와일드카드 문자를 포함하는 스트링 데이타 사이의 포함관계 확인을 위한 효율적인 알고리즘

(An Effective Algorithm for Checking Subsumption Relation on String Data Containing Wildcard Characters)

김 도 한 † 박 희 진 †† 백 은 옥 †††
(Dohan Kim) (Heejin Park) (Eunok Paek)

요 약 와일드카드 문자를 포함하는 스트링 데이타는 텍스트에 나타나는 특정 패턴을 표현하는 데에 사용될 수 있다. 임의의 두 패턴 사이의 포함 관계는 각 패턴과 매칭이 가능한 모든 스트링의 집합 사이의 포함관계로 나타낼 수 있으며, 포함 관계를 결정하는 것은 패턴이 나타내는 스트링의 집합을 중복성 없이 표현하기 위해 필요하다. 본 논문에서는 이와 같이 패턴의 중복성을 판단하기 위해 와일드카드 문자를 포함하는 스트링 데이타 사이의 포함 관계를 결정하기 위한 효율적인 알고리즘을 제안한다. 먼저 기존의 접미사 트리 알고리즘을 단순하게 확장하여 와일드카드 문자를 포함하는 스트링 데이타 사이의 포함 관계를 확인할 수 있도록 하는 방법과 이러한 접미사 트리를 스트링 데이타의 각 위치 별로 나누어 구성하여 포함 관계를 확인하는 방법을 제안한다.

키워드 : 포함 관계, 접미사 트리, 트라이, 와일드카드 문자

Abstract String data containing wildcard characters may represent certain patterns in texts. A subsumption relation between two patterns can be defined by a subset relation between sets of strings that match those patterns. Thus, the subsumption relation check is important to determine whether each pattern represents a set of strings without any overlap with another pattern. In this paper, we propose an effective algorithm that can determine subsumption relation between strings with wildcard characters. First, we consider a simple extension of the suffix tree algorithm so that it may include wildcard characters and then we propose another method that checks the subsumption relation by dividing a suffix tree structure at each location of string data.

Key words : subsumption relation, suffix tree, trie, wildcard character

1. 서 론

와일드카드 문자를 포함하는 스트링 데이타의 처리는 다양한 응용에서 요구된다. 대표적인 문제로는 와일드카드 문자를 포함하는 두 개의 스트링 데이타가 서로 매치될 수 있는지를 판단하는 것이 있다. 예를 들어 "axbbcxd"와 "cbbxd"의 두 스트링 데이타가 있을 때 ('x'는 와일드카드 문자), 두 번째 스트링 "cbbxd"가 첫 번째 스트링 "axbbcxd"의 두 번째 위치부터 여섯 번째 위치에서 매치되는 것을 확인하는 문제이다.

그러나 본 논문에서 살펴보고자 하는 문제는 이와 달리, 와일드카드 문자를 포함하는 두 스트링 데이타 사이에 포함 관계가 성립하는지를 판단하는 것이다. 여기서 포함 관계는 두 스트링 데이타 "abbxcd"와 "abxxcd"가 있을 때, 와일드카드 문자로 인해 두 번째 스트링 데이타가 첫 번째 스트링 데이타보다 표현할 수 있는 스트링 데이타 집합이 더 많게 되어 첫 번째 스트링 데이타를 포함하는 관계를 말한다.

포함관계에 대한 검사(subsumption test)는 주로 Description Logic이나 XML 스키마 등에서 타입 계층구조(type hierarchy)와 같이 계층정보가 있는 경우, 새로 정의된 것이 계층구조와 부합하는지를 확인하는 경우에 사용되거나[1,2], 데이타베이스에 대한 질의 문장과

· 본 연구는 IMT2000 프로젝트인 IMT2000-C5-2(IT-BT)에 의하여 연구되었음
† 학생회원 : 서울시립대학교 기계정보공학과
dhkim@uos.ac.kr
†† 종신회원 : 한양대학교 정보통신대학 교수
hjpark@hanyang.ac.kr
††† 종신회원 : 서울시립대학교 기계정보공학과 교수
(Corresponding author임)
paek@uos.ac.kr
논문접수 : 2005년 1월 24일
심사완료 : 2005년 7월 21일

같은 논리적 표현에서 변수에 대한 치환이나 부분집합 관계를 통해 포함관계를 확인하는 경우에 사용되어 왔다[3]. 이와 같은 포함관계 모두 와일드카드를 포함하는 스트링에 대한 포함관계 보다 일반적인 문제에 해당하며 복잡도가 매우 높은 문제이다. 예를 들어 논리적 표현에 대한 포함관계 확인 문제는 NP-Complete 문제로 알려져 있다.

와일드카드 문자를 포함하는 스트링 데이타 사이의 포함관계 확인 문제는 단백질 서열을 나타내는 스트링 사이에 존재하는 패턴을 찾아내는 응용프로그램을 개발하는 과정에서 그 필요성이 대두되었다. 단백질 서열을 대상으로 하는 패턴을 발견하는 것이 중요한 이유는 공통된 기능을 갖는 것으로 알려진 단백질 군의 서열 사이에 종종 공통된 서브스트링이 존재하며, 이 공통된 서브스트링이 단순한 형태의 정규 표현식(regular expression)으로 표현될 수 있기 때문이다. 같은 군에 속하는 것으로 알려진 단백질들의 서열 사이에 존재하는 패턴을 일종의 정규 표현식으로 나타내고 이를 데이타베이스화한 것으로 PROSITE 데이타베이스를 들 수 있다[4]. 패턴 발견(pattern discovery) 알고리즘의 대표적인 예로는 PROSITE 패턴 형태와 같은 형태의 패턴을 찾는데 사용된 Pratt 알고리즘이나[5], IBM에서 개발한 SPLASH 등을 들 수 있다[6].

이와 같은 패턴 발견 알고리즘에 의해 생성되는 후보 패턴에는 와일드카드 문자가 흔히 포함되는데 이들 후보 패턴 사이의 포함 관계를 고려하지 않게 되면 각 후보 패턴이 나타내는 스트링 데이타의 집합 사이에 포함관계가 성립함에도 불구하고 후보 패턴 모두가 패턴 발견 알고리즘의 결과로 제공되어, 불필요하게 많은 결과를 생성하게 되고 따라서 사용자의 입장에서는 매우 불편한 요인이 된다. 다음의 예는 위에서 언급한 SPLASH 수행결과의 일부이다. 여기서 '[ ]'는 그룹 기호를 의미하는데, 예를 들어 '[rqk]'는 'r', 'q' 또는 'k'가 매치될 수 있다는 것을 의미한다.

① e x x [rqk] x x x e x x x x x [nde]
② e x x [rqk] x x x e x x x x x [de]
③ e x x [rqk] x x x e

위의 3가지 후보 패턴을 살펴보면 ②번 후보패턴이 가장 구체적인(specific) 스트링 데이타 집합을 나타냄을 알 수 있고, ①번과 ③번은 모두 ②번 후보 패턴을 포함하는 것을 확인할 수 있다. 일반적으로 같은 서열 스트링의 집합을 나타내는 패턴들 사이에서는 패턴의 길이가 더 길고 더 구체적인 패턴이 더 좋은 패턴으로 여겨진다. 따라서 알고리즘이 찾아낸 패턴들 사이의 포함 관계를 파악하여 더 구체적인 후보 패턴만을 선택하는 것이 필요하다..

본 논문에서는 기존의 접미사 트리(suffix tree)를 이 문제에 비교적 단순히 적용하여 와일드카드 문자를 포함하는 스트링 데이타를 처리하는 알고리즘과 스트링 데이타를 위치별 압축 트라이의 집합으로 표현하는 방법 등 두 가지 방법을 제안하고, 이들의 계산복잡도를 비교하여 본다.

2. 개요

앞서 언급한 바와 같이, 본 논문에서 고려하는 스트링 데이타는 패턴 발견 알고리즘에 의해 생성되는 것으로서 다음과 같은 특성을 갖는다. (1) 스트링 데이타를 구성하는 기호는 알파벳이며 이중 기호 'x'는 와일드카드 문자를 나타낸다. (2) 스트링 데이타는 초기에 한꺼번에 주어지지 않고, 외부의 알고리즘에 의해 하나씩 순차적으로 생성된다. (3) 스트링 데이타가 나타내는 패턴은 "구체적 -> 일반적" 순서에 어긋나지 않게 생성된다. 여기서 "구체적/일반적"이라고 하는 것은 부분순서 관계(partial order)를 나타내므로, 새로 생성되는 스트링 데이타가 기존의 어느 스트링 데이타와도 비교할 수 없는 경우도 있다.

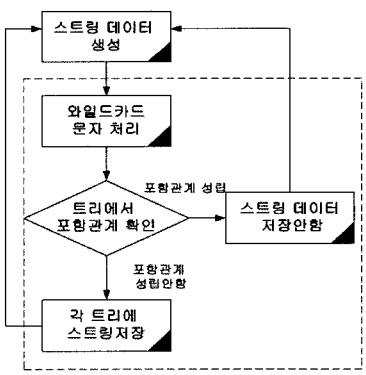

그림 1 알고리즘 흐름도

2.1 기본 정의 및 표기법

와일드카드 문자 'x'는 기호 집합의 어떠한 기호와도 매치되는 것을 의미하고, 그로 인해 스트링 데이타 사이에 포함 관계가 성립할 수 있게 된다. 여기서 포함 관계는 다음과 같이 정의한다.

정의 1. *Ground string* S는 와일드카드 문자를 포함하지 않는 스트링 데이터이다.

정의 2. 스트링 데이타 S1, S2가 있을 때, S1과 매치되는 모든 ground string의 집합을 SS1, S2와 매치되는 모든 ground string의 집합을 SS2라 할 때,

S1 *subsumes* S2 if and only if SS2 ⊆ SS1

예를 들어, 두 개의 스트링 데이타 "aaaxa"와 "axaxa"이 있을 때, 위의 정의에 의하면, "axaxa"가 "aaaxa"를 subsume하게 된다. 같은 의미로, "axaxa"가 "aaaxa"를 "포함한다"고도 하며, "aaaxa"가 "axaxa"보다 더 "구체적", "axaxa"가 "aaaxa"보다 더 "일반적"이라고 한다.

앞서 언급한 바와 같이 본 논문에서 제안하는 알고리즘이 적용될 응용프로그램에서는 스트링 데이타가 외부의 알고리즘에 의해 하나씩 생성이 되고 이때, 좀 더 구체적인 데이타가 늘 먼저 생성된다. 이 때, 스트링 데이타의 생성 특성 상 나중에 생성되는 스트링 데이타가 더 구체적이기 때문에 이미 생성되어 있는 스트링 데이타를 대체하는 경우는 발생하지 않는다. 즉 스트링 데이타의 포함 관계를 파악하여 이미 저장된 스트링 데이타를 새로 생성된 스트링 데이타가 포함하는 경우에는 새로 생성된 스트링 데이타를 버리고, 그렇지 않은 경우에는 스트링 데이타를 추가로 저장한다. 위의 그림 1은 이러한 알고리즘의 흐름을 보여주고 있다.

2.2 와일드카드 문자를 포함한 스트링 데이타 처리

서론에서 소개했던 와일드카드 문자를 포함한 스트링 데이타 사이의 매칭 알고리즘은 텍스트(검색 대상)와 패턴(검색 스트링)을 와일드카드 문자까지 하나의 문자로 간주하여 그대로 접미사 트리에 저장한다. 접미사 트리란, 텍스트의 모든 접미사를 압축된 트라이(trie) 형태로 표현한 것으로 정의한다[7]. 이는 위치 트리의 단순화된 형태로 설계되었는데[8], 길이가 n인 텍스트 T에 대한 접미사 트리는 $O(n)$공간을 필요로 하며, $O(n)$시간에 구축할 수 있다[7,9-13]. 와일드카드 문자가 포함된 텍스트를 접미사 트리 형태로 저장한 이후 longest common extension 알고리즘을 활용하여 와일드카드 문자를 처리하고 패턴이 텍스트에 나타나는지를 확인한다[14]. 이 알고리즘은 패턴이 텍스트에 존재하는지를 확인하기 위한 것으로 패턴과 텍스트 사이의 포함 관계를 결정하는 문제에 직접 적용할 수 없다. 만약 이 접미사 트리에 대해서 포함 관계 확인을 위한 알고리즘을 수정하여 적용한다고 하더라도, 여기서는 와일드카드 문자를 하나의 기호로 간주하여 접미사 트리에 포함시키고 있기 때문에 포함 관계 확인을 위한 탐색을 선형시간에 수행할 수가 없다. 따라서 와일드카드 문자를 포함하는 스트링 데이타 사이의 포함 관계 확인을 위한 새로운 자료구조와 알고리즘이 필요하게 되었다.

와일드카드 문자를 포함한 스트링 데이타를 처리하기 위해서는 우선 스트링 데이타를 와일드카드 문자를 가지지 않는 서브스트링으로 나누어서 고려한다. 서브스트링 $S_{i,j}$ $(1 \le i \le j \le L)$는 스트링 데이타 S = s_1, ..., s_L의 서브스트링 si,...,sj를 의미한다. 예를 들어 S1 = "aaaxa", S2 = "axaxa"일 때, 이들을 와일드카드 문자를 포함하지 않는 서브스트링으로 나누면 다음과 같다.

$$S1_{1,3} = 'aaa',\ S1_{5,5} = 'a'$$
$$S2_{1,1} = 'a',\ S2_{3,3} = 'a',\ S2_{5,5} = 'a'$$

이와 같이 하나의 스트링을 여러 서브스트링으로 나눈 다음 각 서브스트링을 스트링 데이타에서의 위치에 맞추어 비교를 하면 포함 관계를 확인할 수 있다. 위의 예에서 $S1_{1,3}$과 $S1_{5,5}$는 $S2_{1,1}$, $S2_{3,3}$과 $S2_{5,5}$를 스트링 데이타에서의 위치에 맞게 포함하는 수퍼스트링이므로 S2가 S1을 포함한다는 것을 알 수 있다.

이 때 스트링 데이타의 서브스트링 사이의 포함관계를 확인하는 과정은 서브스트링의 접미사(suffix) 또는 접두사(prefix)에 해당하는지를 확인하는 것에 해당하므로 여기서 고려하는 서브스트링을 접미사 트리 자료구조 형태로 저장하면 효율적으로 포함 관계를 파악할 수 있을 것이라 예상할 수 있다.

3. 알고리즘

3.1 접미사 트리를 이용한 포함 관계 확인

2.2절에서 소개한 포함 관계를 확인하는 문제에서 서브스트링의 접미사의 존재를 확인할 때, 스트링 데이타 내에서의 위치에 맞추어 확인해야 한다. 그런데 접미사 트리 알고리즘에서는 위치가 달라도 기호가 같으면 같은 에지(edge)를 공유하여 접미사를 나타내고, 접미사의 위치정보는 말단(leaf) 노드(node)에서 확인할 수 있다. 이 때문에 접미사의 존재를 확인할 때 기호를 먼저 확인하고 말단 노드에서 해당 접미사가 원래의 스트링 데이타 내에서 시작되는 위치를 확인해야 한다. 따라서 접미사 트리 알고리즘을 그대로 사용할 경우, 트리 내에서 접미사를 찾은 이후 접미사에 해당하는 노드의 모든 하위(descendant) 말단 노드에 저장된 접미사의 텍스트 내에서의 시작위치를 확인해야 하므로 탐색에 필요한 수행시간이 길어지게 된다.

다음과 같은 스트링 데이타 집합을 가지고 접미사 트리를 이용하여 포함 알고리즘을 구현하는 경우를 살펴보자.

S1 = "aaaxa"
S2 = "axaaa"
S3 = "bbaxa"
S4 = "aaxaa"
S5 = "axaxa"

위의 스트링 데이타를 와일드카드 문자를 포함하지 않는 서브스트링으로 표현하면 다음과 같다.

$S1_{1,3}$ = 'aaa', $S1_{5,5}$ = 'a'
$S2_{1,1}$ = 'a', $S2_{3,5}$ = 'aaa'
$S3_{1,3}$ = 'bba', $S3_{5,5}$ = 'a'
$S4_{1,2}$ = 'aa', $S4_{4,5}$ = 'aa'
$S5_{1,1}$ = 'a', $S5_{3,3}$ = 'a', $S5_{5,5}$ = 'a'

여기서 스트링 데이타 S1, S2, S3, S4는 어떤 임의의 쌍 사이에도 포함 관계가 존재하지 않기 때문에 이들은 버려지지 않고 남아서 접미사 트리에 저장되지만, 스트링 데이타 S5는 스트링 데이타 S1, S2를 포함하므로 버려지게 된다. 이를 그림으로 나타내면 아래의 그림 2와 같다.

그림 2에서 각 노드는 접미사를 나타내기 위해 사용되었고, 말단 노드에 표시된 숫자 쌍은 각각 스트링 데이타의 번호와 접미사의 시작 위치를 나타낸다. 접미사의 시작 위치는 2.2절에서 보았던 포함 관계 확인을 위해 필요한 정보이고, 스트링 데이타의 번호는 포함 관계가 성립하기 위해서는 스트링 데이타의 모든 서브스트링이 하나의 같은 스트링 데이타에 존재하여야 하기 때문에 이를 확인하기 위해 필요하다.

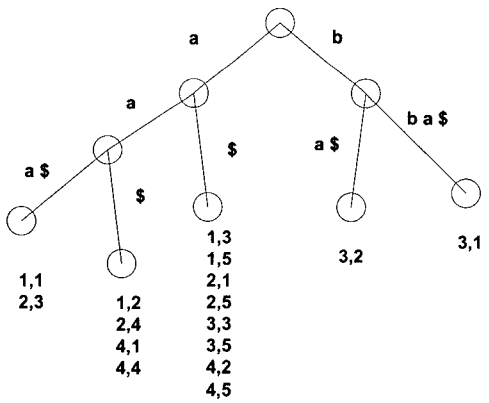

그림 2 접미사 트리 적용

그림 2에서 확인할 수 있는 것처럼, 스트링 데이타 S5의 서브스트링 $S5_{1,1}$이 접미사 트리에 포함되어 있는 다른 스트링 S1, S2, S3, S4 중 어느 하나를 포함하는지를 확인할 때 $S5_{1,1}$의 스트링 값 'a'를 접미사 트리에서 우선 찾은 다음, $S5_{1,1}$를 접미사나 접두사로 갖는 스트링에서 'a'가 어느 위치에서 시작하는지를 확인하기 위해서는 'a' 이하의 모든 말단 노드에 대해서 'a'를 포함하는 서브스트링의 시작위치를 확인해야 한다. 즉, 'a$', 'aa$', 'aaa$'를 서브스트링으로 갖는 모든 스트링 데이타에 대해서 'a$', 'aa$', 'aaa$'가 각각 어느 위치에서 시작되는지를 확인해야 하는 것이다. 이러한 시작위치를 확인하는 문제를 좀 더 효율적으로 처리하기 위해 서브스트링 데이타를 스트링 데이타의 각 위치에 따라 트리로 구성하는 방법을 다음에서 제안한다.

3.2 접미사 트리를 변형한 포함관계 확인

스트링 데이타로부터 만들어진 와일드카드 문자를 포함하지 않는 서브스트링의 위치정보를 쉽게 확인하기 위해 스트링 데이타의 각 위치마다 접미사를 저장하는 압축 트라이를 생성하는 방법에 대해 살펴본다.

3.1절에서 정의한 스트링 데이타의 서브스트링에 대해서 각각 접미사를 구하고 이를 각 위치별로 나타내면 다음과 같다. 길이가 L인 스트링 데이타 S의 서브스트링 $S_{i,j}$의 각 접미사를 S에서의 위치 k를 이용해서($k \le L$, $i \le k \le j$, i, j는 각 서브스트링 범위) 나타내며 이를 S_k로 표현한다. 이를 3.1절의 예에 대해 적용해 보면 다음과 같다.

$S1_1$ = 'aaa', $S1_2$ = 'aa', $S1_3$ = 'a', $S1_5$ = 'a'
$S2_1$ = 'a', $S2_3$ = 'aaa', $S2_4$ = 'aa', $S2_5$ = 'a'
$S3_1$ = 'bba', $S3_2$ = 'ba', $S3_3$ = 'a', $S3_5$ = 'a'
$S4_1$ = 'aa', $S4_2$ = 'a', $S4_4$ = 'aa', $S4_5$ = 'a'
$S5_1$ = 'a', $S5_3$ = 'a', $S5_5$ = 'a'

S1, S2, S3, S4에 대해 위와 같이 각 위치별로 나타나는 모든 서브스트링의 접미사를 원래 스트링에서의 위치에 따라 압축 트라이로 표현하면 그림 3과 같이 나타낼 수 있다

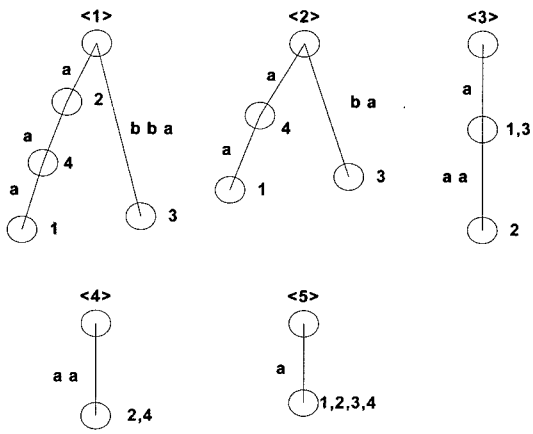

그림 3 접미사 트리 변형 적용

그림 3에서 괄호 안의 숫자는 원래 스트링 데이타에서 각 접미사가 시작되는 위치를 나타낸다. 압축 트라이의 노드에 표시된, 괄호가 없는 숫자는 저장된 접미사를 가지고 있는 스트링 데이타의 인덱스를 나타낸다. 이와 같은 정보는 3.1절에서와 마찬가지로 어떤 스트링의 각 서브스트링을 포함하는 하나의 스트링이 접미사 트리 안에 있는지를 확인하기 위함이다.

앞의 예에 대해서 S51(서브스트링 'a')의 경우를 3.1절의 알고리즘과 비교하여 보면, 서브스트링 $S5_1$의 시작 위치가 '1' 이므로 첫 번째 압축 트라이에서 바로 서브스트링의 존재여부를 확인할 수 있다. 따라서 3.1절의 알고리즘에서와 달리 위치에 따라 해당 트라이를 검색함으로써 서브스트링의 존재여부를 효과적으로 확인할 수 있게 된다.

3.3 알고리즘 개선 방안

여기에서는 3.2절에서 제안한 자료구조의 성능을 개선하기 위한 방안에 대해서 살펴본다. 첫 번째는 기존의 접미사 트리 알고리즘에서 사용하는 접미사 링크(suffix link)의 적용이다[14]. 스트링의 i번째 위치에 해당하는 트라이에서 접미사를 생성하고, i+1 번째 트라이에서 접미사를 생성할 때, 인접한 트라이 사이에는 접미사 링크를 구현하여 이후에 다른 접미사를 생성할 때 에지를 탐색하는 시간을 줄일 수 있다. 그림 4는 그림 3의 예에 대해 접미사 링크를 구현한 결과를 보여준다.

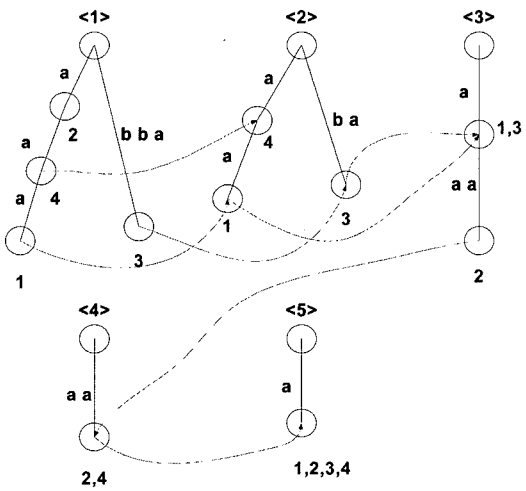

그림 4 접미사 링크 적용

또 다른 개선 방안은 새로운 스트링 데이타의 포함관계를 확인할 때 탐색했던 공통노드에 대한 정보를 저장하여, 새로운 스트링 데이타가 이전에 생성된 스트링 데이타들과 포함 관계가 성립하지 않아서 트라이에 삽입될 때 동일한 확인작업을 수행하지 않고 바로 트라이에 접미사를 생성할 수 있도록 하는 것이다.

예를 들어 새로운 스트링 데이타를 S6 = "aabxa"라고 하면, 스트링 데이타 S6는 와일드카드 문자를 포함하지 않는 두 개의 서브스트링 $S6_1$ = 'aab'와 $S6_5$ = 'a'로 나누어지게 되고 이 서브스트링들이 첫 번째 트라이와 다섯 번째 트라이에 존재하는지 확인하게 된다. 그림 4에서 확인할 수 있듯이 첫 번째 트라이에서 서브스트링 $S6_1$의 첫 번째 기호 'a'와 두 번째 기호 'a'는 트라이에 존재하고 있지만 마지막 기호 'b'는 존재하지 않기 때문에 생성된 스트링 데이타 S6은 이전에 생성된 스트링 데이타들과 포함 관계를 가지지 않는 것으로 판단되어 트라이에 삽입되어야 한다. 이 때, 첫 번째 트라이에 대해서는 이미 포함 관계 확인을 위해서 비교가 첫 번째와 두 번째 문자에 대해 이루어졌으므로 생성된 스트링 데이타의 서브스트링 $S6_1$의 두 번째 기호 'a'와 첫 번째 트라이의 왼쪽 에지의 두 번째 노드는 공통된 정보를 나타낸다는 것을 알고 있다. 따라서, 이 위치를 기억하고 있으면, 해당 스트링 데이타를 트라이에 삽입할 때 동일한 비교작업을 수행하지 않고 기억되어 있는 위치에서 서브스트링 $S6_1$의 마지막 기호 'b'에 대한 생성만을 처리하면 된다. 이렇게 처리함으로써 새로운 스트링 데이타를 트라이에 삽입할 때 좀 더 효율적으로 처리할 수 있게 된다.

다음은 3.2절에서 제안한 접미사의 압축 트라이를 이용해서 스트링 데이타를 표현했을 때, 포함관계를 확인하는 알고리즘을 서술한 것이다.

```
Loop (서브스트링 개수 만큼)
  해당 트라이에 생성된 노드가 없으면 포함 안됨
    return false
  해당 트라이에 접미사의 가장 최대 끝 위치가 비교하려는
  서브스트링의 끝보다 적으면 포함 안됨
  return false  //접미사의 최대 끝 위치 정보 유지하고 있음
                //노드도 있고 접미사의 끝 위치도 포함하면
                //root 노드에서 비교시작
  Loop (서브스트링의 시작위치에서 끝 위치까지)
    //현재 노드 아래 에지 중 시작 기호가 같은 에지를 찾음
    if 기호가 같은 에지가 없을 때
    {
      이전 까지 공통된 노드 정보를 현재 트라이 정보에 저
      장함
      중복 포함 안됨 return false
    }
    else
    {
       Loop (해당 노드의 시작 위치에서 끝 위치까지)
          //해당 노드 안에서의 기호를 계속 비교
          다른 기호가 나타나면 포함 안됨 return false
       Loop end
    }
  Loop end
  //하나의 서브스트링이 포함된 상태
  현재 트라이 정보에 공통 노드 정보를 저장
  //서브스트링 전부가 하나의 데이타에 포함되어야 전체가
  //포함되었다고 할 수 있으므로 현재 서브스트링을
  //포함한 데이타정보를 저장
  저장된 정보와 계속 스트링 데이타 정보를 비교하고 공통적
      인 데이타 정보가 없게 되면 포함 안됨
  return false
Loop end
공통된 데이타 정보가 하나 이상 있을 경우는 중복 포함
```

위의 포함관계 확인 알고리즘에 의해 포함 관계가 없다고 판명된 스트링 데이타를 트라이에 삽입하는 알고리즘을 아래에 나타냈다.

```
Loop (생성할 서브스트링의 시작위치부터 끝 위치 까지)
  if현재 노드아래 자식(child) 노드 없으면
   { 새로운 노드 생성
     현재 노드와 부모/자식 연결(parent /son connect)
     생성된 노드를 접미사 링크로 사용할 수 도 있기 때문에
        저장해 둠
     현재 노드에 접미사 링크 정보 있으면 다음 위치 (현재
        위치 + 1) 트라이 정보에 저장
   }
  else //현재 노드아래 자식 노드 있으면
   { //기호가 같은 에지를 찾음, 접두어 공유하기 위해서
     if 기호가 같은 에지 없으면
      {
        새로운 노드를 생성하고 현재 노드의 자식노드와 형
           제 연결(sibling connect)
        생성된 노드를 접미사 링크로 사용할 수 도 있기 때
           문에 저장해 둠
        현재 노드에 접미사 링크 정보 있으면 다음 위치 트
           라이 정보에 저장
      }
      else
      {
        Loop (해당 노드의 시작위치에서 끝 위치 까지)
          //기호가 같은지를 계속 비교
          if 에지 중간에서 달라지고 현재 서브스트링 길이
             가 끝나면
          {
            내부 노드를 생성하고 부모/자식/형제 연결
            생성된 노드를 접미사 링크로 사용할 수도 있기
               때문에 저장해 둠
            현재 노드에 접미사 링크 정보 있으면 다음 위
               치 트라이 정보에 저장
          }
          if 해당 노드에서 끝나면
            해당 노드에 현재 서브스트링의 데이타 정보(데
               이타 인덱스)만을 저장
          if 에지 중간에 달라지고 현재 서브스트링 길이가
            남아 있으면
            내부 노드를 생성하여 에지 중간에 연결하고 새
               로운 노드를 생성하여
            새로운 말단 노드로 만들고 부모/자식/형제 연결
            생성된 노드를 접미사 링크로 사용할 수 도 있
               기 때문에 저장해 둠
            현재 노드에 접미사 링크정보 있으면 다음 위치
               트라이 정보에 저장
        Loop end
     }
   }
Loop end
```

3.4 접미사 트리 구조와 달라진 점

와일드카드 문자에 대한 처리 알고리즘과 서브스트링의 위치정보를 효율적으로 확인하기 위한 방법이 적용되면서 기존의 접미사 트리 알고리즘이 가지는 특성과 달라진 점은 종료문자(terminal character)를 사용하지 않는 것이다.

기존의 접미사 트리 알고리즘에서는 종료문자를 사용해서 한번 말단 노드로 생성된 노드는 계속해서 말단 노드로 남아 있게 되는 특성을 이용한다. 이는 스트링 데이타 일부의 접미사를 접미사 트리에 생성하고 생성한 접미사의 길이를 증가시키며 접미사 트리를 생성하는 알고리즘에서 효율성을 높일 수 있었다[14]. 그러나 본 논문에서 제안한 알고리즘은 스트링 데이타를 와일드카드 문자를 포함하지 않는 서브스트링으로 분리한 뒤 이것에 대한 접미사를 압축 트라이로 생성하게 된다. 이 때문에 접미사를 생성할 때 한번에 전체 길이의 접미사를 트라이에 삽입하게 되고 따라서 접미사의 길이를 증가시켜 가면서 접미사 트리를 생성했던 방법과 달리 종료문자를 사용할 필요가 없게 되었다. 이러한 변화로 접미사는 항상 말단 노드에서 끝나지 않게 되었고 접미사 링크도 내부 노드(internal 노드)가 아닌 곳에서 생성되게 되었다.

3.5 알고리즘의 복잡도(complexity) 비교

본 논문에서 제안하는 두 가지 알고리즘의 복잡도를 고려해 보면 다음과 같다. 비교 대상인 스트링 데이타의 개수를 n (모든 스트링 데이타의 길이는 포함관계를 확인할 스트링 데이타의 길이보다 같거나 길다고 가정.), 저장된 각각의 스트링 데이타를 $R_i(1 \le i \le n)$, R_i를 구성하는 기호 개수를 $|R_i|$, 와일드카드 문자를 제외한 기호의 개수를 $|R_i|'$, 포함관계를 확인할 스트링 데이타를 P, P를 구성하는 기호 개수를 $|P|$, 와일드 카드 문자를 제외한 기호의 개수를 $|P|'$라고 하자.

우선 매우 단순하게 모든 R_i에 대해서 P가 각 R_i를 포함하는지를 반복적으로 비교하는 알고리즘을 생각해 보자. 이 경우 P의 와일드카드 문자를 제외한 모든 위치의 기호가 R_i와 같아야 한다. 이러한 단순 알고리즘의 시간 복잡도는 $O(|P|')$이고, 이러한 비교를 n번 수행하여야 하므로 전체 검색 시간은 $O(n|P|')$가 된다. 그리고 공간 복잡도는 각 스트링 데이타의 길이의 합인 $O(\sum|R_i|)$가 된다.

이 단순 알고리즘과 비교하여 본 논문에서 제시한 알고리즘의 공간 및 시간 복잡도는 다음의 표 1과 같다.

표 1에서 occ_i는 P의 i번째 서브스트링이 전체 비교 대상 스트링 데이타에서 발견되는 횟수, $locc_i$는 occ_i에서 P와 동일한 위치에 서브스트링이 발견되는 횟수를 의미한다.

검색시간의 분석을 쉽게 하기 위해, 검색을 두 단계로 나누어 생각하자. 트리를 따라서 패턴의 특정 서브스트링이 있는지 확인하는 단계와, 서브스트링이 트리에 나타나는 경우, 이것이 비교 대상이 되는 스트링 데이타에

표 1 알고리즘 복잡도 분석

알고리즘	접미사 트리 적용	접미사 트리 변형
① P에서 와일드카드 문자를 제외한 각 서브스트링을 접미사 트리에서 확인	$O(\|P'\|)$	$O(\|P'\|)$
② 초기화	$O(n)$	$O(n)$
③ P의 i번째 서브스트링을 해당 위치에 가지고 있는 스트링 데이타가 있는지를 검색하는 데 필요한 시간	$O(occ_i)$	$O(locc_i)$
④ 검색 시간	$O(\|P'\| + n + \sum occ_i)$	$O(\|P'\| + n + \sum locc_i)$
⑤ 공간 복잡도	$O(\sum \|R_i\|')$	$O(\sum \|R_i\|')$

서 해당 위치에 나타나는 경우가 있는지를 검사하는 단계로 나누어 보자. ①은 검색의 대상이 되는 P의 i번째 서브스트링이 접미사 트리나 해당 위치의 압축 트라이에 나타나는지 노드를 따라 이동하는 데에 필요한 시간을 나타낸다. ②와 ③이 스트링 데이타에서 해당위치에 나타나는 경우가 있는지를 검사하는데 필요한 시간을 나타낸다.

접미사 트리 적용 알고리즘의 경우, 접미사 트리의 단말노드에서 어느 스트링 데이타의 어느 위치에서 해당 접미사가 나타나는지를 연결리스트 (linked list)를 사용해서 유지한다고 하자. 패턴 P의 각 서브스트링에 대해서 해당 위치에 서브스트링을 가지고 있는 스트링 데이타들에 대한 색인을 구하기 위해서는 서브스트링이 스트링 데이타에 나타나는 회수인 $O(occ_i)$만큼, 즉 해당 서브트리의 단말에 저장된 색인의 수에 해당하는 시간이 필요하다. 각 서브스트링을 포함하는 스트링 데이타의 색인들에 대해 교집합을 구하는 과정이 필요한데, 이는 n개의 통을 0으로 초기화 한 다음, 각 서브스트링에 대해 구한 스트링 데이타의 색인에 해당하는 통에 서브스트링의 인덱스를 덮어 씀으로써 (이때 이전 서브스트링 인덱스가 있는 경우에만 덮어 쓰고, 그렇지 않으면 그대로 둔다) 구할 수 있다. 이렇게 할 경우, 마지막 서브스트링의 인덱스가 남아 있는 통이 있으면 패턴을 포함하는 스트링 데이타가 이미 저장되어 있는 것이다.

접미사 트리 변형 알고리즘의 경우에는 단말 노드 뿐만 아니라 내부 노드에도 스트링 데이타의 색인이 저장된다. 이때에도 연결리스트 형태로 색인을 유지하게 되면 해당 위치에 서브스트링을 갖는 스트링의 개수인 $O(locc_i)$ 만큼의 시간이 필요하고, 교집합을 구하는 과정은 동일하게 진행한다고 하면, n개의 통을 0으로 초기화하는데 $O(n)$ 시간이 필요하다. ④는 총 검색 시간으로 ①, ②, ③을 모두 합한 시간이다.

마지막으로 ⑤는 알고리즘이 요구하는 공간을 나타내는데 두 알고리즘 모두 접미사 트리 알고리즘과 같이 스트링 데이타의 크기에 대해 선형적인 공간을 사용하게 된다. 또한 포함 관계를 확인한 후에 트리에 추가하는 시간도 두 알고리즘 모두 패턴의 길이에 대해 선형인 시간에 가능하다.

위의 두 알고리즘의 복잡도를 비교해 보면 occ_i와 $locc_i$에 있어서만 차이가 나는데, 정의에 의하면 $locc_i$는 반드시 occ_i보다 작거나 같으므로 접미사 트리를 변형한 알고리즘이 검색시간에 있어서 좀더 효율적임을 알 수 있다. 또한 앞에서 간단히 언급한 단순비교 알고리즘의 검색시간 $O(n|P|')$와 비교해 본다면, $\sum locc_i$는 최악의 경우에도 $n|P|'$보다 작다. 최악의 경우는, 알파벳이 단 하나뿐인 경우에 해당한다. 그러나 현실적으로 단백질 서열에 대한 패턴을 고려하는 경우, 알파벳의 크기는 적어도 20이 되기 때문에, 확률적으로 특정한 위치에 길이 2인 서브스트링이 나타날 확률은 1/400, 길이 3인 서브스트링이 나올 가능성은 1/8000로 급격히 그 가능성이 줄어든다는 점을 감안한다면, 제안하는 알고리즘이 효율적임을 알 수 있다.

4. 결 론

와일드카드 문자를 포함한 스트링 데이타 사이의 포함 관계를 파악하는 데 있어, 스트링 데이타에서 와일드카드 문자를 제거하여 생성된 여러 개의 서브스트링으로 나누어 처리를 하였다. 이러한 방법은 서브스트링을 스트링 데이타의 위치에 맞추어 확인해야 하기 때문에, 효율적으로 서브스트링을 확인할 수 있도록 접미사 트리 알고리즘을 활용한 두 가지 방법을 제안하였다.

두 가지 방법의 알고리즘 복잡도는 큰 차이를 보이지 않지만, 트리의 노드에서 유지해야 하는 정보와 노드에 저장된 정보를 검색하는 방법을 고려하면 두 번째 알고리즘이 좀 더 효율적임을 알 수 있다.

이 두 가지 방법은 단백질 서열에 대한 패턴 발견 알고리즘에서 후보패턴을 저장하고 포함관계를 확인하기 위해 고안되었으나 일반적으로 와일드카드 문자를 포함하는 스트링 데이타 사이의 포함 관계를 검사하는 경우에 효율적인 알고리즘으로 활용할 수 있다.

참 고 문 헌

[1] I. Horrocks and P. F. Patel-Schneider, Optimising description logic subsumption, Journal of Logic

and Computation, 9(3), 267-293, 1999.
[2] G. M. Kuper and J. Simeon, Subsumption for XML types, Proc. Of International Conference on Database Theory, London, 2001.
[3] C. Chang and R. Lee, Symbolic logic and mechanical theorem proving, Academic Press, 1973.
[4] C. Sigrist, L. Cerutti, N. Hulo, A. Gattiker, L. Falquet, M. Pagni, A. Bairoch, and P. Bucher, PROSITE: A documented database using patterns and profiles as motif descriptors, Brief Bioinformatics, Vol. 3 no. 3, 265-274, 2002.
[5] Inge Jonassen, Efficient discovery of conserved patterns using a pattern graph, CABIOS, 13, 509-522, 1997.
[6] Andrea Califano, SPLASH: structural pattern localization analysis by sequential histograms, Bioinformatics, Vol. 16 no. 4, 341-357, 2000.
[7] E. M. McCreight, A psace-economical suffix tree construction algorithm, JACM 23, 262-272, 1976.
[8] P. Weiner, Linear pattern matching algorithms, Proc. 14th IEEE Symp. Switching and Automata Theory, 1-11, 1973.
[9] M.T. Chen and J. Seiferas, Efficient and elegant subword tree construction, In A. Apostolico and Z. Galil, editors, Combinatorial Algorithms on Words, NATO ASI Series F: Computer and System Sciences, 97-107, 1985.
[10] M. Farach, Optimal suffix tree construction with large alphabets, FOCS, 137-143, 1997.
[11] M. Farach-Colton, P. Ferragina and S. Muthukrishnan, On the sorting-complexity of suffix tree construction, JACM 47, 987-1011, 2000.
[12] S. Kurtz, Reducing the space requirement of suffix trees, Software Practice and Experience, 29, 1149-1171, 1999.
[13] E. Ukkonen, On-line construction of suffix trees, Algorithmica, 14, 249-260, 1995.
[14] Dan Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge University Press, 1997.

김 도 한
2001년 서울시립대학교 정밀기계공학과 학사. 2003년 서울시립대학교 기계정보공학과 석사. 2005년~현재 삼성전자 반도체총괄 메모리사업부 연구원. 관심분야는 컴퓨터알고리즘, 시스템프로그래밍

박 희 진
정보과학회논문지 : 시스템 및 이론 제 32 권 제 5,6호 참조

백 은 옥
1985년 서울대학교 전자계산기공학과 학사. 1991년 Stanford University 전산과 박사. 1992년~1995년 서울대학교 컴퓨터신기술공동연구소. 1995년~2000년 엘지종합기술원 책임연구원. 2001년~현재 서울시립대학교 기계정보공학과 교수. 관심분야는 생물정보학, 인공지능, 컴퓨터알고리즘