creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

Full-Text Index 자료구조들의 비교 분석

Comparative Analysis of Full-Text Index Data Structures

2018년 2월

서울대학교 대학원

컴퓨터공학부

공 서 우

Full-Text Index 자료구조들의 비교 분석

Comparative Analysis of Full-Text Index Data Structures

지도교수 박 근 수

이 논문을 공학석사 학위논문으로 제출함
2017년 12월

서울대학교 대학원
컴퓨터공학부
공 서 우

공서우의 공학석사 학위논문을 인준함
2017년 12월

위 원 장 문 병 로 (인)

부위원장 박 근 수 (인)

위 원 김 선 (인)

요약

텍스트 인덱스 자료구조는 주어진 문자열에 대하여 문자열을 가능한 작게 압축하여 저장하는 동시에 패턴 검색과 추출을 처리하는 자료구조를 말한다. 최근 텍스트 데이터의 크기가 나날이 커져감에 따라 대용량 텍스트 데이터의 공간 효율적인 인덱스 자료구조 구축의 중요성이 커지고 있다. 그에 따라 인덱스 자료구조에 대한 다양한 연구들이 진행되었다. 제안된 대표적인 텍스트 인덱스 자료구조로는 FM-Index와 Compressed Suffix Array(CSA), Compressed Suffix Tree(CST), 그리고 Lempel-Ziv 압축 알고리즘을 기반으로 인덱스를 구축하는 LZ-Index등이 있다.

본 논문에서는 첫 번째로 다양한 텍스트 데이터에 대해 인덱스 구축 시간, 인덱스 크기, 메모리 사용량, 패턴 검색 및 추출 시간을 측정하는 실험을 통해 인덱스 자료구조들의 실제 성능을 비교 분석한다. 두 번째로 앞의 분석 결과 압축률이 상대적으로 좋은 FM-Index에 대해 알파벳 사이즈가 작은 경우 압축률을 개선할 수 있는 방법을 알아본다. 실험 결과, 공간 효율적인 인덱스 구축 측면에서 일반 텍스트 데이터는 FM-Index가, 고반복 텍스트 데이터는 LZ-Index가 가장 작은 인덱스를 구축했다.

주요어 : 문자열 인덱싱, 압축, 접미사 배열, 접미사 트리, 패턴 검색
학 번 : 2016-21184

목 차

1. 서론 ····· 1
2. 사전지식 및 실험 데이터 ····· 4
2.1 실험 데이터 ····· 4
2.2 버로우즈-휠러 변환 ····· 5
2.3 Rank/Select 질의 ····· 7
2.4 웨이블릿 트리 ····· 8
3. 대표적인 Full-Text Index 자료구조 ····· 11
3.1 FM-Index ····· 11
3.2 Compressed Suffix Array ····· 13
3.3 Compressed Suffix Tree ····· 15
3.4 Lempel-Ziv 압축 알고리즘을 이용한 Index 자료구조 ····· 16
3.4.1 Lempel-Ziv 압축 알고리즘 ····· 16
3.4.2 LZ-Index ····· 18
4. 실험 결과 및 분석 ····· 20
4.1 인덱스 구축 시간 비교 ····· 20
4.2 인덱스 크기 비교 ····· 22
4.3 메모리 사용량 비교 ····· 24
4.4 패턴 검색 시간 비교 ····· 25
4.5 문자열 추출 시간 비교 ····· 28
4.6 실험 결과 분석 ····· 29
5. 알파벳 사이즈가 작은 경우 FM-Index 압축률 향상법 ·· 30
6. 결론 ····· 35

참고문헌 ····· 36
Abstract ····· 39

1. 서 론

기술의 발전과 더불어 많은 정보가 컴퓨터로 처리되면서 디지털로 사용되고 있는 데이터의 크기가 기하급수적으로 증가하고 있다. 이러한 많은 데이터 중에서도 텍스트 데이터가 가장 많은 비중을 차지하고 있는데, 그 이유는 단지 언어 데이터뿐만 아니라 프로그램 소스코드, 음악 데이터, 생물학적 데이터 등 다양한 정보가 텍스트를 사용하여 저장하기 때문이다.

일반 언어 외에도 다양한 정보를 텍스트로 이용해서 표현하는 이유는 어떤 정보를 텍스트로 바꾸어 저장하면 그로부터 새로운 정보를 추출할 수 있기 때문이다. 이렇게 데이터로부터 새로운 정보를 추출하기 위해 가장 중요한 것은 특정 문자열의 출현 수 파악 및 출현 위치의 비교 분석이다. 예를 들어 최근 많이 사용되고 있는 생물학적 데이터는 DNA, 단백질 또는 유전자 서열과 같이 다양한 생물학적 정보를 A, C, G, T의 텍스트로 표현하고 있는데, 여기서 특정 염기서열의 수, 위치를 파악함으로써 다양한 생물학적 정보와 화학적 구조 등과 같은 정보를 알아낼 수 있다.

이와 같이 텍스트 데이터에서 새로운 정보를 추출하기 위해서 가장 기본적인 것은 문자열 검색이다. 문자열 검색은 전체 텍스트에서 특정 문자열(패턴)을 찾는 것을 말한다. 문자열 검색은 크게 두 가지 방법으로 나누어진다. 하나는 전처리가 되지 않은 텍스트에 대해 순차적으로 모든 텍스트에 대해서 패턴을 검색하는 방법이고 다른 하나는 주어진 텍스트에 대해 인덱스 자료구조를 만드는 것이다. 여기서 Full-Text Index 자료구조란, 주어진 텍스트 $T[1,n]$에 대하여 효율적인 패턴 검색을 도와주는 자료구조를 말한다. 이 자료구조를 사용하면 전체 텍스트를 훑어보지 않고도 패턴을 검색할 수 있다. 따라서 텍스트에 대한 인덱스 자료구조는 기존 검색 방법이 텍스트의 크기가 커질 때 검색 시간이 길어지는 단점을 보완해준다. 하지만 인

덱스 자료구조를 만든다는 것은 말 그대로 텍스트 데이터 이외에 새로운 자료구조가 필요하다는 것을 의미하기 때문에 효율적인 패턴 검색이 가능하며 크기가 작은 인덱스 자료구조 구축을 위해 많은 연구가 진행되어 왔다.

대표적인 텍스트 인덱스 자료구조로는 FM-Index[1][2]와 Compressed Suffix Array(CSA)[3][4], Compressed Suffix Tree(CST)[5], 그리고 Lempel-Ziv 압축 알고리즘을 기반으로 인덱스 자료구조를 만드는 LZ-Index[6][7][8]가 있다. Ferragina와 Manzini[1][2]는 버로우즈-휠러 변환(BWT)[9]을 기반으로 텍스트 인덱스 자료구조를 구축하는 방법을 제안했다. Grossi와 Vitter[3]는 분할과 재귀적 구조의 아이디어를 사용하여 접미사 배열을 압축하는 CSA를 제안했으며 Sadakane[4]는 Grossi와 Vitter가 제안한 CSA를 self-index가 되도록 제안했다. 또한 Sadakane[5]는 LCP와 위의 CSA, 그리고 접미사 트리를 parenthesis를 통해 간결하게 표현하는 방법[13] 등을 사용하여 CST를 제안했다. LZ-Index는 Lempel-Ziv 압축 알고리즘을 사용하여 구축하는 인덱스 자료구조이다. Lempel-Ziv 77 압축 알고리즘[10]은 Lempel과 Ziv가 1977년 고안한 압축 알고리즘으로 다양한 파생형이 존재한다. Navarro[6]는 LZ78 압축 알고리즘을 기반으로 하는 LZ-Index를 제안했고, S.Kreft, Navarro[7]는 일반적인 텍스트 데이터가 아닌 유사한 반복이 많이 발생하는 텍스트 데이터에 대해서 LZ77 압축 알고리즘과 LZ-End 압축 알고리즘을 사용하여 인덱스 자료구조를 구축하는 방법을 제안했다.

본 논문에서는 첫 번째로 다양한 텍스트 데이터를 입력으로 각 자료구조의 구축 시간, 인덱스 크기, 패턴 검색 시간, 메모리 사용량을 측정하는 실험을 통해서 인덱스 자료구조들의 실제 성능을 비교 분석한다. 또한 최근에는 문서 데이터, 소스 코드 등의 버전화된 데이터와 같이 유사하게 반복되는 데이터가 증가하고 있기 때문에 일반적인 텍스트 이외에 유사한 반복이 많은 텍스트 데이터에 대해서도 실험을 진행하여 어떤 성능을 보이는지 확인한다. 두 번째로 텍스트 인덱스 자료구조는 알파벳의 사이즈에 영향을 받

기 때문에 알파벳 사이즈에 따라 인덱스 크기가 변하게 된다. 유전체 데이터의 경우에는 알파벳 사이즈가 4개의 문자 A, C, G, T와 기타 소수의 문자로 매우 작은 편에 속하는데, 이와 같이 텍스트 데이터를 구성하는 알파벳 사이즈가 작은 경우, FM-Index[2]의 웨이블릿 트리를 대체할 수 있는 자료구조들을 제시한다. 그리고 이를 적용하여 인덱스 자료구조의 크기를 줄일 수 있는지 알아본다.

2. 사전 지식 및 실험 데이터

텍스트 인덱스 자료구조는 주어진 문자열 $T[1,n]$에 대하여 문자열 T 를 가능한 작게 압축하여 저장하는 동시에 세 가지 질의 Count, Locate, Extract를 처리하는 자료구조를 말한다. 위의 각 질의는 다음과 같은 일을 처리한다.

- Count : Count 질의는 주어진 패턴이 $P[1,m]$이라고 할 때, 패턴 P가 T에 출현하는 총 개수를 구한다.
- Locate : Locate 질의는 주어진 패턴이 $P[1,m]$이라고 할 때, 패턴 P가 T에 출현하는 위치를 구한다.
- Extract : Extract 질의는 주어진 위치 $[i,j]$에 대하여 $T[i,j]$에 있는 문자열을 추출한다.

또한, 위와 같은 텍스트 인덱스 자료구조가 질의를 수행할 때 주어진 문자열 $T[1,n]$을 사용하지 않으면 이를 self-index 라고 부른다.

2.1 실험 데이터

본 논문의 실험을 위해 사용된 데이터는 http://pizzachili.di.unipi.it/texts.html 에서 제공되는 데이터를 사용한다. 각 데이터는 English, DNA, Einstein, chr20의 이름을 갖는다. English 데이터는 알파벳으로 구성된 일반 텍스트 데이터로 크기별로 50MB, 100MB, 200MB이다. DNA 데이터는 A, C, G, T로 구성된 유전자 염기 서열을 갖고 있는 데이터로 알파벳 A, C, G, T와 일부 특정 문자로 구성되어 있으며, English와 마찬가지로 50MB, 100MB, 200MB로 나누어진다. Einstein 데이터는 유사한 반복이 많은 고반복 텍스트 데이터로 특정 기간 동안 위키피디아의 Albert Einstein 항목 내용 수정이 일어날 때마다의 전체 데이터를 모아둔 것이다. chr20 데이터는 위

의 DNA 데이터가 염기서열 외에 다른 문자가 어느 정도 포함하고 있기에 조금 더 알파벳 사이즈를 줄인 데이터이다. 각 데이터들의 상세한 정보는 [표 1]과 같다.

[표 1] 실험 데이터 상세

	File Size	Alphabet Size	Total character
English	50MB	239	52,428,800
	100MB		104,857,600
	200MB		209,715,200
DNA	50MB	16	52,428,800
	100MB		104,857,600
	200MB		209,715,200
Einstein	446MB	230	467,626,544
chr20	60MB	5	63,025,520

2.2 버로우즈-휠러 변환

버로우즈-휠러 변환[9]은 문자열과 문자열 사이의 변환으로 문자열 전처리 알고리즘이다. 접미사 배열이란 문자열 $T[1,n]$이 있을 때 T에 대한 모든 접미사 $T[i,n]$ $(1 \le i \le n)$을 사전 순으로 정렬한 것을 말한다. 따라서 $SA[]$를 접미사 배열이라고 하면, $SA[j]=i$는 i번째 접미사 $T[i,n]$이 사전 순서로 j번째임을 의미한다.

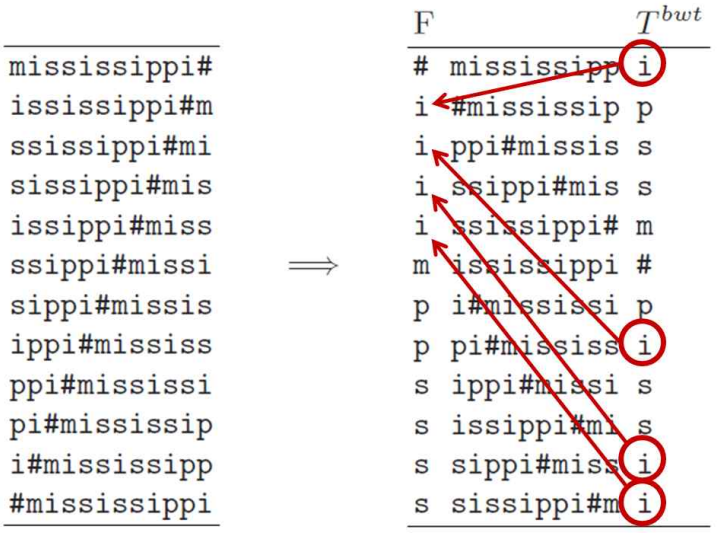
문자열 $T[1,n]$의 버로우즈-휠러 변환은 먼저 T의 마지막에 #이나 $와 같이 T의 전체에 나타나지 않는 특정 문자를 붙여준다. 그리고 T의 회전 쉬프트 연산으로 만들어지는 모든 문자열에 대해서 사전 순서로 정렬하는 것을 버로우즈-휠러 변환이라고 하며 이때 정렬된 문자열들의 마지막 문자

를 차례대로 취한 것을 T^{bwt}라고 한다. [그림 1]은 mississippi#에 버로우즈-휠러 변환을 적용한 예시를 보여준다.

		F	T^{bwt}
mississippi#		# mississipp	i
ississippi#m		i #mississip	p
ssissippi#mi		i ppi#missis	s
sissippi#mis		i ssippi#mis	s
issippi#miss		i ssissippi#	m
ssippi#missi	$\Longrightarrow$	m ississippi	#
sippi#missis		p i#mississi	p
ippi#mississ		p pi#mississ	i
ppi#mississi		s ippi#missi	s
pi#mississip		s issippi#mi	s
i#mississipp		s sippi#miss	i
#mississippi		s sissippi#m	i

[그림 1] mississippi# 의 버로우즈-휠러 변환

이렇게 생성된 T^{bwt}는 몇 가지 중요한 특징을 갖고 있다. 첫 번째 특징은 T^{bwt}만을 가지고도 원래의 문자열 T를 복원을 할 수 있다는 것이고, 두 번째 특징은 버로우즈-휠러 변환 상에서 첫 번째 열인 F 와 마지막 열 T^{bwt} 사이에 연관성이 있다는 것이다. 그 연관성은 [그림 2]와 같이 T^{bwt}에서 어떤 문자가 나타나는 순서와 그 문자로 시작하는 접미사, 즉 F에서의 문자 순서가 동일한 것을 말한다. 이런 특징 때문에 BWT는 압축 효율을 높이는 전처리 알고리즘 중 한 가지로 제시된 변환 방법인 동시에 패턴 검색에도 많이 활용되고 있는 문자열 변환 방법이다.



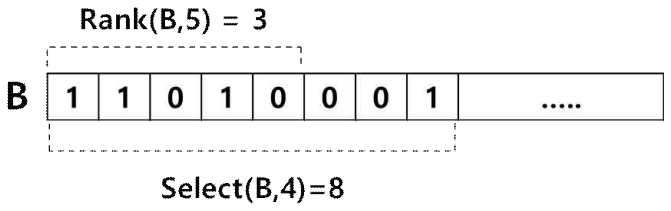
[그림 2] T^{bwt}와 F 사이의 문자 순서 연관성

2.3 Rank/Select 질의

Rank와 Select는 0과 1로 구성된 비트 스트링 $B[1,n]$에 대해서 다음과 같은 일을 수행한다.

- $Rank(B,i)$: $B[1,i]$에 포함된 1의 개수를 계산한다.
- $Select(S,i)$: 비트 스트링 B의 처음부터 i번째 1의 위치를 계산한다.

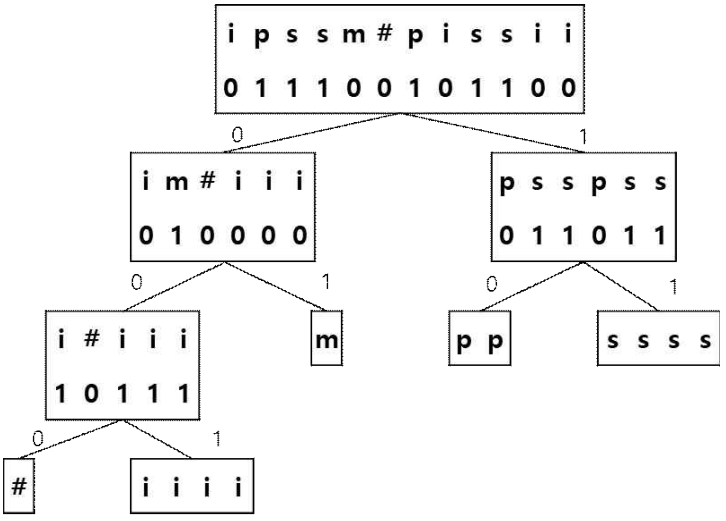
예를 들어 [그림 3]과 같이 비트 스트링이 주어졌을 때, $Rank(B,5)=3$이고 $Select(B,4)=8$이다.



[그림 3] 비트 스트링 $B[1,n]$에 대한 Rank/Select 예시

2.4 웨이블릿 트리

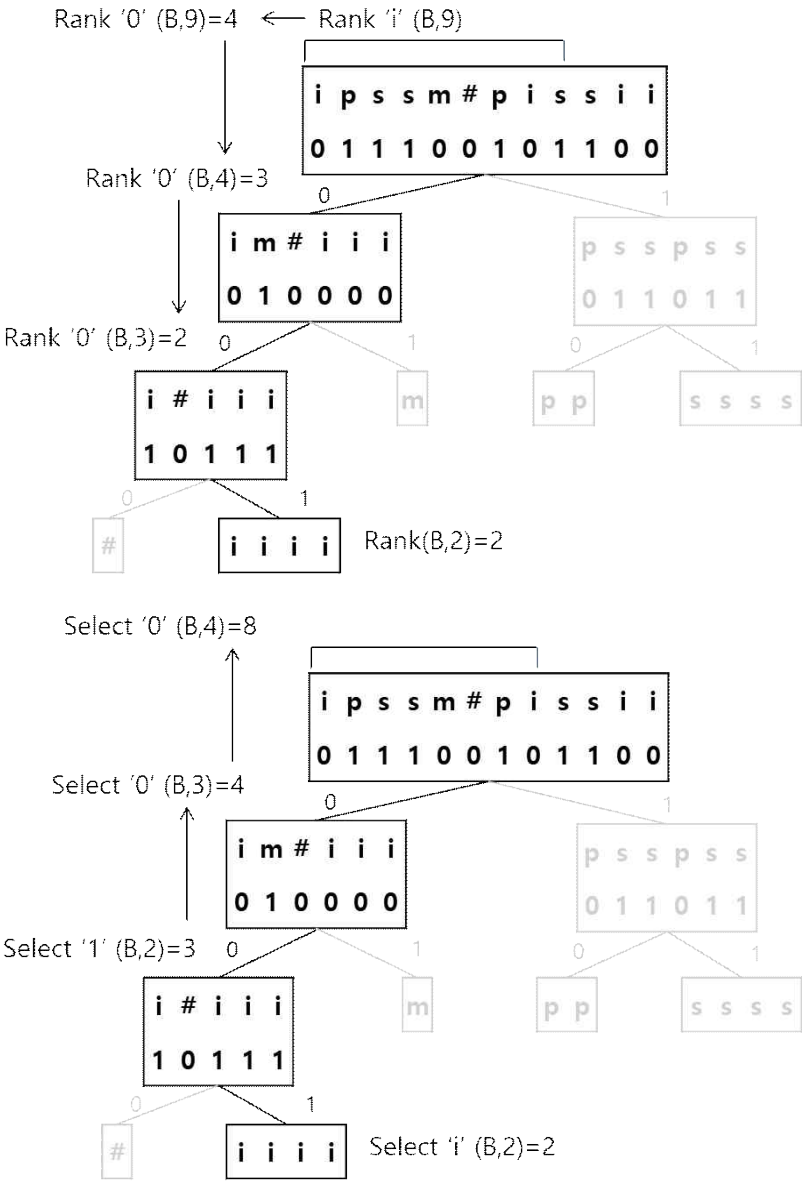
웨이블릿 트리[11]는 다수의 알파벳으로 구성된 스트링에서 Rank/Select 질의를 조금 더 효율적으로 처리하기 위해서 고안된 자료구조이다. 웨이블릿 트리의 구조는 계층적 구조의 비트 스트링으로 이루어지며 비트 스트링에서 Rank 질의가 $O(1)$시간에 처리되므로 웨이블릿 트리를 이용하면 알파벳 사이즈를 σ라고 할 때 Rank 질의는 $O(\log\sigma)$시간에 처리할 수 있게 된다. 웨이블릿 트리의 구축 방법을 단순히 설명하면 알파벳으로 구성된 스트링을 알파벳 집합의 절반은 0으로 표기하고, 나머지 절반은 1로 표기하여 이진 트리 형식으로 표현하는 것이다.



[그림 4] ipssm#pissii 의 웨이블릿 트리

알파벳 집합 Σ으로 이루어진 문자열에 대한 웨이블릿 트리 구축 방법은 다음과 같다. 첫 번째로 알파벳 집합 Σ을 절반으로 나눌 수 있는 문자 c를 선택한다. 그 후 선택된 문자 c를 기준으로 $\Sigma_l = \{a \in \Sigma : a \le c\}$과 $\Sigma_r = \{a \in \Sigma : a > c\}$ 두 개의 알파벳 집합으로 분할한다. 그리고 기존 문자열을 각각 Σ_l로 이루어진 문자열과 Σ_r로만 이루어진 문자열로 분할하여

왼쪽과 오른쪽 자식으로 트리를 만든다. 이 작업을 알파벳 집합을 더 이상 분할할 수 없을 때까지 진행하여 웨이블릿 트리를 만든다. [그림 4]는 2.2에서 보았던 'mississippi#' 의 버로우즈-휠러 변환인 'ipssm#pissii'를 웨이블릿 트리로 만든 결과이다.



[그림 5] wavelet tree에서 Rank/Select질의 수행과정
$Rank_i(B,9)$(위) / $Select_i(B,2)$(아래)

이와 같이 구축한 웨이블릿 트리에서 Rank/Select 질의를 처리하는 과정은 다음과 같다. 웨이블릿 트리에서의 Rank는 기존 Rank 질의에 특정 문자 c에 대한 조건이 추가되어 $Rank_c(B, i)$가 된다. 먼저 주어진 문자 c가 알파벳 집합 Σ에서 어느 집합에 속하는지 확인하고 왼쪽이면 0의 수를, 오른쪽이면 1의 수를 세어 다음 자식 노드에서의 새로운 i값을 계산한다. 이를 반복하여 마지막까지 진행하면 하나의 문자만이 남게 되는데 이때 값이 i위치까지 문자 c가 나타나는 횟수이다. Select는 Rank와 반대로 마지막 노드에서 시작하게 된다. 먼저 마지막 노드에서 주어진 문자 c의 i번째 위치를 구한다. 그리고 부모 노드로 이동하면서 앞서 구한 위치를 Select 질의의 새로운 i값으로 설정하고 이전에 부모의 왼쪽, 오른쪽 자식이었는지 여부에 따라 각각 0과 1로 설정하여 계산한다. 이를 루트 노드까지 진행하면 i번째 문자 c가 나타나는 위치를 계산할 수 있다. [그림 5]는 웨이블릿 트리에서 Rank/Select 질의 과정의 예시이다.

3. 대표적인 Full-Text Index 자료구조

이 장에서는 본 논문에서 비교 분석에 사용할 대표적인 인덱스 자료구조인 FM-Index, CSA, CST, LZ-Index에 대해 살펴본다.

3.1 FM-Index

FM-Index[1][2]는 Ferragina와 Manzini가 제안한 텍스트 인덱스 자료구조로 2.2에서 설명한 버로우즈-휠러 변환(BWT)의 특징으로 얻어지는 역방향 검색에 아이디어를 둔 방법이다.

2.2의 [그림 2]를 통해 버로우즈-휠러 변환을 거친 T^{bwt}에서 나타나는 문자의 순서와 F에서 나타나는 문자 순서가 동일하다는 관계를 알 수 있었다. 위의 두 문자열 사이의 관계를 표현하기 위해 몇 가지를 정의한다.

- $C[\,]$: T^{bwt}에서 각 문자보다 사전 순서가 낮은 문자의 전체 개수를 저장하는 배열
- $Occ(c, i)$: T^{bwt}에서 i번째 위치까지 문자 c가 나타나는 수를 계산하는 함수

이 두 가지를 이용하면, 두 문자열의 관계를 아래와 같이 표현할 수 있다.

$$LF(i) = C[\,T^{bwt}[i]\,] + Occ(\,T^{bwt[i]}, i\,)$$

위 식을 사용하면 i번째 T^{bwt}문자의 F 에서 위치를 계산할 수 있게 되며, 이를 LF-매핑이라고 부른다.

예를 들어 [그림 1]에서 $LF(10) = C[s] + Occ(s, 10) = 12$를 계산할 수 있고 따라서 $T^{bwt}[10] = F[12]$와 매핑 되는 것을 알 수 있다. 또한 LF-매핑을 사용하면 $T[k] = T^{bwt}[i]$일 때, $T[k-1] = T^{bwt}[LF(i)]$가 성립하게 되어 역방향 검색(backward search)을 사용할 수 있다.

Algorithm 1 [2]. get_rows($P[1,p]$)

1. $i \leftarrow p$, $c \leftarrow P[p]$, $First \leftarrow C[c]+1$, $Last \leftarrow C[c+1]$;
2. **while** $((First \leq Last)$ **and** $(i \geq 2))$ **do**
3. $\quad c \leftarrow P[i-1]$;
4. $\quad First \leftarrow C[c] + Occ(c, First-1) + 1$;
5. $\quad Last \leftarrow C[c] + Occ(c, Last)$;
6. $\quad i \leftarrow i-1$;
7. **if** $(Last < First)$ **then return** "no rows prefixed by $P[1,p]$" **else return** $(First, Last)$

[그림 6] FM-Index[2]의 역방향 검색을 이용한 패턴 검색 알고리즘

이 방법에 착안하여 FM-Index가 Count 질의를 처리하는 방법은 [그림 6]과 같다. 패턴의 마지막 글자부터 역방향 검색을 진행하여 결과로 얻는 $(First, Last)$구간의 길이를 계산하고, 그 길이 값이 곧 패턴이 출현하는 횟수가 된다. 이렇게 얻은 패턴이 존재하는 구간의 모든 값에 대해 위치를 접미사 배열을 통해 구하면 패턴이 출현하는 모든 위치를 얻을 수 있다. 그러나 접미사 배열을 그대로 저장하는 것은 매우 큰 공간을 사용하므로 일정 간격으로 샘플링(marked row)하여 저장한다.

결과적으로 FM-Index는 $C[\,]$와 T^{bwt}에서의 $Occ(c,i)$함수, 그리고 샘플링된 접미사 배열과 역 접미사 배열로 구성된다. 초기 FM-Index[1]가 제안될 때는 알파벳 사이즈가 커지는 경우 사용되는 공간이 커지는 문제를 갖고 있었으나 [2]에서 웨이블릿 트리를 사용하여 알파벳 사이즈가 커지더라도 효율적인 공간 사용이 가능하도록 했다.

3.2 Compressed Suffix Array

접미사 배열이란 문자열 $T[1,n]$이 있을 때 T에 대한 모든 접미사 $T[i,n]$ $(1 \le i \le n)$을 사전 순으로 정렬한 것을 말하며 따라서 $SA[j]=i$는 i번째 접미사 $T[i,n]$이 사전 순서로 j번째임을 의미한다. 하지만 접미사 배열은 $O(nlogn)$비트의 저장 공간을 필요로 하므로 대용량 텍스트 데이터에 대해서 큰 공간을 사용하는 문제점이 있다. 이 문제점을 개선하기 위해서 Grossi와 Vitter[3]는 기존 접미사 배열을 압축하여 $O(nlog\Sigma)$비트를 사용하는 알고리즘을 제안했다. Sadakane[4]는 Grossi와 Vitter가 제안한 CSA를 self-index로 바꾼 자료구조를 제안했다. 접미사 배열을 압축하는 방법은 분할과 재귀적 구조를 사용하여 매 단계마다 부가적인 자료구조를 만들며 접미사 배열을 절반씩 압축해 나가는 방법이다. 각 압축 단계를 $k(0 \le k \le l)$라고 하면 다음과 같이 크게 네 가지 단계로 나눌 수 있다.

- Step 1 : 접미사 배열의 값이 짝수면 1을, 홀수면 0을 값으로 갖는 비트 벡터 B_k 생성한다.
- Step 2 : 함수 Ψ_k를 아래와 같이 정의하고 비트 벡터 B_k의 0값과 1값을 매핑한다.

$$\Psi_k(i) = \begin{cases} j & \text{if } SA_k[i] \; is \; odd \text{ and } SA_k[j] = SA_k[i]+1; \\ i & otherwise. \end{cases}$$

- Step 3 : 비트 벡터 B_k의 1값을 더해서 $rank_k$를 생성한다.
- Step 4 : 각 단계 접미사 배열 SA_k의 짝수 값만을 골라서 접미사 배열을 절반으로 압축하고 이전의 접미사 배열은 삭제한다.

위의 압축 과정을 재귀적으로 계속 진행하면서 매 단계의 B_k, Ψ_k, $rank_k$를 저장하고, SA_k는 저장하지 않으며 오직 마지막의 접미사 배열인 SA_l만 저장한다. 특히 Ψ_k는 압축 접미사 배열에서 SA_{k+1}로부터 이전 단계의 접미사 배열 SA_k 값을 계산하기위해 필요한 자료구조이다. $\Psi_k(i)$는 $SA_k[i]$값이

홀수 인 경우 다음 순서의 접미사 위치, 즉 $SA_k[i]+1$의 위치를 기록해둔다. 따라서 다음과 같은 식으로 각 단계의 SA_k를 복원할 수 있다.

$$SA_k[i] = 2 \times SA_{k+1}[rank_k(\Psi_k(i))] + (B_k[i]-1)$$

[그림 7]은 접미사 배열을 압축하는 첫 단계와 SA_1로부터 SA_0의 값을 계산하는 예시를 나타낸 것이다.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
T	a	b	b	a	b	b	a	b	b	a	b	b	a	b	a	a	a	b	a	b	a	b	b	a	b	b	b	a	b	b	a	#
SA_0	15	16	31	13	17	19	28	10	7	4	1	21	24	32	14	30	12	18	27	9	6	3	20	23	29	11	26	8	5	2	22	25
B_0	0	1	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	1	0	1	0	0	0	1	1	0	1	1	0
$rank_0$	0	1	1	1	1	1	2	3	3	4	4	4	5	6	7	8	9	10	10	10	11	11	12	12	12	12	13	14	14	15	16	16
Ψ_0	2	2	14	15	18	23	7	8	28	10	30	31	13	14	15	16	17	18	7	8	21	10	23	13	16	17	27	28	21	30	31	27

↓

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA_1	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11
B_1	1	1	0	1	1	1	0	0	1	0	0	1	0	1	0	0
$rank_1$	1	2	2	3	4	5	5	5	6	6	6	7	7	8	8	8
Ψ_1	1	2	9	4	5	6	1	6	9	12	14	12	2	14	4	5

	1	2	3	4	5	...
SA_1	8	14	5	2	12	...

B_0	0	1	0	0	0	0	1	1	0	1	0	0	...
$rank_0$	0	1	1	1	1	1	2	3	3	4	4	4	...
Ψ_0	2	2	14	15	18	23	7	8	28	10	30	31	...

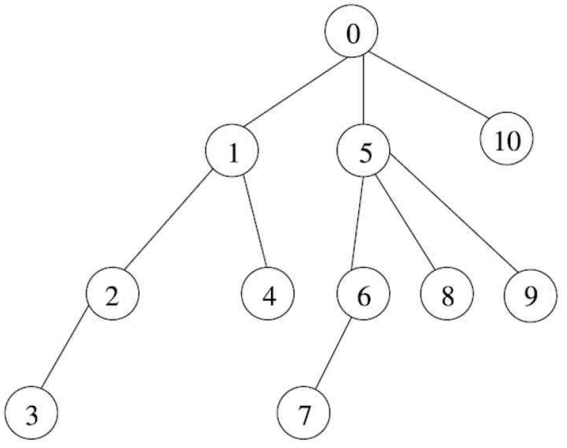
$$SA_0[2] = 2 \times SA_1[rank_0(\Psi_0(2))] + (B_0[2]-1)$$
$$SA_0[2] = 2 \times SA_1[1] + (1-1)$$
$$SA_0[2] = 2 \times 8 = 16$$

[그림 7] 압축 접미사 배열을 이용한
T = abbabbabbabbabaaababababbabbbabba# 의 압축 첫 단계와
SA_1로부터 $SA_0[2]$ 값 계산 과정 예시

압축 접미사 배열의 패턴 검색은 기존의 접미사 배열과 구조만 다를 뿐 동일한 방법을 사용한다. 패턴 검색은 이진 검색 방법을 통해서 이루어지며 패턴의 길이가 m일 때 비교하기 위해서 압축 접미사 배열을 복원 하는데 $O(m)$시간이 소요되므로 패턴 검색하는데 걸리는 총 시간은 $O(mlogn)$이다.

3.3 Compressed Suffix Tree

접미사 트리는 문자열 $T[1,n]$이 있을 때 T에 대한 모든 접미사 $T[i,n]$ $(1 \le i \le n)$을 사용하여 만든 트라이 구조로 전처리를 통해 패턴 검색하는데 있어서 패턴의 길이가 m일 때 $O(m)$의 시간 복잡도를 갖는다. 구조의 특징은 n개의 말단 노드를 가지며 각 노드는 T의 접미사들과 대응된다. 또한 각 간선은 알파벳 순서로 정렬되어있기 때문에 접미사를 나타내는 말단 노드들은 정렬되어 있는 구조이다. 접미사 트리는 McCreight[12]가 제안한 알고리즘을 통해 선형시간에 구축할 수 있다. 하지만 접미사 트리는 주어진 텍스트보다도 많은 공간을 사용하기 때문에 크기를 줄이기 위한 연구가 진행되었다.



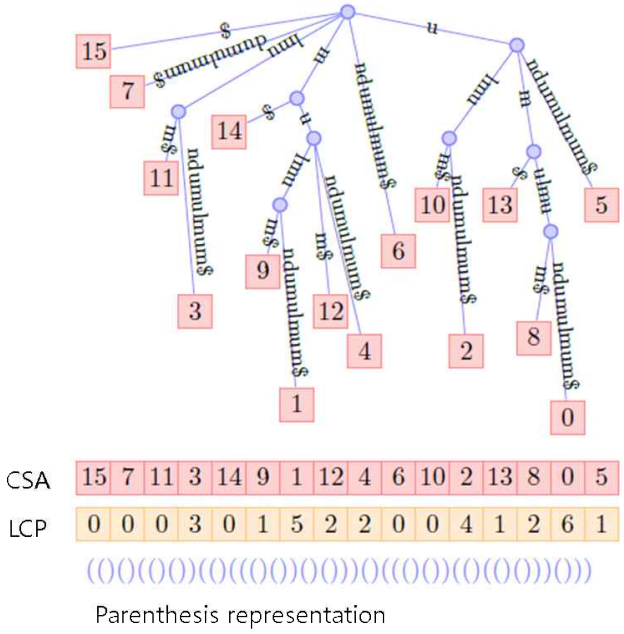
0 1 2 3 3 2 4 4 1 5 6 7 7 6 8 8 9 9 5 10 10 0

(((( )) ( )) ((( )) ( ) ( )) ( ))

[그림 8] parenthesis를 이용한 트리 표현 예시

Munro[13]는 이진 트리와 rooted ordered tree의 isomorphism 관계에 착안하여 접미사 트리를 parenthesis를 통해 $4n+o(n)$비트로 간결하게 표현하는 법을 제안했다. 이 표현방법은 트리를 전위 순회하면서 처음 노드를 방문하면 '('를, 그리고 다시 돌아오면서 ')'를 표기하는 방식이다. [그림 8]

은 parenthesis를 이용한 트리의 표현 예시이다. Sadakane[5]는 LCP 배열을 위한 새로운 자료구조와 함께 CSA와 접미사 트리의 간결한 표현법들을 사용하여 기존 접미사 트리보다 작은 크기를 사용하는 $|CSA|+6n+o(n)$비트 크기의 CST를 제안했다. [그림 9]는 접미사 트리를 CST로 구축할 때의 자료구조의 예시이다.



[그림 9] CST 자료구조의 예시

3.4 Lempel-Ziv 압축 알고리즘을 이용한 인덱스 자료구조

3.4.1 Lempel-Ziv 압축 알고리즘

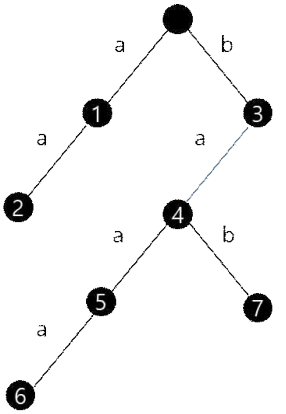
Lempel-Ziv 77(LZ77) 알고리즘[10]은 1977년 Lempel과 Ziv가 제안한 압축 알고리즘으로 현재 압축하려는 문자열과 그 전의 문자열 내에서 가장 길게 일치하는 부분의 상대적 위치와 길이를 찾아서 저장하는 방식으로 압축을 수행한다. LZ78, LZ-End 등과 같은 압축 알고리즘들은 모두 LZ77 알고리즘을 기반으로 파생된 알고리즘이다.

입력	a b c a b c b a c b a b ...
출력	(,0,a) (,0,b) (,0,c) (3,3,b) (4,1,c) (3,2,b) ...

[그림 10] LZ77 압축 알고리즘을 이용한 abcabcbacbab 압축

LZ77 압축 알고리즘의 압축 과정은 문자열 $T[1,n]$이 있을 때, 먼저 현재 압축을 할 문자 위치를 i라고 하면 압축을 마친 부분은 $T[1,i-1]$이고 압축할 부분은 $T[i,n]$으로 나눌 수 있다. 그러면 $T[i,n]$의 접두사 부분과 $T[1,i-1]$ 내에서 가장 길게 일치하는 문자열을 찾고 상대적 위치 p와 일치하는 문자열 길이를 l 구한다. 그리고 압축할 문자열에서 일치하는 문자열 다음의 문자 c하나를 추가하여 포인터 (p,l,c)로 표현하고 이 과정을 계속 반복하여 압축이 끝날 때까지 진행한다. [그림 10]은 LZ77 압축 과정을 나타낸 것이다.

입력	a	aa	b	ba	baa	baaa	bab
Block 번호	1	2	3	4	5	6	7
출력	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

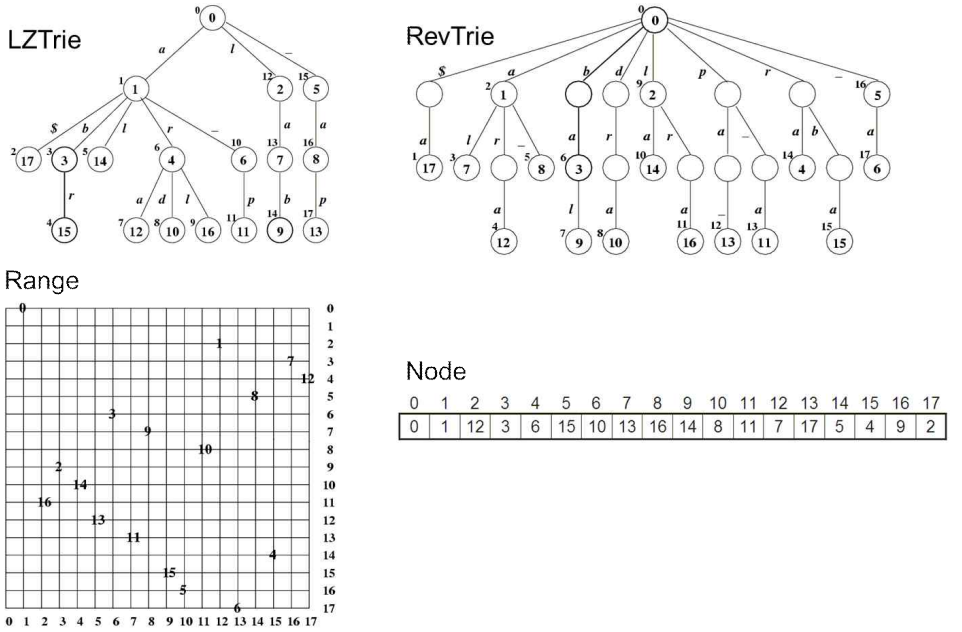


[그림 11] LZ78 압축 알고리즘을 이용한 aaabbabaabaaabab 압축 및 사전구조

LZ78은 LZ77과 유사하지만 현재 압축 문자 이전의 $T[1,i-1]$내에서 매칭을 찾는 것이 아니라 블록(phrase)중에서 가장 긴 매칭을 찾는 점이 다르다. LZ78은 압축을 하면서 각 포인터를 블록으로 만들어 사전을 만든다. 그리고 사전에서 가장 긴 매칭이 이루어지는 블록에 문자를 하나씩 추가하여 새로운 블록을 만들어나가는 방식으로 압축한다. [그림 11]은 LZ78 압축 과정과 그로 인해 만들어진 사전을 나타낸 것이다.

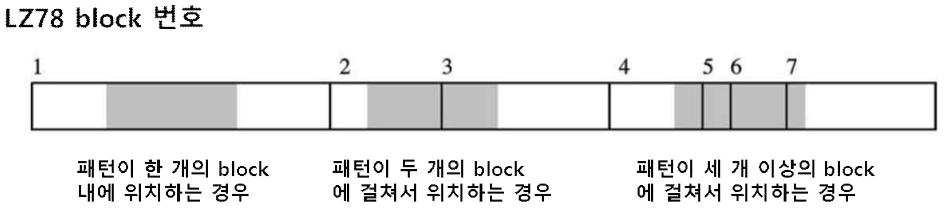
3.4.2 LZ-Index

LZ-Index는 3.4.1에서 소개한 Lempel-Ziv 압축 알고리즘을 사용하여 구축하는 인덱스 자료구조이다. Navarro[6]는 LZ78 압축으로 만들어지는 블록에 대한 트라이 구조와 부가적인 자료구조로 LZ-Index를 제안하였다. Navarro[6]은 크게 네 가지의 자료구조를 만드는데, LZ78의 결과로 만들어진 블록을 사용하여 LZTrie와 RevTrie의 두 개의 트라이 자료구조와 LZTrie의 노드와 블록의 번호를 매핑하는 Node 배열, 그리고 연속된 블록에 걸쳐있는 패턴 검색을 위한 Range 자료구조를 만든다. [그림 12]는 Navarro[6]에서 구축하는 LZ-Index의 자료구조들이다.



[그림 12] LZ-Index[6]가 블록을 사용하여 구축하는 네 가지 핵심 자료구조

Navarro[6]의 인덱스 검색 알고리즘은 먼저 패턴이 LZ78로 압축되어 블록으로 구성된 문자열에서 어떻게 위치해있을 것인가에 따라 달라진다. 주어진 패턴이 위치하는 경우는 [그림 13]과 같이 총 세 가지로 나눌 수 있다.



[그림 13] 블록 내에서 패턴 위치의 세 가지 경우

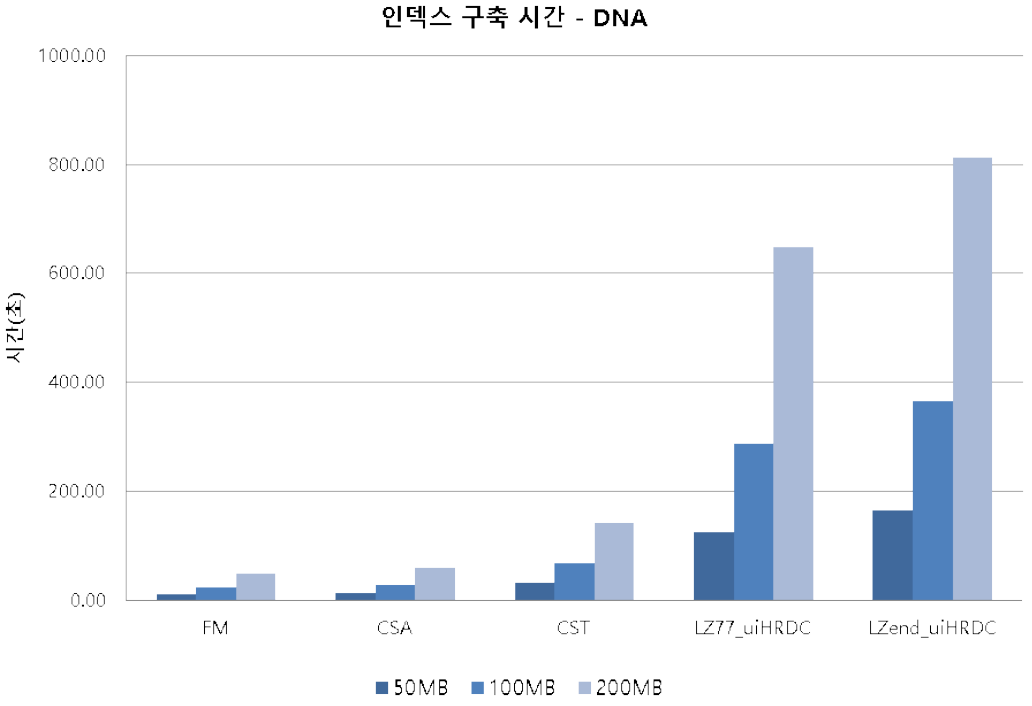
S.Kreft와 Navarro[7]는 일반적 텍스트가 아닌 유사한 텍스트가 많이 반복되는 텍스트 데이터에 대해서 효율적인 인덱스 자료구조를 구축하는 방법을 연구했다. 고반복 텍스트 데이터는 문서의 버전, 소스 코드의 버전, 주기적인 간행물, 생물학적 정보와 같은 데이터가 있으며, 이에 대해 [7]은 LZ77과 LZ-End 압축 알고리즘을 사용하여 인덱스를 구축하는 방법을 제안했다. LZ-End 압축 알고리즘은 LZ77과 유사하지만 가장 긴 매칭을 찾는 범위가 이미 만들어진 블록이 $B[1]...B[q]$라고 할 때 $B[1]...B[q]$의 접미사 범위에서만 찾는다는 점이 다르며, 문자열 추출 속도가 다른 LZ 압축 알고리즘보다 빠르다는 장점이 있다. 검색 알고리즘은 패턴 $P[1,m]$을 가능한 모든 $P[1,i]$와 $P[i+1,m]$로 분할하고 패턴의 왼쪽분할은 블록의 접미부분과 비교하고 패턴의 오른쪽분할은 블록의 접두부분과 비교하여 검색을 수행한다.

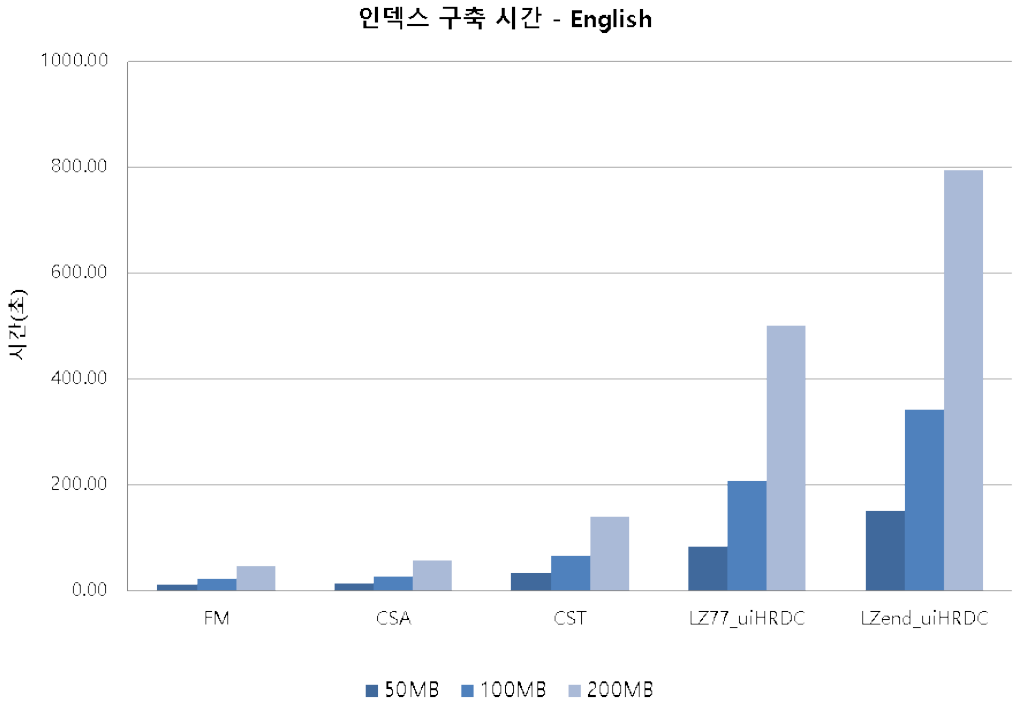
4. 실험 결과 및 분석

본 장에서는 3장의 네 가지의 인덱스 자료구조를 2장에서 설명한 실험 데이터를 사용하여 실제 성능 비교한 결과를 분석한다. 실험 측정 요소는 인덱스 구축 시간, 인덱스 크기, 메모리 사용량, 패턴 검색 시간, 그리고 문자열 추출 시간이다. FM-Index와 CSA, CST의 경우는 sdsl-lite[14] 라이브러리를 이용하여 구현했으며, LZ-Index의 경우는 [7]의 구현을 최적화한 uiHRDC[8]의 구현을 사용한다. 또한 LZ-Index는 LZ77, LZ-End 두 가지 경우로 나누어 실험한다. 실험은 각각 30번씩 측정하고 그 평균값을 사용했으며 실험에 사용된 시스템은 Intel Xeon X5672 3.20GHz, 32GB 메모리, Debian Linux를 사용하였다.

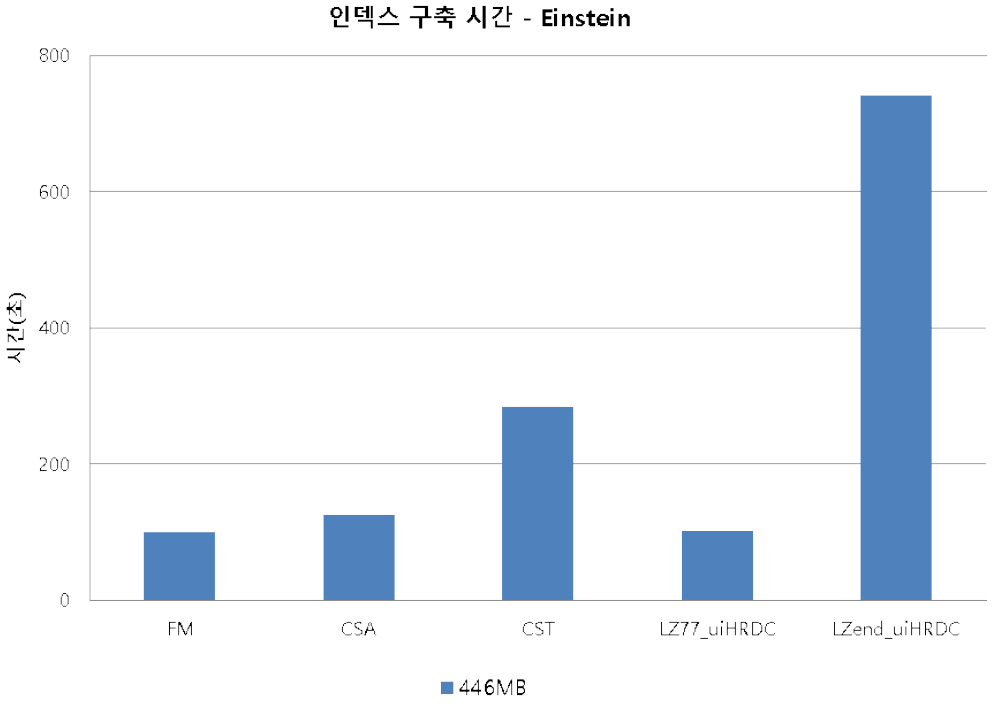
4.1 인덱스 구축 시간 비교

[그림 14]는 각 인덱스 자료구조에 대해 DNA, English 데이터의 50, 100, 200MB에 대한 인덱스 구축시간을 나타낸 그래프이다. 두 가지 데이터 모든 경우에서 FM-Index가 가장 빠른 시간에 인덱스를 구축하고 있으며 CSA의 경우도 빠른 구축시간을 보여준다. CST는 FM-Index보다 약 3배정도 느렸으며, LZ-Index는 LZ77와 LZ-End 두 가지 경우 모두 FM-Index에 비해 10배 이상 느린 속도를 보였다. [그림 15]는 고반복 텍스트 데이터인 Einstein의 경우인데, 마찬가지로 FM-Index가 가장 빠른 시간을 보이며 전체적으로 DNA, English데이터와 비슷한 양상을 보이나 이 경우에는 LZ77을 사용하는 LZ-Index가 FM-Index만큼 빠른 구축시간을 보여주었다.





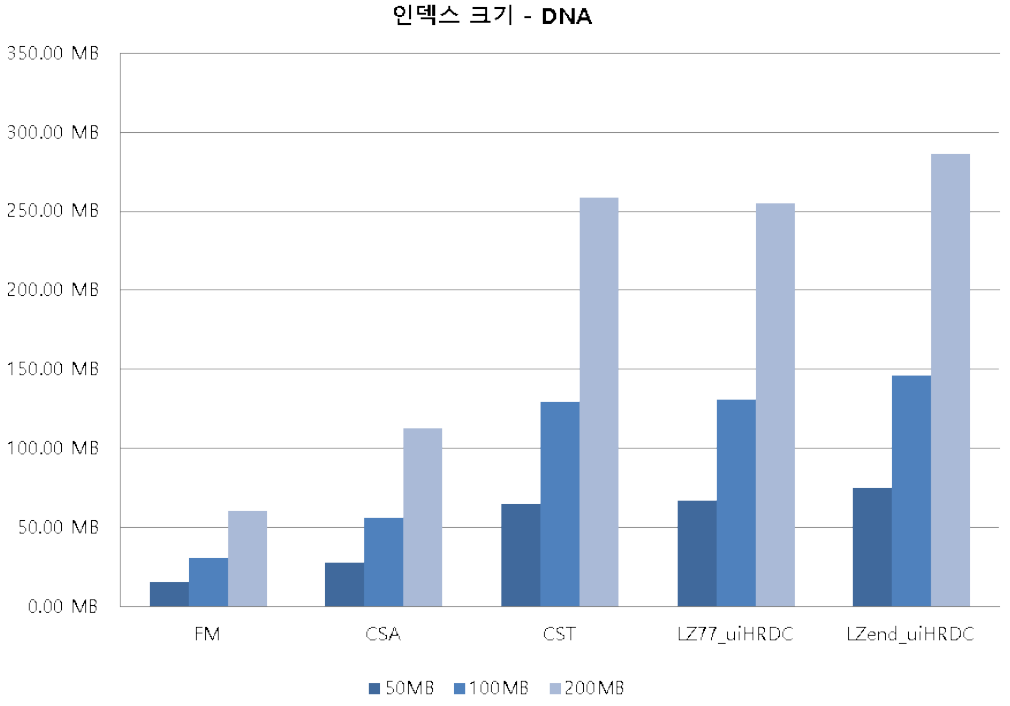
[그림 14] DNA(위), English(아래) 50,100,200MB에 대한 각 인덱스 자료구조 구축시간 비교 그래프

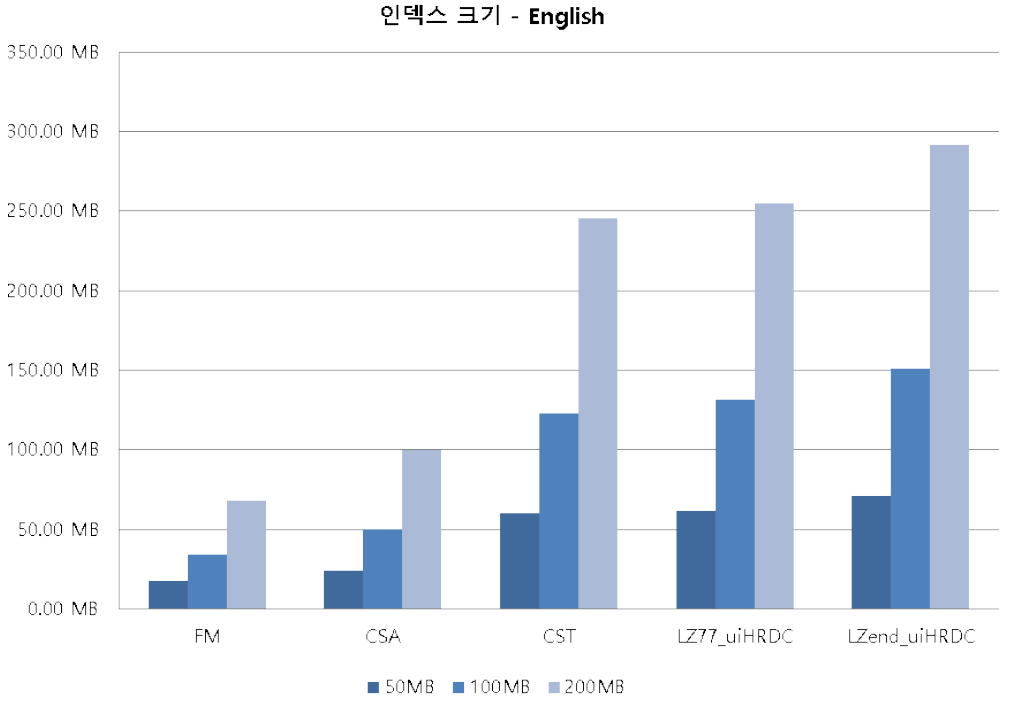


[그림 15] Einstein 446MB 데이터에 대한 각 인덱스 자료구조 구축시간

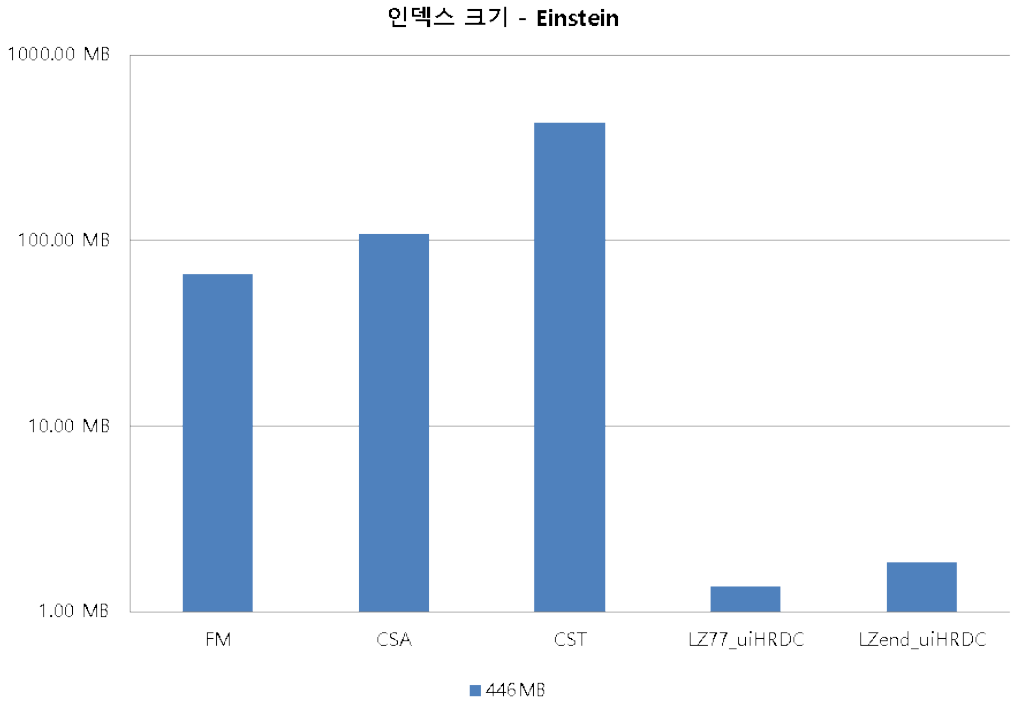
4.2 인덱스 크기 비교

[그림 16]은 각 인덱스 자료구조의 크기를 비교한 그래프이다. 인덱스 구축시간과 유사하게, 인덱스의 크기 또한 모든 실험 데이터에 대해서 FM-Index가 가장 작은 인덱스를 구축하는 것을 확인할 수 있다. 그러나 CST, LZ-Index의 경우는 오히려 입력 실험 데이터의 크기보다 더 큰 인덱스를 결과로 만드는 것을 확인할 수 있다. [그림 17]은 고반복 텍스트 데이터를 입력으로 사용한 결과이다. 여기서 눈여겨 볼 점은 LZ-Index는 LZ77의 경우 1.37MB, LZ-End의 경우 1.86MB라는 인덱스 크기로 놀라운 압축률을 보여준다는 점이다. 이 실험 결과를 통해 LZ-Index가 고반복 텍스트 데이터에 대해서 다른 인덱스 자료구조들보다 압축률이 굉장히 좋다는 것을 알 수 있다. 반대로 LZ압축 알고리즘 특성상 반복 문자열이 많이 없으면 압축률이 떨어지는데 이를 DNA, English데이터 실험 결과로 확인할 수 있다.





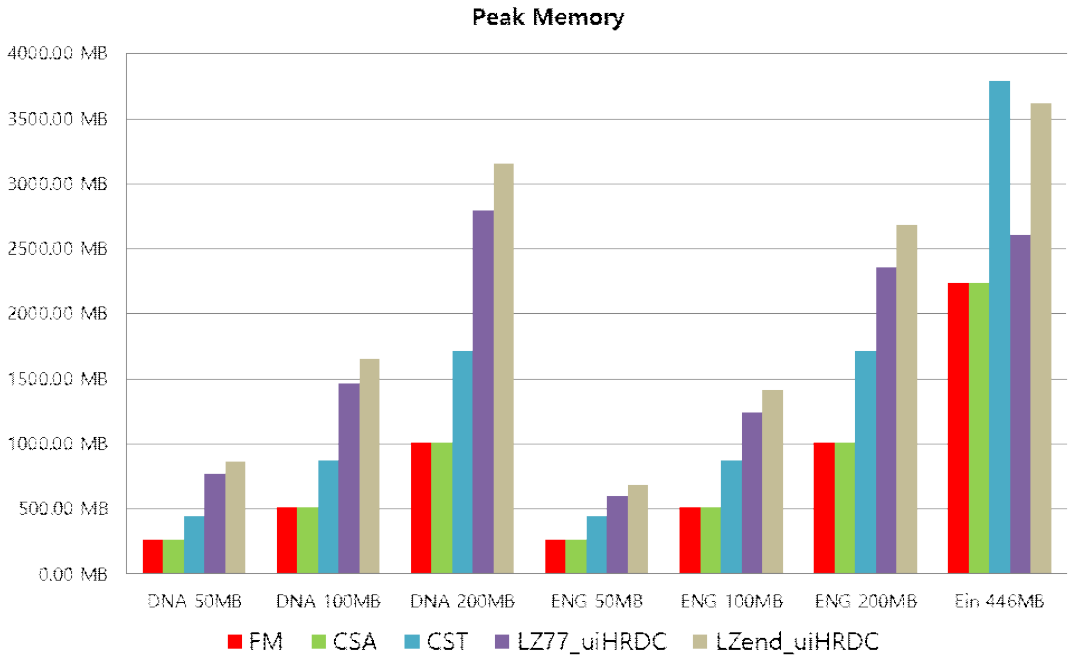
[그림 16] DNA(위), English(아래) 50,100,200MB에 대한 각 인덱스 자료구조 크기 비교 그래프



[그림 17] Einstein 446MB 데이터에 대한 각 인덱스 자료구조 크기

4.3 메모리 사용량 비교

[그림 18]은 각 실험 데이터에 대해 인덱스 자료구조 구축 시 최대 메모리 사용량을 비교한 그래프이다. 모든 경우에 대해 FM-Index와 CSA가 사용 메모리가 거의 동일하게 가장 적었고 CST, LZ77을 사용하는 LZ-Index, LZ-End를 사용하는 LZ-Index 순으로 메모리 사용량이 많았다. 고반복 텍스트 데이터의 경우는 인덱스 구축 시간 실험과 비슷하게 LZ77을 사용하는 LZ-Index가 메모리 사용량이 적어지는 결과를 보여주었다.

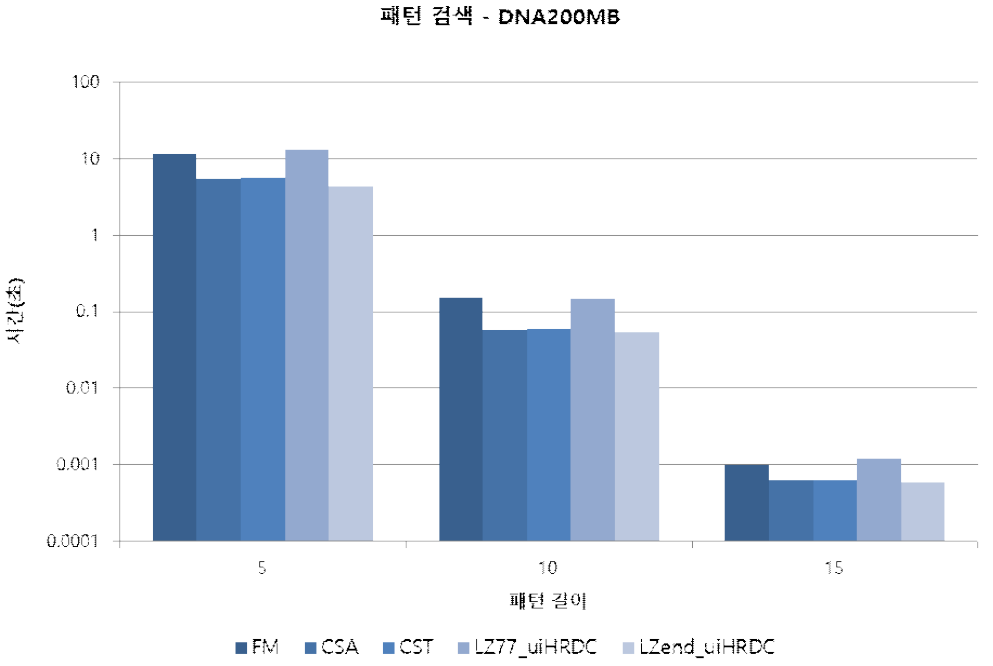


[그림 18] 모든 실험 데이터에 대한 각 인덱스 자료구조 구축 시 최대 메모리 사용량

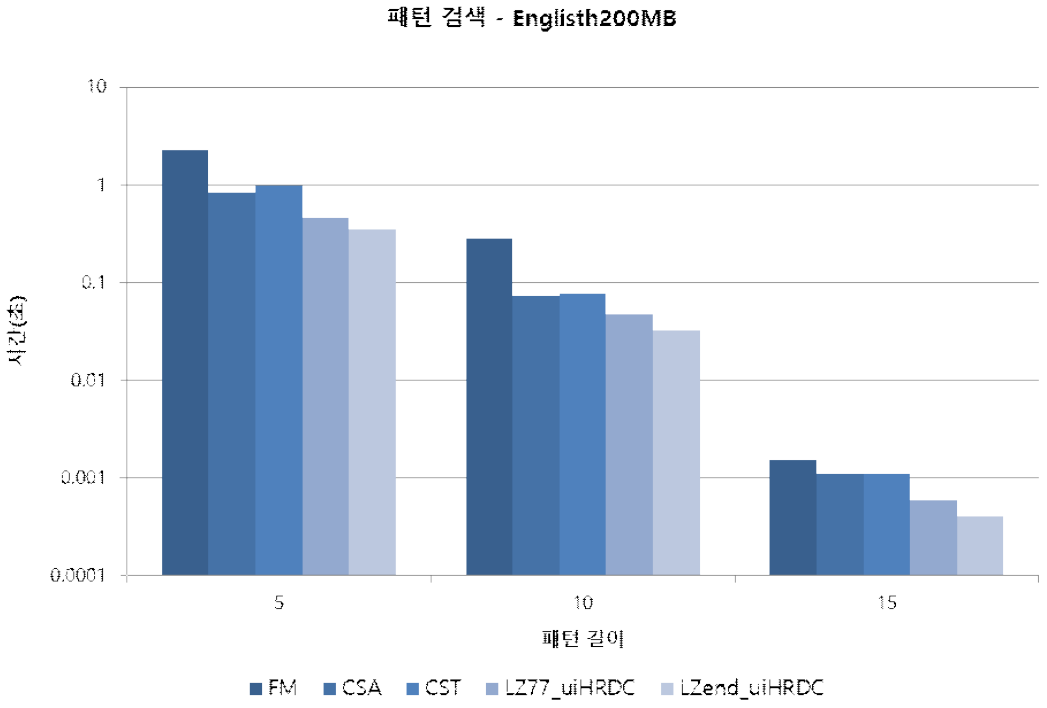
4.4 패턴 검색 시간 비교

이 실험에서는 각 인덱스 자료구조의 패턴 검색 시간을 측정하고 비교한다. DNA, English 50, 100MB 데이터의 경우 패턴 검색 시 검색되는 패턴 수가 작아서 검색 시간이 너무 작은 값들이 많이 나오기 때문에 200MB 데이터만 실험한 결과를 사용했다. 패턴은 무작위로 길이를 5, 10, 15의 세 가지 경우로 실험했으며 총 30번 패턴 검색 시간의 평균값을 사용했다.
[그림 19], [그림 20]은 각각 DNA 200MB, English 200MB의 패턴 길이 5, 10, 15의 패턴 검색 시간을 나타낸 그래프이다. 패턴의 길이가 길어짐에 따라 패턴 검색 시간이 크게 감소하는 것을 볼 수 있는데 그 이유는 패턴의 길이가 2, 3배 길어지면 텍스트에서 검색되는 패턴의 수가 2, 3배보다 훨씬 높은 비율로 감소하기 때문이다. DNA데이터의 패턴 검색 시간의 경

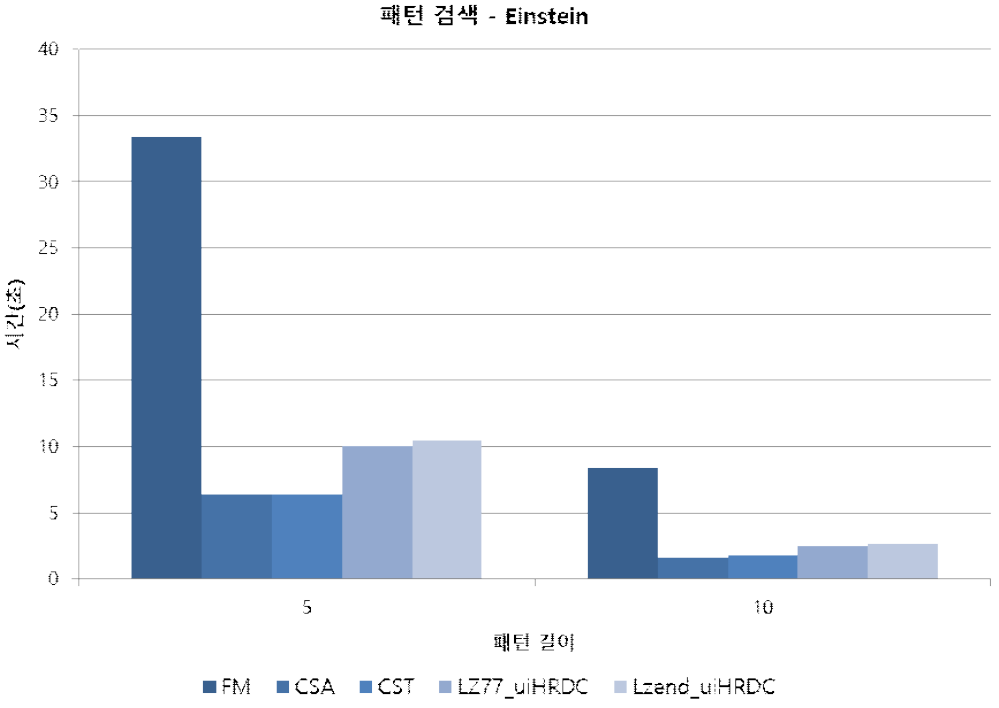
우는 LZ-End를 사용하는 LZ-Index가 가장 빠른 성능을 보여주고 있으며 그 뒤로 CSA, CST, FM-Index, LZ77을 사용하는 LZ-Index가 이어지고 있다. English데이터에 대해서는 두 개의 LZ-Index가 모두 빠른 검색 시간을 보여준다. 고반복 텍스트 데이터가 아닌 일반 텍스트 데이터에 대해서 LZ-Index가 패턴 검색에서 가장 좋은 성능을 보여주는 것은 눈여겨볼만한 결과이다. 하지만 CSA와 시간상으로는 그렇게 큰 차이가 있는 것이 아니며 또한 uiHRDC[8]의 LZ-Index 구현이 여러 방면으로 최적화가 잘 되어있다는 점도 무시할 수 없는 부분이다. [그림 21]은 Einstein 데이터에 대한 패턴 검색 시간 그래프이다. 여기서는 패턴의 길이를 5, 10으로만 실험했으며 15에 대해 하지 않은 이유는 검색되는 패턴 수의 차이가 길이가 10인 경우와 거의 없기 때문이다. Einstein의 경우 패턴 검색 시간은 CSA가 가장 빨랐으며 그 뒤로 CST, LZ-Index, FM-Index 순으로 나타났다. 고반복 텍스트 데이터에 효율적인 LZ-Index가 CSA보다 검색 속도가 느리긴 하지만 약 1.5배 정도 느린 데에 비해 FM-Index에 비해서는 4배에서 5배 정도 월등한 속도를 보여주고 있다.



[그림 19] DNA 200MB에 대한 패턴 검색 시간 비교 그래프



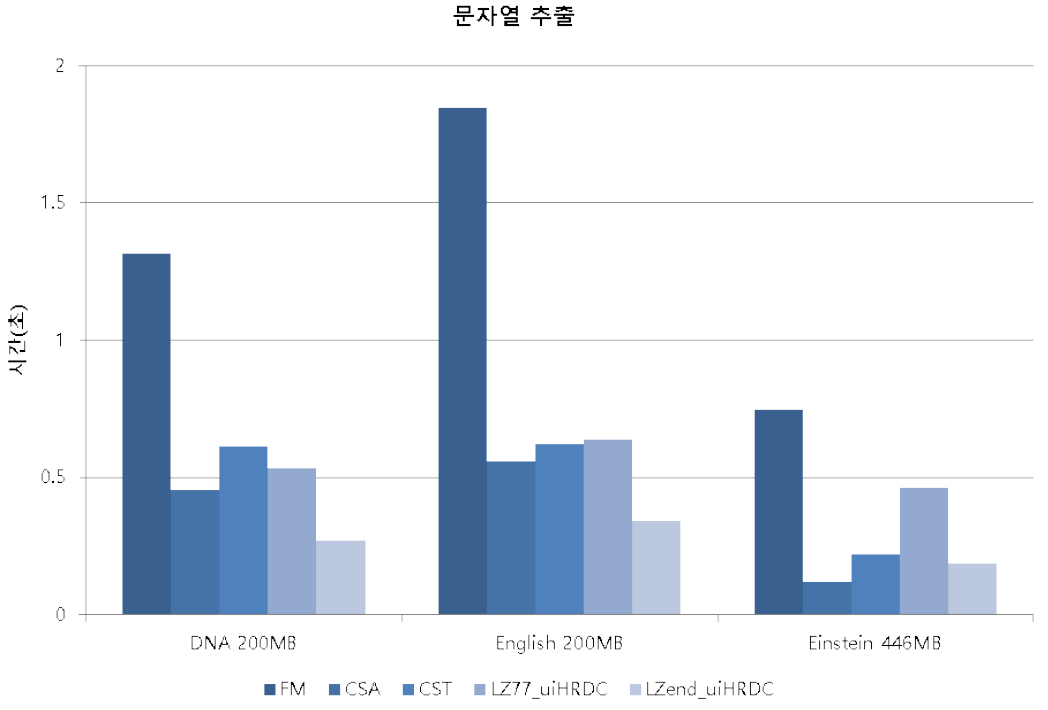
[그림 20] English 200MB에 대한 패턴 검색 시간 비교 그래프



[그림 21] Einstein 446MB에 대한 패턴 검색 시간 비교 그래프

4.5 문자열 추출 시간 비교

문자열 추출 시간은 DNA 200MB, English 200MB, Einstein 446MB의 데이터로 만들어진 인덱스 자료구조에 대해서 임의의 위치부터 100,000개의 문자를 추출하여 실험한다. [그림 22]는 실험 결과를 그래프로 나타낸 것이다. LZ-End를 사용하는 LZ-Index와 CSA가 문자열 추출 속도가 가장 빨랐으며 FM-Index는 다른 자료구조들에 비해 다소 느린 추출 시간을 나타냈다. LZ-End 압축 알고리즘이 문자열 추출에 효율적이라는 부분을 이 실험 결과를 통해 확인할 수 있다.



[그림 22] DNA, English, Einstein 데이터에 대한 길이 100,000의 문자열 추출 실험 그래프

4.6 실험 결과 분석

앞에서 각 인덱스 자료구조를 각 실험 항목에 대해서 실험을 진행하였다. 구축 시간과 패턴 검색 시간, 문자열 추출 시간은 짧을수록, 인덱스 크기와 최대 메모리 사용량은 적을수록 성능이 좋은 인덱스 자료구조라고 할 수 있다. 이를 토대로 실험 항목별로 인덱스 자료구조의 순위를 1~5의 숫자로 매겨보면 [표 2]와 같다.

DNA	구축 시간	인덱스 크기	메모리 사용량	패턴 검색 시간	문자열 추출
FM-Index	1	1	1	4	5
CSA	2	2	2	2	2
CST	3	3	3	3	4
LZ77_uiHRDC	4	4	4	5	3
LZEnd_uiHRDC	5	5	5	1	1

English	구축 시간	인덱스 크기	메모리 사용량	패턴 검색 시간	문자열 추출
FM-Index	1	1	1	5	5
CSA	2	2	2	3	2
CST	3	3	3	4	3
LZ77_uiHRDC	4	4	4	2	4
LZEnd_uiHRDC	5	5	5	1	1

Einstein	구축 시간	인덱스 크기	메모리 사용량	패턴 검색 시간	문자열 추출
FM-Index	1	3	1	5	5
CSA	3	4	2	1	1
CST	4	5	5	2	3
LZ77_uiHRDC	2	1	3	3	4
LZEnd_uiHRDC	5	2	4	4	2

[표 2] DNA(위), English(중간), Einstein(아래) 데이터에 대한 실험 항목별 인덱스 자료구조 순위

[표 2]를 보면 DNA와 English 데이터에 대해서 인덱스 구축 시간, 인덱스 크기, 메모리 사용량 항목에서 FM-Index가 전반적으로 가장 좋은 성능을 보였다. 하지만 패턴 검색과 문자열 추출에서는 성능이 다른 자료구조에 비해 좋지 않은 결과를 보였다. Einstein 데이터의 경우 눈여겨볼 부분은 LZ-Index를 사용했을 때의 인덱스 크기이다. 다른 항목에서는 다른 인덱스 자료구조에 비해 좋은 성능을 보여주지 않지만 인덱스 크기에서 LZ-Index를 제외하고 가장 작은 인덱스 크기와 비교하여도 약 60배 뛰어난 실험 결과를 보여준다. 따라서 압축률이 매우 좋기 때문에 다른 항목의 실험 결과로 LZ-Index가 고반복 텍스트에 대해서 성능이 좋지 않은 인덱스 자료구조라고 판단하기는 어렵다.

인덱스 자료구조는 가능한 문자열 T를 작게 압축하는 동시에 패턴 검색 등의 질의를 처리하는 자료구조이다. 따라서 작은 공간을 차지하는 인덱스 구축 측면에서 분석하면, FM-Index가 다른 자료구조에 비해 가장 좋은 성능을 보였다. 또한 고반복 텍스트 데이터에 대해서는 작게는 60배, 크게는 300배 정도의 압축률 차이를 보이는 LZ-Index가 작은 인덱스를 구축하는데 적합하다고 볼 수 있다.

5. 알파벳 사이즈가 작은 경우 FM-Index의 압축률 향상법

앞의 실험 결과 중 일부를 정리해보면 [표 3]과 같다. 알파벳 사이즈가 239인 English 50MB의 FM-Index 크기는 17.20MB, 알파벳 사이즈가 16인 DNA 50MB의 경우는 15.25MB, 그리고 알파벳 사이즈가 5 인 chr20 60MB는 13.92MB 크기의 FM-Index를 구축한다. 이를 통해 FM-Index는 알파벳 사이즈가 작으면 인덱스 크기가 작아지는 것을 알 수 있다. 그런데

이 결과는 모두 동일한 FM-Index[2] 구현을 사용한 것이므로 알파벳 사이즈가 작은 경우에 맞추어 변형하면 더 작은 인덱스를 구축할 수 있을 것이다. 먼저 FM-Index의 각 부분의 사용 공간을 파악을 하여 개선할 점을 찾아보았다. [표 4]는 A, C, G, T, N으로 이루어진 DNA데이터 chr20(60MB)으로 FM-Index[2]를 구축할 경우 각 자료구조의 크기이다.

알파벳 사이즈	파일 크기	FM-Index 크기
239	50 MB	17.20 MB
16	50 MB	15.25 MB
5	60 MB	13.92 MB

[표 3] 알파벳 사이즈에 따른 FM-Index[2] 크기

	크기
$C[\,]$	64 B
Wavelet tree	13.35 MB
SA sample	0.38 MB
ISA sample	0.19 MB
Total	13.92 MB

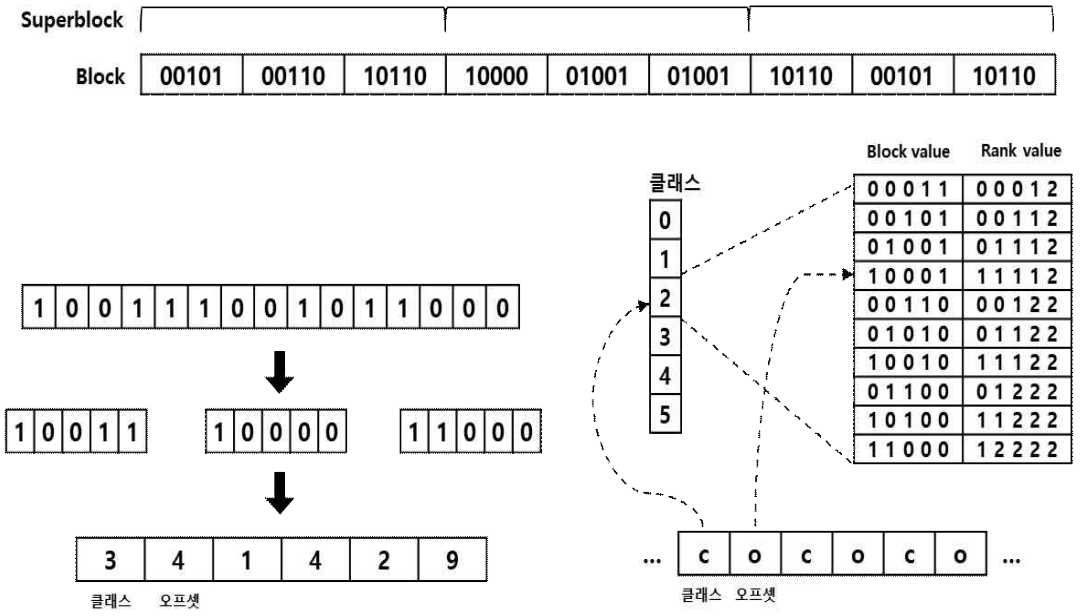
[표 4] chr20(60MB)의 FM-Index[2] 구축 시 데이터 구조

[표 4]에서 보면 알 수 있듯이 전체 인덱스 자료구조에서 가장 공간을 많이 차지하는 부분은 $Occ(c,i)$의 계산을 위한 자료구조인 웨이블릿 트리임을 알 수 있다. 2.4에서 설명했듯이 웨이블릿 트리는 다수의 알파벳으로 구성된 스트링에서 Rank/Select 질의를 조금 더 효율적으로 처리하기 위해서 고안된 자료구조이다. 그런데 알파벳 사이즈가 작으면 다수의 알파벳에 효율적인 웨이블릿 트리를 사용하지 않아도 되며, FM-Index에서 $Occ(c,i)$는 i번째 위치까지 문자 c의 개수를 세는 것이므로 이를 계산하기 위한 Rank 질의만 구현되면 충분하기 때문에 Select 질의를 위한 자료구조는 따로 필요하지 않다. 이 부분에서 알파벳 사이즈가 적은 경우 FM-Index의 압축률 향상 가능성을 생각할 수 있다. 만약 웨이블릿 트리보다 작고 Rank

질의를 처리할 수 있는 자료구조로 $Occ(c,i)$를 계산할 수 있다면 기존 FM-Index보다 작은 인덱스 자료구조를 구축할 수 있게 된다. 이 아이디어를 적용하기 위해 웨이블릿 트리를 대체하여 $Occ(c,i)$를 수행할 수 있는 자료구조는 다음과 같다.

- 5개의 알파벳에 대해서 각각 Rank연산이 가능한 일반 비트 벡터 구축
- 5개의 알파벳에 대해서 RRR[15] 벡터 구축
- Na et al.[16]의 Rank 연산이 가능한 자료구조 구축
- 초기 FM-Index[1]의 $Occ(c,i)$자료구조 사용

RRR[15] 벡터는 Rank 질의를 상수시간에 처리할 수 있는 압축된 비트 벡터 자료구조이다. RRR을 구축하는 방법은 먼저 정해진 수의 큰 블록(superblock)으로 주어진 비트 벡터를 나누고 나누어진 큰 블록을 다시 한 번 작은 블록(block)으로 나눈다. 그리고 사전에 블록의 Rank 값이 계산된 테이블을 만들고 각각의 작은 블록을 클래스 값 C와 오프셋 값인 O로 매핑한다. Rank 질의는 (C,O)를 통해 Rank 값이 계산된 테이블에 접근하여 처리한다. [그림 23]은 RRR벡터의 구조를 도식화한 것이다.



[그림 23] RRR벡터의 구조와 예시

Na et al.[16]은 비트 벡터에 대해서 Rank/Select 질의를 위한 자료구조를 제안했다. [그림 24]를 참고하여 설명하면, 먼저 주어진 비트 벡터 A를 일정 크기로 블록으로 분할 후 0으로만 이루어진 zero-block과 1이 하나라도 있는 nonzero-block으로 구분한다. 그리고 nonzero-block에 해당하는 비트 벡터 B와 zero, nonzero-block을 구분하는 비트 벡터 C 정보만을 저장한다. 그리고 B, C 에 대해 각각 2단계의 Rank 디렉토리를 만들어서 입력 비트 벡터 A 없이 Rank 질의를 가능하도록 한다. 예를 들어 [그림 23]에서 $Rank_A(28)$을 계산하는 과정은 먼저 B에서의 블록위치 $\lceil 28/5 \rceil = 6$을 얻고 $28-(6-1)*5=3$을 통해 28번째의 위치가 6번째 블록의 3번째라는 것을 계산한다. 이제 28번째가 B 벡터에서 몇 번째인지 $Rank_C(6-1)*5+3*C[6]=13$ 을 통해 계산한다. 마지막으로 $Rank_B(13)$을 통해 $Rank_A(28)$을 계산할 수 있다. 여기서 Select 질의를 위한 자료구조는 FM-Index에 필요하지 않으므로 더 작은 크기로 구현이 가능하다.

						↓	
A	01101	00000	00000	11010	00000	01001	00100
C	1	0	0	1	0	1	1
B	01101	11010	01001	00100			
			↑				

[그림 24] Na et al.[16]의 비트 벡터 분할 방법

초기 FM-Index[1]에서는 [2]와 다르게 $Occ(c,i)$계산을 위해 T^{bwt}를 일정 크기의 bucket으로 나누어 2단계로 구성한다. 그리고 각 bucket 내 알파벳의 Rank 값을 미리 계산하여 저장하고 누적 Rank 값을 모두 저장해두는 방식이다. 이 방법은 알파벳 사이즈가 커질 때 많은 공간을 차지하는 문제가 있었는데, 알파벳 사이즈가 작은 조건 하에서는 웨이블릿 트리보다 효과적일 수도 있기 때문에 실험에 사용한다. 위 네 가지 자료구조를 chr20 데이터에 대한 인덱스 크기를 측정하여 기존의 웨이블릿 트리보다 더 작은지 실험을 진행한다. 실험 환경은 4장에서 실험한 환경과 동일하며 좀 더 정확한 비교를 위해 되도록 sdsl-lite 라이브러리를 사용하여 구현한다.

[표 4]는 앞서 소개한 네 가지 방법으로 chr20 데이터의 $Occ(c,i)$ 자료구조 크기를 측정한 표이다. 기존의 웨이블릿 트리의 경우는 13.35MB이므로 네 가지 방법 모두 웨이블릿 트리보다 크기를 줄이지 못했다. 가장 웨이블릿 트리에 근접한 결과는 Na et al.[16]을 사용한 것으로 15.90MB의 공간을 사용한다. 비트 벡터의 경우는 모든 알파벳에 대해 입력 문자열의 길이만큼 비트 벡터를 사용하기 때문에 사이즈가 커졌고, RRR벡터의 경우는 알파벳 N과 같이 1의 빈도가 낮은 경우는 압축률이 높아져서 0.47MB까지 압축이 가능하지만 다른 알파벳의 경우는 그렇지 못했기 때문에 13.35MB보다 더 작아지지 못했다. 전체적으로 분석해보면 웨이블릿 트리의 경우 여러 알파벳에 대해서 하나의 자료구조를 만들지만 실험한 방법들은 모두 각각의 알파벳에 대해서 개별적인 자료구조를 가져야하고 이뿐만 아니라 Rank 질의를 위한 추가적인 자료구조를 만들어야하기 때문에 더 작아질 수 없는 것으로 분석된다.

	sdsl-lite 비트 벡터	sdsl-lite RRR 벡터	Na et al.[16] Rank only	초기 FM-Index[1]
A	7.51MB	5.86MB	3.18MB	.
C	7.51MB	4.95MB	3.18MB	.
G	7.51MB	5.23MB	3.18MB	.
T	7.51MB	5.82MB	3.18MB	.
N	7.51MB	0.47MB	3.18MB	.
Rank support	1.88 x 5 = 9.40MB	.	.	.
Total	46.95MB	22.33MB	15.90MB	18.30MB

[표 4] 비트 벡터, RRR벡터, Na et al.[16], 초기 FM-Index[1] 를 사용한 chr20데이터 $Occ(c,i)$ 자료구조 크기

따라서 알파벳 사이즈가 작은 경우 FM-Index의 크기 축소를 위해 웨이블릿 트리를 대체하여 실험한 결과, 알파벳 사이즈에 크게 관련 없이 기존의 구현처럼 웨이블릿 트리로 구현하는 것이 가장 작은 인덱스를 구축했다.

6. 결론

본 논문에서는 텍스트 인덱스 자료구조를 일반적인 텍스트 데이터와 고반복 텍스트에 대해서 인덱스 구축 시간 및 크기, 메모리 사용량, 패턴 검색과 문자열 추출 시간을 중점으로 비교 분석해보고 알파벳 사이즈가 작은 경우 FM-Index의 압축률을 높일 수 있는 방법을 적용하여 실험해보았다.

실험 결과 일반적인 텍스트 데이터에 대해서 FM-Index가 가장 작은 인덱스를 구축하며 전반적으로 뛰어난 성능을 보였다. 하지만 패턴 검색과 문자열 추출에서는 다른 인덱스 자료구조보다 다소 성능이 떨어지는 것을 확인할 수 있었다. 고반복 텍스트 데이터의 경우는 LZ-Index가 다른 인덱스 자료구조보다 작게는 60배, 최대로는 300배 압축된 인덱스를 구축함으로써 공간 효율적 측면에서 뛰어난 성능을 보여주었다. 또한 알파벳 사이즈가 작은 경우 FM-Index의 압축률 향상을 위해 웨이블릿 트리를 대체하여 다양한 자료구조를 실험한 결과, 알파벳 사이즈에 관련 없이 웨이블릿 트리를 사용하는 FM-Index가 가장 작은 인덱스를 구축하는 것을 확인했다.

참 고 문 헌

[1] Ferragina, Paolo, and Giovanni Manzini. "Opportunistic data structures with applications." Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. IEEE, 2000.

[2] Ferragina, Paolo, et al. "An alphabet-friendly FM-index." String Processing and Information Retrieval. Springer Berlin/Heidelberg, 2004.

[3] Grossi, Roberto, and Jeffrey Scott Vitter. "Compressed suffix arrays and suffix trees with applications to text indexing and string matching." SIAM Journal on Computing 35.2. 378-407. 2005.

[4] Sadakane, Kunihiko. "New text indexing functionalities of the compressed suffix arrays." Journal of Algorithms 48.2. 294-313. 2003.

[5] Sadakane, Kunihiko. "Compressed suffix trees with full functionality." Theory of Computing Systems 41.4. 589-607. 2007.

[6] Navarro, Gonzalo. "Indexing text using the Ziv-Lempel trie." Journal of Discrete Algorithms 2.1. 87-114. 2004.

[7] Kreft, Sebastian, and Gonzalo Navarro. "On compressing and indexing repetitive sequences." Theoretical Computer Science 483. 115-133. 2013.

[8] Claude, Francisco, et al. "Universal indexes for highly repetitive document collections." Information Systems 61. 1-23. 2016.

[9] Burrows, Michael, and David J. Wheeler. "A block-sorting lossless data compression algorithm." (1994).

[10] Ziv, Jacob, and Abraham Lempel. "A universal algorithm for sequential data compression." IEEE Transactions on information theory 23.3. 337-343. 1977.

[11] Grossi, Roberto, Ankur Gupta, and Jeffrey Scott Vitter. "High-order entropy-compressed text indexes." Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2003.

[12] McCreight, Edward M. "A space-economical suffix tree construction algorithm." Journal of the ACM (JACM) 23.2. 262-272. 1976.

[13] Munro, J. Ian, Venkatesh Raman, and S. Srinivasa Rao. "Space efficient suffix trees." Journal of Algorithms 39.2. 205-222. 2001.

[14] Succinct Data Structure Library. https://github.com/simongog/sdsl-lite

[15] Raman, Rajeev, Venkatesh Raman, and S. Srinivasa Rao. "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets." Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2002.

[16] J. C. Na, J. E. Kim, K. Park, D. K. Kim, Fast Computation of Rank and Select Functions for Succinct Representation, IEICE TRANSACTIONS on Information and Systems Vol. E92-D No.10, 2025-2033, 2009.

Abstract

Comparative Analysis of Full-Text Index Data Structures

Seo Woo Gong
Department of Computer Science and Engineering
The Graduate School
Seoul National University

Full-text index data structure is data structure which is for compressing a text string while making searching and extracting on the compressed object possible. Recently, as the size of the text data increases, the importance of constructing a space-efficient index data structure for large text data is increasing. As a result, various studies on the index data structure have been carried out. Representative text index data structures include FM-Index, Compressed Suffix Array(CSA), Compressed Suffix Tree(CST), and LZ-Index based on the Lempel-Ziv compression algorithm.

In this paper, we first compare and analyze the actual performance of index data structures by experimenting with construction time, index size, memory usage, pattern search and extraction time for various text data. Secondly, the previous analysis shows that FM-Index has

relatively good compression ratio so we carried out a study about how to improve the compression rate when the alphabet size is small for FM-Index. As a result, in terms of space efficiency, FM-Index is suitable for general text data and LZ-Index is suitable for highly repetitive text data.

keywords : text indexing, compression, suffix array, suffix tree, pattern searching
Student Number : 2016-21184