Reconstructing the Knuth-Morris-Pratt algorithm

François Pottier

December 20, 2013

In my view, the Knuth-Morris-Pratt string searching algorithm is one of those "magic" algorithms that do something wonderful but are difficult to explain or understand in detail without a significant time investment. The pedagogical question that I would like to address is not just: "how does it work?", or "why does it work?", or "does it work?". More importantly, it is: "how did they come up with it?". Of course, I wasn't there, and I am no historian, so let me instead address the somewhat related question: "how do I reconstruct it?".

Of course one can prove that the algorithm works; this can be done with pen-and-paper or with a machine-checked proof. This is very useful, but does not answer my question.

I know of several ways of explaining and reconstructing the algorithm. In one of them, one views the pattern as a non-deterministic finite-state automaton; by eliminating the ϵ-transitions, one obtains a deterministic automaton, whose transitions form the "failure function". In the second approach, which is the one I would like to describe here, one begins with a naïve implementation of string searching, and by a series of program transformations, one derives an efficient algorithm, namely the Knuth-Morris-Pratt algorithm.

I found this idea in Ager, Danvy, and Rohde's paper, where it is somewhat buried in Appendix A. I rephrase it here in a slightly different way.

The naïve algorithm

Let us begin with a supposedly idiomatic Java implementation of naïve string searching. We are looking for an occurrence of a string, known as the "pattern", within another string, known as the "text". Throughout, I write m for the length of the pattern and n for the length of the text.

```
public static int search00 (String pattern, String text)
{
  final int m = pattern.length();
  final int n = text.length();
  loop: for (int k = 0; k + m <= n; k++) {
    for (int j = 0; j < m; j++)
      if (pattern.charAt(j) != text.charAt(k + j))
        continue loop;
    return k;
  }
  return -1;
}
```

The code returns the index in the text of the leftmost occurrence of the pattern, if one exists, and -1 otherwise. In the worst case, its time complexity is $O(mn)$, that is, quadratic.

Let us first write this algorithm in the form of a single infinite loop. This makes the control flow much simpler, and means that the current state of the algorithm is now entirely encoded in the variables j and k. (With respect to the previous code, the meaning of k has changed. Now, j is an index into the pattern, while k is an index into the text.)

```
public static int search01 (String pattern, String text)
{
  final int m = pattern.length();
  final int n = text.length();
  int j = 0, k = 0;
  while (true) {
    if (j == m)
      // End of the pattern reached: success.
      return k - j;
    if (k == n)
      // End of the text reached: failure.
      return -1;
    if (pattern.charAt(j) == text.charAt(k)) {
      // Character match. Advance both indices by one.
      k++;
      j++;
    }
    else {
      // Character mismatch. Backtrack.
      k = k - j + 1;
      j = 0;
    }
  }
}
```

The source of inefficiency is the last case, where we backtrack and (if j is greater than zero) the index k does not increase. This leads the algorithm to read again a part of the text that it has already read.

A pure formulation

Before attempting any algorithmic change, let us transform the code to a purely functional form, which is both more general and easier to reason about. We switch to OCaml, and instead of a loop, we write a tail-recursive function.

```
let rec search pattern m text n j k =
  if j = m then
    Found (k - j)
  else if k = n then
    Interrupted j
  else if pattern.[j] = text.[k] then
    search pattern m text n (j + 1) (k + 1)
  else
    search pattern m text n 0 (k - j + 1)
```

This is a function of six parameters, but the first four are invariant, so it costs nothing to carry them around – they are just stored in registers. The code produced by the OCaml compiler is about as tight as one might hope (try `ocamlopt -unsafe -S` and look at the assembly file).

I have taken this opportunity to enrich the result type of the `search` function. When the search fails, by reaching the end of the text, instead of returning just -1, it returns the current value of j. In other words, it returns the state of the algorithm at that point.

For the sake of completeness, let me show the definition of the type `result`, as well as a destructor that extracts `j` out of a result when we know that this result is `Interrupted _`. This destructor will be useful later on.

```
type result =
  | Found of int       (* the offset k where a match lies *)
  | Interrupted of int (* the state j that was reached *)

let assertInterrupted = function
  | Found _        -> assert false
  | Interrupted j -> j
```

In this form, the code looks almost identical to the previous version, which was written in Java. It is, however, much more powerful. Thanks to the extra parameters that it exposes and thanks to the extra information that it returns, the algorithm is now interruptible and resumable. In other words, a search within a long text can be split up as a sequential composition of two searches. If the index l lies between k and n (that is, $k \leq l \leq n$), then the call:

```
  search pattern m text n j k
```

is equivalent to the sequence:

```
  let result = search pattern m text l j k in
  match result with
  | Found _ ->
      result
  | Interrupted j' ->
      search pattern m text n j' l
```

The first call searches the text interval $[k, l)$, while the second call searches the interval $[l, n)$, beginning in the state j' that was reached by the first call.

Identifying and exposing the redundant work

It is now time to do something about the inefficiency of this algorithm. Recall that the root of the problem is the backtracking that takes place in the last case when j is greater than zero. It is stupid to go back (or stay where we are) in the text, because we already know what characters are there. We have just read these characters and successfully compared them with a prefix of the pattern!

To clarify what I mean, and to allow my claim to be tested, let me enrich the code with extra assertions that document our knowledge. (The function `sub` is `String.sub`; it extracts a substring out of a string.) In technical lingo, these assertions form a precondition of the function `loop`; in the second Java version above, they would form an invariant of the `while` loop.

```
let rec search pattern m text n j k =
  assert (0 <= j && j <= m);
  assert (j <= k && k <= n);
  assert (sub pattern 0 j = sub text (k - j) j);
  if j = m then
    Found (k - j)
  else if k = n then
    Interrupted j
  else if pattern.[j] = text.[k] then
    search pattern m text n (j + 1) (k + 1)
  else
    search pattern m text n 0 (k - j + 1)
```

The last recursive call is performing partly redundant work, because we already know what characters are found in the text interval $[k - j, k)$. Fortunately, we are now in a position to do something about it. Instead of searching the interval $[k - j + 1, n)$ in one go, let us split this search into the two intervals $[k - j + 1, k)$ and $[k, n)$. We will then be able to identify the first search as entirely redundant.

We set apart the sub-case where j is zero, as it does not pose a problem. Thus, the code becomes (assertions now removed, for brevity):

```
let rec search pattern m text n j k =
  if j = m then
    Found (k - j)
  else if k = n then
    Interrupted j
  else if pattern.[j] = text.[k] then
    search pattern m text n (j + 1) (k + 1)
  else if j = 0 then
    search pattern m text n 0 (k + 1)
  else
    let result = search pattern m text k 0 (k - j + 1) in
    match result with
    | Found _ ->
        result
    | Interrupted j' ->
        search pattern m text n j' k
```

We can immediately simplify this code a little bit. Indeed, the call `search pattern m text k 0 (k - j + 1)` searches for the entire pattern, which has length m, in a text interval of length $j - 1$, where $j < m$, so it cannot possibly succeed. Thus, the first branch of the last `match` construct is dead, and the whole construct can be replaced with a more concise call to `assertInterrupted`:

```
let rec search pattern m text n j k =
  if j = m then
    Found (k - j)
  else if k = n then
    Interrupted j
  else if pattern.[j] = text.[k] then
    search pattern m text n (j + 1) (k + 1)
  else if j = 0 then
    search pattern m text n 0 (k + 1)
  else
    let j' = assertInterrupted (search pattern m text k 0 (k - j + 1)) in
    search pattern m text n j' k
```

We now take a more essential step. As previously noted, the sub-strings `sub pattern 0 j` and `sub text (k - j) j` are equal. Thus, the call `search pattern m text k 0 (k - j + 1)` searches for the pattern within (a prefix of a suffix of) the pattern itself. As a result, this call can be rewritten to mention the pattern, instead of the text! It is equivalent to the call `search pattern m pattern j 0 1`. We have replaced `text` with `pattern`, and adjusted the indices into the text by subtracting $k - j$, so that they are now indices into the pattern. The code is now:

```
let rec search pattern m text n j k =
  if j = m then
    Found (k - j)
  else if k = n then
    Interrupted j
  else if pattern.[j] = text.[k] then
    search pattern m text n (j + 1) (k + 1)
  else if j = 0 then
    search pattern m text n 0 (k + 1)
  else
    let j' = assertInterrupted (search pattern m pattern j 0 1) in
    assert (j' < j);
    search pattern m text n j' k
```

The new assertion $j' < j$ is justified as follows. We are searching in a text fragment of length $j - 1$, and we begin in state 0, so, at best (if all characters match), this search will reach the final state $j - 1$.

The code is now roughly analogous to Figure 8 (page 42) of Ager, Danvy, and Rohde's paper. Our version is more compact, compared to theirs, as we have avoided some code duplication, but the idea is the same.

It is worth noting that, at this point, our code is not tail-recursive, due to the manner in which we perform two searches in sequence. This is not a problem, however, as our very next step (as I am sure you, the reader, have guessed) is to remove the call `search pattern m pattern j 0 1` entirely. Indeed, this value does not depend on the text in any way, so it can be precomputed before the search begins.

The search phase

Let us *assume*, for the time being, that we have precomputed the value of `search pattern m pattern j 0 1`, for every j in the range $0 < j < m$, and stored this information in an array `table`. (As noted above, for every j in this range, we have the property that `table.(j)` is less than j.) Then, the code can be simplified as follows:

```
let rec loop pattern m text n table j k =
  if j = m then
    Found (k - j)
  else if k = n then
    Interrupted j
  else if pattern.[j] = text.[k] then
    loop pattern m text n table (j + 1) (k + 1)
  else if j = 0 then
    loop pattern m text n table 0 (k + 1)
  else
    loop pattern m text n table table.(j) k
```

Cool. The code is tail-recursive again. This is cool. This is way cool. Wait a minute. Is this cool? A shadow of a doubt should cast itself in your mind.

Is this code efficient? We still seem to have a recursive call, on the last line, where k is not incremented. Thus, it is not obvious that this code has good time complexity. In fact, a rapid analysis indicates that, in the worst case, `table.(j)` could be $j - 1$, so, in the worst case, the algorithm could stutter m times (minus one or two) before it finally increments k. This leads to a worst-case upper bound of $O(mn)$, no better than the naïve algorithm.

Fortunately, this analysis is overly pessimistic. The variables j and k are linked: whenever j is incremented, k is incremented too. Thus, the worst-case scenario, where j has reached a high point and requires many steps to go down, arises only if k has been incremented as many times as j has been incremented. Thus, speaking in terms of amortization, the cost of backtracking can be charged, ahead of time, to the steps where k was incremented. Speaking in plain mathematics, it is not difficult to see that the quantity $2k - j$ grows strictly at every recursive call. Since this quantity is initially 0 and is bounded by $2n$, we find that the time complexity of the search phase is $O(n)$.

The phenomenon that we have identified, and that gave rise to a doubt, is that the algorithm is not real-time. In the worst case, it can take up to $O(m)$ steps to process one character of the text, even though, in an amortized sense (i.e., averaged over the entire text), processing one character only requires constant time.

The preprocessing phase

We now need to program the preprocessing phase, which initializes the array `table`. Recall that `table.(j)` is supposed to contain the integer value `assertInterrupted (search pattern m pattern j 0 1)`. How do we efficiently compute this information?

Of course one could invoke the inefficient `search` function at every `j`, and store the results. But this naïve approach would have time complexity $O(m^3)$, which is not great.

In order to do better, we use two simple ideas.

First, it is quite obvious that the desired value at j can be computed in terms of the desired value at $j-1$, simply by exploiting (again) the fact that a search can be split into two consecutive sub-searches. The base case, $j = 1$, corresponds to searching an empty interval; the desired value is then the initial state 0.

Second, instead of calling the inefficient `search` function, let us use the efficient `loop` function. Granted, `loop` reads the array `table`, which we have not fully initialized yet, but a moment's thought reveals that this is safe: the array will be read only at indices that have been computed already. (The proof is not difficult; let me skip it.) The code is as follows:

```
let init pattern m =
  let table = Array.create m 0 in
  for j = 2 to m - 1 do
    table.(j) <- assertInterrupted (
      loop pattern m pattern j table table.(j - 1) (j - 1)
    )
  done;
  table
```

This is a beautiful bootstrapping process: the same code, `loop`, is used in the preprocessing phase and in the search phase. We did not even consciously plan ahead for that, but the function was sufficiently general to begin with that it did not need to be modified. Thus, we re-use the code. We can re-use also its complexity argument, and assert that `init` has time complexity $O(m)$.

Packing it all up

I leave it as an exercise for the reader to put everything together and produce an implementation that satisfies the following signature:

```
(* An abstract type of compiled patterns. *)
type pattern

(* An abstract type of states. *)
type state

(* The initial state is used to begin a search. *)
```

```
val initial : state

(* A search result is either an offset in the text where the pattern
   was found, or a state, which means that the end of the text was
   reached in that state. *)
type result =
| Found of int
| Interrupted of state

(* [compile p] turns the pattern [p], represented as a string, into
   a compiled pattern. Its time complexity is linear in the size of
   the pattern. *)
val compile: string -> pattern

(* [find p s text k n] searches for the compiled pattern [p] in the
   string [text], between the offsets [k] included and [n] excluded. Its
   time complexity is linear with respect to [n - k].  A search normally
   begins in the [initial] state, but a state [s] returned by a previous
   search can also be used, if desired; this feature allows interrupting
   and resuming a search. *)
val find: pattern -> state -> string -> int -> int -> result
```

I should note that the algorithm that we have obtained is not quite the Knuth-Morris-Pratt algorithm. The latter includes one extra optimization, which I have not attempted to incorporate here. This optimization builds shortcuts in the failure table, and is implemented by modifying one line in the preprocessing phase, as follows:

```
let init pattern m =
  let table = Array.create m 0 in
  for j = 2 to m - 1 do
    let s = assertInterrupted (
      loop pattern m pattern j table table.(j - 1) (j - 1)
    ) in
    table.(j) <-
      if s > 0 && pattern.[s] = pattern.[j] then table.(s)
      else s
  done;
  table
```

As another exercise, the reader can translate the code back to imperative style (and to Java, if desired).

Related work

Chengdian (1985) and van der Woude (1989) derive an implementation of the Knuth-Morris-Pratt algorithm from its specification, expressed as a pre/postcondition pair. In other words, they construct, at the same time, the code and the proof of its correctness in Hoare logic. Again, this establishes that the code is correct, but does not (in my opinion) lead to a pedagogical explanation of the algorithm.

Bird, Gibbons, Jones (1989) and Bird (2005) use equational reasoning to transform a specification of the algorithm (that is, a simple but inefficient implementation) into an efficient one. Unfortunately, it is not obvious to a casual reader that the final algorithm is the Knuth-Morris-Pratt algorithm.

Ager, Danvy, and Rohde (2002) explain how, once the algorithm has been manually put under a suitable form, the standard techniques of partial evaluation can be used to specialize the code with respect to a fixed pattern. Thus, the partial evaluation phase is the preprocessing phase. Their paper provided the inspiration for this post. They also prove, in great detail, that the algorithm thus obtained is exactly the Knuth-Morris-Pratt algorithm.