압축된 써픽스 배열 구축 알고리즘의 성능 분석

박치성[O1] 조준하[1] 심정섭[2] 김동규[3]
[1] 부산대학교 컴퓨터공학과
[2] 인하대학교 컴퓨터공학과
[3] 한양대학교 전자통신컴퓨터공학부
[1] {cspark, jhjo}@islab.ce.pusan.ac.kr
[2] jssim@inha.ac.kr
[3] dqkim@hanyang.ac.kr

Performance Analysis of Construction Algorithms for Compressed Suffix Arrays

Chi Seong Park[O1] Junha Jo[1] Jeong Seop Sim[2] Dong Kyue Kim[3]
[1] Dept. of Computer Science & Engineering, Pusan National University, Pusan 609-735, Korea
2 Dept. of Computer Science & Engineering, Inha University, Incheon 402-751, Korea
[3] Dept. of Electronics and Computer Engineering, Hanyang University, Seoul 133-791, Korea

요 약

써픽스 배열은 사전적 순서로 정렬된 써픽스들의 인덱스를 저장한 인덱스 자료구조로서, 긴 텍스트에서 반복되는 패턴 검색 시 효율적으로 사용될 수 있다. 하지만 $O(n\log|\Sigma|)$ 비트의 텍스트보다 큰 $O(n\log n)$ 비트 공간을 차지하기 때문에 대용량의 텍스트에 대해서는 큰 공간을 필요로 하는 문제점이 있다. 이를 해결하기 위해 압축된 써픽스 배열이 제안되었지만, 구축 시 이미 만들어진 써픽스 배열을 이용하기 때문에 실제 사용 공간을 줄이지는 못했다. 최근 써픽스 배열 없이 텍스트에서 직접 압축된 써픽스 배열을 구축할 수 있는 두 가지 알고리즘이 개발되었다. 본 논문에서는 이 두 가지 알고리즘을 구현한 후, 구축 시간과 사용 공간 등의 실험을 통해 기존의 써픽스 배열들과의 성능을 비교하고 분석한다.

1. 서 론

패턴(pattern) 검색 문제는 문자집합의 수가 $|\Sigma|$이고, 길이가 n인 텍스트(text) 내에서 길이가 m인 패턴이 존재하는 모든 위치를 찾는 문제이다. 이는 $O(m)$ 시간에 패턴을 전처리(preprocessing)한 후 $O(n)$ 시간에 텍스트에서 검색하거나, 텍스트에 대한 인덱스 자료구조(index data structure)를 $O(n)$ 시간에 구축한 후 $O(m)$ 시간에 패턴을 검색하는 두 가지 방법으로 해결할 수 있다. DNA 시퀀스(sequence) 검색과 같이 패턴의 길이에 비해 텍스트의 길이가 매우 길고, 고정된 텍스트에서 빈번한 검색이 이루어지는 경우에는 전자보다 후자의 방법이 적합하다.

대표적인 인덱스 자료구조로 써픽스 트리(suffix tree)[1, 2, 3, 4]와 써픽스 배열(suffix array)[5, 6, 7, 8, 9]이 있다. 써픽스 트리는 텍스트의 모든 써픽스들을 압축된 트라이(trie) 구조로 표현한 것이며, 써픽스 배열은 사전적 순서(lexicographical order)로 정렬된 써픽스들의 인덱스(index)를 리스트(list)로 표현한 것이다. 써픽스 트리의 경우 McCreight[2], Ukkonen[3] 등이 제안한 알고리즘을 통해 선형 시간(linear time)에 구축(construction)될 수 있으며, 써픽스 배열의 경우에도 Kim et al.[7], Ko and Aluru[8], Kärkkäinen and Sanders[9] 등이 제안한 알고리즘을 통해 선형 시간에 구축 가능하다. 특히, Kim et al.[10], Larsson and Sadakane[11], Manzini and Ferragina[12], Schurmann and Stoye[13] 등은 비록 선형 시간 알고리즘들은 아니지만, 보다 실용적인 써픽스 배열 구축 알고리즘들을 각각 제안하였다.

하지만 기존의 인덱스 자료구조들은 구축 시 $O(n\log n)$ 비트의 사용 공간(working space)을 필요로 하며, 이는 텍스트를 저장하는 $O(n\log|\Sigma|)$ 비트에 비해 매우 크다는 문제점이 있다. 따라서 대용량의 텍스트에 대해서는 보다 적은 공간을 사용하는 인덱스 자료구조가 필요하다.

이에 Munro et al.[14]은 $O(n\log|\Sigma|)$ 비트를 사용하여 선형 시간에 압축된 써픽스 트리(compressed suffix tree)를 구축하는 알고리즘을 제안하였다. 또한 Grossi and Vitter[15]는 $O(n\log|\Sigma|)$ 비트를 사용하는 압축된 써픽스 배열(compressed suffix array)을, Ferragina and Manzini[16]는 $O(n\log|\Sigma|)$ 비트를 사용하는 FM-index 자료구조를 제안하였다. 하지만 Grossi and Vitter와 Ferragina and Manzini는 자료구조 구축 시 이미 구축된 써픽스 배열을 이용하기 때문에 결국 $O(n\log n)$ 비트의 공간을 필요로 하게 된다.

최근 $O(n\log|\Sigma|)$ 정도의 공간 복잡도(space complexity)를 만족하면서, 텍스트에서 직접 압축된 써픽스 배열을 구축할 수 있는 두 가지 알고리즘이 제안되었다. Hon et al.[17]은 Farach[4], Kim et al.[7, 10] 등이 사용하는 odd-even 기법으로, Na[18]는 Kärkkäinen and Sanders[9]의 skew 기법으로 텍스트에서 직접 압축된 써픽스 배열을 구축한다.

본 논문에서는 이 두 가지 알고리즘들을 구현하여 기존의 써픽스 배열들과 그 성능을 비교한다. 구축 시간과 구축 시 최대 사용 공간, 구축된 써픽스 배열의 크기 등, 이론적인 시간 및 공간 복잡도가 아닌 실제 성능의 비교를 통해 압축된 써픽스 배열의 효율성을 판단한다.

본 논문은 총 3장으로 구성된다. 먼저 2장에서 텍스트에서 직접 압축된 써픽스 배열을 구축하는 두 가지 알고리즘을 간단히 살펴본 후, 3장에서 실험을 통해 기존의 써픽스 배열과 2장에서 언급된 두 가지 알고리즘의 성능을 비교한다.

2. 압축된 써픽스 배열 구축 알고리즘

본 장에서는 압축된 써픽스 배열 구조에 필요한 Ψ함수를 정의한 후, 텍스트에서 직접 압축된 써픽스 배열을 구축하는 두 가지 알고리즘이 각각 사용하는 기법을 살펴보고, 구현 시 발생하는 문제점과 해결 방법 등을 간단히 언급한다.

텍스트 $T[1..n]$의 써픽스 배열을 SA_T, i번째 써픽스를 S_i라고 할 때, S_i의 Ψ값은 SA_T에서 S_{i+1}의 인덱스이며, 함수 Ψ는

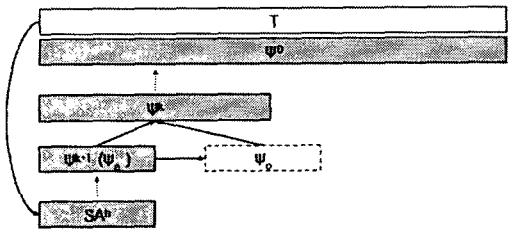

그림 1. 압축된 써픽스 배열의 구조

아래와 같이 정의된다.

$$\Psi^h[i]=\begin{cases} SA_T^{h-1}[SA_T[i]+1] & (\text{if } SA_T[i]\neq n) \\ SA_T^{h-1}[1] & (\text{otherwise}) \end{cases}$$

이 때, n개의 Ψ 값은 unary coding과 같은 방법을 통해 $O(n\log|\Sigma|)$ 비트에 압축하여 저장할 수 있다.

2.1 Odd-even 기법

odd-even 기법은 먼저 짝수 번째 써픽스들에 대한 자료구조를 구축하고, 이를 이용하여 홀수 번째 써픽스들에 대한 자료구조를 구축한 뒤, 두 자료구조를 합병(merge)하여 전체 써픽스들에 대한 자료구조를 구축하는, 재귀적 분할 정복 기법(divide-and-conquer)으로 진행된다. Hon et al.[17]은 이와 같은 odd-even 기법을 사용하여 다음과 같이 텍스트 T에서 직접 압축된 써픽스 배열을 구축한다.

$h=\lceil \log_2 \log_{|\Sigma|} n \rceil$이라 하고, T의 $2^k(0\le k\le h)$개의 캐릭터(character)를 하나의 캐릭터로 합쳐(concatenation) 만들어지는 텍스트를 T^k라 하면, 먼저 h 단계에서 T^h에 대한 써픽스 배열 SA^h를 만들고, 이를 이용하여 Ψ^h를 구축한다. $\Psi^{k+1}=\Psi_e^k$이므로 $h-1$ 단계에서부터 0 단계까지 이전 단계의 Ψ^{k+1}, 즉 Ψ_e^k를 이용하여 Ψ_o^k를 구축하고, Ψ_e^k와 Ψ_o^k를 합병하여 Ψ^k를 구축하는 과정을 반복한다. 이 때 각 단계의 Ψ_e^k만 저장하고 Ψ_o^k는 저장하지 않는다. 최종적으로 0단계가 완료되면 그림 1과 같이 SA^h와 각 단계의 Ψ_e^k로 이루어진 압축된 써픽스 배열이 구축된다. 전체 구축 시간은 $O(n\log\log|\Sigma|)$, 사용 공간은 $O(n\log|\Sigma|)$ 비트이다.

2.2 Skew 기법

skew 기법 역시 odd-even 기법과 유사한 재귀적 분할 정복 기법을 사용한다. 하지만 전체 써픽스들을 짝수 번째와 홀수 번째 써픽스로 분할하는 odd-even 기법과 달리 skew 기법은 써픽스의 인덱스 i에 대하여 $i \bmod 3\neq 0$을 만족하는 써픽스 S_{12}와 $i \bmod 3=0$을 만족하는 써픽스 S_3으로 분할한다. 따라서 skew 기법은 먼저 S_{12}에 대한 자료구조를 구축하고, 이를 이용하여 S_3에 대한 자료구조를 구축한 뒤, 두 자료구조를 합병하는 순서로 진행된다. Na[18]는 odd-even 기법과 유사한 순서로 진행하되 이와 같은 skew 기법을 사용하여 텍스트 T에서 직접 압축된 써픽스 배열을 구축한다.

$h=\lceil \log_3 \log_{|\Sigma|} n \rceil$이라 할 때, 역시 h 단계에서 써픽스 배열 SA^h를 만들고, 이를 이용하여 Ψ^h를 구축한다. 알고리즘의 특성에 따라 $\Psi^{k+1}=\Psi_{12}^k$를 만족하게 되므로 $h-1$ 단계에서부터 0 단계까지 이전 단계의 Ψ^{k+1}, 즉 Ψ_{12}^k를 이용하여 Ψ_3^k을 구축하고, Ψ_{12}^k와 Ψ_3^k을 합병하여 Ψ^k를 구축하는 과정을 반복한다. 최종적으로 0단계가 완료되면 Hon et al.과 유사한 전체 구축 시간은 $O(n)$, 사용 공간은 $O(n\log|\Sigma|\cdot\log_{|\Sigma|}n)$ 비트이다.

2.3 구현 시의 문제점 및 해결

Hon et al.의 압축된 써픽스 배열 구축 알고리즘을 구현하던 중 다음과 같은 몇 가지 문제들을 발견하였다. 첫 번째는 h 단계에서 T^h의 캐릭터, 즉 T의 2^h개의 캐릭터가 합쳐진(concatenated) 캐릭터 하나의 크기가 일반적인 word 단위(32비트)를 넘는 경우가 발생하여 기존의 알고리즘으로 써픽스 배열 SA^h를 구축할 수 없다는 것이다. 두 번째 문제는 역시 합쳐진 캐릭터 하나의 크기로 인해, Ψ^h를 구축하는 과정 중 내부적으로 사용되는 stable 정렬 시 필요한 연산의 수가 많고, 그 과정에 필요한 자료구조의 크기가 생성 불가능할 정도로 커지는 경우가 발생할 수 있다는 것이다. 세 번째 문제는 합병 과정 중 필요한 backward 검색 연산이 사용하는 auxiliary 자료구조의 구현이 쉽지 않다는 것이다. 네 번째 문제는 구축 과정에서 $O(n\log|\Sigma|)$ 비트에 압축되어 저장되어 있는 Ψ^h를 사용하기 위해 succinct 표현의 기본 함수인 rank와 select 함수를 반복 수행해야 하는 것인데, 이 rank와 select 함수의 성능이 압축된 써픽스 배열의 구축 시간에 큰 영향을 미친다는 것이다.

첫 번째와 두 번째 문제에 대해서는 캐릭터들의 인코딩(encoding) 과정을 통해 문자집합의 수를 줄이는 방법을 사용하였다. 인코딩 과정 중 word 단위를 넘어서는 저장 공간이 필요한 경우 C99 표준에서 지원하는 long long integer 타입(64비트)을 사용하였으며, 인코딩 과정과 인코딩 후 stable 정렬 과정에 필요한 캐릭터들의 정렬과 검색 연산 내에는 $O(n\log n)$ 시간의 정렬 알고리즘과 $O(n\log|\Sigma|)$ 시간의 검색 알고리즘을 사용하였다. 정렬 알고리즘을 수정하는 방법을 통해 stable 정렬 과정에 필요한 자료구조의 크기를 더 줄일 수도 있었지만, 구축 시간이 지나치게 길어지는 문제점으로 인해 사용하지 않았다. 세 번째 문제에 대해서는 auxiliary 자료구조 구축 대신 케이스(case) 구분 없이 이진(binary) 검색을 사용하였으며, 네 번째 문제에 대해서는 최근 발표된 Kim et al.[19]에서 제안한 바이트(byte) 기반의 빠른 rank와 select 함수를 사용하여 그 영향을 최소화하였다.

3. 실험 및 결과

본 장에서는 실험을 통해 2장에서 언급된 두 가지 압축된 써픽스 배열 구축 알고리즘과, 기존의 써픽스 배열 구축 알고리즘 다섯 가지와의 성능을 비교한다.

3.1 실험 환경

실험에 사용된 시스템은 표 1과 같다.

CPU	Intel Pentium IV 3.00GHz
RAM	512M
OS	Fedora Core 4
Compiler	gcc, g++

표 1. 실험에 사용된 시스템

실험에 사용된 알고리즘들의 경우 저자가 공개한 소스코드를 되도록 사용하되, 공개되어 있는 코드가 존재하지 않는 경우에는 직접 구현하였다. 또한 모든 실험은 주기억장치만을 사용하였으며, 각각 10번씩 반복 측정하여 그 평균값을 결과로 사용하였다.

3.2 실험 결과

그림 2는 길이가 {1M, 5M, 10M}이고 문자집합의 크기가 {8, 20}인 랜덤(random) 텍스트에 대해 압축된 써픽스 배열과 써픽스 배열의 구축 시간(construction time)을 비교한 결과이다. 2장에서 언급된 압축된 써픽스 배열 odd-even, skew 외에 odd-even(kjp), skew(ks), qsufsort, deep-shallow, bpr은 각각

Kim et al.[10], Kärkkäinen and Sanders[9], Larsson and Sadakane[11], Manzini and Ferragina[12], Schurmann and Stoye[13]의 써픽스 배열 구축 알고리즘을 의미한다. 그림에서

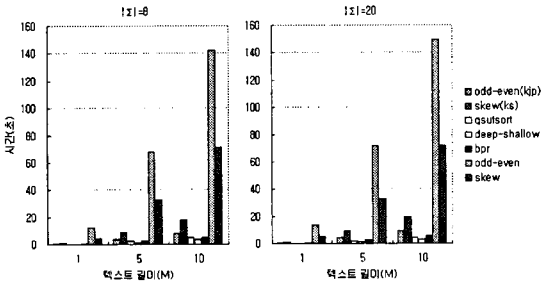

그림 2. (압축된) 써픽스 배열 구축 시간 비교 결과

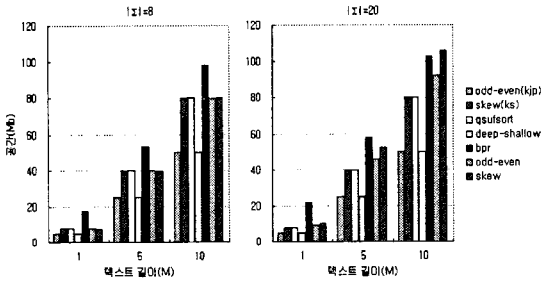

그림 3. (압축된) 써픽스 배열 구축 시 사용 공간 비교 결과

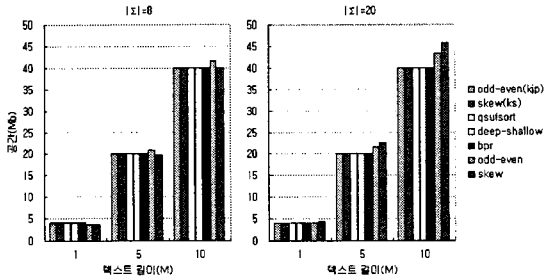

그림 4. (압축된) 써픽스 배열의 크기 비교 결과

알 수 있듯이 실험 결과 압축된 써픽스 배열 알고리즘들의 구축 속도가 기존의 써픽스 배열 알고리즘들에 비해 현저히 느리다는 것을 확인할 수 있었다. 특히 odd-even의 경우 구현을 위해 추가적으로 사용한 인코딩 과정과 stable 정렬 연산 때문에 skew의 2배 정도 되는 시간을 필요로 하였으며, skew의 경우에도 기존의 써픽스 배열 알고리즘들에 비해 4배 이상 느렸다.

그림 3은 동일한 랜덤 텍스트에 대해 구축 시 사용된 공간(working space)이 최대가 되는 경우(peak memory usage)를 비교한 결과이다. 문자집합이 큰 경우 odd-even이 skew보다 다소 작은 공간을 사용하는 차이를 보이지만, 두 가지 다 대부분의 경우 bpr을 제외한 기존의 써픽스 배열들에 비해 큰 공간을 사용하고 있었다.

그림 4는 역시 동일한 랜덤 스트링에 대한 압축된 써픽스 배열과 써픽스 배열의 크기(space)를 비교한 결과이다. 써픽스 배열의 경우 텍스트를 제외한 SA 즉 pos 배열만을 측정하였으며, 압축된 써픽스 배열의 경우에도 SA^h와 각 단계의 ψ^h만을 측정하였다. 문자집합이 큰 경우 기존의 써픽스 배열보다 압축된 써픽스 배열의 크기가 오히려 큰데, 이는 h가 작아 SA^h가 차지하는 공간이 크기 때문인 것으로 보인다. 따라서 상대적으로 odd-even보다 h가 작은 skew의 경우 문자집합의 크기가 20일 때 odd-even보다 5% 정도 더 큰 공간을 필요로 하였다.

4. 결론

본 논문은 기존의 써픽스 배열과, 사용 공간의 효율을 목적으로 텍스트에서 직접 구축되는 압축된 써픽스 배열의 성능을 실제 구축 시간과 실제 사용 공간을 중심으로 비교해 보았다. 실험 결과 odd-even과 skew 둘 다 기존의 써픽스 배열에 비해 대부분의 경우 구축 속도와 사용 공간 모두 효율적이지 못함을 확인할 수 있었다. 특히 사용 공간의 효율을 위해 제안된 것임에도 실제 구현 시에 발생하는 문제점들로 인해 구축 시간뿐만 아니라 사용 공간 측면에서도 $O(n \log n)$ 비트의 공간을 사용하는 기존의 써픽스 배열들에 비해 많은 공간을 필요로 한다는 사실에는 주목할 필요가 있다.

참고문헌

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

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

[3] E. Ukkonen, On-Line construction of suffix trees, Algorithmica 14 (1995), 249-260

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

[5] U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Comput. 22, (1993), 935-948

[6] D. Gusfield, An "Increment-by-one" approach to suffix arrays and trees, Report. CSE-90-39, Computer Science Division, University of California, Davis, 1990

[7] D.K. Kim, J.S. Sim, H. Park and K. Park, Linear-Time construction of suffix arrays, Symp. Combinatorial Pattern Matching (2003), 186-199

[8] P. Ko and S. Aluru, Space-efficient linear time construction of suffix arrays, Symp. Combinatorial Pattern Matching (2003), 200-210

[9] J. Kärkkäinen and P. Sanders, Simple linear work suffix array construction, Int. Colloq. Automata Languages and Programming (2003), 943-955

[10] D.K. Kim, J. Jo and H. Park, A fast algorithm for constructing suffix arrays for fixed-size alphabets, Workshop on Efficient and Experiemntal Algorithms (2004), 301-314

[11] N.J. Larsson and K. Sadakane, Faster Suffix Sorting, Technical Report, number LU-CS-TR:99-214, Department of Computer Science, Lund University, Sweden (1999)

[12] G. Manzini and P. Ferragina, Engineering a lightweight suffix array construction algorithm, Algorithmica 40 (2004), 33-50

[13] K. Schürmann and J. Stoye, An incomplex algorithm for fast suffix array construction, Algorithmica Engineering and Experiments (2005)

[14] J.I. Munro, V. Raman and S.S. Rao, Space efficient suffix trees, J. of Algorithms 39 (2001), 205-222

[15] R. Grossi and J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, ACM Symp. Theory of Computing (2000), 397-406

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

[17] W.K.Hon, K. Sadakane and W.K.Sung, Breaking a time-and-space barrier in constructing full-text indices, IEEE Symp. found. Computer Science (2003), 251-260

[18] J.C. Na, Linear-time construction of compressed suffix arrays using O(nlog'n)-bit working space for large alphabets, Symp. Combinatorial Pattern Matching (2005), 57-67

[19] D.K. Kim, J.C. Na, J.E. Kim and K. Park, Efficient implementation of rank and select functions for succinct representation, Workshop on Efficient and Experiemntal Algorithms (2005), 315-327