Succinct Data Structures for Tree Adjoining Grammars

James King
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC, V6T 1Z4, Canada
king@cs.ubc.ca

Abstract

We present a set of data structures for the succinct representation of tree-adjoining grammars. The savings in space requirements compared to a naive representation depend on the grammar, but will in general be between a constant factor and a logarithmic factor. We also present a modification of the PATRICIA tree structure and propose a possible modification of the Directed Acyclic Word Graph (DAWG) structure. These modifications enable more efficient queries regarding lexicographical rank.

1 Introduction

Tree-adjoining grammars (TAGs) are tree generation systems first introduced in (Joshi et al., 1975). A comprehensive survey can be found in (Joshi and Schabes, 1997). Tree adjoining grammars generate mildly context-sensitive languages and can also be used to generate context-free languages in a far more concise and intuitive way than CFGs.

In the realm of tree-adjoining grammars, much work has been done with regards to time efficiency. However, little has been done towards space-efficient representation of TAGs. In this paper we present a space-efficient representation format for TAGs.

There are five components that make up a tree adjoining grammar.

1. Σ is the set of terminal symbols
2. NT is the set of non-terminal symbols
3. S is the special 'start' non-terminal symbol
4. I is the set of *initial trees*
5. A is the set of *auxiliary trees*

Storage of Σ and NT is covered in Sections 2 and 4. Storage of I and A is covered in Section 3. Storage of S is trivial and is mentioned in Section 2.1. In Section 5 we discuss the difficulty of parsing with this data structure and in Section 6 we discuss the usefulness of this structure in practice.

2 Symbol Representation

2.1 The Start Symbol

The start symbol is a non-terminal symbol. Its only significance is that it must be the root of any derived tree, and it cannot appear anywhere else in a derived tree. If we consider the start symbol to be a standard non-terminal, the other mechanisms of tree-adjoining grammars (i.e. constraints on adjoining) can ensure that the start symbol behaves properly. Therefore we do not need to consider it as special, though the grammar certainly will.

2.2 Terminals and Non-Terminals

In a TAG, terminals and non-terminals are generally not labeled succinctly. That is, their labels generally use more bits than necessary. In terms of succinct representation we have two main options: fixed length representation and variable length representation. Variable length representation is potentially more concise (depending on the relative frequencies of symbols in the grammar), but fixed length representation allows us to navigate through trees more easily.

2.2.1 Fixed Length Representation

In a fixed length representation, terminals and non-terminals can be represented with as few as $\lceil \log |\Sigma| \rceil$ bits and $\lceil \log |\mathrm{NT}| \rceil$ bits respectively. We will use τ to denote the number of bits used to represent a terminal and π to denote the number of bits used to represent a non-terminal.

We will use a bit string of length τ to represent each type of terminal node in our structure. The intuitive way

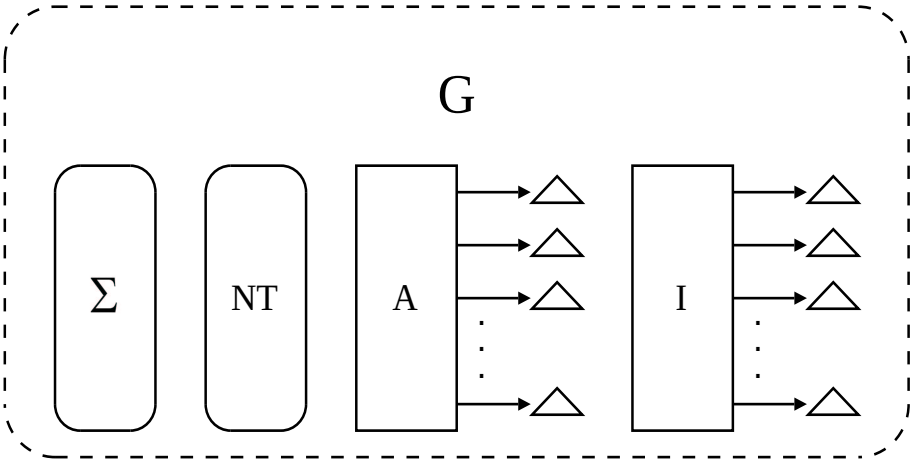

Figure 1: A high level view of the TAG data structure for a grammar G. The boxes represent arrays, the rounded boxes represent RIP arrays (see Section 4.3), and the triangles represent tree blocks (see Section 3.5).

to do this is to store the full labels of all terminals in an array sorted in lexicographical order, though we will actually use a more sophisticated and succinct array structure (see Section 4, especially 4.3). The bit string representation of a terminal will then be its index in that array. We can do the same with non-terminals, storing them in a separate array. It should be noted that this array is not actually necessary since non-terminal labels do not actually matter for the purposes of generation, parsing, etc. However, many would find it an advantageous feature to know that a non-terminal is, for example, of the type "NP" as opposed to simply "001010".

Using bit sequences of different lengths to represent terminals and non-terminals enables us to store trees more succinctly. However, when reading a node of a tree, we will not know in advance how many bits are in its representation. This is problematic when using a succinct data structure; since we waste no space there are no delimiters, so we will not know when the representation of a given node ends and the representation of the next one begins. In parse trees every internal node is a non-terminal and every leaf is a terminal, so we can tell from the structure of the tree how many bits to read. However, this is not the case with the elementary trees and partially derived trees of a TAG — non-terminal nodes can be leaves. For this reason we will prefix every terminal with a 0 and every non-terminal with a 1. Reading a 0 tells us that the representation of this node is $\tau + 1$ bits long, reading a 1 tells us that the representation of this node is $\pi + 3$ bits long. The extra 2 bits in the representation of a non-terminal are explained in Section 3.4.

2.2.2 Variable Length Representation

Variable length encoding of terminals and non-terminals can be done via Huffman coding (Huffman, 1952). Symbols that appear more frequently in the representation of the grammar will be given shorter codes. The encodings of the symbols that minimize the total amount of space used can be determined very efficiently given the frequency of each symbol. The encodings are stored in a binary tree with the symbols at the leaves, where the path to a leaf determines the encoding of the corresponding symbol. Starting at the root and moving towards the leaf, simultaneously starting at the first bit of the encoding and moving towards the last, following the edge to a left child denotes a 0 in the current position of the encoding and following the edge to a right child denotes a 1 in that position.

The structure of this Huffman tree can be stored in $2n + o(n)$ bits using the technique of (Munro and Raman, 1997). This representation still enables navigation operations in constant time. The leaves of the tree can be stored in a separate array. These leaves will not be symbols, but rather lexicographical ranks of symbols (see Section 4). Since the Huffman tree will not necessarily be a complete binary tree (in fact we gain nothing from compression if it is), retrieval of a leaf's label will take linear time in the worst case. These labels will not be needed often; one label query per symbol in a string to be parsed is sufficient. Therefore this linear query time is acceptable.

When reading a variable length label, we will not know in advance when the label ends. We must traverse the Huffman tree while reading it. We will know that we have read the last bit of the label when our traversal of the Huffman tree reaches a leaf.

3 Tree Representation

There are three major components of the representation of each elementary tree. The first is the structure of the tree, the second is the labeling of the tree's nodes, and the third is the set of adjoining constraints on each node (see Section 3.4). An overview of the conversion process from a tree to a succinct bit string is shown in Figure 4.

3.1 Tree Structures

It is well known that the structure of a general tree can be stored using $2n$ bits, where n is the number of nodes in the tree. It is also known that this representation is within an additive logarithmic term of the lower bound. This succinct representation is constructed using balanced parentheses. A leaf is represented by a closed pair $()$. Recursively, an internal node is represented by a closed pair with its child information inside. This child information will simply be the concatenation of the representations of the node's children. It is not difficult to see that this representation corresponds to a depth-first traversal of the tree, where a left parenthesis means a node is being visited for the first time and a right parenthesis means all of a node's children have been visited. An example of this representation is given in Figures 2 and 3. This parenthesis representation can be converted to a string of $2n$ bits by replacing every left parenthesis with a 0 and every right parenthesis with a 1.

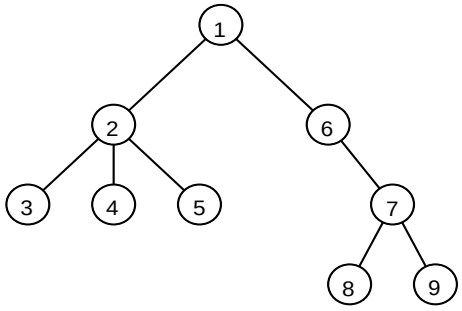

Figure 2: A tree.

```
123445526788997 61
((()()())((()())))
0001010110001011 11
```

Figure 3: Representations of the same tree.

Though this structure is very compact, traversal of a tree in this representation is cumbersome. In the case of binary trees, the structure can be augmented to enable navigational operations in constant time using only sublinear overhead (Munro and Raman, 1997). Elementary trees in a tree-adjoining grammar are not necessarily binary, but we could use dummy nodes to convert every elementary tree to a binary tree, less than doubling the number of nodes in the worst case.

This augmented structure could be considered when storing a TAG with extremely large elementary trees, but elementary trees of a TAG are typically far too small to make this worthwhile. In practice an elementary tree with more than 10 nodes would be anomalous, whereas the primary application of the augmented succinct structure is extremely large suffix trees. We will make the assumption that the elementary trees are small enough that not only will they easily fit in memory, but also it is not prohibitively expensive to look at the entire tree when a tree is accessed at all. Because the trees can be reconstructed from their succinct representations in linear time, reconstructing a tree from a space efficient representation before it is read is at most a constant factor slower than reading the entire tree from any time efficient representation.

Based on our small tree assumption we choose to store tree structure using the simple $2n$ bit representation. We note that, if this small tree assumption were removed, we could use the augmented structure of Munro and Raman. This would ensure efficient traversal of the trees regardless of size. However, it could cost us up to twice as many bits in structural representation (to make our trees binary) and would force us to use fixed length representation for terminal and non-terminal symbols.

3.2 Foot Nodes

Every auxiliary tree has a foot node. The foot node is a non-terminal leaf with the same label as the root of the auxiliary tree. Because the foot node has the same label as the root of the same tree, we need not store its label. As described in Section 3.4 every non-terminal is prefixed with a 2 bit code (after the 1 that tells us it is a nonterminal). We can use one of these codes to label the foot node of an auxiliary tree. This special code essentially says that the node is the foot node, the node's true label is that of the root, and the node's label is omitted. In this way we handle foot nodes at no storage cost except for the three bit code.

3.3 Marking Non-terminals for Substitution

In a partially derived tree, certain non-terminal nodes are marked for substitution. Substitution must occur at these nodes, adjunction cannot occur at these nodes, and substitution cannot occur at any other nodes. However, by the definition of a tree-adjoining grammar, a node is marked for substitution if and only if it is a non-terminal leaf node that is not the foot node of an auxiliary tree. These markings are therefore implicit and, in any given tree, marked nodes can easily be determined. For this

reason we do not need to worry about the issue of marking for substitution.

3.4 Adjoining Constraints

Adjoining constraints are potentially the most space-consuming component of tree-adjoining grammars. Every internal node in an elementary tree has adjoining constraints that dictate which auxiliary trees can be adjoined at that node. The types of adjoining constraints are as follows:

- Null Adjunction (NA): no adjunction is allowed at this node.
- Obligatory Adjunction ($OA(T)$): an auxiliary tree from the set $T \subseteq A$ *must* be adjoined at this node.
- Selective Adjunction ($SA(T)$): an auxiliary tree from the set $T \subseteq A$ can be adjoined at this node, though no adjunction is required.

The type of adjoining constraint will be stored with every non-terminal in a 2 bit code, after the 1 that tells us it is a non-terminal and before the node's label. We also use one of these codes to identify foot nodes as mentioned in Section 3.2. The codes are as follows:

- 00 - Null Adjunction
- 01 - Foot Node
- 10 - Selective Adjunction
- 11 - Obligatory Adjunction

Null adjunction requires no additional representation, but with obligatory adjunction and selective adjunction we must store the set $T \subseteq A$. At first glance it seems that there are $2^{|A|}$ possibilities for this set and that its representation requires $|A|$ bits. However, using R to denote the non-terminal symbol on which we are considering adjunction, we make an observation that can save us a considerable amount of space. We observe that T can be more strictly be defined as a subset of A_R, where $A_R \subseteq A$ is defined as the set of auxiliary trees that have the non-terminal symbol R as their roots. This restricts the number of possibilities of T to $2^{|A_R|}$, allowing us to represent T using only $|A_R|$ bits.

In order to use this optimally compact representation of T, we must sort the auxiliary trees by root. This is not a problem since the auxiliary trees need not be in any other order. The auxiliary trees will therefore be indexed by numbers from 0 to $|A|$; in a table of size $|NT| \log |A|$ we can store a map from each non-terminal symbol to the index of the first tree with that symbol as the root. We store T using a bit string of length $|A_R|$. The tree in A_R with the i^{th} smallest index is a member of T if and only if there is a 1 in the i^{th} position of this string.

We must store the size $|A_R|$ for each non-terminal R. When we read the nodes of a tree to reconstruct it, we can construct a list of the nodes that require set storage (i.e. the nodes marked for OA(T) or SA(T)) and make note of the sizes of the sets. This is necessary so that, when reading the sets, we know when a set representation begins and ends.

Storing adjoining constraints will be the greatest cost for a typical TAG. Even assuming that the largest elementary tree is of constant size, it could cost us on the order of

$$|A| \cdot |A \cup I|$$

bits. Our reduced subset representation from $|A|$ to $|A_R|$ bits is therefore extremely important.

3.5 Overview

Now that we have described how to store the more complex components, we must explain how they are brought together. Each elementary tree will be stored in a single block that is a contiguous string of bits. The structure of these tree blocks is simple. Each consists of the tree's structure, followed by the tree's nodes, followed by the adjoining constraints of the nodes (See Figure 5). We will store a header for each tree that holds three pieces of information: the address of the start of the block, the number of terminals in the tree, and the number of non-terminals in the tree. This is sufficient for the reconstruction of a tree.

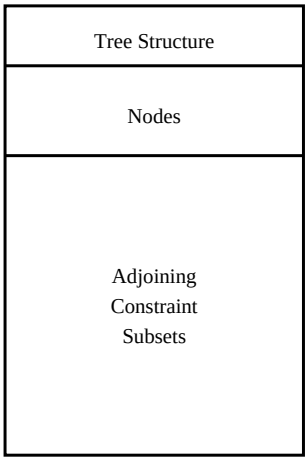

Figure 5: The block of bits for a tree.

The sets I and A of elementary trees will each be stored as an array of these headers. The format of the headers is the same for both. When looking at a tree we will know whether it is an initial tree or an auxiliary tree based only on what set we came from. It is crucial to know what type of tree it is — auxiliary trees will have a foot node

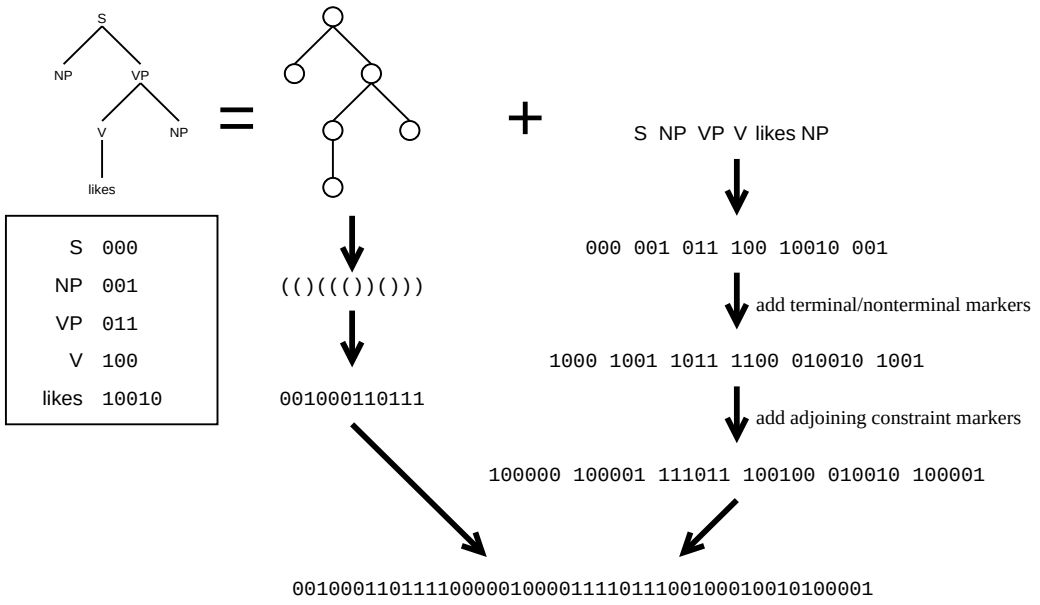

Figure 4: The conversion process from a tree to a succinct bit string. The VP node is marked for obligatory adjunction, though its adjunction set is omitted from the diagram. All other nodes are marked for null adjunction.

with no label, which means that the adjoining constraints will start π bits before they would in an initial tree with corresponding nodes.

4 Symbol Set Representation

Two storage options are immediately apparent for the sets Σ and NT.

4.1 Sorted Arrays

The first option is sorted arrays. This is the simplest way to store these sets. Since the index of a symbol is stored implicitly (by the symbol's location in the array), a sorted array has the advantage that there no overhead in terms of space requirements. Given an index, we can find the corresponding terminal or non-terminal in constant time. Given a terminal or non-terminal, we can find its index in $\log|\Sigma|$ or $\log|NT|$ time respectively via binary search. Since few index lookups will ever be performed (basically only one per symbol of a string being parsed), this logarithmic time cost is acceptable.

4.2 PATRICIA Trees

The second storage option for our symbol sets is PATRICIA trees. PATRICIA trees are trees in which each edge is labeled with one or more characters and each leaf corresponds to a string in our set. The string at a leaf will be the string formed by concatenating the characters of the edge labels on the path from the root to that leaf. For a detailed explanation of PATRICIA trees, see, for example, (Knuth, 1998). The most attractive feature of PATRICIA trees is that any prefix of a string in the set is stored only once. For large string sets this can eliminate a great deal of redundant data. See Figure 6 for an example of a PATRICIA tree and how it eliminates redundant data.

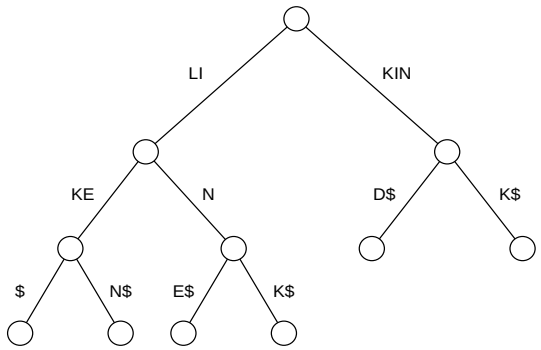

Figure 6: A PATRICIA tree storing the words LIKE, LIKEN, LINE, LINK, KIND, and KINK.

Testing if a string is stored in a PATRICIA tree is very fast. However, speed is not of great concern to us for our current application, especially when the difference is only a logarithmic factor. Elimination of redundant prefixes is the only significant advantage that PATRICIA trees give to us. Since the lexicographic ranks of strings in a PATRICIA tree are not stored implicitly, we must store the rank with every leaf. This, combined with the pointers involved in a typical implementation of a PATRICIA tree,

require storage that we would rather not use. Moreover, to determine the string corresponding to a given rank, we must essentially do a binary search of the leaves of the PATRICIA tree.

4.3 RIP Arrays

To combine the advantages of sorted arrays with those of PATRICIA trees, we introduce a hybrid data structure which we will call the Rank Indexed PATRICIA (RIP) array. A RIP array stores strings in an array in sorted order, but also includes pointers to eliminate storage of redundant prefixes. The trick to using these pointers is to store the strings in reverse order, with the last character appearing first. In this way, prefixes of the words become suffixes in the RIP array. If two strings in the set with ranks i and j have a prefix in common, the representation of the j^{th} word will contain the different ending, stored in reverse order, followed by a pointer to the common prefix as represented for the i^{th} word. An example of this structure is shown in Figure 7.

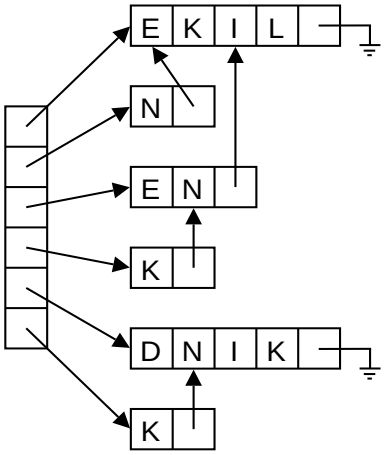

Figure 7: A RIP array storing the words LIKE, LIKEN, LINE, LINK, KIND, and KINK in sorted order.

Compared to PATRICIA trees, RIP arrays use slightly less storage (the pointers to leaf nodes in the PATRICIA tree are emulated for free in the corresponding RIP array). The main advantages of RIP arrays are that the lexicographic rank of a string is stored implicitly and rank to string queries require only constant time. The disadvantage is that, to compare a given string to a string in a RIP array, the entire string must be read from the RIP array, whereas PATRICIA trees can detect a different character after only reading the string up to that character. This is inconsequential to us; we will store our word sets in RIP arrays because of the storage savings.

4.4 DAWGs

There exists another data structure for string sets, the Directed Acyclic Word Graph (DAWG), that is even more compact than PATRICIA trees or RIP arrays (Blumer et al., 1985). A DAWG is a directed acyclic graph one source (the start node) and one sink (the end node). Each other node has a character associated with it. For each string in the set there is a path from the start node to the end node such that the internal nodes of the path spell out the string (see Figure 8).

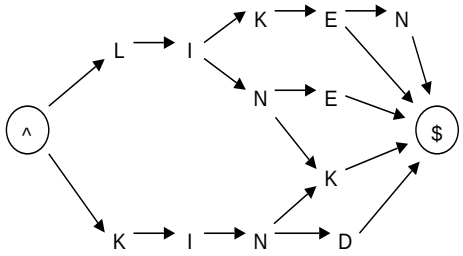

Figure 8: A DAWG storing the words LIKE, LIKEN, LINE, LINK, KIND, and KINK in sorted order.

DAWGs eliminate not only redundant prefix data, but also redundant suffix data. Unfortunately DAWGs are such that determining the lexicographical rank of any string in the set would require a depth-first search for that string. This could take linear time, even with the assumption that the longest string is of constant length.

An issue that we consider is this: can we augment a DAWG, at most multiplying its size by a constant, such that we can perform rank queries in logarithmic time, or even $\log^2$ time? Perhaps. One idea is this: we construct a series of sub-DAWGs. Each sub-DAWG has at most half the vertices of the previous one, until we get to a DAWG of constant or logarithmic size. For efficient rank querying we require that each sub-DAWG represents a word set that has a *representative sample* of the word set of the previous one. We define a representative sample as follows:

G' is a sub-DAWG of a DAWG G. S is the word set represented by G. $S' \subseteq S$ is the word set represented by G'. We say that S' has a representative sample of S if and only if, for each pair of strings i, j in S' that are lexicographically adjacent in S' (i.e. their ranks in S' have difference 1), their rank difference in S is at most logarithmic in the size of S.

We can build a series of sub-DAWGs that enable efficient querying if, for any DAWG G, we can build a sub-DAWG G' such that the word set encoded by G' has a representative set of the word set encoded by G. This

can be done trivially, but to ensure that our entire series of sub-DAWGs does not increase the original DAWG by more than a constant factor we need to add the stipulation that G' has at most half as many vertices as G.

Assuming we can create such a series of sub-DAWGs, we can do a rank query for a string s as follows. We start at the smallest DAWG. Searching for a string s we find the last word in that DAWG that comes before s (or if we find s we are done. We then move on to the next smallest DAWG and do the same, though we use that word as a starting point for our search. We find the last word in that second DAWG that comes before s, then move on to the next smallest, etc.

We can store an array for each DAWG with one entry per string. Given the rank of a string s in G', this array will tell us the rank of s in G. From these arrays we can determine the total number of strings we skip in our query, thereby determining the rank of s. The fact that each sub-DAWG encodes a representative set of its 'parent' DAWG ensures that we never spend more than logarithmic time in any given DAWG. Since every sub-DAWG has at most half as many vertices as its parent DAWG, there are at most a logarithmic number of DAWGs in our series. Therefore a rank query will take at most $\log^2 |S|$ time.

It is not immediately clear that a good series of sub-DAWGs can be built. This is something we will continue to investigate.

5 Parsing

Unfortunately efficient parsing is not possible with a TAG stored in this format. All efficient TAG parsers require at least efficient querying of the form "Given this symbol, which trees could it have come from?" See, for example, (Joshi and Schabes, 1997), (Nederhof, 1999), (Satta, 1994). In the worst case this information could require

$$\Theta\left(|\Sigma \cup \mathrm{NT}| \cdot |\mathrm{I} \cup \mathrm{A}|\right)$$

additional storage. In practice, however, less storage will generally be required. Each terminal symbol will exist in only a small number of elementary trees, so we can store this information for terminals in lists. The number of non-terminals will typically be far smaller than the number of terminals ($|\mathrm{NT}| < \sqrt{|\Sigma|}$), so we can store the information for non-terminals in an array.

6 Practical Considerations

A group of researchers at UPenn has constructed tree-adjoining grammars for both English (Group, 2001) and Korean. In practice, TAGs are unlikely to be much bigger than the TAG for the English language. The XTAG group's representation of this grammar is about 2MB large. Any reasonably powerful computer can therefore hold the entire grammar in memory. For this reason, it is unlikely that a succinct TAG representation would be useful in practice. It is somewhat conceivable that certain portable computers with less than 2MB of memory could be used to store a large TAG (a very sophisticated pocket translator, for example), but it is extremely unlikely that such a computer would be powerful enough to do anything with such a grammar. TAG parsing, for example, cannot be done in much less than $O(n^6)$ time (Satta, 1994).

The XTAG system also uses implicit morphology in its grammar. This saves a large amount of space in the storage of Σ. This morphology is not in the strict definition of tree-adjoining grammars, and it cannot be ported to our TAG structures. In short, practical TAGs make use of mechanisms that are not in the strict definition of TAGs, so this makes our structure even less viable in practice.

7 Conclusion

We have presented a set of data structures for the efficient storage of tree-adjoining grammars. Though the savings in storage is at least a significant constant factor, the usefulness of the succinct representation is questionable; TAGs are not extremely large in practice, so time efficiency is far more important than space efficiency.

More interesting, perhaps, are the representation of PATRICIA trees and proposed possible representation of DAWGs that enable faster queries related to lexicographical rank. We are particularly interested in pursuing the augmented DAWG structure since rank related queries in DAWGs currently require linear time — an improvement of this time to logarithmic would be significant.

References

A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. 1985. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40(1):31–55, July.

XTAG Research Group. 2001. A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania.

David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September.

Aravind K. Joshi and Yves Schabes. 1997. Tree-adjoining grammars.

Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. 1975. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163, February.

Donald E. Knuth. 1998. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc.

J. Ian Munro and Venkatesh Raman. 1997. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science: October 20–22, 1997, Miami Beach, Florida*, pages 118–126. IEEE Computer Society Press.

Mark-Jan Nederhof. 1999. The computational complexity of the correct-prefix property for TAGs. *Computational Linguistics*, 25(3):345–360.

Giorgio Satta. 1994. Tree adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics*, 20(2):173–192.