AVERAGE-TIME ANALYSIS OF STRING-SEARCHING ALGORITHMS

by

Paul Joseph Bayer

S.B., Massachusetts Institute of Technology
(1973)

S.M., Massachusetts Institute of Technology
(1975)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1979

Signature of Author . . . . Paul J. Bayer
Dept. of Electrical Engineering and Computer Science, 14 May 1979

Certified by. . . . . Ronald L. Rivest
Thesis Supervisor

Accepted by . . . .
Chairman, Department Committee

Acknowledgement

I would like to thank Ron Rivest for his encouragement and assistance over the last five years. I am indebted to Fred Hennie for his counsel and aid throughout my graduate career. Len Adleman provided useful comments. Leo Guibas contributed some useful ideas in the early part of this research. I would especially like to thank D.Knuth, J.Morris, V.Pratt, P.Weiner, E.McCreight, R.Boyer, and J.Moore, for without their algorithms this thesis would not have been possible.

Finally, I thank my wife Terry for putting up with my travails and changing more than her share of the diapers.

This research was conducted at the MIT Laboratory for Computer Science and was supported in part by the National Science Foundation under grants MCS76-14294, MCS77-19754, and MCS78-05849. Some formulas were computed using MACSYMA, a research project at the MIT Laboratory for Computer Science, sponsored by the Department of Energy under contract number E(11-1)-3070 and the National Aeronautics and Space Administration under grant NSG 1323.

I. Introduction

In this thesis we study algorithms for string searching. That is, we study algorithms which, when given two strings of characters, the pattern and the text, determine where in the text the pattern occurs as a substring. Our particular emphasis is on the average amount of time used by these algorithms.

String searching algorithms are in wide use today, and their use will increase in the future. All text editing systems have some facility for finding a pattern in the text being processed. As computers are used more and more in offices, especially for text processing, the use of string-searching algorithms will increase. The goal of our thesis research is to further the search for efficient algorithms through theoretical analysis.

We also hope to increase our understanding of the design and analysis of algorithms in general. Our techniques of analysis may be applicable to other problems. The issues of time-space trade-offs and local versus global information are important in string searching, and our increased understanding of these issues may aid in the study of other problems.

I.A Formal Statement of the Problem

A string-searching algorithm has as input a pattern $p=p_1p_2\cdots p_m\in\Sigma^m$ and text $t=t_1t_2\cdots t_n\in\Sigma^n$ for some fixed, finite alphabet Σ. (Throughout this thesis the letters p, t, m, and n will be used only as defined above. Also, we will assume that Σ does not contain $*$ and \$, so that we may use these characters for special purposes.) We say that p occurs in t at j if for all i, $1\le i\le m$, $t_{j-1+i}=p_i$. An all-occurrences algorithm finds all j, $1\le j\le m-n+1$, such that p occurs in t at j. A first-occurrence algorithm finds the least such j. Unless stated otherwise we will always be considering all-occurrences algorithms. This allows us to express the time used by an algorithm in terms of m and n without worrying about where the pattern first occurs. However, all the algorithms we study can be modified to find the first occurrence, and most of our analysis is applicable to this case.

There are two definitions of "time" that we will study. Ultimately we would like time to mean actual time on a real computer. For our purposes we will assume that our algorithms are executed on some random access machine on which one might run programs written in Algol or PASCAL, for example. Our times will be stated using the O, Θ, Ω notation as described by Knuth [Kn3], which neatly sidesteps the issue of particular computers and their speeds.

In most of our analysis we use a simpler definition for time -- the number of times text characters are examined. This simplifies the analysis greatly. This analysis does aid in the "computer time" analysis, however, since all the algorithms we study have the property that if k is the number of text examinations made, then, except for some preprocessing of the pattern, the running time is $\Theta(k)$.

Finally, when we speak of average time we are assuming that the text string is random, but the pattern may not be. Either the pattern will be fixed, or we will be interested in the average time for the worst pattern. Initially one can assume that all text strings of length n are equally likely. In fact, our results hold for a general class of probability distributions, but assuming "equally likely" may aid in understanding the proofs.

1.B Summary of Results

Our research has produced results in three principal areas. First, we have defined a restricted model for a string-searching algorithm, a finite-state machine called a string-searching automaton (SSA). For this model we have developed techniques for computing the exact average-time behavior of an algorithm, for computing the asymptotic average-time behavior, and for finding an optimal algorithm for a given pattern. The analysis techniques use the theory of Markov chains, while the optimization technique uses Markov chains, linear programming, and a kind of randomization.

The second focus of our study has been a new algorithm for string searching. The algorithm is an elaboration of an idea of Knuth [KMP]. We can prove that the average number of text characters examined by this algorithm is within a constant factor of optimal. A variation of the algorithm can be used to solve a related problem, finding any of a set of patterns.

Our third area of study has been into the simplest special case of string searching, when $\Sigma=\{0,1\}$ and $p=0^m$. In this case we can prove that the average number of characters examined by our new algorithm is minimum, even when a very general model for an algorithm is considered. We get an interesting time-space trade-off as a corollary of this result, and also see one instance where an interesting new conjecture is true. The conjecture is that an optimal all-occurrence algorithm can be found by extending an optimal first-occurrence algorithm so that it continues on to find all occurrences.

I.C Previous Research

There are three widely known algorithms for string searching. They all use the following outline:

1. A window of length m is placed over the text, initially over $t_1 \cdots t_m$.
2. The characters in the window are examined and compared with the corresponding pattern characters in some order, as long as they match.
3. If all the characters match, the pattern has been found.
4. When all the characters have been examined, or a mismatch found, the window is shifted to the right, and the procedure continued, if more text remains.

Each particular algorithm determines the order in which characters are examined in step 2, the distance the window is shifted in step 4, and the characters remembered when the window is shifted. Note that if the window is ever placed over an occurrence of the pattern, then the algorithm will find that occurrence. Therefore, an algorithm using this outline will be correct if the window is never shifted past an occurrence of the pattern.

The particulars of the three algorithms are described below.

The Naive Algorithm

This is the obvious algorithm. In step 2 the characters are matched from left to right. When a mismatch occurs, the window is shifted right by one position, and all characters in the new window are forgotten. This algorithm is correct, since the window is placed over every text position. The worst-case number of text character examinations happens when the pattern is, say, a^m, and the text is a^n. Then the number of examinations is $n \cdot m - m^2 + m$. Flajolet [Fla] has shown that the average number of characters examined is $O(n)$. The worst-case and average times are $O(n \cdot m)$ and $O(n)$ respectively.

The Knuth-Morris-Pratt Algorithm [KMP]

Knuth, Morris, and Pratt discovered an algorithm with linear worst-case running time. In step 2 the characters are matched from left to right. When a mismatch is found the algorithm shifts as far as possible based on the characters already matched. That is, if the window is over $t_j \cdots t_{j+m-1}$ and a mismatch is found between t_k and p_{k-j+1}, then the algorithm can shift to $t_i \cdots t_{i+m-1}$, where i is the least number such that $i>j$, $p_1 \cdots p_{k-i} = t_i \cdots t_{k-1}$, and $p_{k-j+1} \neq p_{k-i+1}$. If the algorithm shifted less, then it would already "know" that there was a mismatch. If the algorithm shifted more, it might miss an occurrence of the pattern. After the shift, the algorithm continues scanning the text at t_k, since the preceding characters in the window are known to match the pattern.

The amount to be shifted when there is a mismatch with p_ℓ depends only on the pattern, not the text. Therefore, it can be precomputed, and in [KMP] it is shown how the preprocessing can be done in time $\Theta(m)$.

The algorithm can be programmed so that the worst-case number of text characters examined is n, and the average is between n-m+1 and n. The worst-case and average times are both $\Theta(m+n)$. Galil and Seiferas [G&S] have discovered a spectrum of related algorithms with time complexity better than the naive algorithm and space complexity better than KMP.

The Boyer-Moore Algorithm [B&M]

Boyer and Moore noted that a great deal of time can usually be saved by scanning the window backwards. In step 2 the characters are examined from right to left. When a mismatch is found the window is shifted in a way similar to KMP. As an added heuristic the algorithm shifts even further, if possible, when the mismatched text character does not occur in the pattern at all. After the shift, all characters in the window are "forgotten" and the process restarted. As in KMP the shifts can be precomputed in time $O(m)$.

This algorithm is faster than KMP for two reasons. First, because of the right-to-left scan, the shifts may be significantly longer than in KMP. More important, however, is the fact that most of the time some of the characters shifted out of the window will never be examined at all. This gives rise to the possibility of a sublinear average running time.

The analysis of this algorithm is greatly muddied by its "forgetfulness." Two complex proofs have been published that show that the the number of times text characters are examined is

$\Theta(n)$ in the worst case if the pattern does not occur in the text [KMP,G&O]. Galil has shown that the algorithm can be modified in a simple way so that the worst-case time is $\Theta(n)$ even if p occurs in t. Analysis of the average running time is also complex. No good closed-form bound for the time is known, but $\Theta(n \cdot \log(m)/m)$ seems likely.

The KMP and Boyer-Moore algorithms showed two important facts about string searching. KMP showed that no character need be examined twice, and Boyer-Moore showed that some characters need never be examined at all. This led to some interesting results on lower bounds. Rivest [Riv] has shown that for any pattern, any algorithm, and any n there exists a $t \in \Sigma^n$ for which the algorithm examines at least n-m+1 characters. Moreover, for infinitely many n, all n characters must be examined in the worst case. This author found an infinite set of patterns for which the worst case for any algorithm is n for all but finitely many n, even when the pattern does not occur in the text. (Note that for $p=a^m$, $t=a^n$, $n \geq m$, all positions must be examined by any algorithm.) Tuza [Tuz] found this independently, and proved the stronger result, that this is the case for most patterns.

For the average case Yao [Yao] derived a lower bound of

$\Omega(n \cdot \log_q(m)/m)$ characters examined, where $q=|\Sigma|$, for almost all patterns. His proof, in fact, shows that this is a lower bound for every text string, not just a randomly selected string. Knuth [KMP] sketched an algorithm that examines $\Theta(n \cdot \log_q(m)/m)$ characters on the average. Thus the optimal average number of characters examined is known to within a constant factor. However, Knuth did not describe an implementation of his scheme that would take time $\Theta(n \cdot \log_q(m)/m)$ on a computer.

These bounds are summarized below, along with the results for our new algorithm.

	worst-case characters	worst-case time	average characters	average time
lower	$n-m+1$	$\Omega(n)$	$\Omega(n \cdot \log_q(m)/m)$	$\Omega(n \cdot \log_q(m)/m)$
bounds	n for most patterns			
naive	$n \cdot m - O(m^2)$	$\Theta(n \cdot m)$	$\Theta(n)$	$\Theta(n)$
KMP	n	$\Theta(n+m)$	$n-O(m)$	$\Theta(n+m)$
BM	4n	$\Theta(n+m)$	?	?
Knuth's	n	?	$\Theta(n \cdot \log_q(m)/m)$	?
Our Alg.	n	$\Theta(n+m)$	$n \cdot \log_q(m)/m+O(n/m)$	$\Theta(m+n \cdot \log_q(m)/m)$

1.D Related Problems

The string-searching problem as we have defined it is the simplest of a large number of pattern matching problems that people have studied. Most of the other problems are simple generalizations of the string-searching problem. A number of the problems are described in [AHU]. These include searching for any of a set of strings [A&C] and matching a regular expression. Fischer and Paterson consider patterns with "don't cares" [F&P]. Generalizations to other combinatorial objects are presented in [KMR]. Another approach to the problem in which the text is preprocessed has been studied by Harrison [Har].

One might argue that these generalizations pose more interesting questions than the problem we are considering, but this thesis will show that abundant questions remain even for the simple case.

II. Theoretical Models for Algorithms

The Knuth-Morris-Pratt and Boyer-Moore algorithms operate in two phases, preprocessing the pattern and scanning the text. In most uses of string searching the text is significantly longer than the pattern, so the second phase takes more time. With this in mind, our theoretical study focuses on the question: given a fixed pattern p, what are efficient algorithms for finding p? We might hope that there is an efficient preprocessing scheme which produces a near-optimal algorithm for each p. However, we will mainly be concerned in this thesis with the efficiency of the algorithms for each p, not the preprocessing time.

II.A Time

When p is fixed the number of text characters examined is an accurate measure for the time that an algorithm uses. (At least this is true of all the real and theoretical algorithms that we have studied.) Therefore, we fix upon this as our definition of time.

We now formalize our notion of average time. Let $T_A(t)$ be the time used by algorithm A on text t. For each t, let prob(t) be the probability that t is the text being searched. Then the average time $T_A = \sum_{t \in \Sigma^n} \text{prob}(t) \cdot T_A(t)$. Allowing any probability distribution would make our problem intractable, but we can be more general than simply assuming that each t is equally likely. Let $P: \Sigma \to (0,1)$ be a probability distribution on Σ. (We assume that for all $\sigma \in \Sigma$, $P(\sigma) > 0$, since otherwise we could take $\Sigma - \{\sigma\}$ as the alphabet.) The probability distributions on Σ^n that we will study are those for which the characters t_i are independent, identically distributed random variables with probability distribution P. (In information theory terms we would say that the text is from a discrete memoryless source.) This is equivalent to defining $\text{prob}(t) = \prod_{i=1}^{n} P(t_i)$.

II.B Decision Trees

Since we are assuming that the pattern and text are given to an algorithm simply as strings of characters, we can represent an algorithm as a decision tree. This is the most general model we will study.

We define a <u>decision tree algorithm</u> A for finding a (fixed) $p \in \Sigma^m$ in text strings $t \in \Sigma^n$ to be a rooted $|\Sigma|$-ary tree. Each (internal) node is labeled with an integer i, $1 \le i \le n$. From each node descend $|\Sigma|$ branches, each labeled with a distinct $\sigma\in\Sigma$. Every branch leads either to another node or to a leaf. Each leaf is labeled with a set $S\subset\{1,\cdots,n\}$. The operation of A on input $t\in\Sigma^n$ is

1. Start at the root.
2. When at a node labeled i, A examines t_i and moves along the branch labeled with t_i to the next node or leaf.
3. If the branch leads to a node, go back to 2.
4. If the branch leads to a leaf, then A halts and declares the set S labeling the leaf to be the set of positions where p occurs in t.

We sometimes refer to an execution of step 2 as a probe. We define the time for A on t to be the number of probes made.

Example A decision tree for $\Sigma=\{0,1\}$, p=01, n=4 is diagrammed in Figure II.1.

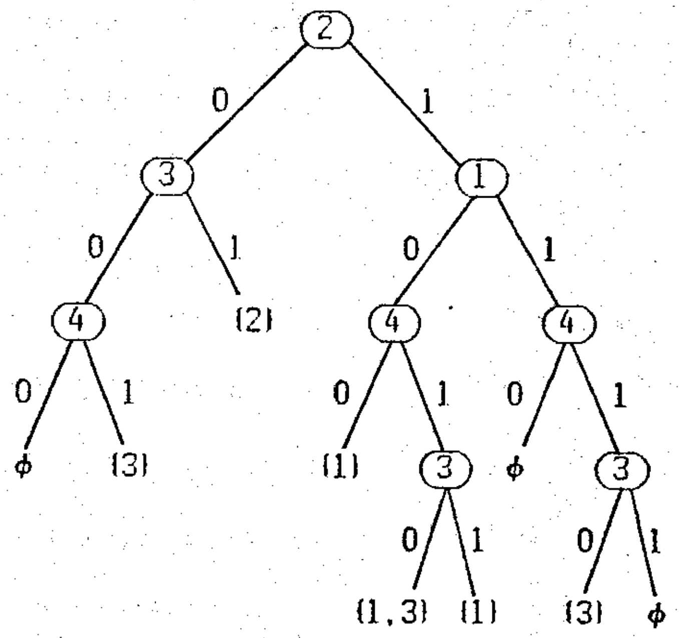


Figure II.1

In order to prove lower bounds on the time an algorithm must take it is necessary to determine the minimum amount of work that an algorithm must do to be correct. We formalize this notion below.

Definition A partial certificate (p.c.) is a string $u\epsilon(\Sigma\cup\{*\})^n$.

Definition A partial certificate for t is a p.c. in which for all i, $u_i{=}*$ or $u_i{=}t_i$. (In a sense, u matches t if you consider *'s to be "don't cares.")

Definition In a p.c., u, a position i such that $u_i{=}*$ is called unknown; one such that $u_i{\neq}*$ is known.

Definition A (complete) certificate for p is a p.c. u such that for all j, $1 \le j \le m-n+1$, either

(1) $u_{j-1+i}=p_i$ for $1 \le i \le m$ (p occurs in u at j)

or (2) $u_{j-1+i} \ne * \wedge u_{j-1+i} \ne p_i$ for some i, $1 \le i \le m$ (p does not match u at j).

Claim If a p.c., u, for t is a complete certificate for p then the set S of positions in t where p occurs is precisely the set of j for which (1) is satisfied.

Proof If p occurs in t at j, then (2) cannot be satisfied for j since for all i, $1 \le i \le m$, either $u_{j-1+i}=*$ or $u_{j-1+i}=t_{j-1+i}=p_i$ from the definition of partial certificate for t. Therefore, (1) must be satisfied. Conversely, if (1) is satisfied, then for all i, $1 \le i \le m$, $p_i=u_{j-1+i}=t_{j-1+i}$, so p occurs in t at j.◊

Corollary A partial certificate u for t is a complete certificate for p iff the set of positions in t where p occurs can be determined from u.◊

Definition The partial certificate u associated with node or leaf x of A is defined recursively as follows. If x is the root then $u=*^n$. Otherwise, x is the σ-child of some node y with partial certificate v and label j. Then we define $u_j=\sigma$ and $u_i=v_i$

for $i \neq j$.

<u>Claim</u> If A visits x when the text is t, then the p.c. associated with x is a p.c. for t.

<u>Proof</u> Follows easily from the operation of A and the definitions above.◊

<u>Definition</u> A node or leaf x of A is <u>reachable</u> if there is some t such that A visits x at some point when given text t.

<u>Claim</u> A decision tree algorithm is correct iff for all reachable leaves x, the p.c. associated with x is a complete certificate and the set S labeling x is the set of j satisfying (1) in the definition of complete certificate.

<u>Proof</u> Follows from previous claims.◊

The above derivation can be summarized as follows. For an algorithm to be correct, it must examine sufficient characters of t to construct a (complete) certificate for p which matches t. For an algorithm which never examines a text character twice, the time taken by the algorithm on t is simply the number of knowns in the certificate found. Therefore, we can shift our viewpoint slightly and define the output of a string-searching algorithm to be a certificate, rather than a list of positions where the pattern occurs.

II.C String-Searching Automata

There are two problems with studying decision-tree algorithms. First, the class is so large that we might have a hard time finding optimal algorithms. Second, the algorithms are so large that it would not be practical to construct one for arbitrary patterns. Also, it should be clear that a preprocessing algorithm of bounded complexity can only build decision trees which are representable in a bounded amount of space. Therefore, instead of studying decision trees, we will define a finite-state model for a string-searching algorithm. This model was first proposed by Knuth [KMP].

Informally, we define a <u>string-searching automaton</u> (SSA) for a pattern $p \in \Sigma^n$ to be a machine with a one-way, read only input tape, a finite-state control unit, and a memory of m cells called the window. The window can be randomly read by the control unit, but is changed only by shifting text characters from the tape through the memory (it acts like a shift register).

The machine starts with the first m characters of text in the window. At any step, an unknown character in the window is examined and compared with the corresponding pattern character. This continues until all characters in the window are known, or a mismatch occurs. In either case, the window is shifted, with new text entering at the right. (One can view this as the window shifting across the text, or the text shifting into the window.)

The description above is very similar to the outline for the algorithms described in Chapter 1. However, we put further restrictions on SSA's. First, they never forget; all relevant characters known before a shift are remembered after the shift (in their new locations). Second, each shift is as far as possible. Third, we require the SSA to operate based only upon the known characters in the window -- every time any particular set of knowns and unknowns is present in the window, the SSA must examine the same position. All of these restrictions seem consistent with the goal of finding fast algorithms. This is clearly true of the first two restrictions. When we argue that the third restriction yields fast algorithms, we rely on the intuition that the best position to examine in some configuration is the best position whenever that configuration recurs.

With these restrictions we can construct a simple formal definition for an SSA. For a fixed pattern $p \in \Sigma^m$, define a state set as $S = \{u \in (\Sigma \cup \{*\})^m \mid \forall i,\ 1 \le i \le m,\ u_i = *$ or $u_i = p_i\} - \{p\}$. (That is, S is the set of partial certificates for p with at least one unknown.) A <u>string-searching automaton</u> A is defined by a probe function $A: S \rightarrow \{1, \cdots, m\}$ such that for all $u \in S$, $u_{A(u)} = *$. The operation of A on input $t \in \Sigma^n$ is the following

1. $t_1 \cdots t_m$ is read into the window $w_1 \cdots w_m$. If $m<n$ A halts.
2. When in state u, A examines $w_{A(u)}$.
3. Let $u' \in (\Sigma \cup \{*\})^m$ be defined by $u'_{A(u)} = w_{A(u)}$, and $u'_i = u_i$ if $i \ne A(u)$.
4. If $w_{A(u)} = p_{A(u)}$ and u' has at least one unknown, set the current state to u' and go to 2.
5. If $w_{A(u)} = p_{A(u)}$ and u' has no unknowns, then the pattern has been found.
6. Let d be the least number, $1 \le d \le m-1$, such that for all i, $1 \le i \le m-d$, $p_i = u'_{d+i}$. If no such d exists, let $d=m$.
7. Shift the characters in the window to the left by d positions (i.e. set $w_i = w_{d+i}$) and move the next d text characters into $w_{m-d+1} \cdots w_m$. If insufficient text characters remain, A halts.
8. Define $u'' \in S$ by $u''_i = u'_{d+i}$ for $1 \le i \le m-d$ and $u''_i = *$ for $i > m-d$. Set the current state to u'' and go to 2.

It should be noted that an SSA never examines the same character twice. The time for A on text t is the number of times step 2 is executed.

We will diagram SSA's as illustrated in Figure II.2. Each state is represented by an oval containing the state name. The probe position A(u) is underlined in the name. For each $\sigma\epsilon\Sigma$ there is an arrow from the state to the next state, labeled with σ/d if σ causes a shift of length d, and labeled with σ/d◊ if the pattern is found. In some (maybe all?) SSA's certain states are unreachable. Those states are omitted in the diagram.

<u>Example</u> Below are three SSA's for Σ={0,1}, p=001.

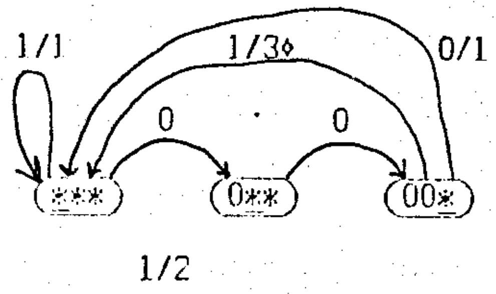


Figure II.2 (a)

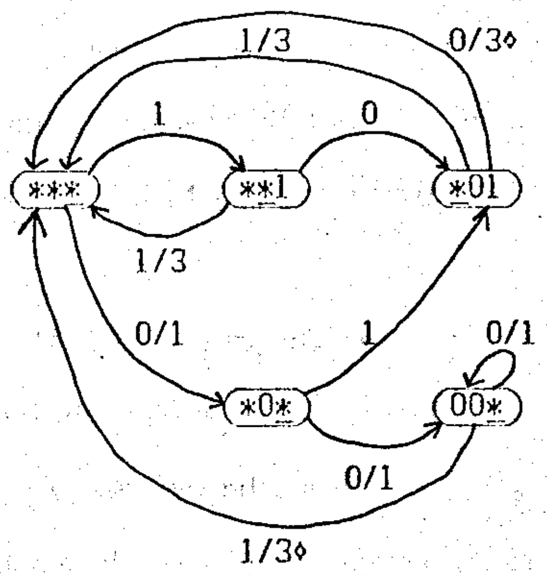


Figure II.2 (b)

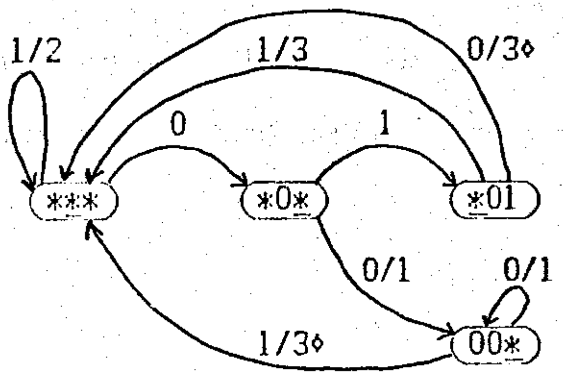


Figure II.2 (c)

For a given pattern we give names to two SSA's. The Knuth-Morris-Pratt SSA (KMP-SSA) is the one in which the leftmost unknown is examined in each state. Figure II.2 (a) shows a KMP-SSA. Note that a KMP-SSA always has m-1 states. We say that an SSA is a reduced-Boyer-Moore SSA (rBM-SSA) if in each state the rightmost unknown is examined. (We use the term "reduced" because, unlike the original Boyer-Moore algorithm, an rBM-SSA never examines the same character twice.) Figure II.2 (b) shows an rBM-SSA.

We conjecture that the class of SSA's contains an optimal algorithm for every pattern, optimal in the sense of examining the fewest text characters on the average of any algorithm. We cannot prove this in general, but for the case of $p=0^m$ it is true, as shown in Chapter 5. Nevertheless, for any of the known algorithms we can find an SSA which is at least as good. In addition, the average time for these algorithms can be analyzed in detail, as illustrated in the next chapter.

III. Markov Analysis of SSA's

In this chapter we will show how the constrained structure of SSA's leads to a method for analyzing their average time in detail. Through a trivial transformation an SSA can be made into a Markov chain. Then all the techniques of Markov analysis can be used. Most of the facts that we use about Markov chains are from Feller [Fel].

The Markov chain associated with an SSA A consists of the states of A and the transition probabilities

ϕ_{uv} = {the probability that A will move to state v after the next probe, given that A is in state u}.

In other words, if there are arrows labeled $\sigma_1,\sigma_2,\cdots,\sigma_k$ from u to v, then $\phi_{uv}=P(\sigma_1)+\cdots+P(\sigma_k)$. For each state u let $\phi_u(k)$={the probability of being in state u after k probes}. We can compute $\phi_u(k)$ from the recurrences:

$$\phi_u(0) = \begin{cases} 1 & \text{if } u=*^m \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_u(k) = \sum_v \phi_v(k-1)\cdot\phi_{vu} \quad \text{for } k>0.$$

Let s_u be the expected number of shifts on the arrows leaving u. Then the average number of characters shifted in doing the first k probes is $s(k)=m+\sum_{j=0}^{k-1}\phi_u(j)\cdot s_u$ (we include the initial loading of the window). The quantity s(k) is essentially the inverse of the quantity that we are really interested in -- the average number of characters examined in doing the first n shifts. We will show later exactly how these quantities are related.

Example In Figure III.1 we have an SSA for $\Sigma=\{0,1\}$, p=001.

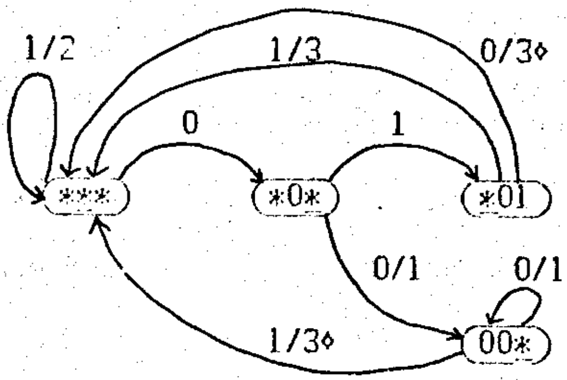


Figure III.1

Assume that P(0)=P(1)=0.5. Then we have

$$\begin{array}{c|cccc|}
u \quad v & \ast\ast\ast & \ast 0\ast & \ast 01 & 00\ast \\
\hline
\ast\ast\ast & 0.5 & 0.5 & 0 & 0 \\
\ast 0\ast & 0 & 0 & 0.5 & 0.5 \\
\ast 01 & 1 & 0 & 0 & 0 \\
00\ast & 0.5 & 0 & 0 & 0.5 \\
\hline
\end{array}$$

ϕ_{uv}

In Figure III.2 we diagram the Markov chain with the arrows labeled with the transition probabilities.

$$\begin{tikzcd}
\ast\ast\ast \arrow[loop above, "0.5"] \arrow[r, bend left, "0.5"] & \ast 0\ast \arrow[r, bend left, "0.5"] \arrow[dr, "0.5"] & \ast 01 \arrow[ll, bend right, "1"'] \\
 & & 00\ast \arrow[loop above, "0.5"] \arrow[ull, bend left, "0.5"]
\end{tikzcd}$$

Figure III.2

u \ k	0	1	2	3	4
***	1	0.5	0.25	0.5	0.5
0	0	0.5	0.25	0.125	0.25
*01	0	0	0.25	0.125	0.0625
00*	0	0	0.25	0.25	0.1875

$\phi_u(k)$

u	***	*0*	*01	00*
s_u	1	0.5	3	2

k	0	1	2	3	4	5
s(k)	3	4	4.75	6.375	7.8125	9

If a Markov chain has certain nice properties, it can be analyzed in an elegant way. A subchain is a subset of the sets of a Markov chain. A subchain is closed if every state reachable from a state in the subchain is in that subchain. A subchain is irreducible if every state in the subchain is reachable from every other state of the subchain.

Lemma III.1 The Markov chain for an SSA contains exactly one irreducible, closed subchain.

Proof Let f be the state entered when the pattern is found (it is unique). The set of states reachable from f is closed, by definition of closed. State f is reachable from every state, since an occurrence of the pattern will take the SSA to f. Finally, the states reachable from f must form the only closed subchain, since any closed subchain must contain f. Therefore, the states reachable from f constitute the one irreducible, closed subchain of the Markov chain.◊

In Figure III.3 is the rBM-SSA for 0100 which has an irreducible, closed subchain of {0***, 0**0, 0*00, *1**, *1*0, *100, *10*, 01**, 01*0}. (In all our other examples, the entire chain is irreducible.)

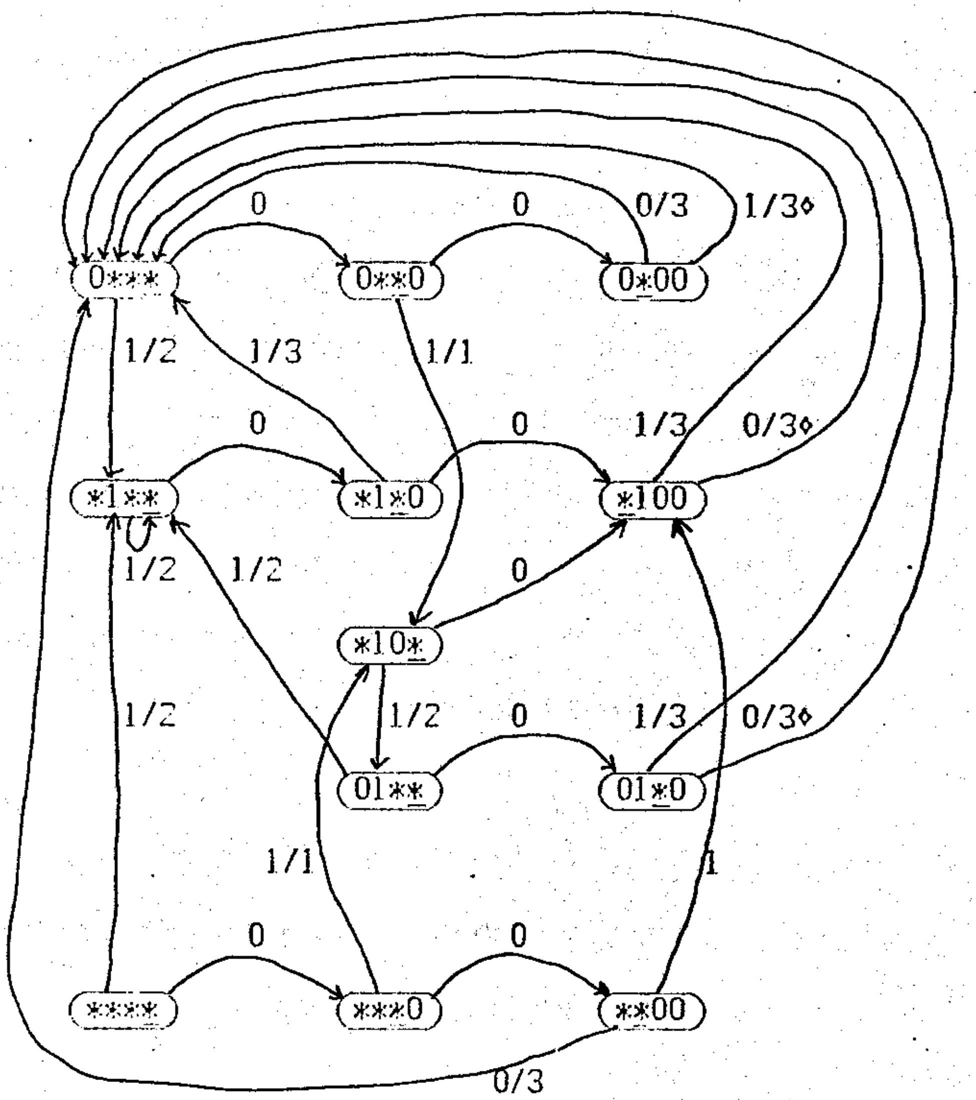


Figure III.3

Because the Markov chain has one irreducible, closed subchain, we get from Feller [Fel] that there exists an invariant probability distribution ϕ_u such that $\phi_u = \sum_v \phi_v \phi_{vu}$ and $\sum_u \phi_u = 1$. Moreover, the distribution ϕ_u is the unique solution to those equations. Thus, we have a method of computing ϕ_u. If A satisfies another nice property (aperiodicity), then $\phi_u = \lim_{k\to\infty} \phi_u(k)$. (We conjecture that A is never periodic, but have been unable to prove it.)

Let $s = \sum_u \phi_u s_u$, the average number of characters shifted per character examined given the invariant distribution. Then if A is aperiodic, this is the asymptotic number of characters shifted per character examined. (If A is periodic, this is still true, but showing it is not so easy.) However, what we are really interested in is the number of characters examined per shift. It is true, but not obvious, that this is 1/s. We show this later.

III.A Two Other Markov Models

We are interested in the number of characters examined per character shifted, but this cannot be computed directly from the Markov model. We need the dual of the Markov model, reversing the roles of shifts and probes. To get this dual it is convenient to use an intermediate model.

The complete Markov chain for an SSA A is derived from A by adding a state for each shift. The most concise way of doing this is to add states u_1, u_2, ···, u_d for each state u for which d is the maximum number of characters shifted on the arrows entering u. Then it is assumed that the text is shifted by one position when any of these shift states is entered. There are transitions with probability 1 from u_{i+1} to u_i and u_1 to u, and a transition from v to u with a shift of k is redirected to u_k. In addition, if $u=*^m$ there will be shift states $u_1,\cdots,u_m$, and u_m will be the new initial state. An example of a complete Markov chain is given in Figure III.4 with shift states drawn as rectangles.

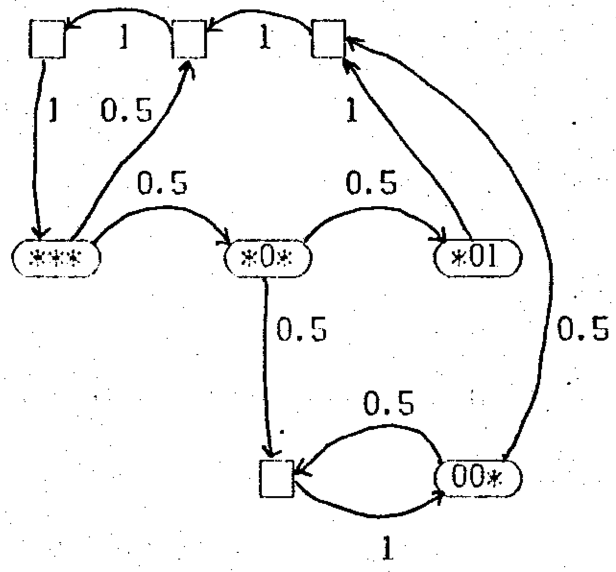


Figure III.4

We can formulate the average time for an algorithm in terms of the complete Markov chain. That is, T(n) is the expected number of probe states visited before reaching the n+1'st shift state (that is when the SSA halts). Again, it is not obvious how to compute this value, so we construct a third Markov model.

The <u>dual Markov chain</u> is constructed from the complete chain by deleting the probe states. The transition probabilities ψ_{ij} are determined by finding all paths in the complete chain from shift state i to shift state j containing no intermediate shift states. (We now assume that the shift states are numbered 1,2,··· with 1 being the initial state.) Then the transition probability ψ_{ij} is the sum of the probabilities of these paths. Note that every path from i to j includes at most m probe states, since the window is of length m and no character is examined twice. This makes the procedure for constructing the dual chain effective (and efficient). It is also convenient to define c_i as the average number of probe states on the paths leaving state i. The dual chain for our example is illustrated in Figure III.5.

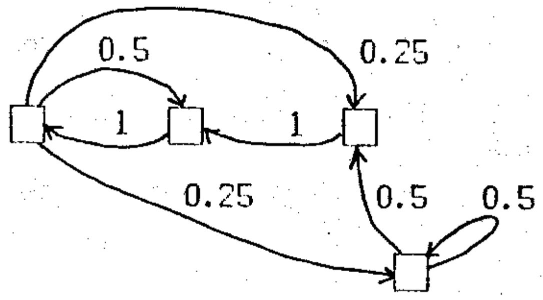


Figure III.5

From the dual Markov chain we can (finally) compute T(n). As in the original chain we define $\psi_i(n)$ to be the probability of being in state i after n shifts. Thus we have

$$\psi_i(1) = \begin{cases} 1 \text{ if } i=1 \\ 0 \text{ otherwise} \end{cases}$$

$$\psi_i(n+1) = \sum_j \psi_j(n) \cdot \psi_{ji}.$$

Then $T(n) = \sum_{k=1}^{n} \sum_i \psi_i(k) \cdot c_i$. We have reduced the problem to that of computing $\psi_i(n)$.

A number of properties of the dual chain allow us to concisely analyze $\psi_i(n)$. As before, the dual chain has one irreducible, closed subchain. Moreover, the dual chain has the property that we could not prove for the original chain -- it is aperiodic. A state i in this chain is <u>periodic</u> if it is a member of the closed subchain, and if there exists a k>1 such that each cycle from i back to itself is of length $\ell \cdot k$ for some integer ℓ.

The Markov chain is periodic if it contains a periodic state, otherwise it is aperiodic.

Lemma III.2 The dual chain is aperiodic.

Proof Let f be the shift state entered when the pattern is found. From [Fel] we know that if the chain is periodic, then f must be periodic. Also from [Fel], we know that if we can find two cycles from f back to f with lengths that are relatively prime, then f cannot be periodic. In fact there are cycles from f to f of lengths m and m+1 as witnessed by the text strings pp and pσp where p is the pattern and σ is any character. Since m and m+1 are always relatively prime, f must be aperiodic as must be the whole chain.◊

With these results we can now employ all the algebraic techniques of Feller [Fel] to produce the following results:

Lemma III.3

1. There exists an invariant distribution ψ_i such that

   $\psi_i = \sum_j \psi_i \psi_{ji}$, $\sum_i \psi_i = 1$, and $\psi_i = \lim_{k \to \infty} \psi_i(k)$.
2. The generating functions $\Psi_i(z) = \sum_{k \geq 1} \psi_i(k) \cdot z^k$ satisfy the equations $\Psi_i(z) = \psi_i(1) \cdot z + \sum_j z \Psi_j(z) \psi_{ji}$.

$\sum_i \Psi_i(z)=z+z^2+z^3+\cdots=z/(1-z)$.

3. The equations of (2) can be solved yielding $\Psi_i(z)=f_i(z)/g(z)$ for some polynomials f_i and g.

4. Using partial fraction techniques, we find that

   $\Psi_i(z)=\psi_i/(1-z)+\sum_j a_{ij}/(1-\lambda_j)^{e_j} + h_i(z)$

   where the a_{ij} are complex constants, the e_j are non-negative integers, the $h_i(z)$ are polynomials, and the λ_j are the complex roots of $g(z)$ with $|\lambda_j|<1$.

5. The form of $\Psi_i(z)$ in (5) implies that $\psi_i(n)=\psi_i+O(\lambda^n)$ where λ is any positive real number such that $\lambda<1$ and $\lambda>|\lambda_j|$ for all λ_j. Thus, the difference between $\psi_i(n)$ and ψ_i decreases exponentially.

6. In the (unfortunately few) cases where the roots λ_j can be exactly computed, we can find explicit formulas for $\psi_i(n)$.

Proof See Feller [Fel]. For more on generating functions see [Knl].◊

Using Lemma III.3 we can derive a concise form for the asymptotic average time for any SSA.

Theorem III.1 For an SSA the average time $T(n)$ satisfies $T(n)=cn+d+O(\lambda^n)$, where $c=\sum_i \psi_i c_i$ is the expected number of characters examined per text character for the invariant distribution, and d and λ are constants with $0 \le \lambda < 1$.

Proof

$$
\begin{aligned}
T(n) &= \sum_{k=1}^{n} \sum_{i} \psi_i(k) c_i \\
&= \sum_{i} c_i \sum_{k=1}^{n} \psi_i(k) \\
&= \sum_{i} c_i \sum_{k=1}^{n} (\psi_i + O(\lambda^k)) \\
&= n \cdot \sum_{i} \psi_i c_i + \sum_{i} c_i \cdot (c' + O(\lambda^n)) \text{ for some } c' \\
&= cn + d + O(\lambda^n) \text{ for some } d. \diamond
\end{aligned}
$$

Example We will use our usual example to illustrate a complete derivation of T(n). Recall that $\Sigma = \{0,1\}$, $P(0) = P(1) = 0.5$, $p = 001$. The SSA, Markov chain, complete Markov chain, and dual Markov chain are shown in Figures III.1, III.2, III.4, and III.5.

From the dual Markov chain we see:

$$
\begin{array}{c|cccc}
i \quad j & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 1 & 0 \\
3 & 0.25 & 0.5 & 0 & 0.25 \\
4 & 0.5 & 0 & 0 & 0.5
\end{array}
$$

ψ_{ij}

i	1	2	3	4
c_i	0	0	1.75	1

The invariant probabilities satisfy

$\psi_1 = \psi_3/4 + \psi_4/2$

$\psi_2 = \psi_1 + \psi_3/2$

$\psi_3 = \psi_2$

$\psi_4 = \psi_3/4 + \psi_4/2$

$\psi_1 + \psi_2 + \psi_3 + \psi_4 = 1$

The solution to these equations is

i	1	2	3	4
ψ_i	1/6	1/3	1/3	1/6

The generating functions satisfy

$\Psi_1 = z + z\cdot\Psi_3/4 + z\cdot\Psi_4/2$

$\Psi_2 = z\cdot\Psi_1 + z\cdot\Psi_3/2$

$\Psi_3 = z\cdot\Psi_2$

$\Psi_4 = z\cdot\Psi_3/4 + z\cdot\Psi_4/2$

$\Psi_1 + \Psi_2 + \Psi_3 + \Psi_4 = z/(1-z)$

They have solutions

$\Psi_1 = [z(2-z^2)(2-z)]/[2(1-z)(2+z)]$

$\Psi_2 = [2z^2(2-z)]/[2(1-z)(2+z)]$

$\Psi_3 = [2z^3(2-z)]/[2(1-z)(2+z)]$

$\Psi_4 = [z^4]/[2(1-z)(2+z)]$

They can be expressed in partial fractions as

$\Psi_1 = 1/[6(1-z)] + 4/[3(1+z/2)] - (z^2-3z+3)/2$

$\Psi_2 = 1/[3(1-z)] + 8/[3(1+z/2)] + z - 3$

$\Psi_3 = 1/[3(1-z)] - 16/[3(1+z/2)] + z^2 - 3z + 5$

$\Psi_4 = 1/[6(1-z)] + 4/[3(1+z/2)] - (z^2-z+3)/2$

If we let $\psi_i^*(n) = \sum_{k=1}^{n} \psi_i(k)$, then the generating function satisfies

$\Psi_i^*(z) = \sum_{n=1} \psi_i^*(n) \cdot z^n = \Psi_i(z)/(1-z)$

For our example, a partial fraction representation of Ψ_i^* is:

$\Psi_1^* = z/[6(1-z)^2] + 5/[9(1-z)] + 4/[9(1+z/2)] + (z-2)/2$

$\Psi_2^* = z/[3(1-z)^2] + 1/[9(1-z)] + 8/[9(1+z/2)] - 1$

$\Psi_3^* = z/[3(1-z)^2] - 2/[9(1-z)] - 16/[9(1+z/2)] - z + 2$

$\Psi_4^* = z/[6(1-z)^2] - 4/[9(1-z)] + 4/[9(1+z/2)] + z/2$

Let $T(z) = \sum_{n\geq 1} T(n)\cdot z^n = c_1\Psi_1^* + c_2\Psi_2^* + c_3\Psi_3^* + c_4\Psi_4^*$

For our example

$$T(z) = 3z/[4(1-z)^2] - 5/[6(1-z)] - 8/[3(1+z/2)] - 5z/4 + 7/2$$

This leads to the explicit form

$$T(n) = \begin{cases} 0 & \text{if } n<3 \\ 3n/4 - 5/6 - 8\cdot(-1/2)^n/3 & \text{for } n\geq 3 \end{cases}$$

The coefficient of n can be more easily computed as

$$c = \sum_i \psi_i c_i = (1/6)\cdot 0 + (1/3)\cdot 0 + (1/3)\cdot(7/4) + (1/6)\cdot 1 = 3/4.$$

III.B The Original Markov Chain Revisited

It would be nice if the dominant coefficient c in $T(n)=cn+d+o(1)$ could be computed directly from the original Markov model, saving the complexity of constructing the dual chain. We described in the beginning of this chapter how to compute s, the asymptotic average number of text characters shifted per character examined. We will now show that $c=1/s$.

Consider, once again, the complete Markov chain. Assume that the states are numbered $1,2,\cdots$ and let X denote the set of probe states and Y the set of shift states. We will use π_{ij} to denote the transition probabilities and π_i to denote the invariant distribution. Note that the number c in which we are interested can be defined as c=(the expected number of X states visited between two Y states). Similarly, s=(the expected number of Y states visited between two X states). We will show that c=1/s by proving the following theorem.

Theorem III.2 $c = (\sum_{u \in X} \pi_u)/(\sum_{i \in Y} \pi_i)$ and symmetrically, $s = (\sum_{i \in Y} \pi_i)/(\sum_{u \in X} \pi_u)$.

Proof The theorem holds in general for irreducible Markov chains, but we will use a property of this Markov chain to simplify the proof.

Note that in the complete chain there is no path with more than m consecutive probe states or shift states. This is due to the fact that an SSA never examines the same character twice and must examine at least one of every m consecutive text characters. Therefore, the set of X states or Y states forms an acyclic graph. This implies that for $u \in X$, $\pi_u = \sum_{i \in Y} \pi_i \sum_{w \in W(i,u)} \mathrm{prob}(w)$ where W(i,u) is the set of paths from i to u composed only of probe states. The probability of getting to a shift state from u is 1 (because of the acyclic nature of X), so we could just as well have defined W(i,u) to be the set of paths from i through u back to a shift state. Alternatively, if we let W(i) be the set of all paths from i through probe states back to a shift state, then

$$\pi_u = \sum_{i \in Y} \pi_i \sum_{w \in W_i} \text{prob}(w) \cdot (1 \text{ if } u \text{ is on } w,\ 0 \text{ otherwise})$$

Thus,

$$\sum_{u \in X} \pi_u = \sum_{i \in Y} \pi_i \sum_{w \in W(i)} \text{prob}(w) \sum_{u \in X} (1 \text{ if } u \text{ is on } w,\ 0 \text{ otherwise}).$$

But

c_i=(the average number of probe states on the paths leaving i)

$$= \sum_{w \in W(i)} \text{prob}(w) \sum_{u \in X} (1 \text{ if } u \text{ is on } w,\ 0 \text{ otherwise}),$$

so $\sum_{u \in X} \pi_u = \sum_{i \in Y} \pi_i c_i$. The probability $\pi_i / \sum_{j \in Y} \pi_j$ is the invariant probability of being in state i given that the chain is in a shift state, which is ψ_i. Therefore,

$(\sum_{u \in X} \pi_u)/(\sum_{i \in Y} \pi_i) = \sum_{i \in Y} \psi_i c_i = c$. The formula for s follows by symmetry.◊

We have now come full circle. To compute the asymptotic average time for an SSA the following procedure is used.

1. Compute the invariant probabilities ϕ_u from the transitions of the SSA.

2. Compute $s = \sum_u \phi_u s_u$.

4. Let $c = 1/s$. Then $T(n) = cn+O(1)$.

Example Returning to our usual example (Figures III.1 and III.2), we have

u	***	*0*	*01	00*
ϕ_u	4/9	2/9	1/9	2/9
s_u	1	1/2	3	2

so $s=4/3$ and $c=3/4$ (as we discovered before).

We now have a method for computing the asymptotic average time used by a SSA. This also gives us a way of comparing two SSA's -- we compute the respective c's and the algorithm with the smaller value is the faster algorithm. (If there is a tie, we must go back to the generating functions to compute the constant d. However, this should usually be a term which can be ignored.) We can now find an optimal SSA for a given pattern. This will be treated in the next section.

III.C Optimal SSA's

We now address the problem of determining the fastest SSA for a particular pattern. We will ignore the lower order terms and simply optimize the parameter c in $T(n)=cn+O(1)$. There is an obvious algorithm for finding an optimal SSA -- construct all possible SSA's, analyze their average time, and find the fastest one. However, there are $\binom{m}{i}$ possible sates with i unknowns, and in each of these we could examine i possible positions. Thus, the total number of SSA's is roughly $\prod_{i=1}^{m} i^{\binom{m}{i}} > (m/2)^{2^{m-1}}$. For m=10, we get roughly 10^{689} possible SSA's. (Note that these are overestimates when one considers unreachable states, but the number of possible SSA's should still be astronomical.)

We present a procedure for finding optimal SSA's, which is certainly better than the brute force method, but is still only practical for short patterns. However, the technique has some interesting points in its development.

III.D Random SSA's

Our procedure for finding optimal SSA's involves defining a more general class of algorithms, designing a procedure for finding optimal algorithms in this class, and showing that the optimal algorithm found is, in fact, an SSA. The more general type of algorithm is a kind of probabilistic algorithm. A randomized SSA (RSSA) R operates just like an SSA, except that instead of examining a fixed position in each state, the RSSA examines a position i, selected randomly. Specifically, for each state u, the RSSA R defines a probability distribution R_u on the unknown positions of u. Then when R is in state u, the position examined, i, is chosen with probability $R_u(i)$. It is clear that an SSA is a special case of an RSSA in which $R_u(i)$ is always either 0 or 1.

Example Let $\Sigma=\{0,1\}$, p=001. In Figure III.6 let $R_{***}(3)=0.5$, $R_{***}(2)=0.5$, and all the other arrows have probability 1.

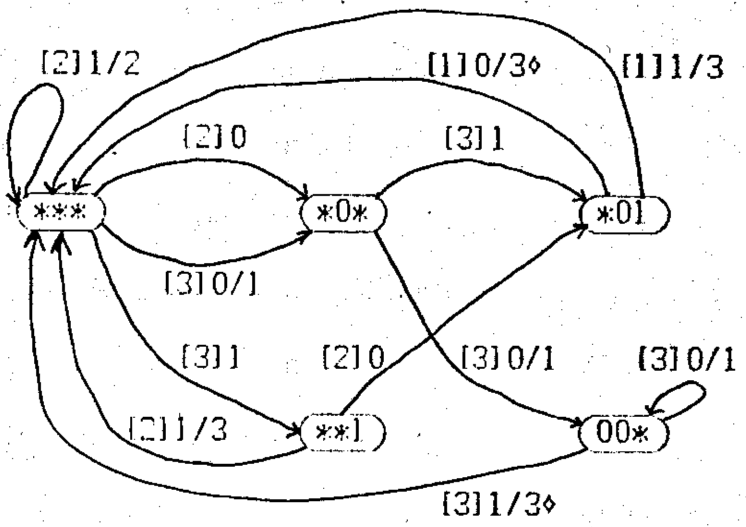


Figure III.6

The techniques of Markov analysis generalize easily to RSSA's. Only the transition probabilities change. The results of the Markov analysis hold also for RSSA's:

1. The average number of characters examined satisfies $T(n)=cn+d+o(1)$.
2. The asymptotic number of characters shifted per character examined s satisfies $s=1/c$.

The value s can be easily computed. Let ϕ_u be the invariant distribution for the RSSA, and let $s_u(i)$ be the average number of characters shifted when position i is examined in state u. then $s=\sum_u \sum_i \phi_u R_u(i) s_u(i)$. For the RSSA of Figure III.6, $s=4/3$ and $T(n)=3n/4+O(1)$.

To find an optimal RSSA, we must maximize s. We will set up a linear program to do this. A linear program (LP) is composed of a set of variables which can take on non-negative real values, a set of linear equations on those variables, called constraints, and a linear function of the variables, call the objective. A feasible solution to an LP is an assignment of values to the variables which satisfies the constraints. To solve a linear program means to find a feasible solution which maximizes the objective. The principal algorithm for solving LP's is the simplex method. Its time complexity is exponential in the size of the LP in the worst case, but for most practical problems, its time is very satisfactory. See [Lue] for the basic definitions and results about LP's and the simplex method.

We can state the problem of finding an optimal RSSA as

$$\text{maximize } s=\sum_u \sum_i \phi_u R_u(i) s_u(i)$$
$$\text{subject to } \sum_u \phi_u=1$$
$$\sum_i R_u(i)=1$$
$$\phi_u=\sum_v \phi_v \phi_{vu}$$
$$\phi_u,\ R_u(i)\geq 0.$$

However, this is not a linear program since $\phi_u R_u(i) s_u(i)$ is not a linear term. Fortunately, we can change variables and get a linear program. We substitute the variables ρ_{ui} for $\phi_u R_u(i)$ and get

$$\text{maximize } s=\sum_u \sum_i \rho_{ui} s_u(i)$$
$$\text{subject to } \sum_u \sum_i \rho_{ui}=1$$
$$\sum_i \rho_{ui}=\sum_v \sum_j \rho_{vj} \rho_{vju}$$
$$\rho_{ui}\geq 0$$

where ρ_{vju}=(the probability of going from v to u when position j is examined). (Note that $s_u(i)$ and ρ_{vju} are not variables, but are determined by the pattern and the probabilities of the characters.) Any feasible solution to the LP will define an RSSA with

$$\phi_u=\sum_i \rho_{ui}$$
$$R_u(i)=\rho_{ui}/\phi_u \text{ if } \phi_u>0.$$

If $\phi_u=0$ then $R_u(i)$ could be anything, but for definiteness we will define $R_u(i)$ to be 1 if i is the rightmost unknown of u, and 0 otherwise. Conversely, any RSSA defines a feasible solution to the LP with $\rho_{ui}=\phi_u R_u(i)$, since the equations which constrain ϕ_u are equivalent to the equations which constrain ρ_{ui}. Thus, we have the following procedure for finding an optimal RSSA:

1. Construct the LP as described above.
2. Solve the LP.
3. Compute $R_u(i)$ from the solution.

In general, LP's do not necessarily have any feasible solutions, and even if they do, the objective function may not be bounded. The LP's described above always have a feasible solution (use the KMP-SSA for instance), and the objective function is always bounded since the variables are bounded. Thus, there is always an optimal solution.

Example Let $\Sigma=\{0,1\}$, $p=01$, $P(0)=P(1)=0.5$. We will find an optmal RSSA. The general RSSA is shown below.

$$\begin{tikzcd}
\ast\ast \arrow[loop left, "{[1]1/1}"] \arrow[r, bend left, "{[2]1}"] \arrow[dr, bend right, "{[1]0}"'] \arrow[dr, "{[2]0/1}"] & \ast 1 \arrow[l, bend right=60, "{[1]0/2\diamond}"'] \arrow[l, bend right=90, "{[1]1/2}"'] \\
 & 0\ast \arrow[loop right, "{[2]0/1}"] \arrow[ul, bend left=60, "{[2]1/2\diamond}"]
\end{tikzcd}$$

Figure III.7

Let $u=\ast\ast$, $v=\ast 1$, $w=0\ast$. Then $\rho_{u1u}=\rho_{u1w}=\rho_{u2v}=\rho_{u2w}=\rho_{w2u}=\rho_{w2w}=0.5$, $\rho_{v1u}=1$, $s_u(1)=s_u(2)=0.5$, $s_v(1)=2$, $s_w(2)=1.5$. The LP is

maximize $s=\rho_{u1}/2+\rho_{u2}/2+2\rho_{v1}+3\rho_{w2}/2$

subject to $\rho_{u1}+\rho_{u2}+\rho_{v1}+\rho_{w2}=1$

$\rho_{u1}+\rho_{u2}=\rho_{u1}/2+\rho_{v1}+\rho_{w2}/2$

$\rho_{v1}=\rho_{u2}/2$

$\rho_{w2}=\rho_{u1}/2+\rho_{u2}/2+\rho_{w2}/2$

$\rho_{u1},\rho_{u2},\rho_{v1},\rho_{w2}\geq 0.$

The simplex method yields the optimal solution $\rho_{u1}=0$, $\rho_{u2}=0.4$, $\rho_{v1}=0.2$, $\rho_{w2}=0.4$, $s=1.2$. This implies that the following SSA is an optimal RSSA with $T(n)=5n/6+O(1)$.

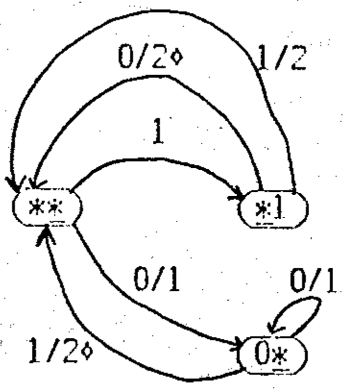


Figure III.8

III.E SSA's as Optimal RSSA's

The example above was a case where an SSA was an optimal RSSA. This was not a coincidence, as we show below.

The set of feasible solutions to an LP is a convex set. A feasible solution which corresponds to an extreme point of this convex set is called <u>basic</u>.

<u>Lemma III.4</u> The RSSA defined by a basic feasible solution to the LP is an SSA (that is, $R_u(i)$ is always 0 or 1).

<u>Proof</u> We will show the converse -- if ρ_{ui} is a feasible solution to the LP such that in the corresponding RSSA $0<R_u(k)<1$ for some u and k, then the feasible solution to the LP is not basic.

Assume that the LP has a feasible solution ρ_{ui} with $0<R_u(k)<1$ in the corresponding RSSA. Let $\lambda=R_u(k)$. Define two other RSSA's R^1 and R^0 such that

$$R^1_u(i) = \begin{cases} R_u(i) & \text{if } u\neq w \\ 0 & \text{if } u=w,\ i\neq k \\ 1 & \text{if } u=w,\ i=k \end{cases}$$

$$R^0_u(i) = \begin{cases} R_u(i) & \text{if } u\neq w \\ R_u(i)/(1-\lambda) & \text{if } u=w,\ i\neq k \\ 0 & \text{if } u=w,\ i=k \end{cases}$$

(Note that $\sum_{i\neq k} R_u(i)=1-\lambda$, so $\sum_i R^0_w(i)=1$ as required.) These RSSA's determine the corresponding ϕ^1_u, ϕ^0_u, ρ^1_{ui}, ρ^0_{ui}. Let $\alpha=1/((1-\lambda)\phi^1_u/\phi^0_u+1)$.

Note that in an RSSA, $\phi_u>0$ iff u is reachable from the state f entered when the pattern is found. In particular, $\phi_w>0$, because of the way that R is derived from ρ_{ui}, so w is reachable from f. Moreover, the shortest path from the found state to w does not contain w, so w is reachable in both R^1 and R^0, also. Therefore, $\phi^1_u>0$ and $\phi^0_u>0$ and α is well defined and not equal to 1. Also, if $\phi_u>0$ in R, then either

1. u is reachable from f without passing through w, in which case $\phi^1_u>0$ and $\phi^0_u>0$; or
2. u is reachable through w when position $i\neq k$ is examined, in which case $\phi^0_u>0$; or
3. u is reachable through w when position k is examined, in

which case $\phi^1_u>0$.

Thus, $\phi_u>0$ implies that at least one of ϕ^1_u or ϕ^0_u is non-zero. Conversely, if $\phi_u=0$ then $\phi^1_u=\phi^0_u=0$.

Now consider $\rho''_{ui}=\alpha\rho^1_{ui}+(1-\alpha)\rho^0_{ui}$ and $\phi''_u=\alpha\phi^1_u+(1-\alpha)\phi^0_u$. The values ρ''_{ui} are a solution to the LP, since they are a convex combination of two other solutions. We will show that $\rho''_{ui}=\rho_{ui}$ by showing that the derived $R''_u(i)=R_u(i)$. First, if $\phi_u=0$ then $\phi^1_u=\phi^0_u=0$, so $\phi''_u=0$ and $R''_u(i)=R_u(i)$ as explained in the derivation of R. So, assume that $\phi_u>0$, and so, from the note above, we must have at least one of $\phi^1_u>0$ or $\phi^0_u>0$, so $\phi''_u>0$. There are three cases.

1. If $u\neq w$ then $R''_u(i)=\rho''_{ui}/\phi''_u$

   $=(\alpha\rho^1_{ui}+(1-\alpha)\rho^0_{ui})/(\alpha\phi^1_u+(1-\alpha)\phi^0_u)$

   However, $R^1_u(i)=R^0_u(i)=R_u(i)$ if $u\neq w$, so

   $R''_u(i)=(\alpha\phi^1_u+(1-\alpha)\phi^0_u)R_u(i)/(\alpha\phi^1_u+(1-\alpha)\phi^0_u)=R_u(i)$.

2. If $u=w$, $i\neq k$, we have $R''_u(i)=(\alpha\rho^1_{wi}+(1-\alpha)\rho^0_{wi})/(\alpha\phi^1_u+(1-\alpha)\phi^0_u)$.

   But $\rho^1_{wi}=0$ if $i\neq k$, so

   $R''_u(i)=(1-\alpha)\phi^0_u R_u(i)/(1-\lambda)/(\alpha\phi^1_u+(1-\alpha)\phi^0_u)$

   but $(1-\alpha)\phi^0_u/(\alpha\phi^1_u+(1-\alpha)\phi^0_u)=1-\lambda$ (as can be verified with tedious algebra), so $R''_u(i)=R_u(i)$.

3. For $u=w$, $i=k$ we have

   $R''_u(k)=(\alpha\rho^1_{uk}+(1-\alpha)\rho^0_{uk})/(\alpha\phi^1_u+(1-\alpha)\phi^0_u)$

$=\alpha\phi_u^1/(\alpha\phi_u^1+(1-\alpha)\phi_u^0)=\lambda=R_u(k)$.

We conclude that $\rho''_{ui}=(\alpha\rho^1_{ui}+(1-\alpha)\rho^0_{ui})=\rho_{ui}$ Therefore, ρ_{ui} is a convex combination of ρ^1_{ui} and ρ^0_{ui}, which implies that ρ_{ui} is not an extreme point of the convex set, i.e. it is not a basic feasible solution.◊

With this lemma we can now use a fundamental theorem of linear programming to get our final result.

Theorem III.3 For every pattern there exists an SSA which is an optimal RSSA. Moreover, when the simplex method is used to solve the LP, the resulting RSSA is an SSA.

Proof The fundamental theorem of LP is that if there exists an optimal solution, then there exists an optimal solution which is basic. Moreover, the solutions found by the simplex method are always basic. Since the basic solutions correspond to SSA's, the theorem is proved.◊

Thus we have a procedure for finding optimal SSA's. In the worst case the simplex method takes time exponential in the number of variables (which in our case is $m2^{m-1}$). However, it usually is much faster. Also, the added structure of our particular LP may make it easier to solve using a specially designed algorithm. We leave this as an open problem.

IV. A New Algorithm

In this section we present a new alorithm for string searching. We can prove that this algorithm has good average and worst-case running time. Knuth [KMP] proposed the basic idea of the algorithm, but did not work out all the details.

The algorithm is similar to the Boyer-Moore algorithm in that it scans the characters in the window from right to left. However, instead of stopping when a mismatch is found, the algorithm continues scanning as long as the piece it has found occurs somewhere in the pattern. Intuitively, one may feel that this strategy is not as good as the Boyer-Moore strategy of shifting when a mismatch is found, but as we will show, it is possible with this strategy to "remember" all relevant characters without excessive overhead. As a result, this new algorithm never examines the same character twice, which aids the efficiency of the algorithm and its analysis. The algorithm is described below.

Algorithm B: Input: pattern p of length m and text t of length n.

1. Place a window of m characters over $t_1 \cdots t_m$ and set ℓ to 0.

2. At this point the window is in the following configuration:

$$\cdots\ t_{j-m}\ \underbrace{\overbrace{t_{j-m+1}\ \cdots\ t_{j-m+\ell}}^{\ell\text{ knowns}}\ \overbrace{t_{j-m+\ell+1}\ \cdots\ t_j}^{m-\ell\text{ unknowns}}}_{\text{window}}\ t_{j+1}\ \cdots$$

for some j. The algorithm examines t_j, t_{j-1}, $\cdots$, $t_{j-m+\ell+1}$, stopping if it is found that $t_i \cdots t_j$ is a substring of p, but $t_{i-1} \cdots t_j$ is not a substring of p, for some $i>j-m+\ell+1$. If such an i exists, go to 3a; otherwise, go to 3b.

3a. At this point it is known that $t_{i-1} \cdots t_j$ is not a substring of p. This implies that p does not occur in t at positions $j-m+1$, $j-m+2$, $\cdots$, $i-1$. Therefore, the window can be shifted at least to position i. In fact, the window can be shifted to position i′ defined as the least $i' \geq i$, such that $t_{i'} \cdots t_j = p_1 \cdots p_{j-i'+1}$. Go to 4.

3b. At this point all the characters in the window are known, but they might not match the pattern. This can be checked easily by rescanning the window. In any case, the algorithm shifts the window to the next possible occurrence of the pattern, a position i′, where i′ is the least $i' > j-m+1$ such that $t_{i'} \cdots t_j = p_1 \cdots p_{j-i'+1}$. Go to 4.

4. The window is actually shifted by setting $\ell=j-i'+1$ and $j=i'+m-1$. If $j>n$ the algorithm halts; otherwise, go back to 2.

The correctness of Algorithm B is due to

1. if the window is ever placed over an occurrence of p, then B finds it;
2. B only shifts the window past positions where p cannot occur.

Example We illustrate the operation of Algorithm B for p=abca and t=aabcaabcdabc. Initially we have:

$\ell=0$ $j=4$ $t=\underline{aabc}aabcdabc$

B finds that $t_2\cdots t_4$=abc is in p, but $t_1\cdots t_4$=aabc is not. B shifts to:

$\ell=3$ $j=5$ $t=a\underline{abca}abcdabc$

B finds that t_5=a is in p and determines that $t_2\cdots t_5$=p. B shifts to:

$\ell=1$ $j=8$ $t=\text{aabca}\underline{\text{abc}}\text{dabc}$

B finds that $t_6 \cdots t_8=abc$ is in p, but $t_5 \cdots t_8 \neq p$. B shifts to:

$\ell=3$ $j=9$ $t=\text{aabca}\underline{\text{abcd}}\text{abc}$

B finds that $t_9=d$ is not in p. B shifts to:

$\ell=0$ $j=13$ $t=\text{aabcaabcd}\underline{\text{abc\ }}$

and halts.

IV.A Implementation

The key to implementing Algorithm B efficiently is the use of two other algorithms, Weiner's algorithm [Wei,McC] and the Knuth-Morris-Pratt algorithm [KMP]. Weiner's algorithm can be used to preprocess the pattern from right to left to produce a compact position tree [AHU]. From this tree, B can answer the question "is $t_i \cdots t_j$ a substring of p?" in time which is constant for each character examined. KMP can be used to find "the least i' such that $t_{i'} \cdots t_j=p_1 \cdots p_{j-i'+1}$." (This is in fact what is done in the inner loop of KMP.) The time used by KMP is proportional to the number of characters processed.

We now present the algorithm in more detail.

Algorithm B: Input: $p \in \Sigma^m$ and $t \in \Sigma^n$.

```
1.  initialize the next array of KMP on the string p$
2.  initialize the position tree on $p from right to left
3.  j:=m  { the end of the window }
4.  ℓ:=0  { the number of known characters
                at the left of the window }
5.  while j≤n do
6.      r:=0  { the number of known characters
                at the right of the window }
7.      reset tree pointer to the top of the position tree
8.      found:=true
9.      while ℓ+r<m do
10.         x:=t_{j-r}  { get the next character }
11.         a_{m-r}:=x  { save it for later use by KMP }
12.         if x≠p_{m-r} then found := false
13.         if x does not come next in the position tree
                then go to 15
14.         r:=r+1  { end of loop }
15.     if found then print "found at j-m+1"
16.     if ℓ+r<m then ℓ:=0  { the left part can be ignored }
17.     for x:=a_{m-r+1},a_{m-r+2},...,a_m do ℓ:=KMP(ℓ,x)
18.     j:=j+m-ℓ  { end of loop }
```

where KMP(ℓ,x) is:

```
    $\ell' := \ell + 1$
    while $\ell' > 0 \wedge p_{\ell'} \neq x$ do $\ell' :=$ next($\ell'$)
    if $\ell' = m$ then $\ell' :=$ next($m+1$)$-1$
    return $\ell'$
```

This is not an optimal form for the algorithm. We leave the various optimizations to the programmer.

IV.B Time Analysis of Algorithm B

Theorem IV.1 Algorithm B can be implemented on a random access machine with uniform cost criterion to run in time $\Theta(m+k)$, where k is the number of times "$x:=t_{j-r}$" is executed.

Proof Steps 1 and 2 can be done in time $\Theta(m)$ as outlined in [KMP,Wei,McC]. Steps 3 and 4 take constant time. If $j \leq n$ then step 10 must be executed at least once. Therefore, the number of times that steps 5-8, 15-16, and 18 are executed is $O(k)$ and each step takes constant time. Clearly, the inner loop (steps 9-14) is executed k times. Since the position tree step can be done in constant time, the total time for the loop is $\Theta(k)$. Finally, the analysis in [KMP] shows that the total time for the loop of step 17 is $\Theta(k)$. Thus, the time for the entire algorithm is $\Theta(m+k)$.◊

With this theorem we have reduced the time analysis of Algorithm B to determining the number of times that the text is examined. We study the worst-case and average-case times below.

Theorem IV.2 Algorithm B never examines a text character more than once.

Proof We simply note that before step 18 of B, all text characters examined have been in positions $i \le j$ (the old j), and after step 18, no position $i \le j-m+\ell$ (the new j) will be examined in the future. However, (old j)=(new j)$-m+\ell$, so the characters previously examined cannot be examined again.◊

We can conclude from these two theorems that the worst-case time is $O(m+n)$. When combined with the lower bound of Rivest [Riv] we see that the worst-case time must be $\Theta(m+n)$.

IV.C Average Time

The analysis of average time is more complex. Let $T(n)$ be the average number of characters examined by Algorithm B on text strings of length n. We first analyze the simplest case.

Theorem IV.3 For $\Sigma=\{0,1\}$ and $P(0)=P(1)=0.5$, $T(n)=n\cdot \lg(m)/m+O(n/m)$. $(\lg(m)=\log_2(m))$

Proof In the sequel let $k=\lceil 2\lg(m)\rceil$ and assume $m\geq 18$, so $m\geq 2k$.

Let $T^{(\ell)}(n)$ be the average number of characters yet to be examined, given that Algorithm B has just shifted the window, there are ℓ known characters at the left of the window, and n is the total number of characters remaining, including the entire window. If $\ell\leq m-k$ then we say that B is in state 1, otherwise we say that B is in state 2. Let $T_1(n)=\max\limits_{\ell\leq m-k}\{T^{(\ell)}(n)\}$ and $T_2(n)=\max\limits_{\ell>m-k}\{T^{(\ell)}(n)\}$. Note that $T(n)=T^{(0)}(n)\leq T_1(n)$.

Claim $T^{(\ell)}(k+1)\geq T^{(\ell)}(k)$.

Proof B operates exactly the same on strings of length k and k+1 until it tries to examine t_{k+1}. In the former case, B halts, so its time must be less on these strings.◊

Corollaries $T_1(k+1)\geq T_1(k)$, $T_2(k+1)\geq T_2(k)$, and $T(k+1)\geq T(k)$. (We will use these facts implicitly in the rest of the proof.)◊

To prove the theorem we will prove by induction that $T_1(n) \le n(\lg(m)+d)/m$, and $T_2(n) \le n(\lg(m)+d)/m+2k$ for some constant d (independent of m and n).

If $n<m$ then $T_1(n)=T_2(n)=0$, so we have the basis for our induction. Assume, then, that $n \ge m$.

Consider the operation of B when it is in state 1. It examines positions in the window from right to left until the substring found does not occur in p, or until all positions in the window are known. Then the window is shifted. Let r be the number of characters examined, excluding the last character if the first stopping condition holds. Let ρ_r be the probability that r is this number.

Claim $\rho_r \le (m-r+1)/2^r$.

Proof The string of length r must occur somewhere in the pattern. There are 2^r strings of length r, but at most $m-r+1$ occur in p.◊

When the next shift occurs, there are four cases, a-d, which occur with probability ρ_a-ρ_d. The time $T_1(n)$ will be equal to $T_a+T_b+T_c+T_d$. We analyze these cases.

a. $r \le \lceil \lg(m) \rceil$. B must be in state 1 after the shift, and the shift must be by at least $m-r \ge m-\lceil \lg(m) \rceil$. Thus,

$T_a \le \rho_a \cdot (\lceil lg(m) \rceil + 1 + T_1(n-m+\lceil lg(m) \rceil))$.

b. $\lceil lg(m) \rceil < r \le \lceil 2\ lg(m) \rceil = k$. B must enter state 1 after the shift and shift the window by at least m-r. Thus,

$T_b \le \sum_{r=\lceil lg(m) \rceil+1}^{k} \rho_r \cdot (r+1+T_1(n-m+r))$.

c. r>k and B enters state 1. The shift is by at least k. Thus,

$T_c \le \rho_c \cdot (m+T_1(n-k))$.

d. B enters state 2. The shift is by at least 1. Thus,

$T_d \le \rho_d \cdot (m+T_2(n-1))$. Note that for either (c) or (d), r>k, so $(\rho_c+\rho_d) \le (m-k+1)/2^k = (m-\lceil 2\ lg(m) \rceil+1)/2^{\lceil 2\ lg(m) \rceil} < 1/m$.

Summing the cases, we find that

$$T_1(n) \le \rho_a(\lceil lg(m) \rceil+1+T_1(n-m+\lceil lg(m) \rceil)) + \sum_{r=\lceil lg(m) \rceil+1}^{k} \rho_r(r+1+T_1(n-m+r)) + \rho_c(m+T_1(n-k)) + \rho_d(m+T_2(n-1)).$$

From the induction hypothesis we get

$$T_1(n) \le \rho_a(\lceil lg(m) \rceil+1+(n-m+\lceil lg(m) \rceil)(lg(m)+d)/m)) + \sum_r \rho_r(r+1+(n-m+r)(lg(m)+d)/m) + \rho_c(m+(n-k)(lg(m)+d)/m) + \rho_d(m+(n-1)(lg(m)+d)/m+2k).$$

Using $\sum_{r=\lceil lg(m) \rceil+1}^{k} \rho_r = \rho_b$ and $\rho_a+\rho_b+\rho_c+\rho_d=1$ and some algebra, we get

$$T_1(n) \le n(lg(m)+d)/m + \rho_a(2-d+(lg(m)+1)(lg(m)+d)/m) + \rho_b(1-lg(m)-d) + (1+(lg(m)+d)/m) \cdot \sum_r \rho_r r + \rho_c(m-k(lg(m)+d)/m) + \rho_d(m-(lg(m)+d)/m+2k).$$

For $r > \lceil lg(m) \rceil$, $\rho_r \le (m-r+1)/2^r < m/2^r \le 1/2^{r-\lceil lg(m) \rceil}$, so with a change of index. $\sum_{r=\lceil lg(m)\rceil+1}^{k} \rho_r r$

$$=\sum_{r'=1}^{k-\lceil lg(m)\rceil} \rho_{r'+\lceil lg(m)\rceil}(r'+\lceil lg(m)\rceil)$$

$$=\rho_b\lceil lg(m)\rceil+\sum_{r'=1}^{k-\lceil lg(m)\rceil} r'/2^{r'}$$

$$\le\rho_b\lceil lg(m)\rceil+2.$$

Therefore,

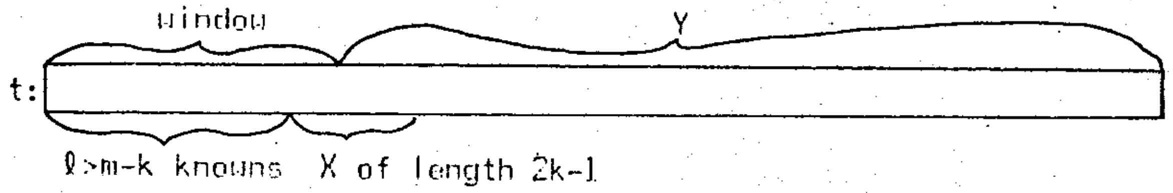
$$T_1(n)\le n(lg(m)+d)/m+\rho_a(2-d+(lg(m)+1)(lg(m)+d)/m)+$$
$$\rho_b(1-lg(m)-d+\lceil lg(m)\rceil+(lg(m)+1)(lg(m)+d)/m)+$$
$$2+2(lg(m)+d)/m+(\rho_c+\rho_d)m+\rho_d 2k$$
$$\le n(lg(m)+d)/m+(1-\rho_c-\rho_d)(2-d+(lg(m)+1)(lg(m)+d)/m)+$$
$$2+2(lg(m)+d)/m+(\rho_c+\rho_d)m+\rho_d 2k$$
$$\le n(lg(m)+d)/m+4-d+(lg(m)+3)(lg(m)+d)/m+$$
$$(\rho_c+\rho_d)(m+d-2-(lg(m)+1)(lg(m)+d)/m)+\rho_d 2k.$$

If we use $\rho_d\le\rho_c+\rho_d\le 1/m$ we get

$$T_1(n)\le n(lg(m)+d)/m+5-d+((lg(m)+3)(lg(m)+d)+2\lceil 2\ lg(m)\rceil+d-2)/m.$$

Since $lg(m)/m$ decreases for $m\ge e=2.718\cdots$ and $lg^2(m)/m$ decreases for $m\ge e^2=7.389\cdots$, the last term above decreases for $m\ge e^2$, and goes to 0 as m goes to ∞. Thus, for $m\ge e^2$ and suitably large d, $T_1(n)\le n(lg(m)+d)/m$. For $m\ge 18$, $d=14$ is suitably large.

Now consider the case where B is in state 2. We wish to show that $T_2(n)\le n(lg(m)+d)/m+2k$ ($k=\lceil 2\ lg(m)\rceil$). We diagram the situation below



Let X be the region of length 2k-1 immediately following the knowns as diagrammed above, and let Y be the remainder of the text. (If $n<\ell+2k$ then $T_2(n)\le 2k\le n(\lg(m)+d)/m+2k$.) B will examine positions in X and eventually shift into Y. Let ρ be the probability of being in state 2 when the first probe is made into Y. Then $T_2(n)\le 2k+(1-\rho)T_1(n-k)+\rho T_2(n-k)$. If B is in state 2, then, since $m\ge 2k$, there exists a substring of length k in X which occurs in the last 2k positions of p. The probability of this does not exceed k^2/m^2, so $\rho\le k^2/m^2$. Therefore,

$$\begin{aligned}T_2(n)&\le 2k+(1-\rho)(n-k)(\lg(m)+d)/m+\rho((n-k)(\lg(m)+d)/m+2k)\\&\le n(\lg(m)+d)/m+2k-k(\lg(m)+d)/m+\rho 2k\\&\le n(\lg(m)+d)/m+2k+2k^3/m^2-k(\lg(m)+d)/m\\&\le n(\lg(m)+d)/m+2k+k(2k^2-m(\lg(m)+d))/m^2\end{aligned}$$

The final term above is negative for $m\ge 18$, $d=14$, so $T_2(n)\le n(\lg(m)+14)/m+2k$.

We have shown that $T(n)\le n(\lg(m)+14)/m$, for $m\ge 18$. We also know that $T(n)\le n$ for all m. Then, since for $m<18$, $(\lg(m)+14)/m>1$, we have $T(n)\le n(\lg(m)+14)/m$ for all m.◊

We have proved an upper bound on the average running time for Algorithm B. We can use Yao's result [Yao] to get a lower bound which is within a constant factor of the upper bound. In fact, if one analyzes the constants in Yao's bound, the bound works out to $n \cdot \lg(m)/4m$. So our upper bound is only four times the lower bound, and the lower bound is for the best case! However, Yao's constant of 1/4 holds for $m > 10^{30}$, which means that the bounds are not quite so close.

We can also get an explicit example for which Algorithm B takes nearly the upper bound.

Theorem IV.4 Let $\Sigma = \{0,1\}$. For all m there exists a pattern $p \in \Sigma^m$ for which B examines $n \cdot \lg(m)/m - O(n/m)$ characters in all text strings.

Proof For every j there exists a string $p' \in \Sigma^k$, where $k = 2^j + j - 1$, such that all elements of Σ^j occur as substrings of p'. Such a string is called a DeBruijn sequence. See [Hal] for a proof of the existence of such sequences. For a given $m \geq 4$, let $j = \lfloor \lg(m) \rfloor - 1$, and let $p = 0^{m-k} p'$, where k and p' are defined as above. Note that $k = 2^j + j - 1 = 2^{\lfloor \lg(m) \rfloor - 1} + \lfloor \lg(m) \rfloor - 2 < m$.

Let f(n) be the best-case time for B on p. B scans backwards from t_m as long as it finds a substring of p. By construction, every string of length $\lfloor \lg(m) \rfloor - 1$ occurs in p, so B must examine at least this many characters. When it shifts the window it cannot shift by more than m characters, so $f(n) \geq \lfloor \lg(m) \rfloor - 1 + f(n-m)$. Thus,

$$f(n) \geq \lfloor n/m \rfloor (\lfloor \lg(m) \rfloor - 1) \geq n \cdot \lg(m) - O(n/m).\diamond$$

In summary we conclude the following.

Theorem IV.5 For $\Sigma = \{0,1\}$, $P(0)=P(1)=0.5$, and for all patterns $p \in \Sigma^m$, the average running time for Algorithm B on text strings of length n is $O(n \cdot \lg(m)/m)$, and for most patterns B has average running time $\Theta(n \cdot \lg(m)/m)$.$\diamond$

These results generalize to larger alphabets and different probability distributions.

Theorem IV.6 Let $q = 1/P(\sigma)$ where σ has the highest probability of any element of Σ. Then for any $p \in \Sigma^m$, the average number of characters examined by B on text strings of length n is no more than $n \cdot \log_q(m)/m + O(n/m)$, and for most patterns it is $\Theta(n \cdot \log_{|\Sigma|}(m)/m)$.

Proof A generalization of the proofs of Theorems IV.1-5.$\diamond$

IV.D Notes on Algorithm B

It should be noted that Algorithm B, when specialized for a pattern p, is not an SSA, since it does not shift when a mismatch occurs. As a result it sometimes will examine a character that it does not have to examine. Correcting this deficiency would produce an algorithm that examines fewer characters. However, it would still examine $n \cdot \log_{|\Sigma|}(m)/m+O(n/m)$ characters on the average; the improvement would only be in low order terms. Also, even though Algorithm B does not yield SSA's, its average number of characters examined still is of the form $T(n)=cn+d+o(1)$, for some c and d, since it does yield finite-state machines for which our Markov analysis holds.

Due to time limitations in the preparation of this thesis, there is a significant gap in the presentation of Algorithm B. We have not actually programmed the algorithm and empirically compared its speed with that of known algorithms. In this section we present some thoughts on issues that might arise in the actual programming.

Algorithm B is very simple except for the uses of the Knuth-Morris-Pratt algorithm and Weiner's algorithm. KMP is well described in [KMP] and programming it is straight forward and efficient. Weiner's algorithm [Wei] on the other hand, is not so transparent. McCreight's presentation [McC] is clearer, but it is still a challenge to wade through. It appears, though, that except for one problem, a program for McCreight's algorithm will be complex, but efficient. The one problem area is due to the fact that the position tree built by the algorithm can have nodes with up to $|\Sigma|$ branches. Thus, if one is not careful in programming, the time to determine whether a string occurs in the pattern can be large.

The key to efficient implementation is the underlying structure of the position tree. That is, it must be possible to determine efficiently, given a node x of the tree and a symbol $\sigma \epsilon \Sigma$, whether there is a branch labeled σ from node x. The simplest implementation of the position tree as a binary tree [Kn1] would yield a program with worst-case time $O(|\Sigma| \cdot n+m)$ and average time $O(|\Sigma| \cdot n \cdot \log(m)/m+m)$ (for the worst pattern). This can be improved using binary search trees [Kn2] to worst-case $O(\log(|\Sigma|) \cdot (n+m))$ and average $O(\log(|\Sigma|) \cdot n \cdot \log(m)/m+\log(|\Sigma|) \cdot m)$ A space inefficient scheme where each node is represented by an array of size $|\Sigma|$ yields time of worst-case $O(n+|\Sigma| \cdot m)$ and average $O(n \cdot \log(m)/m+|\Sigma| \cdot m)$. Hashing is probably the best scheme to use in implementing position trees (as suggested in [McC]).

using an analogous procedure AC instead of KMP. The only additional change is that the algorithm must keep track of the state of the Aho-Corasick automaton as well as ℓ, between shifts.

If the patterns are not of equal length, the situation is more complex. The following is a sketch of the algorithm.

1. Set $\ell=0$, $k=1$, and $s=$ {the initial state of the Aho-Corasick automaton (AC)}.
2. Let $k'=k+f(s)-1$, where $f(s)$ will be defined later. Examine $t_{k'}, t_{k'-1}, \cdots, t_{k+\ell}$, stopping if $t_i \cdots t_{k'}$ occurs in some pattern, but $t_{i-1} \cdots t_{k'}$ does not occur in any pattern. If such an i exists, go to 3a; otherwise, go to 3b.

3a. Determine the least $i' \geq i$ such that $t_{i'} \cdots t_{k'} = p^x_1 \cdots p^x_{k'-i'+1}$ for some pattern p^x. Do this using AC, setting s to the resulting state. Go to 4.

3b. Determine the least $i' \geq k$ such that $t_{i'} \cdots t_{k'} = p^x_1 \cdots p^x_{k'-i'+1}$, for some p^x. Do this using AC, setting s to the resulting state. As a by-product, any occurrences of the patterns completely within $t_k \cdots t_{k'}$ can be found. Go to 4.

4. Set $\ell = k'-i'+1$, and if $k+f(s)-1 \leq n$, then go to 2.

For this algorithm to work, we must define f(s) correctly. We could define $f(s)=\ell+1$, in which case every character would be examined, so the algorithm would be correct. (Note that ℓ is uniquely determined by s, so this definition makes sense.) Alternatively, if m is the length of the shortest pattern, then we could define $f(s)=\max\{m,\ell+1\}$. This would guarantee that no occurrences would be skipped.

The best value for f(s) is found by considering the least number of characters, d, that can be appended to $t_k \cdots t_{k+\ell-1}$ to get an occurrence of the pattern. Then f(s) can be defined as $\ell+d$. In terms of the Aho-Corasick automaton, d can be expressed simply as the length of the shortest path from s to an accepting state. This can be computed from the automaton by starting with the accepting states, and doing a breadth-first search of all the other states. The details of this are left to the reader.

The time for this algorithm can be expressed in terms of the length of the shortest pattern m, the sum of the lengths of all the patterns M, and the length of the text n. As in the single pattern case, no text character is examined more than once, the actual time per character examined is constant, and the preprocessing takes time $\Theta(M)$. Therefore, the worst-case actual time is $\Theta(n+M)$. Analysis similar to the single pattern case shows that the average number of characters examined is no more than $n\cdot\log_q(M)/m+O(n/m)$ if $m\geq 2\cdot\log_q(M)$, where $q=1/P(\sigma)$ and σ is the most probable element of Σ. The average time is then $O(n\cdot\log_q(M)/m+M)$. The only known lower bound on the average number of characters examined is the Yao lower bound [Yao] for one pattern, $\Omega(n\cdot\log_{|\Sigma|}(m)/m)$. However, it appears that for all patterns of equal length, Yao's technique yields a lower bound of $\Omega(n\cdot\log_{|\Sigma|}(M)/m)$, but the details of this have not been worked out.

As in the single pattern case, we have not actually programmed this algorithm. Nevertheless, we believe that it can be implemented efficiently and is an improvement over the Aho-Corasick algorithm. It appears to be especially useful for the bibliographic search application described in [A&C], where the patterns should be reasonably long. The algorithm also has the nice property that the more explicit one is in specifying the queries (i.e. the longer the patterns), the faster the algorithm will run.

V. The Case of the All-Zero Pattern

In this section we study the simplest special case of string-searching -- finding $p=0^m$ in $t\in\{0,1\}^n$. While the simplicity of this problem might make it uninteresting, the strong results and the techniques we develop justify its study.

We will show that Algorithm B when specialized to $p=0^m$ is optimal even when considering the class of decision tree algorithms. A consequence of this is that this is a problem where increasing the memory size or program complexity cannot produce a faster algorithm. Conversely, restricting memory or program size does lead to slower algorithms. We will study this time-space trade-off.

V.A Certificates

The case of $p=0^m$ imposes a great deal of structure on the certificates. This structure is summarized in:

Theorem V.1 Let $u\in\{0,1,*\}^n$ be a certificate for $p=0^m$. then the string $1u1$ can be partitioned into segments $1u1=u^1u^2\cdots u^k$ such that each segment u^i is either a positive segment -- $u^i\in 0^m0^*$, or a negative segment -- u^i starts and ends with 1 and every substring of length m contains at least one 1.

Proof Follows easily from the definition of certificate.◊

Corollary V.1a Any substring of u of length at least m not containing a 1 must be all 0's.◊

Corollary V.1b If $t_j=1$ and the total number of 0's immediately preceding and following t_j is at least m-1, then $u_j=1$ in all certificates u for t.◊

V.B Algorithm B for 0^m.

When specialized to the case of $p=0^m$, Algorithm B has a very simple form. It was used in [Riv] to show an algorithm with good worst-case behavior. It can be stated as:

```
Algorithm B0:  Input: m and $t \in \Sigma^n$
    $\ell$:=0;     { the number of 0's found
                     on the left of the window }
    r:=0;     { the number of 0's found on the right }
    j:=m;     { the end of the current
                     text-matching position }
    while $j \le n$ do
while $\ell+r<m \wedge t_{j-r}=0$ do $r:=r+1$
if $\ell+r=m$ then { found at $j-m+1$ }
   $j:=j+1$;
   $\ell:=m-1$;
   $r:=0$;
else { not found at $j-m+1$ through $j-r$ }
   $j:=j-r+m$;
   $\ell:=r$;
   $r:=0$;
```

The algorithm starts at position m of the text and scans backwards until either a 1 is found or until m 0's are found. In the former case the algorithm can skip ahead to m places past the 1 without missing any occurrences of the pattern. In the latter case the pattern has been found and the algorithm moves ahead by one searching for the next occurrence. In either case it restarts the scan, remembering the relevant 0's already found.

This algorithm is optimal over the class of decision tree algorithms no matter what the probabilities are of 0 and 1. The essence of the algorithm's "goodness" is summarized in the following:

This would yield worst-case time $O(|\Sigma|\cdot n+|\Sigma|\cdot m)$ and average time $O(n\cdot \log(m)/m+m)$. It may be possible to improve the worst-case time by choosing a hashing scheme particularly suited to this application. (The restricted domain of the hash function makes this likely.)

We believe that Algorithm B can be implemented efficiently, yielding a practical algorithm. Whether it is better than the Boyer-Moore algorithm remains to be seen.

IV.E Multiple Patterns

Algorithm B can be generalized to handle the problem of finding any of a set of patterns. This problem was studied by Aho and Corasick [A&C], who generalized the idea of KMP to produce a fast algorithm. For this problem the input is a list of patterns $p^1, p^2, \cdots, p^j$ and text t, and the algorithm must find all occurrences of all the patterns in t. The generalization of Algorithm B to handle multiple patterns is simple if $|p^1|=\cdots=|p^j|=m$. As preprocessing, the algorithm builds a position tree for $\$p^1\$p^2\cdots\$p^j$ from right to left and builds the Aho-Corasick automaton [A&C] for the patterns. The algorithm then shifts a window of length m over the text, just as before.

Theorem V.2 For all $t\in\{0,1\}^n$, every certificate for t must have at least as many 1's as the certificate found by BO.

Proof Let t be some text string with BO-certificate α and another certificate β such that β has fewer 1's then α. We will derive a contradiction.

If β has no 1's then either $n<m$ or $t=0^n$. In either case α has no 1's either, so β cannot have fewer 1's. Also, if α has no 1's then β cannot have fewer. So assume that α has 1's at positions $i_1<i_2<\cdots<i_k$ and β has 1's at positions $j_1<j_2<\cdots<j_\ell$. For notational convenience, let $i_0=j_0=0$ and $i_{k+1}=j_{\ell+1}=n+1$.

Lemma If $i_u\geq j_v$ then $i_{u+1}\geq j_{v+1}$.

Proof If $i_u\geq j_{v+1}$ then $i_{u+1}>i_u\geq j_{v+1}$ and we are done. So assume that $j_{v+1}>i_u\geq j_v$. If $j_{v+1}-j_v>m$ then by Corollary V.1a, $\beta_i=0=t_i$ for $j_v<i<j_{v+1}$, and so i_{u+1} cannot be less that j_{v+1}. If $j_{v+1}-j_v\leq m$ then consider the operation of BO after finding $t_{i_u}=1$. It starts at i_u+m and scans backwards looking for a 1. Since $i_u+m\geq j_v+m\geq j_{v+1}$, it will find a 1 at some position $b\geq j_{v+1}$, and never look to the left of b again. Thus, $i_{u+1}=b\geq j_{v+1}$.◊

Returning to the theorem, we can partition the set $\{0,1,\cdots,n\}$ into $\ell+1$ subsets $[j_0,j_1)\cup[j_1,j_2)\cup\cdots\cup[j_\ell,j_{\ell+1})$. The elements of $\{i_0,\cdots,i_k\}$ fall into these subsets. Since we are assuming that $\ell<k$, there must be some pair i_u,i_{u+1} in the same

set $[j_v, j_{v+1})$, by the pigeon-hole principle. This implies that $j_v \le i_u < i_{u+1} < j_{v+1}$ which contradicts the lemma. Therefore, β cannot have fewer 1's. The BO-certificate must have the least number of 1's of all certificates.◊

In the next section we will show that the total number of 0's found in all text strings is proportional to the total number of 1's. Thus, since BO minimizes the number of 1's, it also minimizes the number of 0's and the total.

V.C The Decision Tree Lemma

We have defined the average time for a decision tree algorithm A to be $T_A = \sum_t \text{prob}(t) \cdot (\text{number of characters of } t$ examined by A$)$. We can view this another way; at every node x of the decision tree we "charge" a cost of 1 to every t which takes A to that node sometime in its computation. Then the average time is the average total cost charged to the $t \in \Sigma^n$. Note that exactly half the t which take A to x take A to the 1-child of x. Moreover, if $P(0)=P(1)$ then each child of x accounts for the same cost in T_A. Therefore, another way to compute the average cost is to charge 2 to the text strings in which a 1 is found at x. Thus, when $P(0)=P(1)$, $T_A=\sum_t \text{prob}(t)\cdot 2\cdot \{\text{number of 1's in t found by A}\}$. We generalize and prove this as follows.

<u>Lemma V.1</u> $T_A=\sum_t \text{prob}(t)\cdot \{\text{number of 1's in t found by A}\}/P(1)$.

<u>Proof</u> For every node x in the tree A define

$S(x)=\{t\in\Sigma^n \mid t \text{ takes A to x}\}$

$S_1(x)=\{t\in\Sigma^n \mid t \text{ takes A to x and a 1 is found in t at x}\}$

$S_0(x)=\{t\in\Sigma^n \mid t \text{ takes A to x and a 0 is found in t at x}\}$

and define $S'(t)=\{x \mid t\in S(x)\}$, with S'_1 and S'_0 defined analogously. Then

$$\begin{aligned}T_A &= \sum_t \text{prob}(t)\,|S'(t)| \\ &= \sum_t \text{prob}(t) \sum_{x\in S'(t)} 1 \\ &= \sum_x \sum_{t\in S(x)} \text{prob}(t) \\ &= \sum_x \Big(\sum_{t\in S_1(x)} \text{prob}(t) + \sum_{t'\in S_0(x)} \text{prob}(t')\Big)\end{aligned}$$

Note that for every $t\in S_1(x)$ there exists a unique $t'\in S_0(x)$ which differs from t in exactly one position -- that examined at x. (We assume, here, that the decision tree never examines the same position twice.) That is, for some j, $t_j=1$, $t'_j=0$, and $t_i=t'_i$ if $i\neq j$. Therefore

$$\begin{aligned}\text{prob}(t') &= \text{prob}(t'_j)\prod_{i\neq j}\text{prob}(t'_i) \\ &= (\text{prob}(t'_j)/\text{prob}(t_j))\cdot(\text{prob}(t_j)\prod_{i\neq j}\text{prob}(t'_i)) \\ &= (P(0)/P(1))\cdot\text{prob}(t)\end{aligned}$$

Thus

$$\begin{aligned} T_A &= \sum_x \sum_{t \in S_1(x)} (\text{prob}(t) + (P(0)/P(1)) \cdot \text{prob}(t)) \\ &= (1+P(0)/P(1)) \sum_x \sum_{t \in S_1(x)} \text{prob}(t) \\ &= (1/P(1)) \sum_t \text{prob}(t) \sum_{x \in S'_1(t)} 1 \\ &= \sum_t \text{prob}(t) \cdot |\text{number of 1's found in } t \text{ by } A| / P(1). \diamond \end{aligned}$$

This lemma holds when $|\Sigma|>2$ and, of course, when any $\sigma \in \Sigma$ is substitued for 1 in its statement.

V.D Optimality of BO

We now prove our principal theorem from the BO theorem and the decision tree lemma.

Theorem V.3 BO is average-time optimal over the class of decision tree algorithms for any probabilities, P(0) and P(1), where $0<P(0)<1$.

Proof We have shown that for any algorithm A

$T_A = \sum_t \text{prob}(t) \cdot |\text{number of 1's found in } t \text{ by } A|/P(1)$. We have also shown that for any A and all $t \in \Sigma^n$

$$\begin{aligned} &|\text{the number of 1's found in } t \text{ by } BO| \\ &\quad \leq |\text{the number of 1's found in } t \text{ by } A| \end{aligned}$$

Therefore, $T_{BO} \le T_A$.◊

This theorem has some interesting consequences in terms of space-time-program-size trade-offs. When we think of BO in the form af an SSA we see that it uses m cells of random access memory. This theorem shows that having more memory does not help. However, we will see that having less memory forces any algorithm to be slower. In addition, increasing the complexity of the program -- even using a decision tree -- cannot produce a faster algorithm.

V.E Other Optimal Algorithms

We have shown that BO is an optimal algorithm, but not that it is the only optimal algorithm. In fact, there are others. For example, running BO from right to left instead of left to right will have the same average-time behavior. However, we will show that all optimal algorithms are just combinations of these two strategies.

Consider some intermediate point in the operation of an algorithm and the partial certificate u computed at that point. We define:

<u>Definition</u> A <u>forced position</u> j is a text position such that $u_j=*$ and one of the following hold:

(1) $u_{j-1}=1$ and for $j<i<j+m$, $u_i=0$.

(2) $u_{j+1}=1$ and for $j-m<i<j$, $u_i=0$.

In either case, t_j must be examined sometime in the future by the algorithm, but its value has no affect on whether any other position need be examined.

<u>Definition</u> Let k be the least number such that

(1) k is not a forced position,

(2) for all i, $k\le i<k+m$, $u_i=0$ or $u_i=*$, and

(3) for some i, $k\le i<k+m$, $u_i=*$.

That is, k is the leftmost position where the pattern may or may not occur, but which is not forced. The <u>left window</u> of t is the substring $t_k\cdots t_{k+m-1}$. The <u>right window</u> is defined analogously.

<u>Definition</u> A position is an <u>ending position</u> if it is either in both windows or is adjacent to both windows and surrounded by at least m-1 0's.

<u>Definition</u> A probe j is <u>acceptable</u> if it is one of:

(1) a forced position

(2) the rightmost unknown in the left window

(3) the leftmost unknown in the right window

(4) an ending position.

Theorem V.4 An algorithm A is optimal iff every probe it makes is acceptable.

Proof (if) This is similar to the BO Theorem and is left to the reader.

(only if) What we will show is that if A makes an unacceptable probe then there is some t in which A finds more than the minimal number of 1's. We will use an adversary argument.

Note that every unacceptable probe either is between the windows or is in only the left (right) window but is not the rightmost (leftmost) unknown.

Assume A makes an unacceptable probe and consider the first (highest in the decision tree) such probe. Assume it is to position j. The adversary will answer that $t_j=1$, but also answer that $t_i=t_k=1$ for some i and k, with $i<j<k$ and $k-i<m$. Moreover, the adversary will force both t_i and t_k to be examined, so that examining t_j will be unnecessary.

If j falls in between the windows and is not adjacent to either window, then we must have $u_{j-1}=u_j=u_{j+1}=*$. (If either u_{j-1} or u_{j+1} were known, they would have been found by an unacceptable probe, but we assume that j is the first one.) In this case our adversary answers $t_{j-1}=t_j=t_{j+1}=1$ and every other unknown is 0. Since everything else is 0, A must probe both t_{j-1} and t_{j+1} at some point. The resulting certificate will have $u_{j-1}=u_j=u_{j+1}=1$. Thus, $u_j=1$ will be redundant and A will have found more than the minimum number of 1's.

If the unacceptable probe is between the windows, but adjacent to, say, the left window, then we have $u_j=u_{j+1}=*$ and $u_i=*$ for some i, $j-m<i<j$. In this case the adversary says $t_i=t_j=t_{j+1}=1$ and everything else 0. Again, both t_i and t_{j+1} will be examined, $u_j=1$ will be redundant, and the number of 1's will not be minimal.

Finally, if the unacceptable probe is, say, in the left window but not the rightmost unknown in the left window, then we have $u_{k-1}=1$, $u_j=u_i=*$, where k is the position of the left window and i is the rightmost unknown in the window. As before, the adversary says $t_j=t_i=1$ and everything else 0. The algorithm must examine t_i, $u_j=1$ will be redundant, and A will not find the minimum number of 1's.

The last case, where j is adjacent to both windows but not surrounded by m-1 0's is similar.

Since A does not find the minimum number of 1's for some t, it must find more total 1's in all t than B0. Therefore, by the decision tree lemma, it must not be optimal.◊

V.F First-Occurrence Algorithms

We get similar but simpler results for algorithms which halt after finding the first occurrence of 0^m.

Theorem V.5 BO is an optimal first-occurrence algorithm for $p=0^m$.

Proof First, note that the decision tree lemma applies in this case also. A modification of the BO theorem shows that BO finds the minimum number of 1's in any t among first-occurrence algorithms. Therefore, BO is optimal.◊

For the first-occurrence case we define:

Definition A probe j is acceptable if j is a forced position, or if there is no forced position and j is the rightmost unknown in the left window or an ending position.

Theorem V.6 A first-occurrence algorithm is optimal iff every probe is acceptable.

Proof The proof is similar to the all-occurrences case. In the case where the unacceptable probe is inside the left window, the adversary answers as before. In the cases where either there is a forced position and the probe is to some other position or the probe is to the right of the window, answering $t_j=1$ and all else 0 puts the first occurrence of the pattern to the left of j. Therefore, $u_j=1$ is redundant.◊

V.G Time Analysis of BO

For the record, we would like to precisely analyze the time taken by BO for $\Sigma=\{0,1\}$, $P(0)=P(1)=0.5$, $p=0^m$. For any specific m we can use the techniques of Chapter 3 to compute T(n), but we have not been able to generalize the results for all m in a simple way. All we can say is that $T(n)=cn+d+o(1)$, for some c and d. In this section we will derive bounds on c in terms of m.

Due to the optimality of BO, we can prove certain uniformity conditions on its running time. Let $T_\ell(n)$ be the time used by BO given that it has found ℓ 0's at the left of the window, and the total number of characters left, including the window, is n.

Lemma V.2

1. $T_\ell(n+1) \geq T_\ell(n)$
2. $T_{\ell+k}(n+k) \geq T_\ell(n)$ for $\ell+k<m$
3. $T_{\ell+k}(n+k) \leq T_\ell(n)+2$ for $\ell+k<m$
4. $T_{\ell+1}(n) \leq T_\ell(n)$ for $\ell+1<m$.

Proof

1. This was proved in general for Algorithm B in Chapter 4.

2. An algorithm for strings of length n+k with ℓ+k leading 0's can be converted into an algorithm for strings of length n with ℓ leading 0's, simply by appending k leading 0's to t. Then, since B0 is optimal, we must have $T_{\ell+k}(n+k) \geq T_\ell(n)$.

3. We will construct an algorithm for finding 0^m in strings of length n+k with ℓ+k known 0's at the left, that has average time $\leq T_\ell(n)+2$. Then, since B0 is optimal, the result is proved. The algorithm operates just like B0, starting at t_{k+1} with ℓ 0's known. When it has finished it has found all occurrences of p in t except for possible occurrences starting at $t_1, t_2, \cdots, t_{j-m}$ where j is the leftmost position examined by the algorithm. If $j=k+\ell+1$ then it is done. If $k+\ell+1<j<m$, then $t_j=1$ and it is done in this case also. If $j>m$, the algorithm examines $t_{k+\ell+1}, \cdots, t_{j-1}$ stopping when it finds a 1. These positions determine if p occurs at $t_1, \cdots, t_{j-m}$. The expected number of characters

examined before finding a 1 is $1\cdot 1/2+2\cdot 1/4+3\cdot 1/8+\cdots<2$. Thus, $T_{\ell+k}(n+k)\le T_\ell(n)+2$.

4. Compare the operation of BO when ℓ 0's are known and when $\ell+1$ 0's are known. They are exactly the same if a 1 is found in $t_{\ell+2}\cdots t_m$. The probability that no 1 is found in $t_{\ell+2}\cdots t_m$ is $2^{-(m-\ell-1)}$. Thus, the difference in average times is

$$\begin{aligned}T_\ell(n)-T_{\ell+1}(n)&=2^{-(m-\ell)}(m-\ell+T_{m-\ell-1}(n-\ell-1))+2^{-(m-\ell)}(m-\ell+T_{m-1}(n-1))\\&\quad-2^{-(m-\ell-1)}(m-\ell-1+T_{m-1}(n-1))\\&=2^{-(m-\ell-1)}-2^{-(m-\ell)}(T_{m-1}(n-1)-T_{m-\ell-1}(n-\ell-1))\end{aligned}$$

If $\ell=0$, then $T_{m-1}(n-1)-T_{m-\ell-1}(n-\ell-1)=0$ and the result holds. If $\ell>0$, then $T_{m-1}(n-1)\le T_{m-\ell-1}(n-\ell-1)+2$ [from 3], so $T_\ell(n)-T_{\ell+1}(n)\ge 2^{-(m-\ell-1)}-2^{-(m-\ell)}\cdot 2\ge 0$.◊

We can now bound c in terms of m.

<u>Theorem V.7</u> The average number of characters examined by BO for $p=0^m$ is $T(n)=cn+d+o(1)$, where c and d are constants with $(2-2^{1-m})/m\le c\le(2-2^{1-m})/(m-1+2^{1-m})$.

<u>Proof</u> We know that for any positive ϵ, there exists n_0 such that $cn+d-\epsilon\le T(n)\le cn+d+\epsilon$ for all $n\ge n_0$ from the form of T(n). We will assume that $n\ge n_0+m$.

For the lower bound we have

$$cn+d \geq T(n)-\epsilon$$
$$\geq \sum_{i=1}^{m} 2^{-i}(i+T_{i-1}(n-m+i-1))+2^{-m}(m+T_{m-1}(n-1))-\epsilon$$
$$\geq \sum_{i} 2^{-i}(i+T(n-m))+2^{-m}(m+T(n-m))-\epsilon \quad \text{[from Lemma V.2 2]}$$
$$\geq 2-2^{1-m}+T(n-m)-\epsilon$$
$$\geq 2-2^{1-m}+cn-cm+d-2\epsilon \quad \text{[since } n-m \geq n_0\text{]}$$

So $c \geq (2-2^{1-m}-2\epsilon)/m$. Since ϵ can be arbitrarily small, we conclude that $c \geq (2-2^{1-m})/m$.

For the upper bound we use

$$cn+d \leq T(n)+\epsilon$$
$$\leq \sum_{i=1}^{m} 2^{-i}(i+T_{i-1}(n-m+i-1))+2^{-m}(m+T_{m-1}(n-1))+\epsilon$$
$$\leq 2-2^{1-m}+\sum_{i} 2^{-i}T(n-m+i-1)+2^{-m}T(n-1)+\epsilon \quad \text{[from Lemma V.2 4]}$$
$$\leq 2-2^{1-m}+\sum_{i} 2^{-i}(c(n-m+i-1)+d+\epsilon)+2^{-m}(c(n-1)+d+\epsilon)+\epsilon$$
$$\leq 2-2^{1-m}+cn+d-cm(1-2^{-m})-c+c(2-(m+2)/2^{m})+2\epsilon$$
$$\leq 2-2^{1-m}+cn+d-c(m-1+2^{1-m})+2\epsilon$$

So $c \leq (2-2^{1-m}+2\epsilon)/(m-1+2^{1-m})$. Since ϵ can be arbitrarily small, we conclude that $c \leq (2-2^{1-m})/(m-1+2^{1-m})$.◊

This analysis has shown that as m grows, T(n) approaches $(2-2^{1-m})n/m+d+o(1)$. In fact for any pattern, any algorithm must examine at least $(2-2^{1-m})\lfloor n/m \rfloor$ characters on the average. To see this consider a fixed pattern and algorithm. Now consider any m consecutive positions in the text, $t_j \cdots t_{j+m-1}$. No character outside this segment can affect whether the pattern occurs starting at j. Therefore, the algorithm must keep examining characters in this segment until it has found a mismatch or examined all the characters. Thus, on the average, it must examine at least

$$1\cdot 1/2+2\cdot 1/4+3\cdot 1/8+\cdots+(m-1)\cdot 1/2^{m-1}+m\cdot 1/2^{m-1}=2-2^{1-m}$$

characters in this segment. We can divide the text into $\lfloor n/m\rfloor$ such segments, and this must be true in all these segments. So $T(n)\geq(2-2^{1-m})\lfloor n/m\rfloor$. So, as one might imagine, the running time of the optimal algorithm for $p=0^m$ is essentially the least for all patterns.

VI. Summary

Our study has been divided into two areas, theoretical analysis of string-searching algorithms and the design and analysis of a new algorithm. In the theoretical study we have concentrated on the question of how many text characters are examined on the average by algorithms. By defining a finite-state model, string-searching automata, we have found a structure which permits detailed anaylsis of average running time. Using the techniques of Markov analysis we can compute the exact average time for an algorithm. More important is the fact that we can readily compute the asymptotic average time, $T(n)=cn+O(1)$. From this it is possible to find optimal algorithms using linear programming.

We also studied the simplest special case of string searching, when $p=0^m$. In this case the optimal SSA is also an optimal decision tree algorithm. This result reveals some important insights into the nature of the problem. In this special case the text can be processed left to right, never looking more than m characters ahead, and still yield an optimal algorithm. Also, the algorithm has a finite number of states -- the added power of decision trees is not needed.

The more practical part of our research has dealt with a new algorithm for string searching. This algorithm has worst-case running time of $\Theta(n+m)$ and average running time $O(n \cdot \log(m)/m+m)$ on a random access machine. These times are within a constant factor of optimal. The sizes of these constant factors must still be learned before the practicality of the new algorithm is determined.

There remain many open problems concerning SSA's and the nature of string searching. We have developed some intuition about some of these problems, and will state them as conjectures.

<u>Conjecture</u> For every pattern an optimal SSA is also an optimal decision tree algorithm. Thus, there exists an optimal decision tree algorithm which processes the text from left to right, with a lookahead of at most m characters.

<u>Conjecture</u> An optimal all-occurrences algorithm, when appropriately modified, is an optimal first-occurrence algorithm.

<u>Open Problem</u> Is there a simple, structural characterization of optimal SSA's? The hypothesis that the reduced-Boyer-Moore SSA is always optimal is proved incorrect by considering the patterns

$0^{m-1}1$ for $m>3$. In these cases, it is better to make the first probe at position m-1 instead of m.

Conjecture The worst-case number of states in the reduced-Boyer-Moore SSA grows exponentially with m, or at least non-polynomially. Thus, it is not practical to construct the rBM-SSA in the preprocessing step of the algorithm. We have computed the maximum number of states and the corresponding pattern for $\Sigma=\{0,1\}$, $m=1,\cdots,16$. They are:

m	states	pattern
1	1	0
2	3	00
3	6	000
4	12	0100
5	20	01000
6	30	010000
7	42	0100000
8	57	01010000
9	83	011101100
10	106	0010100011
11	155	01101011100
12	196	011011110100
13	281	0110101110100
14	361	01000001000101
15	517	011010111011100
16	634	0111010110111100

This appears to be exponential growth.

Conjecture For all patterns and all n, the rBM-SSA is worst-case optimal.

Open Problem Find a good lower bound on the average time to find multiple patterns.

Finally, the techniques of this thesis may be applicable to other computational problems. Specifically, the Markov analysis of Chapter 3 and the decision tree lemma of Chapter 5 may be useful in analyzing the average time for algorithms for other problems.

References

[A&C] Aho, A.V. and Corasick, M.J., "Fast Pattern Matching: An Aid to Bibliographic Search." Communications of the ACM 18, 6 (June 1975), pp. 333-340.

[AHU] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Massachusetts, 1974.

[B&M] Boyer, R.S. and Moore, J.S., "A Fast String Searching Algorithm." Communications of the ACM 20, 10 (October 1977), pp. 762-772.

[Fel] Feller, W., An Introduction to Probability Theory and Its Appliications. John Wiley & Sons, New York, 1968.

[F&P] Fischer, M.J. and Paterson, M.S., "String Matching and Other Products." Project MAC TM-41, MIT, Cambridge, Massachusetts, 1974.

[Fla] Flajolet, P., private communication.

[G&S] Galil, Z., and Seiferas, J., "Saving Space in Fast String-Matching." IEEE 18th Annual Symposium on Foundations of Computer Science (1977), pp. 179-188.

[Gal] Galil, Z., "On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm." Conference on

Automata, Languages, & Programming V (Lecture Notes in Computer Science 62), Springer-Verlag, 1978.

[G&O] Guibas, L.J. and Odlyzko, A.M., "A New Proof of the Linearity of the Boyer-Moore String Searching Algorithm." IEEE 18th Annual Symposium on Foundations of Computer Science (1977), pp. 189-195.

[Hal] Hall, M., Jr., Combinatorial Theory, Blaisdell, Waltham, Massachusetts, 1967.

[Har] Harrison, M.C., "Implementation of the Substring Test by Hashing." Communications of the ACM 14, (December 1971), pp. 777-779.

[KMR] Karp, R.M., Miller, R.E., and Rosenberg, A.L., "Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays." Proceedings of the 4th Annual ACM Symposium on Theory of Computing (1972), pp. 125-136.

[Kn1] Knuth, D.E., The Art of Computer Programming: Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, Massachusetts, 1973.

[Kn2] Knuth, D.E., The Art of Computer Programming: Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Massahcusetts, 1973.

[Kn3] Knuth, D.E., "Big Omicron and Big Omega and Big Theta."

SIGACT News 8, 2 (June 1976), pp. 18-24.

[KMP] Knuth, D.E., Morris, J.H., and Pratt, V.R., "Fast Pattern Matching in Strings." SIAM Journal on Computing 6, 2 (June 1977), pp. 323-350.

[Lue] Luenberger, D.G., Introduction to Linear and Nonlinear Programming. Addison-Wesley, Reading, Massachusetts, 1973.

[McC] McCreight, E.M., "A Space-Economical Suffix Tree Construction Algorithm." Journal of the ACM 23, 2 (April 1976), pp. 262-272.

[Riv] Rivest, R.L., "On the Worst-Case Behavior of String-Searching Algorithms." SIAM Journal on Computing 6, 4 (December 1977), pp. 669-674.

[Tuz] Tuza, Z., "'Very Bad' Worst-Case Behavior of String-Searching Algorithms." manuscript in preparation, 1978.

[Wei] Weiner, P., "Linear Pattern Matching Algorithms." IEEE 14th Annual Symposium on Switching and Automata Theory (1973), pp. 1-11.

[Yao] Yao, A.C-C., "The Complexity of Pattern Matching for a Random String." Stanford University Computer Science Department CS-77-629, 1977.

Biograghical Note

Contrary to popular belief, Paul Bayer was born in Chicago, not at MIT. Through accidents of moving and aging, he entered and left eight schools in Illinois and New Jersey before landing at MIT in 1969. Another popular belief will be dispelled if, as seems likely, he leaves MIT after only ten years.

Paul met his first computer in high school and has been courting a variety of IBM's, PDP's, and CDC's ever since. He has also courted his wife Terry for five years with an amazing result -- Andrew Bayer, age 16 months.

While an undergraduate, his extracurricular life centered on the MIT gymnastics team. He was a letter winner for four years, the captain for one, and an assistant coach for two years during graduate school.

His academic interests are focused on, but not limited to the study of algorithms. He is a member of ACM, a number of its special interest groups, SIAM, and IEEE. His work experience has been mainly in the development of minicomputer operating systems and other software. After graduation he plans to continue work in this area as a sidelight to his principal occupation of playing with his wife and son.

Paul's family of two parents, one brother and two sisters are scattered around the globe in Houston, Chicago, Ithaca, and Petersfield, England. He will be survived at MIT by a cousin in the class of 1983.