Quantum Leap Pattern Matching
A New High Performance Quick Search-Style Algorithm

Bruce W. Watson[1,2], Derrick G. Kourie[1,2], and Loek Cleophas[1,3]

[1] FASTAR Research Group, Department of Information Science, Stellenbosch University,
Private Bag X1, 7602 Matieland, Republic of South Africa
[2] Centre for Artificial Intelligence Research,
CSIR Meraka Institute, Republic of South Africa
[3] Department of Computer Science, Umeå University,
SE-901 87 Umeå, Sweden
{bruce,derrick,loek}@fastar.org

Abstract. Quantum leap matching is introduced as a generic pattern matching strategy for the single keyword exact pattern matching problem, that can be used on top of existing Boyer-Moore-style string matching algorithms. The cost of the technique is minimal: an additional shift table (of one dimension, for shifts in the opposite direction to the parent algorithm's shifts), and the replacement of a simple table lookup assignment statement in the original algorithm with a similar conditional assignment. Together with each of the conventional shift table lookups, the additional shift table is typically also indexed on the text character that is at a distance of z away from the current sliding window. Under conditions that are identified, the returned values from the two shift tables allow a "quantum leap" of distance more than the length of the keyword for the next matching attempt. If the conditions are not met, then there is a fall back is to the traditional shift.
Quick Search (by Sunday) is used as a case study to illustrate the technique. The performance of the derived "Quantum Leap Quick Search" algorithm is compared against Quick Search. When searching for shorter patterns over natural language and genomic texts, the technique improves on Quick Search's time for most values of z. Improvements are also sometimes seen for various values of z on larger patterns. Most interestingly, under best case conditions it performs, on average, at about *three times faster* than Quick Search.

Keywords: high-speed pattern matching single keyword matching, Boyer-Moore algorithms, Sunday's algorithm, faster pattern matching

1 Introduction

We consider the well-known single keyword string matching problem. We adopt the convention that string s is treated as an array whose length is $|s|$. Its first element is at index 0 and the element at index i is denoted by $s[i]$. A substring of length n starting at index i is denoted[1] by $s[i, i+n)$.

Given an alphabet Σ, a text string $t \in \Sigma^*$ and a single keyword or pattern string $p \in \Sigma^+$ of length $|p| = m$, the string matching problem is to find all indices of t where a *match* of p occurs. A match of p at index i *occurs* if $p[0, m) = t[i, i+m)$.

Following Cantone and Faro [1], we call $t[i, i+m)$ the *current window* of the text when $p[0]$ is aligned with $t[i]$. To solve the string matching problem, 'Boyer-Moore style' algorithms *slide* (or 'shift') the current window in t in a given direction

[1] A set of successive integers $\{i, i+1, \ldots j\}$ is commonly represented in interval notation format as one of the following: $[i, j]$, $[i, j+1)$, $(i-1, j+1)$ or $(i-1, j]$. Our motivation for this substring notation is to simplify +1 and −1 subscript expressions using square and round parentheses.

— normally in a forward direction (from left to right), but it could also be in a backward direction (from right to left). After shifting the current window to index i a *match attempt* at i is made. If the match attempt is successful then i is recorded as a match location. Regardless of whether the attempt is successful, an offset value from i (to the right of i in classical algorithms proceeding from left to right) is found to indicate where the next match attempt should take place. Naturally, such an offset must not miss any intervening matches, and it is called a *safe* offset or shift.

The offset is usually given by a *shift table* that is precomputed from the structure of p — see [6] for examples of such a shift functions. The simplest such table is accessed by indexing by a character in the substring $t[i, m+2)$, although certain algorithms instead use more information, such as two characters in this range and use a two-dimensional shift table. Due to its simplicity and efficiency, we specifically focus on Sunday's Quick Search shift table [9]. Shift tables are covered in books such as [2,4,8]. See [1] for a recent survey of related algorithms and shift tables, while [3] gives a *calculus* for arriving at all of the known shift tables as well as designing new ones.

When a current window at i is slid in a forward direction, a *forward shift table* provides an integer $\mathit{shf} \in [1, m+2)$ indicating that the next match attempt should be at a window at $i + \mathit{shf}$. The heuristics used to set up the shift table guarantees that no match is missed — there is no match in the index range $[i+1, i+\mathit{shf})$. Note this must hold regardless of previous match attempts in the range $[0, i+1)$.

Dually, if we were sliding the current window at i in a backward (right to left) direction, a *backward shift table* provides an integer $\mathit{shb} \in [1, m+2)$ indicating that the next match attempt may be made at a window at $i - \mathit{shb}$, since no match at an index in the range $(i - \mathit{shb}, i)$ is possible. Again, the validity of this assertion is independent of whether match attempts have previously been made at indices in the range $[i, |t|+1)$.

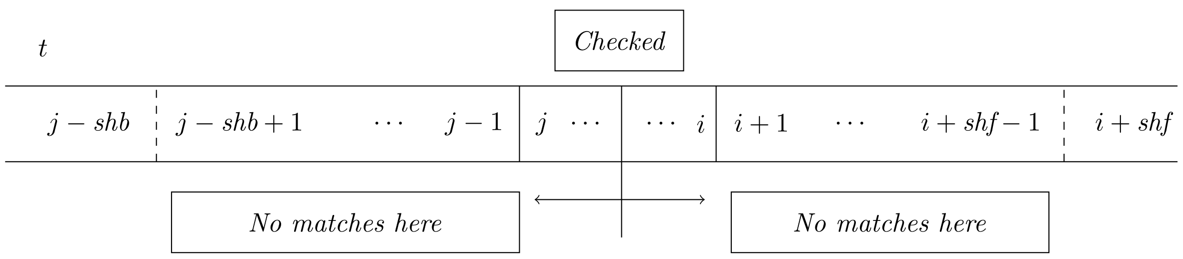
Some algorithms partition the search space over t at one or more indices of t — see [10]. Windows are placed to the right and left of such indices and, after match attempts, they are slid in a forward and backward direction respectively. The scenario is loosely depicted in Figure 1a. The figure assumes that the indices in the interval (j, i) have already been checked. It further assumes that the offset shb results from a match attempt at j followed by a backward table lookup, while the offset shf results from a match attempt at i followed by a forward table lookup.

Suppose that Figure 1b, a variant of Figure 1a, is the result of executing some abstract string matching algorithm. (Ignore for the moment the entries at the top of the figure that refer to z. They shall be addressed in Section 2.) The figure was inspired by our experience that algorithms based on Figure 1a incur penalties (including cache miss penalties) because of the bookkeeping required in respect of the partitions over t. Figure 1b assumes that a forward scan has already checked all indices in the interval $[0, i)$ for matches, thus avoiding the cache miss problem encountered by algorithms based on the latter figure. Additionally, this figure assumes that all information in Figure 1a is available and that $i < j$.

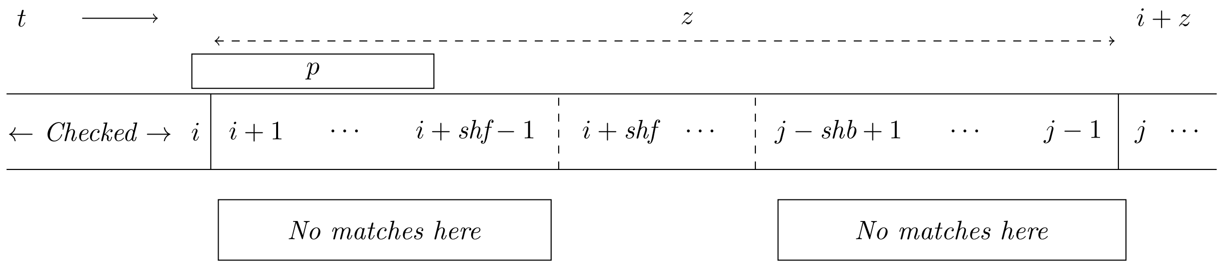
Clearly, if the predicate

$$i + \mathit{shf} \geq j - \mathit{shb} + 1 \tag{1}$$

was true, then the abstract algorithm could safely resume further processing at j, thus making a right shift that is larger than shf. If predicate (1) is false, then $i + \mathit{shf}$ serves as a fall back position from which to resume further processing.



(a) Dead Zone



(b) The Quantum Leap Matching Concept

Figure 1: Dead Zone vs Quantum Leap Matching Concept

The intuition reflected in Figure 1b served as the starting point for exploring whether and how an algorithm can be developed to exploit the possibility of *leapfrogging* over *shf* to produce larger shifts. Because this is a possibility and not a given, we call our approach the *quantum leap* strategy.

Reliance on dual shift tables may be used to update existing string matching algorithms. In Section 2 we outline how this can be done for the well-known Quick Search (QS) algorithm proposed by Sunday [9]. We call this the Quantum Leap QS algorithm (QLQS). Then Section 3 describes the empirical results yielded by this algorithm. Reflections on these results are presented in a final concluding section.

2 The QLQS Algorithm

An abstract algorithm that relies on predicate (1) is entirely generic in that it does not depend on how *shf* and *shb* are obtained. All that matters is that their values should ensure that matches are not possible in the two regions in Figure 1b marked "*No matches here*". As mentioned earlier, to derive the concrete QLQS algorithm, we decided to rely on QS's forward and backward shift tables as the dual shift tables needed for *shf* and *shb*. We compare our results against QS.

2.1 QLQS derived from QS

After a match attempt at i, QS uses the character at $t[i+m]$ as an index into its (forward) shift table to find *shf*—the offset from i for the next match attempt. It is well-known that for QS, *shf* will lie in the range $[1, m+2)$.

To proceed, an abstract algorithm based on QS would determine the value of *shb* by indexing into QS's backward shift table at the character $t[j-1]$, where j has some suitably chosen value. Unfortunately, it is not clear how to choose such a value for j. It would be pleasant if a "magical" choice of j guaranteed two conditions at *every* match attempt: firstly, that the length of the interval $[i+1, j)$ is at a maximum; and secondly, that for the chosen j the predicate (1) continues to hold. To guarantee just the first condition would mean to have foreknowledge of the next match index and to choose j exactly at that index—something which is clearly infeasible. To guarantee the second condition without incurring the expense of additional probes between i and j is only possible for trivial choices of j. (Subsection 2.2 will examine possible ranges of j.) To avoid such computational expense, a compromise action is to rely on some fixed offset ahead of i whose compliance with predicate (1) is stochastically determined.

Let us call this offset z, and assume that $j = i + z$. This is depicted at the top of Figure 1b. In principle, at every new match attempt in a search, a different value for z could be selected according to some criterion. However, to keep things simple, our research is based on a preselected z value over the entire search. Empirical results discussed in Section 3 examine the consequences of selecting various values for z.

With elementary algebraic manipulation, predicate (1) can be rewritten in terms of z as

$$\begin{array}{rc} & \mathit{shf} > z - \mathit{shb} \qquad (2) \\ \text{or equivalently} & \mathit{shf} + \mathit{shb} > z \end{array}$$

After every match attempt, it is now necessary to check whether predicate (2) holds for the preselected value of z. To carry out this check the value of *shb* has to be obtained by looking up QS's backward shift table for the character $t[i + z - 1]$ and then evaluating the revised predicate (2). If predicate (2) turned out to be true, then the next match attempt could take place at $i + z$; otherwise the next match attempt must necessarily be at $i + \mathit{shf}$. Note, however, that to save on algorithmic computations during run time, the minus operation can be avoided in predicate (2) by precomputing and storing $\mathit{shb}' = z - \mathit{shb}$ instead of *shb* as the backward shift table.

A new algorithm can now be very simply derived from QS by carrying out the following four steps.

1. Select a suitable value for z;
2. Precompute the table for shb';
3. Replace the QS assignment statement that unconditionally increments i by *shf* for the next match attempt index with a conditional statement incrementing i by z if $(\mathit{shf} > \mathit{shb}')$ or by *shf* otherwise.
4. Pad the tail end of t as necessary to ensure that the reference to $t[i + z - 1]$ does not cause an array bound error in the last iteration of the main algorithm loop.

Figure 2 gives the resulting C code for this revised algorithm. It contains a counter, `count`, for the number of matches. The array `shf` stores the precomputed forward table and the array `shb` stores the precomputed values for the shb' values. The instructions differ from the conventional QS algorithm only in that the conditional assignment statement in lines 9 to 11 has replaced a conventional assignment statement, `i += shf[T[i+m]]`, that would be used in QS. The variable `z` is global to the code.

We conjecture that these simple steps, appropriately adapted, could be used to modify practically any of the common string matching algorithms.

```
static int search(const unsigned char *P, int m, const unsigned char *T, int n) {
    int k, i, count;
    i = 0;
    count = 0;
    while(i<=n-m) {
        k=0;
        while(k<m && P[i]==T[i+k]) i++;
        if (k==m) count++;
        i +=    (shf[T[i+m]] > shb[T[i+z-1]]
                  ? z
                  : shf[T[i+m]]);
    }
    return count;
}
```

Figure 2: C Code for QLQS

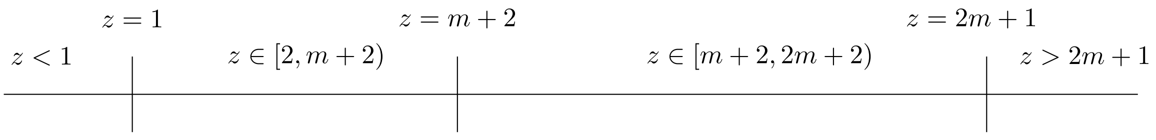


Figure 3: z Ranges Considered

2.2 Range of z Values

In QLQS outlined above, z is a constant. Variants of this algorithm might adjust the value of z dynamically to reflect text characteristics that manifest as it is being searched. It is therefore important to consider the range of values that z may legitimately and meaningfully assume, whether chosen as a constant or dynamically changed during a variant of the algorithm. In the discussion to follow, without loss of generality and for the sake of simplicity, reference to backward tables should be construed to mean those whose values are represented by *shb* and not those whose values are represented by *shb*$'$.

For given values of *shf* and *shb*, it is easy to see that $z = \mathit{shf} + \mathit{shb} - 1$ is the largest value of z that complies with predicate (2). Since the range of possible values for both *shf* and *shb* in QS shift tables is $[1, m + 2)$, the minimum and maximum of these largest possible z values are respectively attained when $\mathit{shf} = \mathit{shb} = 1$ and when $\mathit{shf} = \mathit{shb} = m + 1$. In the former instances, $\mathit{shf} + \mathit{shb} - 1$ evaluates to 1 and in the latter case $\mathit{shf} + \mathit{shb} - 1$ evaluates to $2m + 1$. The points, $z = 1$ and $z = 2m + 1$, are indicated in Figure 3, as well as various other points and ranges that will visually support the discussion that follows.

Clearly, if $z > 2m+1$ then predicate (2) cannot be satisfied for any values assumed by *shf* and *shb*, and consequently QLQS will always slide the current window ahead to $i + \mathit{shf}$. It will therefore execute in exactly the same way as QS, but with an additional overhead.

On the other hand, if z is selected in the range $[1, 2m + 2)$ then for some values of *shf* and *shb* within their permissible ranges predicate (2) may be satisfied, and for others, not. Whenever the predicate is satisfied, QLQS will slide its window to $i + z$, and otherwise to $i + \mathit{shf}$.

However, satisfying this predicate does not necessarily mean that $z > \mathit{shf}$ and so in these instances QLQS will not slide the window as far to the right as QS. A necessary and sufficient condition for QLQS to slide further than QS is a conjunction of the predicates (2) and $z > shf$, namely the predicate

$$\begin{aligned} & (z < shf + shb) \wedge (z > shf) \\ \text{or equivalently}\quad & z \in [shf + 1, shf + shb) \end{aligned} \tag{3}$$

Since the maximal QS value for shf is $m + 1$, any selection of z in the range $[m + 2, 2m+2)$ guarantees compliance with the lower bound of the interval in predicate (3). If, in addition, the current values for shf and shb result in compliance with the upper bound—equivalently, if predicate (2) is satisfied—then QLQS will slide the window further than QS.

If $z \in [1, m + 2)$ then the current values for shf and shb may or may not render z compliant with predicate (3). If the predicate is indeed satisfied, then QLQS will slide further than QS from the current window. If predicate (3) is false but predicate (2) is satisfied, then QLQS will slide less than (or the same as, if $z = shf$) QS from the current window.

If $z = 1$, then predicate (2) is always satisfied, but predicate (3) is never satisfied. As a result, the window will always slide to $i + z = i + 1$. Thus, QLQS degenerates to the most naïve string matching algorithm—one that merely slides the window by one position in each iteration.

Finally, if $z < 1$ then inspection will confirm that predicate (2) is satisfied for all possible values of shf and shb, while predicate (3) is never satisfied. QLQS executed with $z < 1$ will therefore always slide to $i + z < i + 1$. If $z = 0$ then this means staying in the same window as before. The algorithm effectively ends up in an infinite loop, carrying out a match attempt in the same place. If $z < 0$ and the current value of $i + z \geq 0$ then the slides to the left and a region already checked before will be rechecked—the region marked *Checked* in Figure 1b. This is obviously redundant. If $z < 0$ and the $i + z < 0$, then the next match attempt will involve an out-of-range index of t. This will be the case if $z < 0$ is used in the first iteration of the algorithm. Any algorithm built around a dynamically changing value of z should account for these boundary problems.

Table 1 summarises the foregoing discussion. Columns represent differing possible choices of z as given in the column heading. The first three rows indicate whether the predicate in the row heading (on the left) is always true, always false, or possibly either depending on the specific values of shf and shb, indicated by **true**, **false** or **depends** respectively. (Note that row 2 corresponds to predicate (2) and row 3 corresponds to predicate (3).) Row 4 indicates whether the offset from i will definitely be z, or definitely shf or either one of these values, depending on whether or not predicate (2) is satisfied. The final row indicates the worst outcome for the z in each respective column.

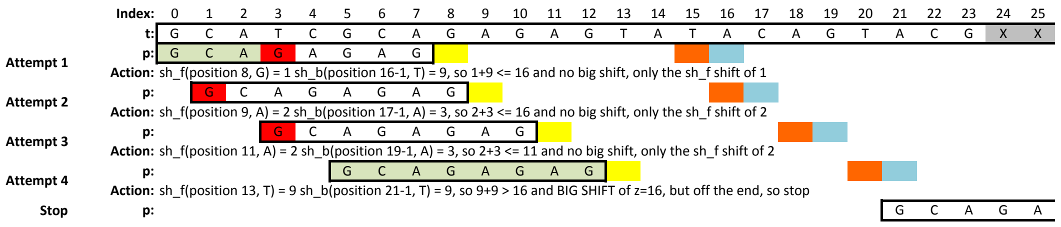
z is	< 0	$= 0$	$= 1$	$\in [2, m + 2)$	$\in [m + 2, 2m + 2)$	$> 2m + 1$
$z > shf$	**false**	**false**	**false**	**depends**	**true**	**true**
$z < shf + shb$	**true**	**true**	**true**	**depends**	**depends**	**false**
$z \in [shf + 1, shf + shb)$	**false**	**false**	**false**	**depends**	**depends**	**false**
Offset of i	z	z	z	z or shf	z or shf	shf
Worst case outcome	Array error	Loop error	Naïve alg	$(z < shf) \wedge i = i + z$	$(z \geq shf + shb) \wedge i = i + shf$	Needless work

Table 1: Consequences of different z value choices

2.3 QLQS Behaviour

The foregoing provides a basis for theoretically assessing the behaviour of QLQS. Note that, in comparison to QS, every shift of this QLQS algorithm requires an additional table lookup and the execution of a conditional statement instead of a simple assignment statement. The potential gain for the extra computational workload is longer shifts.

For illustrative purposes, Figure 4 shows an example (taken from from [2]) of four match attempts that occur when searching for matches of $p = \texttt{GCAGAGAG}$ in a string $t \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}^{24}$. Note that t is appropriately padded at the end with X's and the forward and backward shift tables are provided in Table 2.


Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
t: G C A T C G C A G A G A G T A T A C A G T A C G X X
Attempt 1
p: G C A G A G A G
Attempt 2
p: G C A G A G A G
Attempt 3
p: G C A G A G A G
Attempt 4
p: G C A G A G A G
Stop
p: G C A G A

Action: sh_f(position 8, G) = 1 sh_b(position 16-1, T) = 9, so 1+9 <= 16 and no big shift, only the sh_f shift of 1

Action: sh_f(position 9, A) = 2 sh_b(position 17-1, A) = 3, so 2+3 <= 16 and no big shift, only the sh_f shift of 2

Action: sh_f(position 11, A) = 2 sh_b(position 19-1, A) = 3, so 2+3 <= 11 and no big shift, only the sh_f shift of 2

Action: sh_f(position 13, T) = 9 sh_b(position 21-1, T) = 9, so 9+9 > 16 and BIG SHIFT of z=16, but off the end, so stop

Figure 4: An example from [2]

Σ	A	C	G	T	X
shf	2	7	1	9	9
shb	3	2	1	9	9

Table 2: Shift tables for $p = \texttt{gcagagag}$

Clearly, in the best case, the maximum shift should occur at every iteration. This will be the case if z is at its maximum $(2m+1)$ and t and p are such that predicate (3) is satisfied in each iteration. Such a scenario can be constructed by choosing t and p to have disjoint character sets. In such a case, each shift of QLQS will be $2m+1$, exceeding QS's maximal shift of $m+1$ by m. The upper bound on the total number of shifts is $\lceil \frac{|t|}{(2m+1)} \rceil$ compared to $\lceil \frac{|t|}{(m+1)} \rceil$ for QS.

Figure 5 illustrates how windows slide under best conditions, both for QS and QLQS . Text $t = \texttt{a}^{23}$ is padded at the end with X's and $p = \texttt{01234}$ is used. The four match attempts required by QS are shown, where *shf* is is looked up at $t[5], t[11], t[17]$ and $t[23]$. QLQS requires two match attempts. The first needs *shf* and *shb* lookups at $t[5]$ and $t[11]$ respectively; and the second at $t[16]$ and $t[22]$ respectively.

Worst case behaviour is manifested in both QLQS and QS if every window in t matches p. This is the case for both algorithms when $|\Sigma| = 1$. We have already pointed out that this worst case behaviour will also be exhibited if QLQS is run with z chosen as 1. In such instances, both algorithms can slide ahead by only one position at each iteration, each therefore execute $(|t| - |p|)$ iterations.

For randomly chosen t and p when $|\Sigma| > 1$, behaviour will be consistent with the analysis given in Subsection 2.2 and summarised in Table 1. Randomness implies that the values for *shf* and *shb* are randomly distributed over the interval $[1, m+2)$—i.e. choosing a random character from Σ and indexing either shift table on this character, is equally likely to return any of the integers in the interval $[1, m+2)$.

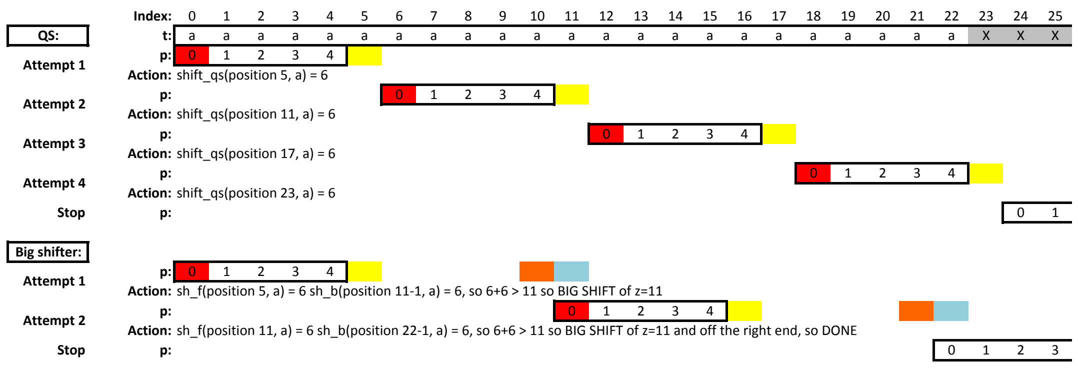

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
QS:
t: a a a a a a a a a a a a a a a a a a a a a a a X X X
Attempt 1
p: 0 1 2 3 4
Action: shift_qs(position 5, a) = 6
Attempt 2
p: 0 1 2 3 4
Action: shift_qs(position 11, a) = 6
Attempt 3
p: 0 1 2 3 4
Action: shift_qs(position 17, a) = 6
Attempt 4
p: 0 1 2 3 4
Action: shift_qs(position 23, a) = 6
Stop
p: 0 1
Big shifter:
Attempt 1
p: 0 1 2 3 4
Action: sh_f(position 5, a) = 6 sh_b(position 11-1, a) = 6, so 6+6 > 11 so BIG SHIFT of z=11
Attempt 2
p: 0 1 2 3 4
Action: sh_f(position 11, a) = 6 sh_b(position 22-1, a) = 6, so 6+6 > 11 so BIG SHIFT of z=11 and off the right end, so DONE
Stop
p: 0 1 2 3

Figure 5: Example of best case behaviour

Noting that *shf* and *shb* are bound from above, and that predicate (2) is satisfied if their sum exceeds z, it is clear that the larger z that is chosen, the smaller is the probability that $shf + shb > z$ will hold and thus, the smaller the chance that z will be used as the offset for the next match attempt. This observation mitigates against using large z values in the presence of randomness. On the other hand, as indicated in Table 1, if $z < m + 2$ then $z < shf$ may be selected as the offset. The smaller the value of z that is chosen, the more likely this is to happen, whereas it definitely does not happen if $z \in [m + 2, 2m + 2)$. This observation mitigates against using small values of z.

It is beyond the scope of this research to take further these observations about random behaviour and derive symbolic expressions for statistical metrics such as the expected value or standard deviation. Instead, Section 3 reports on the empirical analysis undertaken in respect of the performance of QLQS on commonly used benchmarking data.

2.4 Degenerate Forms of QLQS

It is instructive to consider in more detail the behaviour of QLQS in the degenerate case when $|p| = 1$. Table 3 reflects this behaviour. It shows the shifts when $z = q, \ldots 4$, the pattern is the single character c and the current window in t is at i and $t[i, i+4) = c_1c_2c_3c_4$. Under these circumstances, *shf* and *shb* can ony assume the values 1 (when indexed by c) or 2 (when indexed by anything other than c). The table then shows the various possible outcomes.

- As pointed out above, when $z = 1(= m)$ the shift is always 1. Sometimes this will be when QS could do a shift by 2 and so a non-optimal shift will occur.
- When $z = 2(= m + 1)$ then QLQS shifts always correspond to QS shifts.
- When $z = 3(= 2m + 1)$ then QLQS in one instance shifts ahead by 3 positions—something that QS could not do.
- When $z = 4(= 2m + 2)$ then behaviour is as expected: only conventional QS shifts can be made, at slightly additional computational expense.

z	$shf = 1 \wedge shb = 1$	$shf = 1 \wedge shb = 2$	$shf = 2 \wedge shb = 1$	$shf = 2 \wedge shb = 2$
$z = 1$ Shift	$c_2 = c \wedge c_1 = c$ z	$c_2 = c \wedge c_1 \neq c$ z	$c_2 \neq c \wedge c_1 = c$ z Non-optimal	$c_2 \neq c \wedge c_1 \neq c$ z
$z = 2$ Shift	$c_2 = c$ shf	Infeasible z?	Infeasible z?	$c_2 \neq c$ z
$z = 3$ Shift	$c_2 = c \wedge c_3 = c$ shf	$c_2 = c \wedge c_3 \neq c$ shf	$c_2 \neq c \wedge c_3 = c$ shf	$c_2 \neq c \wedge c_3 \neq c$ z
$z = 4$ Shift	$c_2 = c \wedge c_4 = c$ shf	$c_2 = c \wedge c_4 \neq c$ shf	$c_2 \neq c \wedge c_4 = c$ shf	$c_2 \neq c \wedge c_4 \neq c$ shf

Table 3: QLQS shifts for $m = 1$, $p = c$ and $t[i, i + 4) = c_1c_2c_3c_4$.

3 The Results

3.1 Experimental Design

The hardware platform used for this study is a 17-inch Macbook Pro (early 2011), 2.2 GHz (peaks at 3.0 GHz turbo) Intel Core i7 Quad-core (with another 4 virtual cores), 8 GB of 1333 MHz DDR3 RAM, 256 KB of L2 Cache per core, 6 MB of L3 Cache. C Code was compiled with Gnu g++ Apple LLVM version 6.1.0 using optimisation -O3 and also optimisations `unroll-loops` and `unit-at-a-time` for performance.

A software framework reads the text, t, from a specified file. Each pattern, p, needed in the test is a substring of t of a designated length that starts at an index whose value is determined by a pseudo-random number generator. An arbitrary string matching function for a given t and p can be plugged into the framework, as well as routines to set up forward and backward shift tables based on p. The framework allows for specifying the number runs to take over the same data, for specifying the number of random patterns of a given length to generate and for specifying a range of different pattern lengths to use.

The framework was configured to always execute 5 runs over the same data and to generate 30 randomly selected patterns for each specified length. This configuration is in line with findings reported in [7] about appropriate statistical sample sizes and appropriate times to repeat a run over given data. It was additionally configured to generate patterns of length $m = 1, \ldots, 32, 256, 1024$.

C-coded versions of both QS and of QLQS (slightly altered for display purposes in Figure 2) were provided. In the case of QLQS, runs were executed for all z values in the range $[m, 2m+3)$. This provides data about scenarios described in the last two columns of Table 1, headed "$\in [m + 2, 2m + 2)$" and "$> 2m + 1$" respectively. It also provides for limited data about scenarios described in the preceding two columns of the table, namely those headed "$= 1$" and "$\in [2, m + 2)$". Although the boundary cases (i.e. when $z = m$ and $z = m + 1$) are always covered, data for $z \in [1, m)$ is only available for limited values of m.

We realised *ex post facto* that more complete data for $z \in [1, m)$ over *all* pattern lengths could also be of interest (albeit somewhat marginal) to investigate empirically how frequently QLQS selects shifts smaller than QS for such z values. Such data will be included in future investigations. Of course, there is no practical value in empirically investigating QLQS behaviour when $z < 1$ since that will lead to algorithmic errors — as indicated in the first two columns of Table 1.

As text data sources, the Bible file (approximately 4MB) and the Ecoli file (approximately 4MB) from the SMART corpus were used [5] were used. The distribution of alphabet symbols over the indices of the text from the latter file is conjectured to be random and so may be characterised as a random text. We shall refer to it as t_s to indicate that its alphabet is relatively small. The text based on the Bible file could be said to approximate randomness with respect to English text, but not with respect to the distribution of its alphabet symbols of text index positions. Since it contains 63 different symbols — roughly eight times more than the Ecoli file — it serves to represent QLQS behaviour over a (relatively) large alphabet and so we refer to it as t_ℓ. For convenience, the shift tables needed for QS and QLQS were implemented as arrays of size 256 for both these texts. It is conjectured that the speed and space implications for these overly large tables are negligible.

In addition to the foregoing arrangements for measuring "random" behaviour, the behaviour of QLQS and QS on best case data was also measured. Such data for both algorithms is easily constructed by using disjoint alphabets for p and t. Here a 4MB text was used and pattern lengths ranged over $m \in [1, 257)$. The theoretical best case performance for QLQS, is when $z = 2m + 1$. However, as a sanity check, times were taken for all z values in the range $[1, 2m + 2)$.

All timing data is gathered in nanoseconds but for reporting purposes these times are converted into milliseconds. In all our reporting we use the *minimum time* over the 5 runs on the same data item to eliminate possible outlier timings caused by unscheduled operating system effects.

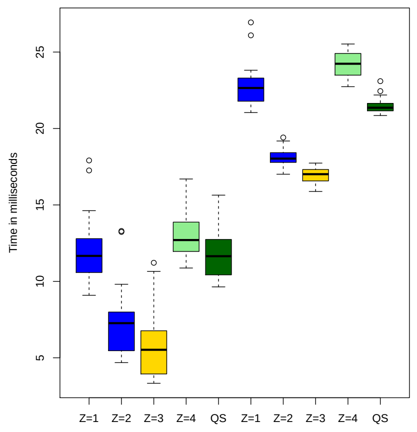
The timing for these runs *excludes* precomputational time required to set up the shift tables — in contrast to benchmarking frameworks such as SMART. In practical use-cases, such as network security, antivirus, etc., the precomputation of the shift tables is done once (per keyword, often offline on a server), while the string matching is conducted repeatedly over large (sometimes unending) input strings. As a result, the time required is quickly overwhelmed by the string processing time for a large input string.

3.2 Outcomes: The Broad Picture

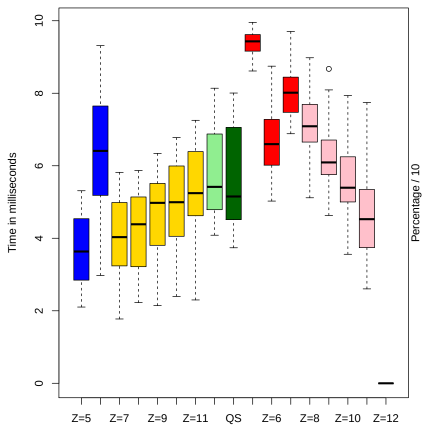
The subfigures of Figures 6 and 7 have been selected to illustrate one or more representative features of the data. Each subfigure contains, for a specified pattern length, several box-and-whisker plots. Such a plot indicates the median, quartiles and outlier regions in relation to measurements (in y-axis units) over a sample. Generally these measurements pertain to a sample of QLQS data for the z value given on the x-axis. Where the measurements relate instead to QS, this is also indicated on the x-axis. In the present instance, the sample is the set of 30 randomly generated patterns of the length under consideration in the subfigure. These lengths are $m = 1, 5, 1024$ and 1024 for Subfigures 6a, 6b, 7a and 7b respectively.

The plots in Subfigure 6a on the left hand side refer to time performance of QLQS and QS for runs over t_ℓ. The same plots are given on the right hand side for runs over t_s. In each case $m = 1$ and $z = 1, \ldots, 4$. Subfigures 6b and 7a give similar plots for time performance for QLQS and QS, but for $m = 5$ and $m = 1024$ respectively. Subfigures 6b and 7b incorporate plots for data described in the captions as %QLQS shifts. By this is meant the percentage of all shifts in a run to the window $t[i + z]$ instead of the conventional QS shift to $t[i + \mathit{shf}]$. (Note that the data in Subfigure 6b has been scaled by a factor of 10 to keep within the range of the performance data in the same subfigure.

As a visual aid, plots of particular interest are coloured according to the following guidelines: plots relating to time performance are in blue, gold or green and plots relating to %QLQS shift data are in red or pink. The blue and red plots relate to z values that merit particular attention. Plots for $z = 2m + 2$ are coloured in light green and QS plots are coloured in dark green. The plots in Subfigure 6a are for



(a) t_ℓ vs t_s times
$m = 1$ and $z = 1, \ldots, 4$



(b) Time compare to % QLQS shifts
$m = 5$ and $z = 5, \ldots, 12$

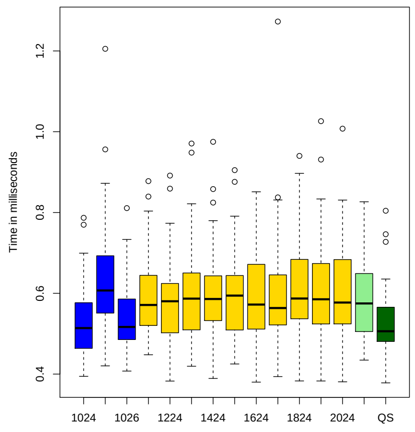
Figure 6: QLQS performance for smaller patterns

patterns of length 1. There are undoubtedly more efficient single character search algorithms than these degenerate instances of QS and QLQS. Nevertheless, the data usefully illustrates a number of relevant broader trends and issues. Median times for t_ℓ range between about 50 and 130 milliseconds compared to a range of about 160 to 240 milliseconds for t_s. Also, t_s plots show data spread over shorter ranges than corresponding t_ℓ plots. These differences in behaviour in respect of strings from small and large alphabets is consistent with other string matching algorithms and can be explained from first principles. However, the subfigure also illustrates that, despite these differences, trends in the data from t_s to t_ℓ are similar. For example, the ranking of the medians for $z = 1, \ldots 4$ and QS is exactly the same for the t_s and t_ℓ data. Comparison of the other t_s and t_ℓ data confirmed this broad correspondence. Consequently, nothing of interest is missed by limiting further discussion here to the t_ℓ-derived data.

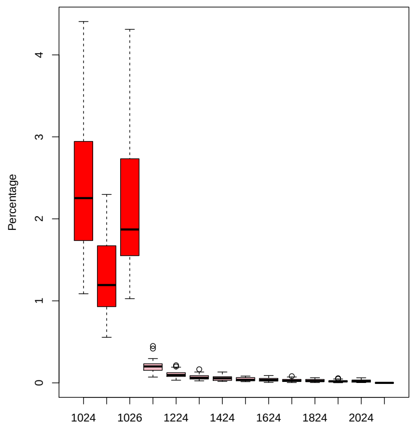
Subfigure 6a shows that QLQS significantly improves on the speed of QS for $z = 2$ and 3, more than doubling it for $z = 3$. Its speed is more or less the same as QS for $z = 1$ and worse for $z = 4$. This is consistent with the following generalisation about data from all pattern lengths:

> QLQS's *best performance over all z values* tends to be better than QS for relatively small m but that advantage is eventually lost as m increases.

By the time $m = 1024$, Subfigure 7a confirms that QS outperforms QLQS for all values of z. (At $m = 32$, QLQS outperforms QS for several z values. This data is not shown here.)



(a) Time for $m = 1024$,
$z = 1024, 1025, 1026, 1124, \ldots, 2024, 2050$



(b) % QLQS shifts for $m = 1024$,
$z = 1024, 1025, 1026, 1124, \ldots, 2024, 2050$

Figure 7: QLQS performance for larger patterns

Visual inspection of the subfigures that the median of the light green plot is always greater than that of the dark green plot. This points to a general feature that is evident in all the data, namely that QS always outperforms QLQS when $z = m+2$. Of course, this is to be expected because, as observed in the last column of Table 1, when $z > m + 1$ no QLQS shifts are made. Thus the QLQS algorithm behaves just as QS, but incurs additional computational complexity.

The %QLQS plots in Subfigures 6b and 7b show explicitly that there are no QLQS shifts when $z = 2m + 2$. Furthermore, both these subfigures show that as z increases, the probability diminishes of doing a %QLQS shift. There is one exception to this trend and that is when $z = m + 1$. Once again the subfigures are typical of all pattern sizes. The general trend is explicable. As z increases it becomes less and less likely to comply with predicate (2), i.e. large values of z are less likely to be smaller than $shf + shb$, and therefore less likely to be selected for the next shift.

When $z = m+1$, there is a significant dip in %QLQS shifts, and there is an accompanying worsening of QLQS time performance. This peculiar behaviour is manifested in all the data for QLQS when $m \geq 4$. It can be seen in the performance plots in Subfigures 6b and 7a as well as in the %QLQS plots in Subfigures 6b and 7b.

To explain the dip in %QLQS shifts, note that when $z = m + 1$ then the same character, say c, in t is being used to index into the forward and backward shift tables and retrieve a value for *shf* and *shb*. If c does not occur in p then *shf* $=$ *shb* $= m + 1$ so that $z <$ *shf* $+$ *shb* $= 2m + 2$. Since predicate (2) holds, z will be used as the offset for the next move.

Suppose c occurs one or more times in p. By definition *shf* is the length of the suffix in p that begins at the leftmost occurrence of c and *shb* the length of the prefix of p that has the leftmost occurrence of c as its last element. The maximal value of *shf* $+$ *shb* $= m + 1$ and occurs when there is only one instance of c in p. In this case predicate (2) is not satisfied (equality holds) and so the offset used for the next move is *shf*.

We conjecture that the dip in time performance relates to the way in which the operating system and compiler handle the aliasing that arises in line 9 of the code in Figure 2 — the same location in text vector `T` is referenced by two different index expressions.

The foregoing means that all the plots for %QLQS shifts when $z = m + 1$ reflects the percentage of times that a character does not appear in the pattern but appears just to the right of a window used in the text.

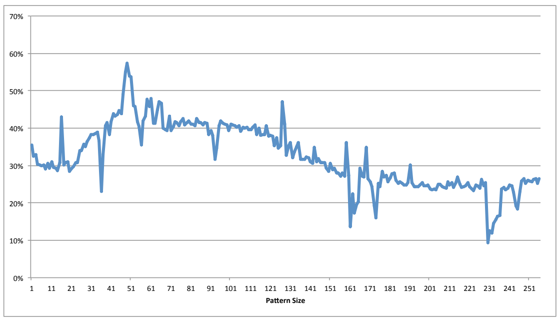


Figure 8: Best Case Performance Ratios: $\frac{QLQS}{QS}\%$ for $m \in [1, 257)$

Figure 8 graphs the performance ratio of QLQS time divided by QS time under the best case data previously outlined — disjoint pattern and text alphabets and $z = 2m + 1$. The average and standard deviation of these observations is 23% and 9% respectively, with maximum and minimum values of about 58% and 9%. Thus, QLQS on average performs at about one third of QS's speed over all pattern lengths tested.

4 Conclusion

We have presented a new algorithm for single-keyword string pattern matching. The algorithm has a number of interesting properties:

- It outperforms Sunday's QS algorithm in most cases with an appropriate choice of z.
- QLQS significantly outperforms QS when the pattern consists of letters not appearing in the input text. In the best case, the two subalphabets are disjoint and the QLQS double's QS's performance, making half the number of match attempts.
- While large z choices appear to violate the principle that safe shifts larger than $m + 1$ are not possible, QLQS in fact makes the same number of table lookups as QS — though uses them considerably more efficiently thanks to instruction-level parallelism.
- Significant instruction-level parallelism is used by modern processors (in this case Intel i7) to enable simultaneous shift lookups and simple arithmetic.
- The algorithm structure is as simple as Sunday's QS, and considerably simpler than many similar recent algorithms.
- The shift tables are easily computed and closely related to Sunday's QS.
- This appears to be the first left to right algorithm using a backward shift distance.
- QLQS is an example of a *speculative execution* (take a Quantum Leap/shift, then check if it was valid) algorithm.

There are several possible enhancements to this algorithm, as well as other areas to use the Quantum Leap principle:

- Simplify QLQS to be a probabilistic algorithm in which the validity of a z shift is not checked. Measure such an algorithm over a range of z to determine the probability of missed matches.
- Explore opportunities for coarse-grained parallelism in this style of algorithm.
- Benchmark QLQS using two dimensional shift tables (as opposed to two one dimensional tables).
- Characterize the performance of QLQS on processors unable to use instruction-level parallelism or with vastly different cache memory sizes (compared to the i7).
- Apply the Quantum Leap principle to Boyer-Moore style algorithms in other pattern matching areas such as multiple-keyword, regular expression, tree, and multi-dimensional pattern matching.
- The shift tables used in QLQS should be formally derived in a correctness-by-construction algorithm formalism.

References

1. 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.
2. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's College Publications, 2004.
3. 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.
4. M. A. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific Publishing Company, 2003.
5. S. Faro and T. Lecroq: *2001–2010: Ten years of exact string matching algorithms*, in Proceedings of the Prague Stringology Conference 2011, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2011, pp. 1–2.
6. A. Hume and D. Sunday: *Fast string searching.* Software — Practice & Experience, 21(11) 1991, pp. 1221–1248.
7. D. G. Kourie, B. W. Watson, T. Strauss, L. Cleophas, and M. Mauch: *Empirically assessing algorithm performance*, in Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014, SAICSIT '14, New York, NY, USA, 2014, ACM, pp. 115–125.
8. W. F. Smyth: *Computing Patterns in Strings*, Addison-Wesley, 2003.
9. D. M. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) Aug. 1990, pp. 132–142.
10. 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.