CSE6339 3.0 Introduction to Computational Linguistics
Tuesdays, Thursdays 14:30-16:00 – South Ross 101
Fall Semester, 2011

Earley Parser

Adapted from Wikipedia,
the free encyclopedia (Oct 6, 2011)

The Earley parser is a type of chart parser mainly used for parsing in computational linguistics, named after its inventor, Jay Earley. The algorithm uses dynamic programming[1].

Earley parsers are appealing because they can parse all context-free languages. The Earley parser executes in cubic time ($O(n^3)$, where *n* is the length of the parsed string) in the general case, quadratic time ($O(n^2)$) for unambiguous grammars, and linear time for almost all LR(k) grammars. It performs particularly well when the rules are written left-recursively.

The algorithm

In the following descriptions, α, β, and γ represent any string of terminals/nonterminals (including the empty string), X and Y represent single nonterminals, and *a* represents a terminal symbol.

Earley's algorithm is a top-down dynamic programming algorithm. In the following, we use Earley's dot notation: given a production $X \rightarrow \alpha\beta$, the notation $X \rightarrow \alpha \bullet \beta$ represents a condition in which α has already been parsed and β is expected.

For every input position (which represents a position *between* tokens), the parser generates an ordered *state set*. Each state is a tuple ($X \rightarrow \alpha \bullet \beta$, *i*), consisting of

- the production currently being matched ($X \rightarrow \alpha\ \beta$)
- our current position in that production (represented by the dot)
- the position *i* in the input at which the matching of this production began: the *origin position*

(Earley's original algorithm included a look-ahead in the state; later research showed this to have little practical effect on the parsing efficiency, and it has subsequently been dropped from most implementations.)

The state set at input position *k* is called S(*k*). The parser is seeded with S(0) consisting of only the top-level rule. The parser then iteratively operates in three stages: *prediction*, *scanning*, and *completion*.

- **Prediction**: For every state in S(*k*) of the form ($X \rightarrow \alpha \bullet Y\ \beta$, *j*) (where *j* is the origin position as above), add ($Y \rightarrow \bullet\ \gamma$, *k*) to S(*k*) for every production in the grammar with Y on the left-hand side ($Y \rightarrow \gamma$).

---

[1] In mathematics and computer science, dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), and then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations. This is especially useful when the number of repeating subproblems is exponentially large.

Top-down dynamic programming simply means storing the results of certain calculations, which are later used again since the completed calculation is a sub-problem of a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

- **Scanning**: If *a* is the next symbol in the input stream, for every state in S(*k*) of the form (X → α • *a* β, *j*), add (X → α *a* • β, *j*) to S(*k*+1).
- **Completion**: For every state in S(*k*) of the form (X → γ •, *j*), find states in S(*j*) of the form (Y → α • X β, *i*) and add (Y → α X • β, *i*) to S(*k*).

These steps are repeated until no more states can be added to the set. The set is generally implemented as a queue of states to process (though a given state must appear in one place only), and performing the corresponding operation depending on what kind of state it is

Pseudo code

Adapted from Speech and Language Processing by Daniel Jurafsky and James H. Martin:

```
function EARLEY-PARSE(words, grammar)

  ENQUEUE((γ → •S, [0,0]), chart[0])
  for i ← from 0 to LENGTH(words) do
    for each state in chart[i] do
      if INCOMPLETE?(state) then
        if NEXT-CAT(state) is a nonterminal then
          PREDICTOR(state)        // non-terminal
        else do
          SCANNER(state)          // terminal
      else do
        COMPLETER(state)
    end
  end
  return chart

procedure PREDICTOR((A → α•B, [i, j])),
  for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
    ENQUEUE((B → •γ, [j, j]), chart[ j])
  end

procedure SCANNER((A → α•B, [i, j])),
  if B ⊂ PARTS-OF-SPEECH(word[j]) then
    ENQUEUE((B → word[j], [j, j + 1]), chart[j + 1])
  end

procedure COMPLETER((B → γ•, [ j, k])),
  for each (A → α•Bβ, [i, j]) in chart([j]) do
    ENQUEUE((A → αB•β, [i, k]), chart[k])
  End
```

Example

Consider the following simple grammar for arithmetic expressions:

```
P → S       # the start rule
S → S + M | M
M → M * T | T
T → number
```

With the input:

2 + 3 * 4

This is the sequence of state sets:

```
(state no.) Production (Origin) # Comment
----------------------------------------
== S(0): • 2 + 3 * 4 ==
(1)  P → • S          (0)    # start rule
(2)  S → • S + M      (0)    # predict from (1)
(3)  S → • M          (0)    # predict from (1)
(4)  M → • M * T      (0)    # predict from (3)
(5)  M → • T          (0)    # predict from (3)
(6)  T → • number     (0)    # predict from (5)

== S(1): 2 • + 3 * 4 ==
(1)  T → number •     (0)    # scan from S(0)(6)
(2)  M → T •          (0)    # complete from S(0)(5)
(3)  M → M • * T      (0)    # complete from S(0)(4)
(4)  S → M •          (0)    # complete from S(0)(3)
(5)  S → S • + M      (0)    # complete from S(0)(2)
(6)  P → S •          (0)    # complete from S(0)(1)

== S(2): 2 + • 3 * 4 ==
(1)  S → S + • M      (0)    # scan from S(1)(5)
(2)  M → • M * T      (2)    # predict from (1)
(3)  M → • T          (2)    # predict from (1)
(4)  T → • number     (2)    # predict from (3)

== S(3): 2 + 3 • * 4 ==
(1)  T → number •     (2)    # scan from S(2)(4)
(2)  M → T •          (2)    # complete from S(2)(3)
(3)  M → M • * T      (2)    # complete from S(2)(2)
(4)  S → S + M •      (0)    # complete from S(2)(1)
(5)  S → S • + M      (0)    # complete from S(0)(2)
(6)  P → S •          (0)    # complete from S(0)(1)

== S(4): 2 + 3 * • 4 ==
(1)  M → M * • T      (2)    # scan from S(3)(3)
(2)  T → • number     (4)    # predict from (1)

== S(5): 2 + 3 * 4 • ==
(1)  T → number •     (4)    # scan from S(4)(2)
(2)  M → M * T •      (2)    # complete from S(4)(1)
(3)  M → M • * T      (2)    # complete from S(2)(2)
(4)  S → S + M •      (0)    # complete from S(2)(1)
(5)  S → S • + M      (0)    # complete from S(0)(2)
(6)  P → S •          (0)    # complete from S(0)(1)
```

The state (P → S •, 0) represents a completed parse. This state also appears in S(3) and S(1), which are complete sentences.

References

- J. Earley, "An efficient context-free parsing algorithm", *Communications of the Association for Computing Machinery*, **13**:2:94-102, 1970.
- J. Leo, A general context-free parsing algorithm running in linear time on every LR(*k*) grammar without using look-ahead, *Theoretical Computer Science*, **82**:165-176, 1991.
- J. Aycock and R.N. Horspool. Practical Earley Parsing. *The Computer Journal*, **45**:6:620-630, 2002.
- Daniel M. Roberts Earley Parsing for Context-Sensitive Grammars