

File ID 344015
Filename 7: Parsing with episodic memory

SOURCE (OR PART OF THE FOLLOWING SOURCE):

Type Dissertation
Title The neural basis of structure in language: bridging the gap between symbolic and connectionist models of language processing
Author G. Borensztajn
Faculty Faculty of Humanities
Faculty of Science
Year 2011
Pages xv, 233
ISBN 90-5776-233-1

FULL BIBLIOGRAPHIC DETAILS:

<http://dare.uva.nl/record/400259>

Copyright

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use.

Chapter 7

Parsing with episodic memory

While the previous chapter described the episodic probability model and used it as a reranking system, in this chapter I will take the episodic framework one step further and develop a full episodic left corner chart parser. As a first step a non-episodic probabilistic left corner chart parser that computes prefix probabilities is introduced in reasonable detail. Subsequently I describe an episodic ‘spreading activation’ instantiation of the left corner chart parser (ELCCP) where episodic probabilities are computed on-the-fly at parse time through spreading trace activations from stored derivations to (the traces in) its states. I discuss an efficient Viterbi algorithm for dynamically computing the shortest derivation within the ELCCP, which is implemented and evaluated on the Wall Street Journal.

7.1 An Earley-style probabilistic left-corner chart parser that computes prefix probabilities

In this section and the following I will gradually build an episodic left corner chart parser for HPN. Recall from section 5.3.1 that top-down parsing is not an option for HPN, because no privileged TOP category nor any other labels are assumed, and there are no restrictions on binding between HPN units, hence the search could go on indefinitely. As a first step, this section describes an Earley-style probabilistic left-corner chart parser that computes prefix probabilities. The left corner chart parser described here follows in broad lines the work of van Uytzel

et al. [2001] (which unfortunately I learned about only after I had developed a very similar left corner chart parser myself), and it is based on the Earley chart parser [Earley, 1970] (see section 3.1.11) and the probabilistic version thereof [Stolcke, 1995] (see section 7.1.6).

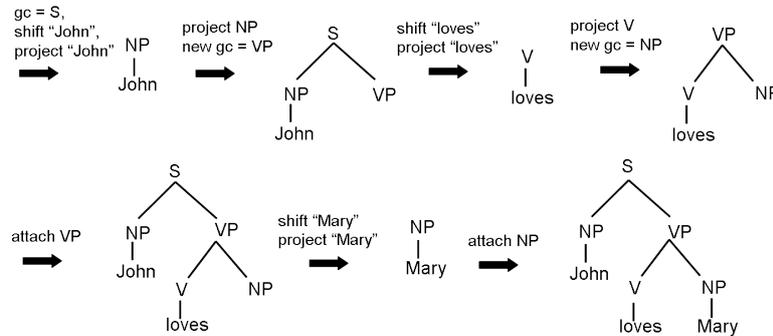


Figure 7.1: Left corner derivation of *John loves Mary*. *gc* is short for goal category.

Whereas in Stolcke [1995] a top-down parsing process is assumed (and a PCFG probabilistic model) the chart parser of van Uytsel et al. [2001] follows a left-corner parsing (LCP) strategy (see section 3.1.3), and a left corner probabilistic model, which will in general return different string probabilities than the top-down model. As discussed in section 3.1.3 the left corner parser has three operations: *shift*, *attach* and *project*. As a reminder, Figure 7.1 (repeated from section 3.1.3) illustrates the left corner parsing process.

7.1.1 States of the left corner parser

The LC parsing process is executed as a search through a network of *states* in a chart, where state transitions are effectuated by the *shift*, *project* and *attach* operations. States represent an instantiation of a grammar rule within a derivation: they are distinguished with respect to their position in the input string and their goal category, and are of the form

$$q = \{G; X \leftarrow_j \lambda \bullet_i \mu\} \quad (7.1)$$

where G is a goal category, $X \leftarrow \lambda \mu$ is a grammar rule¹, and λ and μ are (possibly empty) strings of terminals and nonterminals. The position of the dot indicates which daughters of the rule have already been processed in the state. If the dot is on the right of all daughters of a rule the state is called *complete*; otherwise, if it is not complete, it is called a *goal state*. The *left span index* j of state q

¹To prevent confusion I have adopted a notation where the arrow points left, instead of right, as customary in context free rewrite rules. The right arrow would be suggestive of a top-down prediction, whereas in left corner parsing the rule is accessed first through bottom-up projection.

($lspan(q)$, for short) corresponds to the word position in the sentence when the rule was first *projected* (i.e., words x_0, \dots, x_{j-1} have been processed before the projection); the *right span index* i ($rspan(q)$, for short) denotes the word position of the dot with respect to the input string. In plain language, $X \leftarrow_j \lambda \bullet_i \mu$ means that the processed part of the rule spans $\langle j, i \rangle$ in the input string.

Most probabilistic left corner parsers [e.g., Manning and Carpenter, 1997] compute the parse probabilities *given* the sentence (see section 3.1.7). If one is however interested in sentence or string probabilities (e.g., for a language model, or to compute prefix probabilities) this introduces some additional challenges, because a left corner derivation may produce many *unconnected* partial substructures (section 3.1.3). In order to compute the probability of a certain prefix one needs to make assumptions about how the substructures relate to each other in the generation process of the tree.

To deal with this problem one may postulate that in a LC derivation a goal state is followed by a special *word state* (a state of the parser after a shift), and that this transition is mediated by a *shift rule* from the *goal category* (the first unprocessed daughter of the goal state) to the word (for instance *loves* \leftarrow *VP* in Figure 7.1). With this assumption a LC derivation always stays connected, since it defines a unique and fixed linear order of processing. It is convenient to write the shift rule as

$$word \leftarrow G$$

where G is the goal category; the word state is then given as

$$\{G; word \leftarrow_i G \bullet_{i+1}\}$$

The left corner grammar augmented with a shift rule will be called *Left Corner Shifting Grammar* (LCSG).

7.1.2 Probabilistic left corner shifting grammar

A left corner shifting grammar can be extended to a probabilistic LCSG (PLCSG) by assigning a probability to every operation. Whereas in a PCFG there is only one type of probability, associated with top-down prediction and conditioned on the parent in the tree, in the PLCSG there are three types of probabilities, corresponding to different parser moves (project, attach and shift). The left corner probabilities are conditioned on previous steps in the derivation (history-based parsing), rather than based on the tree structure.

Often one or two prominent features are selected from the history to condition upon. For instance, Manning and Carpenter [1997] condition the probability of a projection and of an attachment on the parent category (Y) of a completed rule and the goal category (G) in the stack of the parser. Their example will be followed in the LCSG parser, but in a later section I will show that an *episodic* LCSG parser is able to condition on the entire history of the derivation.

As discussed in section 3.1.7, given a *complete* state with parent category Y , and given a goal category G , one can either project a rule $r : Z \leftarrow Y \alpha$ with left corner Y , or attach to a goal state if its first unprocessed daughter G equals Y , hence

$$P_{att}(Y, G) + \sum_{Z, \alpha} P_{proj}(Z \leftarrow Y \alpha | Y, G) = 1 \quad (7.2)$$

If there are no complete states then one must shift from a goal state to the next word in the sentence, conditioned on the goal category G (the first unprocessed daughter of the goal state). The *shift probability* is therefore given as

$$\sum_{word} P_{shift}(word | G) = 1 \quad (7.3)$$

The above probabilities can be estimated from the treebank using relative frequency estimation, after converting the treebank parses to their left corner derivations. Given shift probabilities one can calculate the joint probability of the derivation and the sentence, which is given by

$$\begin{aligned} P(der_{lc}, S) = \prod_{shifts} P_{shift}(w | G) &\times \prod_{attachments} P_{att}(Y, G) \\ &\times \prod_{projections} (1 - P_{att}(Y, G)) \times P_{proj}(r | Y, G) \end{aligned} \quad (7.4)$$

7.1.3 Probabilistic left corner chart parsing

As one can define a dynamic programming chart parser for the probabilistic top-down Earley-style chart parser of [Stolcke, 1995], one can do so as well for the LCSG grammar, in a way that allows for computing prefix probabilities and find the most probable parse efficiently. In section 7.1.6 I gave definitions for the prefix probability, the forward probability (for short, P_{fw}) and the inner probability (for short, P_{inn}) for the top-down parser. These definitions can be accommodated for the left corner parser, taking into account the different branching process of the left corner search. Given a state q as defined in Equation 7.1, then one defines

Forward probability The forward probability $P_{fw}(q)$ is the sum of the probabilities of all constrained paths of length j that end in state q , start in the initial state and generate $x_0 \dots x_{j-1}$.

Inner probability The inner probability $P_{inn}(q)$ is the sum of the probabilities of all paths of length $k - j$ generating the input symbols $x_j \dots x_{k-1}$, ending in q and starting with a *shift* of w_j .

(The prefix probabilities can be easily computed from the forward probabilities, as will be discussed in section 7.1.4.) One may also make use of the LCSG chart structure to efficiently find the most probable parse of the sentence (a.k.a.

the *Viterbi parse*). We define the Viterbi probability of a state, $P_{Vit}(q)$, as the probability of the most probable path (the *Viterbi path*), constraint by the sentence string, that goes through state q . To find the Viterbi path, the states in the chart must update their Viterbi probabilities, and keep track of a *Viterbi pointer* to the predecessor states associated with the Viterbi path. After completing the chart one can reconstruct the Viterbi parse from the final state by tracing back the pointers.

The LCSG chart parser moves through the sentence from left to right, and it keeps a set of states for each position in the input. Starting from state set 0, and as long as there are complete states in the current state set, the parser adds new states to the chart by exhaustively, and recursively performing *project* and *attach* operations on complete states.² Then it adds states to the next state set by *shifting* to the next input symbol from all goal states in the current state set.

A derivation starts with the initial state

$$q_I = \{TOP; TOP \leftarrow_{-1} SB \bullet_0 S\}$$

which is placed in state set 0 (SB is a dummy nonterminal that marks the Sentence Beginning).³ Since this is a goal state, the first operation will be a shift operation, invoking a shift rule $x_0 \leftarrow TOP$ (with x_0 the first word of the sentence). The parse completes when it reaches the final state

$$q_F = \{TOP; TOP \leftarrow_{-1} SB S \bullet_N\}$$

where N equals the number of words in the sentence.

Following are the details of the LCSG chart parser operations, specifying the conditions for adding new states to the chart, and the dynamic update rules for the forward, inner and Viterbi probabilities.

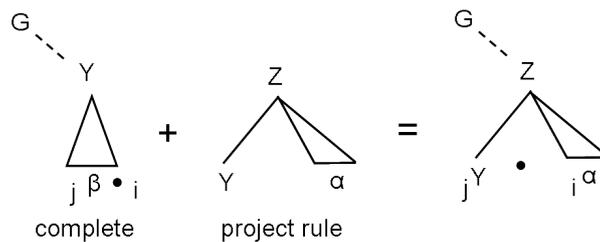


Figure 7.2: Project operation

- The *project* operation (Figure 7.2)

Given a complete state $C = \{G; Y \leftarrow_{j\beta} \bullet_i\}$ with parent category Y

²There are some intricacies involved here, concerning keeping the correct order and generating states recursively. These will be dealt with in section 7.1.6.

³This assumes that all parses must have a root category S .

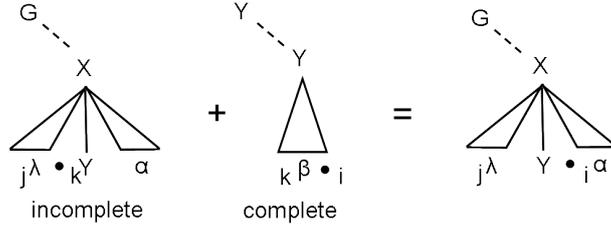


Figure 7.3: Attach operation

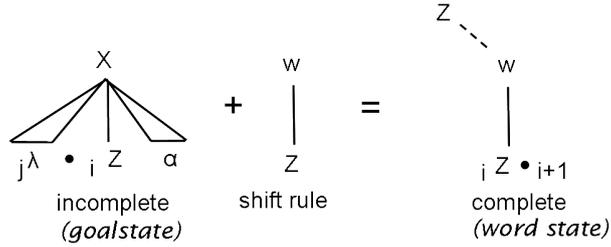


Figure 7.4: Shift operation

and a grammar rule $Z \leftarrow Y \alpha$ with left corner Y

If it does not yet exist, then create a new projected state

$\mathcal{N} = \{G; Z \leftarrow_j Y \bullet_i \alpha\}$ with the dot in the second position, and add it to the chart.

The forward, inner and Viterbi probability are updated according to:

$$P_{inn}(\mathcal{N}) += P_{inn}(\mathcal{C}) \times P_{proj}(r|G) \quad (7.5)$$

$$P_{fw}(\mathcal{N}) += P_{fw}(\mathcal{C}) \times P_{proj}(r|G) \quad (7.6)$$

$$P_{Vit}(\mathcal{N}) = \max((P_{Vit}(\mathcal{C}) \times P_{proj}(r|G)), P_{Vit}^{old}(\mathcal{N})) \quad (7.7)$$

The notation $+=$ means that the probabilities are set to the value on the right hand side of the equation if \mathcal{N} does not yet exist in the chart; otherwise their values are incremented with the same amount. If \mathcal{N} already exists in the chart, the Viterbi probability is only updated if it is higher than the state's previous Viterbi probability $P_{Vit}^{old}(\mathcal{N})$. In that case the Viterbi pointer is also replaced.

- The *attach* operation (Figure 7.3):

Given a complete state $\mathcal{C} = \{Y; Y \leftarrow_k \beta \bullet_i\}$ and an incomplete goal state $\mathcal{I} = \{G; X \leftarrow_j \lambda \bullet_k Y \alpha\}$, where $rspan(\mathcal{I}) = lspan(\mathcal{C}) = k$.

If it does not yet exist in the chart, create a new state

$\mathcal{N} = \{G; X \leftarrow_j \lambda Y \bullet_i \alpha\}$, in which the dot is moved behind Y .

Let $P_{att}(Y, G)$ be the probability of attaching a complete state with parent category Y to goal category $G = Y$. Then the inner, forward and Viterbi probability are updated according to

$$P_{inn}(\mathcal{N}) += P_{inn}(\mathcal{C}) \times P_{inn}(\mathcal{I}) \times P_{att}(Y, G) \quad (7.8)$$

$$P_{fw}(\mathcal{N}) += P_{inn}(\mathcal{C}) \times P_{fw}(\mathcal{I}) \times P_{att}(Y, G) \quad (7.9)$$

$$P_{Vit}(\mathcal{N}) = \max((P_{Vit}(\mathcal{C}) \times P_{Vit}(\mathcal{I}) \times P_{att}(Y, G)), P_{Vit}^{old}(\mathcal{N})) \quad (7.10)$$

The new state can be either complete (if $\alpha = \emptyset$) or incomplete (if $\alpha \neq \emptyset$). The rationale for updating the forward probabilities from the inner probabilities will be explained with an example in section 7.1.5.

- The *shift* operation (Figure 7.4):

Given an *incomplete* goalstate $\mathcal{I} = \{G; Y \leftarrow_j \lambda \bullet_i Z \alpha\}$, and given a shift rule $w \leftarrow G$

If it does not yet exist in the chart, create the complete state

$$\mathcal{N} = \{Z; w \leftarrow_i Z \bullet_{i+1}\}$$

$$P_{inn}(\mathcal{N}) = P_{shift}(w|G) \quad (7.11)$$

$$P_{fw}(\mathcal{N}) += P_{fw}(\mathcal{I}) \times P_{shift}(w|G) \quad (7.12)$$

$$P_{Vit}(\mathcal{N}) = P_{shift}(w|G) \quad (7.13)$$

To comply with [Stolcke, 1995] I will call a complete state that results from a shift a *scanned state*.

7.1.4 Prefix probabilities

Recall from section 7.1.6 that the prefix probability $P(S \leftarrow_L^* x)$ is the sum of the probabilities of all derivational paths starting with the initial state, that have $x = x_0, \dots, x_{k-1}$ as a prefix. Since all derivations of sentences with prefix x_0, \dots, x_{k-1} (and $|x| = k$) have to go through a scanned state $\{G; x_{k-1} \leftarrow_{k-1} G \bullet_k\}$, one may compute the prefix probability as the sum of the forward probabilities of all such scanned states. Hence,

$$P(S \leftarrow_L^* x) = \sum_{G: \{G; x_{k-1} \leftarrow_{k-1} G \bullet_k\}} P_{fw}(\{G; x_{k-1} \leftarrow_{k-1} G \bullet_k\}) \quad (7.14)$$

Figure 7.5: Treebank parses of *Peter runs* (PRN = pronoun).

7.1.5 An example that explains why inner probabilities are necessary

Suppose the left corner parser of van Uytsel et al. [2001] is trained on a treebank containing the two parses of Figure 7.5. Table 7.1 gives the states that the left corner chart parser goes through when parsing the sentence *Peter runs*.

nr	$op.$	$from$	$state$	p	$P_{fw}(\mu)$	$P_{inn}(\nu)$
q_1	-	-	$TOP; TOP \leftarrow_{-1} SB \bullet_0 S$	-	1.0	1.0
q_2	sh	q_1	$S; Peter \leftarrow_0 S \bullet_1$	$P_{sh}(Peter S) = 1.0$	1.0	1.0
q_3	pr	q_2	$S; NP \leftarrow_0 Peter \bullet_1$	$P_{pr}(NP Peter, S) = 0.6$	0.6	0.6
q_4	pr	q_2	$S; PRN \leftarrow_0 Peter \bullet_1$	$P_{pr}(PRN Peter, S) = 0.4$	0.4	0.4
q_5	pr	q_3	$S; S \leftarrow_0 NP \bullet_1 VP$	$P_{pr}(S, VP NP, S) = 1.0$	0.6	0.6
q_6	pr	q_4	$S; S \leftarrow_0 PRN \bullet_1 VP$	$P_{pr}(S, VP PRN, S) = 1.0$	0.4	0.4
q_7	sh	q_5, q_6	$VP; runs \leftarrow_1 VP \bullet_2$	$P_{sh}(runs VP) = 1.0$	$1.0(\mu_5 + \mu_6) \cdot P_{sh}$	1.0
q_8	pr	q_7	$VP; VP \leftarrow_1 runs \bullet_2$	$P_{pr}(VP runs, VP) = 1.0$	1.0	1.0
q_9	att	$q_5 + q_8$	$S; S \leftarrow_0 NP VP \bullet_2$	$P_{att}(VP, VP) = 1.0$	$0.6(\mu_5 \cdot \nu_8 \cdot P_{att})$	$0.6(\nu_5 \cdot \nu_8 \cdot P_{att})$
q_{10}	att	$q_6 + q_8$	$S; S \leftarrow_0 PRN VP \bullet_2$	$P_{att}(VP, VP) = 1.0$	$0.4(\mu_6 \cdot \nu_8 \cdot P_{att})$	$0.4(\nu_6 \cdot \nu_8 \cdot P_{att})$
q_F	att	$q_9 + q_{10}, q_{10} + q_1$	$TOP; TOP \leftarrow_{-1} SB S \bullet_2$	$P_{att}(S, S) = 1.0$	$1.0(\mu_1 \cdot \nu_9 \cdot P_{att} + \mu_1 \cdot \nu_{10} \cdot P_{att})$	$1.0(\nu_1 \cdot \nu_9 \cdot P_{att} + \nu_1 \cdot \nu_{10} \cdot P_{att})$

Table 7.1: Left corner chart of the sentence *Peter runs*. (Note that in the Table P_{fw} is abbreviated as μ and P_{inn} as ν .)

Why can one not compute the forward probabilities efficiently without inner probabilities? This has to do with the existence of non-local dependencies in the left corner derivations. One can see from the chart and from Figure 7.6, that after a shift to *runs*, both the path through state q_5 and the path through q_6 converge on state q_7 . Therefore, q_7 receives a contribution to its forward probability from both μ_5 and μ_6 . However, when the same paths attach back to q_5 and q_6 respectively (in q_9 and q_{10} respectively) they are again separated. Therefore, one cannot use the (summed) forward probability of q_7 in the calculation of both μ_9 and μ_{10} , because that would double the forward probability. The inner probability ν_7 of state q_7 however is common to both paths, so it can be used in the calculation.

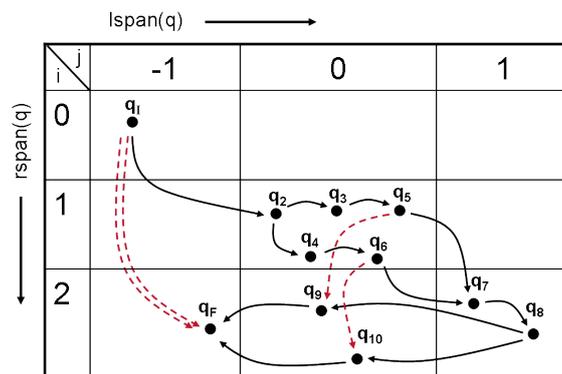


Figure 7.6: States and their transitions in the LCG chart (the q 's refer to Table 7.1).

7.1.6 Implementation issues

Retrieving the Viterbi parse

After constructing the chart, the Viterbi parse can be retrieved by following back the Viterbi pointers stored in the chart cells, starting from the final state. Stolcke [1995, p.22] describes a recursive procedure to recover the Viterbi parse for the Earley-style PCFG chart parser (section). The left corner version of the chart parser differs from the PCFG version in that the chart also stores states corresponding to shift transitions. These should however be ignored when retrieving the Viterbi parse, and treated as leaves of the tree. In their place the shifted word is inserted in the parse tree, but no recursive call is made.

Left recursion

In a top-down chart parser left-recursive *predictions* pose a problem when updating string probabilities, because the same state may appear several times within a given derivation, giving rise to so-called prediction loops. In left corner chart parsing the problem is less urgent, because in general left-recursive *projections* do not stay within the same state set but are followed by a *shift* to the next state set. Only *unary* projections could still directly and indirectly result in left recursion. Indeed, upon inspection of the Wall Street Journal one finds several such unary rules, for example $NP \leftarrow NP$ and $S \leftarrow VP$ and $VP \leftarrow S$. To deal with this problem we first remove all reflexive unary rules, such as $NP \leftarrow NP$ from the grammar. The remaining indirect left recursion can be dealt with by precomputing a matrix of probability sums for the reflexive, transitive unit-production relation [Stolcke, 1995, p.16]. An alternative, but only approximate solution is to restrict the number of successive unary projections. In this work the latter option was chosen, with a maximum of 2 successive unit projections.

Prioritized queues

When complete states from a certain state set are attached to a goal state, it is possible that the resulting state ends up in the same state set as the complete state. In this case one must take care that new states are added in a specific order to ensure that all contributions to a state's forward and inner probabilities are summed before that state is used as input to further projections or attachments within the same iteration. To achieve this, the newly derived state is inserted into a so-called prioritized queue [Stolcke, 1995, p.32]. States are ordered in this queue according to their left span index, from high to low, such that states that span a smaller part of the input sentence are completed before they are attached to states that contain them.

Unless the transitive unit projection matrix option is adopted, a similar problem may occur with unit projections (that occur within the same iteration as attachments), because unit projections always result in complete states in the same state set. To ensure that unit projections are processed in the correct order a second prioritized queue is created within the first one. In the latter queue unit projections are ordered according to their 'depth of projection', with a maximum of 2.

Beam search

To prevent the chart from growing too large states can be pruned from the chart. This restricts the search space, but may lead to a sub-optimal Viterbi parse, and under estimation of the inner and forward probabilities. One approach to pruning is to maintain a beam of states of a fixed size throughout the chart. States of a certain state set are ranked according to their forward probabilities, and if the maximum beam size is exceeded the states with the lowest forward probabilities are pruned from the state set. The optimal beam size should be assessed empirically. Again, one should take care that all contributions to a state's forward probability are accumulated before one decides whether to prune the state or not.

7.2 Evaluation of the basic probabilistic LCSG chart parser

The development of the PLCSG chart parser in the previous sections was not a goal in itself, but an intermediate step and base model on top of which an episodic left corner chart parser will be built in the next section. Still it is useful to have an idea of its performance before continuing further, because it provides a baseline for comparison to later versions. The PLCSG parser was evaluated on the standard task of parsing the Wall Street Journal, while measuring labeled precision and

recall. As usual, sections 2-21 were used for training the parser, and section 22 for development. Table 7.2 compares the performance of the PLCSG chart parser implemented in this thesis to that implemented by van Uytsel et al. [2001], and to a standard PCFG implementation with lexical and syntactic smoothing⁴ (but note that the other results are on section 23 of WSJ, while my results are on section 22). All reported results are for sentences up to length 40 (ca. 1700 in section 22), which took approximately 1.5 hours to parse.

Parsing model	LR	LP	F
This work	70.4	76.7	73.4
van Uytsel (2001)	79	79	79
PCFG	–	–	78.5

Table 7.2: Comparison of different implementations of the left corner parser with the PCFG parser.

Table 7.2 shows that the van Uytsel parser performs more than 5 percentage points better than my implementation of the PLCSG parser, and in the same range as the PCFG parser. Note however, that van Uytsel et al. [2001] did not implement smoothing (as a consequence his parser could not parse 4% of the sentences), while in this work (as well as in the PCFG) the train sentences were horizontally Markovized, and 2 levels of back-off smoothing were performed ($\lambda = 0.2$ for each level). Thus, the current implementation can parse all sentences, but the smoothing may have an overall negative impact on performance. Further, it should be noted that the scores reported by van Uytsel et al. [2001] were obtained with an advanced submodel of the parser, which included additional conditioning history to calculate the probabilities, whereas my implementation is based on the basic PLCSG probability model.

7.3 The episodic left corner chart parser

This section introduces the *episodic* left corner chart parser, an episodic extension of the probabilistic LCSG chart parser of the previous sections. I will discuss two versions of the episodic left corner chart parser, but I have implemented only the second version: the first version computes the most probable parse from the episodic traces, and the second version computes the shortest derivation, that is the derivation containing the least number of episodic fragments. In both cases training proceeds in the same manner as with the episodic reranker (see section 6.2.3): first, the treebank parses are converted to left corner derivations, and treelets are created for unique left corner productions (with specific register positions) in the treebank derivations (including treelets for shift productions).

⁴I am grateful to Federico Sangati for providing these results.

Then, for every left corner derivation in the treebank, and for every step in the derivation the treelet associated with the rewrite rule is filled with traces that encode the sentence number and position in the derivation (see the detailed algorithm in section 6.2.2).

The treelets filled with traces constitute an episodic grammar; the ‘rules’ of this grammar are thus no longer symbolic rewrite rules, but objects with a local memory where traces are contained. The treelet keeps track of the next required operation in the queue by means of a local register. The states of the LCSG parser are replaced by *treelet states*, which are of the form

$$q = \{G; X \leftarrow_j \lambda \bullet_i \alpha, E_q\} \quad (7.15)$$

Here the register position of the treelet is indicated by the dot, and G is again a goal category included in the state to enforce valid parses. The main difference with the LCSG states is that treelet states are enriched with a set E_q of (activated) traces. When a new state is added for the first time to the chart, all the traces are copied from the ‘treelet type’ to the treelet state, and receive a certain activation.

7.3.1 Most probable episodic parse

The most probable episodic parse is computed in much the same manner as the Viterbi parse in the LCSG chart parser, except that the base probabilities (P_{shift} , $P_{project}$ and P_{attach}) are not estimated beforehand from the treebank, but computed on the fly, as a function of spreading activation of traces in treelet states. This works as follows (but note that it has not been implemented):

When a new treelet state q is first added to the chart (as a result of a shift, project or attach operation) all traces from the ‘treelet type’ are copied to the treelet state. Then, for every trace separately, its common history (CH) is updated from a predecessor trace according to the update rule 6.2.3 (section 6.4), which is repeated here for convenience: Let q' be the predecessor state (either a complete state in case of project or attach, or a goal state in case of shift), and let $e_{q',i} = (s_{q',i}, n_{q',i})$ be the i^{th} trace in the predecessor state (associated with treelet $t_{q'}$), and $e_{q,j} = (s_{q,j}, n_{q,j})$ a trace in the current state q (associated with treelet t_q). (As before, s denotes the sentence number in the treebank, and n the position in the derivation.) Then

$$CH(e_{q,j}) = \begin{cases} CH(e_{q',i}) + 1 & \text{if } \exists (s_{q',i}, n_{q',i}) \text{ such that} \\ & (s_{q,j} = s_{q',i} \wedge n_{q,j} = n_{q',i} + 1) \\ 0 & \text{otherwise} \end{cases} \quad (7.16)$$

(i.e., the CH of the trace is incremented by 1 if there is a direct predecessor of the trace in the predecessor state.) When the same state is reached multiple times a weighted average is computed for the CH of its traces: Let $CH(e_{q \leftarrow q',j})$ denote

the common history of trace $e_{q,j}$ originating from the path through predecessor state q' . Then

$$CH_{average}(e_{q,j}) = \frac{\sum_{q'} CH(e_{q \leftarrow q',j}) P_{fw}(q')}{\sum_{q'} P_{fw}(q')} \quad (7.17)$$

(The rationale for weighing with the (non-episodic) forward probability is that it is a measure of the number of paths that have converged at q' .) Once the average CH's of the traces in a state are known, one can compute their activation $A(e_{q,j})$ according to Equation 6.1. Now it is straightforward to dynamically update the forward, inner and Viterbi probabilities in the chart, using Equations 7.5 to 7.13. In each of the latter equations the base probability (P_{shift} , $P_{project}$ or P_{attach}) is a function of the current state q , which is given by the relative fraction of traces in q that prefer moving to another treelet t_r (corresponding to a shift, project or attach operation), weighted by their activations. This probability was given in Equation 6.2, which is repeated here:

$$P_{episodic}(r|q) = \frac{\sum_{e_i \in E_q^r} A(e_i)}{\sum_{e_j \in E_q} A(e_j)} \quad (7.18)$$

As before, E_q^r denotes the set of traces in state t_q associated with the current state q that points to treelet t_r , and E_q is the full set of traces in state q .

One should take care that all incoming contributions to the CH's of traces in a state are considered for the averaged before continuing to update the CH's of traces in states further down in the derivation. This can be dealt with by a priority queue, as was explained in section 7.1.6.

There is a complicated issue concerning updating the common histories across a shift operation. This again has to do with non-local dependencies in left corner parsing: when you start a shift a new substructure is created for which it is not known yet which state it will attach to. A possible way around this problem is to keep track of an 'inner' CH, like the inner probability in the non-episodic LCSG parser, that starts to count from CH=0 (for every trace) at the shift operation. Upon an attach the 'inner' CH of traces in the complete state can be added to the total CH of traces in the goal state, as is done with forward probabilities.

The difficulty here is that, unlike the inner and forward probability, the 'inner' CH is not independent of the CH of the goal state. One cannot simply add the inner CH's of traces in the complete state to the CH's of traces in the goal state, because whether they should be added or not depends on whether the traces of the goal state have a successor trace after the shift operation. Can one not make the decision to add or not the 'inner' CH's at attach time, once the goal state is known? Unfortunately, it is not as simple as that. One of the complications is that the CH's are not additive: they can be set to 0 at any time during the shift loop. Things get more complicated if one also takes the average CH of contributing paths. As of writing this section the problem remains unsolved.

Remark about episodic parsing

Given that in the episodic grammar one may reconstruct a parse from sequences of successive traces pointing to treelets, one might wonder why it is actually still necessary to use a parser, rather than simply let the parse of the test sentence be decided by a Markov process between treelets? The reason is that there exist non-local syntactic (tree-)constraints that can not be captured by the Markov process. Without such constraints it could happen that, as soon as one combines treelets from different exemplars, in the resulting derivation a treelet state attaches to a state that has not occurred before in the derivation. The Earley chart structure guarantees that only those successions of treelets are allowed that constitute valid parses.

7.3.2 Shortest derivation parse

While in the *most probable* episodic parse traces receive activation values based on their common histories, in the shortest derivation parse the activation of a trace corresponds to the length of the shortest derivation up to the current state and trace. The Shortest Derivation Length of trace j in state q , abbreviated as $SDL(e_{q,j})$, is measured as the number of distinct episodic fragments that are used in the derivation path up to trace $e_{q,j}$ in state q .

Whereas in the previous chapter (section 6.3.2) I described a straight forward way to compute the shortest derivation of a parse tree in a greedy fashion, here I will develop a non-greedy dynamic programming approach that efficiently computes the shortest derivation in the left corner episodic chart parser. It uses a Viterbi-style algorithm that is very similar to the well-known algorithm for finding the most likely sequence of states in a Markov Model (MM) for language.⁵

Whereas in a MM successive time steps on the trellis correspond to successive words in a sentence, in the episodic grammar successive time steps correspond to successive treelets $\langle t_0, \dots, t_n \rangle$ in the derivation (see Figure 7.7). In the episodic grammar there is a distinct ‘state’ associated with every trace in a treelet, and one can define transition probabilities (or rather, transition costs) between all pairs of traces in treelet t_k and treelet t_{k+1} .

Let $e_{k,i} \equiv \langle s_{k,i}, n_{k,i} \rangle$ be a trace of a stored exemplar derivation, where s_i is the sentence number of the exemplar in the training corpus, and n_i is the position in the exemplar derivation. Define a transition cost C between two traces $\langle s_{k,i}, n_{k,i} \rangle$ and $\langle s_{k+1,j}, n_{k+1,j} \rangle$ of successive treelets t_k and t_{k+1} in a derivation

⁵The shortest derivation chart parser was developed independently to the work of Bansal and Klein [2011] to which it is very much related, although their work is formulated in a very different framework.

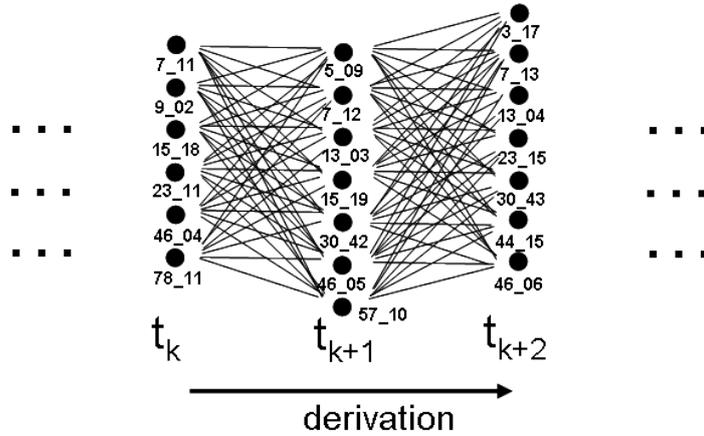


Figure 7.7: Trellis through the traces of the treelets in a derivation

$$C(\langle s_{k+1,j}, n_{k+1,j} \rangle | \langle s_{k,i}, n_{k,i} \rangle) = \begin{cases} 0 & s_{k+1,j} = s_{k,i} \wedge n_{k+1,j} = n_{k,i} + 1 \\ & \text{(direct successor)} \\ 1 & \text{otherwise} \end{cases} \quad (7.19)$$

This defines the aggregate ‘cost’ of any possible path through the trellis (i.e., through a sequence of treelets in a given derivation) in terms of the number of different episodic fragments used. In order to dynamically compute the shortest derivation (the one with the lowest cost), in every trace of treelet t_{k+1} one keeps a pointer to a single trace in the predecessor treelet t_k which, including the current transition cost, follows the path through the trellis with the lowest overall cost. Further, one stores the lowest overall cost (that is, the SDL) thus computed in the current trace. In the final treelet of the derivation one must then find the trace among all traces which has the lowest SDL, and follow the pointers back to the first treelet in the derivation to reconstruct the episodes used in the shortest derivation.

Computing the shortest derivation with discontinuous episodes

The Viterbi-algorithm for finding the shortest derivation can be generalized to the case that discontinuous episodes are allowed. I will briefly describe a possible implementation, but note that it has not been implemented or evaluated thus far, because it is computationally very expensive; the discontinuities give rise to a multiplication of the number of ‘trace states’ in the treelets for every ‘gap’ in an episode/fragment that one wants to account for. In the discontinuous shortest derivation case, one may reuse an exemplar that has been used earlier in the derivation, but that has been interrupted by fragment(s) from (an)other exemplar(s), at lower cost than using an entirely new exemplar (provided one

continues the earlier exemplar at a later position n_{k+1} in its derivation than where it was interrupted).

As an example I will describe a ‘trigram discontinuity model’, which can account for discontinuities that are a result of interrupting an exemplar by fragments from 2 other exemplars at most. In this model a ‘trace state’ corresponds to the final traces of three distinct exemplars that have been used in the final part of the shortest derivation. (To avoid clutter I left out the trace indices i and j .)

Define $(\mathbf{s}_k, \mathbf{n}_k) = \{\langle s_k^{-2}, n_k^{-2} \rangle, \langle s_k^{-1}, n_k^{-1} \rangle, \langle s_k^0, n_k^0 \rangle\}$. Here $\langle s_k^0, n_k^0 \rangle$ indicates the final trace of the last exemplar used in the shortest derivation, $\langle s_k^{-1}, n_k^{-1} \rangle$ indicates the final trace of the second most recently used exemplar, etc. Then one can define a transition cost function which charges a (small) prize for reusing one of the 3 most recently used exemplars in the shortest derivation, and which takes their relative order into account

$$C(\langle s_{k+1}, n_{k+1} \rangle | \langle \mathbf{s}_k, \mathbf{n}_k \rangle) = \begin{cases} 0 & s_{k+1} = s_k^0 \wedge n_{k+1} = n_k^0 + 1 & (\text{direct successor}) \\ 0.1 & s_{k+1} = s_k^0 \wedge n_{k+1} > n_k^0 + 1 & (\text{same exemplar w. gap}) \\ 0.2 & s_{k+1} = s_k^{-1} \wedge n_{k+1} > n_k^{-1} & (\text{second most recent}) \\ 0.3 & s_{k+1} = s_k^{-2} \wedge n_{k+1} > n_k^{-2} & (\text{third most recent}) \\ 1 & \text{otherwise} & \end{cases} \quad (7.20)$$

As before, for every trace in treelet t_k one selects a single trace state in the previous treelet, which minimizes the overall derivation length, and stores a pointer to it. Subsequently, one updates the states of all traces in treelet t_{k+1} , such that they include the final traces of the 3 most recently used exemplars, and one stores the aggregate cost in the updated states.

7.3.3 Shortest derivation left corner chart parser

Having explained how the Shortest Derivation Length (SDL) is computed *given* a derivation, it is only a small step to implement it in a chart parser. The basic control structure is again the non-episodic (probabilistic) left corner Earley parser, that uses shift, project and attach operation to fill a chart. When moving from trace $e_{q',i} = \langle s_{q',i}, n_{q',i} \rangle$ in predecessor state q' to trace $e_{q,j} = \langle s_{q,j}, n_{q,j} \rangle$ in the new state q , the transition cost $C(e_{q,j} | e_{q',i})$ is computed as in Equation 7.19, replacing the treelets t_{k+1} and t_k by t_q and $t_{q'}$ respectively.

To compute the SDL of trace $e_{q,j}$ one must find the trace $\widehat{e}_{q',i}$ in the predecessor state q' that, including the transition cost to the current trace, yields the shortest derivation length

$$SDL(e_{q,j}) = \min_{e_{q',i}} (SDL(e_{q',i}) + C(e_{q,j} | e_{q',i})) \quad (7.21)$$

After this is found, the SDL is stored in trace $e_{q,j}$, together with a pointer to trace $\widehat{e}_{q',i}$ in state q' , that will be used to reconstruct the Viterbi path going

through trace $e_{q,j}$. Once the chart is constructed one can reconstruct the shortest derivation parse as usual: in the final state one selects the single trace with the shortest derivation length, and starting from this trace one follows the (trace-level) Viterbi pointers back through the predecessor states and traces until the start state.

Updating the shortest derivation in the chart

In case more than one derivation passes through the same state q (i.e., q has multiple predecessor states q' in the chart), one compares for every trace $e_{q,j}$ in q the SDL of path through the most recent predecessor state q' and trace $e_{q',i}$ with the best SDL thus far, as one would do with the Viterbi probability. If the current SDL is lower one replaces the SDL of trace $e_{q,j}$ as well as its pointer with a pointer to $e_{q',i}$ and state q' . Note that unlike the episodic Viterbi probability, which is associated with the state as a whole, the SDL is associated with a trace, hence within a single state distinct traces may point to different predecessor states.

The updates of the SDL 's of the traces follow the control structure of the LCSG chart parser described in the previous section; Below I give the update equations for the shift, project and attach operators, where the following notation is used (referring to Equations 7.5 to 7.13): $SDL(e_{\mathcal{N},j})$ is the SDL of the j^{th} trace $e_{\mathcal{N},j}$ in the new state \mathcal{N} ; $SDL(e_{\mathcal{C},i})$ and $SDL(e_{\mathcal{I},k})$ are similar definitions for traces in the complete state \mathcal{C} and the goal state \mathcal{I} ; $\mathbf{update}(SDL(e_{\mathcal{C},i}), \mathcal{N})$ is an abbreviation of the update Equation 7.21 (involving a transition cost function).

- The *shift* operation (Figure 7.4):

$$SDL_{inn}(e_{\mathcal{N},j}) = 0 \quad (7.22)$$

- The *project* operation (Figure 7.2)

$$SDL_{inn}(e_{\mathcal{N},j}) = \mathbf{update}(SDL_{inn}(e_{\mathcal{C},i}), \mathcal{N}) \quad (7.23)$$

- The *attach* operation⁶ (Figure 7.3):

Let $SDL_{min}(\mathcal{I})$ be the length of the shortest derivation to reach any of the traces in the goal state \mathcal{I} ; Upon attach, the SDL of a trace $e_{\mathcal{N},j}$ in the new state \mathcal{N} is updated according to the equations below

$$SDL_{inn}(e_{\mathcal{N},j}) = SDL_{min}(\mathcal{I}) + \mathbf{update}(SDL_{inn}(e_{\mathcal{C},i}), \mathcal{N}) \quad (7.24)$$

⁶Note that this is only an approximation of the actual shortest derivation, because the cost of the shift transition is ignored. See the end of this section for a discussion of the shift problem.

Care should be taken that all incoming paths to a state are considered as candidates for the *SDL*'s of the state's traces before continuing to update *SDL*'s in states further down in the derivation. To this end a priority queue is used, as was explained in section 7.1.6.

Tie breaking

In case of ties with respect to the *SDL* of trace $e_{q,j}$ in state q (i.e., there is more than one predecessor trace that gives rise to the same *SDL*) one can resort to a tie-breaking heuristic, that selects one predecessor trace $e_{q',i}$ in state q' from among all predecessor traces that result in the same *SDL*. The tie is broken in favor of the trace with the highest Viterbi probability of the path through $e_{q',i}$ and leading up to $e_{q,j}$. Note that within a certain state of the chart every trace has its *own* Viterbi path and Viterbi probability. (Recall that traces from one state may store pointers to different predecessor states, depending on their shortest derivation.) Thus, the Viterbi probability is computed for every trace individually according to the non-episodic LCSG probability model, irrespective of the overall Viterbi path and probability associated with the state.

The Shift problem

As was discussed in section 7.3.1, for the shortest derivation parse, too, there is an issue that the updates of the *SDL* are complicated by non-local dependencies: the attach operation must operate on a goal state that has been derived earlier in the path of the same derivation. If one would naively update the *SDL* after every shift, project and attach operation one might end up with a shortest derivation that is not a possible path (namely, by attaching an episodic fragments to a goal state that is not in the path). To deal with this problem one must keep track of an 'inner' *SDL*, which is set to 0 for every trace upon a shift operation. Upon attach the 'inner' *SDL* is added to the total *SDL* of the goal state.

However, still the solution is not complete: if one resets the 'inner' *SDL* of every trace to 0 upon a shift operation, one ignores the fact that a shift sometimes continues an existing episode, and sometimes starts a new episode. Yet, which case applies is only known at the time of attach, because of the non-local dependencies. Unfortunately, I have thus far not been able to address this problem in an efficient manner. Therefore, the current implementation, pertaining to the results reported in the experimental section, ignores switch costs at the shift transition, hence it cannot be guaranteed to always find the shortest derivation.

7.3.4 Implementation issues of the shortest derivation parser

- **Duplicate sentences are not allowed.** When parsing a new sentence with the shortest derivation, the trivial solution, which uses a parse of

the identical sentence as the sole fragment in the derivation, should be avoided. To this end, before parsing duplicate sentences are removed from the exemplar set.

- **Time complexity.** The time complexity of parsing a sentence with the shortest derivation LCSG parser is $O(n^3XN)$, where n is the sentence length, and N is the number of sentences in the train corpus. The standard left corner chart parser has time complexity $O(n^3)$, because there are in the order of n^2 cells in the chart, and every chart cell can be derived in the order of n ways. The episodic left corner chart parser further iterates over the traces in a state, whose number is in the order of N (if trained on the full WSJ, certain states may contain up to 100,000 traces).
- **Improving the efficiency.** To make the updates of the transition costs in Equation 7.19 more efficient, one may choose to consider as candidate predecessor traces only those traces in a predecessor state that have minimal SDL, since other traces can never give rise to shorter derivations. The ‘minimal SDL traces set’ (in short *minSDLset*) of a state can be computed beforehand, and this needs to be done only once for every state. Although this substantially improves efficiency, the downside of this option is that it introduces a certain bias because it may miss a candidate trace outside the *minSDLset* if it happens to be the direct predecessor trace of the current trace (hence has zero transition costs, whereas the traces in the *minSDLset* have transition cost 1). The latter trace is excluded from tie-breaking even though it may have across the board the same SDL as traces in the *minSDLset* of the predecessor state. Thus, the heuristic prefers traces in the *minSDLset* that do a ‘late switch’ (i.e., to a different fragment) over a trace that has switched earlier.
- **Goal categories in treelets.** Although the treelet *states* are distinguished by a goal category, in theory a goal category should not be included with a treelet *type*. However, empirical tests indicate that adding a goal category to treelets hardly impacts performance, while on the other hand it significantly speeds up the parser (because the same number of traces is distributed over many more treelets).
- **Sampling at the trace level.** If one is interested to introduce some randomness into the system but still preserve the shortest derivation, one may choose to sample among predecessor traces that participate in a tie-break. Sampling in the episodic parser is different than standard sampling because it is done at the trace level, rather than at the state level (recall that every trace defines its own Viterbi probability and path). Sampling will become important in the next chapter when the episodic parser is invoked

in learning, and a certain amount of noise is needed to break the symmetry of the parses.

7.4 Experiments with the shortest derivation left corner parser

The episodic left corner shortest derivation parser (ELCSD) was trained on the Wall Street Journal, sections 2-21, and tested on section 22. Labeled precision and recall was computed for the shortest derivation parses, with 2 levels of probabilistic tie-breaking, as described above. As the research is still in its developmental phase I present only preliminary results; the parser was trained only on the first 50 % of the WSJ train set, and tested only for sentences up to length 20. This took approximately 60 hours for the 709 sentences of section 22. To have a fair baseline, the standard (non-episodic) probabilistic LCSG chart parser was also trained on 50 % of the WSJ. Table 7.3 summarizes the results. (See Table 7.2 for the results of the non-episodic left corner parser trained and tested on the full WSJ.)

Parsing model	LR	LP	F	EM
ELCSD	82.3	81.1	81.7	26.9
SL-DOP [Bod, 2003]	90.7	90.8	90.7	—
AFG [Bansal and Klein, 2011]	—	—	86.9	31.5
PLCSG (baseline)	76.3	81.2	78.7	16.8
van Uytsel (2001)	79	79	79	—

Table 7.3: Comparison of the episodic left corner shortest derivation parser (ELCSD) with state-of-the-art shortest derivation parsers, and with the baseline PLCSG and the van Uytsel left corner parser. AFG= all-fragments grammar; SL-DOP=Simplicity-Likelihood-DOP. Note that all other parsers were tested on section 23, and on sentences up to length 40, while the ELCSD and the baseline were tested on section 22, and sentence length ≤ 20 .

From Table 7.3 it is clear that the ELCSD parser is not (yet) competitive with the state-of-the-art, but it outperforms the baseline PLCSG, and even the more sophisticated implementation of a PLCSG by van Uytsel et al. [2001] by a significant margin. Further, it should be noted that the results of Bansal and Klein [2011] were obtained after a coarse-to-fine pruning preprocessing step, and that their simple shortest derivation implementation scored badly, with $F = 66.2$. The high variance in the F-scores of our results ($\sigma = 18.7$) gives hope that a similar approach could also work for the ELCSD parser (see section 7.4.1).

7.4.1 Coarse-to-fine parsing and pruning

An example from the WSJ illustrates that too heavy a reliance on the shortest derivation for parsing a novel sentence can sometimes have a detrimental effect on the accuracy of the parse. Figure 7.8 (b) shows that the shortest derivation parse has literally copied a fragment from train sentence 11121 containing an analysis of the words *an average* as separate constituents. However, this analysis of *an average* in the shortest derivation parse is highly unusual. As a result it scores much worse (F=0.43) than the standard PLCSG model parse (b), which scores F=1.0.

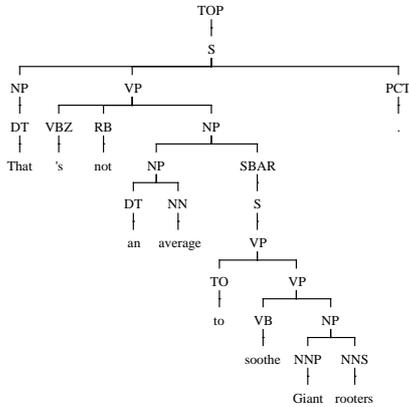
Ideally, one would like to avoid the use of fragments in the shortest derivation that occur only rarely in the train set. From the results of Bansal and Klein [2011] it can be learned that coarse-to-fine parsing has a surprisingly large positive effect on the shortest derivations, because it filters out idiosyncratic exemplars with low frequency. It is expected that pruning the states of the chart by using the forward probability of the LCSG probability model will have a similar effect, and it can be done in a single pass through the data. This is left for future work.

7.4.2 Chapter conclusion and discussion

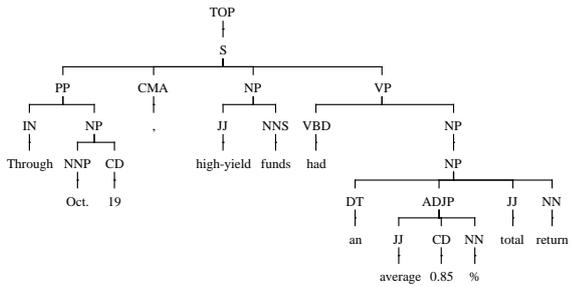
Whereas in the previous chapter the episodic grammar was implemented as a reranking system, meaning that its performance depended on a third party parser, the current chapter demonstrated that episodic parsing is a viable approach in its own right, and computationally tractable. In the preliminary evaluation its performance is lower than state-of-the-art, but there are reasons to be optimistic about the future, in particular if the full power of episodic parsing can be unleashed, once we succeed to solve the problem of the shift transitions. Otherwise, it is currently the best performing left corner parser, and hopefully will restore confidence in left corner parsing as an attractive, cognitively plausible alternative to top-down parsing.

Relation to other work

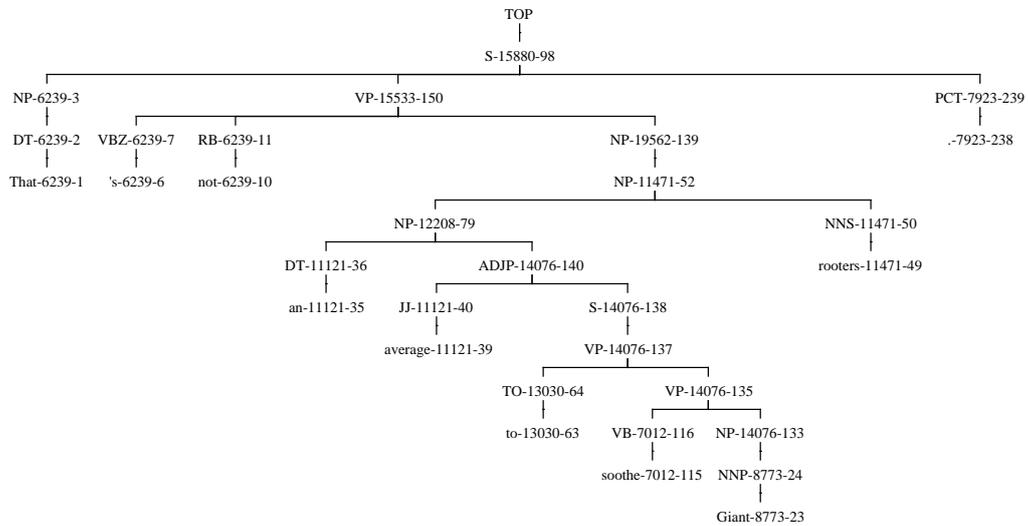
The episodic shortest derivation parser bears a strong resemblance to the work of Bansal and Klein [2011], even though it is formulated in quite a different framework, and it was developed independently from that work. Their All-Fragments Grammar (AFG) uses a technique known as Goodman Reduction [Goodman, 1996], and takes it to its extreme: every local subtree in a parse tree of the train corpus defines a unique rule ($X_p \rightarrow Y_q Z_r$), with unique labels. As a result the AFG grammar has very many specific rules, but only 2 general rule schemes: *CONTINUE* ($X_p \rightarrow Y_q Z_r$) and *SWITCH* ($X_p \rightarrow X_q$). *CONTINUE* follows unique rules along training trees, whereas *SWITCH* changes between trees. In the shortest derivation application, a *CONTINUE* rule is associated with a transition costs of 0, whereas the *SWITCH* rule has associated cost 1.



(a) Gold Standard parse



(b) Fragment of train parse tree used in the shortest derivation



(c) Shortest derivation parse

Figure 7.8: Illustration of where the shortest derivation parse goes wrong. (a) Gold Standard parse. (b) Shortest derivation parse. The numbers in the nodes of the tree represent the traces: 11121-39 indicates that this fragment originates from position 39 in the derivation of sentence number 11121 in the train set. (c) Fragment of train parse tree (sentence 11121) used in the shortest derivation

An interesting insight can be obtained if one realizes that the unique local trees of the Goodman reduction in AFG can in fact be interpreted as traces, because they point from one rule in a certain exemplar to a unique successor rule in the same exemplar. For this reason the AFG approach is probably broadly equivalent to the shortest derivation version of the episodic grammar. Yet, the episodic grammar also allows for defining alternative probability models over the traces, such as the probabilistic episodic grammar proposed in section 7.3.1. Further, it is important to note that, by contrast to AFG, episodic-HPN is a processing model: the order of traversal through the path in a derivation is a determining factor in the search for its shortest derivation. Another difference, of course, is that the current work is implemented as a left corner parser, whereas Bansal and Klein [2011] assume a top-down parsing strategy.

Simplicity-DOP, or S-DOP, is the DOP implementation of a shortest derivation parser [e.g., Bod, 2003, 2000]. It works by assigning equal probabilities to all DOP-fragments, large or small, such that the system will exhibit a preference for parsing with a minimal number of fragments. Bod [2003] proposes two variations of the tie-breaking heuristic: Simplicity-Likelihood-DOP, or SL-DOP, selects the shortest derivation tree among the n likeliest trees, while Likelihood-Simplicity-DOP, or LS-DOP, selects the likeliest among the n shortest derivation trees. As Table 7.3 shows, SL-DOP is currently the best performing system among the shortest derivation parsers. For a further comparison between DOP and the episodic grammar framework please refer back to section 6.4.

Interpretation of the episodic grammar as a syntactic network

To conclude this chapter let me bring back in the reader's mind the original motivation behind the episodic grammar agenda, which was to find a solution for syntactic parsing within a connectionist framework. To this end I demonstrated that in the episodic grammar all conditioning events can be accessed locally (i.e., the contextual history is content-addressable through the traces), in line with the constraints of a connectionist design. I mentioned before that the treelets of the episodic grammar fulfill the role of traditional grammar rules. They should be regarded as physical network units, that possess a local memory containing the traces. Further, the treelets possess a local register to keep track of which of their children has last been processed. The entire set-up is compatible with a view of episodic grammar as a physical network, consisting of multiple autonomous treelets that work together to produce the macro-behavior of a syntax, without central control. In this view a treelet is a kind of micro-processor, that locally enforces the correct order of execution of a sequence of operations through the register (i.e., starting with a projection, and followed by zero or more shifts and attachments).

While in the current chapter the treelets were created by copying rules from an annotated treebank in a supervised manner, and identified by symbolic labels,

in the next chapter the labels will be removed as well, as the episodic grammar will be integrated with HPN. The treelets will then be replaced by compressor nodes, and the labels by vectors in a high-dimensional substitution space, which learn their position in the space from experience. Hence the episodic framework generalizes to a *connectionist* syntactic network that implements a left corner parser, and learns in a fully unsupervised manner.