접미사 배열을 이용한 선형시간 탐색
(Linear-Time Search in Suffix Arrays)

심정섭 † 김동규 †† 박희진 ††† 박근수 ††††
(Jeong Seop Sin) (Dong Kyue Kim) (Heejin Park) (Kunsoo Park)

요 약 계산 생물학이나 문자열 연구 분야에 다양하게 응용되는 패턴 탐색 문제에 접미사 트리와 접미사 배열과 같은 인덱스 자료구조가 널리 사용되어 왔다. 접미사 트리를 이용한 패턴 탐색이 접미사 배열을 이용한 탐색보다 시간 복잡도 관점에서 더 빠른 것으로 알려져 왔다. 즉, 상수 크기의 알파벳에 대해 패턴 P를 길이 n인 텍스트에서 탐색하기 위해 접미사 트리는 $O(|P|)$시간이 필요한 반면 접미사 배열은 $O(|P|+\log n)$ 시간이 필요하다. 본 논문에서는 상수 크기 알파벳에 대해 접미사 배열을 이용한 선형시간 탐색 알고리즘을 제시한다. 본 알고리즘은 일반적인 알파벳 Σ에 대해서는 $O(|P|\log|\Sigma|)$시간이 필요하다.

키워드 : 문자열 처리, 패턴 탐색, 접미사 배열, 접미사 트리

Abstract To search a pattern P in a text, such index data structures as suffix trees and suffix arrays are widely used in diverse applications of string processing and computational biology. It is well known that searching in suffix trees is faster than suffix arrays in the aspect of time complexity, i.e., it takes $O(|P|)$ time to search P on a constant-size alphabet in a suffix tree while it takes $O(|P|+\log n)$ time in a suffix array where n is the length of the text.

In this paper we present a linear-time search algorithm in suffix arrays for constant-size alphabets. For a general alphabet Σ, it takes $O(|P|\log|\Sigma|)$ time.

Key words : string processing, pattern search, suffix arrays, suffix trees

1. Introduction

Suffix trees and suffix arrays are important index data structures in diverse applications of string processing and computational biology. The suffix tree due to McCreight [1] is a compacted trie of all the suffixes of a string T. It was designed as a simplified version of Weiner's position tree [2]. The suffix array due to Manber and Myers [3] and independently due to Gonnet et al. [4] is basically a sorted list of all the suffixes of a string T.

Despite simplicity of suffix arrays, suffix trees have been the most fundamental index data structures in the literature [5], [6] because suffix arrays were inferior to suffix trees in the following aspects.

(1) Construction time: Suffix trees can be constructed in linear time for an integer alphabet, while constructing suffix arrays takes $O(n\log n)$ time even for a constant-size alphabet [3], [7]. (Suffix arrays can be constructed from suffix trees in linear time, but it has been an open problem whether suffix arrays can be constructed in $O(n\log n)$ time without using trees.)

(2) Search time: In suffix trees a search for a pattern P can be done in $O(|P|\log|\Sigma|)$ time for an alphabet Σ, while it is done in $O(|P|+\log n)$ time in suffix arrays. In search time, suffix trees are better than suffix arrays for a constant-size alphabet, but the opposite is true for other cases.

Recently, however, there has been vigorous research on suffix arrays [8-12]. For the construction

---

· This work was supported by Korea Research Foundation grant KRF-2003-03-D00343.

† 종신회원 : 인하대학교 컴퓨터공학부 교수
jssim@inha.ac.kr
†† 종신회원 : 부산대학교 컴퓨터공학부 교수
dkkim1@pusan.ac.kr
††† 종신회원 : 한양대학교 정보통신대학 교수
hjpark@hanyang.ac.kr
†††† 종신회원 : 서울대학교 컴퓨터공학부 교수
kpark@theory.snu.ac.kr

논문접수 : 2005년 1월 24일
심사완료 : 2005년 3월 16일

of suffix arrays, Kärkkäinen and Sanders [10] and Kim et al. [11] independently developed a linear-time suffix array construction algorithm. The two algorithms are both using divide-and-conquer approach [13], [5], [14], i.e., (i) recursively construct partial suffix arrays, (ii) construct the suffix array of the remaining suffixes, (iii) merge the two suffix arrays into one. Almost at the same time, Ko and Aluru devised an interesting linear-time suffix array construction algorithm [12]. They used simple and nice properties of suffixes of a string.

For the search time in suffix arrays, Abouelhoda et al. developed an $O(|P||\Sigma|)$-time search algorithm [15]. In this paper, we present an $O(|P|\log|\Sigma|)$-time search which uses an interesting idea developed by Ferragina and Manzini in the context of compressed pattern matching [16]. Our algorithm is faster and simpler than Abouelhoda et al.'s algorithm [15]. The additional space needed to search P efficiently in our algorithm is $O(n)$.

This paper is organized as follows. In Section 2, we define some notations. In Section 3, we explain our search algorithm and the data structures. In Section 4, we conclude.

2. Preliminaries

2.1 Basics

We first give some definitions and notations that will be used in our algorithm. Consider a string T of length n over an alphabet Σ. $T[i]$ denotes the ith symbol of string T and $T[i,j]$ the substring starting at position i and ending at position j in T. We assume that $T[n]$ is a special symbol # which is lexicographically smaller than any other symbol in Σ and appears only once in T. Let S_i, $1 \le i \le n$, denote the suffix of T that starts at position i.

The suffix array A_T is the lexicographically ordered list of all suffixes of T. That is, $A_T[i] = j$ if S_j is lexicographically the ith suffix among all suffixes $S_1, S_2, \ldots, S_n$ of T.

2.2 *U* and *V*

Consider the problem of searching T for a pattern P over alphabet Σ. Let $p=|P|$ and $n=|T|$. Let σ_j be the jth smallest symbol in Σ and assume σ_0 is the special symbol #.

We define two arrays: $V[i]=T[A_T[i]]$ and $U[i]=T[A_T[i]-1]$ for $1 \le i \le n$ (assume $T[0]=\#$ for convenience), i.e., V is the array of the first symbols in the sorted list of all suffixes of T and U is the array of previous symbols of V. See Figure. 1. We use U and V only conceptually; we do not make them but access them in constant time with T and A_T. The idea of searching with U and V was developed by Ferragina and Manzini [16] to find patterns in a compressed file. A similar idea in a compressed suffix array was given by Sadakane [17].

U	V	
b	#	–
b	a	ababbb#
b	a	baababbb#
a	a	babbb#
#	a	bbabaababbb#
b	a	bbb#
b	b	#
a	b	aababbb#
b	b	abaababbb#
a	b	abbb#
b	b	b#
a	b	babaababbb#
a	b	bb#

Fig. 1 Arrays U and V when T=*abbabaababbb#*

Let $M[\sigma_j]$, $0 \le j \le |\Sigma|$, be the position of the first occurrence of σ_j in V. When σ_j does not occur in T, $M[\sigma_j]=M[\sigma_{j'}]$ where $\sigma_{j'}$ is the lexicographically smallest symbol that occurs in T and $\sigma_j < \sigma_{j'}$. Array M logically partitions V by each symbol σ_j occurring in T. We define a function $N:\{0,1,\ldots,n\} \times \{\sigma_0,\ldots,\sigma_{|\Sigma|}\} \rightarrow \{0,1,\ldots,n-1\}$. $N(i,\sigma_j)$ is the number of occurrences of σ_j in $U[1,i]$. For convenience, we assume $N(0,\sigma_j)=0$ for $0 \le j \le |\Sigma|$.

For example, if T=*abbabaababbb#* and $\Sigma=\{a,b\}$, then $A_T = \{13,6,4,7,1,9,12,5,3,8,11,2,10\}$, $V = (\#,a,a,a,a,a,b,b,b,b,b,b,b)$, and $U = (b,b,b,a,\#,b,b,a,b,a,b,a,a)$. See Figure. 1. Also, $M=(1,2,7)$ and $N(1,\#)=0$, $N(1,a)=0$, $N(1,b)=1,\cdots$, $N(7,\#)=1$, $N(7,a)=1$, $N(7,b)=5,\cdots$, $N(13,\#)=1$, $N(13,a)=5$, $N(13,b)=7$.

3. Linear-Time Search

3.1 Search Algorithm

We search for P from the last symbol to the first symbol of P. Note that P occurs at position i of T if and only if the suffix S_i of T has P as its prefix. Assume we know all the positions where $P[h+1, p]$ occurs in T. Then we can find the positions where $P[h,p]$ appears in T as follows. If there exists any suffix of T that has $P[h+1, p]$ as its prefix and its previous symbol is $P[h]$, then $P[h,p]$ appears in T. Since U is the array of previous symbols of the sorted suffixes and the positions where $P[h+1, p]$ occurs in T are contiguous in A_T, we check if $P[h]$ exists in the corresponding positions by using U. If it does, we find the positions where $P[h,p]$ occurs in T using the following lemma.

Lemma 1. Let S_k be the ith lexicographically smallest suffix of all the suffixes of T that start with σ_j. Then S_k is the $(M[\sigma_j]+i-1)$st lexicographically smallest suffix among all the suffixes of T.

Our algorithm is divided into p phases. Assume that M and N are available. At the beginning of the hth phase from $h=p$ to 1, we know all the positions where $P[h+1, p]$ occurs in T. In fact, we maintain the start position p_s and end position p_e of the contiguous block where $P[h+1, p]$ occurs. Initially, $p_s = 1$ and $p_e = n$. (At the beginning when $h = p$.) In the hth phase, we find all the positions where $P[h, p]$ occurs in T. That is, we update p_s and p_e so that all occurring positions of $P[h, p]$ in T are $A_T[p_s],\ldots,A_T[p_e]$. The new start and end positions are set to $M[P[h]]+N(p_s-1,P[h])$ and $M[P[h]]+N(p_e,P[h])-1$, respectively.

We now show the correctness of this algorithm. Consider the hth phase of our algorithm. Assume there are i number of occurrences of $P[h]$ in the current block of $U[p_s,p_e]$ and the first occurrence of them is the jth occurrence of $P[h]$ in U. Consider the suffixes that start with $P[h]$ in $U[p_s,p_e]$. By Lemma 1, the new start and end positions are $M[P[h]]+j-1$ and $M[P[h]]+j-1+i-1$, respectively. By definitions of M and N, $j=N(p_s-1,P[h])+1$ and $i=N(p_e,P[h])-N(p_s-1,P[h])$. Hence, this algorithm correctly sets the new values of p_s and p_e. Note that if $p_s > p_e$, then there exist no occurrences of P in T.

For the above example when T=*abbabaababbb#* and P=*aba*, the initial value of p_s and p_e are 1 and 13, respectively. In the phase of $h=3$, we find all the positions where $P[3,3]=a$ occurs in T. That is, the new $p_s=2$ and the new $p_e=6$. See Figure. 2. Next, in the phase of $h=2$, we find all the positions where $P[2,3]=ba$ occurs in T. All the occurrences of *ba* must appear between p_s and p_e in S_{A_T}, the sorted suffixes of T. Thus, the new $p_s=M[P[2]]+N(p_s-1,P[2])=M[b]+N(2-1,b)=7+1=8$, and the new $p_e=M[P[2]]+N(p_e,P[2])-1=M[b]+N(6,b)-1=7+4-1=10$. It means that *ba* occurs at $A_T[8]$, $A_T[9]$, and $A_T[10]$. See Figure. 3. In the phase of $h=1$, we find all the positions where $P[1,3]=aba$ occurs in T. Thus, the new $p_s=M[P[1]]+N(p_s-1, P[1])=M[a]+N(8-1,a)=2+1=3$, and the new $p_e=M[P[1]]+N(p_e,P[1])-1=M[a]+N(10,a)-1=2+3-1=4$. Therefore, P appears at the positions $A_T[3]=4$ and $A_T[4]=7$ in T. See Figure. 4.

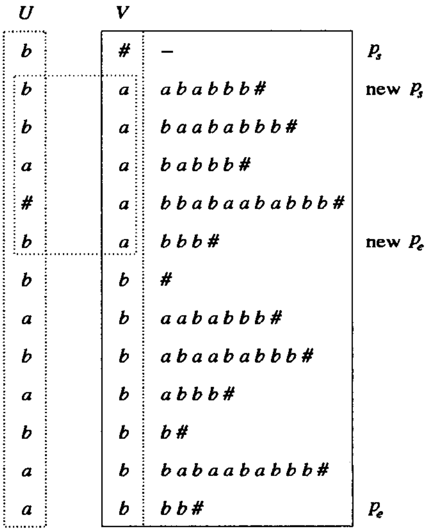

Fig. 2 Search P=*aba* in T=*abbabaababbb#*.(h=3)

3.2 Preprocessing and query for N(i,σ_j)

We now explain how to preprocess A_T to construct array M and make data structures for function N. We can construct array M in $O(n)$ time by scanning V once and it needs $O(|\Sigma|)$ space. We first explain how to answer a query $N(i,$

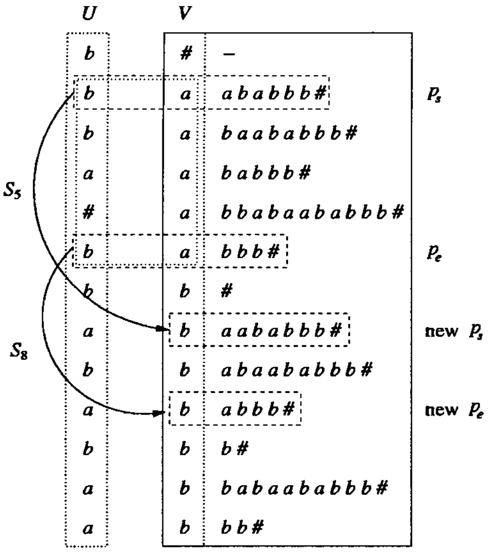

Fig. 3 Search $P=bab$ in $T=abbabaababbb\#$.(h=2)

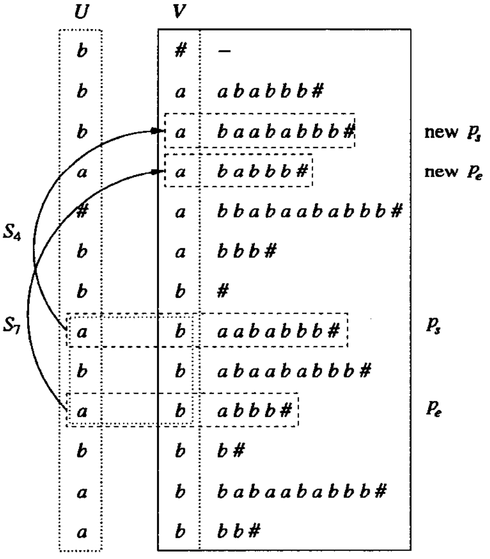

Fig. 4 Search $P=aba$ in $T=abbabaababbb\#$.(h=1)

σ_j) in $O(\log n)$ time; then we give a solution that takes $O(\log|\Sigma|)$ time.

First, we describe an $O(\log n)$ time solution for query $N(i,\sigma_j)$. We make two arrays: Y of size $O(n)$ and Z of size $O(|\Sigma|)$. Let n_j denote the number of total occurrences of σ_j in U and $a_j=\sum_{0\le k<j} n_k$ (assume a_0=0). For the above example of $T=abbabaababbb\#$, a_0=0, a_1=1, a_2=6. For $0\le j\le|\Sigma|$, $Z[j]$ stores a_j and $Y[a_j+m]$ stores the place of mth occurrence of σ_j in U. See Figure. 5. We can make Y and Z in $O(n)$ time by scanning U.

With Y and Z, we can answer a query $N(i,\sigma_j)$ in $O(\log n)$ time. First, we find a_j and a_{j+1} by accessing $Z[j]$ and $Z[j+1]$. Then we do a binary search on $Y[a_j+1],\cdots,Y[a_{j+1}]$ to find the maximum k such that $Y[k]\le i$. Then, the number of occurrences of σ_j in $U[1,i]$ is $k-a_j$, which is the answer to query $N(i,\sigma_j)$. For the query $N(8,a)$ on the above example, we first access $Z[1]$ and $Z[2]$ to find a_1=1 and a_2=6. Now we do a binary search on $Y[2,6]$ to find $Y[3]$=8. Thus, the number of occurrences of a in $U[1,8]$=3-1=2. See Figure. 5. Since the portion of Y we search can be of size $O(n)$, this solution takes $O(\log n)$ time in the worst case for a query $N(i,\sigma_j)$.

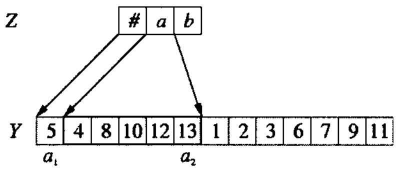

Fig. 5 Arrays Y and Z when $T=abbabaababbb\#$

For an $O(\log|\Sigma|)$-time solution for query $N(i,\sigma_j)$, we divide U into blocks of size $|\Sigma|$ to reduce the size of Y to $|\Sigma|$. Let U^i for $1\le i\le n/|\Sigma|$ be the ith block of U and s_i and e_i be the start position and end position of U^i, respectively. First, we make a two dimensional array X of size $O(n)$. $X[i,j]$, $1\le i\le n/|\Sigma|$ and $0\le j\le|\Sigma|$, stores the number of occurrences of σ_j in $U[1,e_i]$. Array X can be made in $O(n)$ time by scanning U. Second, as in the previous $O(\log n)$-time solution, we make two arrays Y^i and Z^i for each U^i. Let n_j^i be the number of occurrences of σ_j in U_i and $a_j^i=\sum_{0\le k<j} n_k^i$. For $0\le j\le|\Sigma|$, $Z^i[j]$ stores a_j^i and $Y^i[a_j^i+m]$ stores the place of mth occurrence of σ_j in U^i. Both Y^i and Z^i can be made in $O(|\Sigma|)$ time by scanning U^i since U^i has $|\Sigma|$ elements.

Now we can answer query $N(i,\sigma_j)$ in $O(\log|\Sigma|)$ time. First, we find the block U^w such that $w=\lceil i/|\Sigma|\rceil$. Then, we access Z^w to find a_j^w and do a binary search on $Y^w[a_j^w+1],\cdots,Y^w[a_{j+1}^w]$ to find the maximum k such that $Y^w[k]\le i$. Then, $N(i,\sigma_j)=X[w-1,j]+k-a_j^w$.

4. Conclusion

In this paper, we proposed an $O(|P|\log|\Sigma|)$-time algorithm for searching P in a text on an alphabet Σ. This result equips suffix arrays with the same search time as suffix trees. Therefore, the suffix array is more powerful than the suffixtree in the sense that it has a choice of $O(|P|\log|\Sigma|)$ or $O(|P| +\log n)$ search time depending on the alphabet size.

References

[1] E.M. McCreight, A space-economical suffix tree construction algorithm, *J. Assoc. Comput. Mach.*, vol. 23, pp. 262-272, 1976.

[2] P. Weiner, Linear pattern matching algorithms, *Proc. 14th IEEE Symp. Switching and Automata Theory*, pp. 1-11, 1973.

[3] U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, *SIAM J. Comput.*, vol. 22, pp. 935-938, 1993.

[4] G. Gonnet, R. Baeza-Yates, and T. Snider, *New indices for text: Pat trees and pat arrays.* In W. B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pp. 66-82. Prentice Hall, 1992.

[5] M. Farach-Colton, P. Ferragina and S. Muthukrishnan, On the sorting-complexity of suffix tree construction, *J. Assoc. Comput. Mach.*, vol. 47, pp. 987-1011, 2000.

[6] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge Univ. Press, 1997.

[7] D. Gusfield, An "Increment-by-one" approach to suffix arrays and trees, *manuscript*, 1990.

[8] S. Burkhardt and J. Kärkkäinen, Fast lightweight suffix array construction and checking, *Symp. Combinatorial Pattern Matching*, LNCS 2676, pp. 55-69, 2003.

[9] W. Hon, K. Sadakane, and W. Sung, Breaking a time-and-space barrier in constructing full-text indices, *Proc. IEEE Symp. Found. Computer Science*, pp.251-260, 2003.

[10] J. Kärkkäinen and P. Sanders, Simple linear work suffix array construction, *Int. Colloq. Automata Languages and Programming*, LNCS 2719, pp. 943-955, 2003.

[11] D. Kim, J.S. Sim, H. Park, and K. Park, Linear-time construction of suffix arrays, *Symp. Combinatorial Pattern Matching*, LNCS 2676, pp. 186-199, 2003.

[12] P. Ko and S. Aluru, Space efficient linear time construction of suffix arrays, *Symp. Combinatorial Pattern Matching*, LNCS 2676, pp. 200-210, 2003.

[13] M. Farach, Optimal suffix tree construction with large alphabets, *IEEE Symp. Found. Computer Science*, pp. 137-143, 1997.

[14] R. Hariharan, Optimal parallel suffix tree construction, *J. Comput. Syst. Sci.*, vol. 55, pp. 44-69, 1997.

[15] M.I. Abouelhoda, E. Ohlebusch, and S. Kurtz, Optimal exact string matching based on suffix arrays, *International Symposium on String Processing and Information Retrieval*, LNCS 2476, 31-43, 2002.

[16] P. Ferragina and G.. Manzini, Opportunistic data structures with applications, *IEEE Symp. Found. Computer Science*, 390-398, 2001.

[17] K. Sadakane, Succinct representation of lcp information and improvement in the compressed suffixarrays, *ACM-SIAM Symp. on Discrete Algorithms*, pp. 225-232, 2002.

심 정 섭

1995년 서울대학교 컴퓨터공학과 학사
1997년 서울대학교 컴퓨터공학과 석사
2002년 서울대학교 컴퓨터공학부 박사
2002년 3월~2004년 8월 한국전자통신연구원 바이오정보연구팀 선임연구원. 2004년 9월~현재 인하대학교 컴퓨터공학부 전임강사. 관심분야는 컴퓨터 이론 및 알고리즘, 생물정보학

김 동 규 ·

1992년 서울대학교 컴퓨터공학과 학사
1994년 서울대학교 컴퓨터공학과 석사
1999년 서울대학교 컴퓨터공학과 박사
1999년~현재, 부산대학교 컴퓨터공학과 조교수. 관심분야는 암호학 및 알고리즘, 컴퓨터 보안 시스템, Bioinformatics.

박 희 진

1990년 3월~1994년 2월 서울대학교 컴퓨터공학과 학사. 1994년 3월~1996년 2월 서울대학교 컴퓨터공학과 석사. 1996년 3월~2001년 2월 서울대학교 컴퓨터공학과 박사. 2001년 3월~2003년 2월 서울대학교 컴퓨터연구소 전문연구원. 2003년 2월~2003년 8월 이화여자대학교 컴퓨터학과 BK연구교수. 2003년 9월~현재 한양대학교 정보통신대학 전임강사 관심분야는 알고리즘, 정보보호

박 근 수

1983년 서울대학교 컴퓨터공학과 학사
1985년 서울대학교 컴퓨터공학과 석사
1991년 미국 Columbia 대학교 전산학 박사. 1991년 11월~1993년 8월 영국 런던대학교 King's College 조교수. 1995년 7월~1995년 8월 호주 Curtin 대학교 방문연구원. 1993년 8월~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 컴퓨터이론, 생물정보학, 암호학