Wildcard character를 포함하는 String Data 사이의 Subsumption 관계 확인을 위한 효율적인 알고리즘

김도한[0], 박희진[1], 백은옥[2]
[0.2] 서울시립대학교 기계정보공학과
dhkim@uos.ac.kr[0], paek@uos.ac.kr[2]
[1] 한양대학교 정보통신학부
hjpark@hanyang.ac.kr

An effective algorithm for checking subsumption relation on string data containing wildcard characters

Dohan Kim[0], Heejin Park[1], Eunok Paek[2]
[0.2] Department of Mechanical and Information Engineering, University of Seoul
[1] College of Information and Communications, Hanyang University

요 약

본 논문에서는 wildcard character를 포함하는 문자열의 집합을 대상으로, 이들 사이의 subsumption 관계를 파악하여 더 구체적인 정보를 가지는 문자열들의 집합을 구하고자 하는 것이다. 이를 위해 기존의 suffix tree 알고리즘이 wildcard character를 포함하는 문자열을 처리할 수 있도록 단순 적용한 방법과 trie의 집합을 이용하여 wildcard character를 포함한 문자열을 처리하는 두 가지 방법을 고려하였다.

1. 서 론

Wildcard character를 포함하는 string의 처리는 다양한 응용에서 요구된다. 대표적인 문제로는 wildcard character를 포함하는 두 개의 string 사이에 match가 가능한지를 판단하는 알고리즘이 있다. 그러나 본 논문에서는 wildcard character를 포함하는 두 string 사이에 subsumption 관계가 성립하는지를 판단하는 문제를 살펴보고자 한다.

이와 같은 문제는 단백질 서열을 나타내는 string들 사이에 존재하는 pattern을 찾아내는 application을 개발하는 과정에서 그 필요성이 대두되었다. 단백질 서열을 대상으로 하는 pattern discovery algorithm의 대표적인 예로는 Prosite pattern을 [1] 찾는데 사용된 Pratt 알고리즘이나 [2] IBM에서 개발한 SPLASH 등을 들 수 있다. [3] 이와 같은 pattern discovery 알고리즘에 의해 생성되는 candidate pattern에는 wildcard character가 흔히 포함되는데 이들 candidate pattern 사이의 subsumption 관계를 고려하지 않게 되면 각 candidate pattern이 나타내는 string의 집합 사이에 포함관계가 성립함에도 불구하고 두 candidate pattern 모두가 pattern discovery의 결과로 제공되어, 불필요하게 많은 결과를 생성하게 되고 따라서 사용자의 입장에서는 매우 불편한 요인이 된다.

두 개의 서로 다른 pattern이 같은 coverage를 갖는 경우, pattern의 길이가 더 길거나 더 구체적인 (specific) pattern을 더 좋은 pattern으로 여긴다. 따라서 알고리즘이 찾아낸 pattern들 사이의 subsumption 관계를 조사하여 더 좋은 pattern만을 결과에 포함시키는 것이 필요하다.

본 논문에서는 기존의 suffix tree를 이 문제에 비교적 단순히 적용하여 wildcard character를 포함한 문자열을 처리하는 알고리즘과 문자열을 Trie의 집합으로 표현하는 방법 두 가지를 살펴보고 이들의 장단점을 비교한다.

2. 개 요

2.1 기본 정의 및 표기법

본 논문에서 고려하는 문자열 data는 다음과 같은 특성을 갖는다. (1) 문자열을 구성하는 symbol은 영어의 alphabet이며 이중 symbol 'x'는 wildcard character를 나타낸다. (2) 문자열 data는 초기에 한꺼번에 주어지지 않고, 외부의 알고리즘에 의해 하나씩 순차적으로 생성된다. (3) 문자열 data는 "specific-to-general"의 순서로 생성된다. 여기서 specific/general은 partial order이므로, 새로 생성되는 문자열이 기존의 어느 문자열과도 비교할 수 없는 경우도 있다.

Wildcard character 'x'는 symbol set의 어떠한 symbol과도 match가 되는 것을 의미하고, 그로 인해 문자열 data 사이에 subsumption 관계가 존재하게 된다. 여기서 subsumption 관계는 다음과 같이 정의한다.

정의 1. *Ground string* S는 wildcard character를 포함하지 않는 문자열이다.

정의 2. 문자열 S①, S②가 있을 때, S①과 match되는 모든 ground string의 집합을 SS①, S②와 match되는 모는 ground string의 집합을 SS②라 할 때,

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

예를 들어, 두 개의 문자열 "aaaxa"와 "axaxa"이 있을 때, "axaxa"가 "aaaxa"를 subsume 하며, 이 경우, "aaaxa"가 "axaxa" 보다 specific 하다고 한다.

앞서 언급한 바와 같이 본 논문에서 제안하는 알고리즘이 적용될 application에서는 문자열 data가 외부의 알고리즘에 의해 하나씩 생성이 되고 이때, specific data가 늘 먼저 생성된다. 외부 application에서는 이렇게 생성된 문자열 data 중에서 specific이라는 partial order 관계에 의해 least element에 해

당하는 것만을 유지하여야 하는데, 문자열의 생성 특성 상 나중에 생성되는 data가 specific하여 이미 생성되어 있는 data를 대체하는 경우는 발생하지 않는다. 즉 문자열 data의 subsumption 관계를 파악하여 이미 저장된 data를 새로 생성된 문자열 data가 subsume 하는 경우에는 새로 생성된 문자열 data를 버리고 그렇지 않은 경우에는 subsumption 관계가 없으므로 문자열 data 저장장소에 추가한다.

2.2 Wildcard character 처리 방법

서론에서 소개했던 wildcard를 포함한 string match 알고리즘은 text(검색 대상)와 pattern(검색 string)을 그대로 suffix tree에 저장한다. 이후 longest common extension 알고리즘을 활용하여 wildcard character를 처리하고 pattern이 text에 나타나는지를 확인한다.[4] 이 알고리즘은 pattern이 text에 존재하는지를 확인할 수는 있지만 text와 pattern 사이에 subsumption 관계를 파악할 수는 없다. 또한 wildcard character를 하나의 symbol로 간주하여 suffix tree에 포함시키고 있기 때문에 subsumption test를 위한 탐색을 linear time에 수행할 수가 없다. 따라서 wildcard character를 포함하는 string 사이의 subsumption test를 위한 다른 알고리즘이 필요하게 되었다.

Wildcard character를 포함한 문자열을 처리하기 위해 문자열을 wildcard character를 가지지 않는 substring으로 나누어서 고려한다. Substring $S_{i,j}$ ($1 \le i \le j \le L$)는 문자열 $S = s_1, \ldots, s_L$의 substring si,...,sj를 의미한다.

예를 들어 S① = "aaaxa", S② = "axaxa"일 때, 이들의 각 substring은 다음과 같다.

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

이 상태에서 각 substring들을 문자열 data에서의 위치에 맞추어 비교를 하면 subsumption 관계를 확인할 수 있다. 위의 예에서 $S①_{1,3}$과 $S①_{5,5}$는 $S②_{1,1}$과 $S②_{3,3}$, $S②_{5,5}$를 문자열에서의 위치에 맞게 모두 가지고 있으므로 S②가 S①을 subsume한다는 것을 알 수 있다.

이 경우, S①이 저장되어 있는 문자열 data set에 들어 있고, S②가 새로이 생성되어 subsumption 관계를 파악하는 상황이 가능한데, 이는 문자열 data 중 더 specific 한 것이 항상 더 먼저 생성되기 때문이다.

이 때 문자열의 substring이 나타나는 것을 확인하는 과정은 substring의 suffix (또는 prefix)의 존재를 확인하는 것에 해당하므로 여기서 고려하는 substring을 suffix tree의 형태로 저장하면 효율적으로 subsumption 관계를 파악할 수 있을 것이라 예상할 수 있다.[4]

3. 알고리즘

3.1 Suffix tree 이용한 방법

2.2에서 소개한 subsumption test 문제에서 substring의 suffix의 존재를 확인할 때 문자열 내에서의 위치를 함께 확인해야 한다. 그런데 suffix tree 알고리즘에서는 위치가 달라도 symbol value가 같으면 같은 edge로 suffix를 나타내기 때문에, suffix를 비교할 때 symbol value를 먼저 확인하고 leaf node에서 해당 suffix의 문자열 내에서의 시작위치를 비교해야 된다. 그런데, suffix의 시작위치에 대한 정보는 leaf node에서만 파악할 수 있기 때문에, suffix를 찾은 이후 suffix node의 모든 descendant node에 대해서 시작위치를 비교해야 한다는 문제가 있다.

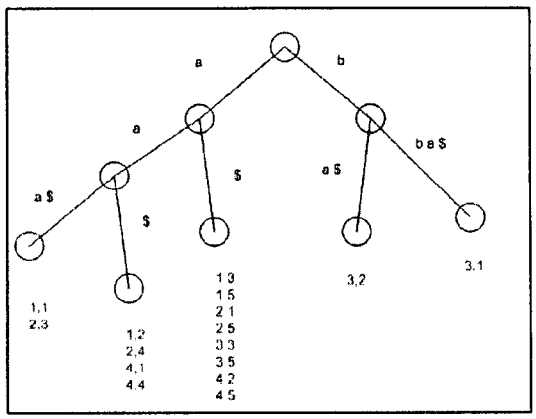
예를 들어, 다음과 같은 data set을 가지고 suffix tree를 구현하는 경우를 살펴보자.

① "aaaxa"
② "axaaa"
③ "bbaxa"
④ "aaxaa"
⑤ "axaxa"

위의 data를 substring으로 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 문자열 ①, ②, ③, ④ 사이에는 어떠한 subsumption 관계도 존재하지 않기 때문에 이들은 버려지지 않고 남아서 suffix tree에 저장되지만, 문자열 ⑤는 문자열 ①, ②를 subsume하므로 버려지게 된다. 이를 그림으로 나타내면 다음과 같다.



[그림 1] Suffix tree 적용 예

[그림1]에서 확인할 수 있는 것처럼, 문자열⑤의 $S⑤_{1,1}$을 확인할 때 'a'를 Tree에서 찾은 후에 시작위치를 확인하기 위해서는 모든 descendant node에 대해서 시작위치를 검사해야 한다. 따라서 문자열 data를 suffix tree로 구성하는 것은 linear time에 가능하지만, substring을 확인하는 것은 linear time에 불가능하게 된다.

이 문제는 suffix tree가 symbol 값을 위치정보보다 우선적으로 비교하기 때문에 발생하는 것이므로 suffix tree에서 위치정보를 쉽게 파악할 수 있도록 자료구조를 변경할 필요가 있다. 다음 절에서는 trie를 이용해 위치정보를 우선적으로 처리하는 방법에 대해 소개한다. [5]

3.2 Suffix Tree를 변형한 방법

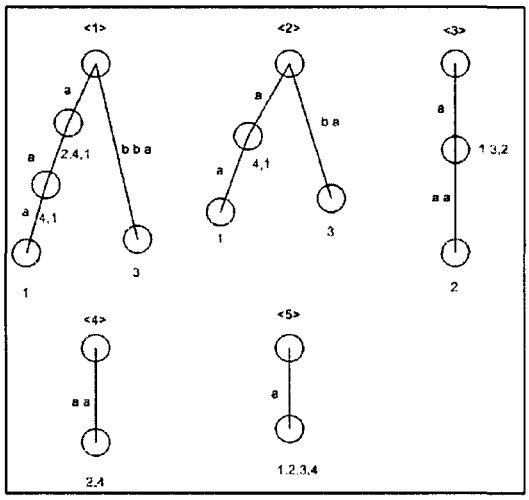
문자열 data로부터 만들어진 substring에 대한 위치정보를 쉽게 확인하기 위해 문자열 data의 각 위치마다 suffix를 가지는 trie를 생성한다.

3.1절에서 사용한 문자열 data의 substring에 대해 각각 suffix를 구하고 이를 각 위치별로 나타내면 다음과 같다. 아래에서 길이가 L인 문자열의 k 번째($k \le L$, $i \le k \le j$, i,j는 각 substring 범위) 위치에서 나타나는 suffix는 S_k로 표현한다.

$S①_1$ = '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'

[그림2]는 이와 같이 각 위치에서의 substring의 suffix를 trie로 구현한 것을 보여준다.



[그림 2] Suffix tree 변형

위 그림에서 괄호 안의 숫자는 문자열 data의 각 위치를 나타낸다. 괄호가 없는 숫자는 저장된 suffix를 가지고 있는 문자열 data에 대한 index number를 나타낸다. Subsumption 관계가 성립하는 substring이 같은 data에 존재하여야 subsume 한다고 할 수 있기 때문에 문자열 data의 index 정보를 각 노드에 유지하여야 한다.

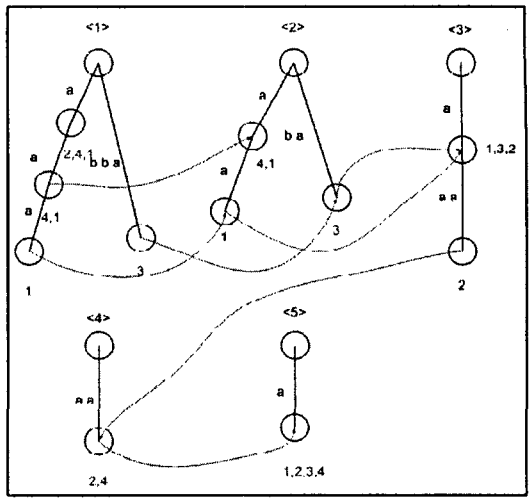
[그림2]에서 확인할 수 있는 것처럼 같은 위치에서 시작하는 suffix들은 같은 trie에 저장되고 symbol value가 같을 경우 edge를 공유하게 된다. 이 경우 trie를 구성하는 것은 linear하지 않게 되었지만 ($O(\sum|S_{i,j}|^2)$), substring을 확인하는 것은 linear하게 되었다. 따라서 이 알고리즘이 적용될 application이 trie의 생성보다 search가 더 많은 경우라면 3.1의 방법보다 효과적인 알고리즘이 된다.

3.3 Performance 향상을 위한 개선방안

첫 번째는 기존의 suffix tree 알고리즘에서 사용하는 suffix link의 적용이다. [4] Substring의 i번째 위치에 해당하는 trie에서 suffix를 생성하고, i+1 번째 trie에서 suffix를 생성할 때, 인접한 trie 사이에 suffix link를 구현하여 이후에 다른 suffix를 생성할 때 edge를 탐색하는 시간을 줄일 수 있다.

두 번째는 subsumption 관계를 파악할 때 탐색했던 공통노드에 대한 정보를 저장하여, subsumption 관계가 성립하지 않아 새로 trie에 저장될 때 동일한 비교작업을 수행하지 않고 바로 trie에 suffix를 생성할 수 있도록 하는 것이다.

[그림3]은 [그림2]에 suffix link를 구현한 것이다.



[그림 3] Suffix link 적용 예

4. 결 론

Wildcard character를 포함한 문자열 data 사이의 subsumption 관계를 파악하는 데 있어서 기존의 suffix tree 구조를 그대로 사용하는 것은 tree의 구성은 linear time에 가능하나 탐색이 linear하지 못하다는 단점이 있다. Subsumption test를 사용하는 application이 한번 구성된 자료구조에 대해 탐색을 많이 하는 경우에는 문자열 data의 각 위치마다 suffix를 저장하는 trie를 생성하는 알고리즘이 더 효율적이다.

이 두 가지 방법은 단백질 서열에 대한 pattern discovery 알고리즘에서 candidate pattern을 저장, 탐색하기 위해 고안되었으나 일반적으로 wildcard character를 포함하는 문자열 사이의 subsumption 관계를 검사하는 경우에 효율적인 알고리즘을 제시한다.

5. 참고 문헌

[1] 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.

[2] Inge Jonassen, Efficient discovery of conserved patterns using a pattern graph, CABIOS, 13, 509-522, 1997.

[3] Andrea Califano, SPLASH: structural pattern localization analysis by sequential histograms, Bioinformatics, Vol. 16 no. 4, 341-357, 2000.

[4] Dan Gusfield, Algorithm on Strings, Trees and Sequences, Cambridge University Press, 1997.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms 2nd ed., MIT Press, 2001.