# Efficient Generalized LR Parsing of Word Lattices

Alon Lavie and Masaru Tomita
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

email : lavie@cs.cmu.edu

**Abstract**

A word lattice is an efficient representation of a large set of possible sentence candidates, of which only a few are grammatical and thus parsable. Word lattices are a common output of some speech recognizers, and may also arise as a result of multiple part-of-speech tags of sentence words. Parsing a word lattice involves finding a path of connecting words within the lattice that is grammatical. In typical cases, where the words of the lattice are assigned probabilities or confidence scores, the goal of the parser is to find the grammatical path of highest overall score within the lattice. We describe an efficient algorithm for parsing such word lattices. Our algorithm is based on a Generalized LR style substring parser, that can parse an input string in arbitrary order. An efficient computation strategy is achieved by using an $A^*$ heuristic to determine the order in which words of the lattice are processed.

## 1 Introduction

This paper is concerned with the problem of parsing word lattices. A word lattice is an efficient representation of a large set of possible sentence candidates, of which only a few are grammatical and thus parsable. Word lattices are a common output of some speech recognizers, and may also arise as a result of multiple part-of-speech tags of sentence words. In the speech case, individual word hypotheses are characterized by a time interval (marking the beginning and ending times of the word) and a likelihood score. In the case of multiple part-of-speech tags, the order of words of the original sentence determines an order on the assigned parts-of-speech, and each possible part-of-speech tagging of a word is assigned a probability. In both cases, the lattice consists of a two dimensional grid, on which all the hypotheses are represented according to their time and probability features.

Parsing a word lattice involves finding a path of time-wise connecting words within the lattice that is grammatical. The goal of the parser is to find the grammatical path of highest overall score within the lattice. We describe an efficient algorithm for parsing such word lattices. Our algorithm is based on a Generalized LR style substring parser, that can parse an input string in arbitrary order. An efficient computation strategy is achieved by using an $A^*$ heuristic to determine the order in which words of the lattice are processed.

Without the grammaticality constraint, the highest scoring path of time-wise connecting words through the lattice can be computed in time linear in the number of words in the lattice using a Dynamic Programming Algorithm [Thompson, 1990],[Thompson, 1989].

Previous work by Tomita, Kita, Saito and others [Tomita, 1986a] [Kita *et al.*, 1989] [Saito and Tomita, 1988] has focussed on how to use the predictive power of the Generalized LR parser in order to guide the search through the huge space of word hypotheses in speech recognition systems, and has also tried to deal with the problem of missing words. Saito [Saito, 1990] suggested an algorithm that can start parsing from an identified anchor word, from which the parsing can proceed sequentially in both directions. These parsing methods are rigid due to the fact that the parser must scan and process the input in a sequential uni-directional fashion.

Chow and Roukos [Chow and Roukos, 1989] describe a bottom-up CYK-style parsing algorithm that does not suffer from the uni-directionality restriction. The algorithm uses Dynamic Programming and Chart Parsing techniques in order to parse the word lattice and find the highest scoring grammatical path.

The word lattice parsing algorithm we present in this paper has several advantages over the Chow and Roukos algorithm. First, our algorithm is founded on Generalized LR (GLR) parsing, which is very efficient in practice due to the utilization of parsing tables that are pre-compiled in advance from the grammar. Second, our algorithm is based on an algorithm for parsing substrings that we have developed. This substring parsing algorithm follows from previous work by Bates and Lavie [Bates and Lavie, 1992],[Bates and Lavie, 1991] on recognizing substrings of LR languages, and from work by Rekers and Koorn [Rekers and Koorn, 1991]. The substring parser is converted into a GLR parser that can parse the words of an input sentence in any arbitrary order. Words of the input are parsed as substrings and are combined with other neighboring substrings as these become available. A unique feature of our substring parsing algorithm is that it can parse arbitrary substrings, irrespective of phrase boundaries. This allows neighboring substrings to be combined even when they span several partial phrases. This property provides complete flexibility in determining the order in which to parse the words of the input.

In order to achieve an efficient computation strategy for parsing the word lattice, we develop an $A^*$ style heuristic. The heuristic determines the order in which lattice words are parsed so that potentially more probable substrings are pursued first. The heuristic guarantees a halting condition, by which it can be determined when the best full parse found so far is the most probable one in the lattice.

The remainder of the paper is organized in the following way. Section 2 describes the substring parsing algorithm. Section 3 presents our GLR arbitrary word order parser. A running example of the parser is presented in section 4. In Section 5 we describe the $A^*$ style heuristic and how it is incorporated with the parser in order to efficiently parse the word lattice. Finally, our conclusions and future directions are presented in section 6.

## 2 The Substring Parsing Algorithm

LR parsing techniques have long been popular as efficient and powerful methods for processing context free languages. Tomita's Generalized LR parsing algorithm [Tomita, 1986b] extended these techniques to deal with the ambiguities normally present in natural language grammars, while preserving the efficiency of the parser that stems from it being driven by the pre-compiled LR parsing tables. Ambiguous grammars are not LR, and as a result, the standard parsing tables derived from them contain conflicting actions, known as "Shift/Reduce" conflicts. Tomita's algorithm uses a Graph Structured Stack (GSS) in order to efficiently pursue in parallel the different parsing possibilities that arise as a result of such conflicts in the parsing tables. A second data structure uses pointers to keep track of all possible parse trees throughout the parsing of the input, while sharing common subtrees of these different parses.

Our substring parsing algorithm is similar in principle to the one described by Rekers and Koorn [Rekers and Koorn, 1991], and follows from an algorithm for recognizing substrings of LR languages developed by Bates and Lavie [Bates and Lavie, 1992],[Bates and Lavie, 1991]. For simplicity, we assume the parser is SLR(1), although the principles described here are applicable to the other LR parsing variants as well. Given an input $x = x_1 x_2 \cdots x_n$, the algorithm first accesses the parsing table in search of all states that wish to shift the first input symbol $x_1$. These states are entered into the GSS as initial states. Parsing continues from all of these initial GSS states in the ordinary way specified by the GLR parsing algorithm. For each top state in the GSS, the next action (or actions) is determined from the parsing tables, according to the state and the next input symbol. Each action may be either a *shift*, an *error* or a *reduce*, and is treated in the following manner:

- A *shift* action of the form `sh`$k$ (shift to state $k$) is treated normally. The input symbol is shifted into the GSS, and a new top state node with state $k$ is added to the GSS.
- An *error* action indicates that the input cannot continue to be parsed from this top node, and this path in the GSS is discarded.
- A *reduce* action of the form `r`$i$ (reduce by rule $i$) is treated normally, as long as the reduction can be completed with the existing stack symbols in the GSS. If this is not the case, the reduce action is a *long reduction*, and is handled in a special way, as shall be described below.

The major difference between our algorithm and that proposed by Rekers and Koorn is in the handling of long reductions. Long reductions are reductions that attempt to pop states and symbols beyond the bottom of the GSS. They thus correspond to reductions that include symbols that are prior to the beginning of the given input string. Our algorithm uses an additional parsing table, the *long reduction goto table*, to handle such cases. The idea behind the long reduction goto table is to determine the set of states in which the parser could find itself after completing the reduction (or perhaps multiple reductions), and from which the next action would be a shift. It therefore enables the parser to postpone the actual performance of the reduction, and to continue parsing the input by shifting the next input symbol. For each possible state $k$ and rule $i$, the table specifies the state (or states) to which the parser would goto after completing a reduction of rule $i$ from top state $k$. The long reduction goto table is easily constructible in advance from the grammar in a way similar to the other parsing tables.

When our substring parser encounters a long reduction, it marks the top state in which the long reduction occurred, determines the set of continuation states from the long reduction goto table and adds these states as new top states to the GSS, connecting the new states with the old marked state. This action is in fact equivalent to delaying the actual reduction from taking place, and allows the parser to continue parsing the input as if the reduction had occurred. It is compatible with the action performed by the Rekers and Koorn algorithm in this case, but intentionally does not remove the reduced nodes from the GSS.

If the algorithm succeeds to reach the end of the input string $x$ and has processed the last input symbol $x_n$, it is guaranteed by properties of the LR parsing paradigm that $x$ is a valid substring of some sentence in the language described by the grammar, and as such is accepted by the substring algorithm. The algorithm does not produce a full or partial parse tree of the substring. However, the parse information is represented in the parser's GSS when the substring algorithm terminates, and is utilized by the arbitrary word order full-string parsing algorithm, that is based on our substring parser and presented in the following section.

# 3 Arbitrary Word Order Generalized LR Parsing

We now describe our arbitrary word order parsing algorithm that is based on the substring parsing algorithm presented in the previous section. The primary advantage of this algorithm is that the input word sequence may be parsed in an arbitrary chosen order. Furthermore, this order need not be determined prior to the start of the parsing process, and decisions on which word of the input to handle next can be made dynamically, based on any kind of relevant information or heuristic. This property enables us to use an $A^*$ heuristic to efficiently parse word lattices.

The main idea behind the algorithm is to efficiently parse islands of the input as substrings. Thus, key features of the substring recognition algorithm described in the previous section are used. Parsed islands of the input must be correctly combined with neighboring islands as these become available. Eventually, the parsed islands combine to a fully connected substring parse of the input. Additional constraints may be applied at this point in order to guarantee that the algorithm accepts only full-strings of the language.[1]

## 3.1 Description of the Algorithm

Due to space limitations and for the sake of simplicity, we describe here only the recognition aspect of the algorithm. However, the manipulation of pointers to maintain and eventually produce the parse tree (or trees) of the input are similar in nature to the corresponding actions in Tomita's Generalized LR parsing algorithm. To simplify the description, we assume the input is an $n$ word sequence, where the $i$-th word is time tagged by the interval $[i-1, i]$. Parsed islands are marked with the interval $[i, j]$ which they span.

The beginning of an island is with the startup of a substring parse of a single word $[i-1, i]$. As in the initial stage of the substring parsing algorithm, the parsing table is searched for states that wish to shift the input word. These states are entered into the GSS and the shift action is performed.

After the initial shift action, reductions are performed. Normal reductions occur as usual. Long reductions are handled in the way described by our substring algorithm. The top state in the GSS is marked, and the long reduction goto table is accessed to determine the continuation states. When the parsing of an island reaches the stage where no more reductions can be performed on its GSS (the actions specified from all of the top nodes of the GSS are all shift actions), the processing of the island is stalled until it can be combined with a neighboring island.

When two neighboring islands are combined, their GSSs are "glued" together. The GSS of the left island serves as the "lower" part, while that of the right island is the "upper" part. Each top state in the lower GSS is matched with corresponding bottom states of the upper GSS and the structures are then merged. Parts of either of the two GSSs that find no matching part in their counterpart GSS are discarded at this point.

After the two GSS structures are combined, the upper part of the GSS (the part that originally belonged to the right island) is re-scanned in search of nodes marked by long reductions. A reduction previously marked as long, that can now be performed due to the more complete merged GSS, is executed at this point. This operation may rule out some of the continuation states that were determined when the long reduction occurred. The GSS is pruned accordingly in such cases. The result of the merging of the two GSSs and the post-processing described above is a GSS appropriate to the newly constructed joint island.

[1] However, the parse through the entire input may in fact be valuable even in cases where the input is merely a substring (but not a full-string) of a sentence in the language.

(1) S → NP VP
(2) NP → n
(3) NP → NP PP
(4) VP → v NP
(5) PP → p NP

Figure 1: An Example Grammar

Attempts to combine islands occur in both directions. An island first attempts to combine with a neighboring left island. If no left neighboring island is available, the island attempts to combine with a neighbor to its right. When neither a left or a right neighboring island is available, the processing of the current island is stalled. The island will be picked up again when the parsing of a neighbor to either its left or right attempts to combine with it.

The fact that island combinations are attempted from both left and right directions guarantees that the algorithm will not deadlock, as long as some progress can be made. Thus, if the entire sentence is indeed parsable, the algorithm will eventually combine all islands into a single parsed island.

We assume the parser is able to distinguish if the input segment being processed starts at the beginning of the input or reaches the end of the input. If the algorithm is to accept only full-strings, this information can be used to constrain the parsing process in the following way. If the island does not reach the end of input, parsing actions for all possible following words (other than the end-of-input symbol "$") are considered and pursued. However, if the island does reach the end of the input, only the actions indicated for the end-of-input symbol ("$") need be performed. Similarly, if the segment spanned by the island starts at the beginning of the input, reductions that would require symbols prior to the beginning of the input can be ignored.[2]

The algorithm terminates when no progress can be done on any of the existing islands. The input is accepted if there exists a single island at this point, and the GSS contains the single accepting state. If on the other hand there exist two or more islands, none of which can be combined, the input in whole is not a valid substring and is rejected. The existing islands at this point correspond to the largest valid substrings that could be found within the input string.

# 4 An Example

To clarify how our proposed arbitrary word order parsing algorithm operates in practice, we now present an example. The grammar in Figure 1 is a simple natural language grammar. From this grammar, we construct the standard SLR(1) parsing table of Table 1. Note that the table contains a "Shift/Reduce" conflict for state number 9, in the case of a preposition (terminal symbol "p"). This is due to an ambiguity with prepositional phrase attachments. The long reduction goto table for this parsing table is presented in Table 2. Note that reductions are unique per state in this case, therefore the long reduction goto states are a function of state only (and not of state and rule).

We now follow the first few steps of the arbitrary word order parsing algorithm on the input x = n v n p n p n. Each word of the input is tagged with its appropriate interval of the form

[2] If, on the other hand, we wish the algorithm to accept inputs that are merely substrings (but not full-strings) of a sentence in the language, actions for all possible input words must be pursued at all times.

| | Action | | | | Goto | | | |
|---|---|---|---|---|---|---|---|---|
| State | n | v | p | $ | NP | VP | PP | S |
| 1 | sh2 | | | | 3 | | | 4 |
| 2 | | r2 | r2 | r2 | | | | |
| 3 | | sh6 | sh5 | | | 7 | 8 | |
| 4 | | | | acc | | | | |
| 5 | sh2 | | | | 9 | | | |
| 6 | sh2 | | | | 10 | | | |
| 7 | | | | r1 | | | | |
| 8 | | r3 | r3 | r3 | | | | |
| 9 | | r5 | r5,sh5 | r5 | | | 8 | |
| 10 | | | sh5 | r4 | | | 8 | |

Table 1: Standard Parsing Table for Grammar in Figure 1

| Top state | Goto states after reduction |
|---|---|
| 1 | |
| 2 | 3 9 10 |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | 3 9 10 |
| 9 | 3 9 10 |
| 10 | |

Table 2: Long reduction goto table for the parsing table in Table 1

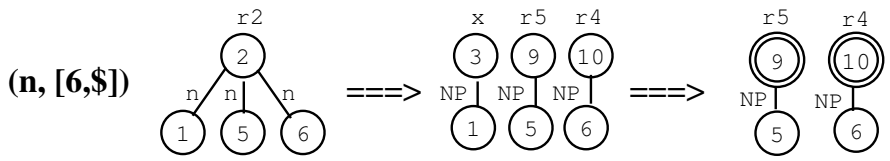


Figure 2: GSS of Island (n, [6, $])

$[i-1, i]$ (for $0 \leq i \leq 7$ in our case). Let us assume that the order of word processing chosen is that in which we process the input from the last word to the first.

We thus begin with the island (n, [6, $]). The tree on the left of Figure 2 is the initial GSS constructed, after shifting the input symbol "n". A normal reduction by rule 2 is then performed, resulting in the GSS shown in the middle of the figure. Since this island borders the end of the input, only further actions on the end-of-input symbol "$" are pursued. The action on state 3 indicates an error, therefore the left tree of this GSS is discarded, and we remain with the GSS on the right side of the figure. The other actions are reductions that are all long at this point. Thus, the two top state nodes are marked (in the figure marked nodes are indicated by a double circle). Since this island borders the end-of-input, it will not be combinable with anything to its right. Therefore, there is no need to determine continuation states in this case. Since no neighboring islands are available at this point, the processing of the island is stalled.

We next continue with the island (p, [5, 6]). The initial shift action results in the GSS shown in Figure 3. The only action from state 5 is a shift. At this point the island is ready to be combined with neighboring islands. Since there is no left neighbor available, the island is combined with the neighbor to its right to form the island (p n, [5, $]). The GSS that results from the combination is shown on the left of Figure 4. The delayed reduction by rule 5 from state 9 can now be performed, and this results in the GSS shown on the right side of the figure. The reduction by rule 3 from state 8 cannot be done (it is a long reduction), so the node is marked. Once again, the long reduction goto table is not accessed in this case, since the island borders the end-of-input.

The processing now proceeds to the island next in line, which is (n, [4, 5])). The tree on the left of Figure 5 represents the initial GSS constructed, after the shifting of the input symbol "n". A normal reduction by rule 2 is then performed, resulting in the GSS shown in the middle of the figure. Note that the top node of state 9 now has two conflicting actions that need to be pursued. To achieve this we graphically split this node into two separate nodes.[3] One of the actions is a reduction by rule 5. This is a long reduction, so the node is marked (graphically by a double circle), and the long reduction goto table is accessed to determine the continuation states. The continuation states are 3, 9 and 10. Since there already exist top level nodes of all three of these states, the long reduction node is connected with these three nodes. The resulting GSS is displayed on the right of Figure 5. The island is now ready to be combined with its neighbors. Since there is no left neighbor available, the island combines with its right neighbor to form the island (n p n, [4, $]). The resulting GSS is shown in Figure 6. The delayed reduction from the top node with state 8 can partially be executed, and the resulting GSS is displayed on the right side of the figure. The action on the top node of state 3 indicates an error, and this part of the graph is thus deleted. The other reductions are long, but since the island borders the end of the input, no continuation states are added. the resulting GSS is shown at the bottom of Figure 6. Processing then moves on to the island (p, [3, 4])) and continues from there on.

[3] In practice, the node does not have to be explicitly separated.

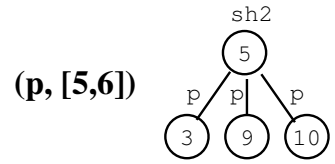


Figure 3: GSS of Island (p, [5, 6])

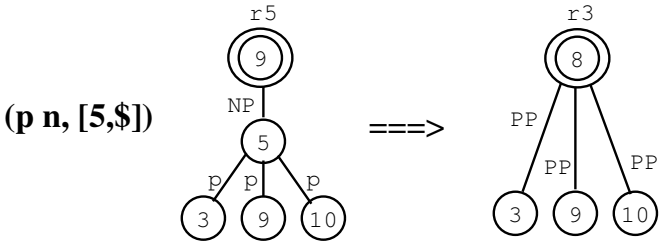


Figure 4: GSS of Island (p n, [5, $])

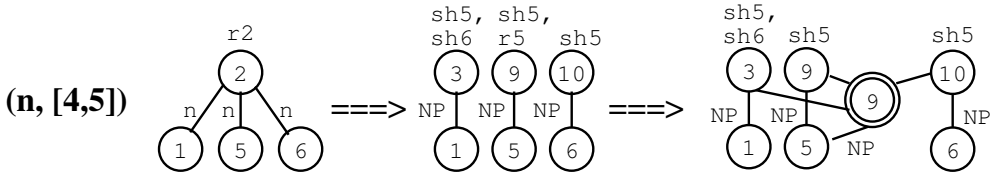


Figure 5: GSS of Island (n, [4, 5])

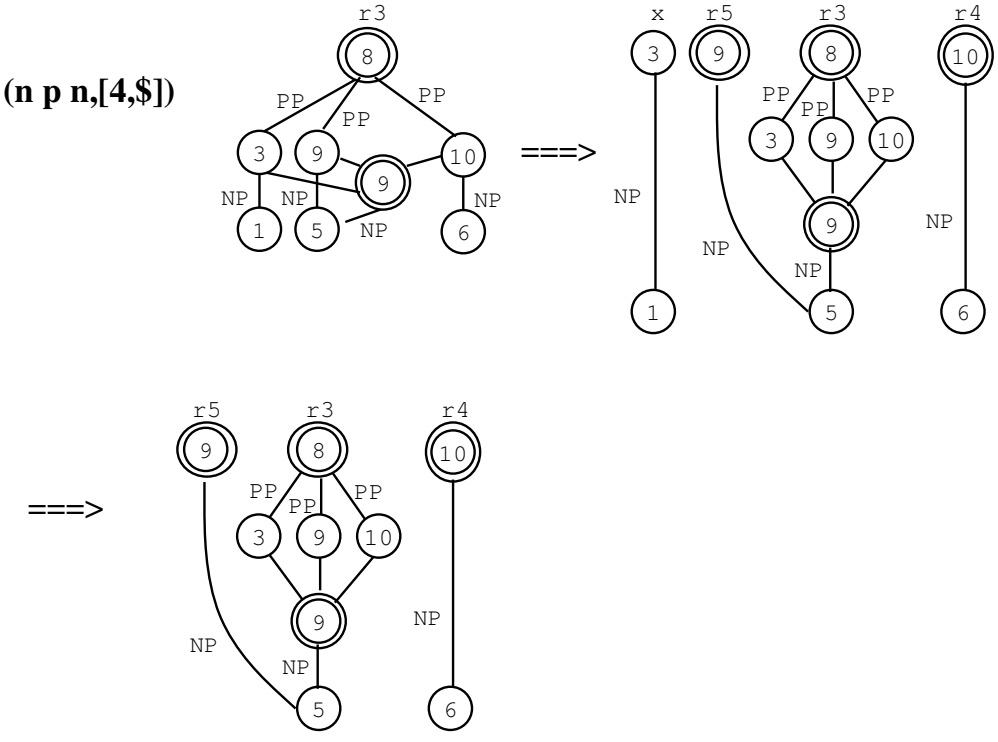


Figure 6: GSS of Island (n p n, [4, $])

# 5 Efficient Word Ordering Using an $A^*$ Heuristic

We now turn to describe how to efficiently parse a word lattice, by determining an optimal ordering on the processing of the lattice words. The aim of the parser is to find a complete path of connecting lattice words that is parsable and is maximal in overall score. We assume that the overall score of a string of lattice words is simply the product of their individual scores. However, our analysis is just as valid with any other monotonically increasing function of the individual scores.

## 5.1 The Heuristic

We suggest an $A^*$ style heuristic to determine the ordering of the words. The lattice is then parsed using the parser described in the previous section, with the order of words determined by the heuristic.

The idea behind the heuristic is to attach an upper-bound score $P^*$ to each lattice word and to each substring parsed by the algorithm. $P^*$ is the maximal score of a path through the lattice that includes the word or substring. This path need not itself be grammatical. The $P^*$ score of a word is defined in the following way:

- Let $w$ be a given word of time span $[t_i, t_j]$ and score $P(w)$.
- Let $P_0^{i*}$ be the score of the best path of connecting words from the beginning of the lattice ($time = 0$) to $time = t_i$.
- Let $P_j^{e*}$ be the score of the best path of connecting words from $time = t_j$ to the end of the lattice ($time = end$).
- Then $P^*(w) = P_0^{i*} \cdot P(w) \cdot P_j^{e*}$

The $P^*$ score of a parsed substring is defined similarly, where the product of $P$ scores of the words of the substring is used instead of $P(w)$.

Note that any path that includes the word $w$ and is parsable is guaranteed to have an overall score that is lesser or equal to $P^*(w)$. This condition holds for parsed substrings as well. Therefore, if the lattice words are processed by order of their $P^*$ values, the following termination condition will hold:

- If the $P^*$ score of all remaining unprocessed words of the lattice is less than or equal to the actual overall score of the best parsable full path found so far, then the algorithm can terminate.

The reason that this condition holds is that the current best parsable path is guaranteed to be better in score than any path that includes any of the words that haven't been yet processed, and thus the best parsable path has already been found.

## 5.2 Computing the $P^*$ Scores

The $P^*$ scores of words and substrings can be efficiently computed. In order to use the above mentioned equation, we must show how to compute, for each word $w$ of time span $[t_i, t_j]$, the values $P_0^{i*}$ and $P_j^{e*}$. For part-of-speech word lattices, this task is trivial, due to the simple time spans of the words. $P_0^{i*}$ is simply the product of scores of the part-of-speech tags of greatest score that precede $w$. Similarly, $P_j^{e*}$ is the product of scores of the part-of-speech tags of greatest score that follow $w$.

In the case of speech produced word lattices, we can use the simple Dynamic Programming algorithm that finds the highest scoring (not necessarily grammatical) path of words through the lattice [Thompson, 1990], [Thompson, 1989]. Thompson's algorithm actually computes the desired values of $P_0^{i*}$ as a by-product. By executing Thompson's algorithm time-wise in reverse (from the end of the lattice to its start), the values of $P_j^{e*}$ are similarly computed as a by-product. The complexity of this algorithm is linear in the number of words in the lattice.

### 5.3 Parsing a Word Lattice using the Heuristic

Prior to the parsing itself, an initial phase must scan the lattice and assign to each word in the lattice its corresponding $P^*$ score. Subsequently, the lattice words are sorted by their $P^*$ scores. Words that have the same $P^*$ score are ordered by their $P$ score. The sorting allows the parsing algorithm to efficiently select the next word of the lattice that is to be parsed.

The parser processes the lattice word after word, in the order determined by the pre-sorting. The first word chosen to be processed is the word that has the greatest $P$ score among the words of greatest $P^*$ score. Each individually parsed word creates an initial island, which is then combined with all existing neighboring islands. Islands are stored in a list, which is also sorted according to the $P^*$ scores of the islands.

Islands that correspond to full paths through the lattice, that are found to be valid full-parses, are stored in a third list, ranked by their actual overall score. Once the algorithm reaches the point where the $P$ score of an existing full parse is greater or equal to the $P^*$ score of the next word yet to be processed, it may terminate. The current best full parse is the desired solution.[4]

## 6 Conclusions and Future Directions

In this paper we presented a new efficient algorithm for parsing word lattices. Based on an algorithm for parsing substrings, we developed a Generalized LR style parser that can parse an input string in any given word order. This algorithm parses words of the input as substrings, and combines these parsed islands with other neighboring islands as they become available. We described an $A^*$ heuristic that can be used to impose an ordering on the lattice words. This heuristic guarantees an efficient computation strategy for finding the most probable grammatical path of words through the lattice.

The algorithm presented in this paper has been implemented and tested on a small set of preliminary examples. However, we have yet to conduct a large scale experiment to evaluate the performance of the proposed algorithm in parsing actual speech produced word lattices. The efficiency of the parser with realistic large scale grammars will need to be tested as well.

## References

[Bates and Lavie, 1991] J. Bates and A. Lavie. Recognizing Substrings of LR(k) Languages in Linear Time. *Technical Report CMU-CS-91-188*, 1991.

[Bates and Lavie, 1992] J. Bates and A. Lavie. Recognizing Substrings of LR(k) Languages in Linear Time. In *Proceedings of POPL'92*, pages 235–245, Albuquerque, NM, 1992. ACM press.

[4] If *all* best solutions are desired, the algorithm must continue until the $P$ score of the best full parse is strictly greater than the $P^*$ score of the next word to be processed.

[Chow and Roukos, 1989] Y. Chow and S. Roukos. Speech Understanding Using a Unification Grammar. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'89)*, pages 727–730, 1989.

[Kita *et al.*, 1989] K. Kita, T. Kawabata, and H. Saito. HMM Continuous Speech Recognition Using Predictive LR Parsing. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'89)*, pages 703–706, 1989.

[Rekers and Koorn, 1991] J. Rekers and W. Koorn. Substring Parsing for Arbitrary Context-Free Grammars. In *Proceedings of Second International Workshop on Parsing Technologies*, pages 218–224, Cancun, Mexico, 1991.

[Saito and Tomita, 1988] H. Saito and M. Tomita. Parsing Noisy Sentences. In *Proceedings of 12th International Conference on Computational Linguistics (COLING)*, pages 561–566, Budapest, Hungary, 1988.

[Saito, 1990] H. Saito. Bi-directional LR Parsing from an Anchor Word for Speech Recognition. In *Proceedings of 13th International Conference on Computational Linguistics (COLING)*, Helsinki, Finland, 1990.

[Thompson, 1989] H. Thompson. A Chart Parsing Realisation of Dynamic Programming, with Best-first Enumeration of Paths in a Lattice. In *Proceedings of European Conference on Speech Communication and Technology (Eurospeech'89)*, pages 378–381, Paris, France, September 1989.

[Thompson, 1990] H. Thompson. Best-first Enumeration of Paths through a Lattice - an Active Chart Parsing Solution. *Computer Speech and Language*, 4(3):263–274, 1990.

[Tomita, 1986a] M. Tomita. An Efficient Word Lattice Parsing Algorithm for Continuous Speech Recognition. In *Proceedings of IEEE-IECEJ-ASJ International Conference on Acoustics, Speech and Signal Processing (ICASSP'86)*, pages 1569–1572, Tokyo, Japan, April 1986.

[Tomita, 1986b] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Hingham, Ma., 1986.