To the Graduate Council:

I am submitting herewith a thesis written by Eric Wright Richardson entitled "A Shortest Common Supersequence Approach to Error Correcting Parsing with Regular Grammars." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr. Jens Gregor, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

A SHORTEST COMMON SUPERSEQUENCE APPROACH TO ERROR CORRECTING PARSING WITH REGULAR GRAMMARS

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Eric Wright Richardson

December 1996

Abstract

We are concerned with those problems where information can be encoded as strings, and classes of information can be modeled by regular grammars. When the string encoding is imperfect, it may be that classification is impossible unless some form of error correction is used. To handle this situation, we study the Lyon parser which is an error correcting modification of the Earley parser for context-free grammars. In particular, we simplify this parser from working with context-free grammars to working only with regular grammars. Once simplified, we try different approaches to reducing the number of deletion edits to the string.

We compare two modifications of the Lyon parser that reduce the number of deletions: one that uses an adjustable queue, and one that uses a shortest common supersequence approach. The focus of this thesis is the comparison of these two approaches. We compare these approaches by parsing strings using three different types of regular grammars: feed forward, regular expression, and free-form grammars. Also, parsing test strings that generate many deletions when using a free-form grammar are examined. For these three types of grammars we compare the number of deletions for each approach and, for the free-form grammars, their timings. In conclusion, for grammars requiring multiple deletion passes, we find using the shortest common supersequence to be the better one.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 General Introduction

Many applications of syntactic analysis of strings call for some type of error correction. Keying in information, transmission of encoded information, and sequence comparison in molecular biology are just some of the many problems that involve some sort of error correction [1]. We are concerned with those problems where information is easily encoded into strings and can be categorized into groups. For each of these groups i, we can derive a grammar G_i such that each string in the group is in the language $L(G_i)$. We discard the case where a string can belong to two or more distinctively different languages.

The grammars we will be working with are regular grammars. Figure 1.1 shows two representations of a finite automaton for the grammar in part A; part B has

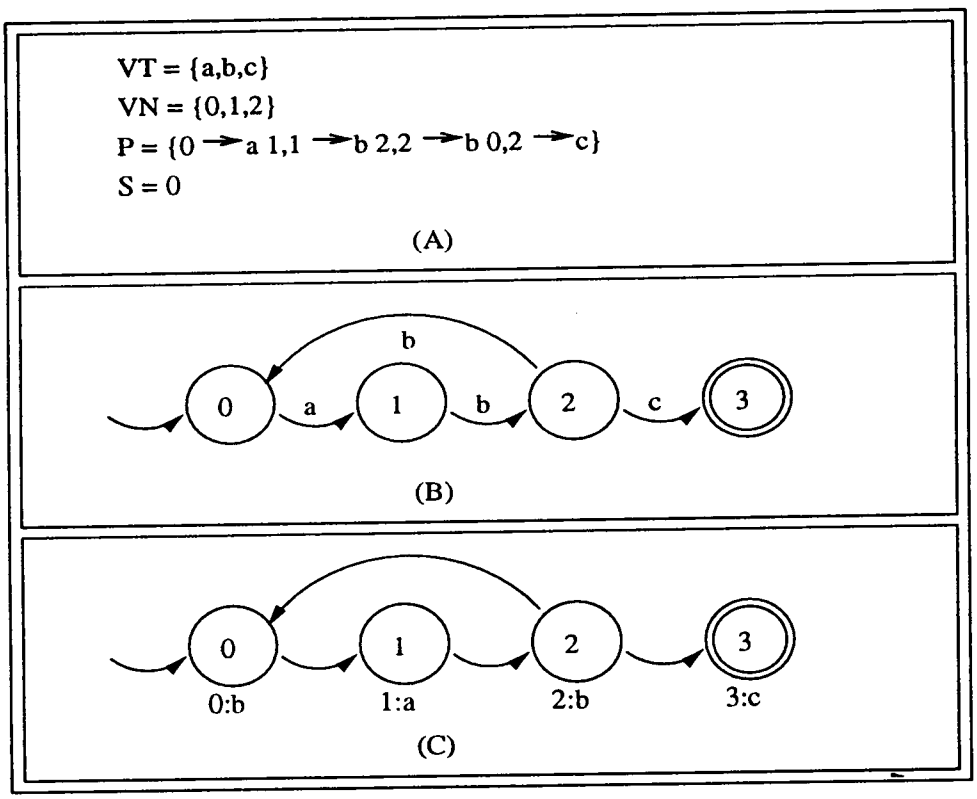

Figure 1.1: Finite automaton representations

states that are numbered and has letters on its transitions to other states; part C moves the terminals, or letters, into the states. A transition is represented by an arc in the automaton. If we consider the letters on the transitions in part B as output, then this represents a Mealy machine. In part C, with the output kept in the states, this represents a Moore machine. The regular grammar and the two finite automata are equivalent [6]. In terms of explaining algorithms, and the grammars themselves, we will represent automata as Moore machines.

1.2 Problem Formulation

Given a test string, we must determine which group, or language, it belongs in. If the test string is not found in any of the languages, then we have a problem in classification. Rather than rejecting the string, we would like to know which group it is closest to. In many applications in which errors may occur, it is desirable to compute how close the test string is to one of the strings in $L(G)$. This requires error-detecting parsing which, for the regular case, can be described as an alignment problem. Using error correction, we can develop an alignment based on the string edits match, substitution, insertion, and deletion. These edits allow us to edit the test string for its closest fit to one of the strings in $L(G)$.

Figure 1.2 shows a grammar, its edit costs, and some test strings. Of the

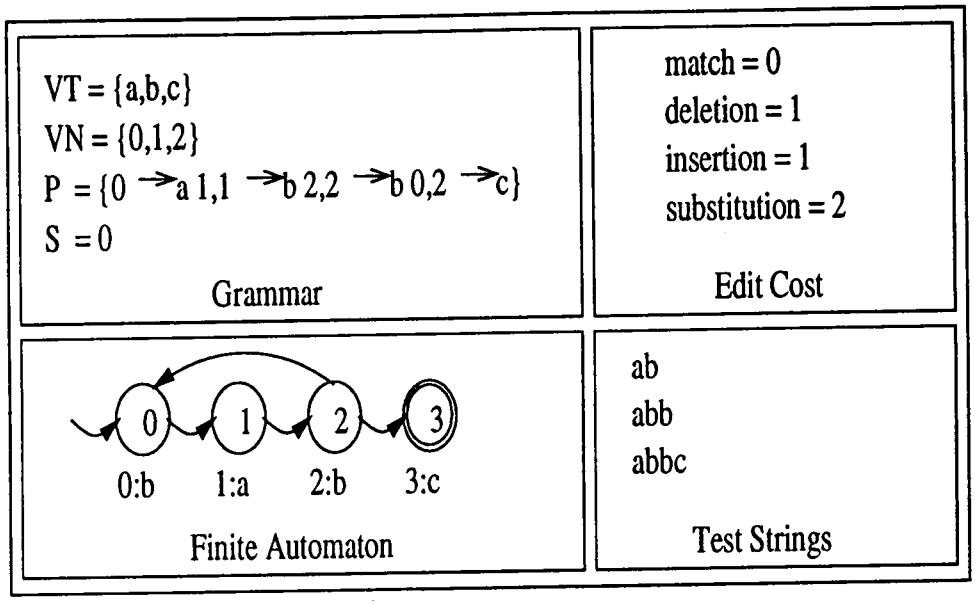

Figure 1.2: Sample grammar and strings

many edit sequences that can be derived Table 1.1 shows just some of the ways of mapping these test strings into $L(G)$ via edits. Note that different sequences of edit operations may result in the same edit cost. We are only looking for one with the lowest cost.

One common setting for these edit costs assigns deletions and insertions cost one; substitutions, cost two; and matches, cost zero. Another cost setting, called Levenshtein, assigns edit operations cost one while matches have a cost of zero [1]. The effect of this setting is that the lowest cost is a count of the fewest edits required.

Table 1.1: Parsing test strings into an automaton using error correction

Test string	Alignment				Total cost
ab	a mat +0	b mat +0	(c) del +1		1
abb	a mat +0	b mat +0	(c) sub +2		2
abb	a mat +0	b mat +0	(b) ins +1	(c) del +1	2
abb	a mat +0	(b) ins +1	b mat +0	(c) del +1	2
abbc	a mat +0	b mat +0	(b) ins +1	c mat +0	1
abbc	a mat +0	(b) ins +1	b mat +0	c mat +0	1

1.3 Previous Work

The Earley algorithm for a context-free language $L(G)$ is a recognizer [3]. If the input string is in $L(G)$, then the algorithm accepts it, otherwise it rejects it. Others have implemented the Earley algorithm as a parser for weighted grammars [2], and as an error-correcting parser [9].

There are many different types of alignment problems. One aligns a string against another string where we see how closely these two strings are related [13]. Building on this, another approach aligns a string against a directed network that has no cycles [1]. A different approach aligns a string to a left-to-right network which may contain self-loops but not cycles [4]. Parsing a string using a regular expression is yet another approach, where only two passes are needed to calculate deletions correctly [11]. Two approaches that parse a string using a regular grammar are covered in this paper. These two approaches are derived from the Lyon parser for context-free grammars, which is a version of Earley's modified to handle error correction [9].

1.4 Alternative Approaches

After modifying the Lyon parser for context-free grammars to work only with regular grammars and changing the underlying data structure (linked lists) to work within a matrix, we can lower the number of edit operations needed to be

performed. In the list approach, it is often the case that many deletion operations are tried more than once on the same situation. A list keeps track of those operations taken to process a particular symbol in the test string. In converting the list data structure to a matrix, we associate a list to a column. This means that each column represents how a particular symbol in the test string is processed.

We propose two new parsers that reduce the number of deletions attempted: an adjustable queue approach, and a shortest common supersequence approach. The two parsers are quite similar in coding, only the deletion routines differ. Both handle matches, insertions, and substitutions in a straightforward manner in one pass over the column. For each production, we try a match or substitution, and an insertion. Which ever edit operation has the lowest cost is what makes a link in the matrix (as well as costs that tie).

One parser that reduces the number of deletions attempted by the list approach is a queue approach. We check all transitions from the current state being processed for any deletions that can be done. If we can do a deletion operation on a state a transition takes us to, then we add that state to the end of our queue. We process the rest of the states in the grammar, then process the queue. Using this method, we add a state to the queue only if a deletion operation is performed. Eventually, the costs in the column will settle and our queue will empty.

Another parser that is an improvement on the list approach is the use of the shortest common supersequence in handling deletions. The shortest common

supersequence (SCSS) is derived from a grammar. This sequence is a list of states to visit while checking for deletions in a column. The SCSS has a fixed length, unlike the queue approach which is dependent on the test string in determining its size.

1.5 Thesis Outline

Chapter 2 deals with the parser and the background behind it. Starting with the Earley parser for context-free grammars and seeing this develop into the Lyon parser. First we convert it to a parser that handles error correction for context-free languages, then we modify it to only work for regular grammars. Once modified to work only with regular grammars, we change the underlying data structure from linked lists to a matrix. Finally, once in this matrix form, we discuss our queue approach for handling deletions.

Chapter 3 discusses the shortest common supersequence approach. Starting with the basics, we explain what it is and why we want to use it. We also show how we can use a candidate CSS which is easier to find than the SCSS. If we want to find the shortest CSS, we can use the candidate to prune away the search space needed to find the shortest CSS. Once we have the SCSS, we can use it in our modified Lyon parser. This parser also uses a matrix as its data structure for keeping track of the parse information. Like the queue approach in the previous

chapter, we use the SCSS here for calculating deletions.

Finally, Chapter 4 shows a comparison between the three approaches. This comparison is done on three types of regular grammars which are represented as finite automata: feed forward (no cycles), regular expression (no crossed transitions), and free-form regular grammars (massive pumping with no restrictions). The focus is on the latter type of grammars as they show the queue approach having to check some paths in a column more than twice when parsed with certain test strings. From these comparisons, we draw our conclusions.

Chapter 2

Background

2.1 Earley Parser

In simple terms, Earley describes his algorithm with the use of three functions in the building of lists [3]. Each list corresponds to one of the input strings' symbols and the last list represents the end of the input string. A list can be kept in a linked list and is initially empty. Figure 2.1 lists the Earley algorithm in pseudocode. Using this algorithm, we can only tell if a parse of a string is successful or not. A starting element $(\$ \rightarrow \cdot S, 0)$ is placed into the first list. From here, the predictor, completer, and scanner functions start their work.

Before we go over the three functions, we need to define the union operator first since this is how an element gets added to a list. For $\text{list}[j] \cup \{(X \rightarrow \alpha \cdot Y\beta, i)\}$ we can define the union function $\cup$ in two ways [12]: if $(X \rightarrow \alpha \cdot Y\beta, k) \in \text{list}[j]$ and

```
01  list[0] = { ($ → ·S, 0) }
02  for j = 1 to n do list[j] = NULL
03  j = 0 /* current list we are in */
04  while j <= n do /* length of test string is n */
05     for each (X → α · Yβ, i) ∈ list[j] do
06        if Y ∈ N then /* predictor */
07           for each Y → γ ∈ P
08              list[j] = list[j] ∪{(Y → ·γ, j)}
09        else if (Y ∈ T and j ≠ n) then /* scanner */
10           if Y = a_{j+1} then
11              list[j+1] = list[j+1] ∪ { (X → αY · β, j) }
12        else if (Yβ = λ) then /* completer */
13           for each (A → δ · Xζ, k) ∈ list[i]
14              list[j] = list[j] ∪{(A → δX · ζ, k)}
15     if (no new item has been generated in list[j]) then j = j + 1
16  if (($ → S·, 0) ∈ list[n]) then accept string
17  else reject string
```

Figure 2.1: Earley algorithm (adapted from [12])

$k \leq i$, then do nothing, else add $(X \rightarrow \alpha \cdot Y\beta, i)$ to list[j]; or if $(X \rightarrow \alpha \cdot Y\beta, k) \in$ list[j] and $k = i$, then do nothing, else add $(X \rightarrow \alpha \cdot Y\beta, i)$ to list[j]. If we accept the input string, then the first rule will only allow us to find one edit sequence or parse. With the second rule, if we accept the input string, then we will have all edit sequences or parses of the input string.

The predictor basically loads the possible paths onto the current list that the input string may or may not take. When the pointer, represented by a dot, in the current element points to a nonterminal Y, the predictor will try to add all productions $(Y \rightarrow \cdot\gamma, j)$ that have that nonterminal in the left hand side (LHS) of the production (lines 7-8). The pointer, for each of these new elements in the list, will point to the first symbol in the right hand side (RHS) of that production.

The Greek letters $\{\alpha, \beta, \delta, \gamma, \zeta\}$ represent any combination of terminals T and nonterminals N from the grammar, or even the empty symbol λ.

The completer is used to update back traces in this or previous lists by adding an element or elements to the current list. What this really means is that when an element in a list currently being looked at has scanned past all the RHS symbols, we try to add to this list a new element or elements. Checking which list this current element was generated from, we add only those elements from previous lists that have their pointers in the RHS to symbols that match the current elements LHS symbol. These newly added elements now have their pointer pointing to the next symbol in their RHS.

The scanner examines the input symbol associated with this list and compares it to the symbol being parsed in the current element. If there is a match, then a new element is unioned wtih the next list and that element's pointer is moved to the next symbol in the RHS of the production. In short, the scanner 'matches' symbols.

Once we have built all the lists (incrementally) we can check the last list to see if we should accept the string. If the completer does not trickle $(\$ \rightarrow S\cdot, 0)$ through from the first list, then we reject the string, otherwise we accept it. This algorithm is noted for having time complexity of $O(n^3)$ and space complexity of $O(n^2)$ for a test string of length n [12]. For unambiguous grammars, the completer operates in time $O(n^2)$ [3].

2.2 Lyon Parser

Lyon added error correction to the Earley parser [9]. Figure 2.2 shows that by adding an associated weight to the list element and by modifying the scanner to allow for edits, we can check how close a match a string is to one of the strings in $L(G)$. The basic changes were: the addition of another field in the element (weight of the accumulated error so far), a back pointer to the element that generated this the element, a special ending symbol # in the start element ($\$ \rightarrow \cdot S\#, 0, 0$), the union operator has been modified, and the scanner has been expanded to allow for error correction. By setting a maximum weight, we can accept or reject the input string depending on its accumulated weight in the last list.

For the Lyon parser, the union operator is more involved than the Earley union operator. Now when we have list$[j] \cup \{(X \rightarrow \alpha \cdot Y\beta, i, w)\}$ the union operator can be defined 2 ways [12]: if $(X \rightarrow \alpha \cdot Y\beta, k, w') \in$ list$[j]$ and $k \leq i$ and $w' \leq w$, then do nothing else if $(X \rightarrow \alpha \cdot Y\beta, k, w') \in$ list$[j]$ and $k \leq i$ and $w' > w$, then replace $(X \rightarrow \alpha \cdot Y\beta, k, w')$ with $(X \rightarrow \alpha \cdot Y\beta, i, w)$ otherwise if $w \leq W_{max}$, then add $(X \rightarrow \alpha \cdot Y\beta, i, w)$ to list$[j]$; or if $(X \rightarrow \alpha \cdot Y\beta, k, w') \in$ list$[j]$ and $w' \leq w$, then do nothing else if $(X \rightarrow \alpha \cdot Y\beta, k, w') \in$ list$[j]$ and $w' > w$, then replace $(X \rightarrow \alpha \cdot Y\beta, k, w')$ with $(X \rightarrow \alpha \cdot Y\beta, i, w)$ otherwise if $w \leq W_{max}$, then add $(X \rightarrow \alpha \cdot Y\beta, i, w)$ to list$[j]$. If we accept the input string, then the first rule will only allow us to find one edit sequence or parse. With the second rule, if we

```
01  list[0] = { ($ → ·S#, 0, 0) }
02  for j = 1 to n do list[j] = NULL
03  j = 0 /* current list we are in */
04  while j <= n do
05     for each (X → α · Yβ, i, w) ∈ list[j] do
06        if Y ∈ N then /* predictor */
07           for each Y → γ ∈ P
08              list[j] = list[j] ∪{(Y → ·γ, j, 0)}
09        else if Y ∈ T then /* scanner */
10           if j ≠ n then
11              if Y = a_{j+1} then
12                 list[j+1] = list[j+1] ∪{(X → αY · β, j, w)} /* match */
13              else
14                 list[j+1] = list[j+1] ∪{(X → αY · β, j, w + p)} /* substitution */
15              list[j+1] = list[j+1] ∪{(X → α · Yβ, j, w + q)} /* insertion */
16           list[j] = list[j] ∪{(X → αY · β, j, w + r)} /* deletion */
17        else if (Yβ = λ) then /* completer */
18           for each (A → δ · Xζ, k, u) ∈ list[i]
19              list[j] = list[j] ∪{(A → δX · ζ, k, u + w)}
20        else (Y = #) then
21           list[n] = list[n] ∪{($ → S#·, i, w + (n − j) * q)}
22     if (no new item has been generated in list[j]) then j = j + 1
23  if (($ → S#·, 0, w) ∈ list[n]) then accept string
24  else reject string
```

Figure 2.2: Lyon algorithm for context-free grammars (adapted from [12])

accept the input string, then we will have all edit sequences or parses of the input string.

Now, when the scanner checks the RHS symbol against the current symbol in the input string, we can add edit costs to the weight of those elements added in the next list due to matches, substitutions, and insertions. In the case of deletions, we add the deletion cost to the weight of the current element and skip over the scanned symbol, this element is then unioned with our current list. The predictor and completer work similar to the Earley parser except the predictor zeroes out the weight for new elements and the completer adds the current elements weight with the weight of the element that generated the current element. The Greek letters are treated the same as in the Earley algorithm.

If we add another field to the element in the list, we can label what type of edit operation was done on each element. After parsing a string using a grammar, we can trace back through the ending element of a parse to the beginning and see what edit operations were used in the alignment. The time and space complexities are the same as without error correction [12].

2.3 From Context-Free to Regular Grammars

When modifying the Lyon algorithm to work for just regular grammars, it simplifies how the program works (Figure 2.3). Since we have only two types of

```
01  list[0] = { ($ → S, 0) }
02  for j = 1 to n do list[j] = NULL
03  j = 0 /* current list we are in */
04  while j <= n do
05     while any (X → αY, w) ∈ list[j] is replaced do
06        for each (X → αY, w) ∈ list[j]do
07           for each (Y → δZ) ∈ P
08              list[j] = list[j] ∪{(Y → δZ, w + r)} /* deletion */
09     if (j ≠ n) then
10        for each (X → αY, w) ∈ list[j]do
11           if (Y = λ) then
12              list[j+1] = list[j+1] ∪{(X → αY, w + q)} /* insert in next list */
13           else for each (Y → δZ) ∈ P
14              if δ = a_{j+1} then
15                 list[j+1] = list[j+1] ∪{(Y → δZ, w)} /* match */
16              else
17                 list[j+1] = list[j+1] ∪{(Y → δZ, w + p)} /* substitution */
18              list[j+1] = list[j+1] ∪{(X → αY, w + q)} /* insertion */
19     j = j + 1
20  if ({(X→ β, k)} ∈ list[n] and β ∈ T ) (chose item with lowest k) then
21     accept it with its weight, otherwise reject it
```

Figure 2.3: Lyon algorithm for regular grammars

productions i.e. A $\rightarrow$ a or A $\rightarrow$ a B, this allows us to combine the predictor with the scanner. The scanner operations wind up adding all the productions to a list. Also, the completer is not needed. Each element is generated from a previously scanned element and can be easily back traced from the last list if the string is accepted. When we have the element, $(X \rightarrow \beta, k)$, in the last list, we can look for a trace starting back from that element (see line 20). Here the Greek letters $\{\alpha, \beta, \delta\}$ represent a terminal from the grammar. The union operator $\cup$ is the same as the second union rule for the Lyon parser for context-free languages. The maximum number of elements in a list is equal to the number of productions in the grammar plus the special element $(\$ \rightarrow S, w)$ for some weight w. Each one of these elements happen to have been generated from one of the scanner operations. For a simple regular grammar (represented as an automaton in Figure 2.4) we see how much space the Lyon parser for CFGs takes as opposed to the Lyon parser for RGs (Figure 2.5).

Two versions of this parser follow the algorithm in Figure 2.3. One version must search the list for a matching element on a union operation, while the other does a table lookup on a union operation. The time complexity for our Lyon regular

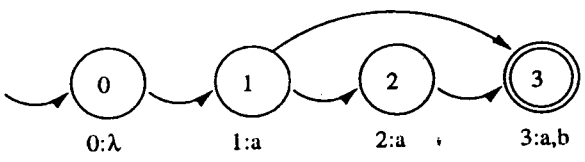

Figure 2.4: Simple finite state automaton

Lyon for context-free grammar

```
Input string is: b
-LIST[0]----------------------------
 1   $ ->  . S0  #  0 0 INIT
 2  S0 ->  .  a T1  0 0 PRED (1)
 3  S0 ->  a  . T1  0 1 DEL  (2)
 4  T1 ->  .  b     0 0 PRED (3)
 5  T1 ->  .  a T2  0 0 PRED (3)
 6  T1 ->  b  .     0 1 DEL  (4)
 7  T1 ->  a  . T2  0 1 DEL  (5)
 8  S0 ->  a T1  .  0 2 COMP (6,3)
 9  T2 ->  .  a     0 0 PRED (7)
10   $ -> S0  .  #  0 2 COMP (8,1)
11  T2 ->  a  .     0 1 DEL  (9)
12  T1 ->  a T2  .  0 2 COMP (11,7)
-LIST[1]----------------------------
13  S0 ->  .  a T1  0 1 INS  (2)
14  S0 ->  a  . T1  0 1 SUB  (2)
15  T1 ->  .  b     0 1 INS  (4)
16  T1 ->  b  .     0 0 MAT  (4)
17  T1 ->  .  a T2  0 1 INS  (5)
18  T1 ->  a  . T2  0 1 SUB  (5)
19  T2 ->  .  a     0 1 INS  (9)
20  T2 ->  a  .     0 1 SUB  (9)
21   $ -> S0  #  .  0 1 INS  (32)
22  S0 ->  a  . T1  1 2 DEL  (13)
23  T1 ->  .  b     1 0 PRED (14)
24  T1 ->  .  a T2  1 0 PRED (14)
25  T1 ->  b  .     1 1 DEL  (23)
26  S0 ->  a T1  .  0 1 COMP (16,3)
27  T1 ->  a  . T2  1 1 DEL  (24)
28  T2 ->  .  a     1 0 PRED (18)
29  T2 ->  a  .     1 1 DEL  (28)
30  T1 ->  a T2  .  0 2 COMP (20,7)
31  S0 ->  a T1  .  1 3 COMP (25,22)
32   $ -> S0  .  #  0 1 COMP (26,1)
33  T1 ->  a T2  .  1 2 COMP (29,27)
weight of string is 1
```

Lyon for regular grammar

```
Input string is: b
-LIST[0]------------------
 1   $ -> S0    0  INIT
 2  S0 ->  a T1 1  DEL  (1)
 3  T1 ->  b    2  DEL  (2)
 4  T1 ->  a T2 2  DEL  (2)
 5  T2 ->  a    3  DEL  (4)
-LIST[1]------------------
 6   $ -> S0    1  INS  (1)
 7  S0 ->  a T1 1  SUB  (1)
 8  T1 ->  b    1  MAT  (2)
 9  T1 ->  a T2 2  SUB  (2)
10  T2 ->  a    3  SUB  (4)
weight of string is 1
```

Figure 2.5: Comparison of Lyon for context-free grammars and regular grammars

grammar parser is $O(n^2)$ when the union operation does not have to search the list. When having to search the list for matching items we search a maximum of $\sum_{i=0}^{n} i$ times, where n is the number of productions in the grammar. This takes $O(n^2)$ time, which means our parser runs in $O(n^3)$ time.

2.3.1 Matrix Version of the Lyon Parser

Here we focus on a Lyon parser for regular grammars using a matrix instead of linked lists. Now we can use a basic dynamic programming approach in filling in the matrix. In our matrix, we have the states in the grammar represent the rows in the matrix. The columns in the matrix represent one of the symbols in the input string. Each column in the matrix can be thought of as one list in the list approach. A link from one column to the next is like the match/insertion/substitution operations in the list approach. Deletions effect only the column we are currently working on.

2.3.2 Algorithm Explained

We initialize the matrix in a manner that is similar to string matching [13]. The key difference is that instead of aligning a string to a string, we are parsing a string using a regular grammar. Initially, we start out with an empty matrix. We initialize the top row by filling in element [0,0] with a cost of zero and adding insertions across the first row. For the first column we only calculate deletions.

From Figure 2.6, symbol X represents a nonterminal from the grammar and it is also associated with a row in the matrix. Also, the Greek letter α represents a terminal from the grammar. The first symbol of the input string starts at the second column. This is why we fill the first column with deletions, it allows us to calculate matches, substitutions, and insertions for the next column. From Figure 2.6, we see the match, substitution, and insertion parts (lines 10-17) are done separately from the deletions (lines 18-23). This allows all non-deletion operations to be done in one pass. We check all transitions going from the previous column to the state being processed in this column for each non-deletion operation. We do this for all the states in the automaton. On processing one of these transitions, if a symbol at the state a transition takes us to matches the test string symbol for this column, then we have a match. For this match, we copy the cost from the transitions' previous state to this state. This involves the cost matrix, where the previous state is in the previous column. Not only do we copy the cost, but if it happens to be a lower cost than what was originally at our current state, then we remove the old edit links and add a new 'match' link from the previous state to the current state. If on processing one of these transitions, no symbol matches the symbol for this column, then we check to see if the cost from the previous state is less than the current state. If it is less, we change the cost in the matrix and either replace or add to the old edit links a 'substitution' link. Also, if the same state in the previous column has a lower cost than the current state, we change

```
01  cost[0,0] = 0
02  for j = 1 to n   cost[0,j] = cost[0,j-1]+q /* insertion /
03  for i = 1 to m
04     for j = 0 to n   cost[i,j] = ∞
05  for each X ∈ N
06     for each (Y → αX) ∈ P
07           if cost[Y][0]+r <= cost[X][0] then
08              cost[X][0] = cost[Y][0]+r /* deletion */
09  for j = 1 to n
10     for each X ∈ N
11        for each (Y → αX) ∈ P
12           if a_i = α then
13              cost[X][j] = cost[Y][j-1] /* match */
14           else
15              cost[X][j] = cost[Y][j-1]+p /* substitution */
16           if cost[X][j-1]+q <= cost[X][j] then
17              cost[X][j] = cost[X][j-1]+q /* insertion */
18     copy N into Queue
19     for each X ∈ Queue
20        for each (Y → αX) ∈ P
21           if cost[Y][j]+r <= cost[X][j] then
22              cost[X][j] = cost[Y][j]+r /* deletion */
23              for each (X → βZ) ∈ P
24                 add Z to Queue
```

Figure 2.6: Modified Lyon algorithm for regular grammars - queue approach

its cost and either replace or add to the old edit links an 'insertion' link,

2.3.3 Queue Approach

The first approach to handling deletions is a queue approach. This routine has two parts (see lines 18-23 in Figure 2.6). The first part runs through a regular grammar's automaton, trying all deletions, one state at a time. If the cost changes at a state, then we add that state to the queue. Once all states have been processed linearly we can start the second part. The second part is the queue, which we use as a visitation schedule for moving the altered costs. Taking a state off the front of the queue, we examine its outgoing transitions and if any state on an outgoing transition has a cost change, then we add that state to the queue. Eventually, since we are reducing the costs and the costs can never become negative, the costs will settle. This means that the queue will eventually empty and we can start on the next column filling in non-deletion operations.

Chapter 3

Shortest Common Supersequence Approach

3.1 Introduction

With the queue approach, we have an efficient method for trying deletions. Sometimes, we will have to pass over the column more than twice in order to insure that the costs will settle. Another approach fixes the number of states we will visit when trying deletions. This approach uses a general visitation schedule, which is the shortest possible, that can cover any arrangement of costs in a column.

3.1.1 SCSS Defined

Before we define what a SCSS is, we first define the terms subsequence and supersequence. Hirschberg defines a subsequence by means of a mapping function [1]. Given two sequences X = $x_1x_2...x_m$ and Y = $y_1y_2...y_n$, where their symbols are from a finite set, we can check if there is a mapping from one sequence to the other. If we have a mapping F: X $\rightarrow$ Y where F(i) = j only if $x_i = y_j$ and F(i) = u, F(j) = v and i<j implies u<v, then X is a subsequence of Y. Which also means that Y is a supersequence of X. In short, if a sequence Y can generate a sequence X by having zero or more symbols deleted from it, then Y is a supersequence of X.

A SCSS is built from a number of sequences, n, such that the SCSS has each one of these sequences as a subsequence and there is no other sequence shorter than this SCSS that contains all n of these sequences as subsequences. To generate a SCSS from a regular grammar, we construct it out of sequences derived from that regular grammar's automaton.

3.1.2 Prefix Strings

A particular kind of sequence that we can derive from a finite state automaton are prefixes [5]. A simple definition of what is a prefix is: for any particular state in the automaton, a prefix to that state is a sequence of two or more states that form a directed cycle-free path ending at that state. For example, Figure 3.1 shows a

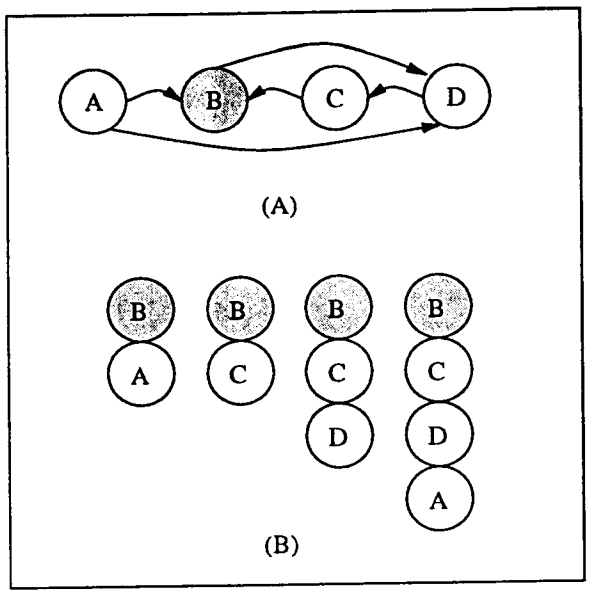

Figure 3.1: Prefix string example

finite state automaton with four states. Selecting state 'B', we find only 4 prefixes for it. These 4 prefixes are listed in Figure 3.1. For every state in the automaton, we can get a list of all its prefixes. We can get rid of many prefixes that happen to be subsequences of other longer prefixes. From here, we can build the SCSS from the remaining prefixes.

DAG Construction

A DAG is a directed graph with no cycles. We use the DAG to represent all prefixes to a particular state [5]. This particular state is the root of the DAG and all prefixes hang off from it. Figure 3.2 shows the DAGs for the reduced set of prefixes. Part A of Figure 3.2 is the automaton the prefixes are derived from. The following set contains all prefixes of size two or greater for this automaton:

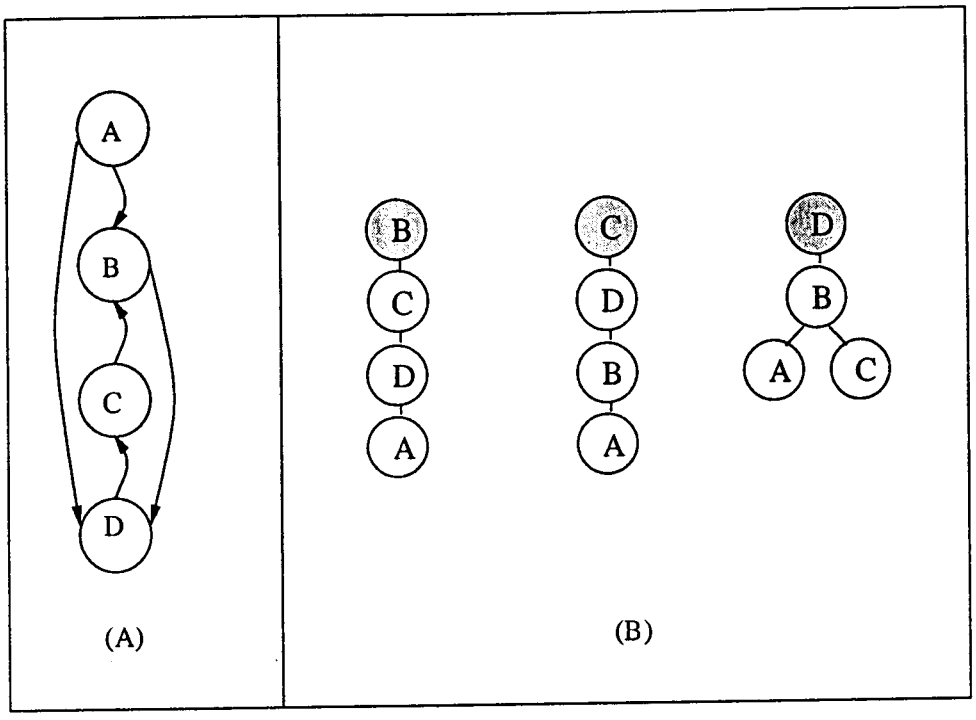

Figure 3.2: Example automaton and its associated prefix DAGs

{ AB, CB, DCB, ADCB, DC, ADC, BDC, ABDC, AD, BD, CBD } . We can reduce this set to: { ADCB, ABDC, ABD, CBD } . Those prefixes left out were subsequences of one of the prefixes in this reduced set. State A does not have a DAG since there are no prefixes of length two or more for it. We read the prefixes in the DAG from the leaves on up to the root.

Combining DAGS

Looking at Figure 3.3, we see that we can combine these DAGs into one large DAG. Notice how in part B of Figure 3.3 that all the prefixes are preserved in the resulting DAG. From this DAG, we can create a visitation schedule [5]. We build

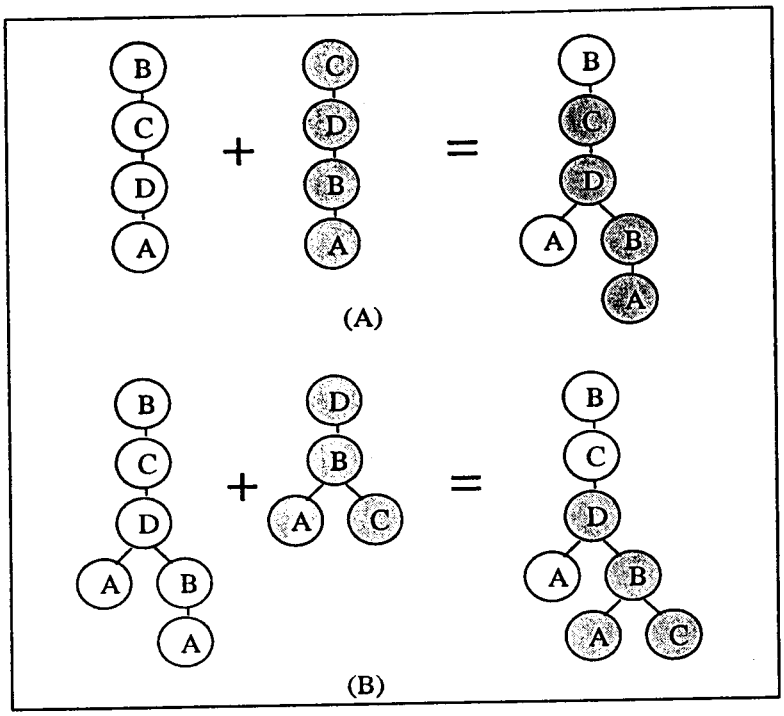

Figure 3.3: Combining DAGs into one DAG

this visitation schedule by adding those lowest level nodes in the DAG to our schedule. Nodes at higher levels in the final DAG are ones that can be effected by lower level nodes. Nodes on the same level do not directly effect each other. Those nodes on the same level can be read in any order and placed on the right hand side of our schedule. We continue doing this and going up a level until we reach the root. The root happens to be the last node added to the schedule. Notice how node A in part B of Figure 3.3 falls into two leaves. Since the next to the lowest level contains nodes A and B, by ordering these with node A before B in the schedule we can combine the two node A leaves. One schedule from this DAG is, "CABDCB". In this case, this schedule happens to be one of many shortest

common supersequences for the automaton in Figure 3.2.

3.1.3 NP-Completeness and Approximations

Finding the SCSS for n sequences is NP-hard even when the length of each input sequence is two and total number of times each letter shows up in all the sequences is three [8]. In general, the SCSS problem is NP-complete when the alphabet size is 5 or more [10]. Even finding an approximation is difficult. If P $\neq$ NP, then a "polynomial time linear approximation algorithm" does not exist for the SCSS problem [8]. Some of the approximations methods used are the Tournament and Greedy algorithms [7].

The Tournament algorithm first pairs up the n sequences and finds a SCSS for each pair. Once done, all those SCSSs are paired up and a SCSS is found for each new pair. This process continues until we have one resulting CSS for all n sequences. For this algorithm to work we need to have $n = 2^i$ sequences, where i is a positive integer.

The Greedy algorithm first finds a SCSS for two out of the n sequences. Next, this algorithm takes another of the n sequences and with our current SCSS it generates a new SCSS. This process is continued until all n sequences have been used. This results in a CSS for all n sequences.

3.2 SCSS Brute-Force Approach with Pruning

There are many ways to go about finding the SCSS. One simple method finds a short CSS. This alone can take some time to find, if the intent is getting close in length to a SCSS. Once we have a short CSS, we can use it to ignore anything not smaller than it when searching for the SCSS. We will find the SCSS if given enough time and resources. Also, we want to emphasize why this can be used as a visitation schedule.

3.2.1 A Candidate Common Supersequence

A Greedy algorithm is used to find an approximation for a SCSS. Once we have an approximation, or CSS, it can be used to find a SCSS. Figure 3.4 explains the algorithm used. From part (*i*) in Figure 3.4, after finding all the unique long prefixes in the automaton, where no prefix is a subsequence of a longer prefix, we can now try to build the candidate CSS. For part (*ii*), we can find the shortest common supersequence of two prefixes (done in polynomial time). This can be done in a matrix using dynamic programming. Now taking the newly generated SCSS, we use this and the next prefix to generate a new SCSS. We do this for all prefixes and we will come up with a common supersequence for those prefixes. This gives us a candidate CSS that we can use to limit our search for a real SCSS. For part (*iii*) in Figure 3.4, we can use the candidate CSS for pruning in an

(*i*)	Derive prefix strings for all nonterminals in our grammar. Remove all prefixes that are subsequences of other prefixes.
(*ii*)	Align all prefix strings sequentially. Keep only the SCSS at each alignment. Align this SCSS with the next prefix string. The candidate is equal to the CSS that is a supersequence to all the prefixes.
(*iii*)	Align all prefix strings sequentially. Keep all CSSs at each alignment. Align each CSS with the next prefix string. The candidate string is used to remove those CSS's that are the same size or longer. Also, if a CSS is a supersequence to all the prefixes and is shorter than the candidate, then replace the candidate with this shorter CSS.

Figure 3.4: Shortest common supersequence algorithm

exhaustive search for a shortest CSS or it could be used as a visitation schedule.

3.2.2 Exhaustive Search with Pruning

Since we do not really know if the CSS is the shortest, we do an exhaustive search for the SCSS and use the length of the CSS to prune our search space. Figure 3.5 shows the role of the CSS string. If any sequence that we build from the prefixes is longer than or the same size as the CSS, then we stop searching on that sequence (prune that branch in the tree).

From the figure, each level of the tree, from the top down, represents the combination of a certain number of prefixes. At the root, there is only one prefix. At the next level down, all children from the root are the result of two prefixes and all their combinations into supersequences. If we have N prefixes, then there will be N levels of the tree. On each level down, there is generally an exponential growth in the number of nodes at this level. To make the role of the candidate

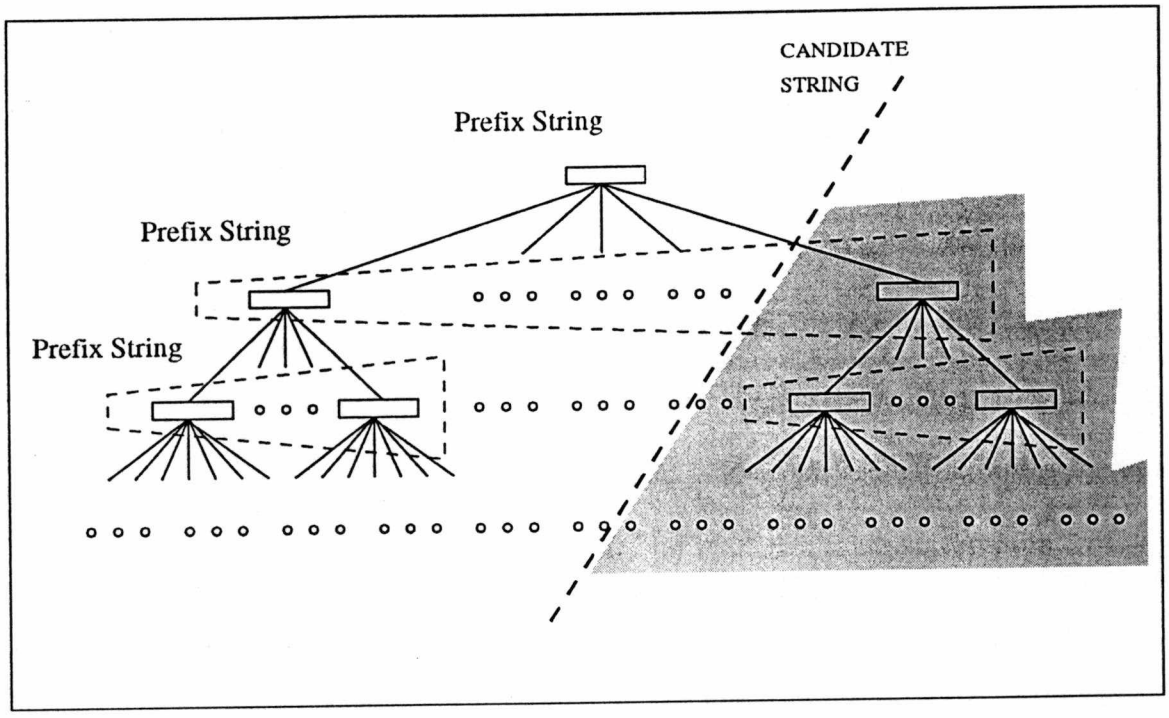

Figure 3.5: Role of the candidate string in the search space

string more effective, the combinations of prefixes on a level below a node are ordered by length from the shortest on the left to the longest on the right. When doing a depth-first search, this helps check shorter CSSs first.

A leaf node in the tree, where all prefixes have been combined, represents a CSS for those prefixes. On reaching one of these leaf nodes that happen to have a shorter CSS than our candidate string, we replace the candidate with the shorter CSS. This allows us to prune even more of the tree. Eventually, when all the leaf nodes that have CSS's smaller than the candidate have been examined, we will then have a SCSS for these prefixes. If no shorter string than our candidate was found, then the candidate is the SCSS. In other words, the last candidate string

selected will be our SCSS.

3.2.3 Using the SCSS as a Visitation Schedule

A SCSS can be used as a visitation schedule [5]. We use the SCSS as a visitation schedule for our deletion routine in the matrix version of the Lyon parser for regular grammars. That is, we can step through each state in the schedule and check all outgoing transitions from that state for deletions. This works because any cycle free path in a grammar's automaton, will be a subsequence of the SCSS and thus will be covered.

We want to find all the cycle free paths for a particular state, since we do not need paths that have cycles. These cycle free paths happen to be prefixes for that state. A schedule that contains, as subsequences, all the prefixes to a state will be able to find the lowest cost for that state [5]. Earlier, we have seen that we can combine each states' schedule into one. A sequence of states that have had their costs changed due to deletion operations may effect states that we have previously processed. This is not a problem when using the SCSS, since it will have all the information from the prefixes and will know which states need to be checked again (those states common to more than one prefix string).

To arrive with the least error cost for aligning a string into one of the strings in $L(G)$, we do not delete substrings more than once. We will use the automaton from Figure 3.6 as an example on repeating substrings. Assume that all edit

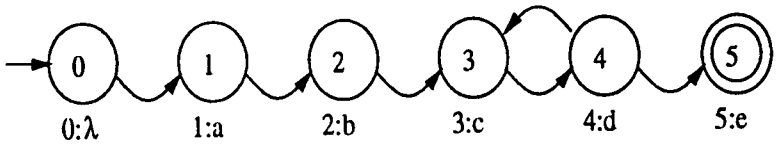

Figure 3.6: Example automaton with one repeating substring

costs are one, except for matches. With the test string "abe" this string should align with a string from this automaton with a cost of two. This string from the automaton is "abcde". If we deleted a substring more than once then we would also be able to align the test string to strings generated by this automaton that have higher costs. Another string from the automaton "abcdcde" would have an alignment cost of four. This is why using prefixes to build a visitation schedule is important, since they do not have repeating substrings.

If two prefixes have no states in common, then they are not dependent on each other. If any state or states are in common, then there is a dependency at that state or states. The supersequence of these prefixes will filter the lowest cost to those common states. This lowest cost is then carried on to the rest of the states in the supersequence. By examining all the prefixes for a state for movement of deletion costs, this will ensure that this state settles on the lowest cost. The same can be done for all states in the automaton. This inherently tells us that in scanning the schedule from left to right, we never increase costs, we can only lower them.

Harris used a constructive proof using DAGs for showing that a CSS can be a

visitation schedule [5]. When converted into schedules, those prefix DAGS for the individual states guarantee that those states would settle if we processed each one individually. We can combine each state's DAG into one large DAG. The resulting schedule from this DAG will have enough information in the state sequences such that the deletion costs will settle. This large DAG can be constructed where, with some manipulation, we can generate a SCSS from it. The combining of all the individual DAGs for the states into one large DAG requires many combinations to be tried in order to find the SCSS. The method for deriving the SCSS in this paper used a brute force approach with pruning.

3.3 Complexity

Depending on the number of prefixes and their length (and somewhat on how the prefixes are ordered), the search tree grows exponentially. For some eight state automata with many cycles, even machines with 128 megabytes of RAM are not able to hold all parts of the search tree. We do order the sequences generated from the prefix and current common subsequence so that we work on the shorter sequences first. One thing that can save us some space and time is if the next prefix to combine with the current CSS happens to be a subsequence of the current CSS, then we can skip to the next prefix. If we have prefixes that don't fallout, that is, are not subsequences of previous combinations, then the search space

grows for those prefixes. This implies that a different ordering of the prefixes may make the search tree smaller. Running different orderings on many machines may give a faster result (since one may finish before any of the others do).

3.3.1 Worst Case

When combining the first prefix with the second prefix, we get a worst case performance that actually corresponds to Pascal's Triangle (Figure 3.7). For instance, if the first prefix and second prefix are four symbols long and do not have any states in common, then 70 combinations would be generated. Each one of these would be a CSS for the two prefix strings and would be eight symbols long. Combining one of these with a third prefix that is four in length, with no states in common, we get 330 combinations for each of the 70 CSS's. Picking a constant, C, we have at least C^N combinations for N prefixes in the worst case. This is definitely exponential in nature. Even worse, the value of C is a function of N and it becomes larger with increasing N.

The search tree is effected by many factors. How much is in common between two prefixes, their length, the number of prefixes, and whether we have prefixes that are subsequences of previous combinations (which can be effected by the order in which we combine prefixes) all have some influence in the size of the search tree. The only thing that is guaranteed to reduce the size of the search tree is if we can prune off long combinations using a short candidate string. If the

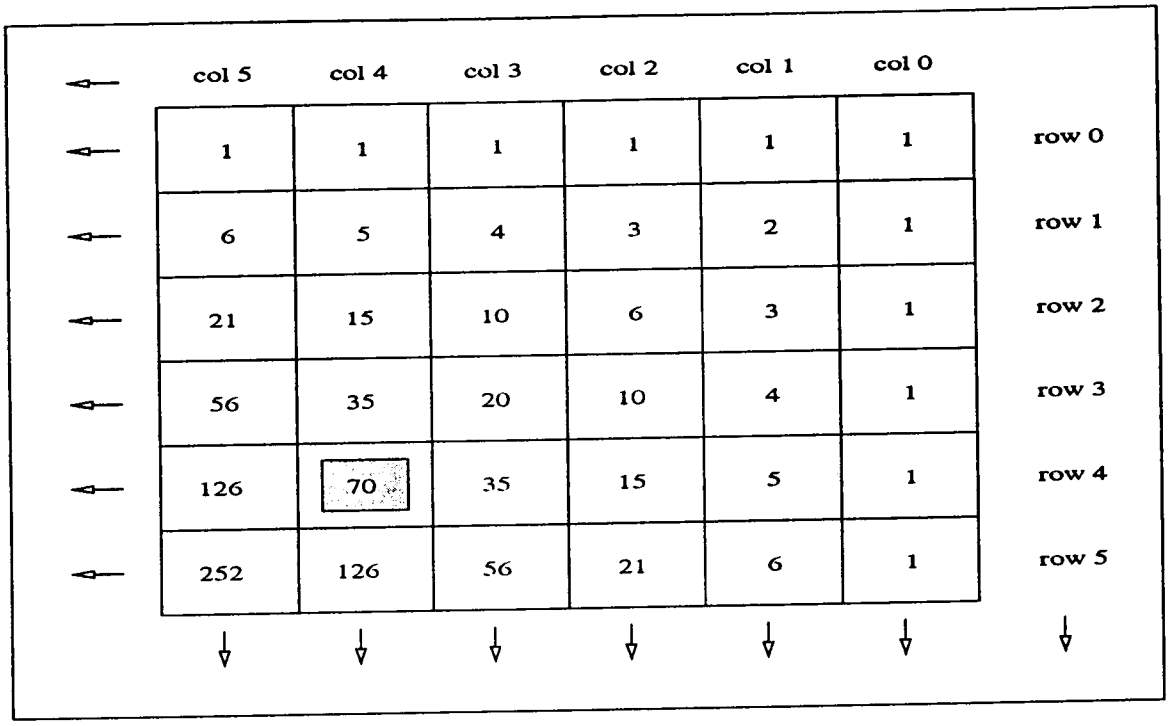

Figure 3.7: Worst case performance when combining two sequences

candidate string is nearly as long as all the prefixes concatenated, then this too is a bad approach to reducing the number of combinations computed. Overall, we can generalize the complexity of finding the SCSS, by saying it is $O(C^N)$ where C is function of N. Given that the prefixes are from an automaton, there will be some states in common among the prefixes. Thus C will be less than the worst case derived from Pascals Triangle than would be generated if all the prefixes had no states in common with any other prefix.

3.4 SCSS Deletion Approach

For the SCSS approach to handle deletions, we use the SCSS as a visitation schedule for a regular grammar's automaton (Figure 3.8). The only difference between the queue approach and this approach is the deletion routine (lines 18-21 in Figure 3.8). Running through each one of the states in the SCSS, we check to see if we can push a deletion cost down its transitions. After checking all outgoing transitions for a state, we then try the next state in the SCSS. The costs in a column are guaranteed to settle once we run out of states in the SCSS.

Which is the better approach? That depends on the grammar and test string. On certain grammars, if the string matches one of the strings generated by the grammar, it is quite possible that the queue approach has an advantage. The reason why there is no clear winner will be explained in the last chapter. The SCSS is the shortest visitation schedule that covers any possible deletion path in a column. When there are a large number of deletions to move in a column, the SCSS approach tends to win out overall, since the queue can wind up becoming longer than the length of the SCSS. Some free-form automata with many cycles show this to be the case. The queue approach is unbounded and there are some automata where deletions can make many passes over a column before settling.

```
01  cost[0,0] = 0
02  for j = 1 to n   cost[0,j] = cost[0,j-1]+q /* insertion /
03  for i = 1 to m
04     for j = 0 to n   cost[i,j] = ∞
05  for each X ∈ N /* N is a topological ordering of T */
06     for each (Y → αX) ∈ P
07           if cost[Y][0]+r <= cost[X][0] then
08              cost[X][0] = cost[Y][0]+r /* deletion */
09  for j = 1 to n
10     for each X ∈ N /* N is a topological ordering of T */
11        for each (Y → αX) ∈ P
12           if a_i = α then
13              cost[X][j] = cost[Y][j-1] /* match */
14           else
15              cost[X][j] = cost[Y][j-1]+p /* substitution */
16           if cost[X][j-1]+q <= cost[X][j] then
17              cost[X][j] = cost[X][j-1]+q /* insertion */
18     for each X ∈ SCSS
19        for each (Y → αX) ∈ P
20           if cost[Y][j]+r <= cost[X][j] then
21              cost[X][j] = cost[Y][j]+r /* deletion */
```

Figure 3.8: Modified Lyon algorithm for regular grammars - SCSS approach

Chapter 4

Experimental Work

4.1 Three Types of Grammars

Three major forms of regular grammars are studied: feed forward (no cycles), regular expression (no crossed transitions in its associated automaton), and free-form regular grammars (massive pumping with no restrictions). Why were these chosen? The deletion routines are apt to go into a loop when there are cycles in an automaton. This explains why there are at least two groups of grammars to study. For regular expressions we need, at most, two passes through a list/column to correctly calculate deletions [11]. The free-form grammars can require more than two passes over a column to have deletions calculated correctly. Figure 4.1 show some feed forward and regular expression automata. Notice that automaton A is the same as B, only the labels in the states are changed. Automaton C has

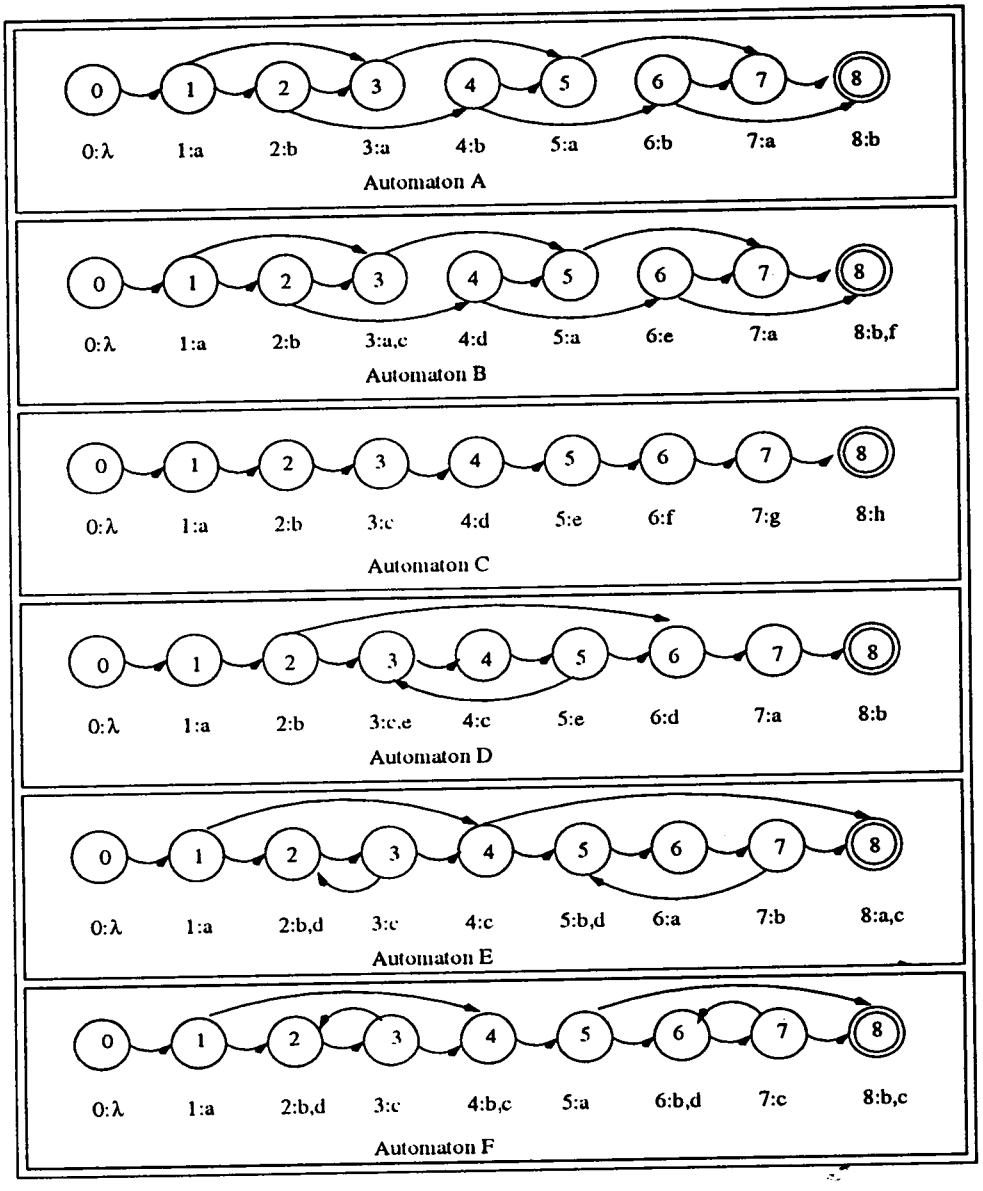

Figure 4.1: Feed forward and regular expression automata

a straight forward path from the start state to the ending state. Automata D, E, and F represent simple regular expressions. Figure 4.2 covers those automata that are neither feed forward or represent regular expressions. Automata G, H, I and J were chosen because they tend to repeat some transitions more than 2 times in the queue approach. This is only on certain test strings entered. It is important to note that the edit costs used with the free-form grammars were 2 for a substitution and 1 for insertion/deletion, where matches cost zero. We used a Levenshtein edit cost for the other grammars. Finally, automaton K, has a very interesting behavior, it took about one hour to find its SCSS on a SPARC 10. Its candidate string was 17 symbols long and took only about one second to find. It found a shorter one, 16 characters long, and eventually decided that it was the shortest common supersequence.

4.2 Number of Deletions

On various grammars examined, the timings were too small to show significant differences over the two deletion approachs. A count of the number of deletions attempted, along with their timings, is a better indicator of performance. The queue approach will run quickly for columns with very few or no deletions. The SCSS approach will run quickly if the length of the SCSS is the same or just slightly greater than the number of states in the grammar.

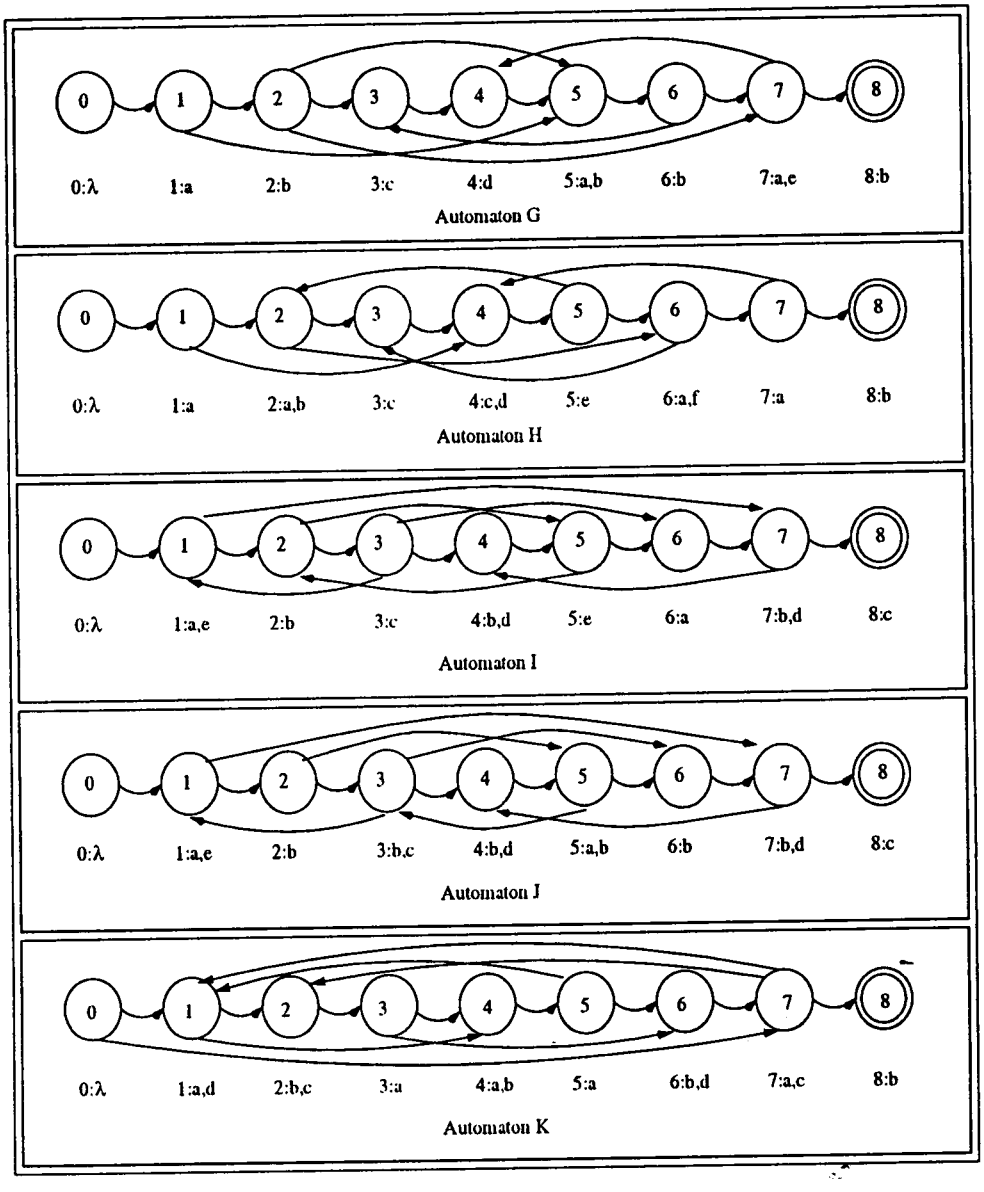

Figure 4.2: Free-form automata

Comparing both Lyon regular grammar parsers (list approach) to the queue approach for deletions, the queue approach will usually have fewer deletions attempted than the Lyon list versions. Both Lyon regular grammar list versions do not specifically know ahead of time which states to check for deletions, unlike the queue approach which only adds to the queue those states whose costs were changed because of deletion operations.

We compare the number of deletions attempted by the SCSS, queue, and two Lyon list deletion routines. List version A does a table lookup in a list on a union operation, while list version B searches in a list on a union operation. In the regular expression and free-form automata, it looks like the queue is more efficient. Keep in mind that the execution times usually favor the SCSS routine when it's visitation schedule is just a little bit longer than the number of states in the grammar.

From Table 4.1, we see that the SCSS has fewer deletions attempted than the other approaches. Here the queue approach sometimes adds states to the queue when it has not stepped through those states in the first pass. Both list approaches will always calculate the same number of deletions since their deletion routines operate the same. Feed forward automata have SCSSs equal in length to the number of nonterminals in their associated grammar. Since many of the prefix strings will fall into longer ones. If there is a straight path through all the nodes, this would be an indicator for a short SCSS. Automata A and B have

Table 4.1: Number of deletions attempted for regular expression automata

Automaton	String	SCSS	Queue	List A	List B
Automaton D	abda	48	50	108	108
	abecec	72	73	144	144
	abececce	96	96	192	192
	abececceda	120	115	252	252
Automaton E	abcd	64	55	128	128
	acbabd	96	83	208	208
	acbabdab	128	108	272	272
	abcdccbabd	160	130	320	320
Automaton F	abcd	56	55	120	120
	acabcd	84	81	180	180
	abcdcbac	112	105	255	255
	abcbabcdcb	140	131	315	315

the same shortest common supersequence. The only difference between the two is that automaton B has a larger set of terminals. When there are only a few terminals in a set, the queue approach tends to have fewer deletions attempted.

Looking at Table 4.2, we see that not all strings generate fewer deletions for the SCSS routine. With longer strings it appears that the queue approach does well. Again, more operations are required in handling a deletion in the queue approach than the SCSS approach. There are some columns in the matrix which executed faster in the queue approach than the corresponding column in the SCSS method. These times are rather close in comparison to actually have any real significance. This behavior also showed up in some of the feed forward grammars

Table 4.2: Number of deletions attempted for feed forward automata

Automaton	String	SCSS	Queue	List A	List B
Automaton A	abaa	48	50	135	135
	aaaa	48	50	135	135
	abbaab	72	74	180	180
	abbbab	72	75	180	180
Automaton B	abca	48	56	135	135
	aaaa	48	50	135	135
	abdaab	72	80	165	165
	abdeab	72	81	150	150
Automaton C	abcd	32	46	56	56
	abde	32	44	48	48
	abdefg	48	59	72	72
	abcdeh	48	62	72	72

too. Looking at automaton E in the table, we see that its SCSS is the longest visitation schedule of the three here. This is inferred by dividing the number of deletions by the number of symbols in a test string. Thus we have a visitation schedule of length 16, which should be the maximum number of states visited for a regular expression grammar with eight states (not counting the final state).

Finally, we have interesting results for our free-form automata in Table 4.3. Although the SCSS approach tries more deletions in these cases, automata G, H, I, and J all executed faster with the SCSS deletion routine than with the queue routine. Those test strings from Table 4.3 emphasize looping on some transitions more than twice in the queue approach for automata G, H, I, and J. With larger free-form automata, this kind of behavior can get even worse for the queue approach.

For automaton K, we have about twice as many deletions attempted for the SCSS routine than the queue approach. Once we reach this point the queue approach will usually execute faster than the SCSS approach. Since automaton K has 13 transitions, we see that very few deletions were done in the queue approach for a test string of 4 symbols (since 52 deletions is the lowest number possible). Every transition has at least one deletion operation attempted. When the queue routine does not push any deletion costs onto other states, then we can expect it to operate efficiently. Also, automaton K has a shortest common supersequence that is 16 symbols in length which is twice the number of states in the automaton

Table 4.3: Number of deletions attempted for free-form automata

Automaton	String	SCSS	Queue	List A	List B
Automaton G	aab	54	56	152	152
	adab	72	67	152	152
	acadb	90	77	152	152
Automaton H	abc	63	52	140	140
	bcd	63	52	160	160
	adeafa	126	95	280	280
Automaton I	dcbe	116	68	192	192
	bcab	116	70	216	216
	cdebcab	203	113	360	360
Automaton J	bea	87	56	175	175
	dcbe	116	66	200	200
	bcab	116	70	225	225
Automaton K	abab	112	57	216	216
	aaaaba	168	81	312	- 312
	aaaabada	224	111	408	408

(not counting the final state).

4.3 Timings

Deletion counts alone are not sufficient to tell the whole story. Table 4.4 shows that even when fewer deletions were encountered with the queue approach, it still had to take time to generate and add a state to the queue. The shortest common supersequence approach is just a schedule and does not have to generate or add states. Still, we see that automaton K from Table 4.3 makes the queue approach look better. Those strings did not make deletions loop more than twice in any state in the automaton. The other free-form automata are good representatives of the kind of grammars that our SCSS is best suited for. Automata that take more than two passes in order to make sure the deletions settle will benefit from the visitation schedule as opposed to the queue.

4.4 Conclusion

Over all, using a SCSS as a visitation schedule will operate efficiently for many regular grammars. In the above cases studied, we did not optimize the use of the SCSS. By itself, the SCSS does not have any prior knowledge of the costs distributed across a regular grammar's automaton. In this instance, it is the shortest general visitation schedule where the costs in a grammar's automaton

Table 4.4: Timings of deletion routines for free-form automata

Automaton	String	SCSS	Queue	List A	List B
Automaton G	aab	0.664m sec	0.886m sec	1.275m sec	1.503m sec
	adab	0.720	0.957	1.467	1.582
	acadb	0.788	0.987	1.661	1.795
Automaton H	abc	0.592m sec	0.726m sec	1.293m sec	1.382m sec
	bcd	0.571	0.684	1.619	1.741
	adeafa	1.376	1.434	2.776	2.952
Automaton I	dcbe	0.778m sec	0.819m sec	2.282m sec	2.608m sec
	bcab	0.866	0.934	1.702	1.934
	cdebcab	1.369	1.476	3.394	3.864
Automaton J	bea	0.689m sec	0.773m sec	1.300m sec	1.649m sec
	dcbe	0.780	0.850	2.134	2.604
	bcab	0.879	0.918	1.736	2.157
Automaton K	abab	0.719m sec	0.741m sec	1.887m sec	2.128m sec
	aaaaba	1.468	1.093	2.572	2.878
	aaaabada	1.479	1.462	3.621	4.130

will always settle. Knowing how the costs are distributed ahead of time, we can come up with shorter schedules. Generating the SCSS can be a problem, since we know it to be an NP-complete problem. Even finding close approximations is time consuming. The candidate CSS that was used to prune the SCSS search space took a lot less time to generate. It too can be used as a visitation schedule. Since it takes a long time and many resources to generate the SCSS, it is not recommended for large free-form grammars. Our candidate CSS may not be too impractical to use. Although, it can be a lot longer than the SCSS, it does not take nearly the same amount of time to find. The execution time for parsing a string is very short, implying that larger visitation schedules would not be too costly. If we were going to use the same grammar for constructing many error-correcting parses of strings, we may find that having the SCSS practical. It is a one time cost in calculating this shortest general visitation schedule. In other words, if you can find it in a reasonable amount of time, you will always have it for that grammar.

Bibliography

Bibliography

[1] In D. Sankoff and J. B. Kruskal, editors, *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, Reading MA, 1983. Addison-Wesley.

[2] F. Casacuberta and E. Vidal. A parsing algorithm for weighted grammars and substring recognition. In G. Ferraté et al, editor, *Syntactic and Structural Pattern Recognition*, Berlin, 1988. Springer-Verlag.

[3] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[4] J. Gregor and R. S. Harris. String matching with left-to-right networks. *Pattern Recognition Letters*, 16(1995):213–218, February 1995.

[5] R. S. Harris. Approximate Matching Strings to Automata. Unpublished memo, Dept. of Computer Science, University of Tennessee, Knoxville, January 1994.

[6] J. E. Hopcroft and J. D. Ullman. *Introduction to Automota Theory, Languages, and Computation.* Addison-Wesley, Reading MA, first edition, 1979.

[7] R. W. Irving and C. B. Fraser. On the worst-case behavior of some approximation algorithms for the shortest common supersequence of k strings. In A. Apostolico et al, editor, *Combinatorial Pattern Matching*, Berlin, 1993. Springer-Verlag.

[8] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, October 1995.

[9] G. Lyon. Syntax-directed least-errors analysis for context free languages: a practical approach. *Communications of the ACM*, 17(1):3–14, January 1974.

[10] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, April 1978.

[11] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[12] E. Tanaka. Parsing for string grammars. In H. Bunke and A. Sanfeliu, editors, *Syntactic and Structural Pattern Recognition: Theory and Applications*, Singapore, 1990. World Scientific.

[13] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974.

Vita

Eric Wright Richardson was born in Chicopee, Massachusetts in 1966. He lived on three different continents and went to public schools in Virginia, Germany, Colorado, Maine, and Florida. He graduated from Leto Comprehensive High School in 1985. He received his Bachelor of Science in Computer Science and a Bachelor of Arts in Mathematics at Lyndon State College in 1993. In that same year, he entered the graduate program in Computer Science at the University of Tennessee, Knoxville. He received his Master of Science in Computer Science at the University of Tennessee, Knoxville in 1996. He is presently working as a programmer in Atlanta, Georgia.