

Handling the infinite candidate set

A finite state implementation of optimality theory for loanword phonology

Thora Daneyko

Contact: thora.daneyko@student.uni-tuebingen.de

October 18, 2017

Abstract

Manual construction of Optimality Theory (OT) tableaux, as is still the default procedure in contemporary linguistics, has the deficit of not covering the complete, i.e. infinite, set of candidates. Even though proposals of how to handle the infinite set computationally by using finite state transducers (FSTs) have been around for almost 20 years, current OT software provides no means of checking against infinitely many candidates. I present a new OT implementation based on finite state technology which is able to derive a winner for any input from an infinite candidate set, while being accessible and easy to use.

1 Introduction

Ever since its introduction in 1993, Optimality Theory (OT; Prince and Smolensky 2008) has been a popular framework in phonology, being able to successfully explain a range of phonological phenomena. However, most of the work associated with OT, especially the generation of output candidates, is usually done by hand and rarely automated. This leads to many possible

candidates being missed: The real set of candidates is in fact infinite, and while the majority of these possible candidates is completely unrelated to the input or even unpronounceable, we cannot speak of a thorough analysis if we are not sure that all of the infinite bad candidates have been dealt with by our constraints. Even worse, concentrating on a handful of “human-generated” examples, it is easy to overlook a likely candidate that outperforms the desired winner under the current order of constraints.

To avoid such a faulty analysis due to missed candidates, Karttunen (1998) suggests to use finite state transducers (FSTs) to generate an infinite candidate set and reduce it to a single winning candidate. An FST is able to encode mappings between an infinite amount of strings and multiple transducers can be combined into a single one which simply yields the correct output for an arbitrary input. Karttunen’s approach has later been improved by Gerdemann and Noord (2000); however, there is no comprehensive implementation of their algorithms yet.

Most of the contemporary OT software focuses on finding the optimal order of constraints for a given input and optimal candidate, and these programs usually ask the user to provide the candidates. The OT implementation contained in the phonetics software *Praat* (Boersma and Weenink 2017) even needs the violation counts for each candidate and constraint, requiring the user to construct the entire tableau by hand. Another popular constraint ranking program, *OTSoft* (Hayes, Tesar, and Zuraw 2013), is able to also apply user-defined constraints itself to get the number of violations, but allows only for very simple context-independent constraints. It too needs to be provided with user-generated candidates.

Few OT implementations are able to generate candidates automatically and none of them can handle an infinite candidate set. In *OT-Help* (Staubs et al. 2010), the user can define complex context-sensitive generation operations to be applied to the input. Since these are formulated as regular expressions, it is theoretically possible to generate infinitely many. However, while the program can count violations for most of the constraints automatically, faithfulness constraints are an exception: Instead, for each generation operation (such as replacing any plosive by a nasal), the user has to define which faithfulness constraints (such as ID(PLOSIVE)) are violated by the operation. It is therefore not feasible to have a single operation mapping any input to any output.

The powerful *OTKit* (Biró 2010), in contrast, provides a large range of constraint schemes that can be applied automatically, including faithfulness

constraints, as well as a number of generation functions, including an infinite one. In addition, there is the possibility to define custom constraints and generators using the underlying Java library. However, the system is not finite state based, and hence, an infinite generator has to be limited to generate only the first n candidates to prevent memory from overflowing. While the system can nevertheless provide an extremely large amount of candidates that should cover all feasible possibilities, there is no truly infinite candidate set. Also, the complexity of the program requires some time to become acquainted with it and renders it rather inaccessible to a technically not so versed person.

In this paper, I introduce my finite state based Python implementation of an OT tableau that is able to generate an infinite and completely unrestricted candidate set, mark violations of constraints and filter out losing candidates. The final FST is small and fast, and can produce the winner(s) for any input. Intermediate FSTs additionally provide the possibility to view violation marks assigned by any (finite) constraint and watch selected candidates win and lose across the tableau. Straightforward and simple methods and classes enable easy usage, and many options as well as the possibility to define custom generators and constraints provide the means for building complex tableaux.

In the following sections, I explain the usage of my implementation using the example of English loanwords in Hawaiian. Section 2 gives a brief overview over English-to-Hawaiian loanwords and introduces some manual tableaux for selected borrowings. In section 3, I then go through the individual parts of my implementation and reproduce these manual tableaux in the program.

2 Example: English-to-Hawaiian loanwords

The Hawaiian language (*‘Ōlelo Hawai‘i*) is known for having one of the smallest consonant inventories among the languages of world. This and its restricted (C)V(V) syllable structure often lead to foreign words being altered drastically when they enter the language as loans. Because of this, Hawaiian is a popular example for illustrating loanword phonology. This section gives a brief overview over Hawaiian’s phonology and the adaption of English loanwords into the language.

	Labial	Alveolar	Velar	Glottal
Plosive	p	(t)	k	ʔ
	<i>p</i>	<i>k</i>	<i>k</i>	‘
Nasal	m	n		
	<i>m</i>	<i>n</i>		
Fricative	(v)			h
	<i>w</i>			<i>h</i>
Approximant	w	l		
	<i>w</i>	<i>l</i>		

Table 1: The consonants of Hawaiian (orthographic representation in italics).

2.1 Phonology of Hawaiian

Table 1 shows the consonant inventory of Hawaiian (Elbert and Pukui 2001). There are only eight consonant phonemes. The velar plosive [k] is in free variation with the alveolar plosive [t] in the Ni‘ihau dialect (Elbert and Pukui 2001, p. 24f), but since it is written *k* and usually pronounced [k] in the standard dialect, I will only transcribe it as [k] from now on. Likewise, the labials [v] and [w] are in free variation in all dialects, even though some tend more towards one variant than the other (Elbert and Pukui 2001, p. 12f). Since the occurrence of either of them is not entirely predictable, I will consistently transcribe them both as [w].

Hawaiian has a standard five vowel system consisting of [a], [e], [i], [o] and [u], which can be both short and long (marked by a macron in orthography), yielding ten distinct vowel phonemes (Elbert and Pukui 2001). I will not discuss Hawaiian’s diphthongs here because my program is not yet able to handle diphthongs.

As briefly mentioned before, Hawaiian only allows (C)V(V) syllables, i.e. maximally one consonant in the onset and no consonants in the coda, while the nucleus may contain a single short or long vowel or a diphthong (Elbert and Pukui 2001). To handle consonant clusters from source languages with more complex syllable structures such as English, additional vowels are inserted or consonants, usually continuants such as [s] or [n] in clusters or coda position, are deleted.

2.2 Borrowings from English

Hawaiian has many English loanwords, most of them stemming from the missionaries studying the language in the 19th century and the resulting Hawaiian Bible translation (Parker Jones 2009). English is one of the official languages of Hawaii and used to be the only official language for decades, so new loanwords from English continued to enter Hawaiian until today.

Over time, English loanwords have been adapted quite inconsistently into the Hawaiian language, rendering it impossible to construct a single set of constraints correctly predicting all borrowings. The first loans were created by the missionaries who were not entirely proficient in Hawaiian and often borrowed English words according to their own ideas. Many adaptations were based on orthography rather than pronunciation (e.g. *hymn* [hɪm] → *hīmeni*, *lion* [laɪən] → *liona*) (Elbert and Pukui 2001, p. 28). The vowels inserted to break up consonant clusters and coda consonants differed between borrowers (e.g. *Martha* → *Malaka*/*Maleka*, *Gertrude* → *Kelekuluke*/*Kekaluka*) as well as the choice which input consonants to drop and which to retain (e.g. coda *r* retained in *March* → *Malaki*, but dropped in *crowbar* → *kolopā*) (Elbert and Pukui 2001, p. 27ff). Due to the insufficient historical spelling, glottal stops were inserted into many places where they did not belong (e.g. *Aaron* → *‘A‘alona*) (Elbert and Pukui 2001, p. 32).

Since this paper is not primarily about the problem of English loanwords in Hawaiian, but rather about a tool for solving this problem, I will only treat a few straightforward and consistent borrowings in the following sections. The constraints and tableaux I establish are by no means supposed to be exhaustive, but only serve illustrative purposes.

2.3 Example tableaux

The three example loanwords that we are going to investigate using my program are *letter* [lɛ.tə], *frog* [fɹɒg] and *kangaroo* [kaŋ.gə.ru:] (IPA according to *Wiktionary* 2017) which have been adapted into Hawaiian as *leka* [le.ka], *poloka* [po.lo.ka] and *kanakalū* [ka.na.ka.lu:] (Parker Jones 2009, IPA by myself based on section 2.1). In this section, I will briefly construct a manual tableau for each of them to obtain the necessary constraints. In the following section, I am going to implement these constraints in my program to see whether it yields the same results.


le.tə	*FOREIGN	ID(CONSONANTAL)	ID(+STOP)	ID(+LINGUAL)	ID(BACKNESS)	ID(HEIGHT)	ID(PLACE)
a. le.tə	3*!					*	
b. le.ia		*!				*	
c. le.na			*!			*	
d. le.pa				*!		*	*
e. le.ke					*!		*
f. li.ka						**!	*
 g. le.ka						*	*

Table 2: Tableau for English *letter* adapted into Hawaiian as *leka*.

2.3.1 Letter → leka

The adaption of *letter*, as shown in Table 2, only requires a mapping from English to Hawaiian sounds, since it does conform to Hawaiian’s (C)V(V) syllable structure. Instead of establishing markedness constraints for all non-Hawaiian sounds in the input, I bundle them up into *FOREIGN to keep the tableau simple. In order to ensure that [t] becomes [k] and not a vowel, a non-plosive or a labial plosive, we need ID(CONSONANTAL), ID(+STOP) and ID(+LINGUAL). To keep the vowels close to their input counterparts, I further establish ID(BACKNESS) and ID(HEIGHT). So far, the order of the constraints seems to be irrelevant, except that ID(BACKNESS) has to be ranked above ID(HEIGHT) to prevent [le.ke] from winning. Another common constraint, ID(PLACE), must also be ranked low in Hawaiian since violations of it are quite frequent due to the small consonant inventory.

2.3.2 Frog → poloka

Frog (see Table 3) has a complex onset and a coda consonant, both of which are not allowed in Hawaiian, hence I add the constraints NOCOMPLEXONSET and NOCODA. To ensure that they are resolved by inserting additional vowels and not by deleting consonants, a MAX(IO) constraint and the two syllable constraints have to be placed above a DEP(IO) constraint. ID(CONSONANTAL) must be ranked above all four so that the offensive syllable is not improved by replacing some of the consonants with vowels. To force [f] to become [p] and not [m] or [k], we can extend ID(+STOP) to ID(OBSTRUENT) and ID(+LINGUAL) to

fɹɔg	*FOREIGN	ID(CONSONANTAL)	MAX(IO)	NoCOMPLEXONSET	NoCODA	DEP(IO)	ID(OBSTRUENT)	ID(+LIQUID)	ID(ARTICULATOR)	ID(BACKNESS)	ID(HEIGHT)	ID(PLACE)
a. fɹɔg	4*!			*	*							
b. plok				*!	*						*	*
c. piou		**!					*	*			*	*
d. po.lok					*!	*					*	*
e. po			*!								*	
f. mo.lo.ka						**	*!				*	*
g. po.no.ka						**		*!			*	*
h. ko.lo.ka						**			*!		*	*
i. po.la.ka						**				*!		*
j. po.lu.ka						**					**!	*
☞ k. pi.lo.ki						**					*	*
☞ l. po.lo.ka						**					*	*

Table 3: Tableau for English *frog* adapted into Hawaiian as *poloka*.

ID(ARTICULATOR). Also, we need ID(+LIQUID) to make sure that [ɹ] becomes [l].

Note that our constraints do not make any prediction about the nature of the epenthetic vowels. Hence, *piloki* is just as optimal as *poloka*, and the real winner is *pVlokV* (where *V* is any vowel). Since the inserted vowels in actual Hawaiian loanwords vary greatly and do not seem to underlie consistent rules (Parker Jones 2009), we cannot define constraints to fixate them either, so this is as precise as we can get.

2.3.3 Kangaroo → kanakalū

The transformation from *kangaroo* to *kanakalū* is mostly covered by the previous constraints, as can be seen in Table 4. In addition, we require ID(+NASAL) to prevent [ŋ] from becoming [l] instead of [n], and ID(LENGTH) to preserve the final long vowel. Again, the epenthetic vowel is unconstrained and the real winner is *kanVkalū*.

These 14 constraints should be sufficient to correctly predict the adaption of the three example loanwords.

kaŋ.gə.lu:	*FOREIGN	ID(CONSONANTAL)	MAX(IO)	NoCOMPLEXONSET	NoCODA	DEP(IO)	ID(OBSTRUENT)	ID(+LIQUID)	ID(+NASAL)	ID(ARTICULATOR)	ID(LENGTH)	ID(BACKNESS)	ID(HEIGHT)	ID(PLACE)
a. kaŋ.gə.lu:	4*!				*									
b. ka.la.ka.lu:						*			*!				*	*
c. ka.na.ka.lu						*					*!		*	*
d. ka.na:ka.lu						*					*!		*	*
☞ e. ka.ni.ka.lu:						*							*	*
☞ f. ka.na.ka.lu:						*							*	*

Table 4: Tableau for English *kangaroo* adapted into Hawaiian as *kanakalū*.

3 Architecture

My program is completely written in Python 3. For the finite state transducers, it uses the Python API of the Helsinki Finite State Toolkit (HFST), an open source wrapper around most of the modern FST implementations (Koskenniemi and Yli-Jyrä 2008). HFST is available for Windows, Linux and Mac; however, the Windows bindings are only available for 32-bit Python, which limits the amount of RAM the program can use to 2 GB. For large and complex tableaux, it is therefore recommended to use a Linux or Mac machine.

In this section, I am going to explain the individual modules of my program, all while constructing an example tableau for Hawaiian using the described classes and methods. In section 3.1, I explain how to refer to phonological features in constraints. Section 3.2 introduces the candidate generator and its various settings and convenience functions. The constraint types and how to define them is treated in section 3.3. Finally, in section 3.4, the previously established generator and constraints will be used to build a tableau and investigate the three example loanwords.

3.1 Phonological representation

For reasons of readability and simplicity, my program works directly on phonetically transcribed strings. However, IPA symbols such as [e] or [ɟ] do not have any inherent information about the features of the sound they represent. Hence, constraints such as ID(PLACE) or *+VOICE would require the user to explicitly spell the affected phonemes out (e.g. *b,d,g,...for the latter). Since

	IPA:	a	e	i	o	u	:
	X-SAMPA:	a	e	i	o	U	:
height	high	-	-	+	-	+	0
height	low	+	-	-	-	-	0
backness	front	-	+	+	-	-	0
backness	back	-	-	-	+	+	0
	round	-	-	-	+	+	0
	tense	0	+	+	+	-	0
	long	-	-	-	-	-	+

Table 5: Excerpt of `phon_symbols.tsv` (a, e, i, o, U) and `phon_diacritics.tsv` (:) for the vowel features.

this is quite inconvenient, the module `phonemes.py` contains a representation of a `PhonemeInventory` which can spell out our features for us.

3.1.1 The feature tables

The `PhonemeInventory` class relies on two feature tables stored in `phon_symbols.tsv` and `phon_diacritics.tsv`. Table 5 shows part of these tables. An example table with all features can be found as Table 6 in the appendix. `phon_symbols.tsv` contains the base characters with their associated phonetic feature matrices. Each character has a value for each feature, which can be either +, - or 0 (not applicable). `phon_diacritics.tsv` then contains diacritics which can be combined with the base characters (like ʷ for labialization or : for vowel/consonant length). `PhonemeInventory` will interpret these additively and flip all features of the accompanying base character to the non-0 values of the diacritic.

The features and feature values stores in the tables are loosely based on the system described by Hayes (2009). Some features that are non-obvious for a user not very familiar with the underlying phonetic theory such as [DELAYED RELEASE] or [STRIDENT] have been left out, while other intuitive features such as [STOP] or [VELAR] were added. These features do by no means aim to be scientifically sound and should be seen as a convenient starting point for intuitive constraint declaration. The advanced phonetician can easily edit and extend both tables to new phonemes and features.

3.1.2 Usage

The `PhonemeInventory` may be instantiated without any arguments. In this case, it will contain all possible combinations of phonemes and diacritics listed in its two tables. However, this is far more than we need. Usually, a loanword will be composed of the phonemes of the source and recipient language. Thus, we will restrict our `PhonemeInventory` for English-to-Hawaiian loanwords to the sounds that do actually occur in English and Hawaiian:

```
alph_en = {'A','A:','Q','{','V','e','E','@','i','i:','I',
          '1','0','0:','u','u:','U','m','n','N','p','b','t',
          'd','k','g','t)S','d)Z','f','v','T','D','s','z','S',
          'Z','h','l','r\\','j','w'}
alph_hw = {'a','e','i','o','u','a:','e:','i:','o:','u:',
          'p','k','m','n','l','w','h','?'}
phono = PhonemeInventory(phonemes=alph_en|alph_hw)
```

By default, `PhonemeInventory` uses X-SAMPA transcription (Wells 1995), but it also supports IPA (add another argument `representation=PhoneticAlphabet.IPA` for this). For ease of typing, I am going to write X-SAMPA in the code snippets.

Now that we have created our phoneme inventory, we can use it to directly extract phonemes with certain features:

```
phono.get_phonemes("+labial")
> {'f', 'b', 'p', 'v', 'w', 'm'}
phono.get_phonemes(["+syllabic","+round"])
> {'0:', 'o:', '0', 'u:', 'o', 'A:', 'A', 'U', 'u'}
```

You might have noticed that the excerpt in Table 5 has an extra column labeled **height** for the features **high** and **low**, and **backness** for the features **front** and **back**. These are feature bundles that can be used to conveniently refer to a group of related features. Other feature bundles are e.g. **place2** and **manner2**¹. We can access them using the `get_feature_bundles` method:

```
phono.get_feature_bundles("high", filtr="+long")
> [{'0:', 'A:', 'e:', 'a:', 'o:'},
   {'u:', 'i:'}]
phono.get_feature_bundles("height", filtr="+long")
> [{'0:', 'o:', 'u:', 'e:', 'i:'},
```

¹Have a look at Table 6 in the appendix to see all available bundles and which features they encode.

```
{'A:', 'a:'},  
{'O:', 'A:', 'e:', 'a:', 'o:'},  
{'u:', 'i:'}]
```

Asking for `high` returns a set with all `-high` phonemes and another with all `+high` phonemes. Similarly, `height` returns `-low`, `+low`, `-high` and `+high` sets. In this example, I have restricted the output to long vowels using the `filtr` option. Without `filtr`, `get_feature_bundles` would have sorted all vowels into the bundles.

As we will see in section 3.3, these phoneme sets and feature bundles can be fed directly to our constraints, allowing for very intuitive constraint declaration. The `PhonemeInventory` will also come in handy in the next step, namely for the generation of candidates, especially if we want to set reasonable syllable boundaries.

3.2 Generating candidates

The `gen.py` module contains the `Generator` and `Syllabifier` classes. The `Generator` is the most important part of the program, since it is responsible for generating the infinite candidate set. It accepts any input string (composed of the phonemes specified in our `PhonemeInventory`) and is able to manipulate it via substitutions, deletions and insertions to create a infinite number of candidates. Most of these will be nonsensical, but it is the task of the linguist to create a set of constraints that will quickly rule these out and only leave the desired winner(s) to survive.

3.2.1 String format

If we were to only apply markedness constraints, the `Generator` could just directly mutate the input string. However, faithfulness constraints need information about the input characters as well, so a number of special symbols is inserted into the string during candidate generation to differentiate between input and output. They will be removed again before printing the final winner(s), but it is often useful to ask the tableau for intermediate output and look at the raw strings to understand where something unexpected happens. This is how the candidate *kula* [kula] for the input *school* [sku:l] would look like (whitespaces inserted for readability):

```
# . >s<- >k<k , >u:<u , . >l<l , >-<a , . #
```

All original input symbols are preceded by `>`, while all output symbols are preceded by `<`. `-` stands for ‘no symbol’, and therefore marks deletion (as in `>s<-`) or insertion (as in `>-<a`). Word boundaries are marked by `#`. If you have syllabification enabled, you will also find syllable boundaries (`.`) and nucleus boundaries (`,`) in your string.

3.2.2 The Generator

The sole generator currently implemented in `gen.py` is `MorphGen`. The only argument it requires is a `PhonemeInventory` object to retrieve the usable phonemes from:

```
gen = MorphGen(phono)
```

However, it also has a range of optional arguments to extend or restrict its functionality.

Often, a recipient language only uses its native phonemes in loanwords and does not borrow additional phonemes from the donor language. Since this is the case in Hawaiian, we can spare ourselves the `*FOREIGN` constraint and restrict `MorphGen` to only use Hawaiian phonemes in the output by setting the parameter `out_alph` to a new Hawaiian-only `PhonemeInventory`:

```
hw_phono = PhonemeInventory(phonemes=alph_hw)
gen = MorphGen(phono, out_alph=hw_phono)
```

We can also get a finite candidate set (e.g. for debugging purposes) by specifying a maximum number of insertions (parameter `max_ins`) or we can switch off insertions altogether (parameter `allow_ins`). Similarly, we can disallow deletions (parameter `allow_del`), so that we do not need to impose a maximality constraint later. However, Hawaiian uses both insertions and deletions to adapt loanwords, so we will not make use of these functions here.

Finally, we can specify a `Syllabifier` with the `syllabifier` parameter.

3.2.3 The Syllabifier

The `gen.py` module comes with two implementations of the `Syllabifier` class. A `RandomSyllabifier` will insert syllable and nucleus boundaries randomly and the user has to make sure it generates reasonable syllables via constraints.

It can be instantiated without any arguments. Alternatively, we can create a `NuclearSyllabifier` which ensures that each syllable contains exactly one nucleus, i.e. one +syllabic phoneme. Its constructor minimally requires a `PhonemeInventory` to extract the nuclei from:

```
syl = NuclearSyllabifier(hw_phono)
```

We could also have given it `phono`, but since the syllabifier works on the output characters which we have restricted to Hawaiian phonemes, the additional English phonemes in `phono` are redundant.

Similar to `MorphGen`, our `NuclearSyllabifier` can be further specialized to make some basic constraints unnecessary. The parameter `fill_onset` may be set to `True` to avoid C.V syllable boundaries, i.e. to push a coda consonant into an empty adjacent onset. Also, we can enable a `sonority_filter`. This will make sure that syllable boundaries are set according to the sonority hierarchy and that no syllables violating this hierarchy are generated (removing candidates like [akl.wa], leaving only [ak.lwa] and [a.klwa]). These two settings are thought to remove only unpronounceable candidates and should thus be compatible with the ‘unrestricted candidate set’ principle, but the same effect can of course be achieved via constraints.

Since Hawaiian has no codas or consonant clusters, none of them are necessary in our example. If we needed them, however, we could declare our syllabifier as follows:

```
syl = NuclearSyllabifier(hw_phono, fill_onset=True,
                        sonority_filter=True)
```

Now we can construct our generator with a syllabifier:

```
gen = MorphGen(phono, out_alph=hw_phono, syllabifier=syl)
```

3.3 Defining constraints

Now that we have created a generator, we need some constraints to narrow our infinite candidate set down to a winner. The constraint classes are contained in `constraints.py`. It defines two main types of constraints: Markedness and faithfulness constraints. There is a number of other predefined constraint classes, which however all go back to these two constraint types.

3.3.1 Markedness Constraints

The `MarkednessConstraint` class represents a categorical markedness constraint in the sense that it assigns a violation mark to an offensive structure no matter where in the string it occurs (Karttunen 2006). Constraints like `*ɹ` or `*(+STOP,+VOICE)` will simply mark all [ɹ]s or voiced stops in a string as a violation. Hawaiian does not allow either, so we could declare these two constraints as following:

```
no_r = MarkednessConstraint(phono.get_phonemes(["+rhotic"]))
no_voiced_stops = MarkednessConstraint(
    phono.get_phonemes(["+voice", "+stop"]))
```

For `no_r`, it might be more convenient to refer to the phoneme [ɹ] (X-SAMPA `r\`) directly, since it is only a single symbol. There are two ways to do this:

```
no_r = MarkednessConstraint(["r\\"])
no_r = MarkednessConstraint("[r%\\"])
```

We can write out the list that `phono.get_phonemes()` returns for `+rhotic` directly (`["r\\"]`) or we can write out the regular expression which the constraint will create from this list directly (`"[r%\\"]`)². Note that a string will always be interpreted as an XFST regular expression (Beesley and Karttunen 2003), while a set or list of strings will always be interpreted as a collection of phonemes. If you decide to hand it a regular expression, remember to escape special characters properly or you might get an error. The safe way is to specify your phonemes in a list, because then the constraint will handle the special characters itself.

Sometimes, a marked structure might not be a single phoneme, but rather a specific sequence of sounds. Hawaiian's syllable structure is too simple to find an example for this, so let us consider German. In its native words, it does not allow a sequence of [s] and a plosive in the onset of a syllable, and this constraint will also sometimes apply to loanwords. Hence, the English word *story* [stɔ:ɹi] might be nativized as [ʃtɔβi] instead of [stɔβi]. We could declare this constraint as follows:

```
no_sC = MarkednessConstraint(
    [["s"], phono.get_phonemes("+stop")])
```

A list of sets/lists of strings is interpreted as a sequences of phonemes, in this case as a sequence of `s` and `+stop` consonants. However, this will also mark the

²The double backslashes are necessary to escape the single backslash in the Python string.

sequence across syllable boundaries (as in *Kasten* [kas.tən]) or in the coda (as in *Last* [last]), where it is perfectly acceptable.

To solve the first problem, we can edit the `scope` parameter of our constraint:

```
no_sC = MarkednessConstraint(
    ["s"], phono.get_phonemes("+stop"),
    scope=Scope.SYLLABLE)
```

We have now told the constraint to only match within a syllable, and not across syllable boundaries. A word like [kas.tən] is no longer violating our constraint. By default, the `Scope` of all constraints is `WORD`.

To allow words like [last], we further need to specify that we only want to match in the onset, or rather on the right side of a syllable boundary. To do this, we can set the `left` context of our violation to the syllable boundary mark. While we are at it, we can also move the stop into the `right` context of the violation, since it not really the sequence '[s] plus stop' that is marked, but rather an [s] in the context of a syllable boundary on the one and a stop on the other side. Hence:

```
no_sC = MarkednessConstraint(
    ["s"], left=["."], right=phono.get_phonemes("+stop"),
    scope=Scope.SYLLABLE)
```

Finally, the constraint can be given a name to make it easier to identify it in a save file (see section 3.4.3).

3.3.2 Faithfulness Constraints

The other main type of constraints next to markedness constraints are faithfulness constraints. These are violated if an output symbol does not share certain properties with the corresponding input symbol. The constraint `ID(+LIQUID)`, for instance, is violated whenever a liquid from the input became a non-liquid in the output. In section 2.3 we discovered that this is one of the constraints we need for Hawaiian. Hence:

```
faith_liquid = FaithfulnessConstraint(
    phono.get_phonemes("+liquid"))
```

Just like markedness constraints, the constructor of `FaithfulnessConstraint` also has the `left`, `right`, `scope` and `name` parameters.

3.3.3 Constraint Bundles

Both of the two constraint types may also be declared as a `ConstraintBundle`, i.e. as a bundle of constraints to be applied simultaneously. This is most useful for faithfulness constraints, as we will often want to punish multiple faithfulness violations at the same time. The constraint `ID(PLACE)`, for example, actually is a bundle of the constraints `ID(+LABIAL)`, `ID(+ALVEOLAR)`, `ID(+VELAR)`, etc. This is where the `get_feature_bundles()` method of our `PhonemeInventory` comes in handy:

```
faith_place = FaithfulnessConstraintBundle(
    phono.get_feature_bundles("place2"))
```

By default, `get_feature_bundles()` does not only return the `[+FEATURE]` phonemes, but also the sets of `[-FEATURE]` phonemes. Because of this, `faith_place` will also execute e.g. `ID(-LABIAL)` and `ID(-VELAR)` and mark all non-labials and non-velars that became labials or velars. A `[p]` that became a `[k]` will therefore receive two violation marks, one for not being faithful to `+labial` and one for not being faithful to `-velar`. Since `faith_place` will consistently give two violation marks to all violations, it does not matter technically, but might be irritating to see. You can ask `get_feature_bundles()` to only return `[+FEATURE]` sets by setting its parameter value to `"+"`:

```
faith_place = FaithfulnessConstraintBundle(
    phono.get_feature_bundles("place2", value="+"))
```

Now, `faith_place` will only apply one violation mark per violation. Note that for other constraint bundles like `ID(HEIGHT)`, it is crucial to evaluate both `[+FEATURE]` and `[-FEATURE]` deviations, since the `-` values are actually meaningful here. Consider a language with only `[o]` (`-high`, `-low`) and `[a]` (`-high`, `+low`) receiving a loanword with an `[u]` (`+high`, `-low`) inside, trying to be faithful to height. If we were to only evaluate faithfulness to `+` values, `[o]` and `[a]` would be equally good (or bad) candidates, because in both cases we would switch from `+high` to `-high`. `[o]`, however, is in fact the closer vowel and better candidate, because it does not change `[u]`'s `-low` feature while `[a]` does, so we want `[a]` to receive two violation marks and `[o]` only one. Hence, we must use the `value` parameter with care.

3.3.4 More predefined constraints

The `constraint.py` module contains a few more predefined constraints, which however all go back to markedness and faithfulness constraints or constraint bundles.

The `DependencyConstraint` punishes insertion of new output phonemes, while the `MaximalityConstraint` punishes deletion of input phonemes. Both can be instantiated without any arguments, in which case they apply to any phoneme, or with a set of phonemes that should not be inserted or deleted.

```
max_io = MaximalityConstraint()
dep_c = DependencyConstraint(
    phono.get_phonemes("+consonantal"))
```

The `AssimilationConstraint` punishes consonants that are dissimilar in a given feature to their left neighbor. `ASSIM(place)`, for example, will mark the sequence `[ntp]` with one violation mark (`n t p*`), `[npt]` with two (`n p* t*`), and `[npm]` with one (`n p* m`).

```
assim_place = AssimilationConstraint(
    phono.get_feature_bundles("place2", value="+"))
```

The `ComplexOnsetConstraint` and `ComplexCodaConstraint` punish onsets and codas of a certain complexity. Hawaiian, for instance, allows one consonant in the onset, so we want to accept the first and mark all subsequent consonants. The coda is always empty, so all consonants should be marked as violations in there. Hence, we want to set the parameter `max_size`, which denotes the maximum number of a given structure that is accepted, to 1 for the onset, but to 0 for the coda:

```
no_complex_onset = ComplexOnsetConstraint(
    phono.get_phonemes(["-syllabic"]), max_size=1)
no_coda = ComplexCodaConstraint(
    phono.get_phonemes(["-syllabic"]), max_size=0)
```

These are actually the default `max_size` values for the two constraint types, so we can also leave the parameter out and just write:

```
no_complex_onset = ComplexOnsetConstraint(
    phono.get_phonemes(["-syllabic"]))
no_coda = ComplexCodaConstraint(
    phono.get_phonemes(["-syllabic"]))
```

The `GradientConstraint` is the superclass of the two syllable constraints and works in the same way. It sets out from a `border`, which by default is the boundary of its `Scope` (e.g. from the word boundary `#` for `Scope.WORD`). Apart from the `Scopes` we already encountered in the section about markedness constraints, `WORD` and `SYLLABLE`, there are also `ONSET`, `NUCLEUS` and `CODA` which can be useful here. Starting from this border, it can accept and mark phonemes going to the left or right, depending on whether the parameter `left_oriented` is set to `True` or `False`. Hence, we could also write our syllable constraints for Hawaiian as:

```
no_complex_onset = GradientConstraint(  
    phono.get_phonemes(["-syllabic"]), scope=Scope.ONSET,  
    left_oriented=True, max_size=1)  
no_coda = GradientConstraint(  
    phono.get_phonemes(["-syllabic"]), scope=Scope.CODA,  
    left_oriented=False, max_size=0)
```

3.4 Using the tableau

The actual optimality theory tableau is defined in the `tableau.py` module. The `Tableau` class is responsible for putting together the generator and the constraints into a single small and quick finite state transducer and evaluate the winner(s) for any input based on this transducer. It can also provide intermediate sets of surviving candidates for debugging and may be saved to and loaded from files for repeated usage.

3.4.1 Building the tableau

The constructor of the `Tableau` class only requires a generator as an argument:

```
tab = Tableau(gen)
```

Additionally, it has another optional argument, `penal_method`, which sets the method by which losing candidates are removed from the pool of survivors after each constraint. By default, the system employs the "matching" approach of Gerdemann and Noord (2000). Alternatively, Karttunen's (1998; 2006) "counting" approach may be used. Since "matching" is more precise and produces much smaller transducers, it is usually advised to stay with the default, so we will not change it here either.

Now our Tableau has to be filled with constraints. First, we define all of the necessary constraints for our examples that we identified in our manual tableaux in section 2.3:

```
no_foreign = MarkednessConstraint(alph_en-alph_hw)
faith_cv = FaithfulnessConstraintBundle(
    phono.get_feature_bundles("consonantal"))
max_io = MaximalityConstraint()
no_complex_onset = ComplexOnsetConstraint(
    phono.get_phonemes(["-syllabic"]))
no_coda = ComplexCodaConstraint(
    phono.get_phonemes(["-syllabic"]))
dep_io = DependencyConstraint()
faith_obstruent = FaithfulnessConstraintBundle(
    phono.get_feature_bundles("sonorant"))
faith_liquid = FaithfulnessConstraint(
    phono.get_phonemes("+liquid"))
faith_nasal = FaithfulnessConstraint(
    phono.get_phonemes("+nasal"))
faith_artic = FaithfulnessConstraintBundle(
    phono.get_feature_bundles(["+labial", "+lingual"]))
faith_length = FaithfulnessConstraintBundle(
    phono.get_feature_bundles("long"))
faith_backness = FaithfulnessConstraintBundle(
    phono.get_feature_bundles("backness"))
faith_height = FaithfulnessConstraintBundle(
    phono.get_feature_bundles("height"))
faith_place = FaithfulnessConstraintBundle(
    phono.get_feature_bundles("place2", value="+"))
```

Since we restricted the generator to output only Hawaiian phonemes, `no_foreign` is not actually needed.

The Tableau can be provided with the constraints one by one:

```
tab.add_constraint(faith_cv)
tab.add_constraint(max_io)
```

Alternatively, we can hand it multiple at once in an ordered list:

```
constraints = [faith_cv, max_io, no_complex_onset, no_coda,
    dep_io, faith_obstruent, faith_liquid, faith_nasal,
    faith_artic, faith_length, faith_backness, faith_height,
    faith_place]
```

```
tab.add_constraints(constraints)
```

Once we have added all the constraints we need, we can tell the `Tableau` to compile the FST:

```
tab.build()
```

This may take up to a few minutes depending on the number and complexity of the generator and the constraints. To get some intermediate feedback from the building process, we can set the `verbosity` parameter of the `build()` method. At 0, it is completely silent. At 1, it will print the index of the constraint it is at and the overall build time once it is done, all of this to just a single line. At 2 and above, it will print the number of states and arcs after applying each constraint as well as the time needed to do this, so you can see which constraints take longer than others and watch the FST grow to see where it might reach a problematic size. By default, `verbosity` is set to 1.

On a Windows 8.1 machine running 32-bit Python, building the English-to-Hawaiian tableau takes 17.33 seconds. The final FST consists of only 24 states and 195 arcs.

3.4.2 Running the tableau

Once the tableau has finished building its FST, we can test the three example loanwords $[le.tə] \rightarrow [le.ka]$, $[fɪŋg] \rightarrow [po.lo.ka]$ and $[kaŋ.gə.ru:] \rightarrow [ka.na.ka.lu:]$. The `run()` method of `Tableau` takes two arguments: The input form, i.e. the English word, and the desired winner, i.e. the Hawaiian loanword.

```
tab.run("lEt@", "le.ka")
> 'le.ka' wins!
tab.run("fr\\Qg", "po.lo.ka")
> Too many winners:
> 'po.lo.ka:'
> 'pe.lo.ki:'
> 'pe.lo.ke'
> 'po.lo.ku'
> 'po.lo.ki'
> 'pe.lo.ki'
> 'pe.lo.ka:'
> 'po.lo.ko:'
> 'pe.lo.ku:'
> 'po.lo.ku:'
```

```

> etc.
tab.run("kaNg@r\\u:", "ka.na.ka.lu:")
> 'ka.na.ka.lu:' wins, but there are also other winners:
> 'ka.no:.ka.lu:'
> 'ka.no.ka.lu:'
> 'ka.ni.ka.lu:'
> 'ka.na:.ka.lu:'
> 'ka.nu.ka.lu:'
> 'ka.ne.ka.lu:'
> 'ka.ni:.ka.lu:'
> 'ka.nu:.ka.lu:'
> 'ka.ne:.ka.lu:'

```

The constraints correctly capture the transformation of *letter* into *leka*. For *frog* and *kangaroo*, there is more than one winner, because our constraints do not define the quality of the epenthetic vowel(s). For *kangaroo*, this is not a problem, because we can see that the nine additional winners returned are all variations of *kanVkalū*. For *frog*, this seems to be the case as well, but we cannot be sure, because `run()` only looks up ten candidates by default. We could increase this number to the 100 possible realizations of *pVlokV*:

```
tab.run("fr\\Qg", "po.lo.ka", n=100)
```

However, this will print out all of the 100 winners and we will have to investigate them manually to decide whether they follow the desired pattern. To avoid this, we can formulate the desired winner as a Python regular expression and set the option `regex` of `run()` to `True`:

```

pVlokV = "p(a|e|i|o|u):?\\.lo\\.k(a|e|i|o|u):?"
tab.run("fr\\Qg", pVlokV, n=100, regex=True)
> /p(a|e|i|o|u):?\\.lo\\.k(a|e|i|o|u):?/ wins!

```

The constraints we have established in section 2.3 seem to hold even against an infinite candidate set. However, we do not really see what they do and whether they actually assign the correct violation marks.

To investigate this, we can use the method `trace_candidates()`. Instead of a single desired winner, it takes a list of candidates which we would like to trace along the constraints. For each constraint, it prints the number of survivors and fatalities, and the survivors and fatalities themselves with the violation marks assigned by the constraint, separated into traced and untraced candidates. As long as the candidate set is still infinite, i.e. before a dependency constraint has punished unbounded insertions, this is of course not possible. Afterwards, the

number of remaining candidates might still be large. Hence, just like the `run()` method, `trace_candidates()` has a parameter `n` which defines the number of candidates it should try to look up. If there are more, it will not print any survivors and fatalities either, so try to set this number as high as possible. By default, it is as low as 10. With high `n`, the speed of `trace_candidates()` can be significantly increased by suppressing the printing of untraced candidates (`show_traced_only=True`).

Let us check whether our finite state tableau correctly replicates the manual tableau from section 2.3 for *frog*:

```
tab.trace_candidates("fr\\Qg",
    ["plok", "po.lok", "po", "po.no.ka", "mo.lo.ka",
     "ko.lo.ka", "po.la.ka", "po.lu.ka", "po.lo.ka"],
    n=100000, show_traced_only=True)
> 0
> Candidates: (infinite)
```

As expected, the candidate set is infinitely large. This holds until the fifth constraint, `DEP(IO)`:

```
> 5
> Fatalities: (infinite)
> Survivors: > 100000
> 6
> Fatalities: > 100000
> Survivors: 64000
>   traced: {'#.po.lo.ka.#', '#.po.lu.ka.#',
>           '#.po.la.ka.#', '#.ko.lo.ka.#', '#.po.no.ka.#'}
```

The survivors of `DEP(IO)` are more than the 100,000 candidates we requested, so they are not printed. After applying `ID(OBSTRUENT)`, we can see that we are left with the expected five traced candidates, while the others have been correctly eliminated before.

```
> 7
> Fatalities: 48000
>   traced: {'#.po.n*o.ka.#'}
> Survivors: 16000
>   traced: {'#.po.lu.ka.#', '#.po.la.ka.#',
>           '#.ko.lo.ka.#', '#.po.lo.ka.#'}
```

The next constraint, `ID(+LIQUID)`, correctly marks the `[n]` in `[po.no.ka]` as a violation, since this used to be a liquid in the input, and removes the candidate

while the others survive.

```
> 12
> Fatalities: 100
>   traced: {'#.po.lu**.ka.#'}
> Survivors: 100
>   traced: {'#.po.lo*.ka.#'}
```

At the penultimate constraint, $ID([HEIGHT])$, only two of the traced candidates, [po.lu.ka] and the desired winner [po.lo.ka], are left. Neither is a perfect candidate: The vowel [ɒ] is lower than both [o] and [u]. However, [o] is closer to it than [u], so [po.lu.ka] (and all other pV.lu.kV candidates) receives two violation marks, while [po.lo.ka] (pV.lo.kV) is only marked once and therefore survives.

```
> 13
> Fatalities: 0
> Survivors: 100
>   traced: {'#.p*o.lo.ka.#'}
```

Finally, $ID(ID(PLACE))$ gives one violation mark to all remaining candidates and therefore does not contribute anything to the result. Hence, it could also be omitted.

In case the tableau gives seemingly wrong results, it can be beneficial to not only view the output itself, but the complete aligned input-output string formatted as described in section 3.2.1. The `trace_candidates()` method can display these raw representations with the option `verbose` set to `True`. Now the output for e.g. the 12th constraint will look like this:

```
> 12
> Fatalities: 100
>   traced: {'#.>f<p,>-<o,.>r\\<l,>Q<u**,.>g<k,>-<a,.#'}
```

3.4.3 Saving and loading

A Tableau object can be saved to a file and restored in a later session. This is particularly useful for Tableaux with larger FSTs that took long to compile. This way, they do not have to be recompiled when we only wish to test a new input. Saving works very straightforward with the `save()` method which takes as an argument the file name:

```
tab.save("en2hw")
```

This will create two files with the specified path: One which contains the settings of the constraints, `en2hw.tableau`, and one which contains the built FSTs, `en2hw.hfst`. Investigating `en2hw.tableau` might also be useful for debugging, because it spells out the regular expressions representing each constraint. `en2hw.hfst` stores the transducers in binary HFST format, and may thus also be loaded by HFST in any other application. It contains a stack of multiple transducers: First the generator, then the intermediate FSTs used by `trace_candidates()` (one after inserting the violation marks and one after removing the losers for each constraint), and finally the complete FST representing the whole tableau.

To restore the `Tableau`, simply use the class method `load()` with the path to the `.tableau` and `.hfst` file:

```
tab = Tableau.load("en2hw.tableau", "en2hw.hfst")
```

4 Conclusion

In this paper, I have presented a new implementation of Optimality Theory. Because it is based on finite state transducers, it is able to extract the winner(s) out of an infinite set of candidates, and can therefore greatly aid the linguist to overcome the limits of manual tableau construction. The ability to generate and handle a truly unrestricted candidate set is unique amongst other OT software, which mostly concentrates on finding the optimal constraint ranking. By combining these two kinds of OT programs, one for ranking the constraints and one for testing these constraints against an unbounded candidate set, it should be possible to create a complete and thorough OT analysis of the (loanword) phonology of a language.

5 Eigenständigkeitserklärung

Ich versichere, dass ich die Arbeit ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Stellen und Personen, welche mich bei der Vorbereitung und Anfertigung der Abhandlung unterstützten, wurden genannt und Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Tübingen, den 18. Oktober 2017

Thora Daneyko

References

- Beesley, Kenneth R. and Lauri Karttunen (2003). *Finite state morphology*. Center for the Study of Language and Information.
- Biró, Tamás (2010). *OTKit. Tools for Optimality Theory*. URL: <http://www.biroth.hu/OTKit/> (visited on 10/13/2017).
- Boersma, Paul and David Weenink (2017). *Praat: doing phonetics by computer. Version 6.0.34*. URL: <http://www.praat.org/> (visited on 10/13/2017).
- Elbert, Samuel H. and Mary Kawena Pukui (2001). *Hawaiian Grammar*. University of Hawai'i Press.
- Gerdemann, Dale and Gertjan van Noord (2000). "Approximation and Exactness in Finite State Optimality Theory". In: *Finite-State Phonology*, p. 34.
- Hayes, Bruce (2009). *Introductory phonology*. John Wiley & Sons. URL: <http://linguistics.ucla.edu/people/hayes/IP/>.
- Hayes, Bruce, Bruce Tesar, and Kie Zuraw (2013). *OTSoft 2.5*. URL: <http://linguistics.ucla.edu/people/hayes/otsoft/> (visited on 10/13/2017).
- Helsinki Finite-State Transducer Technology (HFST)* (2017). URL: <http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/> (visited on 06/29/2017).
- Karttunen, Lauri (1998). "The proper treatment of optimality in computational phonology: plenary talk". In: *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*. Association for Computational Linguistics, pp. 1–12.
- Karttunen, Lauri (2006). "The insufficiency of paper-and-pencil linguistics: the case of Finnish prosody". In: *Intelligent linguistic architectures: Variations on themes by Ronald M. Kaplan*, pp. 287–300.
- Koskenniemi, Kimmo and Anssi Yli-Jyrä (2008). "CLARIN and Free Open Source Finite-State Tools." In: *FSMNLP*, pp. 3–13.
- Parker Jones, 'Ōiwi (2009). "Loanwords in Hawaiian". In: *Loanwords in the World's Languages: a comparative handbook*. URL: http://www.academia.edu/3769002/Loanwords_in_Hawaiian.
- Prince, Alan and Paul Smolensky (2008). *Optimality Theory: Constraint interaction in generative grammar*. John Wiley & Sons.
- Staubs, Robert et al. (2010). *OT-Help 2.0*. URL: <http://people.umass.edu/othelp/> (visited on 10/13/2017).
- Wells, John C. (1995). *Computer-coding the IPA: a proposed extension of SAMPA*.
-

Wiktionary (2017). *Wiktionary, the free dictionary*. URL: <https://en.wiktionary.org/> (visited on 10/08/2017).

A Feature table

	IPA	a	e	i	o	u	b	ç	d	j	k	ʎ	ŋ	r	ʁ	?
	X-SAMPA	a	e	i	o	U	b	C	d`	j	k	5	N	r	t)S	?
	consonantal	-	-	-	-	-	+	+	+	-	+	+	+	+	+	+
	syllabic	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-
m2	nasal	0	0	0	0	0	-	-	-	-	-	-	+	-	-	-
m2	stop	0	0	0	0	0	+	-	+	-	+	-	-	-	-	+
m2	fricative	0	0	0	0	0	-	+	-	-	-	-	-	-	-	-
m2	approximant	0	0	0	0	0	-	-	-	+	-	+	-	-	-	-
m2	trill	0	0	0	0	0	-	-	-	-	-	-	-	+	-	-
m2	flap	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
m2	affricate	0	0	0	0	0	-	-	-	-	-	-	-	-	+	-
m1	continuant	0	0	0	0	0	-	+	-	+	-	+	+	+	+	-
m1	sonorant	+	+	+	+	+	-	-	-	+	-	+	+	+	-	-
	lateral	0	0	0	0	0	-	-	-	-	-	+	-	-	-	-
	rhotic	0	0	0	0	0	-	-	-	-	-	-	-	+	-	-
m1	liquid	0	0	0	0	0	-	-	-	-	-	+	-	+	-	-
p1	labial	0	0	0	0	0	+	-	-	-	-	-	-	-	-	-
p2	bilabial	0	0	0	0	0	+	-	-	-	-	-	-	-	-	-
p2	labiodental	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
	lingual	0	0	0	0	0	-	+	+	+	+	+	+	+	+	-
p1	coronal	0	0	0	0	0	-	-	+	-	-	+	-	+	+	-
p2	dental	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
p2	alveolar	0	0	0	0	0	-	-	-	-	-	+	-	+	-	-
p2	post-alveolar	0	0	0	0	0	-	-	-	-	-	-	-	-	+	-
p2	retroflex	0	0	0	0	0	-	-	+	-	-	-	-	-	-	-
p1	dorsal	0	0	0	0	0	-	+	-	+	+	+	+	-	-	-
p2	palatal	0	0	0	0	0	-	+	-	+	-	-	-	-	-	-
p2	velar	0	0	0	0	0	-	-	-	-	+	+	+	-	-	-
p2	uvular	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
p1	laryngeal	0	0	0	0	0	-	-	-	-	-	-	-	-	-	+
p2	pharyngeal	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
p2	epiglottal	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
p2	glottal	0	0	0	0	0	-	-	-	-	-	-	-	-	-	+
ph	egressive	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
ph	voice	+	+	+	+	+	+	-	+	+	-	+	+	+	-	-
ph	nasalization	-	-	-	-	-	-	-	-	-	-	-	+	-	-	-
ph	aspiration	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ph	ejective	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
ph	click	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
he	high	-	-	+	-	+	0	0	0	0	0	0	0	0	0	0
he	low	+	-	-	-	-	0	0	0	0	0	0	0	0	0	0
ba	front	-	+	+	-	-	0	0	0	0	0	0	0	0	0	0
ba	back	-	-	-	+	+	0	0	0	0	0	0	0	0	0	0
	round	-	-	-	+	+	0	0	0	0	0	0	0	0	0	0
	tense	0	+	+	+	-	0	0	0	0	0	0	0	0	0	0
	long	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 6: A list of all the features in `phon_symbols.tsv` and `phon_diacritics.tsv` with a few example sounds and their values.