Finding Minimum Cost Derivation Trees for Regular Tree Grammars

Maya Madhavan Priti Shankar

IISc-CSA-TR-1998-7
http://archive.csa.iisc.ernet.in/TR/1998/7/

Computer Science and Automation
Indian Institute of Science, India

September 1998

Finding minimum cost derivation trees for regular tree grammars

Maya Madhavan
Priti Shankar
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012
India

September 29, 1998

Abstract

Regular tree grammars have been used extensively as specification mechanisms for retargetable code generation. Conventional code generation tools construct a finite state tree pattern matching automaton from a regular tree grammar specification. This automaton generates optimal code by traversing an intermediate code tree and constructing a mimimum cost derivation tree for it. Cost computations can be carried out either at the grammar preprocessing phase, or during the matching phase, the former choice usually resulting in larger tables. We propose a construction that augments the precomputation step of a regular tree pattern matching algorithm to include cost analysis. The matching device generated is a pushdown automaton in contrast with the conventionally generated tree pattern matching automaton. The technique can be viewed as an extension of the well known Graham-Glanville technique for table driven code generation, as it is essentially based on LR parsing techniques, augmented to handle ambiguous string grammars derived from regular tree grammars. Our technique can handle a larger class of cost augmented regular tree grammars than can be preprocessed by conventional methods, and has been tested on some input problem instances representing instruction sets of two processors.

1 Introduction

The use of regular tree grammars and bottom up rewrite systems for the description of machine instructions for use in code generation tools is well known [4] [2] [7] [11] [12] [13]. The problem addressed in this paper can be stated as follows: *Given a regular tree grammar whose productions are augmented with costs, and an input subject tree, compute a minimum cost derivation tree for the input tree.* In a typical application, the subject tree is an intermediate code representation for a source program undergoing compilation; the minimum cost derivation tree yields a cheapest assembly level code sequence for the tree.

The class of languages generated by regular tree grammars includes those accepted by conventional tree automata, or defined by tree grammars [8]. For the latter class, once the set of operators and terminals is specified, the set of trees in the language comprises all those that can be constructed using these operators and terminals. If one wishes to restrict the set of trees that are to be included in the language, then one way of doing so, is by using a regular tree grammar; the set of trees in the language is then restricted to those that can be generated from the start symbol of the grammar. The problem of regular tree pattern matching is generally divided into the following two phases, originally suggested by [8] for the problem of ordinary tree pattern matching:

1. A preprocessing phase, which constructs tables that encode the automaton from a description of the input patterns in the form of a regular tree grammar
2. A matching phase, which traverses the input subject tree and reports matches of patterns at the appropriate nodes.

The problem of optimal regular tree pattern matching is somewhat more complex. In general, a subject tree may be tiled in several ways. (This is equivalent to saying that there may be several derivation trees for the subject tree, or that the grammar is ambiguous). In finding the minimum cost derivation tree, the processing of cost information may be carried out either during the preprocessing phase or during the matching phase. The former choice has the advantage that cost analysis is moved from matching time to preprocessing time, the latter usually being considered a one time operation. The constructions described in [4] [2] and [12] all construct bottom up, finite state tree pattern matching automata for the purpose of optimal regular tree pattern matching.

We propose an alternate approach to the problem of optimal regular tree pattern matching; the matching device is a pushdown automaton. The idea of table driven code generation had been suggested by Graham and Glanville[6], who constructed $LR(0)$ parsing tables from linearized tree patterns, and performed matching on a linearized version of the intermediate code tree,using a device that was essentially a pushdown automaton. However, their technique cannot be applied to regular tree pattern matching in general, as it does not carry forward all parsing choices until an optimal decision can be made. Our approach can be viewed as an extension of the $LR(0)$ technique to handle input grammars, that are, in general, highly ambiguous. However, restrictions on the form of the grammar permit the design of a linear time parsing technique. We have incorporated a variation of the technique proposed in [2] in an algorithm proposed in [15] [10] to precompute costs into the states of the deterministic finite state machine that controls the parsing process. Thus cost analysis can be performed at pattern preprocessing time. There are(pathological) examples where the finite state tree pattern matching automaton constructed using conventional techniques is exponential in the size of the pushdown automaton constructed for the same problem instance.

For the problem of optimal regular tree pattern matching, it is known that conventional algorithms do not terminate if cost differences for derivation trees for the same subject tree are unbounded. A sufficient condition to ensure termination is given in [4]. The technique proposed here is also subject to similar limitations; however it can handle a larger class of input grammars than can be handled by previous techniques. Section 2 provides some background; section 3 presents the new algorithm and a condition for termination. Finally section 4 concludes the paper with a discussion on the efficiency of this technique and a presentation of some empirical results.

2 Background

Let A be a finite alphabet consisting of a set of operators OP and a set of terminals T. Each operator *op* in OP is associated with an *arity*, $arity(op)$. Elements of T have arity 0. The set $TREES(A)$ consists of all trees with internal nodes labeled with elements of OP, and leaves with labels from T. The number of children of a node labeled *op* is $arity(op)$. Special symbols called *wildcards* are assumed to have arity 0. If N is a set of wildcards, the set $TREES(A \cup N)$ is the set of all trees with wildcards also allowed as labels of leaves.

We begin with a few definitions drawn from [2].
Definition 2.1 A *regular tree grammar* G is a four tuple (N, A, P, S) where
1. N is a finite set of *nonterminal* symbols
2. $A = T \cup OP$ is a *ranked alphabet*, with the ranking function denoted by arity. T is the set of *terminal* symbols and OP is the set of *operators*.
3. P is a finite set of *production rules* of the form $X \to t$, where $X \in N$ and t is an encoding of a

tree in $TREES(A \cup N)$.
4. S is the *start symbol* of the grammar.

A *tree pattern* is thus represented by the righthand side of a production of P in the grammar above. A production of P is called a *chain rule*, if it is of the form $A \to B$, where both A and B are nonterminals. For purposes of the discussion here we will be dealing with grammars in *normal form* defined below:

Definition 2.2 A production is said to be in normal form if it is in one of the three forms below.
1. $A \to opB_1B_2 \dots B_k$ where $A, B_i, i = 1, 2, \dots k$ are all non terminals, and op has arity k.
2. $A \to B$, where A and B are non terminals.
3. $B \to b$, where b is a terminal.
A grammar is in normal form if all its productions are in normal form.

Any regular tree grammar can be put into normal form by the introduction of extra nonterminals. Below is an example of a regular tree grammar in normal form. For the time being, we ignore the costs associated with the productions.
Example 2.1
$G = (\{V, B, G\}, A, P, V)$ where, $A = \{a, b\}$ and the arities are 2 and 0 respectively. The productions P, augmented with cost are as follows:

$V \to aVB$	[0]
$V \to aGV$	[1]
$V \to G$	[1]
$G \to B$	[1]
$V \to b$	[7]
$B \to b$	[4]

Derivation sequences are defined in the usual way. However, we note that the objects being derived are trees. Two derivation trees for the subject tree $a(a(b, b), b)$ in the language generated by the regular tree grammar of example 2.1 above, are displayed in Figure 1.

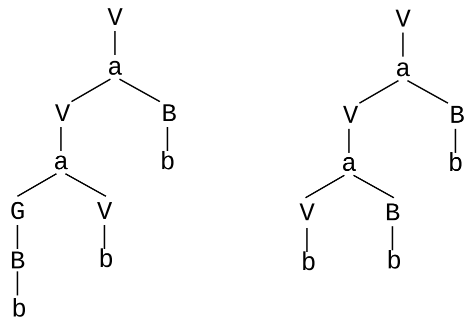

Figure 1: Two derivation trees for the subject tree $a(a(b, b), b))$ in the grammar of example 2.1.

A tree pattern is said to *match* at a node in the subject tree, if a production with righthand side corresponding to the tree pattern is used at that node in the derivation tree. Each derivation tree thus defines a set of matches, (a *set* because there may be chain rules that match) at each node in the subject tree. (Actually, as we shall see later, conventional regular tree pattern matching algorithms compute, in general, a superset of the tree patterns that actually match). The regular tree pattern matching problem may thus be phrased as :*Given a regular tree grammar G and a subject tree t, find a representation of all derivation trees for t in G.*

The regular tree pattern matching problem for a grammar G can be converted into a string parsing problem for a context free grammar by constructing a new grammar G' from G, by converting righthand sides of all productions of G into postorder listings of the corresponding tree patterns. Any listing will work; however, a postorder listing is appropriate for a bottom up algorithm. Since each operator has a fixed arity, the postorder listing uniquely defines the tree. Let $post(t)$ denote the postorder listing of a tree t. We note that the grammar G' is in general ambiguous. It is shown in [15] that finding all derivation trees in G corresponds to finding all derivations in G'.
We assume that the reader is familiar with the notions of right sentential forms, handles, viable prefixes, and $LR(0)$ items being valid for viable prefixes. Definitions may be found in [9].
The three key ideas used in the algorithm in [15] are the following:
1. Refinement of the notion of a pattern matching at a particular node to that of a *pattern matching in a left context* at that node.
2. Equivalence of the problems of detecting pattern matches in left contexts in regular tree grammars, and obtaining rightmost derivation sequences in reversein the corresponding context free grammars.
3. The observation that all viable prefixes induced by any input string are of the same length.(By a viable prefix *induced* by an input string we mean the string representing the stack contents after consuming the input string using a shift reduce parsing strategy. The normal form of the grammar is essential for the last property).

Definitions and theorems formalizing these ideas from [15] are presented below. By an X-derivation tree we mean a subtree of a derivation tree with root labeled X.:

Definition 2.4 Let $G = (N, T, P, S)$ be a regular tree grammar in normal form, and t be a subject tree. Then pattern β represented by production $X \to \beta$ matches at node j in left context $\alpha, \alpha \in N^*$ if
1. β matches at node j or equivalently,

$$X \Longrightarrow \beta \Longrightarrow^* t'$$

where t' is the subtree rooted at j.
2. If α is not ϵ, then the sequence of maximal complete subtrees of t to the left of j, listed from left to right is $t_1, t_2, \ldots t_k$, with t_i having an X_i-derivation tree,$1 \leq i \leq k$, where $\alpha = X_1 X_2 \ldots X_k$.
3. The string $X_1 X_2 \ldots X_k X$ is a prefix of the postorder listing of some tree in $TREES(A \cup N)$ with an $S-$derivation.

Example 2.2 In the derivation tree on the left in figure 1, pattern represented by production $B \longrightarrow b$ matches in left contexts ϵ and V. In the derivation tree on the right both instances match in left context V.

Theorem 2.1 Let $G = (N, T, P, S)$ be a regular tree grammar, and G' the context free grammar constructed as before. Let t a subject tree with postorder listing $a_1 \ldots a_j w, a_i \in A, w \in A^*$. Then pattern β represented by production $X \to post(\beta)$ of G' matches at node j in left context α if and only if there is a rightmost derivation in the grammar G' of the form

$$S \Longrightarrow^* \alpha X z \Longrightarrow \alpha post(\beta) z \Longrightarrow^* \alpha a_h \ldots a_j z \Longrightarrow^* a_1 \ldots a_j z, z \in A^*$$

where $a_h \ldots a_j$ is the subtree rooted at node j.

Theorem 2.2 Let G' be a context free grammar constructed from a regular tree grammar in normal form. Then for any input string w which is a prefix of a string in $L(G')$, all viable prefixes induced by w are of the same length.
Proof The proof rests on the following four observations.
1. A shift of any symbol is always followed by a reduction.
2. If the symbol shifted is a terminal symbol, then the length of the viable prefix remains the

same, as the handle is of length 1.
3. If the symbol is an operator op, then the viable prefix reduces by length $arity(op)$.
4.Reduction by chain rules does not change the length of the viable prefix.
The first three observations are a consequence of the fact that the grammar is in normal form, and do not depend on the rightmost derivation sequence used. Therefore all parsing sequences for an input string yield viable prefixes of the same length.

The property stated in the last theorem suggests an augmentation of the LR parsing technique for preprocessing the context free grammar. The construction of the deterministic finite automaton(dfa) that recognizes viable prefixes of an LR grammar can be modified to produce one that recognizes *sets* of viable prefixes of a context free grammar derived from a regular tree grammar. The construction of this dfa may either be viewed as an extension of the technique used to construct a dfa to recognize canonical sets of LR items, [5] or as a restriction of the Earley algorithm [10]. In either case, the dfa has two kinds of transitions, those on terminals, and those on *sets* of nonterminals. Any path from the start state of the dfa to a reduce state will spell out a string of the form $S_1S_2\ldots S_na$, where the S_i are sets of nonterminals. The associated viable prefixes are of the form $X_1X_2\ldots X_na$ where $X_i \in S_i$. (Note, however, that not all concatenations of elements of the sets S_i are viable prefixes). The dfa satisfies the following property:

Property Item $[A \longrightarrow \alpha.\beta]$ is valid for viable prefix $\gamma = X_1X_2\ldots X_n$ if and only if $\delta(q, S_1S_2\ldots S_n)$ contains $[A \longrightarrow \alpha.\beta]$, where $X_i \in S_i$.

Once the dfa is constructed, matching reduces to a parsing process. There are no shift reduce conflicts (except where there are chain rules, and these are handled separately). Multiple reductions applicable at a point are all handled simultaneously.

With regular tree pattern matching reduced to parsing, all we need to store during matching, is the equivalent of the LR parsing table for the input grammar.

A relevant question that arises is: Are there any gains in succinctness of representation that arise from using a pushdown automaton instead of a finite state tree pattern matching automaton? The following theorem answers the question.

Theorem 2.3 There exist families of languages for which the input patterns have height h, the finite state tree tree pattern matching automaton has number of states $O(2^{2^h})$, whereas the auxiliary automaton has size $O(2^{h+2})$.
Proof We modify slightly, the family of balanced binary tree patterns described in [8], which constitute a worst case input instance for algorithms that construct bottom up finite state tree pattern matching automata. Let $P_j^i, i \geq 0, 1 \leq j \leq 2^i$ be a class of balanced binary tree patterns of height i with all internal nodes labeled $a \in OP$. There are two non terminals, S, the start symbol, and V. In P_j^i, all the leaves are labeled V, except the j^{th} leaf from the left which is labeled b. From the set of tree patterns of height i, we construct a set of tree patterns of height $i+1$ as follows:

$$R_j^{i+1} = op(l_j, P_j^i), l_j \in T, 1 \leq j \leq 2^i$$

To these patterns , we add the patterns b and c. There will be a total of $2^{i+2} + i$ normalized productions. Example 4.1 below shows the set of unnormalized productions for the case $i = 3$. For each subset of patterns, containing distinct elements $\{P_{j_1}^i, P_{j_2}^i, \ldots P_{j_k}^i\}, j_l \in \{1, 2, \ldots 2^i\}, i \leq l \leq k$, there is a subject tree for which exactly this set of patterns matches at the root. (This is just the binary tree of height i with a at all internal nodes, a b at each leaf position j_l and c at every other leaf. Thus a finite state tree pattern matching automaton contains at least 2^{2^i} states. However, each pattern P_j^i matches in a different left context and therefore no two or more patterns can appear together in any matchset associated with the LR automaton. The number of matchstates and LCstates are both $O(2^{i+2})$ in this case.

Example 2.3
The regular tree grammar(not in normal form) below, illustrates the case $i = 3$. The number of normalized productions is 35.

$S \longrightarrow op(l_1, a(a(a(b,V), a(V,V)), a(a(V,V), a(V,V))))$
$S \longrightarrow op(l_2, a(a(a(V,b), a(V,V)), a(a(V,V), a(V,V))))$
$S \longrightarrow op(l_3, a(a(a(V,V), a(b,V)), a(a(V,V), a(V,V))))$
$S \longrightarrow op(l_4, a(a(a(V,V), a(V,b)), a(a(V,V), a(V,V))))$
$S \longrightarrow op(l_5, a(a(a(V,V), a(V,V)), a(a(b,V), a(V,V))))$
$S \longrightarrow op(l_6, a(a(a(V,V), a(V,V)), a(a(V,b), a(V,V))))$
$S \longrightarrow op(l_7, a(a(a(V,V), a(V,V)), a(a(V,V), a(b,V))))$
$S \longrightarrow op(l_8, a(a(a(V,V), a(V,V)), a(a(V,V), a(V,b))))$
$V \longrightarrow c$
$V \longrightarrow b$

The above is an example of a regular tree grammar for which the match sets computed by the algorithm that generates a finite state tree pattern matching automaton are approximations to the actual match sets, except at the last level. In other words, the match sets generated for trees of height less than 4 are bigger than the actual match sets that actually occur in any S-derivation tree.

In the next section we show that we can modify the preprocessing algorithm so as to produce an extended $LR(0)$ automaton which is augmented with costs.

3 Definitions

Our starting point here is a context free grammar derived from a regular tree grammar augmented with costs. Thus each production is associated with a non negative integer which is the cost of the instruction represented by the tree pattern. We denote this by $rulecost(p)$ for production p. Our aim is to create a LR like parsing table as in the previous section, where each state of the dfa also has relative cost information encoded into it. The technique used in this section is adaptated from [2]. We begin by defining certain entities. Let $G = (N, T, P, S)$ be a cost augmented regular tree grammar in normal form.

Definition 3.1 The absolute cost of a nonterminal X matching an input symbol a in left context ϵ, is represented by $abscost(\epsilon, X, a)$. For a derivation sequence d represented by $X \Longrightarrow X_1 \Longrightarrow X_2 \ldots \Longrightarrow X_n \Longrightarrow a$, let $C_d = rulecost(X_n \longrightarrow a) + \sum_{i=1}^{n-1} rulecost(X_i \longrightarrow X_{i+1}) + rulecost(X \longrightarrow X_1)$, then, $abscost(\epsilon, X, a) = min_d(C_d)$

Definition 3.2 The *absolute cost* of a nonterminal X matching a symbol a in left context α is defined as follows:
$abscost(\alpha, X, a) = abscost(\epsilon, X, a)$ if X matches in left context α
$abscost(\alpha, X, a) = \infty$ otherwise

Definition 3.3 The *relative cost* of a nonterminal X matching a symbol a in left context α is: $cost(\alpha, X, a) = abscost(\alpha, X, a) - min_{y \in N}\{abscost(\alpha, Y, a)\}$.

Having defined costs for trees of height one we next look at trees of height greater than one. Let t be a tree of height greater than one.

Definition 3.4 $abscost(\alpha, X, t) = \infty$ if X does not match t in left context α;
If X matches t in left context α, let $t = a(t_1, t_2, \ldots t_q)$ and $X \longrightarrow aY_1Y_2 \ldots Y_q$ where Y_i matches $t_i, 1 \leq i \leq q$;

Define $abscost(\alpha, X \longrightarrow Y_1Y_2 \ldots Y_q a, t) = rulecost(X \longrightarrow Y_1 \ldots Y_q a) + cost(\alpha, Y_1, t_1) + cost(\alpha Y_1, Y_2, t_2) + \ldots + cost(\alpha Y_1 Y_2 \ldots Y_{q-1}, Y_q, t_q)$
Then $abscost(\alpha, X, t) = min_{X \Longrightarrow \beta \Longrightarrow^* t}\{abscost(\alpha, X \longrightarrow \beta, t)\}$

Definition 3.5 The relative cost of a nonterminal X matching a tree t in left context α is:
$cost(\alpha, X, t) = abscost(\alpha, X, t) - min_{Y \Longrightarrow * t}\{abscost(\alpha, Y, t)\}$

Having defined the cost of a nonterminal matching in a left context, we next introduce the notion of an augmented rightmost derivation of a context free grammar derived from the regular tree grammar as in the previous section. To avoid complicating notation, we assume from now on, that $G = (N, T, P, S)$ is the context free grammar

Definition 3.6
A single *augmented derivation step* is of the form $\alpha < A, c > w \Longrightarrow_A \alpha < X_1, d_1 > \ldots < X_k, d_k >< a, 0 > w$ where

1. α is a sequence of $< nonterminal, cost >$ pairs, such that if α' is the concatenation of first members of pairs of α then $S \Longrightarrow^* \alpha' A w \Longrightarrow \alpha' X_1 X_2 \ldots X_k a w$
2. $\exists t s.t. A \Longrightarrow X_1 X_2 \ldots X_k a \Longrightarrow^* t$ with $X_i \Longrightarrow^* t_i, i \leq i \leq k$ and $t = t_1 t_2 \ldots t_k a$
3. $cost(\alpha', X_1, t_1) = d_1, cost(\alpha' X_1, X_2, t_2) = d_2, \ldots cost(\alpha' X_1 X_2 \ldots X_{k-1}, X_k, t_k) = d_k$ and $c = rulecost(A \longrightarrow X_1 \ldots X_k) + \sum_{i=1}^{k} d_i - min_B\{abscost(\alpha\prime, B, t)\}$.

Definition 3.7 An *augmented viable prefix* is any prefix of an augmented right sentential form such that the concatenation of the first elements yields a viable prefix.

Definition 3.8 An augmented item $[A \rightarrow \alpha \,.\, \beta,\ c]$ is valid for an augmented viable prefix $\gamma =< X_1, c_1 >< X_2, c_2 > \cdots < X_n, c_n >$ if,

- $[A \rightarrow \alpha \,.\, \beta]$ is valid for viable prefix $X_1 X_2 \cdots X_n$.
- If $\alpha = X_i \cdots X_n$ then, if $\beta \neq \epsilon$ then $c = c_i + c_{i+1} \ldots + c_n$ else $c = c_i + c_{i+1} \ldots + c_n + rulecost(A \longrightarrow \alpha)$

We are actually interested in a similar definition of validity for sets of augmented viable prefixes (which we term *set viable prefixes*), where the costs associated with the items are relative costs. Thus we are naturally led to the following definition.

Definition 3.9 An augmented item $[A \rightarrow \alpha.\beta, c]$ is valid for set viable prefix $S_1 S_2 \ldots S_n$ if

- There exists a viable prefix $\gamma =< X_1, c_1 >, < X_2, c_2 > \ldots, < X_n, c_n >$ in $S_1 S_2 \ldots S_n$ with $[A \rightarrow \alpha.\beta, c']$ valid for γ
- If $\beta \neq \epsilon$ then $c = c'$ else $c = c' - min_{\gamma' \in S_1 S_2 \ldots S_n}\{c\prime\prime : [\beta \rightarrow \delta., c\prime\prime]$ is valid for γ'.

4 Construction of the Auxiliary Automaton

The definitions above suggest the construction of a finite state machine similar to that used for the recognition of viable prefixes of an $LR(0)$ grammar. If the grammar is $LR(0)$, each prefix of a string in the language, induces exactly one viable prefix before any reduction, and there is at most one reduction possible at any step. In this case there may be several augmented viable prefixes induced by the same string. However, as these are all of the same length [15], shifts and reductions may be carried out using a single stack. This permits the use of a simple extension of the $LR(0)$ parsing technique. We first augment the grammar with the production $Z \longrightarrow S\$, [0]$ where \$ is a symbol not in the alphabet, in order to make the language prefix free. We next precompute sets of augmented items that are valid for all augmented viable prefixes induced by the same input string. All these are included in a single state. We can then construct a finite state automaton M which we will call the Auxiliary Automaton(AA) as follows:

$M = (Q, \Sigma, \delta, q_0, F)$
where $Q = \{I | I$ is a distinct set of augmented items$\}$
$\Sigma = A \cup 2^{<N,c>}$, where $< N, c >$ is a (*non terminal,cost*) pair
q_0 is the state associated with the initial set of augmented items.
F is the state containing the augmented item $[Z \longrightarrow S\$., 0]$.
δ is the transition function made up of δ_A, the set of transitions on elements of A, and δ_{LC}, those on sets of $(nonterminal, cost)$ pairs. The function δ can be extended to strings of symbols in the usual way.

Assume that we can precompute the states and transitions of this auxiliary automaton. This is then the controlling automaton for a parser for the language generated by the grammar G', the parser performing the additional role of minimum cost tree selection. The auxiliary automaton plays a role analogous to the dfa for canonical sets of $LR(0)$ items in an $LR(0)$ parser.

The auxiliary automaton to be constructed will satisfy the following property:
Property
An augmented item $[A \to \alpha\,.\,\beta,\, c]$ is valid for an augmented viable prefix $\gamma =< X_1, c_1 >< X_2, c_2 > \cdots < X_n, c_n >$, if and only if, $\delta(q_0, S_1 \cdots S_n)$ contains $[A \to \alpha\,.\,\beta,\, c]$, where $\gamma \in S_1 S_2 \ldots S_k$

As a consequence of the property, we are assured that if we use the LR parsing technique of shifts and reductions (with the difference that several reductions may be applied simultaneously), a derivation sequence in reverse which yields a state containing $[S \to \alpha.\, 0]$ is a cheapest (optimal) matching sequence. Although we can use a technique similar to the technique for construction of the automaton for recognizing viable prefixes of an LR grammar, there is one striking difference. The transitions out of a state of the automaton that recognizes viable prefixes of the LR grammar are determined by just the items associated with the state. Here, the transitions on sets of $(nonterminal, cost)$ pairs can be deduced only *after computing sets of such pairs that match in the same left context.* There are three functions required. The first, called the *goto* function, which is similar to the corresponding function of an $LR(0)$ automaton, restricted to only terminal symbols, computes the new set of augmented items obtained by making a transition from the old set on a terminal or operator symbol. Because our grammar is in normal form, all such transitions will always lead to complete items. Such a set of items in our terminology, is called a *matchset*.
The *reduction* function has two parameters. The first is a set of complete items all of which call for reductions. The second is a set of augmented items $\mathcal{I}$ associated with a state which will be uncovered on stack when the righthand sides of all these productions are simultaneously popped off the stack. (For the time being let us not concern ourselves with how exactly such a set of items is identified). We call such a set of items an $LCset$, and we say that a state associated with such a set is a *valid left context state* for the matchset. *reduction* first collects the left hand side $(nonterminal, cost)$ pairs associated with a *matchset* into a set $\mathcal{R}$, and computes relative costs. It then augments $\mathcal{R}$ with $(nonterminal, cost)$ pairs that can be added using chain rules that are applicable, updating costs as it proceeds. Finally, it creates a new set of augmented items by making transitions on elements of $\mathcal{R}$ from $\mathcal{I}$. We describe these functions formally below. For a given matchset m we use S_m to denote the set of lefthand side non terminals of complete items.

$$\begin{aligned}
&goto(itemset, a) = \{[A \to \alpha\, a\,., c] /\ [A \to \alpha.a, c'] \in itemset \text{ and} \\
&\quad c = c' + rule_cost(A \to \alpha\, a) - \\
&\quad \min\{c'' + rule_cost(B \to \beta\, a) /\ \ [B \to \beta.a, c''] \in itemset\}\}
\end{aligned}$$

The *reduction* operation on a set of complete augmented items $itemset_1$ with respect to another set of augmented items, $itemset_2$ is encoded in the form of the following function.

```
function reduction(itemset_2, itemset_1) begin
    // Costs of non-terminals in matchsets
    S = S_{itemset_1}
    cost(X) = min{c_i / [X -> α_i., c_i] ∈ itemset_1} if X ∈ S
                    ∞ otherwise
    // Process chain rules and obtain costs of non-terminals
    temp = ⋃{[A -> B., c] / ∃[A -> .B, 0] ∈ itemset_2 and [B -> γ., c_1] ∈ itemset_1
                        and c = c_1 + rule_cost(A -> B)}
    repeat
        S = S ∪ {X / [X -> Y., c] ∈ temp}
        for each X ∈ S do
            cost(X) = min(cost(X), min{c_i / ∃ [X -> Y_i., c_i] ∈ temp})
        temp = {[A -> B., c] / ∃[A -> .B, 0] ∈ itemset_2
                        and [B -> Y., c_1] ∈ temp and c = c_1 + rule_cost(A -> B)}
    until no change to cost array or temp = φ
    // Compute reduction
    reduction = ⋃{[A -> α B .β, c]  / [A -> α .B β, c_1] ∈ itemset_2 and B ∈ S
                    and c = cost(B) + c_1
end
```

The *closure* function remains the same except that the definitions are modified to take into account the cost. The function is encoded as follows:

```
function closure(itemset)
begin
    repeat
        itemset = itemset ⋃{[A -> .α, 0] / [B -> .Aβ, c] ∈ itemset}
    until (no change to itemset);
    closure = itemset;
end
```

As defined earlier, $ClosureReduction(itemset_1, itemset_2)$ is just, $closure(reduction(itemset_1, itemset_2))$.
To identify states that can be valid left context states for a given matchset, we follow a simple technique that tests a condition that is necessary but not sufficient. As a consequence some spurious states may be generated (which are unreachable at run time). The choice of this over the exact technique [15] was dictated by considerations of efficiency, and the fact that for all the examples tested out by us, the approximate test gave the same results as the exact test, except in one case where it produced one extra state. Informally speaking we check if each item of the form $A \longrightarrow \alpha.$ in a matchset has a corresponding item of the form $A \longrightarrow .\alpha$ in the state under test for the valid left context property, and that there is a tally of such items. The condition is not sufficient because there may be another production of the form $B \longrightarrow \beta$ that always matches in this left context along with the other productions, but whose corresponding complete item is not in the matchset. We omit a formal definition of this test here, and refer to it as a boolean function $validlc(p, m)$ (where p is an LCset and m is a matchset) which returns true if p satisfies the condition to be eligible as a valid LCset for m.

Having defined these functions, we now present the routine for precomputation.

Algorithm *Preprocess*
Input A cost augmented context free grammar $G' = (N, T, P, S)$ representing a regular tree grammar G in normal form.
Output The auxiliary automaton represented by tables δ_A, δ_{LC}, which represent transitions on elements of A and 2^N respectively
begin
$lcsets := \phi$;
$matchsets := \phi$;
$list := closure(\{[S \to .\alpha] \mid S \to \alpha \in P\})$;
  while $list$ is not empty **do**

    delete next element q from $list$ and add it to $lcsets$;

    for each $a \in A$ such that $goto(q, a)$ is not empty **do**

      $m := goto(q, a)$;

      $\delta_A(q, a) := (match(m), S_m)$;

      if m is not in $matchsets$ **then**

        $matchsets := matchsets \cup \{m\}$;
        for each state r in $lcsets$ **do**
          if $validlc(r, m)$ **then**
          $p := ClosureReduction(r, m)$;
          $\delta_{LC}(r, S_m) := (match(p), p)$;
          if p is not in $list$ or $lcsets$ **then** append p to $list$ **endif**
          endif
        endfor

      endif

    endfor

    for each state t in $matchsets$ **do**

      if $validlc(q, t)$ **then**

        $s := ClosureReduction(q, t)$;
        $\delta_{LC}(q, S_t) := (match(s), s)$;
        if s is not in $list$ or $lcsets$ **then** append s to $list$ **endif**;

      endif

    endfor

  endwhile
end

5 An Example

We now look at an example with cost precomputation.

The grammar is a transformed version of that in Example 2.1, and is, $G = (\{V, B, G\}, A, P, V)$ where, $A = \{a, b\}$ and the arities are 2 and 0 respectively. The productions P, augmented with cost are as follows:

$V \to VBa$ [0]
$V \to GVa$ [1]

$V \to G$ [1]
$G \to B$ [1]
$V \to b$ [7]
$B \to b$ [4]

The automaton is shown in Figure 2 below.

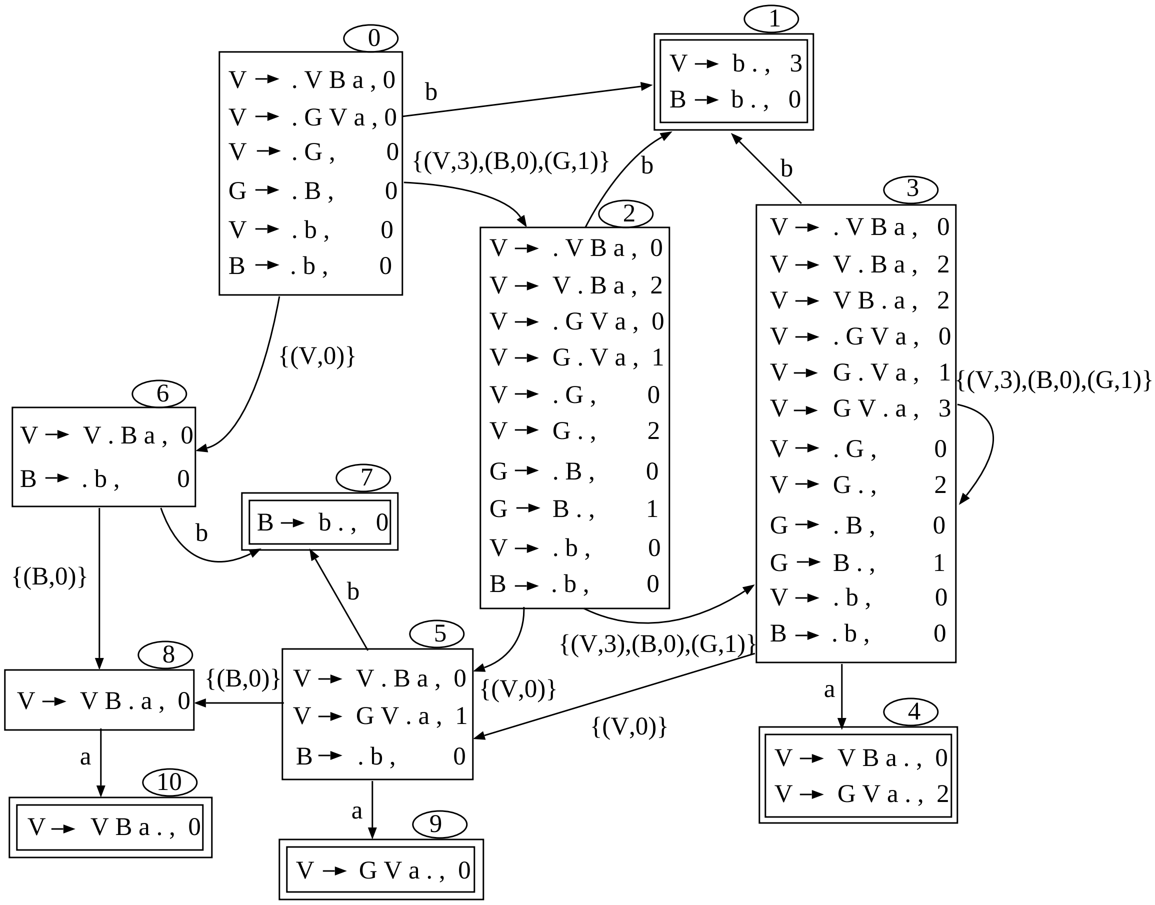

Figure 2: Example with costs

Let us look at a typical step in the preprocessing algorithm. Let the starting state be q_0, i.e. the first LCset.

$q_0 = \{\, [V \to .\, V\, B\, a,\, 0], [V \to .\, G\, V\, a,\, 0],\, [V \to .\, G,\, 0], [G \to .\, B,\, 0],$
$[V \to .\, b,\, 0], [B \to .\, b,\, 0]\, \}$

Using the definition of the *goto* operation, we can compute the matchset q_1 as,

$q_1 = goto(q_0, b) = \{\, [V \to b\, .,\, 3], [B \to b\, .,\, 0]\, \}$

In the matchset, the set of matching non-terminals S_{q_1} is,

$S_{q_1} = \{V,\, B\}$, with costs 3 and 0 respectively.

Now, we can compute the set $ClosureReduction(q_0, q_1)$. First we compute $reduction(q_0, q_1)$.

Initialization :

$\mathcal{S} = S_{q_1} = \{V,\, B\}$
$cost(V) = 3$
$cost(B) = 0$
$cost(G) = \infty$

$temp = \{[G \rightarrow B\,.,1]\}$
Processing chain rules:
Iteration 1:
$\mathcal{S} = \mathcal{S} \cup \{G\} = \{V,\, B,\, G\}$
$cost(V) = 3$
$cost(B) = 0$
$cost(G) = 1$
$temp = \{[V \rightarrow G\,.,2]\}$
Iteration 2:
$\mathcal{S} = \mathcal{S} \cup \{V\} = \{V,\, B,\, G\}$
$cost(V) = 2$
$cost(B) = 0$
$cost(G) = 1$
$temp = \phi$
Computing reduction:

$$reduction = \{[V \rightarrow V\,.\,B\,a,\, 2],\, [V \rightarrow G\,.\,V\,a,\, 1],\, [V \rightarrow G\,.,\, 2],\, [G \rightarrow B\,.,\, 1]\}$$

Once we have $reduction(q_0, q_1)$, we can use the *closure* function and compute *ClosureReduction*. Therefore,

$$\begin{aligned} q_2 &= ClosureReduction(q_0, q_1) \\ &= closure(reduction(q_0, q_1)) \\ &= closure(\{[V \rightarrow V\,.\,B\,a,\, 2],\, [V \rightarrow G\,.\,V\,a,\, 1],\, [V \rightarrow G\,.,\, 2],\, [G \rightarrow B\,.,\, 1],\})\\ &= \{[V \rightarrow .\,V\,B\,a,\, 0],\, \{[V \rightarrow V\,.\,B\,a,\, 2],\, [V \rightarrow .\,G\,V\,a,\, 0],\, [V \rightarrow G\,.\,V\,a,\, 1],\\ &\quad [V \rightarrow .\,G,\, 0],\, [V \rightarrow G\,.,\, 2],\, [G \rightarrow .\,B,\, 0],\, [G \rightarrow B\,.,\, 1],\, [V \rightarrow .\,b,\, 0],\, [B \rightarrow .\,b,\, 0]\}\end{aligned}$$

6 Correctness of the Construction

We next prove that the automaton constructed by the algorithm satisfies the property below:

Property
Augmented $LR(0)$ Item $[A \rightarrow \alpha.\beta, c]$ is valid for augmented viable prefix $\gamma = Y_1Y_2 \ldots Y_k$ in $S_1S_2 \ldots S_k$ if and only if $\delta(q_0, S_1S_2 \ldots S_k)$ contains $[A \rightarrow \alpha.\beta, c]$.
Proof We give an informal proof that the property is satisfied, ignoring costs. The argument is based on the fact that a similar property (differing from the one above only in that the function δ is applied on a single viable prefix instead of a set of viable prefixes) holds for a dfa for canonical set of $LR(0)$ items [9]. The construction using algorithm *Preprocess* may be viewed as a subset construction beginning with the $LR(0)$ dfa. We begin with the start state. Suppose m is a matchstate reached during some point in the algorithm, with $S_m = \{X_1, X_2, \ldots X_l\}$. In the $LR(0)$ dfa we would have individual transitions from valid left context states on each of $X_1, X_2, \ldots X_l$ to different states. Here, we collect the items in all those states into a single state, and then augment the items with those obtained with the application of chain rules and got by the closure operation.(This is exactly what *ClosureReduction* does given a matchstate and an LCstate). Thus the state we reach in the auxiliary automaton on a label sequence $S_1S_2 \ldots S_k$ is got by merging individual states we would have reached by following individual viable prefixes of the form $Y_1Y_2 \ldots Y_k$ in the dfa for $LR(0)$ item sets induced by the same input string. The property follows from the fact that it holds for each of the viable prefixes in the $LR(0)$ automaton. That the property holds when costs are included in the itemsets follows directly from the definition of an augmented item being valid for an augmented viable prefix.

7 Table Compression

It is possible to compress the tables by defining certain equivalence relations on the set $lcsets$.

Let $arityset$ be the set of arities of symbols of A. Define a set of equivalence relations $\{R_i \mid i \in arityset\}$ on the set $lcsets$ as follows.

If p and q are in $lcsets$, then pR_iq if $\delta_A(p,a) = \delta_A(q,a)$ for all a with $arity(a) = i$. The table δ_A now splits into several tables δ^i_A, one for each arity. The rows of the table δ^i_A correspond to the equivalence classes of R_i. The columns correspond to columns of A_i where A_i is the set of symbols of A with arity i.

Define the set NT_i as

$$NT_i = \{B \mid B \to \alpha a \in P, arity(a) = i\}$$

Equivalence relation U_i is defined on $lcsets$ as follows. For states p and q, pU_iq if for all (nonterminal,cost) pairs $< B, c >$ in $NT_i \times C$, where C is the set of relative costs, $\delta_{LC}(p, < B, c >) = \delta_{LC}(q, < B, c >)$. The table δ_{LC} now splits into several tables δ^i_{LC}, one for each value of arity. The table δ^i_{LC} has one row for each equivalence class of U_i and a column for each distinct set of matching (nonterminal,cost) pairs that is a subset of $NT_i \times C$.
The compression of the tables can be done on line, while the tables are generated, as the equivalence classes can be computed from the sets of items associated with the states. Each equivalence relation would need an index map which maps from original indices to indices in that equivalence class, which are then used to access table entries. During matching, if the next symbol is in A_i, then table δ^i_A is first consulted through its index map, followed by a lookup of table δ^i_{LC} through the appropriate index map.

8 Complexity of the Algorithm

8.1 Auxiliary Space Complexity

Let $lcsets$ be the set of LC sets, $\mid R_i \mid$ ($\mid U_i \mid$) be the number of equivalence classes of $R_i(U_i)$, and $RC_i(UC_i)$ the equivalence classes of $R_i(U_i)$, $i \in arityset$.

The following maps and tables have to be maintained for use during matching, for $i \in arityset$.

- $\delta^i_A : RC_i \times A_i \to (matchsets \times 2^{NT_i \times C})$
- $\mu_i : lcsets \to RC_i$
- $\lambda_i : lcsets \to UC_i$
- $\delta^i_{LC} : UC_i \times 2^{NT_i \times C} \to lcsets$

It is evident that the sizes of $lcsets$ and $matchsets$ determine the sizes of the tables, so we first compute loose bounds on the sizes of these sets. Let h be the maximum height of a tree pattern. This, for a regular tree grammar, is the maximum height of trees in $TREES(A)$ to be considered using the conventional algorithm, to obtain all the matchsets. Let nLC be the size of the set $LCsets$, and nM that of $matchsets$.

Lemma 8.1. $nLC + nM \leq 2^{|<NT,c>| \times (maxarity-1) \times h}$, where $|NT, c > |$ is the size of the set of (nonterminal,cost) pairs.
Proof Consider a node in any derivation tree for a subject tree of height bounded by h. The number of maximal complete subtrees to the left of the node represents the length of a path in the auxiliary automaton, encoding a set of viable prefixes induced by the prefix of the input postorder

string ending at the postorder predecessor of the terminal or operator associated with the node. The edges of this path are labelled by sets consisting of matching (nonterminal,cost) pairs. The maximum length of any such path is $(maxarity - 1) \times h$. Since each edge on the path may be labeled by a set of matching (nonterminals,cost) pairs, the a bound on the total number of states of the auxiliary automaton is $2^{|<NT,c>| \times (maxarity-1) \times h}$.

8.2 Time Complexity

We next estimate the time complexity of our algorithm.
We estimate the complexity of Algorithm *SimplePreprocess* as the auxiliary space bounds estimated above hold for this algorithm as well.

The **while** loop is executed $\mid lcsets \mid$ times. Within the **while** loop the first **for** loop is executed $\mid A \mid$ times, the complexity of each execution being bounded by $\mid lcsets \mid \times \mid NT \mid \times patsize$ assuming that a *ClosureReduction* operation and computation of *validlc* each take time $\mid NT \mid patsize$ and that all set and table entry operations take constant time. The last **for** loop is executed $\mid matchsets \mid$ times for each execution of the **while** loop, the complexity of each execution being bounded by $\mid NT \mid \times patsize$. The first **for** loop dominates the execution within the **while**. Hence the overall time complexity is bounded by $\mid lcsets \mid^2 \times \mid A \mid \times \mid NT \mid \times patsize$.

9 A Sufficient Condition for Termination

In some cases, the input augmented grammar may be such that the algorithm *Preprocess* does not terminate. Thus, there is no finite automaton that can store the precomputed cost information. In such cases, the only alternative is to perform dynamic cost computations.

Recall that the states of the auxiliary automaton consist of sets of $(LR(0)item, cost)$ pairs. Thus the number of states is unbounded if and only if the cost component is unbounded. This will happen whenever there is a pair of nonterminals, such that both elements of the pair match in the same left context, and derive the same sequence of infinite trees with increasing cost differences. This is illustrated in the example below.

Example 9.1
Consider the grammar, $G = (\{S, A, B, C\}, \{d, b, a, e\}, P, S)$, where the arities of the terminals (in the order that they appear in the set) are 2,0,0 and 0, and the augmented productions are,

$$
\begin{array}{lll}
S & \rightarrow A\,S\,d & [0] \\
S & \rightarrow B\,S\,d & [0] \\
A & \rightarrow C\,A\,d & [0] \\
B & \rightarrow C\,B\,d & [5] \\
A & \rightarrow b & [1] \\
B & \rightarrow b & [1] \\
C & \rightarrow a & [1] \\
S & \rightarrow c & [1]
\end{array}
$$

The nonterminals A and B match an infinite sequence of input trees in the same left context, with increasing relative costs(which increase in multiples of 5). As a result, Algorithm *Preprocess* will not terminate on this input instance.

The use of left-contextual information, results in the generation of finite parsing tables for problem instances on which conventional techniques that produce finite state tree pattern matching automata do not terminate. One such example is given below.

Example 9.2

$$\begin{array}{lcll} S & \rightarrow & X\,S\,d & [0] \\ S & \rightarrow & Y\,S\,d & [0] \\ X & \rightarrow & L1\,A\,e & [0] \\ Y & \rightarrow & L2\,B\,e & [0] \\ A & \rightarrow & C\,A\,d & [0] \\ B & \rightarrow & C\,B\,d & [5] \\ A & \rightarrow & b & [1] \\ B & \rightarrow & b & [1] \\ C & \rightarrow & a & [1] \\ S & \rightarrow & c & [1] \\ L1 & \rightarrow & l1 & [0] \\ L2 & \rightarrow & l2 & [0] \end{array}$$

Since the non-terminals A and B do not match in the same left context, it does not matter that these non-terminals match the same infinite sequence of trees with unbounded cost difference. The table construction for this example using our algorithm terminates with 25 states (13 LCsets and 12 matchsets). However, the computation using any bottom algorithm that constructs a bottom up finite state tree pattern matching automaton will not terminate.

We now give a sufficient condition for termination of the algorithm. The condition is *sufficient* but not *necessary*, as it disqualifies a grammar if there is a pair of nonterminals that match in the same left context, and generate the same sequence of infinite trees, without checking the condition on costs. We first motivate the development of the sufficiency condition.

Let m be a matchset. For each valid left context state p for m, one can associate an enhancement of m by associating with each complete item $A \longrightarrow \alpha.$in m a set of nonterminals representing the completion of A using all chain rules applicable in p. Thus, we can associate with each matchset, a collection of enhanced matchsets. Each element of an enhanced matchset is a set of complete items, each item associated with a set of nonterminals, obtained as described above. Let us use the symbol $\mathcal{C}_m$ to describe the collection of nonterminal sets associated with enhanced matchset m. Let $\mathcal{M}$ be the collection of enhanced matchsets associated with the auxiliary automaton without costs.

We also define an *aligned nonterminal set* below:

Definition 9.1 Let m be a matchset with complete items $[A_1 \rightarrow X_{11}X_{12}\cdots X_{1n}.]$, $[A_2 \rightarrow X_{21}X_{22}\cdots X_{2n}.]$, $\ldots$, $[A_k \rightarrow X_{k1}X_{k2}\cdots X_{kn}.]$.

The sets $\{X_{11}, X_{21}, \ldots, X_{k1}\}$, $\{X_{12}, X_{22}, \ldots, X_{k2}\}$, $\ldots$,$\{X_{1n}, X_{2n}, \ldots, X_{kn}\}$ are all termed aligned non-terminal sets for matchset m. Aligned nonterminal sets are sets of non terminals that match in the same left context.

We next construct a directed graph $\mathcal{G} = (\mathcal{M}, \mathcal{E})$ from the automaton (without costs) [15]. $\mathcal{M}$, the vertex set consists of all enhanced matchsets. There is an edge from node m_1 to node m_2 if and only if it is possible to form a set by selecting at least one element from the nonterminal set associated with each complete item in m_1 so that the set thus formed is an aligned non-terminal set for node m_2.

The graph constructed above, encapsulates certain properties of *fused* derivation trees defined below. Consider a subject tree t, and a derivation tree for it. The derivation tree induces a labeling of the nodes of the tree as follows. Each node is labeled with the set of nonterminals that match at that node. (In general, there will be a *set* because of the existence of chain rules). Suppose now, that we superimpose all derivation trees for this subject tree. Instead of a set of nonterminals, there will be a *collection* of sets labeling each node. Let us call this tree with a collection of sets labeling each node a *fused* tree for t.

The following relationship exists between the graph $\mathcal{G}$ and the set of fused derivation trees for the grammar:

1.Each node of a fused derivation tree corresponds to an enhanced matchset.

2.There is a node with label set $\mathcal{C}_1$ with parent with label set $\mathcal{C}_2$ in some fused derivation tree for the grammar if and only if there is a edge from an enhanced matchset with associated collection $\mathcal{C}_1$ to another with associated collection $\mathcal{C}_2$ in the graph $\mathcal{G}$.

A moments reflection will convince the reader that both these statements are true. Using the second property inductively, we can conclude that it holds with the word *parent* replaced by *proper ancestor* and *egde* replaced by *path*. Finally if the sets $\mathcal{C}_1$ and $\mathcal{C}_2$ are the same, then the path is a cycle. The last statement implies that a cycle exists in $\mathcal{G}$ if and only if there is an ancestor-descendant path in some fused derivation tree with both the nodes having the same collection as label. This in turn implies that the tree can nest itself, which is the same thing as saying that there is an infinite sequence of trees matching a set of nonterminals in the same left context. We have given an informal argument in support of the following theorem:

Theorem 9.1 Algorithm *Preprocess* will terminate if the graph $\mathcal{G}$ constructed above has no cycles.

That the condition is only sufficient and not necessary can be verified by modifying the grammar of example 3.1 so that the costs of the recursive productions for the nonterminals A and B are the same. The corresponding graph $\mathcal{G}$ has a cycle but algorithm *Preprocess* terminates.

10 Conclusion

We have given an algorithm that solves the problem of optimal linear regular tree pattern matching by constructing a pushdown automaton that performs matching in linear time, and precomputes costs. The size of the automaton so constructed can be exponential in the size of all patterns in the worst case [15]. There are, however, pathological input instances where the finite state tree pattern matching automaton in exponential in the size of the pushdown automaton constructed using our algorithm. We have also shown that the algorithm is able to handle a larger class of cost augmented input grammars than conventional algorithms.

We have run, both, the algorithm proposed here, and the conventional algorithm[14] on some input instances that represent instruction sets for two machines. The results are displayed in Figures 3 and 4 below.

	Intel Pentium		IBM R6000	
Productions	351		60	
Nonterminals	26		13	
Terminals & operators	212		30	
	With costs	Without costs	With costs	Without costs
Number of states of AA (matchsets + LCsets)	421 + 144	266 + 67	65 + 77	65 + 47
Total table size (after compression)	3559	1965	802	738
Size of index maps	864	402	462	282
Total preprocessing time (seconds)	27.3	6.3	1.0	0.7

Figure 3: Table sizes and execution times using our algorithm.

	Intel Pentium		IBM R6000	
	With costs	Without costs	With costs	Without costs
Total table size	46566	32505	2747	2747
Total preprocessing time (seconds)	2.29	1.50	0.22	0.22

Figure 4: Table sizes and execution times for the conventional algorithm.

Our algorithm generates smaller tables for these instances, but requires more preprocessing time. More experimentation is necessary to study the space/time tradeoffs between static and dynamic cost computations. The fact that our optimal matcher(or instruction selector), operates like an LR parser(quite different from the Graham Glanville technique, as it carries forward all choices, finally selecting an optimal one), allows the use of attributes, which may prove useful for carrying out other tasks in code generation.

References

[1] Aho, A.V., and Ganapathi, M., Efficient tree pattern matching: an aid to code generation, in: *Proc. 12th ACM Symp. on Priciples of Programming Languages* (1985) 334-340.

[2] Balachandran, A., Dhamdhere, D.M., and Biswas, S., Efficient retargetable code generation using bottom up tree pattern matching, *Comput. Lang.* **3**(15)(1990) 127-140.

[3] Chase, D., An improvement to bottom up tree pattern matching, in: *Proc. 14th Ann. ACM Symp. on Principles of Programming Languages* (1987) 168-177.

[4] Ferdinand, C., Seidl, H., and Wilhelm, R., Tree Automata for Code Selection, *Acta Informatica*, **31**,(1994) 741-760.

[5] Gantait, A., Design of a bottom up tree pattern matching algorithm and its application to code generation, ME Project Report, Department Of Computer Science and Automation, Indian Institute of Science, Bangalore, 1996.

[6] Glanville, R.S., and Graham, S.L., A new approach to compiler code generation, in: *Proc. 5th ACM Symposium on Principles of Programming Languages*, (1978) 231-240.

[7] Hatcher, P., and Christopher, T., High-quality code generation via bottom-up tree pattern matching, in:*Proc. 13th ACM Symp. on Principles of Programming Languages* (1986) 119-130.

[8] Hoffmann, C., and O'Donnell, M.J., Pattern matching in trees, *J. ACM* **29**(1)(1982) 68-95.

[9] Hopcroft, J.E., and Ullman, J.D., An Introduction to Automata Theory, Languages and Computation, Addison Wesley(1979).

[10] Madhavan, M., Optimal linear regular tree pattern matching using pushdown automata, MSc (Engineering) Thesis(submitted), Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 1998.

[11] Nymeyer, A., and Katoen, J.P., Code generation based on formal BURS theory and heuristic search, *Acta Informatica*, **34**(8),(1997) 597-636.

[12] Pelegri-Llopart, E., and Graham, S.L., Optimal code generation for expression trees: An application of BURS theory, in:*Proc. 15th ACM Symposium on Principles of Programming Languages*, (1988) 119-129.

[13] Proebsting, T.A., BURS automata generation, *ACM Trans. on Prog. Lang. and Sys.* **3**(17),(1995) 461-486.

[14] Ravi Kumar, S., Retargettable code generation using bottom up tree pattern matching, M.E Project Report, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 1992.

[15] Shankar, P., Gantait, A., Yuvaraj, A.R., and Madhavan, M., A New algorithm for linear regular tree pattern matching, (To appear in *Theoretical Computer Science*).