Search strategies
We can think of the generation task as a search problem: how do we find the output candidate that best satisfies the constraint ranking?

(1) Depth-first searching

Benefit: you don’t have to remember too many forks in the road at once.
Drawback: if the right solution is A, you have to try every other solution till you get to it.

(2) Breadth-first searching

Benefit: if you know that the problem is structured appropriately, you may be able to write off a whole branch after probing just one of its leaves.
Dynamic programming

Benefit: if the problem is structured appropriately, you can write off a whole branch without even getting down to the leaf level.

Example from Tesar: calculating the best route between X and Y

Chart parsing

<table>
<thead>
<tr>
<th>shortest route from X to...</th>
<th>A:</th>
<th>D:</th>
<th>G:</th>
<th>Y:</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:</td>
<td>E:</td>
<td>H:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C:</td>
<td>F:</td>
<td>I:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DP is used in spell-checkers, speech recognition, machine translation, parsing....

Tesar & Smolensky (2000), ch. 8—roughly the same as Tesar (1995)

Presents an example of how dynamic programming can choose the optimal candidate without considering an infinite set: basic syllabification.

Assumes containment

Segments aren’t actually deleted or inserted.

Deletion (“underparsing”) = failing to attach segment to any prosodic structure
Violates PARSE

Insertion (“overparsing”) = blank prosodic structure (featural values are “filled in” by the “phonetic component”)
Violates FILL
(6) **Context-free grammar for syllable structure**

\[
S \to e \mid oO \mid nN \\
O \to nN \\
N \to e \mid dD \mid oO \mid nN \\
D \to e \mid oO \mid nN
\]

\(o = \text{onset, } n = \text{nucleus, } d = \text{coda}\)

We could add, for the terminals:

\(o \to C\)
\(n \to V\)
\(d \to C\)

Candidates are derived by constructing a tree for the input string of Vs and Cs. At any point in the derivation, we can be in one of four states: the nonterminal character is \(S\), \(O\), \(N\), or \(D\).

Tree construction is guided by Operations Set (p. 119, p. 128), which tells you what your options for the next step are.

(7) **Grammar is actually regular**

Although the syllable language is defined with a context-free grammar, it could also be described with just a regular expression:

\[\left( (o\mid\varepsilon)n(d\mid\varepsilon) \right)^*\]

(8) **Algorithm**

1. Set cell \([S, i_{0}]\) to blank: no structure, no violations
2. For each row \(i_{0}\) (proceeding from left to right)
3.   { For each row \(X\)
4.     { Fill \([X, i_{j}]\) with the result of the underparsing operation for \(X\)
5.     For each parsing operation for \(X\)
6.       { If the result of applying the operation is more harmonic than what’s currently in the cell, replace it.
7.     }
8.   }
9. { For each cell \([X, i_{i}]\)
10.   { For each overparsing operation for \(X\)
11.     { If the result of applying the operation is more harmonic than what’s currently in the cell, replace it.
12.   }
13. }
14. }

p. 3
The cycle (one syllable) limits insertion
Maximum number of passes through #7 is three, because of the cycle:
... is the same as ..., but with three extra violations of FILL.

This means we don’t really have to consider an infinite number of candidates here.

Example
Let’s step through how the algorithm would work for CCV (Tesar shows VC).

Complexity
The maximum number of steps for each column is equal to:

\[(1 + \text{Under}_S) + (1 + \text{Under}_O) + (1 + \text{Under}_N) + (1 + \text{Under}_D) + 3(\overline{\text{Over}_S} + \overline{\text{Over}_O} + \overline{\text{Over}_N} + \overline{\text{Over}_D})\]

This is a constant (how big depends on the number of operations) so the maximum number of steps is “linear in the size of the input” (the length of the input string times a constant number). The maximum amount of cells to keep in memory is 8 (you can ignore row \(i\) once you’re at \(i+2\), because none of the operations require looking back more than one column).

Simplifying assumptions
Locality: A constraint is local if it can be evaluated by looking only at two adjacent positions.
The most nonlocal constraint is ONSET (still local under above definition), which requires looking back from a nucleus to the previous position to see if it’s an onset. Can you think of some constraints that are truly nonlocal?

Cycles: assumes that a whole blank syllable will never be inserted. What if you had a language requiring minimally disyllabic words?

Ways to extend this
• Include feet in the prosodic structure, and do stress assignment.
• Remove the assumption that nuclei must be vowels, and add constraints like \(*P/X\) and \(*M/X\).
• Allow complex onsets and complex codas to be considered (though constraints may rule them out).
• Under what circumstances is the locality assumption not met? Can the procedure still be made to work?
• Metathesis? Coalescence?
(14) Example (not meant to be realistic) of strictly context-free grammar

\[
\begin{align*}
S &\Rightarrow F \mid e \\
F &\Rightarrow Y \mid YF \\
Y &\Rightarrow P \mid MFM \\
M &\Rightarrow m \\
P &\Rightarrow p \\
\end{align*}
\]

\(m = \text{margin}, \ p = \text{peak}\)

Generates the candidate space \((C^aVCA^d)^*\)

(15) Locality

Defined now as a constraint that requires looking only at a ‘local region’, where local region =

- non-terminal plus its child non-terminals
  - or
- non-terminal plus a child terminal and the input segment (if any) filling that terminal

(16) Dynamic programming table

Now it’s 3-dimensional.

In addition to a category for each nonterminal (S, O, N, D above), we have a category for each prefix consisting of nonterminals of a string on the right side of a rule (S, F, Y, M, P, MF).

For each category, there’s a two-dimensional table of starting-input-segment by ending-input-segment:

<table>
<thead>
<tr>
<th>String length</th>
<th>3</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>(i_1)</td>
<td>(i_2)</td>
</tr>
</tbody>
</table>

starting segment

Notation for cells: \([X, a, c]\), where \(X\) is the category, \(a\) is the index of the starting segment, and \(c\) is the index of the ending segment.

Operations compete to fill each cell with the most optimal partial parse, one row at a time (starting from the top rows).
(17) **Operations set**

As before, derives from the grammar.

- Rules whose right side is one terminal (P => p):
  
  May fill only cells for string-length 1 (top row).
  
  parse: cell [P,a,a] gets P(p(i\_a))
  
  overparse: cell [P,0] gets P(p(\_))

  where [X,0] is the optimal unfilled structure for category X

  (all the [X,0]s can be calculated in advance, since they don’t depend on the input)

- Rules whose right side non-terminals (Y => MFM):
  
  parse: cell [MF,a,c] gets treated as ordered pair [M,a,b],[F,b+1,c]
  
  overparse: cell [MF,a,c] gets treated as ordered pair [M,a,c],[F,0]

- Underparsing doesn’t use the production rules (just add an underparsed segment to the left or to the right of the string in any cell).

(18) **Locality**

Harmony of a cell entry is the marks inherited from cells used to make it up, plus marks created by the production rule used to create the new cell entry.

Constraints must be local = new violations can be determined just by looking at the production rule being used.

And if constraints are local, we have a cycle again: adding overparsed material to a category X until you eventually get back to having category X can never increase harmony (and if you have Fill constraints, it will decrease harmony).

Since there are a finite number of categories, for any structure there’s a known limit to how much overparsed material you can profitably add to it.

(19) **Example**

Let’s try CCV again, with the ‘silly’ CFG above.

**Next time: Generation in OTP**

**For next time**

- **Exercise:** give a DPT (just like the one on T&S’s page 123, showing structure, constraint violations, and which cell it comes from) for the sequence CCVV, using the ranking
  
  ONSET >> NOCODA >> FILL\_ONS >> PARSE >> FILL\_NUC.

  Also give a standard tableau (the winner should be the same in both!!). You probably want to do the tableau first so that you can tell if something’s going wrong in your DPT.

- **Reading:** Eisner (2000), Efficient generation in Primitive Optimality Theory