Journal of the KIMST, Vol. 22, No. 2, pp. 266-277, 2019
ISSN 1598-9127(print), 2636-0640(online)
DOI http://dx.doi.org/10.9766/KIMST.2019.22.2.266

Research Paper 정보 · 통신 부문

함정전투체계 표적 색인을 위한 TPR-Tree 상향식 갱신 기법

고 영 근*,1)

1) 국방과학연구소 제6기술연구본부

A Study on Bottom-Up Update of TPR-Tree for Target Indexing in Naval Combat Systems

Youngkeun Go*,1)

1) *The 6th Research and Development Institute, Agency for Defense Development, Korea*

(Received 16 October 2018 / Revised 27 February 2019 / Accepted 8 March 2019)

ABSTRACT

In modern warfare, securing time for preemptive response is recognized as an important factor of victory. The naval combat system, the core of naval forces, also strives to increase the effectiveness of engagement by improving its real-time information processing capabilities. As part of that, it is considered to use the TPR-tree in the naval combat system's target indexing because spatio-temporal searches can be performed quickly even as the number of target information increases. However, because the TPR-tree is slow to process updates, there is a limitation to handling frequent updates. In this paper, we present a method for improving the update performance of TPR-tree by applying the bottom-up update scheme, previously proposed for R-tree, to the TPR-tree. In particular, we analyze the causes of overlaps occurring when applying the bottom-up updates and propose ways to limit the MBR expansion to solve it. Our experimental results show that the proposed technique improves the update performance of TPR-tree from 3.5 times to 12 times while maintaining search performance.

Key Words : Naval Combat System(함정전투체계), TPR-Tree(TPR-트리), Bottom-Up Update(상향식 갱신), Target Indexing(표적 색인), Spatio-Temporal Indexing(시공간 색인)

1. 서 론

함정전투체계는 함정에 탑재되는 무장, 센서, 통신체계 등을 통합 자동화한 시스템으로, 표적의 탐지부터 식별, 위협평가, 무장 할당, 교전에 이르는 일련의 과정을 최적화함으로써 함정의 전투력을 극대화하는 것을 목적으로 한다.

이를 위한 함정전투체계의 성능요구사항중 하나는 승조원이 복잡한 전장상황을 신속정확하게 판단할 수 있도록 표적정보를 실시간으로 처리할 수 있어야한다는 것이다. 함정전투체계는 네트워크를 통해 연동된

* Corresponding author, E-mail: ykgo@add.re.kr
Copyright ⓒ The Korea Institute of Military Science and Technology

다수의 센서, 통신 체계로부터 표적정보를 수신하며, 제한된 시간 내에 이를 취합하여 식별처리한 후 전시하는 작업을 수행해야 한다.

하지만 최근 국방과학기술의 발전으로 무장 성능 및 개별 센서의 탐지/추적능력이 나날이 향상됨에 따라 이러한 요구사항을 만족하는 것이 점점 어려워지고 있다. 교전공간과 교전대상의 범위가 확대되면서 전투체계에서 처리해야하는 표적의 수가 지속적으로 증가하고 있기 때문이다[1]. 기존 국내 함정전투체계에서는 표적정보를 관리하기 위해 갱신처리 속도가 빠른 해시테이블을 주로 사용해왔는데, 데이터 수에 비례해 검색처리시간이 늘어나는 해시테이블의 특성으로 인해 실시간성이 위협받게 되었다.

관련하여, 표적정보와 같이 시간의 흐름에 따라 연속적으로 위치를 변경하는 특성을 갖는 데이터를 관리하기 위한 다양한 색인 기법[2-10]들이 연구되어왔다. 이들 연구는 시공간적(Spatio-temporal) 특성을 갖는 이동객체들을 효율적으로 색인함으로써 영역검색을 빠르게 수행하는 것을 목적으로 한다. 그중에서도 TPR-트리[5,6]는 현재위치 및 미래 예측위치에 대한 영역검색을 위해 제안된 대표적인 시공간 색인기법으로, 실험결과 함정전투체계에서 빈번히 발생하는 검색(예를 들어 ‘기준 반경 00 km 이내에 위치한 모든 표적을 구하라’던지 ‘t시간 후에 특정 구역을 지날 것으로 예상되는 표적을 구하라’ 등)을 해시테이블 대비 50배 이상 빠르게 처리할 수 있는 것으로 확인되었다. 하지만 TPR-트리는 갱신과정의 복잡성으로 인해 빈번한 갱신을 처리하기에는 무리가 있어 함정전투체계에 실제로 적용하기 위해서는 갱신처리성능을 개선하는 연구가 선행되어야 할 것으로 판단되었다.

TPR-트리에서 객체의 갱신은 루트노드에서 시작해 하향식(Top-down)으로 기존객체를 찾아 삭제한 후 다시 하향식으로 새로운 객체를 삽입하는 방식으로 동작하는데, 이 과정에서 발생하는 검색 및 트리 구조변경 오버헤드가 갱신처리성능 저하를 유발하는 것으로 알려져 있다. 이는 TPR-트리의 기반이 된 R-트리[7,8]의 갱신 알고리즘 고유의 문제로, 앞선 연구[9,10]에서는 상향식(Bottom-up) 접근을 통해 R-트리의 갱신성능을 개선하는 방안이 제안되었다. 상향식 접근법은 보조색인을 통해 갱신하려는 객체를 포함하는 단말노드에 직접 접근하여 트리 하부에서부터 갱신을 시도하는 방법으로, 단말노드 내에 다시 갱신 가능한 객체는 제자리갱신만으로 갱신처리를 완료함으로써 갱신과정을 간소화한다. 해당 논문들은 이러한 방법을 통해 R-트리의 검색성능을 유지하면서도 갱신성능은 크게 향상시킬 수 있었으며 동일한 기법은 R-트리 계열의 다른 트리에도 적용할 수 있을 것이라고 밝혔다. 하지만 R-트리의 상향식 접근을 TPR-트리에 그대로 적용하면 트리 내부노드 간 중복영역이 넓어지면서 검색성능이 악화되는 문제가 발생한다. 시간의 흐름에 따른 객체의 이동을 내부구조에 반영하는 TPR-트리의 특성이 고려되지 않았기 때문이다.

이에 본 논문에서는 TPR-트리의 갱신 과정에 상향식 접근을 적용함에 있어 노드 간 중복영역이 넓어지는 원인에 대해 분석하고 해결방안으로 문제를 유발하는 객체를 제자리갱신 대상에서 제외하는 기법을 제안한다. 또한 제자리갱신에서 제외된 객체를 상향식이 아닌 하향식 삽입으로 처리하되 객체의 삭제를 마지막까지 유예하는 방법을 통해 상위노드 간 중복영역을 줄이고 트리구조변경을 최소화하는 방법을 제안한다. 제안하는 갱신 알고리즘은 기존 TPR-트리의 검색성능은 유지하면서도 갱신성능을 월등히 향상시킬 수 있다.

본 논문의 2장에서는 제안하는 기법을 이해하는데 필요한 배경연구에 대해 설명한다. 3장에서는 TPR-트리에 상향식 접근방식을 적용함으로 인해 발생하는 문제에 대해 분석하고 4장에서는 이를 해결하기 위해 제안하는 갱신 기법을 소개한다. 5장에서는 실험을 통해 제안 기법의 성능개선 효과를 제시한다.

2. 배경 연구

2.1 R-트리

R-트리[7,8]는 공간 데이터를 저장하기 위해 고안된 데이터분할(Data partitioning) 색인기법 중 하나로, 검색성능이 매우 뛰어날 뿐만 아니라 데이터쏠림(Data skew)에도 강한 특성을 보여 여러 연구의 기반이 되었다.

R-트리는 인접한 객체들을 그룹으로 묶어주는 최소경계사각형인 MBR(Minimum Bounding Rectangle)을 높이균형트리 계층구조로 구성하여 객체들을 관리한다. Fig. 1은 10개의 객체[a, b, …, j]를 색인하는 R-트리 예제를 보여준다. 트리구조에서 인접한 객체 [a, b, c]는 단말노드 N_{11}의 MBR에 포함되며 [d, e]는 N_{12}의 MBR에 포함된 것을 확인할 수 있다. N_{11}과 N_{12}는 다

시 트리 상위 계층의 노드 N_1의 MBR에 포함되며 N_1은 더 넓은 영역을 포함하는 루트노드 R_0의 MBR에 포함된다.

R-트리의 이러한 구조는 영역검색을 빠르게 수행하기 위함이다. 검색은 루트노드에서 시작하여 자식노드로 내려가면서 해당노드의 MBR이 검색영역을 포함하는지 반복해서 비교하는 과정으로 이루어진다. 검색영역이 MBR에 포함되면 자식노드로 내려가서 다시 비교하고, 포함되지 않으면 이하 트리는 검색하지 않는다. 검색영역에 해당하지 않는 서브트리를 가지치기함으로써 검색대상을 추려내는 것이다. 이때 MBR간 겹침 정도에 따라 검색경로가 복잡해질 수 있다. 최종적으로 단말노드에 이르면 객체들의 위치를 비교해 검색영역에 포함되는 객체만 가려낸다.

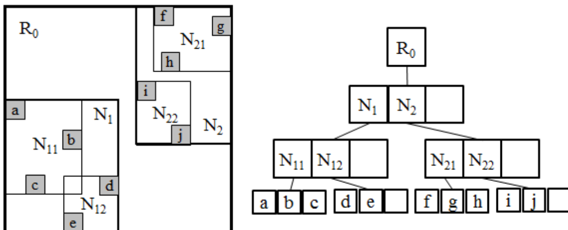

Fig. 1. Example R-tree

R-트리는 검색에 최적화된 트리를 구성하기 위해 4개의 페널티 메트릭(MBR의 크기, MBR의 둘레, MBR간 오버랩, MBR과 내부객체 중심 간의 거리)을 최소화하는 방향으로 객체의 갱신위치를 결정한다[11]. 객체를 갱신하기 위해서는 먼저 기존객체를 트리에서 찾아 삭제해야 하는데, 객체를 찾는 과정은 검색과 동일하다. 객체를 삭제한 후 단말노드에 일정비율 이하의 객체만 남아있다면 언더플로우(Underflow)가 발생했다고 판단하고 해당 노드를 삭제하는 작업을 수행한다. 남아있던 객체는 트리에 다시 삽입한다. 다음으로 새로운 객체를 삽입하기 위해 루트노드에서부터 내려가며 자식노드들 중 객체의 삽입으로 인한 MBR, 오버랩의 확장을 가장 적게 유발하는 노드를 반복해서 찾는다. 단말노드에 이르면 객체를 삽입한 후 MBR을 재계산한다. 해당 노드가 일정 수 이상의 객체를 포함하게 되면 오버플로우(Overflow)가 발생했다고 판단하고 중심에서 가장 먼 객체들을 삭제 후 재삽입한다. 그럼에도 오버플로우가 다시 발생하면 MBR의 둘레와 MBR간 오버랩이 가장 작아지도록 해당 노드를 둘로 분할한다. 단말노드의 언더플로우나 오버플로우는 경우에 따라 상위노드로 전파될 수 있으며 상위노드에서도 단말노드와 동일한 메커니즘으로 처리한다. 이와 같은 갱신 방법은 검색에 적합한 색인 구조를 구성할 수는 있으나 트리 구조변경이 자주 발생하므로 빈번한 갱신을 처리하기에는 적합하지 않다.

2.2 TPR-트리(Time Parameterized R*-tree)

TPR-트리[5,6]는 갱신이 빈번한 이동객체를 색인하기 위해 R-트리를 확장한 연구로, 객체의 위치를 좌표가 아닌 시간에 대한 위치함수로 저장하는 방법을 제안하였다. 위치함수를 이용하면 이동객체의 위치 변화를 모두 트리에 반영하는 것이 아니라 속도 성분이 변할 때에만 갱신할 수 있으며, 객체의 이동을 고려해 현재와 미래시점의 위치를 예측할 수 있다는 장점이 있다.

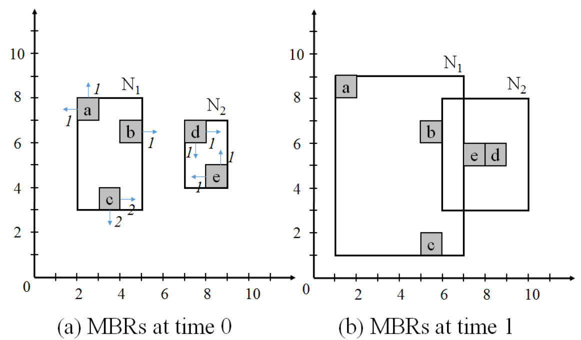

Fig. 2. Entry representations in a TPR-tree

Fig. 2는 시간의 흐름에 따른 TPR-트리 내부 객체와 MBR의 변화를 보여준다. (a)에서 (b)로 1초가 흐르면서 객체들이 각자의 위치함수를 따라 이동하였으며 객체들을 포함하는 MBR도 이를 반영해 확장되었다. 하지만 (b)에서 MBR은 최소경계가 되도록 조여지지 않은 것을 알 수 있는데, 이는 MBR을 최소경계사각형으로 유지하는데 필요한 연산 오버헤드를 피하기 위해 의도적으로 선택한 정책적 결과이다. TPR-트리에서는 시시각각 변하는 MBR을 매번 최소경계사각형으로 재계산하는 방법 대신 VBR(Velocity Bounding Rectangle)이라고 하는 속도벡터를 통해 MBR을 관리한다. VBR은 상하좌우 네 방향에 대한 속도성분으로 구성되며 노드에 포함된 객체(또는 자식노드)들의 속도 성분 중 가장 큰 값을 갖는데, 예를 들어 N_1의 VBR(x_-, x_+, y_-, y_+)은 (-1, 2, -2, 1)이 된다. 이에 따라

N_1의 MBR이 (2, 5, 3, 8)에서 1초 후 (1, 7, 1, 9)로 VBR만큼 확장된 것이다. 이러한 방법은 시간이 흐르더라도 간단한 연산만으로 MBR이 계속해서 모든 객체를 포함할 수 있도록 한다. 하지만 MBR이 불필요하게 계속 커지기만 하면 MBR 간 중복영역이 생기게 되고, 검색성능이 저하되는 문제가 발생한다. TPR-트리에서는 이러한 트레이드오프를 감안하여 해당 노드에 갱신이 발생하는 경우에만 MBR을 최소경계로 재조정한다.

검색 및 갱신처리과정은 R-트리와 동일하다. 다만 검색이나 삽입/삭제 시 접근하는 노드의 MBR은 현재 시점으로 계산 후 사용된다. 페널티 메트릭의 경우 T 시간 후의 미래 예측질의에 최적화 될 수 있도록 [t_0 ~ t_0 + T] 구간에 대한 적분계산으로 구해진다.

TPR-트리의 문제는 객체의 실제 기동을 위치함수로 나타내기 어렵다는 것이다. 객체가 급격한 속도 변화를 보이며 매번 위치함수와 다른 위치로 갱신되는 상황에서는 TPR-트리도 모든 갱신을 처리할 수밖에 없다. 하지만 앞서 R-트리에서 설명한 바와 같이 하향식 갱신 방식으로는 빈번한 갱신을 처리하기 어렵다.

2.3 상향식 접근법(Bottom-up Approach)

R-트리의 갱신성능을 개선하기 위해 상향식으로 갱신을 시도하는 방안이 연구되었다.[9,10] 메인 트리 외에 별도의 2차 색인을 추가함으로써 갱신하려는 객체의 과거 정보를 포함하고 있는 단말노드에 직접 접근하는 것이다. 기존에는 객체를 삭제하기 위해 트리에서 검색을 수행해야 했지만 제안된 방법을 이용하면 검색처리를 생략하여 접근시간을 줄일 수 있다. 또한 상향식 접근법에서는 객체들의 이동이 공간지역성을 갖는다는 점을 고려하여 트리 하부에서부터 갱신을 시도한다. 우선 갱신하려는 위치가 기존객체를 포함하는 단말노드의 MBR에 포함되는지 확인하여 제자리에 갱신이 가능하면 객체의 정보만 새로운 값으로 갱신하고 바로 갱신을 종료한다. 단말노드의 MBR은 변하지 않기 때문에 단말노드 내 정보는 변하지 않으며 당연히 트리구조도 변하지 않는다. 이러한 방법은 불필요하게 삭제 후 삽입하는 과정을 거치지 않고, 특히 그 중에서도 오버헤드가 큰 언더플로우 및 오버플로우 처리를 덜 수행하게 되므로 갱신성능을 크게 향상시킬 수 있다. 제자리갱신 대상이 아니라면 부모노드로 올라가며 MBR 포함 여부를 확인하여 최대한 낮은 레벨의 서브트리에서 삽입을 시도한다.

해당 연구들은 제안하는 방법이 R-트리 계열의 다른 트리에도 적용될 수 있을 것이라고 밝히고 있는데, 실제로 HTPR*-트리[12,13], VTPR-트리[14], HVTPR-트리[15], BU-TPR*-트리[16] 등의 연구에서 TPR-트리에 상향식 접근법을 적용하였다고 기술하고 있다. 하지만 대부분 새로운 기법을 제안하는 것에 초점을 두고 있어 TPR-트리에 적용한 상향식 접근법에 대한 깊이 있는 분석은 수행되지 않았다.

3. 문제점 분석

아래는 R-트리의 상향식 갱신 방식 그대로를 TPR-트리에 적용한 알고리즘을 보여준다. MBR을 재조정하는 과정과 제자리갱신 결정을 위한 VBR 비교조건이 추가되었을 뿐 기본 동작방식은 R-트리의 것과 동일하다. 하지만 이러한 갱신 알고리즘을 TPR-트리에 실제로 적용한 결과 갱신성능은 개선되지만 검색성능이 기존 대비 4~7배 가까이 저하되는 문제를 확인할 수 있었다. 실험결과에 관한 내용은 5장에서 자세히 설명하도록 하고 우선 본 장에서는 문제가 발생하는 3가지 원인에 대해 분석한다.

Algorithm 1. Naive bottom-up update for TPR-tree

```
Update(o, o')

Input : An object o and its new position o'
1.  leaf ← lookupTable.getLeafNode(o)
2.  recalculate the leaf.MBR as current time
3.  if (o' ⊂ (leaf MBR & VBR))
4.    update location of o in the leaf
5.    tighten all nodes in the path from leaf to root
6.  else
7.    delete(leaf, o)
8.    if (removal of o causes underflow)
9.      resolve underflow
10.       insert(root, o')
11.   else
12.     parent ← getParent(leaf)
13.     while (o' ⊄ (parent MBR & VBR))
14.       parent ← getParent(parent)
15.     insert(parent, o')
```

– 원인 1. 제자리갱신 결정을 위한 MBR 비교

R-트리에서는 제자리갱신 여부를 판단할 때 과거 제일 마지막으로 갱신이 완료된 시점의 MBR을 사용한다. 객체가 갱신되지 않으면 MBR이 바뀌지 않기 때문이다. 반면 TPR-트리에서는 시간에 따른 객체의 이동을 고려해 MBR이 동적으로 변하므로 제자리갱신 결정의 판단기준으로 사용하는 MBR을 어떤 방식으로 구하느냐에 따라 상향식 접근의 결과가 달라진다.

TPR-트리에서는 노드 접근 시 현재시점의 MBR을 이용하며, 일반적으로 현재시점의 MBR을 구하는 방법은 위치함수를 통해 내부 객체들의 위치를 현재 시점으로 계산한 뒤 MBR을 최소경계사각형으로 재계산하거나 VBR을 통해 기존에 계산되어 있던 과거시점의 MBR을 현재와의 시간차만큼 확장하는 것이다. 하지만 두 방법으로 구한 MBR 중 어느 것을 비교에 사용하더라도 대부분의 객체가 제자리갱신 처리되어버리는 문제가 발생한다. MBR이 갱신하려는 객체의 과거 위치함수를 반영하고 있기 때문이다. 객체가 제자리갱신 대상에서 제외되는 경우는 객체의 과거 위치함수 대비 급격한 위치/속도 변화가 있는 경우에 한한다.

그 결과 대부분의 객체들이 한번 결정된 노드에서 벗어나지 못하고 자신이 포함된 노드에 반복해서 갱신되면서 MBR을 계속해서 확장시키는 악순환에 빠지게 된다. 물론 제자리갱신이 많이 될수록 갱신성능 측면에서는 좋다. 그러나 MBR을 확장시켜 검색성능을 악화시키는 객체는 제자리갱신에서 제외해야 함에도 불구하고 기존 알고리즘은 이를 판단하지 못한다. 그렇다고 R-트리처럼 과거 시점의 MBR을 비교대상으로 사용하는 것이 대안이 되는 것도 아니다. 정적인 R-트리와는 달리 TPR-트리에서는 다른 객체들도 모두 움직이기 때문이다. 과거의 MBR은 현재 시점에서 더 이상 최적의 위치임을 보장하지 못한다.

– 원인 2. 제자리갱신 결정을 위한 VBR 비교

TPR-트리에 상향식 접근을 적용한 일부 논문들[12-16]에서는 MBR에 더해 VBR을 비교조건에 포함하여 제자리갱신 대상을 제한하는 방법을 사용하였다. 이는 속도가 가장 빠른 객체를 제자리갱신에서 제외함으로써 MBR이 확장되는 것을 막으려는 시도로 보인다.

하지만 단순히 갱신하려는 객체의 속도가 기존노드의 VBR에 포함되는지 비교하는 방법만으로는 제자리에 갱신되어야 하는 객체와 제자리에 갱신되지 말아야하는 객체를 정확하게 구별하지 못한다. Fig. 3의 (a)는 제자리에 갱신되어야 할 객체를 제외하는 상황을 보여준다. 예제를 단순화하기 위해 a객체만 MBR 내 최고속도인 (1, 1)로 움직이고 있고 다른 객체들은 이동하지 않는다고 가정하자. 이때 a가 기존보다 더 빠른 (2, 2)의 속도로 갱신되는 경우 a`는 다른 객체들과의 거리가 가까워지고 MBR의 확장을 유발하지 않음에도 불구하고 제자리갱신 대상에서 제외된다. 물론 a`는 삭제 후 삽입하는 과정에서 같은 노드로 재삽입될 가능성이 높기 때문에 검색성능을 악화시키지는 않을 것이다. 하지만 제자리갱신으로 처리할 수 있는 객체를 제외한 것이므로 추가적인 갱신성능개선의 여지가 있음을 의미한다.

반면 (b)의 경우는 제자리갱신하지 말아야하는 객체를 제자리갱신하는 경우를 보여준다. 마찬가지로 d만 MBR내 최고 속도 (2, 2)로 움직이고 나머지 객체는 고정되어 있다고 가정하자. 이때 d가 기존보다 느린 (1, 1)의 속도로 갱신되는 경우 d`의 제자리갱신은 MBR의 확장을 유발함에도 불구하고 재계산된 MBR에 포함되기 때문에 제자리갱신으로 처리된다. 물론 MBR이 한두 번 확장된다고 해서 검색성능이 크게 악화되는 것은 아니고 이후에 속도가 다시 빨라진다면 제자리갱신 대상에서 제외되어 최선의 자리에 삽입될 수 있을 것이다. 그러나 급격한 기동의 변화 없이 한 방향으로 일정한 가속을 하는 시간인 길어질 경우 (a)나 (b)와 같은 경우가 연속해서 발생하면서 트리의 성능이 저하될 수 있다.

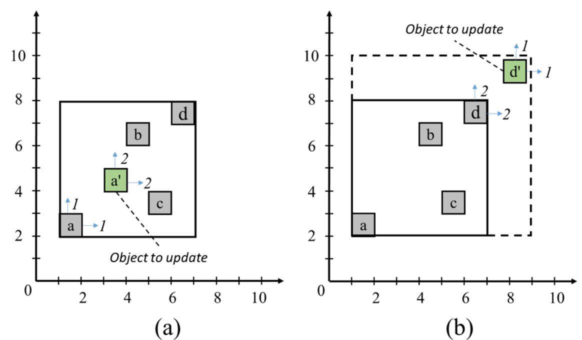

Fig. 3. Problems caused by VBR comparison

– 원인 3. 제자리갱신 제외 대상의 처리 방법

상향식접근에서는 제자리갱신에서 제외된 객체를 단말노드에서 삭제한 후 트리하부에서부터 상위노드로 올라가며 갱신을 시도한다. 공간지역성을 고려할

때 같은 서브트리 내에 포함된 인접노드로 이동할 가능성이 높기 때문이다. 서브트리의 크기가 작을수록 전체트리에 하향식으로 삽입하는 것 대비 소모시간을 줄일 수 있으므로 이러한 시도는 유효하다.

트레이드오프도 존재한다. 단말노드에서 루트노드까지의 경로를 따라 올라가며 해당 노드의 MBR만으로 판단하기 때문에 다른 노드로 삽입되는 것이 효율적이더라도 그쪽으로 이동하지 못하고 중복을 유발하는 위치에 삽입하는 경우가 발생할 수 있다. 갱신성능 향상을 위해 약간의 검색성능을 희생하고 지역 최적값(Local optimum)을 선택하는 것이다. 그렇지만 이러한 방법을 사용하더라도 R-트리에서는 중복영역이 국부적으로 증가할 뿐이지 계속해서 넓어지는 것은 아니기 때문에 실제로 검색성능에는 큰 영향이 발생하지 않았다.

반면 TPR-트리에서는 그 이상의 문제가 발생한다. 부모노드의 MBR은 하부노드의 MBR을 바탕으로 계산되는데 중간노드의 MBR은 보통 실제 내부 객체들보다 확장되어있는 경우가 많다. 갱신이 발생하지 않은 노드의 MBR을 현재시점으로 계산할 때에는 VBR이 사용되기 때문이다. 게다가 상위 노드로 갈수록 최고 속도가 더 빨라지므로 MBR은 더 빨리 확장된다. 따라서 갱신되는 객체 거의 대부분이 루트노드 또는 상위노드까지 도달하지 못하고 단말로부터 가까운 서브트리에 다시 갱신된다. 그 결과, 중간노드에서도 단말노드에서처럼 노드 간 중복영역을 넓히는 객체를 제대로 제외시키지 못하고 같은 노드로의 갱신이 빈번하게 발생하면서 검색처리성능이 악화되는 문제가 발생한다.

4. 제안 기법

결국 TPR-트리에 상향식 접근을 적용하면서 발생하는 모든 문제는 MBR이 커지면서 발생함을 알 수 있다. 따라서 TPR-트리의 특성을 고려하여 갱신과정에서 MBR의 확장을 적절히 제한하는 방안이 필요하다.

먼저 단말 노드에서 과도한 제자리갱신으로 인해 발생하는 문제를 해결하기 위해서는 MBR 확장을 유발하는 객체를 구별하여 제자리갱신 대상에서 제외해야 한다. 이때 어떤 기준으로 MBR의 확장이 발생했다고 판단할지가 중요하다. TPR-트리에서 MBR은 대체로 확장중이기 때문이다. 앞장에서 설명했듯이 자신의 과거 위치함수를 포함한 현재의 MBR, 과거의

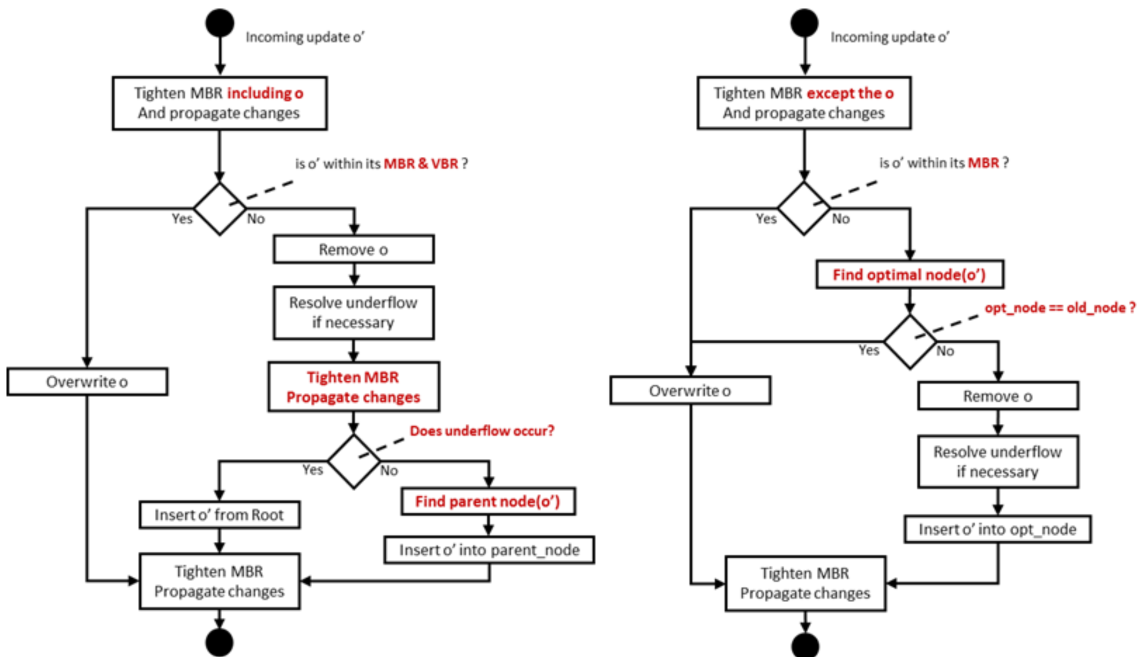

Fig. 4. Comparison of naive bottom-up algorithm and proposed update algorithm

MBR은 최적의 위치임을 보장하지 못하며 VBR 또한 판단의 기준이 될 수 없었다. 본 논문에서 제안하는 방법은 갱신하려는 객체를 제외한 MBR을 비교기준으로 사용하는 것이다. 현재 시점으로부터 어떤 T 시간 동안에 갱신하려는 객체의 위치가 자신을 제외한 객체들이 구성한 MBR에 포함된다면 MBR의 확장을 일으키지 않는 객체로 판단하여 제자리갱신으로 처리한다. 이러한 방법은 MBR에서 최외곽에 위치하며 다른 객체들과 멀어지려하여 MBR을 확장시킬 가능성이 있는 객체를 제자리갱신에서 제외한다.

중간노드에서 발생하는 MBR 확장 문제는 부모노드로 올라가면서 서브트리를 찾는 것이 아니라 루트노드에서 삽입하는 방법을 통해 해결한다. 앞서 설명한 바와 같이 MBR이 확장되는 TPR-트리에서는 부모노드의 MBR 비교만으로 갱신 위치를 결정하는 방법이 적합하지 않기 때문이다. 물론 루트노드에서 하향식으로 삽입하더라도 MBR은 확장되어 있다. 상위 노드일수록 MBR이 크게 확장되어 여러 노드가 중복된 위치에 삽입을 시도하는 경우가 자주 발생한다. 삽입할 위치를 결정할 때에는 객체의 삽입이 MBR을 적게 확장시키는 방향으로 선택하는데 중복된 영역에 삽입되는 경우 MBR 확장은 동일하게 0이 된다. 하지만 상위노드로부터 삽입하는 경우 자식노드 간 타이브레이크(Tie-break) 조건을 추가로 비교하여 더 최적의 위치를 결정할 수 있다는 장점이 있다. 본 연구에서는 인메모리 환경임을 고려하여 네 가지 메트릭 중 가장 계산이 간단한 중심노드와의 거리를 추가 비교하는 방법을 사용하였다.

부모노드 대신 루트노드에 삽입하며 발생하는 갱신 오버헤드를 보완하기 위해서는 삭제를 지연하는 기법을 적용할 수 있다. 실제로 객체를 삭제하지는 않고 삭제한 것처럼 MBR을 트리 내부에 반영한 뒤 갱신하려는 객체를 먼저 삽입해봄으로써 제자리갱신에서 제외되었지만 다시 한 번 같은 단말노드로 삽입되는 경우를 구분하는 것이다. 만약 같은 단말노드로 다시 삽입되는 경우 기존객체를 갱신함으로써 불필요한 삭제를 하지 않을 수 있으며 그로인한 언더플로우 처리도 줄일 수 있다.

제안하는 업데이트 알고리즘의 처리과정은 다음과 같다. 우선 단말노드로 직접 접근하여 MBR을 현재 시점으로 재조정한다. 이때 갱신하려는 객체를 제외한 MBR을 구한다. 이후 변화된 MBR을 루트노드까지 반영한 다음 제자리갱신이 가능한지 판단한다. 갱신하려는 객체가 가까운 미래 시간 T 동안 MBR에 포함되면 객체의 정보만 변경하고 갱신처리를 종료한다. 제자리갱신 대상이 아니라면 기존객체를 삭제하기 이전에 먼저 갱신하려는 객체를 삽입하기 위한 최적의 위치를 탐색한다. 앞서 객체가 삭제된 것처럼 MBR을 루트노드까지 반영해두었기 때문에 추가적인 처리 없이 바로 탐색을 진행할 수 있다. 삽입위치를 찾는 과정은 루트노드에서 시작하며 기존 TPR-트리의 삽입과정과 동일하다. 만약 같은 노드에 다시 삽입되는 경우라면 삽입되어 있는 기존객체에 정보를 수정한다. MBR은 갱신위치까지 포함하도록 확장한다. 끝으로 변경된 MBR을 루트노드까지 반영하고 갱신을 종료한다. 삽입위치가 기존 단말노드와 다른 경우 기존객체를 삭제하고 필요에 따라 언더플로우 처리를 완료한 후 새로운 위치에 갱신하려는 객체를 삽입하고 갱신을 완료한다.

Algorithm 2. Update process proposed in this paper

```
Update(o, o')

Input : An object o and its new position o'
1.  leaf ← lookupTable.getLeafNode(o)
2.  tighten the leaf.MBR except the o
3.  tighten all nodes in the path from leaf to root
4.  if (o' ⊂[t0 ~ t0+T] leaf MBR)
5.     update location of o in the leaf
6.  else
7.     opt_node ← findOptimalNode(root, o')
8.     if (opt_node = leaf)
9.        update location of o in the leaf
10.       expand leaf.MBR
11.       tighten all nodes in the path from leaf to root
12.    else
13.       delete(leaf, o)
14.       resolve underflow if necessary
15.       insert(optNode, o')
```

5. 성능 실험

5.1 실험 환경

실험에는 Intel Xeon CPU E5-2670 v3(2.30 GHz) 4코어, 8 GB 메모리를 갖는 PC가 이용되었으며 운영체

제는 Windows Server 2012를 사용하였다. 모든 자료구조 및 표적정보는 메모리 상에 위치한다.

제안 기법을 검증하기 위해 다음 표와 같이 총 6가지 타입의 TPR-트리를 구현하여 실험을 진행하였다. TD-트리는 하향식 갱신을 수행하는 기본 TPR-트리이며 BU-트리는 알고리즘 1에서 소개한 상향식 갱신으로 동작하는 TPR-트리를 말한다. Proposed-트리는 제안하는 기법을 적용한 트리를 지칭한다. 앞장에서 지목한 세 가지 문제의 영향성을 각각 분석하기 위해 제안하는 기법에서 한 가지씩만 다르게 구성한 Type 1, Type 2, Type 3 트리를 함께 비교하였다.

Table 1. TPR-trees for comparison

	MBR에 자신 포함	VBR 비교	부모노드 삽입
TD	-	-	-
BU	O	O	O
Type 1	O	X	X
Type 2	X	O	X
Type 3	X	X	O
Proposed	X	X	X

실험은 개발한 표적 시뮬레이터 상에 300×300 km^2 크기의 이차원 공간을 설정하고 랜덤분포로 표적을 배치하여 진행하였다. 표적은 공중(30 %), 수상(60 %), 수중(10 %) 표적으로 구별되며 각 표적의 초기 속도는 타입 별 평균 속도(1000 km/h, 50 km/h, 5 km/h)에 30 %의 표준편차를 갖도록 가우시안 분포로 설정하였다. 각 표적의 속도는 갱신될 때마다 기존 대비 최대 30 %까지 변하되 평균속도의 3배는 넘지 못하도록 설정하였다. 이외의 실험 변수 및 변수 값은 아래 표와 같다. 각 변수가 통제변인(Control variable)으로 설정되는 경우 굵게 표시한 값을 이용하였다.

Table 2. Simulation parameters

실험 변수	변수 값
데이터 수	20 k, 40 k, **60 k**, 80 k, 100 k
평균 갱신 주기	20, 40, **60**, 80, 100
질의 영역 크기	2, 4, **6**, 8, 10

5.2 갱신성능 실험결과

갱신성능 시험은 평균 갱신 주기의 6배 시간만큼 지나도록 하여 모든 표적이 최소 1회 이상 갱신되도록 한 후 다시 평균 갱신주기의 6배 시간만큼을 지나는 동안 걸리는 시간과 노드접근 횟수를 측정하였다.

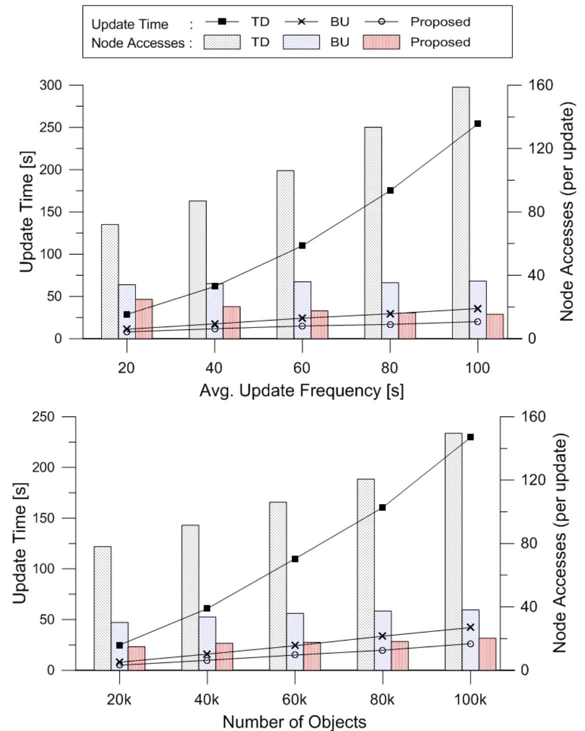

Fig. 5. Update performance for varying update frequency and number of objects

갱신주기에 따른 갱신처리성능 그래프와 총 데이터수에 따른 갱신처리성능 그래프를 보면 제안 기법을 적용한 Proposed-트리는 기본 TD-트리 대비 3.5배에서 최대 12배 이상 갱신처리가 빠른 것으로 나타났다. 갱신 당 노드 접근횟수에서도 Proposed-트리가 TD-트리보다 3배~10배 가량 적어 갱신시간과 유사한 경향을 보였다. TD-트리에서는 갱신 주기가 길어지거나 객체의 수가 많아질수록 갱신시간이 길어지고 노드접근횟수가 많아지는 경향을 보였는데, 이는 두 변수의 값이 커질수록 중복영역이 넓어지기 때문이다. 반면 상향식 갱신을 수행하는 BU-트리와 Proposed-트리는 갱신주기나 객체의 수의 변화에도 상대적으로 일정한 성능이 유지되는 결과를 보였다. 노드 간 중복 정도에

관계없이 단말노드로 접근하여 제자리갱신을 시도하는 비율이 높기 때문이다.

Proposed-트리는 예상 밖으로 BU-트리보다도 30~40 % 가량 갱신 시간이 줄어드는 결과를 보였다. 제안 기법이 검색성능을 위해 삭제한 객체를 루트노드로부터 삽입하는 방식을 취함에도 불구하고 갱신성능이 향상된 이유는 제자리갱신에 성공하는 비율이 높기 때문인 것으로 나타났다.

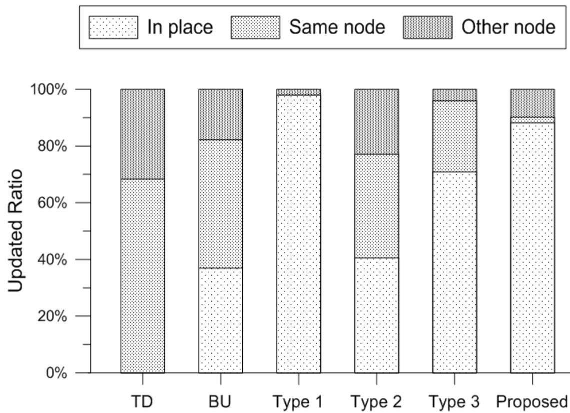

Fig. 6. Updated ratio for each tree type

Fig. 6은 전체 갱신처리 중 제자리갱신이 되는 비율(In place)과 제자리갱신 실패 후 다시 제자리로 삽입되는 비율(Same node), 다른 노드로 삽입되는 비율(Other node)을 측정한 결과를 나타낸다. BU-트리는 36.9 % 가량만 제자리갱신을 하고 45.3 % 가량은 부모노드로 삽입 후 다시 같은 노드로 들어가는 결과를 보인 반면, Proposed-트리는 제자리갱신에 88.1 % 가량이 들어가고 삭제 후 삽입되는 객체가 같은 노드로 들어가는 비율은 2.1 % 밖에 되지 않는 것으로 나타났다. 이는 결국 BU-트리가 제자리갱신으로 처리할 수 있는 객체를 불필요한 추가 연산을 통해 갱신했다는 것을 의미한다. BU-트리와 가장 비슷한 경향을 보이는 Type 2를 통해 이러한 문제가 발생한 원인이 VBR을 비교조건으로 사용하였기 때문임을 추론할 수 있다. Type 1은 자신을 포함한 MBR을 제자리갱신 비교조건으로 사용하면서 약 97.9 %의 객체가 제자리갱신되는 것을 확인할 수 있다. Type 3의 경우 Proposed-트리 대비 삭제 후 같은 노드로 재삽입되는 비율이 높은 것을 알 수 있다. 제자리갱신에 실패한 객체를 부모노드에 삽입하다보니 단말노드와 가까운 중간노드에 삽입된 대부분의 객체들이 다시 원래 위치로 되돌아간 것이다. Type 3에서 제자리갱신 제외 객체가 트리의 어느 레벨에서 갱신되는지를 측정한 결과 단 1.1 %만이 루트노드까지 도달하는 것을 확인할 수 있었다.

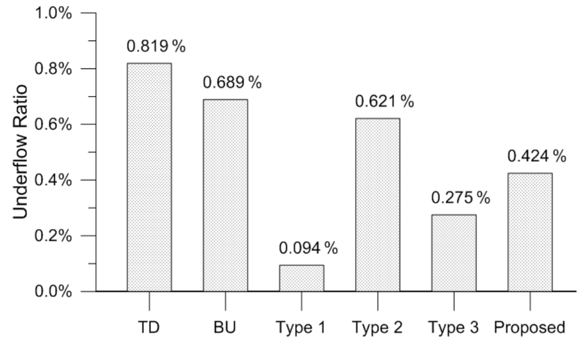

Fig. 7. Underflow ratio for each tree type

제안 기법에서 갱신성능이 향상된 또 하나의 이유는 언더플로우가 줄었기 때문이다. Fig. 7은 갱신 당 언더플로우가 발생하는 비율을 나타낸다. 대부분의 객체가 원래 자리에 갱신되는 Type 1과 Type 3을 제외하고는 Proposed-트리가 언더플로우 비율이 가장 낮음을 알 수 있다. Proposed-트리는 제자리갱신 비율이 높아 삭제되는 객체의 수가 적을뿐더러 삭제 후 삽입이 아니라 삽입을 먼저 시도하여 같은 노드로 삽입되는지 비교하는 기법을 사용함으로써 언더플로우의 발생을 최소화했기 때문이다. 측정결과 Proposed-트리에서 삽입을 먼저 시도함으로써 회피한 불필요한 언더플로우 수는 전체 발생 횟수의 약 5.5 %에 해당하는 것으로 확인되었다.

5.3 검색성능 실험결과

검색성능 실험은 갱신성능 실험을 완료한 후 가로세로 100×100개의 지점에 대해 10회씩 영역검색을 반복 시도하여 소요되는 시간을 측정하였다.

갱신주기 별, 총 데이터 수 별, 검색영역 별 검색처리성능을 각각 측정한 결과(Fig. 8) Proposed-트리는 TD-트리와 거의 유사한 검색처리 능력을 보였다. 검색 당 노드접근 횟수를 측정한 결과에서도 비슷한 경향이 나타나는 것을 확인할 수 있었다. 반면 BU-트리의 경우 4배~7배 가량 검색처리가 느린 것을 확인할 수 있다.

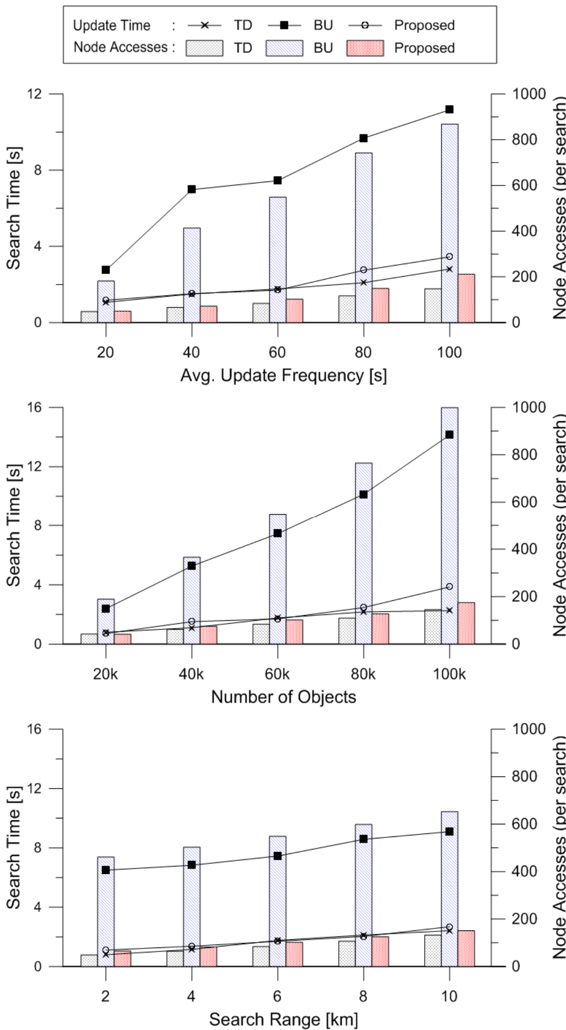

Fig. 8. Search performance for varying update frequency, number of objects and search range

이러한 결과가 나타난 이유는 MBR과 중복영역의 크기에서 찾을 수 있다. 갱신 실험 완료 후 트리 레벨별 MBR과 노드의 평균 중복영역의 넓이를 측정한 결과(Fig. 9)를 보면 BU-트리가 기존 TD-트리 대비 10배 이상의 MBR을 갖고 중복영역의 넓이는 100배 이상 차이나는 것으로 나타났다. 원인은 Type 1과 Type 3에서 찾을 수 있다. 단말노드에서 대부분 제자리갱신을 수행하는 Type 1은 비교에 사용한 6가지 트리 중 가장 넓은 MBR과 중복영역이 발생하는 것으로 나타났다. Type 2는 MBR 계산에 자신을 포함하지 않았기

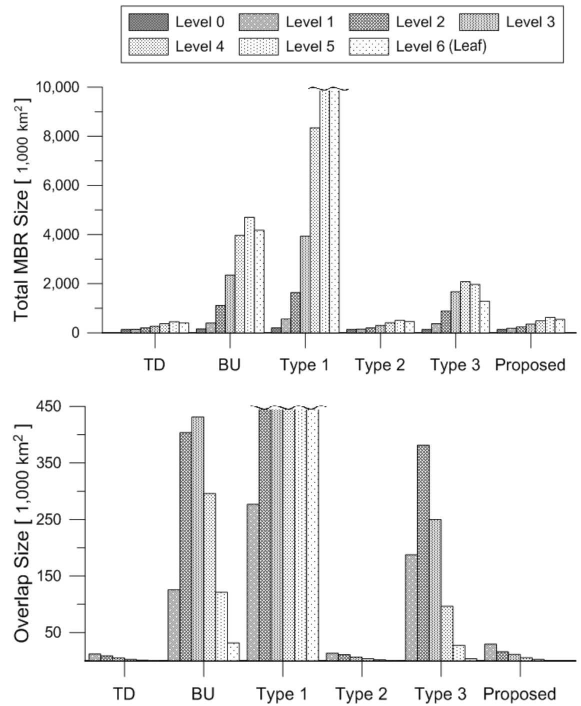

Fig. 9. MBR and overlap size for each tree type

때문에 제자리갱신으로 인해 MBR이 확장되는 현상은 나타나지 않았다. 그보다는 VBR 조건으로 인해 많은 객체들이 제자리갱신에서 제외됨으로써 하향식 갱신으로 처리되어 TD-트리와 유사한 경향을 보였다. Type 3는 부모노드에 삽입하는 방식 때문에 MBR이 확장되었다. 특히 Type 1과는 달리 중간노드에서의 MBR 확장이 두드러지게 나타났다. BU-트리는 Type 1, Type 2의 방식이 혼합되어 제자리갱신 여부를 결정하므로 Type 1 방식으로 인한 MBR 확장이 일부 상쇄되었지만 부모노드에 삽입하는 방식도 같이 사용하므로 결과적으로 MBR과 중복영역이 크게 나타났다.

6. 결 론

현대전에서는 선제 대응을 위한 시간 확보가 승리의 중요한 요소로 인식되고 있다. 해군의 핵심전력인 함정전투체계 또한 실시간 정보처리능력을 확대하여 교전 효과도를 높일 수 있도록 개발되고 있다. 하지만 처리해야할 표적 정보의 수가 늘어남에 따라 기존의 자료구조로는 실시간 표적검색처리가 힘들어지고 있

으며 그 대안으로 TPR-트리를 표적색인에 사용하는 방안이 고려되었다.

그러나 TPR-트리와 같이 하향식 갱신방법을 사용하는 R-트리 계열의 색인 방식은 갱신성능이 느려 빈번한 갱신을 처리하기에는 적합하지 않다. 이에 본 논문에서는 상향식 갱신을 적용하여 TPR-트리의 갱신성능을 개선하는 방안을 제안하였다. 특히 상향식 갱신 과정에서 중복영역이 넓어지는 원인을 분석하고 이를 해결하기 위한 기법을 제시함으로써 검색성능은 유지하면서도 갱신성능을 3.5배 이상 개선하는 결과를 보였다.

이외에도 R-트리와 TPR-트리의 갱신성능을 향상시키기 위해 TPR*-트리[6], STAR-트리[17], RUM[18,19], R^{sb}-트리[20], 능동적 재조정 기법[21] 등의 연구가 수행된 바 있다. 하지만 대부분 디스크 기반을 염두에 두고 연구가 진행되어 실시간으로 표적처리를 수행해야 하는 전투체계에는 적합하지 않다. 메모리상에서의 색인연구로는 캐시미스를 줄이기 위해 데이터사이즈를 줄이는 기법을 제안한 CR-tree[22] 등이 있다. 해당 기법은 본 논문의 제안 기법과 병행하여 사용될 수 있을 것이다.

References

[1] H. Lee, T. Kim, H. Shin, “Multi Sources Track Management Method for Naval Combat Systems,” Journal of Institute of Control, Robotics and Systems 20(2), 126-131, 2014.

[2] M. F. Mokbel, T. M. Ghanem, and W. G. Aref, “Spatio-Temporal Access Methods,” IEEE Data Eng. Bull., Vol. 26, No. 2, pp. 40.49, Jun. 2003.

[3] L. V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel, “Spatio-Temporal Access Methods: Part 2(2003.2010),” IEEE Data Eng. Bull., Vol. 33, No. 2, pp. 46.55, Jun. 2010.

[4] John, A., M. Sugumaran, and R. S. Rajesh, “Indexing and Query Processing Techniques in Spatio-Temporal Data,” ICTACT Journal on Soft Computing 6.3 : 1198-1217, 2016.

[5] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez, “Indexing the Positions of Continuously Moving Objects,” SIGMOD 2000.

[6] Y. Tao, D. Papadias and J. Sun, “The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries,” Proc. 29th Very Large Data Bases Conference, pp. 790-801, 2003.

[7] A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching,” Proc. of ACM SIGMOD, Vol. 14, No. 2, pp. 47-57, June 1984.

[8] Beckmann, Norbert, et al., “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles,” Acm Sigmod Record, Vol. 19, No. 2, Acm, 1990.

[9] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo, “Supporting Frequent Updates in R-Tree: A Bottom-Up Approach,” Proc. of VLDB '03, Vol. 29, pp. 608-619, Sep. 2003.

[10] D. Kwon, S. Lee, and S. Lee, “Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree,” Proc. of MDM '02, pp. 113-120, Jan. 2002.

[11] Pagel, B., Six, H., Toben, H., Widmayer, P, “Towards an Analysis of Range Query Performance in Spatial Data Structures,” PODS, 1993.

[12] Fang, Ying, et al., “HTPR*-Tree: An Efficient Index for Moving Objects to Support Predictive Query and Partial History Query,” International Conference on Web-Age Information Management, Springer, Berlin, Heidelberg, 2011.

[13] Fang, Ying, et al., “Indexing Partial History Trajectory and Future Position of Moving Objects Using HTPR*-Tree,” International Conference on Database Systems for Advanced Applications, Springer, Berlin, Heidelberg, 2012.

[14] Liao, Wei, et al. “An Efficient Indexing Method for Moving Objects with Frequent Updates,” International Conference on Conceptual Modeling, Springer, Berlin, Heidelberg, 2006.

[15] Liao, Wei, et al., “Hybrid Indexing of Moving Objects based on Velocity Distribution,” Chinese Journal of Computers-Chinese Edition-30.4 : 661, 2007.

[16] Lau, Alex, “Processing Frequent Updates with the TPR*-Tree Using Bottom-Up Updates,” Diss. Master’s Thesis, University of Waterloo, Ontario Canada N2L 3G1, 2005.

[17] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. In Proc. of the Workshop on Alg. Eng. and Experimentation, ALENEX, pages 178–193, Jan. 2002.
[18] Xiaopeng Xiong and Walid G. Aref, "R-Trees with Update Memos," Proceedings of the 22nd International Conference on Data Engineering, 2006.
[19] Yasin N. Silva, Xiaopeng Xiong, and Walid G. Aref, "The Rumtree: Supporting Frequent Updates in R-Trees Using Memos," VLDB J., 18(3):719.738, 2009.
[20] MoonBae Song and Hiroyuki Kitagawa, "Managing Frequent Updates in R-Trees for Update-Intensive Applications," IEEE Transactions on Knowledge and Data Engineering, Vol. 21, No. 11, pp. 1573-1589, 2009.
[21] S. W. Kim, S. C. Lim, "Active Adjustment: An Effective Method for Keeping the TPR*-Tree Compact," Journal of Information Science and Engineering, 2010.
[22] K. Kim, S. K. Cha, and K. Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," SIGMOD Rec., 30.2):139.150, 2001.