Generación de Numeros Aleatorios

Adolfo Tamayo Briceño
Jose Manuel Valdivia Romero

February 8, 2014

Abstract

La generación de numeros aleatorios por computadora es de vital importancia para muchas ramas del conocimiento, como por ejemplo, la simulacion de gases, fluidos y por supuesto en la criptografia. Esto ha ocasionado el desarrollo de varias tecnicas para generarlos. A continuacion analizamos algunos algoritmos que consideramos son muy buenos por varios motivos.

Contents

1 Marco Teorico

1.1 Numeros Pseudo-aleatorios

En realidad, las computadoras no pueden generar numeros verdaderament aleatorios. Lo que se hace es una funcion, que generalmente es de la forma:

$$X_i = (a_0 X_{i-1} + b)\%m$$

Ya que estamos usando el modulo, eventualmente se van a generar patrones, o periodos, en los que se va a empezar a repetir la sequencia. Ademas si el valor de X_0 es siempre el mismo, la secuencia también sera siempre igual.

1.2 Semillas

La semilla es el estado inicial del generador de numeros, es el X_0 que permite que la sequencia generada por la funcion sea distinta cada vez que se genere. Generalmente se utiliza el numero de milisegundos desde Enero de 1970 (Unix epoch), o tambien se puede utilizar una semilla generada por ruido ambiental. Esto puede ser cualquier cosa desde el clima, el sonido, los electrones producidos por algun material radioctivo, etc.

1.3 Propiedades

Generalmente se quiere que los numeros cumplan con ciertas propiedades para considerarlos "aleatorios".

- Deben ser estadisticamente aleatorios, lo que quiere decir que su distribucion es uniforme (que todos los elementos del conjunto tienen la misma probabilidad de aparecer).
- Impredecibles, es decir que no se pueda predecir el siguiente numero de la secuencia luego de observar anteriores numeros.
- Deben tener periodos largos, ya que como estamos utilizando el modulo, los numeros eventualmente se van a repetir y se prefiere que este periodo este en el orden de numeros como 2^{64} aunque hay algoritmos que permiten periodos mucho mayores
- Eficiencia: Se desea que el generador sea eficiente, para disminuir la carga en algoritmos que podrian ser paralelos.

1.4 Pruebas

Existen conjuntos de pruebas estadisticas que se pueden aplicar a los generadores para probar que tan uniforme es su distribucion. Estas pruebas generalmente son hechas por academicos para probar los algoritmos que desarrollan. Estas pruebas pueden ser cualitativas o cuantitavas. Algunos ejemplos:

- Cuantitativas:

- Pruebas X^2: Dividen el intervalo de numeros y calculan el esperado de la distribucion y lo comparan con el generado para ver si esta bien distribuido.
- Lagged Correlation: Estas pruebas comprueban si es que existe una relacion obvia entre dos numeros de la secuencia.

- Cualitativas:
  - Analisis Visual de la Muestra (Scatter Plots): Se genera un grafico con la sequencia generada y se busca patrones en la imagen, lo que indicaria una distribucion pobre. Ejemplo:

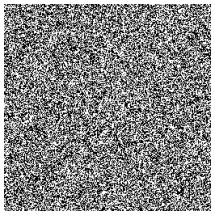

Figure 1: Random.org

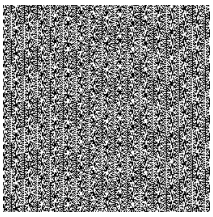

Figure 2: PHP rand() en Windows

  - Pruebas Diehard : Es una bateria de pruebas estadisticas que prueban la calidad de los numeros generados.

2 Algoritmos

2.1 XOR-shift

2.1.1 Descripcion

Es un algoritmo muy rápido ya que se basa en hacer la secuencia (vista en el marco teórico) haciendo corrimientos de bits y funciones XOR ("o" exclusivo), estas funciones a nivel de procesador son muy rápidas y no gastan mucho espacio. Para la semilla podemos usar cualquier numero y le hacemos luego los corrimientos de bits y funciones XOR necesarias para crear el patron

2.1.2 Implementacion

Para el XOR-Shift necesitamos cuatro valores x, y, z, w (definidos como "xorhift_x, xorshif_y, xorshift_z,xorshift_w" en el código).
En nuestra implementación como semilla usamos al tiempo con la libreria time.h, con ella sacamos segundos desde enero 1 del 2000 hasta la fecha actual.
x será el tiempo actual, y, z, w seran corrimientos de bits o funciones xor de este tiempo, aqui el seed (notese que la funcion XOR esta denotada por el operador ^ y el corrimiento de bits por el operador <<)

```
1 void SeedXorShift()
2 {
3     time_t timer= time(NULL);
4     struct tm y2k;
5     ZZ seconds;
```

```
6       y2k.tm_hour = 0;
7       y2k.tm_min = 0;
8       y2k.tm_sec = 0;
9       y2k.tm_year = 100;
10      y2k.tm_mon = 0;
11      y2k.tm_mday = 1;
12      seconds= conv<ZZ> (difftime(timer,mktime(&y2k)));
13      xorshift_x=seconds;
14      xorshift_y=seconds^123456789;
15      xorshift_z=seconds<<31;
16      xorshift_w=seconds>>28321;
17
18  }
```

una vez obtenido el seed para el XOR-shift se procede a hacer los corrimientos de bits y las funciones XOR de la siguiente forma: Obtenemos una variable temporal *tmp* que sera un XOR entre x y el corrimiento con 11, luego $x = y \quad y = z \quad z = w$, al final actualizamos w que sera $wXOR(w >> 19)XOR(tmpXOR(tmp >> 8))$ y retornamos el modulo de w con un numero MAX_XOR, todos los numeros generados seran menores a este. El codigo es el siguiente:

```
1   ZZ XorShift()
2   {
3
4       ZZ tmp;
5       tmp=xorshift_x^(xorshift_x<<11);
6       xorshift_x=xorshift_y;
7       xorshift_y=xorshift_z;
8       xorshift_z=xorshift_w;
9       xorshift_w=xorshift_w^(xorshift_w>>19)^(tmp^(tmp>>8));
10      return Modulo(xorshift_w,conv<ZZ>(MAX_XOR));
11
12  }
```

La forma de usarlo será la siguiente

```
1   SeedXorShift();
2   for (int i=0;i<10;i++)
3   {
4     cout<<XorShift()<<endl;
5   }
```

Llamamos al Seed primero y luego a la misma funcion XorShift para que nos genere los numeros aleatorios, esta nos generara diez numeros aleatorios.

2.1.3 Pruebas XOR-Shift

El XOR-Shift es tan rapido que uno solo no puede ser medido ya que los Corrimientos de bits y las funciones XOR a nivel de bits son demasiado rapidas para el procesador y no requieren de mucho, pero para sacar 25 numeros se demoro 0.000270669 segundos, lo malo de este algoritmo que es facilmente predictible porque solo tenemos que hayar el tiempo para poder saber el seed y simplemente seguimos generando numeros y sabremos perfectamente cual es la secuencia de este algoritmo pudiendo predecir cual sera el siguiente.

2.2 Blum Blum Shub

2.2.1 Descripcion

Es un algoritmo un poco mas lento que el XOR-Shift u otros ya que este usa exponenciaciones modulares pero este es mas seguro, fue creado por Lenore Blum, Manuel Blum y Michael Shub, este algoritmo genera bits con la formula $x_i = x_{i-1}^2 mod N$ sacando el bit menos significativo del x_i. Este algoritmo es mas seguro ya que la variable N es grande y dificil de factorizar ya que se haya de la misma manera que en RSA (obtenemos dos primos grandes P y Q, los multiplicamos y asi hayamos N)

2.2.2 Implementacion

Para hallar el x_0 necesitamos hallar el P y Q, para esto usamos nuestro XOR-Shift y el test de primalidad de Miller-Rabin, El XOR-Shift genera Numeros y se prueban en el Test de Miller-Rabin,hallamos $N = P * Q$, luego hallamos un numero aleatorio con el XOR-Shift para que sea el Seed S, entonces tenemos que $x_0 = s^2 mod N$
Esta es la implementacion:

```
1  void SeedBBS()
2  {
3    ZZ p =GeneratePrimeXOR2();
4    ZZ q = GeneratePrimeXOR2();
5    //generamos los Primos P y Q
6    n_bbs =p*q;
7    ZZ seed= XorShift();
8    xi_bbs=ModularExponentiation(seed,conv<ZZ>(2),n_bbs);
9    //Hallamos el Xo
10 }
```

Una vez obtenido el x_0 podemos tener los demas con la formula ya expuesta ($x_i = x_{i-1}^2 mod N$), luego de hallar esto sacamos el bit menos significativo de x_i y lo almacenamos en un vector de bits de el tamaño ya establecido por la cantidad de bits que queremos sacar. Para esto usamos una estructura de datos llamada BitVector que almacenara los bits que pongamos y la funcion "SetBit" que pondra el bit menos significativo en la posicion que le indiquemos, luego de tener el BitVector lleno lo volvemos un numero con la funcion de la libreria NTL "ZZFromBytes".

```
1  ZZ BlumBlumShub(int tam)
2  //tam es la cantidad de bits que tendra el numero aleatorio
3  {
4    ZZ res;
5    BitVector bits(tam);
6    for(int i=tam-1;i>=0;i--)
7    {
8      xi_bbs=ModularExponentiation(xi_bbs,conv<ZZ>(2),n_bbs);
9      bits.SetBit(i,(xi_bbs%2)==conv<ZZ>(1));
10   }
11   res=ZZFromBytes(bits.vector,tam/8);
12
13   return res;
14 }
```

la forma de utilizar el Blum Blum Shub es sencilla:

```
1 SeedXorShift();
2 SeedBBS();
3 int tam =64;
4 for (int i=0;i<10;i++)
5 {
6   cout<<BlumBlumShub(64)<<endl;
7 }
```

Como dijimos, usamos el XOR-Shift para hallar el x_0 del Blum Blum Shub, para eso usamos su seed, y luego llamamos a SeedBBS(), de ahi a la misma funcion BlumBlumShub(tam)donde tam es el tamaño de la cadena de bits que se generara.

2.2.3 Pruebas Blum Blum Shub

Le hicimos pruebas a Blum Blum Shub ya que se afirma que es un algoritmo que no genera numeros, sino genera secuencias de bits aleatorias, permitiendo generar numeros de los bits que queramos, para estas pruebas hemos generado cien numeros aleatorios de la cantidad de bits especificada, aqui los tiempos:

64 bits: 0.000550542 segundos.

128 bits: 0.001876300 segundos.

256 bits: 0.003737900 segundos.

512 bits: 0.005616870 segundos.

1024 bits: 0.006317080 segundos.

2048 bits: 0.012843500 segundos.

4096 bits: 0.025477500 segundos.

8192 bits: 0.0508103 segundos

a pesar de ser mucho mas lento en generar un numero el Blum Blum Shub es muy seguro y casi impredictible de saber ya que primero tenemos que genera bits no numeros, luego el x_0 se haya con un seed de un numero aleatorio que no sabremos cual es y habria que hacer una investigacion sumamete profunda para saber exactamente cual fue.

2.3 Mersenne Twister

2.3.1 Descripcion

El Mersenne Twister es el algoritmo de generacion de numeros pseudo-aleatorios mas utilizado. Su nombre proviene del hecho que su periodo es un primo Mersenne (primos de la forma $M_p = 2^p - 1$).
Fue desarrollado en 1997 por Makoto Matsumoto y Takuji Nishimura
Es el generador de aleatorios utilizado por varios lenguajes de programacion incluyendo R, Python, Ruby, Pascal, Matlab, GNU, y C++ desde su version C++11.

2.3.2 Implementacion

El Mersenne twister consta de tres partes que nos permiten generar los numeros aleatorios. Primero declaramos un vector estatico de 624 numeros, en este caso variables ZZ de la libreria NTL para trabajar numeros grandes, y un indice que nos va a permitir navegar el vector. Ademas tenemos la inicializacion del vector a partir de una semilla.

```
1  /*--------------------Mersenne Twister----------------------*/
2  // Declaraciones
3  ZZ mersenne_state[624];
4  int m_index;
5  //Inicializador del vector
6  void MT_init(ZZ seed){
7      m_index = 0;
8      mersenne_state[0] = seed*seed;
9      for (int i = 1; i < 624; ++i){
10         ZZ x;
11         x = mersenne_state[i-1]<<30;
12         x = mod(x,conv<ZZ>(MAX_XOR));
13         ZZ tmp = mersenne_state[i-1];
14         ZZ ans;
15         ans = x^tmp;
16         string mult = "18124332531873678163";
17         ans *= conv<ZZ>(str_to_ZZ( mult));
18         mersenne_state[i] = abs(ans);
19
20     }
21
22 }
```

El vector se inicializa con el valor de la semilla al cuadrado y el indice con 0. Luego llenamos el vector en las posiciones 1 a la 623 haciendo una serie de shifts, xor y multiplicaciones.
Luego vamos a necesitar una funcion que regenere el vector cuando este se quede sin numeros aleatorios nuevos.

```
1  //Regenera el vector
2  void mt_regenerate() {
3      for (int i = 0; i < 624; ++i)
4      {
5          ZZ y = (mersenne_state[i] & 0x80000000)+(mersenne_state[(i
               +1)%624] & 0x7fffffff) ;
6          mersenne_state[i] = mersenne_state[(i+397)%624]^(y>>1);
7           if( mod(y,conv<ZZ>(2)) != 0 ){
8               mersenne_state[i] = mersenne_state[i]^(2567483615);
9           }
10     }
11  }
```

Esta funcion se va a llamar una vez cada 624 pedidas de numeros como vamos a ver a continuacion en la funcion que es la que usamos para pedir numeros.

```
1  //Retorna un numero aleatorio, regenera el vector cada 624 llamadas
2  ZZ mt_rand() {
3       if (m_index == 0){
4           mt_regenerate();
5       }
6
7       ZZ y = mersenne_state[m_index];
8       y = y^(y>>11);
```

```
9         y = y^(y<<7 & (conv<ZZ>(2636928640)));
10        y = y^(y<<15 & (conv<ZZ>(4022730752)));
11        y = y^(y>>18);
12
13        m_index++;
14        m_index = m_index%624;
15        return y;
16    }
```

En esta funcion igualmente hacemos uso del XOR binario y los corrimientos para que los numeros sean aun mas distintos.
La forma de utilizar el Mersenne Twister que hemos implementado es la siguiente:

```
1 int main(){
2       MT_init(conv<ZZ>(time(NULL)));
3       for(int i=0; i<10 ; i++){
4         cout<<mt_rand()<<endl;
5       }
6 }
```

Este codigo imprimira 10 numeros aleatorios y esta utilizando el tiempo del sistema como semilla.

2.3.3 Pruebas Mersenne Twister

El Mersenne Twister genera rapidamente numeros al igual que el XOR-Shift y generando 25 numeros se demora 0.000712404 segundos, sin embargo este no genera numeros muy grandes ya que no podemos controlar bien eso ya que este no genera bits sino numeros

2.4 Dual Elliptic Curve (Dual EC DRBG)

2.4.1 Descripcion

El Dual EC DRBG, es el algoritmo estandard presentado por la NIST(National Institute of Standard Technology) para la generacion de bits aleatorios.
Este algoritmo requiere unos ciertos valores especificados, los cuales son la curva eliptica, y los dos puntos P y Q. La NSA requeria que todos los que querian utilizar este algoritmo de forma aprobada utilizen los siguientes valores.

- La curva $y^2 = x^3 - 3x + b(modp)$
- Ademas debias utilizar ciertos valores de $P, Q, P_x, P_y, Q_x, Q_y, p, nyb$

Esta basado en el problema del logaritmo discreto de curvas elipticas donde dados unos puntos P y Q en una curva eliptica en un orden de n encontresmos un a que cumpla que $Q = aP$

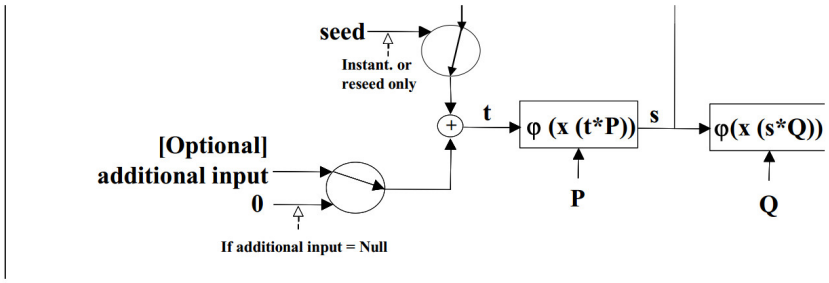

Figure 3: Diagrama de Flujo de Dual EC DRBG .

3 Conclusiones

Rapidez Vs. Seguridad: A pesar que algunos algoritmos demoren muy poco como el XOR-Shift o el Mersenne Twister aveces es mejor (en especial en la criptografia) tener algoritmos mas seguros como el Blum Blum Shub o el Dual EC DRGB que permiten generar numeros mas grandes.

Mersenne Twister Casi Perfecto: Decimos que es asi porque el periodo de este es de $2^{19937} - 1$ haciendo que sea improbable llegar a la repeticion, lo unico malo es que genera solo numeros de 32 bits.

References

[1] *Handbook of Applied Cryptography.* Book by Alfred Menezes, Scott Vanstone, Paul van Oorschot, and A. J. Menezes

[2] *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random generator by Makoto Matsumoto and Takuji Nishimura.* http://www.math.sci.hiroshima-u.ac.jp/ 20m-mat/MT/ARTICLES/mt.pdf

[3] *Randomness.* www.random.org/randomness

[4] *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf