Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.28, NO.1, Feb. 2018 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2018.28.1.25

NIST SP 800-90B의 최소 엔트로피 추정 알고리즘에 대한 고속 구현 및 효율적인 메모리 사용 기법*

김 원 태,[1†] 염 용 진,[1,2] 강 주 성[1,2‡]
[1]국민대학교 금융정보보안학과, [2]국민대학교 정보보안암호수학과

High-Speed Implementation and Efficient Memory Usage of Min-Entropy Estimation Algorithms in NIST SP 800-90B*

Wontae Kim,[1†] Yongjin Yeom,[1,2] Ju-Sung Kang[1,2‡]
[1]Dept. of Financial Information Security, Kookmin Univ.
[2]Dept. of Information Security, Cryptology, and Mathematics, Kookmin Univ.

요 약

최근 NIST에서는 암호학적 난수발생기의 핵심 요소인 엔트로피 소스의 안전성을 평가하기 위한 방법을 다루고 있는 SP 800-90B 문서의 두 번째 수정안과 이를 Python으로 구현한 코드를 제공하였다. SP 800-90B에서의 엔트로피 소스에 대한 안전성 평가는 엔트로피 소스의 출력 표본 수열로부터 도출한 여러 가지 추정량(estimator)에 기반 하여 최소 엔트로피를 추정하는 과정이다. 최소 엔트로피 추정 과정은 IID 트랙과 non-IID 트랙으로 대별되어 진행된다. IID 트랙의 경우 MCV 추정량만을 사용하여 속도 측면에서 무리가 없다. 반면 non-IID 트랙에서는 MCV를 포함한 총 10 가지의 추정량을 적용해 최소 엔트로피를 추정하게 된다. NIST의 코드에서 non-IID 트랙의 1 회 구동 시간은 약 20 분이 소요되고, 사용되는 메모리는 5.5 GB를 넘긴다. 이는 다양한 잡음원으로 반복적인 평가를 수행해야 하는 평가 기관 또는 여러 환경에서 실험을 수행해야 하는 개발자나 연구자 입장에서는 NIST에서 제공한 Python 코드를 이용하는 것이 불편할 수 있으며, 환경에 따라 실행이 불가할 수도 있다.

본 논문에서는 SP 800-90B의 최소 엔트로피 추정 방법에 대한 고속 구현과 효율적인 메모리 사용 기법을 제시한다. 주요 연구 결과로 MultiMCW 추정 방법에 C++ 코드의 장점을 적용한 고속화 방법, MultiMMC 추정 방법의 데이터 저장 방식을 재구성하여 메모리 사용량을 현저하게 감소시킴과 동시에 고속화한 방법, LZ78Y 추정 방법에 데이터 저장 방식의 재구성을 통한 고속화 기법 등을 제안한다. 우리의 개선된 방법이 종합적으로 적용된 C++ 코드는 NIST에서 제공한 기존의 Python 코드와 비교할 때, 속도는 14 배 빠르고 메모리 사용량은 1/13로 감소하는 효과를 보인다.

ABSTRACT

NIST(National Institute of Standards and Technology) has recently published SP 800-90B second draft which is the document for evaluating security of entropy source, a key element of a cryptographic random number generator(RNG), and provided a tool implemented on Python code. In SP 800-90B, the security evaluation of the entropy sources is a process of

Received(09. 28. 2017), Modified(11. 30. 2017), Accepted(11. 30. 2017)
* 본 논문은 2017년도 한국정보보호학회 하계학술대회에 발표한 우수논문을 개선 및 확장한 것임
* 이 논문은 2017년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2014-6-00908, 난수발생기 및 임베디드 기기 안전성 연구)
† 주저자, kwt123@kookmin.ac.kr
‡ 교신저자, jskang@kookmin.ac.kr(Corresponding author)

estimating min-entropy by several estimators. The process of estimating min-entropy is divided into IID track and non-IID track. In IID track, the entropy sources are estimated only from MCV estimator. In non-IID Track, the entropy sources are estimated from 10 estimators including MCV estimator. The running time of the NIST's tool in non-IID track is approximately 20 minutes and the memory usage is over 5.5 GB. For evaluation agencies that have to perform repeatedly evaluations on various samples, and developers or researchers who have to perform experiments in various environments, it may be inconvenient to estimate entropy using the tool and depending on the environment, it may be impossible to execute.

In this paper, we propose high-speed implementations and an efficient memory usage technique for min-entropy estimation algorithm of SP 800-90B. Our major achievements are the three improved speed and efficient memory usage reduction methods which are the method applying advantages of C++ code for improving speed of MultiMCW estimator, the method effectively reducing the memory and improving speed of MultiMMC by rebuilding the data storage structure, and the method improving the speed of LZ78Y by rebuilding the data structure. The tool applied our proposed methods is 14 times faster and saves 13 times more memory usage than NIST's tool.

Keywords: SP 800-90B, Min-Entropy estimation, High-Speed implementation, Memory reduction, Entropy source

I. 서 론

암호시스템과 보안 응용 프로그램의 안전성을 위해 사용되는 난수는 예측 불가능성(unpredictability)과 독립성(independence)을 만족하도록 암호학적 난수발생기에 의해 생성된다. 암호학적 난수발생기는 진난수생성기(TRNG)와 의사난수발생기(PRNG)가 안전하게 결합하여 구성된다. 이 중에서 진난수생성기 부분은 초기 엔트로피를 생성하므로 보통 엔트로피 소스(entropy source)로 분류된다.

1.1 NIST의 암호학적 난수발생기 구성 표준 : SP 800-90A, B, C

암호학적으로 안전한 난수란 암호학적 난수발생기에 의해서 생성되는 난수를 의미한다. 미국 NIST(National Institute of Standards and Technology)의 SP 800-90A, B, C에서는 이러한 암호학적 난수발생기를 구성하는 방법과 그 안전성에 대한 내용을 다루고 있다. SP 800-90C[1]는 암호학적 난수 발생기의 설계방법에 대한 문서로, 암호학적 난수를 생성하기 위해 결정론적 난수발생기(Deterministic Random Bit Generator, DRBG)와 엔트로피 소스(entropy source)를 결합하는 방법을 다루고 있다. 결정론적 난수발생기에 대해 다루는 문서인 SP 800-90A[2]에 따르면 결정론적 난수발생기에서 출력된 의사 난수(pseudo random bits)의 안전성은 엔트로피 소스의 안전성에 의존한다.

1.2 SP 800-90B 최소 엔트로피 추정 과정

2016년 NIST는 엔트로피 소스의 안전성 평가를 위한 문서인 SP 800-90B의 두 번째 수정안[3]을 공표하였다. 이 문서는 잡음원(noise source)과 엔트로피 소스의 난수성(randomness)을 나타내는 최

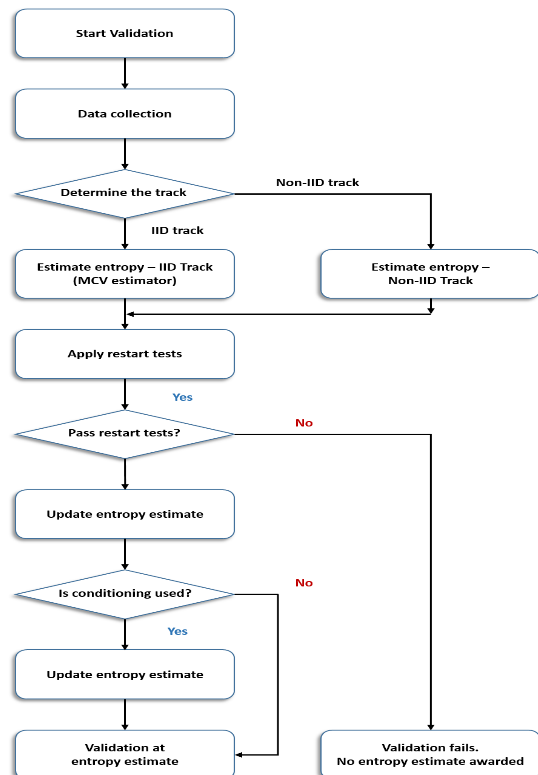

Fig. 1. Entropy estimation strategy[3]

소 엔트로피를 추정하기 위한 방법을 다루고 있으며, NIST에서는 그 방법을 Python으로 구현한 도구 [4]를 추가로 제공하고 있다.

SP 800-90B 최소 엔트로피 추정 과정은 Fig.1과 같다. SP 800-90B에서는 엔트로피 소스의 최소 엔트로피 추정을 위해 엔트로피 소스로부터 100만 개 이상의 출력 표본 수열을 입력하도록 권장하고 있다. 이때 표본은 고정된 길이의 비트열을 의미하며, 표본 값이란 표본의 바이너리(binary) 값을 의미한다. 입력된 표본 수열은 먼저 IID(independent and identically distributed) 검정과정을 거치게 되며, 그 결과에 따라 엔트로피 소스는 IID 또는 non-IID로 판정되어 각각의 트랙을 따라 최소 엔트로피 추정 과정을 진행한다.

1.3 SP 800-90B non-IID 트랙 고속 및 메모리 경량 구현의 필요성

IID 트랙은 MCV(Most Common Value) 추정량만을 이용하며, NIST에서 Python으로 구현한 기존 코드에서 그 추정 시간은 1 초가 안 되어 속도 측면에서 문제가 없다. 그러나 실제 사용되는 일반적인 잡음원(타임스탬프, CPU 클록 등)의 확률 분포는 non-IID를 따르는 경우가 대부분이다. non-IID 트랙은 표본 크기 8-bit를 구동하는 데 약 20 분의 시간을 소요한다. 이에 더해 엔트로피 소스는 하나 이상의 잡음원을 이용해 난수를 생성하는데, 이 경우 SP 800-90B에서는 각각의 잡음원에 대해 최소 엔트로피를 추정하도록 명시하고 있어 엔트로피 소스의 구성에 따라 더 많은 추정 시간이 요구될 것이다. 이뿐 아니라 기존 NIST 코드에서 non-IID 트랙을 구동할 때 RAM 8 GB 컴퓨팅 환경의 경우 메모리 문제로 구동에 실패하는 경우를 확인하였다. SP 800-90B 기반으로 다양한 표본에 대해 반복적인 평가를 수행해야 하는 평가 기관이나 개발자 또는 제한된 컴퓨팅 환경에서 최소 엔트로피 추정 실험을 수행해야 하는 연구자의 경우 기존 코드를 이용하는 것에 많은 불편함이 있을 것으로 사료된다.

1.4 논문의 구성

본 논문의 구성은 다음과 같다. Ⅱ장에서는 SP 800-90B의 non-IID 트랙에 대한 고속화와 메모리 경량 구현을 위한 배경지식을 제시한다. Ⅲ장에서는 SP 800-90B의 non-IID 트랙을 고속 및 메모리 경량 구현한 방법과 이를 적용한 C++코드를 제시한다. Ⅳ장에서는 Ⅲ장에서 제시한 C++코드의 성능 측정 결과를 제시한다. Ⅴ장에서는 논문의 결론 및 추후 연구 주제를 제시하도록 한다.

II. 고속 및 메모리 경량 구현을 위한 배경지식

2.1 SP 800-90B의 Python 구현에 대한 적절성 분석

Python은 가독성과 구현의 편의성에 장점이 있다. 다시 말하면 Python은 코드의 직관적 이해가 쉽도록 만들어졌으며, 상대적으로 다른 프로그램 언어들보다 풍부한 라이브러리를 제공하여 개발의 편의를 제공한다[5]. 이에 더해 SP 800-90B에 제시된 프리딕터를 이용한 최소 엔트로피 추정 방법의 기반이 된 문서인 [6]에서는 추후 연구로 프리딕터 추정법과 머신러닝의 연계성을 제시하였는데, 이는 풍부한 라이브러리를 제공하는 Python이 현재 머신러닝 알고리즘 개발의 핵심 언어로 사용되고 있기 때문이다. 이와 같은 이유로 NIST에서 SP 800-90B를 Python 코드로 개발했을 것으로 생각된다.

하지만 Python은 인터프리트(interpreted) 언어로써 컴파일 언어보다 느리다는 문제가 제기되고 있으며, 도구에 사용되는 Python 함수는 필요 이상의 기능을 제공하여 엔트로피 추정에 불필요한 연산이 소요된다. 불필요한 연산은 속도 지연 문제 뿐 아니라 과도한 메모리를 사용하여 기존 도구 사용 시 메모리 문제[7]의 원인이 되었다.

2.2 최적화를 위한 프로그램 언어 : C++

Python은 가독성이 좋고 구현이 편한 장점을 갖지만, 상대적으로 컴파일 언어보다 느리고 환경에 따라 구동에 문제를 가질 수 있어 최적화에 적절하지 않다고 판단된다. 일반적으로 Python은 대표적인 컴파일 언어인 C 시리즈(series)와 비교되고 있으며, 특히 최적화에는 많은 경우 C 시리즈가 더 적절하다는 결과들을 어렵지 않게 볼 수 있다[8,9]. 나아가 본 논문에서 제안한 MultiMCW의 경우 개선한 알고리즘이 Python보다 C 시리즈에 더 적절하였다. 따라서 제안하는 고속 구현 방법에는 Python보다 C 시리즈로 구현하는 것이 적절하다고 판단하였고, 구현의 편의성 등을 고려해 C++를 본 논문의 최적화

를 위한 프로그램 언어로 선택하였다.

2.3 고속 및 메모리 경량 구현 과정 요약

본 논문에서는 non-IID 트랙 최소 엔트로피 추정 알고리즘의 고속 및 메모리 경량 구현 방법과 이를 적용한 C++ 코드를 제안하며, 그 과정은 다음과 같다.

1) 기존 NIST의 SP 800-90B Python코드를 C++ 코드로 단순변환
2) 단순 변환된 C++ 코드의 속도 및 사용 메모리 측정
3) 측정된 결과 구동 시간이 상대적으로 가장 큰 상위 3 개 추정법과 메모리 사용량이 가장 큰 추정법에 대한 알고리즘 분석을 통해 고속 및 경량 가능 요소 도출
4) 분석결과를 기반으로 기존 알고리즘을 재구성하여 고속 및 경량 구현된 C++ 코드 구현

2.4 프리딕터를 사용한 엔트로피 추정법

NIST에서 공표한 SP 800-90B 두 번째 수정안과 초안[10]의 최소 엔트로피 추정에서 중요 차이점 중 하나는 non-IID 트랙의 최소 엔트로피 추정 방법에 4 가지의 프리딕터(predictor) 추정법이 추가된 것이다. 프리딕터를 이용한 추정법은 특정 예측함수를 이용하여 엔트로피 소스의 확률모델을 구성한 후 이를 바탕으로 최소 엔트로피를 추정하는 방법이다. 예측함수는 관측된 표본을 기반으로 다음 표본 값을 예측하는 함수이다. 일반적으로 예측함수의 예측 성공 횟수가 높을수록 최소 엔트로피는 낮게 추정된다[6]. 2.3 절에서 제시한 기준에 의해 고속화 및 메모리 경량 구현 대상으로 선정한 3 개의 추정법 MultiMCW, MultiMMC, LZ78Y는 모두 프리딕터를 이용한 추정법이다.

III. SP 800-90B 고속 및 메모리 경량 구현

3.1 NIST의 SP 800-90B Python 코드에 대한 C++ 코드 단순 변환 결과

NIST의 10 개의 최소 엔트로피 추정법에 대한 Python 코드와 이를 단순 변환한 C++ 코드의 구동 시간과 사용 메모리를 비교한 결과는 Table 1과 같다. 이 중 C++ 코드로 단순 변환시켰음에도 상대적으로 구동 시간이 긴 MultiMCW, MultiMMC, LZ78Y, 세 가지의 알고리즘을 고속화하기 위한 분석을 진행하였다.

특히 MultiMCW 알고리즘의 경우 Python 코드에서 600 초가 넘는 시간이 소요되어 엔트로피 추정 시간 지연의 가장 큰 요인이었으며, 단순 변환 이후에도 약 300 초로 가장 큰 속도 지연의 요인이 되었다. 한편, MultiMMC 알고리즘의 경우 속도 지연 문제 뿐 아니라 5.5 GB가 넘는 메모리를 사용해 과도한 메모리 사용 문제가 발생되었으며, C++ 코드로 단순 변환시킨 이후에도 약 2.5 GB로 메모리 사용량이 많다는 것을 확인하였다. 실제로 NIST의 Python 코드에 대한 의견이 제시된 웹페이지[7]에는 RAM 8 GB의 환경에서 MultiMMC 구동 중 메모리 문제로 구동에 실패한 사례들이 다수 제기되었다. 그래서 MultiMMC 알고리즘에 대해서는 메모리 경량 구현 측면에 비중을 두고 분석을 진행하였다. LZ78Y의 경우는 MultiMMC의 분석 결과를 유사하게 적용할 수 있었다.

Table 1. Running times and memory usages of estimators in non-IID track

	Estimator	MCV	Collision	Markov	Compression	t-Tuple
NIST's Python	Time(seconds)	0.04	0.45	1.44	27.59	0.42
	Memory(MB)	0.21	3.19	0.06	32.16	5.14
Converted C++	Time(seconds)	0.001	0.008	0.002	2.27	0.55
	Memory(MB)	0.02	3.81	1.08	22.93	5.7
	Estimator	**LRS**	**MultiMCW**	**Lag**	**MultiMMC**	**LZ78Y**
NIST's Python	Time(seconds)	3.70	671.56	33.13	411.27	119.08
	Memory(MB)	103.86	8.00	7.70	5789.16 (5.65 GB)	65.48
Converted C++	Time(seconds)	2.72	307.70	0.18	88.92	21.63
	Memory(MB)	103.57	103.83	7.69	2549.20 (2.50GB)	127.01

3.2 MultiMCW 알고리즘에 대한 고속화 기법

3.2.1 MultiMCW 추정법 소개

프리딕터 추정법 중 하나인 MultiMCW(Multi Most Common in Window)는 엔트로피 소스로부터 출력된 수열의 최빈값이 확연히 도출되면서 시간에 따라 그 값이 다양하게 변하는 경우에 효과적인 최소 엔트로피 추정법이다[6]. 여기서 최빈값이란 표본 값 중 발생 빈도가 가장 높은 값을 의미한다.

MultiMCW 프리딕터는 4 개의 부분 프리딕터로 구성되며, 각각의 부분 프리딕터는 고정된 예측 길이를 갖는 예측함수(*mostCommon*)를 사용한다. 순서에 따른 예측함수와 예측 길이는 Table 2와 같다. MultiMCW 프리딕터의 예측함수는 입력된 예측 길이의 표본 수열에서 최빈값을 계산해 예측값으로 결정한다. 만약 최빈값이 두 개 이상이라면 최빈값을 표본 값으로 갖는 표본 중 가장 늦게 예측 함수에 입력된 표본 값을 예측값으로 결정한다. MultiMCW 예측함수의 의사코드는 Fig.2에 나타나 있다.

Table 2. Prediction function with the prediction length

Table Head	Prediction functions with the prediction length			
Prediction function	*mostCommon[1]*	*mostCommon[2]*	*mostCommon[3]*	*mostCommon[4]*
Prediction length	63	255	1023	4095

```
Pseudo-code of the mostCommon[i]
Step.1. Input sequence X = {x[1], ..., x[w[j]]}
Step 2. if number of modes in X = 1
            return (the mode in X)
        else
            return (the lastest same value
                        among the modies in X)
```

Fig. 2. The pseudo-code of the *mostCommon[i]*

3.2.2 NIST의 MultiMCW 구현 알고리즘을 개선한 고속 C++ 코드

MultiMCW의 *mostCommon[j]*를 예측값으로 갖는 j번째 부분 프리딕터를 *Sub_predict[j]*라 할 때, MultiMCW 알고리즘에서 소요되는 연산의 대부분을 *Sub_predict[j]*에서 차지한다. *Sub_predict[j]*는 Table 3에 제시한 5 개의 내부함수로 구성된다.

본 논문에서는 MultiMCW 알고리즘을 분석하였으며, 분석을 바탕으로 크게 두 가지의 고속화 방법으로 알고리즘을 재구성하였다. 첫 번째 방법은 초기에 결정되는 표본 크기에 따라 기존 Counter 클래스를 단순 배열로 표현하고 이를 기반으로 알고리즘을 효율적으로 재구성하는 것이다. 두 번째 방법은 불필요한 연산을 소요하는 내부 함수를 최적화하는 것이다. 기존 MultiMCW의 Python 코드에 사용된 최빈값을 구하는 함수와 예측값을 구하는 함수의 경우 Counter 클래스를 이용하는데, 이때 필요한 연산 이상의 연산이 소요된다.

한편, Python 배열의 경우 유리수, 문자열 등 고정되지 않은 다양한 형태(type)의 입력 값을 받을 수 있다. 이와 반대로 C++ 배열의 경우 고정된 형태의 입력 값만을 받을 수 있고, 이에 따라 입력 값에 대한 메모리는 개발자에 의해 고정되어야 한다. Python의 이러한 특징은 구현의 편리성에는 도움을 주지만 제안하는 첫 번째 방법과 같이 형태와 길이가 고정된 배열을 사용하는 알고리즘은 C++ 코드에서 구현하는 것이 효과적이다. 이에 대한 실험적 결과는 Ⅳ장에 제시한다.

NIST의 MultiMCW Python 코드에서 *counters[j]*는 Counter 클래스로 입력된 표본 값에 대응되는 빈도수를 자동으로 저장하는 기능을 갖는다. 최소 엔트로피 추정 초기에 설정된 표본 크기에 의해 *counters[j]*에 입력되는 표본 값에 대한 경우의 수는 고정된다. 예를 들어 표본의 크기를 8-bit로 지정하였다면, *counters[j]*는 길이가 $256(=2^8)$인 배열로 표현할 수 있다. 이처럼 경우의 수가 고정되는 특징을 이용하여 기존 Counter 클래스로 표현된 *counters[j]*를 고정된 길이의 단순 배열로 변경할 수 있으며, 기존 MultiMCW에서 사용한 Counter 클래스의 기능을 동일하게 표현할 수 있다.

Table 3. The Internal functions of Sub_predict[j]

Internal functions of Sub_predict[j]
Counter(value)
Counter.subtract(value)
Counter.update(value)
Counter.most_common()[0][1]
mostCommon(tuple, value)

Counter 클래스의 기능 중 입력되는 표본 값의 빈도수를 자동으로 기록하는 기능을 위하여 배열의 순서를 표본 값에 대응시키고, 순서에 따른 입력 값은 표본 값의 빈도수에 대응시킬 수 있다.

Counter 클래스는 dict 클래스의 부분 클래스(subclass)로 데이터를 해시 테이블(hash table) 형태로 기록한다. 데이터를 기록 및 탐색을 위하여 1 회 이상의 해시함수를 사용한다. 그러나 여기서 제안하는 방식대로 고정된 크기의 배열로 구현할 경우 단순 배열 입출력만으로 *Sub_predict[j]*의 모든 내부 함수를 구현할 수 있다. 해시함수의 1 회 구동 시간과 배열 입·출력 속도는 크게 차이가 난다. 그 차이로 인해 기존 MultiMCW 코드와 제안하는 MultiMCW 코드의 구동 시간이 현저하게 차이 나는 것으로 분석된다.

Counter 클래스의 *subtract* 함수는 *counters[j]*에서 입력된 표본 값에 대응하는 빈도수를 찾아 1을 빼는 함수이다. *subtract* 함수는 Fig.3에 표현되어 있다. Counter 클래스의 *subtract* 함수는 *counter[j]*를 배열 형식으로 바꿔줄 경우 배열 값에 대한 뺄셈 연산으로 구현할 수 있으며, *update* 함수에 대해서도 유사하게 구현할 수 있다. update 함수와 subtract 함수를 배열로 구현한 결과는 Fig.4와 같다. 기존 *subtract* 함수의 경우 입력 값을 해시 테이블형태로 기록하게 된다. 이에 따라 최소 1 회의 해시함수를 구동시키게 되며, 이외에도 몇몇 기능을 위해 비교 연산을 약 5 회 진행하게 된다. 그러나 제안하는 *subtract* 함수는 Fig.4와 같이 배열 입력 1 회, 배열 출력 1 회로 표현이 된다.

most_common 함수는 최빈값을 결정하는 데 사용된다. *most_common* 함수의 Python 코드는 Fig.5와 같다. Counter 클래스의 *most_common* 함수는 *counters[j]*에 입력된 표본 수열을 빈도수에 따라 순서를 재배치한 결과를 출력한다. 이를 위하여 평균 시간 복잡도가 $O(k\times\log k)$인 *sorted* 함수를 사용한다. 그러나 MultiMCW에서는 최빈값만 구하면 되기 때문에 Fig.6과 같이 시간 복잡도 $O(k)$로 구현이 가능하다.

기존 코드의 경우 빈도수를 호출하기 위하여 매번 1 회 이상의 해시함수 구동이 필요하다. 이로 인해 *sorted* 함수를 구동하는데 약 $k\times\log k$ 회의 해시함수 구동 및 비교 연산이 이루어진다. 본 논문과 같이 표본 크기가 8-bit인 경우 k는 $2^8=256$이다. 그러나 제안하는 코드는 Fig.6과 같이 k 회의 고정된 횟수의 배열 호출 및 비교 연산만으로 구동된다.

mostCommon 함수는 가장 나중에 입력된 최빈값을 출력하는 함수로 Fig.7과 같다. *mostCommon* 함수는 Counter 클래스의 *most_common* 함수를 내부에서 호출하여 최빈값을 찾아 *maxcount*에 입력한다. 이후 입력된 부분 표본 수열에서 표본 값에 대응하는 빈도수를 *maxcount*와 비교하여 같다면 표본 값에 대응하는 순서를 *lastindex*에 입력한다. 이후 다른 최빈값이 발견되면 순서를 *lastindex*와 비교해

Counter.subtract(*args, **kwds)

```
self, *args = args
iterable = args[0] if args[0] if args else None
if iterable is not None:
  self_get = self.get
  if instance(iterable, Mapping):
    for elem, count in iterable,items():
      self[elem]=self_get(elem,0)-count
    else:
      for elem in iterable:
        self[elem] = self_get(elem,0)-1
if kwds:
  self.subtract(kwtd)
```

Fig. 3. The subtract code of Counter Class

subtract(counters, value)

```
counters[j][value] -= 1;
```

update(counters, value)

```
counters[j][value] += 1;
```

Fig. 4. The proposed subtract and update code

Counter.most_common(self,n=Non)

```
if n is None:
  return sorted(self.items(), key=_itemgetter(1),
         reverse=True)
return  _heapq.nlargest(n,self.items(),key=_item
        getter(1))
```

Fig. 5. The most_common code of Counter class

most_common(counters,k)

```
maxcount = 0;
for t in rage(k){
  if counters[t] > maxcount
    maxcount = counters[t];
}
return maxcount;
```

Fig. 6. The proposed most_common code

가장 나중에 입력된 최빈값을 찾는다.

*mostCommon*은 입력된 표본 수열 *S*의 내부 표본 값을 끝에서부터 거꾸로 호출하여, 호출된 표본 값에 대응하는 빈도수를 *maxcount* 값과 비교하는 것으로 바꿀 수 있다. 이 경우 최초 *maxcount*와 같은 빈도수를 갖는 표본 값을 찾는 것으로 알고리즘은 끝나게 된다. 구현된 코드는 Fig.8과 같다.

결론적으로 제안하는 *Sub_predict[j]* 코드는 Fig.9와 같이 구현된다. 제안된 MultiMCW 고속 구현 C++ 코드의 경우 1 초 이하로 시간을 단축해, 기존 NIST의 MultiMCW Python 코드 대비 700 배 이상의 속도가 향상했다. 이 결과는 SP 800-90B non-IID 트랙의 구동 시간을 절반 이하로 줄이는 결과를 보여준다.

3.3 MultiMMC 알고리즘에 대한 효율적인 메모리 사용 기법

3.3.1 MultiMMC 추정법 소개

프리딕터 추정법 중 하나인 MultiMMC(Multi Markov Model with Counting)는 엔트로피 소스의 출력이 길이가 N인 마코브(markov) 모델로 표현 가능할 경우에 효과적인 최소 엔트로피 추정법이다[6]. 마코브 체인이라고도 불리는 마코브 모델은 확률 모델 중 하나로 과거와 현재의 상태가 주어졌을 때, 미래 상태의 조건부 확률 분포가 과거 상태와는 독립적으로 현재 상태에 의해서만 결정된다는 것을 뜻한다.

MultiMMC는 길이 1에서 16까지의 마코브 모델을 모두 기록한다. 각각의 마코브 모델은 MultiMMC의 하나의 부분 프리딕터로써 입력받은 지정된 길이의 표본 수열과 표본 수열의 다음 표본 값, 그리고 표본 수열과 표본 값 쌍의 입력 횟수를 기록한다. 각각의 부분 프리딕터는 예측값을 결정하기 위하여 지정된 길이의 표본 수열을 입력받아 기록된 표본 수열의 다음 표본 값으로 가장 입력 횟수가 높은 값을 예측값으로 결정한다.

```
mostCommon(S, counters)
maxcount = counters.most_common()[0][1]
maxsymb = None
latindex = len(S)
revers=S[::-1]
for s in set(S):
  if(counters[s]==maxcount)and(reverse.index
  (s)<lastindex):
    lastindex = reverse.index(s)
    maxsymb = s
return maxsymb
```

Fig. 7. The mostCommon code of Counter class

```
mostCommon(S, counters)
maxcount = most_common(counters,k);
for(int i=len(S) - 1; i >=0; i--)
  if counters[i]==maxcount
    return i;
```

Fig. 8. The proposed mostCommon code

```
Sub_predict[j]
counters[j][S[i-W[j]-2]-=1;
counters[j][S[i-2]+=1;
sub_predict[j] = mostCommon(counters[j],S,k)
```

Fig. 9. The proposed Sub_predict[j] code

3.3.2 NIST의 MultiMMC 구현 알고리즘의 메모리 문제 분석

MultiMMC 추정법의 가장 큰 특징은 프리딕터에 입력되는 모든 표본 수열을 기록하고 있어야 한다는 것이다. 결과적으로 최소 엔트로피 추정을 위해 입력되는 100만 표본의 수열에 대한 길이 1~16의 표본 부분 수열을 저장할 수 있어야 한다. NIST의 MultiMMC 알고리즘은 프리딕터에 입력되는 표본 수열을 사전(dict, dictionary)형식으로 기록한다. 사전이란 키(key)를 값(value)에 대응시키는 일방향 치환(map)을 기록한다. Python의 사전은 해시 테이블을 사용하는데, 입력되는 키의 해시값으로 키에 대응하는 값의 저장 위치를 결정하게 된다. 해시값이란 해시함수에 의해 산출된 값을 의미한다. 해시 테이블은 충돌이 발생하지 않는다면 키의 탐색 시간은 해시함수의 구동 시간으로 고정되게 된다는 특징을 갖는다. 이에 따라 해시 테이블로 구현된 사전 데이터 구조는 메모리가 풍부한 환경에서 많은 데이터의 치환을 저장할 때 효과적이다.

NIST의 MultiMMC 알고리즘에서 지정된 길이의 표본 수열과 다음 표본의 값 그리고 그 발생 빈도수를 기록하는 Python 코드는 Fig.10에 나타나있다. 길이 d는 1부터 16까지를 사용하며, 길이에 따라 사전 형식으로 길이가 d인 마코브 체인을 구성하여 Fig.10의 M[d]로 구현된다. M[d]는 길이 d의

The MultiMMC Python code's record process

```
if M[d-1].get(tuple(S[i-d-2:i-2]),0)==0:
  M[d-1][tuple(S[i-d-2:i-2])] = Counter()
M[d-1][tuple(S[i-d-2:i-2])][S[i-2]] += 1
```

Fig. 10. The process to record tuples, corresponding samples and the observation frequency of the samples in MultiMMC Python code

표본 수열을 입력받아 Counter 클래스로 치환하는 사전을 구성한다. Counter 클래스는 길이 d의 표본 수열 다음에 오는 표본 값과 그 빈도수를 저장한다. 입력되는 표본 수열은 최소 엔트로피 추정에 사용되는 100만 표본 수열의 부분 수열로, 순서대로 길이 d씩 입력된다. 표본 수열(1,2,3,1,2,3,2)을 예로 들면 M[2]와 M[3]의 경우 순서대로 각각 길이 2, 3을 입력받아 사전 형식으로 Fig.11과 같이 저장된다. Fig.10과 같이 사전 형식으로 메모리를 저장할 경우 Counter 클래스로 할당된 메모리를 제외하더라도 대량의 표본 수열을 저장하게 된다. 예를 들어 M[16]의 경우 길이 16의 표본 수열을 극단적인 경우 약 100만 개를 기록하는 많은 메모리가 사용될 것이다.

실험을 통해 NIST의 MultiMMC 코드는 약 5.5 GB의 메모리를 사용하여 RAM 8 GB의 환경에서 기존 최소 엔트로피 추정 프로그램의 메모리 문제가 발생하는 것을 확인하였다. Python의 사전 형식은 해시 테이블에서 저장하고 있는 메모리 크기의 2 배 이상의 메모리가 가용 되어야 한다. 이러한 이유로 5.5 GB의 해시 테이블 메모리 사용은 약 11 GB의 RAM 메모리가 필요하다. 따라서 RAM 8 GB의 컴퓨팅 환경에서는 NIST의 코드를 구동하는데 메모리 문제를 갖게 된다. 이러한 이유로 본 논문에서는 MultiMMC에 대해서만 RAM 32 GB의 컴퓨팅 환경을 이용한 실험 결과를 제시하였다. 이외에도 프로그램 구동에 있어 다양한 환경적 요인에 의한 메모리 사용이 동시에 이루어져 메모리 문제가 발생할 수 있다.

기존 NIST 코드에 대한 실험적 분석 결과 같은 길이의 마코브 모델을 구성하는 표본 수열의 많은 부분 수열들이 중복되는 특징이 확인되며, Fig.11과 같이 서로 다른 길이의 마코브 체인 내부에 저장된 표본 수열끼리도 많은 부분이 중복되었다.

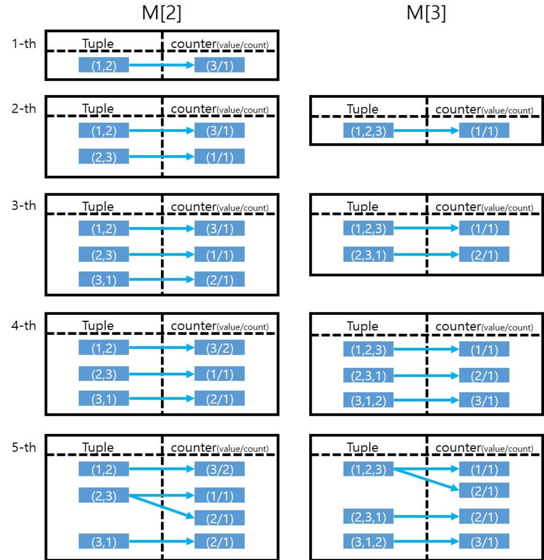

Fig. 11. The memory example of M[2] and M[3]

3.3.3 트리 구조를 활용한 마코브 체인 구성

트리(tree) 데이터 구조란 뿌리(root)노드, 자식(children)노드, 부모(parent)노드가 계층에 따라 분류되며 관계가 있는 노드끼리 연결되어 뿌리부터 시작되는 각각의 길(path)을 기록한다. 트리 구조는 길 사이의 중복되는 부분이 많을 경우 모든 길을 따로 저장하는 것과 비교해 메모리 사용량이 적다는 특징을 갖는다.

MultiMMC 알고리즘은 엔트로피 추정을 위해 기록하는 표본 수열끼리 중복되는 부분이 많다는 특징이 확인된다. 이러한 특징에 의해 기존 사전 데이터 구조를 트리 구조로 재구성할 경우 메모리를 사용량을 효과적으로 줄일 수 있다.

MultiMMC에 트리 구조를 적용하기 위하여 트리의 노드들은 Fig.12와 같이 구조체로 선언하였다. 각 노드는 자신에 대응되는 표본 값(value), 자신이 길의 끝 노드로 관측된 빈도수(count), 같은 계층의 다음 노드에 대한 메모리 주소(branch_next), 자신의 자식 노드에 대한 메모리 주소(branch_up)의 정보를 기록한다. 마코브 모델의 구성은 다음의 과정을 따른다. 먼저 표본 크기에 따라 뿌리를 생성하여 0계층을 이룬다. 예를 들어 표본 크기가 8-bit인 경우 뿌리는 256($=2^8$) 개를 생성하게 된다. 다음으로 각 뿌리는 첫 번째 자식의 메모리 주소를 갖는다. 뿌리의 첫 번째 자식은 자신과 같은 계층에 있는 다른 노

tree

unsigned char value;
unsigned int count = 1;
tree* branch_next = NULL;
tree* branch_up = NULL;

Fig. 12. The structure of a tree node

드의 메모리 주소를 가지며, 각 노드는 자신이 길 끝의 노드로 발생한 빈도를 기록하는 것으로 1 계층을 구성한다. 구성된 0 계층과 1 계층을 잇는 각각의 길(path)은 길이 1의 표본 수열과 다음 표본 값 하나를 기록한 것과 같으며, 이 길들의 집합이 길이 1의 마코브 모델을 구성하게 된다. 1 계층의 노드들은 각각 자신의 첫 번째 자식 노드의 메모리 주소를 가진다. 첫 번째 자식 노드는 자신과 같은 계층에 있는 다른 노드의 메모리 주소를 가지며, 자신이 길 끝의 노드로 발생하는 길의 빈도수를 기록하는 것으로 2 계층을 구성한다. 구성된 0, 1, 2 계층을 잇는 각각의 길(path)은 길이 2의 표본 수열과 다음 표본 값 하나를 기록한 것과 같으며, 이 길들의 집합이 길이 2의 마코브 모델을 구성한다. 이러한 방식으로 계층 16까지의 트리, 즉 길이 16까지의 마코브 모델을 구성한다. 결과적으로 모든 길의 집합은 기존 MultiMMC 알고리즘에서 기록하는 데이터와 같은 데이터를 기록하게 된다.

3.3.4 트리 구조를 활용한 메모리 경량 MultiMMC C++ 코드

제안하는 마코브 모델을 트리 구조로 기록하는 함수는 Fig.13과 같다. 길이 d의 마코브 모델의 경우 길이 d의 입력된 표본 수열과 그다음 표본 값을 기록한다. 입력된 수열에 대응하는 길을 찾아 길의 끝에 있는 노드의 자식 노드 중에 입력된 다음 표본 값을 기록한 노드가 있는지 확인한다. 대응하는 자식 노드가 있다면 해당되는 자식 노드의 발생 빈도에 1을 더하게 되며, 대응하는 자식 노드가 없다면 새로운 자식 노드를 생성하여 길을 잇고, 발생 빈도를 1로 초기화한다.

마코브 모델에 기록된 데이터를 이용하여 다음 예측값을 결정하는 기존 Python 코드는 Fig.14에 나타나 있다. 기존의 Python 코드는 사전 형식으로 입력된 표본 수열을 찾아 대응되는 Counter에서 가장 빈도수가 높은 값을 찾아 예측값으로 결정한다. 한편, SP 800-90B 문서에는 언급되지 않았지만, NIST에서 제공하는 MultiMMC 추정 Python 코드에 따르면 빈도수가 가장 높은 값이 두 개 이상일 때 표본 크기가 큰 값을 예측값으로 결정한다.

update_tree(root, S, d)

```
tree* pointer;
if (root[S[0]].branch_up == NULL) {
  root[S[0]].branch_up=pointer=new tree;
  pointer->value = S[1];
  return 1;
}
pointer = root[S[0]].branch_up;
for (int i = 1; i < d; i++) {
  while (pointer->value!= S[i])
    pointer = pointer->branch_next;
  if (pointer->branch_up == NULL) {
    pointer = pointer->branch_up = new tree;
    pointer->value = S[i+1];
    return 1;
  }
  pointer = pointer->branch_up;
}
while (pointer->value != S[d])
  if (pointer->branch_next == NULL) {
    pointer = pointer->branch_next = new
    tree;
    pointer->value = S[d];
    return 1;
  }
  else
    pointer = pointer->branch_next;
pointer->count++;
return 1;
```

Fig. 13. The proposed process to update the markov model of length d

predict(M, S)

```
if M[d-1].get(tuple(S[i-d-1:i-1]), None) != No
ne:
  for y in sorted(M[d-1][tuple(S[i-d-1:i-1])].ke
  ys()):
    if M[d-1][tuple(S[i-d-1:i-1])][y] >= M[d-1]
    [tuple(S[i-d-1:i-1])][ymax]:
      ymax = y
  if M[d-1][tuple(S[i-d-1:i-1])][ymax] > 0:
    subpredict[d-1] = ymax
  else:
    subpredict[d-1] = None
```

Fig. 14. The existing process to decide predict value

제안하는 트리 구조를 활용해 예측값을 결정하는 과정을 *sub_predict*라 했을 때, *sub_predict*는

Fig.15와 같이 구현될 수 있다. 먼저 트리에서 입력받은 길이 d의 표본 수열에 대응하는 길을 탐색한다. 탐색 된 길의 마지막 노드의 자식 노드들의 발생 빈도를 비교하여 가장 높은 빈도수를 반환한다. 만약 탐색되는 길이 없다면 예측할 수 없음을 의미하고 −1을 반환한다. 한편 두 개 이상의 노드가 같은 수의 가장 높은 발생 빈도를 가질 경우 노드에 대응하는 표본 값 중 큰 값을 반환한다.

3.4 LZ78Y 알고리즘에 대한 고속화 기법

3.4.1 LZ78Y 추정법 소개

프리딕터 추정법 중 하나인 LZ78Y 추정법은 엔트로피 소스의 출력이 유사 LZ78Y 압축 알고리즘에 의해 효과적으로 압축될 경우 효과적인 최소 엔트로피 추정법이다[6].

LZ78Y 추정법은 프로세스 측면에서 보면 기록하는 표본 수열 개수에 제한이 있는 MultiMMC 추정법으로 볼 수 있다. MultiMMC 알고리즘과 같이 1~16의 예측 길이로 입력되는 표본 수열과 다음 표본 값 그리고 표본 값의 발생 빈도수를 기록한다. 다만 기록되는 표본 수열의 개수가 65,536 개로 제한된다.

3.4.2 LZ78Y 알고리즘 분석을 통한 고속화 요소 도출

NIST의 LZ78Y는 MultiMMC의 같이 사전 데이터 구조를 이용해 마코브 체인을 구성하였다. 사전 구조의 탐색 시간은 충돌이 없다면 사용되는 해시함수의 1회 구동 시간과 같아지게 된다. 그러나 제안하는 MultiMMC에 적용된 트리 구조의 탐색은 선형 탐색방식으로 기록된 데이터가 많을수록 구동 시간은 길어지게 된다. 그럼에도 제안한 MultiMMC C++ 코드는 단순 변환한 MultiMMC C++ 코드보다 빠른 속도를 가졌다. 따라서 MultiMMC보다 기록하는 표본 수열의 양이 현저하게 적은 LZ78Y의 경우, 트리 구조를 적용하는 것이 고속화에 더욱 효과적일 것을 예상할 수 있다.

NIST의 LZ78Y Python 코드는 Fig.16과 같이 최초 입력된 65,536개의 수열에 대해서만 사전 형식으로 기록한다. NIST의 ZZ78Y 예측 알고리즘은 Fig.17과 같이 입력되는 표본 수열을 사전에서 탐색하여 가장 많은 빈도수가 기록된 다음 표본 값을 예측값으로 결정한다. 한편 동점인 경우 표본 값이 큰 값으로 결정한다.

sub_predict(root, S, d)

```
tree* pointer;
  if (root[Samples[0]].branch_up == NULL)
    return -1;
  pointer = root[Samples[0]].branch_up;
  for (int i = 1; i < length; i++) {
    while (pointer->value != Samples[i])
    pointer = pointer->branch_next;
    if (pointer->branch_up == NULL)
      return -1;
    pointer = pointer->branch_up;
  }
unsigned char ymax = 0;
int ycount = 0;
while (pointer != NULL) {
  if (ycount < pointer->count) {
  ymax = pointer->value;
  ycount = pointer->count;
  }
  else if ((ycount == pointer->count) &&
  (ymax < pointer->value)) {
    ymax = pointer->value;
  }
  pointer = pointer->branch_next;
}
return ymax;
```

Fig. 15. The proposed process to decide the predict value

record(D, S)

```
k = tuple(S[i-j-2:i-2])
if k not in D and dictionarySize < maxDictiona
   rySize:
  D[k] = dict()
  dictionarySize = dictionarySize + 1
if k in D:
  D[k][S[i-2]] = D[k].get(S[i-2], 0) + 1
```

Fig. 16. The process to record in LZ78Y

predict(D, S)

```
for y in sorted(D[prev].keys(),reverse=True):
  if D[prev][y] > maxcount:
    predict = y
    maxcount = D[prev][y]
```

Fig. 17. The process to decide predict value in LZ78Y

3.4.3 트리 구조를 활용한 고속 LZ78Y C++ 코드

기록하는 표본 수열의 개수가 65,536 개로 제한된

LZ78Y의 경우 데이터 기록 방식을 트리 구조로 구성하여 고속 구현하였다. 트리의 노드는 MultiMMC에서 사용하는 노드의 구성을 일부 수정하여 구성하였으며, 이에 따른 트리의 기록 및 탐색 과정 알고리즘 또한 유사하게 구성하였다. 따라서 여기서는 제안한 MultiMMC C++ 코드와의 차이에 대해서만 제시하도록 하겠다.

LZ78Y 알고리즘과 MultiMMC 알고리즘의 가장 큰 차이는 기록하는 표본 수열 개수에 제한이 있다는 것이다. 이를 고려하여 트리 구조를 활용해 구성한 LZ78Y 알고리즘의 C++ 코드는 Fig.18과 같다.

트리 구조를 활용한 LZ78Y C++ 코드에 사용된 내부 함수인 *ini_tree_LZ78Y*는 입력된 수열을 기록하는 함수로 지정된 수의 표본 수열을 기록할 때 까지만 사용된다. *update_tree_LZ78Y* 함수는 *ini_tree_LZ78Y* 함수에서 표본 수열을 기록하는 과정을 제거하고 기존에 기록된 표본 수열을 탐색해 입력된 표본 수열과 비교하는 기능만 가진 함수이다. *predict_LZ78Y* 함수는 기록된 표본 수열을 탐색해 입력된 표본 수열의 길을 찾는다. 입력된 표본 수열에 대응하는 길을 탐색했다면 끝 노드에 연결된 모든 자식 노드의 값에 대한 발생 빈도 중 가장 높은 값 찾아 기존 예측값의 발생 빈도와 비교해 더 크면 예측값을 갱신한다. 사용된 내부 함수인 *ini_tree_LZ78Y*, *update_tree_LZ78Y*는 제안한 MultiMMC C++ 코드의 *update_tree*와 유사하게 구현되었으며 predict_LZ78Y는 제안한 MultiMMC C++ 코드의 *subpredict*와 유사하게 구현되었다.

LZ78Y entropy estimator C++ code

```
for (int d = 15; d >= 0; d--)
  if (dictionarySize < maxDictionarySize) {
    if (ini_tree_LZ78Y(root, pDataset + i - 1 -
       d, d, k))
      dictionarySize++;
  }
  else
    update_tree_LZ78Y(root, pDataset + i - 1
      - d, d, k);
for (int d = 15; d >= 0; d--)
  predict[1] = predict_LZ78Y(root, pDataset +
           i - d, d, predict, k);
```

Fig. 18. The proposed process to record and to predict in LZ78Y

IV. 고속화 및 효율적 메모리 사용 기법을 적용한 SP 800-90B 최소 엔트로피 추정법의 C++ 코드 실험 결과

논문의 실험은 Table 4와 같은 환경에서 이루어졌다. 본 논문의 모든 실험 결과는 10 회 이상의 실험을 통해 얻은 결과의 평균값이며, 각각의 실험은 서로 다른 100만 개의 표본 수열을 대상으로 했다.

제안하는 고속 SP 800-90B의 모든 최소 엔트로피 추정 C++코드는 기존 NIST 코드와 기능적인 면에서 완전히 동일하게 동작한다. [5]에 포함된 모든 테스트 소스(100만 바이트(bytes)의 바이너리 파일 4 개)에 대해 NIST의 코드와 동일한 결과를 출력하였으며, 진난수생성기[11]를 통해 출력한 서로 다른 100만 바이트 출력 표본에 대해 10 회 이상의 실험에서 모두 같은 출력을 보이는 것을 확인할 수 있었다.

Table 4. Experimental environment

Computing environment	
Processor	inter(R) Core(TM) i7-6700
CPU	3.40 GHz
RAM	8.0 GB
OS	Windows 10 / 64-bit
Using Program	Python 3.6 32-bit Visual Studio 2015 C++
Input data	
Length of samples	8-bit
Number of samples	1 million samples
Number of experiment	
Over 10 times	

4.1 고속 및 메모리 경량 SP 800-90B non-IID 트랙 C++ 코드 구현 결과

본 논문에서는 프로그램 언어를 C++로 사용하는 것이 최적화에 유의미할 것으로 판단하였다. 이에 따라 먼저 모든 추정법에 대하여 C++코드로 단순 변환하였으며, 변환 후에도 상대적으로 구동 시간이 긴 MultiMCW, MultiMMC, LZ78Y 세 가지 알고리즘에 대하여 분석 및 고속화를 진행하였다. 특히 메모리 문제의 원인이 되었던 MultiMMC의 경우 메모리 경량 구현에 초점을 맞췄다.

Table 5. Running times and memory usage of proposed estimators in non-IID Track

	Estimator	MCV	Collision	Markov	Compression	t-Tuple
Python	Time(seconds)	0.04	0.45	1.44	27.59	0.42
	Memory(MB)	0.21	3.19	0.06	32.16	5.14
Converted C++	Time(seconds)	0.001	0.008	0.002	2.27	0.55
	Memory(MB)	0.02	3.81	1.08	22.93	5.7
Proposed C++	Time(seconds)	0.001	0.008	0.002	2.27	0.55
	Memory(MB)	0.02	3.81	1.08	22.93	7.63
	Estimator	LRS	MultiMCW	Lag	MultiMMC	LZ78Y
Python	Time(seconds)	3.70	671.56	33.13	411.27	86.01
	Memory(MB)	119.30	8.00	7.70	5789.16 (5.65 GB)	25.04
Converted C++	Time(seconds)	2.72	307.70	0.18	88.92	21.63
	Memory(MB)	103.57	103.83	7.69	2549.20 (2.50 GB)	127.01
Proposed C++	Time(seconds)	2.72	0.90	0.18	73.23	8.18
	Memory(MB)	7.65	3.88	7.69	442.01	137.07

모든 추정법에 대한 Python 코드, 단순 변환 C++ 코드, 제안하는 C++ 코드 각각에 대한 구동 시간 및 메모리 사용량 측정 실험 결과를 Table 5에 제시한다. 제안된 SP 800-90B의 모든 최소 엔트로피 추정법을 구동하는데 88 초의 구동 시간과 442 MB의 메모리가 사용된다. 기존 코드 대비 14 배의 속도가 향상되었으며, 메모리 사용량은 1/13 배로 줄었다.

기존 추정법의 알고리즘을 재구성하여 고속 및 메모리 경량 구현한 MultiMCW, MultiMMC, LZ78Y 세 가지 추정법에 대한 실험적 결과와 그 분석 내용은 다음과 같다.

4.2 고속 MultiMCW C++ 코드 실험 결과

제안하는 MultiMCW의 고속 구현 알고리즘에 대한 C++ 코드와 Python 코드에서의 계산적 효율성을 실증적으로 확인하기 위하여, 제안하는 알고리즘을 Python 코드와 C++ 코드로 구현하여 비교했으며 그 결과는 Table 6과 같다.

고속 MultiMCW 알고리즘을 적용한 Python 코드의 경우 기존 Python 코드보다 약 5 배의 속도 향상으로 유의미한 결과를 보였다. 같은 알고리즘으로 구현한 MultiMCW C++ 코드의 경우 기존 NIST 코드의 700 배 이상의 속도 향상을 보였다. 또한, Table 5를 보면 알 수 있듯이 단순 변환한 MultiMCW C++ 코드 대비 300배 이상의 속도가 향상되어 제안한 알고리즘 개선 방법이 Python 보다 C++에 적절하다는 것을 확인할 수 있었다. 결과적으로 제안한 고속 MultiMCW C++코드를 사용할 경우 기존 non-IID의 구동 시간을 반으로 줄이는 결과로 충분한 고속화 결과로 판단된다.

Table 6. The comparison result for Python and C++ code using the proposed method

Algorithm	Python		C++
	NIST	Propo-sed	Proposed
Time (seconds)	671.56	136.36	0.90
Speed	1	4.92	746.17

4.3 메모리 경량 MultiMMC C++ 코드 실험 결과

기존 MultiMMC의 Python 코드는 메모리 사용량이 5.5 GB를 넘어 메모리 문제를 갖는다. 메모리 사용량이 많은 이유는 입력되는 길이 1~16의 모든 표본 수열을 저장하기 때문이다. 본 논문에서는 NIST의 MultiMMC 알고리즘의 특징상 기록되는 표본 수열 사이에 중복되는 부분이 많다는 것을 확인하였다. 트리 데이터 구조는 이러한 특징에 효율적 구현이 가능하다. 따라서 기존 사전 데이터 구조의 저장 방식을 트리 데이터 구조로 재구성하여 메모리 경량 MultiMMC를 C++ 코드를 구현하였다. 그 결과 Table 7과 같이 메모리 사용량을 기존의 1/13 배로 현저하게 감소시켜, 기존 Python 코드의 메모리 문제를 해결하였다. 나아가 제안하는 코드는 기존 코드 대비 약 5 배의 속도 향상을 보였으며, 단순 변환한 코드보다도 빠르게 구동되는 것을 확인하였다. 이러한 결과는 메모리 사용량 감소에 따라 탐색에 소

요되는 평균 시간이 해시함수의 1 회 구동 시간 보다 감소한 결과로 사료된다.

Table 7. The comparison results for time and memory usage of existing code, converted C++ code and proposed C++ code for MultiMMC

	Python	C++	
Algorithm	NIST	Converted	Proposed
Time (seconds)	411.27	88.92	72.23
Memory (GB)	5.65	2.50	0.43

4.4 고속 LZ78Y C++ 코드 실험 결과

NIST의 LZ78Y 추정법의 Python 코드는 약 65 MB의 메모리가 사용되며, 구동에는 약 120 초의 시간이 소요된다. 기존 LZ78Y Python 코드는 65,536 개로 제한된 표본 수열을 사전 형식으로 저장하여, 저장된 데이터가 많을 때 탐색에 효과적인 사전 형식의 장점이 무의미하다. 기존의 저장 방식을 트리 구조로 재구성한 고속 LZ78Y C++ 코드는 기존 Python 코드 대비 10 배 이상, 단순 변환 C++ 코드 대비 2 배 이상 속도가 향상된 것을 실험을 통해 확인하였다(Table 8). 이러한 결과는 LZ78Y의 기록하는 표본 수열 개수가 크지 않기 때문이다. 선형 탐색을 진행하는 트리 구조의 특성상 기록되는 데이터 크기와 비례하게 탐색 속도가 향상된다. 이러한 트리 구조의 특징에 의해 LZ78Y에서는 MultiMMC 보다 속도 측면에서 효과적인 향상을 확인할 수 있었다.

Table 8. The comparison result for existing python code, simple converted C++ code and proposed C++ code of LZ78Y

	Python	C++	
Algorithm	NIST	Converted	Proposed
Time (seconds)	86.01	21.63	8.31
Memory (GB)	65.48	127.01	137.07

Table 9. The methods of the high-speed implementation or memory reduction of entropy estimator in non-IID track

Estimator	Method of high-speed implementation or memory reduction
MultiMCW	To fix the parameters according to the setting of the sample size and to improvement of existing code
MultiMMC LZ78Y	To rebuild the markov model with the proposed tree structure
LZ78Y	To rebuild dictionary structure with the proposed tree structure

V. 결론 및 추후 연구

본 논문에서는 SP 800-90B의 최소 엔트로피 추정 방법에 대한 고속 구현과 효율적인 메모리 사용 기법을 제안하였다. 먼저 고속화를 위해 기존 NIST SP 800-90B의 모든 최소 엔트로피 추정법에 대한 기존 Python 코드를 C++ 코드로 단순 변환하여 구현했다. 나아가 C++코드로 단순 변환 후에도 상대적으로 구동 시간이 긴 MultiMCW, MultiMMC, LZ78Y의 경우 기존 알고리즘을 분석하여 고속화 가능 요소를 제시하였으며, 특히 구동 시 메모리 문제를 일으키는 MultiMMC에 대해서는 메모리 경량 구현 관점에서 분석한 결과를 제시하였다.

분석을 기반으로 도출한 각 추정법에 대한 고속화 및 메모리 경량화 방법은 Table 9와 같다. MultiMCW는 C++ 코드의 장점에 적합하도록 알고리즘을 재구성하였으며, MultiMMC는 데이터 저장 구조를 트리 구조로 재구성하여 메모리 사용량을 현저하게 줄였다. 또한, LZ78Y는 MultiMMC와 유사한 방식으로 데이터 저장 구조를 트리 구조로 재구성하여 탐색 속도를 높였다.

결과적으로 제안하는 방법이 적용된 C++ 코드의 총 구동 시간은 약 88 초로 기존 NIST의 코드 대비 약 14 배의 속도가 향상되었으며, 메모리 사용량은 440 MB로 이는 기존 NIST 코드 대비 메모리 사용량이 약 1/13로 감소하였다.

본 논문의 결과는 다양한 잡음원 또는 엔트로피 소스에 대해 반복적인 평가를 수행해야 하는 평가 기관 또는 개발자의 효율 향상에 도움이 될 것으로 기대한다. 이에 더해 메모리 문제의 해결로 RAM 8 GB 이하 컴퓨팅 환경에서도 실험이 가능하게 되어 다양한 제한된 환경을 가진 연구자들에게 도움이 될 것으로 기대한다.

본 논문에서는 다음의 두 가지 주제를 추후 연구과제로 남긴다. 첫째로 Lempel-Ziv 알고리즘 분석을 통한 LZ78Y 및 MultiMMC에 대한 추가적인

고속화 가능성을 타진하는 것이다. SP 800-90B에서 사용하는 LZ78Y는 압축 알고리즘인 Lempel-Ziv와 유사하게 구성된 최소 엔트로피 추정법이다. Lempel-Ziv 알고리즘은 이미 UNIX 파일 압축이나 GIF 이미지 등에서 널리 쓰이는 압축 방법으로, 다양한 최적화된 구현 결과가 존재한다. 이에 대한 분석을 통해 LZ78Y 및 알고리즘 상 유사한 MultiMMC에 대한 추가적 개선이 가능할 것으로 생각된다.

둘째, 제안한 방법을 적용한 SP 800-90B의 Cython[12] 코드 구현이 있다. 현재 SP 800-90B는 두 번째 드래프트 단계로 NIST는 [6]에서 머신러닝을 이용한 프리딕터 엔트로피 추정법을 업데이트 예정 사항으로 언급하였고, Python에는 다양한 머신러닝 프로그램이 구현되어 있기 때문에 NIST에서 SP 800-90B를 Python으로 구현하였다고 생각된다. 하지만 본문에서 언급한 것과 같이 Python에서 제공하는 함수는 엔트로피 추정 목적 이외의 기능을 수행하기 때문에 고속 구현에 적합하지 않다. 반면 C++의 경우 머신러닝관련 프로그램이 Python에 비해 상대적으로 부족하기 때문에 많은 기능을 모두 구현해야 하는 단점을 가진다. 이에 대한 해결책으로 Cython 언어를 이용한 구현 방법을 추후 연구 과제로 남긴다. Cython이란 기존 Python 언어를 기반으로 데이터 타입을 C++과 같이 지정하여 프로그램을 효율적으로 구현할 수 있도록 만들어진 프로그래밍 언어로 C++에 가까운 성능과 기존 Python의 가독성, 방대한 라이브러리를 이용한 쉬운 구현 등의 장점을 갖는 것으로 알려져 있다. Cython을 이용해 제안된 방법으로 SP 800-90B를 구현할 경우 기존의 Python의 장점과 향후 머신러닝과의 호환성 등은 그대로 가지고 가면서 제안된 고속 및 메모리 경량 SP 800-90B C++ 코드에 가까운 성능을 낼 수 있을 것으로 생각한다.

References

[1] E. Barker and J. Kelsey, "Recommendation for Random Bit Generator (RBG) Constructions", (Second Draft) NIST SP 800-90C, Apr. 2016.

[2] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", NIST SP 800-90A Revision 1, Jun. 2015.

[3] M.S. Turan, E. Barker, J. Kelsey, K.A. McKay, M.L. Baish, and M.Boyle, "Recommendation for the Entropy Sources Used for Random Bit Generation", (Second Draft) NIST SP 800-90B, Jan. 2016.

[4] NIST, SP 800-90B_EntropyAssessment, https://github.com/usnistgov/SP800-90B_EntropyAssessment, accessed Dec. 2017.

[5] H.C. Shin, S.J. Woo and D.J, Choi, Python3.2 programming, 3rd Ed., Wikibook, Jun. 2015.

[6] J. Kelsey, K.A. McKay, and M.S. Turan, "Predictive Models for Min-Entropy Estimation" International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2015, pp. 373-392, Sep. 2015.

[7] issues #31 : noniid_main.py Memory leak?, https://github.com/usnistgov/SP800-90B_EntropyAssessment/issues/31, accessed Dec. 2017.

[8] A speed comparison test of C, Julia, Python, Numba, and Cython LUFactorization, https://www.ibm.com/developerworks/community/blogs/jfp/entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization?lang=en, accessed Dec. 2017.

[9] PerformancePython that is an article of scipy.org, http://scipy.github.io/old-wiki/pages/PerformancePython, accessed Dec. 2017.

[10] E. Barker and J. Kelsey, "Recommendation for the Entropy Sources Used for Random Bit Generation", (First Draft) NIST SP 800-90B, Aug. 2012.

[11] Information of TrueRNG2, http://ubld.it/products/truerng-hardware-random-number-generator/, accessed Dec. 2017.

[12] Cython homepage, http://cython.org/, accessed Dec. 2017.

〈저자소개〉

김 원 태 (Wontae Kim) 학생회원
2017년 2월: 국민대학교 수학과 학사
2017년 2월~현재: 국민대학교 일반대학원 금융정보보안학과 석사과정
〈관심분야〉 난수성 분석 및 평가, 정보보안 프로토콜, 대칭키 암호 분석

염 용 진 (Yongjin Yeom) 종신회원
1991년 2월: 서울대학교 수학과 학사
1994년 2월: 서울대학교 수학과 석사
1999년 2월: 서울대학교 수학과 박사
2000년 4월~2012년 2월: ETRI 부설연구소 책임연구원/팀장
2006년 12월~2007년 12월: Columbia 대학교 방문 연구원
2012년 3월~현재: 국민대학교 정보보안암호수학과 부교수
2013년~현재: 국민대학교 BK21+ 미래 금융정보보안 인력양성사업단 교수
〈관심분야〉 암호구현 및 분석, 보안시스템 평가

강 주 성 (Ju-Sung Kang) 종신회원
1989년 2월: 고려대학교 수학과 학사
1991년 2월: 고려대학교 일반대학원 수학과 석사
1996년 2월: 고려대학교 일반대학원 수학과 박사
1997년~2004년: 한국전자통신연구원 선임연구원/팀장
2001년~2002년, 2010년: 벨기에 루벤대학 COSIC 방문 연구원
2004년~현재: 국민대학교 정보보안암호수학과 교수
2013년~현재: 국민대학교 BK21+ 미래 금융정보보안 인력양성사업단 교수
〈관심분야〉 암호이론, 정보보안 프로토콜, 안전성 분석 및 평가