제 26 회 한국정보처리학회 추계학술발표대회 논문집 제 13 권 제 2 호 (2006. 11)

FP-tree 연관 규칙 탐사 알고리즘의 구현 및 성능 특성

이 형 봉
강릉대학교 컴퓨터공학과
e-mail : hblee@kangnung.ac.kr

An Implementation and Performance Characteristics of the FP-tree Association Rules Mining Algorithm

Hyung-Bong Lee
Dept. of Computer Science & Engineering, Kangnung National University

요 약

FP-tree(Frequent Pattern Tree) 연관 규칙 탐사 알고리즘은 DB 스캔에 대한 부담을 획기적으로 절감시킴으로써 전체적인 성능을 향상시키고자 제안되었다. 그런데, FP-tree 는 DB 에 저장된 거래 내용 중 빈발 항목을 포함하는 모든 거래를 트리에 저장해야 하기 때문에 그만큼 많은 메모리를 필요로 한다. 이 논문에서는 범용 운영체제인 유닉스 시스템을 사용해서 메모리 사용 측면에서 F.P. Tree 알고리즘의 타당성과 이에 따른 성능 특성을 관찰하였다. 그 결과, F.P. Tree 알고리즘은 현대 컴퓨터에서 보편화된 512MB~1GB 의 주메모리 시스템에서 무리는 없으나, 메모리 소요량이 DB 의 크기나 빈발 항목 집합의 수 보다는 거래의 길이 등 DB 의 특성에 따라 급격하게 증가하는 것으로 나타났다.

1. 서론

가장 알기 쉬운 연관 규칙 탐사 알고리즘은 전체 항목 집합에 대한 모든 부분 집합에 대응되는 계수 공간을 마련한 후, DB 에 저장된 각각의 거래를 읽어 거기에 포함된 항목들로 구성 가능한 항목 집합에 대한 계수를 실시하는 것이다. 이를 흔히 원시 알고리즘이라 하는데, 이 알고리즘은 다음과 같이 두 가지 큰 문제점을 안고 있다.

첫 번째 문제점은 많은 양의 계수 공간을 필요로 한다는 점이다. 예를 들어 전체 항목 집합의 크기 $|I|$ 가 5,000 개라면 약 $2^{5,000}$ 개의 계수 공간이 필요한데 이를 지원할 수 있는 시스템은 흔하지 않다.

두 번째 문제점은 계수 공간이 마련되었다 하더라도 대응되는 계수 지점을 찾기 위해서는 많은 시간이 소요된다.

위와 같은 근본적인 문제점을 해결하고, 보다 나은 성능을 얻기 위해 연관 규칙 탐사 알고리즘은 꾸준히 진화해 왔다. *AIS(finding All frequent Item-Stes)* 알고리즘[1]은 탐색을 단계별로 진행하되, 현재까지의 빈도 통계를 적용하여 전 단계의 빈발 항목 집합으로부터 후보 빈발 항목 집합을 도출하여 다음 단계를 결정함으로써 한꺼번에 필요로 하는 계수 공간을 줄였고, *Apriori* 알고리즘[2]은 빈도 통계 대신 각 단계별 빈발 항목 집합의 길이를 1, 2, 3, ...과 같이 정확히 1 씩 늘려가면서 후보 빈발 항목 집합을 유도하여 규칙적인 단계로 진행하여 실행 시간을 단축시켰다. *DHP(Direct Hash and Pruning)* 알고리즘[3]은 직접 해시 테이블(direct hash table)이라는 사전 전지 정보를 활용하여 후보 빈발 항목 집합의 수를 줄임으로써 계수 공간 및 검색 시간을 단축하였을 뿐만 아니라 거래 중에서 빈발이 아닌 항목들을 제거해 나감으로써 다음 단계에서의 DB 크기를 줄였다.

DB 스캔 횟수 측면에서 원시 알고리즘은 단 1 회면 충분하나, *AIS, Apriori, DHP* 알고리즘은 최대 빈발 항목 집합의 크기 만큼의 DB 스캔 횟수가 필요하기 때문에 소요 기간의 많은 부문이 DB 입·출력에 소비된다.

FP-tree 알고리즘[4]은 지금까지의 알고리즘들이 필요로 했던 여러 번의 DB 스캔 횟수를 단 2 회로 단축시킴으로써 DB 입·출력 부담을 획기적으로 줄였다.

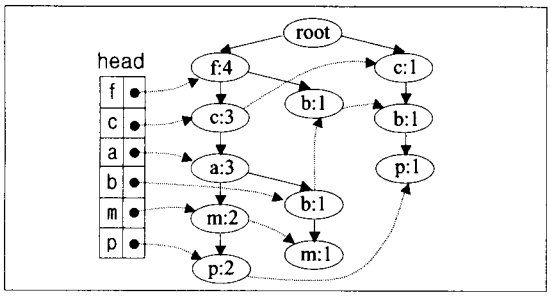
이 알고리즘은 단계 1 에서 빈발 항목이 아닌 항목들을 제거하고, 단계 2 에서 빈발 항목들로만 구성된 거래들을 모두 FP-tree 구조에 저장한 다음, 단계 3 에서 FP- tree 로부터 빈발 항목 집합을 유도해 낸다.

즉, *FP-tree* 알고리즘은 DB 의 거래 내용을 필터링한 모든 거래 내용을 메모리에 구축한 후 그 곳으로부터 빈발 항목 집합을 추출한다. 따라서, *FP-tree* 알고리즘은 다른 알고리즘보다 많은 양의 메모리를 필요로 하지만, 전체적인 성능은 우수한 것으로 알려져 있다. 이 논문은 *FP-tree* 알고리즘을 C 언어로 구현하여 실행하고, 사용 시스템의 메모리 크기에 따라 그 성능이 어떤 추이를 보이는가를 관찰한다.

2. FP-tree 알고리즘의 구현

2.1 FP-tree 자료구조

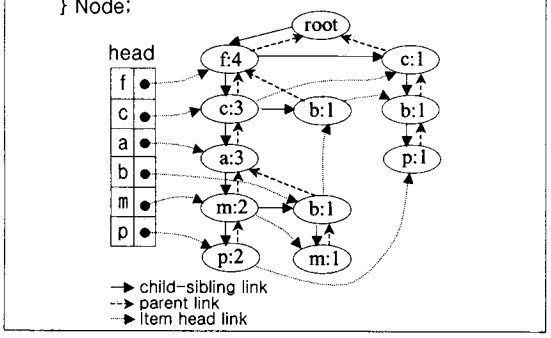
(그림 1)에 [4]에서 제시한 FP-tree 자료구조를 보였다. 이 그림에서 트리의 각 노드는 root 로부터 출발하는 트리 고유의 링크(실선)와 header table 로부터 출발하는 해당 항목 링크(점선) 등 두 개의 링크를 유지한다.



(그림 1) FP-tree 자료구조

아래의 (그림 2)에는 (그림 1)을 위한 C 언어 코드와 구현된 링크 모습을 보였다. 여기서 sibling link 는

```
typedef struct  node {
    u_short     count, item_name;
    u_short     flag;    /* tag root etc,... */
    struct node  *clink;  /* child link */
    struct node  *plink;  /* parent link */
    struct node  *slink;  /* sibling link */
    struct node  *ilink;  /* item head link */
} Node;
```



(그림 2) (그림 1)의 FP-tree 의 구현 자료구조

보모가 동일한 노드들을 연결하고, child link 는 자식 노드들 중 첫 노드를 연결한다. Parent link 는 자신의 부모 노드를 연결하고, 따라서 형제 노드들의 parent link 는 모두 동일하다.

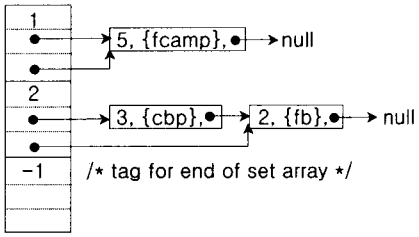
2.2 패턴과 패턴 집합 자료구조

(그림 3)에 패턴(빈발 항목 집합)을 저장하기 위한 패턴과 패턴 집합의 구현된 자료구조를 보였는데, 이 그림의 패턴 집합 배열은 길이 5 인 패턴 {fcamp}, 길이 3 인 패턴 {cbp}, 길이 2 인 패턴 {fb} 등 총 3 개의 패턴을 저장하고 있다. (그림 4)에는 이 자료구조를 바탕으로 구현된 패턴 연산 프로시저 원형을 보였다.

```
typedef struct  pattern {
    struct pattern  *next;  /* pattern link list */
    u_short     item_len;  /* pattern length */
    u_short     item_name[1];   /* item list */
} Pattern;

typedef struct  patternset {
    short       nset;       /* # of t patterns */
    struct pattern  *first;  /* front,pattern list */
    struct pattern  *last;   /* rear,pattern list */
} PatternSet;
```

pattern set array



(그림 3) 패턴과 패턴 집합의 구현 자료구조

```
MergePattern(Pattern *fp1,  Pattern *fp2)     {
 /* 패터 fp1항목들과 패턴 fp2의 항목들을 결합하고,
    중복된 항목은 제거하며, 빈도에 따라 정렬한 후,
                새로운 패턴으로 생성하여 반환함 */
}
InsertPatternToSet(PatternSet *ps,
                                    Pattern *fp) {
 /* 패터 fp를 패턴집합 ps에 추가함에 추가함 */
}
InsertPatternToSetArray(PatternSet *psarray,
                                PatternSet *ps) {
 /* 패터집합 ps에 있는 모든 패턴들을
                패턴집합 배열 psarray에 추가함 */
}
PatternSet *PatternSetUnion(PatternSet *ps1,
                                PatternSet *ps2) {
 /* 두 패턴집합 배열 ps1, ps2를 합친 새로운
                패턴집합 배열을 생성하여 반환함 */
}
PatternSet *PatternSetCross(PatternSet *ps1,
                                PatternSet *ps2) {
 /* 두 패턴집합 배열 ps1, ps2에 포함된 양쪽 모든 패
    턴들이 조합된 패턴집합 배열을 생성하여 반환함 */
}
```

(그림 4) 주요 패턴 관련 구현 연산 프로시저

2.3 FP-tree 생성 알고리즘

(그림 5)에는 [4]에서 제시한 "Algorithm 1(*FT-tree construction*) insert()" 알고리즘을 (그림 2)와 (그림 3)에 구현한 자료구조를 바탕으로 구현한 C 코드를 보였다. [4]에서는 표현 [*p*|*P*] 즉, 거래의 항목들을 빈도에 따라 정렬한 패턴 중 앞 쪽의 한 항목 *p* 와 그 나머지 항목들로 구성된 패턴 *P* 라는 표현을 (그림 5)의 구현 알고리즘에서는 정렬된 전체 패턴 fp 에서 ci 번째 항목과 그 이후의 항목들로 표현하였다. 또한 [4]에서는 FP-tree 를 단순히 root 로 표시하였지만 실제로는 (그림 2)에서 보는 바와 같이 항목 링크를 유지하는 head 도 필요하므로 (그림 5)의 구현에서는 head 가 추가되었다.

```
insert(Pattern *fp,  int ci,
                    Node *root, Node *head[])
{
   Node  *tp, *np;
   int     citem;
   if (ci == fp->len) return;
   else  citem = fp->item_name[ci];
   for(tp=NULL, np=root->clink; np != NULL;
                        tp=np, np=np->slink) {
     if (np->item_name == citem) {
       np->count++;
       insert(fp, ci +1, np, head);
       return;
     }
   }
   np = get_node();  /* allocate a new node */
   np->item_name = citem;  np->count = 1;
   if (tp) {tp->slink=np; np->plink=tp->plink; }
   else   {rt->clink=np; np->plink=rt; }
   for(tp=NULL,  hp=head[citem]; hp != NULL;
                        tp=hp, hp=hp->ilink) ;
   if (tp) tp->ilink = np;
   else   head[citem] = np;
   insert(fp, ci +1, np, head);
}
```

(그림 5) insert() 알고리즘의 구현

2.4 FP-tree 탐색 알고리즘

FP-tree 탐색 알고리즘은 (그림 5)에서 구축된 FP-tree 를 대상으로 모든 빈발 항목 집합(frequent pattern set)을 추출한다. [4]에서 제시한 FP-tree 탐색 알고리즘 "Algo- rithm 2(*Mining frequent patterns*) FP_growth()"의 주요 부분을 구현한 내용을 (그림 6)에 보였다. 즉, 이 알고리즘에서의 핵심은 single prefix path 와 multi path 부분을 어떻게 식별할 것인가 하는 문제인데, 이는 (그림 6)에 보인 바와 같이 root 의 직계 자손 중 slink(sibling link)가 없는 노드까지를 설정하면 얻을 수 있다. 이와 같이 얻어진 single prefix path 의 끝 노드는 multipath part 의 null 노드 역할을 겸하게 되고, 이를 표시하기 위해 해당 노드의 flag 를 1 로 설정한다. (그림 4)에 보인 PatternSetCross(ps1, ps2) 프로시저는 패턴 집합 ps1, ps2 를 보존하는 반면, PatternSetUnion(ps1, p2)는 패턴 집합 ps1 과 ps2 를 파괴하고 새로운 패턴 집합을 생성하여 반환한다.

```
PatternSet *FP_growth(Node *root, Node *head[],
                                  Pattern *alpha)
{
   Node *sp, *mp;
   PatternSet  *P=NULL, *Q=NULL, *R;
   if (!root->clink->slink) { /* single prefix path */
     for(sp=root,mp=sp->clink;mp;mp=mp->clink)
       if (mp->slink)
         break;     /* end of single prefix part */
     if (mp) {
       mp = mp->plink;    /* null root node for  */
       mp->flag = 1;      /* multipath part      */
     }
     P = grow_single_part(sp, alpha);
     if (mp)
       Q = grow_multi_part(mp, head, alpha);
   }
   else {                    /* multipath part only */
     root->flag = 1;
     Q = grow_multi_part(root, head, alpha);
   }
   R = PatternSetCross(P, Q);
   R = PatternSetUnion(R, P);
   R = PatternSetUnion(R, Q)
   return(R);
}
```

(그림 6) FP_growth() 알고리즘의 분리부분 구현

3. FP-tree 알고리즘의 성능 특성 분석

3.1 성능 특성 분석 환경

구현된 FP-tree 알고리즘의 성능을 분석하기 위하여 <표 1>의 시스템을 사용하였고, [5]에서 제공된 시험용 데이터 생성 프로그램을 활용하여 <표 2>의 데이터들을 성능 분석에 사용하였다. 이 때, 데이터는 {거래 길이, 항목 1, 항목 2, ... 항목 n} 형태의 레코드로 구성된 파일로 구성하였다.

<표 1> 성능 측정 시스템 환경

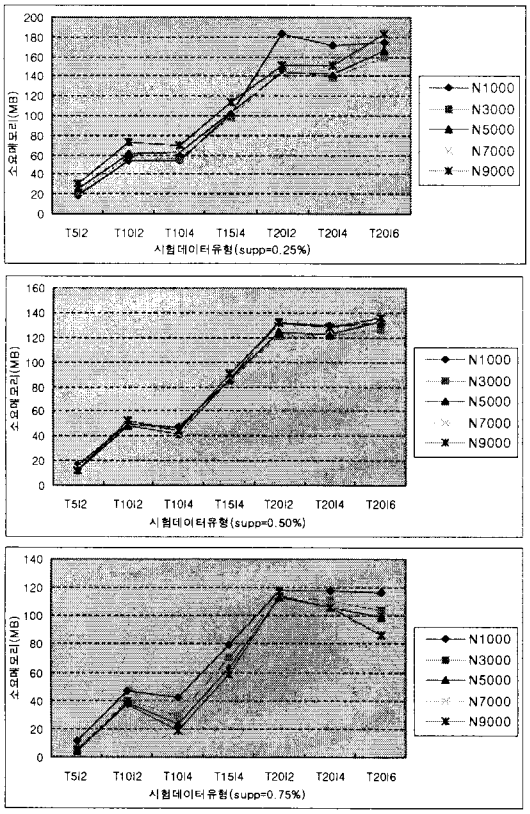
항 목	사 양
모 델	COMPAQ WS au600
C P U	Alpha Ev67 667MHz
메 모 리	1GB/512MB
운 영 체 제	Digital Tru64UNIX 4.0F

<표 2> 시험 데이터의 유형

분 류 기 준	분 류 내 용
데이터 특성	T5I2D100, T10I2D100, T10I4D100, T15I4D100, T20I2D100, T20I4D100, T20I6D100
전체 항목 개수(I/I)	N1000, N2000, N3000, N5000, N7000, N9000

3.2 성능 특성 측정

(그림 7)에 <표 1>, <표 2>의 환경에서 실시한 FP-tree 알고리즘의 메모리 소요 성능 측정 결과를 보였다. 이 그림으로부터 FP-tree 알고리즘은 대체적으로, 거래의 평균 길이와 빈발 항목 집합의 평균 길이가 길수록 소요 메모리가 급격하게 증가한다는 사실을 알 수 있다. 즉, DB 의 크기는 거래의 길이에 따라 어느 정도 증가할 수 밖에 없으나 이를 처리하기 위한 알고리즘의 요구 메모리는 그 보다 훨씬 가파른 메모리 증가율을 보인다는 것이다. 따라서 거래의 길이나 빈발 항목 집합의 길이가 극단적을 큰 경우에는 [4]에서 언급한 바와 같이 B+-tree 구조를 바탕으로 하는 disk-resident FP-tree 개념의 도입이 불가피하고 이는 결국 또 다른 DB 스캔 부담을 유발한다..

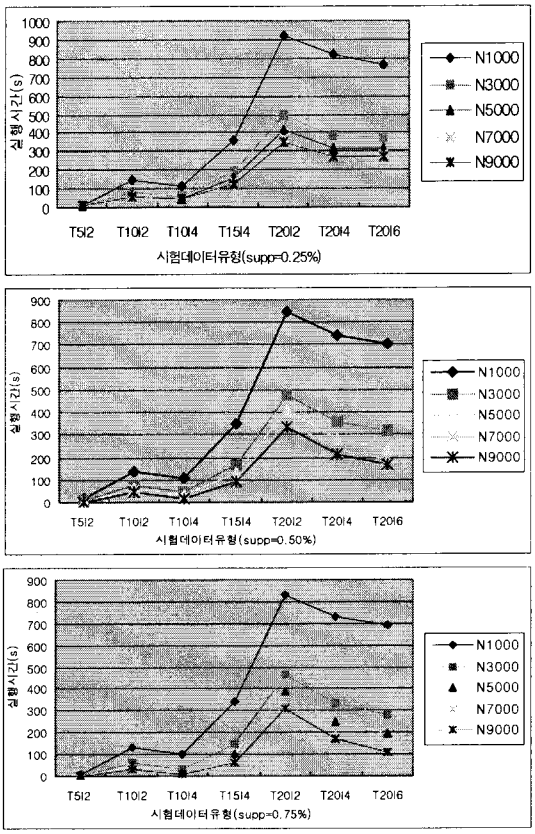


(그림 7) FP-tree 알고리즘의 메모리 성능 특성

(그림 8)에는 (그림 7)과 동일한 데이터에 대한 FP-tree 알고리즘의 실행 시간 성능의 측정 결과를 보였다. 이 그림으로부터 알고리즘의 실행 소요 시간이 요구 메모리와 반드시 비례하지 않음을 알 수 있는데, 이는 (그림 5)에서 작성한 FP-tree 가 어떤 경우에는 효과적이지 못할 수도 있음을 의미한다.

4. 결론

[4]는 *Apriori* 알고리즘에 대한 FP-tree 알고리즘



(그림 8) FP-tree 알고리즘의 실행시간 성능 특성

의 성능이 월등히 우수한 것으로 결론짓고 있으나, 앞에서 보인 (그림 7), (그림 8)에서 그 비교 결과는 메모리 등 처리 시스템의 환경이나 DB 의 특성에 따라 달라질 수 있는 가능성이 존재함을 보였다. 즉, 특정 연관 규칙 탐사 알고리즘에게 어떤 상황에서도 적용될 수 있는 절대적 우수성은 주어지지 못할 것이다.

참고문헌

[1] R. Agrawal, T. Imielinski and A. Swami, "*Mining Association Rules between Sets of Items in Large Databases*", Proceedings of ACM SIGMOD on Management of Data, pp.207~216, 1993.

[2] R. Agrawal and R. Srikant, "*Fast Algorithms for Mining Association Rules*", Proceedings of the 20th International Conference on Very Large Databases, pp.487-499, 1994.

[3] J. S. Park, M.-S. Chen and P. S. Yu, "*An Effective Hash-Based Algorithm for Mining Association Rules*", Proceedings of ACM SIGMOD, pp.175-186, 1995.

[4] Jiawei Han, Jian Pei, and Yiwen Yin, "*Mining frequent patterns without candidate generation*", Proceedings of 2000 ACM SIGMOD Int. Conf. Management of Data(SIGMOD'00), Dallas, TX, pp. 1-12

[5] R. Agrawal and et al, "*Synthetic Data Generation Code for Associations and Sequential Patterns*", http://www.almaden.ibm.com/cs/quest, 1999.