Basic Algorithms for Computational Linguistics

Project Report

Anna Schmidt, Tim Krones and Marc Schulder

September 29, 2011

1 Introduction

As project work for the lecture *Basic Algorithms for Computational Linguistics* our group implemented a probabilistic bottom-up chart parser titled *BOP* using the Python programming language. Chapter 2 of this report documents this work by describing the main components and algorithms of our system and giving instructions on how to run it. Additionally, chapter 3 describes an experimental n-best parsing strategy and its implementation. The report is concluded in chapter 4 with a final discussion and thoughts on future work.

2 Bottom-Up Parser

A bottom-up chart parser usually consists of two main components, namely a *chart* and a *queue*, which contain *edges* representing words or constituents of sentences. In order to be able to successfully parse a sentence, the parser also needs a grammar and a parsing *strategy* that determines in which order the edges are taken from the queue and placed in the chart during the parsing process. In our implementation, each of these components is represented by a distinct class.

The chart is represented by a two-dimensional array. The queue is simply a list of edges which is ordered according to the parsing strategy selected by the user.

Edges are complex objects. They consist of a start and end point, a *production rule*, and a *dot*. For the production rule associated with a given edge, the dot marks which elements on its right-hand side (RHS) have already been found by the parser. If there are no remaining RHS elements to the right of the dot, the edge is said to be *complete* or *inactive*; if there are, it is considered *incomplete* or *active*.

In addition to a particular production rule, every edge has an edge *probability* associated with it. This probability is the product of the probability assigned to its production rule and the probabilities of all RHS elements **before** the dot, i.e. those that have already been found.

Every edge also separately stores a list of these RHS elements. Without this, the system would not be a full-fledged parser but rather a recognizer: While it **would** be able to accept sentences which are licensed by a given grammar and reject those that are not, it would **not** be able to associate grammatical sentences with their underlying syntactic structure.

In order to recover the tree structure associated with a particular parse, the parser needs to store the immediate substructure of each edge and use this information to "work its way down" from the S edge spanning the whole input sentence.

2.1 Parser rules

The main functionality of a bottom-up chart parser is provided by three rules: Initialization rule ("init rule" for short), predict rule and fundamental rule. These rules use the components described above to handle the parsing process.

The init rule, which is run only once, starts the parsing process by generating complete edges for every single token of the input sentence and adding them to the queue. Since bottom-up parsers begin by operating on the word level, the edges representing the individual tokens must be put on the chart first. There are different ways of ensuring this behavior depending on the parsing strategy.

The predict rule takes only complete edges as input. It consults the grammar and creates a new incomplete edge for every rule that has the LHS of the input edge as a first daughter on its RHS. All edges created by the predict rule are self-loop edges, since they are only predictions and have no substructures yet. Just like the initialization rule, the predict rule pushes each edge it creates to the queue.

The fundamental rule is responsible for pairing incomplete edges with appropriate complete ones. It does so by first checking for each incomplete edge which complete edges directly follow it. If the element after the dot on the RHS of an incomplete edge matches the LHS of the production rule associated with a given complete edge, it creates a new edge - which can be complete or incomplete - with the dot advanced over the previously missing RHS element, and pushes it to the queue.

After initializing the parsing process using the init rule, predict rule and fundamental rule are applied repeatedly until the maximum number of parses specified by the user has been found or the queue is empty. During the whole parsing process, the parser is not allowed to add the same edge to the chart or the queue twice. This restriction ensures that the system can deal with left-recursive grammar rules; without it, the system would enter an infinite loop every time it encounters such a rule.

2.2 Queue strategies

The order in which edges are taken off the queue and placed on the chart during the parsing process is determined by the parsing strategy selected by the user. In our system, three different strategies can currently be used: *first-in, first out* (*FIFO*), *best-first* and *alt-search*.

When using the FIFO strategy, the ordering of edges on the queue simply results from the order in which they are pushed to it. In other words, the first element added to the queue is always the first one that is taken off the queue and put in the chart.

Under the best-first strategy, the queue is ordered with respect to the probabilities associated with the edges on the queue: The queue element with the highest probability always takes precedence over all the others during the parsing process. This guarantees that the first complete S edge spanning the whole sentence that the system finds is always the best (i.e., the most likely) parse for this sentence. It can not ensure that the second found parse also is the second best, though.

Alt-search is an experimental n-best strategy that we developed as part of this project. It is described in detail in chapter 3.

2.3 Execution

In order to run the parser, all you need to do is

1. extract the archive;
2. `cd` into the extracted folder;
3. ensure the file `bop.py` is executable and run it by issuing `python bop.py` or `./bop.py`; and
4. follow the prompts.

3 Alt-Search Parser Extension

The parser presented in section 2 is capable of two things: Finding the best parse and finding a given number of parses. However, it is not given that the second parse found is also the second-best parse, neither the third found the third-best, etc. Such an n-best parse usually requires the computation and subsequent ranking of all parses. This is computationally expensive. To avoid it, we suggest an extension to the parser that will allow it to find, if not the n-best parses, at least a good approximation of them, while hopefully speeding up the parsing process.

3.1 Theory

The base assumption of our extension is that the second-best parse will probably have a similar base structure to the best parse and only differ in a bit of its sub-structure. Thus, after a best parse is found, the system should prioritize edges that can directly replace edges of the best parse while re-using the rest of its sub-structure. Hence the extensions name *alt-search parsing*, signifying its search for alternative sub-structures.

To give an example, imagine the best parse has an upper structure of S[NP(0:3) VP(3:8)] Replacing the NP with a different NP that also spans nodes 0 through 3, but with a different sub-structure, allows the system to re-use the whole VP (which after all was the best for this span).

Of course, the same can be done for all other sub-sections of the parse tree and only stops when no edge can be replaced by any other.

3.2 Application

We now present a first tentative approach to implementing this theory. It represents a compromise between our two goals of increasing speed and quality of the parsing process, while trying to preserve the relative simplicity of chart parsing.

To this end, we expand the main algorithm to call an additional alt-search rule (see algorithms 2 and 3 in the appendix). We also introduce a new queue strategy that allows special manipulations through the new rule.

We will first define the new queue to establish its functionality, followed by a description of alt-search rule, which explains the purpose of the new functions.

3.2.1 Alt-Search Queue

The new queue does in fact consist of two sub-queues, the base queue and the priority queue. Both are sorted by likelihood, just like the queue of the *best first* strategy.

At first, the base queue is accessed normally, until the priority queue is activated by the alt-search rule (see below). From then on, all standard requests (e.g. push and pop actions) are redirected to the priority queue. When the priority queue is emptied, the queue automatically switches back to the base queue.

The second addition to the alt-search queue is the particular-pop command. It allows the specification of a left hand side, starting node and end node of the edge that will be popped from the queue[1]. Additionally, the edge in question must be complete (i.e. its dot is at the right end of the rule). Note also that particular-pop always accesses the base queue, while the normal pop command accesses either base or priority queue, depending on which is activated.

The new queue is considered empty when both its sub-queues are empty. It is also possible to check whether the priority queue is empty.

3.2.2 Alt-Search Rule

In the main algorithm, the new alt-search rule is called right after the predict and fundamental rules, but only if the most recent addition to the chart is a complete s-edge that spans the whole sentence (i.e. is a complete parse). Its first action is to activate the priority queue. This ensures that all edges added through the alt-search will be processed before those remaining from the conventional parsing process.

It then requests a further complete parse from the base queue via particular-pop[2] (which ignores the priority queue activation). If one is returned, it is pushed to the priority queue and the rule terminates. During the next iteration of the main parsing loop (lines 7–17 in algorithm 2), it is then popped from the (otherwise empty) queue and processed by the normal rules. Thereafter, the alt-search rule is invoked again, as the newest edge on the chart is once more a complete parse.

This process is repeated until no more complete parses remain on the queue. In that case, the alt-search rule moves on to its main part, the search for alternative sub-trees. This search is repeated for all complete parses in the chart, starting with the most likely one.

If at least one alternative is found for a parse, the search is stopped and the remaining parses are not checked. The rule is exited and the alternative parses are generated by processing the priority queue in the main loop. Afterwards, the alt-search rule is re-entered and the search begins anew.

[1] To allow a fast search of the queues via particular-pop, we have introduced a secondary data structure which allows directed access via the given parameters. The downside of this is that we need to maintain this secondary structure throughout the normal parsing process as well.

[2] This step is optional. We have included it as it is a quick way of finding further parses that are still stuck in the queue. It does not directly belong to our agenda of finding alternate sub-trees. It can be skipped by moving from line 1 of algorithm 3 directly to line 7.

If no alternative sub-trees can be found for any of the parses, the rule ends without filling the priority queue, causing the system to move back to the base queue at the next iteration and continue normal parsing until yet another complete parse is found.

Sub-tree search Let us now look at what happens during the search for sub-tree alternatives of a parse. Starting with the direct daughters of the parse's root edge, the tree is searched breadth-first. For each daughter, it is checked whether the base queue contains alternative complete edges with the same left hand side, starting node and end node as the daughter. The resulting list contains all known alternative edges for all daughters at the current tree depth. If any alternative edges are found, the search is stopped at this point, the found edges are pushed to the priority queue and the alt-search rule terminates.

If no alternatives could be found, the search depth is increased by one, meaning that the current daughter's own direct daughters are checked for alternatives. This is repeated until either some alternatives are found or the leaf level is reached, at which point the search moves on to the next complete parse tree until all trees are searched and alt-search is completed.

4 Conclusions

We have presented a bottom-up parser with two conventional parsing strategies (*first-in, first-out* and *best first*) as well as an experimental extension. We accompanied this with a complete implementation in Python.

In chapter 2 we described the bottom-up parser's structure and given instructions for the implementation's execution. In chapter 3 we have introduced its extension, the *alt-search parser*, which prioritizes parses with a structure similar to the first found (and best) parse.

4.1 Future Work

Due to time constraints, the alt-search parser could not be extensively tested or its efficiency be shown computationally or mathematically. Preliminary tests have shown that alt-search can reduce the number of main loop iterations[3] (i.e. edges added to the chart), although no tests were made as to the ratio of speed gains from this against the speed loss of the additional search rule.

Several trade-offs remain where we are uncertain which approach would be favourable. Generally we have chosen in favor of fast parse generation (see footnote 2) and search (breadth-first instead of computing all potential sub-trees) rather than trying to ascertain that the next parse will definitely be the best one.

[3]The sentence "Jack saw small dogs and mice with small telescopes and big telescopes" has 9 possible syntax structures under the given example grammar. Computing all possible parses requires 300 iterations. When limited to 5 parses, the best-first strategy requires 283 iterations, while for alt-search 233 iterations suffice.

One point where we might still improve on speed is the search of all complete parses. Right now, the best parses are always searched before worse ones, no matter whether the queue can still potentially contain alternative sub-trees for them or not. For the sake of simplicity and to avoid logical pitfalls, we have omitted a special treatment of this, but adding it to the implementation would be possible.

References

S. Bird, E. Klein, and E. Loper. *Natural language processing with Python.* O'Reilly Media, 2009.

G. Gazdar and C.S. Mellish. *Natural language processing in PROLOG: an introduction to computational linguistics.* Addison-Wesley Pub. Co., 1989.

Appendix

Algorithm 1 Basic Bottom-Up Parsing Algorithm

```
1: Tokenize sentence
2: Initialize chart and queue
3: for each token do
4:   Initialization rule(token)
5: end for
6: while queue is not empty and not enough parses found do
7:   Pop next edge from queue
8:   Add edge to chart
9:   if edge is complete then
10:    Predict rule(edge)
11:  end if
12:  Fundamental rule(edge)
13: end while
14: return found parses
```

Algorithm 2 Bottom-Up Parsing Algorithm with Alt-Search Extension

```
1: Tokenize sentence
2: Check for unknown tokens
3: Initialize chart and queue
4: for each token do
5:   Initialization rule(token)
6: end for
7: while queue is not empty and not enough parses found do
8:   Pop next edge from queue
9:   Add edge to chart
10:  if edge is complete then
11:    Predict rule(edge)
12:  end if
13:  Fundamental rule(edge)
14:  if edge is complete parse or edge was last priority queue element then
15:    Alt-Search rule()
16:  end if
17: end while
18: return found parses
```

Algorithm 3 Alt-Search Rule

Require: Empty lists: `dtrs`, `mthrs`

```
Activate priority queue
Pop next complete parse of the sentence                  # via particular-pop
Particular-pop next complete parse of the sentence       # Search complete parses
if a complete parse was popped then
   push complete parse to priority queue
else                                                     # Search alternative sub-trees
   Get list of complete parses from chart
   Sort list of parses by decreasing likelihood
   for each parse in list of parses do
      Add parse to mthrs list
      while mthrs list is not empty do
         if priority queue is not empty then           # Abort condition
            Terminate rule
         end if
         for each edge in mthrs list do                # Breadth-first search of tree
            Append all daughters of edge to dtrs list
         end for
         for each daughter edge in dtrs list do        # Process current tree depth
            repeat
               Particular-pop edge with same start, end and LHS as daughter
               Push new edge to priority queue         # Skipped if no edge returned
            until no edge is returned by particular-pop
         end for
         Copy content of dtrs list to mthrs list
         Empty dtrs list
      end while
   end for
end if
```