Disjunctive Partial Deduction of a Right-to-Left String-Matching Algorithm

Manuel Hernández and David A. Rosenblueth

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Universidad Nacional Autónoma de México
Apdo. 20-726, 01000 México D.F.

Abstract

In spite of its practical importance, the Boyer–Moore string-matching algorithm has hardly been studied in the context of partial evaluation. We show how to derive the search stage of a variant of this algorithm using "disjunctive" partial deduction.

Key words: Program derivation, Partial deduction, String matching

1 Introduction

Partial deduction aims at producing an efficiently executable implementation by applying a sequence of meaning-preserving rules to a logic program together with some of its input. Pettorossi, Proietti, and Renault [5] illustrate their "disjunctive" partial deduction with a strategy deriving the search stage of the (left-to-right) Knuth–Morris–Pratt string-matching algorithm.

Although this algorithm is important for its theoretical elegance, in practice it is not significantly better than the naive method [6, p. 10]. Instead, variants of a different (right-to-left) algorithm, devised by Boyer and Moore, are considerably faster. In this paper, we give a minimal common generalization of the strategy of [5] deriving the search stage of the Knuth–Morris–Pratt algorithm and the strategy of [3] deriving the search stage of a variant of the Boyer–Moore algorithm. This common generalization uses an elaboration of the ordinary deterministic unfolding rule: to decide whether or not a subgoal A is deterministic, a set of subgoals that will have succeeded *before* A is considered ("lookbehind").

2 String-matching algorithms

The *string-matching problem* consists in finding all occurrences (or only one occurrence, if any) of a string called the *pattern* within another string called the *text*. A *naive algorithm* for this problem would be to line up the pattern over the beginning of the text, and successively compare, from left to right, each symbol of the pattern with its corresponding symbol of the text. If a mismatch occurs, we then shift the pattern one symbol to the right and start the comparisons again. If mismatches happen every time we almost finish comparing all symbols of the pattern, we may obtain a *quadratic* performance. This situation, however, is unlikely, and the expected performance is *linear*.

Knuth, Morris, and Pratt [4] devised an algorithm (*KMP*) having linear complexity even in the worst case. Their method is similar to that of the naive algorithm, except that through a preprocessing stage, information about the pattern is used so that the pointer to the text is never moved backwards. Essentially, whenever there is a mismatch, the pattern is shifted to the right as many symbols as necessary so that a smaller prefix of the pattern already matches corresponding symbols of the text.

Boyer and Moore [1] drastically improved the average case in a *sublinear* algorithm (*BM*). In their method, which also has a preprocessing stage, the pattern is as well slid from left to right. The comparisons, however, proceed starting from the *rightmost* symbol of the pattern and continue to the left. In this case no memory is kept of the possible partial matches obtained in the previous steps. The search is fast on average, because in many cases the number of symbols shifted is close to the length of the pattern.

We now describe our variant of the BM algorithm in more detail. We denote by s_k the k^{th} symbol of a string s (s_1 being the first symbol of s). In addition, t denotes the text, with length n and index i, and p denotes the pattern, with length m and index j.

Comparisons of corresponding pattern and text symbols are performed from right to left until a complete match is obtained or a mismatch at p_j and t_i, say, occurs. We can determine, in a manner similar to that of the KMP algorithm, a reoccurrence of the pattern suffix that matched. In this case, $p_{j+1} \cdots p_m = t_{i+1} \cdots t_{i-j+m}$ and $p_j \neq t_i$. If the suffix $p_{j+1} \cdots p_m$ also appears in p as a substring $p_{j+1-k} \cdots p_{m-k}$, with $p_{j-k} \neq p_j$, and it is the rightmost such occurrence, then the pattern may be shifted k symbols to the right ($1 \leq k < j$). It may also happen that such a reoccurrence "falls off" the left end of p, that is, a suffix of $p_{j+1} \cdots p_m$ appears as a prefix $p_1 \cdots p_{m-k}$ of p ($j \leq k \leq m$). The

```
initialize(p, δ); i := m; j := m;
while (j > 0) and (i ≤ n) do
    if t_i = p_j
    then  begin i := i − 1; j := j − 1 end    {partial match}
    else  begin i := i + δ(j); j := m end     {mismatch}
if j < 1 then i := i + 1 else i := 0
```

Figure 1. A variant of the Boyer–Moore algorithm

number of symbols to be shifted can thus be precomputed as:

$$\begin{aligned}\delta(j) = \min\{k + m - j \mid k \geq 1 \\ &\text{and } (k < j \Rightarrow p_{j-k} \neq p_j) \\ &\text{and } ((k < d \Rightarrow p_{d-k} = p_d) \quad \forall d\ [j < d \leq m])\}\end{aligned} \tag{1}$$

If the reoccurrence does not fall off, then $p_{j-k} \neq p_j, p_{j-k+1} = p_{j+1}, \ldots, p_{m-k} = p_m$, which results from the second conjunct and from varying d from $j + 1$ to m in the third conjunct. If the reoccurrence does fall off, then $p_1 = p_{k+1}, \ldots, p_{m-k} = p_m$, which results from varying d from $k + 1$ to m in the third conjunct (when no equation holds, $k = m$).

Boyer and Moore [1] use, in addition to δ, another criterion based on occurrences of the current text symbol t_i in the pattern, and compute the maximum of both these functions when a mismatch happens. This additional function, however, has little effect if the alphabet is small [2, p. 59]. Our variant consists precisely in omitting such a criterion (see Figure 1).

3 Disjunctive partial deduction and the D strategy

"Disjunctive" partial deduction [5] employs a folding rule using a multiple-clause definition operating on disjunctions of conjunctions of literals. As illustrated in [5] by Pettorossi, Proietti, and Renault, this "extended" folding may decrease nondeterminism, and these authors are able to derive the search part of the KMP algorithm from a naive, nondeterministic specification.

The application of the transformation rules is mechanized in the "D" strategy, shown (with minor departures from [5]) in Figure 2. An unfolding is often called *deterministic* if the selected subgoal A unifies with exactly one head, and *nondeterministic* if it does so with more than one head. Sometimes *lookahead* is used to establish if the subgoal can be considered deterministic, by unfolding subgoals that will succeed after A. Instead, we use *lookbehind*, by accumulating subgoals that will have succeeded before A.

Each cycle in the D strategy comprises:

$i := 0$; $\mathit{Def}_0 := \{C\}$;   /* C is the specification (specialized clause) */
repeat
(a) $Cs := \mathit{case\_split}(\mathit{det\_unfold}(\mathit{unfold\_once}(\mathit{Def}_i)))$;
(b) $\mathit{Def}_{i+1} := \mathit{required\_defs}(\bigcup_{j=0}^{i} \mathit{Def}_j, Cs)$;
(c) $F_{i+1} := \mathit{fold}(\bigcup_{j=0}^{i+1} \mathit{Def}_j, Cs)$;
(d) $U_{i+1} := \mathit{unit\_clauses}(\mathit{Def}_{i+1})$;
     $i := i + 1$
until $\mathit{Def}_i = \{\}$;
$P_{residual} := \bigcup_{j=1}^{i} U_j \cup \bigcup_{j=1}^{i} F_j$

Figure 2. The D strategy

(a) exactly one possibly nondeterministic unfolding of each clause in Def_i, followed by zero or more applications of deterministic unfolding, case split, subsumption, and inequation simplification;
(b) zero or more definition introductions;
(c) zero or more foldings; and
(d) extraction of the unit clauses.

In the next section we will see that a strategy deriving the search stage of our variant of the BM algorithm is similar to the D strategy, except that:

i. we omit the possibly nondeterministic unfolding if a certain condition (given below) holds, and
ii. we first apply case split and then deterministic unfolding with lookbehind as many times as possible.

To obtain a common generalization, named the *D' strategy*, we replace (a) by:

(a') $Cs := \mathit{det\_unfold'}(\mathit{case\_split}(\mathit{det\_unfold}(\mathit{possibly\_unfold\_once}(\mathit{Def}_i))))$;

where *det_unfold'* uses lookbehind, and *possibly_unfold_once* unfolds once only if a condition given below holds.

4 Deriving the search stage of a variant of the BM algorithm

The derivation can be conceptually thought of as (1) the addition of constraints through case split and (2) repeated deterministic unfolding until the constraints are satisfied. To conform to a cyclic strategy, however, it will be more convenient to alternate both these processes incrementally.

We represent the text by a predicate $t(I, X)$ meant to hold when the I^{th} text symbol is X. Standard Prolog implementations normally index the predicates on their first argument place, so that subgoals $t(I,X)$ having a constant first argument succeed in constant time. $t(J{+}d,X)$ abbreviates I *is* $J+d,\ t(I,X)$.

Assume as an example that the pattern is *abb*. A specification could then be:

$$match_0(I,K) \leftarrow between(I,J,K),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a)$$

where $between(I,J,K)$ is intended to hold iff $I \leq J \leq K$. The goal would be: $\leftarrow match_0(1,n)$. For conciseness, however, we will prefer a nonterminating variant of this specification by omitting one argument place of the *between* predicate (which we name *leq*):

$$match_0(I) \leftarrow leq(I,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) \tag{2}$$

where $leq(I,J)$ is intended to hold iff $I \leq J$:

$$leq(I,I) \leftarrow$$
$$leq(I,J) \leftarrow leq(I{+}1,J)$$

First iteration. By unfolding (2) we get (annotations are explained below):

$$match_0(I) \leftarrow t(I{+}2,b)^{*},\ t(I{+}1,b),\ t(I,a) \tag{3}$$
$$match_0(I) \leftarrow leq(I{+}1,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) \tag{4}$$

Applying case split to (4) w.r.t. $t(I{+}2,b)$ we get:

$$match_0(I) \leftarrow \quad t(I{+}2,b)^{*},\ leq(I{+}1,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) \tag{4a}$$
$$match_0(I) \leftarrow not(t(I{+}2,b)),\ leq(I{+}1,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) \tag{4b}$$

Our first departure from the D strategy appears here, where such a strategy does not prescribe any unfolding. However, an important feature of the BM algorithm is a shift of the text pointer of possibly more than one symbol in case of a mismatch, and such a shift is readily obtained by unfolding (4b) (as t is unknown, all unfoldings are w.r.t. *leq*):

$$match_0(I) \leftarrow not(t(I{+}2,b)),\ t(I{+}3,b),\ t(I{+}2,b),\ t(I{+}1,a) \tag{4b$'$}$$
$$match_0(I) \leftarrow not(t(I{+}2,b)),\ leq(I{+}2,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) \tag{4b$''$}$$

Since (4b′) has contradictory subgoals, we can delete (4b′), so that this unfolding step is in fact deterministic. (Only one deterministic unfolding step is possible, as another unfolding would derive two clauses, neither of which has contradictory subgoals.) Deterministic unfolding cannot be applied to (4a).

For folding (3) and (4a) w.r.t. $t(I{+}2,b)$ (ovals labelled $*$), we introduce:

$$match_1(I) \leftarrow \underline{t(I{+}1,b)}^{**},\ t(I,a) \tag{5}$$
$$match_1(I) \leftarrow\ leq(I{+}1,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) \tag{6}$$

By folding (3) and (4a) using (5) and (6), we get:

$$match_0(I) \leftarrow t(I{+}2, b),\ match_1(I) \tag{7}$$

By folding (4b″) using (2) we get:

$$match_0(I) \leftarrow not(t(I{+}2, b)),\ match_0(I{+}2) \tag{8}$$

Second iteration. A second departure from the D strategy happens at this point, where such a strategy prescribes a (possibly nondeterministic) unfolding step. Note that if we perform such a step, we create an undesirable nondeterminism. To see this, let us unfold (6):

$$match_1(I) \leftarrow \underline{t(I{+}3, b)},\ t(I{+}2, b),\ t(I{+}1, a) \tag{6′}$$
$$match_1(I) \leftarrow leq(I{+}2, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{6″}$$

Clauses (5) and (6′) have two essentially distinct subgoals (underlined) which would generate a profusion of cases upon application of case split. A criterion for not unfolding can be: *if by unfolding a clause defining a predicate we obtain a leftmost subgoal A different from the leftmost subgoal B of another clause defining the same predicate, such that $\leftarrow A, B$ can succeed, then we do not unfold.* For example: $\leftarrow t(I{+}3, b),\ t(I{+}1, b)$ can succeed, but not so $\leftarrow t(I{+}2, b),\ t(I{+}2, a)$ because of the functionality of t w.r.t. its first argument. By case split of (6) w.r.t. $t(I{+}1, b)$, we get:

$$match_1(I) \leftarrow t(I{+}1, b),\ leq(I{+}1, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{6a}$$
$$match_1(I) \leftarrow not(t(I{+}1, b)),\ leq(I{+}1, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{6b}$$

As in the first iteration, we deterministically unfold (6a) and (6b) if possible. This time, however, we need to "look behind" and use the fact that $t(I{+}2, b)$ holds when $match_1$ is called (cf. (7)). Now (6a) can be deterministically unfolded, but (6b) cannot.

$$match_1(I) \leftarrow t(I{+}1, b)^{**},\ leq(I{+}2, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{6a′}$$

For folding (5) and (6a′) w.r.t. $t(I{+}1, b)$ (ovals labelled $**$), we introduce:

$$match_2(I) \leftarrow t(I, a)^{***} \tag{9}$$
$$match_2(I) \leftarrow leq(I{+}2, J),\ t(J{+}2, b),\ t(J{+}1, b),\ t(J, a) \tag{10}$$

By folding (5) and (6a′) using (9) and (10), we get:

$$match_1(I) \leftarrow t(I{+}1, b),\ match_2(I) \tag{11}$$

By folding (6b) using (2), we get:

$$match_1(I) \leftarrow not(t(I{+}1, b)),\ match_0(I{+}1) \tag{12}$$

Third iteration. For the same reason as above, we do not unfold. By case split of (10) w.r.t. $t(I,a)$ we get:

$$\begin{array}{llr} match_2(I) \leftarrow & t(I,a),\ \ leq(I{+}2,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) & \text{(10a)} \\ match_2(I) \leftarrow & not(t(I,a)),\ leq(I{+}2,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) & \text{(10b)} \end{array}$$

Next we deterministically unfold (10a) and (10b), considering that $t(I{+}2,b)$ and $t(I{+}1,b)$ hold when $match_2$ is called:

$$\begin{array}{llr} match_2(I) \leftarrow & t(I,a)^{***},\ leq(I{+}3,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) & \text{(10a}'\text{)} \\ match_2(I) \leftarrow & not(t(I,a)),\ leq(I{+}3,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) & \text{(10b}'\text{)} \end{array}$$

For folding (9) and (10a′) w.r.t. $t(I,a)$ (ovals labelled $***$), we introduce:

$$match_3(I) \leftarrow \tag{13}$$

$$match_3(I) \leftarrow leq(I{+}3,J),\ t(J{+}2,b),\ t(J{+}1,b),\ t(J,a) \tag{14}$$

By folding (9) and (10a′) using (13) and (14), we get:

$$match_2(I) \leftarrow t(I,a),\ match_3(I) \tag{15}$$

By folding (10b′) using (2), we get:

$$match_2(I) \leftarrow not(t(I,a)),\ match_0(I{+}3) \tag{16}$$

Fourth iteration. By folding (14) using (2), we get:

$$match_3(I) \leftarrow match_0(I{+}3) \tag{17}$$

The residual program consists of (7, 8, 11, 12, 15, 16, 13, 17).

In general, the residual program is of the form:

$$\begin{array}{lllr} match_{m-j}(I) \leftarrow & t(I{+}j{-}1,p_j),\ \ match_{m-j+1}(I) & j = m,\ldots,1 & \\ match_{m-j}(I) \leftarrow & not(t(I{+}j{-}1,p_j)),\ match_0(I{+}c(j)) & j = m,\ldots,1 & \text{(18)} \\ match_m(I) \leftarrow & & & \\ match_m(I) \leftarrow & match_0(I{+}c(0)) & & \text{(19)} \end{array}$$

for some function c, related to δ as follows.

Theorem $c(j) = \delta(j) - m + j$.

Proof. We will first deal with mismatches. In the clauses of the form (18), $c(j)$ is an accumulated number of deterministic unfoldings. We can equivalently perform such unfoldings only at the end, once we have derived the clauses:

$$\begin{aligned}
match_0(I) &\leftarrow t(I{+}m{-}1, p_m),\ match_1(I)\\
&\ \ \vdots\\
match_{m-j-1}(I) &\leftarrow t(I{+}j, p_{j+1}),\ match_{m-j}(I)\\
match_{m-j}(I) &\leftarrow not(t(I{+}j{-}1, p_j)),\ leq(I{+}1, J),\ agree(J, 0)
\end{aligned}$$

Here and throughout $agree(K, l)$ abbreviates $t(K{+}m{-}1, p_m), \ldots, t(K{+}l, p_{l+1})$. The deterministic unfoldings are then applied to clauses of the form:

$$match_{m-j}(I) \leftarrow \boxed{agree(I, j),}\ not(t(I{+}j{-}1, p_j)),\ leq(I{+}1, J),\ agree(J, 0) \quad (20)$$

We use a rectangle to indicate the lookbehind subgoals.

Assume that starting from (20), deterministic unfolding can be applied $c(j)-1$ times so that the first nondeterministic unfolding derives:

$$match_{m-j}(I) \leftarrow \boxed{agree(I, j),}\ not(t(I{+}j{-}1, p_j)),\ agree(I{+}c(j), 0) \qquad (21)$$

$$match_{m-j}(I) \leftarrow \boxed{agree(I, j),}\ not(t(I{+}j{-}1, p_j)),\ leq(I{+}c(j){+}1, J),\ agree(J, 0)$$

Note that (21) does not have contradictory subgoals (otherwise we could perform another deterministic unfolding step), so we fold the clause of the step previous to this last unfolding, obtaining:

$$match_{m-j}(I) \leftarrow \boxed{agree(I, j),}\ not(t(I{+}j{-}1, p_j)),\ match_0(I{+}c(j)) \qquad (22)$$

We distinguish now two cases:

(1) (The disequation holds.) Assume that for some k such that $1 \le k < j$,

$$p_{j-k} \neq p_j \text{ and } (p_{d-k} = p_d) \quad \forall d\ [j < d \le m].$$

Then (21), the first clause with no contradictions, has the subgoals:

$$\begin{array}{llll}
 & t(I{+}m{-}1, p_m), \quad \ldots, & t(I{+}j, p_{j+1}), & not(t(I{+}j{-}1, p_j)),\\
\ldots, & t(I{+}m{-}1, p_{m-k}), \quad \ldots, & t(I{+}j, p_{j+1-k}), & t(I{+}j{-}1, p_{j-k})
\end{array}$$

The deterministic unfoldings halt with the least such k. Hence,

$$\begin{aligned}
c(j) &= \min\{k \mid k \ge 1 \text{ and } p_{j-k} \neq p_j \text{ and } (p_{d-k} = p_d) \quad \forall d\ [j < d \le m]\}\\
&= \delta(j) - m + j
\end{aligned}$$

(2) (The disequation does not hold for any k and some equations may hold.) Assume that for some k such that $j \le k \le m$,

$$(k < d \Rightarrow p_{d-k} = p_d) \quad \forall d\ [j < d \le m]$$

(where $k = m$ if no equation holds). Then (21) has the following subgoals:

$$\begin{array}{l} \quad t(I{+}m{-}1, p_m), \quad \ldots, \ t(I{+}k, p_{k+1}), \ \ldots, \ t(I{+}j{-}1, p_j), \ \ldots \\ \ldots, \ t(I{+}m{-}1, p_{m-k}), \ \ldots, \ t(I{+}k, p_1) \end{array}$$

Hence,

$$\begin{aligned} c(j) &= \min\{k \mid k \geq 1 \text{ and } ((k < d \Rightarrow p_{d-k} = p_d) \quad \forall d\ [j < d \leq m])\} \\ &= \delta(j) - m + j \end{aligned}$$

Similarly, in (19), $c(0) = \min\{k \mid k \geq 1 \text{ and } ((p_{d-k} = p_d) \quad \forall d\ [k < d \leq m])\} = \delta(0) - m + 0$ specifies the shift size in case of a complete match. □

The D' strategy should also be able to derive the search part of the KMP algorithm. It is possible to confirm that such a strategy does so from:

$$match_0(I) \leftarrow leq(I, J),\ t(J, a),\ t(J{+}1, b),\ t(J{+}2, b)$$

5 Conclusions

We have extended the D strategy [5] so as to derive not only the search part of the Knuth–Morris–Pratt string-matching algorithm, but also that of a variant of the Boyer–Moore algorithm.

References

[1] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[2] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[3] M. Hernández and D. A. Rosenblueth. Development reuse and the logic program derivation of two string-matching algorithms. In *Proc. PPDP'01*, pages 38–48, Florence, Italy, 2001.

[4] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[5] A. Pettorossi, M. Proietti, and S. Renault. Enhancing partial deduction via unfold/fold rules. In *Proc. 6th LOPSTR*, pages 146–168, Stockholm, Sweden, 1996. Springer-Verlag. LNCS 1207.

[6] G. A. Stephen. *String Searching Algorithms*, volume 3 of *Lecture Notes Series on Computing.* World Scientific Publishing, 1994.