Dealing with the infinite candidate set

A finite state implementation of optimality theory

Thora Daneyko

July 18, 2017

"The Insufficiency of Paper-and-Pencil Linguistics" (Karttunen 2006)

- Introduction and ranking of OT constraints is based on the candidates
- New candidates can always disprove your theory
- 'Manual method' cannot account for all possible candidates
- But computers can!

→ Finite state transducers to generate and deal with infinitely many candidates

Structure

1. Introduction to finite state transducers
2. Implementation
   - Generating candidates
   - Applying constraints
3. Demo
4. Outlook

Introduction to finite state transducers (FSTs)

Finite state transducer (FST)

- A machine that accepts a set of strings and maps them to some output, while rejecting the rest
- Consists of states (accepting vs. non-accepting) and transitions (with characters)
- Reads each character of a string and takes the matching transition to the next state
- Accepts the string if the string can be read completely and the machine ends up in an accepting state

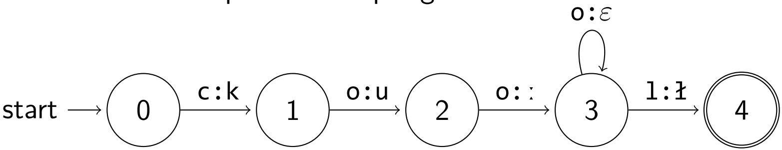

Figure: An FST transforming the strings ‘cool’, ‘coool’, ‘cooool’, etc. into ‘kuːł’

Why are FSTs useful?

- Finite state transducers are very powerful

- Can deal with infinitely many possible inputs
- Comparatively small (5 states, 5 transitions)
- Usually only need to read the input once

Composition

- Output of FST 1 becomes input of FST 2
- The resulting FST accepts the same as FST 1 and outputs the same as FST 2

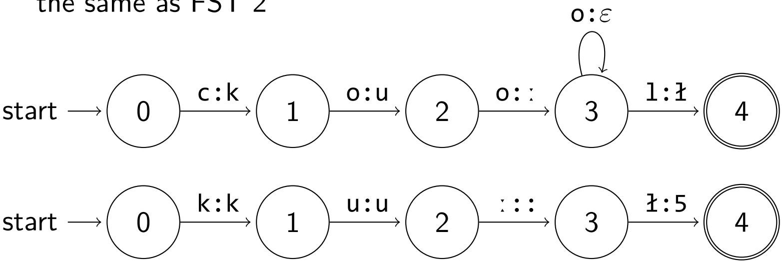

Figure: One FST transcribing coo+l into IPA kuːɫ, another one converting the IPA string to X-SAMPA ku:5.

Composition

- Output of FST 1 becomes input of FST 2
- The resulting FST accepts the same as FST 1 and outputs the same as FST 2

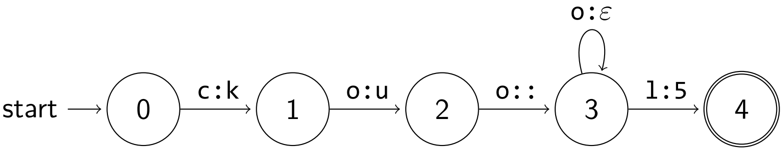

Figure: An FST directly transcribing coo+l into X-SAMPA ku:5.

Composition

- Output of FST 1 becomes input of FST 2
- The resulting FST accepts the same as FST 1 and outputs the same as FST 2

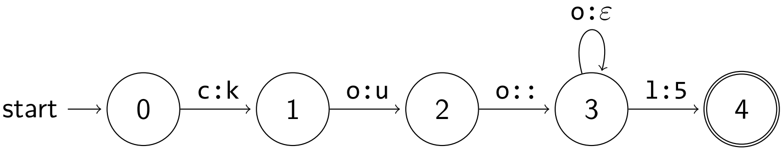

Figure: An FST directly transcribing coo+l into X-SAMPA ku:5.

- Lenient composition (Karttunen 1998): Composition only applies if it results in some output

A finite state implementation of optimality theory

Finite state implementation of OT

- Idea (Karttunen 2006): Construct FSTs for
  1. generating candidates,
  2. marking violations of constraints,
  3. and eliminating candidates with violations,
- and compose all of these FSTs into a single one that does all in one step.

Materials

- Helsinki Finite State Toolkit (HFST)
  - Open source FST implementation
  - Has lenient composition
- Programmed in Python using HFST's Python API

Components

- Tableau
  - Generates candidates
  - Stores and applies constraints
- Constraint
  - Marks violations in the input string
  - Penalizes marked candidates and possibly removes them

The GEN function: Requirements

- Create all possible combinations of the phonemes in the alphabet
- Must retain a representation of the input to evaluate faithfulness constraints
- Syllabification of output to apply onset and coda constraints

The GEN function: Implementation

Example: (whitespaces inserted for readability)
Input: `ku:5`

The GEN function: Implementation

Example: (whitespaces inserted for readability)

`# >k >u: >5 #`

1. Insert input symbols (>) and word boundaries (#)

The GEN function: Implementation

Example: (whitespaces inserted for readability)

`# >k<k >u:<a >5 #`

1. Insert input symbols (>) and word boundaries (#)
2. Manipulate input
   - **Substitutes** each phoneme in the input for each phoneme in the alphabet (output symbol <)

The GEN function: Implementation

Example: (whitespaces inserted for readability)

```
# >k<k >u:<a >5<- #
```

1. Insert input symbols (>) and word boundaries (#)
2. Manipulate input
   - **Substitutes** each phoneme in the input for each phoneme in the alphabet (output symbol <)
   - May **delete** any number of phonemes in the input (deletion mark -)

The GEN function: Implementation

Example: (whitespaces inserted for readability)

`# >k<k >u:<a >5<- +n +a #`

1. Insert input symbols (>) and word boundaries (#)
2. Manipulate input
   - **Substitutes** each phoneme in the input for each phoneme in the alphabet (output symbol <)
   - May **delete** any number of phonemes in the input (deletion mark -)
   - May **insert** an infinite number of any phoneme in the alphabet at any position in the input (insertion mark +)

$\rightarrow$ Infinitely many outputs

The GEN function: Implementation

Example: (whitespaces inserted for readability)

`# >k<k >u:<a . >5<- +n +a #`

1. Insert input symbols (>) and word boundaries (#)
2. Manipulate input
   - **Substitutes** each phoneme in the input for each phoneme in the alphabet (output symbol <)
   - May **delete** any number of phonemes in the input (deletion mark -)
   - May **insert** an infinite number of any phoneme in the alphabet at any position in the input (insertion mark +)

   → Infinitely many outputs
3. Add syllable boundaries (.)

The GEN function: Implementation

Example: (whitespaces inserted for readability)

`# >k<k >u:<a . >5<- +n +a #` → candidate: `ka.na`

1. Insert input symbols (>) and word boundaries (#)
2. Manipulate input
   - **Substitutes** each phoneme in the input for each phoneme in the alphabet (output symbol <)
   - May **delete** any number of phonemes in the input (deletion mark -)
   - May **insert** an infinite number of any phoneme in the alphabet at any position in the input (insertion mark +)

   → Infinitely many outputs
3. Add syllable boundaries (.)

Applying constraints (Karttunen 1998, 2006)

1. Marking violations inside the string with *
2. Removing candidates with violation marks using lenient composition
   - Define upper bound *n* of violations to be eliminated
   - First remove strings with *n* or more violations,
   - then remove strings with $n - 1$ violations, …
   - then remove strings with 1 violation.
   - Lenient composition: Elimination stops as soon as it would delete all candidates!

Demo

Example languages

- Donor language
  - 5 vowels: `a e i o u`
  - 9 consonants: `p t k b d g m n r`
  - Syllable structure: (C)(r)V(C)
  - Some native words: `degor`, `mitgra`, `bratak`
- Recipient language
  - 3 vowels: `a i u`
  - 7 consonants: `p t k m n N l`
  - Syllable structure: (C)V(S)
  - Some native words: `taka`, `miNul`, `kumpil`

Paper-and-Pencil: degor → tikul

de.gor	*R	FAITH(LIQUID)	*[+STOP,+VOICE]	*[-LOW,-HIGH]	FAITH(MANNER)	FAITH(PLACE)	FAITH(BACKNESS)
a. de.gor	*!		**	**			
b. de.gol			**!	**	*		
c. de.gon		*!	**	**	*		
d. ne.Nol				**!	***		
e. ni.Nul					***!		
f. pi.tul					*	**!	
g. ta.kal					*		**!
☞ h. ti.kul					*		

Paper-and-Pencil: bratak → palataka

bra.tak	Max(IO)	NoComplOnset	NoCoda([-son])	Dep(IO)	*R	Faith(liquid)	*[+stop,+voice]	*[-low,-high]	Faith(manner)	Faith(place)	Faith(Backness)
a. bra.tak		*!	*		*		*				
b. pla.tak		*!	*						*		
c. pa.tak	*!		*						*		
d. pa.la.tak			*!	*					*		
☞ e. pa.la.ta.ka				**					*		

Demo time!

Paper-and-Pencil: bratak → palataN?

bra.tak	MAX(IO)	NOCOMPLONSET	NOCODA([-SON])	DEP(IO)	*R	FAITH(LIQUID)	*[+STOP,+VOICE]	*[-LOW,-HIGH]	FAITH(MANNER)	FAITH(PLACE)	FAITH(BACKNESS)
a. bra.tak		*!	*		*		*				
b. pla.tak		*!	*						*		
c. pa.tak	*!		*						*		
d. pa.la.tak			*!	*					*		
e. pa.la.ta.ka				**!					*		
☞ f. pa.la.taN				*					**		

Paper-and-Pencil: bratak → palataka!

bra.tak	Max(IO)	NoComplOnset	NoCoda([-son])	*R	Faith(liquid)	*[+stop,+voice]	*[-low,-high]	Faith(manner)	Dep(IO)	Faith(place)	Faith(Backness)
a. bra.tak		*!	*	*		*					
b. pla.tak		*!	*					*			
c. pa.tak	*!		*					*			
d. pa.la.tak			*!					*	*		
☞ e. pa.la.ta.ka								*	**		
f. pa.la.taN								**!	*		

Outlook

To do

- Immediate:
  - Find the factors that inflate the transducers when reordering constraints
  - Simplify faithfulness constraints (generally inflating FST size)
  - Improve syllabification
- Long-term:
  - Test on real languages (with many constraints)
  - Combine with constraint ordering algorithm
  - Automatically generate constraints

References

Beesley, Kenneth R. and Lauri Karttunen (2003). *Finite state morphology*. Center for the Study of Language and Information.

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.