Improving Upon Earley's Parsing Algorithm In Prolog

Matt Voss
Artificial Intelligence Center
University of Georgia
Athens, GA 30602

May 7, 2004

Abstract

This paper presents a modification of the Earley (1970) parsing algorithm in Prolog. The Earley algorithm presented here is based on an implementation in Covington (1994a). The modifications are meant to improve on that algorithm in several key ways. The parser features a predictor that works like a left-corner parser with links, thus decreasing the number of chart entries. It implements subsumption checking, and organizes chart entries to take advantage of first argument indexing for quick retrieval.

1 Overview

The Earley parsing algorithm is well known for its great efficiency in producing all possible parses of a sentence in relatively little time, without backtracking, and while handling left recursive rules correctly. It is also known for being considerably slower than most other parsing algorithms[1]. This paper outlines an attempt to overcome many of the pitfalls associated with implementations of Earley parsers. It uses the Earley parser in Covington (1994a) as a starting point. In particular the Earley top-down predictor is exchanged for a predictor that works like a left-corner predictor with links,

[1]See Covington (1994a) for one comparison of run times for a variety of algorithms.

following Leiss (1990). Chart entries store the positions of constituents in the input string, rather than lists containing the constituents themselves, and the arguments of chart entries are arranged to take advantage of first argument indexing. This means a small trade-off between easy-to-read code and efficiency of processing.

The paper assumes a general understanding of chart parsing, and other standard parsing algorithms. A brief overview of the Earley algorithm is presented, followed by an implementations that makes the above modifications to the original.

2 Earley's Algorithm

An Earley parser is an active chart parser[2]. It eliminates backtracking completely by making a chart entry for everything that it has parsed: completed constituente (inactive constituents), as well as constituents being parsed (active constituents). It has several other attractive features:

1. It has an upper time bound proportional to n^3, and a lower time bound near n. This is close to optimal.

2. It does not specify a grammar, or a form for the grammar. It works on any.

3. It does not loop on left recursive rules, a problem that arises for parsers using Definite Clause Grammars.

4. Since it puts information about active as well as inactive constituents in the chart, it pursues all possible parses concurrently. At the end of the parse, all the alternatives are also stored in the chart.

Earley chart entries keep track of four pieces of information:

1. The current goal.

2. Its starting position.

3. A list of constituents left to parse to complete the goal.

[2]For an introduction to chart parsing in Prolog the reader is referred to Covington (1994a) and Gazdar and Mellish (1989)

4. The endpoint of the current goal.

Earley represented this information as a set of production rules of the form:

$$S \rightarrow \bullet NP\ VP\ 0\ 0$$

$$S \rightarrow NP \bullet VP\ 0\ 2$$

The dot represents the current position of the parser in the input stream. The first rule means the parser is at the beginning of the S constituent. The second rule above means the parser has found an NP and is now looking for a VP. The numbers repres

As an active chart parser Earley's algorithm needs to do three things:

1. Add "active" entries, more current goals, to the chart. It will do this by looking for rules that expand current goals, and making chart entries based on these rules.

2. Update current goals after looking at the next word in the input stream. This means adding chart entries that reflect the state of the parser after accepting a word.

3. Use any completed constituents to complete goals requiring those constituents. For example, if you have completed an NP and a VP, then complete an S.

These three parts are called the predictor, the scanner, and the completer. They will be the three main components of the parser.

Earley's original proposal was a top-down predictor. This predictor is known to over-predict because it does not look ahead at the next word in the input stream. It will assert chart entries for all of the rules that have the current goal as their main goal, no matter what their subgoals are. Changing this feature is the primary focus of the implementation here. However there are some other considerations to be made before changing this feature.

3 Chart Entries

We need a representation for chart entries in Prolog. Sacrificing some readability, we can take advantage of Prolog's first argument indexing to represent chart entries as follows:

```
chart(0,0,[np,vp],s).
chart(1,0,[vp],s).
```

These entries express the same information as the chart entries in Earley notation above, but they do it in reverse order. The first argument is the current position in the input stream, the second argument is the start point of the current constituent, the third argument is a list of subgoals yet to be processed, and the fourth argument is the current goal. The idea is that there many fewer constituents that end at a given position in the list than constituents that begin there, so we should index by the argument with more distinct values to make looking these chart entries up more efficient.

We can look at an alternative to see the benefit of this approach. Chart entries could be presented as follows:

```
chart(s,0,[np,vp],0).
```

Comparing the run times for parsing the sentence "that dogs chase cats surprises me" 500 times with each representation shows how striking the improvement can be. The alternative averaged around 80 seconds runtime, while the chosen representation averaged around 10. The test was conducted on a Pentium 2 733 MHz machine. The numbers are not so important as the large difference between them. Simpler sentences will produce less of a difference in runtime. This is by no means an extensive test but it at the very least presents some evidence in favor of the chosen representation.

Gazdar and Mellish (1989)proposes chart entries also include a fifth argument representing a list of all the constituents that have been parsed. this way, when the parse is finished, the user can display the parse tree. This approach,however does not give the option of leaving out that step. The parse tree can be retrieved by adding arguments to the nodes in the rules. Hence the fifth argument is left off of the chart in favor of a more flexible parser.

3.1 Grammars

The grammar is the only part of the program that the user supplies. A sample grammar is included along with the source code for the parser. If you want to make a custom grammar, then include the clause

```
:- ensure_loaded('earley.pl').
```

at the top of your file. Consult your grammar file to use the parser.

Words and rules follow the representation used by Covington (1994a). Words are represented by a two place predicate as follows:

```
word(d,the).
word(n,cat).
word(v,jumped).
```

The first argument is always the part of speech of the word. The second argument is the word itself.

Rules are represented similarly:

```
rule(s,[np,vp]).
rule(np,[d,n,]).
rule(vp,[v,np]).
```

The first argument is the constituent; the second argument is a list of the constituents that comprise it.

Representing rules and words in this manner allows for maximum flexibility in the grammar. It is very easy to add arguments to the constituents or words for use with WordNet (Princeton 2003), GULP (Covington 1994b), or to produce tree structures from the grammar.

3.2 Using WordNet to Supply Words

Using WordNet is simply a matter of adding another clause `word/2` clause to the grammar. It should be as follows:

```
word(Cat,Word) :-
  s(_,_,Word,Cat,_,_).
```

The user should make sure to take note of the category markers in WordNet and construct rules accordingly. The user should also be aware that none of the closed-class words are in WordNet. These the user will have to include.

3.3 Adding Arguments in GULP Notation

Adding arguments in GULP notation is fairly straight forward. The parser treats any rule, with or without arguments on the goals, in the same way. To use GULP just add arguments to the appropriate goals. For example:

```
rule(np(number:N..case:C),[n(number:N..case:C)]).
word(n(number:sg..case:nom),she).
```

3.4 Using ProNTo_Morph

ProNTo_Morph (Schlacter 2003) is a morphological analyzer for English and is availible at the ProNTo website:

`http:\\www.ai.uga.edu\mc\ProNTo`

A large lexicon is easy to create using ProNTo_Morph in conjunction with WordNet and GULP. The idea is to run each word through ProNTo_Morph, strip off the endings, look up the root in WordNet and use the suffix to take care of agreement issues. An example of a grammar that uses these three tools to check for subject/verb agreement can be found in the file `grmr.glp`. The following is an example to help make the key idea clear.

```
word(n(number:pl),Word) :-
   morph_atoms(Word,[[W,-es]]),
   s(_,_,W,n,_,_)
   ;
   fail.

word(n(number:pl),Word) :-
   morph_atoms(Word,[[W,-s]]),
   s(_,_,W,n,_,_)
   ;
   fail.

word(n(number:pl),Word) :-
   morph_atoms(Word,[[W,-pl]]),
   s(_,_,W,n,_,_)
   ;
   fail.
```

We find a plural noun in the following way. ProNTo_Morph distinguishes three possible endings for a plural noun: '-es','-s',and '-pl'. For each case we have one `word/2` clause. I will explain the first one. `morph_atoms/2` is non-deterministic. It returns each possible morphological analysis only upon

backtracking. So if the first one does not match what we are looking for we need to fail and check other posibiities. In the first clause, we break off endings until we get an '-es' ending, then look the root up in WordNet to make sure we have a valid word.

A sample grammar that illustrates the use of ProNTo_Morph in conjunction with GULP and WordNet is included with the parser in a file called `grmr.glp`. In addition an updated version of GULP is required and also availible with the parser in a file called `gulp3swi.pl`. To use this version of GULP simply consult `gulp3swi.pl` into SWI-Prolog, then consult the appropriate `.glp` file.

4 The Parser

gssugi.llh

These predicates say there is a link between the first argument and the second.

The parse is finished when `process/2` has reached the index of the last word in the sentence, and there is a chart entry whose main goal is our initial goal, and which has no more subgoals to try.

Example queries:

```
?- parse(s,[the,dog,chases,the,cat]).
Yes

?- parse(np,[the,dog]).
Yes
```

4.2 store(+ChartEntry)

We need to do a little more than just add chart entries whenever we find a candidate. In particular we check to make sure there's not already a chart entry that contains the information we are trying to add. We also do a subsumption check on both the current goal and the list of subgoals. The subsumption check is important when we have arguments on the nodes. If the goal or its subgoals have an argument that is an uninstantiated variable we want to make sure the variable remains that way before it goes into the chart. If the variable would unify with one of the arguments of a node in a chart entry, the subsumption check prevents that new entry from going into the chart. If it was allowed to go in, that variable would be instantiated, possibly to something undesirable

```
store(chart(A,B,C,D)) :-
  \+ ( chart(A,B,C1,D1),
       subsumes_chk(C1,C),
       subsumes_chk(D1,D) ),
  asserta(chart(A,B,C,D)).
```

As packaged, the parser includes a subsumption checker from Covington (1994a); however, some Prolog implementations have a predefined version by the same name. If this is the case with your particular Prolog, just comment out the line `:- ensure_loaded('subsumes.pl').` in the file `earley.pl`. This is a slight inconvenience but ensures you will be running the most efficient version of the predicate.

4.3 process(+StartPosition,-EndPosition)

This predicate oversees the parsing process. It calls the three parts of the parser: the predictor, the scanner, and the completer until the end of the sentence is reached.

```
process(End,End) :- !.

process(Position,End) :-
  predictor(Position),
  scanner(Position,NewPosition),
  completer(NewPosition),
  process(NewPosition,End).
```

4.4 predictor(+Position)

The predictor works like a left–corner parser with links. We have already seen how the links are defined. Now we see how they are put to use. The first step is to peek at the next word in the input stream, then make predictions based on that word. The idea is to look up each goal at the current position that has the category of the first word as one of its subgoals and use these two pieces of information to make new chart entries that link to the main goal. There are several tasks to be completed. This predicate simply orchestrates the predicting process. It calls predicates that do the predicting.

```
predictor(Position) :-
  chart(Position,_,_,[Goal|_]),
  c(Position,_,W),
  predict(W,Goal,Position),
  fail.

predictor(_).
```

4.4.1 predict(+W,+Goal,+Position)

Make predictions based on the current word.

If the current goal is the category of the current word, then there are no more predictions to make.

```
predict(W,Cat,_) :-
  word(Cat,W),!.
```

Otherwise use the category of the current word to restrict predictions. Predictions need to be made that link the current word to the current main goal. `predict_all/3` is called to make these predictions.

```
predict(W,Goal1,Position) :-
  word(Cat,W),
  predict_all(Goal1,Cat,Position),
  predict(W,Goal1,Position),
  fail.

predict(_,_,_).
```

4.4.2 predict_all(+Goal,+Category,+Position)

This predicate finally makes all the predictions for the given input category. It is a type of left–corner parser with links inspired by one proposed by Leiss (1990). It stops making predictions when the current goal is the same as the category of constituent it is looking for. Otherwise it finds rules whose first subgoal is the current category, checks to make sure there is a link between the goal it found and the current main goal, then enters the appropriate chart entry into the knowledge base.

```
predict_all(Goal,Goal,_).

predict_all(Goal1,Cat,Position) :-
  rule(Goal,[Cat|T]),
  l(Goal,Goal1),
  store(chart(Position,Position,Goal,[Cat|T])),
  predict_all(Goal1,Goal,Position).
```

The predicate also makes sure it did not miss a null constituent when it looked at the first word. It checks the second subgoal of each rule for a constituent of the current category. If it finds one, it checks if the first subgoal can be a null constituent. If it can it makes sure there is a link between the main goal and the proposed goal, then enters the appropriate information in the chart. At this point it also needs to call the completer since it knows it completed a null constituent.

```
predict_all(Goal1,Cat,Position) :-
  rule(Goal,[D,Cat|T]),
  rule(D,[]),
  l(Goal,Goal1),
  store(chart(Position,Position,Goal,[D,Cat|T])),
  predict_all(Goal1,Goal,Position),
  store(chart(D,Position,[],Position)),
  complete(D,Position,Position).
```

4.5 scanner(+End,-End2)

The scanner actually eats the next word in the input stream. To do this it looks up each word that begins at the current point in the input stream. If the word is of the same type as the first current subgoal, then the parser asserts a new chart entry with updated information. The new chart entry reflects the state of the parser after it has consumed the word it was looking at.

```
scanner(End,End2) :-
  chart(End,Start,C,[G|Goals]),
  c(End,End2,Word),
  word(G,Word),
  store(chart(End2,Start,C,Goals)),
  fail.
```

The scanner returns the input position after accepting the word it was looking at.

```
scanner(End,End2) :-
  End2 is End+1.
```

4.6 completer(+Position)

The completer looks for chart entries that have no subgoals left to process. For each one it finds, it tries to use that chart entry to complete higher goals.

```
completer(Position) :-
  chart(Position,PC,C,[]),
  complete(C,PC,Position),
```

```
  fail.

completer(_).
```

4.6.1 complete(+Constituent,+Start,+End)

This predicate actually completes subgoals. The completer has just found a completed constituent. So now it can find chart entries that have subgoals with one subgoal left to complete. Advance one step in the input string and update the goal list to reflect another subgoal processed. Assert this change as a chart entry.

```
complete(C,PC,Position) :-
  chart(PC,PC0,C0,[C|Goals]),
  store(chart(Position,PC0,C0,Goals)),
  Goals == [],
  complete(C0,PC0,Position),
  fail.

complete(_,_,_).
```

5 Utilities

It is fairly easy to access parse trees if the rules of the grammar have arguments on their nodes. Here are two predicates that retrieve all of the completed constituents from the parse.

5.1 get_constituents(+Constituent,+Sentence)

This predicate is called just like `parse/2`. It parses a sentence then prints out all of the completed constituents from the parse.

```
?- get_constituents(s(X),[the,dog,chases,the,cat]).
s(np(d(the), n(dog)), vp(v(chases), np(d(the), n(cat))))
vp(v(chases), np(d(the), n(cat)))
np(d(the), n(cat))
np(d(the), n(dog))
```

```
X = s(np(d(the), n(dog)), vp(v(chases), np(d(the), n(cat))))
```

5.2 get_constituents_list(+Con,+Sen,-List)

This predicate is the same as above except that instead of printing the constituents, it returns them in a list.

```
?- get_constituents_list(s(X),[the,cat,chases,the,dog],List).

X = s(np(d(the), n(cat)), vp(v(chases), np(d(the), n(dog))))
List = [s(np(d(the), n(cat)), vp(v(chases), np(d(the),
n(dog)))), vp(v(chases), np(d(the), n(dog))),
np(d(the), n(dog)), np(d(the), n(cat))] ;

No
```

6 Limitations

This algorithm was meant to improve upon another, and hence overcome many of its limitations. There are however a few improvements that could be made to this version.

It was a design decision not to store part of the chart in a list. Having some chart entries in the list might complicate the code somewhat, but may allow for better parsing in some cases.

This parser does not implement restriction.[3] This is not seen as a big limitation, since restriction is used to solve a very particular problem with Earley parsers. Fixing the problem would mean a less efficient parser. A trade-off had to be made.

References

Covington, Michael(1994a) *Natural language processing for Prolog programmers.* Englewood Cliffs, N.J.: Prentice Hall.

[3]See Covington (1994a, 186–187) for an explanation of this concept.

Covington, Michael (1994b) GULP 3.1: An Extension of Prolog for Unification-Based Grammar. Research Report AI-1994-06, Artificial Intelligence Center, The University of Georgia.

Earley, Jay (1970) An efficient context-free parsing algorithm. *Communications of the ACM,* 13:94–102.

Gazdar, Gerald and Chris Mellish (1989) *Natural language processing in Prolog: an introduction to computational linguistics.* Wokingham: Addison-Wesley.

Leiss, Hans (1990) On Kilbury's modification of Earley's Algorithm. *ACM Transactions on Programming Languages and Systems,* 12:610–640.

Princeton (2003) WordNet 1.7.1. http://www.cogsci.princeton.edu/∼wn/.

Schlacter, Jason (2003) *ProNTo_Morph: Morphological Analysis Tool.* University of Georgia. http://www.ai.uga.edu/mc/ProNTo.