SIMPLE AND EFFICIENT BURS TABLE GENERATION

by

Todd A. Proebsting

Computer Sciences Technical Report #1065

December 1991

Simple and Efficient BURS Table Generation

Todd A. Proebsting
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
(608) 262-0018 todd@cs.wisc.edu

November 20, 1991

Abstract

A simple and efficient algorithm for generating bottom-up rewrite system (BURS) tables is described. A small prototype implementation produces tables 10 to 30 times more quickly than the best current techniques. The algorithm does not require novel data structures or complicated algorithmic techniques, yet it generates compact tables very quickly. The algorithm is optimized to exploit knowledge specific to BURS table generation. In addition, previously published methods for on-the-fly elimination of states are generalized and simplified to create a new method, *triangle trimming*, that is employed in the algorithm.

1 Introduction

Tree pattern matching combined with dynamic programming can be used in code generators to create locally optimal code for expression trees [AGT89]. Code generators based on bottom-up rewrite system (BURS) theory can be extremely fast because all dynamic programming is done when the BURS pattern matching automaton is built. At compile time, it is only necessary to make two traversals of the subject tree: one bottom-up traversal to label each node with a *state* that encodes all optimal matches, and a second top-down traversal that uses these states to select and emit code. Fraser and Henry [FH91b] report that careful encodings can produce an automaton that executes fewer than 50 VAX instructions per node to do both traversals.

Two difficulties arise in creating a BURS-style code generator: efficiently generating the states and state-transition tables, and creating an efficient encoding of the automata for use in a compiler. A solution to the encoding problem is described by Fraser and Henry in [FH91b]. (Their techniques were used in the implementation described in this paper.) Because *all* potential dynamic programming decisions are done at table-generation time, they must be done efficiently. This paper describes a simple and efficient table generation algorithm whose implementation is an order of magnitude faster than the best current systems.

To efficiently generate BURS tables, we have designed and implemented a simple algorithm. Simplicity has increased, not decreased, efficiency. Efficiency has been enhanced, and tables sizes kept small, by the development of a new technique, *triangle trimming*, for eliminating most redundant states. Triangle trimming is an uncomplicated optimization that, for complex grammars, reduces *both* the table generation time and table sizes by over 50%. We also describe optimizations that take advantage of special properties of BURS states.

2 Related Work

Naively generating BURS states and state-transition tables fails because the tables become too large. A typical CISC machine description will generate over 1000 states.[1] Directly encoding the transition table for a *single* binary operator would, therefore, require over 1,000,000 entries.

[1] The integer *subset* of a Motorola 68000 grammar has over 800 states (Figure 6).

Rule#	Simple Grammar LHS		RHS	Cost	Canonical Form LHS		RHS	Cost
1.	goal	→	reg	(0)	goal	→	reg	(0)
2.	reg	→	Reg	(0)	reg	→	Reg	(0)
3.	reg	→	Int	(0)	reg	→	Int	(0)
4.	addr	→	Plus(reg, Int)	(0)	addr	→	Plus(reg, n.1)	(0)
4a.					n.1	→	Int	(0)
5.	reg	→	Fetch(addr)	(2)	reg	→	Fetch(addr)	(2)
6.	reg	→	Plus(reg, reg)	(2)	reg	→	Plus(reg, reg)	(2)
7.	reg	→	Plus(Plus(reg, reg), reg)	(3)	reg	→	Plus(n.2, reg)	(3)
7a.					n.2	→	Plus(reg, reg)	(0)

Figure 1: Simple Grammar and Its Canonical Form

Chase [Cha87] discovered that many of the rows (and columns) of such transition tables are identical and proposed the idea of ***index maps*** to allow the creation of much smaller tables. Index maps are vectors that map states of the automaton to *representer states* for indexing a transition table. Many states may then share a given row or column of a transition table at the cost of a single indirection. Chase demonstrated that these maps can be produced on-the-fly during table generation so that no superfluous work will be performed.

Pelegri-Llopart, the originator of BURS theory ([PLG88], [PL88]), encorporated Chase's ideas into a system that that added cost information for dynamic programming at table generation time. Balachandran, Dhamdhere, and Biswas [BDB90] also generalized Chase's ideas to use cost information.

Henry [Hen89] developed optimization techniques that can be used to limit the number of BURS states produced during table generation. With fewer states, a smaller automaton is produced more quickly. His techniques are much more aggressive than the simple index map techniques, and therefore produce smaller automata more quickly, but at the cost of increased complexity. In [Hen89], Henry states, "The table builder uses space and time voraciously, even though it uses very complex algorithms designed to minimize these resources." Our algorithm generalizes and simplifies his work, resulting in substantial speed gains with reduced complexity in our implementation. Our system can be directly compared to his on a variety of machine specifications, and shows a factor of 10 to 30 improvement in speed.

The algorithm described in this paper differs from earlier work in its simplicity and efficiency. Previous algorithms rely on bit-vectors that encode information about pattern matching, and rely on auxiliary data structures to maintain cost information. Our algorithm has a unified structure that maintains cost information as well as a description of the pattern to be matched.

3 BURS Model

The input to a BURS-style code generator generator is a set of rules. Each rule indicates a tree pattern, a cost, a replacement symbol, and an action. The set of all the rules is called the *grammar*. Figure 1 gives a small sample grammar (without actions). The replacement symbol is a *nonterminal* on the left of the rule—the linearized tree pattern it derives is on the right. In the sample, *goal, reg* and *addr* are nonterminals. In addition to nonterminals, the grammar has *operators* of varying arities. In the sample, *Reg, Int, Fetch*, and *Plus* are operators with respective arities of 0, 0, 1, and 2.

The goal of a BURS pattern matcher is to find a least-cost parse of a subject tree for the grammar that reduces to the *goal* nonterminal. Every node in the tree will be labeled with a state that indicates which rule is to be used when that node is to be reduced to a given nonterminal.

3.1 Normal Form Patterns

To simplify the generation of BURS tables, all patterns are put into the *canonical form* introduced in [BDB90]. This form requires that all patterns be of the form "$n \rightarrow m$" where both n and m are nonterminals, or of the form "$n_0 \rightarrow op(n_1, \ldots, n_k)$" where n_i are all nonterminals, $k \geq 0$, and op is an operator. This regular form for rules does not reduce the expressiveness of the grammars—any set of rules not in canonical form can be put into canonical form by introducing new nonterminals. Putting the previous rules into canonical form gives the rules on the right of Figure 1.

4 Algorithm to Generate BURS Tables

The method we use to compute the states and state transition tables is an uncomplicated *work-list* algorithm. This algorithm is outlined below in procedure *Main*(). Initially, the states corresponding to each leaf operator (arity = 0) are computed, and added to the set of known states, *States*, and to the list of states to be processed, *WorkList*. Then, one by one, each new state is removed from *WorkList* and processed. For each operator with arity greater than 0, the state must be examined to determine what transitions are induced by that state when combined with each of the already processed states. These transitions may create new states to be added to the *WorkList*.

```
1   procedure Main()
2       States = ∅
3       WorkList = ∅
4       ComputeLeafStates()
5       while WorkList ≠ ∅ do
6           state = Pop(WorkList)
7           ∀ op ∈ Operators do
8               ComputeTransitions(op, state)
9           end ∀
10      end while
11  end procedure
```

4.1 Data Structures Used to Generate BURS Tables

The set of known states, *States*, is a table that maintains a one-to-one mapping from individual states to non-negative integers. These integers are used as indices into state transition tables via index maps.

States in a BURS code generator encode three pieces of information at any node in a subject tree: the nonterminals derived from patterns that match a rule at that node, the relative costs of those nonterminals, and which rules generated each nonterminal (at a minimal cost). Such triples are called *items*, and a collection of items describing a particular state is called an *itemset*. Itemsets are implemented as arrays of $\{cost, rule\}$ pairs that are indexed by a nonterminal. Itemsets are, therefore, states. A cost of infinity (∞) indicates that, in this state, no rule derives the given nonterminal. The empty state ($\emptyset$) has all costs equal to infinity. The relative costs are called *delta costs* and are always normalized so that the nonterminal with the lowest cost derivation has a delta cost of 0. Costs within an itemset are normalized by the routine *NormalizeCosts*() below.

```
1   procedure NormalizeCosts(state)
2       delta = min_∀i {state[i].cost}
3       ∀ n ∈ Nonterminals do
4           state[n].cost = state[n].cost − delta
5       end ∀
6   end procedure
```

4.2 Chain Rules

Itemsets are computed in a two-step process. Initially, nonterminals derived from the direct application of rules of the form "$n \rightarrow op(\ldots)$" are generated (by procedure *ComputeTransitions*()). Next, it is necessary to compute the closure of this set by finding all applicable *chain rules*. Chain rules are rules of the form "$n \rightarrow m$" where both n and m are nonterminals. These rules may introduce new nonterminals into an itemset, or they may introduce cheaper ways of deriving nonterminals already in the set. Finding the closure of the set is done by iteratively trying all the chain rules and repeatedly applying those that add new or cheaper nonterminals, until no changes are made. *Closure*() below implements this procedure. Because all costs are non-negative, and because a change is made only if a strictly less expensive derivation is found, this process must terminate.

One nonterminal may be derived from another by zero or more chain rule applications—and the least cost derivation is denoted "$n \stackrel{*}{\Rightarrow} m$." The cost of such least cost derivations is denoted by "$Cost(n \stackrel{*}{\Rightarrow} m)$" and these values can be computed efficiently using a shortest path algorithm.

```
1   procedure Closure(state)
2       repeat
3           ∀ r : n → m such that m ∈ Nonterminals do
4               cost = r.cost + state[m].cost
5               if cost < state[n].cost then
6                   state[n] = { cost, r }
7               end if
8           end ∀
9       until no changes to state
10  end procedure
```

4.3 Computing States and Transitions

The computation of the states and the state transition tables begins by generating a state for each leaf operator (with arity of 0) in the routine *ComputeLeafStates*(). Once these leaf states have been created, they must be combined as children of each non-leaf operator, and new states will be created. Each new state is added to the *WorkList* and will be subsequently processed to determine the transitions that it induces.

Computing the state to label each leaf is straightforward. Inspection of all rules with a right hand side of the given leaf operator determines which nonterminals are directly generated into the itemset. Normalizing the costs and finding the closure of the itemset completes the computation of the state corresponding to the leaf operator.

```
1   procedure ComputeLeafStates()
2       ∀ leaf ∈ Leaves do
3           state = ∅
4           ∀ r : n → leaf do
5               if r.cost < state[n].cost then
6                   state[n] = { r.cost, r }
7               end if
8           end ∀
9           NormalizeCosts(state)
10          Closure(state)
11          WorkList = Append(WorkList, state)
12          States = States ∪ {state}
13          leaf.state = state
14      end ∀
15  end procedure
```

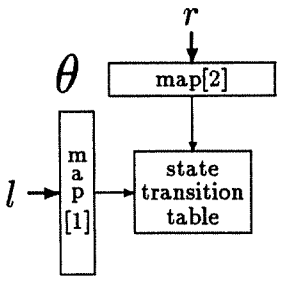

Figure 2: Computing Transitions for $\theta(l, r)$ Using Index Maps.

For each dimension of a non-leaf operator[2], an index map of *representer states* is maintained. Representer states are constructed from an itemset by retaining only those nonterminals that may contribute to a match in the given dimension for the given operator ([Cha87], [BDB90]). Suppose that, for a given grammar, there is no rule with a tree pattern for the binary operator, θ, that has a left child of nonterminal n. In this case, we would project n out of any state when that state is to be examined as a possible left child (in the 1st dimension) of θ.

Project() will retain only those nonterminals in a given state that may be used in determining the transitions that may be induced by that state as a given child of a particular operator. A representer state also discards the *rule* field of each item because that information does not affect transitions (only reductions). For each dimension, d, a table of representer states, $op.reps[d]$, is maintained that encodes a one-to-one mapping between those states and non-negative integers. Each dimension's $op.map[d]$ table maintains a mapping from global states to representer states ($op.map[d][s]$ is the representer state to which s maps in the d^{th} dimension of op.)

Figure 2 illustrates the relationship between index maps and transition tables. Given the states l and r for the children of binary operator θ, an indirection is used to lookup the state transition for the θ node.

```
1   function Project(op, i, state)
2       pState = ∅
3       ∀ n ∈ Nonterminals do
4           if ∃ r : m → op(n1,...,ni−1, n, ni+1,...,nop.arity) then
5               // Nonterminal n may be used in the ith dimension of op.
6               pState[n].cost = state[n].cost
7           end if
8       end ∀
9       NormalizeCosts(pState)
10      return pState
11  end procedure
```

Transition tables are computed based on representer states, not on the original states. This provides a tremendous compaction in transition table size because many states may map to the same representer state. At tree-matching time the cost of using this technique is the extra level of indirection necessary to compute transitions. For each new state, *ComputeTransitions*() (given below) is used to find all the transitions that this state may induce when used in combination with other known states for a given operator.

Once a representer state has been constructed, it is checked to see if it has already been processed. If the representer state has been previously processed, then no additional work must be done with this

[2]Each operator of arity n has a transition table of n dimensions.

state for this dimension. If the representer state is new, the transition table must be extended along the given dimension for all possible combinations of the representer states of other dimensions (along with this representer state). This is done by generating all such combinations and then searching for rules that can apply to the given combinations. Once these rules have been applied, the delta costs are normalized, and the itemset is closed. If the generated state has not been previously seen, then it is added to *States* and *WorkList*.

(The postponement of *Closure*() until after the check for the state's existence in *States* is an optimization justified in §5.4. *Trim*(), the routine responsible for reducing the number of states produced, is discussed in §4.4.)

```
1   procedure ComputeTransitions(op, state)
2       ∀ i ∈ 1..op.arity do
3           pState = Project(op, i, state)
4           op.map[i][state] = pState
5           if pState ∉ op.reps[i] then
6               op.reps[i] = op.reps[i] ∪ {pState}
7               ∀ (s1, ..., si−1, pState, si+1, ..., sop.arity) such that each sj ∈ op.reps[j] do
8                   result = ∅
9                   ∀ r : n → op(m1,...,mop.arity) do
10                      cost = r.cost + pState[mi].cost + Σj≠i sj[mj].cost
11                      if cost < result[n].cost then
12                          result[n] = { cost, r }
13                      end if
14                  end ∀
15                  Trim(result)
16                  NormalizeCosts(result)
17                  if result ∉ States then
18                      Closure(result)
19                      WorkList = Append(WorkList, result)
20                      States = States ∪ {result}
21                  end if
22                  op.transition[s1, ..., si−1, pState, si+1, ..., sop.arity] = result
23              end ∀
24          end if
25      end ∀
26  end procedure
```

4.4 State Trimming

Many of the states created by the *ComputeTransitions*() are nearly identical. The state-generation algorithm can be made to run faster if it can increase the likelihood that two created states will, in fact, be identical. Two states can often be made identical by trimming *unessential* nonterminals from the itemset. A nonterminal is unessential (in a particular state) if it can be proven that it will never be needed to produce a least-cost cover of any subject tree. Henry devised two *ad hoc* techniques, "sibling," and "demand" trimming [Hen89], to identify when one "{ *cost, rule* }" item (representing a nonterminal) can be safely removed from a state because another item subsumes it.

4.4.1 Triangle Trimming

By generalizing Henry's trimming techniques, we have developed a new technique, *triangle trimming*, for safely removing unessential nonterminals from an itemset. Triangle trimming considers all pairs

of nonterminals in a particular itemset and determines if, for each pair, given their respective costs, one of the nonterminals can be removed. A nonterminal can be removed if, in all dimensions of all rules where it is applicable, another nonterminal can be used (in a possibly different rule) to generate the same resulting nonterminal at no greater cost. Informally, a nonterminal, i, can be removed from an itemset if it can be shown that everywhere i can lead to a pattern match, another nonterminal, j, in the itemset can also lead to a comparable pattern match at no greater cost.

Determining if j subsumes i requires comparisons that have a *triangular* shape (see Figure 3). For a given operator, θ, and in a given dimension, d, two (possibly identical) rules must be found such that both rules represent patterns for θ, and one rule, r, can employ i as its d^{th} child, and the other rule, t, can employ j as its d^{th} child. (It is not necessary that these rules use i and j directly—they may use nonterminals that are derived from i and j via chain rules.)

Since rule r reduces to nonterminal n_r, it must be shown that t can also produce n_r at no greater cost. We, therefore, start by assuming that rule r has matched. From this it can be determined if rule t can also match. Rule t can also match if its children in dimensions other than d can be derived via chain rules from the corresponding children of rule r. (All we are assuming is that r matches, therefore all we may assume in determining if $p_{t,k}$ exists for a match of rule t is whether $p_{t,k}$ may derived from $p_{r,k}$ via chain rules.)

Figure 3 shows how i and j, and the rules r and t must relate for j to subsume i. Once rule r is found to use i to derive n_r, a (possibly different) rule must be found that can employ j and can also derive n_r. Notice that for any rule r that employs i, it is only necessary to find one such rule t employing j for j to subsume i.

Subsumption is based not only on feasibility, but also on costs. A nonterminal cannot be removed if its removal would force more expensive reductions to be found than had it been retained. For the pair of rules, r and t, in Figure 3, it is possible to remove i from the itemset containing j if the inequality in Figure 4 holds. The cost of using r is the sum of the cost of i, the cost of deriving $p_{r,d}$ from i, and the cost of r. Since our premise is only that rule r matches and that i and j are present in some itemset, the computation of the cost of using t with j to indirectly produce n_r will require not only the costs of t, j, and $p_{t,d} \stackrel{*}{\Rightarrow} j$, but will also require the costs of deriving the other $p_{t,k}$ from $p_{r,k}$ and the cost of deriving n_r from n_t.

The inequality in Figure 4 is the basis for finding the *minimal* cost difference between two nonterminals to allow one of them to be removed for a given rule. In general, to safely remove i, it is necessary to examine *all* contexts in which i can be used and find the cost difference that is sufficient to guarantee that i can be removed based on the relative costs of i and j. The routine, *Triangle*(), calculates this minimal difference for any pair of nonterminals. (When it is impossible for nonterminal j to be used in place of i, regardless of their respective costs, *Triangle*() returns ∞.)

```
// Compute C, such that if state[i].cost ≥ state[j].cost + C then i can safely be removed from state.
1   function Triangle(i, j)
2       if i = Goal then
3           return ∞        // Do not remove the goal nonterminal, tempting as it may be.
4       end if
5       Max = −∞
6       ∀ n ∈ Nonterminals−{i} do
7           if Max < Cost(n ⇒* j) − Cost(n ⇒* i) then
8               Max = Cost(n ⇒* j) − Cost(n ⇒* i)
9           end if
10      end ∀
11      ∀ op ∈ Operators do
12          ∀ d ∈ 1..op.arity do
13              ∀ r : n_r → op(p_{r,1},...,p_{r,op.arity}) do
14                  C_i = Cost(p_{r,d} ⇒* i)
15                  if C_i < ∞ then
```

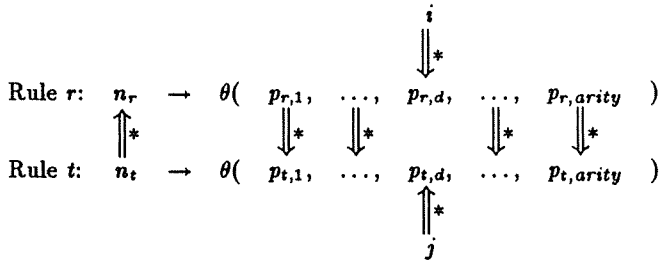

Figure 3: Triangle Trimming Relationship (for j to subsume i)

$$state[i].cost + r.cost + Cost(p_{r,d} \stackrel{*}{\Rightarrow} i)$$
$$\geq state[j].cost + t.cost + Cost(n_r \stackrel{*}{\Rightarrow} n_t) + Cost(p_{t,d} \stackrel{*}{\Rightarrow} j) + \sum_{k \neq d} Cost(p_{t,k} \stackrel{*}{\Rightarrow} p_{r,k})$$

Figure 4: Inequality that must hold for i to be removed if j is present.

```
16                          LocalMin = ∞
17                          ∀ t : n_t → op(p_{t,1},...,p_{t,op.arity}) do
18                              C_{r,t} = Cost(n_r ⇒* n_t)
19                              C_j = Cost(p_{t,d} ⇒* j)
20                              C_k = Σ_{k≠d} Cost(p_{t,k} ⇒* p_{r,k})
21                              C = C_{r,t} + C_j + C_k + t.cost − r.cost − C_i
22                              if C < LocalMin then
23                                  LocalMin = C
24                          end ∀
25                          if LocalMin > Max then
26                              Max = LocalMin
27                          end if
28                      end if
29                  end ∀
30              end ∀
31          end ∀
32          return Max
33      end procedure
```

4.4.2 Chain Rule Trimming

Two states are identical if they represent the same nonterminals at the same costs with each respective nonterminal generated by the same rule. Triangle trimming removes nonterminals from states whenever possible, thereby eliminating the possibility that two states differ on the particular costs or rules involving those nonterminals. To further minimize the number of states, it is necessary to bias the algorithm towards using the same rules whenever possible. We have chosen to bias the algorithm towards using chain rules whenever possible to increase the likelihood that two states will have used the same rules to derive a given nonterminal. This bias can be forced by removing nonterminal entries from an itemset prior to closure when it can be determined that *Closure*() will restore those nonterminals at an equal or lesser cost using chain rules.

The routine, *Trim*(), uses both triangle and chain rule trimming to prune nonterminals from itemsets so that they will be more likely to be identical, thereby reducing the size of the generated tables and the table generation time.

```
1   procedure Trim(state)
2       ∀ n ∈ state do
3           ∀ m ∈ state (m≠ n) do
4               C = Cost(n ⇒* m)
5               if state[n].cost ≥ state[m].cost + C then
6                   state[n] = { ∞, ⊥ } // Remove n from state.
7               end if
8           end ∀
9       end ∀
10      ∀ n ∈ state do
11          ∀ m ∈ state (m≠ n) do
12              C = Triangle(n, m)
13              if state[n].cost ≥ state[m].cost + C then
14                  state[n] = { ∞, ⊥ } // Remove n from state.
15              end if
16          end ∀
17      end ∀
18  end procedure
```

5 Speed Optimizing Techniques

The previous routines provide many opportunities for speed optimization. Some of the improvements are general techniques not specific to BURS table generation; other improvements rely on subtle knowledge of the problem of BURS table generation.

5.1 Attempt Cheaper Alternatives First

It may appear that the two sets of nested loops in *Trim*() could be jammed into a single pair of nested loops for improved efficiency. Both loops have the intended side-effect of removing nonterminals from the states. Since the loops iterate over only the nonterminals that remain in the state, the second set of loops will normally iterate fewer times that the first set. This should speed the program because triangle trimming is an expensive operation relative to chain rule trimming. It is much more efficient to try to remove all possible nonterminals via chain rule trimming and then attempt triangle trimming only on the remaining nonterminals.

5.2 Precompute Values

In the previous routines, many situations exist where values can be computed once and used many times. For instance, *Project*() requires the knowledge of which nonterminals can appear in the i^{th} dimension of operator *op*. Because this list is invariant for a given rule set, it can be computed once and used repeatedly.

Efficiency is also enhanced if the list of rules is partitioned by the operator of the pattern, so that *ComputeTransitions*() will only iterate over the list of applicable rules.

The cost of transitive closure rules ($Cost(n \stackrel{*}{\Rightarrow} m)$) is precomputed advantageously since it is used often by *Trim*() and *Triangle*().

5.3 Lazy Computations

There are $O(N^2)$ possible pairs of 2 nonterminals that may be used in a call to *Triangle*(), but in practice only very few pairs are ever used. Our original implementation precomputed the results of calling *Triangle*() with all possible combinations of nonterminals and then used table lookup for these values. Using this strategy, *Triangle*() consumed over 75% of the execution time generating tables for a VAX grammar. With 179 nonterminals in the (canonical form) grammar, *Triangle*() was

called 32041 times, but fewer than 1000 of those values were ever referenced! Changing the program to compute those values by need increased the speed tremendously. Once computed, these values are cached for subsequent calls with the same arguments.

5.4 Defer Closure

If two itemsets are equal before closure, then they must be equal after closure. Because two itemsets are chain-rule trimmed before closure, it is also the case that if two itemsets are equal after closure, they must have been equal before closure. By maintaining both pre-closure and post-closure copies of an itemset in a table, we can check for the existence of an itemset in the table by comparing their pre-closure representations. This allows the closure computation to be deferred until it is known that the state is indeed new and must be added to the table.

5.5 Itemset Equivalence

Determining whether an itemset is already in a table of states is an expensive operation, and this test is done for every entry in every transition table. The integer subset 68000 grammar required over 425,000 calls to determine itemset equivalence. Making itemset equivalence testing efficient is extremely important. For two itemsets to be equal, they must be equal for all of their items. Fortunately, two observations make testing for equivalence much more efficient: two itemsets created as members of transition tables for different operators can never be equal, and for any given operator it is only necessary to compare the entries corresponding to the left-hand sides of the rules for that operator.

By keeping a reference to the generating operator as part of an itemset's representation, many itemsets can be determined to be unequal by recognizing that those entries differ. Should those entries be the same, it is only necessary to check that the nonterminal entries for the relevant nonterminals are equal for both itemsets. This check must be done after the states have been trimmed.

The same routines are used to implement the global *States* table, and each of the local *op.reps*[] tables. These tables are implemented as hash tables. Computing the hash function is also made more efficient by examining only the relevant nonterminals.

Calling *NormalizeCosts*() after *Trim*(), but before *Closure*(), allows it to limit the nonterminals it must inspect. Again, the same nonterminals that are relevant to determining itemset equivalence are those that must be normalized prior to a call to *Closure*().

5.6 Specialize Memory Allocation

Our program allocates and deallocates an enormous amount of memory during the computation of the itemsets and transition tables. The primary source of allocation and deallocation of memory in the algorithm is the tentative allocation of itemsets by *ComputeTransitions*() and *Project*(). Only after the itemset is allocated and computed can it be determined if an equivalent state has already been seen, thereby allowing the deallocation of the itemset. Redundant itemsets really *must* be deallocated—for a 68000 grammar the program computed over 100,000 redundant itemsets.

Fortunately, knowledge of the the allocation/deallocation pattern of particular data can lead to very efficient memory management [Han90]. This is the case with itemsets. Itemsets, after allocation, are computed and then either retained forever or immediately released. It can never be the case, therefore, that two itemset deallocations occur sequentially without an intervening allocation. This allows the creation of specialized deallocation and allocation routines for itemsets. The deallocation routine simply maintains a reference to the last discarded itemset, and does not return the space to the heap. Allocation checks this reference, and if the reference is not null, it returns the reference to the previously deallocated value (and clears the reference); only if the reference is null does the allocator request space from the heap.

5.7 Minimize space

On a machine without enormous amounts of RAM, it is important to avoid over-allocating memory and thrashing. The single biggest user of memory is the itemset representation for all of the computed states. It is important to keep the representation of itemsets as small as possible. This can be done, in part, by minimizing the number of nonterminals in the canonical form grammar. A naive translation of a grammar into canonical form may produce too many nonterminals if it creates different nonterminals that represent identical patterns. It is important (and easy) to reuse previously created nonterminals as much as possible.

6 Unprofitable Optimizations

Two additional techniques were not implemented because either the speed-up did not merit the additional complexity, or because the resulting code would only reduce the number of states, without also speeding up the code.

6.1 Closure Speedup

Because least-cost transitive chain rules are precomputed for use by *Trim*(), they are available for speeding up the *Closure*() routine. *Closure*(), however, represents less than 4% of the execution time of the program, and using these transitive rules only speeds that routine by 10-20%.

6.2 Post-pass State Minimization

It is possible to further eliminate states after they and the transition tables have been generated by isolating and removing states that differ only in the respective costs of each constituent nonterminal. State minimization for BURS is similar to DFA state minimization. Because state minimization is a post-pass, it cannot make the program faster—it must make it slower.[3] We decided the space savings was not worth the additional complexity or time and, therefore, did not attempt to add a state minimization pass.

7 Implementation Results

Our algorithm has been implemented in ANSI C. The input has two parts: a description of the operators (including the arity and identifying value of each), and a list of grammar rules. The operators are limited to being nullary (leaf), unary, or binary. (The arity was limited because the intended application required only nullary, unary, and binary operators.) Each rule includes an arbitrarily complex pattern, the nonterminal the pattern derives, its cost, and a unique external rule number (for identification). The front end of the table generator puts the rules into canonical form.

As output the program creates C routines and tables for labeling and reducing a subject tree. The program can output either a simple table driven tree labeler and reducer, or a hard-coded labeler and reducer. The hard-coded routines incorporate the time and space saving techniques in [FH91b].

The entire program is under 4000 lines of code that splits evenly between table generation routines and input/output routines. Figure 5 gives the number of lines of code used to implement the table generator.

Our program runs quickly on both simple and complex inputs. We compare our system to Henry's table generator (his system was derived from the CodeGen system [Hen89].) His system consists of over 20,000 lines of C code. It is not clear, however, how much of this code is a direct consequence of algorithm design, and how much is an indirect consequence of the fact that his BURS system was derived from the much bigger CodeGen distribution.

Figure 6 gives a description of 4 sample input grammars and the execution times for each system on each grammar. The first two grammars (used to generate code generators for `lcc` [FH91a]) are

[3] Henry [Hen89] found that the additional time for the post-pass was negligible (< 1%) in his system.

Function	Lines (C/Yacc)
Table Generation	1981
Front End	633
Table Output	1345
Total	3959

Figure 5: Code size for our BURS table generator

Grammar Description			Henry's System		Our System		Ratio
Machine	#Rules	#Nonterms	#States	Generation Time (seconds)	#States	Generation Time (seconds)	
vax	291	48	1017	467.7	1015	14.4	32
MIPS	136	9	125	21.4	125	0.6	36
vax.bwl	524	179	493	146.8	586	15.5	9
mot.bwl	462	80	499	251.5	838	14.4	14

Figure 6: Timings

for the VAX and the MIPS R3000 RISC processor. Two others that were developed as part of the CodeGen project are integer (byte, word, and long) subsets of the VAX and Motorola 68000 processors. The timings were taken on a DECstation 5000 with 96Mb of RAM.[4]

The differences in the number of generated states between the two systems for the CodeGen grammars can be attributed to the presence of a state minimization post-pass in Henry's system that is not present in our system.

It should be noted that if triangle trimming is *disabled*, the number of states generated and the running times are about 100–200% higher than those reported here.

8 Conclusion

The algorithm presented is a simple and efficient method of producing BURS tables. To the best of our knowledge our system is significantly faster than any other BURS system that does aggressive state trimming. The prototype implementation required fewer than 2000 lines of C code for producing the BURS automata. It was able produce these tables over 30 times more quickly than the previous "state of the art" optimizing system. Our system does not sacrifice table compaction optimizations to achieve this speed—to the contrary, the compaction techniques increase the overall speed of the implementation by reducing the number of states that must be examined.

The algorithm employs only simple data structures and routines to generate these tables quickly. We believe that, to a large degree, this design simplicity increases efficiency. To further increase speed, optimizations that exploit the specific nature of BURS table generation were isolated and are described here.

To reduce the number of states created a new technique of trimming states, triangle trimming, has been developed to isolate nonterminals that can be removed from a state. This trimming provides a many-fold reduction in the number of states and a commensurate speed-up in table generation.

9 Acknowledgements

We would like to thank Robert Henry for making his system available for comparison and for explaining much of his early work in BURS table generation.

[4] The timings are *more* favorable towards our implementation on machines with limited amounts of RAM.

References

[AGT89] Alfred V. Aho, Mahedevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[BDB90] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[Cha87] David R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 168–177, 1987.

[FH91a] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(1), September 1991.

[FH91b] Christopher W. Fraser and Robert R. Henry. Hard-coding bottom-up code generation tables to save time and space. *Software—Practice and Experience*, 21(1):1–2, January 1991.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.

[Hen89] Robert R. Henry. Encoding optimal pattern selection in a table-driven bottom-up tree-pattern matcher. Technical Report 89-02-04, University of Washington, 1989.

[PL88] Eduardo Pelegri-Llopart. *Rewrite Systems, Pattern Matching, and Code Generation.* Phd Thesis, Technical Report UCB/CSD 88/423, Computer Science Division, University of California, Berkeley, 1988.

[PLG88] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Proceedings of the 15th Annual Symposium on Principles of Programming Languages*, pages 294–308, 1988.