DGR-Tree : u-LBS에서 POI의 검색을 위한 효율적인 인덱스 구조

(DGR-Tree : An Efficient Index Structure for POI Search in Ubiquitous Location Based Services)

이 득 우* 강 홍 구**
(Deuk-Woo Lee) (Hong-Koo Kang)

이 기 영*** 한 기 준****
(Ki-Young Lee) (Ki-Joon Han)

요 약 유비쿼터스 컴퓨팅 환경에서의 LBS, 즉 u-LBS는 실세계의 수많은 객체가 위치정보와 밀접히 연관된 대용량 데이타를 대상으로 한다. 특히, 사용자의 위치정보와 관련하여 검색하려고 하는 객체인 POI에 대한 빠른 검색이 중요하다. 따라서 u-LBS에서 POI의 효율적인 검색을 위한 인덱스 구조에 대한 연구가 필요하다. 본 논문에서는 u-LBS에서 정적 POI를 대상으로 이를 효율적으로 검색하기 위한 DGR-Tree를 제시한다. DGR-Tree는 변형된 R-Tree를 기본 인덱스로 하고 동적 레벨 그리드를 보조 인덱스로 사용하는 구조이다. DGR-Tree는 점 데이타에 적합하도록 최적화하고 있으며 리프 노드 간 겹침 문제를 해결한다. DGR-Tree에서 동적 레벨 그리드는 점 데이타의 밀집도에 따라 동적으로 구성되며, 각 셀은 DGR-Tree의 리프 노드와 연계를 위한 포인터를 저장하여 리프 노드를 직접 접근하도록 함으로써 인덱스 접근 성능을 향상시킨다. 또한, 본 논문에서는 DGR-Tree를 위한 KNN 검색 알고리즘을 제시한다. 이 알고리즘에서는 KNN 검색 시 후보 셀에 빠르게 접근하기 위하여 동적 레벨 그리드를 활용하며, 후보를 노드별로 구분하여 저장함으로써 후보 리스트 내에서의 정렬 비용을 감소시킨다. 마지막으로 실험을 통해 DGR-Tree의 우수성을 입증하였다.

키워드 : u-LBS, DGR-Tree, 동적 레벨 그리드, KNN

Abstract Location based Services in the ubiquitous computing environment, namely u-LBS, use very large and skewed spatial objects that are closely related to locational information. It is especially essential to achieve fast search, which is looking for POI(Point of Interest) related to the location of users. This paper examines how to search large and skewed POI efficiently in the u-LBS environment. We propose the Dynamic-level Grid based R-Tree(DGR-Tree), which is an index for point data that can reduce the cost of stationary POI search. DGR-Tree uses both R-Tree as a primary index and Dynamic-level Grid as a secondary index. DGR-Tree is optimized to be suitable for point data and solves the overlapping problem among leaf nodes. Dynamic-level Grid of DGR-Tree is created dynamically according to the density of POI. Each cell in Dynamic-level Grid has a leaf node pointer for direct access with the leaf node of the primary index. Therefore, the index access performance is improved greatly by accessing the leaf node directly through Dynamic-level Grid. We also propose a K-Nearest Neighbor(KNN) algorithm for DGR-Tree, which utilizes Dynamic-level Grid for fast access to candidate cells. The KNN algorithm for DGR-Tree provides the mechanism, which can access directly to cells enclosing given query point and adjacent cells without tree traversal. The KNN algorithm minimizes sorting cost about candidate lists with minimum distance and provides NEB(Non Extensible Boundary), which need not consider the extension of candidate nodes for KNN search.

Keywords : u-LBS, Dynamic-level Grid based R-Tree, Dynamic-level Grid, K-Nearest Neighbor

1. 서 론

유비쿼터스 컴퓨팅 환경에서의 LBS(Location Based Services), 즉 u-LBS를 원활히 지원하기 위해서는 사용자의 위치정보와 관련하여 검색하려고 하는 객체인 POI (Point of Interest)에 대한 효율적인 검색이 매우 중요하다[5]. u-LBS에서는 POI의 개념이 점차 확장됨으로 인해 POI의 종류가 점차 다양해지고 증가하는 추세이며 공간적으로 특정 지역에 연관되고 밀집되는 경향을 보인다. 따라서 u-LBS에서 POI의 효율적인 검색을 위해서는 POI의 특성을 고려한 인덱스 구조에 대한 연구가 필요하다.

POI와 같은 공간상의 데이타를 검색하기 위한 인덱스 구조로는 공간 분할 인덱스 구조 및 데이타 분할 인덱스 구조와 하이브리드 인덱스 구조가 있다. 공간 분할 인덱스 구조는 삽입 알고리즘이 단순하고 삽입 시간이 적은 반면 데드 스페이스로 인해 공간 효율성이 낮고 대용량 데이타 처리가 어렵다[4]. 데이타 분할 인덱스 구조는 공간 효율성이 높으며 KNN(K-Nearest Neighbor)과 같은 복잡한 검색이 빠르다. 그러나 삽입 알고리즘이 복잡하여 삽입 비용이 크며, 노드 간에 겹침으로 인한 다중 검색 경로가 발생한다[1,10]. 하이브리드 인덱스 구조는 노드 간 겹침 문제를 일부 해소하였지만 여전히 노드 간 겹침 문제가 발생하여 검색 속도가 느리다는 단점을 가지고 있다[3,7].

따라서 본 논문에서는 u-LBS에서 POI를 효율적으로 검색하기 위한 새로운 인덱스 구조인 DGR-Tree

† 이 논문은 2009 GIS 공동추계학술대회에서 'DGR-Tree : u-LBS에서 POI의 효율적인 검색을 위한 인덱스 구조'의 제목으로 발표된 논문을 확장한 것임

* 건국대학교 컴퓨터·정보통신공학과 공학박사, deukwoo@db.konkuk.ac.kr
** 건국대학교 컴퓨터공학과 강의교수, hkkang@db.konkuk.ac.kr
*** 을지대학교 의료산업학부 교수, kylee@eulji.ac.kr(교신저자)
**** 건국대학교 컴퓨터공학과 교수, kjhan@db.konkuk.ac.kr

논문접수 : 2009.08.07
수 정 일 : 2009.09.10
심사완료 : 2009.09.25

(Dynamic-level Grid based R-Tree)를 제시한다. DGR-Tree의 기본 인덱스인 변형된 R-Tree에서 넌 리프 노드의 분할은 기존의 R-Tree와 동일하나 리프 노드 분할은 그리드를 기반으로 최대 4개까지 분할함으로써 노드 분할 알고리즘이 단순하며 리프 노드 간 겹침 문제를 해결한다. 동적 레벨 그리드(Dynamic-level Grid)는 DGR-Tree에 대한 접근 속도를 개선하기 위한 보조 인덱스로서 POI의 분포에 따라 동적으로 구성된다.

또한 본 논문에서는 DGR-Tree를 위한 KNN 검색 알고리즘을 제시한다. KNN 검색 알고리즘에서는 KNN 검색 시 후보 셀을 빠르게 접근하기 위하여 동적 레벨 그리드를 통해 질의점(Query Point)을 포함한 셀 및 주변 셀에 빠르게 접근한다. 또한, 후보를 노드별로 구분하여 저장함으로써 후보 리스트 내에서의 정렬 비용을 감소시킨다. 마지막으로 성능 실험을 통해 DGR-Tree의 우수성을 입증하였다.

2. 관련 연구

본 장에서는 점 데이타를 위한 대표적인 인덱스 구조와 KNN 검색 알고리즘에 대해서 분석한다.

2.1 점 데이타를 위한 기존 인덱스 구조

점 데이타를 위한 대표적인 공간 분할 인덱스 구조에는 Quad-Tree, KD-Tree[2], PK-Tree[11], KDB-Tree[8] 등이 있다. Quad-Tree 및 메모리 기반 다차원 데이타 구조로서 k차원 점 객체를 위한 이진 검색 트리인 KD-Tree의 구조는 높이 균형 인덱스가 아니기 때문에 공간 데이타가 균등하게 분포되지 않으면 트리가 한쪽으로 치우쳐 성능이 떨어지는 문제를 가지고 있다. 따라서 편향 분포 형태를 가지는 대용량 공간 객체를 다루는 u-LBS 환경에 적합하지 않다.

PK-Tree는 PR-Quad 트리 또는 KD-Tree의 특성을 결합한 인덱스 구조로서 불필요한 노드를 제거함으로써 편향 분포를 갖는 고차원 점 데이타에 대해 성능을 높였다. 그러나 높은 차원을 갖는 점 데이타가 아닌 2차원 점 데이타에 대해서는 적합하지 못한 단점이 있다. KDB-Tree는 KD-Tree와 B-Tree의 특성을 결합한 인덱스 구조이며, 넌 리프 노드가 분할되면서 분할 차원과 분할 위치에 따라 하위 노드를 연쇄적으로 분할시키는 하향 연쇄 분할 문제를 가지고 있다. 따라서 공간 객체가 증가하면서 노드 분할이 급격히 증가하는 단점이 있다.

R-Tree, R^*-Tree, R^+-Tree 등 점 데이타를 포함한 크기를 갖는 공간 데이타를 처리하는 데이타 분할 인덱스 구조는 공간 효율성이 높으며 KNN과 같은 복잡한 검색에 빠르다. 그러나 삽입 알고리즘이 복잡하고 삽입 시간이 많으며, 노드 간에 겹침으로 인한 다중 검색 경로가 발생하는 문제를 가진다[1,10].

LBS에서 보편적으로 많이 사용되던 R-Tree를 보완하기 위한 QR-Tree[7], iQR-Tree(Improved QR-Tree)[3] 등은 대표적인 하이브리드 인덱스 구조이다. QR-Tree에서 상위 구조는 Quad-Tree이며, 하위 구조인 R-Tree는 Quad-Tree의 리프 노드마다 연계되어 실제로 공간 데이타를 저장하는 역할을 수행한다. QR-Tree에서 일부 객체는 Quad-Tree의 분할 경계에 걸쳐 있으며 해당 공간 데이타를 여러 Quad-Tree 노드에 중복하여 저장한다. 이처럼, QR-Tree는 공간 데이타가 중복 저장되기 때문에 인덱스 크기가 커지고 공간 낭비가 심해지는 문제를 가지고 있다.

iQR-Tree는 QR-Tree 중복 저장 문제를 해결하기 위해 변형한 인덱스 구조이다. iQR-Tree는 Quad-Tree의 분할 경계에 걸쳐진 공간 데이타를 새로운 R-Tree에 저장함으로써 중복 저장 문제를 해결하고, 질의 처리 성능과 저장 공간 활용성을 향상시킨다. 그러나 Quad-Tree 노드에 연계된 R-Tree 사이의 겹침으로 인해 Quad-Tree의 레벨이 높아질수록 질의 처리 성능이 급격히 떨어지는 문제가 있다.

2.2 KNN 검색 기법

LBS 환경에서 트리-기반의 인덱스에 적용 방법으로서 MinMax 거리를 이용한 가지치기(Branch and Bound) 방법이 있다[9]. 그러나 이 방법은 연산의 복잡성으로 인해 일반적으로 거리 기반 KNN 검색 방식이 많이 사용된다[6]. 거리 기반 KNN 검색 방식은 최소 거리(Minimal Distance)와 우선순위 큐(Priority Queue)를 사용하는 방식으로서 Quad-Tree, KD-Tree, R-Tree 등의 다양한 인덱스 구조에 적용할 수 있다는 장점이 있다.

거리 기반 KNN 검색 방식은 우선순위 큐에 객체를 저장하고 질의 객체와의 거리 순서대로 정렬한 후 가장 거리가 가까운 객체 또는 노드를 확장하면서 KNN 검색을 수행한다. 노드가 확장될 경우 질의점으로 부터 자식 노드의 MBB까지 각각의 거리를 계산하고 이들 자식 노드를 우선순위 큐에 추가하고, 객체가 확장될 경우 객체를 기록하고 확장을 계속 진행한다.

3. DGR-Tree

3.1 DGR-Tree의 특성

DGR-Tree는 변형된 R-Tree를 기본 인덱스로 하고 데이타의 밀집도에 따라 구성되는 동적 레벨 그리드를 보조 인덱스로 사용한다. 그림 1은 DGR-Tree의 예를 보여준다.

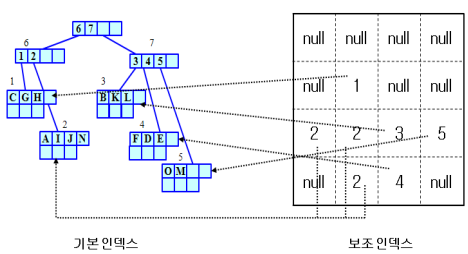

그림 1. DGR-Tree의 예

그림 1에서 화면의 왼쪽 그림은 DGR-Tree의 기본 인덱스(이하 DGR-Tree라 칭함)를 보여주며, 오른쪽 그림은 동적 레벨 그리드를 보여준다. 동적 레벨 그리드를 구성하고 있는 각 셀은 해당 셀과 연관되는 DGR-Tree의 리프 노드를 가리키고 있으며, 이를 통해 인덱스 접근 성능을 크게 향상시킨다. 그림 2는 DGR-Tree와 동적 레벨 그리드의 연결 예를 보여준다.

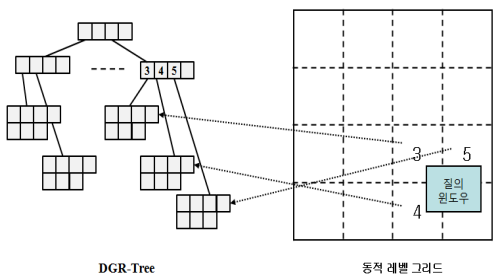

그림 2. DGR-Tree와 동적 레벨 그리드의 연결 예

그림 2에서 보인 바와 같이 사용자가 윈도우 검색을 요청한 경우, 트리의 검색 없이 동적 레벨 그리드를 통해 질의 윈도우와 겹치는 DGR-Tree의 리프 노드 3, 4, 5를 빠르게 찾을 수 있으며, 동적 레벨 그리드의 각 셀이 저장하고 있는 리프 노드의 연결 정보를 통해 해당 노드에 직접 접근할 수 있게 된다. 이를 통해 POI 검색 성능을 크게 향상시킬 수 있다. 또한 DGR-Tree는 리프 노드 분할 시 동적 레벨 그리드를 기반으로 분할함으로써 기존 인덱스에서 존재하는 리프 노드 간 겹침 문제를 해결한다. 그림 3은 DGR-Tree의 리프 노드 간 겹침 문제 해결을 설명하기 위한 노드 분할의 예를 보여준다.

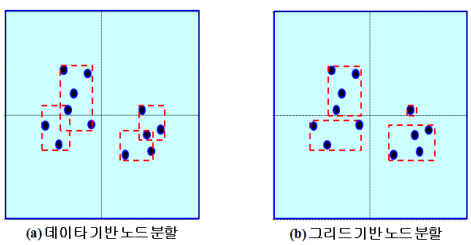

그림 3. 노드 분할의 예

그림 3(a)와 그림 3(b)는 동일한 데이타에 대해 노드의 차수를 4로 가정하였을 경우 각각 데이타 기반 노드 분할과 그리드 기반 노드 분할의 예를 보여준다.

DGR-Tree의 리프 노드에는 객체의 MBB를 저장하지 않고 점 데이터만을 저장함으로써 리프 노드에서는 넌 리프 노드의 엔트리에 비해 많은 엔트리를 저장할 수 있다. 이로 인해 저장 공간의 효율성을 높일 수가 있으며, 리프 노드의 분할 지연으로 인해 DGR-Tree 및 동적 레벨 그리드의 레벨을 낮출 수가 있다.

3.2 DGR-Tree 인덱스 구조

그림 4와 그림 5는 각각 DGR_Tree 노드의 자료 구조와 넌 리프 및 리프 노드의 엔트리 자료 구조를 보여준다.

parent node pointer	MBB	entry number	entries

그림 4. DGR-Tree 노드의 자료 구조

child node MBB	child node pointer

(a) 넌 리프 노드의 엔트리 자료 구조

point	OID(object identifier)

(b) 리프 노드의 엔트리 자료 구조

그림 5. DGR-Tree 노드의 엔트리 구조

그림 4에서 처럼 DGR-Tree의 노드는 부모 노드의 포인터, 노드의 MBB, 엔트리 개수, 엔트리들로 구성되며, 그림 5에서 처럼 고유의 엔트리를 가진다.

DGR-Tree에서는 리프 노드가 POI에 적합하게 설계되었기 때문에 리프 노드와 넌 리프 노드의 엔트리 개수의 범위가 기존의 인덱스와는 다르다. 표 1은 DGR-Tree 노드의 엔트리 개수 범위를 보여준다.

표 1. DGR-Tree 노드의 엔트리 개수

분류	엔트리 개수	엔트리 개수의 범위
넌 리프 노드	n	$m/2 \le n \le m$ 단, 루트 노드 제외
리프 노드	n′	$1 \le n' < 2m$

표 1에서 m은 DGR-Tree의 차수를 의미한다. DGR-Tree의 차수는 4 이상의 값을 가진다. DGR-Tree에서 넌 리프 노드의 엔트리 개수 범위는 R-Tree와 동일하나 리프 노드 엔트리 개수의 범위는 1보다 크거나 같으며 2m보다 작다.

3.3 동적 레벨 그리드 구조

동적 레벨 그리드는 Grid(i)로 표현되며, Grid(i)는 i 레벨을 가지는 동적 레벨 그리드로서 X축과 Y축이 각각 2^i 개로 분할이 이루어진 셀로 구성된 그리드를 의미한다.

그림 6은 Grid(i)의 예로서 각 셀에는 검색 성능 향상을 위해 연계된 DGR-Tree 리프 노드의 포인터 및 MBB와 각 셀의 논리적 레벨을 저장한다. 논리적 레벨이란 해당 셀의 논리적인 그리드 분할 레벨을 의미하고, 물리적 레벨은 해당 셀의 물리적인 그리드 분할 레벨을 의미하며 동적 레벨 그리드의 레벨과 같은 값을 가진다.

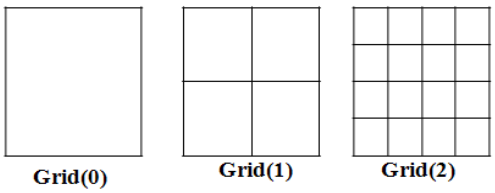

그림 6. Grid(i)의 예

그림 7. 논리적 레벨의 예

그림 7은 Grid(2), 즉 레벨이 2인 동적 레벨 그리드를 물리적으로 구성하고 있는 각각의 셀에 서로 상이한 논리적 레벨 값이 저장되어 있음을 보여주고 있다. 각 셀이 저장하고 있는 논리적 레벨 값은 노드 분할의 기준이 되는 동적 레벨 그리드의 레벨이 된다.

3.4 알고리즘

본 절에서는 DGR-Tree의 삽입, 삭제, 윈도우 검색 알고리즘에 대해 기술한다.

3.4.1 삽입 알고리즘

Algorithm : Insert(Point *point*, **OID** *oid*, **DYNAMICGRID** *dg*, **DGRTREE** *dt*)

```
Begin
1.  CELL cell ← FindCell( point, dg )
2.  NODE node ← cell.node;
3.  IF( node == NULL )
4.    IF( cell.level <> dg.level )
5.      node ← find node which has same logical level
and contain point
6.    END IF
7.    IF( node == NULL )
8.      node ← make new node
9.      insert node into dt
10.   END IF
11. END IF
12. IF ( node.count == LORDER )
13.   IF( cell.level == dg.level )
14.     gridExpand( dg )
15.   END IF
16.   split( node, cell, point, oid )
17. END IF
18. insert data{ point, oid } into node
End
```

그림 8. 삽입 알고리즘

그림 8에서 보인 바와 같이 삽입 알고리즘은 먼저 FindCell(*point, dg*) 함수를 호출하여 동적 레벨 그리드를 통해 삽입하려는 POI의 좌표를 포함하고 있는 대상 셀을 검색한 후, 대상 셀이 DGR-Tree의 리프 노드를 가리키는지 검사한다.

검색된 셀이 DGR-Tree의 리프 노드를 가리킬 경우 데이타를 삽입할 노드로 지정하고, 검색된 셀이 리프 노드를 가리키지 않을 경우 대상 셀의 논리적 레벨 정보와 동적 레벨 그리드의 물리적 레벨 정보를 비교한다. 대상 셀의 논리적 레벨 정보와 물리적 레벨 정보가 같지 않은 경우, 논리적 레벨 상의 셀을 찾는다. 논리적 레벨 상의 셀과 연관된 DGR-Tree 노드를 찾은 경우, 해당 노드를 데이타를 삽입할 노드로 지정한다. 찾지 못한 경우, DGR-Tree에 해당 셀과 연관될 새로운 노드를 생성하고 이를 데이타를 삽입할 노드로 지정한다.

이때 리프 노드의 엔트리 개수가 기 정의된 리프 노드의 최대 엔트리 개수인 LORDER와 같지 않다면 해당 노드에 POI를 삽입한다. 만일 리프 노드의 엔트리 개수가 LORDER와 같다면 해당 노드가 데이타 삽입을 위해 노드 분할이 필요한 경우로서, 해당 셀의 논리적 레벨과 동적 레벨 그리드의 레벨을 비교한다. 만약 셀의 논리적 레벨이 동적 레벨 그리드의 레벨과 같다면 노드 분할하기 전에 이를 위한 동적 레벨 그리드의 분할이 필요한 경우로써 gridExpand(*dg*) 함수를 호출하여 그리드 확장을 수행하고, split(*node, cell, point, oid*) 함수를 호출해 노드 분할을 수행한다. 마지막으로 노드 분할이 수행된 후, POI를 해당 노드에 삽입한다.

3.4.2 삭제 알고리즘

Algorithm : Delete(Point *point*, OID *oid*, DYNAMICGRID *dg*, DGRTREE *dt*)

```
Begin
1.  CELL cell ← FindCell( point )
2.  NODE node ← cell.node;
3.  IF( node == NULL )
4.    RETURN
5.  END IF
6.  FOREACH LENTRY entry in node.entires
7.    IF( (entry.point == point) && (entry.oid == oid) )
8.      IF( node.count == 1 )
9.        cell.node ← NULL
10.       DeleteNode( node, dt )
11.     ELSE
12.       delete data{point, oid} from node
13.     END IF
14.   END IF
15. END FOR
End
```

그림 9. 삭제 알고리즘

그림 9에 보인 바와 같이 삭제 알고리즘은 먼저 FindCell(*point*) 함수를 이용하여 삭제하고자 하는 POI의 위치에 해당하는 셀을 검색한 후 검색한 셀에 연결된 리

프 노드의 존재 여부를 판단한다. 만일 노드가 없다면 종료한다. 검색한 셀에 연결된 리프 노드가 있는 경우, 리프 노드의 엔트리 중 포인트 및 OID 정보와 삭제하고자 하는 POI가 일치하는 엔트리가 존재하는지 검사한다. 삭제할 엔트리가 존재하지 않는 경우 종료한다. 삭제할 엔트리가 존재하는 경우, 리프 노드의 엔트리가 두 개 이상이면 해당 엔트리만 삭제한다. 그러나 만약 리프 노드 엔트리 개수가 한 개라면 해당 노드를 삭제해야 된다. 해당 노드 삭제는 대상 셀의 노드 연결 정보를 NULL로 초기화하고 노드 삭제를 위한 DeleteNode(*node, dt*) 함수를 이용하여 노드를 삭제한다.

3.4.3 윈도우 검색 알고리즘

Algorithm : RangeSearch(MBB *mbb,* DYNAMICGRID *dg*)

```
Begin
1.  RESULT result ← ∅
2.  CELL celllist[] ← find set of cell which contain mbb
in dg
3.  FOREACH CELL cell IN celllist
4.    IF( mbb overlap cell.mbb )
5.      IF( mbb contain cell.mbb )
6.        add entry of cell.node to result
7.      ELSE
8.        FOREACH LENTRY entry IN cell.node
10.         IF( mbb contain entry.point )
11.           add entry to result
12.       END IF
13.     END FOR
14.   END IF
15.  END IF
16.  END FOR
17.  RETURN result
End
```

그림 10. 윈도우 검색 알고리즘

그림 10의 윈도우 검색 알고리즘에서는 우선 동적 레벨 그리드로부터 질의 윈도우를 포함하는 셀들의 집합 celllist를 얻어와 각 셀의 mbb와 질의 윈도우의 겹침을 비교한다. 만일 각 셀의 mbb와 질의 윈도우가 겹칠 경우 질의 윈도우가 각 셀의 mbb를 포함하고 있으면 셀이 가리키는 리프 노드의 모든 엔트리들을 검색 결과에 추가한다. 각 셀의 mbb를 포함하지 않고 겹치는 경우 셀과 연결된 리프 노드의 모든 엔트리 위치, 즉 point와 질의 윈도우를 비교하여 질의 윈도우가 point를 포함하고 있으면 해당 엔트리를 결과에 추가한다. 더 이상 검색할 추가 엔트리가 없거나 겹침이 없을 경우 검색 결과를 반환한다.

4. DGR-Tree를 위한 KNN 검색 알고리즘

본 장에서는 DGR-Tree를 위한 KNN 검색 알고리즘의 자료 구조 및 세부 알고리즘에 대해 기술한다.

4.1 KNN 검색 알고리즘의 자료 구조

KNN 검색 알고리즘을 수행하기 위해서는 후보 노드를 관리하는 후보 리스트, 검색이 완료된 노드를 저장하는 검색 완료 리스트, 검색 결과를 저장하는 결과 리스트 등이 필요하다.

minimum distance	offset	entry number	entries	pointer of next node

그림 11. 후보 노드의 자료 구조

그림 11에서 보인 바와 같이 후보 리스트는 후보 노드의 링크드 리스트(Linked-List)로 구성되며, 각각의 후보 노드는 질의점과의 최소 거리, 다음에 비교할 엔트리의 위치를 나타내는 옵셋, 후보 노드의 엔트리 개수와 엔트리, 다음 후보 노드의 포인터로 구성된다. 또한 후보 리스트는 후보 노드의 최소 거리를 기준으로 정렬된다. 후보 노드의 최소 거리는 질의점과 후보 노드 MBB 간의 최단 거리로서, 질의점이 해당 객체에 포함되는 경우 최단 거리는 0이 된다.

후보 리스트는 질의점이 위치한 셀과 연결된 DGR-Tree의 리프 노드가 있는 경우 해당 리프 노드를 후보 노드로서 후보 리스트에 추가한다. 현재의 후보 리스트에서 KNN 검색을 완료하지 못한 경우, 질의점이 위치한 셀에 인접된 최대 8개까지의 셀을 검색한다. 이때 검색된 각각의 셀과 연결된 DGR-Tree 리프 노드가 있는 경우 해당 리프 노드를 후보 노드로서 후보 리스트에 추가한다. 만약 8개까지 셀을 확장한 후에도 현재의 후보 리스트에서 KNN 검색을 완료하지 못한 경우, DGR-Tree의 루트 노드를 후보 리스트에 추가한다.

검색 완료 리스트는 검색 완료된 노드를 추후 검색하지 않기 위한 자료 구조로서, 검색 완료된 노드 정보를 저장하는 배열 형태로 구성된다. 결과 리스트는 KNN 검색의 결과를 저장하기 위한 자료 구조로서 KNN 검색 시 사용자가 요청하는 K개만큼의 결과 엔트리의 배열로 구성된다. 그림 12는 결과 리스트에 저장되는 결과 엔트리의 자료 구조로서 해당 POI와 질의점과의 최소 거리와 OID로 구성된다.

minimum distance	OID

그림 12. 결과 엔트리의 자료 구조

DGR-Tree를 위한 KNN 검색 알고리즘에서는 노드 내에 있는 질의점으로부터의 엔트리의 최소 거리가 다른 노드에 있는 엔트리의 최소 거리보다 큼에도 불구하고 근접 객체로 결정되는 오류를 방지하기 위해서 NEB를 저장한다. NEB는 노드 검색 시 확장을 고려하지 않아도 되는 최소 거리로서 그림 13은 NEB의 예를 보여준다.

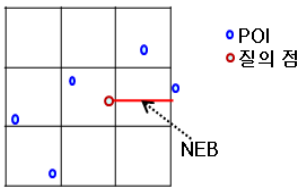

그림 13. NEB의 예

NEB는 노드 검색 시 확장을 고려하지 않아도 되는 최소 거리이며 루트 노드를 검색한 이후에는 불필요하다.

4.2 KNN 검색 알고리즘

Algorithm : KnnSearch(POINT *qp*, INT *k*, DYNAMICGRID *dg*, DGRTREE *dt*)

```
Begin
1.  RESULT result ← Ø
2.  DONELIST dl ← Ø
3.  CANDIDATELIST clist ← Ø
4.  CELL cell ← FindCell( qp, dg )
5.  IF( cell.node <> NULL )
6.      Insert( cell.node, qp, clist, dl )
7.      Search( qp, clist, dl, result )
8.      IF( number of result == k )
9.          RETURN result
10.     END IF
11. END IF
12. CELL celllist[] ← find set of cell which adjacent cell
                      of cell
13. FOREACH cell IN celllist
14.     Insert( cell.node, qp, clist, dl )
15. END FOR
16. Search( qp, clist, dl, result )
17. IF( number of result == k )
18.     RETURN result
19. END IF
20. Insert( dt.root, qp, clist, dl )
21. Search( qp, clist, dl, result )
23. RETURN result
End
```

그림 14. KNN 검색 알고리즘

그림 14에서 보인 바와 같이 먼저 질의 결과를 저장하기 위해 result, 후보 리스트와 검색 완료 리스트를 저장하기 위한 후보 리스트 객체 clist, 검색 완료 리스트 객체 dl을 선언한다. 다음 FindCell(*qp, dg*) 함수를 호출하여 질의점이 위치한 곳의 셀을 찾아 후보 리스트에 노드를 삽입하기 위해 Insert(*node, qp, clist, dl*) 함수를 호출한다. 그리고 Search(*qp, clist, dl, result*) 함수를 이용하여 현재까지 후보 리스트에서 입력된 노드 데이타를 기반으로 실제 KNN 검색을 수행하게 된다.

KNN 검색을 수행한 후 사용자가 원하던 K개의 데이타를 얻은 경우 결과값을 반환하나, 사용자가 원하던 K개의 데이타를 얻지 못한 경우 기존에 찾은 셀의 주변 셀을 탐색하여 앞의 과정을 반복한다. 사용자가 원하던 K개의 데이타를 여전히 얻지 못한 경우 DGR-Tree의 루트 노드를 후보 리스트에 입력한 후 전체 트리를 검색하게 된다.

5. 성능평가

본 논문에서는 Windows XP 환경에서 DGR-Tree와 비교 대상으로서 LBS에서 많이 사용되고 있는 R-Tree 및 LBS를 위해 개발된 iQR-Tree를 동일 환경에서 구현하였다. 본 논문에서 실험 결과의 당위성을 위해 실험 데이타로서 균등 분포를 가지는 가상 POI, 편향 분포를 가지는 서울시에 존재하는 9만개의 음식점 POI, 그리고 편향 분포를 가지는 서울시에 존재하는 40만개의 산업체 POI를 사용하였다.

5.1 삽입 성능

그림 15, 그림 16, 그림 17은 POI 삽입 실험 결과로서 입력되는 POI 개수에 따른 노드 접근 횟수를 보여준다.

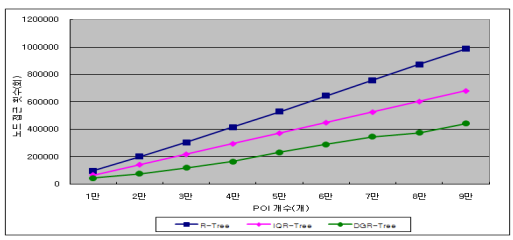

그림 15. 가상 POI(균등 분포)

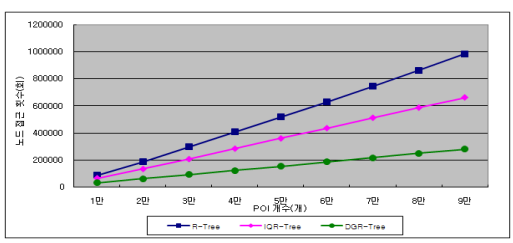

그림 16. 서울시 음식점 POI(편향 분포)

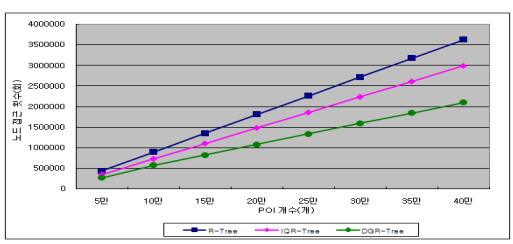

그림 17. 서울시 산업체 POI(편향 분포)

위 그림에서 보인 바와 같이 모든 경우에서 DGR-Tree의 성능이 가장 우수하고, 삽입할 POI 개수가 증가함에

따라 DGR-Tree와 iQR-Tree의 성능 격차 비율이 점차 커지는 것으로 나타났다.

5.2 검색 성능

5.2.1 윈도우 검색 성능

그림 18, 그림 19, 그림 20은 윈도우 크기를 $100m^2$에서 $1000m^2$까지 $100m^2$씩 증가시키면서 윈도우 검색 동안 노드 접근 횟수를 보여준다. 이때, 실험의 정확성을 위해 윈도우의 위치를 변경하면서 100번씩 수행한 평균 결과를 사용하였다.

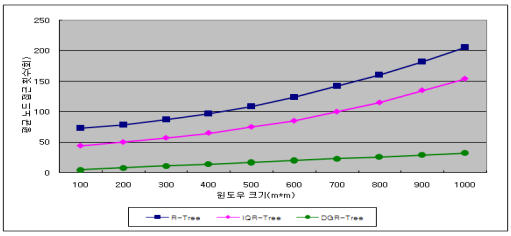

그림 18. 가상 POI(균등 분포)

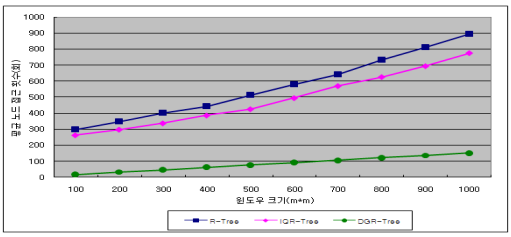

그림 19. 서울시 음식점 POI(편향 분포)

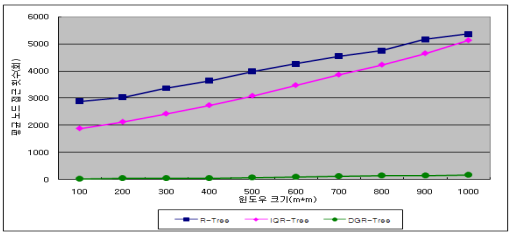

그림 20. 서울시 산업체 POI(편향 분포)

위 그림에서 보인 바와 같이 윈도우 크기에 관계없이 모든 경우에서 DGR-Tree의 성능이 가장 우수한 것으로 나타났다. 또한 윈도우 크기가 증가할수록 성능 격차 비율이 커지는 것으로 나타났다.

5.2.2 KNN 검색 성능

그림 21, 그림 22, 그림 23은 K를 10에서 100까지 10개씩 증가시키면서 KNN 검색하는 동안 평균 노드 접근 횟수를 보여준다.

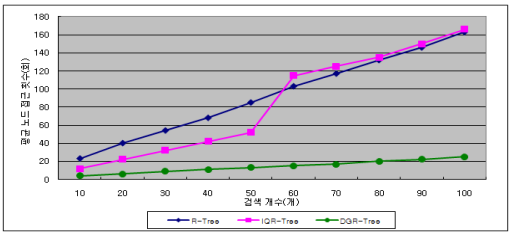

그림 21. 가상 POI(균등 분포)

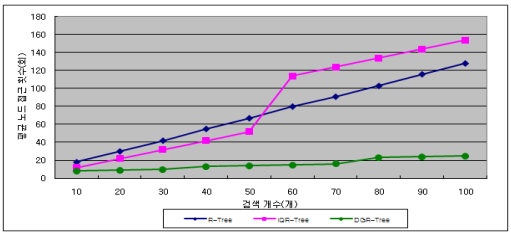

그림 22. 서울시 음식점 POI(편향 분포)

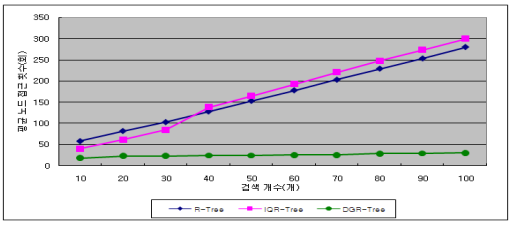

그림 23. 서울시 산업체 POI(편향 분포)

위 그림에서 처럼 모든 경우에서 검색 개수가 증가할수록 DGR-Tree의 노드 접근 횟수는 약간의 소폭 증가에 그치는 반면 iQR-Tree와 R-Tree에서는 현저히 증가하고 있다. 이처럼, DGR-Tree의 KNN 검색 성능이 다른 인덱스에 비해 월등히 좋은 이유는 동적 레벨 그리드를 통해 트리 검색 없이 데이타가 존재하는 리프 노드에 직접 접근할 수 있기 때문이며, 후보 객체가 있을 가능성이 가장 높은 인접 셀을 먼저 검색하기 때문이다.

5.3 삭제 성능

그림 24, 그림 25, 그림 26은 POI를 각각 10개에서 100개까지 삭제하는 동안의 노드 접근 횟수를 보여준다.

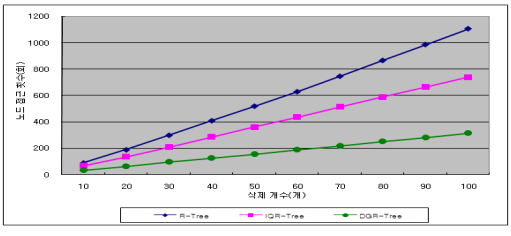

그림 24. 가상 POI(균등 분포)

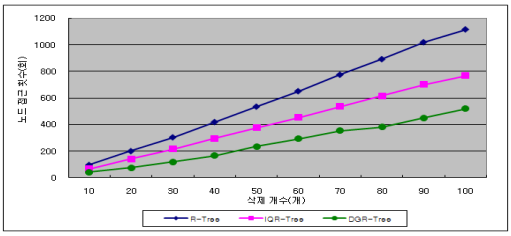

그림 25. 서울시 음식점 POI(편향 분포)

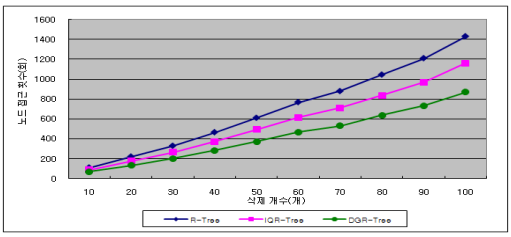

그림 26. 서울시 산업체 POI(편향 분포)

위 그림에서 보인 바와 같이 POI 삭제 시 노드 접근 횟수 실험 결과, 모든 경우에서 DGR-Tree의 성능이 가장 우수하게 나타났다. 그리고 공간 데이타 개수가 증가하여도 iQR-Tree 및 R-Tree와의 성능 격차 비율은 크게 변하지 않는 것으로 나타났으나, 편향 분포 POI의 경우 균등 분포 POI에 비해 성능 격차 비율이 감소하는 것으로 나타났다.

6. 결 론

본 논문에서는 u-LBS에서 대용량화 및 밀집되는 경향을 보이는 정적 POI의 효율적인 검색을 위한 DGR-Tree와 DGR-Tree를 위한 KNN 검색 처리 기법을 제시하였다. DGR-Tree는 점 데이타에 적합하도록 최적화하고 있으며 리프 노드 간 겹침 문제를 해결하였다. 보조 인덱스로서 사용되는 동적 레벨 그리드는 DGR-Tree의 리프 노드로 직접 접근하도록 함으로써 인덱스 접근 성능을 향상시켰다.

DGR-Tree를 위한 KNN 검색 알고리즘은 동적 레벨 그리드를 활용하여 질의점에 존재하는 셀 또는 인접 셀을 통해 검색 후보 노드에 직접 접근하여 빠른 응답이 가능하다. 또한 정렬에 따른 오버헤드 및 셀 검색으로 인한 오버헤드를 최소화하고 있다. 또한 실험을 통하여 DGR-Tree의 삽입, 삭제, 윈도우 검색, KNN 검색 성능이 우수함을 입증하였다. DGR-Tree는 윈도우 검색 및 KNN 검색에서 특히 뛰어난 성능을 가지고 있다. 그러나 메모리 사용이 약간 크다는 문제가 발생하였다.

본 논문에서 제시한 DGR-Tree는 유비쿼터스 컴퓨팅 환경에서 위치를 기반으로 사용자의 상황에 맞는 서비스를 제공하는 분야에서 실제 유용하게 활용될 수 있다. 향후 발전 방향으로서 동적 POI를 대상으로 하는 추가 연구가 필요할 것이라 예상된다.

참 고 문 헌

[1] Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B., “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles”, Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 1990, pp.323-331.

[2] Bentley, J. L., “Multidimensional Binary Search Trees Used for Associative Searching”, Communications of the ACM, Vol.18, No.9, 1975, pp.509-517.

[3] Bo, H., and Qiang, W., “A Spatial Indexing Approach for High Performance Location Based Services”, The Journal of Navigation, Vol.60, No.1, 2007, pp.83-93.

[4] Chern, H. H., and Hwang, H. K., “Partial Match Queries in Random Quadtrees”, SIAM Journal on Computing, Vol.35, No.6, 2003, pp.904-915.

[5] Frentzos, E., Gratsias, K., and Theodoridis, Y., “Towards the Next Generation of Location-based Services”, Proc. of the Intl. Workshop on Web and Wireless Geographical Information Systems, 2007, pp.202-215.

[6] Hjaltason, G. R., and Samet, H., “Distance Browsing in Spatial Databases”, ACM Transactions on Database Systems, Vol.24, No.2, 1999, pp.265-318.

[7] Manolopoulos, Y., Nardelli, E., Papadopoulos, A., and Proietti, G., “QR-Tree: A Hybrid Spatial Data Structure”, Proc. of the Intl. Conf. on Geographic Information Systems in Urban, Regional and Environmental Planning, 1996, pp.3-7.

[8] Robinson, J. T., “The K-D-B-Tree: A Search Structure for Large Multi-dimensional Dynamic Indexes”, Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 1981, pp.10-18.

[9] Roussopoulos, Nick., Kelley, S., and Vincent, F., “Nearest Neighbor Queries”, Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 1995, pp.71-79.

[10] Sellis, T. K., Roussopoulos, N., and Faloutsos, C., “The R+-Tree: A Dynamic Index for Multi-dimensional Objects”, Proc. of the Intl. Conf. on VLDB, 1987, pp.507-518.

[11] Wang, W., Yang, J., and Munts, R., “PK-Tree: A Spatial Index Structure for High Dimensional Point Data”, Proc. of the Intl. Conf. on Foundations of Data Organization and Algorithms, 1998, pp.27-36.