1 Three versions of the episodic left corner shortest derivation chart parser

I have implemented three versions of the episodic left corner shortest derivation (ELCSD) chart parser. The first version, which I used in my PhD (and which achieved an F-score of 81.7 on section 22 of the WSJ (≤ 20)), breaks the fragments after every shift operation (i.e., it resets the derivation length of all traces to 0 in a shift state). The second version greedily looks for the shortest derivation, even across a shift operation. However, because it is greedy it may sometimes find a 'local optimum' and miss the globally shortest derivation (see the examples below). Both the first and the second version are very fast (linear in the number of traces per treelet X the number of states). The third and final version is supposed to find the globally shortest left corner derivation using dynamic programming. Implementation details are given in section 3. This version is computationally much more expensive (quadratic in the number of traces per treelet X the number of states).

2 Some examples of shortest derivations with different versions of the algorithm

The first version of the ELCSD parser, when trained on the treebank of Figure 1,

(2x)

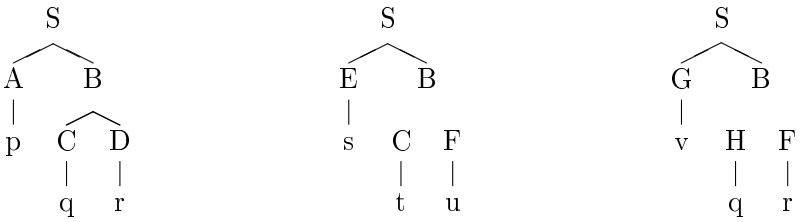

Figure 1: Training set

fails to find the shortest left corner derivation of the sentence *p q u*. The correct shortest left corner derivation, shown in Figure 2, and with length 2, combines a left corner fragment (including a shift operation, as indicated by the =-sign) from the first sentence with a fragment from the second sentence.

However, the first version of the ELCSD parser will assign derivation length 3 to this parse, because it breaks the first fragment in two at the shift to the word *q*. It will prefer the derivation of Figure 3, which also has derivation length 3, but a higher

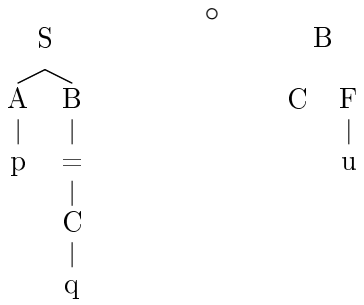

Figure 2: Shortest derivation (length=2)

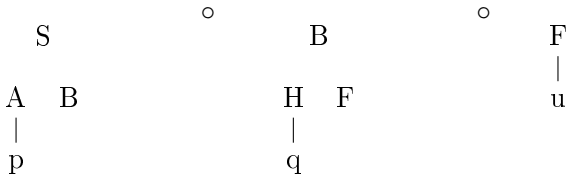

Figure 3: 'Shortest' derivation found by version 1 (length=3)

probability (based on the frequency of the fragment from the third sentence). The second, greedy, version does find the correct shortest derivation in this case.

However, the greedy version fails to find the shortest derivation in the example of Figure 4 (which shows the training set), when parsing the sentence *b r s t u*.

The actual shortest derivation is given in Figure 5 (using a discontiguous fragment from sentence 1). Instead of this derivation, the second, greedy version of the parser finds a local optimum, as shown in Figure 6. It was verified that the third (supposedly correct) version of the ELCSD parser does find the globally shortest derivation of the same sentence.

A final, slightly more complex example is given in Figure 7. Here the globally shortest derivation of the sentence *b r s i j v w* would be as in Figure 8 (using a discontiguous fragment from sentence 1), and this is indeed found by the third implementation of the parser.

Yet, the second (greedy) version prefers the derivation of Figure 9, which has derivation length 4, because of the higher frequency of the fragments in the second and third sentences.

(2x)

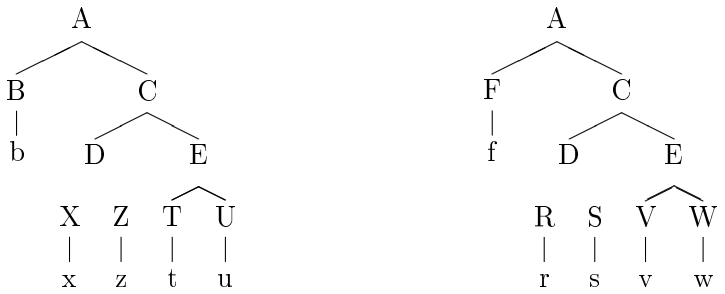

Figure 4: Training set

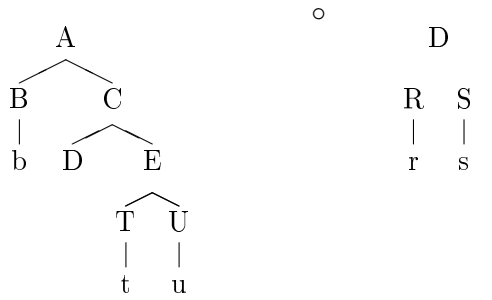

Figure 5: Correct shortest left corner derivation (length=2)

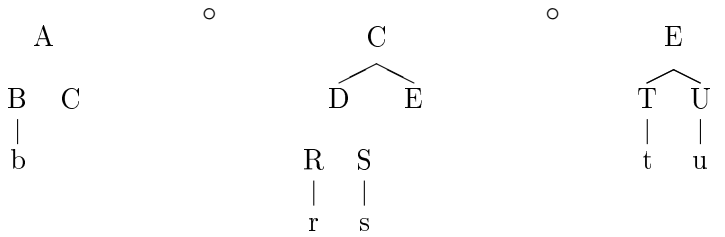

Figure 6: 'Shortest' derivation found by the greedy version of the parser (length=3)

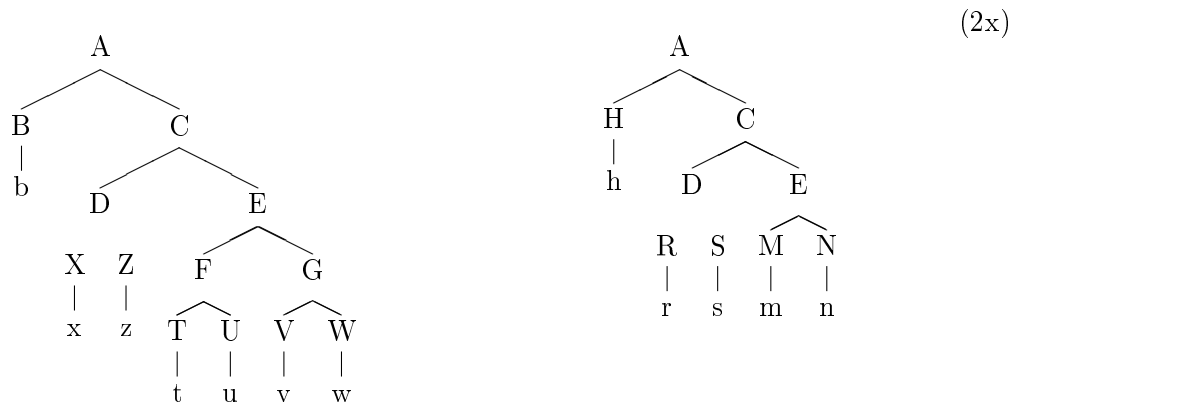

Figure 7: Training set

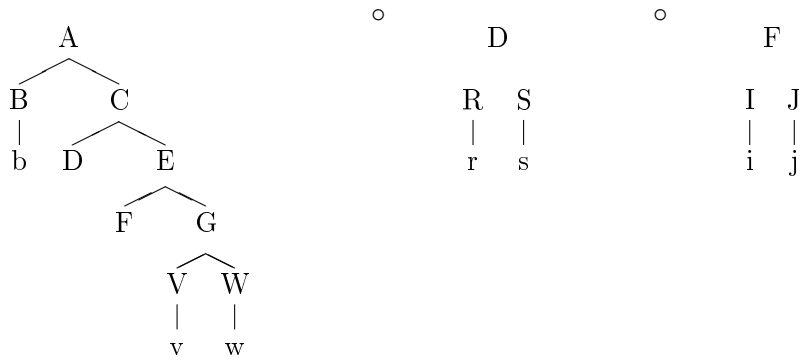

Figure 8: Globally shortest derivation of *b r s i j v w* given the training set of Figure 7.

Figure 9: 'Shortest' derivation of *b r s i j v w* according to the greedy algorithm

3 Implementation details for the full version of the ELCSD chart parser

Following is a short and informal explanation of how the full (third) version of the ELCSD chart parser works (a more formal description including pseudo-code will follow). It is a modified version of the episodic left corner shortest derivation parser algorithm described in my thesis (on page 154, section 7.3.3)[1]. As described in my thesis, in order to efficiently computate the shortest derivation length (SDL) (using dynamic programming), the parser makes use of an 'inner SDL' (by analogy to the inner, or inside probability), which is reset to zero after every shift operation (i.e., at the start of a shift loop), and added to the full SDL of every derivation that shares the shift loop upon attach. This way, the derivation length of the shift loop needs to be computed only once, while the same shift loop can be used multiple times for different (partly overlapping) derivations. However, the SDL of the shift loop evidently does not take into account the 'switch costs' (0 or 1) for the transition between an incomplete *goal state* to a so-called *word state* (the state after a shift), and neither does it take into account the switch cost of the attach at the end of the shift loop, because these costs depend on a particular derivation going through a particular goal state before the shift operation. In the dynamic programming approach these dependencies are not included when one computes the SDL of the shift loop. Thus, only at attach time the parser can know the goal state (and trace) from which it started the shift loop, because only then the choice for a particular derivation (though a particular trace in the attach state) is made, and the dependency (on the free variable introduced by the goal state) is marginalized out.

Some clarification is needed here: although in the episodic grammar formalism in principle the treelet to which the parser attaches at the end of a shift loop may originate from a different sentence than the 'goal treelet' (from which the shift loop started), we will demand from now on that both treelets correspond to a single node in a parse tree of the training set. This ensures that every fragment can be analyzed as a subtree (i.e., that one obtains hierarchically structured, rather than linear fragments), and it ensures compatibility of the linear (Markovian) episodic process with the hierarchical parsing process. To enforce this constraint, during training of the episodic grammar one records pairs of traces that co-occur in a single node of the train parse tree (i.e., the first trace of the pair for the project operation, and the second one for the attach operation). When parsing a novel sentence one must then make sure that attachments are only allowed if the trace in the 'goal state' (i.e., the state that needs to be attached

[1]Note though that equation 7.24 is no longer accurate in the current version, that the so-called 'shift problem' (mentioned on page 156) is solved, and that in the new version discontiguous fragments are automatically discovered (no specialized algorithm is necessary, unlike what is proposed on page 153).

to at the end of a shift loop) and the 'attach state' (i.e., the state that is the result of an attach operation) belong to the same pair.

From the above constraint we have that upon attach every trace in the 'attach state' uniquely determines a single trace in the goal state (from which the shift loop started). This means that at attach one can simultaneously determine the switch costs for the shift transition and for the attach transition, for every derivational path separately, provided the trace in the 'word state' at the beginning of the shift loop is still known. (See Figure xxx). It also means that the uncertainty regarding the switch cost from the goal state to the word state (0 or 1) remains unresolved until the end of the shift loop. As a consequence, instead of a single best shortest derivation path, every trace must store multiple paths of equal SDL, namely a separate shortest derivation path to every trace in the word state (paths longer than the minimum SDL path may safely be discarded). Thus, instead of a single pointer to a predecessor trace (in a predecessor state), a trace must store multiple pointers (so-called *shiftPointers*), one for each possible end point in a different trace of the word state.

Updating the shiftPointers: In the projected state immediately after the shift, shiftPointers are initiated for every trace; depending on whether there exists a direct predecessor of the current trace among the traces in the word state, one must either store a single shiftPointer to that trace, or multiple shiftPointers to all traces in the word state. Upon further projections, for every trace in the new (projected) state one must update the shiftPointers to the shortest derivation paths, while keeping a separate shiftPointer for each path that ends at a different trace in the word state (as long as these paths correspond to the minimal SDL).

Finally, upon attach, all necessary information is locally available to decide on a single unique shortest derivation path (including the switch costs for the shift transition and for the attach transition of the shift loop). Thus, for every trace in the attach state one may choose a single pointer from among the shiftPointers of a trace in the predecessor state. After building the chart, when retrieving the shortest derivation from the pointers (in reverse direction), one must propagate the choice of the shift-Pointer made at the attach operation back all the way to the word state, meanwhile selecting a single pointer (determined at the attach) from the shiftPointers along the way.

I have for now left out from the discussion additional complexity resulting from tie-breaking, according to the Viterbi probabilities of derivations of equal length (and shift trace). This will be covered in a subsequent version of this manuscript.

4 Most probable episodic left corner parser

The episodic probabilistic left corner chart parser (EPLC) uses the same control structure as the van Uytsel left corner Earley parser. However, as opposed to the van Uytsel parser, the base probabilities (P_{shift}, $P_{project}$ and P_{attach}) are not estimated beforehand from the treebank, but calculated on-the-fly whenever a new state is created in the chart, as a function of spreading activation of traces in treelet states. The conceptual innovation is that rule expansion probabilities are no longer associated with 'static' conditioning events, but with states of the chart parser, as the transition probability depends on multiple derivations of the sentence (each with a different history) that end in the the current state. This works as follows:

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 following rule: 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) (where s denotes the sentence number in the treebank, and n the position in the derivation.) Then

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

(i.e., the CH of the trace is incremented by 1 if there is a direct predecessor of the trace in the predecessor state.) In case more than one derivation passes through the same state q (i.e., q has multiple predecessor states q' in the chart), one selects from among the predecessor states the direct predecessor trace with the highest CH, thus maximizing the CH of the current trace. (Note that Equation 7.17 from my thesis is no longer valid.)

Once the maximal CH's of the traces in a state are known, one can compute their activation $A(e_{q,j})$:

$$A(e_{q,j}) = \lambda_0^{CH(e_{q,j})} \quad (2)$$

where λ_0 is a parameter of the model. Alternatively one may use a polynomial decay function for the trace activation:

$$A(e_{q,j}) = CH(e_{q,j})^{\lambda_0} \quad (3)$$

Now it is straightforward to compute the base probabilities P_{shift}, $P_{project}$ or P_{attach} for the parser operations, as functions of (trace activations in) the current state q. The probability of moving from treelet t_q to another treelet t_r is proportional to the relative fraction of traces in q that prefer to move to t_r, weighted by their activations, as given in Equation 4:

$$P_{episodic}(r|q) = \frac{\sum_{e_i \in E_q^r} A(e_i)}{\sum_{e_j \in E_q} A(e_j)} \tag{4}$$

Here, 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.

While Equation 4 gives the 'episodic' component of the transition probabilities, to obtain a smoothed distribution the probabilities are backed off to the non-episodic left corner probabilities (as estimated from the treebank), which in turn are backed off to the rule probabilities with less context (as explained in section 6.2.4 of my thesis)[2].

$$P(t_{q'}|t_q) = (1-\lambda_1) \cdot P_{episodic} + \lambda_1 \cdot ((1-\lambda_2) \cdot P_1 + \lambda_2 \cdot ((1-\lambda_3) \cdot P_2 + \lambda_3 \cdot P_3)) \tag{5}$$

One should take care that all incoming contributions to the *CH*'s of traces in a state are considered before continuing to update the *CH*'s of traces in states further down in the derivation. This can be dealt with by a prioritized queue, as explained in section 7.1.6 of my thesis. Only once the *CH*'s of all traces in a state are maximized one can compute its base transition probabilities.

Given the base probabilities for P_{shift}, $P_{project}$ and P_{attach}, the forward, inner and Viterbi probabilities of the states in the chart are dynamically updated in the same manner as in the van Uytsel parser (see section 7.1.3 of my thesis), and from these prefix probabilities can be computed.

4.1 Dynamic updates of the common history (*CH*) of a trace

The *CH*'s of the traces in the states of the chart are updated dynamically, following the control structure of the LCSG chart parser, as described in section 7.1.3 of my thesis. Unlike in the case of the shortest derivation chart parser, in the EPLC two quantities must be updated, an inner *CH* (CH_{inn}) and an outer *CH* (CH_{out}). This is to ensure that in every state there is an updated value of the outer *CH*, from which the base treelet transition probabilities can be computed[3]. The outer *CH* refers to

[2]However, in the current implementation of the EPLC chart parser thus far only two levels of back-off have been used.

[3]In the shortest derivation chart parser one is not interested in the values of an 'outer' *SDL* in any state except for the final state, hence one may omit the outer *SDL*, since in the final state inner and outer *SDL* coincide.

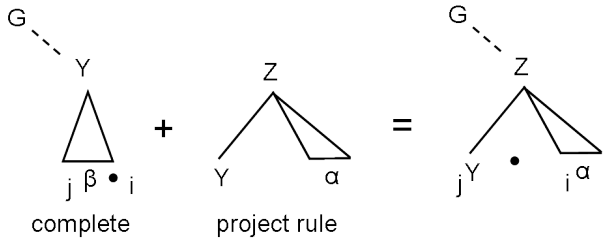

Figure 10: Project operation

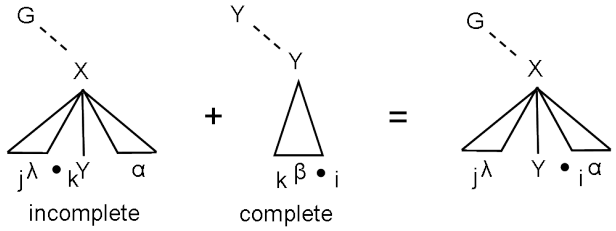

Figure 11: Attach operation

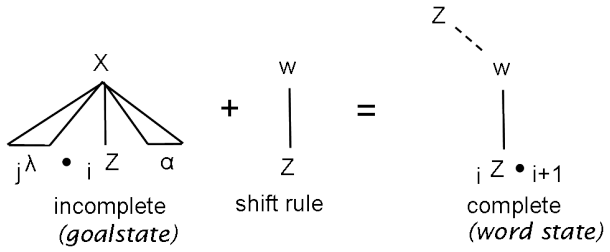

Figure 12: Shift operation

the accumulated common history of a trace, possibly across and including several shift loops. It is this latter quantity that is used to compute the activation of a trace in Equation 2. The inner *CH* is reset to zero at every shift loop, and it is used to enable efficient, dynamic computation of the *CH*'s, because it needs to be computed only once for states within the shift loop that may be visited by multiple derivations (see section 7.1.5 of my thesis for an explanation).

Note that unlike with the computation of the SDL, there is no need to maintain so-called *shiftPointers* (i.e., pointers to different traces in the wordState at the start of a shift loop). This is because at attach time, given a trace in the newly derived state, there is only a single, uniquely determined trace in the wordState that can be reached without a 'break' (i.e., a switch between discontinuous traces). Moreover, if there has not been a break, this trace (in the wordState) is necessarily a direct successor of the trace in the goal state that is a uniquely associated with the given trace in the new state (this follows from the fact that it is required / enforced that the traces in the goal treelet and the subsequent 'attach treelet' belong to the same node of a single parse tree from the training set: see section 3 for an elaboration of this point). On the other hand, if there has been a break in the shift loop of a derivation preceding the attach operation, one may safely forget the trace in the wordState through which the pending derivation passes (this is unlike the case of the episodic SD chart parser).

In order to keep track of whether a path of traces is interrupted or not during the shift loop of a derivation, one sets a flag called *shiftLoopBroken* in every trace. While *shiftLoopBroken* is set to false for each trace in the wordState (at the beginning of a shift loop), it is set to *true* if at some point in the derivation a switch is made to a non-continuous trace. At attach one checks the *shiftLoopBroken* flag of a trace: if it is false, then one can be certain that the current trace can be followed all the way back through the shift loop to the associated trace of the goal state (via a continuous trace in the wordState) without a break, hence in this case alone the outer *CH* of the trace in the goal state may be added to the inner *CH* of the trace in the new state.

Below I give the update equations for the shift, project and attach operators, where the following notation is used: $CH(e_{\mathcal{N},j})$ is the *CH* of the j^{th} trace $e_{\mathcal{N},j}$ in the new state $\mathcal{N}$; $CH(e_{\mathcal{C},i})$ and $CH(e_{\mathcal{I},k})$ are similar definitions for the i^{th} and the k^{th} trace in the complete state $\mathcal{C}$ and the goal state $\mathcal{I}$ respectively. Recall that updates are only executed if they increase the *CH* of a trace (i.e., *CH* is maximized over all predecessor states and predecessor traces).

- The *shift* operation (Figure 12):
  For every trace in the new state $\mathcal{N}$ (a so-called wordState)

$$CH_{inn}(e_{\mathcal{N},j}) = 0 \tag{6}$$

$$CH_{out}(e_{\mathcal{N},j}) = \begin{cases} CH_{out}(e_{\mathcal{I},k}) + 1 & \textit{if } e_{\mathcal{I},k} \textit{ is a direct predecessor trace of } e_{\mathcal{N},j} \\ 0 & \textit{otherwise} \end{cases} \quad (7)$$

Set *shiftLoopBroken* = *false*

- The *project* operation (Figure 10)
  For every trace in the new state $\mathcal{N}$

  If there is a trace $e_{\mathcal{C},k}$ in the predecessor state that is a direct predecessor trace of trace $e_{\mathcal{N},j}$

$$CH_{inn}(e_{\mathcal{N},j}) = CH_{inn}(e_{\mathcal{C},k}) + 1 \quad (8)$$

$$CH_{out}(e_{\mathcal{N},j}) = CH_{out}(e_{\mathcal{C},k}) + 1 \quad (9)$$

  Update *shiftLoopBroken*:

$$shiftLoopBroken(e_{\mathcal{N},j}) = shiftLoopBroken(e_{\mathcal{C},k}) \quad (10)$$

  Otherwise, if there is no direct predecessor trace in the predecessor state

$$CH_{inn}(e_{\mathcal{N},j}) = CH_{out}(e_{\mathcal{N},j}) = 0 \quad (11)$$

$$shiftLoopBroken(e_{\mathcal{N},j}) = true \quad (12)$$

- The *attach* operation (Figure 11):
  For every trace in the new state $\mathcal{N}$
  Find the associated (paired) trace $e_{\mathcal{I},m}$ in the goalState $\mathcal{I}$ (that is uniquely determined via a parse tree in the training set)

  If there is a trace $e_{\mathcal{C},k}$ in the predecessor state that is a direct predecessor trace of trace $e_{\mathcal{N},j}$
  If $shiftLoopBroken(e_{\mathcal{C},k})$ = false

$$CH_{inn}(e_{\mathcal{N},j}) = CH_{inn}(e_{\mathcal{C},k}) + CH_{inn}(e_{\mathcal{I},m}) + 1 \quad (13)$$

$$CH_{out}(e_{\mathcal{N},j}) = CH_{inn}(e_{\mathcal{C},k}) + CH_{out}(e_{\mathcal{I},m}) + 1 \tag{14}$$

else ($shiftLoopBroken(e_{\mathcal{C},k}) = \text{true}$)

$$CH_{inn}(e_{\mathcal{N},j}) = CH_{inn}(e_{\mathcal{C},k}) + 1 \tag{15}$$

$$CH_{out}(e_{\mathcal{N},j}) = CH_{inn}(e_{\mathcal{C},k}) + 1 \tag{16}$$

Otherwise, if there is no direct predecessor trace in the predecessor state

$$CH_{inn}(e_{\mathcal{N},j}) = CH_{out}(e_{\mathcal{N},j}) = 0 \tag{17}$$

In any case, copy the value of the *shiftLoopBroken* flag from the goal trace to the current trace.

$$shiftLoopBroken(e_{\mathcal{N},j}) = shiftLoopBroken(e_{\mathcal{I},m}) \tag{18}$$

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

4.2 Complexity issues

As discussed in my thesis, the space complexity of the episodic shortest derivation LC chart parser (ESDLC) is linear in the corpus size, since the trained episodic grammar stores a single trace in every treelet visited in a derivation. The space complexity of the episodic probabilistic left corner chart parser (EPLC) is increased by a factor $|G|$ (where $|G|$ is the total number of treelet types in the grammar), as a unique probability distribution is associated with every state of the chart parser.

The time complexity is best compared to that of the standard Earley parser, which has an upper bound time complexity of $\mathcal{O}(n^3 \times |G|)$, where $|G|$ equals the number of rules in the grammar, and n equals sentence length. The time complexity of the EPLC parser is then $\mathcal{O}(n^3 \times |G| \times N)$, where N denotes the size of the corpus (which gives an upper bound to the number of traces in a treelet), and $|G|$ denotes the number of unique treelet types. On the other hand, the time complexity of the ESDLC parser is $\mathcal{O}(n^3 \times |G| \times N^2)$, which is caused by the need to maintain shift pointers, as explained above.

Note that not only the space complexity, but also the time complexity (of the EPLC parser alone) compares favorably with DOP, because when computing the probability of a parse, instead of summing over the probabilities of multiple derivations (sum of products), the parse probability equals the probability of a single (Viterbi) derivation, which is computed as a product over sums (where the sums are over trace activations).