22–23 Earley's Algorithm

Drew McDermott drew.mcdermott@yale.edu

2015-11-20, 11-30, 2016-10-28 — Yale CS470/570

The Earley Algorithm[1] can be explained from two different angles:

1. As a simulator of a naive parsing algorithm, that by simplifying its states avoids the consequences of its naiveté.

2. As an application of *dynamic programming*.

Let's start from the second angle. This odd term[2] means "Storing results so they don't have to be recomputed many times." It invests in some memory space to save a lot of computation time.

One of the most obvious applications of dynamic programming is *memoization*. Suppose you want to compute the i'th Fibonacci number, that is, the i'th value in the sequence 1, 1, 2, 3, 5, 8, ..., where the next number is the sum of the previous two. Let's skip past the obvious ways to do this and look at a recursive solution:

```
def fib(i: Int): BigInt = {
  if (i < 3) 1
  else fib(i-1) + fib(i-2)
}
```

The only nonobvious detail is the return type `BigInt`, which we need because the values of `fib` quickly become bigger than will fit in the space allocated to an ordinary `Int`.[3]

Aside from the need to deal in `BigInt`s, there is another problem with `fib`: The same computation will be repeated many times. The call to `fib(i-1)` will require computing `fib(i-2)`, so all that work will be repeated. It may sound like a factor of 2 is not important, but that factor of two will be replicated approximately `i` times, so the net increase in time will be $2^{\texttt{i}}$.

[1] Named for its inventor, Jay Earley,

[2] The word "programming" occurs here in the same sense as in "linear programming"; it means "resource allocation," because these techniques were applied first to that sort of problem. "Dynamic" will be explained later.

[3] Special procedures are needed to add and subtract `BigInt`s, but Scala's slick implicit-conversion machinery hides the need for all that. The same program in Java could not avoid expressions like `BigInt.valueOf(i).plus(BigInt.valueOf(1))`.

One solution is to cache values of `fib` so that when they're needed again they can just be retrieved from the cache. That leads to the following version[4]:

```
val memTab = new HashMap[Int,BigInt]()

def memFib(i: Int): BigInt = {
  val res: BigInt = {
    if (i < 3) 1
    else memTab.get(i) match {
      case Some(v) => v
      case None => {
        val v = memFib(i-1) + memFib(i-2)
        memTab += (i -> v)
        v
      }
    }
  }
  res
}
```

The ClassesV2 file `Resources > Lecture Notes > Memoize.scala` contains an application that runs the two versions of `fib`.[5] For numbers beyond about 40, `fib` takes so long that `Memoize` will refuse to try the computation. Meanwhile, `memFib` runs in about the same amount of time it takes to print the results out.[6]

If this is such a terrific idea, how come it doesn't solve all such problems? Here's a typical case where it comes close, but falls short. Suppose the function to be memoized generates a `Stream` of answers, that is, a "lazy list" of indefinite length whose elements are computed as they are examined. One issue with such lists is that later elements may depend on the values of earlier elements. We can appeal to the Fibonacci numbers again, but this time we will think about a `Stream` of all the Fibonaccis:

[4]This is an interesting example of nonlocal side effects that do not disturb referential transparency. There is no way to detect that `memFib` is any different from `fib` except the shorter runtime.

[5]I believe the term *memoize* was coined by Donald Michie; it means to rewrite a function so to consult a table of previously computed results.

[6]The `Memoize` application runs a third version, `memoizedFib`. This is the result of an attempt to abstract the idea of memoization by using a general `memoize` function. It's surprisingly hard to write and use this abstraction. Check out the code to see how `memoizedFib` is defined.

```
val fibStream: Stream[BigInt] =
  1 #:: 1 #::
    (for {
      p <- fibStream zip fibStream.tail
     } yield {
       p match {
         case (f1, f2) => (f1 + f2)
       }
     })
```

The idea is to give the first two values, then evaluate `fibStream zip FibStream.tail`. If we line the streams up:

```
        1 #::  1 #::  ...
1 #::   1 #::  ...
```

The third number is the sum of the first and second elements:

```
        1 #::  1 #::  2 #::  ...
1 #::   1 #::  2 #::  ...
```

Now that that third number is available, the fourth number can be computed, as soon as someone asks for it:

```
        1 #::  1 #::  2 #::  #::  3 #::  ...
1 #::   1 #::  2 #::  3 #::  ...
```

Clearly, the memoization technique we used before won't help here, not in the sense that we could compute the entire stream and store it the next time someone asks. But this is exactly the problem we have with the parsing techniques used in section 9.3, and many others. Suppose we have a left-recursive rule such as

```
VP -> VP NP
```

(We haven't dealt with how to propagate the requirement that VP involves a transitive verb, but imagine we have.) If there is no other way to generate VPs, the rule is useless at best, a ticket to a stack overflow at worst. But if there is a rule like

```
VP -> V
```

then we can generate, say, VP "Kiss," then we can use the first rule to generate the VP "Kiss a frog." It's a lot like the `FibStream` example.

We can still use the memoization idea, but we have to modify it. There are two ways of thinking about the problem:

1. *Backward, on-demand memoization:* We detect when a goal (find VPs at a certain position) occurs as a subgoal of itself and cut off further search. However, the goal stays alive. Every time an answer to the original goal is generated, (in this case, verb phrases at a certain position) , we feed the answers generated at the supergoal back to the suspended subgoal, which resumes and returns the answers as if it had found them itself. So if the supergoal VP produces "kisses" by other means, it's returned as an answer to the subgoal, which then produces the answer "kisses a frog." Sometimes it's impossible to identify exactly which goals are potential suppliers of useful answers to a suspended subgoal. But if you keep track of a superset of the useful goals and answers, at least you'll be sure of generating every possible answer at least once. This idea is called *tabling* in the logic-programming literature.

2. *Forward, anticipatory, memoization:* Alternatively, if one can break a computation into stages, such that results in stage k depend only on results computed at stages $i < k$, and there are a finite number of stages, one can compute the stages one by one. Often it's impossible to be precise about what inputs will be necessary to compute stage k when working on earlier stages, but if a *superset* of the desired results is stored, then the computation of later stages can rely on the necessary data being available..[7]

Earley's algorithm is an example of *anticipatory memoization.* Suppose a sentence has N words. The algorithm computes a set of phrases ending at position i, for i between 1 and N.[8] We will assume throughout this note that the grammar has no rules of the form P -> ϵ, where P is some nonterminal. If it did, we would have to think about phrases ending at position 0 as well. The set associated with position i may be described as "partial phrases of a certain grammatical category that might be completable (if words to the right of i fall the right way) as a constituent of the S we are looking for." The set ending with word i can be computed from earlier stages, and, if there is a parse of the entire sentence, it will be found at stage N.

[7]This is the form in which dynamic programming was first invented, in the 1950s, by Richard Bellman. The first applications involved optimizing a temporally staged process, which is where the word "dynamic" came from. I think.

[8]The best way to describe subsequences of a sequence is almost always in terms of *positions* in the sequence. A sequence of N objects has $N + 1$ positions, the first lying to the left of all the elements, the last lying to the right of all the elements, and all the others lying between two elements. A position is most naturally designated by the number of elements to its left, from 0 to N.

What's computed at stage i is a set of *parser states,* which are abstract versions of the states of a naive recursive-descent parser. Each is a tuple $\langle j, i, R, p, t\rangle$ that encapsulates an attempt to use grammar rule $R = s_0 \texttt{->} s_1 \ldots s_{n_R}$ to find a phrase of category s_0. (The s_i are nonterminals of a context-free grammar.) So far p contiguous phrases of types s_1, s_2, ..., s_p, covering positions j through i have been found, so if an s_{p+1}, ..., s_{n_R} can be found covering positions through i through k, then we will have found an s_0 covering positions j through k. (The t slot is a general-purpose "result" slot. The result value for phrases is a function of the result values of its subphrases. In what follows we'll assume that t is a list of p syntactic trees, unless $p = n_R$, when t is a list containing a syntactic tree covered by nonterminal s_0, spanning positions j through i, and containing subphrases covered by s_1 through s_{n_R}.)

The parser states resemble the states of a recursive-descent parser up to a point, but certain information has been left out. Specifically, there is no representation of the goals above the goal to find an s_0. This means that all parser states are judged equal if they agree on what's on top of the stack.

A parser state $\langle j, i, (s_0 \texttt{->} s_1 \ldots s_{n_R}), p, t\rangle$ is *complete* if $p = n_R$. Usually p is not written as a separate slot, but is indicated by a big "$\bullet$" at position p. So a state is complete if the $\bullet$ is at the end of the rule. A non-complete state is called *predictive,* because it is "predicting" that the next thing in the word list is an s_{p+1}. For example, a state

$$\langle 0, 3, \text{S} \texttt{->} \text{NP} \ \bullet \ \text{VP}, t\rangle$$

records that, in the course of searching for an S beginning at position 0, an NP was found covering positions 0 through 3; and, in a sense, predicts, or perhaps hopes or wishes, that a VP will be found beginning at position 3 and ending at some as yet unknown position, k. If such a VP is indeed found, then an S will have been found covering positions 0 to k.

There are two key rules for creating new parser states (where σ is a sequence of terminals and nonterminals, possibly empty):

1. *Downward prediction:* Given a predictive state

   $$\langle j, i, s_0 \texttt{->} \ldots \ \bullet \ s_{p+1} \ldots, t\rangle$$

   and a rule in the grammar "$s_{p+1} \texttt{->} \sigma$," generate the state

   $$\langle i, i, s_{p+1} \texttt{->} \ \bullet \ \sigma, \emptyset\rangle.$$

The new state is assumed to be predictive; i.e., σ is assumed to be nonempty. It's possible to eliminate this assumption, but only by making the algorithm more complex, obscuring its essence.

2. *Upward completion:* Given a complete state

   $\langle j, i, s_0 \texttt{ -> } \ldots \bullet, t \rangle$

   and a predictive state

   $\langle l, j, s \texttt{ -> } \ldots \bullet s_0 \sigma, t' \rangle$

   generate the state

   $\langle l, i, s \texttt{ -> } \ldots s_0 \bullet \sigma, t' + t \rangle$

Whether the new state is complete or predictive depends on whether σ is empty or not. Here the expression $t' + t$ is highly schematic. The idea is simply that to the collection of trees t' we add the new elements t. However, if the new state is complete (σ is empty), the result is a (collection containing) a single tree $Tr(s, t + t')$, where Tr is some kind of "tree-making" operation using the collection $t' + t$ as subtrees and s as the root label. As I said, highly schematic.

To get the algorithm started, the pump must be primed with a set of states to which the prediction and completion rules can be applied. One source of states is the words of the sentence. Word i (one-based counting) occupies positions $i-1$ to i. So we can represent the word "can" in position 3 as the state:

$\langle 2, 3, \texttt{Noun -> "can"} \bullet, \{\texttt{"can"}\} \rangle$

The word "can" is ambiguous; it can also be an auxiliary verb, so we should also include this state:

$\langle 2, 3, \texttt{Aux -> "can"} \bullet, \{\texttt{"can"}\} \rangle$

In the unusual event that a sentence is genuinely ambiguous between a syntactic structure in which "can" is a verb or a noun (e.g., "I saw that garbage can fly"), we should get both readings.

We also need a top-level predictive state to provide something for the words to interact with. We need only one:

$\langle 0, 0, \texttt{ -> } \bullet \texttt{ S}, \emptyset \rangle$

The grammar does not really contain a rule with an empty left-hand side. If we ever apply completion to this rule we've found an `S` starting at position 0. If it ends at position N then it's a parse of the entire word sequence as an `S`, which is what we're looking for.

To turn these rules and initial states into an anticipatory dynamic-programming algorithm we must think about the order in which things can be computed. The downward-prediction rule takes states covering positions (i, j) and creates more states covering ranges ending at position j. That suggests that whenever we create a predictive state we immediately create all the predictive states that follow from it.

The upward-completion rule requires two states, a prediction ending at i and a completion from i to j, and creates more states ending at j. Some of the created states may themselves be completions ending at position j, but if so they too generate only states ending at j. That suggests that we examine each word, represented as a completion state at the next position, and generate all the states that follow from it, all of which are in state j.

Because in upward-completion two states interact, it might seem that it is necessary to check for possible interactions in two different situations: the one we just looked at it, when a completion is generated, and the opposite case, where a prediction is generated. Fortunately, this second case will never occur. Completions ending at position j interact only with predictions ending at positions to the left, and generate only further completions ending in position j. So if all parser states ending at i are produced before any ending in position $i + 1$, it is never the case that a prediction state S_p that can interact with a completion state S_c is generated *after* S_c.

Traditionally the Earley algorithm is implemented with a one-dimensional array of size N, where N is the number of words in the string to be parsed. The positions in the array are filled in order. We would like to implement the Earley algorithm with immutable data structures, but we would hate to give up the random-access feature of arrays. That may seem like a small thing, but dynamic programming algorithms quickly escalate in cost as dimensions are added (as they will be), and it's adding insult to injury to make us go hunting down lists to find prediction states matching a completion state.

So we introduce *lazily filled arrays*, or "lazy arrays" for short, as a useful abstraction that gets us the best of both worlds. There are three things you can do with these objects:

1. *Create one:* Use `LazyArray1.create[`T`](`d`)` to create a one-dimensional lazy array of maximum size d holding objects of type T. Its type is `LazyArray1[`T`]`. Its *size*, initially, is 0. But see operation 3.

2. *Access an element:* If a is a lazy array of size k and $0 \leq i < k$, a`(`i`)` is its i'th element.

3. *Add an element at the end:* If a is a lazy array of size k, a`.extend(`t`)` returns a lazy array of size $k+1$, whose k'th element is t.

The lazy array is backed by a real array of size d, so access to an element is truly random. Extending an array usually takes time $O(1)$, assuming that it's extended only once (which is the use case we have in mind). Extending an array that's been extended before requires copying its contents to a new backing array. If execution time is factored out, it's impossible to tell whether the array has been extended and if so how many times.[9] So the one tiny mutable piece of the data structure is inaccessible.

For our purposes, the objects stored in the array are of type `StateSet`, whose implementation is unimportant, provided it allows us to perform two operation:

1. ss`.addIfNew(`s`)`, which tries to adds s to `stateSet` ss if it's not already there. It returns either `Some(`ss'`)`, where ss' is the enlarged state set, or `None` if s is already in the set. (In cases where it is known that s is not in the set, the variant `addNew` just returns the enlarged state set. It raises an exception if s is not new after all.)

2. ss`.getPredictions(`N`)`, which returns all predictive states in the set that are "predicting" nonterminal N.

The Earley algorithm can now be written as follows. Its inputs are a grammar and a list of lexically analyzed strings, called `words`.[10] The lexical system uses F-structures (see lecture 29–30) to represent words, and in general the parser uses lists of F-structures to represent its results. Every such F-structure has a `cat` feature whose value is a terminal or nonterminal symbol of the grammar. A complete parser state, of the form $\langle j, i, s \texttt{ -> } \ldots \bullet, r \rangle$, one representing the discovery of a word or phrase of type s, where s is a terminal or nonterminal symbol of the grammar, has result r = a singleton list whose only element is an F-structure whose `cat` field is s. For the lexical system, the value of `cat` is a terminal. Here's the pseudo-code:

[9] This is not to say that multiple threads could access and extend the lazy array safely; in the current implementation, they cannot.

[10] A few pages back, we suggested that the lexical system could handle lexical ambiguity by producing more than one word at a given position. To simplify the pseudo-code, we've left this wrinkle out. See exercise 1, below.

```
def earleyParse(grammar, words: List[FStructure]): List[FStructure] = {
  val n = words.length
  // The max length is 1 + the number of words because it's
  // indexed by position in the word list –
  var stateSets = LazyArray1.create[StateSet](n + 1)
  // – stateSets(i) is the set of all states ending at position i
  stateSets = stateSets.extend(pursuePredict(⟨0, 0, -> • S,()⟩))
  // Position after next word in word list –
  var i = 1
  while (i ≤ n) {
    word-i = next word in words
    var stateSetHere = new StateSet()
    val wordParserState =
      ⟨i−1, i
       word-i.get('cat) -> word-i.get('lemma) •,
       List(word-i)⟩
    stateSetHere = stateSetHere.addNew(wordParserState)
    stateSetHere =
      pursueComplete(wordParserState, stateSets, stateSetHere)
    stateSets = stateSets.extend(stateSetHere)
    // --Now stateSets(i) is complete and accessible
    // It contains all parser states ending at i i = i + 1
  }
  extract Ss from stateSets(n)
}


def pursueComplete(parser state,
                   stateSets: LazyArray1[StateSet],
                   startStateSet: StateSet): StateSet = {
  parser state match {
    case ⟨start, end, lhs -> ..., subtrees⟩ => {
      var nextStateSet = enlargedStateSet
      val newTrees =
        if (lhs is terminal of grammar) subtrees
        else List({cat: lhs,
                   children:[subtrees]})
      // Find states representing the search for the lhs
      // at points ending at start --
      val predictStates = stateSets(start).getPredictionsFor(lhs)
```

```
      for (⟨pStart, _, pRule, pPos, pTrees⟩ <- predictStates) {
        val newState = ⟨pStart, end, pRule, pPos + 1, pTrees + newTree⟩
        nextStateSet =
          pursueIfNew(newState, sset, nextStateSet,
                      if (newState is complete)
                        pursueComplete
                      else pursuePredict)
      }
    }
  nextStateSet
}

def pursuePredict(parser state,
                  sset: LazyArray1[StateSet],
                  startStateSet: StateSet): StateSet = {
  parser state match {
    case ⟨start, end, rule, pos, subtrees⟩ => {
      var nextStateSet = startStateSet
      val nextCat = nonterminal at pos on rhs of rule
      var nextStateSet = startStateSet
      // Find rules that reduce that category --
      for (grammar rule r  with lhs == nextCat) {
        nextStateSet =
          pursueIfNew(⟨end, end, r, List()⟩, sset, nextStateSet,
                      pursuePredict)
      }
      nextStateSet
    }
  }

def pursueIfNew(parser state,
                sset: LazyArray1[StateSet],
                startStateSet: StateSet,
                pursuer: (Parser State, LazyArray1[StateSet], StateSet)
                        => StateSet): StateSet = {
  startStateSet.place() match {
    case Some(enlargedStateSet) =>
      pursuer(parser state, sset, enlargedStateSet)
    case None => startStateSet
  }
```

}

Exercise 1 *In the program* `Earley`*, what changes are required if* `words`*, instead of being a list of F-structures, is of type* `List[List[FStructure]]`*? (The idea is that each element is a nonempty list of alternative interpretations of the lexical item at that position.)*

Now to answer the question, Why does the Earley algorithm work, if it does? This is actually two questions:

1. Does it find every parse licensed by a context-free grammar, assuming it halts?

2. Does it halt?

The second question is easy to answer. There are only a finite number of alternative parser states ending at position i. No state is pursued if it's already in the state set for position i. There are only a finite number of positions. There's not much else to say.

To answer the first question, it's useful to picture an arbitrary parse tree licensed by a CFG. The *symbol* associated with a node is a terminal symbol s_T if the node is a leaf, and a nonterminal s_{NT} if it's a nonleaf. Each leaf is also associated with a word, of category s_T. Each nonleaf is associated with a a grammar rule $s_{NT} \rightarrow s_1, \ldots, s_k$, and s_i must be the symbol associated with its ith child. (The satisfaction of this constraint is what makes the parse tree a legal parse tree.)

Mark each terminal node with a parser state $\langle i{-}1, i, T \texttt{ -> } w \bullet, \ldots \rangle$, where w is the word at that node, just as the program does. Mark each nonterminal node n (label: nonterminal s_{NT}) except the topmost with a parser state, bottom-up, as follows: if its children are labeled

$\langle p_0, p_1, s_{NT} \texttt{ -> } s_1 \bullet s_1 \ldots s_k, \ldots \rangle$,
$\langle p_1, p_2, s_{NT} \texttt{ -> } s_1 s_2 \bullet s_3 \ldots s_k, \ldots \rangle$,
$\ldots$,
$\langle p_{n-1}, p_n, s_{NT} \texttt{ -> } s_1 \ldots s_{k-1} s_k \bullet, \ldots \rangle$,

then label it

$\langle p_0, p_n, s_{NT'} \texttt{ -> } s'_1 \ldots s_{NT} \bullet s'_{k'}, \ldots \rangle$

where $s_{NT'} \rightarrow s'_1 s'_2 \ldots s'_{k'}$ is the grammar rule associated with its parent, and $s'_j = s_{NT}$ for some j, $1 \leq j \leq k'$.[11] The topmost node, for the symbol is `S`, is labeled similarly, except that the "grammar rule" is the artificial $\langle 0, n, \rightarrow \mathtt{S} \bullet, \ldots \rangle$.[12]

It is now possible to prove

Theorem 1 *For every parse tree of a word string licensed by a CFG, the Earley algorithm finds every state labeling every node of the parse tree.*

Evidently, finding a label for the top node implies that a successful parse exists. The last field of the tuple, the one neglected in our description of the labels on each node, is an F-structure that mirrors the parse tree discovered.

Exercise 2 *Prove the theorem. Hint: Try induction on i, the position after the ith word. Show that every state $\langle j, \mathtt{i}, rule, p, \ldots \rangle$ in the tree will be found on iteration* `i` *of the loop in* `earleyParse`*. in the parse tree.*

Exercise 3 *Correct our neglect of the result fields by showing that for every successful parse, a state will be found whose result field is an F-structure equivalent to the parse tree. That is, at the top the F-structure is of the form* `{cat: S, children:[`c_1`, ..., `c_k`]}`*, where c_i is the i'th child of the top node, and the square brackets around* `[`c_1`, ..., `c_m`]` *stem from the abbreviation convention introduced in lecture 29–30 for* `first-rest` *lists.*

[11] If there is more than one such j, pick the νth one, if n is the ν'th child of its parent with $s_j = s_{NT}$.

[12] We've left out the last element of every state, because these are the "result values" that play a passive role in the algorithm; they're put together into something interesting, but it doesn't affect whether a parse is successful or not. But see exercise 3.