DNA 스트링에 대하여 써픽스 배열을 구축하는 빠른 알고리즘

(Fast Construction of Suffix Arrays for DNA Strings)

조 준 하 † 김 남 희 †† 권 기 룡 ††† 김 동 규 ††††
(Junha Jo) (Namhee Kim) (Ki-Ryong Kwon) (Dong Kyue Kim)

요 약 DNA 스트링과 같은 대용량의 데이타에 대한 빠른 검색을 수행하기 위해서는 전체 텍스트 인덱스 자료구조를 구축하여 검색하는 방법이 효율적이다. 가장 일반적인 인덱스 자료구조는 써픽스 트리와 써픽스 배열이다. 써픽스 배열은 써픽스 트리보다 적은 공간을 사용하기 때문에 DNA 스트링과 같은 대용량의 데이타에 적합한 자료구조이다. 기존의 써픽스 배열 구축 알고리즘들은 정수 문자집합에 적합한 알고리즘들이어서 DNA 스트링에 적합하지 않았다. 본 논문에서는 DNA 스트링의 문자집합이 4로 고정되어 있는 사실을 이용하여 DNA 스트링에 대한 써픽스 배열을 빠르게 구축하는 방법을 제안한다. 고정길이 문자집합에 효율적인 Kim et. al.[1]의 알고리즘의 인코딩 과정과 합병 과정 개선으로 전체 구축 시간을 향상시켰다. 실험 결과 1.3배에서 1.6배 정도 구축 속도가 향상되었으며, 기존의 다른 써픽스 배열 구축 알고리즘들과 비교한 결과에서도 대부분 가장 빠르게 써픽스 배열을 구축하였다.

키워드 : 전체 텍스트 인덱스 자료구조, 써픽스 배열, 써픽스 배열 구축, DNA 스트링

Abstract To perform fast searching in massive data such as DNA strings, the most efficient method is to construct full-text index data structures of given strings. The widely used full-text index structures are suffix trees and suffix arrays. Since the suffix array uses less space than the suffix tree, the suffix array is proper for DNA strings. Previously developed construction algorithms of suffix arrays are not suitable for DNA strings since those are designed for integer alphabets. We propose a fast algorithm to construct suffix arrays on DNA strings whose alphabet sizes are fixed by 4. We reduce the construction time by improving encoding and merging steps on Kim et al.[1]'s algorithm. Experimental results show that our algorithm constructs suffix arrays on DNA strings 1.3-1.6 times faster than Kim et al.'s algorithm, and also for other algorithms in most cases.

Key words : full-text index data structures, suffix arrays, DNA strings

1. 서 론

스트링 검색 문제는 길이가 n인 텍스트 T에서 길이가 m인 패턴 스트링 P를 찾는 문제이다. 이 문제는 많은 응용프로그램에서 사용되고 있으며, 오랫동안 연구가 진행되고 있다[2]. 효과적인 스트링 검색 방법에 대한 연구는 크게 두 방향으로 진행되고 있다. 첫 번째 방법은 $O(m)$ 시간에 패턴을 전 처리하여 $O(n)$ 시간에 검색하는 방법이다. 두 번째 방법은 텍스트에 대한 전체-텍스트 인덱스 자료구조(full-text index data structure)를 $O(n)$ 시간에 구축하고 $O(m)$ 시간에 검색하는 방법이다. DNA 스트링은 패턴에 비해 매우 길고, 많은 패턴을 검색하기 때문에 두 번째 방법으로 검색하는 것이 더 적합하다.

대표적인 인덱스 자료구조에는 써픽스 트리(suffix tree)와 써픽스 배열(suffix array)이 있다. 써픽스 트리는 텍스트의 모든 써픽스들을 압축된 trie로 표현한 것이다. McCreight[3], Ukkonen[4], Farach, Ferragina, 및 Muthukrishnan[5,6]의 알고리즘들에 의해서 $O(n)$

· 본 연구는 한국과학재단 특정기초연구(R01-2006-000-10260-0) 지원으로 수행되었음
· 이 논문은 2006년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(KRF-2006-311-D00762)
† 정 회 원 : (주)소암시스템 연구소 Core 개발팀 연구원
junhajo@soamsys.com
†† 학생회원 : 한양대학교 전자통신컴퓨터공학부
nhkim@esslab.hanyang.ac.kr
††† 정 회 원 : 부경대학교 컴퓨터공학과 교수
krkwon@pknu.ac.kr
†††† 종신회원 : 한양대학교 전자통신컴퓨터공학부 교수
dqkim@hanyang.ac.kr
(Corresponding author임)
논문접수 : 2007년 1월 22일
심사완료 : 2007년 5월 18일

시간에 구축할 수 있다. 써픽스 배열은 텍스트의 모든 써픽스들을 정렬한 리스트로, 써픽스 트리에 비해 적은 공간을 사용하기 때문에 대용량 텍스트인 DNA 스트링의 검색에 적합하다.

써픽스 배열은 써픽스 트리를 구축하여 순회하는 방법을 사용하면 $O(n)$ 시간에 구축할 수 있지만, 많은 공간을 사용해야 하고, 구현이 복잡하다. Manber와 Myers[7]와 Gusfield[8]는 텍스트에서 직접 써픽스 배열을 $O(n\log n)$ 시간에 구축하는 알고리즘을 제안했다. Larsson과 Sadakane[9]는 시간 복잡도는 $O(n\log n)$이지만 Manber와 Myers보다 성능은 훨씬 향상된 알고리즘을 제안하였다. 최근에는 선형 시간에 써픽스 배열을 구축하는 Kärkkäinen과 Sanders[10]의 알고리즘, Kim, Sim, Park 및 Park[11]의 알고리즘, Ko와 Aluru[12]의 알고리즘이 발표되었다. Kim, Jo, Park[1]은 문자집합의 크기가 고정되어 있는 경우 합병을 빠르게 할 수 있는 방법을 이용해서 시간 복잡도가 $O(n\log\log n)$이면서도 실제 구축 속도가 매우 빠른 알고리즘을 제시하였다.

DNA 스트링은 작은 크기의 고정길이 문자집합으로 구성되어 있기 때문에, 정수 문자집합을 위한 기존의 선형 시간 구축 알고리즘들을 이용하여 DNA 스트링에 대한 써픽스 배열을 선형시간에 구축하기에 적합하지 않다. 본 논문에서는 DNA 스트링의 특성을 고려하여 써픽스 배열을 더욱 빠르게 구축하는 방법을 제안한다. 기존 알고리즘들 중에서 고정길이 문자집합에 적합한 Kim, Jo 및 Park의 알고리즘(이하 논문에서 KJP라 칭함)에서의 인코딩 과정과 합병 과정을 DNA 스트링의 특성에 맞게 효율적으로 처리하여 전체 구축 시간을 단축시켰다. 실험 결과 1.35배에서 1.46배 정도 더 빠르게 써픽스 배열을 구축하였다.

2장에서는 기존의 써픽스 배열 구축 알고리즘들을 설명하고, 3장에서는 DNA 스트링에 대한 써픽스 배열을 빠르게 구축하기 위한 방법을 설명하고, 4장에서는 실험을 통해 구축 시간을 측정하여 성능 향상을 보이고, 5장에서 본 논문에 대한 결론을 맺는다.

2. 기존의 써픽스 배열 구축 알고리즘

기존의 써픽스 배열 구축 알고리즘들을 설명하고, 고정 길이 문자집합에 효율적인 KJP를 이용해서 DNA 스트링의 써픽스 배열을 구축하는 경우 개선할 부분을 설명한다.

2.1 $O(n\log n)$ 시간 알고리즘

대표적인 $O(n\log n)$ 시간 알고리즘인 Manber와 Myers[7]의 알고리즘을 설명하고, 이 알고리즘의 단점을 개선한 Larsson과 Sadakane[9]의 알고리즘을 설명한다.

2.1.1 Manber와 Myers 알고리즘

써픽스를 정렬하는 가장 쉬운 방법은 길이가 n인 텍스트의 모든 써픽스들의 모든 문자들을 비교하면서 정렬하는 방법이다. 첫 번째 단계에서 써픽스의 첫 번째 문자들로, 두 번째 단계에서 써픽스의 두 번째 문자들로, 마지막 단계에서는 n 번째 문자들로 정렬하게 되므로 총 n 단계의 정렬이 필요하다. 그러나 써픽스는 다른 써픽스의 써픽스가 되는 성질을 이용하면 n 단계를 $\log n$ 단계로 줄일 수 있다. 텍스트의 i번째 써픽스를 S_i라고 하자. 모든 써픽스가 처음 k개의 문자들로 정렬이 되었다면 S_i, S_j, S_{i-k}, $S_{j-k}(1 \le i, j \le n)$도 k개의 문자들로 정렬되어 있다. S_i와 S_j가 정렬된 순서를 이용해서 S_{i-k}와 S_{j-k}를 정렬하면 모든 써픽스들은 $2k$개의 문자들로 정렬되며, 이 방식을 더블링(doubling) 기법이라고 한다. Manber와 Myers[7]의 알고리즘은 더블링 기법을 사용하며, 정렬 방법은 다음과 같다.

- 모든 써픽스들을 첫 번째 문자로 버킷 정렬(bucket sort)하여 그 인덱스를 배열 A에 저장한다.
- h단계에서 모든 써픽스는 2^{h-1}개의 문자들로 버킷 정렬되어 있으며 $S_{A[i]}$의 순서를 이용해서 $S_{A[i]-2^{h-1}}$를 정렬한다. 정렬이 끝나면 모든 써픽스들은 2^h개의 문자로 정렬된다. h단계에서의 정렬은 다음과 같다.
- 배열 A를 차례로 읽으면서 $S_{A[i]-2^{h-1}}$를 자신이 속한 버킷의 제일 앞으로 이동시킨다.
- $S_{A[i+1]}$이 자신이 속한 버킷의 마지막이면 이동한 써픽스들을 새로운 버킷으로 구분한다.

매 단계마다 정렬되는 문자들의 개수는 이전 단계의 2배이므로 최대 $\log n$ 단계까지의 정렬이 필요하다. 각 단계의 버킷 정렬은 $O(n)$ 시간에 정렬되며 최대 $\log n$ 단계가 필요하므로 전체 써픽스들의 정렬 시간은 $O(n\log n)$이다.

2.1.2 Larsson과 Sadakane 알고리즘

Larsson과 Sadakane[9]는 Manber와 Myers[7]의 알고리즘이 정렬된 써픽스의 모든 인덱스들을 차례로 읽으면서 다른 써픽스를 정렬하기 때문에 매 단계마다 배열 전체를 읽어야 한다는 단점을 보완한 알고리즘을 개발했다. Larsson과 Sadakane 알고리즘은 정렬이 끝난 써픽스의 구간과 정렬이 되지 않은 써픽스들의 구간을 배열 L을 통해서 구분한다. $L[i] = k$이면 i부터 k개의 연속된 써픽스들은 정렬되지 않았다는 것을 의미하고, $L[i] = -k$이면 i부터 k개의 연속된 써픽스들은 이미 정렬이 끝났다는 것을 의미한다. 써픽스들의 순서를 결정하기 위해서 배열 V에 써픽스들의 순위를 저장한다. $V[i] = g$이면 써픽스 i의 순위는 g임을 의미한다. Lar-

sson과 Sadakane 알고리즘 역시 더블링 기법을 사용하며 정렬 방법은 다음과 같다.

(1) 모든 써픽스들을 첫 번째 문자로 버킷 정렬하여 그 인덱스를 배열 $I[1..n]$에 저장한다.
(2) h단계에서 배열 I을 읽으면서 정렬되지 않은 써픽스들이 존재하는 구간을 계산한다.
(3) 구간 내의 모든 $S_{I[i]}$들을 배열 $V[S_{I[i]+2^{h-1}}]$를 이용해서 정렬한다.
(4) 배열 V와 I을 다시 설정한다.
(5) 모든 써픽스들이 정렬될 때까지 (2)-(4)의 과정을 반복한다.

각 단계의 정렬은 $O(n)$ 시간에 수행되고 최대 $O(\log n)$ 단계가 필요하므로 시간 복잡도는 $O(n\log n)$이다.

2.2 선형 시간 알고리즘

선형시간 알고리즘 중 가장 간단하며 효율적인 Kärkkäinen과 Sanders[10]의 알고리즘을 설명한다. Kärkkäinen과 Sanders 알고리즘은 재귀적 분할 정복 기법을 사용한다. 텍스트 $T[1..n]$의 i번째 써픽스를 $S_i = T[i..n]$이라고 하자. $i\bmod 3 \neq 1$인 S_i들을 s_{20}이라 하고, $i\bmod 3 = 1$인 S_i들을 s_1이라고 하자. s_{20}의 처음 3개의 문자들을 인코딩해서 T'을 생성한다. T'의 모든 문자들이 다를 때까지 위의 과정을 재귀적으로 반복한 후, T'을 이용해서 s_{20}의 써픽스 배열 SA_{20}을 직접 구축한다. $S_i = (T[i], S_{i+1})$이므로 s_1의 써픽스 배열 SA_1을 구축하기 위해서는 먼저 첫 번째 문자 $T[i]\,(i\bmod 3 = 1)$를 비교해서 정렬하고 정렬되지 않는 써픽스들은 $S_{i+1}\,(i\bmod 3 = 1)$의 순서를 이용해서 정렬한다. 써픽스의 순서는 이미 구축된 SA_{20}에서 정렬되어 있으므로 빠르게 정렬할 수 있다. SA_{20}의 써픽스 $S_l\,(1 \le l \le 2n/3)$과 SA_1의 써픽스 $S_m\,(1 \le m \le n/3)$을 합병하여 SA를 구축하는 방법은 합병 정렬(merge sort)과 유사하다. S_l과 S_m을 비교해서 S_l(또는 S_m)이 빠르면 S_l(또는 S_m)을 SA에 저장하고 S_{l+1}(또는 S_l)과 S_m(또는 S_{m+1})을 비교한다. S_l와 S_m를 비교할 때는 두 가지 경우를 고려한다. $l\bmod 3 = 2$인 경우에는 $(T[l], S_{l+1})$과 $(T[m], S_{m+1})$을 비교하고, $l\bmod 3 = 0$인 경우는 $(T[l], T[l+1], S_{l+2})$와 $(T[m], T[m+1], S_{m+2})$를 비교해서 두 써픽스의 순서를 결정한다. 마지막에 비교하는 써픽스의 순서는 SA_{20}에 정렬되어 있기 때문에 최대 3번의 비교만으로 S_l와 S_m의 순서를 결정할 수 있어 합병이 빠르게 수행된다. 알고리즘의 모든 과정은 $O(n)$ 시간에 수행되며 T'의 크기는 매 재귀 단계마다 2/3씩 감소하므로 $T(n) = T(2n/3) + O(n)$이 되어 전체 알고리즘의 시간 복잡도는 $O(n)$이다.

2.3 고정길이 문자집합에 효율적인 알고리즘

고정길이 문자집합에 효율적인 KJP를 설명한다. KJP는 Kärkkäinen과 Sanders[10]의 알고리즘과 마찬가지로 재귀적 분할 정복 기법을 사용하지만, 홀수 써픽스와 짝수 써픽스로 나누어서 수행하는 점, 인코딩 및 합병 과정이 다르다. 텍스트 $T[1..n]$의 i번째 써픽스를 $S_i = T[i..n]$라고 하자. 홀수 인덱스를 가지는 써픽스들의 처음 2개의 문자를 인코딩해서 T'을 재귀적으로 생성한다. T'을 이용해서 홀수 써픽스 배열 SA_{Odd}를 직접 구축한다. 짝수 써픽스 S_i는 $(T[i], S_{i+1})$, $(i\bmod 2 \neq 1)$이므로 짝수 써픽스 배열 SA_{Even}을 구축하기 위해서는 짝수 써픽스들의 첫 번째 문자들을 비교해서 정렬하고, 정렬되지 않는 써픽스들은 이미 구축된 SA_{Odd}에 정렬되어 있는 S_{i+1}의 순서를 이용해서 정렬한다. SA_{Odd}와 SA_{Even}을 합병하여 SA_T를 구축하기 위해서 배열 C를 이용한다. 배열 C는 SA_{Odd}의 인접한 두 홀수 써픽스 사이에 포함될 짝수 써픽스들의 개수를 의미한다. 후위 검색을 수행해서 배열 C를 구축하고, 이를 이용해서 두 써픽스 배열을 합병한다.

합병을 제외한 나머지 과정은 $T(n) = T(n/2) + O(n)$이므로 $O(n)$ 시간에 수행되고, 합병 과정은 문자집합이 고정길이인 경우 $O(n\log\log n)$ 시간에 수행되어 전체 알고리즘의 시간 복잡도는 $O(n\log\log n)$이며, 사용 공간은 $O(n)$이다. 이 알고리즘이 선형 시간 알고리즘과 비교해도 느리지 않는 이유는 일반적으로 사용하는 텍스트는 크기가 4G 이하로 $\log\log n$이 6보다 작아 시간 복잡도 $O(n)$의 숨은 상수와 비교할 수 있기 때문이다.

이 알고리즘은 DNA 스트링에 대한 써픽스 배열을 구축하는 경우 인코딩 과정과 합병 과정에서 성능을 향상시킬 수 있다. 문자집합이 작으면 인코딩 과정에서 기수 정렬(radix sort) 대신 다른 빠른 정렬 방법을 사용할 수 있다. 합병 과정에서 이진 검색을 위해서는 추가로 자료구조를 구축해야 하며, 시간 복잡도에 숨은 상수가 크기 때문에 문자집합의 크기가 작은 경우 속도가 빠르지 않다. 다음 장에서는 이 알고리즘의 인코딩 및 합병 과정을 개선하여 DNA 스트링에 대한 써픽스 배열을 더욱 빠르게 구축하는 방법을 제안한다.

3. DNA 스트링에 대한 써픽스 배열의 구축 속도를 향상시키는 방법

KJP의 인코딩 과정과 합병 과정을 개선하여 DNA 스트링의 써픽스 배열 구축 속도를 향상시키는 방법을 설명한다. 인코딩 과정에서 DNA 스트링의 문자집합의 크기가 4인 점을 이용하여 기수 정렬 대신 카운트 정렬

(count sort)을 사용하는 방법과 합병 과정에서 이진 검색을 하지 않고 직접 배열의 값을 읽어서 시간을 단축시키는 방법을 설명한다.

3.1 인코딩의 성능 향상 방법

기존의 KJP의 인코딩 방법을 설명하고, 성능을 향상시키는 방법을 설명한다. 인접한 문자들의 순서쌍(pair) $(T[2i-1], T[2i])$, $1 \le i \le n/2$,를 새로운 문자로 대체하는 아래 두 단계를 통하여, T를 T의 절반의 길이인 T'으로 인코딩한다.

- 크기가 $n/2$인 배열 A에 $1 \le i \le n/2$인 $2i-1$의 값을 차례로 저장하여 초기화 한다($A[i]=2i-1$, $1 \le i \le n/2$). 배열 A에 저장된 값은 각 순서쌍들의 인덱스를 의미한다. 배열 A를 차례로 읽으면서 순서쌍의 두 번째 문자 $T[A[i]+1]$로 모든 순서쌍들을 정렬하고, 이렇게 정렬된 순서쌍들의 인덱스를 배열 B에 저장한다. 배열 B는 두 번째 문자로 정렬된 순서쌍들의 인덱스를 저장하고 있다. 배열 B를 이용해서 순서쌍의 첫 번째 문자 $T[B[i]]$로 모든 순서쌍들을 정렬하고, 정렬된 순서쌍들의 인덱스를 다시 배열 A에 저장한다. 이제 배열 A에는 사전 순으로 정렬된 모든 순서쌍들의 인덱스가 저장되어 있다.
- 크기가 $n/2$인 T'을 생성한다. 배열 A를 차례로 읽으면서 $T'[\lfloor A[i]/2 \rfloor]$에 순서쌍 $(T[A[i]], T[A[i]+1])$를 정수로 맵핑된 값을 저장하여 T'을 생성한다. 배열 A를 처음부터 차례로 읽으면서 현재 순서쌍와 이전 순서쌍를 비교해서 같으면 이전 순서쌍을 맵핑한 정수 값으로 현재 순서쌍을 맵핑하고 다르면 1을 증가시킨 정수 값으로 현재 순서쌍을 맵핑한다. 예를 들어, 순서쌍 $(T[A[i]], T[A[i]+1])$을 정수 k로 맵핑했을 경우 $(T[A[i+1]], T[A[i+1]+1])$이 $(T[A[i]], T[A[i]+1])$과 같으면 k로 맵핑하고, 다르면 $k+1$로 맵핑한다. 모든 순서쌍들에 대한 맵핑이 끝나면 T'이 생성된다.

인코딩 시 기수 정렬을 사용하는 이유는 문자집합의 크기가 n을 넘지 않을 때, $O(n)$ 시간에 $O(n)$ 공간을 사용하면서 모든 순서쌍들을 정렬하기 위해서이다. 그렇지만 기수 정렬 과정에서 배열에 저장되는 순서쌍들의 인덱스 값이 분산되기 때문에, 이를 이용하여 텍스트의 문자를 읽을 때 캐시 효과(cache effect)가 좋지 않아 정렬 속도가 느려진다.

DNA 스트링에 대한 인코딩 과정을 빠르게 할 수 있는 방법을 설명한다. 인코딩된 텍스트의 문자집합의 크기는 최대 $|\Sigma|^2$까지 증가할 수 있으므로 DNA 스트링과 같이 문자집합의 크기가 작으면 $|\Sigma|^2$ 크기를 가지는 테이블을 사용하는 카운트 정렬 방법을 사용하면 더 빠르게 수행할 수 있다. 카운트 정렬을 이용하여 인코딩 하는 방법은 다음과 같다.

- 크기가 $(|\Sigma|, |\Sigma|)$인 테이블을 생성하고 모든 값을 0으로, 변수 k를 0으로 초기화한다. X값은 문자 $T[2i-1]$을 의미하고, Y 값은 문자 $T[2i]$을 의미한다. 텍스트를 처음부터 읽으면서 발생하는 모든 순서쌍 $(T[2i-1], T[2i])$들을 테이블의 $(T[2i-1], T[2i])$의 위치에 0이 아닌 임의의 값을 기록한다. 이제 테이블을 (0, 0)부터 $(|\Sigma|, |\Sigma|)$까지 사전 순으로 읽으면서 0이 아닌 값이 표시되어 있으면 k로 대체하고 k를 증가시킨다. 텍스트 T의 문자들을 처음부터 2개씩 읽으면서 테이블에 저장된 값으로 바꾸어 T'을 생성한다. 카운트 정렬 방법은 기수 정렬 방법보다 간단하게 인코딩하기 때문에 속도가 훨씬 빠르지만 공간을 더 많이 사용한다.

DNA 스트링은 문자집합의 크기가 4로 고정되어 있으므로 세 번째 재귀단계까지 진행되었을 경우 문자집합의 크기는 256이며, 테이블은 260K 정도의 공간만을 사용하므로 이 정렬 방법을 사용할 수 있다. 그러나 네 번째 재귀단계에서는 문자집합의 크기가 최대 65536이 되므로 테이블을 이용하면 최대 16G의 공간이 필요하다. 따라서 네 번째 재귀단계부터는 문자집합의 크기가 너무 크기 때문에 카운트 정렬 방법을 사용할 수 없다.

Kärkkäinen과 Sanders[10]의 알고리즘에 사용하면 성능 향상이 있을 것으로 기대되지만, 3개의 문자를 하나의 정수 값으로 인코딩하기 때문에 $|\Sigma|^3$의 공간이 필요하므로 두 번째 재귀단계까지만 적용할 수 있어 전체 시간 복잡도에 큰 영향을 미치지 못한다.

3.2 합병 과정의 성능 향상 방법

먼저 KJP의 합병 과정을 설명하고, 성능 향상을 위한 방법을 설명한다. 홀수 써픽스 배열과 짝수 써픽스 배열을 합병하기 위해서 배열 C를 사용한다. $C[i]$는 $2 \le i \le n/2$ 인 i에 대해서 홀수 써픽스 배열의 인접한 두 홀수 써픽스 $SA_o[i-1]$보다 순서가 느리고 $SA_o[i]$보다 순서가 빠른 짝수 써픽스들의 개수이다. 합병에 사용되는 배열 C를 구하기 위해서는 U 배열의 처음부터 k까지의 구간$(U[1]..U[k])$에서 문자 c가 나타난 개수인 $occ(U, c, k)$의 값을 계산해야 한다. 가장 간단하고 빠르게 계산하는 방법은 문자집합 Σ에 존재하는 문자들이 구간 $U[1]$ 부터 $U[k]$까지 나타난 횟수를 저장하는 것이다. 이 자료구조를 이용하면 $occ(U, c, k)$를 $O(1)$ 시간에 계산할 수 있지만, 이 자료구조의 사용 공간과 구축 시간은 $O(|\Sigma| \cdot n)$으로 문자집합의 크기에 비례한다. KJP는 매 재귀 단계마다 문자집합의 크기가 기하급수적으로 증가하기 때문에 이 자료구조를 이용하면 구축 시간과 사용 공간이 크게 증가하게 되어 사용할 수 없다.

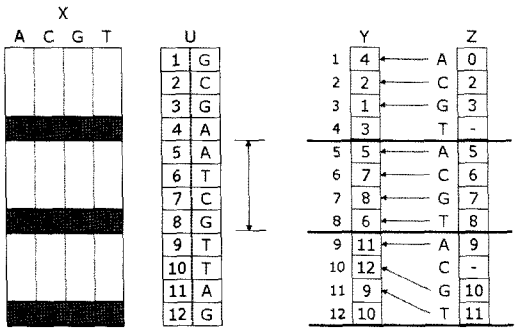

그림 1 KJP의 합병에 사용되는 자료구조

이 문제를 해결하기 위해서 Kim, Jo 및 Park은 문자집합의 크기에 상관없이 $O(n)$ 공간을 사용하면서 비교적 빠른 $O(\log|\Sigma|)$ 시간에 $occ(U, c, k)$를 계산할 수 있는 Sim, Kim, Park, 및 Park[13] 알고리즘의 자료구조를 사용하였다. 이 자료구조는 배열 U를 $|\Sigma|$의 길이를 가진 블럭으로 나누고 매 블럭까지 누적된 c의 개수를 배열 X에 저장한다. $|\Sigma|$ 구간 내에서 c의 개수를 이진 검색을 통해 계산하기 위해서 배열 Y, Z를 구축한다. Sim, Kim, Park 및 Park의 자료구조를 사용하여 $occ(U, c, k)$를 계산하기 위해서는 먼저 k가 속한 블럭 바로 이전 블럭까지 c가 나타난 개수를 배열 X를 읽어서 $O(1)$ 시간에 구하고, k가 속한 블럭 내에서는 배열 Y, Z를 사용해서 이진 검색을 하여 $O(\log|\Sigma|)$ 시간에 c의 개수를 구한 후, 두 값을 더한다. 예를 들어 그림 1에서 문자집합의 크기가 4인 배열 U에서 $occ(U, T, 11)$을 구하는 경우 11은 세 번째 블록에 속해 있으므로 두 번째 블럭 ($U[1]..U[8]$)까지 T의 개수를 배열 X를 이용해서 구한다. 그리고 $U[9]$에서 $U[11]$사이의 T의 개수인 2는 배열 Y, Z를 이용해서 이진 검색을 하여 구한다. 따라서 $occ(U, T, 11)$은 1+2=3이고 $O(\log|\Sigma|)$ 시간에 계산할 수 있다. 이진 검색을 위한 자료구조를 추가로 구축해야 하며 시간복잡도 $O(\log|\Sigma|)$에 숨은 상수가 큰 단점이다.

DNA 스트링에 대한 합병을 빠르게 할 수 있는 방법을 설명한다. 문자집합의 크기가 작은 경우 $O(|\Sigma|)$ 시간에 $occ(U, c, k)$를 계산하는 방법을 사용할 수 있다. $O(\log|\Sigma|)$ 시간에는 길이가 $|\Sigma|$인 구간 내에서 이진 검색을 하지만 $O(|\Sigma|)$ 시간에 계산하는 경우에는 길이가 $|\Sigma|$인 구간의 시작부터 $U[k]$까지 배열을 읽으면서 문자 c의 개수를 구한다. 그림 1에서 예를 들면, $occ(U, c, k)$를 계산할 때 $U[9]$에서 $U[11]$사이의 T의 개수를 구하기 위해서 배열 Y, Z를 이용하여 이진 검색을 하지 않고, $U[9]$에서 $U[11]$까지 배열을 읽어서 T의 개수를 구한다. 이 방법은 추가의 자료구조를 구축하지 않아도 되므로 구축 시간이 단축되며, DNA 스트링의 경우에는 문자집합의 크기가 작아서 이진 검색에 느리지 않다.

DNA 스트링은 문자집합의 크기가 4로 고정되어 있으므로 첫 번째 재귀 단계에서 이 방법을 사용할 수 있다. 첫 번째 재귀 단계에서는 이진 검색을 하는 것보다 배열을 읽으면서 값을 계산하는 것이 빨랐다. 그 이유는 이진 검색의 시간 복잡도 $O(\log|\Sigma|)$의 숨은 상수가 직접 배열을 읽는 시간 복잡도 $O(|\Sigma|)$의 숨은 상수보다 크고, 문자집합의 크기가 16으로 작아서 직접 배열을 읽는 시간이 이진 검색 시간에 비해 크게 느리지 않기 때문이다. 또한, 배열 Y, Z를 구축하지 않아도 되므로 전체 합병 시간을 줄일 수 있다. 두 번째 재귀단계에서는 문자집합의 크기가 최대 256까지 증가하기 때문에 이 방법을 사용할 수 없다.

4. 성능 분석 및 다른 알고리즘들과의 성능 비교

4.1에서 KJP의 인코딩 및 합병 시간과 본 논문에서 제안하는 인코딩 및 합병 시간을 실험을 통해 비교하여 본 논문에서 제안하는 방법을 사용하는 경우 성능향상이 있음을 보인다. 4.2에서 본 논문의 방법을 사용하여 써픽스 배열을 구축했을 경우 KJP에 비해 성능이 향상되었음을 보인다. 마지막으로 4.3에서 본 논문에서 제안한 알고리즘과 다른 알고리즘들의 써픽스 배열 구축 속도를 비교한 결과 대부분의 경우 본 논문에서 제안한 알고리즘이 가장 빠름을 보인다.

4.1 인코딩 및 합병 시간 비교

첫 번째, 두 번째, 세 번째 재귀 단계에서의 인코딩 시간을 실험을 통해서 비교해 보았다. 실험에 사용된 텍스트는 문자집합의 크기가 4이고 길이가 1M, 5M, 10M, 30M, 50M인 랜덤 스트링과 길이가 3.2M, 3.6M, 4.7M, 12.2M, 16.9M, 31.0M, 35.6M인 DNA 스트링이다. DNA 스트링은 NCBI사의 홈페이지에서 있는 것을 사용하였다. Proposed는 논문의 내용을 적용한 알고리즘을 의미한다. 표 1은 랜덤 스트링에 대한 첫 번째, 두 번째, 세 번째 재귀 단계의 인코딩 시간을 비교한 결과와 DNA 스트링에 대한 첫 번째, 두 번째, 세 번째 재귀 단계의 인코딩 시간을 비교한 결과이다.

문자집합의 크기가 4인 랜덤 스트링과 DNA 스트링을 이용해서 첫 번째, 두 번째, 세 번째 재귀 단계의 인코딩 시간을 비교한 결과 KJP에서 사용하는 기수 정렬 방법을 사용하는 방법보다 본 논문에서 제안한 카운트 정렬 방법을 이용하는 방법을 사용할 경우 약 5.6배에서 10배 정도 더 빠르게 인코딩을 수행하는 것을 볼 수 있다.

첫 번째 재귀 단계의 합병 시간을 실험을 통해 비교해 보았다. 인코딩과 마찬가지로 문자집합의 크기가 4이고 길이가 1M, 5M, 10M, 30M, 50M 인 랜덤 스트링과 길이가 3.2M, 3.6M, 4.7M, 12.2M, 16.9M, 31.0M,

표 1 위의 표는 랜덤 스트링에 대한 KJP의 인코딩 시간과 본 논문에서 제시한 인코딩 시간을 비교한 결과이고 아래의 표는 DNA 스트링에 대한 KJP의 인코딩 시간과 본 논문에서 제시한 인코딩 시간을 비교한 결과이다. (단위 : 초)

	1M	5M	10M	30M	50M
	$\|\Sigma\|$=4 (1, 2, 3 재귀 단계)				
KJP 인코딩	0.182	1.015	2.016	5.859	10.063
Proposed 인코딩	0.032	0.109	0.203	0.766	1.188
비 율	17.6%	10.7%	10.1%	13.1%	11.8%

	mito.nt	vector	ecoli.nt	yeast.nt	month.est human	month.est mouse	igseqnt
KJP 인코딩	0.578	0.656	0.859	2.234	3.125	5.703	6.578
Proposed 인코딩	0.093	0.078	0.125	0.313	0.406	0.766	0.891
비 율	16.1%	11.9%	14.6%	14.0%	13.0%	13.4%	13.5%

35.6M인 DNA 스트링을 이용해서 실험하였다. 표 2는 랜덤 스트링을 이용해서 첫 번째 재귀 단계의 합병 시간을 실험을 통해서 비교한 결과와 DNA 스트링을 이용해서 첫 번째 재귀 단계의 합병 시간을 비교한 결과이다.

본 논문에서 제안한 방법인 배열 Y, Z를 구축하지 않고 이진 검색 대신 배열을 읽는 방법을 사용한 첫 번째 재귀 단계의 합병 시간은 KJP의 합병 시간에 비해 1.2배에서 1.6배 정도의 성능 향상이 있음을 알 수 있다. DNA 스트링의 경우에도 1.4배에서 1.7배 정도 성능 향상이 있었다.

4.2 써픽스 배열 구축 시간의 비교

KJP를 이용하여 써픽스 배열을 구축하는 시간과 본 논문에서 제안한 방법을 사용해서 써픽스 배열을 구축하는 시간을 측정했다. 문자집합의 크기가 4이고 길이가 1M, 5M, 10M, 30M, 50M 인 랜덤 스트링과 길이가 3.2M, 3.6M, 4.7M, 12.2M, 16.9M, 31.0M, 35.6M인 DNA 스트링을 이용해서 실험하여 그 결과를 표 3에 나타내었다. 실험을 통해 비교한 결과 랜덤 스트링과 DNA 스트링 모두 본 논문에서 제안한 방법을 사용했을 경우 인코딩 시간과 합병 시간이 단축되어 KJP보다 1.3배에서 1.6배 정도 더 빠르게 써픽스 배열을 구축하였다.

4.3 다른 알고리즘들과의 써픽스 배열 구축 시간 비교

$O(n\log n)$ 알고리즘인 Manber와 Myers[7]의 알고리즘(MM), Larsson과 Sadakane[9]의 알고리즘(LS), 대표적인 선형시간 알고리즘인 Kärkkäinen과 Sanders [10]의 알고리즘(KS), $O(n\log\log n)$ 시간 알고리즘인 KJP와 본 논문에서 제안한 알고리즘(Proposed)의 써픽스 배열 구축 속도를 비교했다. 실험에 사용된 텍스트는 문자집합의 크기가 4이고 길이가 1M, 5M, 10M, 30M, 50M인 랜덤 스트링과 길이가 3.2M, 3.6M, 4.7M, 12.2M, 16.9M, 31.0M, 35.6M인 DNA 스트링이다.

그림 2는 랜덤 스트링에 대한 써픽스 배열 구축 시간을 비교한 결과이다. Manber와 Myers 알고리즘은 가장 느리게 써픽스 배열을 구축하였다. 시간 복잡도의 영향이 나타나 텍스트가 커질수록 구축 속도가 느려진다. Larsson과 Sadakane 알고리즘은 Manber와 Myers 알고리즘보다는 빠르지만 역시 시간 복잡도의 영향이 나타나 텍스트가 커질수록 구축 속도가 느려진다. Kärkkäinen과 Sanders 알고리즘은 선형시간 알고리즘이지만 $O(n\log\log n)$ 시간 알고리즘인 KJP보다 느렸다. 그 이

표 2 위의 표는 랜덤 스트링에 대한 KJP의 합병 시간과 본 논문에서 제시한 합병 시간을 비교한 결과이고, 아래의 표는 DNA 스트링에 대한 KJP의 합병 시간과 본 논문에서 제시한 합병 시간을 비교한 결과이다. (단위 : 초)

	1M	5M	10M	30M	50M
	$\|\Sigma\|$=4 (1 재귀 단계)				
KJP 합병	0.407	1.797	3.672	12.219	20.969
Proposed 합병	0.250	1.422	2.922	9.469	16.860
비 율	61.4%	79.1%	79.6%	77.5%	80.4%

	mito.nt	vector	ecoli.nt	yeast.nt	month.est human	month.est mouse	igseqnt
KJP 합병	1.219	1.672	1.812	5.203	7.156	14.734	18.328
Proposed 합병	0.828	0.953	1.266	3.297	4.140	9.110	10.453
비 율	67.9%	57.0%	69.9%	63.4%	57.9%	61.9%	57.0%

표 3 위의 표는 랜덤 스트링에 대한 KJP과 본 논문에서 제시한 방법의 써픽스 배열 구축 시간을 비교한 결과이고, 아래의 표는 DNA 스트링에 대한 KJP과 본 논문에서 제시한 방법의 써픽스 배열 구축 시간을 비교한 결과이다. (단위 : 초)

	1M	5M	10M	30M	50M
	$\|\Sigma\|=4$				
KJP	1.250	6.703	13.922	44.078	74.203
Proposed	0.813	4.859	10.140	32.406	55.563
비 율	65.0%	72.5%	72.8%	73.5%	74.9%

	mito.nt	vector	ecoli.nt	yeast.nt	month.est human	month.est mouse	igseqnt
KJP	4.344	5.203	6.765	19.203	26.000	52.282	61.484
Proposed	3.140	3.562	4.969	14.109	18.890	38.625	44.094
비 율	72.3%	68.5%	73.5%	73.5%	72.7%	73.9%	71.7%

유는 3개의 문자를 인코딩하기 때문에 시간 복잡도 $O(n)$에 숨은 상수가 크고, $\log\log n$은 실제로 매우 작은 값이므로 $O(n)$에 숨은 상수와 비교할 수 있기 때문이다. 본 논문에서 제안한 알고리즘은 KJP보다 빠르게 써픽스 배열을 구축하였으며, 텍스트의 크기가 작은 일부 경우를 제외하면 가장 빠르게 써픽스 배열을 구축하였다.

그림 3은 DNA 스트링에 대한 써픽스 배열 구축 속도를 비교한 결과이다. Manber와 Myers 알고리즘은 써픽스 배열을 가장 느리게 구축하였다. 이 알고리즘은 매 단계에서 이미 정렬된 써픽스들을 포함한 모든 써픽스들을 읽어서 정렬하기 때문에 속도가 느리다. Larsson과 Sadanake 알고리즘은 $O(n\log n)$ 시간 알고리즘이지만 몇몇 DNA 스트링에 대한 써픽스 배열을 가장 빠르게 구축하고 있다. 그 이유는 DNA 스트링은 랜덤 스트링과는 달리 최장 공통 접두사 (longest common prefix)의 평균값이 작아서 처음 몇 개의 문자들만을 정렬하면 전체 써픽스들을 정렬할 수 있기 때문이다. 하지만 텍스트의 크기가 커질수록 시간 복잡도의 영향이 나타남을 알 수 있다. Kärkkäinen과 Sanders 알고리즘은 선형 시간 알고리즘이지만 Manber와 Myers 알고리즘을 제외한 다른 알고리즘들보다 느리게 써픽스 배열을 구축하고 있다. $O(n\log\log n)$ 시간 알고리즘인 KJP은 비교적 써픽스 배열을 빠르게 구축하고 있지만 Larsson과 Sadakane 알고리즘보다는 느리다. 본 논문에서 제안한 알고리즘을 사용한 경우 최장 공통 접두사의 평균값이 크고, 텍스트의 크기가 커질수록 써픽스 배열을 빠르게 구축한다. 랜덤 스트링과 DNA 스트링을 이용한 실험을 통해 본 논문에서 제안한 방법은 KJP보다 빠르며, 다른 알고리즘들에 비해 전체적으로 가장 좋은 성능을 보임을 알 수 있다.

5. 결 론

본 논문에서는 DNA 스트링의 써픽스 배열을 구축하는 빠른 알고리즘을 제안하였다. KJP는 고정길이 문자 집합에 대해 써픽스 배열을 빠르게 구축하지만, DNA 스트링에 대한 써픽스 배열을 빠르게 구축하는 최적의 알고리즘은 아니었다. 본 논문에서는 DNA 스트링의 문자집합의 크기가 4로 고정되어 있는 사실을 이용해서 KJP의 인코딩 과정과 합병 과정의 성능을 크게 향상시킬 수 있었다.

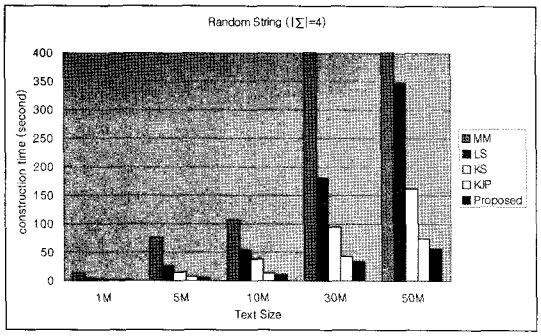

그림 2 랜덤 스트링에 대한 써픽스 배열 구축 시간 비교 (단위 : 초)

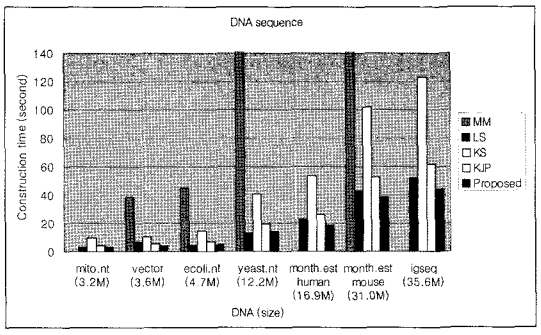

그림 3 DNA 스트링에 대한 써픽스 배열 구축 시간 비교 (단위 : 초)

첫 번째, 두 번째, 세 번째 재귀 단계의 인코딩 과정에서 기존의 기수 정렬 대신 카운트 정렬 방식을 이용하여 빠르게 수행하였으며, 첫 번째 재귀 단계에서 이진 검색을 하지 않고 단순히 배열을 읽는 방법을 사용하여 자료구조 구축 시간과 검색 시간을 줄여 합병을 빠르게 수행하였다. 본 논문에서 제안한 방법을 이용해서 써픽스 배열을 구축한 결과 KJP보다 랜덤 스트링에서 65.0%~74.9%, DNA 스트링에서 68.5%~73.9% 정도로 구축 시간이 단축되었고, 다른 알고리즘들과 DNA 스트링에 대한 써픽스 배열의 구축 시간을 비교한 결과 대부분의 경우 가장 빠르게 써픽스 배열을 구축하였다.

참 고 문 헌

[1] D. Kim, J. Jo, H. Park, A fast algorithm for constructing suffix arrays for fixed-size alphabet, Workshop on Experimental and Efficient Algorithms, LNCS 3059, pp. 301-314, 2004.

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

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

[4] E. Ukkonen, On-line construction of suffix trees, Algorithmica, vol. 14, pp. 249-260, 1995.

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

[6] 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.

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

[8] D. Gusfield, An "Increase-by-one" approach to suffix arrays and trees, manuscript, 1990.

[9] N. Larsson and K. Sadakane, Faster suffix sorting, Manuscript, pp. 1-20, 1999.

[10] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In Proc. 30th International Colloquium on Automata, Languages and Programming, LNCS 2719, pp. 943-955, 2003.

[11] D. Kim, J. 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, S. Aluru. Space-efficient linear time construction of suffix arrays, Journal of Discrete Algorithms, 3(2-4): pp. 143-156, 2005.

[13] J. Sim, D. Kim, H. Park and K. Park, Linear-time search in suffix arrays, Australasian Workshop on Combinatorial Algorithms, pp. 139-146, 2003.

조 준 하

2003년 8월 부산대학교 정보컴퓨터공학과(공학사). 2005년 8월 부산대학교 컴퓨터공학과(공학석사). 2005년 9월~현재 (주)소암시스텔 연구소 Core 개발팀 주임연구원. 관심분야는 컴퓨터 알고리즘, 스트링 처리 및 검색

김 남 희

2007년 2월 한양대학교 전자전기컴퓨터공학과(공학사). 2007년 3월~현재 한양대학교 전자컴퓨터통신공학과 석사과정 관심분야는 컴퓨터 알고리즘, 스트링 처리 및 검색

권 기 룡

정보과학회논문지 : 시스템 및 이론 제 34 권 제 5 호 참조

김 동 규

정보과학회논문지 : 시스템 및 이론 제 34 권 제 5 호 참조