ANÁLISIS DE ALGORITMOS PARA GENERACIÓN DE CUADRADOS LATINOS ALEATORIOS PARA CRIPTOGRAFÍA

por

Ignacio Gallego Sagastume

Directora: Claudia Pons

Tesis presentada para obtener el grado de

MAGISTER EN INGENIERÍA DE SOFTWARE
FACULTAD DE INFORMÁTICA
UNIVERSIDAD NACIONAL DE LA PLATA

LA PLATA, BUENOS AIRES, ARGENTINA
NOVIEMBRE DE 2014

© Copyright by Ignacio Gallego Sagastume, 2014

FACULTAD DE INFORMÁTICA
UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMÁTICA

Los abajo firmantes certifican que han leido el trabajo de grado denominado "**Análisis de algoritmos para generación de cuadrados Latinos aleatorios para criptografía**" por Ignacio Gallego Sagastume.

Fecha: **Noviembre de 2014**

Director: ______________________
Claudia Pons

Codirector: ______________________

Revisión: ______________________

Dedicado a Ale y Emi.

Contenido

Contenido v

Lista de Cuadros viii

Lista de Figuras ix

Resumen xi

1. Introducción: elementos básicos de criptografía 1
- 1.1. Introducción 1
- 1.2. ¿Qué son la seguridad y la privacidad en redes informáticas? 1
- 1.3. Cifrado de los datos 2
- 1.4. Algoritmos simétricos y de clave pública 3
- 1.5. Protocolos criptográficos 4
- 1.6. Tipos de ataques 5
  - 1.6.1. Ataque de solo texto cifrado 5
  - 1.6.2. Ataque de texto plano conocido 5
  - 1.6.3. Modelo de texto plano elegido 6
  - 1.6.4. Modelo de texto cifrado elegido 6
  - 1.6.5. Otros tipos de ataques 6
- 1.7. Tipos de algoritmos y modos de uso 7
  - 1.7.1. Stream Cipher 7
  - 1.7.2. Block ciphers 9
  - 1.7.3. Combinadores de cuadrado Latino (CCL) 9
- 1.8. ¿Por qué interesa el problema de generar cuadrados Latinos de orden 256? 10

2. Cuadrados Latinos y otras estructuras 12
- 2.1. Introducción 12
- 2.2. Algunas estructuras y sus relaciones 12
  - 2.2.1. Otras estructuras 13
- 2.3. ¿Cómo surgieron los CLs y para qué sirven? 14
  - 2.3.1. Diseño de experimentos 14
  - 2.3.2. Aplicaciones criptográficas 15
  - 2.3.3. Entretenimiento 16
- 2.4. CLs ortogonales (MOLS) 16

2.5. Historia . . . 17
2.5.1. Cuadrados Mágicos . . . 17
2.5.2. Cuadrados mágicos en la literatura Islámica . . . 17
2.5.3. Cuadrados Latinos . . . 18
2.5.4. Cuadrados Greco-Latinos . . . 19
2.6. El problema de contar los distinos CLs . . . 20
2.7. ¿Cómo puede usarse un CL aleatorio para cifrar o generar claves? . . . 22
2.8. El problema de generar CLs aleatorios . . . 23

3. Trabajos relacionados **25**
3.1. Introducción . . . 25
3.2. Strube . . . 25
3.3. El enfoque de Jacobson & Matthews . . . 26
3.4. El enfoque de Koscielny . . . 26
3.5. El enfoque de Fontana . . . 27
3.6. Barták . . . 28
3.7. O'Carroll y Selvi (et.al.) . . . 29
3.8. McKay . . . 29
3.9. Wanless . . . 29
3.10. Otras propuestas . . . 30

4. Generación simple con backtracking **31**
4.1. Introducción . . . 31
4.2. Generación de CLs simple por filas . . . 31
4.3. Pseudo-código del algoritmo . . . 33
4.4. Cantidad de colisiones en el peor caso . . . 35
4.5. Cantidad de colisiones esperadas en el caso promedio . . . 38
4.6. Tiempo de ejecución del algoritmo simple . . . 39
4.7. Generación usando producto de dos CLs más pequeños . . . 41

5. Generación con intercambios **43**
5.1. Algoritmo simple con intercambios . . . 43
5.2. Algoritmo simple con intercambios cíclicos . . . 44
5.3. Algoritmo simple con intercambios aleatorios . . . 45
5.3.1. Resultados . . . 46
5.3.2. Pseudo-código resultante . . . 47
5.3.3. Tiempo de ejecución del algoritmo . . . 47
5.3.4. Test de uniformidad del método . . . 49

6. Generación con reemplazos **52**
6.1. Introducción . . . 52
6.2. Descripción del método . . . 52
6.3. El pseudo-código del algoritmo . . . 54
6.4. Tiempo de ejecución del algoritmo . . . 55
6.5. Uniformidad de los resultados . . . 55

7. Implementación del método de Jacobson y Matthews **58**
7.1. Introducción . . . 58
7.2. El método de Jacobson y Matthews . . . 58
7.3. Descripción de los movimientos . . . 60
7.4. Implementación del método . . . 62
7.4.1. Primer implementación: clara pero no tan eficiente . . . 62
7.4.2. Implementación optimizada . . . 64
7.5. Generación paso a paso . . . 66
7.6. Observaciones . . . 67

8. Resultados **69**
8.1. El proyecto Java publicado . . . 69
8.1.1. Sobre la implementación del método simple . . . 70
8.1.2. Sobre la implementación del método con intercambios aleatorios . . . 70
8.1.3. Sobre la implementación del método con grafo de reemplazos . . . 70
8.1.4. Sobre la implementación del método de Jacobson y Matthews . . . 70
8.2. Tiempos de ejecución de los algoritmos . . . 71
8.3. Uniformidad de los métodos . . . 71
8.4. Trabajo futuro . . . 72
8.5. Conclusiones . . . 72

Agradecimientos **81**

Bibliografía **82**

Índice de cuadros

4.1. Pseudo-código con el método iterativo principal 34
4.2. Método con backtracking para generar una fila del CL 35
4.3. Número de colisiones en el peor caso 37
4.4. Implementación del producto de Koscielny 41

5.1. Modificación del método simple para permitir filas con colisiones 47
5.2. El método para eliminar las colisiones 48
5.3. Listado Java del método de Kolmogorov-Smirnov 51

6.1. Construcción del grafo de reemplazos 55
6.2. Tratamiento de colisiones con grafo de reemplazos 56

7.1. Primera versión del cubo de incidencia 62
7.2. Implementación del movimiento 62
7.3. Método conteniendo el bucle principal 63
7.4. Segunda versión del cubo de incidencia 64
7.5. La búsquedas secuenciales son reemplazadas por esta función 64
7.6. Segunda versión del movimiento ± 1 65
7.7. Método para almacenar un 1 en (x, y, z) 65
7.8. Implementación de la aritmética de símbolos 66

Índice de figuras

1.1. Implementación básica de un stream cipher 8
1.2. Cifrado y decifrado en modo CBC 9

2.1. Cifrado de la cadena “abcc” a la manera de Gibson 22

4.1. Ubicación de las colisiones: teorema de la diagonal 36
4.2. Promedio de colisiones por ubicación en 10.001 corridas del algoritmo 38

7.1. CL ejemplo 59
7.2. Diferentes vistas de un cubo de incidencia 59
7.3. Equivalencia entre un cuadrado y su cubo de incidencia 59
7.4. Un movimiento ± 1 60
7.5. Movimiento ± 1 para caso propio e impropio respectivamente 60
7.6. Matriz correspondiente al plano XY de un cubo de incidencia 65
7.7. El gráfico de estados en la generación paso a paso 67
7.8. Ejemplo de la ejecución paso a paso gráficamente 68

Índice de ecuaciones

4.1. Cota superior de la cantidad de colisiones para el peor caso 37
4.2. Cota superior de la cantidad colisiones caso promedio para $n = 16$ 39

5.1. Cantidad de intercambios en el caso promedio (por colisión) 48
5.2. Cantidad de intercambios en el caso promedio por cantidad de colisiones 49

7.1. Invariante: la suma de cada línea del cubo es siempre 1 60

Resumen

Existen estructuras algebraicas aplicables en seguridad informática, como los Latin squares (cuadrados Latinos o CLs), con ciertas propiedades que los hacen muy convenientes para implementar protocolos de comunicación seguros y otras aplicaciones criptográficas como algoritmos de cifrado. En particular, son difíciles de adivinar o calcular por fuerza bruta y sirven como claves de un algoritmo de cifrado simétrico.

Los CLs aleatorios de orden 256 son de particular importancia, porque permiten cifrar y decifrar cualquier carácter de la tabla ASCII, utilizando un recorrido secuencial a la manera "Off the grid" de Gibson.

En un contexto de una aplicación criptográfica como las anteriormente mencionadas (protocolos o algoritmos de cifrado), se debe generar un CL cada cierta cantidad de tiempo o datos transmitidos. Esto no debe representar una sobrecarga de tiempo o recursos, es decir, que el algoritmo de generación debe ser lo más eficiente posible. Es por esto que en el presente trabajo, se analizan distintos algoritmos para generar CLs aleatorios con distribución aproximadamente uniforme.

En la primera parte del trabajo, se hace una introducción a la criptografía aplicada, para revisar conceptos que constituyen el contexto de los algoritmos implementados. En el segundo capítulo, se hace una introducción a los CLs y otras estructuras, su historia y problemas relacionados, como el de generar MOLS (Mutually Orthogonal Latin Squares) o generar Sudokus. En el capítulo 3, se estudian diversos enfoques y trabajos relacionados existentes al problema de la generación de CLs aleatorios con distribución uniforme. En el capítulo 4 se desarrolla un método simple de generación secuencial con backtracking que servirá como base para el algoritmo con intercambios aleatorios desarrollado en el capítulo 5. En el capítulo 6 se muestra otro algoritmo con reemplazos. Estos dos últimos algoritmos se comparan luego con una implementación del método más aceptado para resolver el problema de generación de CLs uniformemente distribuidos, el método de Jacobson y Matthews. En el capítulo 7 se estudia una implementación de este último método y se muestra también una implementación en OpenGL que se utiliza para graficar las estructuras de datos del algoritmo de Jacobson y Matthews, la cual puede ser utilizada con fines didácticos. Por último en el capítulo 8 se analizan los resultados obtenidos, trabajo futuro y algunas conclusiones.

Capítulo 1

Introducción: elementos básicos de criptografía

1.1. Introducción

El presente capítulo introductorio fué construido en base a [Sch95, FSK10, Rit03]. En las siguientes secciones se verán algunos conceptos básicos sobre criptografía y elementos relacionados al contexto del problema de generación de cuadrados Latinos aleatorios.

1.2. ¿Qué son la seguridad y la privacidad en redes informáticas?

Cuando una persona "A" ("Alice") quiere comunicarse con otra persona "B" ("Bob") en otro punto en una intranet o sobre Internet, puede ocurrir que una tercera persona o robot "E" ("Eve" por eavesdropper) pueda interceptar la comunicación y leer los mensajes que se transmitan. Más aún, podría pasar que Eve eliminara los mensajes a mitad de camino y los sustituyera por otros, sin que el emisor (Alice) o receptor (Bob) se dieran cuenta de esto.

Un sistema de comunicación *seguro* o criptográfico debe permitir o cumplir las siguientes propiedades:

- integridad: la información debe ser modificable sólo por las personas autorizadas
- autenticación: debe ser posible para el receptor de un mensaje identificar su origen. Un intruso no debería poder hacerse pasar como alguien más.

- confidencialidad o privacidad: los datos tienen que ser legibles únicamente para los usuarios autorizados
- irrefutabilidad: el usuario no debe poder negar las acciones que realizó

La integridad asegura que los datos enviados no sufren pérdidas o se corrompen, o se modifican a propósito por terceras personas, pudiendo saber en el punto receptor del mensaje si esto sucede, para tomar las acciones correspondientes para reparar el daño.

La autenticación asegura que el emisor se identifique ante el receptor y que nadie pueda tomar su lugar, como se explicó mas arriba.

La privacidad o confidencialidad evita el acceso no autorizado a los datos gracias al uso de algoritmos de cifrado, que dificultan la legibilidad de los datos en un grado lo suficientemente complicado para que no valga la pena el esfuerzo intentar decifrar la información. Un ejemplo podría ser que el tiempo requerido para leer los mensajes fuera de varios años, perdiendo así el valor de la información. La complejidad de decifrar los datos debe ser acorde o congruente con el valor de los datos o el daño que ocasionarían si se dieran a conocer: no es lo mismo la información conteniendo códigos de lanzamiento de misiles nucleares que una receta de cocina o una carta de amor.

La irrefutabilidad indica que si Bob recibió un mensaje de Alice, ésto sería prueba suficiente de que Alice envió el mensaje (Alice no puede refutar el hecho).

1.3. Cifrado de los datos

Para que pueda existir la privacidad en una red de datos, los datos deben viajar cifrados de un punto a otro de la comunicación. Un mensaje es el denominado “texto plano” o “plaintext”. El proceso de “disfrazar” un mensaje para ocultar su escencia es el cifrado o encripción. Un mensaje cifrado o encriptado es el “texto cifrado” o “ciphertext”. El texto plano se denota como M (mensaje) o P (plaintext) y el texto cifrado como C. La función de cifrado se denota E (encripción o encriptación). Ejemplo:

$$E(M) = C$$

la función inversa de E es D:

$$D(C) = M$$

de manera que

$$D(E(M)) = M$$

Todo el propósito de la criptografía es mantener los datos seguros. Los criptoanalistas son los practicantes del criptoanálisis, el arte y ciencia de romper el texto cifrado o códigos.

1.4. Algoritmos simétricos y de clave pública

Los algoritmos simétricos utilizan una misma clave K para cifrar y decifrar los mensjaes (o pueden ser distintas claves pero una es fácilmente calculable teniendo la otra):

$$E_K(M) = C$$

$$D_K(C) = M$$

Se caracterizan por ser más rapidos que los algoritmos de clave pública.

Los algoritmos de clave pública se diseñan de tal manera que existe una clave para cifrar y otra para decifrar los datos, de manera de que teniendo una clave no se puede obtener la otra (en una cantidad de tiempo razonable). Se llaman de clave pública porque la clave de cifrado se hace pública; la clave privada solamente la conoce la persona que debe decifrar el mensaje. En algunos casos, puede ocurrir que al contrario, se utilice la clave privada para cifrar y la clave pública para decifrar.

Un ejemplo de uso de clave pública es si Alice quiere enviar un mensaje M a Bob utilizando el esquema de clave pública, los pasos a seguir serían los siguientes:

1. Alice encripta el mensaje M con la clave pública de Bob K_{Bob1}:

$$E_{K_{Bob1}}(M) = C$$

2. Se envía el mensaje C a traves de la red

3. Bob recibe y desencripta el texto cifrado C utilizando su clave privada K_{Bob2}:

$$D_{K_{Bob2}}(C) = M$$

Por lo general no se distinguirá en la notación de que clave se trate (pública o privada), ya que se utiliza siempre una para cifrar y la otra para decifrar.

Esta serie de pasos nos lleva a la siguiente definición.

1.5. Protocolos criptográficos

Un protocolo es una serie de pasos, incluyendo dos o más partes, designados para cumplir con una tarea. Un protocolo criptográfico, es un protocolo que utiliza criptografía. Es una serie de pasos porque se sigue una secuencia de principio a final, la cual no puede ser alterada. Que incluya dos o más partes significa que una sola persona no constituye un protocolo. Y por último, se debe cumplir con una tarea, es decir que el protocolo debe conseguir algo. Además los protocolos deben cumplir con las siguientes características:

- Todos los participantes deben conocer los pasos y las reglas del protocolo de antemano.
- Los participantes deben estar de acuerdo con los pasos y seguirlos.
- Los pasos no deben ambiguos; los pasos deben ser bien definidos y no debe haber lugar para confusiones.
- El protocolo debe ser completo: debe especificar las acciones a tomar para cada situación posible.

Ejemplos de protocolos pueden ser: ordenar una pizza por teléfono, jugar al póker o votar en una elección. La diferencia con los protocolos criptográficos es que las computadoras necesitan definiciones formales.

Por ejemplo, el siguiente es un protocolo criptográfico para distribución de claves de sesión usando criptografía de clave pública, para luego intercambiar mensajes a través de un algoritmo simétrico (sistema híbrido):

1. Bob le manda a Alice su clave pública
2. Alice genera una clave de sesión aleatoria K, la encripta usando la clave pública de Bob y se la envía a Bob.

$$E_B(K)$$

3. Bob decifra el mensaje de Alice usando su clave privada y obtiene la clave de sesión

$$D_B(E_B(K)) = K$$

4. Ambos encriptan sus comunicaciones usando la misma clave de sesión con un algoritmo simétrico.

Este simple protocolo soluciona un problema muy complejo que es el de intercambio de claves. En nuestro caso, la clave de sesión K puede ser un cuadrado Latino.

Un protocolo de “autoaplicación” (o self-enforcing) es cuando el mismo protocolo garantiza justicia entre las partes. Por ejemplo el protocolo simple de “dividir y tomar”, en donde se tiene una torta a ser dividida en dos partes:

1. El participante 1 divide la torta tratando de aproximarse a que cada parte sea la mitad de la torta
2. El participante 2 elige la parte que desee
3. El participante 1 toma la parte restante

En este caso, si el participante 1 trata de sacar ventaja o desviarse del comportamiento esperado, que es cortar un pedazo más chico que el otro, el participante 2 podrá tomar la pieza más grande, siendo el 1 el más perjudicado.

1.6. Tipos de ataques

Por lo general, al diseñar un sistema seguro, deben tenerse en cuenta varios tipos básicos de ataques. A continuación se explican algunos de ellos.

1.6.1. Ataque de solo texto cifrado

Es el tipo más comun de ataque cuando se habla de romper un sistema criptográfico. Es la situación en la que Alice le envía mensajes a Bob y todo lo que el atacante puede ver es el texto cifrado. Es el tipo de ataque más dificil, porque se tiene la mínima información posible.

1.6.2. Ataque de texto plano conocido

Es cuando el atacante conoce no sólo el texto cifrado, sino también el texto plano. El objetivo más obvio, es obtener la clave de decifrado. Si se logra obtener esa clave, se podrán decifrar todos los mensajes entre Alice y Bob mientras usen la misma clave. ¿Cómo podría conocerse el texto plano? Por ejemplo cuando Alice envía un email incluyendo al atacante o por deducción el atacante podría conocer el encabezado de un email (conocería parte del texto plano).

1.6.3. Modelo de texto plano elegido

Es el siguiente nivel de control. Es más poderoso que el ataque anterior: se tiene la posibilidad de elegir textos planos específicamente diseñados para facilitar el ataque al sistema y conocer su correspondiente texto cifrado. Hay un gran número de situaciones en las que el atacante podría elegir los datos a cifrar. Alice podría obtener información de una fuente exterior (influenciada por el atacante) y transmitir esta información a Bob en forma cifrada. Por ejemplo, el atacante podría enviar un email a Alice, sabiendo que ella enviará este email a Bob.

1.6.4. Modelo de texto cifrado elegido

En realidad el nombre debería ser "ataque de texto plano y texto cifrado elegidos", pero resulta muy largo. En el ataque anterior, el atacante podía elegir el texto plano a cifrar. En este ataque uno elige ambos, el texto plano y el texto cifrado; esto es: para cada texto plano elegido, se puede obtener el texto cifrado correspondiente, y para cada texto cifrado puede obtenerse el correspondiente texto plano que lo generó. El objetivo de este ataque es obtener la clave de cifrado. Con esa clave se podrían decifrar otros mensajes. Cualquier sistema criptográfico debería sobrevivir a este tipo de ataques, ya que no es muy improbable que pudiera darse este escenario.

1.6.5. Otros tipos de ataques

Existen infinidad de otros tipos de ataques, como "DoS" (de las siglas en inglés "Denial of Service") o "DDoS" (de "Distributed Denial of Service"), "Man-in-the-middle" ("hombre en el medio"), "information leakage" ("pérdida de información"), "side-channel attack" ("ataque de canal lateral"), "Birthday attack" ("ataque del cumpleaños"), ataques por "fuerza bruta", etc.

Brevemente, un ataque de DoS consiste en programar un script que genere muchas peticiones a un servicio (por ejemplo cientos de peticiones por segundo), para forzar al servidor del servicio a comportarse de manera anómala o simplemente hacer caer al servicio. DDoS es lo mismo, pero en este caso el ataque es distribuido en varias terminales con el propósito de ser más difícil de rastrear o hacer más eficiente el ataque.

El ataque de "Man-in-the-middle" consiste en borrar los mensajes o paquetes que Alice envía a Bob y enviar a Bob otros paquetes (modificando los originales o creando nuevos). Lo que se pretende es ver, robar o alterar la información transmitida sin ser descubierto.

El ataque del cumpleaños hace uso de la paradoja del cumpleaños para obtener colisiones en las claves de cifrado. La paradoja dice que si hay 23 personas en una habitación , la probabilidad de que 2 personas tengan el mismo cumpleaños es de más del 50 %. Este valor es sorprendentemente alto,

ya que hay 365 posibles cumpleaños. Este mismo hecho se puede hacer para explorar un subconjunto del espacio de claves posibles buscando una colisión.

Los ataques por fuerza bruta, corresponden a explorar todas las posibilidades en un problema, hasta encontrar el caso que se está buscando. Ejemplo, si se tiene una contraseña de 8 bytes, debería explorarse como máximo $256^8 = 18.446.744.073.709.551.616$ posibilidades hasta encontrar el caso que da el acceso al sistema, suponiendo que puedan utilizarse los 256 caracteres de la tabla ASCII en cada byte. Generalmente para el análisis, se considera que pueden explorarse 1 millón de posibilidades por segundo o valores similares, lo cual dará una idea de si el sistema soporta un ataque por fuerza bruta o no. Ejemplo, si se tarda más de un año para adivinar una clave o si se tardan millones de años para adivinar un código secreto podría ser suficiente, teniendo en cuenta el valor y oportunidad de la información.

Lo que Schneier subraya es que un sistema es tan seguro como su punto más débil. Si se tiene un protocolo muy seguro pero se administran mal las claves, por ejemplo, o si se tiene un algoritmo de cifrado excelente pero el tamaño de las claves es demasiado pequeño, el atacante empezará por ese punto débil a tratar de explotar las vulnerabilidades del sistema.

1.7. Tipos de algoritmos y modos de uso

Existen dos tipos básicos de algoritmos simétricos: stream ciphers y block ciphers. Los block ciphers operan en bloques de texto plano y texto cifrado-usualmente de 64 bits y a veces más largos. Los stream ciphers operan en streams de texto plano y texto cifrado de a 1 bit o byte a la vez (a veces también de a palabras de 32 bits). Con un block cipher, el mismo bloque de texto plano se cifrará siempre al mismo texto cifrado, usando la misma clave. Con un stream cipher, el mismo bit o byte de texto plano se encriptará a diferentes valores cada vez que se haga el cifrado.

Un *modo criptográfico* es una forma de usar un cipher: usualmente combina un cipher básico, alguna clase de retroalimentación y algunas operaciones básicas.

1.7.1. Stream Cipher

Los stream ciphers convierten el texto plano en texto cifrado 1 bit a la vez. La implementación más simple de un stream cipher se da en la figura 1.1. Un keystream generator o "generador de stream de clave" (algunas veces llamado running-key generator) libera un stream de bits: k_1, k_2, k_3, ..., k_i. Este stream de clave se "XORea" con un stream de bits de texto plano p_1, p_2, p_3, ..., p_i para producir un stream de bits de texto cifrado. Matemáticamente:

$$c_i = p_i \oplus k_i$$

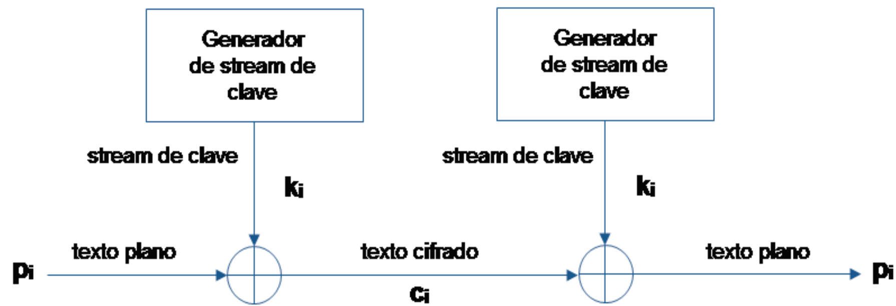


Figura 1.1: Implementación básica de un stream cipher

En el extremo del receptor, el texto cifrado se XORea con el stream de clave para recuperar los bits del texto plano.

$$p_i = c_i \oplus k_i$$

Esto funciona, ya que se tiene que:

$$p_i \oplus k_i \oplus k_i = p_i$$

La seguridad del sistema depende enteramente del generador de stream de clave (RNG). Si genera un sin fin de ceros, el texto cifrado será igual al texto plano, y la operación no tendría sentido. Si el generador libera un patrón de 16 bits que se repitiera siempre, el algoritmo sería un simple XOR con una seguridad ínfima. Si por el contrario el generador de clave liberara una secuencia de de bits aleatorios (no pseudo-aletorios, sino verdaderamente aleatorios), se tendría un OTP (One-Time Pad) y seguridad perfecta. En la realidad práctica, la seguridad queda entre un simple XOR y OTP. El keystream generator genera una secuencia de bits que parece random pero es un stream determinístico que pueda ser reproducido sin problemas al tiempo de decifrado en el otro extremo de la comunicación. En cuanto más se parezca a una secuencia aleatoria, más difícil será romper el código del texto cifrado para un criptoanalista.

Una variedad de diferentes generadores de clave (RNG) pueden utilizarse en un diseño general de stream cipher. Desafortunadamente la mayoría de los RNGs comunes son muy débiles criptográficamente, y necesitan de alguna operación no lineal (ver por ejemplo: jitterizer en [Rit91]). Convencionalmente, el combinador es solamente un XOR o algún otro combinador aditivo, que no agrega seguridad porque es conocido, lineal y fácilmente reversible. Sin embargo, un gran rango de combinadores con clave o no lineales (como sustitución dinámica y combinadores de cuadrados Latinos) pueden agregarse en el diseño del cipher.

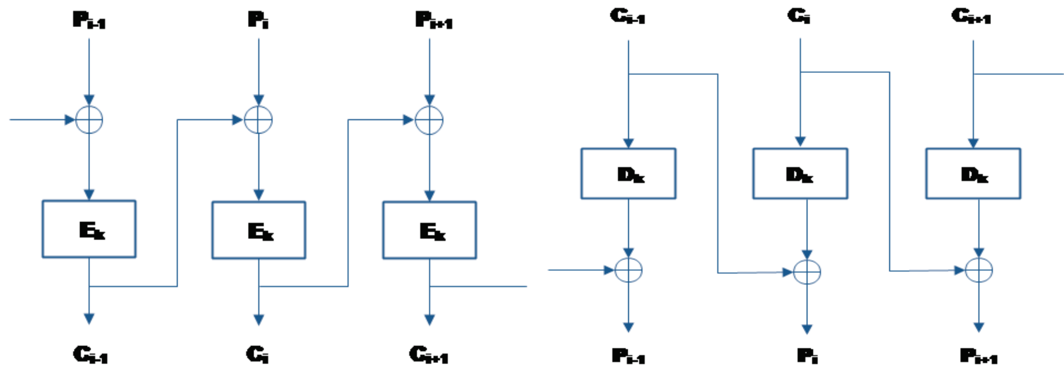


Figura 1.2: Cifrado y decifrado en modo CBC

1.7.2. Block ciphers

Para ver más claramente lo que es un block cipher, se da un ejemplo de su implementación.

Ejemplo de modo de uso: CBC

CBC es por "Cipher Block Chaining Mode". El encadenado de un bloque con el siguiente da al block cipher un mecanismos de feedback; los resultados del cifrado de un bloque se usan para modificar la encripción del bloque siguiente. Cada bloque es dependiente no solamente del texto plano que lo genera, sino también de todos los textos planos de los bloques anteriores. En modo CBC, al texto plano se le aplica una operación XOR con los bloques cifrados antes de encriptarlo. Ver figura 1.2. Matemáticamente sería:

$$C_i = E_K(P_i \oplus C_{i-1})$$
$$P_i = C_{i-1} \oplus D_K(C_i)$$

1.7.3. Combinadores de cuadrado Latino (CCL)

En un contexto criptográfico, un combinador es un mecanismo que mezcla dos fuentes de datos en un solo resultado. Los stream ciphers requieren combinadores reversibles para convertir el texto plano en texto cifrado y viceversa. De esta manera, usando la funcion inversa relacionada (o un mecanismo de extracción) puede decifrarse el texto cifrado. Los clásicos ejemplos son combinadores lineales aditivos sin estado y sin fortaleza, como la suma o el OR exclusivo (XOR), etc.

$$Comb(a, b) \rightarrow c$$

Los combinadores reversibles y no lineales con clave son un resultado de una idea aparentemente revolucionaria de que no toda la seguridad de un stream ciphers debe residir en la secuencia de claves generada. Algunos ejemplos incluyen:

- Combinador de selección de tabla
- Combinador de cuadrado Latino (CCL)
- Combinador de sustitución dinámica

En el caso de los combinadores de cuadrado Latino (CCL), se pueden combinar datos y una secuencia de confusión en el mismo cuadrado: un valor selecciona una fila, y el otro valor selecciona una columna y el resultado es el elemento ubicado en esa fila y columna. Si la secuencia de confusión selecciona columnas, la transformación podría ser revertida en otro CL que tenga como sus columnas la inversa de cada columna en el CL original.

Un CCL es un ejemplo de combinador "balanceado", porque cada valor de salida ocurre con la misma probabilidad (si el CL esta uniformemente distribuido). Cualquier valor de salida puede ser producido por cualquier valor de una entrada, con algún valor en la otra entrada. Esta es la misma clase de combinador producido por XOR, pero con un alfabeto muy grande y la oportunidad de tener complejidad y no linealidad en el mismo combinador (no existe un XOR no lineal). Si el combinador tiene un estado substancial y clave, los ataques de texto plano conocido no exponen inmediatamente la secuencia de confusión como lo harían con un combinador XOR.

1.8. ¿Por qué interesa el problema de generar cuadrados Latinos de orden 256?

Los cuadrados Latinos aleatorios pueden ser usados en protocolos de comunicación (para ver más información sobre protocolos ver [Sch95, FSK10]), para cifrar la información a transmitir, o generar claves para algorimtos de cifrado. Además, ellos pueden ser usados como combinadores (ver sección anterior o [Rit03]) o en mecanismos de autenticación (ver [CP11]). Una técnica para cifrar datos a transmitir fué mostrada en [GS13], a la manera de Gibson "Off the grid" [Gib12]. Se explicará esta técnica nuevamente en la sección 2.7.

El número $L(n)$ de diferentes cuadrados Latinos de orden n es tan grande, que hasta la actualidad sólo se han podido computar hasta $L(11)$ con el poder computacional disponible. Se han establecido cotas superiores e inferiores en general para $L(n)$ (ver [MW05]). Este hecho hace casi imposible que un atacante pueda adivinar un cuadrado Latino mediante ataques por fuerza bruta.

Además, si un cuadrado Latino es de orden 256, sus símbolos se pueden hacer corresponder con códigos de la tabla ASCII y, usando un camino secuencial en el cuadrado (eligiendo elementos adyacentes), puede ser usado para cifrar (y decifrar) cualquier símbolo de un archivo de texto. Esto es posible solamente si el cuadrado es aleatorio y sus elementos están uniformemente distribuidos. Cada elemento debe tener la misma probabilidad de caer en cualquier celda del cuadrado; esto es, los elementos adyacentes no deben estar relacionados entre sí y no deben formar patrones reconocibles en el cuadrado. En otras palabras, el CL debe tener una alta entropía ("entropía" se entiende como el grado de desorden de un sistema).

En el proceso de cifrado, es necesario generar un nuevo cuadrado cada cierta cantidad de tiempo o datos transmitidos, ya que cuanto más se use un cuadrado Latino como clave, mayor el riesgo de que un atacante puede adivinarlo por alguna técnica de deducción o análisis del texto cifrado. Por esta razón, se considera el problema de generar eficientemente cuadrados Latinos de orden 256.

Algunas alternativas para enfrentar este problema serán examinadas en el capítulo 3 (ver [Str88, Kos95, Bar04, Fon13]); éstas se consideran demasiado teóricas, orientadas a matemática, o muy ineficientes para órdenes mayores a 16. Además, los cuadrados Latinos generados en [GS13] no son uniformemente distribuidos.

En el capítulo siguiente se verá la historia y propiedades de los CLs. Luego, en el capítulo 4, se verá un metodo simple de generación y en los capítulos 5 y 6 se verán dos métodos que mejoran el tiempo y la distribución de sus elementos con respecto al algoritmo simple. En el capítulo 7 se dará una implementación del método de Jacobson y Matthews (con dos variantes) y se graficará el proceso de generación con OpenGL en 3D. Se examinarán los resultados obtenidos en el capítulo 8.

Capítulo 2

Cuadrados Latinos y otras estructuras

2.1. Introducción

Existen estructuras algebraicas que tienen aplicaciones en seguridad informática, por ejemplo, los cuasigrupos y los cuadrados Latinos aleatorios, y es útil también conocer sus estructuras relacionadas: tablas de Cayley, lazos (loops), grupos, campos de Galois, etc. En particular, los cuadrados Latinos (CLs) son matrices con propiedades que los hacen útiles para algunas aplicaciones criptográficas ya que pueden ser "shapeless" (o sin forma), es decir que no hay relaciones entre sus elementos ni tampoco se pueden reconocer patrones entre ellos. Al tener una alta entropía o aleatorización de sus elementos, son útiles para usarlos como ciphers (ver sección 2.7).

En las siguientes secciones se analizán algunas estructuras utilizadas en criptografía, como también la historia de los CLs y algunas de sus aplicaciones.

2.2. Algunas estructuras y sus relaciones

Se tiene la siguiente definición de un cuadrado Latino:

Definición 2.2.1. Un Latin square o *cuadrado Latino* (CL) de orden n es una matriz de dimensiones $n \times n$ que se completa con n símbolos distintos, los cuales aparecen (cada uno) n veces en la matriz: exactamente una vez en cada fila y una vez en cada columna ([Fon13]).

Los n símbolos distintos pueden ser los números de 0 a $(n-1)$ o los números de 1 a n, o pueden ser letras o incluso colores. Por ejemplo:

1	3	2
3	2	1
2	1	3

es un CL de orden 3 ($n = 3$), que utiliza los símbolos $1, 2$ y 3. Para más ejemplos y aplicaciones ver [Org04, Rit03, Gib12].

2.2.1. Otras estructuras

Un *cuasigrupo* es un sistema algebraico binario $(Q, *)$ que satisface las condiciones:

- Para cualquier $a, b \in Q$ existe un único $x \in Q$ tal que $a * x = b$.
- Para cualquier $a, b \in Q$ existe un único $y \in Q$ tal que $y * a = b$.

Se escribe a los elementos x e y como $a \backslash b$ y b/a, a estas nuevas operaciones se las llamará *división izquierda* y *división derecha* de b sobre a.

Lo que estas condiciones afirman (expresado más sencillamente) es que cualquier elemento x sólo aparece una vez en cada fila y una vez en cada columna en la tabla de producto de la operación $*$. Es decir que la tabla de multiplicación de la operación $*$ de un cuasigrupo, llamada *tabla de Cayley*, es un cuadrado Latino.

Un elemento a de un cuasigrupo $(Q, *)$ es idempotente si $a * a = a$. El elemento a será llamado *identidad izquierda* si $a * x = x$ $(\forall x \in Q)$, *identidad derecha* si $x * a = x$ $(\forall x \in Q)$ y simplemente *identidad* si es a la vez identidad derecha e izquierda.

Un *lazo* (o loop) es un cuasigrupo que tiene un elemento identidad. Este elemento identidad debe cumplir: $\forall x \in Q : \{i * x = x * i = x\}$. Si además la operación $*$ de la estructura binaria es asociativa, y tiene inverso, es decir: $\forall a \in Q\ \exists \hat{a} \in Q : \{a * \hat{a} = \hat{a} * a = i\}$ (donde $\hat{a}$ se llama opuesto o inverso de a e i es el elemento identidad), entonces el lazo se convierte en *grupo*.

Se tiene entonces que:

$$\{grupos\} \subseteq \{lazos\} \subseteq \{cuasigrupos\}$$

2.3. ¿Cómo surgieron los CLs y para qué sirven?

Parece ser que los CLs fueron originalmente curiosidades matemáticas, pero a principios del siglo XX se encontraron aplicaciones estadísticas, como el "diseño de experimentos".

2.3.1. Diseño de experimentos

El clásico ejemplo es el uso de un CL para poner 3 o 4 variedades de granos diferentes en distintos "parches de prueba". Tener muchos parches para cada variedad de grano ayuda a minimizar los efectos de suelo localizados.

Algo similar ocurre con un ejemplo en tratamientos médicos: el diseño de experimientos basado en CLs puede aplicarse cuando hay exposiciones o tratamientos repetidos y otros dos factores. Un ejemplo de este tipo de diseños sería la respuesta de 5 ratas diferentes (factor 1) a 5 diferentes tratamientos (letras de la A a la E) cuando se los encierra en 5 distintos tipos de jaula (factor 2):

		Rata				
		1	2	3	4	5
	1	B	E	C	D	A
Tipo de	2	E	D	A	B	C
Jaula	3	C	B	D	A	E
	4	D	A	E	C	B
	5	A	C	B	E	D

Este CL diseña una variación sistemática o balance entre jaulas, ratas y los tratamientos que se les aplican (y en qué orden).

Otro ejemplo del uso de los CLs aleatorios en el diseño de experimentos es el de un servicio de citas: se debe agendar citas de tres mujeres (Ann, Bea y Carol) con tres hombres (Al, Brian y Carl), para los días lunes, martes y miércoles, de tal manera que cada mujer se encuentre con cada hombre en los tres días. Una solución podría ser el CL:

	Al	Brian	Carl
Ann	lunes	martes	miércoles
Bea	miércoles	lunes	martes
Carol	martes	miércoles	lunes

Esta misma estructura de CL, podría usarse también realizando una transformación de días (ejemplo: lunes $\rightarrow$ martes, martes $\rightarrow$ miércoles y miércoles $\rightarrow$ lunes).

2.3.2. Aplicaciones criptográficas

En algunas aplicaciones criptográficas, un CL puede ser visto como un "stream cipher combiner" (SCC), un tipo de función XOR generalizada. Un combinador de cuadrado Latino (CCL) es invertible si una de las entradas de la función es conocida, como es generalmente el caso de un SCC. Entre las ventajas de un CCL se encuentran las siguientes: incluyen estado interno masivo, puede soportar claves ("keyed"), y las operaciones de combinación usando claves serían no-lineales. Por esto, las arquitecturas con combinadores lineales, como pueden ser una secuencia de combinadores, o una selección entre combinadores, son posibles aplicaciones de los CLs, ya que estos pueden ser vistos como combinadores no-lineales con estado interno.

En otras aplicaciones criptográficas, un par de CLs ortogonales (ver sección 2.4) pueden ser vistos como un "block mixer" ("mezclador de bloques"), para bloques relativamente pequeños. Los CLs pueden ser construidos o seleccionados desde una plétora de posibles cuadrados, soportando claves. En general, tales cuadrados no serán cíclicos ni triviales, siendo más apropiados para uso criptográfico (esto es, aleatorios y con una alta entropía).

Para bloques más grandes, los CLs ortogonales pueden reemplazar operaciones aritméticas "mariposa" normalmente usadas en FFTs (Fast Fourier Transforms). Como consecuencia, se puede construir un mezclador de bloques parecido a FFT con muy buenas propiedades de mezclado, que soporta claves, es no-lineal, y que puede manejar bloques de ancho dinámicamente diferentes, usando siempre el mismo código de programa. La habilidad de generar bloques de ancho extremadamente grandes es útil, porque permite las características deseadas de claves dinámicas y autenticación inherente por bloque. Para mayor información véase [Rit03].

2.3.3. Entretenimiento

Otra curiosidad y aplicación de los CLs son los Sudokus. Los sudokus, ahora ampliamente difundidos en el mundo como un pasatiempo de tipo rompecabezas de lógica, son CLs con algunas restricciones más (están divididos en n subcuadrados de orden n que deben contener todos los símbolos, etc.). Los sudokus son CLs incompletos (con "agujeros"), los cuales es posible completar hasta obtener un CL completo. Fontana provee formas de generar Sudokus aleatorios en [Fon13].

2.4. CLs ortogonales (MOLS)

Antes de ver la historia en la sección 2.5, se da la definición de CLs ortogonales.

Definición 2.4.1. CL ortogonales: Son dos CLs de orden n que cuando son superpuestos o combinados forman cada uno de los n^2 pares ordenados posibles de n símbolos (producto cartesiano) exactamente una vez.

Todas las entradas de la superposición son distintas, es decir, que conforman las variaciones de n elementos tomados de a 2 con repetición (número A_n^2, donde A es el conjunto de símbolos del cuadrado). Se tienen entonces $\{1,2,3,4\}_4^2 = 4^2 = 16$ elementos distintos.

Se denominan también MOLS (Mutually Orthogonal Latin Squares), es decir, "CLs mutuamente ortogonales" o Greco-Latinos (nombre que les asigna Euler como se verá más adelante). Ejemplo:

$$\begin{array}{cccc} 3 & 1 & 2 & 0 \\ 0 & 2 & 1 & 3 \\ 1 & 3 & 0 & 2 \\ 2 & 0 & 3 & 1 \end{array} \qquad \begin{array}{cccc} 0 & 3 & 2 & 1 \\ 2 & 1 & 0 & 3 \\ 1 & 2 & 3 & 0 \\ 3 & 0 & 1 & 2 \end{array} \quad \text{superposición} \rightarrow \quad \begin{array}{cccc} 30 & 13 & 22 & 01 \\ 02 & 21 & 10 & 33 \\ 11 & 32 & 03 & 20 \\ 23 & 00 & 31 & 12 \end{array}$$

como se vé arriba, se forman todas las combinaciones de 2 símbolos y ninguna se repite (el elemento 11 por ejemplo, aparece solamente una vez).

Pueden existir también un conjunto de MOLS de varios CLs, que cumplen que cualquier par de CLs del conjunto son mutuamente ortogonales entre sí.

2.5. Historia

A continuación se recopila de distintas fuentes una breve historia de cómo surgieron los CLs en la antigüedad y de dónde viene el problema de contar los CLs distintos o MOLS distintos.

2.5.1. Cuadrados Mágicos

Un "cuadrado mágico" es una tabla (de grado primario) donde se dispone de una serie de números enteros en un cuadrado o matriz de forma tal que la suma de los números por columnas, filas y diagonales principales sea la misma, la *constante mágica*. De acuerdo con la revista on-line "Convergence", la idea del cuadrado mágico fue transmitida a los árabes por los chinos, probablemente a través de la India, en el siglo VIII. Fue tratado por Thabit ibn Qurra, conocido por su fórmula para "números amistosos", a principios del siglo IX. Una lista de cuadrados de todos los órdenes desde 3 a 9 se muestra en "La Enciclopedia", recopilada alrededor del año 990 por un grupo de eruditos árabes conocidos como Ikhwan al-safa ("hermanos de la pureza"). Por aquel entonces no aparecieron métodos generales de construcción.

En 1225, coincidiendo con lo mencionado arriba, Ahmed al-Buni mostró cómo construir cuadrados mágicos mediante una sencilla técnica de bordeado, pero puede que él no descubriera el método sólo. Biggs, en referencia a un trabajo de Camman, sugiere que los métodos explicados por Moschopoulos deben tener origen Persa y estarían relacionados con los explicados por al-Buni.

En cambio, Camman alega que los dos métodos dados por Moschopoulos para la construcción de cuadrados mágicos impares eran conocidos por los persas, citando un manuscrito Persa anónimo (Garrett Collection no. 1057, Universidad de Princeton). Aún así, este documento contiene ejemplos pero no métodos explícitos.

El famoso cuadrado mágico de Alberto Durero, grabado en su obra "Melancolía I" está considerado el primero de las artes europeas. Fue realizado en el año 1514.

2.5.2. Cuadrados mágicos en la literatura Islámica

De acuerdo con manuscritos médicos islámicos de la Biblioteca Nacional de Medicina, la primera aparición del cuadrado mágico (conocido en árabe como "wafq") en la literatura islámica se da en el Jabirean Corpus, un grupo de escritos atribuidos a Jabir ibn Hayyan -conocido en Europa como

Gebber-, y que generalmente se piensa que fueron escritos a finales del sigo IX o principios del X después de Cristo.

El Jabirean Corpus recomendaba los cuadrados mágicos como hechizos para facilitar el nacimiento de niños. Estos cuadrados consistían en nueve celdas con los números del 1 al 9 ordenados con 5 en el centro de forma que el contenido de cada fila, columna y las dos diagonales sumaran 15. Los números estaban escritos en letras y números "abjad", y debido a que las cuatro esquinas contenían las letras ba', dal, waw (o "u") y ha', este cuadrado en concreto se conocía como el cuadrado "buduh". Al mismo tiempo, el concepto de cuadrados mágicos llegó a ser tan popular que se asignó propiedades de talismán al nombre "buduh". También se construían cuadrados mágicos de plata como amuletos contra la peste.

En los años posteriores, los escritores islámicos desarrollaron una gran variedad de métodos para la formación de cuadrados mágicos más grandes, en los que no se repetía ningún número y en los que las sumas de cada fila, columna o diagonal eran las mismas. Eran particularmente populares los cuadrados mágicos con celdas de 4×4, 6×6 o 7×7, siendo los cuadrados de 10×10 producidos en el siglo XIII. También parece que los cuadrados mágicos pudieron haber sido introducidos en Europa a través de España por Abraham Ben Meir ibn Ezra (c. 1090-1167), astrólogo y filósofo hispano-judío.

Ben Meir ibn Ezra tradujo varios trabajos árabes al hebreo y tuvo mucho interés en los cuadrados mágicos y la numerología en general. Viajó extensamente por Italia y más allá y debe haber sido una de las personas responsables de la introducción de los cuadrados mágicos en Europa.

2.5.3. Cuadrados Latinos

El concepto de cuadrados Latinos se conoce por lo menos desde los tiempos medievales. Manuscritos árabes del siglo XIII se refieren a veces a los primeros cuadrados Latinos, a menudo dándoles importancia mística "Kabbalika". Un cuadrado Latino, conocido en árabe como "wafq majazi", es un cuadrado que contiene celdas en el cual cada fila y cada columna tienen el mismo conjunto de símbolos, a diferencia del cuadrado mágico, en el que no hay repetición. Estas investigaciones continuaron con el matemático y físico suizo Leonhard Euler (1707-1783). De acuerdo con el Archivo Euler, en su obra "De quadratis magicis" ("Sobre los cuadrados mágicos"), presentada a la academia de San Petersburgo el 17 de octubre de 1776, Euler mostraba como construir cuadrados mágicos

con un cierto número de celdas, en particular 9, 16, 25 y 36. En este documento, Euler empieza con los cuadrados Greco-Latinos (ver 2.5.4) y hace hincapié en los valores de las variables de forma que el resultado es un cuadrado mágico. El nombre de cuadrados Latinos, sin embargo, es debido a un manuscrito de Euler sobre los cuadrados Latinos, llamado "Recherches sur une nouvelle espece de quarre magique" ("Investigaciones de una nueva especie de cuadrados mágicos"). Euler puso letras del latín en una rejilla y lo llamó cuadrado Latino.

2.5.4. Cuadrados Greco-Latinos

Sin embargo, los cuadrados Latinos ortogonales eran bien conocidos antes de Euler. Según lo descrito por Donald Knuth en el Volumen 4 de "El Arte de Programar Computadoras" ([Knu81]), la construcción del conjunto 4×4 fue publicado por Jacques Ozanam en 1725 (en "Récréations mathématiques et physiques") en forma de solitario de cartas. El problema consistía en colocar los ases, reyes, reinas y jotas de una baraja de cartas estándar, en una rejilla de 4×4 de modo que en cada fila y cada columna aparecen los cuatro palos y las cuatro figuras. Este problema tiene varias soluciones, por ejemplo:

$$
\begin{array}{cccc}
A \spadesuit & K \heartsuit & Q \diamondsuit & J \clubsuit \\
Q \clubsuit & J \diamondsuit & A \heartsuit & K \spadesuit \\
J \heartsuit & Q \spadesuit & K \clubsuit & A \diamondsuit \\
K \diamondsuit & A \clubsuit & J \spadesuit & Q \heartsuit
\end{array}
$$

Los cuadrados Latinos ortogonales fueron estudiados en detalle por Leonhard Euler. Él los denominó cuadrados "Greco-Latinos", porque utilizó letras griegas en el primer cuadrado y letras latinas para el segundo. En la década de 1780, Euler demostró métodos para construir cuadrados Greco-Latinos donde n es impar o un múltiplo de 4. Al observar que no es posible constuir cuadrados de orden 2 e incapaz de construir un cuadrado de orden 6 (problema de los treinta y seis oficiales propuesto por él mismo [1]), conjeturó que no existen cuadrados Greco-Latinos para ningún número $n \equiv 2(mod\ 4)$ o, dicho de otra forma, que n sea impar de clase par (multiplo de 2 que no es multiplo de 4). La inexistencia de cuadrados de orden 6 fue confirmado definitivamente en 1901 por Gaston

[1] El problema de los treinta y seis oficiales pregunta si es posible colocar a treinta y seis oficiales de seis regimientos diferentes y de cada uno de los seis grados (en cada regimiento) en un cuadrado de 6×6 de forma que no coincidan dos oficiales del mismo rango o del mismo regimento en ninguna fila y en ninguna columna. Esta disposición forma un cuadrado greco-latino. El problema no tiene solución para $n = 6$.

Tarry a través de la enumeración exhaustiva de todas las posibles combinaciones de símbolos. Sin embargo, la solución a la conjetura de Euler estuvo sin resolverse durante mucho tiempo.

En 1959, R.C. Bose y S. S. Shrikhande construyeron algunos contraejemplos de orden 22 siguiendo puntos de vista matemáticos. Poco más tarde E. T. Parker encontró un contraejemplo del orden 10 utilizando en la búsqueda un UNIVAC, lo que hace que éste sea uno de los primeros problemas de combinatoria resueltos con una computadora digital.

En 1960, Parker, Bose, y Shrikhande (conocidos como los aguafiestas de Euler) demostraron que la conjetura de Euler es falsa para todo $n \geq 10$. Por lo tanto, existen cuadrados Greco-Latinos de orden n para todos los $n \equiv 2(mod\, 4)$, excepto $n = 6$.

Los cuadrados Greco-Latinos se utilizan en el diseño de experimentos, la programación de torneos y la construcción de cuadrados mágicos. El escritor francés Georges Perec estructuró en 1978 su novela "La vida, instrucciones de uso" en torno a un cuadrado ortogonal de 10×10. También son utilizados en algunas aplicaciones criptográficas, como se verá más adelante.

2.6. El problema de contar los distinos CLs

Se conoce como $L(n)$ al número de CLs distintos de orden n. Muchos matemáticos han hecho el intento de calcular $L(n)$, ya sea o bien para un n específico, o bien para obtener una fórmula general. No se conoce una fórmula general que sea computable en la práctica. Se trata de un problema de combinatoria complejo: hasta la actualidad sólo se ha podido obtener hasta $L(n)$, con $n \leq 11$. El número $L(n)$ crece tan rápidamente que el número $L(12)$ no se conoce ni puede ser computable con el poder computacional actual (ver [MW05]). Este hecho hace que usar un CL de orden suficientemente grande hace que sea prácticamente imposible adivinarlo por fuerza bruta. Para tomar dimensión del tamaño de $L(n)$ y lo complejo del problema, pueden mencionarse dos cotas conocidas:

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

Se tiene la siguiente definición:

Definición 2.6.1. CL reducido, $R(n)$: Un CL es *reducido* si la primera fila y la primera columna contienen los símbolos ordenados de 0 a $n-1$.

En la tabla 2.6.1 puede observarse el número de CLs reducidos de orden n hasta $n = 11$ y estimados para $12 \leq n \leq 15$:

n	$R(n)$
2	1
3	1
4	4
5	56
6	9.048
7	16.942.080
8	535.281.401.585
9	377.597.570.964.258.816
10	7.580.721.483.160.132.811.489.280
11	$2^{35} * 3^4 * 5 * 2801 * 2206499 * 62368028479$
12	$1,62 * 10^{44}$
13	$2,51 * 10^{56}$
14	$2,33 * 10^{70}$
15	$1,50 * 10^{86}$

Tabla 2.6.1: CLs reducidos para $n \leq 11$ y estimados para $12 \leq n \leq 15$.

También se tiene la siguiente fórmula a partir de $R(n)$:

$$L(n) = n! * (n-1)! * R(n)$$

Antes de contar, se debe notar que existen CLs que son distintos en nombres de símbolos, pero son iguales en estructura o *isomorfos*. Ejemplo:

$$\begin{matrix} x & y & z \\ y & z & x \\ z & x & y \end{matrix} \quad \text{es isomorfo a} \quad \begin{matrix} y & z & x \\ z & x & y \\ x & y & z \end{matrix}$$

en donde en el segundo cuadrado se ha hecho el siguiente reemplazo de variables: $x \to y$, $y \to z$, y $z \to x$.

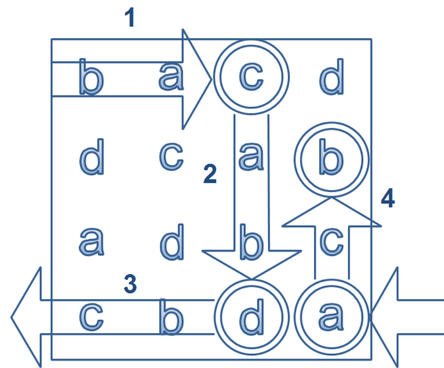


Figura 2.1: Cifrado de la cadena "abcc" a la manera de Gibson

Definición 2.6.2. Isotopismo de CLs: Un *isotopismo* de un CL L es una permutación de sus filas, permutación de sus columnas y permutación de sus símbolos. El CL resultante es llamado *isotópico a* L y el conjunto de todos los cuadrados isotópicos a L es llamado *clase isotópica*. En el caso especial en que la misma permutación se aplica a filas, columnas y símbolos, el isotopismo es llamado *isomorfismo* ([MW05]).

2.7. ¿Cómo puede usarse un CL aleatorio para cifrar o generar claves?

Un CL de orden 256 puede almacenar todos los caracteres de la tabla ASCII y así cifrar casi cualquier texto plano que utilice este juego de caracteres. Nuestro interés es obtener CLs de orden 256 para estos propósitos. Supóngase (para simplificar) que se tiene el CL de orden 4 de la figura 2.1 y se lo quiere utilizar para cifrar la palabra "abcc". Se puede recorrer el CL (a la manera de Gibson [Gib12]) alternando movimientos hacia derecha, abajo, izquierda y arriba. Cuando se termina de recorrer en una dirección y se alcanza el borde del CL, se ingresa cíclicamente por el otro lado.

El proceso para este caso sería el siguiente:

1. Se busca la primera letra a cifrar, la "a", comenzando por la primera fila de izquierda a derecha hasta encontrar esa letra. Se toma la letra siguiente en el CL, la letra "c".

2. Ahora se cambia la dirección y se va hacia abajo. Se busca la segunda letra de la palabra a

cifrar, la “b” y se toma la letra siguiente, la “d”. Se tiene hasta ahora el texto cifrado “cd”.

3. 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 CL cíclicamente por el otro lado. Se toma la siguiente letra del CL, la “a”. Se tiene hasta ahora “cda”.

4. 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 CL (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 CL y se debería generar más frecuentemente. Generar claves aleatorias de esta manera parece razonable: se aprovecha el hecho de que el CL es aleatorio (dos símbolos contiguos no respetan ningún orden) y se toman los símbolos según algún recorrido secuencial. El problema es que el CL usado para cifrar 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.

Un ejemplo de este algoritmo de cifrado se incluye en el paquete Java publicado en internet [GS15].

2.8. El problema de generar CLs aleatorios

Dado un CL aleatorio uniformemente distribuido, se mostró cómo puede hacerse para cifrar información. Existen otras formas de utilizar un CL para cifrar o generar claves (para luego utilizar one-time pad o XOR por ejemplo). Con el objetivo de crear un cipher para un protocolo de comunicación seguro, se debe generar un CL rápidamente y cada cierto tiempo, y esto no debe representar una sobrecarga de tiempo o recursos (memoria, disco rídido, etc.) en el protocolo. Por lo general,

los algoritmos simples de generación de CLs son de orden exponencial, por lo que el problema exige una cierta complejidad para que sea eficiente.

En los siguientes capítulos, se abordará el problema de generar CLs aleatorios y uniformemente distribuidos en un tiempo razonable (polinomial). En el capítulo 3 se verán los distintos enfoques propuestos por los principales autores y el estado del arte del problema.

Capítulo 3

Trabajos relacionados

3.1. Introducción

En las siguientes secciones, se analiza la bibliografía actual e histórica en lo que respecta al problema de la generación de cuadrados Latinos aleatorios. Cabe aclarar que no existen demasiados trabajos en el área y que como los CLs tienen muchas aplicaciones, a veces se encuentran trabajos similares pero que no son útiles para resolver el problema en cuestión.

3.2. Strube

En el año 1988, Michael Strube desarrolló un algoritmo para la micro-computadora Commodore 128 en [Str88], el cual se resume de la siguiente manera:

1. Obtener un CL inicial de tamaño n
2. Reordenar aleatoriamente las columnas
3. Reordenar aleatoriamente las filas
4. Reordenar aleatoriamente los símbolos

Con este algoritmo, sólo pueden alcanzarse un determinado conjunto de CLs (dentro de la misma clase isotópica). El alcanzar todas las clases isotópicas depende del CL inicial, de cómo se genera y si el mismo varía (si se parte o no siempre del mismo CL). Además, el mismo es muy ineficiente, ya

que cada intercambio de columnas llevaría como mínimo n^2 pasos. Para obtener CLs más aleatorios deberían repetirse los pasos de 2 a 4 muchas veces.

Con este método, Strube pudo generar CLs de hasta orden 10, tardando 207 segundos en una Commodore 128.

3.3. El enfoque de Jacobson & Matthews

Este enfoque es el standard de facto en el area para generar CLs aleatorios con distribución *aproximadamente* uniforme. Se basa en un CL inicial al cual se aplican una serie de transformaciones o movimientos para "mezclar" o aleatorizar sus elementos. De acuerdo a Brown [Bro13] hacen falta $(n^3)/8$ movimientos para mezclar el cuadrado completamente (se le da una chance a cada elemento de ser movido, ya que cada movimiento modifica 8 elementos). Los autores recomiendan realizar n^3 movimientos. Es decir que en tiempo polinomial se obtiene un CL con distribución aproximadamente uniforme.

El problema es que no existe una implementación estándar, y la gente que entiende el método y lo implementa, no lo comparte libremente en Internet.

En el capítulo 7 se examinará en detalle como son estos movimientos ± 1 para mezclar un CL hasta obtener uno aleatorio y luego se explicará la implementación que se propone dentro del presente trabajo.

3.4. El enfoque de Koscielny

Koscielny observa el problema desde una perspectiva de la teoría de números (utilizando estructuras algebraicas como campos de Galoise, anillos, grupos, etc.). Citando una frase de su trabajo más reciente en el area ([Kos04]):

" *It is possible to perceive some analogy between contemporary cryptography and the pre-semiconductor era in electronics: generally in all currently proposed and used cryptographic systems encrypting/decrypting procedures compute cryptograms corresponding to given plaintexts, and vice versa, using pure algebraic structures such as groups, rings and fields.* "

es decir, él considera que para la construcción de ciphers nuevos o los actuales se utilizan estructuras como grupos, anillos y campos en estado puro. Esto puede ser cierto, aunque muchos diseñadores de protocolos o algoritmos criptográficos seguramente no tengan el background matemático necesario para comprender estos métodos.

Un avance significativo lo produjo al definir una estructura algebraica nueva en el año 1995, en su trabajo [Kos95], denominada "Spurious Galois Field" (es decir, campo de Galois "adulterado"). Con esta estructura (y otras) ha trabajado desde entonces, desde la perspectiva de estudiar sus propiedades y cómo se aplican a la criptografía en general. En particular, en su trabajo [Kos02] denominado "Generating quasigroups for cryptographic applications", Koscielny desarrolló algunas rutinas en el sistema matemático Maple 7 para generación de CLs aleatorios a partir de estructuras como los campos de Galois o los mismos Spurious Galois Fields y sus operaciones básicas como la suma o el producto de la misma estructura algebraica. A partir de estas rutinas es posible generar CLs de orden 256 en un tiempo aceptable, aunque siempre en un lenguaje matemático y utilizando paquetes Maple (como el paquete "*GF*") como bloques de construcción. Desde entonces ha seguido contribuyendo con nuevas rutinas Maple para uso del público en general en el sitio [Kos14].

Si bien Koscielny dice que estas rutinas sirven para aplicaciones criptográficas, estas rutinas no están implementadas en C o Java (o algún lenguaje de programación de propósito general), con lo que su aplicación en la construcción de un protocolo, sistema criptográfico o elemento de hardware es difícil; se debería traducir todas las rutinas Maple a otro lenguaje, lo cual no sería tarea sencilla. Además se debería tener un amplio background matemático o bibliotecas ya predefinidas de matemáticas avanzadas.

3.5. El enfoque de Fontana

En [Fon13], se enfatiza que el enfoque de Jacobson y Matthews obtiene solamente CLs aleatorios con distribución *aproximadamente* uniforme, pero no realmente uniforme. Construye un grafo basado en todas las permutaciones de n elementos (costoso). Luego propone utilizar una equivalencia entre CLs y los cliques de tamaño máximo en ese grafo. Recordar primero que dado un grafo G, un clique es un subgrafo de G completo.

En la sección 2 de su trabajo, Fontana muestra la equivalencia, demostrando el siguiente teorema:

Teorema 3.5.1. *Los cuadrados Latinos L_n de orden n de $\mathcal{L}_n^{L_n}$ son los cliques ordenados de C_{n-1} de tamaño $n-1$ de $\mathcal{G}_n = (\mathcal{V}_n, \mathcal{E}_n)$.*

Basado en esa equivalencia, propone un método con el que puede generar un CL de orden n aleatorio, con distribución *realmente* uniforme. El problema principal del método, es que debe generar todos los cliques de tamaño máximo de $\mathcal{G}_n$ para luego extraer uno aleatoriamente. Este problema es NP-Completo. Para esto utilizan el paquete *igraph* del lenguaje/sistema *R*. Alternativamente utilizan el procedimiento *Optnet* de *SAS/OR*, o *Cliquer*.

Con un equipo standard con procesador $i7$ y 8GB de RAM se pudo generar CLs verdaderamente aleatorios de hasta orden $n = 7$, en los que encontraron 16.942.080 cliques. El algoritmo pudo extraer un CL de orden 7 de entre los 7!6!16.942.080 = 61.479.419.904.000 posibles. Por esta razón, por el momento este enfoque matemático no es viable para generar CLs de orden 256 en el contexto de desarrollo de una aplicación criptográfica real. Tal vez en los próximos años el poder computacional crezca o se desarrollen alternativas aplicables para el $n = 256$; un ejemplo de esto es el caso de la computación cuántica, que promete cambiar por completo la criptografía moderna.

Una alternativa que propone para tratar de hacer más eficiente el método es tomar subgrafos más pequeños para buscar los cliques y utilizar el CL resultante como punto de partida del método de Jacobson y Matthews.

3.6. Barták

En su trabajo [Bar04], el autor estudia los generadores de problemas aleatorios que pueden ser resueltos (resolubles o satisfactibles), como son el Quasigroup Completion Problem (QCP) y el Quasigroup with Holes (QWH). Esto sería comparable a un generador de sudokus por ejemplo, solamente que generando CLs con "huecos". Estos problemas pueden plantearse de tal manera que sean equivalentes al CSP (Constraint Satisfaction Problem) con n variables.

En el primer problema (QCP), se da un CL parcial y el problema consiste en ver si ese cuadrado puede llenarse hasta obtener un CL completo. Bartak propone un método de generación de estos CLs, con *shallow backtracking* pero luego lo descarta por ser éste menos eficiente que el generador

del segundo problema.

En el segundo problema (QWH), se intenta primero generar un CL y luego extraer algunos elementos para formar agujeros en el CL, de manera que ya se sepa de antemano que el QWH es satisfactible. Para generar esos CLs, el autor utiliza una versión modificada del algoritmo de Jacobson y Matthews, para usar matrices en vez de cubos de incidencia. Lo que hace es posponer asignaciones de valores en estados impropios para tener un solo valor por celda. La eficiencia y eficacia del método, sin embargo, no se modifican.

3.7. O'Carroll y Selvi (et.al.)

En 1963, O'Carroll propone un método [O'C63] para generar CLs que en ciertos casos falla. En 2014 Selvi, Velammal y ThevasahayamArockiadoss modifican el método de O'Carroll agregando backtracking para que pueda completar los cuadrados generados en cualquier caso (ver [SVT14]).

El método genera un cuadrado por filas y luego la fila actual se completa en un orden aleatorio. Se generan CLs aleatorios pero no se habla de la uniformidad de los resultados. Parece un enfoque interesante y se deja como trabajo futuro su implementación en Java.

3.8. McKay

McKay [MW91] propone un metodo muy interesante para generar rectangulos latinos (RLs) de $n \times k$ intercambiando elementos ya colocados realizando "switches" o transformaciones de RLs. Estas trasnformaciones involucran intercambiar elementos en la misma fila (la generación es a partir de un RL con conflictos). Este método es de orden $O(nk^3)$, muy eficiente. Propone también una estrategia de aceptación o rechazo para lograr uniformidad en los resultados. Se propone también como estudio y trabajo futuro.

3.9. Wanless

Wanless [Wan04] propone cambios en CLs para generar otros, construyendo grafos utilizando "trades" de CLs. Los grafos que construye tienen como vertices las clases isotopicas, y los lados del

grafo salen de cambiar ciclos para moverse de una clase a la otra. Se pueden construir estos grafos hasta orden 8 (incluido).

3.10. Otras propuestas

Existen muchos trabajos que sugieren otras aplicaciones de los CLs: en combinatoria [vLW92], en diseño de experimentos (estadística), generación de problemas satisfactibles (ver [AGKS00]), la creación de ciphers usando cuasigrupos (ver [BP13]), o bien como generadores de números pseudo-aleatorios (PRNG) [DM04], etc.; pero la realidad es que no se encontraron muchos enfoques más sobre cómo generar CLs aleatorios con distribución uniforme. La mayoría da por sentado que ya está resuelto el problema con el algoritmo de Jacobson y Matthews.

Capítulo 4

Generación simple con backtracking

4.1. Introducción

En un protocolo seguro de transmisión de datos, pueden utilizarse CLs como claves para encriptar todos los datos que se transfieran. Se deberá generar un nuevo CL cada una determinada cantidad dada de bytes transmitidos o, al menos, una vez por conexión, para no utilizar siempre el mismo y se corra el riesgo de ser adivinado o deducido por algún otro medio. También puede utilizarse un CL como generador de una clave de tamaño igual que el mensaje a transmitir, para luego utilizar OTP (One-Time Pad), cuya clave debe usarse una sola vez y luego descartarse.

Lo que se verá en las siguientes secciones es cuánto cuesta generar un CL, en términos de tiempo computacional. Los resultados de este capítulo fueron expuestos en el trabajo [GS13].

4.2. Generación de CLs simple por filas

El algoritmo más simple posible es ir generando números aleatorios para construir filas de n símbolos distintos e ir agregándolas a una matriz, controlando para cada número generado que no se repita en la misma fila y en la columna. Esto es exactamente como se haría con un lápiz y papel, a prueba y error. El algorimto debe extraer símbolos aleatoriamente desde los disponibles en fila y columna actuales y para el caso en que se quede sin opciones (se llamará a esta situación "colisión"), debe hacer backtracking, es decir, borrar el último símbolo escrito y volver a probar con otro.

Se sabe que el algoritmo siempre termina, pues se conoce el siguiente teorema:

Definición 4.2.1. Un *rectángulo latino* de $k \times n$ es un cuadrado latino parcial, en el cual las primeras k filas están completas y las restantes $n - k$ filas están completamente vacías, para algún valor k, siendo $k < n$.

Teorema 4.2.1. *Todo rectángulo latino puede completarse hasta obtener un cuadrado latino.*

Demostración. Se prueba utilizando el teorema "del casamiento" de Hall (ver [vLW92, MVJ10] y [Kno12]). Simplemente se cumplen las premisas de este teorema. □

Por cada posición, el algoritmo extraerá un símbolo aleatoriamente del conjunto resultante de la intersección de los elementos que tiene disponibles en la fila que se está generando con los disponibles en la columna actual, menos los elementos que ya han sido probados en esa posición y llevan a colisiones. Es decir, se toma aleatoriamente del conjunto:

$$\begin{aligned} \text{Disponible en } (i,j) \quad = (\quad & \{ \text{ símbolos disponibles en fila } i \ \} \bigcap \\ & \{ \text{ símbolos disponibles en columna } j \ \} \ ) \text{ -} \\ & \{ \text{ provocan colisión en } (i,j) \} \end{aligned}$$

Si en algún paso del algoritmo este conjunto de elementos disponibles es vacío, entonces hay "colisión"; es decir, no hay más posibilidades de símbolos para la posición actual. Esto quiere decir que los únicos símbolos disponibles en la fila ya se han usado en la columna actual, o ya se han probado todas los símbolos posibles y todos ellos llevan a la misma situación. En este punto, el algoritmo tiene que borrar el último elemento generado en la fila para probar con otro, a fin de terminar de generar esa fila, y además debe anotar el último símbolo generado como "prohibido" para esa posición, dado que lleva a una colisión. Una colisión se da por ejemplo cuando se ha generado:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \\ 3 & 1 & 2 & \end{array}$$

y el único símbolo disponible para la fila (el 4), ya existe en la última columna. En ese caso, el algoritmo debe hacer "backtracking" (volver hacia atrás) la generación del CL, borrando el último

símbolo de la fila, el 2, y anotar el símbolo 2 como prohibido en la posición (3,3), dado que lleva a una colisión. Entonces el algoritmo puede continuar eligiendo el último símbolo disponible (el 4) en la posición (3,3):

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \\ 3 & 1 & 4 & \end{matrix}$$

generando una nueva colisión, ya que el único símbolo disponible en la fila, el 2, ya existe en la última columna. Se anotan como prohibidos en la posición (3,3) los símbolos [2,4]. Como no se tienen más símbolos para probar en la posición (3,3), el algoritmo debe hacer backtracking otra vez, borrando el símbolo 1 de la última fila y eliminando la lista de símbolos que llevan a colisión en la posición (3,3), ya que estos dependían de la elección del símbolo 1 en la posición (3,2). Ahora se anota el símbolo 1 como prohibido en la posición (3,2) y se prueba con el último símbolo posible, el 4:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \\ 3 & 4 & & \end{matrix}$$

permitiendo así generar el resto de la fila:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \\ 3 & 4 & 2 & 1 \end{matrix}$$

y terminar la generación sin colisiones en la última fila, con el siguiente CL:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \\ 3 & 4 & 2 & 1 \\ 2 & 1 & 4 & 3 \end{matrix}$$

4.3. Pseudo-código del algoritmo

Se implementó en Python ([GS12]) y luego se tradujo a Java el algoritmo antes explicado, obteniendo resultados más que aceptables para un algoritmo tan simple. Se provee un pseudo-código

```
def genCL(n, cl):
  for (int i=0; i<n; i++) {
    fila = generarFila(i, cl)
    if (colisiones[i]>1000000)
      print("Colisiones en fila "+i+"="+colisiones[i])
    cl.setFila(i, fila);
  }
  return cl;
```

Cuadro 4.1: Pseudo-código con el método iterativo principal

por razones de espacio y nivel de detalle. El código Java resultante fue publicado en un repositorio público en Internet junto con los otros métodos para generación de CLs implementados (ver [GS15]).
El primer método que se examina es *genCL()*, que itera de 1 a n para generar las filas; en cada iteración i se genera una fila (invocando al método *generarFila()*) y se copia en el CL resultado. Una vez que finaliza las n filas, termina su ejecución.

El siguiente método es *generarFila()*, que genera una fila del CL. Se dispone de varios conjuntos y variables auxiliares para la generación:

- i_col: es el índice de columna donde se posiciona el algoritmo para llenar el siguiente elemento.
- i_fila: el índice de fila que se está generando
- fila: es el conjunto de símbolos que representa la fila actual generada.
- dispFila: contiene los símbolos que todavía no se utilizaron en la fila que se está generando.
- dispCol: son n conjuntos, cada uno con los elementos que todavía no se utilizó en cada columna (ejemplo: dispCol[1] son los símbolos que quedan disponibles en la columna 1).
- prohib: son n conjuntos indexados por columna, que mantiene los elementos que se han probado y descartado, dado que llevan a colisión (elementos "prohibidos").
- disp: es el conjunto que se explicó mas arriba, de donde se saca un elemento al azar.
- simb: símbolo que se extrae aleatoriamente del conjunto disp.

Además se guarda una matriz de nombre "colisiones" de dimensiones $n \times n$ que cuenta el número de colisiones en cada posición del cuadrado ocurridas durante la generación que se utilizará para

```
def generarFila(i_fila, cl):
  #genero fila de tamaño n en la posición i
  #init. variables con cuadrado latino parcial cl
  while (i_col < n):
    #disponibles es:
    disp = (dispCol[i_col] & dispFila) - prohib[i_col]
    if (disp es no vacío)
      #saco un símbolo al azar:
      simb = random.choice(disp)
      #contabilizo el elegido:
      dispCol[i_col].remove(simb)
      i_col = i_col + 1
      dispFila.remove(simb)
      fila.append(simb)
    else:
      #hay colisión
      #limpio los prohibidos de las columnas de más a la derecha
      #hago backtracking
      i_col = i_col - 1
      #saco el último simbolo de la fila actual
      simb = fila.pop()
      #guardo el símbolo prohibido
      prohib[i_col].append(simb)
      #marco disponible el símbolo en fila y columna
      dispFila.append(simb)
      dispCol[i_col].append(simb)
  return fila
```

Cuadro 4.2: Método con backtracking para generar una fila del CL

hacer un gráfico 3D de la distribución de colisiones por posición del cuadrado (figura 4.2). Si la cantidad de colisiones que demandó la generación de una fila supera un máximo (de 1 millón por ejemplo), se imprime una advertencia en la consola, para saber que la generación tardará más de lo habitual.

El método $generarFila()$ itera hasta obtener n elementos en la fila actual. En cada paso se saca del conjunto de disponibles y en caso de que este conjunto esté vacío hay colisión, y se hace backtracking (se borra el último elemento colocado y se lo anota en el conjunto *prohib* para la columna actual). También en este punto se devuelve el elemento borrado de la fila a los disponibles para usar en la fila y columnas correspondientes.

En las secciones siguientes se examina cuántas colisiones hay en el peor caso y en el caso promedio. Luego se mencionan los resultados de performance para este algoritmo.

4.4. Cantidad de colisiones en el peor caso

Para comenzar se demuestra el siguiente teorema, observando dónde se ubican las colisiones al generar el cuadrado (ver figura 4.1):

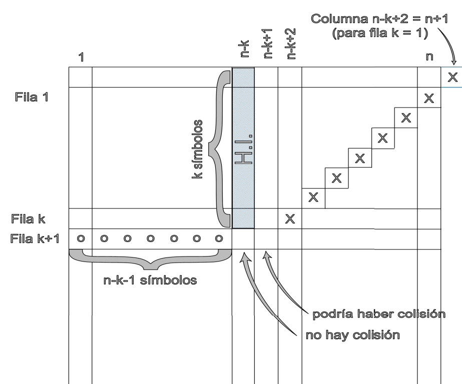


Figura 4.1: Ubicación de las colisiones: teorema de la diagonal

Teorema 4.4.1. *Teorema de la diagonal. En la generación de la fila i sólo puede haber colisiones a partir de la columna* $n - i + 2$ *(y hasta la columna* n*).*

Demostración. Se prueba por inducción sobre el número de fila: para el caso base $p(1)$ se observa que no puede haber colisiones, pues siempre hay símbolos disponibles para elegir en cada columna. Entonces se cumple, pues sólo puede haber colisiones a partir de la columna $n - 1 + 2 = n + 1$, que cae fuera del cuadrado. Se supone $p(k)$: sólo puede haber colisiones a partir de la columna $n - k + 2$. Debe probarse $p(k+1)$: sólo puede haber colisiones a partir de la columna $n-(k+1)+2 = n-k+1$. Supóngase que en la fila $k + 1$ ya se han colocado $n - k - 1$ elementos y se va a colocar el de la columna $n-k$. Por hipótesis inductiva, en la columna $n-k$ no hubo colisiones durante la generación de la fila k, y en esa columna hay k elementos distintos colocados. Supóngase en el peor caso que los colocados en la fila $k+1$ son $n-k-1$ elementos distintos a los k elementos colocados en la columna $n - k$. Se tendrían utilizados $(n - k - 1) + k = n - 1$ elementos distintos. En este caso no habría colisión en la posición $(k + 1, n - k)$, por no haberse utilizado los n símbolos, pero sí podría haber colisión en la posición $(k + 1, n - k + 1)$. □

Se sabe que en la primera y última fila del CL no ocurrirán colisiones: la primera por no tener ninguna restricción previa y la última por estar completamente determinada por el rectángulo latino de $n - 1$ filas $\times$ n columnas.

Si se toma en cuenta además el teorema 4.4.1, que expresa que las colisiones se pueden producir solamente en el triángulo debajo la diagonal que va desde las posiciones $(n, 1)$ a $(1, n)$ (exceptuando

Orden	Cantidad de colisiones (peor caso)
4	3
5	9
6	33
8	873
16	93.928.268.313
20	6.780.385.526.348.313
32	$\approx 274,410 \times 10^{30}$
64	$\approx 3,1986015536 \times 10^{85}$
128	$\approx 2,3911518534 \times 10^{211}$
256	$\approx 1,3192530830 \times 10^{502}$

Cuadro 4.3: Número de colisiones en el peor caso

la primera y última filas), se puede tomar el peor caso (o una cota superior del mismo), es decir, que se tenga una colisión por cada símbolo colocado, sabiendo que ya se han colocado $n-i+1$ símbolos en la fila i. La cantidad de colisiones para la fila i sería como sigue: para la columna $n-i+2$ se tendrían $n-(n-i+1)=i-1$ símbolos disponibles en la fila, que llevarían a $i-1$ colisiones. En la posición $n-i+3$ se tendrían $i-2$ símbolos disponibles (porque ya se ha colocado uno en la posición anterior), que llevarían a $i-2$ colisiones, y así sucesivamente, con lo cual se obtendría la siguiente fórmula considerando todas las filas del CL:

$$\sum_{i=2}^{n-1}(i-1)! \qquad (4.4.1)$$

Cota superior de la cantidad de colisiones para el peor caso

Esto implica que para la generación de un CL de orden 16 habría, en el peor caso, 93.928.268.313 colisiones en total. Para orden 256, el problema es mayor, porque puede haber del orden de 1×10^{502} colisiones en el peor caso y el algoritmo puede tardar varios miles de años en terminar. El número de colisiones crece exponencialmente, como se muestra en la tabla 4.3 (las estimaciones se hicieron usando [Wol12]). De acuerdo a esta tabla, se observa que para generar CLs de orden 16, el número de colisiones en el peor caso todavía puede ser resuelto en un tiempo aceptable. La ecuación 4.4.1 es una cota superior, por lo que en realidad siempre habrá menos colisiones en un caso real. Para órdenes mayores a 30 el algoritmo puede demorar demasiado en terminar, lo que lo hace impráctico.

Se analiza a continuación el número esperado de colisiones o el caso promedio.

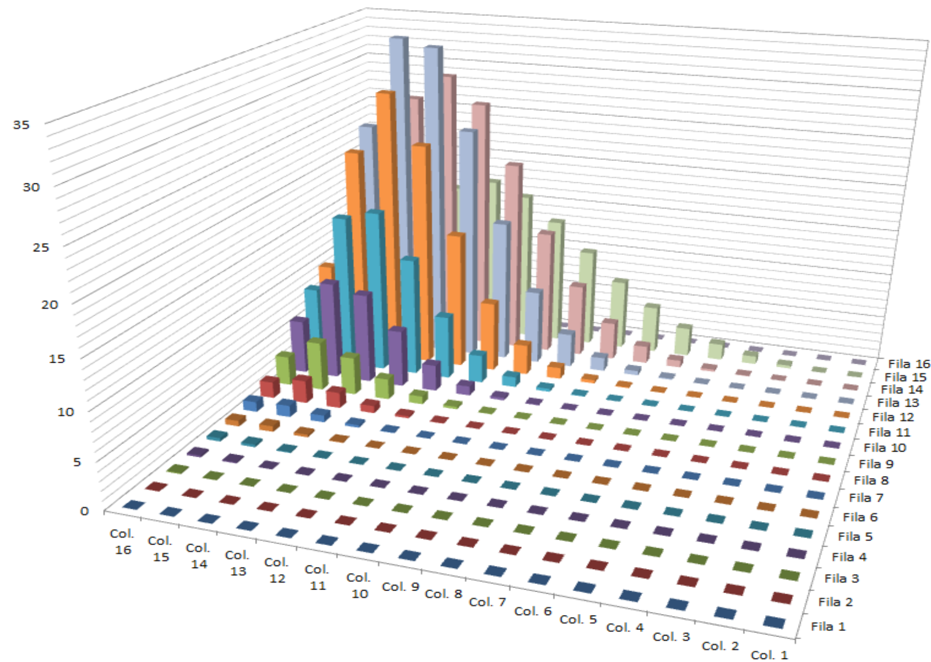


Figura 4.2: Promedio de colisiones por ubicación en 10.001 corridas del algoritmo

4.5. Cantidad de colisiones esperadas en el caso promedio

Dado que en la generación por filas del algoritmo simple se van sacando números al azar, las colisiones que ocurran en cada posición del cuadrado dependen de cuáles sean los valores específicos que se hayan sacado en las filas y columnas previas. Por esto, puede considerarse al número de colisiones en cada posición del cuadrado como variables aleatorias. Se ejecutó 10,001 veces el algoritmo para $n = 16$, pero esta vez contando las colisiones efectivas que se dan en cada generación. Las cantidades de colisiones se almacenan en una matriz para luego calcular el promedio de colisiones para cada posición (i, j), obteniendo así una estimación del valor esperado de colisiones en cada posición (ver [Mey10, JLC11]).

En la figura 4.2, puede observarse cómo se distribuye el promedio de colisiones para cada posición (i, j) del CL. Para las filas 1 y 16 no hay colisiones. Para las demás filas, el valor se va incrementando desde la columna $16 - i + 2 = 18 - i$ (donde i es la fila) hasta la columna 16. El valor máximo se alcanza en la posición $(13, 14)$ y es de 34,47. Se observa que el promedio de colisiones de todos los valores mayores que 0 de la matriz (que contiene promedio de todas las ejecuciones) nunca supera

el valor 9. Se acota entonces el tiempo del algoritmo considerando que en todas las posiciones donde puede haber colisión se produjeran 9 colisiones. La cantidad de colisiones a partir de la columna $n-i+2$ sería entonces la siguiente: dado que los primeros $n-i+1$ elementos no llevan a colisión, se tendrían $n-(n-i+1)=i-1$ elementos que si lo hacen. Si en cada posición hubiera 9 colisiones, se obtendría la siguiente fórmula:

$$\sum_{i=2}^{n-1}(9\times(i-1))=\sum_{i=2}^{n-1}(9i-9) \tag{4.5.1}$$

Cota superior de la cantidad colisiones caso promedio para $n=16$

Esto es, para toda fila distinta de la primera y la última, el número de colisiones será 9 multiplicado por $(i-1)$, lo que para el caso de $n=16$ da un promedio total de 945 colisiones en total para toda la generación. Usando la función estimada para $n=16$ pero aplicándola al caso $n=256$, se tendría un promedio de 289.179 colisiones por ejecución del algoritmo. Sin embargo, se observa que esta estimación es muy irreal, pues en CLs tan pequeños como de orden 30, ya se tienen del orden de 1 millón de colisiones o más en algunos casos para algunas filas.

4.6. Tiempo de ejecución del algoritmo simple

El costo de elegir aleatoriamente un número y ponerlo en una posición del CL es de orden constante. Aún así, se necesitaría en el mejor caso tomar n símbolos y colocarlos en las n filas: eso implica n^2 operaciones (sin considerar las colisiones). Como este es un algoritmo que extrae símbolos aleatoriamente y los coloca en el CL de la manera en que van saliendo en cada fila, el tiempo de ejecución del algoritmo depende de cómo se hayan extraído los símbolos para cada caso en particular y de cuántas colisiones ocurran (variable aleatoria que depende de cada generación). Si la función pseudo-aleatoria tiene alguna tendencia, y su valor esperado no es el valor $n/2$, toda la generación tenderá a ir siempre por casos parecidos. En el contexto de la criptografía, se debe implementar un algoritmo que tendrá una función para extraer símbolos de 1 a n en forma aleatoria, tomando ruido de ambiente (humedad, luz, temperatura) como fuente de datos, o también de fuentes como [Ran12]. Teniendo esto en cuenta, puede suponerse que el algoritmo se comportará más similarmente al caso promedio analizado en la sección anterior.

Supóngase que el tratamiento de cada colisión llevara una constante de c_1 nanosegundos (10^{-9} segundos); se sabe que las colisiones crecen exponencialmente a medida que n crece. Por lo tanto se tendría una función exponencial (que es muy difícil de estimar), sea $b_1^{f(n)}$, multiplicada por la constante de tratamiento de una colisión c_1, siendo b_1 una base constante.

Además, se requieren $n \times n$ operaciones del orden de c_2 nanosegundos para poner los elementos en su posición, que serían $n^2 \times c_2$ nanosegundos. Si a eso se le suman las operaciones de restas e intersección sobre listas para extraer un símbolo aleatorio del conjunto de disponibles que llevan del orden de $n^2 \times c_3$ operaciones. La fórmula quedaría:

$$T(n) = b_1^{f(n)} c_1 + n^2 c_2 + n^2 c_3$$

que sería de orden $O(b_1^{f(n)})$, siempre que $O(b_1^{f(n)}) \geq n^2$.

En la práctica se observa que la generación de un CL de orden 16 lleva entre 0,01 y 0,07 segundos como máximo[1]. Para orden 30, que es un n muy práctico pues pueden utilizarse todas las letras del alfabeto, el algoritmo puede tardar desde 0.5 a 15 segundos aproximadamente en terminar, aunque hay tiempos muy distintos que otros, debido a que el algoritmo toma a veces primero por casos mejores (menor cantidad de colisiones) y otros por casos peores (millones de colisiones por fila). Para un ejemplo de orden 30 se hace la siguiente impresión de consola (cuando en alguna fila se producen más de 1 millón de colisiones, se imprime la fila y el número de colisiones):

```
Colisiones en fila 23=2.522.204
Colisiones en fila 25=3.817.141
Colisiones en fila 26=6.089.187
CL generado en: 12,186879219 segundos.
```

y para otro ejemplo con un caso más rápido (y sin sobrepasar en ninguna fila el millón de colisiones):

```
CL generado en: 0,53406781 segundos.
```

lo que da una idea de las diferencias de tiempo mencionadas anteriormente. Aquí también se observa que la cota del caso promedio (que sería de 3654 colisiones en total) ya no es válida, dado que en algunas filas el número de colisiones supera ampliamente este número.

[1] Los testeos de tiempo se hicieron en una PC de 64 bits con procesador Intel Core i5-3470 @ 3,20Ghz con 4GB de Ram.

```
def multLS(cl1, cl2):
  n1 = len(cl1)
  n2 = len(cl2)
  result = [[0 for i in range(0,n1*n2)]
                for j in range(0,n1*n2)]
  for x in range (0, n1*n2):
    for y in range (0, n1*n2):
      result[x][y]=(n2*(cl1[x//n2][y//n2]))
                   +cl2[x%n2][y%n2]
  return result
```

Cuadro 4.4: Implementación del producto de Koscielny

Para el caso objetivo ($n = 256$) tardaría años en terminar, debido a la gran cantidad de colisiones ocasionadas y la complejidad de los CLs tan grandes. En la sección siguiente, se mejora el tiempo de ejecución usando una noción de producto de dos CLs.

4.7. Generación usando producto de dos CLs más pequeños

Buscando en la literatura, se encontró en un trabajo de Koscielny [Kos02] la noción de generar un CL multiplicando dos CLs más chicos. El producto de dos CLs de orden n_1 y n_2 respectivamente retornará un CL de orden $n_1 \times n_2$. La noción de producto es algo complicada, pero es fácilmente implementable en un lenguaje de programación como Python o Java (ver pseudo-código 4.4).

La implementación del producto es una función que toma dos CLs (cl_1 y cl_2), y retorna un CL que es el producto resultante. En el bucle principal se computa el valor de la posición (x, y) del producto como una función lineal sobre las coordenadas y tamaños de cl_1 y cl_2. El símbolo "//" denota la operación de división entera y "%" es la operación de módulo (resto de la división entera). Como puede verse, el cálculo del producto de dos CLs es de orden $O((n_1 \times n_2) \times (n_1 \times n_2)) = O(n \times n) = O(n^2)$ operaciones. Para generar un CL de orden n se deberían encontrar dos divisores de n, por ejemplo n_1 y n_2, tal que $n_1 \times n_2 = n$ y alguno de ellos se aproxime a $n//2$. Esto es porque es mejor (por ejemplo) multiplicar dos CLs de orden 16 para obtener uno de orden 256, que multiplicar uno de orden 2 y otro de orden 128.

Para el caso objetivo de 256, se utilizarían dos CLs de orden 16 (del cual ya se discutió el tiempo de ejecución y era aceptable) y se aplica la multiplicación en orden de $256^2 = 65536$ operaciones.

Este número es muy pequeño en comparación de las millones de colisiones que se tiene en varias filas para el caso del algoritmo simple con $n > 16$. El tiempo resultante es entonces muy similar al de la generación de un CL de orden 16, es decir, de entre 0,05 a 0,11 segundos.

El problema es que el producto de dos CLs de orden 16 genera un CL de orden 256 con zonas (sub-cuadrados de 16×16) de valores similares. Este hecho, observable a simple vista en un editor de texto, significa que el CL resultante no es uniformemente distribuido.

Para mejorar este resultado se explorarán otras alternativas al método simple en el siguiente capítulo.

Capítulo 5

Generación con intercambios

5.1. Algoritmo simple con intercambios

Para mejorar el tiempo de ejecución del algoritmo simple para $n > 16$, se desarrolló otra versión del mismo, donde se agregó un tratamiento de colisiones mediante intercambio de elementos en vez de hacer backtracking. Lo que se que trata con esto es de evitar las colisiones en cadena, intercambiando elementos ya colocados en la fila o bien uno colocado con el que se está por colocar. Este método busca que cada colisión sea tratada sólo una vez. Se sabe (por el teorema 4.2.1) que dado un rectángulo Latino con k filas completas, siempre es posible encontrar una permutación de elementos en la fila $k+1$ que construya otro rectángulo Latino de dimensiones $(k+1) \times n$. Para encontrar el elemento previo a intercambiar, se busca el primer elemento recorriendo secuencialmente de izquierda a derecha de la fila que cumpla que al hacer el intercambio se elimine la colisión.

Se encontró que existen casos en los que no es posible intercambiar el último elemento con los ya colocados. Por ejemplo:

$$\begin{array}{ccccc} 3 & 1 & 0 & 2 & 4 \\ \textcircled{4} & 3 & 2 & 0 & 1 \\ 1 & 0 & 3 & \textcircled{4} & 2 \\ 0 & \textcircled{2} & \textcircled{1} & 3 & - \end{array}$$

En este caso, se produce una colisión en este punto porque el último elemento restante, el 4, provocaría colisión en la última columna. Cuando se intenta salvar esta colisión intercambiando el elemento que se está por colocar con alguno previamente colocado en la fila, se ve que no es posible:

o bien el 4 se repitiría en la columna previa, o bien el elemento extraido de la columna previa no podría ir en la última columna. Por esta razón se descartó la implementación de este método, porque hay casos como éste en los que la generación no puede completarse. Para salvar este problema, se intenta un nuevo algoritmo, de generación simple con intercambios cíclicos.

5.2. Algoritmo simple con intercambios cíclicos

Para evitar el caso anterior, se busca con este nuevo método intercambiar los elementos ya colocados en la fila actual de forma tal que si al encontrar una colisión, y después de intercambiar el elemento actual sigue habiendo colisión, entonces se vuelve a intercambiar el(los) elemento(s) con colisión, siempre eligiendo la segunda columna a intercambiar de manera secuencial desde el primer elemento de la fila hasta el actual.

Se encuentra otro problema similar: existen casos en los que se entra a un bucle infinito. Por ejemplo, en una generación con $n = 10$:

$$
\begin{array}{cccccccccc}
9 & 8 & 0 & 2 & 5 & 6 & 7 & 4 & 1 & 3 \\
4 & 7 & 6 & 5 & 8 & 1 & 2 & 3 & 9 & 0 \\
8 & 5 & 7 & 6 & 1 & 9 & 0 & 2 & 3 & 4 \\
3 & 9 & 1 & 0 & 7 & 5 & 4 & 8 & 6 & 2 \\
0 & 3 & 5 & 1 & 4 & 8 & 9 & 7 & \textcircled{2} & 6 \\
6 & 2 & 4 & 9 & 0 & 3 & 5 & 1 & \textcircled{8} & 7 \\
7 & 6 & 9 & 8 & 2 & 0 & 3 & 5 & 4 & 1 \\
1 & 4 & 3 & 7 & 6 & 2 & 8 & 0 & \textcircled{5} & 9 \\
5 & 0 & 2 & 4 & 3 & 7 & 6 & 9 & \textcircled{8} & \\
 & & 8 & & & & & & 2 & \\
2 & & & & & & & & 5 & \\
5 & & & & & & & & 2 & \\
 & & 2 & & & & & & 8 &
\end{array}
$$

Si se observa la secuencia de intercambios, se verá que se vuelve a la fila original, donde el 8 estaba repetido en la anteúltima columna.

5.3. Algoritmo simple con intercambios aleatorios

Con la consigna de encontrar cuáles son los casos por los que conviene intercambiar, dado que siempre existe una secuencia de intercambios (una permutación de los n elementos) que salva todas las colisiones, y teniendo en cuenta un concepto de Jacobson y Matthews de permitir colisiones como estados intermedios, se implementó un método que completa primero toda la fila, al igual que lo hacía el algoritmo simple, pero permitiendo colisiones temporalmente. Luego, se ven qué columnas quedaron con colisiones y se intenta cambiar el elemento de esa columna con otro. Simplemente no se puede preveer cuál será el mejor elemento a intercambiar, porque depende totalmente de las filas anteriores, que fueron completadas aleatoriamente y se completaron sin colisiones (las únicas colisiones permitidas se encuentran en forma vertical, con elementos ubicados en la última fila y otros ubicados en filas anteriores). Entonces, lo que se hace es elegir al azar la primera columna del intercambio entre las que tengan colisión y una segunda columna cualquiera (distinta de la primera): si el intercambio de dos esos elementos decrementa o iguala la cantidad de columnas con colisión, entonces lo permitimos; caso contrario, se revierte el cambio. Esto da una idea de dirección en el algoritmo que se había perdido con los dos métodos anteriores. Es importante el concepto de que deje intercambiar cuando la cantidad de colisiones permanezca inalterada por el intercambio, porque esto da la posibilidad de salvar el contraejemplo anterior, donde había un bucle infinito. Por ejemplo, la siguiente es una ejecución posible que salva el problema anterior:

9	8	0	2	5	6	7	4	1	3
4	7	6	5	8	1	2	3	9	0
8	5	7	6	1	9	0	2	3	4
3	9	1	0	7	5	4	8	6	2
0	3	5	1	4	8	9	7	2	6
6	2	4	9	0	3	5	1	8	7
7	6	9	8	2	0	3	5	4	1
1	4	3	7	6	2	8	0	5	9
5	0	2	4	3	7	6	9	8	1
		8						2	
2								5	
								1	5
						1		6	
							6	9	
				9				3	
			3					4	
					4			7	

Se permuta así en cada paso una columna con repetición por otra cualquiera, hasta que se eliminan todas las colisiones y se completa la fila.

5.3.1. Resultados

Este método mejora sensiblemente la generación, ya que cada intercambio se hace en tiempo constante y se elimina el backtracking. Permite generar CLs aleatorios con una distribución aproximadamente uniforme (siempre que el RNG sea realmente aleatorio y no un pseudo-random) de hasta orden 256 en alrededor de 1 segundo [1]. Este método supera en eficiencia al método de Jacobson y Matthews para $n \leq 256$, ya que la cantidad de colisiones converge rápidamente para estos casos con este tratamiento de colisiones por intercambios. Para n más grandes, como 400 o 1000, el método tarda demasiado en converger y el método de Jacobson y Matthews lo supera. Aún así, se considera

[1] Los testeos de tiempo se hicieron siempre en la misma máquina para comparar tiempos

```
@Override
public ArrayList<Integer> generateRow(int i, n, LatinSquare ls, ..) {
  //genera fila de tamaño n en posición i del CL "ls"
  Set<Integer> columnsWithRepetitions = new HashSet<Integer>();
  while (i_col < n) {//mientras falten elementos
    //disponibles es :
    //(disponibles en columna intersección disponibles en fila):

    if (!disponibles.isEmpty()) { //si quedan elementos disponibles
      //tomo uno aleatoriamente de disponibles

      //contabilizo el elegido

      row.add(symbol);
      i_col++;
    } else {//collision
      //tomo uno aleatoriamente de disponibles en fila

      row.add(symbol);
      columnsWithRepetitions.add(i_col);
      i_col++;
    }
  }
  //si hay repeticiones, eliminarlas
  if (columnsWithRepetitions.size()>0)
    this.fixRow(n, ls, row, columnsWithRepetitions, ..);
  return row;
}
```

Cuadro 5.1: Modificación del método simple para permitir filas con colisiones

uno de los aportes originales del presente trabajo. Lo que hay que estudiar, es cuántos intercambios en promedio se deberían hacer, dado n, el orden del CL.

5.3.2. Pseudo-código resultante

El código del cuadro 5.1 muestra cómo se reimplementa el método para generar la fila nuevamente, esta vez permitiendo colisiones y luego el cuadro 5.2 muestra el método $fixRow()$, donde se eliminan las colisiones para terminar la generación de la fila.

5.3.3. Tiempo de ejecución del algoritmo

Lo que hay que ver aquí es cuál es el promedio de cantidad de intercambios por fila (y por colisión) dado un CL de dimensión n. Se presume que el algoritmo se comporta en tiempo polinomial, y es

```
private void fixRow(int n, LatinSquare ls, ArrayList<Integer> row,
                    Set<Integer> columnsWithRepetitions, ...) {
  do {
       //tomar columna con repeticiones
       //tomar otra columna random cualquiera

       //intercambiar los elementos
       if (hay más colisiones) //se empeora estrictamente la situación
            //intercambio de nuevo (dejo como estaban)

  } while (haya columnas con repeticiones);
}
```

Cuadro 5.2: El método para eliminar las colisiones

por eso que soporta grandes valores de n como 256.

A medida que se desciende en la generación por filas de un CL, cada columna tiene menos posibilidades para llenarse en cada caso, por lo que cuesta más que salga sorteada la combinación de columnas justa que elimina las colisiones. Si se tiene por ejemplo un CL de orden $n = 256$ y se está generando la fila 255 (anteúltima) y se tiene una sola colisión en la fila, se tomaría esa columna con colisión primero (sea esa columna i) y se sortearía una segunda columna, sea esa columna j. Supóngase que el elemento repetido en la columna i sólo tiene una posible columna k en donde luego del intercambio se elimina la colisión. Deberían esperarse aproximadamente n sorteos para obtener la columna $j = k$ que elimina la colisión. Si se vuelve a la generación de la fila anterior (la 254) y se tiene una colisión a eliminar, el caso es un poco mejor, porque existen dos columnas k_1 y k_2 por las cuales se eliminaría la colisión, ya que faltan colocar 2 elementos por columna. Se deberían esperar (utilizando el mismo criterio anterior) aproximadamente $n/2$ sorteos para obtener la columna $j = k_1 \vee j = k_2$. Para la fila 253 se tendría un elemento más que haría que se pudiera eliminar la colisión con 3 posibles valores de j, por ejemplo $j = k_1 \vee j = k_2 \vee j = k_3$ y habría que esperar $n/3$ sorteos en promedio. Siguiendo este razonamiento se tiene:

$$\sum_{i=2}^{n-1} \frac{n}{(n-i)} \tag{5.3.1}$$

Cantidad de intercambios en el caso promedio (por colisión)

Faltaría ver cuántas colisiones se generarían en cada fila. Se estima que hay pocas columnas con

colisión (como por ejemplo como máximo $n/4$ por fila), ya que en la generación siempre se intenta sacar el siguiente elemento de los elementos disponibles; en caso de que se agoten las posibilidades de disponibles en fila y columna, se pone un elemento repetido, pero eso no depende del orden de extracción, sino de la calidad del RNG: si hay tendencias, habrá más colisiones en la última fila generada. Si se da como cota máxima de columnas con colisión en $n/4$, se tendría la siguiente fórmula:

$$\sum_{i=2}^{n-1}(\frac{n}{(n-i)} \times n/4) = \sum_{i=2}^{n-1} \frac{n^2}{4(n-i)} \tag{5.3.2}$$

Cantidad de intercambios en el caso promedio por cantidad de colisiones

Esta fórmula da la idea de que toda la generación, considerando el tratamiento de colisiones y si se repite esto para cada fila, es de orden polinomial de aproximadamente $O(n \times \sum_{i=2}^{n-1} \frac{n^2}{4(n-i)}) \approx O(n^3)$. Es por esto que el método mejora notablemente el método con backtracking.

5.3.4. Test de uniformidad del método

Se implementó en Java el test de Kolmogorov-Smirnov para comparar dos distribuciones de probabilidad. La función de referencia FUNC de este test se definió como una función de distribución perfectamente uniforme. Se numeraron los $L(n)$ CLs de 1 a $L(n)$, se generó un número muy superior (10.000.000) de cuadrados examinando para cada CL generado a que numeración pertenecía; se contaron los CLs generados para cada numeración. Se definió cada componente del arreglo de datos del test como:

```
data[i] = "Cantidad de ocurrencias del cuadrado i" / L(n)
```

Los resultados fueron buenos, ya que para $n = 4$ y $n = 5$, con $L(n) = 576$ y $L(n) = 161280$ respectivamente, se generaron todos los cuadrados posibles y en menor tiempo que para el método de J&M. La cantidad de instancias del CL más probable y del menos probable fueron 56926 y 8387 respectivamente para el caso de $n = 4$, y 725 y 2 respectivamente para el caso de $n = 5$. Estos datos indicarían que la distribución no es completamente uniforme, aunque todos los cuadrados son posibles.

Los resultados de este test para los casos de 10 millones de generaciones de CLs, fueron el estadístico PROB (significancia) igual a 1 y distancia máxima entre las dos distribuciones comparadas de aproximadamente 0,983872. El test demoró 12 horas 55 minutos en terminar.

```
private void ksone(int cantExperim, List<Double> data) {
  double en=cantExperim;
  double d=0;
  double fo=0;  //data's c.d.f. before the next step.
  for (int j=1; j<=cantExperim; j++) {     //Loop over the sorted data points.
    double fn=j/en;     //data's c.d.f. after this step.
    double ff= func(data.get(j));   //Compare to the user-supplied function.
    double dt=  Math.max(Math.abs(fo-ff),Math.abs(fn-ff));    //Maximum distance.
    if(dt>d)
      d=dt;
    fo=fn;
  }
  double prob = probks(Math.sqrt(en)*d);     //Compute significance.
  System.out.println("Test concluded. Significance: "+prob+". Maximum distance: "+d);
  return;
}
private double func(double t) {
  if (t<=0)
    return 0;
  else if (t<=1)
        return t;
       else
        return 1;
}
private double probks(double alam) {
  double eps1=0.001;
  double eps2=0.00000001;
  double a2=(-2)*(Math.pow(alam, 2));
  double fac=2;
  double probks=0;
  double termbf=0;     //Previous term in sum.
  for (int j=1; j<=100; j++) {
    double term=fac*Math.exp(a2*Math.pow(j,2));
    probks=probks+term;
    if ((Math.abs(term) < (eps1*termbf)) || (Math.abs(term) < eps2*probks))
      return probks;
    fac=-fac;     //Alternating signs in sum.
    termbf=Math.abs(term);
  }
  probks=1;     //Get here only by failing to converge.
  return probks;
}
```

Cuadro 5.3: Listado Java del método de Kolmogorov-Smirnov

Capítulo 6

Generación con reemplazos

6.1. Introducción

Con el propósito de mejorar el algoritmo con intercambios, surge la idea de realizar reemplazos concientes en vez de intercambios aleatorios (como si no hubiera orden). La idea, es que en vez de hacer reemplazos para corregir una fila completa, cada vez que haya una colisión se utilice algún elemento ya utilizado en la fila para liberar el conflicto y proseguir con la generación. Cabe aclarar que la generación sigue siendo por filas y columnas de manera secuencial.

6.2. Descripción del método

El método hace uso del siguiente hecho: cuando hay una colisión, los elementos disponibles en la fila que se está generando $\{x_1, x_2, .., x_n\}$ y los disponibles en la coluna actual $\{y_1, y_2, .., y_n\}$ son $x_i \neq y_j \forall (i, j)$. Lo que se necesita hacer en este punto es sacar un elemento anterior de la fila que esté dentro de los y_i para liberarlo, y así proseguir con la generación de la fila. Ahora bien, en el hueco del elemento liberado se debe poner alguno de los disponibles en esa columna al momento de iniciar la generación de la fila que sea distinto al que se eligió en un primer lugar. Al realizar el reemplazo del elemento liberado por otro de los disponibles, puede generarse una colisión en la fila. Entonces al elemento repetido (la ubicación que estaba antes de producir la colisión en la fila) y se realizará otro reemplazo por los disponibles en esa columna, y así se repetirá el proceso hasta que no haya colisiones en la fila actual y se haya liberado el elemento que se necesitaba. Puede ocurrir que en

la secuencia de reemplazos, se entre en un ciclo eligiendo un elemento ya antes intercambiado; para evitar esto, se lleva cuenta del camino recorrido (los elementos que se van agregando o reemplazando en la fila). También puede ocurrir que en una posición no haya opciones posibles en la columna actual que no repitan elementos: en ese caso se debe borrar el camino recorrido y tomar otro elemento para comenzar una nueva secuencia de reemplazos. Ver el siguiente ejemplo para clarificar todo esto:

```
          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=[0]
```

En este caso, los x_i serían sólo: $x_1 = 0$ y los y_j serían $1, 2, 3, 5$. Como se ve, al ser estos conjuntos disjuntos, hay una colisión. En este punto se construye un grafo de reemplazos de la siguiente manera:

```
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]
```

Para cada posición se listan los elementos que podrían ir, sin causar colisiones dentro de alguna columna. Entonces se sortea el elemento y_j que se quiere liberar en la fila: por ejemplo el 2. Entonces en la columna 2 y se reemplaza el 2 por alguno de los elementos posibles del grafo anterior ([0,1,3]), por ejemplo tómese el 1. En este punto el camino sería [2,1] y el cuadrado quedaría:

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

en este punto, se debe ir al elemento 1 que estaba originalmente (el de la columna 4) y realizar otro reemplazo para salvar la colisión de la fila. Se toma aleatoriamente de los disponibles en columna

4 ([2,3,4]) pero que no estén en el camino actual (se excluye entonces al elemento 2). Supongamos que se toma el elemento 3. El camino quedaría [2,1,3]:

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

ahora , se debe salvar otra vez la colisión producida en la fila. Se toma el 3 original (columna 3) y se reemplaza por alguno del grafo en esa columna ([0,4,5]). Supongamos que se toma el elemento 0. Se tendría:

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

En este punto, se ha salvado la colisión que había originalmente y el elemento 2 se ha liberado, permitiendo ahora finalizar la fila:

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

6.3. El pseudo-código del algoritmo

Se darán ahora los métodos para el caso del tratamiento de una colisión. Los otros métodos se heredan de la clase *SimpleGen*. El método *constructReplGraph* (que se muestra en el cuadro 6.1) construye el grafo de reemplazo en el momento de una colisión (para reparar ese caso particular). Simplemente se toma el conjunto de disponibles en cada columna al inicio de la generación de la fila actual, y se asigna a cada posición del mapa o grafo. Ejemplo, en la posición j, se pone un conjunto con los elementos inicialmente disponibles en esa columna.

Luego, en el método *makeElemAvailable* (ver cuadro 6.2), se corrige la fila para que el elemento pasado como parámetro quede disponible para usar en la posición donde se originó la colisión.

```
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);
  }
  return map;
}
```

Cuadro 6.1: Construcción del grafo de reemplazos

6.4. Tiempo de ejecución del algoritmo

Se estima que la secuencia de reemplazos es no mayor a n pasos, donde n es el orden del CL. Esto quiere decir que a lo sumo se utilizan n pasos para reparar una colisión. El tiempo total es polinomial y depende de la cantidad de colisiones encontradas en el caso particular de cada generación.

En las pruebas "benchmark" realizadas, el tiempo de ejecución supero al de la versión implementada de Jacobson y Matthews en más de 5 veces. Es decir, que si J&M tardaba 50 segundos en completarse para un orden de 256, este método tarda de 7 a 10 segundos para el mismo orden.

6.5. Uniformidad de los resultados

Se calculó el estadístico del test χ^2 de Pearson que es:

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

Donde k es el número de grupos y grados de libertad. Este estadístico resulta muy grande, fuera de escala. ¿Cuál es el precio que se debe pagar por la eficiencia mejorada? La no uniformidad. Hay CLs que son más probables que otros. Si el test se hace con 10 millones de generaciones para orden 5, los 161280 CLs posibles son generados. El que más veces se generó lo hizo 637 veces, y el que menos

```
public void makeElemAvailable(Integer old, HashMap<Integer, HashSet<Integer>> map,
  ArrayList<Integer> row, Integer col,
  HashSet<Integer>[] availInCol, HashSet<Integer> availableInRow) {

  boolean finished = false;
  int firstElem = new Integer(old);

  //esto es para no volver a poner el elemento que se desea liberar
  this.eraseFirstElemFromGraph(map, firstElem);
  int idx_old = row.indexOf(old);
  int idx_new;

  HashSet<Integer> path = new HashSet<Integer>();
  while (!finished) {
    HashSet<Integer> avail = new HashSet<Integer>();
    avail.addAll(map.get(idx_old));
    avail.removeAll(path);

    if (avail.isEmpty()) {//Path no good, begin again
      path = new HashSet<Integer>();
      avail.addAll(map.get(idx_old));
    }
    int newElem = RandomUtils.randomChoice(avail);
    idx_new = row.indexOf(newElem);//indice del elemento antes de reemplazar

    //reemplazo
    row.set(idx_old, newElem);

    //guardar en camino
    path.add(newElem);

    if (row.indexOf(old)==-1) //si old no esta en la fila
      availableInRow.add(old);
    availableInRow.remove(newElem);

    availInCol[idx_old].add(old);
    availInCol[idx_old].remove(newElem);

    //el elemento está disponible en la fila y no hay repeticiones: termino
    finished = (availableInRow.contains(firstElem) && idx_new==-1);

    idx_old = idx_new;
    old = newElem;
  }
}
```

Cuadro 6.2: Tratamiento de colisiones con grafo de reemplazos

2. El estadístico del test Chi-cuadrado indicaría la no uniformidad de los resultados. Sin embargo, los resultados respecto de la uniformidad son muy similares al método de generación con intercambios aleatorios, por lo que para casos en los que se necesite eficiencia por sobre uniformidad, el método descripto en este capítulo será el elegido.

Además la cantidad de posibles CLs de orden 256 ($L(256)$) es mayor o igual [JM96] a:

$$\frac{(n!)^{2n}}{n^{n^2}} \approx 3{,}047154 \times 10^{101723}$$

entonces la probabilidad de obtener CLs iguales en una generación es muy baja, con lo que podemos admitir una pequeña tendencia en nuestro generador, sin que eso comprometa la seguridad del cifrador.

Capítulo 7

Implementación del método de Jacobson y Matthews

7.1. Introducción

En las siguientes secciones se muestran los detalles de la implementación de Jacobson & Matthews [JM96]. Se han definido estructuras de datos y varios algoritmos en Java, con buenos resultados. Esto permite que en un número polinomial de pasos, se pueda generar un CL con distribución *aproximadamente* uniforme. Como resultado adicional, se implementaron métodos en OpenGL [GO04], que permiten visualizar tanto los resultados de la ejecución del algoritmo, como también el proceso de generación, que puede ser utilizado con fines didácticos.

Los resultados expuestos en este capítulo, fueron presentados en [GS14].

7.2. El método de Jacobson y Matthews

Para explicar cómo funciona el método de Jacobson y Matthews, se debe definir una estructura análoga al CL, que es su cubo de incidencia:

Definición 7.2.1. Cubo de incidencia Un cubo de incidencia de un CL de orden n es un cubo de dimensiones $n \times n \times n$ cuyos 3 ejes corresponden a las filas, columnas y símbolos del CL [Bro13]. El cubo tendrá un 1 en la posición (a, b, c), si el CL desde el cual se derivó tiene un símbolo "c" en la fila a y columna b. En caso contrario, la posición (a, b, c) tendrá valor 0.

Por ejemplo, si se tiene el CL de la figura 7.1, su cubo de incidencia correspondiente sería el de

a	c	b
b	a	c
c	b	a

Figura 7.1: CL ejemplo

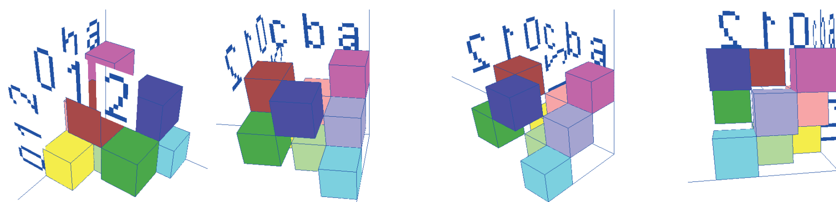


Figura 7.2: Diferentes vistas de un cubo de incidencia

la figura 7.2. En esta última figura se muestra el mismo cubo rotado 3 veces alrededor del eje x, y la última vista corresponde a la vista desde atrás (desde un valor z negativo). El cubo amarillo de dimensiones 1×1 representa el valor 1 de coordenadas $(0, 0, c)$, mientras que el azul representa el 1 en las coordenadas $(2, 2, b)$; el verde oscuro sería el $(1, 2, c)$ y el celeste el $(0, 2, a)$.

Para ver mejor la equivalencia entre un cuadrado latino y su cubo de incidencia, se puede ver el esquema de la figura 7.3. En este esquema, los puntos de colores en el cubo representan valores 1 correspondientes a los círculos del cuadrado del mismo color. Intuitivamente, es como si el cuadrado se hubiera abierto en una dimensión más (hacia atrás en la figura), que es el eje de los símbolos utilizados (en este caso los símbolos $\{a, b, c\}$).

Para preservar la propiedad “Latina” (no tener repeticiones de símbolos en filas o columnas del cuadrado), cada linea del eje x, y, o z debe tener un valor 1 y todas las otras celdas deben contener

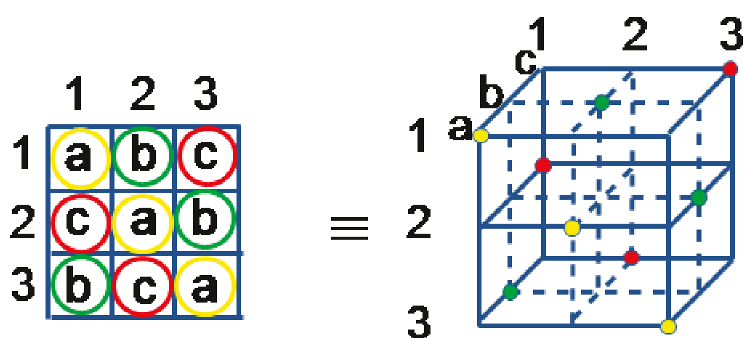


Figura 7.3: Equivalencia entre un cuadrado y su cubo de incidencia

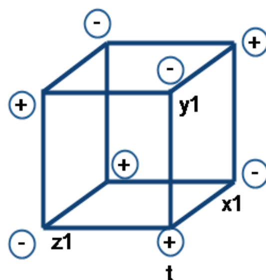


Figura 7.4: Un movimiento ±1

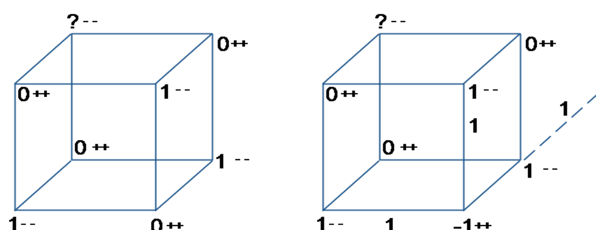


Figura 7.5: Movimiento ±1 para caso propio e impropio respectivamente

0's. Por lo tanto, cada suma de los elementos de una línea debe ser igual a 1. Esto es, si cualquier par de coordenadas se fija en un valor, se tendría la fórmula 7.2.1 (invariante).

$$\sum_{i=1}^{n} cube(i, c_1, s_1) = \sum_{j=1}^{n} cube(f_1, j, s_2) = \sum_{k=1}^{n} cube(f_2, c_2, k) = 1 \tag{7.2.1}$$

Invariante: la suma de cada línea del cubo es siempre 1

Esta invariante se mantendrá en toda la ejecución del algoritmo, pero se permitirá que en algunas líneas haya un valor -1, y dos 1s.

7.3. Descripción de los movimientos

Los movimientos ±1 suman y restan 1 a ciertos elementos de una coordenada de fila, columna o símbolo del cubo de incidencia, de tal manera que el invariante 7.2.1 se mantenga. Sin embargo, un movimiento puede producir una celda que contenga un sólo valor -1. Como la suma no se altera, la coordenada de fila (respectivamente de columna y símbolo) que contiene el valor -1 también tendrá dos valores 1. Los cubos con esta "anomalía" serán llamados "impropios".

Hay dos tipos de movimientos ± 1: desde un cubo propio y desde un cubo impropio:

1. Cuando es desde un cubo propio, se elije aleatoriamente (de forma equiprobable) una celda con valor 0 de las $n^2 \times (n-1)$ celdas 0 disponibles. Sea esa celda t, con coordenadas $(t.x, t.y, t.z)$. Esta tupla t define 3 lineas en cada eje, cada una conteniendo una sola celda con valor 1. Estas celdas con valor 1 determinan un subcubo de dimensiones $2 \times 2 \times 2$ (compuesto no necesariamente de celdas adyacentes). La coordenada x del punto t donde se ubica el valor 1 se llama x_1. Análogamente, los valores 1 de las otras coordenadas son llamadas y_1 y z_1 (ver figura 7.4). El movimiento consiste en sumar 1 a la celda t y restar 1 de las celdas 1 (x_1, y_1 y z_1) de manera tal de no alterar la suma en cada línea. Se propagan las sumas y restas alternadas en los vértices del cubo con este criterio. Si luego de aplicar el movimiento la celda opuesta a t de coordenadas (x_1, y_1, z_1) tiene un valor -1, el nuevo cubo es impropio; en caso contrario el nuevo cubo es propio. Ver en figura 7.5 el cubo de la izquierda (los ++ representan una suma de 1 y los - - representan una resta de 1).

2. Para el caso de cubos impropios, el subcubo se determina de la siguiente manera: la tupla t es la celda que contiene el valor -1 (hay una sola celda -1 en el cubo). La coordenada x correspondiente al punto t contiene un -1 (en $t.x$) y otros dos valores 1. Se elige uno de estos dos 1s de forma aletaoria (equiprobablemente), y esa coordenada x será x_1. Análogamente se elige y_1 y z_1. La suma y resta se hace exactamente como en el caso anterior: se suma 1 al punto t, obteniendo un valor 0, y se resta 1 de las celdas con valor 1 en x_1, y_1 y z_1, de tal manera de no alterar la suma de cada línea. Propagando así las sumas y restas, la única celda indeterminada es otra vez (x_1, y_1, z_1). Si este valor es -1, el nuevo cubo es impropio; en caso contrario el cubo es propio. Si el cubo es impropio, la celda “impropia” se almacena en una variable. Ver en figura 7.5 el cubo de la derecha.

Cualquiera de estos dos tipos de movimientos, puede resultar en un cubo propio o impropio. Si el algoritmo encuentra un cubo propio, aplica el movimiento explicado en el punto 1) de antes; si encuentra un cubo impropio se aplica el punto 2).

Jacobson y Matthews prueban que cualquier par de cubos (propios o impropios) están conectados por movimientos ± 1, y que están separados como máximo por $2(n-1)^3$ movimientos (cota superior).

```
public class IncidenceCube {
  //cada celda contiene 0,1,or -1 (para cubos impropios)
  protected int[][][] cube = new int[n][n][n];//n=order
  protected boolean proper = true;
  //si es impropio, se guarda la celda "impropia"
  protected OrderedTriple improperCell = null;
}
```

Cuadro 7.1: Primera versión del cubo de incidencia

```
protected void move(OrderedTriple t, int x1, y1, z1) {
  cube[t.x][t.y][t.z]++; //suma 1 a la celda seleccionada
  cube[t.x][y1][z1]++;
  cube[x1][y1][t.z]++;
  cube[x1][t.y][z1]++;
  cube[t.x][t.y][z1]--; //restas 1
  cube[t.x][y1][t.z]--;
  cube[x1][t.y][t.z]--;
  cube[x1][y1][z1]--;
}
```

Cuadro 7.2: Implementación del movimiento

En la siguiente sección, se discuten los principales aspectos de la implementación del algoritmo, y se ejemplifica con gráficos OpenGL.

7.4. Implementación del método

La implementación puede ser clara o eficiente, pero es difícil alcanzar estas dos propiedades al mismo tiempo. Se implementaron dos opciones, las cuales se explican en las secciones siguientes.

7.4.1. Primer implementación: clara pero no tan eficiente

La primer versión de un cubo de incidencia para implementar los movimientos ± 1 es una clase con un arreglo de 3 dimensiones de enteros; cada lugar del arreglo contendra un 0, un 1 o un -1. La longitud del arreglo es n, el orden del cubo (ver cuadro 7.1).

Cuando se determina que un cubo es impropio, la bandera "proper" se activa. En el caso de un cubo impropio, la bandera se desactiva y la celda impropia (la celda que contiene el valor -1) se almacena en la variable *improperCell*.

```
public int shuffle() {
  int iterations;
  for (iterations=0; iterations< MIN_ITERATIONS ||
        !this.proper(); iterations++)
    if (this.proper())
      this.moveFromProper();
    else
      this.moveFromImproper();
  return iterations;
}
```

Cuadro 7.3: Método conteniendo el bucle principal

El movimiento ± 1 se implementa muy simplemente en un método que recibe la tupla t y los valores x_1, y_1 y z_1 como parámetros. Para cualquiera de los dos tipos de movimientos, el método suma y resta 1, como se muestra en el código del cuadro 7.2. La primera suma en este cuadro corresponde a la celda 0 o -1 seleccionada (tupla t). Luego de invocar a este método, se debe chequear si la celda ubicada en las coordenadas (x_1, y_1, z_1) contiene un valor -1 para determinar si el cubo es propio o impropio. Si es impropio, entonces esa celda se almacena en la variable *improperCell*.

El método *shuffle()* implementa el bucle principal, cada iteración realizando un movimiento ± 1. Se repite al menos *MIN_ITERATIONS* veces (n^3 por ejemplo), y luego de eso se siguen aplicando movimientos hasta encontrar el próximo cubo propio. El código dentro del bucle en el cuadro 7.3 es como sigue: si el cubo es propio, seleccione una celda 0 y aplicar el movimiento. Si es un cubo impropio, seleccionar la celda impropia y aplicar el movimiento. Se devuelve el número de iteraciones que llevó generar el cuadrado (puede variar) con fines estadísticos.

La ventaja de esta implementación es que es muy simple de explicar y entender. El principal problema que tiene es que, dadas dos coordenadas (x, y), es difícil de encontrar la coordenada z en donde aparece el valor 1 y se debe buscar secuencialmente por todo el eje z muchas veces durante la ejecución del algoritmo. En el peor caso, estas búsquedas son de $O(n)$ pasos.

Para mejorar este punto, se implementa la segunda versión de la clase *IncidenceCube*, la cual no es tan simple, pero elimina las búsquedas secuenciales.

```
public class EfficientIncidenceCube extends IncidenceCube {
  private final int nullInt = -99999999;
  private final int minus0  = -999;//0 tiene negativo!

  private final int max = 3;//como maximo 3 valores: (-z1,z2,z3)
  protected int[][][] xyMatrix = new int[n][n][max];
  protected int[][][] yzMatrix = new int[n][n][max];
  protected int[][][] xzMatrix = new int[n][n][max];
}
```

Cuadro 7.4: Segunda versión del cubo de incidencia

```
@Override
public int plusOneZCoordOf(int x, int y) {
  int z = this.indxOfFirstPositiveElem(xyMatrix[x][y]);
  if (z>=0)
    return xyMatrix[x][y][z];
 . . .
}
```

Cuadro 7.5: La búsquedas secuenciales son reemplazadas por esta función

7.4.2. Implementación optimizada

Como se sugiere en el trabajo [JM96], se implementaron 3 matrices de 2 dimensiones, cada una correspondiente a una de las 3 vistas de los planos positivos de los ejes de fila, columna y símbolo. Cada posición (x, y) (por ejemplo) de la matriz XY, almacenará el valor z (respectivamente x e y para las matrices YZ y XZ) en donde aparece el valor 1 de la línea determinada por los valores fijos x e y. Para las líneas impropias, habrá 3 valores: dos correspondientes a los valores positivos de z (respectivamente de x e y para las matrices YZ y XZ) donde se encuentran los valores 1 del cubo, y un valor negativo, en donde se almacena el valor -1 en el cubo. Esto permite que las búsquedas por los valores 1 se hagan en tiempo constante, o como máximo en 3 operaciones (ver cuadro 7.5).

El rápido acceso de un lado es más costoso en el otro. El movimiento ± 1 es un poco más complicado en este caso, porque hay que actualizar las 3 matrices con cada suma, y se debe implementar una pequeña aritmética, con operaciones para la suma y resta. Por ejemplo, si el algoritmo tiene el conjunto impropio {6, 7, -4} y tiene que sumar -6, debería obtener {7, -4} (los opuestos se cancelan). Si despues tiene que sumar 4 al último conjunto, debería obtener {7}. Y si tuviera que agregar 1 a eso, obtendría {1,7}.

a	b	c
b	b	a-b+c
c	a	b

Figura 7.6: Matriz correspondiente al plano XY de un cubo de incidencia

```
@Override
public void move(OrderedTriple t,  int x1, y1, z1) {
  this.xyzStore(t.x,  t.y, t.z);
  this.xyzStore(t.x,  y1,  z1);
  this.xyzStore(x1,   y1,  t.z);
  this.xyzStore(x1,   t.y, z1);
  this.xyzRemove(t.x, t.y, z1);
  this.xyzRemove(t.x, y1,  t.z);
  this.xyzRemove(x1,  t.y, t.z);
  this.xyzRemove(x1,  y1,  z1);
}
```

Cuadro 7.6: Segunda versión del movimiento ± 1

Una vista del plano XY para un CL de orden 3 podría ser como en la figura 7.6 (los símbolos se expresan en letras). Intuitivamente, para eliminar la impropiedad en esta matriz, los valores z (símbolos) "b" (elemento central) y "-b" podrían ser cancelados entre sí, y la letra "c" (una de las ocurrencias positivas de la misma celda) podría moverse del conjunto {a-b+c} hacia el hueco creado por la cancelación en el elemento central.

El movimiento ± 1 se implementa ahora como se muestra en el código 7.6. Para cada movimiento, estas 3 matrices se deben actualizar. El método *xyzStore()*, almacena un 1 en la posición (x, y, z) del cubo de incidencia.

Y la implementación de la aritmética de símbolos mencionada anteriormente se muestra en la sección de código 7.8. En este fragmento, *minus(elem)* representa el valor negativo del elemento pasado como parámetro. Si el elemento es el entero 0, entonces *minus(elem)* retornará una constante

```
protected void xyzStore(int x, int y, int z) {
  this.add(xyMatrix[x][y], z);
  this.add(yzMatrix[y][z], x);
  this.add(xzMatrix[x][z], y);
}
```

Cuadro 7.7: Método para almacenar un 1 en (x, y, z)

```
private int add(int[] arr, int elem) {
  int idx = ArrayUtils.indexOf(arr, minus(elem));
  if (idx>=0) { //if -element is found
    arr[idx] = nullInt;//-elem+elem = 0
    return idx;
  } else {
    idx = this.getEmptySpace(arr);//for new element
    if (idx==-1) {//if full , fail
      return -1;
    } else {//add the new element
      arr[idx] = elem;
      return idx;//index for the new element
    }
  }
}
```

Cuadro 7.8: Implementación de la aritmética de símbolos

especial que representa al elemento llamado "-0". Esto es porque se utilizan números para representar el conjunto de símbolos. Si el valor negativo del elemento *minus(elem)* se encuentra en el conjunto, se cancela con el elemento *elem* y se elimina del conjunto. Si el elemento negativo no se encuentra, el método busca por un espacio vacío en el arreglo y almacena el nuevo elemento allí. Análogamente, los metodos para resta *xyzRemove(int x, int y, int z)* y *subtract(int[] arr, int elem)* son implementados.

Esta implementación mejora considerablemente el tiempo de generación del CL. La implementación "clara" tomaba en promedio 7,8 segundos para generar el CL de orden 256, con aproximadamente 2.098.000 iteraciones (movimientos ± 1). La implementación "eficiente" explicada en esta sección toma en promedio 1,5 segundos (cerca de 5 veces más rápido) en la misma máquina, usando aproximadamente el mismo número de iteraciones.

7.5. Generación paso a paso

Para comprender qué hace el algoritmo en cada movimiento, se implementó un método *drawIncidenceCube()* en las clases *IncidenceCube* y sus subclases. Este método utiliza OpenGL para graficar la estructura interna del cubo de incidencia en cada paso. Si el usuario presiona las teclas de flechas, la figura 3D rotará alrededor de los ejes x e y. Además el usuario puede presionar las teclas PAGE-UP y PAGE-DOWN para modificar el zoom (acercar y alejar). La barra espaciadora

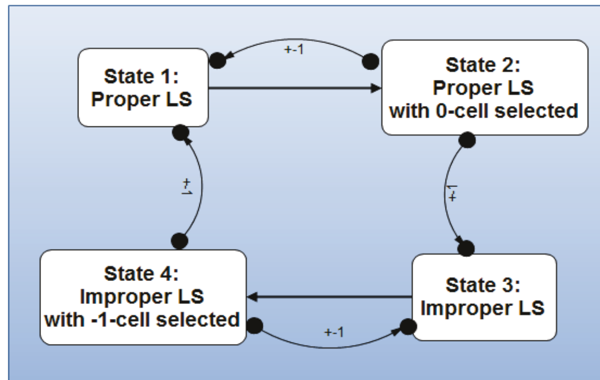


Figura 7.7: El gráfico de estados en la generación paso a paso

disparará un movimiento ±1 en el cubo. El mismo método se implementa en la subclase *IncidenceCubeWithDebugging()*, pero en esta clase dividirá cada movimiento en 2, como se muestra en la figura 7.7.

Cada vez que el usuario presione la barra espaciadora, el estado cambiará al siguiente, de acuerdo al estado actual (cubo propio o impropio) y la tupla seleccionada t. Esto mostrará gráficamente cómo trabaja el algorimto. Se muestra un ejemplo en la figura 7.8: el primer gráfico es del cubo inicial (cíclico), luego se muestran varios movimientos y el resultado final con el cubo totalmente aleatorio. Las líneas en rosa representan el subcubo de $2 \times 2 \times 2$ seleccionado, que es el destino del movimiento ±1, antes de aplicar las sumas y restas. El subcubo rojo representa el valor -1, los verdes representan los valores 1 y los azules son los valores 0 seleccionados.

7.6. Observaciones

La selección de la celda 0 y de la celda 1 aleatoriamente para cada movimiento se implementa usando un generador de números aleatorios criptograficamente fuerte (CS-RNG), *SecureRandom*, que se distribuye con la versión de Java más reciente. Se dice criptográficamente fuerte porque cumple con ciertas normas de seguridad que hacen que los números generados sean impredecibles. Tiene como desventaja que la función de extraer un número al azar es un poco más lenta. Si se lo desea, se puede cambiar la clase que se instancia de *SecureRandom* a *Random* solamente, sin que esto altere al resto de los paquetes o clases Java del proyecto.

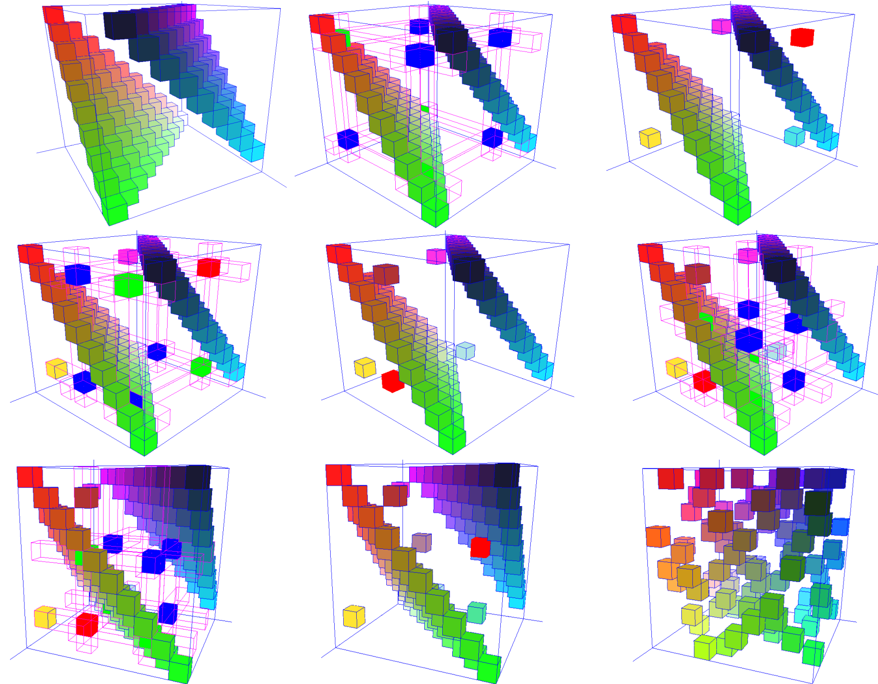


Figura 7.8: Ejemplo de la ejecución paso a paso gráficamente

La implementacion de los métodos en OpenGL son independientes de la representación del CL y los métodos para mezclar los cubos, lo cual hace posible que el usuario trabaje con ambos independientemente según sus necesidades.

La implementación del cubo de incidencia "eficiente" se hizo usando tipos de datos primitivos como *int* e *int*[], para evitar el boxing y unboxing automático de los tipos de datos Objeto, y hacer la comparación de enteros más eficientemente (sin usar el método $compareTo()$). El tiempo de ejecución para la generación de un CL usando este método es polinómica ($O(n^3)$) respecto del orden del cuadrado. Esto es porque cada movimiento ± 1 se hace en tiempo constante (como máximo en 3 operaciones) y es suficiente con n^3 iteraciones para alcanzar un cubo propio que es uniformemente distribuido.

Capítulo 8

Resultados

8.1. El proyecto Java publicado

Se publicó un proyecto Java en un repositorio de Github (ver [GS15]) con las siguientes implementaciones:

1. El método simple con backtracking
2. El producto de CLs de Kocielny
3. El método con intercambios aleatorios (el de intercambios simples y cíclicos estan publicados pero fueron marcados como obsoletos, por no funcionar para cualquier caso)
4. El método simple con grafo de reemplazos
5. El método de Jacobson y Matthews con las dos versiones: la "clara" y la "eficiente"
6. Métodos en OpenGL para graficar cubos de incidencia, con varias opciones distintas de visualización (colores, tamaño, leyendas, etc.)
7. Un cipher llamado "AlexCipher" de ejemplo de uso de los CLs
8. Distintos tests estadísticos para comprobar la uniformidad de los resultados

A continuación se mencionan los principales resultados conseguidos y algunas conclusiones.

8.1.1. Sobre la implementación del método simple

La implementación del método más simple tiene como intención el poder ser comparada con un método ya probado y aceptado como es el de Jacobson y Matthews. Se diseñó también una variante para hacer esa implementación más eficiente: el método del producto de Koscielny. En este caso, el resultado es muy eficiente pero no produce el resultado buscado de uniformidad en la distribución de los elementos del CL.

8.1.2. Sobre la implementación del método con intercambios aleatorios

Esta implementación es muy eficiente (más que la implementación de Jacobson y Matthews) para el caso objetivo de $n = 256$. En la implementación se utilizó la clase Java *SecureRandom*, que es un "cryptographically strong pseudo-random number generator", es decir un PRNG, pero que tiene un período muy grande, lo que lo hace impredecible en la práctica. Sin embargo, el método tiene una tendencia, y genera algunos CLs más frecuentemente que otros.

Para valores de n más grandes (como 400 o 1000), sin embargo, este método tarda demasiado en converger para eliminar todas las colisiones de las últimas filas.

8.1.3. Sobre la implementación del método con grafo de reemplazos

Esta implementación es más eficiente que la anterior, en un factor de 2 a 1 aproximadamente, y es (aproximadamente) 5 veces más eficiente que el método de Jacobson y Matthews. Es la implementación preferida de todas las que no son J&M. También tiene una tendencia, pero para el caso objetivo de 256, podemos utilizarla sin compromiso de la seguridad, dado el inmenso espacio de claves, que tiene como cota inferior:

$$\frac{(n!)^{2n}}{n^{n^2}} \approx 3{,}047154 \times 10^{101723}$$

La probabilidad de generar dos CLs iguales es muy baja, dado el gran espacio de claves.

8.1.4. Sobre la implementación del método de Jacobson y Matthews

La implementación Java del método de Jacobson y Matthews se optimizó lo más posible para que sea útil. El proceso de desarrollo se ha explicado y el código fuente resultante se hizo explícito

en un proyecto en Github [GS15], para que toda persona que esté interesada pueda descargarlo y/o mejorarlo o contribuir de alguna manera al proyecto. Los algoritmos allí expuestos pueden ser compilados en una biblioteca JAR (Java Archive), e incluída en cualquier proyecto que necesite generar o usar cuadrados Latinos para aplicaciones criptográficas u otros propósitos [1].

Como un resultado adicional, se proveen métodos implementados en OpenGL para permitir al usuario graficar cualquier cubo de incidencia intermedio o final, entender como funciona el algoritmo y usarlo con fines didácticos.

8.2. Tiempos de ejecución de los algoritmos

La implementación eficiente de Jacobson y Matthews lleva $O(n^3)$ iteraciones o movimientos. Cada movimiento se produce en tiempo constante, por lo que toda la generación es en tiempo polinomial de $O(n^3)$.

La generación de CLs utilizando el método simple sin mejoras es de tiempo exponencial. Utilizando el producto de dos CLs de orden 16, el tiempo se mejora para el caso de 256 a un tiempo razonable (sigue siendo exponencial pero dependiente de $n = 16$, que es un n chico). También pueden combinarse los métodos de intercambios o reemplazos para generar los cuadrados de orden 16 y luego multiplicarlos.

La implementación del método simple con intercambios aleatorios permite generar CLs en tiempo polinomial (aproximadamente n^3), pero mayor al tiempo de Jacobson y Matthews para el caso general. La implementación del método de reemplazos baja el tiempo del método de intercambios aún más, haciéndolo práctico.

8.3. Uniformidad de los métodos

En la siguiente tabla, se muestran algunas ejecuciones del test de Chi-cuadrado para distintos ordenes y distintos algoritmos. Como se puede observar, a mayor el número de generaciones para el algoritmo de grafo de reemplazos, mayor el estadístico Chi-cuadrado. Para el caso de Jacobson

[1] El código se libera bajo licencia GNU GPL v3, siendo obligatorio nombrar la fuente y hacer público el código si se lo utiliza en productos gratuitos o comerciales.

y Matthews, cuando se aumenta mucho el número de experimentos, es menor el estadístico y los resultados tienden a ser más uniformes.

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	-
Repl Graph	30	3.059.453	49,170 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

8.4. Trabajo futuro

Un problema en el que puede trabajarse en el futuro es cómo utilizar CLs en aplicaciones criptográficas, como pueden ser un cipher, un PRNG o un protocolo criptográfico.

También pueden estudiarse otras formas de generación distintas de Jacobson y Matthews, utilizar CLs en protocolos de conocimiento cero, ver el rol de los CLs como claves de algoritmos simétricos, observar otras formas para encriptar usando CLs distintas a Gibson, utilizar CLs como key-stream generators en stream ciphers y estudiar posibles ataques usando CLs en criptografía.

8.5. Conclusiones

Se investigó el origen e historia de los CLs y los contextos, campos y conceptos sobre los que pueden aplicarse estas estructuras y sus algoritmos de generación.

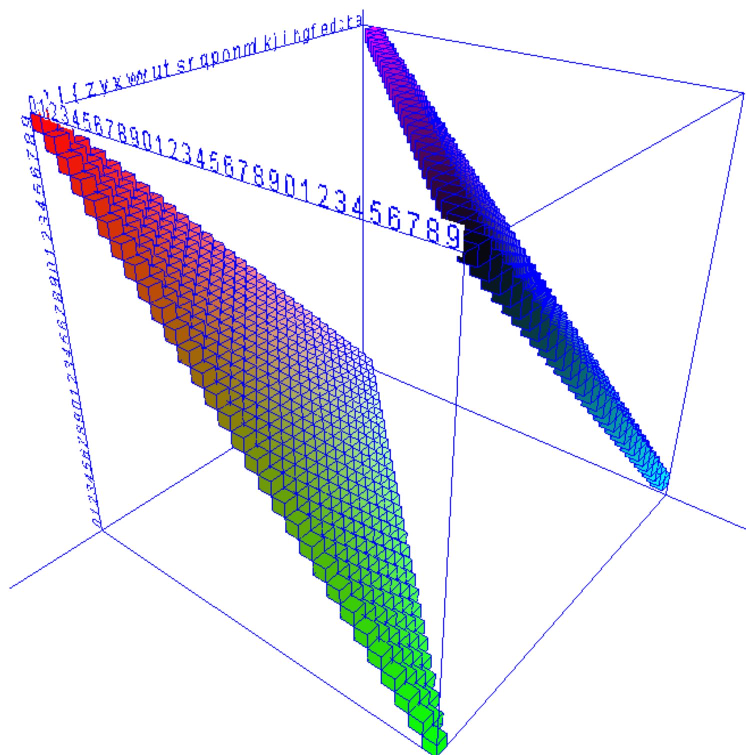
Se desarrollaron varios algoritmos para generación de CLs aleatorios, con distribución aproximadamente uniforme. Algunos de ellos eran conocidos y otros se encontraron mediante prueba y error.

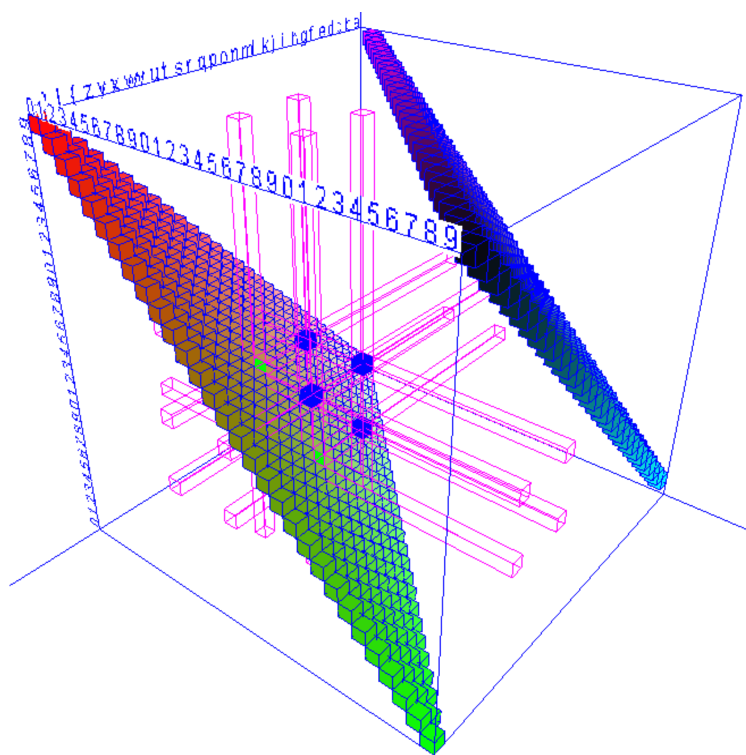
Otro desarrollo original fué la implementación de las gráficas de las estructuras de datos utilizadas, que pueden utilizarse con fines didácticos.

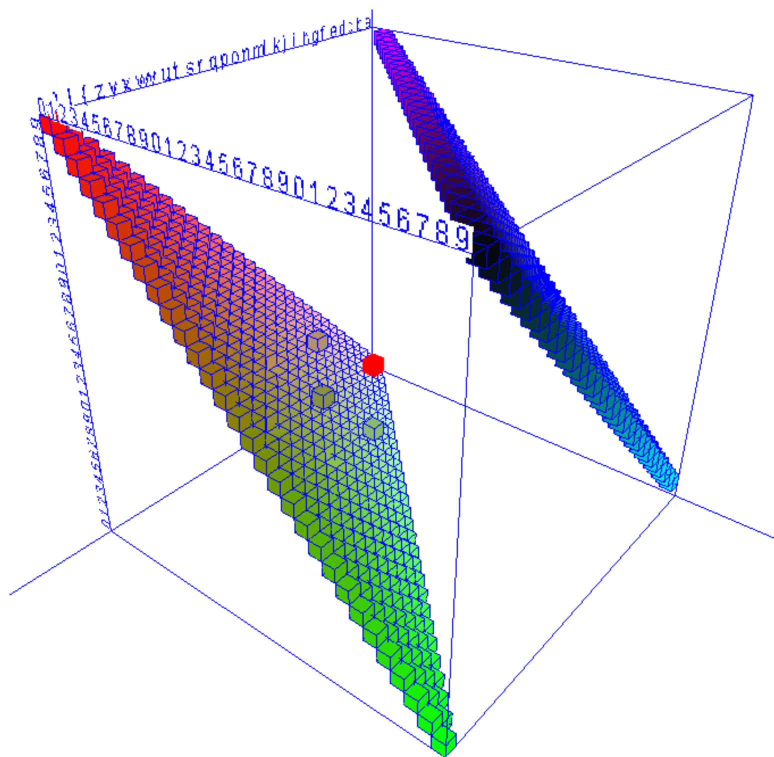
Se espera que el proyecto publicado sea de utilidad a personas que busquen esta clase de implementaciones en Internet, ya sea con fines comerciales, educativos, de entretenimiento o por mera curiosidad o interés.

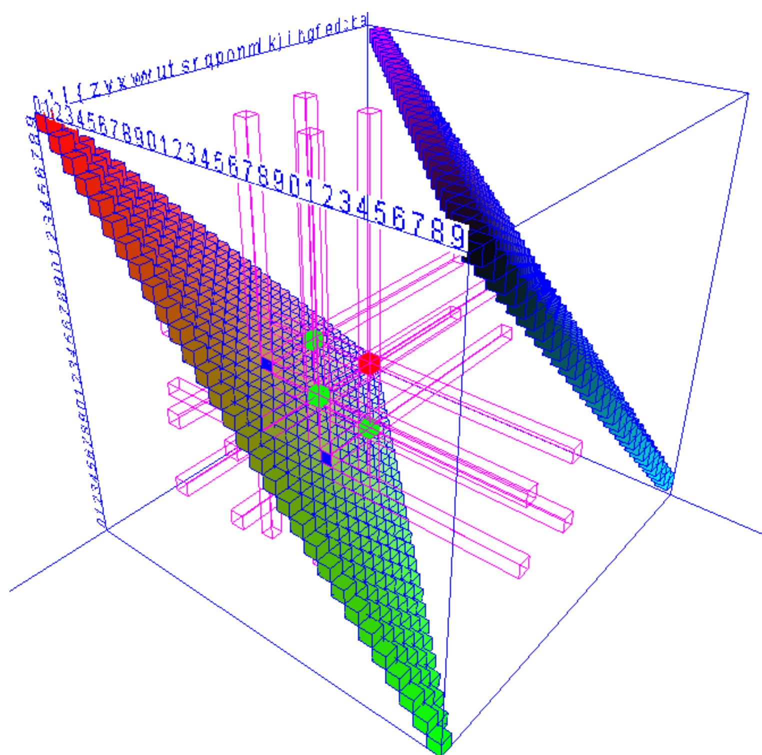
Apéndice 1: Gráficas en OpenGL

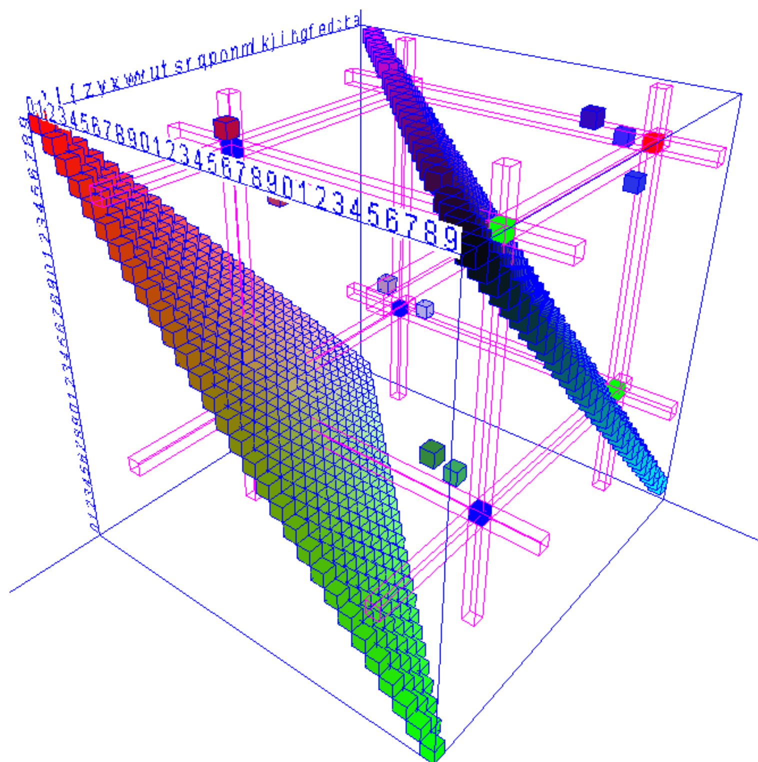
En este apéndice se muestra la generación paso a paso de un CL de orden 30.

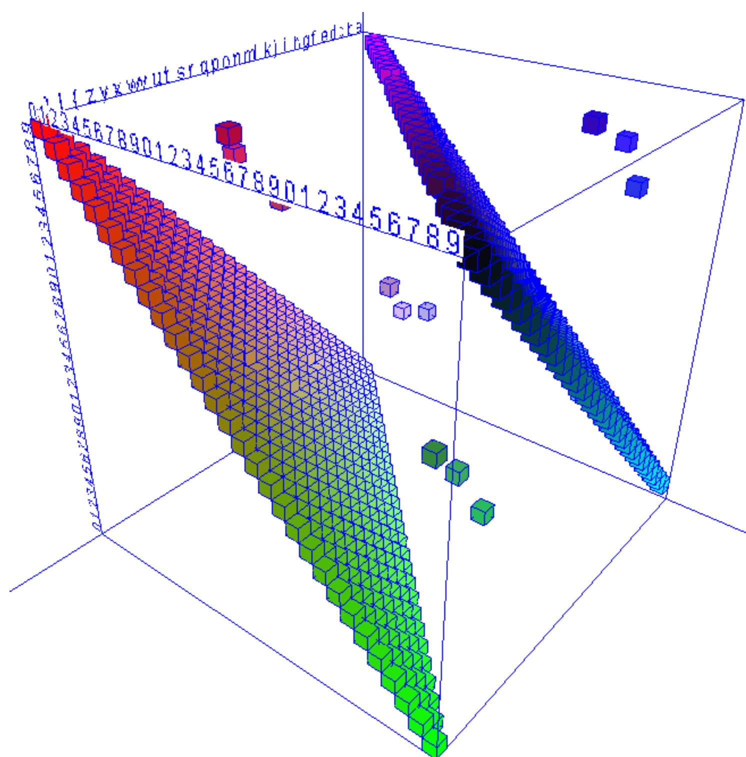


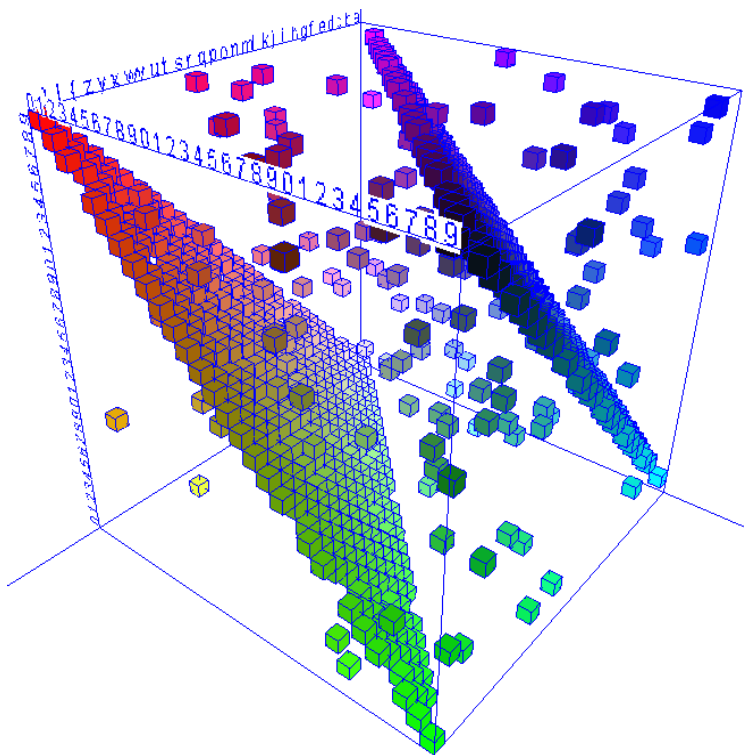


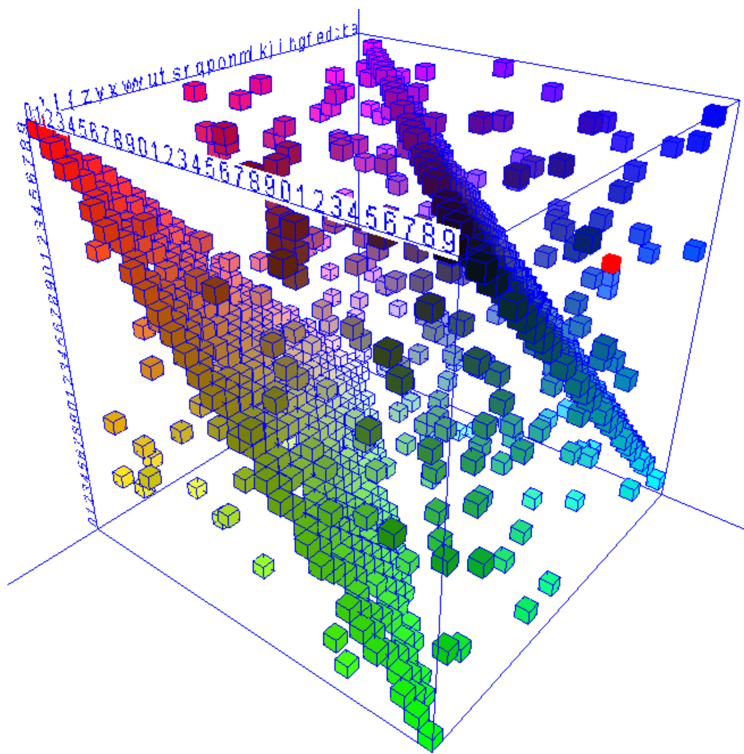


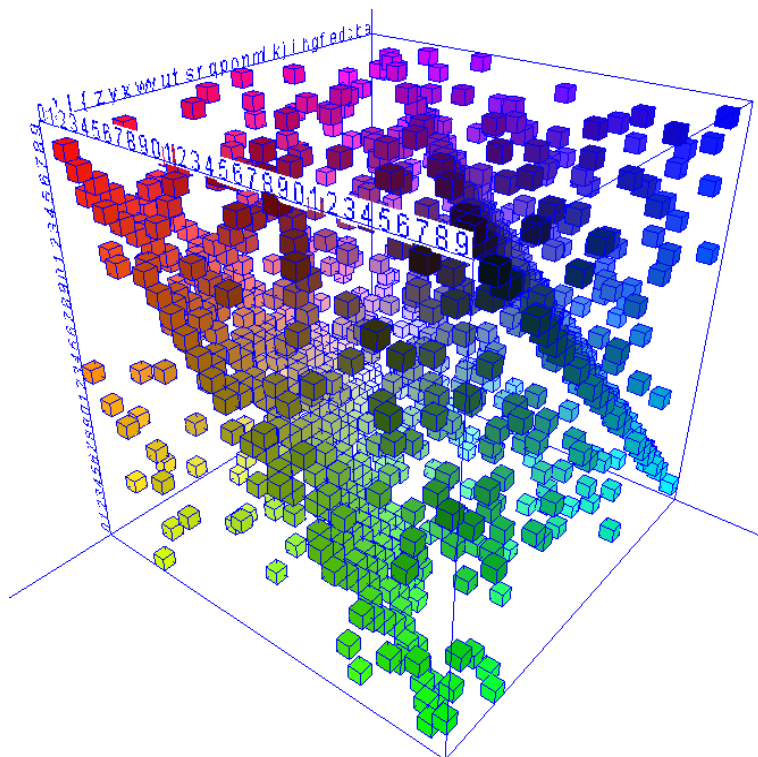


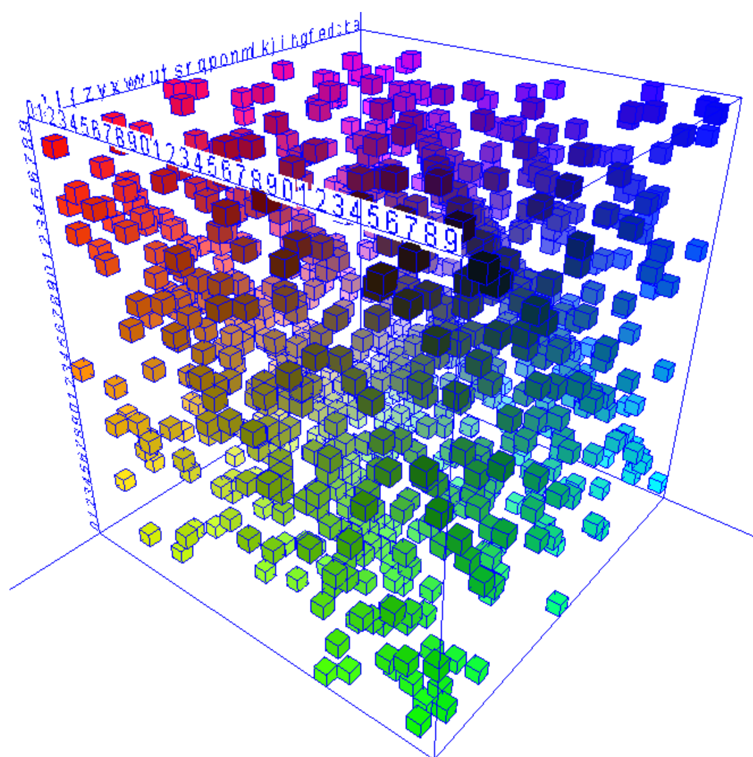


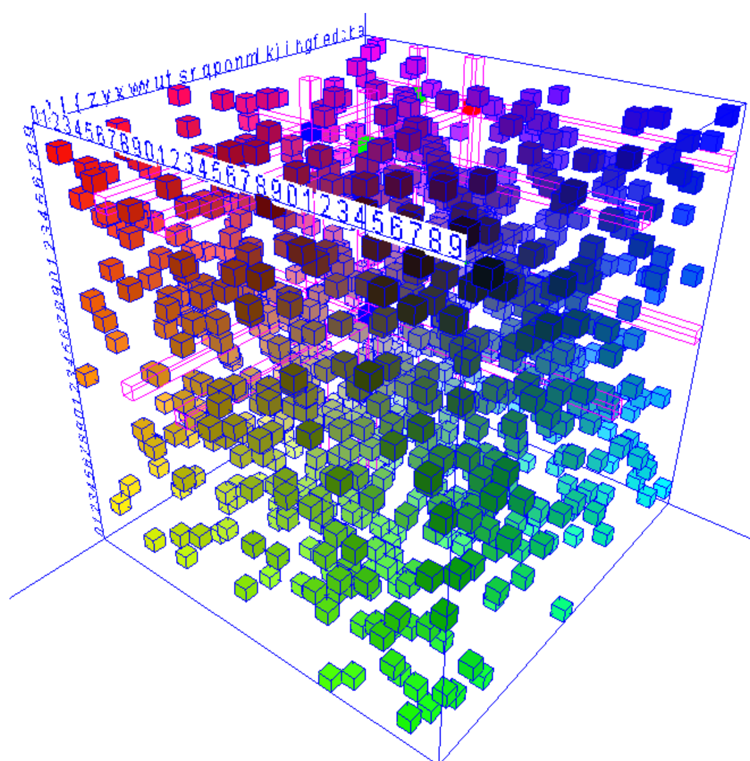












Agradecimientos

Quisiera agradecer a mi esposa, Alejandra, que me apoyó a lo largo de todo este proyecto. A mi madre, Marta Sagastume, por la revisión y ayuda en algunos capítulos. A Alberto Maltz por la ayuda con los tests estadísticos de uniformidad. A Claudia (mi directora) que me guió en todo el proceso de elaboración de la tesis y en las decisiones concernientes a los trabajos incluidos en la misma. A Ariel Ruiz Mateos, que me presentó el problema hace ya un par de años.

Chascomús, 21 de Noviembre de 2014.

Bibliografía

[AGKS00] Dimitris Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman, *Generating satisfiable problem instances*, In AAAI/IAAI, 2000, pp. 256–261.

[Bar04] R. Barták, *On generators of random quasigroup problems.*

[BP13] Matthew Battey and Abhishek Parakh, *An efficient quasigroup block cipher.*, Wireless Personal Communications **73** (2013), no. 1, 63–76.

[Bro13] J. Brown, *An evaluation of "generating satisfiable problem instances".*

[CP11] C. Cortés Pérez, *Propiedades y aplicaciones de los cuadrados latinos. tesis maestro en ciencias de matemáticas aplicadas e industriales*, Master's thesis, Universidad Autónoma Metropolitana, Iztapalapa, D.F., Méjico., Oct 2011.

[DM04] V. Dimitrova and J. Markovski, *On quasigroup pseudo random sequence generator*, Proc. of the 1-st Balkan Conference in Informatics, Thessaloniki, 2004, pp. 393–401.

[Fon13] R. Fontana, *Random Latin squares and Sudoku designs generation*, ArXiv e-prints (2013).

[FSK10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno, *Cryptography engineering - design principles and practical applications*, Wiley, 2010.

[Gib12] Steve Gibson, *Off the grid (online)*, `https://www.grc.com/offthegrid.htm`, May 2012.

[GO04] Guevara A. García O., *Introducción a la programación gráfica con opengl*, 2004.

[GS12] Ignacio Gallego Sagastume, *Generating random latin squares with python (unpublished)*, 2012.

[GS13] ———, *Un método para la generación de cuadrados latinos de orden 256*, Congreso Nacional de Ingeniería Informática y Sistemas de Información (CoNaIISI). UTN Regional Córdoba, Argentina. (2013).

[GS14] ______, *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).

[GS15] ______, *Proyecto igs-lsgp (latin square generation package) in github*, `https://github.com/bluemontag/igs-lsgp/wiki`, 2015.

[JLC11] Gareth J. Janacek and Mark Lemmon Close, *Mathematics for computer scientists*, Ventus Publishing ApS, 2011.

[JM96] 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)

[Kno12] Cut the Knot, *Prueba del teorema de hall*, `http://www.cut-the-knot.org/arithmetic/elegant.shtml`, May 2012.

[Knu81] Donald Ervin Knuth, *The art of computer programming. 2nd edition*, ADDISON-WESLEY PUBLISHING COMPANY, Addison-Wesley series in computer science and information processing, 1981.

[Kos95] Czeslaw Koscielny, *Spurious galois fields*, Int. J. Appl. Math. Comput. Sci. **5** (1995), no. 1, 169—-188.

[Kos02] ______, *Generating quasigroups for cryptographic applications*, Int. J. Appl. Math. Comput. Sci. **12** (2002), no. 4, 559–569.

[Kos04] ______, *Spurious multiplicative group of $\mathcal{GF}(p^m)$: a new tool for cryptography*, no. 12, 61–73.

[Kos14] ______, *Aplicaciones maple*, `http://www.maplesoft.com/applications/author.aspx?mid=16738`, September 2014.

[Mey10] Albert R. Meyer, *Mathematics for computer science*, Creative Commons, Massachusets Institute of Technology, 2010.

[MVJ10] F.P. Miller, A.F. Vandome, and M.B. John, *Hall's theorem*, VDM Verlag Dr. Mueller e.K., 2010.

[MW91] Brendan D. McKay and Nicholas C. Wormald, *Uniform generation of random Latin rectangles*, J. Combin. Math. Combin. Comput. **9** (1991), 179–186. MR 1111853 (92b:05013)

[MW05] Brendan D. Mckay and Ian M. Wanless, *On the number of latin squares*, Ann. Combin **9** (2005), 335–344.

[O'C63] F. O'Carroll, *A method of generating randomized latin squares*, Biometrics (1963).

[Org04] Design Theory Organization, *Latin squares definition and examples (online)*, `http://designtheory.org/library/encyc/latinsq/g/`, October 2004.

[Ran12] Random.org, *Random.org web site*, `http://www.random.org`, May 2012.

[Rit91] T. Ritter, *The efficient generation of cryptographic confusion sequences*, Cryptologia **15** (1991), no. 2, 81–139.

[Rit03] Terry Ritter, *Ciphers by ritter (online)*, `http://www.ciphersbyritter.com/RES/LATSQ.HTM`, 2003.

[Sch95] Bruce Schneier, *Applied cryptography (2nd ed.): Protocols, algorithms, and source code in c*, John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Str88] M. Strube, *A BASIC program for the generation of Latin squares*, Behavior Research Methods, Instruments, & Computers **20** (1988), no. 5, 508–509.

[SVT14] D. Selvi, G. Velammal, and ThevasahayamArockiadoss, *Modified method of generating randomized latin squares*, Journal of Computer Engineering (IOSR-JCE) **16** (2014), 76–80.

[vLW92] J.H. van Lint and R.M. Wilson, *A course in combinatorics*, Cambridge University Press, 1992.

[Wan04] Ian M. Wanless, *Cycle switches in Latin squares*, Graphs Combin. **20** (2004), no. 4, 545–570. MR 2108400 (2005i:05025)

[Wol12] Corporation Wolfram, *Wolfram web site*, `http://www.wolframalpha.com/`, May 2012.