Constraint)
          o
   ~ range(Cands
             o
       mark_violation(Constraint)
             o
        add_violation(Prec))
          o
   { (@ x []),(? - @)}*  ).
```

[8] An alternative approach would be to compose the `permute_marker` transducers before inserting extra markers. Our tests, however, show this alternative to be somewhat less efficient.

```
macro(add_violation,
      {(bracket x []), ? - bracket}*    % delete brackets
                     o
           [[? *,([] x @)]+, ? *]       % add at least one @
                     o
      {([] x bracket), ? - bracket}*    % reinsert brackets
     ).
```

Figure 4: Macro to introduce additional constraint violation marks.

The use of permutation is most effective when constraint violations in alternative candidates tend to occur in corresponding positions. In the worst case, none of the violations may line up. Suppose that for some constraint, the input "bebop" is marked up as:

```
c1:  @ b @ e   b   o   p
c2:    b   e @ b @ o @ p
```

In this case, the precision needs to be two in order for the markers in c1 to line up with markers in c2. Similarly, the counting approach also needs a precision of two in order to count the two markers in c1 and prefer this over the greater than two markers in c2. The general pattern is that any constraint that can be treated exactly with counting precision N, can also be handled by matching with precision less than or equal to N. In the other direction, however, there are constraints, such as those in the Prince and Smolensky syllabification problem, that can only be exactly implemented by the matching approach.

For each of the constraint orderings discussed by Prince and Smolensky, it turns out that at most a single step of permutation (i.e. a precision of 1) is required for an exact implementation. We conclude that this OT analysis of syllabification is regular. This improves upon the result of Karttunen (1998). Moreover, the resulting transducers are typically much smaller too. In §5 we present a number of experiments which provide evidence for this observation.

4.3 Discussion

Containment. It might be objected that the Prince and Smolensky syllable structure example is a particularly simple *containment theory* analysis and that other varieties of OT such as *correspondence theory* (McCarthy and Prince, 1995) are beyond the scope of matching.[9] Indeed we have relied on the fact that *Gen* only adds brackets and does not add or delete anything from the set of input symbols. The filter that we construct needs to compare candidates with alternative candidates generated *from the same input.*

If *Gen* is allowed to change the input then a way must be found to remember the original input. Correspondence theory is beyond the scope of this paper, however a simple example of an OT where *Gen* modifies the input is provided by the problem described in §2.2 (from Frank and Satta (1998)). Suppose we modify *Gen* here so that its output includes a representation of the original input. One way to do this would be to adopt the convention that input symbols are marked with a following 0 and output symbols are marked with a following 1. With this convention *Gen* becomes:

```
macro(gen,
 {[(a x [a,0,b,1])*,(b x [b,0,a,1])*],
  [(a x [a,0,a,1])*,(b x [b,0,b,1])*]})
```

Then the constraint against the symbol `a` needs to be recast as a constraint against `[a,1]`.[10] And, whereas above *add_violation* was previously written to ignore brackets, for this case it will need to ignore output symbols

[9] Kager (1999) compares containment theory and correspondence theory for the syllable structure example.

[10] OT makes a fundamental distinction between *markedness* constraints (referring only to the surface) and *faithfulness* constraints (referring to both surface and underlying form). With this mark-up convention, faithfulness constraints might be allowed to refer to both symbols marked with 0 and symbols marked with 1. But note that the `Fill` and `Parse` constraints in syllabification are also considered to be faithfulness constraints since they correspond to epenthesis and deletion respectively.

(marked with a 1). This approach is easily implementable and with sufficient use of permutation, an approximation can be achieved for any predetermined bound on input length.

Locality. In discussing the impact of their result, Frank and Satta (1998) suggest that the OT formal system is too rich in generative capacity. They suggest a *shift in the type of optimization carried out in OT, from global optimization over arbitrarily large representations to local optimization over structural domains of bounded complexity.* The approach of matching constraint violations proposed here is based on the assumption that constraint violations can indeed be compared *locally.*

However, if *locality* is crucial then one might wonder why we extended the local matching approach with global permutation steps. Our motivation for the use of global permutation is the observation that it ensures the matching approach is strictly more powerful than the counting approach. A weaker, and perhaps more interesting, treatment is obtained if locality is enforced in these permutation steps as well. For example, such a weaker variant is obtained if the following definition of `permute_marker` is used:

```
macro(permute_marker,   % local variant
 {? ,[([] x @),?,(@ x [])],
     [(@ x []),?,([] x @)]}* ).
```

This is a weaker notion of permutation than the definition given earlier. Interestingly, using this definition resulted in equivalent transducers for all of the syllabification examples given in this paper. In the general case, however, matching with local permutation is less powerful.

Consider the following artificial example. In this example, inputs of interest are strings over the alphabet $\{b, c\}$. *Gen* introduces an `a` *before* a sequence of `b`'s, or two `a`'s *after* a sequence of `b`'s. *Gen* is given as an automaton in fig. 5. There is only a single constraint, which forbids `a`. It can easily be verified that a matching approach with global permutation using a precision of 1 exactly implements this OT. In contrast, both the counting approach as well as a matching approach based on local permutation can only approximate this OT.[11]

[11]Matching with local permutation is not strictly more powerful than counting. For an example, change *Gen* in this example to: `{[([] x a),{b,c}*],[{b,c}*,([] x [a,a])]}`. This can be exactly implemented by counting with a precision of one. Matching with local permutation, however, cannot exactly implement this case, since markers would need to be permuted across unbounded sequences.

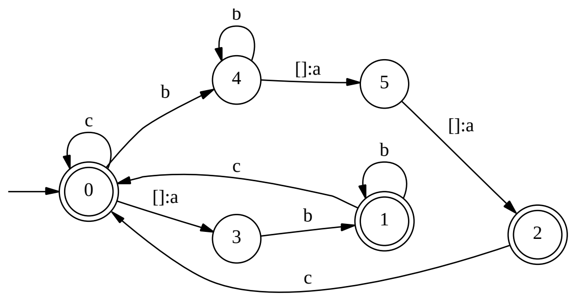

Figure 5: *Gen* for an example for which local permutation is not sufficient.

5 Comparison

In this section we compare the two alternative approaches with respect to accuracy and the number of states of the resulting transducers. We distinguish between *exact* and *approximating* implementations. An implementation is exact if it produces the right result for all possible inputs.

Assume we have a transducer T which correctly implements an OT analysis, except that it perhaps fails to distinguish between different numbers of constraint violations for one or more relevant constraints. We can decide whether this T is exact as follows. T is exact if and only if T is exact with respect to each of the relevant constraints, i.e., for each constraint, T distinguishes between different numbers of constraint violations. In order to check whether T is exact in this sense for constraint C we create the transducer `is_exact(T,C)`:

```
macro(is_exact(T,C),
          T
          o
    mark_violation(C)
          o
  {(? - @) x [], @}*).
```

If there are inputs for which this transducer produces multiple outputs, then we know that `T` is not exact for `C`; otherwise `T` *is* exact for `C`. This reduces to the question of whether `is_exact(T,C)` is ambiguous. The question of whether a given transducer is ambiguous is shown to be decidable in (Blattner and Head, 1977); and an efficient algorithm is proposed in (Roche and Schabes, 1997).[12] Therefore, in order to check a given transducer T for exactness, it must be the case that for each of the constraints C, is_exact(T,C) is nonambiguous.

If a transducer T is not exact, we characterize the quality of the approximation by considering the maximum length of input strings for which T is exact. For example, even though T fails the exactness check, it might be the case that

```
[? ^,? ^,? ^,? ^,? ^]
         o
         T
```

in fact is exact, indicating that T produces the correct result for all inputs of length ≤ 5.

Suppose we are given the sequence of constraints:

```
have_ons >> fill_ons >> parse
   >> fill_nuc >> no_coda
```

and suppose furthermore that we require that the implementation, using the counting approach, must be exact for all strings of length ≤ 10. How can we determine the level of precision for each of the constraints? A simple algorithm (which does not necessarily produce the smallest transducer) proceeds as follows. Firstly, we determine the precision of the first, most important, constraint by checking exactness for the transducer

```
gen oo P :: have_ons
```

for increasing values for P. As soon as we find the minimal P for which the exactness check succeeds (in this case for P=0), we continue by determining the precision required for the next constraint by finding the minimal value of P in:

```
gen oo 0 :: have_ons oo P :: fill_ons
```

We continue in this way until we have determined precision values for each of the constraints. In this case we obtain a transducer with 8269 states implementing:

[12] We have adapted the algorithm proposed in (Roche and Schabes, 1997) since it fails to treat certain types of transducer correctly; we intend to provide details somewhere else.

```
gen oo 0 :: have_ons
    oo 1 :: fill_ons
    oo 8 :: parse
    oo 5 :: fill_nuc
    oo 4 :: no_coda
```

In contrast, using matching an exact implementation is obtained using a precision of 1 for the fill_nuc constraint; all other constraints have a precision of 0. This transducer contains only 28 states.

The assumption in OT is that each of the constraints is universal, whereas the constraint *order* differs from language to language. Prince and Smolensky identify nine interestingly different constraint orderings. These nine "languages" are presented in table 2.

In table 3 we compare the size of the resulting automata for the matching approach, as well as for the counting approach, for three different variants which are created in order to guarantee exactness for strings of length ≤ 5, ≤ 10 and ≤ 15 respectively.

Finally, the construction of the transducer using the matching approach is typically much faster as well. In table 4 some comparisons are summarized.

6 Conclusion

We have presented a new approach for implementing OT which is based on matching rather than the counting approach of Karttunen (1998). The matching approach shares the advantages of the counting approach in that it uses the finite state calculus and avoids off-line sorting and counting of constraint violations. We have shown that the matching approach is superior in that analyses that can only be approximated by counting can be exactly implemented by matching. Moreover, the size of the resulting transducers is significantly smaller.

We have shown that the matching approach along with global permutation provides a powerful technique technique for minimizing constraint violations. Although we have only applied this approach to permutations of the Prince & Smolensky syllabification analysis, we speculate that the approach (even with local permutation) will also yield exact implementations for most other OT phonological analyses. Further investigation is needed here, particularly with recent versions of OT such as cor-

id	constraint order
1	`have_ons` ≫ `fill_ons` ≫ `no_coda` ≫ `fill_nuc` ≫ `parse`
2	`have_ons` ≫ `no_coda` ≫ `fill_nuc` ≫ `parse` ≫ `fill_ons`
3	`no_coda` ≫ `fill_nuc` ≫ `parse` ≫ `fill_ons` ≫ `have_ons`
4	`have_ons` ≫ `fill_ons` ≫ `no_coda` ≫ `parse` ≫ `fill_nuc`
5	`have_ons` ≫ `no_coda` ≫ `parse` ≫ `fill_nuc` ≫ `fill_ons`
6	`no_coda` ≫ `parse` ≫ `fill_nuc` ≫ `fill_ons` ≫ `have_ons`
7	`have_ons` ≫ `fill_ons` ≫ `parse` ≫ `fill_nuc` ≫ `no_coda`
8	`have_ons` ≫ `parse` ≫ `fill_ons` ≫ `fill_nuc` ≫ `no_coda`
9	`parse` ≫ `fill_ons` ≫ `have_ons` ≫ `fill_nuc` ≫ `no_coda`

Table 2: Nine different constraint orderings for syllabification, as given in Prince and Smolensky, chapter 6.

Method	Exactness	Constraint order								
		1	2	3	4	5	6	7	8	9
matching	exact	29	22	20	17	10	8	28	23	20
counting	≤ 5	95	220	422	167	10	240	1169	2900	4567
counting	≤ 10	280	470	1667	342	10	420	8269	13247	16777
counting	≤ 15	465	720	3812	517	10	600	22634	43820	50502

Table 3: Comparison of the matching approach and the counting approach for various levels of exactness. The numbers indicate the number of states of the resulting transducer.

respondence theory. Another line of further research will be the proper integration of finite state OT with non-OT phonological rules as discussed, for example, in papers collected in Hermans and van Oostendorp (1999) .

Finally, we intend also to investigate the application of our approach to syntax. Karttunen (1998) suggests that the Constraint Grammar approach of Karlsson et al. (1995) could be implemented using lenient composition. If this is the case, it could most probably be implemented more precisely using the matching approach. Recently, Oflazer (1999) has presented an implementation of Dependency syntax which also uses lenient composition with the counting approach. The alternative of using a matching approach here should be investigated.

References

Meera Blattner and Tom Head. 1977. Single-valued a-transducers. *Journal of Computer and System Sciences*, 15(3):328–353.

Mark T. Ellison. 1994. Phonological derivation in optimality theory. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*, pages 1007–1013, Kyoto.

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

Dale Gerdemann and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, Bergen Norway.

Ben Hermans and Marc van Oostendorp, editors. 1999. *The Derivational Residue in Phonological Optimality Theory*, volume 28 of *Linguistik Aktuell/Linguistics Today*. John Benjamins, Amsterdam/Philadelphia.

C. Douglas Johnson. 1972. *Formal Aspects of Phonological Descriptions*. Mouton, The Hague.

René Kager. 1999. *Optimality Theory*. Cambridge UP, Cambridge, UK.

Ronald Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–379.

Fred Karlsson, Atro Voutilainen, Juha Heikkilä, and Arto Anttila. 1995. *Constraint Gram-*

Method	Exactness	Constraint order								
		1	2	3	4	5	6	7	8	9
matching	exact	1.0	0.9	0.9	0.9	0.8	0.7	1.5	1.3	1.1
counting	≤ 5	0.9	1.7	4.8	1.6	0.5	1.9	10.6	18.0	30.8
counting	≤ 10	2.8	4.7	28.6	4.0	0.5	4.2	83.2	112.7	160.7
counting	≤ 15	6.8	10.1	99.9	8.6	0.5	8.2	336.1	569.1	757.2

Table 4: Comparison of the matching approach and the counting approach for various levels of exactness. The numbers indicate the CPU-time in seconds required to construct the transducer.

mar: A Language-Independent Framework for Parsing Unrestricted Text. Mouton de Gruyter, Berlin/New York.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–238.

Lauri Karttunen. 1995. The replace operator. In *33th Annual Meeting of the Association for Computational Linguistics*, M.I.T. Cambridge Mass.

Lauri Karttunen. 1996. Directed replacement. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz.

Lauri Karttunen. 1998. The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*, pages 1–12, Ankara.

André Kempe and Lauri Karttunen. 1996. Parallel replacement in the finite-state calculus. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, Copenhagen, Denmark.

Charles Kisseberth. 1970. On the functional unity of phonological rules. *Linguistic Inquiry*, 1:291–306.

John McCarthy and Alan Prince. 1995. Faithfulness and reduplicative identity. In Jill Beckman, Laura Walsh Dickey, and Suzanne Urbanczyk, editors, *Papers in Optimality Theory*, pages 249–384. Graduate Linguistic Student Association, Amherst, Mass. University of Massachusetts Occasional Papers in Linguistics 18.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz.

Kemal Oflazer. 1999. Dependency parsing with an extended finite state approach. In *37th Annual Meeting of the Association for Computational Linguistics*, pages 254–260.

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. MIT Press, To Appear.

Emmanuel Roche and Yves Schabes. 1997. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, Mass.

Bruce Tesar. 1995. *Computational Optimality Theory*. Ph.D. thesis, University of Colorado, Boulder.

Gertjan van Noord and Dale Gerdemann. 1999. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt, H. Juergensen, and L. Robbins, editors, *Workshop on Implementing Automata; WIA99 Pre-Proceedings*, Potsdam, Germany.

Gertjan van Noord. 1997. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*, pages 87–108. Springer Verlag. Lecture Notes in Computer Science 1260.

Gertjan van Noord. 1999. FSA6 reference manual. The *FSA Utilities* toolbox is available free of charge under Gnu General Public License at http://www.let.rug.nl/˜vannoord/Fsa/.

Markus Walther. 1996. OT simple – a construction-kit approach to optimality theory implementation. ROA-152-1096.