# Parameterized Pattern Matching by Boyer-Moore-type Algorithms

Brenda S. Baker*

**Abstract**

This paper investigates generalizations of the Boyer-Moore string pattern matching algorithm to parameterized pattern matching. Parameterized pattern matching was invented by Baker [Bak93b] for the purpose of finding sections of code in a software system that are the same except for a systematic change of parameters. We show that for Boyer-Moore-type algorithms that do not save information about previously matched portions of text, straightforward generalizations to parameterized pattern-matching must have a running time of $\Omega(n \min(m,p))$, where $n$ is the text length, $m$ is the pattern length, and $p$ is the number of parameter symbols in the alphabet. However, we describe a parameterized pattern matching algorithm PturboBM that has the same overall structure as the Boyer-Moore algorithm but saves information about previously matched portions of text and runs in time $O(n \log min(m,p))$, with preprocessing time $O(m \log min(m,p))$, where $n$ is the length of the text, $m$ is the length of the pattern, and $p$ is the number of distinct parameter symbols. Experiments show that the PturboBM algorithm has promise as an efficient means of accomplishing parameterized pattern matching for applications.

## 1 Introduction.

This paper investigates generalizations of the Boyer-Moore algorithm to parameterized pattern matching. Parameterized matching was invented by the author [Bak92, Bak93a, Bak93b, Bak] for the purpose of finding sections of code in a software system that are the same except for a systematic change of parameters. Parameterized matching involves *parameterized strings*, or *p-strings*, which are strings over two alphabets: an alphabet $\Pi$ of parameters and an alphabet $\Sigma$ of non-parameters. P-strings $P$ and $Q$ are a *parameterized match* or *p-match* iff there is a bijection (renaming) between their parameter symbols that transforms one string into the other. Since the p-match relation is an equivalence relation, we write $P \equiv Q$ ($P$ is p-equivalent to $Q$) to denote that $P$ and $Q$ are a p-match. For example, if $x$, $y$, $u$, and $v$ are parameters and $a$ and $b$ are non-parameters, then $yxaxyb \equiv vuauvb$, where $x$ and $y$ are mapped into $u$ and $v$, respectively. The *parameterized pattern matching problem* is the problem of determining for a pattern p-string $P$ and a text p-string $T$ which substrings of $T$ p-match $P$.

Our ultimate goal is to develop a software tool to aid in software maintenance by enabling a programmer to search quickly for other sections of code that are either identical or the same except for a change of parameters. Code copying still occurs in large software systems developed in recent years, despite the well-known danger of bug fixes being made in one copy of a piece of code but not in all copies. However, such copies have frequently had variables renamed [Bak93b], so that they may not easily be found by means of standard string matching algorithms. The algorithms developed in this paper are intended to enable a programmer to find other occurrences of a bug being fixed even if the variable names have been changed.

Our goal is somewhat different from the original goal in [Bak93b] of finding all parameterized duplication over threshold length in a software system, and consequently faster algorithms are possible. The algorithms of [Bak93b] were based on a generalization called *p-suffix trees* of the suffix tree data structure. While p-suffix trees are suited to the problem of looking for all pairs of p-matches in the input, the overhead involved in building p-suffix trees suggests looking for other approaches to parameterized pattern matching.

For strings, the Boyer-Moore (BM) algorithm [BM77] and its variants are the algorithms of choice for pattern matching [Hor80, HS91, Sun90]. Most of these algorithms share the common characteristic of comparing pattern with text from right to left, and when a mismatch occurs, using some form of shift rule to determine how far to the right to shift the pattern before comparing it again with the text. However, the form of the shift rule varies in different variants. The BM algorithm (with the modification proposed by Knuth et al. [KMP77]) runs in linear worst-case time [KMP77, Col91], like the Knuth-Morris-Pratt (KMP) algorithm [KMP77]. However, since the BM algorithm does not need to read every text symbol, it usually runs much faster than the KMP algorithm, which must read every text symbol, and in particular runs three or more

*AT&T Bell Laboratories,600 Mountain Avenue, Murray Hill, NJ 07974, bsb@research.att.com

times faster on English text [Sun90]. The expected performance of certain variants of the BM algorithm can be shown to be sublinear under various assumptions [BM77, KMP77, Sch88, BYGR90, BYR92].

Unexpectedly, the BM algorithm has poor worst-case performance when generalized to p-strings. We show that if a generalization of the BM algorithm preserves the characteristic of the original BM algorithm that once the pattern is shifted, no information is retained about which symbols were looked at previously, then there are examples for which the generalization uses time $\Omega(n \min(m, p))$. Here, and throughout the paper, $n$ denotes the length of the text, $m$ denotes the length of the pattern, $p$ denotes the number of distinct parameter symbols ($|\Pi|$), and $s$ denotes the number of distinct non-parameter symbols ($|\Sigma|$).

Hence, we turn to the *TurboBM* algorithm [CLC+92], which is a variation on the Boyer-Moore algorithm. The original TurboBM algorithm retains two pointers to the portion of the pattern matched in the previous placement and also incorporates an additional shift rule called the turbo shift; it runs in worst-case time linear in text length, independent of pattern length and alphabet size. We derive a generalization to p-strings that we call the PturboBM algorithm, and show that under a comparison model it runs in time $O(n \log \min(m, p))$ and space $O(n)$, with preprocessing time $O(m \log \min(m, p))$. These bounds are the same as for a generalization of the KMP algorithm to p-strings [AFM94, IS94] and are optimal [AFM94].

However, in experiments, for sufficiently long patterns and a variety of choices of alphabet sizes, an implementation of PturboBM performed better than a generalization of the KMP algorithm on random input under a uniform distribution. Careful experiments on input obtained from software have not yet been performed, but a few preliminary trials on software show both algorithms to be practical.

The comparison-based bounds for PturboBM are substantially better than those for algorithms based on p-suffix trees [Bak93b] for variable alphabets. If a p-suffix tree is used to encode the text, p-matching can be accomplished in time $O(m \log(s + p) + k)$, with $O(n(p + \log(s + p)))$ preprocessing time using a practical algorithm or $O(n \log n)$ preprocessing time using auxiliary data structures. If a p-suffix tree is used to represent the pattern, p-matching can be accomplished in time $O(n(p + \log(s + p))$ using a practical algorithm or time $O(n \log(s + p))$ using a complicated algorithm with auxiliary data structures, with $O(m(p + \log(s + p)))$ preprocessing time using a practical algorithm or $O(m \log m)$ preprocessing time using auxiliary data structures.

Deriving the PturboBM algorithm from the TurboBM algorithm is not straightforward because p-strings do not satisfy some of the properties of strings utilized by the TurboBM algorithm. First, whereas the TurboBM algorithm allows processing to jump over portions of the pattern and text known to match from the previous pattern placement, without examining those pattern and text symbols again, PturboBM must look for possible inconsistencies in parameter correspondences that may permit a jump only over part of the previously matched pattern and text. Identifying the inconsistencies may require peeking again at text symbols scanned in the previous placement in order to determine new parameter correspondences resulting from the shifting of the pattern. Second, a clever book-keeping scheme is needed to compute the length of the jump in time proportional to the symbols scanned in the current alignment. Third, in order to get a worst-case bound within a log factor of linear in text size, we introduce a turboBM shift rule that is a hybrid of the BM shift and Turbo shift rules. The proof of the time bounds is based on showing that large local costs are balanced by sufficiently large shifts later, thanks to the relationship of jumps and shifts.

## 2 Straightforward Generalizations of the BM algorithm.

The BM algorithm tries alignments of $P$ with $T$, and for each such alignment, compares the symbols of $P$ from right to left with the aligned symbols of $T$. If no mismatch is found for an alignment, a match is reported. When a mismatch is found or all of $P$ has been matched, the pattern is shifted to the right by an amount determined from the position of mismatch, the mismatched character, and some tables based on $P$ alone.

A straightforward generalization of the BM algorithm to p-strings would have the same structure, except that rather than comparing individual symbols of pattern and text right to left, it would determine whether each new pair of symbols of pattern and text are consistent with parameter pairings of the symbols just scanned to the right. That is, at position $j$ of $P$ aligned with position $i + j$ of $T$, it would determine whether $P[j, m]$ p-matches $T[i + j, i + m]$, possibly by comparing $P[j]$ and $T[i + j]$ with a data structure containing the parameter pairings to the right. If no mismatch is found, it would report a p-match. Finally, it would determine how far to shift the pattern to the right from the mismatched position $j$ of $P$, $P[j, m]$, and $T[i + j, i + m]$, and tables based on these. However, it would not retain any information about $T[i + j, i + m]$ after the shift.

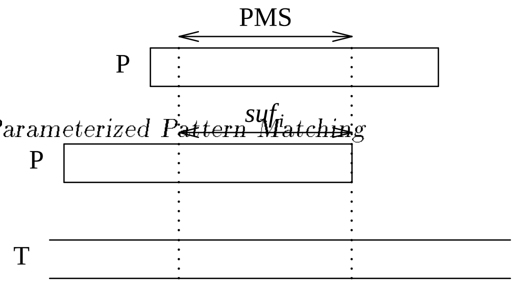


Figure 1: The PMS when $P$ has not been shifted past the left end of the previous longest matching part of $T$. The dotted lines delimit the longest matching part of $P$ and $T$ in the first alignment.

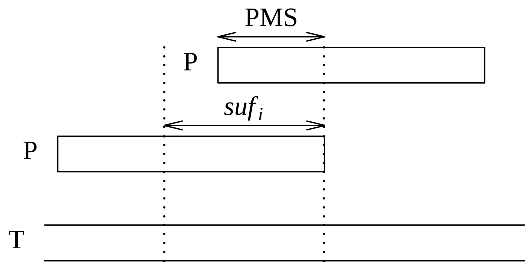


Figure 2: The PMS when $P$ has been shifted past the left end of the previous longest matching part of $T$.

Call an algorithm a memory-less pBM algorithm if it satisfies this design.

PROPOSITION 2.1. *Excluding preprocessing, in the worst case every memory-less pBM algorithm uses $\Omega(n \min(m,p))$ comparisons between pattern and text to determine whether any p-match exists between the pattern and the text, where $n$ is the text length, $m$ is the pattern length, and $p$ is the number of parameters.*

*Proof.* First, for $m-1 < p$, consider patterns of the form $x_1x_2...x_{m-1}x_1$, where the $x_i$'s are distinct parameter symbols, and texts that cycle through $p = m$ distinct parameter symbols repeatedly. For each alignment of $P$ with $T$, the pattern can be shifted right by only one position because of possible texts in which the next symbol would match the one $m-1$ positions to the left.

Next, suppose $p < m$ and $m \equiv k \pmod p$. Let $P = x_{p-k+2} \dots x_p(x_1x_2...x_p)^j x_2$ and $T = (x_1x_2 \dots x_p)^i$ for some $i > j$. At each alignment of $P$ with $T$, a mismatch is detected after reading $p$ symbols right to left. $P$ can be shifted by only one position, because (in some other text) the text symbol in that position might match the one $p-1$ symbols to the left.

In both of these examples, since the algorithm does not remember the symbols of $T$ seen before a shift, it must process symbols one by one from the right until a mismatch is found, thus producing the bounds of the proposition. □

The proposition implies that straightforward generalizations of the BM algorithm will require $\Omega(n \min(m,p))$ time in the worst case to determine whether a p-string pattern of length $m$ p-matches a p-string text of length $n$ over an alphabet with $p$ parameter symbols.

## 3 Generalizing the TurboBM algorithm.

Consequently, we turn to the TurboBM algorithm of Crochemore et al [CLC+92]. Like the BM algorithm, TurboBM tries $P$ at alignments with $T$ that are successively further to the right, beginning with $P$ aligned at the left end of $T$. For each alignment, TurboBM determines the longest suffix of $P$ that matches the aligned part of $T$, reports a p-match if all of $P$ p-matches $T$, and then calculates the size of the shift leading to the next alignment.

However, the TurboBM algorithm remembers two extra integers of extra information from one alignment to the next. The two integers represent which part of the shifted $P$ is aligned with the matching part of $T$ from the previous alignment. We call that substring of $P$ the *previous matching substring* or *PMS*. Figures 1 and 2 illustrate two cases, according to whether $P$ is shifted past the left end of the matching part of $T$. The $suf_i$ in the figures represents the longest matching suffix of $P$ in the earlier alignment. TurboBM has two shift rules, one of which is the BM shift rule, based on the table called $D$ by Rytter [Ryt80] and *dd′* by Knuth et al [KMP77]. If the BM shift rule is used, the PMS matches the aligned part of $T$ after the shift.

Like the BM algorithm, at each alignment of $P$ with $T$, TurboBM compares the pattern to the text from right to left to find the longest match. However, when the PMS is reached, if the BM shift rule was just used, the algorithm jumps over the PMS, because the PMS is known to match the aligned part of the text.

Once the whole pattern is matched (and a match reported) or a mismatch is found, the TurboBM algorithm calculates the size of the shift. In addition to the BM shift rule, the algorithm has a new shift rule, called the turbo shift. The turbo shift rule can be used when the previous shift was a BM shift and the length of the longest matching suffix in the current alignment is less than the length of the PMS, and it shifts the pattern by the difference. When both rules are applicable, the TurboBM algorithm picks the larger shift, and otherwise it uses the BM shift rule.

We wish to define an algorithm called PTurboBM that has the same overall structure as the TurboBM algorithm and runs within a log factor of linear in text length under the comparison model. We need to determine when jumps are feasible and what shift rules may be applied once the longest p-match has been found for $P$ at a current alignment.

We begin with generalizing the definition of table $D$ and defining a BM shift rule for p-strings. Next, we explore how the properties of p-strings can invalidate jumps over the entire portion of $P$ and $T$ known to p-match from the preceding alignment, and explain what definition of jumps will work. We prove that the turbo shift rule still is correct and define a new shift rule that is a hybrid of the turbo shift and the BM-shift. At that point, the main algorithm is given. Finally, we describe the book-keeping data structures needed to keep track of parameter pairings and compute jumps.

**3.1 Generalizing the BM shift.** Both the BM and TurboBM algorithms use table $D$ as one way of determining a lower bound on how far the pattern can be shifted without missing any matches. The definition of $D$ generalizes in a natural way to p-strings using p-equivalence of p-strings rather than comparisons of individual symbols. When $P[j+1,m]$ has been found to be the longest suffix of $P$ that p-matches the aligned part of $T$, the value of $PD[j]$ determines how far the pattern can be shifted. (For simpler exposition, we define $PD$ in terms of pattern shifts rather than the distance by which the text pointer jumps, as in the original definition of $D$. Text pointer jumps are larger by $m-j$.) Informally, for $0 \leq j < m$, $PD[j]$ is the smallest shift of $P$ compared to itself that causes $P[j+1,m]$ to p-match the aligned part of $P$ after the shift, and yet does not have $P[j,m]$ p-matching the aligned part of $P$ after the shift (if all of $P[j,m]$ is still aligned with the shifted $P$). More precisely, for $0 <= j <= m$, $PD[j] = \min\{\Delta | \Delta = m$ or $(0 < \Delta < j$ and $P[j+1-\Delta, m-\Delta] \equiv P[j+1,m]$ but $P[j-\Delta, m-\Delta] \not\equiv P[j,m])$ or $(j <= \Delta < m$ and $P[1,m-\Delta] \equiv P[\Delta+1,m])\}$. The two parts of this definition correspond to Figures 1 and 2.

As is the case for strings, $PD$ can be computed from $P$ alone; the method is a straightforward generalization of Rytter's algorithm [Ryt80]. (Rytter corrected a bug in the original construction given by [KMP77], which was for an improved definition of $D$ compared to the original definition of [BM77].) Parameter correspondences are checked using the *prev* encoding of [Bak93b], which replaces each parameter symbol in $P$ by a 0 if it is the leftmost occurrence of that symbol, or the distance from the previous occurrence to the left otherwise. An analogous *revprev* encoding stores distances to the right. Parameter correspondences can be checked within encoded substrings by comparing the encoded parameters, provided that numbers greater than or equal to the distance to the end of the substring are treated as zeroes. Building $PD$ takes $O(m \log \min(m,p))$ time as part of the preprocessing under the comparison model.

A shift based on $PD$ will be henceforth be called simply a *BM shift.*

**3.2 Jumps and Partial Jumps.** For strings, the turboBM algorithm can "jump" safely over substrings of $T$ and $P$ that are known to match from the previous stage because of a BM shift.

However, this "jump" strategy does not generalize directly to p-strings because the "jump" over previously matched symbols cannot always be made safely in p-strings. The problem is that for p-strings $\beta$, $\gamma$, $\sigma$, and $\tau$, if $\beta \equiv \sigma$ and $\gamma \equiv \tau$, it is not always true that $\beta\gamma \equiv \sigma\tau$. For example, if $x$, $y$, and $z$ are parameters, $x \equiv y$ and $x \equiv z$, but $xx \not\equiv yz$.

Suppose p-string $P = \alpha\beta\gamma$ is aligned with a substring $\rho\sigma\tau$ of p-string $T$ and it is known that $\beta \equiv \sigma$. After finding that $\gamma \equiv \tau$ and arriving at the right ends of $\beta$ and $\sigma$, the algorithm must determine whether $\beta\gamma \equiv \sigma\tau$, and if not, determine the longest p-matching suffixes of $\beta\gamma$ and $\sigma\tau$. Thus, the generalized algorithm must allow for both "full" jumps over all of $\beta$ and $\sigma$ when $\beta\gamma \equiv \sigma\tau$ and "partial" jumps when $\beta\gamma \not\equiv \sigma\tau$. How the algorithm determines how far to jump is described below in the subsection on bookkeeping.

**3.3 Turbo-shifts and TurboBM Shifts.** The turbo shift rule works for strings for the following reason. After a BM shift of $\Delta$ positions, it is known that the pattern has two (possibly overlapping) matching substrings $\alpha$ distance $\Delta$ apart, and that the leftmost one of these is aligned with a matching part of $T$. If a mismatch occurs between the text and a symbol within the rightmost occurrence of $\alpha$, there must be distinct text symbols distance $\Delta$ apart. Therefore, these cannot both match the pattern in any alignment such that identical symbols of the two occurrences of $\alpha$ are aligned with them.

The above turbo shift argument does not apply directly to p-strings, because of its dependence on equality of symbols. However, it is possible to generalize the above argument to p-strings in terms of p-equivalence of p-strings rather than comparisons of single symbols.

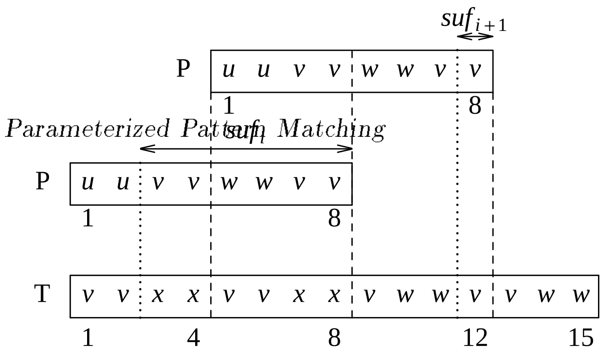


Figure 3: A situation that can be followed by a turbo shift of length 3.

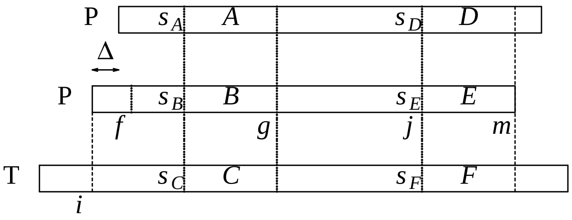


Figure 4: A shift smaller than a turbo shift.

Suppose a BM shift of size $\Delta$ occurred after $P[k, m]$ was found to be the longest suffix of $P$ that p-matched the aligned part of $T$. As we did for the TurboBM algorithm, define the *previous matching substring of P*, or *PMS*, to be $P[k-\Delta, m-\Delta]$ if $k > \Delta$, and $P[1, m-\Delta]$ otherwise, as illustrated in Figures 1 and 2.

A turbo shift is defined for p-strings as follows. Suppose the length of the PMS of $P$ is $a$, and the length of the longest suffix of $P$ that p-matches the aligned part of $T$ is $b$. If $a > b$, then the size of a turbo shift is $a - b$, and otherwise it is zero.

A situation in which a nonzero turbo shift is possible is illustrated in Figure 3. All symbols in Figure 3 are parameter symbols. For the two alignments of $P$, the lengths of the longest suffixes of $P$ p-matching the aligned part of $T$ are denoted by $suf_i$ and $suf_{i+1}$, respectively. In the top occurrence of $P$, the PMS of $P$ is $P[1, 4]$, while $suf_{i+1} = 1$. Therefore, a turbo shift of length 3 may be used. For any smaller shift, either the $xx = T[7, 8]$ or the $wv = T[11, 12]$ will not p-match the aligned part of $P$.

Lemma 3.1. *For p-strings, after a BM shift, no shift smaller than a turbo shift can result in an alignment of P with T in which P p-matches the aligned part of T.*

*Proof.* After a BM shift, let the PMS of $P$ be $P[f+1, g]$, let the current longest suffix of $P$ p-matching the aligned part of $T$ be $P[j + 1, m]$, and suppose $P$ is currently placed so that position 1 of $P$ is aligned with position $i + 1$ of $T$. Suppose $g - f > m - j$ so that the length of the turbo shift is $g - f - m + j > 0$.

Suppose that there is a $\Delta < g - f - m + j$ such that $P$ p-matches $T$ at alignment $i + \Delta$. Define $E = P[j + 1, m]$, so that $E$ denotes the longest suffix of $P$ p-matching the aligned part of $T$ after the BM shift, and $B = P[j+1-m-g, g]$, so that $B$ denotes the substring of the same length ending at $P[g]$. As in Figure 4, define $C$ and $F$ to be the substrings of $T$ aligned with $B$ and $E$, respectively, after the BM shift, and $A$ and $D$ to be the substrings of $P$ aligned with $C$ and $F$, respectively, after an additional shift of $\Delta$. These substrings are all the same length, $m-j < g-f$. The symbol to the left of any of these substrings will be denoted by $s$ subscripted with the substring name, as in $s_A$.

Since $E$ is the longest suffix of $P$ p-matching the aligned part of $T$ after the BM shift, we know that $s_E E \not\equiv s_F F$.

Since $P$ p-matches $T$ at alignment $i + \Delta$, $s_A A \equiv s_C C$ and $s_D D \equiv s_F F$. By definition of $PD$ and the assumption that $m - j < g - f$, $s_B B \equiv s_E E$. By definition of the BM shift, $s_B B \equiv s_C C$. By transitivity, $s_E E \equiv s_A A$. Since the definition of $PD$ implies that $P[f+1, g] \equiv P[m-g+f+1, m]$, and $\Delta < g-f-m+j$ so that the left end of $s_A A$ lies within $P[f + 1, g]$ and the left end of $s_D D$ lies within $P[m - g + f + 1, m]$, $s_A A \equiv s_D D$. From $s_A A \equiv s_D D$, $s_E E \equiv s_A A$, $s_D D \equiv s_F F$, and transitivity, we conclude that $s_E E \equiv s_F F$, a contradiction. □

Finally, to achieve the desired worst-case bound, we define a hybrid shift rule called the turboBM shift rule. The goal is to have a shift almost as large as the turbo shift but to preserve the BM-shift property that a suffix of $P$ p-matches the aligned substring of the shifted $P$.

Suppose the previous shift was of size $\delta$, the PMS is of size $a$, and the longest suffix of $P$ that p-matches the currently aligned part of $T$ is of length $b$. If $a - b = s\delta + r > 0$, where $0 <= r < \delta$, a turboBM shift is of length $s\delta$; otherwise, its length is zero.

Figure 5 illustrates a turboBM shift. From the lower part of the figure it may be seen that the BM shift implies that the structure of $P$ is "p-cyclic"; that is, $P$ has a suffix consisting of a sequence of subtrings (the $\alpha_j$'s) that are all p-equivalent (but not necessarily identical, as they would be for strings). This is a result of the PMS matching the aligned part of $T$.

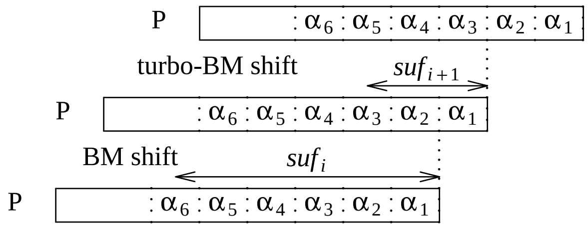


Figure 5: A turboBM shift, where the $\alpha_i$'s are p-strings of equal length.

Using the "p-cyclic" nature of $P$, we prove the following lemma.

LEMMA 3.2. *Suppose a turboBM shift follows a sequence of BM and turboBM shifts, beginning with a BM shift. Suppose the longest suffix of $P$ p-matching the aligned part of $T$ before the final turboBM shift is $P[a,m]$. If $\Delta > 0$ is the size of the resulting turboBM shift, then $a > \Delta$ and $P[a-\Delta, m-\Delta]$ p-matches $P[a,m]$.*

*Proof.* First, suppose the preceding sequence ended with a BM shift. Let $\delta$ be the size of the preceding shift. Choose $r$ and $w$ such that the longest suffix of $P$ p-matching the aligned part of $T$ is of length $(r-1)\delta+w$ before the final turboBM shift, where $0 \le w < \delta$. Choose $s$ and $u$ such that the PMS before the final turboBM shift is of size $(s-1)\delta+u$, where $0 \le u < \delta$ if $w \le |PMS| \pmod{\delta}$, and $\delta < u < w+\delta$ otherwise. In either case, the turboBM shift is of size $(s-r)\delta$ and $u \ge w$.

Since the PMS is of size $(s-1)\delta+u$, and that doesn't include the rightmost $\delta$ symbols, the total length of $P$ is at least $s\delta+u$. The rightmost $s\delta+w \le s\delta+u$ symbols can be written as $\gamma_{s+1}\alpha_s\alpha_{s-1}\ldots\alpha_1$, where each $\alpha_k$ has length $\delta$, and $\gamma_{s+1}$ has length $w$. For each $k$, let $\gamma_k$ be the suffix of $\alpha_k$ of length $w$.

By definition of PMS and the fact that $u \ge w$, the suffix $\gamma_{s-1}\alpha_{s-2}\alpha_{s-3}...\alpha_1$ of $P$ p-matched the aligned part of $T$ before the BM shift of length $\delta$. After the BM shift, $\gamma_s\alpha_{s-1}\alpha_{s-2}\ldots\alpha_2$ is aligned where $\gamma_{s-1}\alpha_{s-2}\alpha_{s-3}\ldots\alpha_1$ used to be and p-matches the aligned text by definition of the BM shift.

By transitivity, we find $\gamma_s\alpha_{s-1}\alpha_{s-2}\ldots\alpha_2 \equiv \gamma_{s-1}\alpha_{s-2}\alpha_{s-3}\ldots\alpha_1$. Then $\gamma_s\alpha_{s-1}\alpha_{s-2}\ldots\alpha_{s-r+1} \equiv \gamma_{s-1}\alpha_{s-2}\ldots\alpha_{s-r} \equiv \gamma_{s-2}\alpha_{s-3}\ldots\alpha_{s-r-1} \equiv \cdots \equiv \gamma_r\alpha_{r-1}\ldots\alpha_1$. By transitivity, $\gamma_s\alpha_{s-1}\ldots\alpha_{s-r+1} \equiv \gamma_r\alpha_{r-1}\ldots\alpha_1$, proving the lemma for the case in which the preceding shift was a BM shift.

After the BM shift of length $\delta$, $\gamma_r\alpha_{r-1}...\alpha_1$ is the longest suffix of $P$ that p-matches the aligned part of $T$. By transitivity $\gamma_s\alpha_{s-1}\ldots\alpha_{s-r+1}$ p-matches the same part of $T$ and is aligned with it after the turboBM shift. Since the above argument only used this property of the preceding BM shifts, the same argument can be used after a turboBM shift. This implies inductively that the lemma holds for the case in which the preceding shift was a turboBM shift. □

Since this lemma states that the turboBM shift satisfies the properties of the BM shift needed for Lemma 3.1, the following corollary follows.

COROLLARY 3.1. *After a turboBM shift, no shift smaller than the turbo shift can result in a p-match of $P$ with the aligned part of $T$.*

Similarly, full and partial jumps may be used after a turboBM shift in the same manner that they may be used after a BM shift.

Since a turboBM shift is no larger than a turbo shift, Lemma 2.1 and Corollary 2.1 imply that a turboBM shift can be safely used when a turbo shift can be.

COROLLARY 3.2. *After a BM shift or a turboBM shift, no shift smaller than the turboBM shift can result in a p-match of $P$ with the aligned part of $T$.*

**3.4 Main Procedure of the PturboBM Algorithm.** At this point, we can define the main procedure of PturboBM, with the book-keeping of parameter pairings left for later.

Like TurboBM, PturboBM tries $P$ at alignments with $T$ that are successively further to the right, beginning with $P$ aligned at the left end of $T$. For each alignment, PturboBM determines the longest suffix of $P$ that p-matches the aligned part of $T$, reports a p-match if all of $P$ p-matches $T$, and then calculates the size of the shift leading to the next alignment.

To determine the longest suffix of $P$ that p-matches the aligned part of $T$, PturboBM works from right to left comparing symbols of $P$ with those of $T$ looking for a mis-pmatch. If it reaches the PMS and the last shift was a BM shift or a turboBM shift, it checks to see how much of the PMS it can jump over, that is, how much of the PMS and the currently aligned part of $T$ has parameter pairings consistent with the portion of $T$ and $P$ already checked to the right. If it can't jump over all of it, it has determined the longest suffix of $P$ p-matching $T$ in that alignment. Otherwise, it jumps over all of it and continues comparing $T$ and $P$ working leftward as before until it finds a mismatch or until it finds all of $P$ p-matches the aligned part of $T$, in which case it reports a p-match.

Finally, the algorithm must determine whether to use a BM shift, a turbo shift or a turboBM shift. If the previous shift was a turbo shift or the new longest p-matching suffix of $P$ is at least as long as the PMS, PturboBM chooses a BM shift. Otherwise, if a full or partial jump just occurred, PturboBM uses the larger of a BM shift and a turboBM shift, while if no jump occurred, it uses the larger of a BM shift and a turbo shift.

Assuming that partial jumps are calculated correctly, correctness of the algorithm follows from the definition of $PD$, Lemma 3.1, Corollary 3.1, and Corollary 3.2, since these show that the algorithm never skips checking an alignment in which there could be a p-match. The calculation of partial jumps is part of book-keeping described in the next subsection.

**3.5 Book-keeping.** The above description still leaves open how to do the book-keeping to keep track of the parameter pairings. We want the time used for book-keeping in an alignment to be proportional (with an additional log factor) only to the number of input symbols scanned in that alignment, not including symbols jumped over.

Several difficulties must be overcome. First, shifting the pattern may change the pairings of parameter symbols for the part of $T$ examined previously, because the new parameter of $P$ aligned with a symbol of $T$ may not be the same as the old one even though the PMS of $P$ is known to be p-equivalent to the aligned substring of $T$ as a result of a BM or turboBM shift. Moreover, jumps can be partial jumps, so that the algorithm must identify which parts of the previous pairing information are relevant as it computes the partial jump.

For $P$, we use precomputed encodings $prev(P)$ and $revprev(P)$ (defined as described for computing $PD$) to identify pairs of positions that must be checked to determine whether each parameter symbol $y$ of $P$ is paired with the same parameter symbol in $T$ as the next usage of $y$ to the right or left, as appropriate.

For $T$, we store for each parameter symbol $x$ that is scanned the rightmost position $t$ at which it was scanned. The value of $t$ is used to determine whether the pairing assigned to $x$ is relevant and if so, to find it by checking the symbol of $P$ currently aligned with position $t$ of $T$. The pairs $(x, t)$ will be stored in a data structure we call a pairstore, which can be implemented as a balanced tree. The tree can be kept to size $O(\min(m,p))$ by deleting out-of-date values after each shift by means of another balanced tree storing values of $t$ so that after a shift, successive minimum out-of-date $t$-values can be found in $O(\log(\min(m,p)))$ time and deleted. This method has $O(\log(\min(m,p))$ overhead per symbol of $T$ scanned elsewhere in the algorithm.

In addition to the main pairstore, which we call "old", we use a temporary "new" one in each alignment. The "old" one retains information from one alignment to the next, while the "new" one holds new pairings while processing is still to the right of the PMS (if this is after a BM or turboBM shift). The new pairstore and $revprev(P)$ are used to check the consistency of the pairings between symbols of $P$ and $T$ as they are encountered. In addition, a list is kept of the leftmost positions of parameters of $P$ as they are encountered. When the previously matched part of $T$ is reached, the new pairings are compared with the old using the pairstores, the list of leftmost positions of parameter symbols, and $prev(P)$ to find the rightmost position of conflict (if any) within the previously matched part of $T$, and the new pairings are inserted into the old pairstore and deleted from the new one and the list. Processing continues to the left using just the old pairstore to check for consistency of parameter pairings. The overhead for each symbol of $P$ and $T$ compared is $O(\log(\min(m,p)))$.

At this point the entire PturboBM algorithm has been described. It may be verified that the above book-keeping operations correctly compute partial jumps, and with the earlier proofs, we have the following theorem.

THEOREM 3.1. *Given a p-string pattern $P$ of length $m$ and a p-string text $T$ of length $n$, the PturboBM algorithm finds all alignments of $P$ at which $P$ p-matches $T$.*

**3.6 Bounding the Running Time.** We say that the $i$th "stage" of the algorithm processes the $i$th alignment that is tried. For stage $i$, define $suf_i$ to be the length of the longest suffix of $P$ p-matching the aligned part of $T$. Define the *cost* of stage $i$, $cost_i$, to be $suf_i$ minus the number of pattern symbols jumped over in that stage (if any). Define $shift_i$ to be the length of the shift at the end of stage $i$.

We first show that the total cost of the computation is linear in $n$. The overall proof strategy is to bound the total cost in terms of the sums of shifts, since the sum of all the shifts is at most $n$.

The proof has two key ideas. First, the entire computation is divided into maximal J-sequences, where a J-sequence is a sequence of stages in which all but the first contain a jump, and bounds are computed for the costs of single J-sequences. If there are only full jumps, the situation can be visualized from Figure 6, which illustrates that the total cost of a sequence ending in stage $k$ is at most $suf_k$, since the successive alignments do not overlap in which symbols of $T$ are compared with $P$. More generally, there may be partial jumps as well and the analysis is more complicated because overlapping can occur when a full jump follows a partial jump. The second key idea is to bound certain costs by

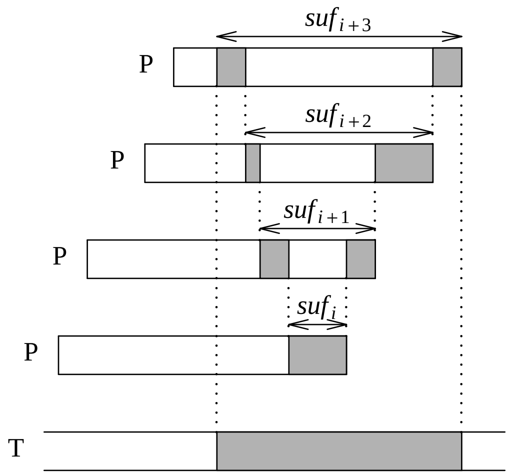


Figure 6: A sequence of alignments in which all but the first have full jumps. Portions of $P$ and $T$ contributing to the cost are shaded.

looking at which kind of shift was after the J-sequence. Finally, summing the bounds gives the desired linear bound on cost.

PROPOSITION 3.1. *If stage $i$ through stage $k$ all have jumps (either full or partial),*

$$suf_{i-1} \leq suf_k + 2shift_{i-1} + shift_k + 3\sum_{j=i}^{k-1} shift_j$$

*Proof.* Let $i \leq j \leq k$. Since a jump can only occur after a BM shift or a turboBM shift, stage $j$ has either a BM shift or a turboBM shift. Let $PMS_j$ denote the length of the PMS in stage $j$. A turboBM shift is used in stage $j$ when $PMS_j - suf_j \geq shift_{j-1}$ and the turboBM shift is bigger than a BM shift. A turboBM shift is of size at least $suf_{j-1} - suf_j - 2shift_{j-1}$, where one $shift_{j-1}$ arises because $PMS_j \geq suf_{j-1} - shift_{j-1}$ and the other arises because the turboBM shift is a multiple of $shift_{j-1}$. So we see that either $PMS_j - suf_j < shift_{j-1}$, or a shift at least as large as the turboBM shift is used and $shift_j \geq suf_{j-1} - suf_j - 2shift_{j-1}$. In either case, $suf_{j-1} \leq 2shift_{j-1} + shift_j + suf_j$. The result holds inductively. □

LEMMA 3.3. *Suppose stages $i$ through $k$ are a J-sequence. Then its total cost is at most*

$$suf_k + 3shift_i + shift_k + 4\sum_{j=i+1}^{k-1} shift_j$$

*Proof.* If stage $j$ has a full jump, its cost is $suf_j - suf_{j-1}$, while if stage $j$ has a partial jump, its cost is at most the length of the shift at the end of the previous stage, namely $shift_{j-1}$.

Suppose a sequence of stages has a full jump in stage $i$ followed by partial jumps in stages $i+1$ through $k$. The cost of the sequence is at most

$$suf_i - suf_{i-1} + \textstyle\sum_{j=i}^{k-1} shift_j$$

Applying Proposition 3.1 and summing over a series of such sequences beginning with a full jump in stage $r$ and ending with a full or partial jump in stage $k$ yields a cost for the series of at most

$$suf_k - suf_{r-1} + shift_k + 4\sum_{j=r}^{k-1} shift_j$$

Adding in the cost of the initial stage, which is at most $suf_i$, and the cost of an initial sequence of partial jumps through stage $r-1$ and applying Proposition 3.1 yields the lemma. □

A maximal J-sequence is one that is not followed by a stage with a jump. Let $t$ be the number of the last stage.

PROPOSITION 3.2. *Suppose a maximal J-sequence ends in stage $k < t$ with a BM or turboBM shift. Then $suf_k \leq 2shift_k + shift_{k+1}$.*

*Proof.* Similar to the proof of Proposition 3.1, for a shift at least as large as a turbo shift. □

LEMMA 3.4. *Given a p-string pattern $P$ of length $m$ and a p-string text $T$ of length $n$, the cost of processing $T$ (not including preprocessing) is at most $5n$.*

*Proof.* Divide the computation into maximal J-sequences.

A maximal J-sequence that ends in a turbo shift has just one stage and no jump, since a turbo shift cannot be used in a stage with a jump. Moreover, a turbo shift must follow a BM shift or a turboBM shift. Therefore, the lack of jump means its cost is at most $shift_{k-1}$. Thus, the total cost of such J-sequences is at most

$$\sum_{j=1}^{t-1} shift_j .$$

Combining Lemma 3.3 and Proposition 3.2, the cost of a J-sequence from stage $i$ through stage $k$ ending in a BM or turboBM shift is at most

$$3shift_i + shift_{k+1} + 4\sum_{j=i+1}^{k} shift_j ,$$

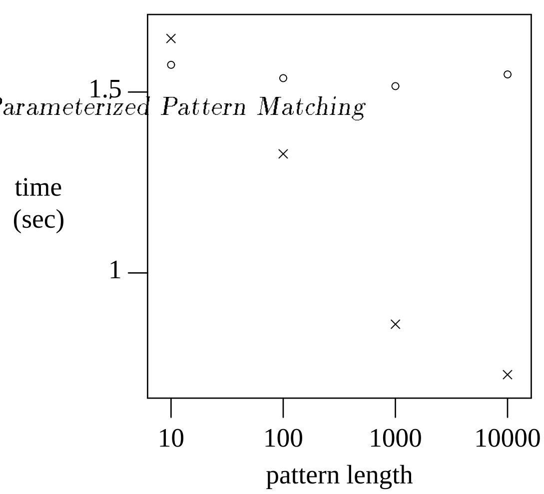


Figure 7: A comparison of PturboBM and pKMP with parameter and non-parameter alphabet sizes of 100. Average running times are shown for PturboBM ($\times$) and pKMP ($\circ$) for $n = 1,000,000$ and various values of $m$.

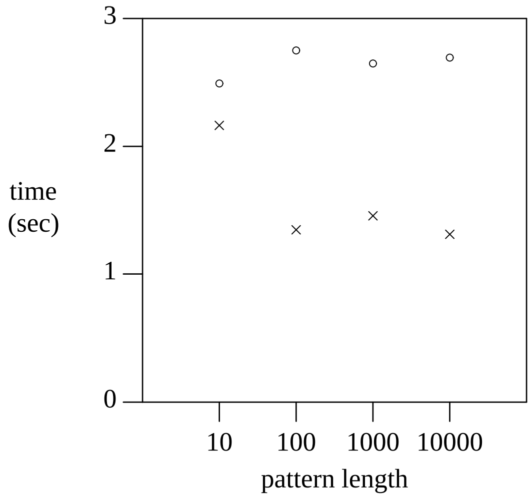


Figure 8: A comparison of PturboBM and pKMP with $p = m$, no non-parameter symbols, and $n = 1,000,000$. Average running times are shown for PturboBM ($\times$) and pKMP ($\circ$) for $n = 1,000,000$ and various values of $m$.

Therefore, all such J-sequences through a stage $k < t$ have a total cost of at most

$$shift_{k+1} + 4\sum_{j=1}^{k} shift_j$$

Adding in a possible J-sequence ending in stage $t$ and applying Lemma 3.3 yields a total cost of at most

$$suf_t + shift_t + 5\sum_{j=1}^{t-1} shift_j$$

Since all the shifts except the last sum to at most $n-m$, the result follows. □

Since the overhead for each unit of cost is $O(\log \min(m,p))$, we obtain the following bound on worst-case running time.

THEOREM 3.2. *Given a p-string pattern $P$ and a p-string text $T$, the PturboBM algorithm uses $O(n \log \min(m,p))$ time and $O(n)$ space under the comparison model to determine the positions at which $P$ p-matches $T$, with preprocessing time $O(m \log \min(m,p))$, where $n$ is the length of $T$, $m$ is the length of $P$, and $p$ is the number of distinct parameter symbols.*

## 4 Experiments.

For experimental comparison, the PturboBM algorithm and a generalization pKMP of the KMP algorithm were implemented in a manner appropriate for the software maintenance application. In particular, the implementations use integer tables of size $p$ rather than balanced trees to keep track of pairings of parameters where necessary. This avoids the $\log p$ factors in the time bounds that come from using balanced trees but results in $O(p)$ space usage. Tables of size $p$ are acceptable for this application since the parameter and non-parameter symbols are integers produced by lexical analysis, $p < n$, and $p$ is known in advance of pattern matching.

The pKMP algorithm implemented is a straightforward generalization of KMP [KMP77]. First, the *prev* encoding of [Bak93b] is applied to $P$ and $T$. Thereafter, where two symbols would be compared for equality in the context of two substrings in KMP, pKMP uses the *prev*-encoded symbols to check for consistency of the parameter pairing (if any) in the context of the two substrings. The proof of correctness is a straightforward generalization of the proof for strings. In timings, computing $prev(P)$ is included in the preprocessing time, but the computation of $prev(T)$ is included in the p-matching time.

Some experiments were performed on random input with a uniform distribution over $[0, p+s-1]$, where the non-parameter alphabet $\Sigma = [0, s-1]$ and the parameter alphabet $\Pi = [s, s+p-1]$, and with $n = 1,000,000$ and $m = 10$, 100, 1000, and 10000. The experiments included all parameters, all non-parameters, and $p = s$, with various values of $p$ and $s$ under these constraints, either constants or equal to $m$. Overall, average times were close for the two algorithms for $m = 10$, with one or the other faster, with PturboBM almost always faster for $m = 100$ and always faster for larger values of $m$. Space does not permit giving the results of all the experiments, but Figure 7 shows average times for $p = s = 100$ and Figure 8 shows average times for $s = 0$ and $p = m$. For each pair of points, pKMP and PturboBM were run on the same 50 random inputs. Preprocessing times were negligible for both algorithms.

These results are not surprising. The advantage of PturboBM over pKMP is that it need not read every text symbol, and this advantage increases with increasing pattern size. For small values of $m$, the extra overhead of PturboBM may outweigh this advantage. The actual cross-over point will depend on tuning and on the nature of the input.

The shift behavior is very pattern-dependent for PturboBM. The dominant shift type arising in the experiments is the BM shift. For certain patterns, turbo shifts occur as well with some frequency. TurboBM shifts are extremely rare.

Carefully designed experiments on input obtained from software have not yet been performed. In four trials on a text of about 2 million symbols produced from a million-line software subsystem by the lexical analyzer of [Bak93b], with patterns of length 200 selected from the text, PturboBM and pKMP had running times of 1-4 seconds, with relative times ranging from PturboBM 50% slower to 63% faster compared to pKMP, with pKMP ahead three out of four times. However, their running times were completely dominated by the three minutes of time for lexical analysis. That lexical analyzer was designed to minimize $n$ by producing semi-line-based output (one non-parameter symbol per line) in order to minimize the size of the subsequent p-suffix trees. A lexical analyzer producing tokens would be much faster and the resulting total running time for lexical analysis and pattern matching would probably be much smaller. Furthermore, for PturboBM, not every input symbol needs to be examined, and a strategy of hashing tokens only as needed might lead to a smaller total time. This strategy would not be feasible for pKMP, which examines every input symbol. Thus, a proper comparison of PturboBM and pKMP on input from code would require some careful thought about the design of lexical analyzers.

## References

[AFM94] Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Info. Proc. Letters*, 49:111–115, 1994.

[Bak] Brenda S. Baker. Parameterized pattern matching: algorithms and applications. *J. Comput. Syst. Sci.* to appear.

[Bak92] Brenda S. Baker. A program for identifying duplicated code. In *Computing Science and Statistics 24: Proceedings of the 24th Symposium on the Interface*, pages 49–57, March 1992.

[Bak93a] Brenda S. Baker. Parameterized duplication in strings: algorithms and an application to software maintenance. Submitted for publication, 1993.

[Bak93b] Brenda S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In *Proceedings of the 25th Annual Symposium on Theory of Computing*, pages 71–80, San Diego, California, 1993.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[BYGR90] Ricardo A. Baeza-Yates, Gaston H. Gonnet, and Mireille Regnier. Analysis of Boyer-Moore-type string searching algorithms. In *Proc. of First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 328–343, 1990.

[BYR92] Ricardo A. Baeza-Yates and Mireille Regnier. Average running time of the Boyer-Moore-Horspool algorithm. *Theoretical Computer Sci.*, 92:19–31, 1992.

[CLC+92] Maxime Crochemore, Thierry Lecroq, Artur Czumaj, Leszek Gasieniec, S. Jarominek, and W. Plandowski. Speeding up two string-matching algorithms. In *9th Annual Symposium on Theoretical Aspects of Computer Science (STACS 92), Lecture Notes in Computer Science 577*, pages 589–600, 1992.

[Col91] Richard Cole. Tight bounds of the complexity of the Boyer-Moore string matching algorithm. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 224–234, 1991.

[Hor80] R. Nigel Horspool. Practical fast searching in strings. *Soft. Pract. and Exp.*, 10:501–506, 1980.

[HS91] Andrew Hume and Daniel Sunday. Fast string searching. *Soft. Pract. and Exp.*, 21(11):1221–1248, 1991.

[IS94] Ramana M. Idury and Alejandro A. Schaffer. Multiple matching of parameterized patterns. In *Proc. of 5th Symposium on Combinatorial Pattern Matching*, pages 226–239, 1994.

[KMP77] D.E. Knuth, J. H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[Ryt80] Wojciech Rytter. A correct preprocessing algorithm for Boyer-Moore string-searching. *SIAM J. Comput.*, 9(3):509–512, August 1980.

[Sch88] R. Schaback. On the expected sublinearity of the Boyer-Moore algorithm. *SIAM J. on Comput.*, 17(4):648–659, August 1988.

[Sun90] Daniel M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–139, August 1990.