효율적인 집합-기반 POI 데이터 검색 알고리즘
(An Efficient Set-based POI Data Search Algorithm)

고 은 비 † 이 종 우 †† 이 재 원 †††
(EunBi Go) (Jong woo Lee) (Jae Won Lee)

요 약 위치 기반 서비스는 특정 위치의 지리 정보를 파악하기 위해 POI 데이터베이스를 사용한다. 모바일 환경에서의 위치 기반 서비스의 경우에는 검색 성능 향상을 위해 사용자의 유동성에서 기인하는 부정확한 POI 쿼리를 개선해야 한다. 이를 위해 기존의 시스템에서는 쿼리 자동 확장 기술과 하드매칭 기법을 사용하지만, 이는 외부 자원을 필요로 하고 성능이 떨어진다는 단점을 지닌다. 그러므로, 본 논문에서는 시스템 자체 내에서 부정확한 POI 쿼리로 인한 성능 저하 문제를 해결할 수 있는 새로운 POI 데이터 검색 알고리즘을 제시한다. 본 알고리즘은 n개의 문자로 이루어진 POI 쿼리를 m개의 블록으로 균등 분할한 뒤, 각 블록에 대해 집합 기반 연산을 적용하고, '차수'라는 개념을 사용하여 블록 간 집합 연산을 수행한다. 실험을 통해 본 논문에서 제시한 집합 기반의 POI 검색알고리즘의 성능이 기존 기법에 비해 83%~96% 우수함을 확인하였다.

키워드: 관심지역정보, POI 데이터베이스, POI 검색 알고리즘, 집합-기반 알고리즘

Abstract Location-based services (LBS) deploy the Point of Interest (POI) databases to locate the positions users want. LBS should support a query correctness enhancing technique because the POI queries are likely to be incorrect due to the user's mobility. The existing systems commonly use an automatic query expansion technique and a hard matching technique, but these techniques have weaknesses such as necessity of outer resources and low performance, respectively. Thus, in this paper, we propose a new POI data search algorithm working well even when the POI queries are incorrect. The algorithm regards a POI query as a sequence of n characters, and divides it into m chunks. The algorithm applies the set-based operation for each chunk and between chunks. We can find by the performance evaluation using real POI database that our algorithm improve the POI search performance for 83%~96%.

Keywords: Point of Interest, POI database, POI search algorithm, Set-based algorithm

1. 서 론

위치 기반 서비스(Location-based services)는 특정 지리적 위치를 파악한 뒤, 이를 바탕으로 사용자에게 관련 정보를 제공하는 플랫폼이다. 일반적인 위치 기반 서비스는 파악해야 할 특정 위치에 대한 위도, 경도 등의 지리 정보를 얻기 위해 POI (Point of Interest) 데이터베이스 또는 GPS (Geographical Positioning System)을 사용한다.

위치 기반 서비스를 제공하는 모바일 기기 중 스마트폰과 차량 내비게이션 시스템이 일반적이다. 스마트폰은 위치 기반 서비스를 지도/네비게이션 어플리케이션의 형태로 제공하는데, 그 중 구글 지도 모바일 서비스[1)]가 대중적이다. 차량 네비게이션 시스템은 GPS로 현 위치를 파악하고, 파악한 위치와 사용자의 POI 간의 최단 거리를 계산한다[1].

POI는 점으로 표현될 수 있는 모든 지리적 객체[2]이

† 학생회원 : 한국과학기술원 전산학과
eunbi_go@kaist.ac.kr
†† 종신회원 : 숙명여자대학교 멀티미디어과학과 교수
bigrain@sookmyung.ac.kr
††† 정 회 원 : 성신여자대학교 컴퓨터정보학부 교수
jwlee@sungshin.ac.kr
(Corresponding author임)
논문접수 : 2012년 12월 17일
심사완료 : 2013년 2월 12일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제19권 제5호(2013.5)

1) Google Maps Mobile. http://www.google.com/mobile/maps/

며, 사용자가 정보를 얻고자 하는 관심 지점을 의미한다. POI는 POI 데이터베이스에 인구 통계 자료와 함께 위도, 경도, 국가 등의 지리적 정보의 형태로 저장된다[3]. 이는 사용자에게 직접 입력 받은 쿼리로 POI 데이터베이스를 검색하여 지리적 위치 정보를 추출하는 지리 정보 시스템(Geographical Information System)에 기반한다.

GPS는 인공위성으로 데이터 신호를 송수신하여 기기의 현 위치를 파악하는 방법으로, 애플의 3G 아이폰이 출시된 2008년[4]부터 급격히 활성화되었다. 애플 사(社)에서 본사의 모바일 기기에 GPS 센서를 내장한 이후 통합 포지셔닝 센서를 내장한 다양한 범용 모바일 기기가 출시되었기 때문이다.

모바일 환경에서 위치 기반 서비스의 성능을 향상시키기 위해서는 사용자의 유동성을 고려하여 POI를 빠르고 정확하게 검색해야 한다. GPS를 이용한 POI 검색 서비스는 내장된 센서를 사용하므로 사용자의 유동성보다 기기의 성능이 서비스의 성능에 많은 영향을 미친다. 그러나 POI 데이터베이스를 이용한 POI 검색 서비스는 사용자의 이동성에 직접적인 영향을 받는다. 유동적 환경은 사용자가 POI 쿼리를 부정확하게 작성할 확률을 증가시켜 데이터 접근성을 떨어뜨리기 때문이다. 그러므로 사용자의 유동적 환경은 GPS보다 POI 데이터베이스를 이용한 지리 정보 검색에 고려되어야 하며, POI 데이터베이스를 이용한 위치 기반 서비스의 경우 성능 향상을 위해 부정확한 POI 쿼리를 개선해야 한다.

부정확한 POI 쿼리 입력에서 기인한 서비스 성능 저하를 개선하기 위해 쿼리 자동 확장 기술이 대부분의 시스템에 적용된다. 쿼리 자동 확장 기술에는 전역적 쿼리 확장(Global Query Expansion)과 지역적 쿼리 확장(Local Query Expansion)[5], 그리고 적합성 피드백(Relevance Feedback)[6]과 같은 방법이 존재한다. 대부분의 쿼리 확장 기술은 계산이 복잡하고 방대한 양의 데이터를 필요로 하기 때문에 차량 내비게이션과 같은 독립형 시스템에는 부적합하다. 그러므로, 대부분의 독립형 시스템에서는 하드매칭(hard matching) 기법이 적용되는데, 이는 POI 쿼리와 POI 데이터베이스 간의 유사도를 고려하지 않은 단순 검색 방법이므로 성능이 매우 떨어진다.

본 논문에서는 부정확한 POI 쿼리 입력으로 인한 POI 검색 서비스 성능 저하 문제를 시스템 자체 내에서 해결하기 위한 새로운 알고리즘을 제시한다. 이는 외부 자원을 사용하지 않고 사용자의 부정확한 POI 쿼리를 알고리즘 차원에서 재구성 하는 방법을 사용하여 POI 검색 서비스 성능을 향상시킨다.

본 알고리즘의 개략적인 동작 과정은 다음과 같다. POI 데이터베이스를 메인 메모리에 로딩한 뒤, 레드-블랙 트리를 이용하여 글자 별 아이디를 생성한다. 사용자의 POI 쿼리가 n개의 문자로 이루어져 있다면, 이를 m개의 블록으로 균등분할 하여 집합의 연산을 적용한다. POI 쿼리의 각 블록에 대해 연산을 수행한 뒤, 블록 간 연산을 수행한다. 각 블록의 글자가 모두 포함된 POI 데이터베이스의 레코드 개수와 번호를 파악하여 블록 내 연산을 수행하고, '차수'라는 개념을 사용하여 블록 내 연산을 통해 얻은 검색 후보에 대해 다른 블록의 정보를 활용하여 추가적인 제약을 가함으로써 검색 후보의 수를 축소시키는 블록 간 연산을 한다.

본 논문은 POI 검색 시스템의 쿼리 처리에서 다음의 공헌을 제시한다:

- 사용자의 POI 쿼리를 m개의 블록으로 나누어 집합의 연산을 적용한다. 집합의 교환법칙에 따라 쿼리 블록의 모든 조합을 통해 검색 후보의 수를 축소한다.
- n개의 문자로 이루어져 있는 POI 쿼리를 m개의 블록($n \geq m$)으로 나눔으로써, 기존 시스템보다 향상된 성능으로 검색 할 수 있도록 한다.

본 논문의 구성은 다음과 같다. 2장에서는 서비스 성능 저하를 개선하기 위해 기존의 POI 검색 시스템에서 사용하는 쿼리 자동 확장 기술을 설명하고, 3장에서는 본 논문에서 제안하는 알고리즘과 이에 대한 실제 예를 기술한다. 4장에서는 제안한 알고리즘의 실효성을 실험으로 입증한 뒤, 5장에서는 결론을 기술한다.

2. 관련 연구

본 장에서는 부정확한 POI 쿼리 입력으로 인한 성능 저하를 개선하기 위해 기존의 시스템에서 사용하는 쿼리 자동 확장 기술을 설명한다. 쿼리 자동 확장은 정보 검색 분야에서 쿼리와 검색 결과의 불일치를 해결하기 위해 연구된 기술이다. 이 기존 기술은 사용자의 쿼리 내 단어와 유사한 단어를 자동으로 찾아내어 이를 쿼리에 대치, 추가 시키는 방법을 사용하며, 사용하는 데이터의 양에 따라 전역적 쿼리 확장 기술과 지역적 쿼리 확장 기술로 구분된다[5].

2.1 전역적 쿼리 확장 기술

전역적 쿼리 확장 기술은 단어 간 관계를 찾기 위해 통계적 연산을 실시한다. 통계적 연산은 가능한 두 단어의 조합에 대한 정보를 계산하는 것으로, 모든 데이터를 대상으로 연산을 실시한다. 전역적 쿼리 확장 기술은 데이터 전체를 이용하여 연산하므로 연산 결과에 대한 신뢰도가 높지만, 데이터를 축적하고 연산하는 데에 시간이 오래 걸려 효율성이 떨어진다. 전역적 쿼리 확장 기술에는 용어 클러스터링 기법(term clustering)[7], 차원 축소 기법(dimensionality reduction)[8] 등이 있다.

용어 클러스터링 기법은 문서에 함께 등장하는 단어들에 기반하여 단어를 클러스터로 묶은 뒤, 생성된 클러스터를 이용하여 사용자의 쿼리를 확장한다. 쿼리 내 단어와 같은 클러스터에 속한 단어들로 해당 단어를 대치하여 다양한 검색 결과를 얻을 수 있다. 그러나 본 기법은 동음이의어를 처리할 수 없다는 단점을 지닌다. 만약 쿼리 내 단어가 여러 뜻을 지니고 있다면, 해당 단어가 동시에 여러 클러스터에 속하게 되면서 쿼리의 의미가 더욱 모호해진다.

차원 축소 기법은 용어 클러스터링 기법과 유사한데, 잠재 의미 색인(latent semantic indexing) 방법[8]이 잘 알려져 있다. 잠재 의미 색인 방법은 사용자의 쿼리를 벡터 공간에서의 벡터로 간주하며, 쿼리 벡터에 특이값 분해(singular value decomposition)[9]를 적용하여 쿼리 벡터를 실제 차원보다 낮은 차원으로 분해한다. 높은 차원에서 직각으로 표현되는 연관 용어들, 즉 유사성이 존재하지 않는 용어들은 낮은 차원에서도 이와 비슷하게 표현될 것이라고 가정하는 것이다. 하지만, 잠재 의미 색인 방법은 벡터 공간에서의 기존 연산과 비교하여 성능 개선이 증명되지 않았다.

2.2 지역적 쿼리 확장 기술

지역적 쿼리 확장 기술은 전역적 쿼리 확장 기술보다 적은 양의 데이터로 연산을 수행한다. 지역적 쿼리 확장 기술은 결과로서 출력된 k개의 상위 데이터를 이용한다. 지역적 피드백(Local Feedback) 기법은 최근에 사용되는 지역적 쿼리 확장 기술 중 하나이다.

지역적 피드백 기법은 정보 검색 분야에서 널리 이용되는 적합성 피드백(Relevance Feedback)[6] 기법에서 기인한다. 적합성 피드백은 추출된 데이터에서 측정된 연관성을 기반으로 쿼리의 단어를 수정하는 기법이다. 지역적 피드백 기법은 k개의 상위 데이터만을 적합성 피드백에서의 추출된 데이터로 간주하여 연관성을 측정한다. 그러므로, 지역적 피드백 기법은 상위 데이터에 자주 등장하는 용어들을 사용자의 쿼리에 추가, 수정함으로써 쿼리를 확장한다.

지역적 쿼리 확장 기술은 관련성이 높은 소수의 데이터를 집약적으로 사용하기 때문에 전역적 쿼리 확장 기술보다 효과적인 방법으로 간주된다. 그러나, 지역적 쿼리 확장 기술은 추출된 소수의 데이터를 대상으로 하기 때문에 추출된 데이터에 대한 의존도가 높다. 만약 사용자가 입력한 쿼리와 추출된 상위 데이터의 연관성이 높지 않으면, 쿼리 확장 기술을 사용하는 것은 무의미하며 쿼리의 의미를 더욱 모호하게 만들 수 있다.

3. 집합-기반 알고리즘

본 장에서는 본 논문에서 제안하는 집합-기반 알고리즘을 순차적으로 설명한다. 먼저, POI 데이터베이스의 구조와 이를 로딩하는 과정, 로딩된 POI 정보를 이용하여 역 인덱스 파일을 생성하는 과정을 설명한다. 그 후에는 POI 쿼리의 각 블록과 블록 간 실행하는 AND 연산의 알고리즘을 기술한다.

No.	POI Data
0	힐하우스@전라북도_군산시
1	힐튼아파트@인천광역시_부평구_부평동
2	희망아파트C동@경상북도_구미시_인의동
3	힐탑트레져아파트@서울특별시_용산구_한남동
4	흰돌마을단지주공아파트@경기도_고양시

그림 1 POI 데이터베이스의 예

본 알고리즘은 POI 데이터베이스에 POI 위치에 대한 주소가 저장되어 있다고 가정한다. POI 데이터베이스는 각 POI의 이름과 주소로 이루어진 레코드의 집합이며, 이름과 주소는 텍스트로 이루어져 있다. 각 레코드는 POI의 이름과 주소를 특수문자인 '@'로 구분하며, 이를 '구분자'로 정의한다. 또한, 구분자의 앞 텍스트를 POI의 '명칭 부분', 뒤 텍스트를 '주소 부분'으로 정의한다. 그림 1은 POI 데이터베이스의 예를 보이고 있는데, 첫 번째 레코드의 명칭 부분은 '힐하우스', 주소 부분은 '전라북도 군산시'이다.

3.1 POI 데이터베이스 로딩 알고리즘

POI 데이터베이스 로딩 단계에서는 POI 데이터베이스 내의 위치 정보와 이후 연산에 필요한 부가적인 정보를 메인 메모리에 로딩한다. 그림 2는 세부적인 POI 데이터베이스 로딩 알고리즘을 나타낸다. 본 알고리즘은 POI 데이터베이스를 입력 받아 각 레코드의 문자를 순차적으로 읽어 명칭 부분 길이와 주소 부분 길이를 알아내면서 글자를 해당 메모리에 저장한다.

명칭 부분의 길이와 주소 부분의 길이는 서로 다른 두 개의 배열에 각각 바이트 단위로 저장되며 레코드 번호가 두 배열의 인덱스로 쓰인다. 한글의 크기는 2바이트 이므로, 읽어 들인 문자가 한글이면 2 바이트를 더하고 해당 문자 자체를 메모리에 로딩한다. 또한, 레코드를 구성하는 문자가 구분자이거나 레코드의 끝이면 1 바이트를 더하고 해당 문자 대신 NULL을 메모리에 로딩한다.

그림 3은 그림 1의 샘플에 대해 POI 데이터베이스 로딩 알고리즘을 적용한 결과이다. 첫 번째 레코드의 명칭 부분인 '힐하우스'의 경우, 모든 한글은 2 바이트로 표현되므로 4*2의 크기를 갖는다. 해당 레코드 내 명칭 부분의 끝을 표시하기 위한 NULL 값 또한 명칭 부분에 포함되므로 1을 더한다. 주소 부분의 경우에는 한글의 개수가 7개이지만, 띄어쓰기와 마지막의 NULL 값을 고려하여 추가적으로 계산해야 한다. 최종적인 메모리는 그림 3의 마지막 배열과 같이 로딩된다.

```
LoadDBinMemory (POI database D)
 1:  name_len ← array for the length of the name part
 2:  addr_len ← array for the length of the address part
 3:
 4:  Initialize the array name_len to zero
 5:  Initialize the array addr_len to zero
 6:  for each record R in the POI database D do
 7:      index ← record number of R
 8:      for each character c in the record R do
 9:          if c is Hangeul then
10:              if c is in the name part then
11:                  name_len[index] ← name_len[index] + 2
12:              else if c is in the address part then
13:                  addr_len[index] ← addr_len[index] + 2
14:              end if
15:              load c in the memory
16:          else if c is @ then
17:              if c is in the name part then
18:                  name_len[index] ← name_len[index] + 1
19:                  load NULL in the memory
20:              else if c is in the address part then
21:                  addr_len[index] ← addr_len[index] + 1
22:                  load c in the memory
23:              end if
24:          else if c is EOL then
25:              if c is in the name part then
26:                  name_len[index] ← name_len[index] + 1
27:              else if c is in the address part then
28:                  addr_len[index] ← addr_len[index] + 1
29:              end if
30:              load NULL in the memory
31:          end if
32:      end for
33:  end for
34:  return name_len, addr_len, memory
```

그림 2 POI 데이터베이스 로딩 알고리즘

name_len	9	11	14	17	23						
addr_len	16	25	23	25	14						
memory	D79	0	D55	8	C6B	0	C2A	4	NULL	D23	6	...
	힐		하		우		스			전		

그림 3 샘플 레코드의 POI 데이터베이스 로딩 결과

3.2 글자 아이디 생성 알고리즘

역 인덱스 파일을 생성하기 위한 선행 작업으로 모든 레코드의 명칭 부분에 등장하는 모든 글자에 아이디를 부여한다. 이를 위해 두 개의 레드-블랙 트리가 사용되는데, 레드-블랙 트리는 모든 노드가 레드나 블랙의 색상 속성을 갖고 있는 이진 트리로서[10] 자료의 삽입과 삭제, 검색에서 일정한 실행 시간을 보장한다[11].

POI 데이터베이스 레코드의 명칭 부분에 존재하는 글자에 아이디를 부여하는 알고리즘은 그림 4와 같다. POI 데이터베이스의 각 레코드를 순차적으로 읽어 데이터베이스에 등장하는 모든 글자를 정수코드로 변환하여 첫 번째 레드-블랙 트리에 저장한다. 그 후, 모든 정수코드를 순차적으로 검색하여 해당 정수코드가 첫 번째

```
CreateIDSearchTree (loaded POI data M)
 1:  rb_tree1 ← red-black tree for characters in the memory
 2:  rb_tree2 ← red-black tree for IDs of each characters
 3:                 in the memory
 4:  for each name part N in the memory M do
 5:      for each character c in the name part N do
 6:          convert c to integer code c'
 7:          rb_tree1.Insert (c')
 8:      end for
 9:  end for
10:
11:  id ← 0
12:  for each integer code icode do
13:      r ← rb_tree1.Search(icode)
14:      if r is not NULL then
15:          rb_tree2.Insert(icode, id)
16:          id ← id + 1
17:      end if
18:  end for
19:  return rb_tree2
```

그림 4 글자 아이디 생성 알고리즘

공	단	돌	동	레	마	망	스	아	우	을
0	1	2	3	4	5	6	7	8	9	10

저	주	지	탑	트	튼	파	하	희	흰	힐
11	12	13	14	15	16	17	18	19	20	21

그림 5 POI 데이터베이스 샘플의 글자 아이디 생성 결과

레드-블랙 트리에 존재하는 경우에만 해당 정수코드와 검색 순서를 두 번째 레드-블랙 트리에 저장한다. 본 논문에서는 두 번째 레드-블랙 트리를 '아이디 검색 트리'라 명한다. 즉, 레코드의 명칭 부분에 등장하는 모든 글자는 정수코드의 형태로 저장되며, 검색된 순서가 각 문자의 아이디가 된다.

그림 5는 그림 1 샘플의 각 글자에 대해 글자 아이디를 생성한 결과이다. 아이디 검색 트리를 완성하기 위해 0부터 시작하여 최대 정수 코드까지 순차적으로 검색하므로, 샘플 레코드 집합의 명칭 부분에 포함되는 모든 글자를 가나다순으로 아이디를 부여한다.

3.3 역 인덱스 생성 알고리즘

POI 데이터베이스의 명칭 부분에 등장하는 모든 글자에 대해 텍스트 검색을 위한 역 인덱스를 생성한다. 역 인덱스에는 각 글자가 등장하는 POI 데이터베이스의 레코드 번호(줄 번호)가 글자 별로 저장된다. 레코드 번호는 3바이트로 표현되며, 이는 시스템이 레코드 수가 2^{24}개 이내인 POI 데이터베이스를 수용할 수 있음을 의미한다.

그림 6은 역 인덱스 생성을 위한 전처리 과정과 역 인덱스를 생성하는 과정을 보이고 있다. 역 인덱스 생성의 전처리 작업으로 각 글자가 POI 데이터베이스의 명칭 부분에 등장하는 횟수와 역 인덱스에 저장할 위치를 파악해야 한다. 이를 위해 두 개의 배열을 사용하며, 각 배열의 인덱스는 3.2의 글자 아이디 생성 알고리즘을 통

```
CreateInvertedIndex (loaded POI data M, rb_tree2)
 1:  inverted ← array for storing record number for each character
 2:  i_count  ← array for the number of appearance for each character
 3:  i_start  ← array for the starting location for each character
 4:                in the inverted index
 5:  Initialize the array i_count to zero
 6:  for each name part N in the memory M do
 7:      for each character c in the name part N do
 8:          convert c to integer code c'
 9:          r ← rb_tree2.Search (c')
10:          if r is not NULL then
11:              i_count [r] ← i_count [r] + 1
12:          end if
13:      end for
14:      for each character c in the name part N do
15:          convert c to integer code c'
16:          r ← rb_tree2.Search(c')
17:          if r is not NULL then
18:              i_start [r] ← i_start [r – 1] + i_count [r – 1] * 3
19:          end if
20:      end for
21:  end for
22:
23:  for each line L in the memory M do
24:      for each character c in the line L do
25:          convert c to integer code c'
26:          r ← rb_tree2.Search(c')
27:          if r is not NULL then
28:              compute the position to store in the array inverted
29:              store the L's number in the computed position
30:          end if
31:      end for
32:  end for
33:  return i_count, inverted
```

그림 6 역 인덱스 생성 알고리즘

해 생성된 글자의 아이디이다.

POI 데이터베이스 내 명칭 부분의 각 글자가 등장하는 횟수를 저장하기 위해 메모리에 로딩된 POI 데이터의 명칭 부분을 한 글자씩 순차적으로 읽는다. 명칭 부분 내 모든 글자의 등장 횟수를 글자 별로 계산한 뒤, 이를 통해 각 글자의 역 인덱스 내 저장위치를 계산한다. 아이디 검색 트리에 정수코드를 순차적으로 입력하여 POI 데이터베이스에 존재하는지 여부를 파악한다. POI 데이터베이스에 존재하는 글자에 한해 역 인덱스 내 저장 위치를 파악하는데, 저장 위치는 이전 아이디의 글자의 저장 위치와 등장 횟수를 통해 알 수 있다.

전처리 작업 후에는 역 인덱스를 생성한다. 메모리에 로딩된 POI 데이터의 명칭 부분을 한 글자씩 순차적으로 읽어 아이디 검색 트리에 넣는다. 아이디 검색 트리를 통해 파악한 글자의 아이디를 사용하여 역 인덱스 내에서의 현재 글자에 대한 역 인덱스의 저장 시작 위치를 알아낸다. 이렇게 파악한 저장 시작 위치에 3*(해당 글자에 대한 현재까지의 역 인덱스 개수)를 더하여 최종 저장 위치를 계산하고, 해당 레코드 번호를 3바이트로 변환하여 저장한다.

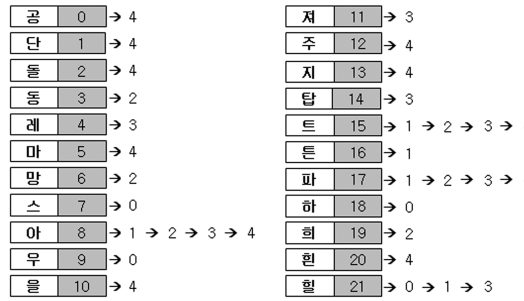


그림 7 샘플 레코드 집합의 글자 아이디 생성 결과

샘플 레코드에 대해 역 인덱스를 생성하면 그림 7과 같다. 역 인덱스에는 아이디가 부여된 각 글자가 포함되어 있는 POI 데이터베이스 내 레코드 번호를 저장한다. 메모리의 명칭 부분의 글자를 순차적으로 읽으므로, 각 글자의 레코드 번호는 오름차순으로 저장된다. 해당 글자가 레코드에 포함된 횟수가 아닌 포함 여부만을 확인하며, 이는 샘플 레코드 집합 중 4번째 레코드의 '힐탑트레져아파트'에서 확인할 수 있다.

3.4 텍스트 검색 알고리즘

사용자의 POI 쿼리는 POI 데이터베이스와 마찬가지로 텍스트로 이루어져 있으며, POI 쿼리는 세 글자 이상으로 이루어져 있다고 가정한다. 사용자의 쿼리를 m 개의 블록으로 나누어 집합의 연산을 적용하는 것이 본 알고리즘의 핵심이다. 각각의 독립된 블록에 대해 집합 연산을 실시한 뒤, 블록 간 연산을 통해 검색 후보를 축소시켜 나간다. 이러한 집합 연산을 실시하기 전에 POI 쿼리에 대한 전처리 작업을 실시하고, 이를 통해 사용자의 POI 쿼리에 대한 기본적인 정보를 파악한다.

POI 쿼리가 입력되었을 때, 쿼리의 글자를 순차적으로 읽어 쿼리의 글자 개수를 계산하고, 각 글자를 바이트 단위로 변환하여 저장한다. 또한, 아이디 검색 트리를 이용하여 각 글자에 대한 인덱스를 파악하고, 역 인덱스와 관련 정보를 통해 POI 쿼리의 각 글자를 포함하는 POI 데이터베이스 내 레코드를 저장한다.

3.4.1 전처리 작업

그림 8은 사용자가 POI 쿼리를 입력했을 때의 전처리 작업에 대한 알고리즘을 나타낸다. 사용자의 쿼리가 입력되면 시스템은 POI 쿼리의 각 글자를 순차적으로 읽어 바이트 단위로 변환한 뒤, 이를 (쿼리의 글자 개수)*3의 크기를 갖는 2차원 배열에 저장한다. 또한, 순차적으로 읽어 POI 쿼리의 각 글자를 아이디 검색 트리에 입력하여 해당 글자에 대한 아이디를 얻어낸 뒤, 이를 쿼리 아이디 배열에 저장한다. 쿼리 아이디 배열의 크기는 쿼리의 글자 개수와 같으며, 각 글자의 아이디가

```
PrepareTextSearch (POI query Q, rb_tree2, inverted)
1:  eo_string ← array for each character in the POI query
2:  eo_index ← array for IDs for each character in the POI query
3:  searched_count ← array for the number of appearance for
4:                   each character in the POI query
5:  searched ←array for storing record number for each character
6:               in the POI query
7:  index ← 0
8:  for each character c in the POI query do
9:      eo_count ← eo_count + 1
10:     eo_string[index] ← c
11:     convert c to integer code c'
12:     r ← rb_tree2. Search(c')
13:     if r is not NULL then
14:         eo_index[index] ← r
15:     end if
16:     index ← index + 1
17: end for
18: eo_count ← index
19: index ← 0
20: Initialize the array searched_count to zero
21: for each character c in the POI query do
22:     searched[index][searched_count[index]] ← inverted[index]
23:     searched_count[index] ← searched_count[index] + 1
24:     index ← index + 1
25: end for
26: return searched_count, searched
```

그림 8 텍스트 검색의 전처리 알고리즘

저장되어 있다.

POI 쿼리의 각 글자에 대한 정보를 해당 변수에 입력하였으므로, 각 글자의 POI 데이터베이스에 대한 정보를 변수에 입력해야 한다. 이를 위해 쿼리의 각 글자가 POI 데이터베이스에 등장한 횟수와 레코드 번호를 이용한다. POI 쿼리의 각 글자가 POI 데이터베이스에 등장하는 횟수를 저장해야 하므로, POI 쿼리의 글자 수만큼의 크기를 갖는 1차원 배열을 사용한다. 또한, (POI 쿼리의 글자 개수)*(최대 등장 횟수)의 크기를 갖는 2차원 배열은 POI 데이터베이스 내에서 각 글자가 등장하는 레코드 번호가 해당 열에 순차적으로 저장된다. 각 글자가 POI 데이터베이스에 등장하는 횟수만큼 루프를 돌며 역 인덱스를 이용하여 POI 쿼리의 해당 글자가 POI 데이터베이스에 등장하는 레코드 번호를 2차원 배열에 저장한다. 또한, 루프를 통해 각 글자가 POI 데이터베이스에 등장하는 횟수를 계산한다.

POI 쿼리로 '힐탑트래저파트'을 입력했을 때의 전처리 결과는 그림 9와 같다. 텍스트 검색을 위해 POI 쿼리에 대한 정보와 이와 관련된 POI 데이터베이스의 레코드 정보를 파악한다. '힐탑트래저아파트'의 글자 개수는 8이며, 각 글자를 바이트 단위로 변환하여 배열에 저장한다. 아이디 검색 트리를 이용하여 각 글자에 대한 아이디를 알아낸다. POI 쿼리의 글자 중 '래'와 '저' 샘플 레코드 집합에 존재하지 않는 글자이므로 아이디가 존재

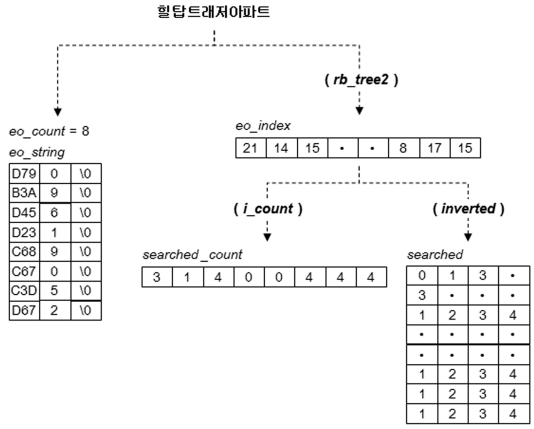


그림 9 POI 쿼리에 대한 텍스트 검색 전처리 결과

하지 않는다. 역 인덱스와 관련 정보의 인덱스로 글자의 아이디를 사용함으로써 각 글자를 포함하고 있는 레코드의 개수와 레코드 번호를 파악할 수 있다. POI 쿼리 중 첫 번째 글자인 '힐'의 글자 아이디는 21이고, 글자를 포함하는 레코드 번호는 0, 1, 3으로 총 3개이다.

3.4.2 블록 내 연산 알고리즘

전처리 과정을 통해 사용자의 POI 쿼리에 대한 기본 정보를 파악한 뒤, 쿼리를 m개의 블록으로 균등 분할한다. 하나의 블록은 두 글자 이상으로 이루어져야 한다. 본 알고리즘은 각 블록 내의 글자가 공통으로 포함된 POI 데이터베이스의 레코드 번호를 파악하며, 이를 위해 3.4.1에서 완성한 배열을 이용한다. 이 때, 해당 배열에 저장되어 있는 POI 쿼리의 글자 별 POI데이터베이스의 레코드 번호는 오름차순으로 정렬되어 있다고 가정한다.

블록 내의 연산 알고리즘은 블록 내의 각 글자를 포함하는 POI 데이터베이스의 레코드 번호를 비교하는 연산을 실시하며, 세부적인 과정은 그림 10에 표현되어 있다. 각 블록의 첫 번째 글자와 두 번째 글자를 각각 포함하는 POI 데이터베이스의 레코드 번호를 비교한 뒤, 두 글자를 모두 포함하는 레코드 번호를 파악한다. 앞의 두 글자를 포함하는 레코드 번호와 세 번째 글자를 포함하는 레코드 번호를 비교하며, 블록 내의 마지막 글자까지 모두 순차적으로 비교한다. 파악한 레코드 번호는 3차원의 배열에 비교 순서 별로 저장된다.

블록 내 연산 알고리즘을 샘플 레코드 집합에 적용하면 그림 11과 같이 나타나며, 점선은 글자에 대한 레코드 번호를 비교하는 연산을 의미한다. 그림 9와 같은 POI 쿼리인 '힐탑트래저아파트'를 3개의 블록으로 분할하고, 각 블록에 대한 연산을 실시한다. 첫 번째 블록의 경우, 글자 '힐'과 '탑'을 모두 포함하는 레코드 번호는 3

```
OperANDwithinChunk (POI query Q, searched)
 1:  chunk ← array for storing record number for each chunk
 2:  chunk_count ← array for string the number of characters
 3:                  for each chunk
 4:  index1 ← 0
 5:  count ← 0
 6:  divide the POI query Q into m chunks
 7:  for each chunk H in the POI query Q do
 8:      chunk[index1][0][ ] ← searched[count][ ]
 9:      max ← the number of characters in the chunk H
10:      for index2 ← 1 upto (max-1) do
11:          compare chunk[index1][(index2)-1][ ] and searched[index2][ ]
12:          index3 ← 0
13:          for each the same number N do
14:              chunk[index1][index2][index3] ← N
15:              index3 ← index3 + 1
16:          end for
17:      end for
18:      count ← count + max
19:      chunk_count[index1] ← count
20:      index1 ← index1 + 1
21:  end for
22:  return chunk, chunk_count
```

그림 10 블록 내 연산 알고리즘

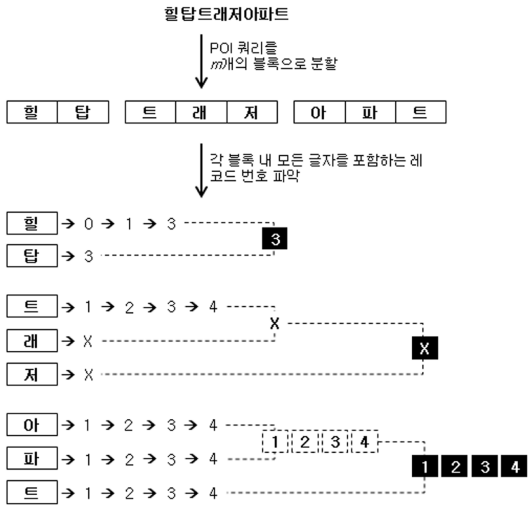


그림 11 POI 쿼리에 대한 블록 내 연산의 결과

이다. 두 번째 블록은 글자 '트'와 '래'의 레코드 번호를 비교한 결과와 '저'의 레코드 번호를 비교한다. 글자 '트'와 '래'는 POI 데이터베이스에 존재하지 않는 글자이므로, 두 번째 블록 내의 글자를 모두 포함하는 레코드 번호는 존재하지 않는다. 마지막 블록의 연산 과정은 두 번째 블록의 연산 과정과 같으며, 세 글자를 모두 포함하는 레코드 번호는 1, 2, 3, 4이다.

3.4.3 블록 간 연산 알고리즘

블록 간 연산은 3.4.2에서 수행한 연산을 통해 얻은 검색 후보에 대해 다른 블록의 정보를 활용하여 추가적인

```
OperANDbetweenChunks
(POI query Q, chunk, chunk_count, searched)
 1:  degree ← array for storing degree for each chunk
 2:  final ← array for storing record number for each chunk
 3:
 4:  index1 ← 0
 5:  for each chunk H in the POI query Q do
 6:      count ← 0
 7:      for each character c in the POI query Q do
 8:          if (count < index1 * chunk_count[index1])
               or (count > (index1 + 1) * chunk_count[index1] – 1) then
 9:              compare chunk[index1][chunk_count[index1]][ ]
                                              and searched[count][ ]
10:              index2 ← 0
11:              for each the same number N do
12:                  degree[index1] ← degree[index1] + 1
13:                  final[index1][index2] ← N
14:                  index2 ← index2 + 1
15:              end for
16:          end if
17:      end for
18:      index1 ← index1 + 1
19:  end for
20:  return degree, final
```

그림 12 블록 간 연산 알고리즘

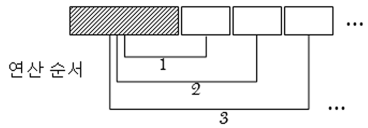
제약을 가함으로써 검색 후보의 수를 축소시킨다. 본 알고리즘에서는 '차수'라는 개념이 사용되는데, 이는 '각 레코드가 포함하고 있는 POI 쿼리의 글자 개수'를 의미한다.

그림 12는 블록 간 연산 알고리즘을 나타낸다. POI 데이터베이스 내에서 해당 블록의 글자들이 공통으로 포함되어 있는 레코드 번호들과 해당 블록이 포함하지 않는 다른 글자를 포함하는 레코드 번호 각각을 비교하여 해당 블록의 검색 후보 레코드에 대한 차수를 계산한다.

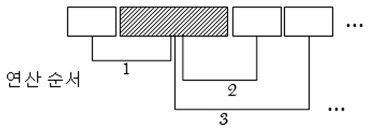
해당 블록의 각 후보 레코드의 차수를 계산하는 과정은 그림 13과 같으며, 회색 사각형은 블록, 흰색 사각형은 글자 하나를 나타낸다. 해당 블록의 각 후보 레코드가 블록 이외의 각 글자를 포함하고 있는지를 판단하여, 포함하는 경우 차수를 1 증가시킨다. 즉, '각 블록 내의 글자를 모두 포함하는 POI 데이터베이스 레코드가 해당 블록을 제외한 글자를 몇 개나 더 포함하는가'를 파악한다.

각 블록의 차수를 계산하고 난 뒤에는, 블록 간 차수를 연산한다. 각각의 블록을 이용하여 검색된 POI 데이터베이스 레코드들의 모든 차수가 계산되며, 최종적으로 최대 차수를 갖는 POI 데이터베이스의 레코드 번호들이 검색 결과로서 출력된다.

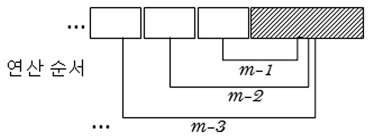
분할한 POI 쿼리의 각 블록에 대해 차수를 구하는 과정은 그림 14에 표현되어있으며, 각각 첫 번째 블록의 경우와 마지막 블록의 경우를 나타낸다. 두 번째 블록은 블록 내 글자를 모두 포함하는 레코드가 존재하지 않으므로, 차수를 계산하지 않는다.



(a) 1번째 블록의 경우

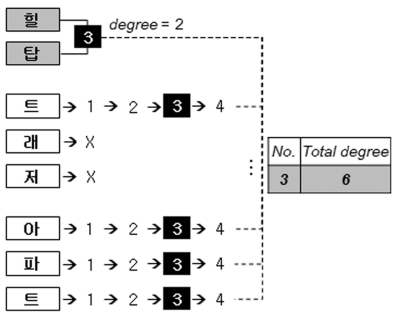


(b) 2~(m-1)번째 블록의 경우

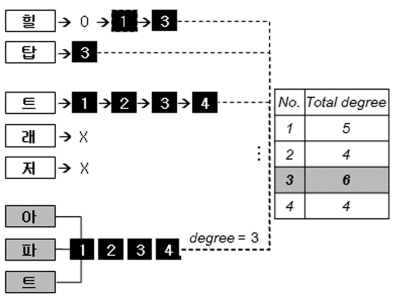


(c) m번째 블록의 경우

그림 13 블록 간 연산 알고리즘의 연산 순서



(a) 첫 번째 블록의 경우



(b) 마지막 블록의 경우

그림 14 쿼리에 대한 블록 간 연산 결과

첫 번째 블록의 경우, 3번 레코드가 블록의 글자 두 개를 기본적으로 포함하므로 첫 번째 블록의 초기 차수는 2이다. 해당 블록 이외의 글자 중 3번 레코드에 포함되어 있는 글자가 존재하면 차수를 1씩 증가시킨다. 그 결과, 첫 번째 블록에 대한 최종 차수는 6이며, 이는 3번 레코드에 POI 쿼리의 글자 6개가 포함되어 있음을 의미한다.

마지막 블록을 구성하는 글자 3개는 레코드 번호 1, 2, 3, 4에 해당하는 레코드에 모두 포함되어 있다. 그러므로 마지막 블록의 초기 차수는 3이며, 각 레코드 번호 각각에 대해 해당 블록 이외의 글자를 포함하는 번호 각각에 대해 해당 블록 이외의 글자를 포함하는 레코드 번호를 비교한다. 1번 레코드는 블록 내 글자와 함께 글자 '힐'과 '트'를 포함하며, 최종 차수는 5이다.

각 블록의 레코드 번호 중 최대 차수를 레코드 번호가 최종 결과가 된다. 그림 14의 경우, 최대 차수 6을 갖는 레코드 3번에 해당하는 '힐탑트래져아파트'는 POI 쿼리 '힐탑트레저아파트'와 가장 근접한 POI 데이터베이스 내 레코드이다.

블록 내 연산 알고리즘은 POI 쿼리의 각 글자를 기준으로 공통으로 포함하는 POI 데이터베이스 레코드 번호를 파악하는 반면, 블록 간 연산 알고리즘은 각 레코드가 갖고 있는 POI 쿼리의 글자 수를 파악한다.

이상의 검색 알고리즘은 POI 쿼리에 등장하는 순서와 POI 데이터베이스에 등장하는 글자의 순서에 상관없이 검색할 수 있다는 장점을 지닌다.

4. 성능 평가

본 논문에서 제시한 알고리즘의 실효성과 성능을 입증하기 위해 실험을 수행하였다. 총 260만 개의 레코드로 구성된 원본 POI 데이터베이스에서 추출한 POI 집합을 대상 데이터로 하였으며, 상용 독립형 시스템에서 사용하는 하드매칭 기법과 본 논문에서 제시하는 알고리즘을 비교하는 방식으로 진행되었다.

실험은 대상 데이터 각각에 대해 예상되는 POI 쿼리를 유추한 뒤, 이를 하드매칭 기법과 본 논문의 알고리즘에 적용하여 검색 결과를 비교한다. 검색의 성공 여부는 예상 POI 쿼리에 대한 검색 결과에서 정답 데이터의 랭킹을 기준으로 판단한다. 정답 데이터가 1위에 랭크되면 성공, 상위 20위 안에 랭크된 경우에는 준성공, 그렇지 않은 경우에는 실패라 간주한다.

실험에 사용된 대상 데이터는 두 개의 그룹으로 나뉜다. 원본 POI 데이터베이스에서 무작위로 추출된 250개의 POI 집합을 '그룹 1', 쿼리의 유추가 어려운 250개의 POI 집합을 '그룹 2'라 한다. '현대자동차'와 같이 쿼리 유추가 용이한 POI는 그룹 1에 포함되고, '비젼헬스크럽'과 같이 정확한 쿼리 유추가 어려운 POI는 그룹 1과 그룹 2에 모두 포함될 수 있다.

그룹 1은 무작위로 추출한 POI의 집합이므로, POI 쿼리 유추가 비교적 쉬운 데이터와 어려운 데이터 모두

를 포함한다. 그러므로, 그룹 1의 POI와 유추한 쿼리의 일치 여부는 각각의 POI에 따라 다르게 나타난다. 쿼리를 쉽게 유추할 수 있는 POI의 경우에는 유추한 쿼리와 일치하지만, 쉽게 유추할 수 없는 POI의 경우에는 일치하지 않는다. 예를 들어, POI가 '현대 자동차'인 경우에는 이와 일치하는 POI 쿼리를 유추하였고, '비젼헬스크럽'을 검색하는 경우에는 쿼리를 '비전헬스클럽'으로 유추한 뒤 검색을 실시하였다. 또한, POI의 쿼리 유추 방법은 POI의 문자 순서를 바꾸는 방법도 포함한다. 예를 들어, '추어탕해장국'이라는 POI의 경우, 쿼리를 '해장국추어탕'으로 작성하여 검색 결과를 확인하였다.

표 1은 그룹 1에 대한 실험 결과를 나타낸다. 하드매칭 기법과 본 논문에서 제시하는 알고리즘의 검색 결과를 수치로 비교하였다. 총 250개의 POI에 대하여 유추한 쿼리로 검색을 실시했을 때의 결과를 성공, 준성공, 실패의 경우로 나누어 각각의 POI 개수와 백분율을 계산하였다. 대상 POI 중 하드매칭 기법과 본 논문의 알고리즘 각각 62%, 88%의 POI를 성공적으로 검색하였으며, 준성공한 경우를 포함하면 각각 62%, 96%의 검색율을 나타내었다.

그룹 2는 그룹 1과 달리 검색이 어려운 POI 중에서 250개를 추출한 데이터 집합으로, '인하횟집', '후렌드치킨' 등이 이에 속한다. 그룹 2에 속하는 POI는 음운현상이나 외래어를 포함하므로, POI와 일치하는 쿼리를 유추하기가 쉽지 않다. 예를 들어, '인하횟집'과 같이 음운현상을 포함하는 POI의 경우에는 발음 그대로 유추하여 '이나횟집'으로 검색을 실시하였으며, '후렌드치킨'과 같은 외래어는 '후랜드치킨', '프렌드치킨' 등 POI 쿼리를 다양하게 유추하였다. 그룹 2에 속하는 250개의 POI 각각에 대해 유사하지만 일치하지 않는 POI 쿼리를 유추하여 검색을 실시하였다. 표 2에 250개 중 10개를 예시로 열거하였으며, POI 데이터와 일치하지 않는 쿼리 글자는 강조하였다.

표 3은 그룹 2에 대한 검색 기법 별 성능을 비교한 결과로, 표 1과 구성이 같다. POI와 일치하지 않는 쿼리를 입력하여 검색을 실시하였으므로, 하드매칭 기법을 사용한 경우에는 올바른 데이터를 전혀 검색하지 못하였다. 그에 비해 본 논문에서 제시하는 알고리즘을 사용한 검색은 36%의 성공률을 나타내었으며, 준성공한 경우를 포함하면 83%의 검색율을 나타내었다.

표 1 실험 데이터 '그룹 1'에 대한 실험 결과

구분	총	성공		준성공		실패	
하드 매칭 기법	250	155	62%	0	0%	95	38%
집합-기반 POI 검색 알고리즘	250	219	88%	21	8%	10	4%

표 2 실험 데이터 '그룹 2'의 실제 예

No	POI Data	POI Query
1	인하횟집	이나횟집
2	후렌드치킨	후랜드치킨
3		프랜드치킨
4		프렌드치킨
5	쥬빌리쇼콜라티에	주빌리쇼콜라티에
6		쥬빌리쇼콜라티애
7		쥬빌리쇼콜라띠애
8		주빌리쇼콜라띠애
9	안의부동산	아니부동산
10	앙떼떼	앙때때

표 3 실험 데이터 '그룹 2'에 대한 실험 결과

구분	총	성공		준성공		실패	
하드 매칭 기법	250	0	0%	0	0%	250	100%
집합-기반 POI 검색 알고리즘	250	90	36%	118	47%	42	17%

이상의 실험 결과를 종합하면, 본 논문에서 제시하는 알고리즘의 성공률과 검색율이 기존의 하드매칭 기법에 비해 현저히 높아졌다. 하드매칭 기법은 문자의 유사도 개념을 사용하지 않으므로, 무작위로 추출한 그룹 1에 대해서는 낮은 검색율을 나타내었고, 검색이 어려운 그룹 2에 대해서는 검색에 모두 실패하였다. 그러나, 본 논문에서 제시하는 알고리즘을 적용한 경우에는 그룹 1에 대해 90% 이상의 높은 검색율을, POI 쿼리를 유추하기 힘든 그룹 2에 대해서도 80%이상의 검색율을 보였다. 따라서, 본 논문에서 제시하는 알고리즘이 모든 POI 데이터 검색에 대한 성공률과 검색율을 현저히 향상시켰음을 알 수 있다.

5. 결 론

본 논문에서는 부정확한 POI 쿼리 입력으로 인한 POI 검색 서비스의 성능 저하 문제를 개선하기 위한 새로운 알고리즘을 제시하였다. 기존의 POI 검색 시스템에서는 정보 검색 분야에서 널리 사용되는 쿼리 자동 확장 기법을 사용하지만, 이는 외부 자원을 사용해야 하기 때문에 독립형 시스템에 대해서는 적용할 수 없다는 한계점을 지닌다. 그러나 본 알고리즘은 사용자의 부정확한 POI 쿼리를 알고리즘 차원에서 재구성하여 POI 검색 서비스의 성능 저하를 개선시킨다.

본 알고리즘은 집합의 연산을 적용하기 위해 사용자의 POI 쿼리를 블록으로 나누어 블록 내 연산, 블록 간 연산을 수행한다. 각 블록의 글자가 모두 포함된 POI

데이터베이스의 레코드 개수와 번호를 파악하여 블록 내 연산을 수행한다. 블록 간 연산에는 '차수'라는 개념이 사용되며, 블록 내 연산을 통해 얻은 검색 후보에 대해 다른 블록의 정보를 활용하여 추가적인 제약을 가함으로써 검색 후보의 수를 축소시킨다.

실험을 통해 본 논문에서 제안하는 알고리즘을 이용한 POI 검색 시스템이 기존의 하드매칭 기법을 사용하는 시스템에 비해 검색 성능이 현저히 높다는 것을 검색율과 성공률로 증명하였으며, 특히 기존의 기법으로는 검색이 되지 않는 POI에 대해 검색율을 향상시키는 것을 확인하였다.

참 고 문 헌

[1] Xinyan Zhu, Chunhui Zhou, "POI Inquiries and data update based on LBS," *Proc. of the International Symposium on Information Engineering and Electronic Commerce (IEEC)*, pp.730-734, 2009.

[2] Steven C.H. Hoi, Jiebo Luo, Susanne Boll, Dong Xu, Rong Jin, Social Media Modeling and Computing, p.245, *Springer*, 2011.

[3] Jinxi Xu, W. Bruce Croft, "Improving the effectiveness of information retrieval with local context analysis," *ACM Transactions on Information Systems*, vol.18, no.1, pp.79-112, 2000.

[4] Paul A Zandbergen, "Accuracy of iPhone Locations: A Comparison of Assisted GPS, WiFi and Cellular Positioning," *Transactions in GIS*, vol.13(s1), pp.5-25, 2009.

[5] Yuen-Hsien Tsieng, Da-Wei Juang, Shiu-Han Chen, "Global and Local Term Expansion for Text Retrieval," *Proc. of the Fourth NTCIR Workshop on Evaluation of Information Retrieval*, 2004.

[6] Ian Ruthven, Mounia Lalmas, "A survey on the use of relevance feedback for information access systems," *Knowledge Engineering Review*, vol.18, no.2, pp.95-145, 2003.

[7] Karen Sparck Jones, Automatic Keyword Classification for Information Retrieval, Butterworths, London, 1971.

[8] Scott Deerwester, Susan T. Dumais, Geroge W. Furnas, Thomas K. Landauer, and Richard Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science*, vol.41, no.6, pp.391-407, 1990.

[9] Parry Husbands, Horst Simon, Chris H. Q. Ding, "On the use of the singular value decomposition for text retrieval," *SIAM Computational information retrieval*, pp.145-156, 2001.

[10] Chris Okasaki, "Functional Pearl: Red-Black trees in a functional setting," *Journal of Functional Programming*, vol.9, no.4, pp.471-477, 1999.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 3rd Ed., pp.308-338, MIT Press and McGraw-Hill, 2009.

고 은 비

2011년 숙명여자대학교 멀티미디어과학과 졸업(학사). 2013년 한국과학기술원 전산학과 졸업(석사). 관심분야는 검색시스템, 자연어처리, 인공지능, 모바일 소프트웨어

이 종 우

1990년 서울대학교 컴퓨터공학과(학사) 1992년 서울대학교 컴퓨터공학과(석사) 1996년 서울대학교 컴퓨터공학과(박사) 1996년~1998년 현대전자(주) 정보시스템 사업본부 과장. 1999년~1999년 현대정보기술(주) 책임연구원. 1999년~2002년 한림대학교 정보통신공학부 조교수. 2002년~2003년 광운대학교 컴퓨터공학부 조교수. 2003년~2004년 아이닉스소프트(주) 개발이사. 2004년~현재 숙명여자대학교 정보과학부 멀티미디어과학전공 조교수. 2008년 뉴욕주립대 스토니브룩 Research Scholar. 2012년~현재 숙명여자대학교 지식정보처장. 관심분야는 Mobile System Software, Storage Systems, Computational Finance, Cluster Computing, Parallel and Distributed Operating Systems, and Embedded System Software

이 재 원

1990년 서울대학교 컴퓨터공학과(학사) 1992년 서울대학교 컴퓨터공학과(석사) 1998년 서울대학교 컴퓨터공학과(박사) 1999년~현재 성신여자대학교 IT학부 부교수. 관심분야는 금융공학, 인공지능, 기계학습, 자연어처리