효율적인 유사문자열 검색을 위한 역리스트 탐색 기법

이은석*, 김종익**
*전북대학교 전자정보공학부
**전북대학교 컴퓨터공학부
e-mail:{les870, jongik}@jbnu.ac.kr

Efficient Approximate String Searches with Inverted Lists through Search Range Reduction

Eunseok Lee*, Jongik Kim**
*Div. of Electronics & Information Engineering, Chonbuk National University
**Div. of Computer Science & Engineering, Chonbuk National University

요 약

유사문자열 검색이란 문자열 집합에서 주어진 문자열과 유사한 문자열들을 검색하는 것으로 정보검색, 데이터 클리닝 등의 분야에서 활용되고 있다. 효율적인 유사문자열 검색을 위해 사전에 문자열 집합에 대한 역리스트를 구성하고 문자열이 주어졌을 때, 주어진 문자열에 관련된 역리스트를 병합하여 유사도 기준을 만족하는 문자열을 찾는다. 이때 비용을 줄이기 위해 일부의 역리스트만 병합하고 나머지 역리스트에 대해서는 이진탐색을 하는 방법이 있다. 본 논문에서는 역리스트를 이진탐색할 때, 불필요한 탐색구간을 제거하여 역리스트 탐색 비용을 줄이는 방법을 제안한다.

1. 서론

텍스트 데이터는 동일한 의미지만 표기방법에 따라 차이가 있을 수도 있고, 오타가 발생 할 수도 있다. 따라서 정확히 일치하는 문자열뿐만 아니라 유사한 문자열도 함께 검색해야할 필요가 있다.

유사문자열 검색에서 유사도 측정 기준은 편집거리, 코사인 유사도, 자카드 유사도 등이 있다. 본 논문에서는 문자열 유사도의 기준으로 많이 사용되는 편집거리를 사용한다. 편집거리는 두 문자열 S1, S2에 대해, S1을 S2로 바꾸는데 필요한 삽입, 삭제, 치환 연산의 최소개수이다.

유사문자열 검색을 위해 주어진 문자열과 문자열 집합 내의 모든 문자열에 대한 편집거리를 구하는 것은 많은 비용이 든다. 그래서 효율적인 처리를 위해 문자열 집합의 문자열들을 특정 길이의 부분문자열로 분해하고, 부분문자열에 대해 해당 부분문자열이 나타나는 모든 문자열의 ID를 리스트로 만들어 검색에 이용한다. 이때 문자열을 고정길이 q로 분해한 부분문자열을 q-그램이라 한다. “search”라는 문자열을 3-그램으로 분해하면 {sea ear arc rch}라는 4개의 부분 문자열을 얻는다. 그리고 그램은 해당 그램이 나타나는 문자열의 ID를 리스트로 가지는데 이를 역리스트라고 한다.

편집거리를 이용하는 유사문자열 검색은 주어진 문자열에서 그램들을 추출하고, 그램들의 역리스트들에서 어떤 문자열 ID가 기준 값 T개 이상 나타난다면, 해당 문자열을 유사문자열로 판단한다. 역리스트들에서 T개 이상 나타난 ID를 찾기 위해서는 역리스트들을 병합해야하지만 전체 역리스트들을 병합하는 것은 많은 비용이 든다. 역리스트 병합비용을 감소하기 위해 길이가 긴 T-1개의 역리스트를 LongList로 그 외의 리스트를 ShortList로 정의한다.[1, 3] 그리고 ShortList만 병합하여 ID의 발생횟수를 세고 부족한 발생횟수는 LongList에서 이진탐색을 하여 ID가 있는 경우 발생횟수를 증가시킨다. 이때 LongList의 수는 T-1개이기 때문에 LongList에만 등장하는 ID의 경우 답이 될 수 없다. 반면 ShortList를 병합한 리스트에 속한 ID는 LongList를 탐색하면 기준 값 T를 만족할 수도 있기 때문에 ShortList를 병합한 리스트를 후보리스트라고 부른다.

역리스트들의 길이 차이가 심하기 때문에 가장 긴 리스트들을 LongList로 분류하여 병합에서 제외하고 ShortList만 병합한다면, 리스트 병합은 많은 비용을 감소시킬 수 있다. 하지만 이 경우 ShortList를 병합한 리스트인 후보리스트의 ID들을 LongList에 속하는 역리스트에서 탐색해야하는 비용이 추가적으로 발생하며 이것은 ShortList를 병합하는 비용과 비슷하다.

본 논문에서는 LongList를 탐색할 때, 탐색범위를 감소시켜 탐색비용을 줄이는 방법을 제안한다. 제안하는 방법은 후보리스트의 모든 ID를 LongList에 속하는 역리스트에 대해 이진 탐색을 할 때, 탐색하는 ID로 후보리스트를 두 개의 리스트로 분리하고, 탐색한 위치로 LongList에 속하는 역리스트를 두 개의 리스트로 분리하여 다음 탐색들의 탐색구간을 줄인다.

본 논문의 구성은 다음과 같다. 2장에서는 역리스트를

이진탐색 할 때 탐색구간을 감소시키는 방법을 설명한다. 3장은 실험을 통한 알고리즘별 성능을 평가한다. 4장은 결론 및 개선 방향을 제시한다.

2. 구간 감소 역리스트 탐색

본 장에서는 유사문자열 검색에서 LongList 탐색비용을 감소시키는 방법을 제안한다. (그림 1)은 유사문자열 검색 처리 과정을 보여준다. 3번째 줄은 ShortList를 병합하여 후보리스트를 만든다. 4번째 줄과 같이, LongList 탐색은 후보리스트를 LongList에 속하는 역리스트들에 대해 순차적으로 탐색하는 과정이므로, 이후 모든 알고리즘은 후보리스트를 하나의 LongList에 속하는 역리스트에 대해 탐색하는 방법만을 설명한다. 그리고 모든 역리스트와 역리스트를 병합한 후보리스트는 오름차순으로 정렬되어 있다.

```
Algorithm Approximate string search
1. long_list = T-1 long lists
2. short_list = remaining lists
3. candidate_list = merge_short_list(short_list);
4. for each long list, L
5.     candidate_list := search(candidate_list, L);
6. return candidate_list;
```

(그림 1) 유사문자열 검색 알고리즘

이 논문은 (그림 1)의 5번째 줄인 search()의 탐색비용을 개선한다. search()함수는 하나의 역리스트에 대해 후보문자열을 탐색하는 함수이다. 전체 LongList탐색은 후보리스트의 모든 ID를 LongList에 속하는 역리스트에서 탐색하여, ID의 발생 횟수가 T개 이상이라면 Result에 ID를 추가하는 과정이다. 후보리스트의 길이를 S, 역리스트의 길이를 L이라고 할 때, 후보리스트의 모든 ID를 역리스트에서 탐색하는 비용은 $Slog_2L$이다.

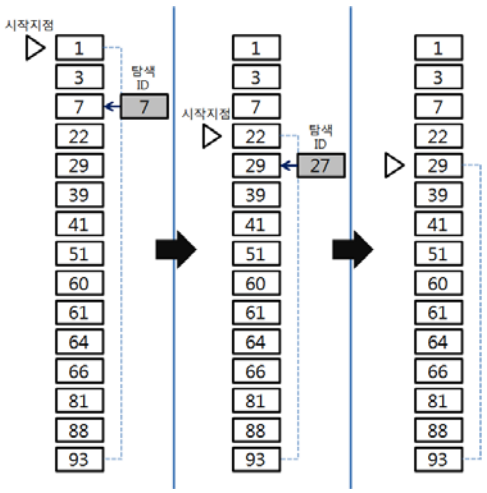
3.1절에서는 이전 탐색위치를 이용하여 탐색구간을 줄이는 방법을 설명하고, 3.2절에서는 후보리스트와 역리스트를 반으로 나눠가며 탐색구간을 줄이는 방법에 대해 설명한다.

3.1 이전 탐색위치를 이용하는 역리스트 탐색

이전 탐색위치를 이용한 역리스트 탐색은 시작지점에 이전 탐색위치를 저장하여 탐색구간 감소에 이용하는 알고리즘이다. 시작지점은 역리스트를 탐색할 때, 탐색구간의 시작을 나타낸다. 시작지점의 초기 값은 역리스트의 시작위치이며, 후보리스트의 ID를 역리스트에서 탐색하면 탐색으로 얻은 위치를 새로운 시작지점으로 갱신한다.

시작지점 이전의 값은 이전의 탐색 ID보다 작은 값들이며 이전의 탐색 ID는 현재의 탐색 ID보다 작다. 따라서 시작지점부터 역리스트의 끝을 탐색구간으로 잡으면, 불필요한 탐색구간은 탐색하지 않게 된다.

(그림 2)는 탐색 과정을 보여준다. 처음 탐색 ID는 7이며 리스트 전체를 탐색 구간으로 가지는 이진탐색을 한다. 이때 리스트 내에서 7이 있으므로 시작포인터는 7 다음 값인 22를 가리킨다. 다음 탐색 ID는 27이며 시작 지점 이전 값들은 탐색 ID보다 작기 때문에 시작포인터가 가리키는 22부터 리스트의 끝까지 이진탐색을 한다. 이때 리스트내에서 27이 없으므로 시작포인터는 27보다 큰 29를 가리킨다.



(그림 2) 이전 탐색위치를 이용한 역리스트 탐색

역리스트 탐색비용은 다음과 같이 예측할 수 있다. 탐색비용을 계산하기 위해 모든 리스트는 ID가 균일하게 분포되었다고 가정한다. 역리스트의 길이가 L, 후보리스트의 길이가 S일 때, 매번 탐색으로 감소하는 탐색구간의 길이는 약 L/S이다. 따라서 후보리스트의 k번째 ID를 이진탐색하는 비용은 $log_2(L-(k-1)*\frac{L}{S})$이다. 전체 후보리스트를 탐색하는 비용은 (수식 1)과 같다. (단 e는 자연로그)

$$\sum_{k=1}^{S} log_2(L-(k-1)*\frac{L}{S})$$
$$\fallingdotseq Slog_2L - Slog_2e + log_2e$$

(수식 1) 이전 탐색위치를 이용한 역리스트 탐색 비용

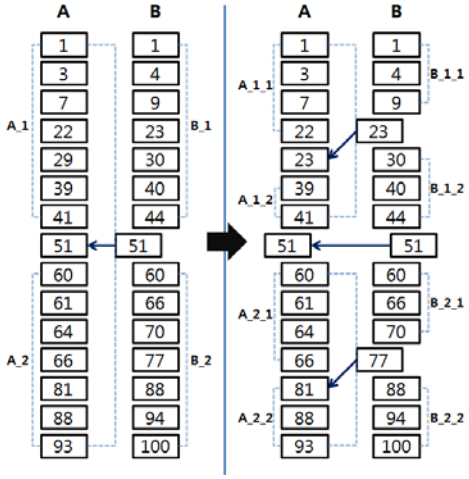
구간감소를 하지 않는 경우, 탐색비용은 $Slog_2L$이었다. 따라서 탐색구간을 줄이는 이 알고리즘의 경우 $Slog_2e - log_2e$만큼의 탐색비용을 감소시킨다.

3.2 리스트 분할을 이용하는 역리스트 탐색

리스트 분할을 이용한 역리스트 탐색은 후보리스트와 역리스트를 절반으로 나누어 탐색구간을 줄이는 알고리즘이다.

후보리스트의 중심에 위치한 ID부터 탐색을 하며, 후보리스트는 중심 ID를 기준으로 두 개의 리스트로 나누어진다. 역리스트는 중심 ID를 역리스트에서 탐색하여 얻은 위치를 기준으로 두 개의 리스트로 나누어진다. 나누어진 리스트들은 중심 ID보다 각각 작은 ID의 리스트와 큰 ID

의 리스트이므로 작은 후보리스트는 작은 역리스트에서 큰 후보리스트는 큰 역리스트에서 탐색을 한다.



(그림 3) 리스트 분할을 이용한 LongList 탐색

(그림 3)은 이 알고리즘의 탐색과정을 보여준다. A는 역리스트, B는 후보리스트를 나타낸다. 처음 탐색 ID는 B의 중심 ID인 51이며 A에서 이진탐색으로 51을 찾는다. 그리고 각각 A에서의 탐색 위치와 B의 탐색 ID를 기준으로 A_1과 A_2, B_1과 B_2로 리스트를 나눈다. 각각이 탐색 ID보다 작은 ID와 큰 ID의 리스트이므로 A_1은 B_1에서 A_2는 B_2에서 탐색한다. 리스트 A_2에 대한 리스트 B_2의 경우, 중심 ID인 77을 탐색하지만 ID가 존재하지 않고 81을 가리킨다. 이때는 큰 리스트인 탐색으로 가리키는 ID인 81을 포함해서 A_2_2를 구성한다.

Algorithm SearchList

```
input: LongList : LL
       LongList begin, end index : Lb, Le
       CandidateList : CL
       CandidateList begin, end index : Cb, Ce
1:  if (Lb > Le || Cb > Ce)     return;
2:  Cm = (Cb+Ce) / 2;
3:  binarySearch(LL, Lb, Le, CL[Cm]);
4:  value = searchResultValue;
5:  Lm = searchResultIndex;
6:  if (CL[Cm]  == value){
7:     CL[Cm]'s count++;
8:     SearchList(LL, Lb, Lm-1, CL, Cb, Cm-1);
9:     SearchList(LL, Lm+1, Le, CL, Cm+1, Ce);
10: }
11: else {
12:    SearchList(LL, Lb, Lm-1, CL, Cb, Cm-1);
13:    SearchList(LL, Lm, Le, CL, Cm+1, Ce);
14: }
15: return ;
```

(그림 4) 리스트 분할을 이용한 역리스트 탐색 알고리즘

(그림 4)는 LongList를 탐색하는 과정을 보여준다. 3번째 줄에서는 후보리스트의 중심 ID을 역리스트에서 이진탐색한다. 6번째 줄에서는 탐색한 ID가 있었는지를 확인하며, 각각의 경우에 따라 카운트 증가와 부분 문제를 실행 시킨다.

이 알고리즘의 역리스트 탐색비용을 예측해본다. 3.1절과 마찬가지로 모든 리스트 내의 ID는 균일분포를 가정한다. 이 알고리즘은 하나의 리스트에서 나누어진 두 개의 리스트는 동일한 길이의 탐색구간을 가지므로 탐색비용이 동일한 ID들을 그룹으로 묶고 이를 탐색레벨이 같다고 표현한다.

첫 번째 탐색의 탐색레벨을 1, 탐색레벨이 k인 탐색으로부터 호출된 탐색의 탐색레벨을 K+1로 정의한다.

역리스트의 길이를 L, 후보리스트의 길이를 S라고 할 때, 탐색레벨이 k인 탐색의 수는 2^{k-1}개이며, 탐색구간의 길이는 $\frac{1}{2^{k-1}} * L$이다. 최대 탐색레벨은 $log_2(S+1)$이다. 하나의 역리스트에 대한 전체 후보리스트의 탐색비용은 (수식 2)와 같다.

$$\sum_{k=1}^{log_2(S+1)} 2^{k-1} * log_2((\frac{1}{2})^{k-1} * L)$$
$$= Slog_2L + 2S - Slog_2(S+1) - log_2(S+1)$$

(수식 2) 리스트 분할을 이용한 역리스트 탐색

구간감소를 적용하지 않은 경우에 비해, $Slog_2(S+1) + log_2(S+1) - 2S$만큼의 비용이 감소되는 것을 알 수 있고, 탐색 비용의 차이는 3.1과 마찬가지로 후보리스트가 증가할수록 커진다.

3. 실험 및 성능평가

본 장에서는 유사문자열 검색에서 LongList 탐색에 대해 구간감소를 하지 않은 경우, 이전 탐색 위치 포인터를 이용한 경우(3.1), 리스트를 분할 한 경우(3.2)의 성능을 수행 시간을 측정하여 비교한다.

실험환경은 Intel-Core 2GHz, 2GB RAM, Ubuntu 10.04 LTS 32bit이다. 3-gram을 사용하는 역리스트를 구성하였고, 1000개의 질의를 처리하는 시간을 측정하였다.

성능측정을 위해 2개의 데이터 셋에 대하여 2부터 5까지 편집거리를 적용하여 실행하였다. 실험에 사용된 데이터는 (표 1)과 같다.

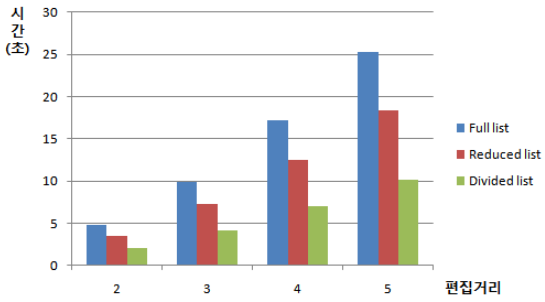
<표 1> 실험 데이터 셋

데이터 셋	문자열 수	평균길이
DBLP Title	1158649	68
Google Web corpus	3000000	21

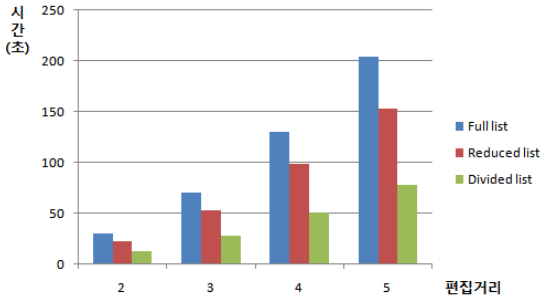
(그림 5)와 (그림 6)은 각각 DBLP Title와 Google Web

corpus 데이터셋에 대해 편집거리를 2~5로 변화시키며 LongList 탐색의 수행시간을 측정한 그래프이다. Full list는 탐색구간을 감소시키지 않은 것이고, reduced list는 3.1의 알고리즘을 적용하였고, Divided list는 3.2의 알고리즘을 적용하였다. 편집거리가 증가하는 경우, 후보리스트의 길이가 증가하기 때문에 수행시간이 증가하는 것을 그래프를 통해 알 수 있다.

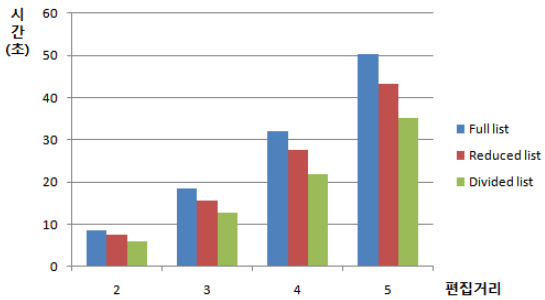
각 알고리즘별 수행속도는 3.2 알고리즘을 적용한 Divide list가 가장 좋은 성능을 나타냈고 전체 리스트를 탐색한 Full list에 비해 110~160% 정도 더 빠른 속도를 나타낸다.



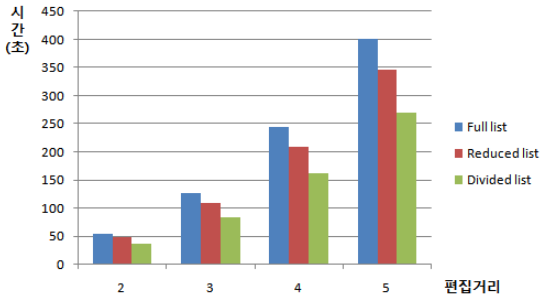
(그림 5) DBLP Title - LongList 탐색 비용



(그림 6) Google Web corpus - LongList 탐색 비용



(그림 7) DBLP Title - 전체 유사문자열 탐색 비용



(그림 8) Google Web corpus - 전체 유사문자열 탐색 비용

(그림 7)과 (그림 8)은 DBLP Title와 Google Web corpus 데이터셋에 대한 전체 유사문자열 검색의 수행시간을 측정한 그래프이다. (그림 5)와 (그림 6)에 비해 ShortList 병합 비용과 기타 비용이 추가되었다. 마찬가지로 3.2 알고리즘을 적용한 Divide list가 가장 좋은 성능을 나타냈고 전체 리스트를 탐색한 Full list에 비해 약 20% 정도의 더 빠른 수행속도를 나타낸다.

4. 결론

본 논문에서는 역리스트 탐색구간을 감소시켜 LongList 탐색비용을 줄이는 기법을 제안하였다. 제안된 기법은 후보리스트의 중간 ID부터 탐색해서 탐색구간을 반으로 줄여나가 탐색비용을 감소시킨다. 구간감소 처리를 하지 않은 경우와 리스트 분할 방법의 성능을 비교했을 때, LongList 탐색 부분은 2.13~2.62배의 수행속도 차이를 나타내었고 전체 유사문자열 검색에서는 1.19~1.28배의 수행속도 차이를 나타내었다.

향후에는 다양한 길이로 리스트를 분할하여 탐색 구간을 줄이는 방법에 대해 연구할 예정이다.

참고문헌

[1] Sunita Sarawagi, Alok Kirpal - "Efficient set joins on similarity predicates"

[2] Roberto J. Bayardo, Yiming Ma, Ramakrishnan Srikant - "Scaling Up All Pairs Similarity Search"

[3] Chen Li, Jiaheng Lu, Yiming Lu "Efficient Merging and Filtering Algorithms for Approximate String Searches"

[4] Chen Li, Bin Wang, Xiaochun Yang - "VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable Length Grams"

[5] Xiaochun Yang, Bin Wang, Chen Li - "Cost-Based Variable-Length-Gram Selection for String Collections to Support Approximate Queries Efficiently"

[6] Chuan Xiao, Wei Wang, Xuemin Lin - Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints