프리픽스 벡터를 사용한 균형 이진 IP 주소 검색 구조

준회원 김 형 기* 정회원 임 혜 숙**

Balanced Binary Search Using Prefix Vector for IP Address Lookup

Hyeong-gee Kim* *Associate Member*, Hye-sook Lim** *Regular Member*

요 약

인터넷 라우터는 라우팅 테이블로부터 입력 패킷의 목적지 주소에 맞는 다음 홉을 찾아 입력 패킷을 내보내는 패킷 포워딩을 수행하여야 한다. 그러나 링크 기술의 발달과 더불어 인터넷 사용자의 급증으로 인하여 입력되는 패킷과 같은 속도로 패킷을 처리하는데 어려움을 겪고 있다. 따라서 보다 빠른 주소 검색을 제공하기 위한 여러 이진 검색 알고리즘들이 제안되었다. 그러나 기존에 나와 있는 대부분의 이진 검색 알고리듬들의 구조는 불균형 구조로서, 메모리 접근 횟수가 많아 빠른 주소 검색이 어렵다는 단점이 있다. 균형 구조를 갖는 경우에는 프리픽스 복사로 인하여 원래 프리픽스 수보다 노드 수가 늘어나는 단점이 있다.본 논문에서는 균형 구조를 갖는 새로운 이진 검색 알고리즘을 제안한다. 제안된 구조에서는 디스조인트 (disjoint)한 프리픽스들 즉, 이진 트라이(binary trie)에서 리프 노드에 위치한 프리픽스들만으로 이진 검색 트리를 구성하고, 각 노드에 프리픽스의 네스팅 (nesting) 관계에 대한 정보를 담는 프리픽스 벡터 (prefix vector)를 두도록 하였다. 그러므로 실제 프리픽스의 수보다 훨씬 적은 수의 노드를 갖는 균형 이진 트리를 형성함으로서, 검색 성능을 매우 향상시킨 새로운 구조이다.

Key Words : IP Address Lookup, Balanced Binary Search, Prefix Vector

ABSTRACT

Internet routers perform packet forwarding which determines a next hop for each incoming packet using the packet's destination IP address. IP address lookup becomes one of the major challenges because it should be performed in wire-speed for every incoming packet under the circumstance of the advancement in link technologies and the growth of the number of the Internet users. Many binary search algorithms have been proposed for fast IP address lookup. However, tree-based binary search algorithms are usually unbalanced, and they do not provide very good search performance. Even for binary search algorithms providing balanced search, they have drawbacks requiring prefix duplication. In this paper, a new binary search algorithm which provides the balanced binary search and the number of its entries is much less than the number of original prefixes. This is possible because of composing the binary search tree only with disjoint prefixes of the prefix set. Each node has a prefix vector that has the prefix nesting information. The number of memory accesses of the proposed algorithm becomes much less than that of prior binary search algorithms, and hence its performance for IP address lookup is considerably improved.

Ⅰ. 서 론

최근 수년 사이 인터넷 사용자가 급증함에 따라 여러 인터넷 서비스 제공업체들이 서로 더 좋은 인터넷 서비스를 제공하기 위하여 경쟁하고 있다. 이에 따라 더 빠른 속도의 서비스를 보장하기 위해

※ 본 연구는 지식 경제부 및 정보통신 연구 진흥원의 대학 IT연구 센터(홈네트워크 연구 센터) 육성, 지원 사업의 연구 결과로 수행되었습니다.
* 이화여자대학교 대학원 전자정보통신학과 SoC연구실(calmsea@ewhain.net),
** 이화여자대학교 전자정보통신학과 부교수 (hlim@ewha.ac.kr)
논문번호 : KICS2008-01-013, 접수일자 : 2008년 1월 7일, 최종논문접수일자 : 2008년 5월 8일

많은 발전이 이루어져 왔다. 그러나 이러한서비스를 제공하기 위해서 링크 속도는 크게 증가한데 반해, 인터넷 라우터에서의 패킷 처리 즉, 링크 속도로 들어온 패킷에 대한 처리는 원활히 이루어 지지 못하고 있다.. 따라서 라우터의 성능이 이 패킷 처리 속도에 의해 제한되고 있다고 해도 과언이 아니다.

라우터의 패킷 처리 속도를 결정하는 가장 중요한 과정은 포워딩이라고 할 수 있다. 패킷 포워딩이란라우터에 저장된 전달 테이블(forwarding table)로부터 라우터에 들어온 패킷의 헤더 정보 중 목적지 주소(destination address)에 맞는 엔트리를 찾아 그 엔트리에 저장된 출력 포트로 패킷을 내보내는 일련의 과정을 말한다. 포워딩 속도는 들어온 패킷의 목적지 주소를 검색하는 IP 주소 검색 속도에 의존한다. 인터넷 사용자의 급증으로 점점 라우팅 테이블의 엔트리가 늘어나고 있어, 검색 속도가 입력되는 패킷과 같은 속도로 패킷을 처리하지 못하는 주요 원인으로 작용하고 있다. 따라서 인터넷 라우터 설계에 있어서 주소 검색 속도를 증가 시키는 것이 주요 과제라 할 수 있다.

인터넷 프로토콜(Internet protocol) 주소는 네트워크 부분과 호스트 부분의 구조로 되어 있다. 클래스를 갖는 주소 할당(class-based addressing) 방식에서는 네트워크 파트를 8, 16, 24 비트로 고정하고 확정 일치(exact match)를 사용하여 주소 검색이 가능하지만 IP 주소가 낭비되는 단점이 있다. 따라서 CIDR(classless inter-domain routing)과 같은 클래스를 갖지 않는 주소할당 방식이 등장하였다. 이 방식은 네트워크 파트인 프리픽스의 길이를 고정시키지 않고 임의로 정할 수 있으므로 IP 주소를 보다 효율적으로 사용할 수 있다. 그러나 CIDR 방식에서는 입력 패킷이 프리픽스 길이에 대한 정보를 가지고 있지 않으므로 목적지 주소와 가장 길게 매치하는 프리픽스를 찾아야 하는 LPM(longest prefix matching) 동작을 수행하여야 한다.

LPM에 의한 주소 검색을 위해 효율적인 IP 주소 검색 알고리즘이 필요하다. 먼저 주소 검색 알고리즘의 효율성 평가 기준을 고려해보면 검색 속도(search speed)와 메모리 요구량(storage requirement)을 대표적으로 들 수 있다. 여기서 검색 속도는 메모리 접근 횟수가 관건이라 할 수 있다. 이를 위해 이진 트라이(binrary trie), 멀티비트 트라이(multibit trie), PC 트라이(PC trie), LC 트라이(LC trie), BST(binary search tree), WBST(weighted binary search tree)등의 여러 구조들이 제안되었으나 이들의 구조는 불균형 (unbalanced) 구조이기 때문에, 트라이(trie) 혹은 트리(tree)의 깊이(depth)가 비효율적으로 커진다. 따라서 최악의 경우(worst-case) 경우, 이 깊이만큼 메모리 접근을 해야 하므로 검색 속도가 느려진다. 이러한 불균형 구조를 막기 위해 DPT(disjoint prefix tree)와 같은 새로운 구조가 제안되었으나 이것은 프리픽스 복사로 인해 트리의 깊이를 줄이는데 한계가 있다.

본 논문에서는 이와 같은 문제들을 해결하기 위한 새로운 알고리즘을 제안한다. 제안된 구조에서는 효율적인 주소 검색을 위해 프리픽스 벡터(prefix vector)를 사용하여 프리픽스간의 네스팅 관계를 표현하고, 디스조인트(disjoint)한 프리픽스들로만 이진 검색 트리를 구성하여 균형 검색 구조를 만들어 검색 성능을 크게 향상시켰다.

본 논문의 구성은 다음과 같다. 먼저 Ⅱ장에서 기존에 연구되어 온 알고리즘들을 간략하게 설명한다. Ⅲ장에서는 제안된 구조를 설명하고, Ⅳ장에서 시뮬레이션 결과로서 향상된 점들을 비교한다. 마지막으로 Ⅴ장에서 결론을 맺는다.

Ⅱ. 기존의 IP 주소 검색 알고리즘

프리픽스 검색(prefix lookup)을 위한 방법으로는 크게 비-알고리드믹(non-algorithmic)한것과 알고리드믹(algorithmic)한 것으로 나눌 수 있다. 먼저 비-알고리드믹 방법에는 캐싱(caching)과 TCAM(ternary content-addressable memories)을 이용한 방법이 있는데 캐싱은 locality가 나쁜 프리픽스들에 대해서는 현저히 낮은 성능을 보인다. TCAM은 입력된 패킷의 목적지 주소와TCAM에 저장된 모든 엔트리의 프리픽스들을 동시에 비교하여 가장 길게 매치한 엔트리의 주소를 선택하는 방식이다. 이것은 구현이 간단하고 빠른 주소 검색이 가능하지만 가격이 비싸고 전력소모가 매우 크며 면적을 많이 차지해 칩 내부에 내장하기 어려운 단점이 있다. 따라서 점점 커지는 데이터베이스들에 대해 한계가 있으므로 앞으로는 더 저렴하고 빠르며 고밀도(denser)의 SRA을 사용한 알고리즘적인 방법이 더 좋은 방안책이라 하겠다.

알고리즘을 이용한 프리픽스 검색 방법으로, 먼저 가장 기본적인 이진 트라이(binary trie)[1] 구조를 살펴본다. 이 구조의 각 노드는 0-포인터(pointer)와 1-포인터(pointer)를 가지고 있다. 루트(root) 노드(node)에서 출발하여 0-포인터가 가르키는 서브 트

표 1. 프리픽스 셋(set) 예시
Table 1. Example: Prefix Set

#Prefix	Prefix	Len	Out_ptr
P0	00*	2	0
P1	010*	3	1
P2	1*	1	2
P3	110100*	6	3
P4	110101*	6	4
P5	1101*	4	5
P6	111*	3	6
P7	11111*	5	7

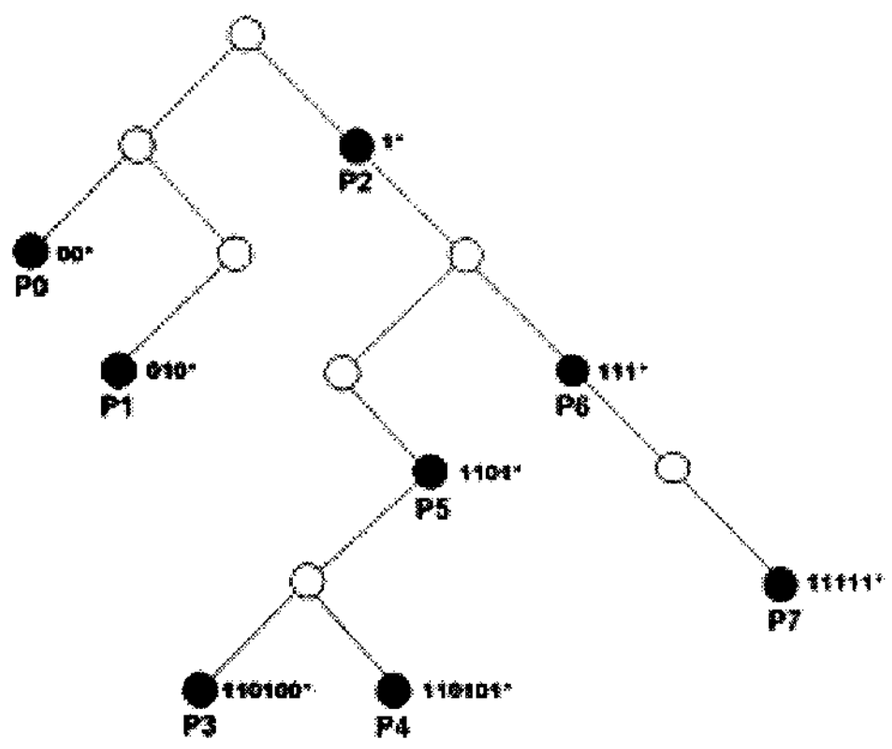


그림 1. 이진 트라이의 예시
Fig. 1. Example: Binary Trie

라이(sub-trie)는 0으로 시작하는 모든 프리픽스들을, 1-포인터가 가르키는 서브 트라이는 1로 시작하는 모든 프리픽스들을 갖게 되며, 프리픽스의 다음 비트에 대해서 같은 과정을 반복한다. 다음의 [표 1]은 주어진 프리픽스 셋(set)의 예이다. [그림 1]은 이 [표 1]의 프리픽스 셋으로 구성된 이진 트라이다. 그림에서 보듯이 이 구조는 프리픽스를 가지고 있지 않은 빈 노드들이 생겨 메모리를 낭비하고 트라이의 깊이가 커져 메모리 접근 횟수를 늘림으로써 검색 속도의 성능이 나빠지는 단점이 있다.

이와 같은 트라이의 깊이를 줄이기 위한 개선책으로 여러 구조들이 제안되었다. 그 중 하나로, 한 번에 한 비트씩 검사하지 않고 여러 비트를 비교하여 메모리 접근 횟수를 줄이는 멀티비트 트라이(multibit trie)[1]가 있다. [그림 2]는 [표 1]의 프리픽스 셋을 이용해 구현한 멀티비트 트라이 구조이다. 이 구조는 트라이의 깊이를 줄이는 반면, 프리픽스 복사로 인해 프리픽스 수가 늘어나는 단점을 갖는다. 다른 방법으로는 하나의 자식 노드를 갖는

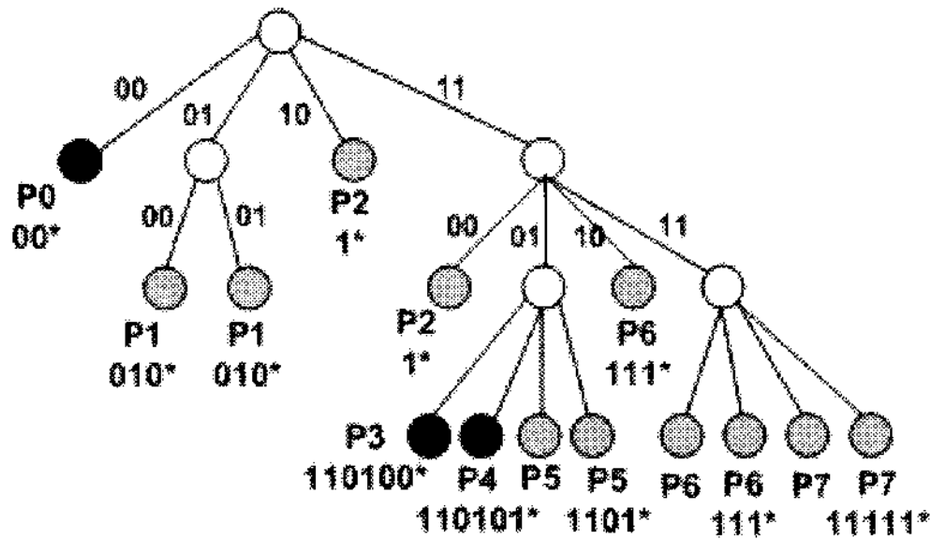


그림 2. 멀티비트 트라이의 예시
Fig. 2. Example: Multibit Trie

비어있는 내부 노드를 제거하여 빈 노드를 줄이는 PC 트라이(path-compressed trie) 구조[1]가 있다. 또한 이 두 가지 방법을 모두 적용한 LC 트라이(level-compressed trie)[2]가 있다. 그러나 이러한 구조들은 여전히 빈 노드들이 존재한다.

이러한 빈 노드들을 완전히 제거한 tree구조가 제안되었다. BST(binary search tree)[3]와 WBST(weighted binary search tree)[4]를 그 예로 들 수 있다. 이들 알고리즘에서는 길이가 서로 다른 프리픽스들에 대하여다음과 같은 정의를 사용한다.

두 프리픽스 A와 B의 길이를 각각 n, m이라 할 때,

정의 1) 비교(Compare)

두 프리픽스의 길이가 같은 경우 (n=m), A와 B의 수학적(numerical) 값이 비교된다. 두 프리픽스의 길이가 다른 경우(n>m), 짧은 프리픽스의 길이까지만 비교한다. 만약 두프리픽스의 서브 스트링(sub-string)이 같은 경우, 긴 프리픽스의 m+1 번째 비트가 1이면 A>B, 아니면 A<B이다.

예) A=100, B=101: B>A

예) A=11011, B=110: A<B

예) A=11011, B=101: A>B

정의 2) 매치(Match)

두 프리픽스가 같거나 짧은 프리픽스 길이까지가 같은 경우, 매치한다. 그렇지 않으면 두 프리픽스는 매치하지 않는다.

예) A=11011, B=110: A와 B는 매치한다.

예) A=11011, B=101: A와 B는 매치하지 않는다.

정의 3) 디스조인트(Disjoint)

두 프리픽스가 매치하지 않으면 A와 B는 디스조인트이다.

예) A=11011, B=101: A와 B는 디스조인트이다.

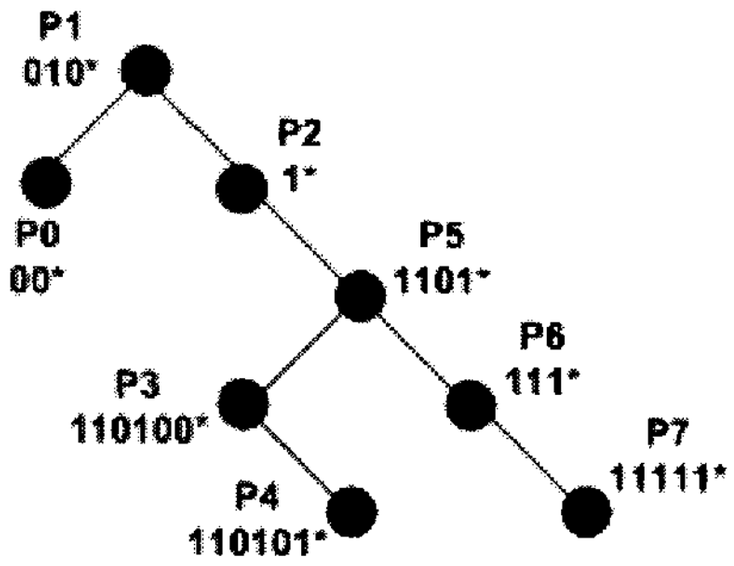


그림 3. BST(Binary Search Tree)의 예시
Fig. 3. Example: Binary Search Tree(BST)

정의 4) 인클로저(Enclosure)

A를 서브 스트링으로 갖는 다른 프리픽스가 하나라도 존재하면 A는 인클로저이다.

예) A=110, B=11011:A는 B의 인클로저이다.

이와 같은 정의를 토대로 트리 구조를 그리면 [그림 3]과 같다. 그림에서 보듯이 하나의 프리픽스가 다른 프리픽스의 인클로져가 되는 프리픽스 네스팅(nesting) 관계가 많으면 트리의 불균형이 심해지는 것을 알 수있다. 이는 LPM에 의해 프리픽스를 매치시켜야 하므로 트리에서 인클로저가 항상 인클로즈드(enclosed) 프리픽스보다 윗 레벨에 위치하여야 하기 때문이다. WBST는 이러한 불균형 트리를 좀 더 균형 트리로 만들어 트리의 깊이를 줄이기 위해 인클로즈드 프리픽스가 많은 인클로저 프리픽스를트리의 노드로 먼저 뽑히게 한다. 따라서 BST보다는 더 균형 구조를 이루지만 이 역시 네스팅 관계 때문에 여전히 불균형 구조라는 한계를 완전히 해결하지 못한다.

BST나 WBST에서는 이진 트라이(binary trie)나 멀티 비트 트라이(multibit trie), PC 트라이, LC 트라이와는 달리 빈 노드들을 없앴지만 네스팅 관계가 있는 프리픽스들 때문에 여전히 불균형 구조를 해결하지 못하는 단점이 있다. 따라서 최악의 경우(worst-case), 깊은 depth의 수만큼 메모리 접근을 해야 하기 때문에 검색 속도가 느려진다. 이를 보완하기 위해 균형구조인 DPT(disjoint prefix tree) 알고리즘[5]이 제안되었다. 이 데이터 구조는 네스팅 관계를 없애기 위해 프리픽스들을 리프 푸싱(leaf-pushing)을통해 모두 디스조인트 프리픽스들로 바꾸는 작업이 선행된다. 이 디스조인트한 프리픽스들로 BST 정의를 토대로 트리를 그리면 균형 구조가 된다. 이 프리픽스들은 더 이상 서브 트리(sub-tree)에 대한 포인터를 저장할 필요가 없고 이진 검색에 의해 검색(search)를 실행하면 된다. 그러나 리프 푸싱 과정에서 프리픽스 복사가 일어나 원래 프리픽스 수보다 더 많은 프리픽스들이 생겨나는 단점이 있다. [그림1]의 이진 트라이를 리프 푸싱을 하여 이진 트라이(disjoint binary trie)[6]를 만들면 [그림 4]와 같다. 이 디스조인트 프리픽스들로 트리를 구성하면 [그림 5]와 같이 균형 구조(DPT)를 이룬다.

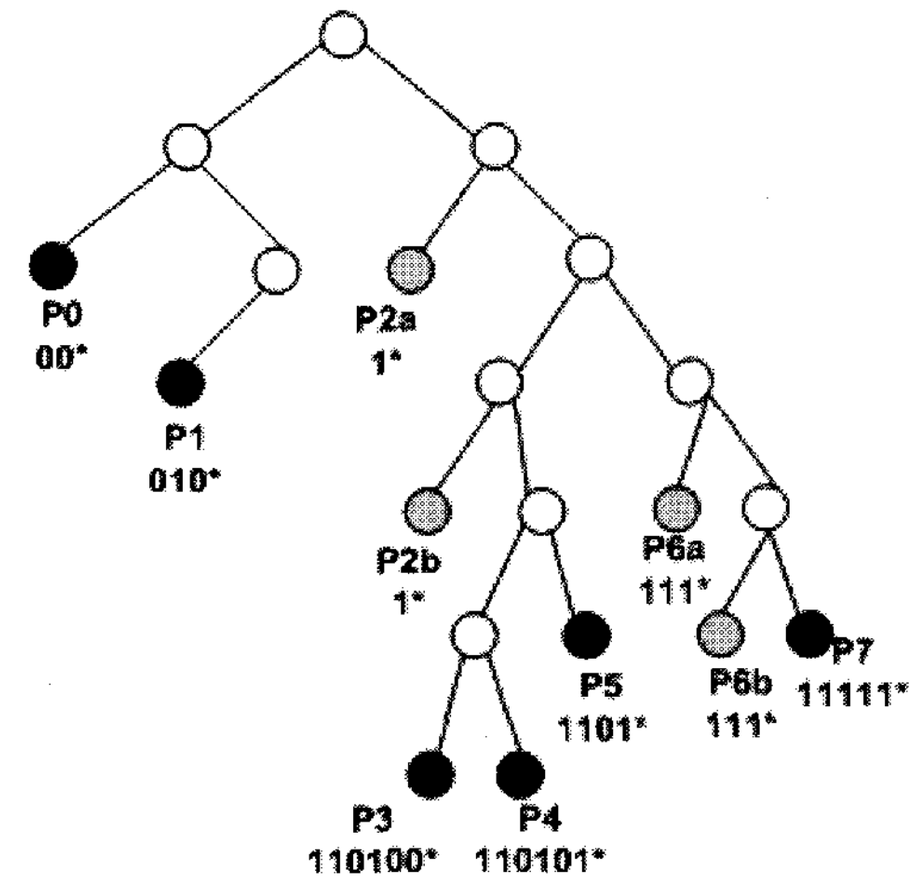


그림 4. 리프 푸싱을 한 이진 트라이의 예시
Fig. 4. Example: binary trie using leaf-pushing)

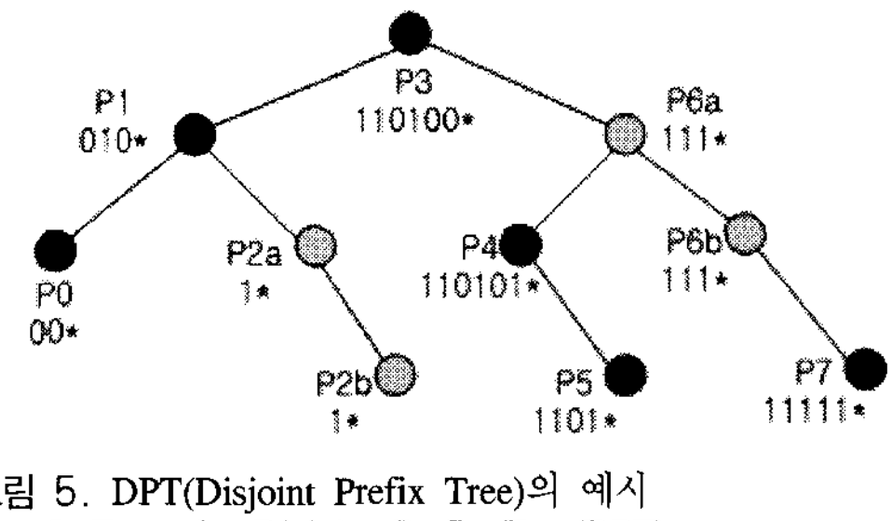


그림 5. DPT(Disjoint Prefix Tree)의 예시
Fig. 5. Example: Disjoint Prefix Tree(DPT)

Ⅲ. 제안된 IP 주소 검색 구조

앞서 본 알고리즘들은 트라이에 빈 노드가 존재하거나 불균형 구조를 하고 있어 그 깊이(depth)가 깊어진다. 따라서 주소 검색 시 최악의 경우, 그 깊이만큼 메모리 접근을 해야 한다. 이러한 점을 보완하기 위해 DPT 구조가 제안되었으나 이는 프리픽스 수가 늘어나는 단점이 있다. 따라서 본 논문에서는 리프 푸싱(leaf-pushing)을 하지 않고 프리픽스

벡터(prefix vector)를 이용하여 디스조인트 프리픽스들로 이루어진 균형 트리 구조를 만드는 보다 효율적인 알고리즘을 제안한다.

3.1 제안된 트리(tree) 구조

앞서 본 DPT에서는 리프 푸싱으로 디스조인트 프리픽스를 만드는 과정에서 프리픽스 복사가 일어나는 단점이 있다. 본 논문에서 제안되는 구조에서는 리프 노드에 있는 모든 프리픽스들은 서로 디스조인트하다는 점을 이용하여 이 프리픽스들로만 BST 정의를 토대로 트리를 구성한다. 따라서 훨씬 적은 수의 디스조인트 프리픽스들로 트리를 구성하므로 깊이가 훨씬 균형적이라는 장점을 살릴 수 있다. 이들을 제외한 나머지 프리픽스들의 정보는 프리픽스 벡터(prefix vector)를 사용해 각 노드에 싣는다. 즉, 이진 트라이의 리프(leaf) 프리픽스들로만 구성된 BST의 각 노드에 프리픽스 벡터를 두어 해당 프리픽스와 이 프리픽스의 인클로저가 되는 모든 프리픽스들의 정보를 담는다.

이 과정을 [표 1]의프리픽스 셋을 예로 들어 설명한다. 이 셋의 프리픽스는 최대 길이가 6이므로 프리픽스 벡터의 크기를 6바이트로 하여 설명한다. 프리픽스 P3의 110100*를 예로 들면 이진 트라이의 리프에 해당하므로 라우팅 테이블 엔트리의 프리픽스 부분에 110100*가 저장된다. 이 프리픽스의 인클로저가 되는 프리픽스들로 P2=1*와 P5=1101*가 있으므로 이 엔트리의 프리픽스 벡터는P2, P5, P3 프리픽스의 정보를 저장해야 한다. 즉, 110100*와 P2는 1번째까지, P5는 4번째까지, P3는 6번째까지 매치하므로 110100*를 프리픽스 정보로 갖는 엔트리의 프리픽스 벡터에는 1, 4, 6번째 bit에 각각 해당하는 출력포트 정보를 저장하다. 본 논문에서는편의상 프리픽스 번호를 저장하는 것을 가정하여 설명하면, 각각 2, 5, 3이 저장된다. 따라서 이 프리픽스에 해당하는 프리픽스 벡터를 보면 [그림 6]과 같다.

[그림 7]은 [그림 1]의 이진 트라이의 리프 프리픽스들로만 구성한 BST이다. 각 노드에서 윗줄에 적힌 숫자는 그 노드에 해당하는 프리픽스 번호이고 괄호안의 숫자들은 프리픽스 벡터에 적힐 프리

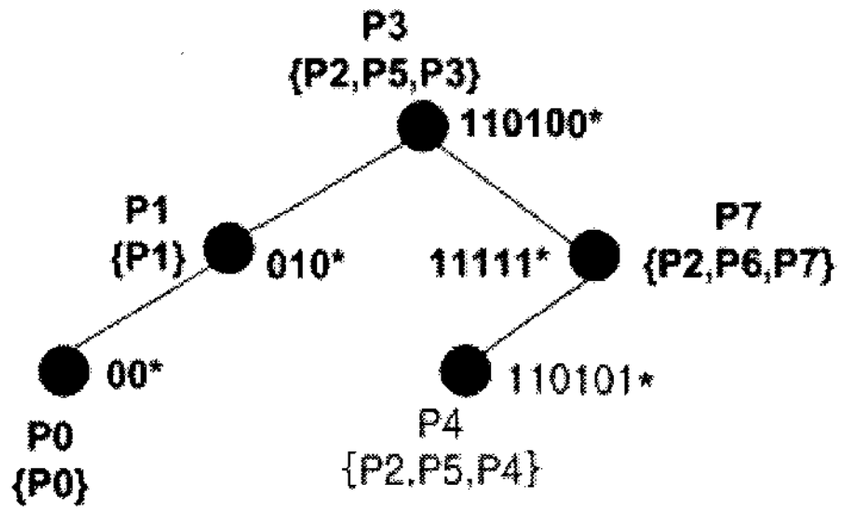


그림 7. 제안된 tree 구조 예시
Fig. 7. Example Proposed Tree Structure

픽스 번호들 즉, 그 노드 프리픽스 자신과 그것의 인클로저 번호들이다. 트리는 앞의 [그림 1]~[그림 5]의 구조와 비교하여 훨씬 적은 수의 디스조인트 프리픽스들로 이루어지므로 깊이(depth)가 작고 균형이라는 점을 볼 수 있다.

3.2 라우팅 테이블 구성(Build)

라우팅 테이블의 각 엔트리는 [그림 8]과 같이 구성된다. 즉, 리프 프리픽스로만 구성한 트리의 각 노드 프리픽스, 프리픽스 길이, 해당 노드의 프리픽스와 그 프리픽스의 인클로저가 되는 모든 프리픽스의 출력 포트 정보를 싣는 프리픽스 벡터로 구성된다. 이를 위해 각 노드의 프리픽스 벡터의 크기는 IPv4의 경우 각 길이 당 한 바이트씩을 할당하되, 최소 프리픽스의 길이는 8비트이므로, 길이 8부터 32까지의 25바이트를 할당하였다.

이 테이블을 구성하는 엔트리들은 다음과 같은 방법으로 형성된다. 각 엔트리의 프리픽스 부분은 이진 트라이에서 리프에 해당하는 프리픽스들만 BST의 정의에 의한 크기 순서로 정렬되어 저장된다. 각각의 프리픽스 벡터는 위에서 말한 바와 같이 해당 엔트리의 프리픽스와 이 프리픽스의 인클로저가 되는 프리픽스들의 정보를 모두 실어야 한다. 즉, 해당 엔트리의 프리픽스에 매치하는 데까지 자릿수를 기억하여 프리픽스 벡터의 해당 자릿수에 인클로저의 출력 포트 정보를 저장한다. [표 2]의 라우팅 테이블에서는 설명의 편의를 위하여 각 프리픽스의 출력 포트 번호를 그것의 프리픽스 번호와 같게 하였다. 이와 같은 과정에 의해 [표 1]의 프리픽스들에 대하여 [그

Prefix	Len	Prefix vector					
110100*	6	2	-	-	5	-	3

그림 6. [표 1]의 110100*의 Prefix Vector 예시
Fig. 6. Example Prefix Vector for Entry Having Prefix 110100*

4bytes	1bytes	25bytes
Prefix	Length	Prefix Vector

그림 8. 라우팅 테이블의 엔트리 구조
Fig. 8. Entry Structure of Routing Table

표 2. 라우팅 테이블의 예시
Table 2. Example: Routing Table

Prefix	Len	Prefix vector					
00*	2	-	0	-	-	-	-
010*	3	-	-	1	-	-	-
110100*	6	2	-	-	5	-	3
110101*	6	2	-	-	5	-	4
11111*	5	2	-	6	-	7	-

림 7]을 이용해 제안된 알고리즘의 라우팅 테이블을 간략하게 나타내면 [표 2]와 같다.

3.3 검색(Search)

제안된 알고리즘은 앞서 본 다른 알고리즘들과 달리 라우팅 테이블의 엔트리에 모든 프리픽스를 저장하는 것이 아니라 프리픽스 벡터에 모든 프리픽스 정보를 싣는다. 이러한 점 때문에 라우팅 테이블의 엔트리에 대하여 LPM을 수행하던 이전 알고리즘의 검색 과정을 그대로 따르지 않는다.

이 데이터 구조의 검색은 프리픽스 벡터도 함께 고려되어야하며 검색 과정에서 현재까지의LMP과 함께 현재까지 가장 길게 일치한 프리픽스의 길이를 나타내는 LLM을 기억하여야 한다. 각 검색 과정에서 검색할 엔트리의 순서는 이진 검색 순서를 따른다.

검색 과정은 [그림 9]와 같은 과정을 따른다. 먼저 설명을 위해 들어온 패킷의 목적지 주소를 A라 하고 이진 검색(binary search) 순서에 따라 비교할 순서의 현재 엔트리 프리픽스를 B라고 가정한다.

[그림 9]에서 보이는 바와 같이 검색을 마치는 조

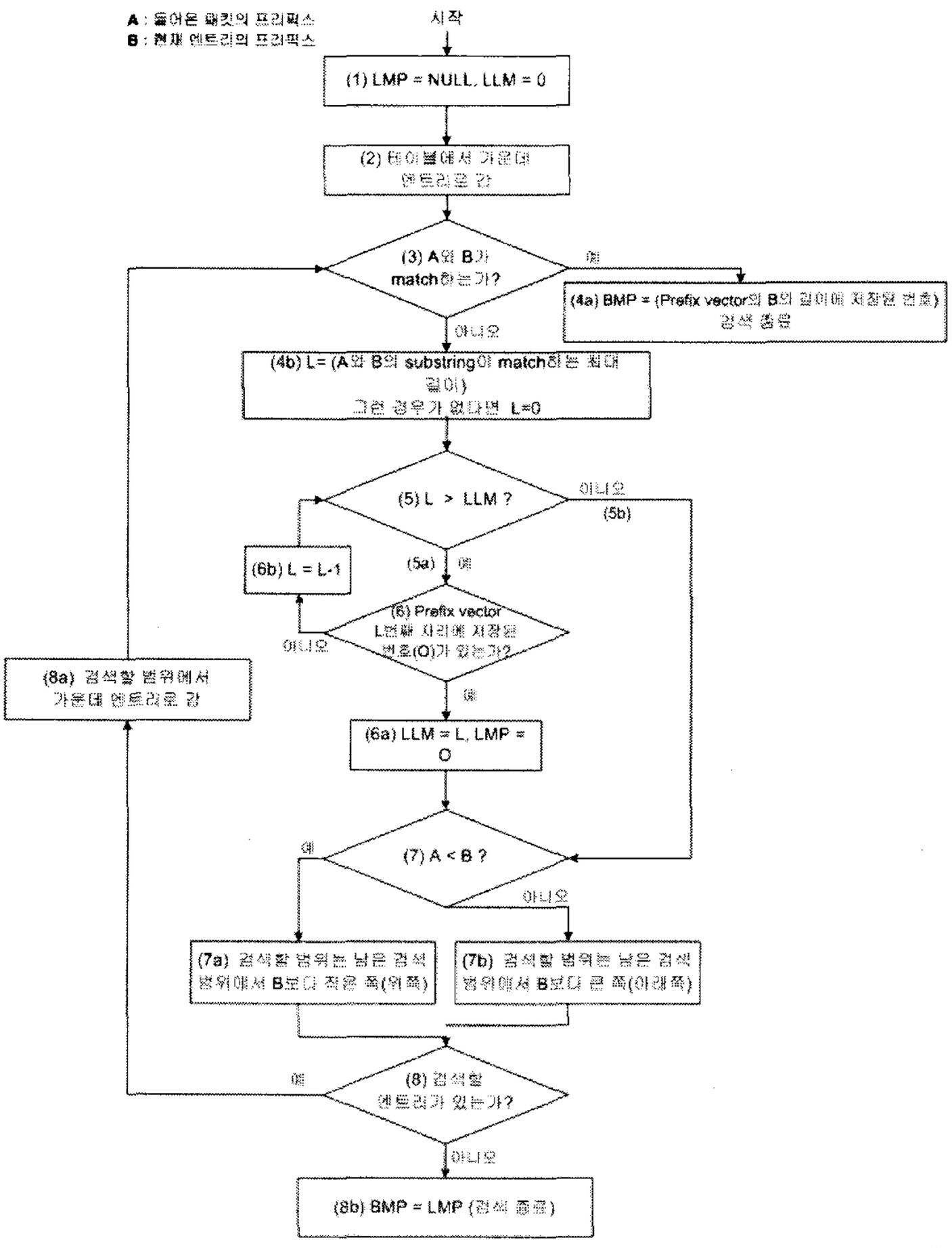


그림 9. 제안된 구조의 검색 과정
Fig. 9. Searching Process of the Proposed Scheme

건은 두 경우로 나눌 수 있다. 즉, 들어온 패킷의 목적지 주소와 현재 엔트리에서의 프리픽스가 매치하는 경우(검색 종료 Case 1)와 가능한 모든 엔트리에서의 검색이 수행된 경우(검색 종료 Case 2)이다.

[표 2]의 라우팅 테이블을 토대로 들어온 패킷의 목적지 주소, 즉 A가 111110인 경우를 예로 설명하면, 먼저(1)LMP=NULL, LLM=0이라 초기화한다. (2)이진 검색 순서에 따라 테이블의 가운데 엔트리 즉, 110100*(B)를 프리픽스로 하는 엔트리로 간다. (3)A와 B가 매치하지 않으므로 (4b) A와 B의 서브스트링이 매치하는 최대 길이 L=2가 저장된다. 이는 A와 B가 매치하지 않더라도 않더라도 B의 인클로저가 되는 프리픽스와 매치할 수 있으므로 프리픽스 벡터를 검사해야 하기 때문이다. 따라서 B의 서브스트링과 A가 매치하는 길이를 기억한다. (5a)현재 LLM이 0으로 초기화되어 있기 때문에 L>LLM이다. 따라서 이 경우, 현재 엔트리에 LLM보다 더 길게 매치하는 프리픽스가 있는지 검사해야 한다. 가장 뒷자리에 저장된 프리픽스 번호(P)가 가장 길게 매치하는 프리픽스 정보이므로 뒷자리부터 검사한다. (6)프리픽스의 2번째 자리에 프리픽스 정보가 없으므로 (6b)에서 볼 수 있듯이 더 짧은 길이 1번째 자리로 가서 (5a)~(6)다시 정보가 있는지 검사하고 이 자리에 정보가 있으므로 (6a)더 이상 이 프리픽스 벡터의 검사를 진행할 필요가 없다. 그 결과, 현재 엔트리에서 가장 길게 매치하는 프리픽스 정보는 LMP=2, LLM=1로 저장된다. (7)현재 엔트리에서의 검사가 끝났으므로 다음 엔트리에 대해 검사를 지속한다. 그러므로 이진 검색 순서에 따라 A>B이므로 다음 검색 범위는 현재 엔트리인A의 아래쪽이 된다. (8) 검색 범위에 남아 있는 엔트리들이 있으므로 가운데 엔트리 11111*로 간다. 이 프리픽스가 다시 B가 되고 위의 검색 과정을 반복한다. (3) A와 B가 매치하므로 다른 엔트리에는 매치하는 프리픽스가 없음을 의미한다. 지금까지 찾은 LMP는 B의 인클로저일 것이다. (4a)따라서 지금까지의 LMP를 무시하고 현재 엔트리의 프리픽스 벡터에서 B의 길이에 저장된 번호가 BMP이며 검색을 종료한다. 즉, 5번째에 7이 저장되어 있으므로 BMP=7로 기억하고 검색을 종료한다. 최종 검색 결과는 11111*의 출력 포트 7이다.

두번째 경우로서, 들어온 패킷의 목적지 주소 A가 110110인 경우를 예로 들어 설명하면, (1)LMP, LLM을 초기화한 다음 (2)먼저 테이블의 가운데 엔트리인 110100*(B)를 프리픽스로 하는 엔트리에서 검색을 시작한다. (3)~(4b)A는 B의 서브스트링 1101*와 매치하므로 L=4가 되고 (5a)~(6a)프리픽스벡터의 4번째 자리부터 프리픽스 정보 유무를 검사한다. 그 결과, 4번째 자리의 프리픽스 정보를 LMP=5, LLM=4로 저장한다. (7b)~(8a)그 다음, 11111*를 프리픽스로 하는 엔트리에서 검색을 지속한다. (3)~(4b)이 경우 A와 매치하는 프리픽스 정보는 길이 1인 P2번 프리픽스 1*이다. (5b)이는 현재 엔트리에서 현재까지 기억된 LMP보다 길게 매치하는 프리픽스 정보가 없음을 의미하므로 프리픽스 벡터를 검사하지 않는다. (7a)~(8a)따라서 다음 검색할 엔트리로 넘어간다. 프리픽스가 110101*인 엔트리에서 검색이 진행되고 이 경우도 마찬가지로 더 길게 매치하는 프리픽스 정보가 없으므로 LMP, LLM을 업데이트하지 않는다. 이 엔트리의 검사가 끝났으므로 다음 엔트리로 넘어가야 하나 (8b)더 이상 검색할 엔트리가 남지 않았으므로 가능한 모든 엔트리를 검사하였다고 볼 수 있다. 따라서 검색은 종료되고 최종 검색 결과는 저장된 LMP가 된다. 즉, BMP=5로 프리픽스 1101*의 출력포트 5이다.

3.4 업데이트(Update)

프리픽스를 업데이트하는 경우는 크게 삽입(insert)과 삭제(delete)로 나누어볼 수 있다.

3.4.1 삽입 (insert)

새로운 프리픽스를 기존의 프리픽스 셋에 삽입하려는 경우, 크게 두 가지로 나눠 볼 수 있다. 즉, 그 삽입하려는 프리픽스가 라우팅 테이블의 프리픽스들과 디스조인트한 경우(삽입 Case 1)와 그렇지 않은 경우(삽입 Case 2)이다.

먼저 첫번째 경우로서, 삽입하려는 프리픽스가 라우팅 테이블의 프리픽스들과 디스조인트한 경우, 이것을 프리픽스로 하는 새로운 엔트리를 추가하고 그에 맞는 프리픽스 벡터 정보를 저장한다. 이 경우, 삽입하려는 프리픽스가 이진 트라이에서 리프에 해당하는 프리픽스이므로 라우팅 테이블의 프리픽스 크기 순서에 맞는 자리에 새로운 엔트리로 추가된다. 이 엔트리의 프리픽스 정보에는 이 삽입 프리픽스가 저장되고 프리픽스 벡터에는 이 프리픽스의 인클로저 정보가 담겨야 한다. 따라서 다음과 같은 과정을 따른다.

첫째, 삽입 프리픽스가 기존의 라우팅 테이블 엔트리의 프리픽스들 중 가장 긴 길이까지 같은 프리픽스를 찾아 이 프리픽스와 매치한 길이와 이 프리픽스가 저장된 엔트리를 기억한다. 둘째, 기억된 엔

트리의 그 길이까지의 프리픽스 벡터를 삽입한 엔트리의 프리픽스 벡터로 복사한다. 셋째, 삽입 프리픽스 자신의 출력 포트 번호를 해당 프리픽스 벡터 자리에 저장한다. 이 과정을 통해 새로운 엔트리의 프리픽스 벡터는 삽입 프리픽스 자신을 포함한 그것의 인클로저 정보를 모두 가지게 된다.

예를 들면, 출력 포트 번호가 8인 11110*를 삽입하려는 경우, 테이블의 모든 엔트리 프리픽스들과 디스조인트하므로 110101*와 11111* 사이에 새로운 엔트리를 추가한다. 가장 긴 길이까지 같은 프리픽스가 11111*로 4번째까지 일치하므로 이 프리픽스 엔트리의 4번째까지의 프리픽스 벡터를 삽입 엔트리의 프리픽스 벡터로 복사한다. 그 다음, 삽입 프리픽스의 출력 포트 번호를 5번째 자리에 저장한다. 결과적으로 이 엔트리의 프리픽스 벡터는 2, 6, 8번 프리픽스 정보를 저장하게 된다.

두번째 경우로서, 삽입하려는 프리픽스가 라우팅 테이블의 프리픽스들과 디스조인트하지 않은 경우, 삽입하려는 프리픽스가현재 라우팅 테이블의어느 프리픽스들의 인클로저일 때(Case 2-1)와인클로즈드일 때(Case 2-2)로 나눌 수 있다.이 두 경우 모두 새로운 엔트리는 생겨나지 않는다.

먼저 (Case2-1)로서 삽입하려는 프리픽스가 현재 라우팅 테이블에서 어느 프리픽스들의 인클로저인 경우 다음 과정을 따른다.

첫째, 테이블에서 그 삽입 프리픽스가 인클로저가 되는 프리픽스를 가진 엔트리들을 찾는다. 둘째, 그 엔트리들의 프리픽스 벡터의 해당 자리 즉, 삽입하려는 프리픽스 길이에 해당하는 자리에 이것의 출력 포트 번호를 삽입한다.

예를 들면, 출력 포트 번호가 110*를 추가하려는 경우 새로운 엔트리는 생기지 않고 110100*와 110101*를 프리픽스로 하는 엔트리에 대해 프리픽스 벡터 3번째 자리에 8이라는 정보를 삽입한다. 이 결과 이들의 프리픽스 벡터는 각각 2, 8, 5, 3과 2, 8, 5, 4의 프리픽스 정보를 갖게 된다

두번째로 (Case 2-2)로서 삽입하려는 프리픽스(A)가 현재 라우팅 테이블에서 어느 프리픽스의 인클로즈드(B)인 경우, A는 binary trie에서의 새로운 리프가 되어야 하고, B는 이것의 인클로저가 되어야 하므로 다음과 같은 과정을 수행한다.

첫째, 테이블의 프리픽스들 중 삽입하려는 프리픽스가 인클로즈드 프리픽스가 되는 엔트리를 찾는다. 단, 테이블의 프리픽스들은 서로 디스조인트하므로하나의 엔트리만 해당할 것이다. 둘째, 이 엔트리의 프리픽스를 삽입 프리픽스로바꾼다. 셋째, 이 엔트리의 프리픽스 벡터에 삽입 프리픽스의 출력 포트 번호를 해당 자리에 추가한다.

이를 통해 삽입하려는 프리픽스 정보와 이프리픽스의 인클로저가 되는 프리픽스들의 정보를 저장하게 된다. 예를 들어 출력 포트 번호가 8인 0101*를 삽입하려는 경우 (Case 2-1)과 마찬가지로새로운 엔트리는 생겨나지 않는다. 대신 기존의 010*를 프리픽스로 하는 엔트리에 대해 프리픽스를0101*로 바꾼다. 다음 이 엔트리의 프리픽스 벡터 4번째 자리에 8을 삽입한다. 이 결과, 이 엔트리는0101*를 프리픽스로 하고 프리픽스 벡터에는 1,8을 저장한다.

3.4.2 삭제 (delete)

라우팅 테이블에서 프리픽스를 삭제하려는 경우에는 삭제하려는 프리픽스가 어느 프리픽스들의 인클로저인 경우(삭제 Case 1)과 삭제하려는 프리픽스가 라우팅 테이블의 프리픽스인 경우(삭제 Case 2)로 나누어 생각하여 볼 수 있다.

먼저 (삭제 Case 1)을 살펴보면, 즉 삭제하려는 프리픽스가 라우팅 테이블의 어느 프리픽스들의 인클로저인 경우에는 해당 엔트리들의 프리픽스 벡터에서 삭제하려는 프리픽스 번호를 삭제한다.

삭제하려는 프리픽스가 라우팅 테이블의 프리픽스인 경우, 이는 다시 두 가지 경우로 나뉜다. 그 프리픽스 엔트리의 프리픽스 벡터에 자신의 정보 외에다른 프리픽스의 번호가 저장된 경우(삭제 Case 2-1)와 그렇지 않은 경우(삭제 Case 2-2)이다.

먼저 (삭제Case 2-1)로서 해당 엔트리의 프리픽스 벡터에 자신의 정보 외에 다른 프리픽스의 번호가 저장된 경우, 프리픽스 벡터에 다른 프리픽스 정보까지 삭제되지 않도록 해야 한다. 따라서 다음 과정을 따른다.

첫째, 삭제하려는 프리픽스의 엔트리 프리픽스 벡터에서 자신의 정보를 제외한 다른 프리픽스의 정보(A)가 저장된 가장 긴 길이를 찾는다. 둘째, 다른 엔트리에 그 A의 정보가 들어 있는지 여부에 따라 두 가지 경우로 나눈다. 해당 엔트리에서 그 길이까지의 프리픽스 서브스트링이 다른 엔트리 프리픽스와 매치하는 경우, 삽입 프리픽스 엔트리 전체를 삭제한다. 다른 엔트리에서 A가 저장되어 있기 때문이다. 그렇지 않은 경우 즉, 해당 엔트리에서 그 길이까지의 프리픽스 서브스트링이 다른 엔트리 프리픽스들과 디스조인트한 경우, 다른 엔트리에 A가 없으므로 이를 현재 엔트리에서 유지해야 한다. 따라서

삭제하려는 프리픽스의 엔트리에 이 서브스트링을 새로운 프리픽스로 저장하고, 프리픽스 벡터에서 삭제하려는 프리픽스 번호를 삭제한다. 이는 B가 새로운 이진 트라이에서 리프가 되기 때문이다.

다음은 (삭제 Case 2-2)로서, 그 프리픽스 엔트리의 프리픽스 벡터에 자신의 정보 외에 다른 프리픽스의 번호가 저장되어 있지 않은 경우, 삭제하려는 프리픽스가 저장된 엔트리 전체를 삭제한다.

Ⅳ. 시뮬레이션 결과 및 성능 평가

본 논문에서는 제안된 알고리즘의 성능을 분석하기 위해, 2006년 5월에 실제 백 본 라우터로부터 내려 받은 몇 개의 데이터베이스 W1, AADS, ,E1, PORT80, GROUPTLCOM, TELSTRA에 대해 알고리즘을 수행하였다[8].

제안된 알고리즘의 시뮬레이션 결과는 [표 3]와 같다. [표 3]에서 Tavg는 평균 메모리 접근 횟수, Tmax는 최대 메모리 접근 횟수, Tmin은 최소 메모리 접근 횟수를 의미한다.

[표 3]에서 리프 비율(leaf rate)은 각각의 데이터베이스에서 주어진 프리픽스 수에 대한 이진 트라이의 리프에 해당하는 디스조인트 프리픽스들의 비율이다. 이 비율이 작다는 것은 라우팅 테이블의 엔트리 수가 작다는 것을 의미한다. 이 프리픽스를 제외한 나머지 프리픽스들의 정보는 프리픽스 벡터에 저장 되고 더 적은 프리픽스들로 구성된트리의 깊이는 현저히 작아진다. 다시 말해서, 검색 속도를 결정하는데 있어서 관건인 메모리 접근 횟수를 줄일 수 있다는 것이다.

[표 4]는 제안된 알고리즘의 메모리 접근 횟수를 다른 알고리즘들의 것과 비교하여 나타낸 값이다. 이 표에서 보듯이 리프 비율이 작은 데이터베이스의 경우, 제안된 알고리즘의 검색 속도 성능이 크게 향상된다. 이는 제안하는 알고리즘이 확장성(scalability)이 뛰어나 현재 점점 더 커지는 데이터베이스들에 대한 처리율이 더 높을 것이라는 가능성을 보인다.

[그림 10]은 시뮬레이션 결과, 여러 알고리즘들의 평균 메모리 접근 횟수를 비교한 그래프이다. 제안된구조는 평균 검색 속도에 있어 LC-Trie다음으로 우수한 성능을 보여주고 있다. LC-Trie의 경우는 [표 5]의 각 알고리즘들의 메모리 요구량에서 보여

표 3. 제안된 알고리즘의 시뮬레이션 결과
Table 3. Simulation Result of the Proposed Algorithm

Input	Prefix 갯수	Leaf 갯수	Leaf rate (%)	Tavg	Tmax	Tmin	Memory (KByte)
W1	14553	14288	98.2	12.9	14	1	418.6
AADS	20204	19832	98.2	13.4	15	1	581.0
W2	29584	27622	93.4	13.9	15	1	809.2
E1	39464	38316	97.1	14.3	16	1	1122.5
PORT80	112310	70418	62.7	15.3	17	1	2063.0
GROUPTLCOM	170601	106959	62.7	16.0	17	1	3133.6
TELSTRA	227223	162103	71.33	16.52	18	1	4749.11

표 4. 여러 알고리즘들과 평균 메모리 접근 횟수 비교
Table 4. Comparison in the average number of memory accesses

Input	Leaf Rate (%)	Prop	B-trie	BST	WBST	P-trie	LC-trie	BSR
W1	98.18	12.87	22.87	14.13	13.89	16.7	2.9	14.21
W2	93.37	13.86	23.08	15.55	14.96	18.2	4.3	15.20
E1	97.09	14.32	23.17	15.79	15.36	18.2	3.2	15.67
PORT80	62.70	15.33	22.15	25.96	20.58	20.4	10.6	16.75
GROUPTLCOM	62.70	15.95	22.31	27.02	21.69	20.8	11.0	17.28
TELSTRA	71.33	16.52	24.64	30.77	23.91	22.9	11.2	17.64

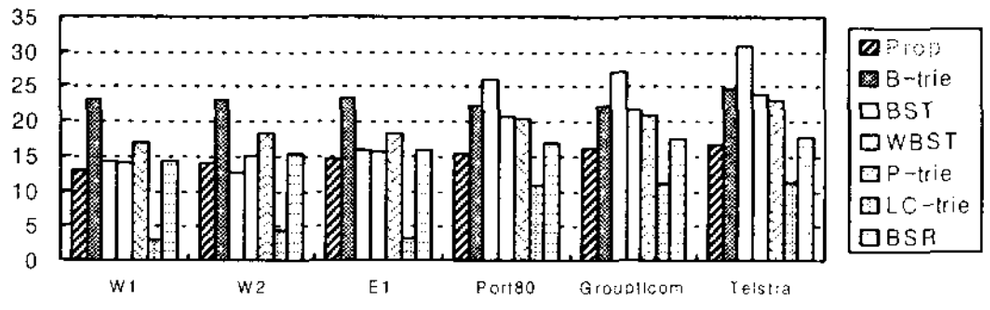


그림 10. 여러 알고리즘들의 평균 메모리 접근 횟수
Fig. 10. Comparison in the average number of memory accesses

주겠지만 지나치게 많은 메모리를 요구하고 구현이 복잡하여, 그 검색 성능이 매우 뛰어남에도 불구하고, 사용되지 못하고 있다.

[표 3]에서 보인 메모리 요구량은 각 데이터베이스에 대해 프리픽스 벡터를 최대치인 25bytes로 동일하게 하여 전체 메모리 요구량을 계산한 결과이다. [표 5]에서는 프리픽스당 요구하는 메모리의 크기, 즉 전체 요구되는 메모리 량을 프리픽스의 개수로 나눈 값을 사용하여 여러 알고리듬의 메모리 요구량을 비교하여 보았다. 본 논문에서 제안된 알고리듬은 프리픽스 벡터로 인하여 다른 이진 검색 알고리듬에 비교하여 조금 큰 메모리를 요구하지만, LC-Trie와 비교할 때, 월등히 적은 메모리를 요구하는 것을 볼 수 있다. 현재 라우터에서 패킷 전달이 링크 속도와 스위칭 속도의 증가만큼 빠르게 수행되고 있지 못한다는 점을 고려하여 볼 때 주소 검색 속도가 메모리 요구량보다 더 중요한 문제라 판단되며, 본 논문에서 제안하는 구조는 LC-Trie보다 월등히 적은 메모리를 사용하여, LC-Trie에 근접하는 검색 성능을 내는 구조로서, 매우 효율적인 구조라고 할 수 있다.

V. 결 론

본 논문에서는 인터넷 라우터의 성능이 링크 스피드(link speed)에 미치지 못하는 패킷 처리 속도에 의해 제한을 받고 있다는 점을 중요하게 보고 이를 향상시키기 위한 알고리즘을 제안하였다. 패킷 처리 속도를 결정하는 가장 큰 요인으로 라우팅 테이블에서의 주소 검색 속도를 들 수 있다. 현재 통용되고 있는 TCAM에 의한 라우팅은 구현이 간단하고 빠른 주소 검색이 가능하지만 가격이 비싸고 전력소모가 매우 크며 칩 내부에 내장하기 어렵고 IPv6로 확장하기 힘든 단점이 있다. 따라서 점점 더 커지는 라우팅 테이블의 정보를 처리하기 위해 알고리즘적인 방안을 모색하였다.

제안된 알고리즘은 검색 속도를 향상시키는데 주력하여 메모리 접근 횟수를 성능의 주요 척도로 삼아 기존의 알고리즘들과 비교, 분석하였다.기존의 알고리즘들은 불균형 구조를 하고 있어 완전균형 이진검색이 불가능하였고 트리의 깊이가 커져 최악의 경우 메모리 접근 횟수가 매우 커져 검색 속도가 매우 느려진다. 이러한 단점을 보완하기 위해 제안된 DPT구조는 원래 프리픽스 수보다 프리픽스가 많아져 트리의 깊이를 줄이는데 한계가 있다.

따라서 본 논문에서 제안된 알고리즘은 프리픽스 수를 늘리지 않으면서 디스조인트 프리픽스들로만 균형 트리를 구성한다. 이 프리픽스들을 제외한 나머지 즉, 트리를 구성하는 프리픽스들과 네스팅 관계가 있는 나머지 프리픽스들의 정보를 각 노드의 프리픽스 벡터에 싣는다. 이러한 과정으로 인해 더 적은 프리픽스들로 트리를 구성하여 깊이를 훨씬 줄일 수 있다. 즉, 더 적은 엔트리로 구성된 라우팅 테이블에서 이진 검색이 가능해진다. 또한 디스조인트 프리픽스들로만 엔트리를 구성하고 이들과 네스팅 관계에 있는 프리픽스들의 정보는 프리픽스 벡터에 저장하기 때문에 네스팅 관계가 많은 데이터베이스에 대해 테이블의 엔트리 수는 상대적으로 적어지는 장점이 있다. 인터넷 사용자가 많아짐에 따라 데이터베이스의 크기가 커지고 있으며, 데이터베이스의 크기가 커질수록 프리픽스들 간에 네스팅 관계 또한 증가하고 있다. 따라서 제안된 알고리즘은 큰 데이터베이스에 대해 훨씬 우수한 성능을 보

표 5. 여러 알고리즘들과 프리픽스 당 메모리 요구량 비교(Byte)
Table 5. Comparison in storage requirement per prefix (Byte)

Input	Prop	B-trie	BST	WBST	P-trie	LC-trie	BSR
W1	24	31	10	10	10	440	12
W2	23	22	10	10	10	159	12
E1	24	26	10	10	10	213	12
Port80	15	12	10	10	10	65	9
Grouptlcom	15	11	10	10	10	48	9
Telstra	17	12	10	10	10	464	9

일 것으로 예측된다.

제안된 알고리즘은 검색 속도가 프리픽스 길이에 의존하지 않고 라우팅 테이블의 엔트리 수에 의해 결정되므로, 따라서 이는 IPv6에 대해서도 좋은 성능을 유지할 것으로 보인다. IPv6의 경우 프리픽스 벡터가 지나치게커짐으로 메모리 요구량에 있어 나쁜 영향을 미칠 것으로 판단되는데, 이러한 문제는 프리픽스 그루핑[7]을 통하여 프리픽스 벡터의 크기를 줄임으로서 해결이 가능하다.

참 고 문 헌

〔1〕 Miguel A. Ruiz-Sanchez, Ernst W. Biersack, "Survey and Taxonomy of IP Address Lookup Algorithms," IEEE Network, Vol.15, No.2, pp.8-23, March 2001.

〔2〕 S. Nilsson, G. Karlsson, "IP Address Lookup Using LC-Tries," IEEE Journal on Selected Areas in Communications, Vol.17, No.6, pp. 1083-1092 June 1999

〔3〕 N. Yazdani, P.S. Min, "Fast and Scalable Schemes for the IP Address Lookup Problem," IEEE HPSR2000, pp.83-92, 2000

〔4〕 C. Yim, H. Lim, and B. Lee, "Weighted Binary Prefix Tree for IP Address Lookup," International SoC Design Conference, October 2004.

〔5〕 Lim, W. Kim, B. Lee, "Binary Search in a Balanced Tree for IP Address Lookup," IEEE HPSR2005, pp.490-494, May 2005.

〔6〕 Srinivasan, G. Varghese, "Fast Address Lookups using Controlled Prefix Expansion," Proceedings of ACM Sigmetrics'98, pp.1-11, June 1998.

〔7〕 H. Lim, H. Kim, and Y. Jung, "Parallel multiple hashing for packet classification," Proc. IEEE HPSR2005, pp. 104-107, May 2005.

〔8〕 http://www.potaroo.net.

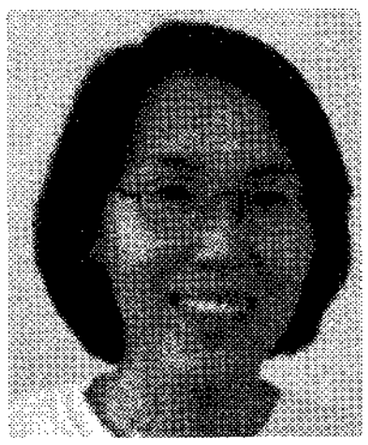
김 형 기 (Hyeong-gee Kim) 준회원

2007년 2월 이화여자대학교 정보통신학과 졸업

2007년 3월~현재 이화여자대학교 전자정보통신학과 석사과정

<관심분야> Router나 switch 등의 Network 관련 SoC 설계

임 혜 숙 (Hye-sook Lim) 정회원



1986년 2월 서울대학교 제어계측공학과 졸업

1986년 8월~1986년 2월 삼성 휴렛 팩커드 연구원

1991년 2월 서울대학교 제어계측공학과 석사

1996년 12월 The University of Texas at Austin, Electrical and Computer Engineering 박사

1996년 11월~2000년 7월 Lucent Technologies Member of Technical Staff

2000년 7월~2002년 2월 Cisco Systems, Hardware Engineer

2002년 3월~현재 이화여자대학교 공과대학 전자정보통신학과 부교수

<관심분야> Router나 switch 등의 Network 관련 SoC 설계, TCP/IP 관련 하드웨어 설계