Algorithms for AI and NLP (Fall 2010, Assignment 2b)

Goals

1. Optimize our generalized chart parser to scale better to the size of the PTB-derived grammar;
2. adapt the generalized chart parser to one-best probabilistic unpacking from the forest;
3. implement the ParsEval metrics and evaluate quantitatively the performance of the parser.

1 Improving the Scalability of the Chart Parser (50 Points)

For this session, our course set-up provides the model solution to the previous assignment as a starting package, called '`pcfg.lsp`'. Please revise and extend the code in this file.

(a) In our starting package, we have made the necessary changes for the chart parser to work with the revised representation of the grammar. Thus, in principle the code is ready to parse sentences using the large, PTB-derived grammar; however, in our current implementation, there are severe scalability issues.

To confirm the basic parsing functionality, try gently (i.e. with a short sentence), for example:

```
? (setf edges (parse '("It" "was" "Monday" ".")))
? (unpack-edge (first edges))
```

With a little bit of patience, the above example calls should complete and show meaningful results. However, with our current implementation it appears impractical to construct the parse forest for more complex sentences, nor is it really feasible to attempt to unpack exhaustively all complete trees.

In recent lectures, we have informally mentioned a few aspects in which our current implementation of the chart parser is naïve (or, some might say, overly cute). One approach to improving its scalability would be to think hard about relevant properties of our grammar and interactions with the various core elements of the parser. A more practical approach to finding the parts of the implementation that most need optimization is to invoke the Lisp profiler. Roughly speaking, *profiling* is the process of executing code in an environment that will try to keep track of how running time (or space allocation) is distributed across differents parts of the code, in our case different functions. To invoke the profiler, one can simply wrap an s-expression to be evaluated into a special form that activates profiling, e.g.

```
? (prof:with-profiling (:type :time)
    (parse '("It" "was" "Monday" ".")))
```

(b) At this point, we are primarily concernced with the forest construction phase; thus, for these optimizations, please ignore the unpacking phase. For testing purposes, our course set-up provides an additional file '`test.mrg`' (Section 23) of the Penn Treebank, which is the commonly used test data. Although we may not quite be able to make our parser process all sentences from this file (on commodity hardware), you should aim to make the `parse()` function sufficiently efficient to be called on any sentence from '`test.mrg`' of up to twenty tokens in length. While you are optimizing the code, try sentences (starting from our trivial example above) of increasing complexity; once the forest for longer sentences can be constructed in a matter of seconds (not minutes), write a function to extract the sequence of leafs (i.e. a list of strings) from the trees in '`test.mrg`' and run those below twenty tokens through the parser. Without unpacking, count the number of edges returned from the `parse()` function; by and large, you should only observe two different counts. Explain why this makes sense.

2 Probabilistic One-Best Unpacking (40 Points)

Even if we managed to optimize our chart parser to allow constructing parse forests for sentences up to some non-trivial length (in reasonable time and memory), seeing the massive ambiguities admitted by our PTB-derived grammar, exhaustive unpacking is inconceivable for all but the shortest parser inputs. Fortunately, typical applications of syntactic parsing are most interested in the most probable analysis (rather than in millions of possible analyses). In this part of the problem set, we will finally put to use the rule and lexeme counts that we worked so hard to acquire from the PTB files earlier.

(a) Augment our function `read-grammar()` to determine rule and lexeme probabilies, using the standard maximum likelihood estimation 'idiom' for conditional probabilities. To avoid costly multiplications (using expensive rational numbers), transpose the probabilities into logarithmic space (so-called *log-probabilities*), as discussed in the lecture. Recall that the following is true: $log\prod_i P_i = \sum_i \log P_i$.

(b) While we are at post-processing the raw grammar (as read from '`wsj.mrg`'), observe that a relatively large proportion of rules is seen with very low counts only. As there is reason to be mildly sceptical of such rules (which might reflect annotation inconsistencies in the training data, rather than true linguistic generalizations) as well as of our ability to estimate reliable probabilities for them, implement a *frequency threshold* on grammar rules: for some value n, we will choose to discard all rules that were observed less than n times in training. To get started, set n to a conservative value of 1, but once we have a complete parsing and evaluation pipeline, feel free to experiment with more aggressive frequency thresholding.

(c) Now that our grammar is a little smaller and all rules and lexemes have probabilities attached to them, we will need to augment the edge structure to keep track of the probabilities of entries in our chart. Ignoring ambiguity packing for the time being, the probability of an edge should reflect the probability of the tree represented by that edge, i.e. (conceptually) the product of probabilities of all lexemes and rules contributing to the tree. Go through the core functions of the chart parser and make sure that all newly created edges have their probabilities computed properly. For the corner case of edges created in chart initialization for the actual input tokens (i.e. the daughter edges to lexemes), assume that they are certainties, in other words each should have the largest possible probability.

(d) Now generalize the Viterbi algorithm to the parse forest, to be able to read off the most probably tree without additional search, once the complete forest is constructed. Observe that forest construction effectively is the equivalent to filling in the HMM trellis, such that the one addition to the management of per-edge probabilities (from part (c) above) is to decide on what to do in the face of ambiguity packing. Much like in the Viterbi algorithm for lattices (i.e. the HMM case), there will need to be a maximization step, combined with a record of which exact 'path through the forest' (i.e. combination of edges into larger edges) leads to the highest-scoring overall tree. This step will likely require adding another component to the *edge* structure, combined with a new function `viterbi()`, to read out (or, in a trivial sense, unpack) the one-best tree from the forest.

3 Evaluating our Parser (30 Points)

To get a better idea of how well our parser actually performs (on unseen inputs, i.e. sentences not contained in the training data), implement the ParsEval metric. To compute ParsEval scores, it might be convenient to have an auxiliary function that decomposes one tree into a set of labelled bracketings, where each bracketing $\langle C, i, j \rangle$ captures the syntactic category C assigned to the sub-string of input starting at i and ending at j. Note that it is customary to *not* include PoS tags (i.e. the preterminal nodes of the tree, or categories of lexemes) in ParsEval scores. In case it seems wasteful to you to explicitly enumerate two sets of bracketings only to count overlapping and non-overlapping elements, consider an alternate scheme of computing, for a pair of trees, the relevant counts.

(a) Parse all sentences of Section 23 of the PTB (i.e. the leafs of the trees in our file '`test.mrg`') using our chart parser, extract the one-best Viterbi tree, and compute ParsEval scores against the gold-standard trees. Recall that the combined F_1 measure in ParsEval is the harmonic mean of precision and recall of labelled bracketings. Owed to unknown words in the test data, our parser will likely fail to parse some sentences; what should be the contribution of these test inputs to the overall ParsEval scores?

(b) To put parser evaluation figures into perspective, it may be useful to construct a so-called *baseline*, i.e. a score reflecting what would be the result without the Viterbi step. Construct a naïve variant of one-best unpacking that effectively ignores all probabilities, i.e. simply returns the tree that the parser happened to find first. Does our probabilistic chart parser improve over this baseline in terms of ParsEval scores?

4 Towards a More Realistic Treebank Parser (30 Points)

There are a number of common optimizations in statistical parsing that our current implementation lacks. It is actually not uncommon for a state-of-the-art parser trained on the PTB to use about one second for parsing a (comparatively complex) input, but nevertheless our current parser probably is one or two order of magnitudes less efficient than cutting-edge statistical parsers. Furthermore, we still fail to provide any syntactic analysis when there is a single unknown word in our input, where unknown tokens include, for example, all names and numbers not observed in training.

(a) To improve robustness to unknown words, combine the chart parser with our HMM tagger. Recall that the tagger was trained on the exact same data and achieved a tagging accuracy of around 96 % on the unseen test data from Section 23. One could imagine at least two different ways of using the tagger to preprocess parser inputs: (a) for unknown words, i.e. input tokens for which there is no lexical entry in our grammar, the PoS of the token could be provided by the tagger; this should effectively make it possible to parse all sentences (within a suitable upper bound on input length, as before) from '`test.mrg`'; (b) to further improve parser efficiency, the tagger could be used to reduce lexical ambiguity, where lexical look-up in the grammar would effectively be replaced with the PoS sequence obtained from most probable path through the HMM. Experiment with both methods of coupling the tagger and parser, and report on your experimental findings.

(b) Another technique to improve the efficiency of the parser is so-called *chart pruning*. The basic idea is, for each chart cell, to discard partial analyses with very low relative probabilities. Typically, this is accomplished by assuming a cut-off beam θ, where for each cell edges whose probability is less than $1/\theta$ of the best-scoring edge in that same cell are discarded. Assume that, at some point during forest construction, for cell $\langle i, j \rangle$ there is an edge e_1 whose probability p_1 is the highest probability for all edges in that cell. The basic intuition in chart pruning is that a new edge e_2 in that same chart cell, with a probability $p_2 < 1/\theta p_1$, is very unlikely to give rise to a tree whose total probability is larger than any tree built using e_1. Think about the assumptions we are making here, and explain why it is possible in principle that a tree containing e_2 might end up with a higher probability than all trees containing e_1.

If you have not done so already, rework the global chart as an abstract data type providing the various types of efficient indexing we need for our chart parser. Add a suitable level of accounting of per-cell maximum probabilities, and then experiment with chart pruning and various levels of θ.

5 Submitting Your Results

Please submit your results for problem set (2b) in email to Johan ('`johanbev@ifi.uio.no`') and Stephan ('`oe@ifi.uio.no`') before the end of the day on Monday, November 29. Please provide all files that you extended (or created) as part of this exercise (e.g. minimally '`pcfg.lsp`'), including all code and answers to the questions above.