Talk at UMass-Amherst this Friday by Tesar & Prince on learning phonotactic distributions. I’ll probably go; let me know if you want a ride.

Successive filtering—Eisner (“Efficient Generation in Optimality Theory”)  
See also Ellison (1994)

(1) Definitions

- **Con**: universal constraint set
- **Grammar**: vector (ordered $n$-tuple), $(C_1, C_2, C_3 ... C_n)$, of (all?) members of **Con**
- **Repns**: universal set of all possible output forms (with containment)
- **Gen**: function from input to subset of **Repns**
- **Constraint**: function from output to natural number (because of containment, output has sufficient information for constraint evaluation)  
  \[ C_i : \text{Repns} \rightarrow \{0, 1, 2, ...\} \quad (C_i \in \text{Con}) \]
- **Filter**: result of applying a constraint to a set of outputs and retaining only those that violate it minimally  
  \[ \text{Filter}(\text{Set}) = \{ R \in \text{Set} : C_i(R) \text{ is minimal}\} \]
- **Winning candidate**: result of successive filter application (Filter$_n$)  
  \[ \text{Filter}_n(... \text{Filter}_2(\text{Filter}_1(\text{Gen}(\text{input})))) \]
- **Recursive definition of winner**  
  \[ S_0 = \text{Gen}(\text{input}) \subseteq \text{Repns} \]
  \[ S_{i+1} = \text{Filter}_{i+1}(S_i) \subseteq S_i \]
  If there are $n$ constraints, winner is $S_n$
(2) Input representations as FSAs

\[ /\text{anba}/ \]

<table>
<thead>
<tr>
<th>V</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lab</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can be written as (infinite) set of strings, where each ‘symbol’ is a combination of tier elements (a vertical slice).

Actually, this is too deterministic for input tiers.
- Because intervals on input tiers end up with zero duration (deletion), the symbols that don’t refer to any edges are not obligatory.
- And because edges on the input tier can be pushed apart (epenthesis), the | edges can get broken apart.
- Similarly, edges that line up on the V and C tiers can get pushed apart.

So, the string below represents a looser specification of the input than the tier drawing does:

\[
\langle -,-,-,\rangle^* \langle [-,,-,\rangle \langle +,-,-,\rangle \langle -,-,-,\rangle \langle [-,,-,\rangle or \langle [,,-,\rangle \langle -,-,+,+\rangle^* \langle [-,,-,\rangle, \rangle \langle -,-,-,\rangle^* \langle [-,,-,\rangle or \langle [,,-,\rangle \langle -,-,+,+\rangle^* \langle [-,,-,\rangle, \rangle \langle -,-,-,\rangle^* \langle [-,,-,\rangle or \langle [,,-,\rangle \langle -,-,+,+\rangle^* \langle [-,,-,\rangle, \rangle
\]

We can express this with an FSA in the obvious way:

(3) Output representations as FSAs

\textbf{Repns}, the set of all well-formed output representations, looks like this if there is just one tier:

p. 2
If there are two tiers, you *intersect* (see how below) two of these FSAs. The intersection of two FSAs, \( A \) and \( B \), is the FSA that accepts all and only the strings accepted by both \( A \) and \( B \).

Keep intersecting for each tier.

This gives you the set of all outputs.

(4) **Candidate sets as FSAs**

To get \( \text{Gen}(\text{input}) \), intersect the FSA for the input with the FSA for \( \text{Repns} \).

This gives you the set of all outputs that contain the input. This includes squishing input intervals down to zero and pushing input \(|s|\)s apart.

(5) **Constraints as edge-weighted FSAs**

An edge-weighted FSA is an FSA where each arc (a.k.a. "edge") has a value, in this case 0 (not bold) or 1 (bold).

The idea is that every time an output form violates the constraint, it crosses a bold arc. Constraint FSAs accept all strings.

They don’t explicitly output the number of violations.
When you intersect a constraint with an FSA representing a set of outputs, you get a new
FSA that accepts just those outputs, but has weighted arcs for constraint violations.

You can then use Dijkstra’s BestPaths algorithm (see how below) to create a new FSA
that accepts only a subset of those outputs—those that pass through the smallest number
of weighted arcs.

(6) Intersection of FSAs
Note first that \( A \cap B = (A^c \cup B^c)^c \) (where \( A^c \) means ‘complement of \( A \) ’)

- How do you take the complement of a FSA?
  Reverse the accept and non-accept states.
  If your FSA is missing some arcs (as ours are), add in those arcs going to a
  non-accept state and turn it into an accept state.

- How do you take the union of two FSAs?
  - Take the cross-product of the states, so that each state in the new FSA
    corresponds to one state from each of the original FSAs.
  - The new start state is the one that corresponds to the two original start states.
  - The new accept states are those that correspond to an original start state in
    either of the two machines (doesn’t have to be both).
  - If you’re in state \((p_i, r_j)\) and reading symbol \(x\), then go to state \((p_k, r_l)\), where \(p_k\)
    is the state that the first machine goes to if it’s in \(p_i\) and reading \(x\), and \(r_l\) is the
    state that the second machine goes to if it’s in \(r_j\) and reading \(x\).

Take the complement of the union of complements, and you’ve got the intersection.

Let’s try the example from (3).

(7) BestPaths (Dijkstra 1959)
Also known as the single-source shortest paths algorithm

This is a dynamic programming algorithm! Instead of calculating all possible paths, you
find the best partial path to \(q\), the best partial path to \(r\), etc.
How to do it:

Let $S$ be the set of states whose shortest path from the start state (‘cost’) has been determined.
Let $V-S$ be the set of other states

1. Set $S$ to empty; set ‘cost’ of start node to 0, all other nodes to infinity; set each node’s predecessor as empty
2. While there are nodes in $V-S$
   a. Find the node in $V-S$ with the lowest current cost; call it $p$. Add $p$ to $S$.
   b. For each arc leaving $p$, if the cost of $p$ plus the cost of the arc is lower than the cost already in the state $q$ at the end of the arc, make that the new cost of $q$ and make $p$ the predecessor of $q$.

To find the best paths from the start node to any node $r$, trace $r$’s predecessors back to the start node.

Good web page, with animation:

(8) The generation algorithm
To repeat from above,
$S_0 = \text{Gen(input)} \subseteq \text{Repns}$
$S_{i+1} = \text{Filter}_{i+1}(S_i) \subseteq S_i$
If there are $n$ constraints, winner is $S_n$

We now know how to do this. Let’s try it with an example from a homework.

(9) Computability
The number of states for $\text{Repns}$ is $2^{|\text{Tiers}|}$. This grows really fast!

For 9 output tiers, Eisner says that intersecting $\text{Repns}$ with the input FSA yields about 5000 states and 500,000 to 775,000 arcs.

Eisner suggests intersecting each tier rule from $\text{Repns}$ only when a constraint mentions that tier. This cuts the number of states by about 90% and the number of arcs by about 99%, and the generation time by about 99%.

Still, even with this ‘trick’, the amount of time needed for generation is more than polynomial on the number of tiers.
(10) Proof of NP-hardness by reduction from Hamiltonian path

A Hamiltonian path is a path that goes through each node in a directed graph exactly once. (This is like the travelling salesman problem, except that the arcs aren’t weighted.)

Determining whether a Hamiltonian path exists between two nodes appears to take more than polynomial time: The problem is a member of $NP$, the class of problems that are solvable in nondeterministic polynomial time; moreover, it’s at least as hard as any other problem in $NP$ (any problem in $NP$ can be reduced to it by a deterministic polynomial-time algorithm), so it’s $NP$-complete and thus believed not to be solvable in deterministic polynomial time.

If you show that solving some problem requires solving the Hamiltonian path problem, then your problem is at least as hard as any of the problems in $NP$ ($NP$-hard), and maybe even harder. This is considered bad news.

Eisner shows that, in some cases, generation requires solving the Hamiltonian path problem. Therefore, generation is at least as hard as the Hamiltonian path problem:

Say the set of tiers is the set of $n$ nodes in some directed graph $G$, plus Stem and $S$.

And say we have the constraints

1. $v \rightarrow S$ \quad (for all nodes $v$ in $G$)
2. $v \rightarrow \_S$ \quad (for all nodes $v$ in $G$)
3. $\text{Stem} \rightarrow v$ \quad (for all nodes $v$ in $G$)
4. $\text{Stem} \perp S$
5. $\_u \perp v$ \quad (for all distinct nodes $u$, $v$ in $G$ that aren’t connected by an arc)
6. $\_S \rightarrow S$

If the input is just $[\text{Stem}]$, then constraints 1-4 leave us with the candidates whose output tiers look like

$$
\begin{array}{llllll}
S & [ & | & | & ... & | ] \\
v_1 & [ & ] \\
v_2 & [ & ] \\
v_3 & [ & ] \\
... \\
v_n & [ & ]
\end{array}
$$

where the $v$ interiors can be in any order.

(Constraints 1-3 are satisfied; 4 is violated $n$ times.)

It’s now up to the remaining two constraints to decide the order of the $v$ interiors.
Constraint 5 says that a sequence of abutting interiors like \([v_1][v_2][v_3]\) is allowed only if it corresponds to a path in \(G\).
The candidates that survive constraint 5 include those in which not all of the \(v_i\) interiors are directly abutting.

But constraint 6 wants all the \(v_i\) interiors to be directly abutting (no gaps between \([S]s\)).

So the best way to satisfy 5 and 6, if there happens to be a path in \(G\) that goes through each vertex exactly once, is to find that path—this is the Hamilton path problem.

If the grammar is known in advance, the path can be figured out in advance.

Is this situation something that happens in human language?

It would be interesting to explore the computational properties of OTP generation in attested grammars versus in unattested but definable grammars.

(11) Factored automata
This is Eisner’s idea for delaying tier intersection and thus making generation faster.

The idea is to represent a candidate set not as a big FSA, but as a collection of small FSAs (a factored automaton), each mentioning just a few tiers.

For \(S_0\), the factors would be \(input\) and each tier rule.

Each constraint is then intersected with only the relevant factors.
Constraints that don’t mention input tiers don’t have to be intersected with \(input\), so the time needed for evaluating those (markedness) constraints is independent of the length of the input.

How do you find the best paths through the intersection of a factored automaton and a constraint? You want to intersect the constraint with the ‘projection’ of the factored automaton onto the relevant tiers:

1. Label the constraint-relevant tiers 0 and all other tiers with distinct positive numbers.
2. Partition the factors according to the highest-numbered tier they mention.
3. If no factors mention any positive-number tiers, then you have to do the intersection of all the factors.
4. Otherwise, take the intersection of all the factors in the highest \((k)\)-numbered member of the partition. Remove mentions of tier \(k\) from the arcs in the resulting FSA. Clean up (minimize and make deterministic) this FSA and assign it to the appropriate member of the partition.
5. Decrement $k$ and repeat from 3. You still have to do as many intersection operations as there are factors (minus one), but you’re pruning as you go.


Next time: Variable and probabilistic grammars

For next time
• Reading: Anttila (1997), and review Boersma & Hayes