Using Correctness-by-Construction to Derive Dead-zone Algorithms

Bruce W. Watson[1], Loek Cleophas[2], and Derrick G. Kourie[1]

[1] FASTAR Research Group, Department of Information Science, Stellenbosch University,
Private Bag X1, 7602 Matieland, Republic of South Africa
[2] Department of Computer Science, University of Pretoria,
Private Bag X20, 0028 Hatfield, Pretoria, Republic of South Africa
{bruce,loek,derrick}@fastar.org

Abstract. We give a derivation, in the form of a stepwise (refinement-oriented) presentation, of a *family* of algorithms for single keyword pattern matching, all based on the so-called *dead-zone* algorithm-style, in which input text parts are tracked as either unprocessed ('live'), or processed ('dead'). Such algorithms allow for Boyer-Moore-style shifting in the input in two directions (left and right) instead of one, and have shown promising results in practice. The algorithms are the more interesting because of their potential for concurrency (multithreading). The focus of our algorithm family presentation is on correctness-arguments (proofs) accompanying each step, and on the resulting elegance and efficiency. Several new algorithms are described as part of this algorithm family, including ones amenable to using concurrency.

Keywords: correctness-by-construction, algorithm derivation, keyword pattern matching, Boyer-Moore, concurrency

1 Introduction

In this paper, we give a stepwise derivation of a *family* of algorithms for single keyword pattern matching. The focus of the derivation is on clarity and confidence in the correctness of each step, which lead to efficiency and elegance. Because of the correctness arguments associated with each step, the presentation forms the essence of a *derivation* of the various algorithms. As such, the presentation forms a case study for the *Correctness-by-Construction* (CbC) approach to software or algorithm construction: we start with an abstract problem specification, in the form of pre- and postcondition, and iteratively refine these to obtain more refined (concrete) specifications [15]. In its strict form, the use of CbC requires that in each refinement step, rules are applied that guarantee and prove correctness of the resulting refined specification. For presentation clarity, and since many of the proof obligations involved are trivial, we will leave out many formal details in the present context. A CbC approach, and the derivations resulting from it, give a clear understanding of the algorithms and concepts involved, and give confidence in correctness, even if applied somewhat informally. Furthermore, new algorithms may arise from the process.

The main result of our presentation is the *derivation* of a substantial and partially new algorithm family, of which significant parts have not been explicitly presented before. In some sense, the algorithms in the family are reminiscent of the Boyer-Moore style of algorithms, in which *shift functions* are used to potentially skip reading parts of the subject string by moving to the right in the input text by *more than one* position. This has earned such algorithms the term *sublinear*.

The algorithm family we consider uses quite different algorithm skeletons from the Boyer-Moore style ones, although the latter form degenerate cases within our

Bruce W. Watson, Loek Cleophas, Derrick G. Kourie: Using Correctness-by-Construction to Derive Dead-zone Algorithms, pp. 84–95.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

algorithm family. It is based on characterizing positions in the text as either *(a)live* and part of a *live-zone* or *dead* and part of a *dead-zone*. A position or range of positions in the text is dead i.e. in a dead-zone, if (based on information obtained by an algorithm so far) it has been deemed to either not match the pattern/keyword or has been deemed to match the pattern and such a match has been reported or registered. Otherwise, it is live i.e. in a live-zone. Initially, the entire input text is live, and upon termination, the entire input text is dead. So-called dead-zone style algorithms such as those in the algorithm family presented here, can do better than Boyer-Moore style algorithms can: based on a single alignment of text to pattern, shifting (i.e. "killing" positions) can occur both to the right *and* to the left, i.e. yielding *two* remaining live text parts (and hence next alignments) to consider. Furthermore, these two parts can be processed independently, offering ample opportunity for concurrency and fundamentally distinguishes the dead-zone algorithm family from classical Boyer-Moore and most more recent single keyword pattern matching algorithms, which typically use a single sliding alignment window on the text. Dead-zone style algorithms can be used with any of the various Boyer-Moore style shift functions known from the literature, and they can even use left and right shift functions that are completely different (above and beyond the obvious mirroring required).

This algorithm family can be seen in the context of earlier work by the authors and their collaborators, based on the use of live-zones and dead-zones for keyword pattern matching. At the Second Prague Stringology Club Workshop (1997), Watson and Watson presented a paper on a new family of string pattern matching algorithms [19], with a later version appearing in 2003 in the South African Computer Journal [20]. In those papers, the focus is on representing liveness and deadness on a per-position basis. The use of ranges to represent liveness and deadness is only mentioned in passing, leading to a *recursive* range-based algorithm. Furthermore, this early work is not explicit about the use of Boyer-Moore style shift functions in the algorithms. At the International Workshop on Combinatorial Algorithms (IWOCA) 2012, Watson, Kourie and Strauss presented a slightly different recursive dead-zone-based algorithm with explicit use of Boyer-Moore style shift functions, as well as a C++ implementation and benchmarking [18]. This work was further extended by Mauch et al. [13], who present the recursive algorithm with some variations, including iterative ones obtained by (tail) recursion elimination. The focus there is on implementing and benchmarking these variants (in C and C++).

In this paper, we present and derive a family of iterative, range-based dead-zone algorithms, including different representation choices for the live-zones in the algorithms. As we will see, the family includes an alternative derivation of the recursive implicit-stack based algorithm used in [18,13]. Furthermore, various representation choices are considered, and various dead-zone-style algorithms using concurrency are presented and sketched, some of them new. The presentation is in the spirit of the CbC style pioneered by Dijkstra, Gries, and others over four decades ago [6,8,14,12]. For this reason, we assume some awareness of this style and we use the well-known Guarded Command Language (GCL; designed by Dijkstra) which was designed to support reasoning about correctness. The presentation starts out from a single abstract algorithm whose correctness is easy to argue based on the formulation of pre- and post-conditions and invariant. Using the CbC approach, this algorithm is then iteratively refined, leading to the derivation and presentation of the family of algorithms.

2 Problem Statement and Initial Solution Sketch

The single keyword pattern matching problem can be stated as follows:

> Given two strings $x, y \in \Sigma^*$ over an alphabet Σ—respectively called the *keyword* or *pattern*, and *(input or search) text*—find all occurrences of x as a contiguous substring of y.

We register any such occurrences or matches by keeping track of the indices in the text at which they occur, i.e. at which they *start*, in a *match set* variable, called *MS* (a set of integers)[1]. To make this more precise, we define a *helper predicate*

$$Match(x, y, j) \equiv (x = y_{[j,j+|x|)}).$$

Note that here and throughout this derivation, we use $[a, b)$ style ranges (inclusive at the left end, exclusive at the right end) to avoid numerous $+1$ and -1 terms in indexing a string.

Using predicate *Match*, the postcondition to be established by any algorithm solving the single keyword pattern matching problem therefore is:

$$MS = \bigcup_{j \in [0,|y|): Match(x,y,j)} \{j\}$$

Note that indexing in y starts from 0, in keeping with many programming languages as well as typical use in correctness-by-construction styles.

Also note that the above does not place any restrictions on x, y; in particular, y may be shorter than x (in which cases there are no matches), and one or both could be of length 0 (and in case x does, it trivially matches at every position of y). Many algorithm presentations in the literature needlessly give such restrictions as part of the problem statement—thereby cluttering the resulting algorithm(s).

As in the earlier derivations and presentations of dead-zone style algorithms mentioned in the introduction, our algorithms will keep track of all indices in y (i.e. members of the set $[0, |y|)$, categorizing each such index k into one of a number of disjoint sets. Here, we will use three (disjoint) sets, although one of them will not be represented explicitly:

1. *MS*, the set of indices where a match has already been found.
2. *Live_Todo*, the set of indices about which we know nothing (i.e. there may or may not be a match at such an index), i.e. that are still *live*.
3. $\neg(MS \cup Live\_Todo)$, the set of indices at which we *know* no match occurs.

The indices in category 2 are called *live* indices, while those in category 1 and 3 are called *dead*. In earlier dead-zone-style algorithms as presented in [19,20], two disjoint sets were used, representing the live (or to do) and dead (or done) indices respectively, with dead indices at which a match occurs being reported as soon as they are found.

The aim of every algorithm in this paper's algorithm family is to start with $Live\_Todo = [0, |y|)$ and reduce this to $Live\_Todo = \emptyset$, meaning that all indices have been checked and are in category 1 or 3; in other words, each such algorithm starts with the entire input text y being *live*, and ends with the entire input being *dead*. This is expressed by the predicates and algorithm skeleton given below.

[1] Of course, in practical implementations, we may print a message to the screen or otherwise report a match instead of accumulating the variable *MS*.

Algorithm 1 (Abstract Live and Dead Indices Matcher)
Live_Todo :=$[0, |y|)$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge(\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg\boldsymbol{Match}(x, y, j))$ }
{ **variant:** $|\boldsymbol{Live\_Todo}|$ }
S
{ **invariant** $\wedge\, |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

Clearly, given an appropriate statement S satisfying the above pre- and postconditions for S, when $|Live\_Todo| = 0$, $Live\_Todo = \emptyset$ and hence the previously mentioned postcondition holds, i.e. $MS = \bigcup_{j \in [0,|y|): Match(x,y,j)} \{j\}$.

3 From Position-based to Range-based Iterative Dead-zone

We can immediately make a practical improvement to Algorithm 1 by representing *Live_Todo* as a *set of (live) ranges* $[l, h)$ (with l, h integers). At first glance this may seem less efficient, but we will benefit from this shortly. As a further conjunct to the original invariant, we also insist that the ranges in *Live_Todo* are pairwise disjoint, i.e. none of the ranges overlap with each other[2]. Furthermore, the ranges are assumed to be maximal, to keep *Live_Todo* small (although ranges may be empty). With this minor change of representation, we need to be clear about what $|Live\_Todo|$ in the variant means: we still use it to refer to the total number of *indices* represented in *Live_Todo*, not the number of ranges in it. The resulting new algorithm skeleton is given below.

Algorithm 2 (Abstract Live and Dead Ranges Matcher)
Live_Todo := $\{[0, |y|)\}$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge\ (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg\boldsymbol{Match}(x, y, j))$ }
{ **variant:** $|\boldsymbol{Live\_Todo}|$ }
do *Live_Todo* $\neq \emptyset \rightarrow$
  Extract some $[l, h)$ *from Live_Todo*;
  S_0
od
{ **invariant** $\wedge\, |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

The question now becomes what needs to be done in S_0 to re-establish the invariant. Clearly some or all of the indices in the range $[l, h)$ need to be checked to gain information about matches found or found to be impossible. Rather than check all indices in the range, we check for a match at the midpoint of the range, i.e. at

[2] Not doing so would lead to inefficiency by considering a live position more than once, or extra booking-keeping to eliminate positions which have already been considered

$m = \lfloor \frac{l+h}{2} \rfloor$ and split the range in two, inserting the remaining portions into *Live_Todo*, i.e. adding $[l, m)$ and $[m+1, h)$ to *Live_Todo*. Note that as indicated above, this does not make variant $|Live\_Todo|$ increase. It should also be noted that other choices than the midpoint of the range could be made, including choices that have the algorithm degenerate to e.g. Boyer-Moore [19], but we will not explore these here.

Either or both of $[l, m)$ and $[m+1, h)$ may be empty ranges. We can detect this before insertion and not insert the empty range(s) into *Live_Todo*. Here, we elect to do such range checking upon extraction of a range from *Live_Todo*, cutting the amount of pseudo-code. However, whenever an empty range is processed in the loop, the earlier variant $|Live\_Todo|$ does not strictly decrease but stays the same. Because of this, we change the variant to the pair $\langle |Live\_Todo|, E \rangle$ with E the number of empty ranges in *Live_Todo*. Using the standard lexicographical ordering on such pairs, this is once more a correct variant: it decreases not only when $|Live\_Todo|$ decreases, but also when $|Live\_Todo|$ stays the same yet an empty range is extracted from *Live_Todo* (since in that case E decreases).

Finally, we note that the last $|x|-1$ indices of input text y cannot contain a match, and we change the initialization of y accordingly. (Note that this could already have been done before, i.e. in Algorithms 1 and 2.)

The above refinements give us Algorithm 3.

Algorithm 3 (Live and Dead Ranges Matcher)
$Live\_Todo := \{[0, |y| - |x|)\}$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg \boldsymbol{Match}(x, y, j))$ }
{ **variant:** $\langle |\boldsymbol{Live\_Todo}|, E \rangle$ }
do $Live\_Todo \neq \emptyset \rightarrow$
    Extract $[l, h)$ *from Live_Todo*;
    if $l \geq h \rightarrow$ { *empty range* } **skip**
    [] $l < h \rightarrow$
        $m := \lfloor \frac{l+h}{2} \rfloor$;
        if $Match(x, y, m) \rightarrow MS := MS \cup \{m\}$
        [] $\neg Match(x, y, m) \rightarrow$ **skip**
        fi;
        $Live\_Todo := Live\_Todo \cup [l, m) \cup [m+1, h)$
    fi
od
{ **invariant** $\wedge\ |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

---

4 Improvements to Range-based Iterative Dead-zone

The preceding algorithm repeatedly performs match attempts, i.e. tests predicate *Match*. Testing this predicate boils down to a loop testing the symbols of x one-for-one against $y_{[m,m+|x|)}$. Such match attempts can be done letter-by-letter from right to left or vice versa, but also in parallel, or using a different order. The particular order used has typically been called the *match order*, with extensive discussions of different orders by Hume & Sunday [11] and elsewhere [16,3].

As with all Boyer-Moore style algorithms, we can eliminate more than one index at a time, i.e. not just index m but consecutive indices next to it as well, by cleverly using information gathered during the match attempt. Shifting more than one index at a time after this match attempt is usually done using a precomputed shift function. We do not consider the details of such "shifters" here. They are extensively covered in the literature [2,9,11], including with correctness arguments in [21,5]. Given that any such shift function is usable with the above algorithm skeleton, the resulting skeleton (not explicitly given here) in fact represents an extensive family of algorithms, even without considering the preceding abstract algorithms or the ones that follow in Section 5.

Since match attempts are made near the middle of a selected range however, our algorithm skeleton, in contrast to classical Boyer-Moore and all its variations, can use two shifters at the same time: one to shift to the right (as per Boyer-Moore and variants), increasing the left bound of range $[m+1, h)$; and a dual one to the left, decreasing the upper bound of range $[l, m)$. Although such left shifters have not been described in detail in the literature, they are in fact straightforwardly computed "duals" of the right shifters, as was pointed out in [18]. It is important to note that it is not even necessary to use a shifter and its dual shifter in the other direction, but that a shifter and the dual of a completely different Boyer-Moore style shifter can be used—thus, the algorithm family is even more extensive than it may seem at first glance. For example, a family member could use Horspool's shifter for its right shifts, and the dual of Sunday's shifter for its left shift, etc.

If we assume two such shift functions, returning values *shl* and *shr* for shift left and shift right, the update of *Live_Todo* near the bottom of Algorithm 3 can be replaced by

$$
\begin{array}{l}
l', h' := m - shl, m + shr; \\
Live\_Todo := Live\_Todo \cup [l, l') \cup [h', h);
\end{array}
$$

We can do a relatively simple running-time analysis, which relies on prior knowledge about the shift functions: they are both bounded above by $|x|$ in most cases[3] meaning we could make as few as $\left\lceil \frac{|y|}{2|x|} \right\rceil$ match attempts in the best case. This contrasts to $\left\lceil \frac{|y|}{|x|} \right\rceil$ match attempts for Boyer-Moore and variants. Note that this does not violate any information-theoretical bounds; in each match attempt, character comparisons from both the left and the right end of the current alignment are made, and following each match attempt, our family of algorithms simply shifts in two directions instead of one. In the best case, just like for the Boyer-Moore algorithm and variants, we have $\left\lceil \frac{|y|}{|x|} \right\rceil$ character comparisons. The worst case for this family, as with all keyword pattern matching variants, is $|y|$ match attempts, and $|y| * |x|$ character comparisons.

[3] The exception is with shift functions as used in e.g. the Berry-Ravindran algorithm [1], which uses characters next to x's current alignment in y as well. In such cases, the shift function is bounded by $|x| + c$ for some constant c.

5 Different Live-zone Representations

Until now, *Live_Todo* has been a set, meaning there is a measure of nondeterminism in the algorithms presented. This can lead to very poor performance in practice, in cases where the algorithm needs read access intermittently all across y.

Alternatively to a set representation, *Live_Todo* can easily be represented using a queue or using a stack. Predictably, these lead to breadth- respectively depth-wise traversals of the ranges in y.

As noted before, the ranges in *Live_Todo* are pairwise disjoint. This also means we can deal with them entirely independently and in parallel. Indeed, each iteration could give rise to new threads, leading to various concurrent versions of the algorithms presented. Two recent papers, [10] and [7], discuss approaches that have some similarities to this; they also process multiple text segments at the same time, but this is limited to two segments in the former, and a fixed number of segments in the latter. In [7], the fixed number of segments is strongly reflected in the structure of the source code, with some coupling between the segments' processing, and no obvious way of making the processing multi-threaded.

In the worst case, a queue can grow to $|y|$ and in the best case to $\left\lceil \frac{|y|}{|x|} \right\rceil$ elements (ranges). Because of this worst case behaviour, we do not further consider this representation choice in the current paper.

A stack representation is much more efficient than a queue one, giving a maximum stack size of $log_2|y|$. The resulting algorithm is a relatively straightforward refinement of (the bidirectional shifting-based refinement) of Algorithm 3, and is given below.

Algorithm 4 (Live and Dead Ranges Matcher using Stack)

$Live\_Todo := \langle [0, |y| - |x|) \rangle$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg \boldsymbol{Match}(x, y, j))$ }
{ **variant:** $\langle |\boldsymbol{Live\_Todo}|, E \rangle$ }
do $Live\_Todo \neq \emptyset \rightarrow$
    Pop $[l, h)$ *from Live_Todo*;
    if $l \geq h \rightarrow$ { *empty range* } **skip**
    $[\!]$ $l < h \rightarrow$
                $m := \left\lfloor \frac{l+h}{2} \right\rfloor$;
                if $Match(x, y, m) \rightarrow MS := MS \cup \{m\}$
                $[\!]$ $\neg Match(x, y, m) \rightarrow$ **skip**
                fi;
                $l', h' := m - shl, m + shr$;
                Push $[h', h)$ *to Live_Todo*;
                Push $[l, l')$ *to Live_Todo*
    fi
od
{ **invariant** $\wedge\ |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

This algorithm in fact is an encoding with an explicit stack of a recursive dead-zone-style algorithm, similar to the ones presented in [18,13].

Algorithm 4 can be further refined to yield a version with information sharing; that is, given that zones may overlap, and since the algorithm considers zones from left to right, we can track a *known dead-zone* $[0, z)$ with z such that $(\forall\, i : 0 \le i < z : i \notin \mathit{Live\_Todo})$, and keep z maximal given what is presently known about the input text based on the algorithm's preceding processing.

Given such a value z, whenever we pop $[l, h)$ from the stack, it can be changed to $[l \max z, h)$, followed by updating z to $l \max z$. These changes are correct due to additional invariants added below, as well as the order in which ranges are pushed onto the stack, which ensures that whenever $[l, h)$ is popped from the stack, all indices to the left of l are already dead.

Algorithm 5 (Live and Dead Ranges Matcher using Stack and Sharing)

$\mathit{Live\_Todo} := \langle [0, |y| - |x|) \rangle$;
$\mathit{MS} := \emptyset$;
$z := 0$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge\, (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg \boldsymbol{Match}(x, y, j))$ }
{ $\wedge\, (\forall\ i : 0 \le i < z : i \notin \boldsymbol{Live\_Todo})$ }
{ $\wedge\, (\boldsymbol{top}(\boldsymbol{Live\_Todo}) = [l, h)) \implies [0, l)$ **is dead** }
{ **variant:** $\langle |\boldsymbol{Live\_Todo}|, E \rangle$ }
do $\mathit{Live\_Todo} \neq \emptyset \rightarrow$
  Pop $[l, h)$ *from Live_Todo*;
  $l := l \max z$;
  $z := l$;
  if $l \ge h \rightarrow$ { *empty range* } **skip**
  $[\!]$ $l < h \rightarrow$
    $m := \left\lfloor \frac{l+h}{2} \right\rfloor$;
    if $\mathit{Match}(x, y, m) \rightarrow \mathit{MS} := \mathit{MS} \cup \{m\}$
    $[\!]$ $\neg \mathit{Match}(x, y, m) \rightarrow$ **skip**
    fi;
    $l', h' := m - \mathit{shl}, m + \mathit{shr}$;
    Push $[h', h)$ *to Live_Todo*;
    Push $[l, l')$ *to Live_Todo*
  fi
od
{ **invariant** $\wedge\, |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

---

6 Using Concurrency for Match Attempts

The use of concurrency as suggested in the preceding section is but one possible use of concurrency in dead-zone-style algorithms. We consider a different, previously undescribed, use of concurrency below. The core observation that leads to this use of concurrency is that for certain shift functions, very little match attempt information is used to determine the shifts to be made (while often still providing competitive performance compared to other shift functions). This means that shifting and match attempt verification can be decoupled to some degree.

For example, Horspool's variant of Boyer-Moore uses shifts based only on the rightmost symbol of the input at the given keyword alignment (and in our context, dually uses the leftmost symbol for the left shifter). Similarly, Sunday's Quicksearch uses shifts solely based on the symbol to the right of that rightmost symbol; and in our context, dually based on the symbol to the left of the leftmost symbol of the alignment for the shift to the left. Thus, it is not necessary to attempt a complete match i.e. to fully evaluate predicate $Match(x, y, m)$ before such shifts can be calculated.

For a given alignment of the pattern to the input text, an alternative algorithm could read the two symbols needed for the respective right shift and left shift (one for the right, one for the left shift), and immediately compute the shifts and make appropriate updates to *Live_Todo*. The current index, at which a match attempt still has to be performed (i.e. for which predicate *Match* has to be evaluated), can be added to a separate set variable, say *Attempt*. The elements of this set can be processed after the main loop of the algorithm has terminated, yielding a different but still sequential dead-zone-style algorithm.

Instead of such a sequential algorithm, a version employing concurrency could be used: one or more separate "matcher" threads could repeatedly extract an element from *Attempt* and perform the match attempt for the alignment indicated by the element. Alternatively, each thread t can have its own thread-private set $Attempt_t$ from which to extract elements, with the main algorithm loop adding positions to the various threads's $Attempt_t$. We present the latter algorithm, including the procedure $Matcher_t$ to be executed by each of the matcher threads $t \in MThreads$. Note that the second conjunct of the invariant has been changed to cover the $Attempt_t$ variables, reflecting the fact that removal of an element from *Live_Todo* does not necessarily mean it represents a non-match, as long as the match attempt for the element still needs to be executed (i.e. the element is in one of the sets $Attempt_t$).

Algorithm 6 (Concurrent Live and Dead Ranges Matcher using Stack)

$Live\_Todo := \langle [0, |y| - |x|) \rangle$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge\ (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo} \cup \bigcup_{t \in \boldsymbol{MThreads}} \{\boldsymbol{Attempt}_t\}) : \neg \boldsymbol{Match}(x, y, j))$ }
{ **variant:** $\langle |\boldsymbol{Live\_Todo}|, E \rangle$ }
do $Live\_Todo \neq \emptyset \rightarrow$
  Pop $[l, h)$ *from Live_Todo*;
  if $l \geq h \rightarrow$ { *empty range* } **skip**
  [] $l < h \rightarrow$
    $m := \lfloor \frac{l+h}{2} \rfloor$;
    Add m *to queue* $Attempt_t$ *for some thread* t;
    $l', h' := m - shl, m + shr$;
    Push $[h', h)$ *to Live_Todo*;
    Push $[l, l')$ *to Live_Todo*
  fi
od
{ **invariant** $\wedge\ |\boldsymbol{Live\_Todo}| = 0 \wedge (\forall\ t : t \in \boldsymbol{MThreads} : \boldsymbol{Attempt}_t = \emptyset)$ }
{ **post** }

proc $Matcher_t$

```
    do Attempt_t ≠ ∅ →
        if Match(x, y, m) → MS := MS ∪ {m}
        [] ¬Match(x, y, m) → skip
        fi
    od
corp
```

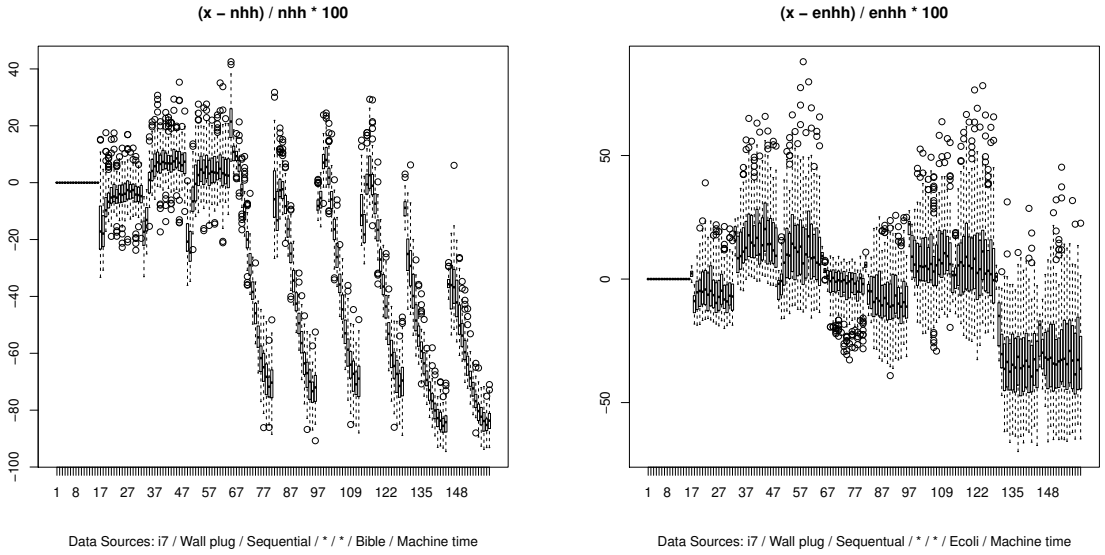
This algorithm assumes each matcher thread has write access to the shared match set variable *MS*. In practice, it is likely that an implementation would instead have a thread-local match set variable, and these would be gathered in shared memory after thread termination.

7 Concluding Remarks

We have given a derivation, in the form of a stepwise presentation, of an extensive family of single keyword pattern matching algorithms. Our exposition serves as a case study for the Correctness-by-Construction style of algorithm development, emphasizing correctness and clarity, which in turn lead to elegant and presumably efficient algorithms. Our presentation included several new algorithm variants, a result typically seen when using the correctness-by-construction style of algorithm derivation [17,4,15].

The algorithms in the family are all based on tracking input text parts as being dead-zones or live-zones, and using shifts to both the left and the right after an alignment and match attempt in the middle of a remaining text part to be considered. This approach offers ample opportunity for concurrency and fundamentally distinguishes the dead-zone algorithm family from classical Boyer-Moore and most more recent single keyword pattern matching algorithms, which typically use a single sliding alignment window on the text.

Previous publications [13,18] and ongoing benchmarking have already shown promising results for some of the members of the algorithm family as well as closely related dead-zone-based algorithms. Figure 1 shows some recent results. The details are not of particular interest here, but Figures 1a and 1b show the differences between each of 10 algorithms—8 dead-zone variants, followed by plain non-dead-zone Horspool and QuickSearch at the right of each figure—when run on Bible and Ecoli data respectively. The box plots reflect results over keyword lengths 2^i for $i \in [1, 16]$ with 100 pseudo-randomly generated keywords of each length—thus, each figure has $10 * 16 = 160$ box plots over 100 values. The results are relative (in percentage terms) to the first (leftmost) algorithm, an iterative dead-zone algorithm using Horspool's shifter (and dual). The results show current dead-zone implementations to already be close to competitive against plain non-dead-zone Horspool and QuickSearch, depending on the data sets pattern length under consideration. Since the new dead-zone algorithms derived and discussed in this paper include ones amenable to using concurrency (multithreading) in various ways, they are all the more interesting to explore further as future work.



(a) On bible data (b) On Ecoli data

Figure 1: Comparisons of relative performance of various dead-zone algorithms and plain Horspool and QuickSearch.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Club Workshop'99, J. Holub and M. Simánek, eds., Czech Technical University, Prague, Czech Republic, 1999, pp. 16–26, Collaborative Report DC-99-05.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 62–72.
3. D. Cantone and S. Faro: *Improved and self-tuned occurrence heuristics*, in Proceedings of the Prague Stringology Conference 2013, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2013, pp. 92–106.
4. L. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Eindhoven University of Technology, the Netherlands, Apr. 2008.
5. L. Cleophas, B. W. Watson, and G. Zwaan: *A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms.* Science of Computer Programming, 75 2010, pp. 1095–1112.
6. E. W. Dijkstra: *A Discipline of Programming*, Prentice Hall, 1976.
7. S. Faro and T. Lecroq: *A multiple sliding windows approach to speed up string matching algorithms*, in SEA, R. Klasing, ed., vol. 7276 of Lecture Notes in Computer Science, Springer, 2012, pp. 172–183.
8. D. Gries: *The Science of Computer Programming*, Springer-Verlag, second ed., 1980.
9. R. N. Horspool: *Practical fast searching in strings.* Software — Practice & Experience, 10(6) 1980, pp. 501–506.
10. A. Hudaib, R. Al-Khalid, D. Suleiman, M. Itriq, and A. Al-Anani: *A fast pattern matching algorithm with two sliding windows (TSW).* Journal of Computer Science, 4(5) 2008, pp. 393–401.
11. A. Hume and D. Sunday: *Fast string searching.* Software — Practice & Experience, 21(11) 1991, pp. 1221–1248.
12. D. G. Kourie and B. W. Watson: *The Correctness-by-Construction Approach to Programming*, Springer Verlag, 2012.
13. M. Mauch, D. G. Kourie, B. W. Watson, and T. Strauss: *Performance assessment of dead-zone single keyword pattern matching*, in SAICSIT Conf., J. H. Kroeze and R. de Villiers, eds., ACM, 2012, pp. 59–68.
14. C. C. Morgan: *Programming from specifications, 2nd Edition*, Prentice Hall International series in computer science, Prentice Hall, 1994.

15. B. Watson, D. Kourie, and L. Cleophas: *Experience with correctness-by-construction.* Science of Computer Programming, 2013, in press.
16. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, Sept. 1995.
17. B. W. Watson: *Algorithms for Constructing Minimal Acyclic Deterministic Finite Automata*, PhD thesis, Department of Computer Science, University of Pretoria, South Africa, 2010.
18. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in IWOCA, S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 236–248.
19. B. W. Watson and R. E. Watson: *A new family of string pattern matching algorithms*, in Proceedings of the Second Prague Stringologic Workshop, J. Holub, ed., Prague, Czech Republic, July 1997, Czech Technical University, pp. 12–23.
20. B. W. Watson and R. E. Watson: *A new family of string pattern matching algorithms.* South African Computer Journal, 30 June 2003, pp. 34–41.
21. B. W. Watson and G. Zwaan: *A taxonomy of sublinear multiple keyword pattern matching algorithms.* Science of Computer Programming, 27(2) 1996, pp. 85–118.