Generación de Cuadrados Latinos de Orden 256 Utilizando un Grafo de Reemplazos

Ignacio Gallego Sagastume
Facultad de Informática
Universidad Nacional de La Plata
ignaciogallego@gmail.com

Abstract

Los cuadrados Latinos (LSs) son estructuras algebraicas con aplicaciones en criptografía. Si los LSs son aleatorios y uniformemente distribuidos, pueden ser usados como claves para algoritmos de encriptación simétricos. En el contexto de un protocolo de comunicación seguro, debe generarse un nuevo LS cada cierta cantidad de tiempo o cantidad de datos transmitida para no correr el riesgo de que un atacante lo deduzca y pueda así descifrar los mensajes transmitidos. El tiempo y recursos requeridos para generar un nuevo LS no deben implicar una gran sobrecarga en la comunicación.

En este trabajo, se propone un algoritmo para generar LSs aleatorios de cualquier orden en tiempo polinomial (menor al tiempo del algoritmo de Jacobson y Matthews). El mismo utiliza un grafo de reemplazos posibles en cada posición para corregir las posibles repeticiones de elementos durante la generación.

Se presenta el pseudocódigo del algoritmo y se hace un análisis de la uniformidad de los resultados.

Palabras Clave
Cuadrados latinos, generación, aleatorios, distribución uniforme

1. Introducción

El marco teórico o modelo de básico de comunicación supone dos entidades: "Alice" y "Bob" que intercambian mensajes en un canal inseguro [2]. Este canal es pasible de ser leído o escrito por un atacante "Eve" (por "Eavesdropper" o escucha secreta).

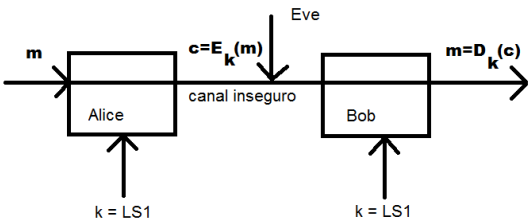

Figura 1: Modelo Básico de Comunicación

Un algoritmo de cifrado de datos simétrico utiliza una clave (privada) ***k*** para transformar una cadena de caracteres (texto plano ***m***) en un texto cifrado $c=E_k(m)$. Usando la clave de cifrado y conociendo el algoritmo de descifrado (***D*** en la **figura 1**) puede accederse al texto plano ***m*** en el otro extremo de la comunicación, haciendo $m=D_k(c)$. Si Eve interceptara el canal de comunicación y no conociera la clave ***k***, no podría en principio acceder al texto plano ***m***.

1.1 Definición de Cuadrado Latino y Propiedades

Un cuadrado Latino (LS) de orden *n* es una matriz de $n \times n$ símbolos, en donde cada símbolo aparece exactamente una vez en cada fila y una vez en cada columna. Por ejemplo:

1 3 2
2 1 3
3 2 1

es un LS de orden 3.

Los LSs pueden utilizarse como claves de algoritmos de cifrado simétricos, en la manera indicada en [5]. Para esto, deben cumplir con algunas propiedades como por ejemplo ser "shapeless" o sin forma; esto es, ser aleatorios y uniformemente distribuidos. Debe ser tan probable un LS como otro cualquiera de todos los posibles.

El conjunto de todos los LSs de orden *n*, *L(n)*, es tan grande que en la actualidad solo se conoce hasta *L(11)*. A medida que crece *n*, *L(n)* crece exponencialmente. Para órdenes mayores a 11 sólo se conocen cotas superiores e inferiores como por ejemplo las siguientes [1]:

$$\frac{n!^{2n}}{n^{n^2}} \leq L(n) \leq \prod_{k=1}^{n} k!^{\frac{n}{k}}$$

Por ejemplo, esta cota inferior de *L(n)* para $n = 256$ es $\cong 3{,}047 \times 10^{101723}$. Esta característica, hace que sea imposible (con el poder computacional actual) realizar un ataque por fuerza bruta para adivinar un LS. Sin embargo, si se reutiliza mucho tiempo el mismo LS para cifrar información, el mismo puede irse deduciendo, por lo que es necesario ir generando nuevos LSs cada cierta cantidad de tiempo o datos transmitidos.

Un LS de orden 256 puede almacenar todos los caracteres de la tabla ASCII y así ser utilizado para cifrar cualquier texto plano. Por lo tanto, el caso objetivo serán los LSs de orden 256. El problema principal es entonces generar *eficientemente* LSs de orden 256 que sean aleatorios y de distribución lo más uniforme posible.

2. ¿Cómo se Usa un LS para Encriptar?

La presente sección se ha incluido para que el trabajo sea auto-contenido; ya había aparecido en trabajos anteriores, por lo que si se desea se puede seguir con la sección 3.

Supóngase (para simplificar) que se tiene el LS de orden 4 de la **figura 2** y se quiere utilizarlo para encriptar la palabra "abcc".

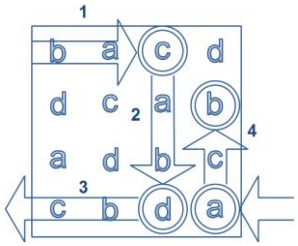

Figura 2: Cifrado de la Cadena "abcc" a la Manera de Gibson

Se puede recorrer el LS, a la manera de Gibson [5], alternando movimientos hacia derecha, abajo, izquierda y arriba. Cuando se termina de recorrer en una dirección y se alcanza el borde del LS, se ingresa cíclicamente por el otro lado.

1. Primer paso: se busca la primera letra a encriptar, la "a", comenzando por la primera fila de izquierda a derecha hasta encontrar esa letra. Se toma la letra siguiente en el LS, la letra "c".

2. Segundo paso: Ahora se cambia la dirección y se va hacia abajo. Se busca la segunda letra de la palabra a encriptar, la "b" y se toma la letra siguiente, la "d". Se tiene hasta ahora el texto cifrado "cd".

3. Tercer paso: Se cambia nuevamente la dirección para ir hacia la izquierda. Se encuentra la letra "c" y se toma la siguiente en la misma dirección, ingresando al LS cíclicamente por el otro lado. Se toma la siguiente letra del LS, la "a". Se tiene hasta ahora "cda".

4. Cuarto paso: Se cambia otra vez la dirección y se va hacia arriba. Se busca la última letra del texto plano, la otra "c", y se toma la letra siguiente, la "b". El texto cifrado correspondiente al texto plano "abcc", resulta entonces "cdab".

Notar que este proceso es reversible, y que recorriendo en el orden inverso (sabiendo la última posición utilizada), se puede volver a obtener el texto plano "abcc". Otras variantes de recorrido podrían ser alternar entre ir hacia izquierda, abajo, derecha y arriba, o recorrer en una dirección y rebotar contra los bordes del LS (en vez de reingresar cíclicamente). También se puede utilizar el mismo esquema de recorrido pero tomando los dos caracteres siguientes en vez de uno. En este caso, el texto cifrado tendría el doble de longitud del texto plano, pero se estaría revelando más información del LS y se debería generar más frecuentemente. Generar claves aleatorias de esta manera parece razonable: se aprovecha el hecho de que el LS es aleatorio (dos símbolos contiguos no respetan ningún orden) y se toman los símbolos según algún recorrido secuencial. El problema es que el LS usado para encriptar debe ser lo más aleatorio posible (sin tendencias o valores similares en celdas adyacentes) y debe generarse uno cada cierto tiempo para que no pueda ser deducido por un atacante, asumiendo que éste conozca el esquema de recorrido que se utiliza.

3. El Método Propuesto

El método se basa en una generación secuencial por filas. En cada paso se extrae un número aleatorio de los disponibles en fila y columna actuales y en caso de no ser posible esto (porque no hay elementos disponibles) se procede a realizar una cadena de reemplazos sobre los elementos ya colocados en la fila que tendrá como objetivo hacer que haya al menos un elemento disponible para continuar la generación.

Ejemplo:

```
          disp. col 5=[1,2,3,5]
3 1 5 2 0 4
2 3 4 1 5 0
4 5 2 3 1       disp. fila actual=[0]
```

En este punto, se ha generado la última fila parcialmente pero entre los elementos disponibles en la columna actual [1, 2, 3, 5] no hay ninguno que esté también disponible en la fila actual [0] (intersección vacía). A la situación cuando dos elementos se repiten o no hay más opciones para la generación se la llamará "colisión". En este punto se construye un grafo de reemplazos posibles para los elementos ya colocados en la fila actual y también para la columna actual:

```
columna j | posibles reemplazos:
    0   --> [0,1,5]
    1   --> [0,2,4]
    2   --> [0,1,3]
    3   --> [0,4,5]
    4   --> [2,3,4]
    5   --> [1,2,3,5]
```

Este grafo se lee como sigue: Al elemento de la posición 0 de la fila actual se lo podría reemplazar por los elementos [0, 1, 5] sin causar una nueva colisión con filas anteriores. Al elemento de la posición 1 por los elementos [0, 2, 4], y así siguiendo.

Se procede a tratar de liberar uno de los elementos posibles de la fila 5; se sortea al azar el elemento 2 (por ejemplo). El primer reemplazo es el elemento 2 que está en la posición 2, por un elemento al azar entre los disponibles en esa posición. Por ejemplo, el elemento 1. El cuadrado parcial quedaría ahora:

```
3 1 5 2 0 4
2 3 4 1 5 0
4 5 1 3 1       disp. fila=[0,2]
```

Como se puede apreciar, no se generan colisiones con

filas previas, pero sí puede ocurrir que se genere una colisión en la fila actual (sino, el proceso estaría terminado y el elemento 2 estaría ahora disponible para usar en la última columna). El segundo reemplazo trata de eliminar la última colisión generada en la fila actual. Se toma el 1 que estaba anteriormente en la fila y se sortea por qué elemento debe ser reemplazado entre los disponibles de la posición 4, que son [2, 3, 4]. Digamos que se obtiene al azar el elemento 3. Entonces se realiza el reemplazo del 1 por el 3 y queda:

```
3 1 5 2 0 4
2 3 4 1 5 0
4 5 1 3 3       disp. fila=[0,2]
```

Se genera una nueva colisión, se procede igual que el caso anterior, reemplazando el elemento 3 que ya se encontraba en la fila (no el último colocado) por alguno de los disponibles [0, 4, 5]; digamos que se sortea al azar el elemento 0. En este reemplazo, del elemento 3 por el 0, se eliminan todas las colisiones y elemento que se deseaba extraer de la fila (el 2), ya se encuentra disponible para continuar con la generación.

```
            disp. col 5=[1,2,3,5]
3 1 5 2 0 4
2 3 4 1 5 0
4 5 1 0 3       disp. fila=[2]
```

Se continúa entonces, poniendo el elemento disponible 2:

```
3 1 5 2 0 4
2 3 4 1 5 0
4 5 1 0 3 2
```

para terminar con la generación de la fila.

4. El Algoritmo

A continuación se lista el pseudo-código del algoritmo implementado en Java. Cuando se produce una colisión, se procede a construir el grafo de reemplazos para el caso actual y luego a hacer un nuevo elemento disponible (método **`makeElemAvailable`**):

```
public HashMap<Integer, HashSet<Integer>> constructReplGraph(ArrayList<Integer> row,
                                                              Integer col,
                                                              HashSet<Integer>[] initialAvailInCol,
                                                              HashSet<Integer>[] availInCol) {

  HashMap<Integer, HashSet<Integer>> map = new HashMap<Integer, HashSet<Integer>>();
  for(int j=col; j>=0; j--) {
    HashSet<Integer> set = new HashSet<Integer>();
    set.addAll(initialAvailInCol[j]);
    if (set.size()>0)
      map.put(j, set);//el elem. en posicion j podría ser cambiado por cualquiera del conjunto set
  }
  return map;
}
```

Figura 3: Método para Construir el Grafo de Reemplazos

Como se puede observar, la construcción del grafo de reemplazos es muy simple, y se basa en el conjunto de números disponibles originalmente en la fila al comenzar su generación. Esta estructura de datos está disponible ya que se necesita para ir colocando los números en el LS.

```
public void makeElemAvailable(elem, ...) {
  while (!finished) {
    avail = Todos los posibles reempl. de elem que no estén en PATH
    if (avail.isEmpty()) {
      //creo un nuevo camino PATH
      //Agrego a avail todos los posibles remplazos de índice idx_old
    }
    int newElem = RandomUtils.randomChoice(avail);
    idx_new = row.indexOf(newElem);
    //Realizo el reemplazo
    row.set(idx_old, newElem);
    path.add(newElem);

    //Actualizo los conjuntos de disponibles en fila actual (availableInRow) y columnas (availInCol)
    //Verifico la condición de fin
    finished = (availableInRow.contains(elem) && idx_new==-1);
    idx_old = idx_new;
    old = newElem;
  }
```

Figura 4: Método que Realiza la Cadena de Reemplazos

```
}
```

En el método de la **figura 4**, se extrae un elemento *elem* de la fila actual para dejarlo disponible para continuar con la generación. Esto siempre es posible, ya que para cada posición en la fila actual existen varias opciones entre los elementos disponibles para esa posición.

El método se lee como sigue: mientras haya colisiones o mientras el elemento *elem* no esté disponible, se itera. Luego se toman todos los posibles reemplazos de *elem* que no estén en el camino recorrido *path*. Se llama *avail* a ese conjunto. Si este conjunto es vacío, se borra el camino actual y se vuelve a empezar. De lo contrario, se toma un elemento de *avail* aleatoriamente, y se lo llama *newElem*. Luego se realiza el reemplazo del elemento viejo por el nuevo y se agrega el nuevo elemento al camino *path*. La condición final, examina si se liberó el elemento *elem*, y si todavía hay colisiones en la fila actual.

5. Tiempo de ejecución

El algoritmo expuesto en la sección anterior se ejecuta en tiempo polinomial de $O(n^3)$, ya que toda la cadena de reemplazos termina a lo sumo en n pasos y hacen falta n^2 pasos para ubicar todos los elementos. La cantidad de colisiones es muy baja para las filas, ya que siempre que se llega a una se repara en n pasos y se puede continuar con la generación.

En comparación con la implementación del algoritmo de Jacobson y Matthews dada en [3,4], el algoritmo de grafo de reemplazos es aproximadamente 5 veces más rápido.

6. Tests de Uniformidad

El problema de la generación con grafo de reemplazos es que no se generan cuadrados uniformemente de todos los posibles, sino que hay algunos más probables que otros. Sin embargo, a los efectos prácticos, puede usarse este método sin temor a repetir LSs dado que el espacio muestral es gigantesco. Además, todos los posibles cuadrados son alcanzables por este método.

6.1 El test de χ^2

El test de χ^2 mide la bondad de ajuste de una distribución con respecto a otra teórica esperada. En este caso, queremos comprobar si la distribución es uniforme.

El procedimiento es como sigue: lo que se hace primero es enumerar (identificar con una etiqueta numérica) todos los LSs posibles del orden n en cuestión. Esto es factible únicamente para $n \leq 7$, pues para órdenes mayores no alcanza con el tipo de datos Long o BigDecimal de Java para representar el valor $L(n)$. Luego, se genera un número muy grande de LSs, como por ejemplo 180 millones (o más). Para cada cuadrado generado, se suma 1 al valor correspondiente para la etiqueta de ese cuadrado. De esta manera se obtiene un vector de cantidades como el siguiente:

cant[ID] = cantidad de LSs generados con ese ID

Un ejemplo para el caso de n=5 (con $L(n)$=161280):

cant[1] = 1.330
cant[2] = 1539
…
cant[161280] = 799

El valor esperado si la distribución fuera realmente uniforme, sería 180.000.000/161280 = 1116,071. Luego, para sintetizar el arreglo *cant*, se genera otro arreglo *observada[]* de dimensión k=10 (por ejemplo), donde k son los grados de libertad con los que se desea trabajar (número de grupos que pueden variar de 10 a 100 según las tablas existentes). Se dividen los intervalos de índices del arreglo *cant* y se suman por grupos, obteniendo:

observada[1] = ∑ (j=1..16128) cant[j]
observada[2] = ∑ (j=16129..32256) cant[j]
…
observada[10] = ∑ (j=145152..161280) cant[j]

Se genera también un arreglo de dimensión k identificado por *esperada[]*, que contiene k valores *cant. de experimentos/k=180.000.000/10=18.000.000*. Es decir que *esperada[i]=18.000.000* $\forall$ *i=1..10*. Este es el valor esperado de la sumatoria de cada grupo.

Entonces se calcula el estadístico de la siguiente manera:

$$\chi^2 = \sum_{i=0}^{k} \{(observada[i] - esperada[i])^2 / esperada[i]\}$$

Para terminar el test, este estadístico χ^2 se compara contra una tabla que dice con qué probabilidad, la muestra de valores observados tiene distribución uniforme.

6.2 Resultados

Se implementó el test descripto en la sección previa. En la siguiente tabla, se muestran algunas ejecuciones del test χ^2 comparadas con el algoritmo de J&M.

Algoritmo	orden	Generaciones	Tiempo	Chi	Max-Min reps	obtenidos
Repl Graph	4	10.000.001	3,166 minutos	341.691,695	53.538 y 9.095	576 sobre 576
Repl Graph	4	62.324.358	19,772 minutos	2.662.444,655	331.498 y 57.480	576 sobre 576
Repl Graph	5	1 millón	3,761 horas	138.733,234	78 y 1	156.538 sobre 161280
Repl Graph	5	1 millón	1,471 horas	140.429,662	78 y 1	156.293 sobre 161.280
Repl Graph	5	10 millones	13,644 horas	678.477,491	637 y 2	161.280 sobre 161.280
Repl Graph	5	50 millones	67,324 horas	3.106.029,432	3.033 y 34	161.280 sobre 161.280
Repl Graph	5	131.490.497	2,804 días	8.059.998,538	7.771 y 98	161.280 sobre 161.280
Repl Graph	30	2.780.330	41,170 horas	-	1 y 1	-
JacoMatt	4	10.000.001	7,224 minutos	1.711.370,968	169.606 y 1.441	576 sobre 576
JacoMatt	4	10.500.001	7,448 minutos	1.660.444,576	177.511 y 1.553	576 sobre 576
JacoMatt	4	20.000.001	14,457 minutos	3.277.473,010	337.417 y 2.931	576 sobre 576
JacoMatt	4	80 millones	2,017 horas	3.620.740,390	170.983 y 45.702	576 sobre 576
JacoMatt	5	1 millón	2,956 horas	1.015.978,235	2.635 y 1	138.015 sobre 161.280
JacoMatt	5	1 millón	6,048 horas	24.678,373	22 y 1	160.945 sobre 161.280
JacoMatt	5	4 millones	13,336 horas	3.519.671,941	10.505 y 1	159.432 sobre 161.280
JacoMatt	5	179.511.626	4,896 días	13.292,942	1.570 y 925	161.280 sobre 161.280

Figura 5: Algunas Ejecuciones del test χ^2 para los Métodos de J&M y del Grafo de Reemplazos

Lo que se hace es generar un número elevado de LSs utilizando el algoritmo de la primera columna para orden 4, 5 o 30. A cada cuadrado se le asigna una firma hash para identificarlo. Lo que se hace es contar cuantos cuadrados repetidos se generan de cada uno. Por ejemplo, para orden 30, el máximo y mínimo número de cuadrados es 1, indicando que nunca se repitió un cuadrado en las 2.780.330 generaciones. Para este caso no se pudo calcular el estadístico, ya que no se conoce *L(30)* (el número total de posibles cuadrados Latinos de orden 30). El estadístico sólo pudo calcularse para órdenes 4 y 5. Este hecho da un indicio de cuán improbable es repetir un LS de orden 256. Calcular la probabilidad de que esto ocurra se deja como trabajo futuro.

En este cuadro, puede observarse como a mayor cantidad de generaciones de cuadrados Latinos el método de Jacobson y Matthews menor es el estadístico χ^2, indicando que el método converge a una distribución uniforme, a diferencia del método de grafo de reemplazos que incrementa el estadístico a medida que se incrementan las generaciones. Además, la cantidad de repeticiones de un cuadrado determinado máxima y mínima están más distanciadas para el caso del método del grafo de reemplazos, indicando la no uniformidad del método.

7. Trabajos Relacionados

Un trabajo muy interesante es el de Fontana [6]. El autor hace incapié en generar LSs que son realmente (y no aproximadamente) uniformemente distribuidos. Para esto construye un grafo sobre todas las permutaciones de *n* elementos y sobre ese grafo encuentra todos los cliquees de tamaño máximo y extrayendo uno construye un LS. Se pudo generar con este método, en una máquina con procesador Intel i7 LSs realmente aleatorios de hasta orden 7; se examinaron 17 millones de cliques. Es decir, se extrajo un LS al azar de entre los 61.479.419.904.000 posibles de orden 7 equiprobablemente.

Otro trabajo interesante es el de Bartak [7]: el mismo propone generadores para problemas QCP (Quasigroup completion problem) y QWH (Quasigroup with holes). En realidad este trabajo utiliza un método de J&M modificado para generar estas instancias de problemas.

Los trabajos de McKay [8] y Koscielny [9] proponen distintos métodos teóricos para generar LSs. Los métodos parecen novedosos y prometedores; sin embargo, no pudieron utilizarse los conceptos en ellos utilizados por ser los mismos muy complejos y donde es necesario un estudio muy detallado y específico que escapa a los objetivos del presente trabajo.

Un trabajo más accesible es el de Selvi (et. al.) [10], que propone una generación por filas similar a la del grafo de reemplazos (que supuestamente soluciona un algoritmo con errores de O'Carroll de 1963), pero con un tratamiento de colisiones distinto. Se propone examinar este trabajo en versiones futuras del algoritmo.

8. Conclusiones

Se ha presentado un algoritmo eficiente, que permite generar cuadrados Latinos en tiempo polinomial (y más bajo que J&M [1], el método más conocido y aceptado). El método no es uniforme, pero la probabilidad de repetir LSs de orden 256 es insignificante. El método puede usarse en la práctica para generar cuadrados y usarlos para encriptar y cada cierto tiempo repetir la generación sin que esto implique una sobrecarga en la comunicación.

9. Trabajo futuro

Se deja como trabajo futuro examinar el sesgo del método aleatorio en más profundidad; esto implicaría calcular la probabilidad de repetir un LS de orden 256 con el método propuesto y/o analizar cuántas generaciones hacen falta en promedio para repetir algún LS de todos los posibles.

Referencias

[1] Mark T. Jacobson and Peter Matthews, Generating uniformly distributed random Latin squares, J. Combin. Des. 4 (1996), no. 6, 405-437. MR MR1410617 (98b:05021)

[2] R. Dahab, J. C. López-Hernández. Técnicas Criptográficas Modernas - Algoritmos e Protocolos. Em: Tomasz Kowaltowski; Karin Breitman. (Org.). Atualizações em Informática 2007. Rio de Janeiro: Editora PUC-Rio, 2007, v. , p. 115-170.

[3] Ignacio Gallego Sagastume, Generation of Latin squares step by step and graphically, Congreso Nacional de Ciencias de la Computación (CACIC) 2014. Universidad de La Matanza, Buenos Aires, Argentina. (2014).

[4] Ignacio Gallego Sagastume, Proyecto igs-lsgp (latin square generation package) en github, https://github.com/bluemontag/igs-lsgp/wiki, 2015.

[5] Steve Gibson. Off the grid (online). https://www.grc.com/offthegrid.htm, Mayo de 2012.

[6] Fontana R.: Random Latin squares and Sudoku designs generation. ArXiv e-prints, 2013.

[7] Barták R.: On Generators of Random Quasigroup Problems, 2004.

[8] Brendan D. McKay, Nicholas C. Wormald: Uniform generation of random Latin rectangles, J. Combin. Math. Combin. Comput. 9 (1991), 179-186. MR 1111853 (92b:05013)

[9] Czeslaw Koscielny: Generating quasigroups for cryptographic applications, Int. J. Appl. Math. Comput. Sci. 12 (2002), no. 4, 559-569.

[10] D. Selvi, G. Velammal, ThevasahayamArockiadoss: Modified Method of Generating Randomized Latin Squares. IOSR Journal of Computer Engineering (IOSR-JCE). e-ISSN: 2278-0661, p- ISSN: 2278-8727 Volume 16, Issue 1, Ver. VIII (Feb. 2014), PP 76-80.