최소 완전 해쉬 함수를 위한 선택-순서화-사상-탐색 접근 방법

(A Selecting-Ordering-Mapping-Searching Approach for Minimal Perfect Hash Functions)

이 하 규 †

(Hagyu Lee)

요 약 본 논문에서는 대규모 정적 탐색키 집합에 대한 최소 완전 해쉬 함수(MPHF: Minimal Perfect Hash Function) 생성 방법을 기술한다. 현재 MPHF 생성에서는 사상-순서화-탐색(MOS: Mapping-Ordering-Searching) 접근 방법이 널리 사용된다. 본 연구에서는 MOS 접근 방식을 개선하여, 보다 효과적으로 MPHF를 생성하기 위해 선택 단계를 새로 도입하고 순서화 단계를 사상 단계보다 먼저 수행하는 선택-순서화-사상-탐색(SOMS: Selecting-Ordering-Mapping-Searching) 접근 방법을 제안한다. 본 연구에서 제안된 MPHF 생성 알고리즘은 기대 처리 시간이 키의 수에 대해 선형적인 확률적 알고리즘이다. 실험 결과 MPHF 생성 속도가 빠르며, 해쉬 함수가 차지하는 기억 공간이 작은 것으로 나타났다.

Abstract This paper describes a method of generating MPHFs(Minimal Perfect Hash Functions) for large static search key sets. The MOS(Mapping-Ordering-Searching) approach is widely used presently in MPHF generation. In this research, the MOS approach is improved and a SOMS(Selecting-Ordering-Mapping-Searching) approach is proposed, where the Selecting step is newly introduced and the Orderng step is performed before the Mapping step to generate MPHFs more effectively. The MPHF generation algorithm proposed in this research is probabilistic and the expected processing time is linear to the number of keys. Experimental results show that MPHFs are generated fast and the space needed to represent the hash functions is small.

1. 서 론

최소 완전 해쉬 함수(MPHF: Minimal Perfect Hash Function)는 N개의 탐색키(search key)를 최소의(minimal) 기억 공간인 N개의 슬롯(slot)에 충돌(collision)이 없이, 즉 완전한(perfect) 형태로 저장하여 검색할 수 있는 일종의 해쉬 함수이다. 일반적인 정보 검색을 비롯하여 CD-ROM 타이틀 검색, 자연어 처리 등의 분야에서 사용되는 탐색키는 대규모이고, 검색이 진행되는 동안에는 삽입, 삭제 등의 연산이 일어나지 않는, 즉 키 집합 자체에는 변화가 없는 정적(static)인 경우가 많다. 이러한 대규모 정적 탐색키 집합의 경우 미리 MPHF를 생성하여 검색에 활용하면 시간과 공간적인 측면에서 검색 효율을 극대화시킬 수 있다.

지금까지 MPHF 생성을 위해 수 이론에 기초한 방법(number theoretical methods), 키 집합의 분할에 기초한 방법(methods based on segmentation), 탐색 공간 제한에 기초한 방법(methods based on restricting the search space), 희소 행렬 압축에 기초한 방법(methods based on sparse matrix packing) 등 다양한 방법들이 시도되어 왔다[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. 이 중에서 FCH 방법이라고 알려진 Fox, Chen, Heath가 1992년에 발표한 방법[4]은 탐색 공간 제한에 기초한 방법으로서, 수백만 개 이상의 대규모 키 집합에 대해서도 MPHF 생성이 가능하고, 문자열 형태의 키도 처리 가능할 뿐만 아니라, MPHF가 차지하는 기억 공간이 작

† 종신회원 : 성공회대학교 컴퓨터정보학부 교수
hglee@green.skhu.ac.kr
논문접수 : 1998년 9월 16일
심사완료 : 1999년 11월 9일

다는 장점이 있다.

FCH 방법에서는 Cichelli, Cercone 등에 의해 제안된 MOS(Mapping-Ordering-Searching) 접근 방식을 취하고 있다[1, 2, 4, 10]. MOS 접근 방식에서는 MPHF 생성의 성공 가능성과 생성 속도를 높이기 위해 크게 사상(mapping), 정렬(ordering), 탐색(searching)의 3단계로 나누어 MPHF 생성을 진행한다. 사상 단계에서는 주어진 키의 모집단 세계를 수(number) 형태로 표현되는 새로운 세계로 변환하는 사상이 수행된다. 정렬 단계에서는 주어진 키 집합을 마지막 단계인 탐색 단계에서 동시에 해쉬 값이 결정되는 키의 부분 집합인 키 레벨(level)들로 분할하고, 이들 키 레벨을 그 크기에 따라 정렬한다. 탐색 단계에서는 각 키 레벨에 대해 충돌이 없는 해쉬 값을 찾아 부여한다.

이 외에도 FCH 방법에서는 문헌 [3], [5], [9], [11] 등에서 제시된 기존의 기법들이 이용되고 있는데, 그 중에서 중요한 것을 살펴보면 다음과 같다. 평균적인 MPHF 생성 성능을 향상시키기 위해, 사상 및 정렬 단계에서 임의성(randomness)을 활용하는 확률적(probabilistic) 알고리즘을 적용한다. 키 레벨 크기의 분산 특성을 고려하여 정렬 단계에서 힙(heap)이라는 자료구조를 이용하여 정렬 시간을 단축시킨다(3.2 참고). MPHF의 탐색 속도와 성공 가능성을 높이기 위해 탐색 단계에서 크기가 큰 키 레벨부터 탐색하는 휴리스틱(heuristic) 탐색 기법을 취한다(3.4 참고). 이 밖에도 FCH 방법에서는 키 집합의 분할 기법이 도입된 MPHF 모형을 정립하여, MPHF 생성의 성공 가능성과 기억 공간 효율의 향상을 꾀하고 있다(2장 참고).

FCH 방법은 대규모 키 집합의 경우 지금까지 제시된 MPHF 생성 방법 중에서 기억 공간이 가장 작은 MPHF를 생성할 수 있는 방법이다. 그리고 비록 엄밀한 증명은 주어져 있지 않지만 이론적 고찰과 실험 결과에 근거하여 기대(expected) 처리 시간이 키 집합의 크기에 대해 선형적(linear)인 알고리즘으로 평가되고 있다[4, 6]. 하지만 MPHF 생성 속도나 검색시의 MPHF 계산 속도는 빠른 편이 아니다.

본 연구에서는 FCH 방법의 개선을 통하여 MPHF 생성과 계산 속도 향상을 시도하였다. 이를 위해 계산 속도가 빠른 MPHF 모형을 개발하였으며, 생성 과정 초기에 충돌이 없는 키의 부분 집합을 미리 걸러내는 선택(Selecting) 단계를 새로 도입하고, 정렬 단계를 사상 단계보다 먼저 수행하는 SOMS(Selecting-Ordering-Mapping-Searching) 접근 방식의 MPHF 생성 알고리즘을 개발하였다.

2. MPHF 모형

MPHF 모형의 기술을 위해 본 논문에서는 다음과 같은 표기법을 사용한다.

- . U : 탐색키의 집합
- . I : 0 이상의 정수 집합
- . N : 키 집합의 크기 (N = |U|)
- . K : 임의의 탐색키
- . K[i] : 키 K의 i번째 문자
- . |K| : 키 K의 길이
- . L : 키의 최대 길이
- . C : 문자 코드의 수
- . M : 레벨 오프셋 함수 G의 크기
- . RG : G의 비율 (RG = M/N)

여기서, U는 주어진 탐색키의 모집단 세계(universe)로서, 일반성을 위해 모든 키는 문자열 형태를 취하는 것으로 가정한다. I는 문자열 형태의 키가 사상되는 수(number) 세계의 집합으로서 0 이상의 정수 값으로 구성된다. K[i]는 키 K에서 i번째 문자 혹은 그 코드(code) 값을 나타내는데, 첫번째 문자는 K[0]이고 키의 최대 길이는 어떤 상수 L 이하인 것으로 가정한다. 키를 구성하는 문자의 기억 장소 크기는 1바이트이며 그 코드 값은 [0, 255] 범위의 값을 지니는 것으로 가정한다. 따라서 문자 코드의 수 C는 256을 넘지 않는다. MPHF 문제에서는 보통 이러한 가정을 도입하고 있는데, 이로 인해 실제 응용에서 키의 표현이 제약을 받는 경우는 거의 없다.

M은 나중에 정의될 레벨 오프셋 함수 G의 크기를 나타내는데, MPHF의 저장 공간과 밀접한 관련성이 있다. RG는 모형 매개변수로서 키 집합의 크기 N에 대한 함수 G의 크기 M의 상대적인 비율을 나타낸다. RG를 작게 설정하면 MPHF가 차지하는 기억 공간은 줄어들지만, MPHF의 생성 속도가 저하되고 MPHF 생성이 실패할 가능성이 많아진다. 대규모 키 집합의 경우 RG는 일반적으로 0.1에서 0.15 사이의 값이 사용된다[3, 4, 5].

FCH 방법에서처럼 본 연구에서도 문자열 형태의 키는 임의 사상(random mapping)을 통해 정수 형태로 변환되는데, 이를 위해 다음 6개의 사상 테이블을 사용한다.

- . RM0, RM1, RM2 : 문자 위치별 사상 테이블
- . CM0, CM1, CM2 : 문자 코드별 사상 테이블

이들은 난수 발생을 통해 생성되는 0 이상의 정수 값을 지니고 있는 일종의 의사 난수 테이블(pseudo random number table)이다. RM0, RM1, RM2는 키를 구성하는 문자의 위치에 따라 사상을 수행하는 데 사용되며 각각의 크기는 키의 최대 길이인 L이다. CM0, CM1, CM2는 키를 구성하는 문자의 코드 값에 따라 사상을 수행하는 데 사용되며 각각의 크기는 문자 코드의 수인 C이다. MPHF 생성에서 임의 사상을 도입하면 고른 분포를 지닌 정수 형태의 키가 주어지므로 문자열 형태의 키를 다루는 것보다는 생성이 용이하고, MPHF 생성기의 자료 구조가 단순해지는 장점이 있다.

탐색키의 사상은 다음 4가지 사상 함수에 의해 이루어진다.

. f0 : U → I

$$f0(K) = \sum_{i=0}^{|K|-1} (RM0[i] \text{ xor } CM0[K[i]])$$

. h0 : U → [0, M)

$$h0(K) = \begin{cases} f0(K) \bmod \lfloor \beta M \rfloor & \text{if } (f0(K) \bmod N) < \lfloor \alpha N \rfloor \\ (f0(K) \bmod (M - \lfloor \beta M \rfloor)) + \lfloor \beta M \rfloor & \text{otherwise} \end{cases}$$

. h1 : U → [0, N)

$$h1(K) = \sum_{i=0}^{|K|-1} (RM1[i] \text{ xor } CM1[K[i]]) \bmod N$$

. h2 : U → [0, N)

$$h2(K) = \sum_{i=0}^{|K|-1} (RM2[i] \text{ xor } CM2[K[i]]) \bmod N$$

여기서 xor는 비트별 배타적 논리합(bit-wise exclusive OR) 연산을, mod는 나머지 연산을 각각 나타낸다. 이상의 정의에서 알 수 있듯이 키 K에 대해 f0, h1, h2를 계산하기 위해서는 각각 |K|번의 xor 연산이 요구되며, h0는 f0의 값이 주어지면 이를 이용하여 키의 참조 없이 직접 계산 가능하다. 그리고 α와 β는 키 집합의 분할 비율(partition ratio)을 나타내는데, 이는 일종의 모형 매개변수로서 다음에 설명될 키 레벨의 구성과 관계가 있다.

탐색 단계에서 동시에 해쉬 값이 결정되는 키의 부분 집합인 키 레벨 LEVEL[t]는 다음과 같이 h0 값이 t인 키의 부분 집합으로 정의된다.

$$LEVEL[t] = \{ K \mid h0(K) = t \}$$

키 집합 분할 비율 α와 β는 평균적인 키 레벨의 크기가 서로 다른 두 집단으로 탐색키를 분할하는 역할을 지닌다. 이러한 키 집단 분할 방법은 FCH 방법에서 처음 도입되었으며, 경험적으로 α는 0.6, β는 0.3이 좋은 것으로 알려져 있다[4]. 그래서 본 연구에서도 이 분할 비율을 그대로 사용하는데, 이 경우 약 60%의 키 집단은 30% 정도의 상대적으로 큰 레벨로 대응되고, 나머지 약 40%의 키 집단은 70% 정도의 상대적으로 작은 레벨로 대응된다. 그 결과 분할 기법을 적용하지 않았을 경우에 비해 큰 레벨과 작은 레벨의 수는 증가하고 크기가 중간인 레벨의 수는 감소하게 된다. 따라서 빈 슬롯이 많아서 MPHF 탐색의 성공 확률이 높은 탐색 단계의 초기에 보다 많은 수의 큰 레벨을 먼저 처리하고, 성공 확률이 낮은 탐색 단계의 말기에는 주로 크기가 1 혹은 2인 많은 수의 작은 레벨을 처리할 수 있게 되어 전체적인 탐색의 성공율을 높일 수 있다.

본 연구에서는 사상 함수 외에도 다음 3가지 보조적인 함수가 MPHF 모형의 정의를 위해 사용된다.

. SM : [0, N) → {0, 1}
. G : [0, M) → [0, N)
. GM : [0, M) → {0, 1}

SM은 선택 마크(selection mark) 함수로서 N개의 비트로 구성되는 테이블 형태를 취하며, 함수 값은 선택 단계에서 결정된다. 키 K의 MPHF 값이 단지 f0 값만을 이용하여 계산 가능한 경우 SM[f0(K) mod N]은 1로 주어지고, 그렇지 않으면 0으로 주어진다.

G는 레벨 오프셋(level offset) 함수로서 해당 레벨에 속한 키의 해쉬 테이블 상의 오프셋을 저장하는데, M개의 정수로 구성되는 테이블 형태를 취한다. G[t]은 LEVEL[t]의 오프셋 값을 나타내며, G의 값은 탐색 단계에서 결정된다.

GM은 레벨 마크(level mark) 함수로서 M개의 비트로 구성되는 테이블 형태를 취하며, 함수 값은 탐색 단계에서 결정된다. LEVEL[t]의 MPHF 계산시에 h1 값을 이용하는 경우 GM[t]는 0으로 주어지고, 그렇지 않으면 1로 주어진다. GM을 도입하여 사용하면 각 키 레벨에 대해 h1과 h2 두 가지 사상 함수 값 집합 중에서 충돌이 없는 하나를 선택적으로 사용할 수 있으므로 자유도가 증대된다. 따라서 그만큼 사상 및 탐색 단계의 성공 가능성이 높아지는데[4], 대신 MPHF를 저장하는 데 M비트의 부가적인 기억 공간이 더 필요하게 된다.

본 연구에서 정립한 MPHF의 모형 h는 다음과 같다.

. h : U → [0, N)

$$h(K) = \begin{cases} f0(K) \bmod N & \text{if } SM[(f0(K) \bmod N)] = 1 \\ (h1(K) + G[h0(K)]) \bmod N & \text{else if } GM[h0(K)] = 0 \\ (h2(K) + G[h0(K)]) \bmod N & \text{otherwise} \end{cases}$$

키 K가 주어졌을 때, SM[f0(K) mod N]이 1인 경우

h는 f0 값만 이용하여 계산하며, 그렇지 않은 경우에는 GM[h0(K)]이 1이면 h는 h0, h1 및 G 값을 이용하여 계산하고, 아니면 h는 h0, h2 및 G 값을 이용하여 계산한다.

f0가 고른 분포를 지닌 임의 사상 함수일 경우, 전체 키 집합 중에서 'f0(K) mod N'의 값이 유일한 키의 비율은 다음 수식으로 계산할 수 있다.

$$N \times \frac{1}{N} \times \left(\frac{N-1}{N}\right)^{N-1} = \frac{1}{\left(1-\frac{1}{N}\right)^{-N}\left(1-\frac{1}{N}\right)}$$

여기서 우측 분모의 극한값이 $e(\approx 2.71828)$이므로 이 수식의 극한값은 $1/e(\approx 0.36788)$이다. 이는 함수 SM을 도입하여 사용하면 전체 키 집합 중에서 36.8% 정도의 키는 f0 값만으로 h의 계산이 가능함을 의미한다. 따라서 MPHF의 생성 과정에서 SM의 값을 결정하는 선택 단계를 도입하여 이를 먼저 수행하면, 약 36.8%의 키는 이 단계에서 미리 걸러지기 때문에 전체적인 MPHF의 생성 속도를 증가시킬 수 있다.(4장 참고)

3. MPHF 생성 알고리즘

전술한 바와 같이 본 연구의 MPHF 생성 알고리즘은 선택, 정렬, 사상, 탐색의 4 단계로 구분되어 차례대로 수행되는 SOMS 접근 방식을 취한다.

3.1 선택 단계

선택 단계에서는 f0에 의해 유일하게 사상되는 키의 부분 집합을 선택하여 미리 걸러낸다. 이렇게 선택된 키에 대해서는 f0에 의해 대응되는 SM의 값을 1로 설정하고 나머지는 0으로 설정한다. 선택 단계의 알고리즘은 다음과 같다.

Algorithm : Selecting_Step

(1) build RM0 and CM0 randomly
(2) initialize SM and CHK
(3) for each K in U do
    if 'f0(K) mod N' is unique then
        SM[f0(K) mod N] = CHK[f0(K) mod N] = 1

여기서 CHK는 N개의 비트로 구성된 테이블로서 이후 단계에서 키의 충돌 여부를 검사하는 데 사용된다. 과정 (1)에서 사상 테이블 RM0와 CM0는 난수 발생을 통하여 구성한다. 과정 (3)에서 유일성에 대한 검사는 크기가 N인 배열을 사용하여 전체 키 집합에 대해 O(N) 시간에 간단히 처리할 수 있으므로 자세한 것은 생략하였다. 위의 알고리즘으로부터 선택 단계는 O(N)의 기억 장소를 사용하여 O(N) 시간에 수행 가능함을 알 수 있다.

다음 그림 1은 N이 10인 경우에 선택 단계의 처리 예를 보여준다. (a)에서는 예제 키의 f0 값이 주어져 있고, (b)는 선택 단계의 수행 결과를 보여준다. 이 예에서는 'a', 'in', 'or'의 'f0(K) mod N' 값이 각각 5, 6, 1로 유일하므로 선택 단계에서 미리 걸러진다.

키 K	a	an	this	that	in	out	of	and	or	not
f0(K)	65	17	32	22	86	37	9	7	41	19

(a) 키의 f0 값

i	0	1	2	3	4	5	6	7	8	9
SM[i]	0	1	0	0	0	1	1	0	0	0
CHK[i]	0	1	0	0	0	1	1	0	0	0
관련 키		or	this that			a	in	an out and		of not

(b) 선택 단계의 수행 결과

그림 1 선택 단계의 처리 예

3.2 정렬 단계

정렬 단계에서는 선택 단계에서 걸러지지 못한 나머지 키 집합에 대해 h0 값이 동일한 키들로 키 레벨을 구성하고, 키 레벨을 크기에 따라 정렬한다. 정렬 단계의 알고리즘은 다음과 같으며, 이는 문헌 [4], [5]에서 적용된 방법과 거의 같다.

Algorithm : Ordering_Step

(1) initialize LEVEL
(2) for each K in U do // build LEVEL
    if SM[f0(K) mod N] = 0 then
        append(K, LEVEL[h0(K)])
(3) find HEAP size S and initialize HEAP
(4) for each LEVEL[t] in LEVEL do
    insert(LEVEL[t], HEAP[|LEVEL[t]| - 1])

여기서 append는 키를 해당 레벨에 추가하는 서브루틴으로 이는 O(1) 시간에 처리 가능하다. HEAP은 키 레벨의 정렬을 위해 사용되는 테이블로, 키 레벨 크기의 최대값이 S일 때 HEAP의 크기는 바로 S가 된다. 최악의 경우 HEAP의 크기는 키 집합의 크기 N이 될 수도 있지만, 거의 대부분은 N보다 훨씬 작다. 왜냐하면, 본 연구에서는 FCH 방법에 따라 키 집합 분할 비율 α와 β를 각각 0.6과 0.3으로 설정하고 있고, 약 36.8%의 키는 선택 단계에서 미리 걸러진다. 따라서 HEAP의 크기는 나머지 약 63.2%의 키 중에서 60%의 키 집단이 30%

의 키 레벨로 대응되는 경우에 키 레벨의 크기 분포에 따라 좌우되는데, 이 분포는 포아송 분포를 따르며 그 평균 p는 다음 수식과 같이 함수 G의 비율 RG에 따라 달라진다. RG가 0.1인 경우 평균 p는 12.64 정도가 된다.

$$p \approx \frac{0.6 \times 0.632 \times N}{0.3 \times M} = \frac{1.264}{M/N} = \frac{1.264}{RG}$$

이러한 포아송 분포에서 실제로 매우 큰 레벨이 나타나는 경우는 거의 없고, 대부분 수십 정도의 크기이므로, 작은 크기의 HEAP으로도 키 레벨의 정렬이 가능하다. 키 레벨 L의 정렬은 그 크기에 해당하는 HEAP의 위치, 즉 HEAP[|L|-1]에 삽입함으로써 수행되며, 위의 정렬 알고리즘에서 insert는 이러한 삽입을 수행하는 서브루틴으로 이는 O(1) 시간에 처리 가능하다. 따라서 정렬 단계는 O(N)의 기억 장소를 사용하여 O(N) 시간에 수행 가능하다.

다음 그림 2는 RG가 0.5로 주어져 함수 G의 크기 M이 5인 경우에 정렬 단계의 처리 예를 보여준다. (a)에서는 선택되지 않은 예제 키의 h0 값이 주어져 있고, (b)와 (c)는 정렬 단계가 수행되고 난 후의 키 레벨과 HEAP을 각각 보여준다.

키 K	an	this	that	out	of	and	not
h0(K)	2	0	0	2	2	4	4

(a) 키의 h0 값

t	0	1	2	3	4
LEVEL[t]	this that		an out of		and not

(b) 키 레벨

i	0	1	2
HEAP[i]		LEVEL[0] LEVEL[4]	LEVEL[2]

(c) HEAP

그림 2 정렬 단계의 처리 예

3.3 사상 단계

사상 단계에서는 사상 테이블 RM1, RM2, CM1, CM2를 난수 발생을 통하여 구성한 다음, 큰 레벨부터 시작하여 각 레벨에 속한 키의 h1과 h2 값을 구하고, 해당 레벨의 h1과 h2 값 집합 각각에 대해 충돌 여부를 검사한다. 사상 단계의 알고리즘은 다음과 같다.

Algorithm : Mapping_Step

```
(1) build RM1, RM2, CM1 and CM2 randomly
(2) for i = S - 1 to 0 do            // S : HEAP size
      for each LEVEL[t] in HEAP[i] do
        if every h1(K) for K in LEVEL[t] is not distinct and
           every h2(K) for K in LEVEL[t] is not distinct
              then
        goto (1)
```

어떤 키 레벨에 대해 h1 값 사이에서도 충돌이 발생하고 h2 값 사이에서도 충돌이 발생하면 MPHF 생성이 불가능하다. 따라서 이러한 경우에는 RM1, RM2, CM1, CM2를 다시 구성하여 충돌이 발생하지 않을 때까지 사상과 충돌 검사를 계속 반복하여 수행한다. 그런데 충돌이 발생할 확률은 키 집합의 크기가 증가하면 매우 빠르게 0으로 접근하는 성질이 있다[4, 5]. 그러므로 대규모 키 집합의 경우 대부분 1번의 수행으로 사상 단계가 종료된다. 그리고 키 집합 전체에 대한 한번의 충돌 검사는 O(N) 시간에 가능하다. 따라서 사상 단계는 O(N)의 기억 장소를 사용하여 O(N) 기대 시간에 수행 가능하다.

본 연구에서는 정렬 단계를 사상 단계보다 먼저 수행하는 SOMS 접근 방식을 취한다. 정렬 단계를 먼저 수행하면 이미 키 레벨이 구성되어 있으므로 보다 효율적으로 충돌 검사를 수행할 수 있다. 뿐만 아니라 키 레벨이 크기에 따라 정렬되어 있으므로 충돌 가능성이 높은 큰 레벨부터 처리할 수 있기 때문에, 충돌이 발생한 경우 보다 조기에 이를 감지하여 반복 수행에 들어갈 수 있다. 따라서 SOMS 방식은 기존의 MOS 방식보다 사상 단계의 수행 속도가 상대적으로 빠르다는 장점이 있다.

다음 그림 3은 사상 단계의 처리 예를 보여준다. 여기서 키 옆에 주어진 것은 해당 키의 h1과 h2 값의 쌍이다. 이 예에서 LEVEL[0]는 h2 값 사이에서만 충돌이 일어났고, LEVEL[2]는 h1 값 사이에서만 충돌이 일어났으며, LEVEL[4]는 전혀 충돌이 없다. 이와 같은 경우라면 일단 MPHF 생성이 가능한 것으로 판단되므로 다음 단계인 탐색 단계로 진행할 수 있다.

t	0	1	2	3	4
LEVEL[t]	this (7,2) that (1,2)		an (1,1) out (3,2) of (3,8)		and (7,3) not (0,2)

그림 3 사상 단계의 처리 예

3.4 탐색 단계

탐색 단계에서는 HEAP을 이용하여 큰 레벨부터 시작하여 각 레벨의 오프셋을 탐색하여 함수 G와 GM을 결정한다. 탐색 단계의 알고리즘은 다음과 같다.

Algorithm : Searching_Step

```
(1) initialize G and GM
(2) for i = S - 1 to 0 do        // S : HEAP size
      for each LEVEL[t] in HEAP[i] do
        collision = true          // collision check flag
        if every h1(K) for K in LEVEL[t] is distinct then
          for j = 0 to N - 1 do
              if every CHK[(h1(K)+j) mod N] for K in
               LEVEL[t] is 0 then
                collision = false, G[t] = j
                for each K in LEVEL[t] do
                  CHK[(h1(K)+j) mod N] = 1
                break loop
        if collision = true and
         every h2(K) for K in LEVEL[t] is distinct then
         for j = 0 to N - 1 do
          if every CHK[(h2(K)+j) mod N] for K
                in LEVEL[t] is 0 then
            collision = false, G[t] = j, GM[t] = 1
            for each K in LEVEL[t] do
                CHK[(h2(K)+j) mod N] = 1
            break loop
        if collision = true then
          FAIL          // fail in MPHF generation
```

h1 값들에 대해 이미 처리된 키와 충돌이 없는 오프셋이 존재하면 GM 값은 0으로 초기화되어 있으므로 그대로 두며, 그렇지 않고 h2에 대해 충돌이 없는 오프셋이 존재하면 GM 값을 1로 설정하고, h1과 h2 모두에 대해 충돌이 없는 오프셋이 존재하지 않으면 탐색이 실패하게 된다. 본 연구에서는 탐색이 실패한 경우 FCH 방법과 같이 백트래킹(backtracking)을 하지 않고, 전체적인 MPHF 생성 실패로 간주하여 MPHF 생성을 처음부터 다시 시작하는 단순한 탐색 방법을 채택하고 있다. 그 이유는 경험적으로 볼 때 대규모 키 집합의 경우 함수 G의 비율 RG가 0.1 이하 정도로 지나치게 작지만 않다면 거의 대부분 탐색이 성공하기 때문이다.(4장 참고)

이와 같은 탐색 방법은 단순히 알고리즘을 문자 그대로 분석하면 $O(N^2)$의 처리 시간을 요하는 것으로 보이는데, 증명이 어려워 아직 완전히 증명되지는 못하였지만 실험 결과등에 근거하여 이러한 탐색 알고리즘이 확률적으로는 O(N) 기대 시간에 처리 가능한 것으로 알려져 있다[4, 6]. 그러므로 본 연구의 탐색 단계 역시 O(N)의 기억 장소를 사용하여 O(N) 기대 시간에 수행 가능한 것으로 볼 수 있다.

다음 그림 4는 탐색 단계의 처리 예를 보여준다. LEVEL[2]는 h2 값들 사이에만 충돌이 없으므로 이에 대해 오프셋을 탐색하면 1로 주어지므로 G[2]와 GM[2]는 모두 1로 설정된다. LEVEL[0]는 h1 값 사이에만 충돌이 없으므로 이에 대해 오프셋을 탐색하면 3으로 주어지므로 G[0]과 GM[0]은 각각 3과 0으로 설정된다. LEVEL[4]의 경우 h1과 h2 모두 그들 값 사이에는 충돌이 없는데, h1에 대한 탐색에서는 이미 처리된 키와 충돌을 유발하지 않는 오프셋이 존재하지 않는다. 따라서 h2에 대해서도 오프셋을 탐색하게 되고 그 결과 5가 존재하므로 G[4]와 GM[4]는 각각 5와 1로 설정된다.

i	0	1	2	3	4
GM[i]	0	0	1	0	0
G[i]	0	0	1	0	0

j	0	1	2	3	4	5	6	7	8	9
CHK[j]	0	1	1	1	0	1	1	0	0	1

(a) LEVEL[2] 탐색 후의 결과

i	0	1	2	3	4
GM[i]	0	0	1	0	0
G[i]	3	0	1	0	0

j	0	1	2	3	4	5	6	7	8	9
CHK[j]	1	1	1	1	1	1	1	0	0	1

(b) LEVEL[0] 탐색 후의 결과

i	0	1	2	3	4
GM[i]	0	0	1	0	1
G[i]	3	0	1	0	5

j	0	1	2	3	4	5	6	7	8	9
CHK[j]	1	1	1	1	1	1	1	1	1	1

(c) LEVEL[4] 탐색 후의 결과

그림 4 탐색 단계의 처리 예

이상을 종합해 보면, 전체적으로 본 연구의 MPHF 생성 알고리즘은 O(N)의 기억 장소와 O(N)의 기대 처리 시간을 요하는, 즉 확률적으로 키 집합의 크기에 대해 선형적인 시간 및 공간을 사용하여 처리 가능한 알고리즘이다. 그리고 이는 실제 실험 결과와도 일치한다.(4장 참고)

4. 실험 및 고찰

MPHF 생성 실험에 사용된 컴퓨터의 사양은 주기억 장치가 64MB이고, CPU는 Intel사의 MMX 200MHz이다. 탐색키 집합으로는 한국어와 영어 사전 및 기타 문서에서 수집된 100만개와 이를 임의 변형을 통해 만들어낸 200만개를 사용하였으며, 키의 평균 길이는 11.5바이트이고 최대 길이 L은 50으로 제한하였다. 사상 테이블 구성에서는 Park와 Miller의 난수 발생 방법[12]을 사용하였다. 그리고 이하 실험 결과에서 주어진 시간은 5번 반복 수행하면서 MPHF 생성이 성공한 경우에 C 언어의 *clock* 함수를 사용하여 측정한 평균값이다.

다음 표 1은 함수 G의 비율 RG가 0.12 및 0.15인 경우에 키 집합의 크기 N에 따른 MPHF 생성 시간을 보여준다. 이로부터 탐색 단계의 처리 시간이 MPHF 생성 시간 중 가장 많이 차지함을 알 수 있다.

표 1 탐색키 집합의 크기 N과 MPHF 생성 시간

RG	N (만개)	50	100	150	200	250	300
0.12	탐색 단계 시간 (초)	37.4	74.5	112.5	150.0	187.3	225.4
	전체 생성 시간 (초)	41.5	82.9	123.6	167.1	208.2	250.5
0.15	탐색 단계 시간 (초)	10.1	19.9	30.5	40.2	50.7	61.0
	전체 생성 시간 (초)	14.2	27.8	43.0	57.2	71.7	86.1

다음 그림 5는 표 1의 결과 중에서 전체 생성 시간 부분을 그래프 형태로 보여준다. 이 그래프를 통해 본 연구의 MPHF 생성 알고리즘은 그 처리 시간이 키 집합의 크기에 대해 확률적으로 선형적임을 알 수 있다.

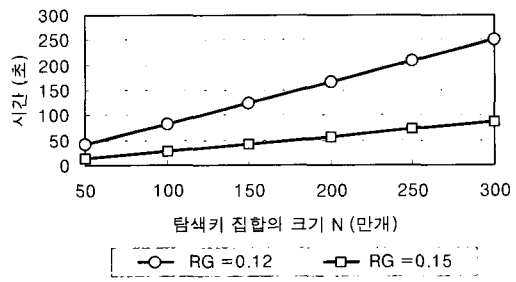

그림 5 탐색키 집합의 크기 N과 MPHF 생성 시간

다음 표 2는 키 집합의 크기 N이 100만일 때, RG에 따른 MPHF 생성 시간을 보여준다. 이 결과로부터 RG를 작게하면 MPHF의 저장 공간은 줄어들지만 생성 시간이 크게 늘어나고 생성이 실패할 가능성이 커짐을 알 수 있다. 따라서 응용 분야의 생성 속도 및 기억 장소 요구에 따라 RG의 값을 적당히 조절할 필요가 있다. 그리고 선택 단계를 도입하면 MPHF 생성 속도가 향상되고 RG가 작을수록 생성 속도 향상 정도가 커지며, 성공 가능성이 높아지는 것으로 나타났다. 이로부터 약 36.8%의 키를 미리 걸러내는 선택 단계의 도입 효과가 크다는 것을 알 수 있다.

표 2 함수 G의 비율 RG와 MPHF 생성 시간

(*N* = 100만, 괄호 안은 선택 단계를 적용하지 않은 경우)

RG	0.10	0.11	0.12	0.13	0.14	0.15
탐색 단계 시간 (초)	387.0 (실패)	147.2 (254.7)	74.5 (115.5)	41.9 (60.3)	23.5 (32.0)	19.9 (25.7)
전체 생성 시간 (초)	395.3 (실패)	155.4 (260.1)	82.9 (120.8)	50.3 (65.8)	31.8 (37.4)	27.8 (31.0)

전체 MPHF 생성 실험에서 선택 단계를 적용하였을 때는 RG가 0.10이고 N이 50만과 100만인 경우에 각각 1번씩의 탐색 실패로 인해 MPHF 생성이 실패하였으며, 사상 단계의 처리에서는 재시도 없이 항상 처음에 성공하였다. 이로부터 RG가 0.1이하 정도로 매우 작지만 않다면, 대부분의 경우 한 번만에 MPHF 생성이 가능하므로 굳이 백트래킹과 같은 복잡한 제어 구조를 도입하지 않아도 됨을 알 수 있다. 그리고 선택 단계에서 미리 걸러진 키의 비율은 평균 36.79%로 나타났는데, 이는 이론적인 계산치 $1/e(\approx 0.36788)$과 일치하는 값이다. 또한 HEAP의 크기는 대부분 30 내외이고, RG가 0.10이고 N이 100만인 경우에 35로 가장 크게 관측되었다. 이로부터 작은 크기의 HEAP으로도 키 레벨의 정렬이 가능함을 알 수 있다.

다음 표 3은 MOS 접근 방식을 취하고 있는 FCH 방법을 적용하였을 때, 함수 G의 비율 RG가 0.12 및 0.15인 경우에 탐색키 집합의 크기 N에 따른 MPHF 생성시간을 본 연구의 방법과 비교하여 보여준다. 본 연구의 방법을 적용하였을 때 소요된 전체 생성 시간은 FCH 방법에 비해 RG가 0.12인 경우에는 평균 67.7%, 0.15인 경우에는 평균 89.2%인 것으로 나타났다. 이로부터 본 연구의 방법은 선택 단계를 도입하여 생성 시간 중에서 가장 많은 부분을 차지하는 탐색 단계의 처리 시간을 줄이기 때문에 전체적인 MPHF 생성 속도가 FCH 방법보다 빠르며, RG가 작을수록 선택 단계의 도입 효과가 크다는 것을 알 수 있다.

표 3 FCH 방법과의 비교

(괄호 안은 본 연구의 방법을 적용한 경우)

RG	N (만개)	50	100	150	200	250	300
0.12	탐색 단계 시간 (초)	57.9 (37.4)	115.6 (74.5)	173.4 (112.5)	232.5 (150.0)	290.3 (187.3)	349.4 (225.4)
	전체 생성 시간 (초)	61.3 (41.5)	122.4 (82.9)	183.0 (123.6)	246.9 (167.1)	307.6 (208.2)	370.0 (250.5)
0.15	탐색 단계 시간 (초)	12.5 (10.1)	25.8 (19.9)	38.5 (30.5)	49.6 (40.2)	62.7 (50.7)	75.8 (61.0)
	전체 생성 시간 (초)	15.9 (14.2)	32.4 (27.8)	48.2 (43.0)	64.1 (57.2)	80.4 (71.7)	96.5 (86.1)

검색시에 각 키의 MPHF 계산 속도를 살펴보면, FCH 방법의 경우 임의 사상을 위해 Pearson의 기법[11]을 사용하므로[4], 키의 길이가 s일 때 키 집합의 크기가 3바이트로 표현되면 6s번, 4바이트로 표현되면 8s번의 xor 연산이 수행되어야만 MPHF를 계산할 수 있다. 이에 반해 본 연구의 MPHF 모형 h의 경우 선택 단계에서 걸러진 약 36.8%의 키는 f0의 계산만으로 MPHF의 값을 구할 수 있으므로 1s번의 xor 연산이 요구되고, 나머지 63.2%의 키는 2s번의 xor 연산이 요구되기 때문에(2장 참고), 키당 평균 1.632s번의 xor 연산이 요구된다.

사상 테이블의 크기를 살펴보면, 키의 인덱스(index)를 4바이트로 표현하고 키의 최대 길이가 L, 문자 코드의 수가 C일 때, 본 연구의 경우 문자 위치별 사상 테이블과 문자 코드별 사상 테이블이 각각 3개씩 필요하므로, 4*3*(L+C)바이트 크기의 기억 공간이 요구된다. 예를 들어 L이 50이고 C가 256일 경우 3,672바이트가 요구된다. 이에 반해 Pearson의 기법[11]을 사용하는 FCH 방법의 경우, 인덱스의 크기나 키의 최대 길이에 상관 없이 문자 코드의 수만큼 원소의 크기가 1바이트인 3개의 사상 테이블이 필요하므로, 3*C바이트 크기의 기억 공간만 요구된다[4]. 예를 들어 C가 256일 경우 768바이트가 요구된다.

대규모 키 집합의 경우 MPHF의 표현을 위해 요구되는 기억 공간의 크기는 주로 함수 G의 크기에 의해 좌우된다. 왜냐하면, 사상 테이블과 함수 GM이나 SM이 차지하는 기억 공간은 함수 G에 비해 훨씬 작기 때문이다. 키의 인덱스 크기가 4바이트이고 키 집합의 크기 N에 대한 함수 G의 비율이 RG일 때, 함수 G는 4*N*RG 바이트, 함수 GM은 N*RG비트의 기억 공간을 각각 차지한다. 예를 들어 N이 100만이고, RG가 0.12일 경우, G와 GM을 위해 48만바이트와 12만비트가 각각 필요하다.

선택 마크 함수 SM의 경우 이론적으로는 키당 1비트의 기억 공간이 필요하지만, 키의 길이가 가변적이어서 별도의 인덱스를 사용하고, 인덱스 부분에 1비트의 여유 공간이 있어 이를 할애하여 SM을 저장할 수 있으면, 사실상 부가의 기억 공간을 사용하지 않고도 SM을 저장할 수 있다. 예를 들어 32비트 크기의 컴퓨터 워드(word)로 인덱스를 표현하고, 인덱스 공간이 2^{31}바이트(2기가바이트)를 넘지 않는다면, 31비트 이하로 인덱스를 표현할 수 있다. 따라서 이와 같은 경우에는 1비트 이상의 여유가 있으므로 여기에 각 키의 SM 정보를 저장할 수 있다.

전체적인 MPHF의 기억 공간 효율을 살펴보면, 본 연구와 FCH 방법 모두 RG를 0.1 정도까지 낮출 수 있는 것으로 나타나고 있다[4]. 따라서 사상 테이블에서 본 연구의 방법이 약간의 기억 공간을 더 요구하는 것을 제외하면, 두 방법은 거의 비슷한 기억 공간 효율을 지니고 있음을 알 수 있다.

5. 결 론

FCH 방법등 기존의 MPHF 생성 방법들은 주로 사상, 정렬, 탐색의 3단계로 진행되는 MOS 접근 방식을 취하고 있다. 본 연구에서는 이를 개선하여 계산 속도가 빠른 MPHF 모형을 정립하고, 선택 단계를 새로 도입하여 선택, 정렬, 사상, 탐색의 4단계로 진행되는 SOMS 접근 방식의 MPHF 생성 알고리즘을 개발하였다. SOMS 접근 방식에서는 약 36.8%의 키 집합을 선택 단계에서 미리 걸러내고, 사상 단계보다 정렬 단계를 먼저 처리함으로써 보다 효과적으로 MPHF를 생성할 수 있기 때문에 전체적인 MPHF 생성 시간을 단축시킬 수 있다.

실험 및 알고리즘 분석 결과 본 연구의 MPHF 생성 방법은 기억 공간과 기대 처리 시간이 탐색키 집합의 크기에 대해 선형적이며, 수백만 개의 키 집합에 대해서도 대부분 MPHF 생성이 성공할 뿐만 아니라, MPHF가 차지하는 기억 공간은 기존의 방법 중 가장 우수한 FCH 방법과 비슷하면서도 MPHF 생성 속도는 보다 빠른 것으로 나타났다. 따라서 본 연구의 방법으로 MPHF를 생성하여 이용하면 자연어 처리, 정보 검색, CD-ROM 타이틀 제작 등 대규모 정적 탐색키 집합의 검색을 요하는 많은 응용 분야에서 검색 성능을 향상시킬 수 있을 것으로 기대된다.

참 고 문 헌

[1] R. Cichelli, "Minimal Perfect Hash Functions Made Simple," *CACM*, Vol. 23, pp. 17-19, 1980.

[2] N. Cercone, M. Krause, and J. Boates, "Minimal and Almost Minimal Perfect Hash Function Search with Application to Natural Language Lexicon Design," *Computers and Mathematics with Applications*, Vol. 9, pp. 215-231, 1983.

[3] E. Fox, Q. Chen, A. Daoud, and L. Heath, "Order Preserving Minimal Perfect Hash Functions and Information Retrieval," *ACM Transactions on Information Systems*, Vol. 9, pp. 281-308, 1991.

[4] E. Fox, Q. Chen, and L. Heath, "A Faster Algorithms for Constructing Minimal Perfect Hash Functions," In *15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval(SIGIR'92)*, pp. 266-273, 1992.

[5] E. Fox, L. Heath, Q. Chen, and A. Daoud, "Practical Minimal Perfect Hash Functions for Large Databases," *CACM*, Vol. 35, pp. 105-121, 1992.

[6] G. Havas and B. Majewski, "Optimal Algorithms for Minimal Perfect Hashing," *Technical Report 234*, Department of Computer Science, The University of Queensland, 1992.

[7] G. Havas and B. Majewski, "Graph Theoretical Obstacles to Perfect Hashing," *Congressus Numerantium*, Vol. 98, pp. 81-93, 1993.

[8] G. Havas, B. Majewski, N. Wormald, and Z. Czech, "Graphs, Hypergraphs and Hashing," In *19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*, Vol. 790 of *Lecture Notes in Computer Science*, pp. 153-165, 1994, Springer Verlag.

[9] T. Sager, "A Polynomial Time Generator for Minimal Perfect Hashing Functions," *CACM*, Vol. 28, pp. 523-532, 1985.

[10] S. Wartic, E. Fox, L. Heath, and Q. Chen, "Hashing Algorithms," *Information Retrieval: Data Structures & Algorithms* (ed. W. Frakes and R. Baeza-Yates), pp. 293-318, Prentice Hall, New Jersey, 1992.

[11] P. Pearson, "Fast Hashing of Variable-Length Text Strings," *CACM*, Vol. 6, pp. 677-680, 1990.

[12] Park and Miller, "Random Number Generators: Good Ones Are Hard to Find," *CACM*, Vol. 31, No. 10, pp. 1192-1201, 1988.

이 하 규

1987년 서울대학교 컴퓨터공학과(학사). 1989년 서울대학교 대학원 컴퓨터공학과(석사). 1994년 서울대학교 대학원 컴퓨터공학과(박사). 1995년 ~ 1998년 한림대학교 컴퓨터공학부 교수. 1999년 ~ 현재 성공회대학교 컴퓨터정보학부 교수. 관심분야는 한국어정보처리, 자연어처리, 정보검색