접미사 트리를 이용한 압축 기법에서 가장 긴 매치 찾기

나중채* , 박근수
jcna@theory.snu.ac.kr, kpark@theory.snu.ac.kr
서울대학교 컴퓨터공학과

Finding the longest match in data compression using suffix trees

Joong Chae Na, Kunsoo Park
Department of Computer Engineering, Seoul National University

요약

Ziv-Lempel 코딩 방식은 문자열이 반복해서 나올 때 뒤에 나오는 문자열을 앞에 나온 문자열에 대한 포인터로 대치시킴으로써 압축을 한다. 따라서 이 방식을 위해서는 앞서 나온 문자열을 유지하는 사전과 문자열 매칭이 필수적이다. 그래서 이 두 가지에 효율적인 자료구조인 접미사 트리를 Ziv-Lempel 코딩 방식에 적용시키려는 연구가 많이 진행되어 왔다. Rodeh, Pratt와 Even은 접미사 트리를 LZ78 코딩에 이용하는 방법을 제안하였고, 그 이후에 Fiala, Greene와 Larsson은 각각 McCreight와 Ukkonen의 접미사 트리 생성 알고리즘을 LZ77 코딩에 이용하였다.

접미사 트리를 이용한 Ziv-Lempel 코딩에는 만들어진 사전, 즉 접미사 트리와 앞으로 압축될 문자열과의 가장 긴 매치를 찾는 과정이 있다. 이는 단순히 접미사 트리의 루트부터 차례로 검색해 나가도 되지만 이렇게 했을 때 걸리는 시간은 노드에서 자식을 찾는데 걸리는 분기 결정 시간에 의해 좌우된다. 즉 분기에 선형 시간 이상이 걸리면 가장 긴 매치를 찾는데도 역시 선형 시간 이상이 걸린다. 게다가 이 방법은 자기 중복(self-overlapping)의 이점을 살릴 수가 없다. Rodeh, Pratt와 Even은 McCreight의 생성 알고리즘을 이용할 때 가장 긴 매치를 바로 찾을 수 있다는 것을 발견했다 그러나 Ukkonen의 알고리즘에 대해서는 아직 어러한 방법이 알려지지 않았다. 본 논문에서는 Ukkonen의 알고리즘에 몇 가지 작업을 추가하여 전체적으로 선형시간 안에 가장 긴 매치를 찾는 방법을 소개한다.

1. 서론

문자열 매칭(matching)은 데이터 압축에서 중요한 부분을 차지한다. 특히 Ziv-Lempel 코딩(coding)[6,7]과 같은 문자열 교체 방식을 이용한 압축 기법에서는 문자열 매칭은 필수적이다.

문자열 매칭에 자주 사용되는 효율적인 자료 구조로서 접미사 트리가 있다. 접미사 트리는 선형시간에 주어진 문자열의 모든 부분 문자열을 표시할 수 있다. 따라서 접미사 트리를 압축에 이용하려는 연구가 많이 진행되어 왔다.

접미사 트리의 선형 시간 생성 알고리즘에는 크게 두 가지가 있다. 하나는 문자열 전체를 입력으로 받은 다음 전체 트리를 만드는 오프라인(off-line) 방식으로 McCreight[3]가 제안하였고, 다른 하나는 한 문자씩 읽으면서 트리를 생성하는 온라인 방식으로 Ukkonen[5]이 제안하였다. 이중 McCreight의 알고리즘을 Ziv-Lempel 코딩에 적용하는 기법을 Rodeh, Pratt와 Even[4]이 제시하였다

또 LZ77[6]과 같이 sliding window 기법을 사용하는 Ziv-Lempel 방식은 압축 시에 비슷한 문자열들이 서로 가까이에 있다는 지역성을 이용하는데 접미사 트리를 이와 같은 방식에 이용하기 위해서는 window의 크기를 벗어나는 문자열, 즉 멀리 떨어져 있는 문자열을 제거하는 방법이 필요하다. 이에 대해서도 역시 많은 연구가 이루어졌는데 Fiala와 Greene[1]는 McCreight의 알고리즘에 window를 벗어나는 문자열을 제거하는 알고리즘을 개발해서 선형시간 안에 자료를 압축하는 방식을 제안하였고, 후에 Larsson[2]이 Ukkonen의 알고리즘에 역시 앞의 문자를 삭제하는 알고리즘을 더해 압축에 응용하였다.

Ziv-Lempel 방식으로 압축을 하기 위해서는 지금까지 압축된 문자열 혹은 window에 속한 문자열과 앞으로 압축을 하려는 문자열과의 가장 긴 매치를 찾아야 한다. 접미사 트리의 특성상 단순히 루트부터 차례로 문자열을 비교해 나가면 가장 긴 매치를 찾을 수 있다. 그러나 이는 별도의 작업을 필요로 한다. 하지만 Rodeh, Pratt와 Even[4]은 McCreight의 생성 알고리즘에서 별다른 작업이 없이 바로 가장 긴 매치를 찾는 방법이 있다는 것을 발견했다. 그러나 온라인 방식인 Ukkonen 알고리즘을 이용한 압축에서는 아직 그러한 방법이 발견되지 않았다. Larsson[2]은 가장 긴 매치를 찾기 위해 루트부터 검색하는 방식을 사용하였다.

본 논문에서는 Ukkonen 알고리즘을 이용한 압축에서도 역시 가장 긴 매치를 찾기 위해 루트부터 검색해 나가지 않고도 Ukkonen의 알고리즘에 약간의 작업만 추가하면 접미사 트리를 생성하면서 자동적으로 가장 긴 매치를 찾을 수 있다는 것을 보인다.

다음 절에서는 우선 접미사 트리의 정의와 성질을 알아보고 다음에 Ukkonen이 개발한 접미사 트리의 온라인 생성 알고리즘을 살펴본다. 그리고 다음에 접미사 트리를 Ziv-Lempel 코딩에 이용하는 방법을 간략히 살펴본 다음에 본 논문의 주 내용인 가장 긴 매치를 찾는 새로운 방법을 설명하고 결론을 맺는다.

2. 접미사 트리

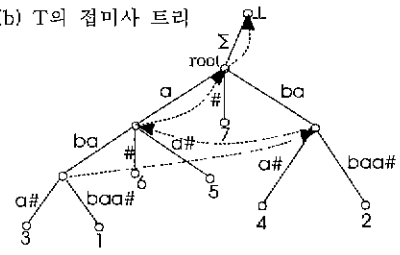
접미사 트리는 고정 크기 알파벳(alphabet) Σ의 문자열 T의 모든 접미사를 표현하는 compacted trie로서 다음과 같은 세 가지 조건을 만족한다.

1) 각 간선(edge)은 문자열 T의 공백이 아닌 부분 문자열로 레이블(label)되어 있는데, 그 레이블은 부분 문자열의 시작 위치와 끝 위치의 쌍으로 저장되어 있다.
2) 루트를 제외한 각 내부 노드는 적어도 두 개 이상의 자식 노드를 가지며 자식 노드로 가는 간선들의 레이블의 시작 문자는 모두 다르다.
3) 모든 단말 노드는 원래 문자열에서 하나의 접미사와 연관되어 있어서, 루트 노드부터 그 단말 노드까지의 경로에 있는 간선의 레이블을 접합시키면 연관되어 있는 접미사와 같게 된다.

여기서 고려되는 문자열 T의 마지막 문자는 유일해야 한다 이 조건은 실제 접미사 트리를 압축에 이용할 때는 필요 없지만 알고리즘의 설명을 간단히 하기 위해 가정한다.

(a) T : 1 2 3 4 5 6 7
a b a b a a #

(b) T의 접미사 트리



<그림 1> 접미사 트리의 예

<그림 1>은 주어진 문자열에 대한 접미사 트리의 예를 보여준다. 단말 노드에 있는 숫자는 접미사의 인덱스이고, 간선에 표시된 문자열은 각 간선이 나타내는 부분 문자열로서 실제로는 두 개의 정수 쌍(시작 위치, 끝 위치)으로 표시된다. 루트 노드 위의 노드 ⊥에 대해서는 다음절에서 설명한다.

접미사 트리의 단말 노드는 T의 각 접미사와 일대일 대응을 이룬다. 따라서 단말 노드의 개수는 접미사 개수의 수와 같다 그리고 모든 내부 노드는 두 개 이상의 자식을 가지므로 전체 노드의 수는 접미사 개수의 2배를 넘지 않는다. 그리고 각 노드는 문자열을 표시하기 위해 시작과 끝 위치를 저장하므로 접미사 트리를 유지하기 위한 저장공간은 문자열의 길이에 비례한다.

3. 접미사 트리의 온라인 생성 알고리즘

상수 문자 집합에 대한 접미사 트리의 생성 알고리즘으로는 서론에서 언급한 두 가지가 있는데 이 중에서 본 논문에서 필요한 Ukkonen의 알고리즘을 간략히 설명한다. 좀더 자세한 내용은 [5]를 참조하기 바란다.

먼저 설명에 필요한 기호를 몇 가지 정의한다 우선 문자열 T의 길이를 n으로 표시한다. 또 T의 각 문자는 t_i로 표시한다 $(1 \le i \le n)$ 그리고 길이가 i인 T의 접두사를 T_i로 표시한다. $STree(T_i)$는 아래에서 설명할 알고리즘에 의해 중간에 생성되는 T_i에 대한 접미사 트리로서 위에서 제시한 접미사 트리의 성질을 완전히 만족하지는 않지만 혼동되지 않는 범위에서 접미사 트리라고 부르겠다

그리고 각 내부 노드는 루트부터 그 노드까지의 경로가 나타내는 문자열에서 맨 첫 문자를 제외한 문자열을 나타내는 노드를 가리키는 포인터를 가진다 이 포인터를 접미사 링크(suffix link)라 부른다. <그림1>에서 각 내부노드의 접미사 링크는 점선으로 표시하였다. 이 접미사 링크는 접미사 트리를 선형 시간 안에 생성할 수 있게 해주는 핵심적인 요소가 된다.

Ukkonen은 알고리즘의 단순화를 위해 루트의 부모 노드로서 더미 노드(⊥)를 첨가했다. 이 더미 노드는 예외적으로 앞 절에서 설명한 접미사 트리가 가져야할 조건을 만족시키지 않는다. 더미 노드는 고려되는 모든 알파벳에 대한 자식으로 루트를 가지고 루트의 접미사 링크는 이 더미 노드를 가리킨다

Ukkonen의 알고리즘은 앞서도 언급했듯이 온라인 알고리즘이다. 즉 문자를 하나씩 읽어나가면서 트리를 확장해 나간다. 즉, i번째 반복(iteration)시 트리 $STree(T_{i-1})$에 문자 t_i를 더해 $STree(T_i)$를 만든다 이 과정을 n번 반복(iteration)하면 $STree(T_n)$ 즉 완전한 접미사 트리 $STree(T)$를 만들 수 있다. 그러므로 i번째 반복은 T_{i-1}의 모든 접미사를 α라 했을 때 이를 αt_i로 바꾸는 과정이라고 볼 수 있다. 이 때 αt_i는 다음 세 가지 중 하나에 속한다.

1. α가 T_{i-1}에 오직 한번 나타난다. 이는 α가 단말 노드로서 존재한다는 것을 의미하므로 이를 αt_i로 바꾸기 위해서 단말 노드가 나타내는 부분 문자열의 끝 위치를 하나 증가시키기만 하면 된다.
2. α가 T_{i-1}에 두 번 이상 나타나지만 αt_i는 나타나지 않는다. 이것은 트리에 αt_i에 대한 단말노드가 생성되어야 한다는 것을 의미한다. 이 때 이 단말노드의 부모로서 내부 노드가 생성될 수도 있다.
3. αt_i가 T_{i-1}에 이미 존재한다. 따라서 아무 작업도 할 필요가 없다.

이 때 1번에 속한 α는 2번에 속한 α보다 길고, 2번에 속한 α는 3번에 속한 α보다 길다는 성질이 있다. 또한 1번에 속한 αt_i를 삽입하기 위한 작업은 단말노드의 끝 위치를 하나 증가 시켜주는 것인데 이 값은 i번째 반복시 항상 i가 된다. 즉 단말 노드의 경우에 명시적으로 끝 위치를 적어 두지 않더라도 우리는 필요한 경우에 항상 단말노드의 끝 위치를 알아 낼 수 있다. 따라서 $STree(T_i)$를 만들기 위해서는 2번에 속하는 αt_i만 고려하면 된다. Ukkonen의 알고리즘은 이를 위해 2번에 속하는 가장 긴 접미사에 대한 정보를 유지한다. 즉 i번째 반복시 $STree(T_{i-1})$에서 변경될 가능성이 있는 가장 긴 접미사 α를 나타내는 포인터를 (노드, 부분 문자열의 시작위치)의 쌍으로 표시한다. 이를 Ukkonen은 *active point*라 명명하였다. 여기서 부분 문자열은 항상 T_{i-1}의 접미사로 표시되기 때문에 끝위치는 표시할 필요가 없다. 2번에 속한 즉 변경되어야할 다른 접미사를 나타내는 포인터는 이 *active point*로부터 접미사 링크를 따라가면서 찾을 수 있다 즉 변경될 필요가 있는 가장 긴 접미사부터 짧은 것까지 차례로 트리에 반영된다 그러다가 마지막으로 3번에 속하는 가장 긴 접미사에 도달한다. 이 위치는 *endpoint*라 한다. 여기까지 오면 i번째 반복시 필요한 작업이 완료되고 $STree(T_i)$가 생성된다. 그리고 단순히 이 *endpoint*로부터 한 문자(t_i)만큼 노드나 간선을 따라 아래로 내려간 위치가 다음 $i+1$번째 반복시 필요한 *active point*가 된다.

그리고 내부 노드에서 자식 노드를 찾는데 상수 시간이 걸린다고 하면 문자열 전체에 대해 이 알고리즘을 수행하는 데 걸리는 시간은 문자열의 길이 n에 비례한다

4. 가장 긴 매치 찾기

이번 절에서는 접미사 트리를 Ziv와 Lempel이 제안한 압축 기법에 어떻게 응용할 수 있는 지를 설명한다.

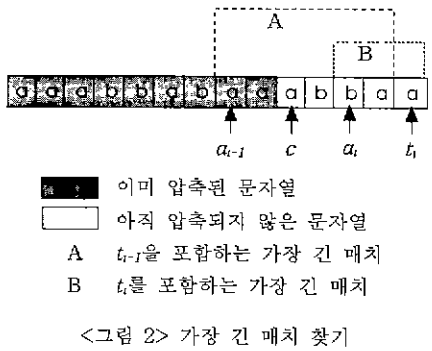
Ziv와 Lempel은 반복되는 문자열을 앞에 나온 문자열에 대한 포인터로 대치시킴으로써 압축할 수 있는 방법들을 제안하였다. LZ77과 같이 제한된 영역에서만 반복된 문자열을 찾는 방법과 LZ78과 같이 지금까지 인코딩된 전체에서 반복성을 찾는 방법 등이 제안되었는데 본 논문에서는 설명을 간단히 하기 위해서 접미사 트리를 LZ78방식의 압축기법에 적용한 경우를 고려하겠다. 하지만 다음에 논의될 내용은 LZ77에도 마찬가지로 적용될 수 있다.

LZ78은 이미 압축된 문자열 전체에 대한 사전을 유지해야 하는데 여기에 접미사 트리가 이용될 수 있다. 즉 이미 $i-1$번째 문자열까지는 인코딩이 되어 있고 i번째 문자가 인코딩될 차례이면 i번째 문자부터 시작하는 부분 문자열과 사전, 즉 접미사 트리와의 가장 긴 매치를 찾아서 인코딩을 한다. 이를 접미사 트리의 생성 알고리즘과 연결시켜 설명하면 $i-1$번째 문자까지 삽입된 $STree(T_{i-1})$를 생성한 다음 가장 긴 매치를 찾아 인코딩하고 인코딩된 문자열까지 또 접미사트리를 생성한다. 그러나 접미사 트리의 생성 알고리즘에 약간의 작업만 추가하면 매치를 따로 찾지 않고도 접미사 트리를 생성하는 과정에서 바로 가장 긴 매치를 찾을 수 있다.

정리 1 문자 t_i를 삽입한 후 즉 $STree(T_i)$에서 *active point*가 나타내는 접미사의 시작위치를 a_i라고 하고 다음 압축될 문자열의 시작위치를 c라고 표시하면 $a_{i-1} \le c < a_i$를 만족하는 인덱스 i $(c \le i \le n)$가 반드시 존재한다 이 때 가장 긴 매치는 부분 문자열 $t_c \ldots t_{i-1}$이 된다. 만약 $i = c$이면 문자 t_c는 문자열 T에서 처음 나오는 것이 된다. 즉 이 때 가장 긴 매치는 빈 문자열이다. (그림2 참조)

증명) *active point*는 항상 접미사 링크를 따라 올라가므로 전체 문자열 T에서 봤을 때 항상 뒤로 움직인다. 즉 $a_{i-1} \le a_i$이다. 따라서 $a_{i-1} \le c < a_i$를 만족하는 인덱스 i를 찾을 수 있다.

*active point*의 정의에 의해 $t_{a_{i-1}} \ldots t_{i-1}$는 T_{i-1}에서 두 번 이상 나타나는 가장 긴 접미사이다 그러므로 부분 문자열 $t_c \ldots t_{i-1}$도 다른 곳에 한 번 이상 나타난다 또한 $t_{a_i} \ldots t_i$는 T_i에서 두 번 이상 나타나는 가장 긴 접미사이다. 즉 문자 t_i를 포함하면서 앞에 한 번 이상 나타난 가장 긴 접미사는 인코딩될 문자 t_c를 포함하지 않는다 따라서 위치 c에서 시작하는 가장 긴 매치는 $t_c \ldots t_{i-1}$이 된다. □



<그림 2> 가장 긴 매치 찾기

예를 들어 <그림 2>에서와 같이 aaabbabaa가 이미 인코딩되어 있고 접미사 트리로 만들어져 있을 때, 다음 문자들을 하나씩 접미사 트리에 삽입을 하면 abba까지는 *active point*가 표현하는 문자열의 시작 위치가 c를 넘지 않는다 하지만 맨 마지막 a가 삽입되고 나면 한 번 이상 앞에서 존재하는 접미사 중 가장 긴 것은 baa가 되어 인코딩될 문자위치 c를 넘어 가게 된다. 따라서 이 때 가장 긴 매치는 abba가 된다.

이를 구현하기 위해서는 a_i의 값을 유지해야 한다 그러나 각 인덱스 i에 대해서 a_i가 전부 유지될 필요는 없고 i번째 반복시에는 a_i값만 있으면 된다 따라서 이 값을 a라는 변수에 유지한다고 했을 때 변수 a의 값은 단조 증가한다. 그리고 이 값은 접미사 링크를 따라 갈 때마다 1씩 증가한다 그리고 접미사 트리를 만들면서 매 반복시마다 $c < a$인지 검사하면 되므로 가장 긴 매치를 찾기 위해 필요한 부가적인 시간은 $O(n)$이 된다.

이 방법은 노드에서 분기를 찾는 것과는 전혀 무관하다. 또한 이 방법은 자기 중복(self-overlapping)을 허용한다는 장점도 있다 자기 중복이란 인코딩될 문자열 일부가 매치에 포함될 수 있는 것을 말한다 예를 들어 a^n인 문자열이 있을 때 맨 처음의 a는 문자열에서 처음 나오므로 이때는 가장 긴 매치는 빈 문자열이다. 그러나 두 번째 인코딩시 중복을 허용하지 않으면 가장 긴 매치는 a이지만 중복을 허용하면 가장 긴 매치는 a^{n-1}이 된다. 즉 두 번째 a는 첫 번째 a와 쌍을 이루고 세 번째 a는 두 번째 a와 네 번째는 세 번째와, 이런 식으로 마지막 a까지 매치를 이룬다. 이렇게 하면 두 번의 인코딩만으로 전체 스트링을 압축할 수 있다.

5. 결 론

본 논문에서는 접미사 트리를 Ziv-Lempel 인코딩 방식에 이용할 때 필수적인 가장 긴 매치를 찾는 새로운 알고리즘을 제시하였다 이 알고리즘은 기존의 루트부터 차례로 비교해 나가는 방식에 비해 몇 가지 장점이 있다. 첫째 본 알고리즘은 따로 매치를 찾는 루틴없이 접미사 트리를 생성하는 과정에서 바로 가장 긴 매치를 찾을 수 있다. 둘째, 자기 중복을 허용함으로써 압축을 좀더 효율적으로 할 수 있다. 셋째, 구현이 매우 간단하다. 본 논문에서 제시된 방법은 항상 1씩 증가하는 카운터 변수만 있으면 되고, 증기를 위한 조건은 접미사 트리 생성 알고리즘에 이미 존재하므로 트리를 검색해 나가야 하는 기존의 방식에 비해 매우 간단하다

본 논문과 관련하여 향후 연구 과제로는 2차원 자료의 압축에 접미사 트리를 이용하는 방법을 개발하는 것이다. 또한 본 논문에서 제시한 알고리즘을 2차원으로 확장할 수도 있을 것이다.

참고 문헌

[1] E. R. Fiala and D. H Greene, Data compression with finite windows, Comm ACM 32(4), 1989, 490-505.

[2] N. J. Larsson, Extended application of suffix trees to data compression, Data Compression Conf., 1996, 190-199.

[3] E. N. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23(2), 1976, 262-272

[4] M. Rodeh, V. R. Pratt and S Even, Linear algorithm for data compression via string matching, J. ACM 28(1), 1981, 16-24.

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

[6] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, IEEE Trans Info Theory 23(3), 1977, 337-343

[7] J. Ziv and A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Info. Theory 25(5), 1778, 523-536