JFP **34**, e3, 10 pages, 2024.
doi:10.1017/S0956796824000017

FUNCTIONAL PEARL

Knuth–Morris–Pratt illustrated

CAMERON MOY
Northeastern University, Boston, MA 02115, USA
(*e-mail:* camoy@ccs.neu.edu)

Abstract

The Knuth–Morris–Pratt (KMP) algorithm for string search is notoriously difficult to understand. Lost in a sea of index arithmetic, most explanations of KMP obscure its essence. This paper constructs KMP incrementally, using pictures to illustrate each step. The end result is easier to comprehend. Additionally, the derivation uses only elementary functional programming techniques.

1 Introduction

Both the Knuth–Morris–Pratt and the Boyer–Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used.

Robert Sedgewick, Algorithms

String search is a classic problem. Given a string, the *pattern*, determine if it occurs in a longer string, the *text*. String search can be solved in $O(n+m)$ time and $O(m)$ space, where n is the size of the text and m is the size of the pattern. Unfortunately, the algorithm that does so, Knuth–Morris–Pratt (KMP) (Knuth et al., 1977), is hard to understand. Its pseudocode is short, but most explanations of it are not.

Standard treatments, like that of Cormen et al. (2009) or Sedgewick & Wayne (2011), contain headache-inducing descriptions. Actually, neither even explain the genuine KMP algorithm. Cormen et al. explain the simpler Morris–Pratt (MP) algorithm and leave Knuth's optimization as an exercise. Sedgewick & Wayne present a related algorithm for minimal DFA construction, with greater memory consumption than KMP, and simply assert that it can be improved.

Alternatively, KMP can be *derived* via program transformation (Takeichi & Akama, 1990; Colussi, 1991; Hernández & Rosenblueth, 2001; Ager et al., 2003; Bird, 2010). Indeed, Knuth himself calculated the algorithm (Knuth et al., 1977, p. 338) from a constructive proof that any language recognizable by a two-way deterministic pushdown automaton can be recognized on a random-access machine in linear time (Cook, 1972).

What follows is a journey from naive string search to the full KMP algorithm. Like other derivations, we will take a systematic and incremental approach. Unlike other derivations,

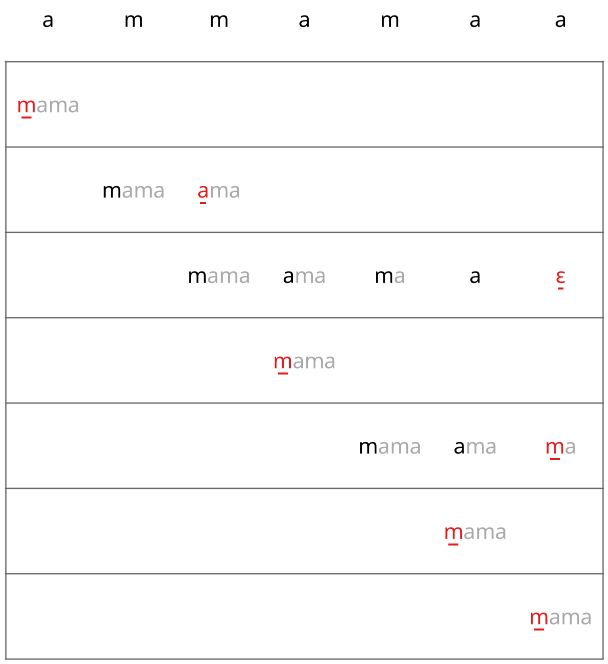

Fig. 1. Naive string search (horizontal).

visual intuition will be emphasized over program manipulation. The explanation highlights each of the insights that, taken together, lead to an optimal algorithm. Lazy evaluation turns out to be a critical ingredient in the solution.

2 Horizontally naive

The naive $O(nm)$ algorithm for string search attempts to match the pattern at every position in the text. Consider the pattern mama and the text ammamaa. Figure 1 visualizes the naive approach on this example.

Each row corresponds to a new starting position in the text. Mismatched characters are colored red and underlined. The third row matches fully, indicated by the underlined ε, so the search is successful. If one just wants to determine if the pattern is present or not, then processing can stop at this point. Related queries, such as counting the number of occurrences of the pattern, require further rows of computation (as shown).

To summarize, naive search finds, if it exists, the left-most *suffix* of the text whose *prefix* is the pattern:

```
horizontal pattern text = any done (scanl step init text)
  where init :: String
        init = text
        step :: String -> Char -> String
        step acc x = tail acc
        done :: String -> Bool
        done acc = isPrefixOf pattern acc
```

The `scanl` function is similar to `foldl` but returns a list of all accumulators instead of just the final one:

```
scanl f acc []     = [acc]
scanl f acc (x:xt) = acc:(scanl f (f acc x) xt)
```

The `any` function determines if some element of the input list satisfies the given predicate:

```
any f []     = False
any f (x:xt) = f x || any f xt
```

Coming back to `horizontal`, the accumulator is initially the entire text. At each step, the accumulator shrinks by one character, generating the next suffix. So, the result of `scanl` is a list containing all suffixes of the text. Then, `any` checks to see if some suffix has a prefix that is the pattern.

In the algorithms that follow, `any` and `scanl` remain the same; they differ only in the choice of `init`, `step`, and `done`.

3 Vertically naive with a set

Figure 2 is identical to Figure 1 except that it uses vertical lines instead of horizontal ones. This picture suggests a different algorithm. Each column is a set of pattern suffixes, all of which are candidates for a match. Calculating the next column involves three steps:

1. Remove suffixes that do not match the current position in the text (colored red and underlined). These suffixes are failed candidates.
2. Take the `tail` of those that do. These suffixes remain candidates.
3. Add the pattern itself, corresponding to the diagonal line of mama. Doing so starts a new candidate at each position.

Let us call the result of this procedure the *successor* of column C on character x. A column containing the empty string, written as ε, indicates a successful match.

Following this picture yields a new approach. Now, accumulators are columns, columns are *sets of strings*, and `step` calculates successors:

```
verticalSet pattern text = any done (scanl step init text)
  where init :: Set String
        init = Set.singleton pattern
        step :: Set String -> Char -> Set String
        step acc x = init ‘Set.union‘ (Set.map tail candidates)
          where candidates = Set.filter (isPrefixOf [x]) acc
        done :: Set String -> Bool
        done acc = Set.member "" acc
```

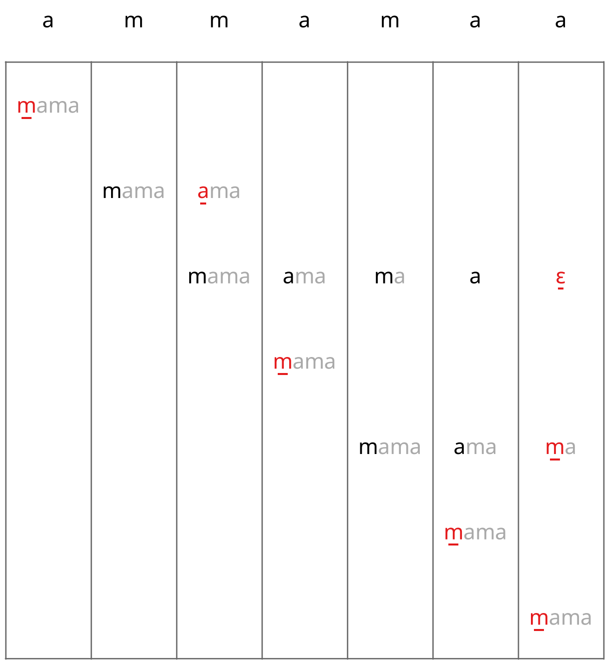

Fig. 2. Naive string search (vertical).

As is, `verticalSet` consumes more memory than `horizontal`. While `step` for `horizontal` does not allocate, `step` for `verticalSet` allocates an entirely new set.

There is a trick to negate this drawback. In the same way that sets of natural numbers can be represented using bitstrings, sets of candidate strings can also be represented in binary. Successors can be calculated using left shift and bitwise or. This optimized algorithm, known as Shift-Or (Baeza-Yates & Gonnet, 1992), performs exceptionally well on small patterns. In particular, Shift-Or works well when the length of the pattern is no greater than the size of a machine word.

Additional notes

String search is equivalent to asking if the regular expression `.*pattern.*` matches. Compiling this regular expression to an NFA and simulating it shows that the columns of Figure 2 are sets of NFA states. The `step` function is then the NFA transition function. Equivalently, columns can be viewed as Brzozowski derivatives (Brzozowski, 1964; Owens et al., 2009) of the regular expression. The `step` function is then the derivative.

4 Vertically naive with a list

Using a set to represent the accumulator has two drawbacks. First, set operations cannot be fused together. Ideally, `step` would traverse the accumulator only once, but with sets it must perform more than one traversal. Second, `done` is not a constant-time operation.

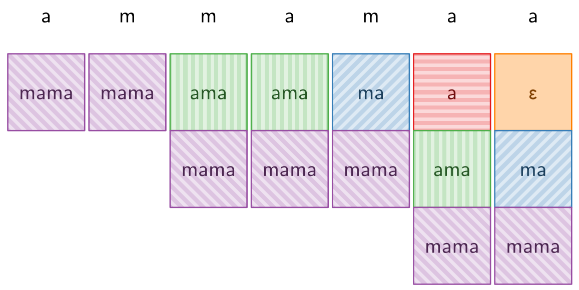

Fig. 3. Columns as lists.

Figure 3 is derived from Figure 2 by removing whitespace from the columns and giving each distinct suffix a unique color and background. This picture suggests representing columns using lists instead of sets, where the first element of the list is the top of the column. The `verticalSet` function can be easily adapted to this new representation:

```
data List a = Nil | Cons { top :: a, rest :: List a }

verticalList pattern text = any done (scanl step init text)
  where init :: List String
        init = Cons pattern Nil
        step :: List String -> Char -> List String
        step Nil x = init
        step acc@(Cons t r) x
          | check acc x = Cons (tail t) (step r (head t))
          | otherwise   = step r x
```

Two features of this snippet may seem unusual now but will be helpful shortly. First, instead of Haskell's built-in lists, the code defines a new datatype. This will be useful in the next section where this datatype is extended. Second, the highlighted expression could more simply be written as `step r x` since `head t` is `x` in this branch. In the next section, `x` will be unavailable, and so `step r (head t)` is the only option at that point.

Now, `step` is a straightforward recursive function that iterates over the list just once. Moreover, each list is automatically sorted by length. Thus, `done` can be completed in constant time since it just needs to look at the first element of the list:

```
done Nil = False
done acc = (top acc) == ""
```

Finally, the `check` function determines whether the candidate at the `top` of `acc` matches the current character of the text:

```
check Nil x = False
check acc x = isPrefixOf [x] (top acc)
```

For the remaining algorithms, `done` and `check` stay the same.

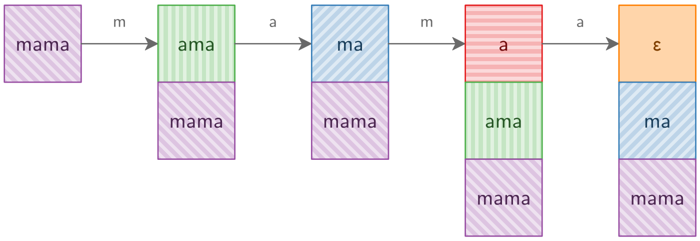

Fig. 4. Column shapes with forward arrows.

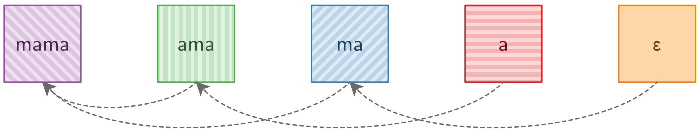

Fig. 5. Column shapes with backward arrows.

5 Morris–Pratt

Take another look at Figure 3. There is yet more structure that can be exploited. In particular, two key properties unlock the secret to KMP:

1. For each pattern suffix, there is only *one* column "shape" where that suffix is `top`.
2. The `rest` field of any column is a *prior* column.

These properties hold for *all* choices of pattern and text; both can be proved inductively using the definition of `step`. Informally:

1. To start with, there is only one accumulator: `init`. A new accumulator can only be generated by calling `step acc x` when `check acc x` holds. Doing so yields a new accumulator, where `top` has shrunk by one character. Additionally, there is only one `x` such that `check acc x` holds. Thus, there is only one accumulator of size n, of size $n-1$, and so forth. Figure 4 shows the five column "shapes" for the pattern mama.
2. The `rest` of column `init` is empty. All other accumulator values must have been generated by calling `step acc x` when `check acc x` holds. Thus, `step` returns a column where `rest` is `step r (head t)`. This expression returns a prior column. Figure 5 shows the five columns where `rest` is indicated by a dashed arrow.

Before, we assumed that columns could be any set of pattern suffixes. There are 2^n such sets. Now we know that only n of these sets can ever materialize. Moreover, each column can be represented as a pair consisting of a pattern suffix and a prior column. Combining Figures 4 and 5 yields a compact representation of all possible columns as a graph, pictured in Figure 6.

All that remains is to construct this graph. Just add a `next` field for the forward edge

```
data Tree a = Nil | Node { top :: a, next :: Tree a, rest :: Tree a }
```

and then compute its value with a "smart" constructor (called `make` here):

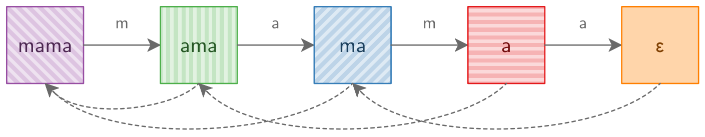

Fig. 6. MP graph.

```
mp pattern text = any done (scanl step init text)
  where make :: String -> Tree String -> Tree String
        make "" r = Node "" Nil r
        make t r  = Node t n r
          where n = make (tail t) (step r (head t))
        init :: Tree String
        init = make pattern Nil
        step :: Tree String -> Char -> Tree String
        step Nil x = init
        step acc@(Node t n r) x
          | check acc x = n
          | otherwise   = step r x
```

Note how the determination of successor columns has been moved from `step` (in `verticalList`) to the constructor (in `mp`). As a result, `init` is now the graph from Figure 6. Then, `step` traverses this graph instead of recomputing successors across the entirety of the text.

In a call-by-value language, this definition would fail because Figure 6 is cyclic. The circularity arises because `init` is defined in terms of `make`, which calls `step`, which returns `init` in the base case. Fortunately, this kind of cyclic dependency is perfectly acceptable in a lazy language such as Haskell.

This algorithm is called Morris–Pratt (MP), and it runs in linear time. Just a small tweak delivers the full KMP algorithm.

Additional notes

One perspective is that Figure 6 depicts a two-way DFA (Rabin & Scott, 1959). Backward arrows represent a set of transitions labeled by $\Sigma \setminus \{x\}$ where x is the matching character. These backward arrows do not consume any input (making it a two-way DFA).

Haskell makes cyclic data construction especially convenient, but it is pretty easy in many eager languages too. Only `next` needs to be lazy. Appendix A gives a Racket implementation of KMP that uses `delay` and `force` to achieve the desired laziness.

Laziness has another benefit. In an eager implementation, the entire graph is always computed, even if it is not needed. In a lazy implementation, if the pattern does not occur in the text, then not of all the graph is used. Thus, not all of the graph is computed.

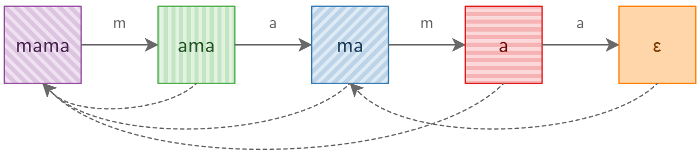

Fig. 7. KMP graph.

```
kmp pattern text = any done (scanl step init text)
  where make :: String -> Tree String -> Tree String
        make "" r = Node "" Nil r
        make t r  = Node t n  r’
          where n  = make (tail t) (step r (head t))
                r’ = if check r (head t) then rest r else r
        init :: Tree String
        init = make pattern Nil
        step :: Tree String -> Char -> Tree String
        step Nil x = init
        step acc@(Node t n r) x
          | check acc x = n
          | otherwise   = step r x
```

Fig. 8. KMP algorithm.

6 Knuth–Morris–Pratt

Take another look at Figure 6. Suppose the current accumulator is the fourth column (where the `top` field is `a`) and the input character is `m`. That is a mismatch, so MP goes back two columns. That is also a mismatch, so it goes back to the first column. That is a match, so the algorithm ends up at the second column.

Note how a mismatch at column `a` *always* skips over column `ama` because the `top` values of the two columns start with the same character. Hence, going directly to column `mama` saves a step. Figure 7 shows the result of transforming Figure 6 according to this insight.

Figure 8 shows the code that implements this optimization, delivering the full KMP algorithm. When constructing a column, KMP checks to see if the first character of `top` matches that of the `rest` field’s `top`. If so, it uses the `rest` field’s `rest` instead. Since this happens each time a column is constructed, `rest` is always going to be the “best” column, that is, the earliest one where `top` has a different first character.

7 Correctness

One way to test that these implementations are faithful is to check that their traces match a reference implementation (Danvy & Rohde, 2006). A *trace* is the sequence of character comparisons performed during a search. Experiments on a large test suite confirm that the code given in Sections 5 and 6 implement MP and KMP, respectively. Moreover, the number of comparisons made in the KMP implementation is always the same or fewer than in the MP implementation, exactly as expected.

8 Conclusion

Naive string search works row-by-row. Going column-by-column yields a new algorithm, but it is still not linear time. MP takes advantage of the underlying structure of columns, representing them as a cyclic graph. This insight yields a linear-time algorithm. KMP refines this algorithm further, skipping over columns that are guaranteed to fail on a mismatched character.

Acknowledgments

The author thanks Matthias Felleisen, Sam Caldwell, Michael Ballantyne, and anonymous JFP reviewers for their comments and suggestions. This research was supported by National Science Foundation grant SHF 2116372.

Conflicts of Interest

None.

References

Ager, M., Danvy, O. & Rohde, H. (2003) Fast partial evaluation of pattern matching in strings. In Partial Evaluation and Semantics-Based Program Manipulation, pp. 3–9.

Baeza-Yates, R. & Gonnet, G. (1992) A new approach to text searching. *Commun. ACM* **35**(10), 74–82.

Bird, R. (2010) *Pearls of Functional Algorithm Design*. Cambridge University.

Brzozowski, J. (1964) Derivatives of regular expressions. *J. ACM* **11**(4), 481–494.

Cook, S. (1972) Linear time simulation of deterministic two-way pushdown automata. *Inf. Process.* **71**, 75–80.

Colussi, L. (1991) Correctness and efficiency of pattern matching algorithms. *Inf. Comput.* **95**, 225–251.

Cormen, T., Leiserson, C., Rivest, R. & Stein, C. (2009) *Introduction to Algorithms*. MIT.

Danvy, O. & Rohde, H. (2006) On obtaining the Boyer–Moore string-matching algorithm by partial evaluation. *Inf. Process. Lett.* **99**(4), 158–162.

Hernández, M., & Rosenblueth, D. (2001) Development reuse and the logic program derivation of two string-matching algorithms. In *Conference on Principles and Practice of Declarative Programming*, pp. 38–48.

Knuth, D., Morris, J. & Pratt, V. (1977) Fast pattern matching in strings. *SIAM J. Comput.* **6**(2), 323–350.

Owens, S., Reppy, J. & Turon, A. (2009) Regular-expression derivatives re-examined. *J. Funct. Program.* **19**(2), 173–190.

Rabin, M. & Scott, D. (1959) Finite automata and their decision problems. *IBM J. Res. Dev.* **3**(2), 114–125.

Sedgewick, R. & Wayne, K. (2011) *Algorithms*. Addison-Wesley Professional.

Takeichi, M. & Akama, Y. (1990) Deriving a functional Knuth-Morris-Pratt algorithm. *J. Inf. Process.* **13**(4), 522–528.

A Racket code

```
;; → Tree
(struct nil ())
;; String, Tree, Tree → Tree
(struct node (top next rest))

;; String, String → Bool
(define (knuth-morris-pratt pattern text)
  (define (make t r)
    (define n
      (delay
        (cond
          [(equal? t "") (nil)]
          [else (make (string-rest t) (step r (string-first t)))])))
    (define r*
      (cond
        [(equal? t "") r]
        [(check? r (string-first t)) (node-rest r)]
        [else r]))
    (node t n r*))
  (define init (make pattern (nil)))
  (define (step acc x)
    (match acc
      [(nil) init]
      [(node t n r) (if (check? acc x) (force n) (step r x))]))
  (fold-until init step done? text))

;; Tree → Bool
(define (done? acc)
  (match acc
    [(nil) false]
    [(node t _ _) (equal? t "")]))

;; Tree, Char → Bool
(define (check? acc x)
  (match acc
    [(nil) false]
    [(node t _ _) (and (not (equal? t "")) (equal? (string-first t) x))]))

;; Tree, (Tree, Char → Tree), (Tree → Bool), String → Bool
(define (fold-until acc step done? text)
  (cond
    [(done? acc) true]
    [(equal? text "") false]
    [else (define acc* (step acc (string-first text)))
          (fold-until acc* step done? (string-rest text))]))

;; String → Char
(define (string-first s) (string-ref s 0))
;; String → String
(define (string-rest s) (substring s 1))
```