A Boyer-Moore (or Watson-Watson) Type Algorithm for Regular Tree Pattern Matching

Bruce W. Watson

Ribbit Software Systems Inc.
(IST Technologies Research Group)
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada

e-mail: watson@RibbitSoft.com

Abstract. In this paper, I outline a new algorithm for regular tree pattern matching. The Boyer-Moore family of string pattern matching algorithms are considered to be among the most efficient. The Boyer-Moore idea of a shift distance was generalized by Commentz-Walter for multiple keywords, and generalizations for regular expressions have also been found. The existence of a further generalization to tree pattern matching was first mentioned in the statements accompanying my dissertation, [Wats95].

Key words: tree pattern matching, tree parsing, code selection, Boyer-Moore algorithms, shift distances.

1 Introduction

The most popular exact pattern matching algorithms[1] (for strings or trees) can be classified into one of two families: the Knuth-Morris-Pratt (KMP) or Boyer-Moore (BM) families.

Interest in Boyer-Moore type algorithms is driven by the fact that they are frequently much faster (in practice) than their KMP counterparts. For a discussion of this phenomenon, see [Wats95]. Since a KMP type algorithm for regular tree pattern matching was presented in [1], the missing piece has been a BM type algorithm for tree pattern matching. This algorithm is an extension (to trees) of one of the algorithms presented at the Prague Workshop in 1996 [4]; it is also related to the algorithm presented at the European Symposium on Algorithms in 1996 [3]. For more background material, consult the links on my homepage at `http://www.RibbitSoft.com/research/watson/index.html`

Instead of providing a set of formal definitions, we introduce most of the concepts using examples.

[1] As opposed to *approximate* pattern matching algorithms.

2 The problem

For the regular tree pattern matching problem, we consider node labeled trees (the labels are taken from a fixed alphabet). All nodes with a particular label have a fixed arity (number of children). We will write all of our trees in a linear (prefix) form, instead of drawing pictures; for example:

$$+(a, *(b, a))$$

Each of the nodes has an associated depth, with the depth of the root being 0.

A tree grammar is a finite set of productions, with nonterminals (written as uppercase letters, as opposed to regular node labels which are written in lowercase letters or as mathematical operators) on the left and tree templates on the right. Nonterminals are permitted to appear at the leaves of the tree templates. For example, each of the following lines is a production:

$$\begin{array}{lcl} A & \longrightarrow & +(B, B) \\ B & \longrightarrow & a \\ B & \longrightarrow & *(C, a) \\ C & \longrightarrow & b \\ C & \longrightarrow & *(b, B) \end{array}$$

The productions match at the input tree nodes in the intuitive way. In our sample input tree $(+(a, *(b, a)))$, we have the following matched patterns:

- The left a is matched by $B \longrightarrow a$.
- The right a is matched by $B \longrightarrow a$.
- The b node is matched by $C \longrightarrow b$.
- The $*$ node is matched by $B \longrightarrow *(C, a)$ and by $D \longrightarrow *(b, B)$.
- The $+$ node is matched by $A \longrightarrow +(B, B)$.

In the next section, we will subdivide the pattern matching problem into a smaller problem which can be solved more readily.

3 Subproblems

One way of reducing the pattern matching problem to a simpler one, is to encode the trees as strings. We do this using so-called *path strings*. In this scheme, the tree is represented as a set of strings (there is one string for each leaf in the tree). Each string consists of alternating node labels and child numbers. For example, our input tree $+(a, *(b, a))$ is represented by $+1a$, $+2 * 1b$, and $+2 * 2a$.

In a similar manner, we encode each of the right sides of the productions as a set of path strings. The only difference is that we omit the nonterminals. For example, production $A \longrightarrow +(B, B)$ is represented by the two path strings $+1B$ and $+2B$, from which we drop the nonterminals to get $+1$ and $+2$. The example set of right sides is encoded as $+1$, $+2$, a, $*1$, $*2a$, b, $*1b$, and $*2$. These path strings can then be mapped back to their corresponding production right sides. These *pattern path*

strings will be used in a reduced problem. Note that the pattern path strings will always begin with a node label.

Given this encoding, we will only concern ourselves with finding matches of the pattern path strings in the set of strings representing the input tree. The matching tree productions can then be easily reconstructed — this is not considered further here. In our example set of input path strings, we have the following pattern path string matches:

- $+1$ and $+2$ match at the root.
- a matches at the left and the right a nodes.
- $*1$, $*2$, $*1b$, and $*2a$ match at the $*$ node.
- b matches at the b node.

From this information, we can then piece together the tree matches. Note that, in effect, we are making use of multiple keyword pattern matching with the pattern path strings as the keywords. To solve this problem, we could use the Commentz-Walter algorithm (among others) [Wats95, Section 4.4].

4 Solving the reduced problem

We begin by presenting a brute-force (naïve) algorithm, solving our simplest subproblem. In presenting the algorithm, we will assume (as in the string pattern matching algorithms presented in my dissertation) the following:

- We use a forward trie τ (constructed from the pattern path strings) for the actual pattern matching. The symbol $\perp$ is used to indicate when the trie takes an undefined value.
- We assume that the start state for the trie is named q_0.
- There is a special procedure RM (for 'register matches') which is used to register matches at nodes in the tree. Precisely how it registers the matches is not relevant.

The mainline of the algorithm is:

$lev := (\mathbf{MAX}\ n : n \in nodes : n.level)$;
do $lev \geq 0 \longrightarrow$
    for $n : n \in nodes \wedge n.level = lev \longrightarrow$
        $AM(q_0, n)$
    rof;
    $lev := lev - 1$
od

This algorithm simply traverses the tree from the bottom up, using procedure AM (for 'attempt match') to check for matches and RM to register the matches. The procedure AM is given as:

```
proc AM(q, n) is
        RM(q, n);
        if τ(q, n.label) ≠ ⊥ then
                q := τ(q, n.label);
                RM(q, n);
                for i ∈ [1, n.arity] ⟶
                        if τ(q, i) ≠ ⊥ then
                                AM(τ(q, i), n.child(i))
                        fi
                rof
        fi
corp
```

This procedure uses the trie and traverses the input tree (starting at node n) top-down to find matches. Note that it is recursive.

As with the other BM type algorithms, we wish to make shifts of more than one level (in the tree) in the mainline program. The shift will be computed in a manner similar to that in the Commentz-Walter family of algorithms — since we are using multiple keyword pattern matching.

Since procedure AM tries a number of possible paths rooted at a node n, there will be a number of potential contributing shifts. In order to make a *safe* shift, we will have to use the smallest of these contributing shifts.

We will use a novel method of implementing the actual shift: if a shift of distance k is required after a match attempt at node n, we will store $n.level - k$ in a location $permit[n]$ ($permit$ is an array which is indexed by n). If a match attempt is initiated at some node n' (above n), then the match attempt will not continue (down in the tree) past node n unless $n' \leq permit[n]$. This can be done safely since all matches that began lower than $permit[n]$ cannot possibly lead to a match below n.

To implement this, we use the following mainline[2]:

```
for n : n ∈ nodes ∧ n.isleaf ⟶
        permit[n] := n.level
rof;
lev := (MAX n : n ∈ nodes : n.level);
do lev ≥ 0 ⟶
        for n : n ∈ nodes ∧ n.level = lev ⟶
                permit[n] := n.level − AM(q0, n, lev)
        rof;
        lev := lev − 1
od
```

Correspondingly, we make use of the shift function *shift* which gives the shift distance in levels in the tree[3].

[2] To abort a match attempt when it is futile, we also pass a third argument to AM — the level at which the match attempt was started. The procedure now returns the integer shift (in terms of levels).

[3] The distance in levels is the ceiling of half of the distance given by the Commentz-Walter shift functions, since the shift distance given for the pattern path strings will be in terms of the path strings and a path string may be up to twice as long as the level of the leaf at which it ends because path strings contain the edge numbers interspersed with node labels.

```
proc AM(q, n, beginlev) returns sh is
        RM(q, n);
        if τ(q, n.label) = ⊥ ∨ beginlev > permit[n] then
                sh := shift(q)
        else
                q := τ(q, n.label);
                RM(q, n);
                if n.arity = 0 then
                        sh := shift(q)
                else
                        for i ∈ [1, n.arity] ⟶
                                if τ(q, i) = ⊥ then
                                        sh := sh min shift(q)
                                else
                                        sh := sh min AM(τ(q, i), n.child(i), beginlev)
                                fi
                        rof
                fi
        fi
corp
```

5 Conclusions

I have outlined a Watson-Watson type algorithm for regular tree pattern matching, thereby backing-up the statement accompanying my dissertation [Wats95]. Unfortunately, this algorithm has not yet been implemented and so nothing is known about its running time performance in practice, though it could potentially be proportional to the product of the input tree size and the size of the largest pattern tree. It therefore appears that the algorithm will not be as efficient as the KMP type algorithm (solving the same problem) given by Aho and Ganapathi in [1], which runs in time linear to the size of the input tree.

Like the other classes of BM type algorithms (for single keyword, multiple keyword, or regular expression string pattern matching), there are likely to be other (perhaps more efficient) variants of this algorithm. For example, it may be possible to devise such an algorithm which operates in a top-down manner, instead of in a bottom-up manner. Alternatively, it may be possible to reduce the primary problem in a different way than what we have done here. These alternatives are left as an exercise for the reader.

Acknowledgements:

I would like to thank Richard Watson (co-developer of the Watson-Watson regular expression pattern matching algorithm for strings) and Nanette Saes for their assistance in preparing this note.

References

[1] AHO, A.V. and M. GANAPATHI. Efficient tree pattern matching: an aid to code generation, in: *Proceedings of the 12th ACM Symposium on Principles of Programming Languages,* New Orleans, p. 334–340, 1985.

[2] WATSON, B.W. *Taxonomies and Toolkits of Regular Language Algorithms,* Ph.D dissertation, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, 1995, ISBN 90-386-0396-7.

[3] WATSON, B.W. A new regular grammar pattern matching algorithm, in: Diaz, J. and M. Serna, eds., *Proceedings of the European Symposium on Algorithms*, Barcelona, Spain, 1996.

[4] WATSON, B.W. A collection of new regular grammar pattern matching algorithms, in: J. Holub, ed., *Proceedings of the First Annual Prague Stringology Club Workshop*, Prague, Czech Republic, 1996.