현재 및 미래 위치 처리를 위한 TPR-tree의 점진적 재구성 기법

박동윤○ 김동현 홍봉희
부산대학교
{parkdy01○, pusrover, bhhong}@pusan.ac.kr

Incremental reorganization Policy of TPR-tree for Querying Predictive Positions

Dongyoun Park○ Donghyun Kim Bonghee Hong
Dept. of Computer Engineering. Pusan National University

요 약

TPR-tree는 이동체의 위치 데이터에 대해 현재 및 미래 위치 질의를 지원하기 위하여 시간 함수 기반의 경계사각형(Bounding Rectangle)으로 이동체를 색인한다. 경계사각형의 각 축은 가장 빠른 속도로 이동하는 이동체의 속도 값을 이용하여 시간에 따라 확장한다. 경계사각형 영역의 확장으로 중복(overlap)이 심화되고 사장영역(dead space)이 커지는 문제가 있다. 따라서 시간이 지날수록 영역질의 시 성능이 떨어진다.

이 논문에서는 시간이 지남에 따라 발생하는 노드 간의 심한 중복과 사장영역을 줄이기 위해 중복이 심한 두 개의 단말노드를 강제 합병하고 재분할하는 강제 합병 정책과 이동체의 삭제가 발생한 노드의 모든 이동체들을 강제적으로 재삽입하는 삭제노드 강제 재삽입 정책과 삭제가 발생한 노드와 중복되는 노드들의 이동체들을 강제적으로 재삽입하는 중복 노드 강제 재삽입 정책을 이용한다. 강제 합병 정책과 삭제 노드 강제 재삽입 정책, 그리고 중복 노드 강제 재삽입은 TPR-tree의 구조를 점진적으로 재구성하기 때문에 이동체의 현재 분포를 고려하여 색인 구조를 동적으로 개선하는 장점을 가진다.

1. 서 론

최근 위치 파악 및 추적 서비스 분야의 시스템 개발이 활발하게 진행되어지고 있다. 대표적인 예로 친구 찾기, 네비게이션등의 서비스 등을 예로 들 수 있다. 이러한 시스템 응용을 위하여 이동체의 위치 정보를 관리하고 현재 위치 질의를 효과적으로 처리하는 방법이 필요하다. 예를 들어 다음의 질의를 신속히 처리해야 한다. " 현재부터 10분 이내에 시청 앞을 지나갈 차량을 검색하라" .

실세계에서 시간에 따라 이동하는 모든 물체는 이동체로 모델화될 수 있다. 이동체는 시간에 따라 객체의 위치 및 모양이 연속적으로 변경되는 데이터이다. 이동체의 현재 및 미래 위치에 대한 시공간 질의를 지원하는 R*-tree를 기반으로 한 TPR-tree[1]가 제안되었다. TPR-tree는 속도와 방향을 R*-tree[2]에 적용한 시간에 대한 함수의 매개 변수로 사용하여, 이동체의 현재와 가까운 미래의 위치를 좀 더 빠르게 검색 할 수 있게 하는 방법을 제시하였다. 경계사각형(BR)은 노드의 모든 엔트리들을 포함하는 시간 함수 기반의 경계 사각형을 말한다. BR의 각 축은 내부 이동체들에 의해 이동 방향에 따라 가장 빠른 속도로 이동하는 이동체의 속도 값으로 증가한다. TPR-tree는 BR의 형태로써 위치 데이터를 모델링 함으로써 시간에 흐름에 따라 BR이 커지는 문제점이 있다.

TPR*-tree[3]는 TPR-tree의 향상된 삽입과 삭제 알고리즘을 제시하고 있다. 이동체의 새로운 위치 보고로 인한 갱신 비용을 감소하는 방법들을 제안하고 있다. TPR-tree의 개선된 BR 최적화(tightening) 방법을 제안하고 있으나 주변 노드와의 시간이 지남에 따른 중

복 증가 문제는 여전히 존재한다.

이 논문에서는 노드 간의 심한 중복과 사장영역의 감소를 위해 두 가지 정책을 제시한다. 첫 번째 정책은 강제 합병 정책이다. 이 정책에서 이동체가 삽입될 때 만약 삽입된 노드의 이웃 노드 간의 중복이 심하면 두 노드를 강제 합병하고 재분할한다. 두 번째 정책으로 삭제 노드 강제 재삽입 정책을 제시한다. 하나의 이동체를 삭제할 때 삭제가 발생한 노드의 모든 이동체를 강제 재삽입한다. 세 번째 정책은 중복 노드 강제 재삽입 정책이다. 삭제가 발생한 노드와 중복되는 노드의 이동체를 강제 재삽입한다. 위의 정책들은 TPR-tree의 구조를 점진적 재구성함으로써 이동체의 분포를 고려하여 동적으로 색인을 개선하는 장점을 가진다.

이 논문의 구성은 다음과 같다. 먼저 2장에서는 관련 연구를 그리고 3장에서 문제정의를 기술한다. 4장에서는 강제 합병 정책을 기술하고 5장에서는 삭제 노드 강제 재삽입 정책과 중복 노드 강제 재삽입 정책에 대하여 기술한다. 그리고 6장에서 결론 및 향후 연구를 기술한다.

2. 관련연구

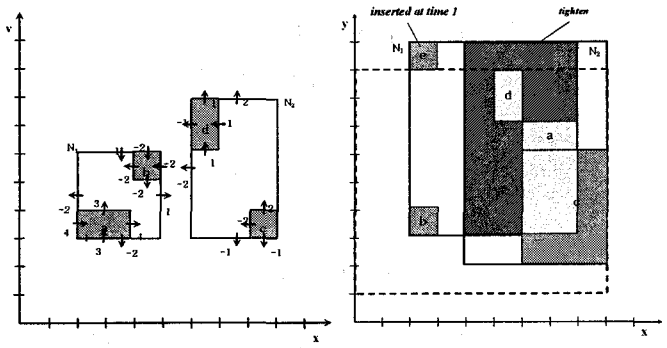
이동체의 현재 및 미래 위치 질의에 대한 색인에 관한 연구로 크게 공간-분할 방법[7,8]과 데이터-분할 방법[1,3]이 있다. 이동체의 분포는 시간에 따라 변하기 때문에 공간-분할 방법은 적합하지 않다. 또한 시간 도메인은 동적으로 증가하기 때문에 시간 도메인을 관리하기가 어렵다는 문제가 있다. 따라서 데이터-분할 방법을 사용한다. 데이터-분할 방법은 TPR-tree와 TPR*-tree가 있다.

TPR-tree[1]는 속도와 방향을 R*-tree에 적용한 시간에 대한 함수의 매개 변수로 사용하여, 이동체의 현재 및 미래 위치의 검색 방법을 제시하였다. TPR-tree는 이동체의 미래 위치 질의를 지원하기 위해서 BR의 형태로 위치 데이터를 모델링 한다. BR은 노드의 모든 엔트리들을 포함하는 시간 함수 기반의 경계 사각형을 말한다. BR의 각 축은 내부 이동체들에 의해 이동 방향에 따라 가장 빠른 속도로 이동하는 이동체의 속도 값으로 증가한다. 따라서 내부 이동체의 공간 위치 보다 BR의 영역이 시간의 경과에 따라 점차적으로 더 커지는 현상이 발생한다. BR 영역이 확장 되고, 노드 간의 중복과 사장영역이 커짐으로써 시간이 경과 함에 따라 영역질의의 성능은 점차적으로 나빠지게 된다. 따라서 중복과 사장영역의 감소가 필요하다.

TPR*-tree[3]는 TPR-tree의 향상된 삽입과 삭제 알고리즘을 제시하고 갱신 비용의 감소를 고려한 색인 구조를 제안하고 있다. 기존의 TPR-tree에서는 중복영역에 이동체 삽입이 발생할 경우 BR의 크기가 적은 노드로 탐색하게 된다. 이러한 방법은 탐색된 하위노드의 BR크기가 더 커지는 문제점이 발생한다. TPR*-tree에서는 삽입할 단말노드 탐색 시 탐색된 노드의 하위 노드를 고려함으로써 BR의 크기가 더 적게 커지는 노드로 탐색하는 방법을 제시하고 있다. 기존의 재삽입 정책의 적용으로 인한 문제점은 이동체의 이동성을 고려하지 않고 현재의 위치정보만으로 재삽입 될 이동체를 선택한다.예를 들어 오버플로우 발생 시 BR 내부의 다른 이동체들은 중심점으로 이동하고, 한 이동체가 바깥 방향의 이동성을 가지고 있고 BR 중심점에 있을 경우 재삽입 이동체로 선택 되지 않는다. 이러한 문제점을 삽입 시점의 이후 시간을 고려한 재삽입 이동체 선택 방법을 제안하고 있다. 삭제 시 TPR-tree의 삭제가 발생 노드의 부모 노드 BR만 최적화되는 방법을 추가 I/O 없이 탐색된 주변 노드의 BR까지 최적화하는 방법을 제시하고 있다. 이러한 방법들은 갱신 비용의 감소 효과를 가진다.

3. 문제 정의

. 이동체의 현재 및 미래 위치 질의에 응답하기 위해서 질의 시간(t_q)을 매개변수로 사용하여 BR을 확장하게 된다. BR의 각 축은 가장 빠른 속도로 이동하는 이동체의 속도 값을 이용하여 시간에 따라 확장하게 된다. 시간의 증가로 인하여 BR의 영역은 점차적으로 커지게 된다. BR의 시간에 따른 증가는 노드 간의 중복(overlap)과 사장영역(dead space)의 크기를 커지게 만든다. 따라서 시간이 지날수록 영역질의의 성능은 점차적으로 나빠지는 결과를 가지게 된다.



(a) t_0 시간의 BR (b) t_1 시간의 BR

그림 1 BR 최적화의 문제점

그림 1은 t_0 시간의 상태에서 t_1 시간에 삽입 발생으로 인한 N_1 노드의 BR 최적화를 나타내고 있다. 미래의 시간인 t_1 시간에 e 이동체의 삽입이 발생 했을 때 탐색되는 각 노드의 BR들을 확장된다. 그림 1의 (b)에서 N_1, N_2 노드의 BR이 확장되는 것을 볼 수 있다. e 이동체가 N_1에 삽입될 때 N_1 노드의 BR은 최적화 된다. 그림 1의 (b)에서 N_1은 e 이동체의 삽입 발생으로 N_1 노드의 BR이 확장된다. 점선으로 표시된 박스가 확장된 형태이다.

TPR-tree에서의 BR 최적화 기법은 삽입과 삭제가 발생한 노드의 부모노드 BR을 최적화한다. 그러나 TPR-tree에서 BR을 최적화 해도 중복이 발생하게 된다. 또한 삽입과 삭제가 발생한 노드의 BR만 최적화되고 주변 노드의 BR 최적화에는 영향을 주지 못한다. 그림1의 (b)에서 BR 최적화의 문제점을 보이고 있다. N_1에 e 이동체의 삽입으로 N_1의 BR은 최적화된다. 그러나 N_1 노드의 BR 최적화에도 불구하고 N_1과 N_2 사이의 중복은 여전히 크다. 주변 노드 N_2의 BR 또한 전혀 변화 없다.

4. 강제 합병 정책

BR은 시간의 증가에 따라 커지게 되고 이동체의 삽입은 삽입되는 노드의 BR의 확장을 유발함으로써 단말 노드 간의 심한 중복 현상이 발생 가능하다. 강제 합병 정책은 가장 중복이 심하게 발생하는 두 단말 노드 간의 강제 합병으로 심한 중복을 제거한다. 만약 강제 합병한 노드에서 오버플로우가 발생 한다면 재분할을 수행한다. 단말 노드 간의 중복 비율(Overlap Rate)의 계산식은 다음과 같다.

$$OverlapRate(E_i, E_j) = area(E_i.BR \cap E_j.BR) / \min(area(E_i.BR), area(E_j.BR))$$

LOR(Large Overlap Rage)은 다음과 같이 정의 한다.

2개의 단말 노드 간의 중복 비율로써 심한 중복의 경우를 말한다.

LOR은 30%,40%,50% 등으로 실험하여 성능이 우수한 값을 실험을 통하여 찾아낸다.

강제 합병의 시기는 중복 비율이 LOR보다 클 경우에 강제 합병한다. 강제 합병 시기는 다음과 같다.

삽입 발생 단말 노드 : $(BR(E_i))$

$BR(E_i)$와 중복이 최대인 단말 노드 : $(BR(E_j))$

$OverlapRate((BR(E_i), BR(E_j)) > LOR$

그림 2는 t_0 시간의 BR 상태를 보여주고 있다. 그림 3의 (a)는 t_1 시간의 강제 합병을 보여주고 있다. t_1 시간의 N_1과 N_2 노드의 BR은 확장된다. 이동체 f는 N_1에 삽입되고 BR이 최적화된다. 삽입되고 BR의 최적화된 상태의 $BR(E_i)$과 $BR(E_j)$는 심한 중복이 발생 하였다. 심한 중복이 발생한 $BR(E_i)$과 $BR(E_j)$를 강제 합병한다. 노드의 팬아웃(fan-out)이 4라고 하면 강제 합병한 결과 $BR(E_m)$은 오버플로우를 발생 시킨다. 따라서 재분할을 수행하게 된다. 그림 3의 (b)는 강제 합병 후 오버플로우로 인한 재분할 결과를 보여주고 있다. 강제 합병은 심한 중복이 발생한 두 단말 노드간의 중복을 제거하고 주변 노드의 BR을 최적화 시키기 때문에 영역질의의 성능을 향상시킨다. 또한 강제 합병 후 재분할은 이동체의 분포에 상관없이 트리 구조를 동적으로 만드는 장점이 있다. 따라서 이동체의 삽입 순서에 종속적인 문제점을 해결한다.

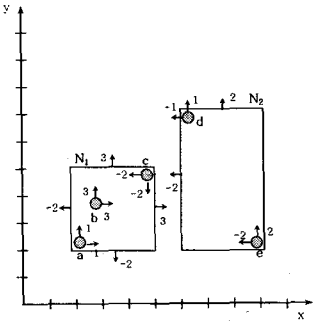
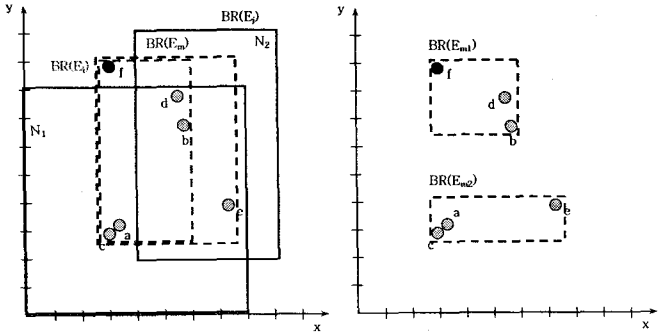


그림 2 t_0 시간의 BR



(a) t_1 시간의 강제 합병 (b) 강제 합병 후 재분할

그림 3 강제 합병 후 재분할

강제 합병 정책 알고리즘은 다음과 같다.

```
Let Ei be tightened the best leaf node to insert new vector of point object
Algorithm Check_Large_Overlap(Ei)
Find all the entries E1, E2, ... En that were overlapped with Ei.BR
Choose Ej ← maximum(OverlapRate(Ei, Ej), for 1 <= i <= n
IF(Ej exists and is largely overlapped over LOR) {
        Merge two node(Ei, Ej) to Em, Em is tightened
        IF Em is overflow {
                Split Em to Em1, Em2;
                Ei ← Em1; Ej ← Em2;
                AdjustTree(Ej);
        }
        ELSE {
                DeleteNode(Ej); Ei ← Em;
        }
```

그림 4 강제 합병 정책 알고리즘

그림 4는 이동체의 삽입으로 LOR을 검사하는 알고리즘이다. 삽입으로 중복 비율(overlap rate)이 최대인 형제 노드가 LOR값 이상이면 강제 합병을 수행한다. 강제 합병 후 오버플로우 발생 시 재분할을 수행한다.

5. 강제 재삽입 정책

TPR-tree는 시간의 증가에 따라 BR이 커지게 되므로 BR들간의 중복과 사장영역이 커지게 된다. 따라서 BR의 크기를 최적화 할 필요가 있다. 삭제가 발생 할 경우 삭제가 발생한 노드 이동체들의 강제 재삽입으로 BR의 최적화와 트리 구조를 점진적 재구성함으로써 공간 활용도를 높이고 영역질의의 성능을 높이게 된다.

강제 재삽입 정책은 다음의 두 가지로 나눌 수 있다.

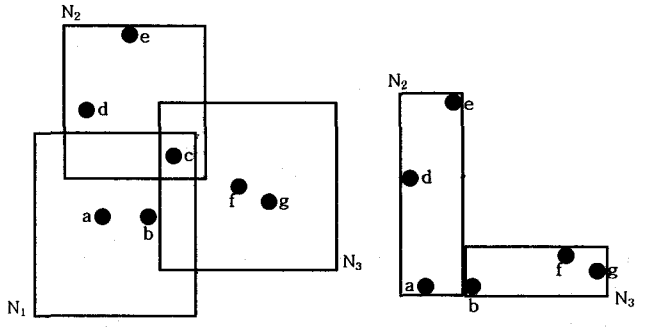
A. 삭제가 발생한 노드의 엔트리들을 재삽입한다.

B. 삭제가 발생한 노드와 중복되는 모든 노드의 엔트리들을 재삽입한다.

두 방법의 실험을 통하여 강제 재삽입에 대한 비용을 측정하고 비용에 대한 원인 분석으로 보다 효과적인 방법을 찾아 낸다.

5.1 삭제 노드 강제 재삽입 정책

삭제 발생 노드의 재삽입은 삭제가 발생한 노드의 이동체들을 강제 재삽입 함으로써 주변 노드 BR 최적화의 효과가 있다.



(a) N_1 노드의 삭제 발생 (b) 강제 재삽입

그림 5 삭제 노드의 강제 재삽입

그림 5의 (a)는 이동체 c의 새로운 위치 보고로 인해 이전 위치의 삭제가 발생 하였다. 삭제 발생 노드의 강제 재삽입 정책에 의해 노드 N_1의 이동체들은 강제 재삽입이 발생하게 된다. 따라서 N_1의 이동체 a와 b는 재삽입 된다. 그림 5의 (b)는 N_1 노드 이동체 c의 삭제 후 이동체들의 재삽입으로 인한 상태를 보여 주고 있다. 노드 N_1의 강제 재삽입은 (b)에서 보는것과 같이 공간활용도를 높이고 주변 노드의 BR의 최적화되고 있다.

다음은 삭제 발생 노드의 강제 재삽입에 대한 알고리즘이다.

```
re-inserted_i ← flase for all levels 1<= i <= h-1 (h is the tree height)
Initialize an empty re-insertion list L_reinsert
E_i is the entry to be deleted
Algorithm Forced_Reinsert(E_i)
N ← FindDeleteNode(E_i, S);   // to find the leaf node N to delete E_i,
                              // Stack S ← retrieves node all the level
DeleteData(N, E_i);
condenseTree(S, L_reinsert);  //Adjust and reinsertion
While(!L_reinsert.empty())
        Insert(L_reinsert);
```

```
Algorithm condenseTree(S, L_reinsert)
IF(isRoot())
        return;
P ←  S;                       // each retrieves the level
L_reinsert ← True;            // all entries of the N
deleteNode(N);
E_j ← find the entry in the parent, that points to this node
IF(P is underflow)
        P→DeleteData(E_j);
        L_reinsert ← True;    // all entries of the P
Else
        AdjustTree(P);        // the entry in 'p' to contain the
                              // new bounding box of this node
P→condeseTree(S, L_reinsert);
```

그림 6 삭제 발생 노드의 강제 재삽입 정책 알고리즘

그림 6은 삭제가 발생 할 때 삭제 발생 노드의 이동체들을 강제 재삽입하는 알고리즘이다. 삭제 발생 노드의 이동체들의 재삽입은 삭제 발생 노드의 공간을 삭제 한다. 노드의 삭제로 상위 노드들의 재조정이 필요하다. condenseTree는 상위 노드의 언더플로우로 발생시 재삽입 될 이동체들을 검사한다.

5.2 중복 노드 강제 재삽입 정책

삭제 발생 노드와 중복되는 노드의 강제 재삽입 방법은 중복 비교를 어디까지 할 것인가에 대한 제약을 가진다. Level Restriction(LR)은 각 트리 레벨의 값으로 설정 가능 하다. LR 값의 설정에 의해 설정된 트리 레벨까지의 중복을 비교한다.

다음은 LR의 값을 level 1로 설정함으로써 단말노드의 형제노드까지 중복 비교하는 강제 재삽입 정책에 대한 예를 보이고 있다. 그림 7은 j이동체의 새로운 위치 보고로 인해 이전 데이터의 삭제가 발생한다. 그림 8은 그림 7의 이동체의 공간적인 위치에 대한 트리 구조를

보이고 있다. 그림 9는 삭제 발생으로 삭제가 발생한 노드와 중복되는 형제 노드 level까지 강제 재삽입의 처리 후의 모습을 보이고 있다. 그림 10은 그림 9에 대한 트리 구조를 나타내고 있다.

아래의 그림에서 보는 것과 같이 삭제 발생 후 삭제 발생 노드와 중복되는 노드의 재삽입으로 인해 BR의 최적화와 노드 간의 중복과 사장영역의 감소한 것을 알 수 있다.

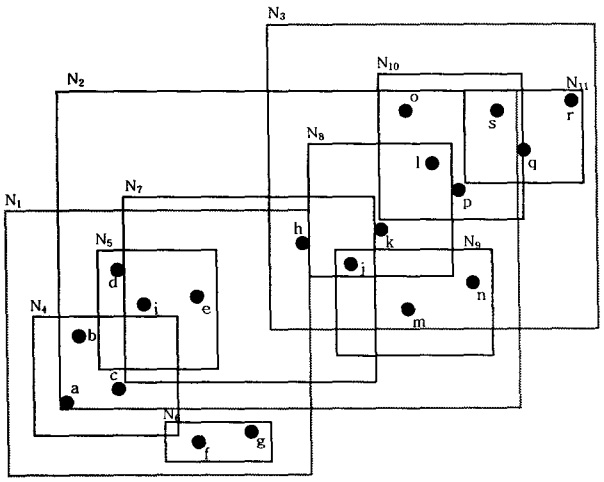


그림 7 N_7 노드내의 j 삭제 발생

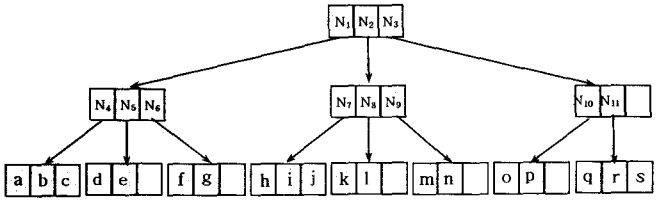


그림 8 j 삭제 발생시의 트리 구조

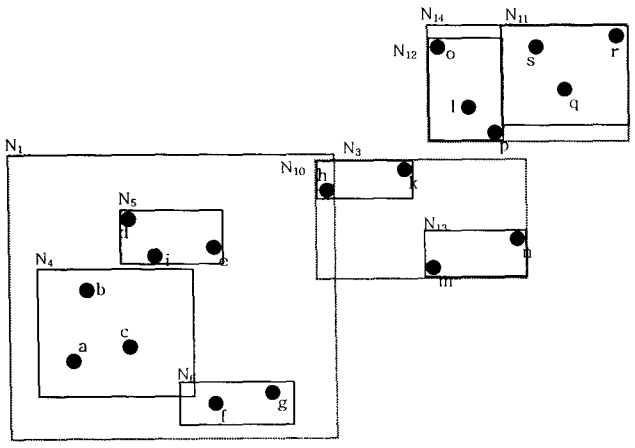


그림 9 j 삭제 발생 후 형제 노드까지 재삽입

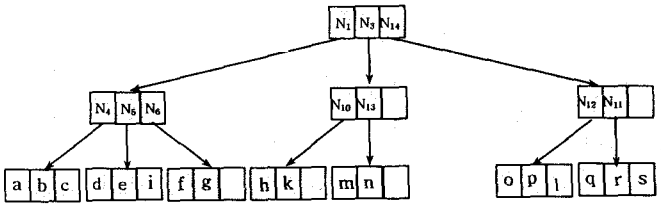


그림 10 형제 노드까지 재삽입 후 트리 구조

중복 노드의 강제 재삽입 정책의 알고리즘은 다음과 같다.

```
re-inserted_i ← flase for all levels 1<= i <= h-1 (h is the tree height)
Initialize an empty re-insertion list L_reinsert
E_i is the entry to be deleted
Algorithm Forced_Reinsert(E_i)
N ← FindDeleteNode(E_i, S);          // to find the leaf node N to delete E_i,
                                     // Stack S ← retrieves node all the leve
DeleteData(N, E_i);
condenseTree(S, L_reinsert);         //Adjust and reinsertion
While(!L_reinsert.empty())
        Insert(L_reinsert);
```

```
Algorithm condenseTree(S, L_reinsert)
IF(isRoot())
        return;
P ←  S;          // point of parent node at each retrieves the level
N ← this;        // point of current node
L_reinsert ← all entris that be deleted the leaf node
E_j ← find the entry in the parent, that points to this node
IF(LS+1 > P.level) {
        For exist each entry of P {
         IF(P != N) {      //LS is overlap check restriction
           O ← Check_Overlap(the node of  be deleted entry, sibling node of P);
           IF(O == True) Lreinsert ←  all entries at sibling node of P
         }
        }
}
IF(P is underflow)
        P→DeleteData(E_j);
        L_reinsert ← all entries of the P;
Else
        AdjustTree(P);     // the entry in 'p' to contain the
                           //new bounding box of this node
P→condeseTree(S, L_reinsert);
```

그림 11 중복 노드의 강제 재삽입 정책 알고리즘

그림 11은 삭제가 발생한 노드와 중복되는 노드의 이동체들을 강제 재삽입하는 알고리즘이다. 삭제 발생

노드와 중복되는 노드들을 검사하여 중복되는 노드의 이동체들을 강제 재삽입한다. 삭제 후 언더플로우 발생시 삭제 발생 노드의 공간을 삭제 한다. 노드의 삭제로 상위 노드들의 재조정이 필요하다. condenseTree는 상위 노드의 언더플로우로 발생시 재삽입 될 이동체들을 검사한다.

6. 결론 및 향후 연구

이 논문에서는 TPR-tree의 구조적인 문제점인 시간에 따른 BR 증가로 인한 노드 간의 중복과 사장영역의 감소 방법을 소개하고 해결 하였다. 강제 합병 정책으로써 강제 합병과 재분할을 통해 중복을 제거하고 사장영역을 감소한다. 강제 재삽입 정책으로써 삭제가 발생할 때 삭제 발생 노드의 이동체들의 강제 재삽입과 삭제 발생 노드와 중복되는 노드의 강제 재삽입으로 BR의 크기를 최적화 하고 트리 구조를 동적으로 재구성 함으로써 공간 활용도와 영역질의 성능 높였다. 강제 합병 정책과 강제 재삽입의 두 정책은 중복과 사장영역의 감소로써 효율적인 영역질의의 성능을 가져 올 것으로 기대된다.

향후 성능 평가를 통해서 강제 합병 정책과 강제 재삽입 정책을 실험하고자 한다. 또한 재삽입으로 인한 삽입 비용의 증가에 대해 비용의 감소 방법에 대한 연구가 필요하다.

7. 참고문헌

[1] S.Saltenis, C. S. Jensen, S.T.Leutenegger, and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects", In Proc. ACM SIGMOD on Management of data, p331-342, 2000.Yannis Theodoridis, Michael Vazirgiannis, Timos Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications", IEEE International Conference on Multimedia Computing and Systems, pp 441-448, 1996.

[2] N.Beckmann and H.P.Kriegel, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles",In Proc.ACM SIGMOC, p332-331,1990.Antonm Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", Proceedings of ACM SIGMOD Conference on the Management of Data, pp. 47-57, 1984.

[3] Y. Tao, D. Papadias, and J. Sun, "The tpr*-tree: an optimized spatiotemporal access method for predictive queries," Proceedings of 29th International Conference on Very Large Data Bases, Berlin, Germany, September 9-12, 2003.

[4] B. Jun, B. Hong, and B. Yu, "Dynamic splitting policies of the adaptive 3DR-tree for indexing continuously moving objects," Int'l Conf. on Databases and Expert Systems Applications, September, 2003.

[5] Z. Song and N. Roussopoulos, "Hashing Moving Object", Intl. Conf. on Mobile Data Management, p161-172, 2001.

[6] J. Tayeb, O. Ulusoy, and O. Wolfson. "A quadtree basesd dynamic attribute indexing method," The Computer Journal, vol. 41, no. 3., pp. 185-200, 1998.

[7] A. Cuttman, "R-trees: A dynamic index structure for spatial searching," In Proc. ACM SIGMOD Int'l Conf. on Management of Data, pp. 47-54, 1984.