Data Communication

Report about

"Reed-Solomon and Convolutional Codes"

과목명	데이터통신
지도교수	민 성 기 교수님
제출일	2006년 4월 11일
제출자	물리학과 2000160054 이 수 형
연락처	019-488-8701 huvx44@korea.ac.kr

1. Introduction

Error Detection 기법은 HDLC와 같은 데이터 링크 프로토콜에서 혹은 TCP와 같은 전송 프로토콜에서 매우 유용한 기술이다. 하지만 이를 무선통신과 같은 통신 시스템에 적용하기에는 다음과 같은 두 가지 이유로 부족한 면이 있다.

- 무선 링크에서의 Error rate는 매우 높기 때문에 재전송이 필요한 데이터가 많아지게 된다.
- 위성 링크와 같은 몇몇 케이스의 경우 propagation delay가 single frame의 전송 시간에 비해 매우 길기 때문에 error detection 기법을 사용할 경우 낭비되는 자원의 양이 많아지게 된다.

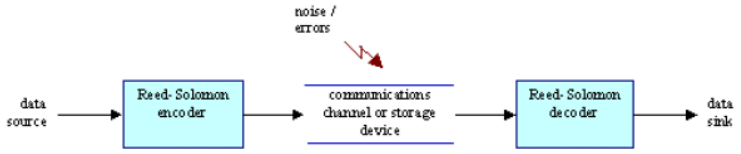
때문에 이러한 전송 시스템을 이용할 경우 error detection 기법 보다는 error correction 기법을 사용하여 에러를 보정하는 방법을 사용하는데 FEC (Forward Error Correction)기법의 하나인 Reed-Solomon Coding과 Convolutional Code가 많이 사용되고 있다.
본 레포트에서는 이 Reed-Solomon Coding과 Convolutional Code에 대해 살펴보고자 한다.

2. Reed-Solomon Code

Reed-Solomon Code는 block에 기반하여 에러를 보정하는 기법으로 현재 디지털 통신과 storage 분야 전반에 걸쳐 광범위하게 사용되고 있다. 일반적으로 다음과 같은 분야에서 Reed-Solomon Code가 많이 사용된다.

- 저장 장치 (Tape, Compact Disk, DVD, 바코드 등)
- Wireless 또는 Mobile 통신 (휴대폰, microwave 링크 등)
- 위성 통신
- 디지털 TV, DVB
- 초고속 모뎀 (ADSL, xDSL 등)

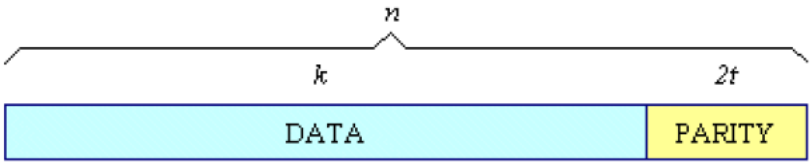
Reed-Solomon code의 기본적인 개념도는 <그림 1.1>과 같다.



<그림 1.1 Reed-Solomon code의 기본적 개념도>

Reed-Solomon encoder는 디지털 데이터를 block 단위로 받아들여 별도의 “중복” 비트를 추가한다. 데이터를 전송하거나 저장하는 과정에서 에러는 늘 발생할 수 있다. 예를 들면 전송 중의 노이즈의 발생이라 던지 CD에 생기는 스크래치 등으로 에러가 발생할 수 있다. 이렇게 에러가 추가된 데이터를 Reed-Solomon decoder는 각 block에 대해 에러를 보정하여 원래의 데이터를 복원시키게 된다.
Reed-Solomon code는 처음에 데이터 심벌을 이용해 polynomial을 생성하고 원본 데이터에 polynomial로부터 샘플링 된 코드를 추가하여 전송하게 된다. 이렇게 추가된 중복된 정보를 통해 원래의 polynomial을 재구축하고 에러를 제거하여 원본 데이터를 복구하게 된다.

Reed-Solomon code는 BCH코드의 부분집합이고 선형적인 block code이다. 일반적으로 Reed-Solomon 코드는 s개의 비트로 구성된 $RS(n,k)$로 표기한다. 이 표기법은 encoder가 s개의 비트마다 k개의 데이터 심벌을 받아들여 n개의 심벌 codeword를 만든다는 의미이다. 즉 각 s개 비트마다 $n-k$개의 패리티 심벌이 존재하게 된다. 이를 바탕으로 Reed-Solomon decoder는 t개($2t=n-k$)의 심벌을 보정하게 된다.
<그림 1.2>는 전형적인 Reed-Solomon codeword를 보여주고 있다.



<그림 1.2 Reed-Solomon codeword>

주어진 심벌 사이즈 s에 대해서 Reed-Solomon code가 갖는 최대 codeword의 길이 $n=2^s-1$이 된다.

예를 들어 가장 널리 쓰이는 Reed-Solomon code의 예인 8비트 심벌의 $RS(255,223)$의 경우를 들어보자. 각 codeword는 255개의 codeword 바이트를 갖게 된다. 여기서 223 byte는 데이터이며 32byte는 패리티를 위해 존재하게 된다. 즉,

$$n=255, k=223, s=8$$

이고,

$$2t=32, t=16$$

이 된다. decoder는 이 코드에서 16개의 심벌 에러를 보정할 수 있게 된다.
Reed-Solomon 코드는 특별한 기법을 통해 좀 더 단축될 수 있다. 데이터에 포함된 심벌들 중 0으로 구성된 부분을 단축시킬 수 있는데, 위에서 예를 든 $RS(255,223)$의 경우 55개의 zero byte를 이용하여 $RS(200,168)$로 단축시킬 수 있다. 즉 168byte의 데이터와 32byte의 패리티로 구성할 수 있게 된다.

한 심벌 내에서 1bit 또는 모든 bit에 에러가 발생하면 하나의 심벌 에러가 발생한 것이 된다. 예를 들어 $RS(255,223)$에서는 앞서 본 것과 같이 16개의 심벌 에러를 보정할 수 있는데 16비트의 에러가 발생하는 최악의 경우 각 심벌마다 에러가 발생하였다면 decoder는 최대 16비트의 에러만을 보정하게 된다. 최상의 경우 완전한 16byte의 에러가 발생하면 decoder는 16×8bit의 에러를 보정하게 된다. 따라서 Reed-Solomon 기법에서는 에러가 여러 개의 심벌에 걸쳐서 나타나는 경우 그 효율이 떨어지게 되고, 연속적인 에러에 대해 최상의 성능을 나타내게 된다.

Reed-Solomon 기법에 의한 decoding 과정은 에러의 보정과 삭제의 부분으로 구성된다. 에러 삭제 과정은 잘못된 심벌의 위치가 알려져 있을 때 일어난다. decoder는 t개 또는 그 이상의 에러를 보정할 수 있다. 삭제 정보는 종종 디지털 통신 시스템의 demodulator에 의해서 주어진다.
codeword가 decode 될 때 다음과 같은 3가지 결과가 가능하게 된다.

1. 만일 $2s+r<2t$ (s개의 에러와 r개의 삭제정보)인 경우 전송된 원본 데이터는 항상 복구될 수 있다.

2. 그 밖의 경우 decoder는 원본 데이터를 복구하지 못하는 것을 감지할 것이다.
3. 또는 decoder는 잘못된 데이터로 복구를 하게 된다.

이 세 가지 결과에 대한 가능성은 Reed-Solomon 기법의 부분에 의존하며 에러가 데이터에 발생된 정도에 따라 다르게 나타난다.
Reed-Solomon 기법을 사용하여 복구된 원본 데이터에 남아있는 에러는 Reed-Solomon 기법을 사용하지 않은 경우에 비해 매우 적은 양의 에러가 남아있게 된다. 이를 보통 coding gain이라고 한다.
Reed-Solomon encoding과 decoding 기법은 다음과 같은 방법을 통해 산출된다.

1) Finite (Galois) Field Arithmetic

Reed-Solomon 기법은 수학의 특정 분야인 Galois field 또는 finite field를 기반으로 하고 있다. Finite field는 사칙연산의 결과가 항상 해당 field에 속하는 성질을 가지고 있다. Reed-Solomon에서의 encoding 또는 decoding은 이러한 연산을 수행하도록 되어 있는데 이 연산은 특정 하드웨어 또는 소프트웨어 함수에 적용 시 필요하게 된다.

2) Polynomial 생성

Reed-Solomon codeword는 특정한 polynomial을 사용하여 생성된다. 모든 적합한 codeword는 생성된 polynomial에 의해 나누어 떨어지는데 이런 polynomial의 일반적인 형태는 다음과 같다.

$$g(x) = (x-\alpha^{i})(x-\alpha^{i+1})\cdot\cdot\cdot(x-\alpha^{i+2t})$$

이러한 polynomial을 사용하여 만들어지는 codeword는 다음과 같다.

$$c(x) = g(x).i(x)$$

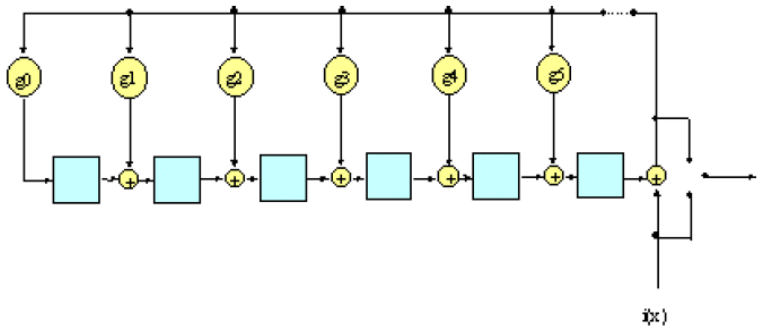
여기서 $g(x)$는 생성된 polynomial이고 $i(x)$는 정보 block을 의미한다. 예를 들어 $RS(255,249)$의 경우 다음과 같은 polynomial을 사용하게 된다.

$$g(x) = (x-\alpha^{0})(x-\alpha^{1})(x-\alpha^{2})(x-\alpha^{3})(x-\alpha^{4})(x-\alpha^{5})$$
$$g(x) = x^{6} + g_5x^{5} + g_4x^{4} + g_3x^{3} + g_2x^{2} + g_1x^{1} + g_0$$

이러한 배경을 둔 Reed-Solomon 기법의 encoding 과정은 다음과 같다. 2t의 패리티 심벌의 수를 갖는 Reed-Solomon codeword는 다음과 같이 주어진다.

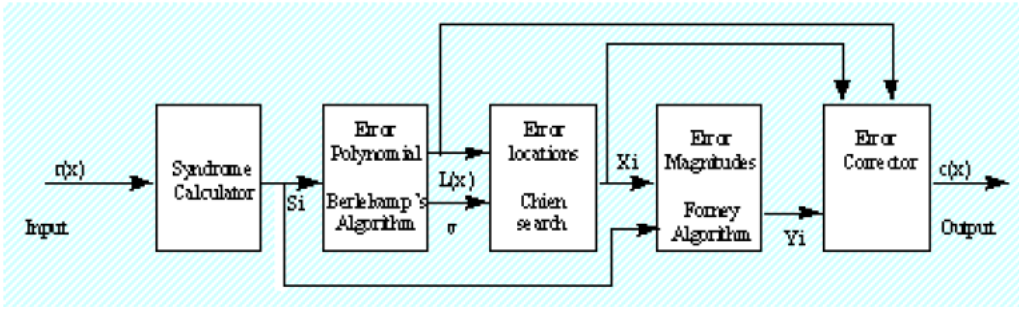
$$p(x) = i(x)\cdot x^{n-k} \bmod g(x)$$

<그림 1.3>은 $RS(255,249)$ encoder의 구조를 보여준다. <그림 1.3>에서 각 6개의 register는 하나의 심벌을 포함하게 된다. 즉 8bit로 구성된 심벌이 나타나며 연산과정은 finite field의 완전한 심벌의 덧셈 또는 곱셈으로 주어지게 된다.



<그림 1.3 $RS(255,249)$ encoder>

<그림 1.4>는 Reed-Solomon code의 decoding 과정을 보여주고 있다.



<그림 1.4 Reed-Solomon decoder>

<그림 1.4>에서 $r(x)$는 입력받은 codeword를 의미하며 Si는 syndrome을, $L(x)$는 error locator polynomial, Xi는 Error location, Yi는 Error magnitude, $c(x)$는 복구된 codeword, v는 에러의 개수를 의미한다. 입력받은 codeword $r(x)$는 원본 codeword $c(x)$에 에러를 더한 것으로 표시할 수 있다.

$$r(x) = c(x) + e(x)$$

Reed-Solomon decoder는 t개의 에러 (또는 2t개의 삭제)의 위치와 크기를 인식하고 에러를 보정하거나 삭제하게 된다.

결론적으로, Reed-Solomon error correction 기법은 finite field에서 정의되는 polynomial과 그 polynomial의 계수를 이용하여 에러를 보정하는 방법으로 Data storage나 무선통신, 위성통신과 같이 error detection 기법만으로 충분치 못한 데이터통신 링크에서 유용하게 사용될 수 있는 높은 신뢰도의 기법이다.

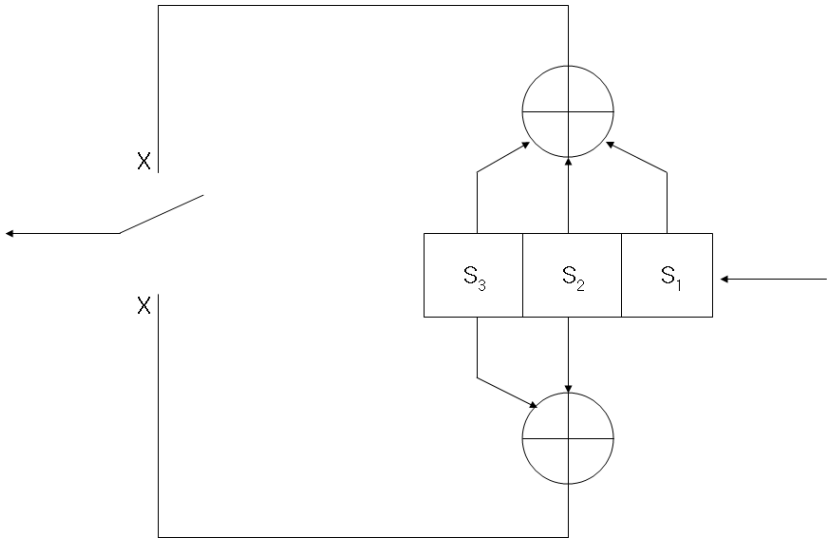
3. Convolutional Code

Convolutional code는 error correcting code의 일종으로 각 m개의 bit 정보 심벌을 n개의 심벌로 변환하여 encoding 한다. 여기서 m/n을 code rate (단 $n \geq m$)라고 한다. 변환 과정은 마지막 k개의 정보 심벌에 대한 함수로 표현되는데 여기서 k를 코드의 constraint 길이라고 한다. Convolutional code는 최소화된 회

로로 구성될 수 있기 때문에 모바일 통신 시스템에서 폭넓게 사용되고 있다. 그밖에 convolutional code는 라디오와 위성 링크에서 많이 사용되고 있다.

Convolutional code는 현재의 입력과 과거의 입력 간의 상호 연관성에 기반해서 부호화하는 방법이다. 따라서 부호기에는 입력 값을 저장하는 register가 있어야 하고 register값을 출력부분과 연결할 때 이용되는 polynomial이 필요하다. register의 개수를 m이라 하고 constraint 길이를 k라 하면 $k = m + 1$이 된다.

<그림 1.5>는 간단한 Convolutional encoder를 보여주고 있다.



<그림 1.5 간단한 Convolutional encoder>

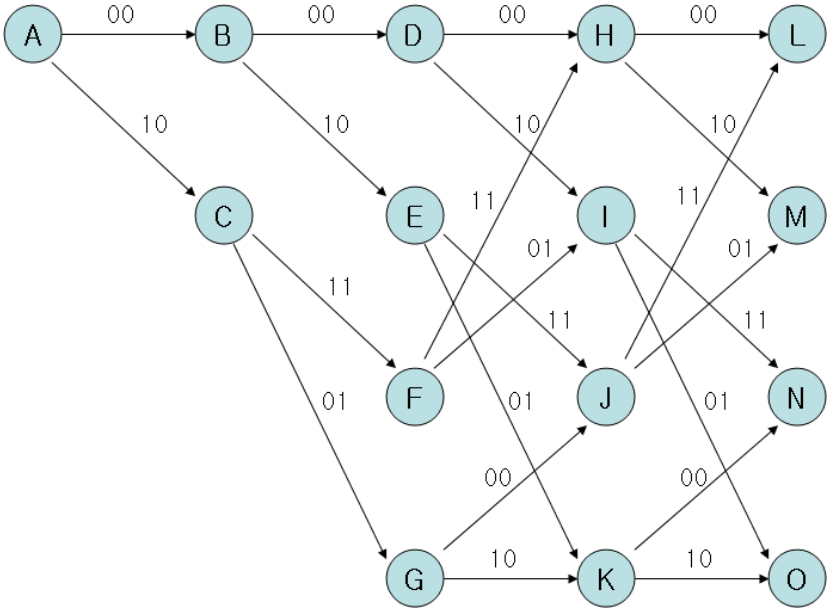
encode될 데이터는 <그림 1.5>의 encoder에 1bit씩 들어오게 된다. 각 입력에 대해 새로운 bit가 들어올 때마다 X와 Y의 값이 바뀌어 출력되는데 <그림 1.5>와 같은 encoder에서는 출력이 다음과 같은 표로 주어지게 된다.

Shift register			New Input	Output		Shift register			New Input	Output	
S_3	S_2	S_1		X	Y	S_3	S_2	S_1		X	Y
0	0	0	0	0	0	0	0	0	1	1	0
0	0	1	0	1	1	0	0	1	1	0	1
0	1	0	0	1	1	0	1	0	1	0	1
0	1	1	0	0	0	0	1	1	1	1	0
1	0	0	0	0	0	1	0	0	1	1	0
1	0	1	0	1	1	1	0	1	1	0	1
1	1	0	0	1	1	1	1	0	1	0	1
1	1	1	0	0	0	1	1	1	1	1	0

이러한 convolutional encoder는 <그림 1.6>과 같은 Trellis diagram으로 표기할 수 있다. Trellis diagram을 잘 살펴보면, 각 state에서 입력 0이 들어오면 위쪽 트리로, 1이 들어오면 아래쪽 트리로 이동하며 각 간선에 표기된 출력 값을 encoding된 데이터 bit으로 사용하게 된다.

Convolutional code의 encoding은 <그림 1.6>의 Trellis diagram을 참조하여 encoding 할 수 있다. Convolutional code를 encoding하는 과정은 이전에 입력되었던 bit값을 참조하여 다음 bit값을 encoding하는데 사용하게 되는데 이는 Trellis diagram의 각 state의 이동을 통해서 쉽게 encoding 할 수 있다.

예를 들어 연속적으로 들어오는 1011이라는 입력 데이터를 고려해보자. 초기상태는 당연히 S_1, S_2, S_3가 모두 0일 것이므로 A state에서 시작하게 된다. 이어서 첫 번째 bit 1이 입력되면 S_1 에 1이 들어가고 state A에서 입력 1이 들어왔으므로 state C로 이동하며 출력으로 10을 출력하게 된다. 이어서 두 번째 bit인 0이



<그림 1.6 Convolutional encoder의 Trellis diagram>

입력되면 C에서 상위 간선인 F로 이동하며 11을 결과로 출력한다. 이어서 1이 들어오면 하위 간선을 타고 I로 이동하며 01을 출력하고 마지막으로 1이 입력되면 하위 간선을 타고 O로 최종 도착하며 01을 출력하게 되어 다음과 같은 결과를 얻게 된다.

입력 : 1011
노드이동 : ACFIO
출력 : 10110101

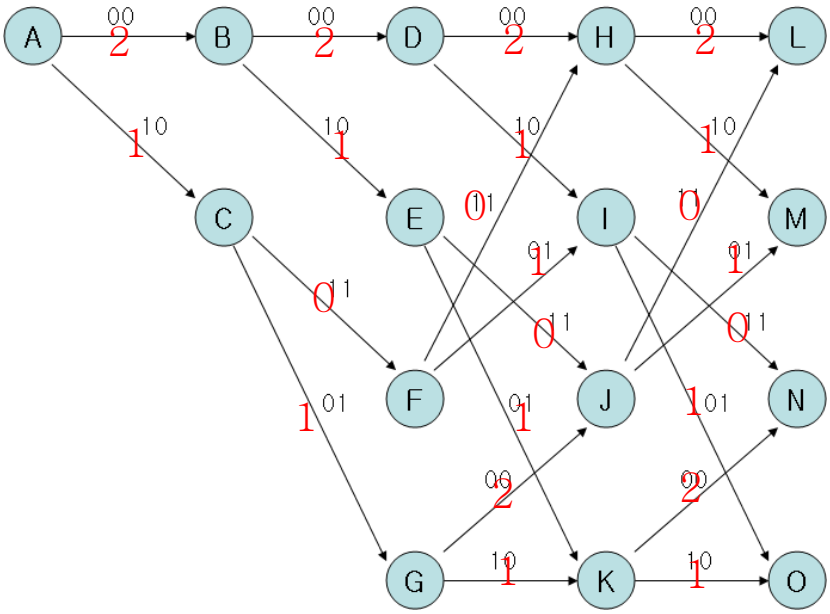
즉 4bit의 입력이 8bit의 출력으로 encoding 되는 것을 알 수 있다.
Convolutional code를 통한 에러 보정은 다음과 같은 공식을 만족시킨다.

$d_{free}-1$ 개의 에러 감지 가능
$\mathrm{mod}(d_{free}-1)/2$ 개의 에러 보정 가능

여기서 d_{free}를 정의하기 위해 무게라는 개념을 도입해야 하는데, Trellis diagram에서의 무게는 각 간선을 통해 만나게 되는 출력 값들 중에서 1의 개수로 정의된다. 예를 들어 <그림 1.6>의 Trellis diagram의 경우 경로 ACFIO를 고려했을 때의 무게는 5가 된다. d_{free}는 Trellis diagram에서 가질 수 있는 모든 경로에서 가장 무게값이 낮은 수를 의미한다. <그림 1.6>에서의 d_{free}는 경로 ACGJL에서 나오는 무게 4가 된다.

이렇게 encoding된 Convolutional code는 마찬가지로 Trellis diagram을 참조하여 쉽게 decoding 가능하다. 예를 들어 00100110을 decoding한 결과는 경로 ABEKO를 참조하여 나타나는 0111가 된다. 만일 데이터 전송 과정에서 에러가 발생한 경우, 예를 들어 encoding된 데이터가 11111110인 경우 Trellis diagram을 참조하면 첫 2 bit부터 맞는 경로가 존재하지 않으므로 데이터에 에러가 생긴 것을 쉽게 알 수 있다. 이러한 에러를 보정하기 위한 방법으로 Viterbi algorithm이나 Turbo code 같은 방법이 존재한다.

Viterbi algorithm은 각 간선의 출력값을 가중치로 환산하여 가장 적은 가중치를 갖는 경로를 취하고 그 경로와 encoding된 데이터를 비교하여 에러를 보정하는 방법이다. <그림 1.6>과 같은 Trellis diagram은 <그림 1.7>과 같은 가중치를 갖게 된다.



<그림 1.7 가중치 환산된 Trellis diagram>

이렇게 가중치로 환산된 Trellis diagram을 참조하여 11111110 데이터의 에러를 보정하여 보자. 먼저 최소 가중치를 갖는 경로를 찾아보면 ACFHM이 되고 이때의 출력 데이터를 살펴보면 10111110인데 이를 입력받은 데이터와 비교하면 두 번째 bit에서 에러가 생겼음을 확인할 수 있다. 따라서 Viterbi algorithm에 의해 11111110을 10111110으로 보정하게 된다.