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

We want an alorithm 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.

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

States

S → • VP
NP → Det • Nominal
VP → V NP •

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."

S $\rightarrow \bullet$ VP, [0,0]
NP $\rightarrow$ Det $\bullet$ Nominal, [1,2]
VP $\rightarrow$ V NP $\bullet$, [0,3]

Each State $\mathbf{s}_i$: <dotted rule>, [<back pointer>,<current posn>]

Graphical States

[Figure 10.15]

Success

Start $\rightarrow \alpha \bullet$, [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.

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.

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?

States and State Sets

Dotted Rule: **State s$_i$** is represented as <dotted rule>, [<back pointer>, <current posn>]

Define: **State Set S$_j$** to be a collection of states s$_i$ with the same <current position>.

Earley Algorithm

[Figure 10.16]

Earley Algorithm (easier to read!)

- Add initial state in dotted form: S_o
  Start $\rightarrow \bullet$ S, [nil,0]
- Apply predict/complete until no more states are added (closure under predict/complete).
- For each word W_i ($i = 1, \ldots, n$), build state set S_i (Main Loop):
  - Apply scan to S_{i-1}
  - Close state set i under predict/complete
  - If state set i is empty, reject; else, continue
- If state set n includes state Start $\rightarrow$ S $\bullet$, [nil,n] then accept; else reject.

Basic operations of the Earley Algorithm

- Predictor
- Completer
- Scanner

SCAN Operation

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

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

PREDICT Operation

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

S_j: $B \rightarrow \bullet\ \gamma$, $[j,j]$

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

[Figure 10.17a]

Example (continued)

[Figure 10.17b]

Example (continued)

[Figure 10.17c]

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.

Complexity Analysis of Earley

1. How many state sets will there be?

2. How big can the state sets get?

Analysis of SCAN, PREDICT, COMPLETE

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

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

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 → V NP •, $[j,i]$

VP → V NP PP •, $[k,i]$

S → NP VP •, $[l,i]$ (from state set j)

S → NP VP •, $[m,i]$ (from state set k)

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

Effect of Grammar Size on Earley Processing Time

Why is grammar size included?

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→VP
VP→V
VP→V NP
VP→V NP PP
VP→V NP NP

Global Ambiguity

Suppose we're parsing the VP "I shot an elephant in my pajamas" ...

[Figure 10.11]

Left Recursion

$A \rightarrow \bullet\ A\ B$

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