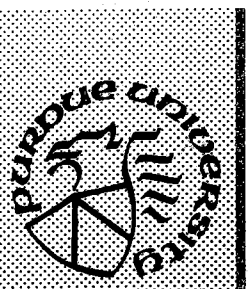
2-1-1989

Searching for Fixed-Length Patterns

R. W. Quong
Purdue University



Searching for Fixed-Length Patterns

R. W. Quong

TR-EE 89-15
February, 1989

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Searching for Fixed-Length Patterns

Russell W. Quong
Dept. of Electrical Engineering
Purdue University

Abstract

We present an algorithm, RQ for finding all occurrences of a fixed-length pattern, $p_1, p_2, \cdots, p_P$, in a text string, where each p_i can match an arbitrary set of characters. Our algorithm is optimal in that it examines the minimum average number of text characters, which is not necessarily the same as being optimal in running time. This paper answers the question of optimal string searching put forth in [KMP77].

Let α = the alphabet size, P= the length of the string matched by the pattern, T= the length of the text, W= the word size in bits of the underlying machine, and $\overline{\Phi(RQ)}$ = the average number of text characters examined RQ. We derive an asymptotic approximation for $\overline{\Phi(RQ)}$ when P$\leq \alpha$. We also show that $\overline{\Phi(RQ)} < (4 \log_\alpha P/3)(T/P)$, when P$\gg \alpha$. In the worst case, RQ examines T characters. Our algorithm requires space $O(\|\Pi\|\lceil P/W \rceil)$. In addition, our method of analysis is applicable to other algorithms modeled by a finite automaton.

We present an efficient implementation of our algorithm when $P < W$. In practice, compared to the Boyer-Moore algorithm, RQ requires slightly more space, accepts a more general range of patterns, and runs in comparable time.

1 Introduction

A fundamental problem is finding all occurrences of a *pattern* in a *text* string. This problem occurs in many applications such as text editing, archival search, DNA-protein matching, and programming. For example, a programmer might search source files for all instances of a procedure name to find where that procedure is defined or called. In many cases, the text string is very long.

The number of text characters examined is a common metric for the running time of a pattern-matching or string-searching algorithm [KMP77] [BM77] [Yao79]. This metric is valid as long as the rest of the algorithm takes $O(1)$ time for each character examined. At one extreme, every character in the text must be examined, if the text consists entirely of (possibly overlapping) concatenations of the pattern. At the other extreme, only $\lfloor T/P \rfloor$ characters need to be examined, if the text consists entirely of characters not occurring in the pattern, where P is the length of the pattern, and T is the length of the text. In this case, examining every P^{th} text character rules out all occurrences. The optimal algorithm minimizes the average number of characters examined over all patterns and text strings.

In general, the pattern may match one or many possible text substrings. A simple "fixed-string" pattern, such as bathroom, matches just one substring. A complex pattern, such as a regular expression, might match a large, possibly infinite, set of substrings. For example, the regular expression a@*z matches all text substrings of length two or greater, starting with an a and ending with a z. The "wildcard" character @ matches any single text character; the kleene closure operator is *. Thus, @* matches any sequence of zero or more text characters.

A *fixed-length pattern* $p_1, p_2, \cdots, p_P$ is similar to a fixed-string, except that each p_i can match an arbitrary set of characters. For example, the pattern [Bb][Aa][Tt][Hh]room matches all occurrences of "bathroom" regardless of the case of the first four letters. Fixed-length patterns are a generalization of fixed-strings, and can match constructs such as "the beginning of a word". For example, if WS is the set of white space characters and ALPHA is the set of alphanumeric characters, the pattern [WS][ALPHA] finds the first letter of every word.

In this paper, we present the optimum algorithm for finding all occurrences of a fixed-length pattern in a (longer) text string. Our algorithm, RQ, is optimal because for any fixed-length pattern, it examines the minimum average number of text characters over all text strings. RQ improves upon previous algorithms because RQ is not limited to fixed-string patterns. In practice, variations of our algorithm can be efficiently implemented for differing pattern lengths.

The reminder of this paper gives definitions and then reviews previous string searching algorithms giving their space and time requirements. Section 4 gives an informal description of our algorithm with examples of its behavior. In Section 5 we present a general method for analyzing pattern-matching algorithms and we derive an asymptotic approximation for the running time of our algorithm when the pattern length is less than the effective alphabet size. Section 6 proves that RQ is optimal. Section 7 views other algorithms as variants of RQ. Section 8 compares our approximations with measured results and compares RQ with the Boyer-Moore algorithm.

2 Definitions

An *alphabet* is a finite, nonempty set of symbols or *characters*. The alphabet, Π, is the set of possible text characters. For standard English text, Π consists of lower case letters, upper case letters, numerical digits, punctuation characters, and whitespace. The size of the alphabet is $\|\Pi\|$. In this paper, c stands for a character in Π; pr_c is the probability that a random text char is c.

A string S of length l is a concatenation of l characters, $c_1 c_2 \cdots c_l$. The length of S is $\|S\|$. If S is a string, $S[i]$ stands for the i^{th} character of S where i is an *index* or *position* of S. The first character of S is $S[1]$. A string S' of length l' is a *substring* of the string S with length l if $l' \leq l$ and $S' = S[i]S[i+1]\cdots S[i+l'-1]$ for some i, $1 \leq i \leq l - l' + 1$. $S[i...j]$ denotes the substring $S[i]S[i+1]\cdots S[j]$. Two strings are equal if they are substrings of each other. If $S =$ abcdefgh, then $S[2] =$ b, $S[4...6] =$ def, and $\|S\| = 8$.

We use the notational convenience $[c_0 c_1 c_2]$ to represent the set $\{c_0, c_1, c_2\}$ where c_0, c_1, and c_2 are single characters. For example, the fixed-length pattern [hs][aio]t matches the six strings: hat, hit, hot, sat, sit, and sot.

Patt[i] represents the set of all characters that match the i^{th} character of the pattern; $\|patt[i]\|$ is the size of the set. If patt[i] includes or "matches" c then $c \sim$ patt[i]; otherwise $c \neq$ patt[i]. The length of a pattern is the length of the strings it matches. The average size of patt[i] is $\overline{\text{patt}[\,]} = (\sum_{i=1}^{P} \|patt[i]\|)/P$. For a fixed-string pattern $\overline{\text{patt}[\,]} = 1$. Thus, for pattern = [hs][aio]t, $\|pattern\| = 3$, patt[1] = {h,s}, $\|patt[2]\| = 3$, a $\sim$ patt[2], e $\neq$ patt[2], and $\overline{\text{patt}[\,]} = 2$.

We set P= ||pattern||, and T= ||text||.

The probability that a random text char will match a random pattern position is q; the probability that they do not match is $\bar{q} = 1 - q$. Usually, the character distribution over all patterns is the same as that over

all text strings, so that

$$q = \overline{\text{patt}[\,]} \sum_{c \in \Pi} \text{pr}_c{}^2.$$

(The distributions would differ if we were to restrict our searches to patterns consisting of the first half of the alphabet, as opposed to the whole alphabet).

The *effective alphabet size*, α, is $1/q$. If the pattern is a fixed-string and all characters in Π have equal probability, $q = 1/\|\Pi\| = 1/\alpha$. With unequal character distributions, α can be much smaller than $\|\Pi\|$. For example, when searching for fixed-strings in English text, $\alpha \approx 11$ [BM77]. In this paper, we assume the text is randomly distributed, so that for any n, all substrings of length n are equally likely. English is not random, because for $n = 3$, ing is much more common than xqz.

A pattern-matching algorithm makes a *decision at (index)* η when it finds that text$[\eta...\eta + P - 1]$ either matches or cannot match the pattern. In the former case, the pattern *completely matches* at η; in the latter case, the text *fails* to match at η. If a decision has not been made at η, there is a *partial match at* η. Initially, there is a partial match at everywhere.

We *examine* or *probe* text characters. Only the probes within text$[\eta...\eta + P - 1]$ can decide at η. Examining a character *looks at* i if it might decide at i. Probing text$[\eta]$ looks at the undecided indices between $\eta - P + 1, \cdots, \eta$. A pattern matching algorithm must make decisions at η, for $1 \le \eta \le$ T-P+1. The following example in Figure 1 shows a search for a@ba. Examining text[9] yields two decisions. Examined text characters are underlined; an unexamined character is shown as an X.

```
pattern = a @ b a,      where @ matches any character
index   1 2 3 4 5 6 7 8 9 10
text = X X X d c a a b a X X X X
(1)                   a @ b a        (decision — MATCH)
(2)                     a @ b a      (decision — NO; fail)
(3)                       a @ b a    (no decision yet — partial match)
(4)                       a @ b a    (mismatch from before; when "b" examined)
```

Figure 1: Making a decision at 6, and 7.

The set of indices of text[ ] where decisions have been made is Δ. We also view Δ as a table indexed by indices of text[ ] with

$$\Delta[x] = \begin{cases} \textit{yes} & \text{pattern completely matches at } x \\ \oslash & \text{pattern fails at } x \\ \diamond & \text{no decision made at } x \text{ yet} \end{cases}$$

We limit our analysis to *monotonic search algorithms* which pick the smallest undecided text index η and examine characters until a decision at η is made. These algorithms monotonically increase η from 1 to $T - P + 1$. In the process of deciding at η, the algorithm may make other decisions. In Figure 1, $\eta = 6$, and examining text[9] makes decisions at 6 and 7. A decision at 8 was made previously. A partial match exists

at 9, and thus, the algorithm continues with $\eta = 9$. Note that we have not specified which character in text$[\eta...\eta + P - 1]$ to examine when trying to decide at η. In practice, all practical algorithms are monotonic.

In the rest of this paper, η = the smallest undecided index of text[ ], and text$'$ = text$[\eta...\eta + P - 1]$. Thus, text$'$ is a moving "window" into text[ ] of size P, which contains the next probe. As η increases, text$'$ moves to the right of text.

For an algorithm, A, the average number of text probes is $\overline{\Phi(A)}$; the worst case number of probes is $\Phi_{wc}(A)$. The average case *running time* of an algorithm A is not necessarily the same as $\overline{\Phi(A)}$, because each probe may require more than $O(1)$ time to process. However, except for RQ, the running time for every algorithm described in this paper is proportional to $\overline{\Phi()}$, and the two measures are interchangeable. The total running time of an algorithm is the sum of the preprocessing time and the search time. Most of the algorithms discussed in this paper preprocess the pattern creating auxiliary tables to speed the search. For long text strings, the search time dominates the total running time. In the rest of this paper, the running time refers to search time alone.

3 Previous Fixed-String Algorithms

In the spring of 1974, R. Boyer and J. Moore [BM77] and (independently) R.W. Gosper developed a breakthrough algorithm, BM, with an average case search time significantly faster than previously known techniques. The key to their algorithm is checking for the pattern backwards. In the typical case, each text character examined makes several decisions. Their algorithm runs in "sublinear" time with respect to P, because significantly fewer than T characters are examined on the average. The average case running time of BM depends on q and P; interestingly, it decreases as P increases. In particular, when the length of the pattern is short compared to α, $\overline{\Phi(BM)}$ is roughly T/P. For example, searching for bath in standard English would require examining approximately one fourth of the text characters. BM preprocesses the pattern creating tables of length P and $\|\Pi\|$.

Analysis of $\overline{\Phi(BM)}$ and $\Phi_{wc}(BM)$ is not simple. Guibas and Odlyzko [GO80] proved the worst case running time of BM is $4T$, if the pattern is not found. Apostolico and Giancarlo [AG86] proved a bound of $2T$ for a Boyer-Moore variant that partially remembers which characters have been examined. Schaback [Sch88] numerically examined the average case running time of a simplified algorithm. Later in this paper, we show that $\overline{\Phi(BM)} \approx Tq/(\bar{q}(1 - \bar{q}^P))$, when $P \leq \alpha$.

Knuth, Morris, and Pratt [KMP77] describe a string search algorithm, KMP, that examines each text character once. Unlike the BM and RQ, their algorithm does not "backtrack". Both $\Phi_{wc}(BM)$ and $\Phi_{wc}(KMP)$ are linear $O(T)$, because of auxiliary tables which guide the algorithms when a mismatch occurs. However, these tables cannot be used in the presence of a wildcard character (@) or character sets such as [abc].

The brute force approach tests whether the pattern matches the text substring starting at text$[\eta]$ for $1 \leq \eta \leq T - P + 1$. The test compares patt[1] with text$[\eta]$, and then patt[2] with text$[\eta + 1]$, and so on until it makes a decision at η. Next, η is incremented by 1, and the process repeats itself. In the worst case, the mismatch does not occur until the last pattern character is checked giving a running time of PT. For example, the worst case occurs when pattern = "aaab" and text = "aaaa$\cdots$aaaa." However, this case is unlikely and the average case running time is $T/(1-q)$. This algorithm works for fixed-length patterns, too.

In practice, Smit [Smi82] found that KMP does not run significantly faster than the brute force algorithm, because for large $\|\Pi\|$, the mismatch usually occurs at patt[1]. Horspool [Hor80] found BM faster than specialized string searching instructions in hardware for $P \geq 5$ when searching English text.

Like KMP, Aho and Corasick [AC75] also used a failure table in their algorithm (AC) to find if any of a set of keywords exists in the text. The failure table is constructed from a trie, which detects common prefixes in the keywords. Compared to KMP, the AC algorithm can search for more than one pattern "in parallel" with no loss in efficiency.

Harrison [Har71] first suggested the use of hashing as a probabilistic method of speeding string search. Every substring of length L in the pattern is hashed into a bit table forming a "key signature". Another signature is formed by hashing each line of the text similarly. If the text line contains the pattern, the signature for the line will "bit-wise" contain the key signature. If this condition succeeds, the pattern and text are compared to check if pattern actually does occur. Otherwise, the line does not contain the pattern. This method assume the text is broken into discrete partitions such as lines, ruling it out for general data searching.

Rabin and Karp [KR81] give another algorithm using hashing with an average case running time of $O(T)$. Their method consists of calculating a hash value, h(j), for every text substring text[j..j+P-1] where $1 \leq j \leq T - P - 1$. Each hash value is compared with the hashed value of the pattern. If the values match, the chances are good that pattern is to be found at that point, and the actual strings are compared. Rabin and Karp found hash functions such that calculating h(j+1) requires $O(1)$ time given h(j). Thus, their average case running time is $O(T)$ with an extremely unlikely worst running time of $O(TP)$. Table 1 summarizes the running time and space requirements of these algorithms.

Algorithm	Space	Running Time			
		Preprocessing	Worst	Avg	Best
brute force	0	0	TP	T/(1-q)	T
KMP	P	P	T	T	T
Aho-Corasick	P	$O(P)$	T	T	T
Boyer-Moore	$\|\Pi\|$+ P	$O(T)$	$O(T)$	*	T/P
Rabin-Karp	O(1)	P	$O(PT)$	T	T

* = no known simple form.

Table 1: Space & time requirements for fixed-string searching algorithms.

Knuth, Morris, and Pratt [KMP77] also pose the question of optimal string searching, and give an answer when P= 2. We call their algorithm OPT2. They state that "the analysis (of OPT2) is not completely trivial even for this case", and show that

$$\overline{\Phi(OPT2)} = \frac{T}{2}\frac{(1+q-q^2)}{(1-\frac{q}{2})} - \frac{(1-q)(1+2q-q^2)}{(2-q)^2} + \frac{(q-1)^T}{(1-q)(2-q)^2}. \quad (1)$$

For large T, the first term dominates Equation 1. In Section 5.2, we shall rederive the first term an asymptotic approximation to the running time of OPT2. Finally, Yao [Yao79] proved that for most fixed-string patterns, the minimum average number of characters examined is $O(T\lceil \log_q P \rceil / P)$, when $T > 2P$.

However, there are "worst-case" patterns that require more characters to be examined regardless of the text string.

Abrahamson [Abr87] explores the lower bound of the time-space requirements when searching for fixed-length patterns. His work differs from ours because he consider alphabets of infinite size, where each character is represented as a string from a finite alphabet.

After discussing OPT2, Knuth, Morris, and Pratt pose the question of finding the optimum searching strategy for patterns of length greater than 2. This paper answers that question.

4 The RQ algorithm

In this section, we describe our algorithm informally by giving an example of its execution. Next, we give a formal description of the algorithm. We also describe how to implement our algorithm efficiently if the pattern is short.

4.1 Informal Description

First, we describe our algorithm by example. Consider the search for abca in the text string abcbacabcaabb. As before, underlined characters have been examined, and the current probe has an arrow over it. We print out the current value of η and underline the indices of text'.

```
index   1 2 3 4 5 6 7 8 9 10          η = 1
text    a b c b a c a b c a a b b     decision at
pattern a b c a                       1 fail
          a b c a                     2 fail
            a b c a                   3 partial match
              a b c a                 4 fail
```

Examing text[4] shows that pattern matches neither text[1...4], text[2...5] nor text[4...7], thus we have made decisions at 1, 2 and 4. We have a partial match at 3. We examine text[6] next.

```
index   1 2 3 4 5 6 7 8 9 10  12      η = 3
text    a b c b a c a b c a a b b     decision at
            a b c a                   3 fail
              a b c a                 4 fail from before
                a b c a               5 fail
                  a b c a             6 fail
```

We find that pattern fails to match text[3...6], text[5...8] or text[6...9]. We already know text[4...7] failed, so that we have made decisions at {1,2,3,4,5,6}. We try to decide at 7, examining the rightmost unexamined

character in text[7...10], namely text[10].

```
index   1 2 3 4 5 6 7 8 9 10  12        η = 7
text    a b c b a c a b c a a b b       decision at
                      a b c ✓a          7 partial match
                        a b c× a        8 fail
                          a b× c a      9 fail
                            ✓a b c a    10 partial match
```

Text[7...10] has a partial match, so we examine text[9] and then text[8] and still have a partial match at 7. Examining text[7] decides at 7 — a complete match. A total of 6 characters have been examined.

```
index   1 2 3 4 5 6 7 8 9 10  12        η = 7
text    a b c b a c a b c a a b b       decision at
                      ✓a ✓b ✓c ✓a       7 COMPLETE match
```

4.2 Formal description

The RQ algorithm is simple: examine the rightmost unexamined character in text′, where text′ = text[η...η+$P-1$] and η is the leftmost undecided index of text[ ]. Figure 2 describes RQ. In the worst case, RQ examines T characters, because it examines each character once.

```
η := 1;
Δ[1...T-P+1] := ◇; (* partial match initially *)
Ex[1...T] := false; (* array of examined characters *)
while (η ≤ T − P + 1) do begin
    e := index of rightmost unexamined char in text[η...η + P − 1];
    examine text[e];
    record decisions in Δ[ ];      (‡‡)
    Ex[e] := true;
    if text[η...η + P − 1] ~ patt[1...P] then
        match found at η;
    end;
    η :=leftmost undecided position;
end;.
```

Figure 2: A high level description of the RQ algorithm.

Unfortunately, this algorithm requires three arrays of size $O(T)$: text[ ], Δ[ ], and Ex[ ]. Let Δ′[1...P] =

$\Delta[\eta...\eta+P-1]$ and $\text{Ex}'[1...\text{P}] = \text{Ex}[\eta...\eta+P-1]$. For all η, all access are to the shorter arrays, Δ', Ex', and text'.

The "hardest" step is (‡‡). However, a precomputed table can tell us what decisions are made upon examining character c. For $c \in \Pi$ and $1 \leq j \leq P$, dvec[c][j] indicates whether a decision at j is made when c is found. Thus,

$$\text{dvec[c][j]} = \begin{cases} \diamond & c \sim patt[P-j] \\ \oslash & c \neq patt[P-j]. \end{cases}$$

Define the binary operator $\oplus$ as follows

Δ'	dvec[c]	$\Delta' \oplus$ dvec[c]
$\diamond$	$\diamond$	$\diamond$
$\oslash$	$\diamond$	$\oslash$
$\diamond$	$\oslash$	$\oslash$
$\oslash$	$\oslash$	$\oslash$.

$\Delta' \oplus$ dvec[c] gives the new value of Δ', because dvec[c] is an array of decisions of length P. In practice, $\diamond$=1, $\oslash$=0, and thus, $\oplus$ is the logical AND operation. Figure 3 shows the algorithm for RQ.

4.3 An efficient implementation when P< W

If P is less than the word size, W, of the computer, then we can maintain $\Delta'[\,]$, and Ex'[ ] in a single register. Similarly, we can hold dvec[c] in a single register. Δ'[j] is the j^{th} bit of Δ'. We use the standard convention that bit 0 (lsb) is the rightmost bit. Incrementing η by η_{sh} shifts Δ' and Ex' by η_{sh} bits. Let ShiftLeft(Ex', s, **false**) be the operation that shifts Ex' left by s bits, filling the vacated locations with **false**. This operation becomes an "arithmetic shift word right by s bits" instruction, which is supported in hardware by most computers. We set text' as a pointer into text[ ] starting at text[η]. Finally, we can update all of Δ' with a single logical AND as shown in Figure 4, which presents a practical algorithm for RQ. On many computers, each step in this algorithm takes $O(1)$ time and thus, the running time is proportional to the number of characters examined.

Finally, we cover other possible implementation choices.

1. Tighten the main loop, by not checking if a character has been previously examined. Instead, probe text'[P], text'[P-1], ..., text'[1] until a decision is made at 1. Some characters may be examined more than once, but the simpler main loop more than compensates for the redundant probes. This algorithm is no longer RQ, and in the worse case might make $O(PT)$ probes. To minimize this case, keep Ex'[ ] when $\alpha \leq 3$, which is when redundant probes are most likely.

2. To determine η_{sh}, use a "shift" tablem, Shift_Δ, indexed by the lowest l bits of Δ. The table has size 2^l, where l is between 8 to 20 depending the space available. (We have used $l = \text{W}/2 = 16$.)

3. If a dynamic right shift is expensive, sacrifice space for time by precomputing the table dvec', where dvec'[c][e] = RightShift(dvec[c], P-e, $\diamond$).

4. To speed large shifts, use a shift table like the Boyer-Moore algorithm after examining text'[P].

Algorithm RQ.

```
η := 1;
Δ'[1...P] := ◇;
Ex'[1...T] := false;
text' = text[1...P];
while (η ≤ T − P + 1) do begin
    e := { i | max i, Ex'[i] = false, 1 ≤ i ≤ P }; (* rightmost unexamined char *)
    c := text'[e];
    Ex'[e] := true;
    for k := 1 to e do begin
        Δ'[k] := Δ'[k] ⊕ dvec[c][(P-e)+k];          (* †††*)
    end;
    if Ex'[1...P] = [true,...,true] and Δ'[1] = ◇ then
        match found at η;
    end;
    η_old := η;
    η := η − 1 + { j | min j, D'[j] = ◇, 1 ≤ j ≤ P} (* rightmost undecided index *)
    η_sh := eta − eta_old;
    if η_sh > 0 then
        text'[1...P-η_sh] := text'[1+η_sh...P];
        text'[P-η_sh+1...P] := text[η + patlen − η_sh...η + patlen − 1];
        Δ'[1...P-η_sh] := Δ'[1+η_sh...P];
        Δ'[P-η_sh+1...P] := ◇;
        Ex'[1...P-η_sh] := Ex'[1+η_sh...P];
        Ex'[P-η_sh+1...P] := false;
    end;
end;.
```

Figure 3: The RQ algorithm

We have tested the first three ideas in an algorithm called RQ'. Our machine supports dynamic shifts, so that option 3 gave no improvement. We show the main loop below, where l = the number of bits examined by the Shift_Δ table.

A practical RQ algorithm when $P < W$

```
η := 1;
Δ'[1...P] := ◇...◇; (* ◇= 1 *)
Ex'[1...T] := false,false,...,false;
text' = text[1...P];
while (η ≤ T − P + 1) do begin
    e:= P; (* find rightmost unexamined char *)
    while Ex'[e] = true do begin
        e := e-1;
    end;
    c := text'[e];
    Ex'[e] := true;
    Δ' := Δ' ∧ ShiftLeft(dvec[c],P-e,◇); (* logical and *)
    if Ex' = true,...,true and Δ'[1] = ◇ then
        match found at η;
    end;
    η_sh := 0;
    while Δ'[1] = ⊘ do begin (* find rightmost undecided index *)
        η_sh := η_sh + 1;
        ShiftLeft(Δ', 1, ◇);
    end
    η := η + η_sh;
    if η_sh > 0 then
        set text' pointing to text[η];
        ShiftLeft(Ex',η_sh,false);
    end;
end;.
```

Figure 4: The RQ algorithm when P≤ W.

An efficient non-optimal algorithm when $P < W$

```
η := 1;
Δ'[1...P] := ◇...◇; (* ◇= 1 *)
e := P;
while (η ≤ T − P + 1) do begin
    c := text[η+e-1];
    Δ' := Δ' ∧ RightShift(dvec[c], P-e, ◇);
    while Δ'[1] = ◇ do begin
        if e = 1 then
            match found at η;
            RightShift(Δ', 1, ◇);
            η := η + 1;
            e := P;
        else
            e := e - 1;
        end;
        c := text[η+e-1];
        Δ' := Δ' ∧ RightShift(dvec[c], P-e, ◇);
    end while;
    do      (* find rightmost undecided index *)
        slide := Shift_Δ[Δ'[1...l]];
        RightShift(Δ', slide, ◇);
        η := η + slide;
    while (Δ'[1] = ∅);
    e := P;
end while;.
```

Figure 5: RQ′, an efficient variation of RQ.

5 The Number of Probes when $P \leq \alpha$

In this section, we present a technique for analyzing the behavior of a finite automaton given certain restrictions. In particular, we show how to derive asymptotic approximations for the running time of a pattern matching algorithm. When the number of probable states is large, the calculation may be intractable. For RQ and BM, restricting $P \leq \alpha$ drastically reduces the number of states.

We model an algorithm as the actions of a finite automaton [ASU86]. Each probe causes a state transition. As the search progress, the finite automaton reaches a probabilistic steady state, or "eigenstate". The probabilistic transition diagram can be written as a matrix with entry b_{ij} corresponding to the probability of moving from state i to state j. The eigenvector of this matrix is the eigenstate of the finite automaton. The entries in the eigenstate are the probabilities of each state. We also calculate the expected number of decisions corresponding to each state. Throwing out states with low probabilities simplifies the analysis and yields the approximation. First, we give definitions, and then we apply our method on three algorithms, OPT2, BM, and RQ.

5.1 Definitions and Approach

We model a monotonic search algorithm as a finite automata, Γ, with *states*, $\xi_{\{E\}}^{\{M\}}$. M and E are sets of indices of text'. M is the *set of positions in* text' *that have partial matches.* E is the *set of character indices in* text' *that have been examined.* In Figure 1 *before* examining text[9], $\eta = 6$ and we are in state $\xi_{\{1,2,3\}}^{\{1,2,4\}}$ with text' = text[6...9] = aaba. There are partial matches at text[6,7,9] = text'[1,2,4], and we have examined text'[1], text'[2], and text'[3]. After examining text[9], we are in state $\xi_{\{1\}}^{\{1,2,3,4\}}$ with $\eta = 9$.

More precisely,

$$\mathrm{M} = \{\ \mathrm{j} \mid \Delta[\eta + \mathrm{j} - 1] = \diamond \quad \text{and} \quad 1 \leq \mathrm{j} \leq \mathrm{P}\},$$

and,

$$\mathrm{E} = \{\ \mathrm{j} \mid \mathrm{text}[\eta + \mathrm{j} - 1] = \text{examined} \quad \text{and} \quad 1 \leq \mathrm{j} \leq \mathrm{P}\}.$$

The starting state is $\xi_{\emptyset} = \xi_{\{\}}^{\{1,\cdots,P\}}$. For the remainder of this paper, $\mathrm{M} = \{m_1 = 1, m_2, \cdots, m_{\|M\|}\}$, and E $= \{e_1, e_2, \cdots, e_{\|E\|}\}$. If the rightmost character in E, $e_{\|E\|} = \gamma$, then $\mathrm{M} = \{m_1, m_2, \cdots, m_i, \gamma{+}1, \gamma{+}2, \cdots, P\}$, because the potential matches starting at $\gamma + 1, \gamma + 2, \cdots, P$ have not been looked at yet. We abbreviate the above state either as $\xi_{\{E\}}^{\{m_1,\cdots,m_i,*\}}$, with the * representing the unexamined patterns, or simply as $\xi_{\{E\}}^{\{m_1,\cdots,m_i\}}$, with the * implied.

The probability of being in state ξ is $\Pr(\xi)$. The set of all possible states is Ξ. Finally, $\xi(\Delta)$ is the *expected number of decisions* to be made in state ξ when the next text character is examined. For RQ determining $\xi(\Delta)$ is trivial but determining $\Pr(\xi)$ is difficult. For BM, neither is trivial.

A *transition* occurs from one state to a second (possibly the same) state for every character examined. For $\xi_1, \xi_2 \in \Xi$, and $c \in \Pi$, if a transition is made from ξ_1 to ξ_2 upon finding c, then the corresponding entry in the *transition table* T, is $\mathrm{T}[\xi_1][c] = \xi_2$. The probability of moving from state ξ_1 to ξ_2 is

$$\Pr(\xi_1 \rightsquigarrow \xi_2) = \sum_{\substack{c \in \Pi \\ T[\xi_1][c] = \xi_2}} \mathrm{pr}_c.$$

The *probabilistic transition table* is gives the probability of moving from ξ_i to ξ_j. We write the probabilistic transition table as an NxN matrix, B, with entries $b_{ij} = \Pr(\xi_i \rightsquigarrow \xi_j)$, where $\mathrm{N} = \|\Xi\|$ = the number of possible

states. The 1xN vector V_τ gives the probability distribution for each state after examining τ characters. V_0 = [1,0,0,...,0] because we are in state $\xi_\emptyset$ initially. We have $V_\tau = V_0 B^\tau$, because $V_\tau = V_{\tau-1}B$. To analyze the finite automaton, Γ, we need to find its probabilistic steady state, $\overline{V}$, which satisfies $V_\tau = V_{\tau-1}B$. Thus, $\overline{V}$ is an eigenvector of B with eigenvalue 1. Because every distinct eigenvalue has an eigenvector, $\overline{V}$ exists if and only if B has an eigenvalue of one.

Lemma 1 *Matrix B has an eigenvalue of one.*

Proof. The sum of each column of B is $\sum_{j=1}^{N} b_{ij} = \sum_{j=1}^{N} \Pr(\xi_i \rightsquigarrow \xi_j)$ which is the probability of moving from ξ_i to any state $\xi_j \in \Xi$ which is 1 by definition. If λ an eigenvector, then $\det(B - \lambda I) = 0$, where I is the NxN identity matrix. Consider B-λI when $\lambda = 1$. The sum of each column is 0. Adding all other rows to the last row yields a new row of all zeroes, proving the lemma. □

Analyzing $\overline{V}$ gives the average behavior for long input text strings, because $\overline{V}$ represents the probabilistic steady state of Γ. We surmise that the steady state is reached quickly, and that analyzing $\overline{V}$ gives an accurate approximation to RQ and BM as long as $T \geq 10P\lceil \log_\alpha P \rceil$. The factor of 10 discounts the effects of starting up.

The n^{th} entry in $\overline{V}$ is $\Pr(\xi_n)$. Let $\overline{d}$ be the average number of decisions made per probe. Summing over all states gives $\overline{d} = \sum_{\xi \in \Xi} \Pr(\xi)\xi(\Delta) / \sum_{\xi \in \Xi} \Pr(\xi)$. The average number of characters examined is $T/\overline{d}$ or

$$\overline{\Phi(A)} = T \frac{\sum_{\xi \in \Xi} \Pr(\xi)}{\sum_{\xi \in \Xi} \Pr(\xi)\,\xi(\Delta)} \qquad \text{as } T \to \infty. \tag{2}$$

In summary, our analysis of $\overline{\Phi(A)}$ consists of the following steps.

Assumption: A random text string, text[ ].
Input: Algorithm A, alphabet Π, character probabilities pr_c, and patt[ ].
Output: The asymptotic number of probes made by A, as $T \to \infty$.

1. Calculate $q = \overline{\mathrm{patt}[\,]} \sum \mathrm{pr}_c{}^2$.
2. For an algorithm A, create the transition table for all states ξ.
3. Create the probabilistic transition table and hence, B.
4. Determine $\xi(\Delta)$ for all $\xi \in \Xi$.
5. Determine $\Pr(\xi)$ for all $\xi \in \Xi$.
6. Evaluate Equation 2.

In practice, RQ has greater than $O(2^P)$ states, which is far too many for exact analysis. Instead, we derive a simpler approximation for $\overline{\Phi(RQ)}$ by counting only the most probable states.

Furthermore, we assume $P < \alpha$, which greatly reduces the number of probable states. Examining a text character will eliminate all or all-but-one match in M on the average. The most probable state will be $\xi_\emptyset$. In general, $\Pr(\xi_{\{E\}}^{\{M',*\}})$ is $O(q^{\|E\| \cdot \|M'\|})$, because $\|E\|$ text characters have each matched $\|M'\|$ pattern

positions so far. In deriving our approximations, we restrict our interest to states with probabilities greater than or equal to some threshold, p_{th}, and calculate probabilities to the same accuracy. For the RQ algorithm we use $p_{th} = O(q^2)$. For BM, we use $p_{th} = O(1)$. For the OPT2 algorithm, the analysis is simple enough to do exactly.

5.2 The OPT2 algorithm

We now derive the leading term given previously for $\overline{\Phi(OPT2)}$. OPT2 is the optimal algorithm for searching for patterns of length two. The OPT2 algorithm [KMP77] is

```
k := 2;                                                    (* ξ_∅ *)
while k ≤ T do begin
   c:= text[k];
   if c = patt[2] then                                     (* ξ^{1?}_{2} *)
      if text[k-1] = patt[1] then
         match found at (k-2)
   while c = patt[1] do begin
      k := k + 1;                                          (* ξ^{12}_{1} *)
      c := text[k];
      if c = patt[2] then
         match found at (k-2);
   end;
   k := k+2;                                               (* ξ_∅ *)
end;.
```

OPT2 is simply RQ when P= 2. For this simple case, let q_1 be the probability that patt[1] matches a random text character; define q_2 similarly. In other words, $q_i = \sum_{c \sim patt[i]} \mathrm{pr}_c$.

Our model of OPT2 uses four states. Let c_0 be the character just examined, and c_1 be the next character to be examined.

- $\xi_\emptyset$: no pending matches, as $c_0 \neq$ patt[1] and $c_0 \neq$ patt[2]. $\xi_\emptyset$ is also the starting state. We could end up in any of the four states after probing c_1.
- $\xi^{\{1\}}_{\{2\}}$: $c_0 \sim$ patt[2] but $c_0 \neq$ patt[1]. c_1 is the text character to the left of c_0, which will force a decision at 1 (either a complete match or a failure). In either case, the next state is $\xi_\emptyset$.
- $\xi^{\{1,*\}}_{\{1\}}$: $c_0 \sim$ patt[1] but $c_0 \neq$ patt[2], giving state $\xi^{\{2\}}_{\{2\}}$. After shifting text$'$ to the right by one, we get the desired state $\xi^{\{1,*\}}_{\{1\}}$. c_1 is the text character to the right of c_0, which also forces a decision at 1. The next state is $\xi_\emptyset$ if $c_1 \neq$ patt[1] and $\xi^{\{1,*\}}_{\{1\}}$ if $c_1 =$ patt[2].
- $\xi^{\{1,2\}}_{\{2\}}$: $c_0 \sim$ patt[1] and $c_0 \sim$ patt[2]. c_1 is the character to the left of c_0, which will decide at 1. The decision at 2 is unaffected, causing the next state to be $\xi^{\{1,*\}}_{\{1\}}$.

The probabilistic transition table for OPT2 is

$$
\begin{aligned}
\xi_{\emptyset} &\rightarrow \bar{q}_1\bar{q}_2\xi_{\emptyset} + \bar{q}_1 q_2 \xi_{\{2\}}^{\{1\}} + \bar{q}_2 q_1 \xi_{\{1\}}^{\{1,*\}} + q_1 q_2 \xi_{\{2\}}^{\{1,2\}} \\
\xi_{\{2\}}^{\{1\}} &\rightarrow \xi_{\emptyset} \\
\xi_{\{1\}}^{\{1,*\}} &\rightarrow \bar{q}_2\xi_{\emptyset} + q_2\xi_{\{1\}}^{\{1,*\}} \\
\xi_{\{2\}}^{\{1,2\}} &\rightarrow \xi_{\{1\}}^{\{1,*\}}.
\end{aligned}
$$

For simplicity, let $\Pr(\xi_{\emptyset}) = 1$, because Equation 2 ignores constant factors in $\Pr(\xi)$. Rearranging terms to collect the probabilities for each state yields

$$
\begin{aligned}
\Pr(\xi_{\emptyset}) &= 1 \\
\Pr(\xi_{\{2\}}^{\{1\}}) &= \bar{q}_1 q_2 \\
\Pr(\xi_{\{1\}}^{\{1\}}) &= \bar{q}_2 q_1 + q_1 \Pr(\xi_{\{1\}}^{\{1,*\}}) + \Pr(\xi_{\{2\}}^{\{1,2\}}) \\
\Pr(\xi_{\{2\}}^{\{1,2\}}) &= q_1 q_2.
\end{aligned}
$$

Thus, $\Pr(\xi)$ and $\xi(\Delta)$ for each state are

State	$\Pr(\xi)$	$\xi(\Delta)$
$\xi_{\emptyset}$	1	$\bar{q}_1 + \bar{q}_2$
$\xi_{\{2\}}^{\{1\}}$	$\bar{q}_1 q_2$	1
$\xi_{\{1\}}^{\{1,*\}}$	$q_1/\bar{q}_1$	$1 + \bar{q}_1$
$\xi_{\{2\}}^{\{1,2\}}$	$q_1 q_2$	1.

Applying Equation 2 gives

$$
\overline{\Phi(OPT2)} = \frac{T}{2}\frac{(1 + q_2 - q_1 q_2)}{(1 - \frac{q_1}{2})}. \tag{3}
$$

Setting $q = q_1 = q_2$ gives the leading term in Equation 1.

5.3 A simple BM approximation

We give a very simple asymptotic approximation for the Boyer-Moore algorithm when $P \leq \alpha$. For short patterns, BM achieves its speed through the use of a "skip table", Skip[x] (called δ_1 in the original article), which contains the distance from the rightmost occurrence of character x in patt[ ] to the end of the pattern. If c is not contained in the pattern, Skip[c] = P. Let c = text'[e] be the last character examined. The amount to increment η so that c aligns to its rightmost occurrence in the pattern is η_{sh}= Skip[c] – (P– e). The number of decisions made is η_{sh}. In the common case, c = text'[P], and η_{sh}= Skip[c]. As an example, assume we are searching for aabc in abcdabcaace.

```
index    1  2  3  4  5  6  7  8  9  10 11
text     a  b  c  d  a  b  c  a  a  c  e
                    ↓
pattern  a  a  b  c                        Skip[d] = 4, η_sh= 4

text     a  b  c  d  a  b  c  a  a  c  e
                             ↓
pattern           a  a  b  c               Skip[a] = 2, η_sh= 2

text     a  b  c  d  a  b  c  a  a  c  e
                                   ↓
pattern              a  a  b  c           Skip[c] = 0  (partial match)

text     a  b  c  d  a  b  c  a  a  c  e
                             ↓
pattern              a  a  b  c           Skip[a] = 2, η_sh= 1

text     a  b  c  d  a  b  c  a  a  c  e
                                      ↓
pattern                 a  a  b  c        Skip[d] = 4, η_sh= 4
```

After aligning itself, BM examines the rightmost character in text'. If a pattern and text character match, BM continues the comparison backwards through the pattern.

Our analysis here only considers the start state, $\xi_\emptyset$, and the result of probing text'[P] = c. When Skip[c] = i for $1 \leq i \leq P-1$, none of the i characters at the end of the pattern are c, but the $i+1^{th}$ character from the end is c. Thus, the probability that Skip[c] = i is $\overline{q}^i q$. The probability that Skip[c] = P is $\overline{q}^P$, as the pattern does not contain c. Finally, c matches patt[P] with probability q, which we view as Skip[c] = 0. The expected number of decisions is the weighted sum of the different shifts: $\sum_{i=0}^{P} i \Pr(Skip[c] = i)$.

Thus, $\xi_\emptyset(\Delta) =$

$$0\,q + \sum_{i=1}^{P-1} i\,\overline{q}^i q + P\,\overline{q}^P$$

As before, setting $\Pr(\xi_\emptyset) = 1$, means that $\overline{\Phi(BM)}$ is $T/\xi_\emptyset(\Delta)$. Simplifying the above equations for $\xi_\emptyset(\Delta)$ gives

$$\overline{\Phi(BM)} \approx T\frac{q}{\overline{q}(1-\overline{q}^P)} \qquad \text{when P} \leq \alpha. \tag{4}$$

(A detailed analysis of BM reveals that this model is too simple, but the approximation remains valid.)

5.4 The RQ algorithm

An exact analysis of RQ would have to consider $O(2^{2P})$ states and simultaneous equations. Instead, we derive an asymptotic approximation to $\overline{\Phi(RQ)}$ by considering only states with probabilities on the order of q^2 or greater, which gives us $O(P^2)$ states. Setting $p_{th} = q^3$ would involve $O(P^3)$ equations and would make the following calculation much more difficult. As before we set $\Pr(\xi_\emptyset) = 1$.

We use $\sum$-notation to represent a collection of states in the obvious way. For example, $\xi_\emptyset \rightarrow q\sum_{i=1}^{P-1} \xi_{\{i\}}^{\{1\}}$ means when in state $\xi_\emptyset$ we go to each of the states $\xi_{\{1\}}^{\{1,*\}}$, $\xi_{\{2\}}^{\{1\}} \ldots \xi_{\{P-1\}}^{\{1\}}$ with probability q. The probabilistic transition table for RQ is

$$\xi_\emptyset \rightarrow \bar{q}^P \xi_\emptyset + \bar{q}^{P-1} q \left(\sum_{i=1}^{P-1} \xi_{\{i\}}^{\{1\}}\right) + \bar{q}^{P-2} q^2 \left(\sum_{e=2}^{P} \sum_{m=2}^{e} \xi_{\{e\}}^{\{1,m,*\}}\right) + O(q^3)\text{terms} \quad (5)$$

states with $\Pr(\xi) = O(q)$

$$\xi_{\{e\}}^{\{1,*\}} \rightarrow q\xi_{\{e,p\}}^{\{1\}} + \bar{q}^{P-e-1} q \left(\sum_{j=1}^{P-e} \xi_{\{j\}}^{\{1,*\}}\right) + O(q^2) \text{ terms}$$

$$\forall\, e : (1 \le e \le P-1)$$

$$\xi_{\{P\}}^{\{1\}} \rightarrow \bar{q}\xi_\emptyset + q\xi_{\{P-1,P\}}^{\{1\}} \quad (6)$$

states with $\Pr(\xi) = O(q^2)$

$$\xi_{\{e\}}^{\{1,m,*\}} \rightarrow \bar{q}^2 \xi_\emptyset + O(q) \text{ terms}$$

$$\forall\, m, e : (2 \le m \le e \le P), \text{excluding } e = m = P$$

$$\xi_{\{P\}}^{\{1,P\}} \rightarrow \bar{q}\xi_{\{1\}}^{\{1,*\}} + O(q) \text{ terms.} \quad (7)$$

The first term in Equation 5 corresponds to deciding all indices in text′. In other words, c = text′[P] is a character not contained anywhere in the pattern. The second term occurs when all but one decision is made — c occurs only once in the pattern. The third term occurs when all but two decisions are made — c occurs twice. We rearrange the terms to collect the probabilities, yielding the following table.

$\approx \Pr()$	*State*	$Pr(\xi)$
1	$\Pr(\xi_\emptyset)$	1
q	$\Pr(\xi_{\{e\}}^{\{1,*\}})$	$\bar{q}^{P-1} q \Pr(\xi_\emptyset) + \sum_{j=1}^{P-e} \bar{q}^{P-j-1} q \Pr(\xi_{\{j\}}^{\{1,*\}}), \quad \forall e : 2 \le e \le P$
q	$\Pr(\xi_{\{1\}}^{\{1,*\}})$	$\bar{q}^{P-1} q \Pr(\xi_\emptyset) + \sum_{j=1}^{P-1} \bar{q}^{P-j-1} q \Pr(\xi_{\{j\}}^{\{1,*\}}) + \bar{q} \Pr(\xi_{\{P\}}^{\{1,P\}})$ (8)
q^2	$\Pr(\xi_{\{e\}}^{\{1,m\}})$	$\bar{q}^{P-2} q^2 \Pr(\xi_\emptyset), \quad \forall m, e : 2 \le m \le e \le P$
q^2	$\Pr(\xi_{\{e,p\}}^{\{1\}})$	$q \Pr(\xi_{\{e\}}^{\{1\}}), \quad \forall e : 1 \le e \le P-1$
$2q^2$	$\Pr(\xi_{\{1\}}^{\{P-1,P\}})$	$q \Pr(\xi_{\{P-1\}}^{\{1\}}) + q \Pr(\xi_{\{P\}}^{\{1\}})$. (9)

In the leftmost column, we list the approximate probabilities of each state. In rightmost column, we use the approximate probabilities, which simplifies the calculation and results in an error on the order of $O(q^3)$. The last term in Equation 8 is due to Equation 7; similarly the last term in Equation 9 is due to Equation 6. The expected number of decisions in state ξ, $\xi(\Delta)$, is simply $\bar{q}$ times the number of indices looked at by the next probe.

State	Pr(ξ)	$\xi(\Delta)$	condition
$\xi_{\emptyset}$	1	$P\bar{q}$	
$\xi_{\{e\}}^{\{1,*\}}$	$\bar{q}^{P-1}q + (P-e)q^2$	$(P-e+1)\bar{q}$	$2 \le e \le P$
$\xi_{\{1\}}^{\{1,*\}}$	$\bar{q}^{P-1}q + Pq^2$	$P\bar{q}$	
$\xi_{\{e\}}^{\{1,m,*\}}$	q^2	$(P-e+2)\bar{q}$	$2 \le e \le m \le P$
$\xi_{\{P\}}^{\{1,P\}}$	q^2	$\bar{q}$	
$\xi_{\{j,p\}}^{\{1\}}$	q^2	$\bar{q}$	$1 \le j \le P-1$
$\xi_{\{1\}}^{\{P-1,P\}}$	$2q^2$	$\bar{q}$	

Using $\bar{q}^k q = q - kq^2 + O(q^3)$ and applying Equation 2, we get

$$\overline{\Phi(RQ)} \approx \frac{T}{P} \frac{1 + Pq + (P+1)q^2}{(1 + \frac{P-1}{2}q - \frac{P^2-5P+2}{2P}q^2)} \qquad \text{when P} \le \alpha. \tag{10}$$

5.5 The number of probes for large P

In this section, we derive an upper bound on the worst case running time of RQ, for arbitrarily large P. We do so by examining blocks of b characters at a time, increasing the effective blocked alphabet size to α^b, as suggested in [KMP77] and [Yao79]. Call this algorithm the blocking version of RQ. We show that in the worse case, the number of characters examined is asymptotic to $(4\log_\alpha P/3)(T/P)$.

For text$'$ = text[η...η+P-1] we probe the characters text$'$[P-b+1...P]. Choosing b = $\lceil \log_\alpha P \rceil$ assures that $\alpha' = \alpha^b >$ P, so that we can apply Equation 10. There are $P - b$ possible indices in M, namely [1...P-b+1]. For large P, $P - b \approx P$.

For our lower bound, we let $P \to \infty$, throw out terms of order $O(1/P)$ or less, and set b = $\log_\alpha P$, which gives $Pq = P/\alpha^b = 1$. Applying Equation 10, gives the desired upper bound to the average case running time,

$$\overline{\Phi(RQ)} \le \frac{4\log_\alpha P}{3}\left(\frac{T}{P}\right) \qquad \text{for } P \gg \alpha,\ T \to \infty. \tag{11}$$

Examining blocks of characters is suboptimal. In many instances, after examining n characters where $n < b$, we will have decided at 1, so that examining the rest of the block is suboptimal, as will be shown in Section 6. However, Yao [Yao79] showed that RQ is better by at most a constant factor than the blocking version of RQ. We surmise that the upper bound of Equation 11 is close (within a factor of 2) to the actual number of characters examined.

6 Proof of Optimality of RQ

The RQ algorithm is optimal in several simpler cases lending credence that it is optimal on the average for all patterns.

- When P= 2, RQ is OPT2.
- When the text consists entirely of characters not in pattern, RQ examines every P^{th} character, thus $\overline{\Phi(RQ)} = \lfloor \frac{T}{P} \rfloor$ characters.
- When the text consists entirely of successive patterns, RQ examines exactly T characters.

Our proof that RQ is optimal consists of determining the best sequence of probes for any given state ξ. We model a search algorithm as a finite automaton with an associated state, (ξ, η, Δ). First, we define how to compare two different probes. Second, we compare probing index i versus index j, and show that when $i > j$ probing i is always better. Next, we compare sequences of one probe followed by another. We show that for $i > j$, probing i first and then j if necessary is better than j first and i second. It follows that RQ is the optimal monotonic search algorithm. Finally, we show that a non-monotonic algorithm can do no better than RQ, proving that RQ is optimal.

6.1 Definitions

As before η = the smallest undecided index, and text′ = text$[\eta ... \eta + P - 1]$. The process of examining or *probing* the character at text′[i] is denoted $\phi[i]$. If $\phi[i]$ might make a decision at x, then $\phi[i]$ *looks at* x. We assume that a probe makes a decision with probability $\overline{q}$ at all undecided indices looked at in text′. Note that applying $\phi[i]$ to ξ_E^M cannot make decisions at the indices in M greater than i. For example, if P= 6 and the current state is $\xi_{\{3,6\}}^{\{1,2,5,6\}}$, $\phi[5]$ might make decisions at 1, 2, or 5, but it cannot make a decision at 6. $\phi[0]$ is the null probe, which corresponds to do nothing. The probe sequence $\phi[i]\phi[j]$ probes i first and then probes j upon the resulting state.

In this section, i and j represent unexamined indices, with $i > j$. In all cases, we shall be proving that $\phi[i]$ is better than $\phi[j]$. The capital letters I, J, I′, J′ denote sets of indices in text′.

For state ξ_E^M, we use the notation E = $\{e_1, e_2, \cdots, e_{\|E\|}\}$= the set of examined indices. $\bar{E}$ = the set of *unexamined indices* of text′; thus $E \cup \bar{E} = \{1, 2, \cdots, P\}$. If we have examined every character but one in text′, then probing that character must make a decision at 1 (either a complete match or a mismatch).

The *two-step algorithm* $\phi[i_1, i_2]$ means to examine i_1 and then to examine i_2 if it is still necessary to decide at 1. $\phi[i_1]$ might decide at 1, rendering $\phi[i_2]$ unnecessary. The n-step algorithm $\phi[i_1, i_2, \cdots, i_n]$ means to examine i_1, and then i_2 if necessary, and then i_3 if still necessary, and so on, down to i_n.

For each possible state ξ_E^M, a monotonic search algorithm specifies a sequence of probes $\phi[i_1, i_2, \cdots, i_n]$, where $i_1, \cdots, i_n$ is some permutation of $\bar{E}$. Thus, $\phi[i_1, i_2, \cdots, i_n]$ specifies what order to probe $\bar{E}$ until a decision is made at 1. If text′[ ] $\sim$ patt[ ], all the indices in $\bar{E}$ will be probed.

We model the actions of the monotonic search algorithm A1 by a finite automaton Γ_{A1}. The current state of Γ_{A1} is ξ_{A1}. The current value of η for A1 is η_{A1}. The set of indices of text (not text′) where decisions have been made by A1 is Δ_{A1}. The current state of A1 is denoted by the *total state*, $(\xi, \eta, \Delta)_{A1}$. Algorithms A1 and A2 are in the same state if $(\xi, \eta, \Delta)_{A1} = (\xi, \eta, \Delta)_{A2}$. For brevity, we say algorithm A1

is in state ξ_1 when Γ_{A1} is in state ξ_1. Intuitively, if algorithm A1 is better than algorithm A2, then *on the average* A1 increments η (and hence, shifts text' to the right of text[]) faster than a A2 does.

Definition 1 *Let $(\xi, \eta, \Delta)_{A1}$ and $(\xi, \eta, \Delta)_{A2}$ be the states of two algorithms. If $\Delta_{A1} \supset \Delta_{A2}$ and $\eta_{A1} > \eta_{A2}$, then $(\xi, \eta, \Delta)_{A1}$ is better than $(\xi, \eta, \Delta)_{A2}$, or $(\xi, \eta, \Delta)_{A1} > (\xi, \eta, \Delta)_{A2}$.*

Definition 2 *Let A1 and A2 be monotonic search algorithms starting in the same state, $(\xi, \eta, \Delta)_0$, with C = a fixed positive constant. If on the average, after n probes, for all $n \geq C$, $(\xi, \eta, \Delta)_{A1} > (\xi, \eta, \Delta)_{A2}$, then A1 is a better algorithm than A2.*

The result of applying probe $\phi[i]$ to total state (ξ, η, Δ) is shown as $\phi[i](\xi, \eta, \Delta) = (\xi_i, \eta_i, \Delta_i) = (\xi, \eta, \Delta)_i$. $\phi[i](\xi_E^M) = \xi_i$ is short for $\phi[i](\xi, \eta, \Delta)_0 = (\xi, \eta, \Delta)_i$, where Δ_i and η_i are understood. For $\phi[i](\xi_E^M)$, $\phi[i]$(M) denotes the resulting set of undecided indices after applying $\phi[i]$. Similarly, $\phi[i]$(E) = E $\cup$ {i}, because probing i adds i to E.

Applying $\phi[i]$ to ξ_E^M increments η by η_{sh}, shifting text' to the right by η_{sh}. Shifting M and E by η_{sh} corresponds to subtracting η_{sh} from all existing indices and discarding those that are less than 1. The discarded indices are the indices shifted out to the left of text'. In addition, the new, not-yet-looked-at indices $\{P - \eta_{sh} + 1, \cdots, P\}$ are introduced into M.

A *transformation* Tr: maps a probe $\phi[z]$ to another (possibly the same) probe $\phi[z']$ denoted by Tr:$\phi[z] = \phi[z']$. There are two types of transformations. The *exchange transformation*, Tr$[x \leftrightarrow y]$: translates $\phi[x]$ to $\phi[y]$ and $\phi[y]$ to $\phi[x]$; all other probes are mapped unchanged. The *nullify transformation* Tr$[x \rightarrow \emptyset]$: maps probe $\phi[x]$ into "nothing" or the null probe $\phi[0]$, with all other probes mapped unchanged. That is, for a probe z,

$$\text{Tr}[x \leftrightarrow y]{:}\phi[z] = \begin{cases} \phi[z] & \phi[z] \neq \phi[x], x \neq \phi[y] \\ \phi[x] & \phi[z] = \phi[y] \\ \phi[y] & \phi[z] = \phi[x] \end{cases}$$

and

$$\text{Tr}[x \rightarrow \emptyset]{:}\phi[z] = \begin{cases} \phi[z] & \phi[z] \neq \phi[x] \\ \phi[0] & \phi[z] = \phi[x] \end{cases}$$

Applying a transformation Tr: to either a sequence of probes or an n-step algorithm applies to the transformation to every probe individually; Tr:$(\phi[x1], \phi[x2], \cdots, \phi[xn])$ gives Tr:$\phi[x1]$, Tr:$\phi[x2], \cdots$, Tr:$\phi[xn]$. The null probe is dropped from a sequence of probes. For example, Tr$[3 \leftrightarrow 4]{:}\phi[2,4,6,3] = \phi[2,3,6,4]$, and Tr$[4 \rightarrow \emptyset]$: $\phi[2,4,6,3] = \phi[2,6,3]$.

6.2 Preliminary Concepts

Lemma 2 *Let A1 and A2 be monotonic search algorithms. Both A1 and A2 start in state $(\xi, \eta, \Delta)_{start}$ and end up in state $(\xi, \eta, \Delta)_{end}$. A1 requires a1 probes; A2 requires a2 probes. If a1 < a2, then A1 is a better algorithm than A2.*

Proof. Let $d = a2 - a1$, the difference in the number of probes executed by the two algorithms. On the average, applying d more probes as prescribed by A1 to $(\xi, \eta, \Delta)_{end}$ increments η_{end} and adds decisions to Δ_{end}. Thus, after $a2$ probes, A1 is a better algorithm than A2. □

In our model, we extract all possible information from each probe, in that we make every decisions possible. Clearly, an algorithm that does use all the information from a probe cannot be optimal. For example, the Boyer-Moore algorithm is non-optimal in this respect. In practice, the set of states for non-optimal algorithms is a subset of the set of all states Ξ.

We are concerned with choosing the right sequence of probes when in state ξ_E^M, when there is more than one choice. If $\bar{E}$ has only one unexamined index, k, from the definition of a monotonic algorithm, we must probe k next. For the remainder of this section, we assume that $\bar{E}$ contains at least two indices.

In comparing states ξ_1 and ξ_2, we are really comparing total states $(\xi,\eta,\Delta)_1$ and $(\xi,\eta,\Delta)_2$. When comparing probes, it is assumed that we start out in the same total state, $(\xi,\eta,\Delta)_0$. We already know how to compare total states and algorithms. The following definition allows us to compare single probes, $\phi[k]$ and $\phi[k']$. In essence, if $\phi[k]$ results in a better state, and if we can maintain $(\xi,\eta,\Delta)_k > (\xi,\eta,\Delta)_{k'}$, then $\phi[k] > \phi[k']$.

Definition 3 *Given state $(\xi,\eta,\Delta)_0$ with unexamined indices k and k', let $\phi[k](\xi,\eta,\Delta)_0 = (\xi,\eta,\Delta)_k$ and $\phi[k'](\xi,\eta,\Delta)_0 = (\xi,\eta,\Delta)_{k'}$. If for any subsequent series of n probes $\phi[K']$ applied to $(\xi,\eta,\Delta)_{k'}$, we can find another series of n probes, such that $\phi[K](\xi,\eta,\Delta)_k > \phi[K'](\xi,\eta,\Delta)_{k'}$, then $\phi[k] > \phi[k']$. Similarly, if $\phi[K](\xi,\eta,\Delta)_k = \phi[K'](\xi,\eta,\Delta)_{k'}$, then $\phi[k] = \phi[k']$.*

Condition 1 In comparing two probes we assume the following conditions.

1. *a state $(\xi,\eta,\Delta)_0$,*
2. *$\bar{E}$ has at least two unexamined indices, i and j, with $i > j$, and*
3. *applying $\phi[i]$ and $\phi[j]$ to $(\xi,\eta,\Delta)_0$ gives resulting states (ξ_i,η_i,Δ_i) and (ξ_j,η_j,Δ_j), respectively. In particular, $\phi[i](\xi_E^M) = \xi_{E_i}^{M_i} = \xi_i$, and $\phi[j](\xi_E^M) = \xi_{E_j}^{M_j} = \xi_j$.*

To analyze this situation, we divide the set of undecided indices M into three regions: M_I = indices looked at by both $\phi[i]$ and $\phi[j]$; M_{II} = the indices looked at only by i, and M_{III} = indices looked at by neither i nor j. Note that M_{II} may be empty, if we have previously decided these indices. Similarly, we divide the set of probed indices, E, into three regions: E_I = indices to the left of j, E_{II} = indices to the right of j up to i, and E_{III} = indices to the right of i. Note that i and j are elements of $\bar{E}$, not E.

$$M = \{\underbrace{M_I}_{I}, \underbrace{M_{II}}_{II}, \underbrace{M_{III}}_{III}\}$$
$$E = \{\overbrace{E_I, j}^{I}, \overbrace{E_{II}, i}^{II}, \overbrace{E_{III}}^{III}\}$$

The following diagram illustrates these ideas for $\xi_{\{3,6\}}^{\{1,2,4,6\}}$, $i = 5$, $j = 2$.

$$\xi \begin{cases} \{M_I = 1,2 & M_{II} = 4 & M_{III} = 6\} \\ \{E_I = \boxed{\text{j}} & E_{II} = 3\,\boxed{\text{i}} & E_{III} = 6\} \end{cases}$$

6.3 A single probe

We now compare two single probes subject to Condition 1. The resulting theorem is the heart of our proof. We expand the resulting states ξ_i and ξ_j probabilistically and compare them term by term, showing that ξ_i is equally good in some cases, and better in the other cases. We break down our argument based on the possible initial condition of M_{II}.

Probing a state ξ gives a probabilistic distribution of resulting states. We break down the distribution by the amount of resultant shift, η_{sh}. Remember that M = $\{m_1 = 1, m_2, \cdots, m_{\|M\|}\}$. If we make decisions at the first n indices in M, then η_{sh}= $m_{n+1} - 1$, because we shift text′ so that m_{n+1} is the first character in the new text′. The resulting set of states when $\eta_{sh} = m_{n+1} - 1$ is denoted $[\xi]_n$. $[\xi]_n$ contains all 2^x permutations for the remaining x indices of M beyond m_{n+1} that were looked at. Pr($[\xi]_n$) is the sum of the probabilities of these states. For example, P= 5, M = {1,2,4,5}, E = {3} and we probe 5. Before shifting text′, the states for $[\xi]_2$ are $\{\xi^{\{4,5\}}_{\{3,5\}}, \xi^{\{4\}}_{\{3,5\}}\}$; after shifting we get $[\xi]_2 = \{\xi^{\{1,2\}}_{\{2\}}, \xi^{\{1\}}_{\{2\}}\}$. All the states in a given $[\xi]_n$ have the same E.

The minimum shift is 0 when no decision is made at 1. After $\phi[k]$, Pr($[\xi]_0$) = q, because text′[k] $\sim$ patt[k]. In general, Pr($[\xi]_n$) = $\overline{q}^n q$, because we make decisions at the first n elements of M and match on the $(n+1)^{th}$ element.

From a probabilistic standpoint, $\phi[i]$ and $\phi[j]$ act upon M_I equally, and we can directly compare the terms between $\phi[i](M_I)$ and $\phi[j](M_I)$. Conceptually, we expand out the result of $\phi[i](M_I)$ into every possible resulting state multiplied by its probability. For every term from $\phi[i]$ there is an identical term from $\phi[j]$. We only compare identical terms. In other words, when we compare $\phi[i]$ and $\phi[j]$, we assume they *make exactly the same decisions* in M_I. Because both probes do not affect M_{III}, $\phi[i]$ differs from $\phi[j]$ only at M_{II}.

M′ is the new set of undecided indices of text′ after the probing and shifting. We partition M′ into three regions similarly to M. $\phi[i](M_I) = \phi[j](M_I) = M'_I$. M_{III} is unaffected, thus M'_{III} is M_{III} shifted by η_{sh}. M'_{new} are the new indices shifted in.

The next lemma shows that when M_{II} is previously decided, $\phi[i] = \phi[j]$.

Lemma 3 *"Equal case" If $\phi[i]$ and $\phi[j]$ are two probes satisfying Condition 1, with $M_{II} = \emptyset$ (the empty set) in state ξ^M_E, then $\phi[i]$ and $\phi[j]$ are equally good probes.*

Proof. We break down ξ_i and ξ_j by the amount of shift. Let $n_I = \|M_I\|$.

$$\xi_i = q[\xi]_0 + \sum_{n=2}^{n_I-1} \overline{q}^n q[\xi]_n + \overline{q}^{n_I} q[\xi]_{n_I}. \tag{12}$$

$$\xi_j = q[\xi]_0 + \sum_{n=2}^{n_I-1} \overline{q}^n q[\xi]_n + \overline{q}^{n_I} q[\xi]_{n_I}. \tag{13}$$

The equations are identical. Thus, the resulting states, ξ_i and ξ_j, are identical except that $E_i = E \cup i$, and $E_j = E \cup j$. The last term in both equations represents the case where all indices in M_I are decided by the probe. Comparing the corresponding terms of Equations 12 and 13 gives the following cases.

- If part of M_I remains undecided, we get either the first term or one of the summed terms in the above equations, with i' = i - η_{sh}, and j' = j - η_{sh}. The resulting states look as follows.

$$\phi[i](\xi) \rightarrow \xi \begin{matrix} \{M'_I & & M'_{III} & M'_{\text{new}}\} \\ \{E'_I & E'_{II}\boxed{i'} & E'_{III} & \} \end{matrix}$$

$$\phi[j](\xi) \rightarrow \xi \begin{matrix} \{M'_I & & M'_{III} & M'_{\text{new}}\} \\ \{E'_I\boxed{j'} & E'_{II} & E'_{III} & \} \end{matrix}$$

For any subsequent series of probes $\phi[J]$ applied to ξ_j, let $\phi[I] = \text{Tr}[i \leftrightarrow j]{:}\phi[J]$. Every decision that $\phi[J]$ makes is also made by $\phi[I]$, and vice versa. The sequence of probes $\phi[I]$ and $\phi[J]$ will eventually decide all of M_I causing a shift past both i and j, resulting in identical states. (Because M_{II} was previously decided, when the last entry in M'_I is decided we shift past both i and j.) In this case, $\phi[i]$ and $\phi[j]$ have lead to the same future state, shown below as Scenario 14, so that $\phi[i] = \phi[j]$.

- If $\phi[i]$ and $\phi[j]$ decide all of M_I, we get the last term in Equations 12 and 13. Because M_{II} is empty, we shift past M_I, j, and i to M_{III}.

$$\phi[i](\xi) \rightarrow \xi \begin{matrix} \{M'_{III} & M'_{\text{new}}\} \\ \{E'_{III} & \} \end{matrix} \qquad (14)$$

$$\phi[j](\xi) \rightarrow \xi \begin{matrix} \{M'_{III} & M'_{\text{new}}\} \\ \{E'_{III} & \} \end{matrix}$$

Clearly, $\xi_i = \xi_j$, so that $\phi[i] = \phi[j]$.

In all cases when M_{II} is previously decided, we obtain identical future states, so that $\phi[i] = \phi[j]$. □

We now consider the case, where M_{II} is not completely decided when we apply $\phi[i]$ and $\phi[j]$. Intuitively, $\phi[i]$ is better here than $\phi[j]$, because $\phi[i]$ might make decisions in M_{II}, but $\phi[j]$ cannot. To simplify the proof, we show that in both the worst case and the best case that $\phi[i] > \phi[j]$. For the remainder of the cases, it suffices to demonstrate the easier proof that $\phi[i]$ is no worse than $\phi[j]$. In the worst case, $\phi[i]$ makes no decisions in M_{II}; in the best case, $\phi[i]$ decides all of M_{II}.

Lemma 4 *"Worst Case." If $\phi[i]$ and $\phi[j]$ are two probes satisfying Condition 1, with $M_{II} \neq \emptyset$ in state ξ_E^M. Assume that $\phi[i]$ makes no decisions in M_{II}. Then $\phi[i] > \phi[j]$.*

Proof. As before, we expand the resulting states by the amount of shift with $n_I = \|M_I\|$. Here, $[\xi]_{n_I}$ means to shift past all of M_I up to M_{II}.

$$\xi_i = q[\xi]_0 + \sum_{n=2}^{n_I-1} \overline{q}^n q[\xi]_n + \overline{q}^{n_I} q[\xi]_{n_I}. \qquad (15)$$

$$\xi_j = q[\xi]_0 + \sum_{n=2}^{n_I-1} \overline{q}^n q[\xi]_n + \overline{q}^{n_I} q[\xi]_{n_I}. \qquad (16)$$

Again, we compare the resulting states term by term.

- In the first two terms in both Equations 15 and 16 some or all of M'_I remains undecided.

$$\begin{array}{l} \phi[i](\xi) \rightarrow \xi \begin{array}{l} \{M'_I \quad M'_{II} \quad M'_{III} \quad M'_{\text{new}}\} \\ \{E'_I \quad E'_{II}\boxed{i'} \quad E'_{III} \quad\quad\ \} \end{array} \\ \hline \phi[j](\xi) \rightarrow \xi \begin{array}{l} \{M'_I \quad M'_{II} \quad M'_{III} \quad M'_{\text{new}}\} \\ \{E'_I\boxed{j'} \quad E'_{II} \quad E'_{III} \quad\quad\ \} \end{array} \end{array}$$

Again, for any subsequent series of probes $\phi[J]$ applied to ξ_j, let $\phi[I] = \text{Tr}[i \leftrightarrow j]{:}\phi[J]$. Every decision that $\phi[J]$ makes is also made by $\phi[I]$, and vice versa. At this point one of two possibilities occurs.

- At some point $\phi[J]$ and $\phi[I]$ might decide all of M'_I with part of M_{II} still undecided. Call these states ξ_J and ξ_I, respectively. That is, we have shifted past j' but not i'. Let m'_{II} be the first undecided in index in the original M_{II} region. m'_{II} is now index 1 in the new state. We have shifted a total of $m'_{II} - 1$ so far, and the old index i is $i' = i - (m'_{II} - 1)$.

$$\begin{array}{l} \phi[I](\xi_i) \rightarrow \xi \begin{array}{l} \{m'_{II} = 1, M'_{II} \quad M'_{III} \quad M'_{\text{new}}\} \\ \{E'_{II}\boxed{i'} \quad\quad\quad E'_{III} \quad\quad\ \} \end{array} \\ \hline \phi[J](\xi_j) \rightarrow \xi \begin{array}{l} \{m'_{II} = 1, M'_{II} \quad M'_{III} \quad M'_{\text{new}}\} \\ \{E'_{II} \quad\quad\quad E'_{III} \quad\quad\ \} \end{array} \end{array} \tag{17}$$

At this point, for any sequence of probes $\phi[J']$ applied to ξ_J, let $\phi[I'] = \text{Tr}[i' \rightarrow \emptyset]{:}\phi[J']$. $\phi[I']$ makes a decision everywhere that $\phi[J']$ does, but potentially does so in one less probe if $\phi[J']$ includes $\phi[i']$. We already know that $\phi[i]$ yields no decisions in M'_{II}. In this case, by Lemma 2 and Definition 3, $\phi[i] > \phi[j]$.

- States ξ_J and ξ_I do not arise, because we decide all of M_{II} before M_I is decided. We shift past j and i in one fell swoop. We end up in identical states as illustrated by Scenario 14. Thus, $\phi[i] = \phi[j]$.

- The probes $\phi[i]$ and $\phi[j]$ decide all of M_I, giving us Scenario 17. We have already shown in this case that $\phi[i] > \phi[j]$.

In all cases $\phi[i]$ leads to a state as good as or better than the state for $\phi[j]$. Thus, on the average, $\phi[i] > \phi[j]$. □

Lemma 5 *"Best Case." If $\phi[i]$ and $\phi[j]$ are two probes satisfying Condition 1, with $M_{II} \neq \emptyset$ in state ξ_E^M. Assume that $\phi[i]$ decides all of M_{II}. Then $\phi[i] > \phi[j]$.*

Proof. As before, we expand the resulting states by the amount of shift, only this time, $\phi[i]$ might shift up to M_{III} whereas $\phi[j]$ can only shift up to M_{II}. $n_I = \|M_I\|$, so that $[\xi]_{n_I}$ shifts up to m_{II} and $[\xi]_{n_I+1}$ shifts past M_{II} to M_{III}.

$$\xi_i = q[\xi]_0 + \sum_{n=2}^{n_I-1} \overline{q}^n q[\xi]_n + \overline{q}^{n_I} q[\xi]_{n_I+1}. \tag{18}$$

$$\xi_j = q[\xi]_0 + \sum_{n=2}^{n_I-1} \overline{q}^n q[\xi]_n + \overline{q}^{n_I} q[\xi]_{n_I}. \tag{19}$$

Again, we compare the resulting states term by term, only this time M_{II} is decided completely by ξ_i, but is undecided in ξ_j.

- The first two terms in both Equations 18 and 19 give the resulting states.

$$\begin{array}{l} \phi[i](\xi) \rightarrow \xi \begin{array}{l} \{M'_I \qquad\qquad\quad M'_{III} \quad M'_{\text{new}}\} \\ \{E'_I \quad E'_{II}\boxed{\text{i}'} \quad E'_{III} \qquad\quad \} \end{array} \\ \hline \phi[j](\xi) \rightarrow \xi \begin{array}{l} \{M'_I \quad M'_{II} \quad M'_{III} \quad M'_{\text{new}}\} \\ \{E'_I\boxed{\text{j}'} \quad E'_{II} \quad E'_{III} \qquad\quad \} \end{array} \end{array}$$

For any subsequent series of probes $\phi[J]$ applied to ξ_j, let $\phi[I] = \text{Tr}[i \leftrightarrow j]{:}\phi[J]$. Every decision that $\phi[J]$ makes either is also made by $\phi[I]$ or was previously made by $\phi[i]$. Thus, $\Delta_i \supseteq \Delta_j$ remains true.

At this point one of two possibilities occurs.

- At some point $\phi[J]$ and $\phi[I]$ decide all of M'_I with part of M'_{II} still undecided by $\phi[J]$. Call these states ξ_J and ξ_I, respectively. Let M^J_{II} = the undecided indices in M'_{II} in ξ_J. Because M'_{II} has already been decided by $\phi[i]$, $\Delta_i = \Delta_j \cup M^J_{II}$. We also shift ξ_I past M'_{II}, thus, $\eta_i > \eta_j$. By Definition 2, $\phi[i]$ leads to a better algorithm than $\phi[j]$.

$$\begin{array}{l} \phi[I] \rightarrow \xi \begin{array}{l} \{M'_{III} \quad M'_{\text{new}}\} \\ \{E'_{III} \quad \} \end{array} \\ \hline \phi[J] \rightarrow \xi \begin{array}{l} \{M^J_{II} \quad M'_{III} \quad M'_{\text{new}}\} \\ \{\qquad\quad E'_{III} \qquad\quad \} \end{array} \end{array} \tag{20}$$

- States ξ_J and ξ_I do not arise, because the probes in $\phi[J]$ decide all of M'_{II} by the time M'_I is decided, causing us to shift past j and i. We end up in identical states as illustrated by Scenario 14. Thus, $\phi[i] = \phi[j]$.

- The probes $\phi[i]$ and $\phi[j]$ decide all of M_I producing Scenario 20. We have already shown that in this case, $\phi[i] > \phi[j]$.

In all cases, $\phi[i] \geq \phi[j]$, so that on the average, $\phi[i] > \phi[j]$. □

Finally, we show that for any state ξ^M_E, $\phi[i]$ is at least as good as $\phi[j]$.

Lemma 6 *"General Case." If $\phi[i]$ and $\phi[j]$ are two probes satisfying Condition 1, then $\phi[i] \geq \phi[j]$.*

Proof. $\phi[i]$ might have made decisions in M_{II}, thus, $\Delta_i \supseteq \Delta_j$. For any subsequent series of probes $\phi[J]$ applied to ξ_j, let $\phi[I] = \mathrm{Tr}[i \leftrightarrow j]{:}\phi[J]$. At some point, if $\eta_I > \eta_J$ and J contains a probe j'' $< \eta_I$, then ignore probe j'' in I, because we have shifted past j'' already in ξ_I. Every decision that $\phi[J]$ makes either is also made by $\phi[I]$ or was originally made by $\phi[i]$. Thus, $\Delta_I \supseteq \Delta_J$ remains true and it follows directly that $\eta_I \geq \eta_J$. Thus $\phi[i] \geq \phi[j]$. □

Theorem 1 *For any state ξ_E^M, with unexamined indices, j, i, with $i > j$, $\phi[i] > \phi[j]$.*

Proof. Lemmas 3, 4, 5, and 6 cover all possible cases. In every case, either $\phi[i] = \phi[j]$,$\phi[i] \geq \phi[j]$, or $\phi[i] > \phi[j]$. Thus, on the average, $\phi[i]$ leads to the better algorithm. □

6.4 Main Proof

We now derive the optimum monotonic algorithm, by comparing n-step algorithms, as opposed to single probes. Let $a_1, a_2, \cdots, a_n \in E$, that is they are potential probes.

Lemma 7 *Let $\phi[i] > \phi[j]$, then $\phi[i,j] > \phi[j,i]$.*

Proof. Assume we are in state ξ_E^M, with $i > k$. Consider the probabilities that $\phi[i,j]$ results in only $\phi[i]$ or that $\phi[i,j]$ results in $\phi[i]\phi[j]$. We get

$$\phi[i,j](\xi) \rightarrow (\bar{q}\phi[i] + q\phi[i]\phi[j])\ (\xi) \quad (21)$$

$$\phi[j,i](\xi) \rightarrow (\bar{q}\phi[j] + q\phi[j]\phi[i])\ (\xi). \quad (22)$$

Equation 21 indicates that only $\phi[i]$ is done with probability $\bar{q}$ (because a decision is made at 1) and that both $\phi[i]\phi[j]$ are done with probability q. The last terms in Equations 21 and 22 are identical because, $\phi[i]\phi[j] = \phi[j]\phi[i]$. Thus, comparing $\phi[i,j]$ with $\phi[j,i]$ amounts to comparing $\phi[i]$ with $\phi[j]$. From Theorem 1, we get $\phi[i,j] > \phi[j,i]$. □

Lemma 8 *For $n > 2$, if $i > j$, then the n-step algorithm $\phi[i,j,a_3,\cdots,a_n] > \phi[j,i,a_3,\cdots,a_n]$.*

Proof. Expand each of the n-step algorithms. The probability that we reach probe a_3 is q^2, because both $\phi[i]$ and $\phi[j]$ must match. Thus, the probability that the 2-step algorithms $\phi[i,j]$ or $\phi[j,i]$ suffice is $(1-q^2)$.

$$\phi[i,j,a_3,\cdots,a_n] \rightarrow (1-q^2)\phi[i,j] + q^2\phi[i]\phi[j]\phi[a_3,\cdots,a_n]$$

$$\phi[j,i,a_3,\cdots,a_n] \rightarrow (1-q^2)\phi[j,i] + q^2\phi[j]\phi[i]\phi[a_3,\cdots,a_n]$$

The last term in both equations are identical, because $\phi[i]\phi[j] = \phi[j]\phi[i]$. From Lemma 7, we know that $\phi[i,j] > \phi[j,i]$, proving this lemma. □

Lemma 9 *For $n > 2$ and $1 \leq k \leq n\text{-}2$, if $i > j$, then the n-step algorithm $\phi[a_1,\cdots,a_k,\underline{i,j},\cdots,a_n] > \phi[a_1,\cdots,a_k,\underline{j,i},\cdots,a_n]$.*

Proof. Expand each of the n-step algorithms as before. The probability that $\phi[a_1,\cdots,a_k]$ does not decide at 1 is $(1-q^k)$. The difference between the equations is underlined.

$$\phi[a_1,\cdots,a_k,i,j,a_{k+3},\cdots,a_n] \rightarrow (1-q^k)\phi[a_1,\cdots,a_k]+q^k\phi[a_1]\cdots\phi[a_k]\phi[\underline{i,j},\cdots,a_n]$$

$$\phi[a_1,\cdots,a_k,j,i,a_{k+3},\cdots,a_n] \rightarrow (1-q^k)\phi[a_1,\cdots,a_k]+q^k\phi[a_1]\cdots\phi[a_k]\phi[\underline{j,i},\cdots,a_n]$$

The first term in equations is identical, thus, the difference between the two equations amounts to comparing $\phi[i,j,\cdots]$ versus $\phi[j,i,\cdots]$. From Lemma 8, we know that $\phi[i,j,\cdots] > \phi[j,i,\cdots]$, thus proving this lemma. □

We now show that if i is the best single probe among $\{i,a_1,\cdots,a_n\} \in E$, then probing i first is optimal. Actually, we show that probing any index other than i first is non-optimal.

Lemma 10 *For $n>1$ and $1 \le k \le n$, if $\max(i,a_1,a_2,\cdots,a_n) = i$, then $\phi[i,a_1,\cdots,a_n] > \phi[a_1,\cdots,a_k,i,a_{k+1},\cdots,a_n]$.*

Proof. We can "exchange" i to the front by applying Lemma 9 k times until i is the first probe. Each application of the Lemma 9 yields an improved (n+1)-step algorithm. For example, the first two applications give $\phi[a_1,\cdots,a_{k-2},\underline{i,a_{k-1}},\cdots,a_n] > \phi[a_1,\cdots,\underline{a_{k-1},i,a_k},\cdots,a_n] > \phi[a_1,\cdots,\underline{a_k,i},a_{k+1},\cdots,a_n]$. □

Because the choice of $\{a_1,\cdots,a_n\}$ was arbitrary, we have shown that an optimal algorithm must probe i first. The final step shows that RQ is the optimal monotonic algorithm.

Theorem 2 *RQ is the optimal monotonic algorithm.*

Proof. We show that for any state ξ_E^M, with $E=\{a_1,\cdots,a_n\}$ and $a_1 > a_2 > \cdots > a_n$, the optimal n-step algorithm is $\phi[a_1,a_2,\cdots,a_n]$. By Lemma 10 the optimal algorithm probes a_1 first. Assume probing a_1 does not decide at 1. We are now in a new state $\xi_{E'}^{M'}$, with $E'=\{a_2,\cdots,a_n\}$. Again by Lemma 10, we know that probing a_2 next is optimal. Thus, the optimal algorithm looks like $\phi[a_1,a_2,?,\cdots,?]$. Repeating this argument n-2 more times, we see that the optimal sequence n-step algorithm is $\phi[a_1,a_2,\cdots,a_n]$. That is, for any state ξ_E^M, the optimal algorithm probes the rightmost undecided index in text′. This is precisely RQ. □

6.5 Nonmonotonic Algorithms

It is possible that a non-monotonic algorithm may be better than RQ. A non-monotonic algorithm is not restricted to examining a character in text′ and may examine characters anywhere. For example, such an algorithm might look at text[P], text[T-P] and then text[T/2]. We give two preliminary reasons why non-monotonic algorithms are not likely to lead to a better approach. Then, we informally discuss the shortcoming of non-monotonic algorithms. Finally we prove that RQ cannot be beaten.

First, a non-monotonic algorithm is not likely to be time or space effective. If the probe must be in memory, we need T space to hold all of text[] in the worst case. When text[] is large, reading the next examined character directly from a file minimizes the required memory, but in practice, it is inefficient to read greatly differing indices of text[] from a file. Secondly, OPT2, which is known to be optimal over all algorithms, is monotonic.

Informally, RQ decides or "gobbles" text[ ] from left to right. One "non-monotonic" algorithm which is equally good to RQ is the reverse algorithm, REV, which gobbles text[ ] from right to left. In fact, any combination of RQ and REV which alternately gobbles "at the ends" and eventually finishes somewhere in the middle is as good as RQ. Each "bite" (probe) of text[ ] gobbles (decides) a random amount between 0 and P of text. Gobbling from the ends assures that we continue where we left off. However, a non-monotonic algorithm which gobbles a part of the middle, leaving a hole, cannot be optimal. The bite that connects the hole and the end might have eaten more, "wasting" part of that bite.

We define η_{REV} = the rightmost (greatest) undecided index, and $\text{text}'_{REV} = \text{text}[\eta_{REV} - P + 1 ... \eta_{REV}]$. REV is the algorithm that probes the leftmost unexamined character in text'_{REV}. We define η and text$'$ as before. A probe $\phi[u]$ is non-monotonic if u falls outside of both text$'$ and text'_{REV}. We now prove that a non-monotonic probe is suboptimal.

There are three types of probes. Non-monotonic probes are denoted $\phi[u]$ or $\phi[u_i]$. Probes within text$'$ are shown as $\phi[f]$ or $\phi[f_i]$. Probes within text'_{REV} are denoted $\phi[r]$ or $\phi[r_i]$. The set of decisions made by the non-monotonic probes $\phi[u_1], \cdots, \phi[u_i]$ is $\Delta_u = \{du_1, \cdots, du_l\}$, where $du_1 < du_2 < \cdots < du_l$. We define Δ_f and Δ_r similarly.

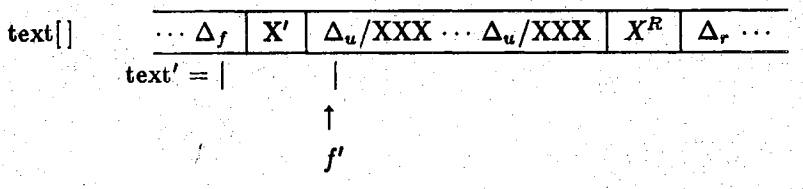
The entire probe sequence S for algorithm can be described as a combination of the three types. For RQ, the probe sequence is $\phi[f_1], \phi[f_2], \cdots, \phi[f_n]$; for REV, it is $\phi[r_1], \phi[r_2], \cdots, \phi[r_n]$. An example sequence for a non-monotonic algorithm might be $\phi[f_1], \phi[f_2], \phi[u_1], \phi[f_3], \phi[u_2], \ldots, \phi[u_n]$. The next lemma shows that shifting a set of probes, U, by i positions merely shifts the set of decisions made by U by i positions.

Lemma 11 *Let $U = \{\phi[u_1], \cdots, \phi[u_n]\}$ = a series of "u" probes. For a given pattern, over all text strings, the probabilistic set of decisions made by U is $\Delta_U = \{du_1, \cdots, du_l\}$, For integer i, let $U + i = \{\phi[u_1 + i], \cdots, \phi[u_n + i]\}$, then $\Delta_{U+i} = \{du_1 + i, \cdots, du_l + i\}$.*

Proof. Adding i corresponds to shifting the set of probes U either left (i < 0) or right (i > 0). Because the text strings are random, U and $U + i$ encounter identical strings from a probabilistic standpoint. The only difference is the shift of i. Thus, for a given pattern, the set of decisions is merely shifted by i. □

Theorem 3 *Any sequence of probes S that contains "u" probes is suboptimal.*

Proof. Eventually, Δ_f, Δ_u, and Δ_r must cover all of $1, \cdots, T - P + 1$, because we must decide all of text. At some point in time, we join $\Delta_f = \{1, 2, \cdots, df_n\}$ and Δ_u, when $df_n = du_1 - 1$. Without loss of generality, assume that (1) the probe that joins Δ_f and Δ_u is $\phi[f']$ and (2) Δ_u and Δ_r have not been joined yet. Let U be the set of "u" probes prior to $\phi[f']$. Before probing $\phi[f']$, the situation looks as follows, where X's represents an undecided region. X$'$ is the set of indices that will be decided by $\phi[f']$ and X^R is the set of undecided indices between Δ_u and Δ_r.



If U contains probes that have jumped around text[], then Δ_u/XXX may consist of many disconnected regions where decisions have been made. In this case, $\phi[f']$ joins the first (leftmost) region in Δ_u with Δ_f. In any case, text′ spans X′, because $\phi[f']$ decides all of X′.

As before, $\phi[f']$ probes text[f'], the rightmost unexamined character in text′. We might have f′ > du_1, so that $\phi[f']$ might have decided at du_1 had the probes in U not already decided du_1. In general, let $du_1, \cdots, du_i$ be the indices that $\phi[f']$ would have decided, were they not already decided. In this case, it follows that $(F, U+i)$ is a better set of probes than (F, U). By Lemma 11, Δ_{U+i} is Δ_U shifted right by i positions, and hence, Δ_{U+i} decides part (or all) of X^R, but is otherwise identical to Δ_u. Thus, in the same number of probes, $(F, U+i)$ decides "everywhere" that (F, U) does, and $(F, U+i)$ decides part of X^R. By Lemma 2, (F, U) cannot be optimal. For other combinations of the assumptions (1) and (2) the reasoning is identical. If follows directly that a non-monotonic algorithm cannot be better than RQ. □

Thus, RQ is optimal over all algorithms.

7 Modeling Other Algorithms

The KMP algorithm is the monotonic algorithm that probes the *leftmost* unexamined character in text′. When a decision is made at 1, η is incremented as usual. If probing text′[e] decides at 1, we know that text′[1...e-1] matches the pattern. For fixed-string patterns, text′[1...e-1] = patt[1...e-1], which is known precisely. We can precompute the decisions that have been made so far in a table, Jump[], of length P. When probing text′[e] decides at 1, η is incremented by Jump[e]. Note that KMP probes text[1], text[2], ..., text[T] in order regardless of the pattern or the text.

The Boyer-Moore algorithm is similar to RQ′, except that it does not make decisions past the first partial match in M, so that M = $\{1, 2, \cdots, P\}$ = {*} all the time. In other words, probing text′[e] makes decisions at 1,2,...,i, $i \geq 0$, which results in a shift of i, and a new M = {*}. BM is suboptimal because it does not make decisions past i. For fixed-string patterns, if examining text′[e] decides at 1, we know that text′[e+1...P] = patt[e+1...P], and we can precompute the other decisions that have been made in table, Jump[] of length P. In addition, BM uses the Skip table described in Section 5.3, which takes into decisions made by text′[e]. BM uses the maximum shift of the two tables.

8 Results

To test our various approximations, we measured $\overline{\Phi(OPT2)}$, $\overline{\Phi(BM)}$, and $\overline{\Phi(RQ)}$ for different P and α. We also compared the actual running time of BM and RQ′. We made all our measurements on a SUN-3/60 workstation running the Unix operating system.

To test the feasibility of RQ, we implemented a version of it for patterns with $P < 32 = W$ (the wordsize of our machine). We generated random text files of length 50,000 for alphabets of size 10 and 26. We varied P from 2 to $\|\Pi\|$. For each value of P, we searched for at least 30 random fixed-string patterns and averaged the results. In all cases, the predicted running time from Equation 10 was in error by less than 1.3% compared to the measured data.

We also measured RQ when $P \gg \alpha$. For each value of P, we searched for 128 or more random patterns on a text file of at least 50,0000 characters. We have normalized the data for a 10,000 character text file.

The rightmost column of the table shows that the upper bound set by Equation 11 is 20–30% too high.

α	P	$X = \log_\alpha(P)T/P$	Measured	Measured / X
2	16	2500	2757	1.10
2	31	1598	1632	1.02
3	27	1111	1196	1.07

To test our OPT2 approximation, we ran RQ on a variety of patterns, using an alphabet size of 10. We ran 2000 *fixed-string* searches on a 50,000 character text file. The measured number of probes was 0.8% greater than Equation 3 predicted. In an informal survey using *fixed-length* patterns such as [ab][acdfgij] or [cde][ac], the predicted number of probes was consistently 1.3%–2.5% below the measured data. In the survey, we tried 4 forms of patterns with an average of 5 runs per pattern type.

We checked the validity of the BM approximation in Equation 4 for different α and P. For searched for at least 100 patterns of each length in a 50,000 character text file, for $\alpha = 10$, and $\alpha = 26$. For $P \leq \alpha$, our approximation was in error less than 1.9% ($\alpha = 26$, P = 24, not shown); for $P \leq 2\alpha$, the maximum error was 3.6%. Our results are summarized below.

α	P	Predicted	Measured	$\frac{(Meas-Pred)}{Pred}$
10	6	11856	11824	-0.27%
10	8	9754	9631	-1.27%
10	10	8529	8401	-1.53%
10	20	6324	6107	-3.55%
10	30	5801	5670	-2.31%
26	6	9538	9516	-0.23%
26	8	7426	7391	-0.47%
26	12	5327	5286	-0.77%
26	16	4291	4244	-1.10%
26	26	3128	3115	-0.42%
26	36	2644	2616	-1.08%
26	42	2477	2444	-1.35%
26	52	2299	2252	-2.06%

Finally, we compared the running time of RQ′ described in Figure 5 with the BM algorithm. Although RQ′ is not optimal, when the pattern is a fixed-string, it probes no more characters than BM, so $\Phi_{wc}(RQ')$ is $O(T)$. We obtained a BM implementation that had a very tight loop as prescribed in [BM77]. We compared the CPU time required for each of the algorithms. For "typical" searches through a list of Internet hosts, BM was from 1.3 to 2 times faster than RQ′. BM was faster than RQ′ on short patterns of length 2–4. Finally, for "long" patterns when $P > \alpha$, RQ′ was typically two to three times faster than BM.

9 Conclusion

We have presented an algorithm for finding all occurrences of a fixed-length patterns in a text string. Our algorithm examines the minimum average number of text characters over all patterns and text strings and is

optimal in this respect. We have also presented a general method for approximating the number of probes, and hence the running time, of many pattern-matching algorithms. As an application of our method, we have derived the asymptotic behavior of $\overline{\Phi(OPT2)}$, $\overline{\Phi(BM)}$, and $\overline{\Phi(RQ)}$. Our approximations correlate well with measured data. We have presented an efficient variant of RQ for short patterns. Finally, we have proven that RQ is optimal over all algorithms.

One area merits future work. Our analysis uses the average q value. For certain alphabets and patterns, RQ is not optimal. For example, when searching for buzz@ in standard English, RQ examines a minimum of $2T/5$ characters, because examining text'[5] never makes a decision at 1 due to the wildcard character. Thus, in the best case 2 probes make 5 decisions. A better algorithm for this particular pattern is $\phi[4, 3, 2, 1]$, which most likely probes $T/4$ characters. The problem is to determine the optimal algorithm given each of the character probabilities and the pattern.

References

[Abr87] Karl Abrahamson. Generalized string matching. *SIAM Journal of Computing*, 16:1039–1048, December 1987.

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[AG86] Alberto Apostolico and Raffaele Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM Journal of Computing*, 15(1):98–105, February 1986.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

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

[GO80] Leo J. Guibas and A. M. Odlyzko. A new proof of the linearity of the Boyer-Moore string search algorithm. *SIAM Journal of Computing*, 9:672–682, 1980.

[Har71] Malcolm C. Harrison. Implementation of the substring test by hashing. *Communications of the ACM*, 14:777–779, 1971.

[Hor80] R. Nigel Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, June 1980.

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1977.

[KR81] R. M. Karp and M. O. Rabin. *Efficient Randomized Pattern-Matching Algorithms.* Technical Report TR-31-81, Aiken Computer Lab., Harvard Univ., Cambridge, MA, 1981.

[Sch88] R. Schaback. On the expected sublinearity of the Boyer-Moore algorithm. *SIAM Journal of Computing*, 17(4):648–658, August 1988.