### Natural Language Processing
### CMSC 723 (spring, 2001)

April 11, 2001

- Review of Dynamic Programming
- Dotted Rule Notation
- Earley Algorithm
- Complexity of Earley
- Key to Efficiency

---

### Dynamic Programming and Parsing

Use a table of size $n + 1$. The table entries sit in the gaps between the words:

- Completed constituents
- In-progress constituents
- Predicted constituents

---

### Dynamic Programming

We want an algorithm that fills a table with solutions to subproblems that:

- Does not do repeated work
- Does top-down search with bottom-up filtering (sort of)
- Solves the left-recursion problem
- Solves an exponential problem in $O(n^3)$ time.

### States

$S \rightarrow \bullet \ VP$

$NP \rightarrow \text{Det} \bullet \text{Nominal}$

$VP \rightarrow \text{V} \ NP \bullet$
States cont.

Keep track of:

- What word it is currently processing.
- Where it is in the processing of the current rule.
- Where it should return to when done w/ current rule.

States cont.

Parse: “Book that flight.”

\[
\begin{align*}
S & \rightarrow \bullet \ VP \ [0,0] \\
NP & \rightarrow \text{Det} \bullet \text{Nominal} \ [1,2] \\
VP & \rightarrow V \ NP \bullet \ [0,3]
\end{align*}
\]

Each State \( s_j \): \(<\text{dotted rule}>, [<\text{back pointer}>, <\text{current posn}>]\)

Graphical States

[Figure 10.15]

Success

Start \( \rightarrow \alpha \bullet \ [\text{nil, n}] \)
**Parsing**

- New predicted states are based on existing table entries that predict a certain constituent at that spot.
- New in-progress states are created by updating older states.
- New complete states are created when the dot moves to the end.

**Toward an Efficient Parsing Algorithm: Earley (1970)**

Top-down parser with bottom-up filtering.

- Ambiguity
- Left recursion
- Repeated parsing of subtrees

What is the key to addressing these issues?

**Memoization and Dynamic Programming**

- Use tables to keep track of previously solved sub-problems.
- Dynamic programming algorithms: oriented around systematically filling these tables.
- Memoization: achieves the same results but allows the algorithm to do so more efficiently.

**States and State Sets**

Dotted Rule: State \( s_i \) is represented as \(<\text{dotted rule}_i, [\text{<back pointer>_i}, \text{<current posn>_i}]>\)

Define: State Set \( S_j \) to be a collection of states \( s_i \) with the same \(<\text{<current position>_i}>\).
Earley Algorithm

[Figure 10.16]

---

Basic operations of the Earley Algorithm

- Predictor
- Completer
- Scanner

---

Earley Algorithm (easier to read!)

- Add initial state in dotted form: \( S_0 \)
  \[ \text{Start} \rightarrow \bullet S_i [n_i, 0] \]

- Apply predict/completer until no more states are added (closure under predict/completer).

- For each word \( W_i \) \( (i = 1, \ldots, n) \), build state set \( S_i \) (Main Loop):
  - Apply \text{scan} to \( S_{i-1} \)
  - Close state set \( i \) under predict/completer
  - If state set \( i \) is empty, reject; else, continue

- If state set \( n \) includes state \( \text{Start} \rightarrow S \bullet [n_i, n] \) then accept; else reject.

---

SCAN Operation

\( S_j: \quad A \rightarrow \alpha \bullet B \beta, [i, j] \)

\( S_{j+1}: \quad A \rightarrow \alpha B \bullet \beta, [i, j + 1] \)
PREDICT Operation

\[ S_j: \ A \rightarrow \alpha \bullet B \beta, \ [i,j] \]
\[ S_j: \ B \rightarrow \gamma, \ [i,j] \]

Example

[Figure 10.17a]

COMPLETE Operation

(Much more complicated! Relies heavily on return address.)

\[ S_k: \ B \rightarrow \delta \bullet, \ [j,k] \]
\[ S_k: \ A \rightarrow \alpha \ B \bullet \beta, \ [i,k], \]

where:

\[ S_j: \ A \rightarrow \alpha \bullet B \beta, \ [i,j] \]

Example (continued)

[Figure 10.17b]
Complexity Analysis of Earley

1. How many state sets will there be?

2. How big can the state sets get?

Another Earley Algorithm Example

Grammar: $S \rightarrow NP \ VP, \ NP \rightarrow N, \ VP \rightarrow V \ NP$

Input: I saw Mary

$S_0$ Word: NIL

$S_1$ Word: I (N)

$S_2$ Word: saw (V,N)

$S_3$ Word: Mary (N)

Sentence Accepted.

Analysis of SCAN, PREDICT, COMPLETE

- **Scan:**
  $S_i: A \rightarrow \alpha \bullet B \beta, [i,j]$
  $S_{i+1}: A \rightarrow \alpha B \bullet \beta, [i,j+1]$

- **Predict:**
  $S_i: A \rightarrow \alpha \bullet B \beta, [i,j]$
  $S_j: B \rightarrow \bullet \gamma, [j,j]$

- **Complete:**
  $S_k: B \rightarrow \delta \bullet, [j,k]$
  $S_k: A \rightarrow \alpha B \bullet \beta, [i,k]$
  where:
  $S_k: A \rightarrow \alpha B \beta, [i,j]$
Effect of Ambiguity on Earley Processing Time

How many ways can we complete a phrase of a given rule in a given state?

Example: I saw the man on the hill

```
VP /
  \ saw NP /
    \ / the man PP /
      \ on NP /
        \ the hill
```

\[ VP \rightarrow V \ NP \ \bullet \ \{i,j\} \]
\[ VP \rightarrow V \ NP \ PP \ \bullet \ \{i,k\} \]
\[ S \rightarrow NP \ VP \ \bullet \ \{j\} \ (from \ state \ set \ j) \]
\[ S \rightarrow NP \ VP \ \bullet \ \{m,i\} \ (from \ state \ set \ k) \]

Unambiguous grammar: \( O(n^2) \).

Key to Efficiency for Earley

- Why efficient?
- Other parsers?
- No grammar conversion.
- Additional efficiency measures
- Efficient for unambiguous grammars.

Local Ambiguity

Suppose we're parsing the VP "gave Mary a book" using the following rules:

\[ S \rightarrow VP \]
\[ VP \rightarrow V \]
\[ VP \rightarrow V \ NP \]
\[ VP \rightarrow V \ NP \ PP \]
\[ VP \rightarrow V \ NP \ NP \]
Global Ambiguity

Suppose we’re parsing the VP “I shot an elephant in my pajamas” ...

[Figure 10.11]

Left Recursion

What about parsing the NP “a flight from denver to boston” with the following rules:

NP → NP PP
NP → Det Nominal
NP → ProperNoun

Left Recursion

$A \rightarrow \bullet A B$