Abstract

We present an improvement on an algorithm for generating tables to drive a bottom-up tree pattern matcher. The improved algorithm has time and space requirements roughly comparable to those of LALR parser generators, and orders of magnitude better than those of the unimproved table-generating algorithm. This work may have application to interpreter construction, code generation, optimization and type-checking.

An Improvement to Bottom-up Tree Pattern Matching

David R. Chase*
Department of Computer Science
Rice University
Houston, Texas

1 The problem

Given a finite set of "tree patterns" (trees with wildcards at some leaves), produce off-line an automaton that will locate all "instances" of those patterns occurring in input trees. An instance of a tree pattern is another (sub)tree, exactly the same except where wildcards occur within the pattern. Such instances are known as "matches".

More carefully:

trees A tree is built out of nodes with zero or more children. A child of a node is another tree. A node with no children is called a leaf. All nodes have labels, and all nodes with the same label have the same number of children. In this paper, the number of children associated with a label A is $arity(A)$, and a tree labelled A with children t_1 through t_n (with $n = arity(A)$) is written $A[t_1, \ldots, t_n]$. The set of labels is finite. Leaves are written either as the label alone (e.g., "A") or as a node with no children (e.g., "$A[\,]$") when there might be some confusion between leaves and labels.

patterns A pattern is a tree that may have "wildcard nodes" for some leaves. All wildcard nodes are indistinguishable, and will be written in this paper as "$*$".

matching A pattern p matches a tree t if either

1. $p = *$
2. $t = A[t_1, \ldots, t_n]$ and $p = A[p_1, \ldots, p_n]$ and for $1 \leq i \leq n$, t_i matches p_i (including the case where n is zero; that is, t and p are two leaves with the same label). A "match" is a (sub)tree of t matched by some pattern.

For example, figure 1 shows the two patterns $A[T, *]$ and $B[A[*, *], *]$. The label A has arity 2, $*$ is a wildcard, and T is a leaf. In figure 2 is the tree $A^1[T, B^2[A^1[T, F], A[F, F]]]$ with the roots of subtrees matched by the two patterns indicated with superscripts. Notice that matches may be nested, and that matches may overlap.

Figure 1: Example patterns

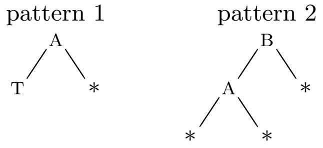
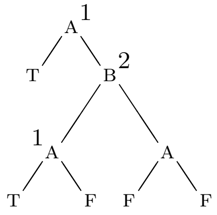


Figure 2: Example tree with matches



2 Background

Hoffmann and O'Donnell describe several techniques for tree pattern matching in [HO82]. One general method given there is known as "bottom-up" pattern matching. It has several advantages over other methods which they describe: the pattern matcher runs in time proportional to the size of the input tree; there are no restrictions on the set of patterns to locate; it can quickly update a tree annotated with pattern matching information after a small change to the tree.

*Supported by IBM and NSF. This report is a revision of a paper by the same name appearing in the 1987 ACM Principles of Programming Languages Conference.

In addition, a bottom-up matcher can be adapted to locate tree patterns in DAGs or general graphs.

The bottom-up algorithm computes sets of patterns that will match nodes of the input tree. The set of patterns to be matched, P, is combined with all subpatterns of patterns in P to form the pattern forest PF. If a non-leaf pattern p is in PF, then all children of p are in PF. For each node of an input tree the bottom-up algorithm produces the set of all patterns in PF that match the subtree rooted at that node. These sets of nodes are called matching sets. The pattern matches within an input tree are exactly those nodes whose matching sets contain some member of P. The bottom-up algorithm appears in figure 3.

Figure 3: Bottom-up matching set algorithm

```
$M(t) =$
    **let** $U = \begin{cases} \{*\} & \text{if } * \text{ is in } PF \\ \{\} & \text{otherwise} \end{cases}$
    **if** $t = A[\,]$
    **then if** $A[\,] \in PF$
        **then** $U \cup \{A[\,]\}$
        **else** $U$
    **else**
        **let** $A[t_1, \ldots, t_n] = t$
        **let** $MSP = M(t_1) \times \cdots \times M(t_n)$
        $PF \cap \left( U \cup \bigcup_{(p_1,\ldots,p_n) \in MSP} \{A[p_1, \ldots, p_n]\} \right)$
```

$M(t)$ returns all patterns in PF matching t. For a leaf t, if t is in PF then $M(t)$ is the set $\{t, *\}$ (or $\{t\}$ if $*$ is not in PF). If t is not in PF, then $M(t)$ is $\{*\}$ (or $\{\}$). In either case, $M(t)$ is exactly the set of patterns in PF matching the leaf t.

For a non-leaf tree t, M is applied to each of t's children to yield their matching sets. The product of these sets is used to construct MSP, a set of pattern tuples. Each tuple $(p_1, \ldots, p_n)$ in MSP is an independent combination of patterns that match t's children; together with t's label, these form a new pattern that matches t. Performing this operation for each member of MSP yields a set of patterns, all matching t. This set does not contain $*$; if $*$ is in PF, then union with U adds it to the set. At this point the set of patterns contains every pattern in PF that matches t (this must be so, since every subtree of every non-leaf pattern in PF is also in PF) plus perhaps a few that are not in PF. To remove the extraneous patterns, the set is intersected with PF to yield exactly those patterns in PF that match t. In the rest of the paper I will use the term *tree-set building* for the process of forming a matching set of patterns from a label and a tuple of pattern sets. Note that "forming a matching set" includes the union with U and intersection with PF. The **else** clause of M then becomes, "build a tree-set from the tuple $(M_1, \ldots, M_n)$."

2.1 Table-driven version

PF is finite, so its powerset is finite. M always returns some subset of PF, so the powerset of PF contains the range of M. Thus, there is a finite number of matching sets for any pattern forest, so the matching sets may be numbered. Because each label has a fixed arity, it is possible to create a table with dimension equal to the arity, and to assign to each entry the number for the tree-set built from the table's label and the matching set tuple corresponding to the entry's coordinates in the table. Leaf labels are associated with the numbers for their matching sets; these may be thought of as tables with dimension zero.

Given a numbering for the matching sets and tables for the labels, all set operations can be removed from the bottom-up algorithm. The new algorithm is shown in figure 4. For a leaf node, it simply returns the corresponding matching set's number. For an interior node, it recursively finds the numbers for the node's children and uses the resulting tuple as an index into the node's label's table.

Figure 4: Table lookup bottom-up matching set algorithm

```
$N(t) =$
    **if** $t = A[\,]$
    **then** $\tau_A$
    **else**
        **let** $A[t_1, \ldots, t_n] = t$
        $\tau_A(N(t_1), \ldots, N(t_n))$
```

In theory, generating all possible matching sets is easy. An algorithm to do this appears in figure 5. It iteratively produces matching sets containing taller and taller patterns until it has produced all possible matching sets. The input to this algorithm is the pattern forest PF and the set of node labels L.

The first step initializes U to the set of patterns in PF that will match every node in the tree. If $*$ is in PF, then $*$ is in U; otherwise, U is empty. The second step of the algorithm builds $\mathcal{R}_0$, the set of possible matching sets of patterns with height equal to zero. These patterns can only be leaves, so all possible matching sets of leaves are generated here[1].

[1] Notice that sets like $\{A[\,], B[\,], *\}$ are not matching sets because there is no tree that every pattern in that set will

Figure 5: Algorithm to generate matching sets

```
let $U = \begin{cases} \{*\} & \text{if } * \text{ is in } PF \\ \{\} & \text{otherwise} \end{cases}$

$\mathcal{R}_0 \leftarrow \{U\} \cup \bigcup_{\substack{A \in L \wedge \\ arity(A)=0 \wedge \\ A[\,] \in PF}} \{U \cup \{A[\,]\}\}$

repeat
    $\mathcal{R}_{i+1} \leftarrow \mathcal{R}_i \cup \bigcup_{A \in L} \left( \bigcup_{\substack{MSP = (R_1 \times \cdots \times R_n) \\ (R_1,\ldots,R_n) \in \mathcal{R}_i^n}} \{R\} \right)$

    where $R = PF \cap \left( U \cup \bigcup_{(p_1,\ldots,p_n) \in MSP} \{A[p_1,\ldots,p_n]\} \right)$

until $\mathcal{R}_{i+1} = \mathcal{R}_i$

$\mathcal{R} \leftarrow \mathcal{R}_i$
```

In subsequent steps the algorithm computes matching sets containing larger patterns. At each iteration of the loop the existing set of matching sets is used to form, for each label A, a product set of n-tuples of matching sets. Each element of this product set is an independent combination of sets containing patterns that might match children of a node labelled A.

For each element of the set of set tuples ($\mathcal{R}_i^n$), form a product set *MSP* containing tuples of patterns. Each one of these tuples contains patterns that might match children of a node labelled A; by using the child-matching patterns as children of a node labelled A, a larger pattern is built. Performing this operation for each member of *MSP* produces a set of patterns that match a node labelled A if its children are matched by the patterns in R_1 through R_n, respectively.

The wildcard is not in the set, so it is added, since it will match any tree. Intersection with *PF* limits the set so that it contains only those matching patterns that are also in *PF*. Repeating this process generates more and more matching sets until all possible matching sets are formed. The algorithm terminates after h iterations of the loop, where h is the height of the tallest pattern in *PF*.

2.2 Correctness

Any set returned by M is in $\mathcal{R}$. This is clearly true for leaves. Inductively, assume that this is true for trees with height less than h. Given a tree $t = A[t_1, \ldots, t_n]$ with height h, the sets $M(t_1)$ through $M(t_n)$ are contained in $\mathcal{R}$. $M(t)$ is the set of patterns formed by using patterns in the $M(t_j)$ as children of a pattern with root label A, adding U, and intersecting with *PF*. However, $\mathcal{R}$ contains the same set of patterns because it contains a set formed in exactly the same way from the same sets. So, every set of patterns returned by M is in $\mathcal{R}$.

Conversely, for every set R in $\mathcal{R}$ there is some tree t such that $M(t) = R$. This is clearly true when R contains only leaves; either $R = U \cup \{A[\,]\}$ or $R = U$. Inductively, assume that this is true for R containing patterns with height less than h. Given a set of patterns R' in $\mathcal{R}$ with height less than or equal to h, there exist sets of patterns R_1 through R_n with height less than h and a label A such that their combination, union with U and intersection with *PF* are equal to R'. By assumption there are trees t_1 through t_n such that $M(t_j) = R_j$. $M(A[t_1, \ldots, t_n])$ is formed in exactly the same way as R' from the same sets, so they are equal.

2.3 Performance of the table generator

In the worst case this algorithm takes time and space exponential in the number of patterns because the number of matching sets can be exponential in the number of patterns. The algorithm is also exponential in the maximum arity of the node labels. In practice, however, the number of matching sets is nowhere near the exponential worst case, and for typical applications label arities are usually quite small (2 or 3). Unfortunately, even a modest number of matching sets and an arity of two can lead to huge tables with size proportional to $|\mathcal{R}|^2$, making the algorithm unpractical.

O'Donnell discusses some attempts to solve this problem in [O'D85], but concludes that efforts so far have been ad hoc or so complex as to be untrustworthy. If the set of patterns is guaranteed to be "simple" (containing no "independent" patterns) the number of matching sets is equal to the number of patterns ([HO82]), but this restriction excludes many interesting pattern sets. Hatcher and Christopher ([HC86]) describe a method for use of a bottom-up matcher in code generation; in some situations their method discards information based on pattern costs, and in

match. Matching sets must also contain *every* pattern in *PF* that matches; if $*$ is in *PF*, $\{A[\,]\}$ is not a matching set.

others it chooses one of two independent patterns. Their method does not discard any information when applied to a “closed template forest” ([Kro75]), but adding enough patterns to a forest to obtain a closed template forest may increase its size exponentially in the same way that the number of matching sets may be exponential in the number of patterns. Note that postprocessing of large tables is not acceptable; often these tables are so large that their construction is not possible on existing computers.

2.4 Example

Figure 6 shows an input set of two patterns. Adding all subpatterns of these patterns to the set gives *PF*, shown in figure 7.

In the first step (not shown), the algorithm creates U, with $U = \{*\}$. Next, it forms the matching sets containing patterns with height equal to zero. These are sets 1, 2 and 3 in figure 8. The first loop iteration builds sets 4, 5 and 6 containing patterns with height less than or equal to 1. After this first iteration $\mathcal{R}_1$ is the set containing matching sets 1 through 6. The next iteration adds matching sets 7 and 8 to form $\mathcal{R}_2$.

Given this particular numbering of the matching sets one can build a table that gives a node's matching set if the matching sets of its children are known. This table is shown in figure 9. The numbers not appearing in the table (2 and 3) are numbers for sets that match the leaves B and C, respectively. Figure 10 shows a tree in which the nodes have been annotated with matching set information.

Notice that the table in figure 9 contains many duplicate rows and columns. It is possible to compress the interior of the table and use “index maps” to convert matching numbers into indices for the compressed table. This is shown in figure 11; here the compressed table is called θ_A and the index maps for the first and second children are called μ_{A1} and μ_{A2}, respectively.

Figure 6: Patterns

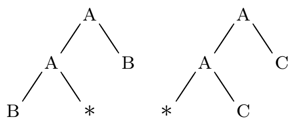


Figure 7: Pattern forest

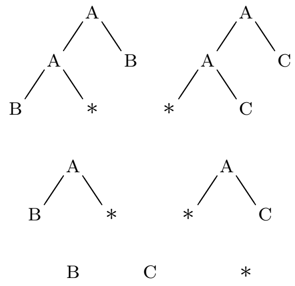


Figure 8: Matching sets

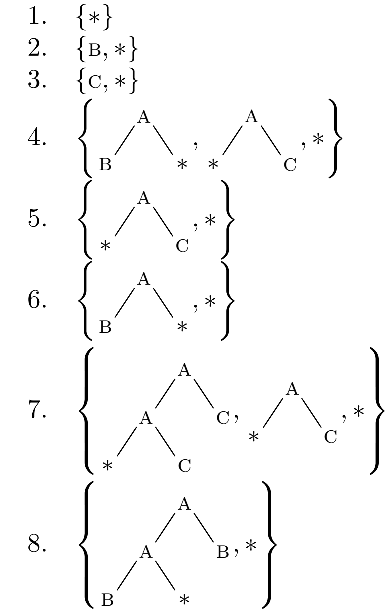


Figure 9: Label A's lookup table

τ_A 2 1	1	2	3	4	5	6	7	8
1	1	1	5	1	1	1	1	1
2	6	6	4	6	6	6	6	6
3	1	1	5	1	1	1	1	1
4	1	8	7	1	1	1	1	1
5	1	1	7	1	1	1	1	1
6	1	8	5	1	1	1	1	1
7	1	1	7	1	1	1	1	1
8	1	1	5	1	1	1	1	1

Figure 10: Application of bottom-up automaton

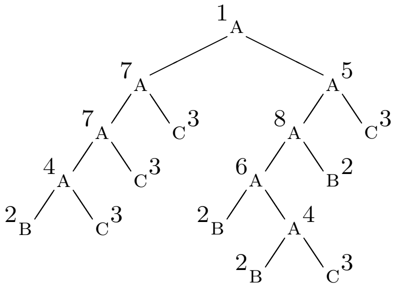


Figure 11: Compressed lookup table

μ_{A2}	1	2	3	4	5	6	7	8
	1	2	3	1	1	1	1	1

μ_{A1}	
1	1
2	2
3	1
4	3
5	4
6	5
7	4
8	1

θ_A 2 1	1	2	3
1	1	1	5
2	6	6	4
3	1	8	7
4	1	1	7
5	1	8	5

Figure 12: Compressed table lookup algorithm

```
$N(t) =$
    if $t = A[\,]$
    then $\theta_A$
    else
        let $A[t_1, \ldots, t_n] = t$
        $\theta_A(\mu_{A1}(N(t_1)), \ldots, \mu_{An}(N(t_n)))$
```

3 Improvements

A table τ_A is said to contain a *duplicate j-subtable* if there exist j, x_j, and y_j, $x_j \neq y_j$, such that

$$\forall(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n), \quad \tau_A(x_1, \ldots, x_j, \ldots, x_n) = \tau_A(x_1, \ldots, y_j, \ldots, x_n) \qquad (1)$$

The tables (naively) generated from the matching sets often contain many duplicate subtables[2]. If the duplicate subtables are known, it is possible to introduce, for each label and child of the label, index maps that map matching sets with equal subtables into a single set. Figure 12 shows an example of the resulting lookup algorithm (the index map for label A and child j is μ_{Aj}; θ_A is the compressed version of τ_A), and figure 11 shows tables suitable for use with this algorithm. Cheng, Omdahl and Strawn ([COS82]) and Hatcher and Christopher ([HC86]) describe identical compression schemes, though they use them in matchers that locate slightly different classes of patterns and generate the tables and index maps in different ways.

As noted above, uncompressed tables may be so large that they cannot be generated, even if the compressed tables would be of manageable size. However, it is possible to compress the tables before they are ever generated by examining the matching sets. Define P_{Aj} to be the set of patterns that appear as the jth child of patterns with label A in PF. Because PF is closed under taking subtrees, P_{Aj} is contained in PF.

Consider the calculation of the table entry for a given tuple of sets' numbers. If the sets are $(R_1, \ldots, R_n)$ and the label is A, then the set (number) R assigned to that table entry is the result of

$$R = PF \cap \left(U \cup \bigcup_{(p_1,\ldots,p_n) \in MSP} \{A[p_1, \ldots, p_n]\} \right)$$
$$\text{where } MSP = R_1 \times \cdots \times R_j \times \cdots \times R_n \qquad (2)$$

Now consider replacing the matching set R_j with the set $R_j \cap P_{Aj}$ in (2). This cannot change the value of R, because the only patterns missing from $R_j \cap P_{Aj}$ are those that do not appear as the jth child of a node labelled A in PF, and thus any pattern tuples removed from MSP will not produce a pattern that is in PF. Note that this holds in general; whatever the value of j, R is unchanged, and whatever the values of the other matching sets, R is unchanged.

If there is another set R'_j such that $R'_j \cap P_{Aj} = R_j \cap P_{Aj}$, then both R'_j and R_j will always produce equal matching sets when substituted for one another in (2), since in either case the result is the same as that obtained by substituting $R_j \cap P_{Aj}$. This observation is the key to easier table compression and other improvements to the algorithm. Intersection with P_{Aj} produces an equivalence relation over $\mathcal{R}$; two sets of patterns R and R' are said to be Aj-equivalent if $R \cap P_{Aj} = R' \cap P_{Aj}$. For each A and j, the set of matching sets ($\mathcal{R}$) is partitioned up into a set of sets of equivalent matching sets. This is cumbersome; it is much easier to represent an equivalence class of sets with the set of patterns $R \cap P_{Aj}$, where R is any member of the equivalence class. Representing the equivalence class with this one set also makes it

[2] If the tables are two-dimensional, a subtable is a row or a column; if they are three-dimensional, a subtable is a two-dimensional section obtained by holding one coordinate constant.

easier to find the class to which a matching set belongs; intersection with P_{Aj} yields the *representer set* for the class. For a given A, j, and $\mathcal{R}$, the set of representer sets will be called $\mathcal{S}_{Aj}$.

To compress the jth dimension of τ_A, then, it suffices to compute $\mathcal{S}_{Aj}$ and maps from the integers associated with the members of $\mathcal{R}$ to the integers associated with the members of $\mathcal{S}_{Aj}$. These maps are the μ_{Aj} of the lookup algorithm in figure 12. Figure 13 illustrates the relationship between the maps and the various sets. The sets $\mathcal{I}$ and $\mathcal{I}_{Aj}$ are the integers corresponding to $\mathcal{R}$ and $\mathcal{S}_{Aj}$.

Figure 13: Sets and maps for table compression

$$\begin{tikzcd}
 & & & \mathcal{S}_{A3} \arrow[dddddd, "\sigma_{Aj}"] \\
\mathcal{R} \arrow[rrru, "\cap P_{Aj}"] \arrow[rr] \arrow[rd] \arrow[dddd, "\rho"'] & & \mathcal{S}_{A2} \arrow[dddd] & \\
 & \mathcal{S}_{A1} \arrow[dd] & & \\
 & & & \\
 & \mathcal{I}_{A1} \arrow[dr, phantom, "\times"] & & \\
\mathcal{I} \arrow[ru] \arrow[rr] \arrow[rrrd, "\mu_{Aj}"'] & & \mathcal{I}_{A2} \arrow[dr, phantom, "\times"] & \\
 & & & \mathcal{I}_{A3} \arrow[lllu, bend left, "\theta_A"]
\end{tikzcd}$$

4 Further improvements

The previous section showed how it is possible to compress the tables of a bottom-up tree automaton without actually forming the tables. However, the algorithm to build the matching sets still iterates over all the coordinates of the uncompressed table, and this can be very time-consuming. This is unnecessary, since two members of the same Aj-equivalence class will have the same effect when generating patterns in the same way that they have the same effect when computing table entries; it is both possible and desirable to iterate over the coordinates of the compressed tables.

To do this, notice that the sets P_{Aj} depend only upon *PF*. Notice also that the Aj-equivalence classes can be generated and maintained at the same time that new matching sets are produced. For each set of sets of patterns with height less than or equal to i, $\mathcal{R}_i$, there are sets of representer sets $\mathcal{S}_{Aji}$. The equivalence class associated with each representer may grow as more matching sets are discovered, but the representer set is unchanged. With this change, the algorithm iterates over the set $\mathcal{S}_{A1i} \times \cdots \times \mathcal{S}_{Ani}$ instead of over the set $\mathcal{R}_i^n$ when generating matching sets.

This also removes one iteration of the main loop from the algorithm. Using matching sets to form the product set, the algorithm generates the last new matching set in the penultimate iteration. However, the set of matching sets has changed, so the algorithm must iterate once more. Even if it only iterates over the new members of the product set, this can be a large and costly computation. The improved method will form the last *representer set* in its penultimate iteration; in its last iteration it will form the last matching set.

Using representer sets instead of matching sets as tuple elements speeds up tree-set building in another way; each representer set is no larger than the smallest matching set in its equivalence class. The set *MSP* formed from a tuple of representer sets will contain no more pattern tuples than any equivalent set formed from a tuple of matching sets. Since the set of tuples is smaller, forming a matching set from the set of tuples and a label is easier.

The use of representer sets generated by the P_{Aj} improves efficiency in four ways:

1. Table compression can be performed before the tables are ever constructed.

2. The algorithm builds fewer combinations of sets of patterns when searching for new matching sets.

3. The last iteration is avoided.

4. For each combination of sets of patterns, the combination contains fewer pattern tuples because the sets combined are themselves smaller.

The improved algorithm for generating matching sets is shown in figure 14.

5 Optimality result

The tables produced by merging together subtables of Aj-equivalent matching sets cannot be further compressed.

Figure 14: Improved matching set generation

```
$U \leftarrow \begin{cases} \{*\} & \text{if } * \text{ is in } \mathit{PF} \\ \{\} & \text{otherwise} \end{cases}$

generate $P_{Aj}$ for each $A$ and $j$

$\mathcal{R}_0 \leftarrow \{U\} \cup \bigcup_{\substack{A \in L \wedge \\ arity(A)=0 \wedge \\ A[\,] \in \mathit{PF}}} \{U \cup \{A[\,]\}\}$

generate $\mathcal{S}_{Aj0}$ for each $A$ and $j$

repeat
    $\mathcal{R}_{i+1} \leftarrow \mathcal{R}_i \cup \bigcup_{A \in L} \left( \bigcup_{\substack{MSP = (S_1 \times \cdots \times S_n) \\ (S_1,\ldots,S_n) \in \prod_{j=1}^{n} \mathcal{S}_{Aji}}} \{R\} \right)$

    where $R = \mathit{PF} \cap \left( U \cup \bigcup_{(p_1,\ldots,p_n) \in MSP} \{A[p_1,\ldots,p_n]\} \right)$

    and generate $\mathcal{S}_{Aj(i+1)}$ for each $A$ and $j$

until $\forall j, \mathcal{S}_{Aj(i+1)} = \mathcal{S}_{Aji}$

$\mathcal{R} \leftarrow \mathcal{R}_i$
$\mathcal{S}_{Aj} \leftarrow \mathcal{S}_{Aji}$ for each $A$ and $j$
```

To show: given two representer sets X_k and Y_k for distinct Ak-equivalence classes, there exists some $(x_1,\ldots,x_n)$ such that $\theta_A(x_1,\ldots,x_k,\ldots,x_n) \neq \theta_A(x_1,\ldots,y_k,\ldots,x_n)$, where x_j is (in general) the integer corresponding to the representer set X_j.

Proof: It helps the proof to make the correspondence between $\mathcal{R}$ and integers explicit, and to make the correspondences between $\mathcal{S}_{Aj}$ and integers explicit. So, ρ maps the set of matching sets $\mathcal{R}$ to $\{1,\ldots,|\mathcal{R}|\}$ one-to-one and onto, and for each Aj, σ_{Aj} maps $\mathcal{S}_{Aj}$ to $\{1,\ldots,|\mathcal{S}_{Aj}|\}$ one-to-one and onto. The inverse maps ρ^{-1} and σ_{Aj}^{-1} are of course defined. Thus $x_j = \sigma_{Aj}^{-1}(X_j)$. Given these functions, the definition of μ_{Aj} is just $\mu_{Aj}(r) = \sigma_{Aj}(P_{Aj} \cap \rho^{-1}(r))$.

Without loss of generality, assume that there is a pattern p_k, with $p_k \in X_k$ and $p_k \notin Y_k$. Since P_{Ak} contains X_k, the pattern p_k must appear as the kth child of some pattern $p = A[p_1,\ldots,p_n]$ in *PF*.

There must be matching sets R_1 through R_n containing p_1 through p_n. This is so because *PF* is closed under taking subtrees, and because every pattern in *PF* must be contained in at least one matching set[3].

For $1 \leq j \leq n$, choose $x_j = \mu_{Aj}(\rho(R_j))$. The pattern p is in

$$\rho^{-1}(\theta_A(x_1,\ldots,x_k,\ldots,x_n))$$

and is not in

$$\rho^{-1}(\theta_A(x_1,\ldots,y_k,\ldots,x_n))$$

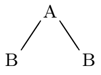
so $\theta_A(x_1,\ldots,x_k,\ldots,x_n)$ and $\theta_A(x_1,\ldots,y_k,\ldots,x_n)$ are not equal.

6 Extension to regular tree patterns

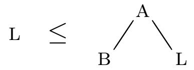
In correspondence, Philip Hatcher ([Hat86]) pointed out how to add "introduced wildcards" to this algorithm. An introduced wildcard is a wildcard that is subsumed by some, but not all patterns in *PF*; the patterns subsuming an introduced wildcard are part of a more general pattern specification. It turns out that this addition produces an extended algorithm capable of building tables for automata that recognize regular tree patterns.

This is an important result; several applications for tree pattern matching make use of introduced wildcards and regular tree patterns. Use of introduced wildcards reduces the size of the pattern and matching sets, and the class of regular tree patterns is larger and more useful than the simpler class of patterns treated above. Regular tree patterns are used by Cheng, Omdahl, and Strawn ([COS82]), appear when Henry's code generation grammars ([Hen84]) are interpreted as tree patterns ([HC86]), and have been proposed for use in analysis and optimization by Jones and Muchnick ([JM81]).

A pattern p *subsumes* another pattern q (written $p \geq q$) if q always matches when p matches. For example, all patterns subsume the wildcard $*$. To extend the algorithm new wildcards and subsumption relations for those wildcards are introduced to the pattern specification. For example, one might specify a linear tree with spine labelled "A" and leaves labelled "B" with

L $\leq$ 

[3] Given a pattern, it is a trivial exercise to construct a tree that the pattern matches. There must be a matching set for that tree, so there must be a matching set containing the pattern.



Wildcard specifications create the *isubsumers* (for *introduced subsumers*) relation between wildcards and patterns. Note that wildcards are also patterns. Form the transitive closure of *isubsumers* to get *isubsumers**. Forming *subsumers** or *subsumers* is not desirable; the structure of matching sets makes this unnecessary[4].

Now consider the reason for union with U in

$$R = PF \cap \left(U \cup \bigcup_{(p_1,\ldots,p_n)\in MSP} \{A[p_1,\ldots,p_n]\} \right)$$

Assuming that $*$ is in U, the union adds to the new set of matching patterns all wildcards ($*$) that match every tree. That is, $*$ is subsumed by every pattern in *PF*, but is not added to the set by forming new patterns from patterns that match children. To ensure that the matching set contains *every* matching pattern for some (potential) tree, $*$ is added.

To handle introduced wildcards, then, an additional step is added to the algorithm. Given R above, add introduced wildcards to form R' by

$$R' = R \cup \bigcup_{\substack{W \in \mathit{wildcards} \wedge \\ R \cap \mathit{isubsumers}^*(W) \neq \emptyset}} \{W\}$$

and use R' in place of R in the rest of the algorithm. Given this change, the rest of the algorithm functions correctly.

7 Implementation

The actual implementation of this algorithm is, of course, much more complicated than the descriptions here.

A bucket hash table mapping trees to integers implements the set *PF*. Bit vectors implement subsets of *PF*. The sets of sets of pattern ($\mathcal{R}$ and $\mathcal{S}_{Aj}$) are implemented with bucket hash tables mapping bit vectors to integers. Both tree and bit vector maps simultaneously maintain their inverse maps. All iterations over sets and set products are of course done over the integers corresponding to the sets and products.

It is not necessary to iterate over all possible representer set products when contructing $\mathcal{R}_{i+1}$ because tuples in $\prod_{j=1}^{n} \mathcal{S}_{Aj(i-1)}$ have already been visited once; a second visit will yield the same results as the first visit. Therefore, the implementation only iterates over members of $\prod_{j=1}^{n} \mathcal{S}_{Aji} - \prod_{j=1}^{n} \mathcal{S}_{Aj(i-1)}$. As new matching sets are discovered, the representer sets are updated. For each label A and index j the height of the tallest subpattern is known; when new matching sets are created containing patterns taller than this no attempt is made to update $\mathcal{S}_{Aj}$ because it will not add any elements.

The compressed tables are constructed as the algorithm iterates over coordinates in $\prod_{j=1}^{n} \mathcal{S}_{Aji}$. Since their final size is unknown, the tables must grow dynamically.

To reduce the number of table expansions, the tables are grown at the beginning of each iteration to be at least as large as the current dimensions of the representer sets. This guarantees that the table will not expand in any dimension more times than the main loop is iterated. Normally, the tables double in size when expanded, but at the last iteration (known in advance because the maximum height of all patterns is known) the tables only grow as large as needed.

In the extended version of the algorithm it is not possible to use height to predict the last iteration because it is not easy to describe the "height" of an introduced wildcard; it depends upon interactions with other patterns. However, it was observed that probes into the vector maps are concentrated on a few elements; an optimization exploiting this reclaimed most of the lost speed.

The implementation avoids generating trees and looking them up in *PF* by precomputing all possible combinations of subtrees for each label, forming the trees, looking them up in *PF* and storing the result in a table. The size of these tables is limited by indexing the jth dimension of A's tree lookup table by numbers less than the size of P_{Aj}. Doing this also requires an alternate representation for the P_{Aj} to avoid searching through bit vectors.

Attention was paid to other information that could be precomputed, and to restricting any transformations on matching sets to newly discovered matching sets. Thus, the implemented algorithm computes *isubsumers** before building any matching sets, and maintains two versions of each matching set. The first version is "as generated" and contains no introduced wildcards; it is used to test for new matching sets. The second version contains introduced wildcards; it is used for constructing sets in $\mathcal{S}_{Aji}$. In general, as much information as possible is computed or cached to speed up the algorithm.

Calculation of the μ_{Aj} index maps is postponed until the results are output to save space; these maps are not needed by the algorithm.

It was determined through profiling that a reasonable amount of time was being spent in memory allocation; to avoid time and space overhead and to

[4] Recall that the matching set $M(t)$ contains all patterns in *PF* matching t; if $p \geq q$ and $p \in M(t)$, then $q \in M(t)$.

insulate the implementation from potentially inefficient memory allocators, allocation of bit vectors and hash buckets is handled by front-end allocators (these structures are not especially dynamic). Where possible, scratch storage is allocated once and re-used.

Most of the optimizations were driven by code profiles produced by *gprof*[GKM82]; profiles obtained by logging memory allocations and measuring hash table behavior were also useful and occasionally surprising. No optimization was applied unless it produced a performance improvement on at least one benchmark.

8 Experimental results

Actual runs of this algorithm have demonstrated (on a few examples) spectacular compression. In fact, this algorithm was discovered as a way around generating impossibly large tables for an interpreter[5]. The table sizes in figure 15 reflect total number of entries stored in the tables independent of the amount of memory used for each entry; typically the index (μ) maps can be stored using one byte per entry, while the tables (θ) require entries large enough to hold the number of matching sets ($|\mathcal{R}|$). The index maps are frequently *very* sparse; in many cases trivial compression techniques should yield substantial reductions in the space used. The timings in figure 16 reflect the time in seconds spent generating all of the maps (ρ, σ, and μ) and all of the tables (θ) on a 16 megabyte Sun-3 with a 16.7 megahertz MC68020 processor, running under Sun UNIX 3.0 and compiled with the -O flag to the C compiler[6].

Figure 15: Table sizes

problem	$\lvert PF\rvert$	$\lvert\mathcal{R}\rvert$	$\sum\lvert\theta_A\rvert$	$\sum\lvert\mu_{Aj}\rvert$
P4	35	65813	77284	131626
P3	18	277	484	554
VAX.BWL.M	377	779	1958	74784
VAX.B–H.M	746	1267	3807	201453
APL-50	133	145	236	10440

Two of the benchmark problems were chosen to create many matching sets for very little input, and three were taken from other work. P3 and P4 are the pathological binary sets of height 3 and 4. A pathological binary pattern set of height n contains (before creating *PF*) 2^n patterns. Each pattern is a full binary tree of height n built out of one variety of binary label. The leaves of each pattern tree i are all wildcards except for the ith leaf; it is some non-wildcard leaf label. The set P2 with binary label A and leaf label B is shown in figure 18. The number of matching sets for Pn is proportional to 2^{2^n} because there is a tree matched by each combination of the 2^n patterns.

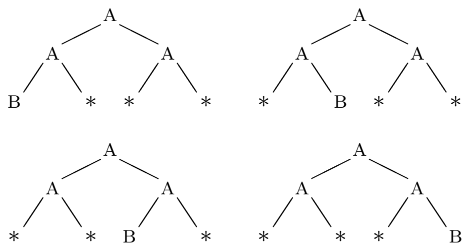
Figure 16: Timings

problem	time
P4	115
P3	0.5
VAX.BWL.M	28
VAX.B–H.M	88
APL-50	4

APL-50 is the set of APL idioms collected by Cheng *et al* for use in idiom-specific optimizations of APL [COS82]. This set includes one regular pattern (*list*), 47 binary and unary labels, and the wildcard $*$. The tallest pattern has height 9. Most of the patterns tend to be right-linear, reflecting the structure of APL.

VAX.BWL.M is one of Robert Henry's grammars for VAX-11[7] machine code generation [Hen84]. This set also uses some regular patterns, but the most common use of introduced wildcards is in pattern factoring. This reduces the size of the pattern forest, improves the performance of the table builder, and makes the machine descriptions easier to read and write. There are 59 unary and binary labels, 23 leaf labels and 63 introduced wildcards. The tallest pattern has height 4.

Figure 18: Pattern set P2



VAX.B–H.M is larger grammar for the VAX-11 (also due to Henry) containing descriptions for the VAX with all its machine datatypes. It contains 97 unary and binary labels, 39 leaf labels, and 109 in-

[5] Bottom-up tree pattern matching was found to be unsuited to this application for other reasons; see [O'D85, p.198] for an explanation.

[6] UNIX is a trademark of AT & T Bell Laboratories; MC68020 is trademark of Motorola Corporation; Sun and Sun-3 are trademarks of Sun Microsystems Inc.

[7] VAX and VAX-11 are trademarks of Digital Equipment Corporation.

Figure 17: Comparison with LALR(1) parser-generators

		VAX.BWL.M		VAX.B–H.M	
method		time	space	time	space
tree matching	plain	110	78.7	333	212.9
table builder	bit-compressed	110	22.9	333	55.1
	run-compressed	110	12.4	333	28.4
LALR parser	*tablegen*	341.2	74.4	826.6	118.4
generators	*yacc*	1190.5	57.5		
	pgs	3217.8	21.2		

troduced wildcards. The tallest pattern has height 4.

The index maps take up most of the space used by the generated tables; fortunately, the index maps provide ample opportunities for compression.

9 Comparison with other work

9.1 Uncompressed tables

This algorithm performs substantially better than algorithms that first construct uncompressed tables. This can be seen by noting that the size of the uncompressed tables is

$$\sum_{A \in labels} |\mathcal{R}|^{arity(A)}$$

For the sample pattern sets above this is usually several orders of magnitude larger than the size of the tables and maps from the compressed algorithm, as shown in figure 19. Table size is measured in number of entries, not number of bytes.

Figure 19: Compressed and uncompressed table sizes

problem	compressed	uncompressed	ratio
P4	208,910	4,331,350,969	20733
P3	1,038	76,729	73
VAX.BWL.M	76,742	22,470,255	292
VAX.B–H.M	205,260	97,969,508	477
APL-50	10,676	528,815	49

9.2 Henry

In [Hen84], Henry generates LALR(1) parsing tables for several machine grammars, including VAX.BWL.M and VAX.BWLFDGH.M. He includes statistics for the parser-generators *tablegen*, *yacc*, and *pgs*. In general, the parser-generators produce somewhat smaller tables but have somewhat larger running times. However, the index maps account for most of the space used by the result of the compressed table builder, and no attempt has been made to compress these in the tables shown in figures 15, 16 and 19. Two naive compression techniques substantially reduce the space used to store index maps. In the first, each map is stored using only as many bits per element (1, 2, 4, 8, 16, or 32) as are needed to express the map's range. In the second, the maps are encoded into two arrays; one contains the "edges" for runs of values, and the other contains the values corresponding to those runs. This method is even more effective than the first at compressing tables, though it is not as fast at run time. These results are shown in figure 17 (the times and speeds for *tablegen*, *yacc*, and *pgs* are taken from [Hen84, fig A2.8, p. 259]. All times are seconds on a VAX-11/750, and space is measured in kilobytes).

9.3 Hatcher and Christopher

Hatcher and Christopher describe a technique in [HC86] that works for a similar class of tree pattern sets annotated with costs. Their method is based on a bottom-up matcher for simple pattern sets (containing no independent pattern pairs) described in [HO82], but has been extended to allow some independence in the pattern set and the use of introduced wildcards. The cost information allows independent patterns; when two independent patterns both match; one is discarded on the basis of its cost. Because their algorithm uses matching patterns instead of matching sets, it produces tables even smaller than the method presented here.

It is appropriate to discard independent matching patterns in code generation, but this is not true in all pattern-matching applications. In some situations their algorithm must discard matching patterns even when it is not justified by cost, or require the user to modify the pattern set. When these things are not acceptable the new algorithm is more appropriate. It also appears that it is possible to modify the method presented here to incorporate cost information and reduce table sizes by discarding some matching sets without significantly increasing the processing time, thus providing the best of both methods.

9.4 Cheng, Omdahl, and Strawn

Cheng, Omdahl, and Strawn construct compressed tables for regular tree pattern forests in [COS82]. Their technique is based on NFA construction, conversion to DFA form, and minimization of the DFA. I believe that the resulting tables are equivalent to those constructed by this paper's method, but their technique is much slower. For example, the method presented here produced tables for 50 APL idioms in 4 seconds on a Sun 3, while their method used 1000 seconds on a NASCO AS/6[8].

10 Still undone

The extended algorithm is no longer optimal, and I suspect that producing optimal tables for regular tree patterns will be much harder. In certain situations it is possible to introduce intersection and complement operations on introduced wildcards, providing a more powerful notation for describing regular tree patterns. It also appears that introducing costs will be a fairly straightforward operation. These things are discussed in the appendix. Though the new algorithm is much more efficient than the uncompressed algorithm, good time and space bounds are even harder to determine.

11 Conclusions

The improved algorithm makes use of a bottom-up tree pattern matcher practical in many situations where it previously was not practical. This is good, because bottom-up pattern matchers are very general, run in time proportional to the size of the input (containing pattern instances) tree, and may be applied to input that is not tree-like.

12 Acknowledgements

Philip Hatcher re-implemented the algorithm from an early description, discovered a bug, provided versions of Henry's grammars suitable for use in tree pattern matching, and pointed out that this algorithm could be extended to handle introduced wildcards; I am very grateful for his help. Robert Henry helped me understand the code generation grammars. Finally, discussion with many people here at Rice helped me refine my description of the algorithm.

References

[COS82] Feng Cheng, Scott Omdahl, and George Strawn. Idiom matching: An optimization technique for an APL compiler. Technical report, Iowa State University, 1982.

[GKM82] S. L. Graham, P. B. Kessler, , and M. K. McKusick. gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[Hat86] Philip Hatcher. correspondence, October 1986.

[HC86] Philip J. Hatcher and Thomas W. Christopher. High quality code generation via bottom-up tree pattern matching. In *Conf. Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, 1986.

[Hen84] Robert Rettig Henry. Graham-glanville code generators. Technical Report UCB/CSD 84/184, University of California,Berkeley, 1984.

[Hen86] Robert Henry. correspondence, October 1986.

[HO82] Christoph Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, pages 68–95, 1982.

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, 1981.

[Kro75] Hans Kron. *Tree Templates and Subtree Transformational Grammars.* PhD thesis, University of California, Santa Cruz, 1975.

[O'D85] Michael J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, 1985.

[PL86] Eduardo Pelegri-Llopart. correspondence, November 1986.

[8] NASCO and AS/6 are probably trademarks of National Advanced Systems.

A Notes on further extensions

A.1 Adding intersection and complement

To do these quickly, it must be possible to compute a partial dependence order on introduced wildcards and patterns. One wildcard depends upon a pattern if it is subsumed by the pattern, the pattern's complement, or an intersection in which the pattern is a member. A topological sort of the partial order gives the correct order to test for subsumption of wildcards when a new set of patterns has been generated. If no partial order exists, then the subsumption relations are cyclic and their solution is too expensive.

It is possible to relax the restriction on the partial order a little bit. Intersection and union are monotonic operations (adding a pattern to a matching set does not exclude one that is already in the set), but complement is not. The dependence graph among wildcards and patterns may contain cycles, but these must be monotonic (complement free). For each matching set the algorithm walks through the dependence graph adding patterns until the process converges. I think it is possible to massage the graph into a form that may be more quickly traversed at set generation time, but I am not certain on the details (collapse along subsuming edges? break cycles?).

A.2 Adding cost information to discard patterns

This is simple enough for patterns that do not appear as subtrees of other patterns, but more difficult for subpatterns. If two patterns p and q satisfy

1. $cost(p) \leq cost(q)$
2. for every label A and child index j, $p \notin P_{Aj}$
3. for every label A and child index j, $q \notin P_{Aj}$

then whenever p and q both match, q may be discarded from the matching set.

For subpatterns the problem is more difficult because subpatterns may appear in different contexts. Suppose that costs have been assigned to the patterns in P3 in the way shown in figure 20. Any two height two patterns (one b and three $*$'s) must both remain in a matching set because one pattern is cheaper in the left-son-of-root context and the other pattern is cheaper in the right-son-of-root context. However, given *three* height two patterns it is always possible to discard one. It seems that the correct, though perhaps expensive, approach is to discard patterns in the intersection over contexts of discarded patterns.

Figure 20: P3 with costs

pattern leaves	cost
B*** ****	1
*B** ****	2
B* **	3
B *	4
**** B***	8
**** *B**	7
**** **B*	6
**** ***B	5

To do this, define a context as a pair $(tree, path)$ where *path* is a path to a node in *tree*. The node thus reached is (say) a wildcard. The subtree at the node is unimportant except for purposes of comparison with other trees. The purpose of contexts is to determine where patterns might appear and the costs that a pattern might generate in such a context; a pattern p appears in a context $c = (tree, path)$ if there is a pattern q in *PF* such that q is equal to *tree* with the node at *path* replaced by p. This same pattern q has an associated cost; this is the cost of p in the context c.

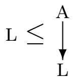
When generating patterns, maintain two sets of sets of patterns. The first is the "unfiltered" matching sets; this contains sets of patterns that result from the simple (no wildcards) phase of the algorithm for generating new matching sets. Doing this acts as a filter to reduce the number of sets given the more expensive treatment that follows. Given a new unfiltered set, add all subsumed wildcards (including those resulting from intersection and complement operations). Build an array indexed by context containing pairs $(cost, patterns)$. The pair at index c contains (as first member) the smallest cost obtained by combining patterns in the matching set with c and (as second member) the set of patterns achieving this cost in context c. After this array is filled, the actual matching set is obtained by taking the union of patterns over all "significant" (see non-optimality result below) contexts. The representer sets are obtained by taking the union of patterns over all contexts associated with label A and child j. Doing this can reduce the number of unfiltered matching sets formed at later stages of the algorithm. Each "unfiltered" set will of course be associated with the resulting "actual" matching set.

Doing this will take more time for each matching set, but will ultimately reduce the table sizes. It may construct significantly fewer unfiltered matching sets, thus reducing the total time needed to construct tables.

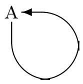
B Application to non-tree input

A bottom-up pattern matcher can be applied to DAGs or general graphs. To process a DAG efficiently, topologically sort the nodes and compute the matching sets in the topological order. This will clearly give the correct matching sets for each node. To process a general graph, each node is initially assigned its height 0 matching set; either $\{*\}$, $\{\}$, or $\{A[\,], *\}$. Given height zero matching sets, matching sets containing taller patterns are computed for each node based on the matching sets of the node's children (successors). If the pattern set is finite (no regular patterns) then this process will converge as soon as the height of the tallest pattern has been reached. If there are regular patterns, then the process will still converge, though the bound on number of iterations is not as good.

Use of the algorithm on graphs points brings up the problem of fixed points. The tables produced by the algorithm in this paper will find smallest fixed points of pattern sets. This is good for trees and DAGs, because these are the only pattern sets that will be produced and this restriction produces smaller tables. If cycles are allowed, however, it is possible to construct a graph with more than one pattern set fixed point. For example, suppose the pattern set is



and the input graph is



In this case, the smallest fixed point of the input tree is the empty set. However, the set {L} is also a fixed point.

C Non-optimality of extension to regular tree patterns

Eduardo Pelegri-Llopart pointed out that the extension to regular trees does not yield optimal automata[PL86]. The extension introduces naming, and it is possible through the use of names to say "these two patterns will have exactly the same effect". This leads to duplicated columns in the tables that are not detected by the equivalence-class method described above. For example, suppose that the two patterns $A[B, *]$ and $A[*, B]$ both subsume the wildcard *foo*, but appear nowhere else in the pattern specification. The algorithm above will make a distinction between matching sets including one pattern, the other pattern, and both patterns. However, those sets can index equal subtables.

This result can be improved (made optimal?) by using or adapting the modification to incorporate costs. If a pattern appears *only* on the right hand side of a subsumption relation, then it should be thrown out of any matching set after all subsumed patterns have been added. Subsumed patterns can be treated in the same way; if they do not identify an interesting pattern instance or appear as part of a pattern, then they should be thrown out of matching sets. I do not know if this produces optimal sets; I suspect not, because names can be used in trickier ways.