GLR Parsing with Scoring

Keh-Yih Su[*], Jong-Nae Wang[**], Mei-Hui Su[**] and Jing-Shin Chang[*]

[*]Department of Electrical Engineering
National Tsing Hua University, Hsinchu, Taiwan, R.O.C.

[**]BTC R&D Center
28 R&D Road II, 2F
Science-Based Industrial Park, Hsinchu, Taiwan, R.O.C.

1. Introduction

In a machine translation system, the number of possible analyses associated with a given sentence is usually very large due to the ambiguous nature of natural languages. But, it is desirable that only the *best* one or two analyses be translated and passed to the post-editor so as to reduce the required efforts of post-editing. In addition, processing time for a sentence is usually limited when processing a large number of sentences in batch mode. Therefore, it is important, in a practical machine translation system, to obtain the *best* syntax tree which has the *best* annotated semantic interpretation within a *reasonably short time*. This is only possible with an intelligent parsing algorithm which can truncate undesirable analyses as early as possible and avoid wasting time in parsing those ambiguous constructions that will eventually be discarded.

Since the selected analysis has to be the *best* in a meaningful sense (for example, best in probabilistic sense), it means that a good *scoring mechanism* will play a very important role. A *score function* which has such optimality property will be described in this chapter.

There are several methods to accelerate the parsing process [Su 88b], one of which is to decrease the size of the *searching space* in the whole parsing state space. This can be accomplished with a *scored* parsing algorithm that truncates unlikely paths as early as possible [Su 87b] and hence decreases the parsing time. Such parsing strategy is referred to as scored truncation.

The searching strategy for a scored parsing algorithm can be either parallel or sequential. A *parallel truncation* algorithm ([Su 87b]) would expand the parsing state space in the breadth-first direction, which allows a number of, say N, alternative paths to expand at the end of each step. A typical parallel truncation algorithm is the beam search algorithm in AI literatures. A *sequential truncation* algorithm, on the other hand, would branch in the depth-first direction. This strategy has some advantages over its parallel counterpart. First, it is faster in getting the *first* parse than the parallel approach. This feature makes it possible to get a fast response if the system is facilitated with a good scoring mechanism, and it can be beneficial for systems which select the first parse as their goal. (This is often the case for small MT systems.) Once the first parse is acquired, we can also use its score to establish a lower bound for all possible scores; any parse whose score is lower than this lower bound can be safely truncated. Thus, we can use this bound to speed up the parsing process further. Secondly, it is much easier to implement a sequential truncation version than a parallel one, and it takes less memory space in the sequential version because no multiple temporary copies of different analyses need to

be saved. Finally, the sequential truncation strategy can cut down more searching space than its parallel counterpart because it will try less paths when a good scoring mechanism is given.

Furthermore, a real system usually set a time limit to stop the parsing process when a sentence is taking too long to parse because of its long sentence length or complicated structure. Under such circumstances, the sequential searching strategy is better than the parallel approach because we are likely to have some complete syntax trees to work with even if the parsing was suspended abnormally when its time expires. On the other hand, the parallel approach will not have this advantage because it may not have any on-going path that traverse to the end.

In this chapter, we will give an informal introduction to the *Score Function* used in our machine translation system. Some desirable features of this score function will be described and its application to Generalized LR parsing is introduced. We will also propose a *sequential truncation* parsing algorithm to reduce the searching space of the parsing process, and hence improving the parsing efficiency. This algorithm employs the score function proposed in [Su 88a], which takes advantages of the probabilistic characteristics of the syntactic and semantic information in the sentences. Preliminary tests on this algorithm were conducted with some special versions of our machine translation system, the ARCHTRAN [Su 87a], and encouraging results were observed. Readers interested in the parallel version of the scored truncation parsing mechanism can refer to [Su 87b] for more details.

2. Making Decisions With The Score Function

2.1. Definition of the Score Function

In a scored parsing system, the best analysis is selected base on its score. Several scoring mechanisms have been proposed in the literatures [Robi 82, Benn 85, Gars 87, Su 88a]. The one we adopt is the *score function* proposed in [Su 88a]. This score function measures the degree of preference of a semantically annotated syntax tree by the following general formula :

$$SCORE\,(Sem, Syn, Lex, Words) \equiv P\,(Sem, Syn, Lex | Words)$$

where *Sem*, *Syn* and *Lex* are the particular *semantic interpretation*, *syntactic structure* and *lexical feature* attached to a given ambiguous construction whose terminal strings are *Words*. In other words, we will assign a higher score to a semantically annotated construction if it is often regarded as the most probable interpretation to the terminal strings. The score function defined in this way is not only linguistically significant but also statistically optimal as we shall show later. Hence, it provides a way to bridge the gap between linguistic knowledge and statistic reality.

2.2. Why Score Function

The above probabilistic model has some advantages over traditional rule-based approaches, which are usually *ad hoc*. First of all, because the score function is statistics-based, it is more objective (in statistic sense) than the certainty factors assigned by linguistic experts (linguists) in a rule-based expert system; a score of 0.8 defined in *statistic sense* will be significantly different from a certainty factor of 0.8 assigned in *expert's sense*. It is also much easier to *train* and *maintain* the probability entries than linguistic rules. Furthermore, the

embedded knowledge in the statistics database is always *consistent* in statistic sense, as opposed to conventional knowledge bases, which usually contain some degree of inconsistency. Another advantage of this score function lies in its flexible *extensibility*. It can be extended easily to lower linguistic levels such as acoustics, phonetics and morphology. On the other hand, it can also be extended to a higher level such as pragmatics. In fact, we have already extended its definition to deal with speech recognition in some applications [Chia 89].

The most striking feature of the above score function is its *optimality* when used as the preference measure of semantically annotated syntax trees. (We will take a syntax tree annotated with semantic features as the representation of a specific interpretation to a given sentence throughout our discussion.) To see why it is *optimal* as the preference measure let's consider the following situation. In an attempt to find the best annotated syntax tree among the ambiguous ones, we must pay some price for possible *misjudgement* no matter what scoring mechanism is used. In general, we want the cost to be minimal. The problem of finding an optimal scoring function can thus be formulated as a *cost minimization* problem. Suppose that we will incur a loss (or cost) of C_1 when we make a right choice on the semantic interpretation (*Sem*$_i$), syntactic structure (*Syn*$_j$) and lexical feature (*Lex*$_k$) for a set of input strings (*Words*), and a loss of C_2 if we make a wrong choice. Then the expected cost, based on any scoring mechanism, would be

$$Cost = C_1 \times P(Sem_i, Syn_j, Lex_k | Words) + C_2 \times (1 - P(Sem_i, Syn_j, Lex_k | Words))$$

Since, in general, we will incur small or no loss when we make a right choice, so C_1 can be set zero (or negative). Furthermore, we can think of C_2 as a positive constant corresponding to the extra efforts for post-editing. Under such circumstances, to minimize the cost of disambiguation would be equivalent to maximizing *P(Sem*$_i$*, Syn*$_j$*, Lex*$_k$ *| Words)*, namely to maximize our *score function*. Therefore, with the score function as the preference measure, we will incur *minimal cost*, in *Bayesian* sense, for disambiguation.

2.3. Decomposition of Score Function

Now that we have a good sense about the optimality property of the score function, how can we use this general formula to deal with the general problems of disambiguation in linguistic level? In the following sections, we will briefly outline some possible ways to apply the score function to different driving mechanisms by decomposing the score function in different ways. In particular, we will map this function to a Generalized LR parser (GLR parser for short) which can handle augmented context-free grammars for natural language processing.

First of all, let's decompose the score function to show how the score function can be related to traditional *stratified analysis* paradigm. To simplify the analysis tasks in different analysis phases, we can usually compute a score by dividing the score function into three components as follows :

$$\begin{aligned} & SCORE(Sem_i, Syn_j, Lex_k, Words) \qquad (1) \\ & \quad \equiv P(Sem_i, Syn_j, Lex_k | w_1 \ldots w_n) \\ & \quad = P(Sem_i | Syn_j, Lex_k, w_1 \ldots w_n) \times P(Syn_j | Lex_k, w_1 \ldots w_n) \times P(Lex_k | w_1 \ldots w_n) \\ & \quad = SCORE_{sem} \times SCORE_{syn} \times SCORE_{lex} \end{aligned}$$

where w_1 through w_n are the input words in the given sentence. In this formula, the three product terms are referred to as *semantic score ($SCORE_{sem}$)*, *syntactic score ($SCORE_{syn}$)* and *lexical score ($SCORE_{lex}$)*, respectively. By making such decomposition, we can model traditional stratified analysis approach easily. In other words, we can perform lexical analysis, syntax analysis and semantic analysis based on lexical score, syntactic score and semantic score, respectively. The total *compositional score* then serves as the global indicator to the degree of preference of a given interpretation.

For simplicity, we will focus only on the syntactic aspect of this score function, namely $SCORE_{syn}$, and show how to compute $SCORE_{syn}$ under GLR parsing environment. These techniques can be extended to the other two components of the compositional score function as well. The goal here is to get the most probable *syntactic structure* when the *input words* and their *lexical feature* (represented by part of speech) are given. We can first simplify the above syntactic score function as $SCORE_{syn}(Syn_j) \equiv P(Syn_j|Lex_k, w_1 \ldots w_n) \approx P(Syn_j|Lex_k) = P(Syn_j|c_1 \ldots c_n)$, where c_1 to c_n are the lexical categories (parts of speech) corresponding to w_1 to w_n. In making the simplification, we assume that the syntactic structure is independent of the terminal strings themselves but depends on their part of speech. This is usually the case in GLR parsing. After the simplification is made, we can get the score by properly decomposing this formula into pieces of probability entries, which can then be evaluated stepwise through the process of GLR parsing. In the following sections, two such formulations for syntactic score are proposed to show the versatility of the score function.

2.4. Syntactic Score in Context Sensitive Model

To show the mechanism for computing syntactic score, first refer to the syntax tree in Fig. 1. This syntax tree is decomposed into a number of *phrase levels* (L_1 ... L_8), each phrase level being a set of terminal or nonterminal symbols that corresponds to a *sentential form* of the sentence. The phrase levels shown here correspond to a canonical derivation sequence that is produced by a generalized LR bottom-up parsing algorithm. Let L_i be the *i*-th phrase level. Then a transition from phrase level L_{i+1} to phrase level L_i corresponds to a *derivation* of a nonterminal at time t_i. On the other hand, transition from L_i to L_{i+1} would be equivalent to a *reduction* at t_i.

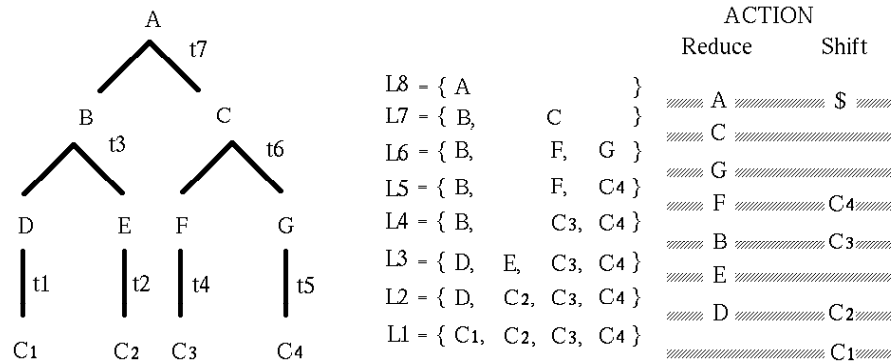

Fig. 1. Decomposition of a syntax tree into phrase levels for score computation in bottom-up GLR parsing

After the syntax tree is decomposed into phrase levels, we can express the syntactic score in terms of these phrase levels. For example, the syntactic score of the syntax tree in Fig. 1

can be formulated as the following conditional probability:

$$\begin{aligned} & SCORE_{syn}(Syn_A) \\ & \quad = P(L_{8,}L_{7...}L_2|L_1) \\ & \quad = P(L_8|L_{7...}L_2,L_1) \times P(L_7|L_{6...}L_1) \times ... \times P(L_2|L_1) \\ & \quad \approx P(L_8|L_7) \times P(L_7|L_6) \times ... \times P(L_2|L_1) \end{aligned} \tag{2}$$

Note that the product terms in the last formula correspond to the sequence of *rightmost derivation* in a GLR parser *with left and right contexts taken into account*. Therefore, such formulation is especially useful for generalized LR parsing algorithm in which context-sensitive processing power is desirable.

To evaluate the transition probability between two phrase levels, say $P(L_7 \mid L_6)$, we can evaluate the product of probabilities of two events. The first probability corresponds to the event that {F, G} are the constituents to be reduced, and the second probability corresponds to the event that they are reduced to C; both events are evaluated under the context of {B, F, G}. The transition probability can thus be expressed as:

$$\begin{aligned} P(L_7 \mid L_6) = & P(F,G\ are\ reduced \mid input\ is\ \{B,F,G\}) \\ & \times P(C \leftarrow FG \mid F,G\ are\ reduced;\ input\ is\ \{B,F,G\}) \end{aligned}$$

To simplify the evaluation, we can approximate the full context {B, F, G} with a window of finite length around {F, G}. The window sizes for the two subevents need not be identical. According to our experiments, the first term is equal to one in most cases, and it has little contribution in discriminating different syntax structures. Hence, we can ignore the first term and approximate the transition probability with the second term. The formulation can thus be expressed as:

$$SCORE_{syn} \approx P(\{A\}|\{l_7,B,C,r_7\}) \times P(\{C\}|\{l_6,F,G,r_6\}) \times ... \times P(\{D\}|\{l_1,c_1,r_1\})$$

where l_i and r_i are the left and right contexts of the symbols to be reduced. The degree of context-sensitivity is specified by the number of context symbols to be consulted. This number is called the *order* of context-sensitivity of the score function. If the order of context-sensitivity for left and right contexts are *m* and *n* respectively, it is said to operate in *LmRn* mode. For example, Eq. (2) can be further reduced to the following equation when operating in L1R1 mode :

$$\begin{aligned} & SCORE_{syn}(Syn_A) \\ & \quad \approx P(\{A\}|\{\emptyset,B,C,\emptyset\}) \times P(\{C\}|\{B,F,G,\emptyset\}) \times ... \times P(\{D\}|\{\emptyset,c_1,c_2\}) \end{aligned} \tag{3}$$

where "$\emptyset$" is the null symbol. In this case, only one immediate context symbol, either to the left or to the right of the reduced symbols, is consulted.

Many systems have tried to adopt L0R0 mode of operation in probabilistic parsing. A probability is assigned to each production rule. The score is then computed as the product of these probabilities. However, even if the language can be represented by a context-free grammar, it does not mean that its rules are used in context-free (statistically independent) manner; the L0R0 model is valid only when the adoption of each rule is independent of the preceding use of the other rules. As a result, when the contextual information of the language becomes more and more important, this operation mode will become less feasible. Although

the context-sensitive model in Eq. (2) provides the capability to deal with *intra-level context-sensitivity*, it fails to catch *inter-level correlation*. In practice, the events {L_2 ... L_8} in Eq. (2) should be jointly considered, instead of being decomposed into seven terms that are assumed to be independent. Otherwise, a syntax tree with more nodes (i.e. more levels) will be less likely to get high score because the multiplication of the large number of probabilities tends to reduce its score. However, the parameter space will be too large if we jointly consider many phrase levels. An alternative to relieve such a *normalization problem* is to compact multiple highly correlated phrase levels in evaluating the score as formulated in the next section.

2.5. Syntactic Score in Run-Time Model

The formulation for syntactic score computation in Eq. (2) and (3) can take place when the parser is executing a **reduce** action. We shall now show another alternative in which the computation takes place when a **shift** action is executed.

In GLR parsing, we may wish the computation of scores to occur after an input word is just fetched. This is often the case in *parallel* parsing algorithms (such as parallel truncation algorithm and Tomita's algorithm) where a number of paths are compared based on the partial cumulative scores of the *same* input subsentence (or more exactly, based on the same *prefix*).

This can be done by compacting multiple **reduce** actions and a single **shift** into one step and formulate the syntactic score $SCORE_{syn}(Syn_A)$ as follows.

$$\begin{aligned} & SCORE_{syn}(Syn_A) \\ & \quad = P(L_8, L_{7\ldots}L_2|L_1) \\ & \quad = P(L_8, L_7, L_6|L_5, L_{4\ldots}L_1) \times P(L_5|L_4, L_{3\ldots}L_1) \times P(L_4, L_3|L_2, L_1) \times P(L_2|L_1) \\ & \quad \approx P(L_8, L_7, L_6|L_5) \times P(L_5|L_4) \times P(L_4, L_3|L_2) \times P(L_2|L_1) \\ & \quad \approx P(L_8|L_5) \times P(L_5|L_4) \times P(L_4|L_2) \times P(L_2|L_1) \end{aligned} \quad (4)$$

The probability entries in the last simplified formula correspond to a sequence of change in the *stack contents* between two **shift**s. In fact, the stack contents immediately after c_1, c_2, c_3, c_4 and \$ (the end-of-sentence symbol) are pushed onto the stack are {c_1, (c_2, c_3, c_4, \$)}, {D, c_2, (c_3, c_4, \$)}, {B, c_3, (c_4, \$)}, {B, F, c_4, (\$)} and {A, \$, ()}, which are the *prefix* of L1, L2, L4, L5 and L8, respectively. (The terminal symbols in the parentheses are the input symbols not yet fetched.) Such formulation makes run-time score computation an easy task because we can simply monitor the status of the stack and compute the probability entries step-by-step after an input word is shifted.

Two assumptions were made in formulating Eq. (2) – (4). First, it is assumed that the formation of phrase level *i* is only dependent on its immediate lower phrase level, since most information percolated from other lower levels is contained in that level. And second, a reduction or derivation is only locally context sensitive to its immediate left or right context at each phrase level. This assumption is also supported in other systems as well [Marc 80, Gars 87].

2.6. Brief Summary

From the previous sections, it is easy to see that the whole scoring mechanism consists of three main components :

1. A *score function* which defines the preference measure of a given interpretation.

2. Different types of *decomposition schemes* which apply the score function to different driving mechanisms or algorithms in different mode.
3. A *simplification scheme* to simplify the computation for each decomposition scheme, which also determines the order of context-sensitivity.

We have introduced only two decomposition schemes in the previous sections. The versatility of this scoring mechanism, however, is beyond what we have described here. For more information about other formulations and the comparison to other probabilistic models in the wide variety of linguistic levels, interested readers can refer to [Su 91] for more formal description.

With such a score function, we can achieve arbitrary degree of context-sensitive parsing capability (in probabilistic sense) with a context-free grammar. Such extension is quite different from other *trainable grammar* like [Bake 79], where N-gram paradigm is used to train a context-free grammar with restricted (and hence unnatural) right hand side symbols. Under our score function paradigm, the grammar rules can be any augmented context-free grammar written by linguists and the contextual information is embedded in the statistic knowledge of the score function (and other augmentation not mentioned here). The score function thus integrates linguistic reality and statistic knowledge in an elegant way.

A simulation based solely on the *syntactic score* was conducted and reported in [Su 88a] with a full-path searching algorithm. The result shows that the correct syntactic structures of over 85% of the test sentences were successfully ranked at the *first* place when a total of 3 local left and right context symbols were consulted. In addition, over 93% of the correct syntax trees are ranked at the *first* or *second* place based on the syntactic score with 2 context symbols. With *semantic score* incorporated into the mechanism, the results should be even more promising.

3. The Sequential Truncation Algorithm

3.1. Basic Algorithm

Using the score function defined in the previous section, we will present the idea of sequential truncation algorithm with Fig. 2.

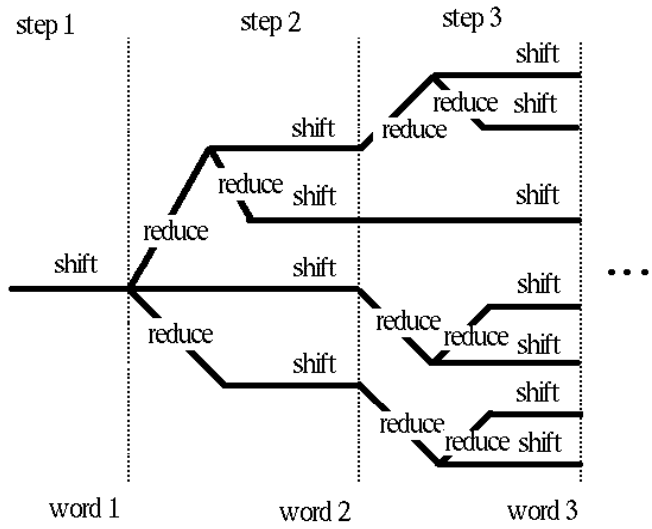

Fig. 2 The searching tree

Each path in Fig. 2 corresponds to a possible derivation of a given sentence. The parser will traverse the searching tree with the depth-first strategy. But during the searching process, the parser compares the accumulated running score of each path with a running threshold constant C(α_i) at each step *i* when the *i*-th word is fetched. If the score of the path is less than the running threshold C(α_i), it will be truncated, i.e. blocked, and the next path will be tried. This process continues until we get the complete parse tree. The fraction of paths to be blocked at each step is determined by the control variable α_i which will be defined later in the next section.

After we obtain the first complete parse tree, a lower bound for the scores, initially set to the score of the first parse tree, is also acquired. The parser will continue to traverse other paths, but from now on, the score of each path will also be compared with the current low bound for the scores in addition to be compared with the running threshold. This additional comparison is similar to the *branch and bound* strategy employed in AI applications [Wins 84] and it will accelerate the parsing process further. If the test fails in either case, this path will be truncated. Continuing in this manner, we will update the lower bound whenever a new complete parse tree has a score higher than current lower bound, and repeat the whole process until the end of the entire searching process. The whole process is shown in the flow chart in Fig. 3.

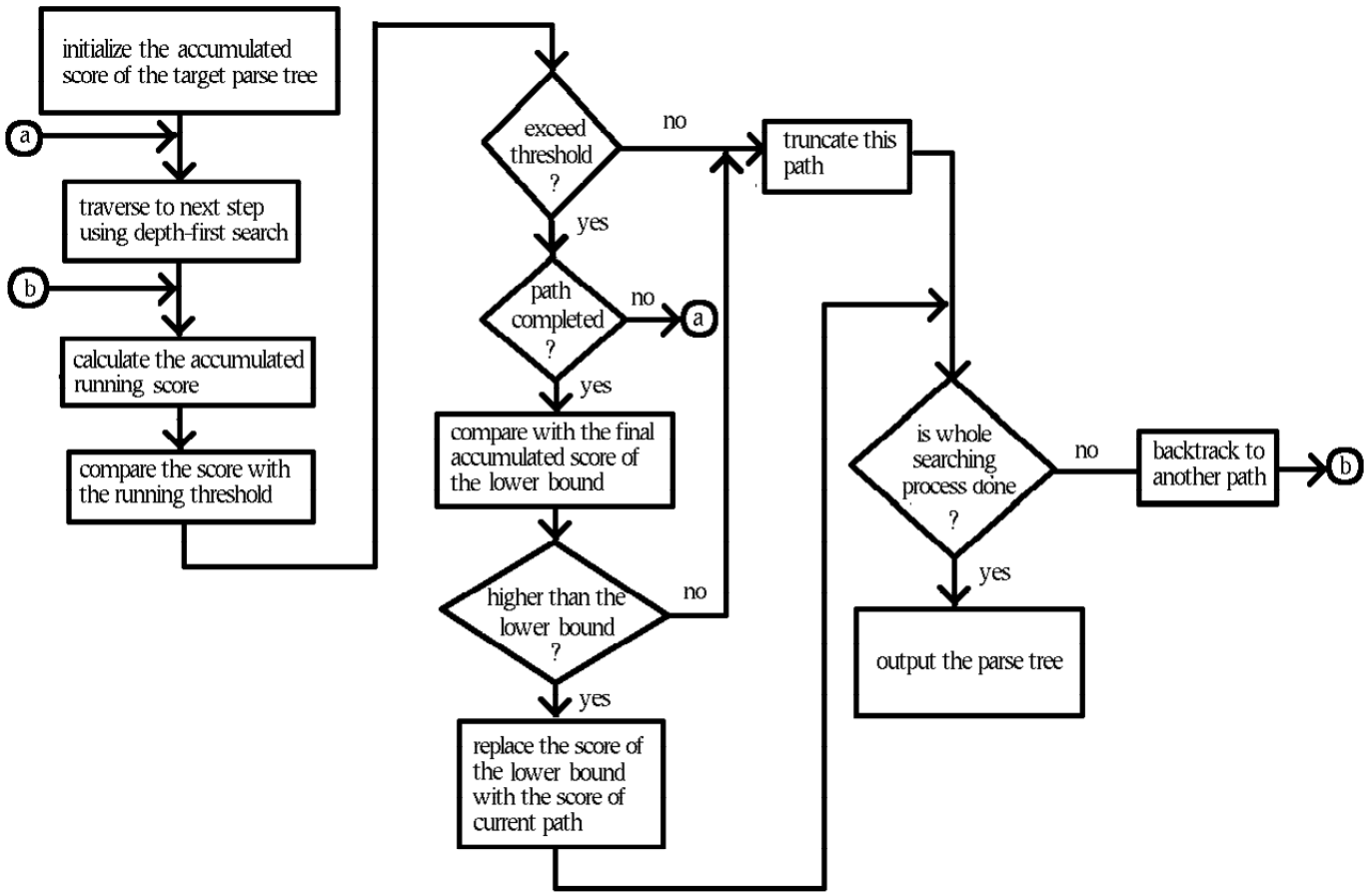

Fig. 3 Flow chart for sequential truncation parsing algorithm

When all the paths are blocked without arriving at any complete parse tree, we can adopt one of two possible strategies. First, we could loosen the running thresholds, i.e. lowering $C(\alpha_i)$, and try the deepest path gone so far again. Second, we can process this sentence in *fail-soft* mode. The fail-soft mechanism will skip and discard the current state and attempt to continue the parsing at some later point.

As we can see, the score of each syntax tree can be expressed as the product of a sequence of conditional probabilities such as the one shown in Eq. (4). Each term in the product corresponds to a transition between two **shift** actions and is evaluated immediately after a **shift**. To avoid mathematical underflow, we shall take the logarithm on both sides of Eq. (4) to get a logarithmic score in the form of :

$$log\,(SCORE_{syn}\,(Syn)) = \sum_{i=1}^{L} log\ P\,(X_i'|X_i) \tag{5}$$

where L is the length of the sentence, X_i denotes the set of phrase levels which have been encountered up to the *i*-th **shift**, and X'_i is the complement of X_i, namely the set of phrase levels not yet encountered up to the *i*-th **shift**.

3.2. Analytic Description

The effectiveness of the sequential truncation algorithm is closely related to the distribution of scores of the database and the distribution of scores of the input sentences. This phenomenon will be illustrated in the following paragraphs.

If we define $y_j \equiv \sum_{i=1}^{j} log\ \ P(X_i'|X_i)$, then y_j is the accumulated logarithmic score up to the *j*-th word, which is also the accumulated score after the *j*-th **shift** of the sentence.

Suppose we have M sentences with their correct parse trees in the database. For each parse tree, we can evaluate y_j by using the logarithmic score function defined in Eq. (5). So for the *k*-th sentence in the database, we obtain a sequence $y_1^k,\ y_2^k,\ \ldots y_{L_k}^k$, where y_j^k is the accumulated logarithmic score of the *k*-th sentence up to the *j*-th step, and L_k is the length of the *k*-th sentence.

Given these accumulative scores found in the database, we can define a set of random variables Y_i corresponding to y_i^k, with k ranged from 1 to M. In other words, Y_i is the random variable corresponding to the accumulative logarithmic scores up to the *i*-th word of the sentences in the database. Using the samples in the database, we can draw a histogram for each Y_i and approximate each histogram by a continuous density function $f_Y^i(y)$.

To allow a fraction α_i, say 99%, of the best parse trees in the database to pass the threshold test at step *i*, we can set a constant $C(\alpha_i)$ such that $P(Y_i \geq C(\alpha_i)) = \alpha_i$, where $C(\alpha_i)$ is the running threshold that we will use to compare with the running accumulated logarithmic score at step *i*. Those paths with running accumulated logarithmic score y_i less than $C(\alpha_i)$ would be blocked at the *i*-th step. Using the notation defined above, the probability of obtaining the desired parse tree for a sentence with length L_k would be $\prod_{i=1}^{L_k} \alpha_i$.

For the input sentences to be parsed, we can also define a set of random variables Z_i corresponding to the distribution of the accumulated logarithmic scores at the *i*-th step and a set of density function $f_Z^i(z)$ associated with Z_i. These probability entries would be evaluated from the ambiguous trees of the input sentences. Fig. 4 shows the relationship between the probability density function $f_Z^i(z)$ (the distribution of the *i*-th running score of the *input text*) and the probability density function $f_Y^i(y)$ (the distribution of cumulative score of the *database*.) In the figure, the dashed lines indicate the means of the two density functions. Since the step-wise cumulative scores in the database are evaluated using the correct parse trees in the database, we would expect that the expectation of Y_i is greater than that of Z_i and the variance of Y_i is less than that of Z_i. In other words, we have $E[Y_i] > E[Z_i]$ and $Var[Y_i] < Var[Z_i]$ in normal situation.

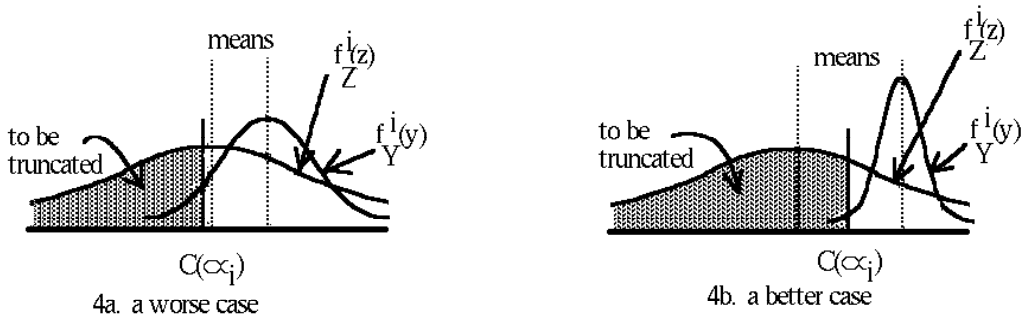

Fig. 4 Relationship between the running score of the input text and the cummulative score of the database

Let β_i denote $F_Z^i(C(\alpha_i))$, where $F_Z^i(z)$ is the cumulated distribution function of Z_i, then β_i is the probability that a path will be truncated at the *i*-th step of the searching process. By using this sequential truncation method, the searching space would then be approximately reduced to $\prod_{i=1}^{L_k}(1-\beta_i)$, which is a small portion of the original searching space generated by a full path searching algorithm. Therefore the efficiency of parsing is increased. Since β_i in Fig. 4a is less than that in Fig. 4b, which correspond to the situation that has a large expectation difference ($E[Y_i]$-$E[Z_i]$) and a small variance ratio ($Var[Y_i]/Var[Z_i]$), the underlying grammar that has the property of Fig. 4b would benefit most from this algorithm. In addition, we can see that if we increase the running threshold $C(\alpha_i)$, we will get a greater β_i and a lower α_i. The parsing efficiency will thus increase, but the probability (i.e. $\prod_{i=1}^{L_k}\alpha_i$) that we will get the desired parse tree would decrease. How to select a good $C(\alpha_i)$ to achieve a desired success rate is thus very important. In the following section, this issue would be discussed in greater details.

4. How To Set The Running Threshold

4.1. Static Running Threshold

Using the model given in the last section, the probability that we will get the global optimal solution, i.e. the parse tree with the largest score, for a sentence of length L is $K_L = \prod_{i=1}^{L}\alpha_i$, where K_L is a constant *pre-selected* by the system designer as a compromise between the parsing *efficiency* and the translation *quality*. Assuming that the average branching factor for each path is a constant N, then the average total number of paths we have to try is :

$$\begin{aligned} g(\alpha_{1\ldots}\alpha_L) &= N + N*(1-\beta_1)*N + N*(1-\beta_1)*N*(1-\beta_2)*N + \ldots \\ &= N*\left(1 + N*h(\alpha_1) + N^2*h(\alpha_1)*h(\alpha_2) + \ldots\right) \\ &= N*\left(1 + \sum_{i=1}^{L-1} N^i \quad *\prod_{j=1}^{i} h(\alpha_j)\right) \end{aligned} \tag{6}$$

where $h(\alpha_i) \equiv (1-\beta_i)$. In order to minimize the path number, we must have $h(\alpha_1) < h(\alpha_2) < \ldots < h(\alpha_L)$ because h(α_i) has more significant influence on the number of paths to be truncated than h(α_{i+1}).

The problem of selecting an appropriate running threshold C(α_i) is now converted into one of minimizing $g(\alpha_1 \ldots \alpha_L)$ under the constraint of $\prod_{i=1}^{L} \alpha_i = K_L$. Taking the logarithm on both sides, we get $\sum_{i=1}^{L} log\ \alpha_i = log\ K_L$. Then the *Lagrange multiplier* λ is introduced to get $g^*(\alpha_{1\ldots}\alpha_L) = g(\alpha_{1\ldots}\alpha_L) + \lambda * \sum_{i=1}^{L} log\ \alpha_i$. Taking the partial derivative of g^* with respect to $\alpha_1 \ldots \alpha_L$, we will get the following simultaneous equations :

$$\frac{\partial g^*}{\partial \alpha_1} = 0, \quad \frac{\partial g^*}{\partial \alpha_2} = 0, \quad \ldots \quad \frac{\partial g^*}{\partial \alpha_L} = 0, \quad and \quad \sum_{i=1}^{L} log\ \alpha_i = log\ K_L \qquad (7)$$

In the above equations, there are (L+1) variables, which are $\alpha_1 \ldots \alpha_L$, and λ, and (L+1) equations. So, $\alpha_1 \ldots \alpha_L$ can be solved by a numerical method. Since α_i is usually very close to 1, we can linearize the function h(α_i) in the region around α_i=1 and approximate it by $h(\alpha_i) \approx a * \alpha_i + b$. In this way, we can substitute h(α_i) in the above equation by $a * \alpha_i + b$ to simplify the calculation.

During our derivation, we have assumed that the average branching factor at each stage is a constant N. This constraint can be relaxed by assuming the average branching factor at the *i*-th stage to be N_i. In this way, we will get a more complicated expression for $g(\alpha_1 \ldots \alpha_L)$, but it can still be solved in the same way.

The running threshold C(α_i) can now be computed off-line by selecting different K_L for different sentence length L. We will call this set of C(α_i) the *static* running threshold, because once they are computed, they will not be changed during the parsing process.

4.2. Dynamic Running Threshold

The static threshold derived in the above section can serve as a limiting factor in the truncation algorithm. However, if we arrive at a complete parse tree with much higher final accumulated running score than the final accumulated running threshold, then the running threshold should be adjusted to reflect the higher final accumulated running score. Such adjustment is necessary because even if a path can pass all the threshold tests it might still be discarded when compared with the much higher final accumulated running score. Therefore, it would be better if the running threshold is changed to $C'(\alpha_i)=C(\alpha_i)+\Delta C(\alpha_i)$, where $\Delta C(\alpha_i)$ is set to $\gamma * (y_i^* - C(\alpha_i))$, $0<\gamma<1$, and y_i^* is the accumulated score of the current best parse tree at the *i*-th step. The tunning constant γ is another control variable pre-selected by the system designer. Since $C'(\alpha_i)$ is adjusted dynamically it will be called the *dynamic* running threshold. Using the dynamic running threshold, the efficiency of parsing would be further improved.

If it so happen that all paths are blocked before a complete parse tree is formed, we can find the deepest path (assuming it to be at the *j*-th step) among the blocked ones and reactivate it with a lowered running threshold of $C'(\alpha_j)=y'_j$, where y'_j is the score of this path at the *j*-th step. Since the procedure to lower the running threshold is quite complicated it might be more convenient just to invoke the fail-soft mechanism for sentences whose paths are all blocked.

5. Scored Truncation vs Charted Parsing

It is interesting to make a comparison between the well-known chart mechanism [Wino 83] and the truncation algorithm. In many current systems, a chart-like approach is used to accelerate the parsing speed by avoiding the reparsing overhead of substructures which had been constructed previously. In the previous sections, we have demonstrated the speedup effects of the scored sequential truncation algorithm, which accelerates parsing speed by cutting down the searching space with the aids of a good scoring function. Is it profitable to combine these two mechanisms into one ? The answer seems to be “yes” if the parsing *speed* is the only concern.

Nevertheless, there is a few words to be said about the combination of these two mechanisms. When the truncation algorithm is adopted in a chart parser, the best tree originally selected by a chart parser might not appear in the trees produced by its truncation version. This phenomenon lies in the fact that *chart* mechanism [Wino 83] is essentially in conflict with the *truncation* mechanism in their operations. The reason for having chart is to be able to *retain* all subtrees that were parsed in previous path traversal. So, when we backtrack to the next path and arrive at the same set of input strings, the same subtrees can be used again without reparsing. On the contrary, the idea behind the truncation mechanism is to *discard* a subtree which has low score as early as possible. Therefore, if we adopt the truncation mechanism during charted parsing, not every possible subtree between a set of input words will be successfully constructed and stored into the chart. For example, in Fig. 5, there are two possible subtrees between *b* and *c*. When the paths in block A are expanded, one of the subtrees is discarded while the other is stored into the chart.

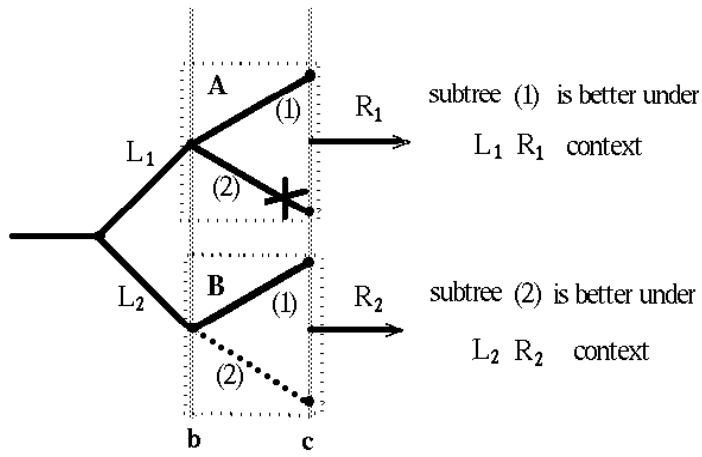

Fig. 5. Chart with truncation mechanism

The subtree may be discarded for the following two reasons. First, it might be due to the semantic constraints on the contextual dependency of the system. Second, the subtree might be discarded because of its small running accumulated score (and thus truncated by the truncation mechanism.) Either will leave us a chart with incomplete subchart. So, this might result in the best possible tree being missed as a side-effect of using this chart under other contextual environment. For instance, the best tree in Fig. 5 might be constructed with the second subtree, the left context L_2 and the right context R_2. But, since the path expansion

starting from the left context L_1 has the second subtree discarded because of its low score under the context of L_1 and R_1, the best tree will never be formed. Therefore, with a chart having incomplete subcharts, the possibility of obtaining the best tree might be affected by the paths truncated before.

One solution to this incompatibility problem is to mark the subcharts as being complete or incomplete. If an incomplete subchart is encountered again, it will be reparsed. On the other hand, if a complete set of chart is encountered, the subtrees can be copied directly from the chart. Another solution is to suspend the truncation mechanism when a chart is being tried the first time. And if subtrees are copied directly from the chart, the truncation mechanism resumes its normal operation. In this way, it is guaranteed that every subchart in the chart is complete. Both of these solutions will introduce some overhead. This compromise, however, is unavoidable if the advantages of using chart are to be retained.

6. Testing

We have completed two preliminary tests on the truncation algorithm with two special versions of our English-Chinese MT system, the ARCHTRAN. The tests use a database which contains the required probability entries acquired from a set of 1430 sentences. A more extended test on a database of 1607 sentences is also conducted. The following paragraphs will describe these tests in detail.

In the first experiment, the parsing time needed by a *chart parser* is compared with the time needed by the same chart parser which is augmented with *truncation* mechanism. The chart parser adopted a *bottom-up parsing with top-down filtering* strategy (TD+BU for short, see [Su 87a]). Its parsing efficiency is about the same order as Tomita's extended chart parser [Tomi 87, Su 90] and it has a mean speed-up factor of about 43 over traditional uncharted LR parser. From the test, we found that the average parsing time by the chart parser with truncation is improved by a factor of about 4. For some sentences, the improvement can go as high as a factor of 20. This result is encouraging.

In the second experiment, we converted the chart parser for the first experiment into a chartless one. Similar test is conducted for this chartless parser but with a smaller analysis grammar. The result shows that the total parsing time for this parser with truncation mechanism added is better than the same parser without truncation by a factor of about 3. Because the uncharted parsers are relatively more slower than the previous charted ones, only a few sentences are tested which account for the above speedup factor.

In the previous two tests, the size of the test samples is small. A more extended test was conducted with even more encouraging results. The results, together with a sentence length distribution curve, are shown in Fig. 6. The speedup factors are shown in logarithmic scale. More detailed test data is listed in the Appendix of this chapter. In the test, two parsers were used to show the effects of *sequential truncation* algorithm with respect to *charted parsing*. The first parser is a TD+BU chart parser, which does not adopt truncation algorithm. The second parser is a truncation-based bottom-up (BU) parser, which does not use chart mechanism. The score database for the second parser contains 1607 sentences which are decomposed into probability entries in L2R1 format (that is, 2 left context symbols and 1 right lookahead are consulted). The augmented context-free grammar used in the test contains a set of 1046 production rules. Under such test environment and taking the sentence length distribution into account, the mean speed-up factor for a set of 165 sentences from

inside the database is about 17 (close test). For another set of test data which contains 196 sentences from outside the database the speed-up factor is about 9 (open test). In general, the speed-up factors vary with the number of ambiguity and the length of the sentences. (Individual comparison shows speed-up factors ranging from 1 to more than 100 in both open and close tests.) Nevertheless, the figure shows a tendency of increasing speed-up factors with length (and hence with structural complexity) of the sentences. This tendency is very encouraging because a truncation algorithm is most beneficial for long sentences in which a large number of paths can be cut down.

In the above tests, only *syntactic* score and *static* threshold were used. We expect that more searching space will be cut down and hence larger speed-up factors will result when *semantic* score and *dynamic* running threshold are incorporated into the truncation algorithm. From the positive results of the above experiments, we have shown the inclusion of the sequential truncation algorithm is advantageous for a MT system.

7. Concluding Remarks

In an operational machine translation system, it is important to arrive at a *good* analysis for a sentence in a *reasonably short time*. One way to achieve this is to decrease the parsing

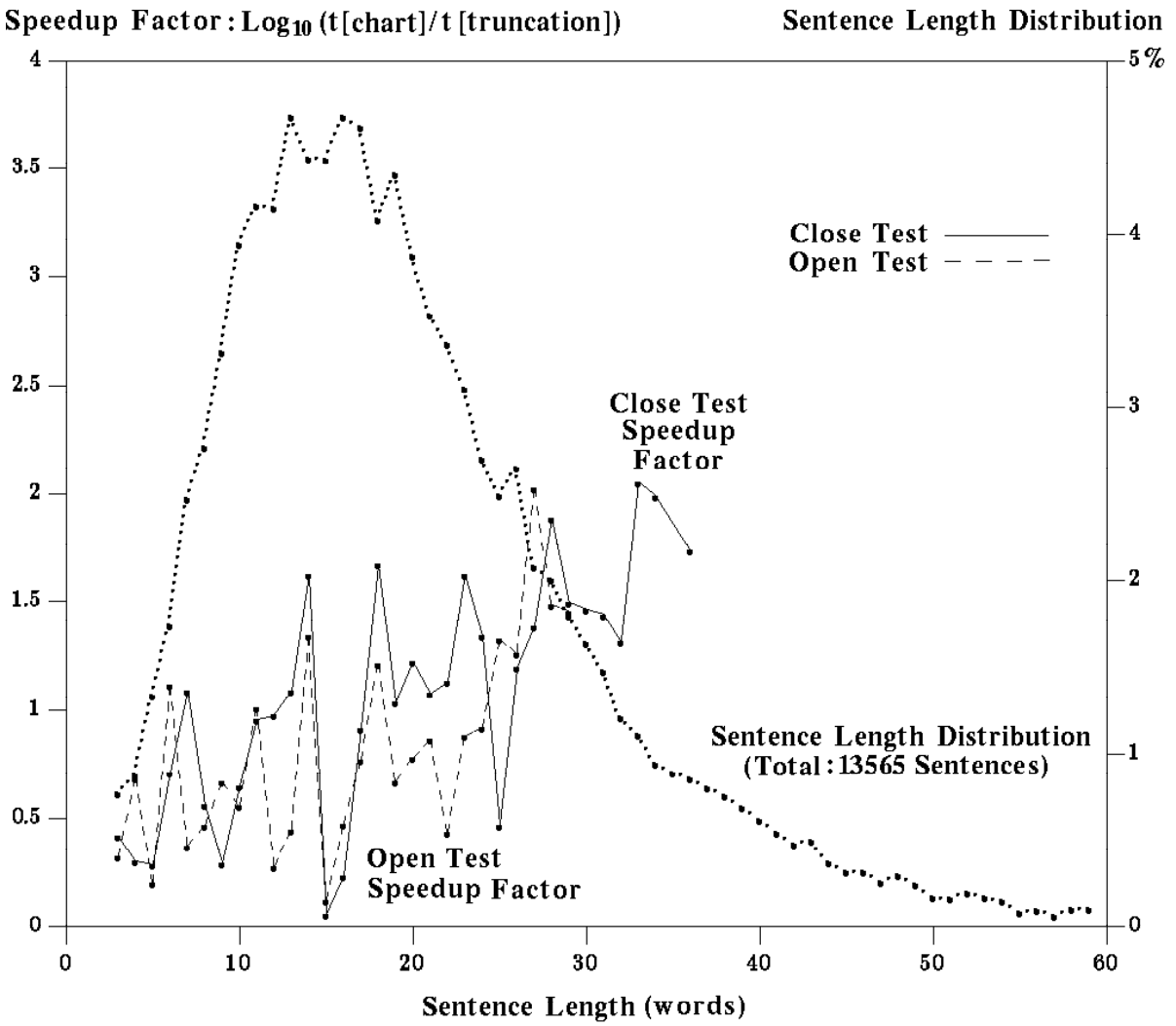

Fig. 6 Comparison of the speed of truncation parser and chart parser

time by reducing the searching space. We have proposed a *sequential truncation* algorithm with a *score function* to achieve this goal. There are several reasons for adopting this strategy. First, the first parse tree with a moderate quality can be found quickly and easily. Second, the running threshold serves to truncate part of the paths that is quite unlikely to lead to the best analysis, and thus greatly reduces the searching space.

The optimality property of the score function has been described in this chapter. Some possible decomposition schemes for tailoring this function to a GLR parser are also introduced. In particular, we have shown how to model the stratified analysis paradigm with the score function, and demonstrated the versatility of the score function by introducing two decomposition schemes about the syntactic score.

In the sequential truncation strategy, a sequence of running thresholds are used to bound the searching space during each step of the scored parsing process. Additional speedup can be acquired by a branch-and-bound strategy if the accumulated score for a parse is lower than a dynamically adjusted lower bound.

We have pointed out the incompatibility problem between the use of *chart* and the *truncation* mechanism. As our current research topic, we shall resolve the incompatibility problem between the chart mechanism and the truncation algorithm, and include the solution into our working MT system, the ARCHTRAN.

We have made a pilot test on the truncation mechanism with a chart parser that adopts bottom-up parsing with top-down filtering strategy. With a database of 1430 sentences, the result indicates an average improvement in the parsing time by a factor of 4 (for some sentences the improvement goes as high as a factor of 20). In another pilot test on the truncation mechanism, the parsing time is tested for a chartless parser that adopts sequential parsing strategy. The result shows an improvement in parsing time by a factor of 3 for the inclusion of the truncation mechanism. These encouraging results demonstrate a great promise for the sequential truncation strategy. In another extended test, we have achieved mean speed-up factors of 9 and 17 for open test and close test, respectively. This result shows a great improvement of the sequential truncation algorithm over conventional chart parser if a good score function is available.

8. References

[Bake 79] Baker, J.K., "Trainable Grammars for Speech Recognition," *Proc. of the Spring Conf. of the Acoustical Society of America*, 1979.

[Benn 85] Bennett, W.S. and J. Slocum, "The LRC Machine Translation System," *Computational Linguistics*, vol. 11, no. 2-3, pp. 111-119, ACL, Apr.-Sep. 1985.

[Chia 89] Chiang, T.-H., *The Design of a Chinese Phonetic Typewriter*, Master thesis, National Tsing Hua University, Hsinchu, TAIWAN, R.O.C., 1989.

[Gars 87] Garside, Roger, Geoffrey Leech and Geoffrey Sampson (eds.), *The Computational Analysis of English : A Corpus-Based Approach*, Longman , New York, 1987.

[Marc 80] Marcus, M.P., *A Theory of Syntactic Recognition for Natural Language*, MIT Press, Cambridge, MA, 1980.

[Robi 82] Robinson, J.J., "DIAGRAM : A Grammar for Dialogues," *CACM*, vol. 25, no. 1, pp. 27-47, ACM, Jan. 1982.

[Su 87a] Su, K.-Y., J.-S. Chang, and H.-H. Hsu, "A Powerful Language Processing System for English-Chinese Machine Translation," *Proc. of 1987 Int. Conf. on Chinese and Oriental Language Computing*, pp.260-264, Chicago, Ill., USA, 1987.

[Su 87b] Su, K.-Y., J.-N. Wang, W.-H. Li, and J.-S. Chang, "A New Parsing Strategy in Natural Language Processing Based on the Truncation Algorithm", *Proc. of Natl. Computer Symposium (NCS)*, pp. 580-586, Taipei, Taiwan. 1987.

[Su 88a] Su, K.-Y. and J.-S.Chang, "Semantic and Syntactic Aspects of Score Function," *Proc. COLING-88*, vol. 2, pp. 642-644, 12th Int. Conf. on Computational Linguistics, Budapest, Hungary, 22-27 Aug. 1988.

[Su 88b] Su, K.-Y., "Principles and Techniques of Natural Language Parsing : A Tutorial," *Proc. of ROCLING-I*, pp.57–61, Nantou, Taiwan. Oct. 1988.

[Su 90] Su, K.-Y. and J.-S. Chang, "Some Key Issues in Designing MT Systems," *Machine Translation*, vol. 5, no. 4, pp. 265–300, 1990.

[Su 91] Su, K.-Y., J.-S. Chang and Y.-C. Lin, "A Unified Approach to Disambiguation Using A Uniform Formulation of Probabilistic Score Functions," In preparation, 1991.

[Tomi 87] Tomita, M., "An Efficient Augmented-Context-Free Parsing Algorithm," *Computational Linguistics*, vol. 13, no. 1–2, pp. 31–46, 1987.

[Wino 83] Winograd, Terry, *Language as a Cognitive Process*, Addison-Wesley, Reading, MA., USA, 1983.

[Wins 84] Winston, P.H., *Artificial Intelligence*, Addison-Wesley, Reading, MA., USA, 1984.

9. Appendix : Open Test and Close Test

The following tables show the results of the open and close tests described in the chapter (see also Fig. 6). The environment under which the tests were conducted is summarized as follows :

- Parser #1 : [-Truncation] [+Chart] [TD+BU]
- Parser #2 : [+Truncation] [-Chart] [BU] (Bottom Up)
- Score Data Base : 1607 sentences (L2R1)
- Grammar : 1046 production rules
- Open Test : 196 sentences (executed at SUN-4/260)
- Close Test : 165 sentences (executed at SUN-4/110)
- Mean Speed-Up : Open Test = 9 & Close Test = 17

CLOSE TEST

Length (words)	No. of Test Sentences	1st Parser Total CPU Time (sec)	2nd Parser Total CPU Time (sec)	Speed-Up (1st / 2nd)
3	2	2.85	1.09	2.61
4	2	5.81	2.92	1.99
5	2	2.63	1.37	1.92
6	2	31.62	6.12	5.17
7	2	33.99	2.77	12.27
8	2	19.60	5.38	3.64
9	2	15.31	7.80	1.96
10	5	175.86	39.50	4.45
11	5	518.89	56.93	9.11
12	4	387.64	40.74	9.51
13	3	448.02	36.33	12.33
14	5	3647.54	86.29	42.27
15	3	143.51	125.65	1.14
16	6	300.34	175.57	1.71
17	11	1950.57	237.51	8.21
18	12	16901.77	360.53	46.88
19	13	4882.46	448.87	10.88
20	18	13148.05	785.17	16.75
21	14	11560.29	957.19	12.08

22	12	7624.84	560.50	13.60
23	9	33879.85	801.12	42.29
24	6	6377.89	292.22	21.83
25	2	407.79	138.53	2.94
26	3	1534.79	97.97	15.67
27	3	8949.71	370.90	24.13
28	2	6651.16	86.40	76.98
29	3	4236.36	135.33	31.30
30	3	2444.76	83.89	29.14
31	2	1749.56	63.50	27.55
32	2	2881.16	140.11	20.56
33	2	8471.91	75.48	112.24
34	2	15669.64	161.78	96.86
36	1	2533.35	46.48	54.50
TOTAL	165	157589.52	6431.94	24.50
MEAN	(take sentence length distribution into account)			17.22

OPEN TEST

Length (words)	No. of Test Sentences	1st Parser Total CPU Time (sec)	2nd Parser Total CPU Time (sec)	Speed-Up (1st / 2nd)
3	4	3.70	1.76	2.10
4	5	7.29	1.50	4.86
5	5	6.31	3.99	1.58
6	5	61.88	4.73	13.08
7	9	39.12	16.62	2.35
8	10	79.52	27.17	2.93
9	10	315.52	67.75	4.66
10	10	266.15	73.72	3.61
11	9	377.72	36.81	10.26
12	9	154.69	81.72	1.89
13	10	359.94	128.87	2.79
14	9	2779.25	126.72	21.93

15	8	179.15	137.21	1.31
16	9	587.23	198.93	2.95
17	8	2121.77	358.72	5.91
18	9	5391.85	328.23	16.43
19	10	1814.09	385.68	4.70
20	11	4888.94	807.30	6.06
21	14	8916.45	1217.73	7.32
22	8	1104.86	406.98	2.71
23	4	1238.93	162.71	7.61
24	4	2249.22	268.99	8.36
25	3	1905.49	90.29	21.10
26	4	10292.94	563.95	18.25
27	2	8759.70	83.29	105.17
28	1	677.92	22.07	30.72
29	6	5969.74	209.76	28.46
TOTAL	196	60549.37	5813.20	10.42
MEAN	(take sentence length distribution into account)			9.07