Generación de números pseudo-aleatorios con autómatas celulares unidimensionales de radio 2

Random number generation with one-dimensional $r = 2, k = 2$ cellular automata

Nelson Enrique Castillo Izquierdo
emQbit Ltda, Colombia.
nelson@emqbit.com

Resumen

El objetivo de este trabajo es encontrar reglas para la generación de números pseudo-aleatorios con autómatas celulares unidimensionales de radio 2. En primera instancia, las reglas son obtenidas automáticamente con un algoritmo genético. Se hace un análisis estadístico preliminar del comportamiento de la función objetivo, y un análisis cualitativo de las reglas que con mayor probabilidad serán útiles para hacer posible posterior exploración exhaustiva del espacio de todos las posibles reglas (2^{32}). Se describe el uso de un cluster de estaciones de trabajo usado para ejecutar un algoritmo genético paralelo.

Abstract

An evolutionary method to find $r = 2, d = 1, k = 2$ cellular automata rules suitable for pseudo-random number generation is shown. A qualitative analysis and preliminary statistical analysis is performed to allow an exhaustive exploration of the relevant subset of all the rules in the considered cellular automata space (2^{32} rules) . A parallel genetic algorithm that ran in a cluster of workstations as part of this work is also described.

1. Introducción

1.1. Números Pseudo-Aleatorios

Los métodos prácticos para obtener números aleatorios suelen están basados en algoritmos determinísticos, y por esta razón, éstos números son llamados pseudo-aleatorios para distinguirlos de los números aleatorios considerados verdaderos, que aparecen en algunos fenómenos físicos naturales. Los números "elegidos al azar" son útiles en diferentes clases de aplicaciones, por ejemplo: simulación, muestreo, análisis numérico, programación de computadores, toma de decisiones, estética, recreación y criptografía.

En la práctica, no es conveniente hablar de un número aleatorio, por ejemplo: ¿es 2 un número aleatorio?. Por esta razón en lugar de hablar de un número aleatorio, se habla de una secuencia de números aleatorios independientes con una distribución determinada, y esto vagamente significa que cada número fue obtenido totalmente al azar y que no tiene que ver nada con los otros miembros de la secuencia, y que cada número tiene una probabilidad determinada de estar en cualquier parte dentro de un rango de valores dado.

Hay diferentes maneras de formular definiciones abstractas de aleatoriedad. El lector interesado puede consultar el libro The Art of Computer Programming, de Donald E. Knuth[4].

Para determinar que tan buenos son los números pseudo-aleatorios generados en este proyecto, se usan pruebas estadísticas de aleatoriedad utilizando programas disponibles sin costo en Internet que han sido usados en proyectos similares: –generación de secuencias pseudo-aleatorias con autómatas celulares–, realizados con anterioridad [7] [8] [2] [5]. El programa más importante en este caso es diehard (la versión en C es diehardc). Estos programas ya no son mantenidos y para futuros desarrollos es conveniente explorar el uso del programa Dieharder (*http://www.phy.duke.edu/ rgb/General/dieharder.php*), que es mantenido por Robert G. Brown, quien es conocido por sus aportes a la comunidad de los clusters Beowulf.

1.2. Autómatas celulares

Un autómata celular (AC) consiste en una retícula regular de celdas, cada una de las cuales puede estar en uno entre un número finito k de estados. Los autómatas considerados en este trabajo permiten dos posibles estados para cada celda ($k = 2$). Los estados se denotan 0 y 1. Estos autómatas también son llamados "binarios". Los estados de un autómata celular son actualizados síncronamente en pa-

sos discretos de tiempo, de acuerdo a una regla de interacción local idéntica para todas las celdas. Los autómatas celulares exhiben tres características notables: paralelismo masivo, interacción local de las celdas y estar formados por componentes básicos simples (celdas). El problema es diseñar el sistema paralelo de interacción loca –la regla– que exhiba un comportamiento específico o resuelva un problema en particular [6].

El primero en proponer formalmente el uso de autómatas celulares para la generación de números aleatorios fue Stephen Wolfram[10], en el año 1985. En este trabajo, para conseguir reglas apropiadas para la generación de números pseudo-aleatorios con los autómatas celulares unidimensionales de $r = 2$, $k = 2$, inicialmente se usó un algoritmo genético. Para esto se usó la librería GALib. El código fuente relevante se encuentra disponible en Internet, con licencia GPL, y se puede descargar del repositorio de subversion *http://svn.arhuaco.org/svn/src/carnd/trunk/*.

1.3. Algoritmos Genéticos

Un algoritmo genético (AG) es un procedimiento iterativo que se compone de una población constante de individuos, cada uno de los cuales se encuentra representado por una cadena finita de símbolos, conocida como el genoma, codificando una posible solución en el espacio de un problema. A éste espacio se le denomina espacio de búsqueda, y consiste en todas las posibles soluciones al problema con el que se está trabajando. Para diferenciar una solución –codificada en un individuo– de otra, se usa una función objetivo o función de aptitud que se encarga de informar al algoritmo qué tanto un individuo se ajusta al criterio de búsqueda, para que esta información sea utilizada en la optimización.

El algoritmo genético estándar procede de la siguiente forma: una población inicial de individuos es generada aleatoria o heurísticamente. En cada paso evolutivo, conocido como generación, los individuos en la población actual son decodificados y evaluados de acuerdo a algún criterio predefinido de calidad, que se conoce como fitness o función de aptitud. Para formar una nueva población (la siguiente generación), los individuos son seleccionados de acuerdo a su fitness. Muchos procedimientos de selección se usan actualmente, uno de los más simples es la selección proporcional al fitness, introducida originalmente por Holland, en donde los individuos son seleccionados con una probabilidad proporcional a su fitness relativo. Esto asegura que el número esperado de veces que un individuo es seleccionado es aproximadamente proporcional a su desempeño con respecto a la población. De esta forma los individuos con un valor alto en el fitness –los buenos– tienen mejores posibilidades para reproducirse, mientras que lo de fitness bajo tienden a desaparecer. La sola selección no puede introducir individuos nuevos en la población, es decir, no puede encontrar nuevos puntos en el espacio de búsqueda. Los nuevos individuos son generados por operadores inspirados en la genética, de los cuales los más conocidos son cruzamiento (crossover) y mutación.

En general, un algoritmo genético se aplica a espacios que son muy grandes para que se realicen búsquedas exhaustivas en ellos. El alfabeto de símbolos es usualmente binario, aunque otras representaciones han sido también usadas.

En una etapa inicial, y como parte de un proyecto final de carrera, se hizo una exploración del espacio de los autómatas celulares tridimensionales [1] (Disponible en Internet). Se obtuvieron resultados satisfactorios en tres dimensiones, pero la opinión del autor es que el resultado más importante fue la obtención de una función objetivo general, que pudo ser aplicada con éxito en un autómata celular unidimensional de radio dos –en el año 2005–. Es importante notar que la intención del autor inicialmente –en el año 2001– fue trabajar con autómatas unidimensionales, pero en las pruebas iniciales exploratorias no se consiguieron resultados prometedores. En el momento, se atribuyó el fracaso de las pruebas iniciales con autómatas celulares unidimensionales a la naturaleza unidimensional del autómata, pero en este trabajo se aporta evidencia que indica que lo que faltaba era contar con una función objetivo apropiada.

2. La función objetivo

Se usó la entropía como función de aptitud del algoritmo genético. También se usa la entropía como un valor a tener en cuenta en la exploración exhaustiva.

Un valor alto de entropía no es una condición suficiente –pero si necesaria– para asegurar que un autómata celular es bueno para la generación de números aleatorios. La función, por si sola, constituye un criterio apropiado para conducir el proceso evolutivo. Para determinar si uno de los autómatas celulares obtenidos es apropiado para la generación de números aleatorios, hay que aplicar diferentes pruebas de aleatoriedad.

2.1. Entropía

Se hace una descripción de la función de entropía, de acuerdo al uso que se le dio en los trabajos de Tomassini et. al. [7] [8].

Se tiene una secuencia de bits. Sea k el número de valores posibles para cada posición de la secuencia (En este caso, 2 posibles estados para cada celda del autómata celular) y h la longitud de una sub-secuencia. La entropía, E_h, medida en bits, para el conjunto de las k^h probabilidades de las k^h posibles sub-secuencias de longitud h está dada por:

$$E_h = \sum_{j=1}^{k^h} ph_j \log_2 ph_j$$

en donde $h_1, h2, \ldots$ son todas las posibles sub-secuencias de longitud h, (Por convención se asume $log_2 0 = 0$ al computar la entropía). E_h alcanza su máximo valor cuando las probabilidades de todas las k^h posibles sub-secuencias de longitud h son iguales a $\frac{1}{k^h}$. E_h alcanzaría su máximo valor (h) si cada posible sub-secuencia apareciera $\frac{N}{k^h}$ veces en la secuencia, en otras palabras, si las sub-secuencias de longitud k^h estuvieran uniformemente distribuidas en la secuencia de longitud N. Se recomienda consultar el código fuente para leer una implementación del cómputo de la entropía, especificada en lenguaje C.

Para medir la entropía se usó $h = 4$, siendo este es el máximo valor de fitness que una regla puede alcanzar (En los trabajos de Sipper y Tomassini se han usado valores de 4 y 7 para h).

2.2. La función objetivo más apropiada

Hasta el momento, las observaciones indican que que la función objetivo más apropiada para usar con un algoritmo genético en el problema en consideración, es medir la entropía mínima en M autómatas celulares de diferentes tamaños (N). En este trabajo, y en particular para el algoritmo genético, se ha usado $M = 3$, incluyendo un número par, un impar y un primo. En las pruebas se escogieron los números 255, 256 y 257. La escogencia de estos números se dio luego de experimentos simples, y sin estadísticas concluyentes. Parte de este trabajo es explorar más esta función objetivo para evaluar la posibilidad de usar autómatas celulares de menor tamaño sin sacrificar la capacidad de la función objetivo para diferenciar los autómatas celulares que interesan de los que no. Los que interesan son los que tiene alta entropía, condición necesaria para lograr buenos resultados en las pruebas estadísticas de aleatoriedad.

La idea de la función objetivo es buscar los autómatas celulares que pueden generar un comportamiento aleatorio a partir de un patrón simple (una grilla inicializada en ceros con una casilla con valor 1). A esta propiedad se le denomina en Inglés *autoplectic behavior* [9]. Sheng-Uei Guan et. al [5] usan una aproximación similar para incluir estos autómatas en sus pruebas, pero no se limitan a esta prueba, como si se hace en este trabajo. Note que usar una sola celda inicializada (en 1) es exagerado, y se puede definir un criterio menos estricto para buscar el *comportamiento autopléctico*.

Para iterar el autómata, se usa una vecindad de 5 celdas, que incluye la celda a iterar, y las dos vecinas de cada lado. Como "regla de transición" se usa el vector con el que se representa el genoma, siendo este el bit de salida de cada uno de los posibles estados de la tabla de transición. Así, si el vecindario considerado es 00001, se usará el segundo elemento del vector como el valor de la celda en el siguiente estado de tiempo.

Una vez inicializado el autómata con un solo 1, se itera 32 veces con la regla a evaluar. Luego se itera el autómata celular 1024 veces y se mide la entropía en cada una de las N celdas del autómata.

Se toma la entropía mínima de todas las celdas, como la entropía del autómata. Esta función objetivo fue útil en las pruebas preliminares con autómatas celulares tridimensionales[1], y se usó luego con buenos resultados en una dimensión.

Para dar la respuesta final, que es la entropía de la regla a considerar, se toma la entropía mínima de los M autómatas considerados.

3. Cluster de estaciones de trabajo

La función objetivo requiere cómputo intensivo, y un algoritmo genético puede ser paralelizado eficientemente en un cluster.

En la etapa preliminar en la que se consideró un autómata celular tridimensional, en el año 2001, se usó un Cluster de Estaciones de Trabajo con procesadores Intel Pentium II (400MHz) para el cómputo de la función objetivo del algoritmo genético. De las 21 estaciones de trabajo usadas, una se usó como servidor con el sistema operativo GNU/Linux (Distribución Debian 2.2r3). Las 20 estaciones restantes se usaron como nodos corriendo Linux con kernel 2.4.4 y un sistema de archivos basado en el sistema de archivos básico de Debian. Los nodos no tenían el sistema operativo Linux instalado en sus discos duros, sino que cargaban la kernel de un diskette. Al arrancar la kernel, esta montaba el sistema de archivos residente en el servidor a través de una red Ethernet, usando NFS. Esto permitió utilizar los 21 equipos de la red local sin modificar la configuración de los discos duros de todos los equipos (solo se modificó el del servidor).

En cuanto a la aplicación, se utilizó una versión modificada del ejemplo "pvmind" que se distribuye con GALib, el cual permite distribuir el computo del fitness del algoritmo genético usando el software PVM (La Parallel Virtual Machine es un software que permite que un grupo de computadoras agrupadas en una red y con el sistema operativo UNIX o Windows NT sean usadas como un único computador paralelo). Para obtener un beneficio real de esta implementación, es necesario que el tiempo empleado en computar el fitness sea menor que el tiempo requerido para transmitir un individuo por la red, el cual es el caso del fitness usado en este trabajo. En el año en el que se hicieron las pruebas ya se prefería usar MPI y no PVM, pero ya que el esqueleto del AG que se necesitaba estaba implementado

en muy pocas líneas de código, se decidió usar PVM.

El empleo del cluster permitió hacer 100 corridas del algoritmo genético en 13.3 horas. El uso de una sola de las estaciones hubiese requerido aproximadamente 187 horas de computo (7.8 días), debido a que cada corrida requiere aproximadamente 1.878 horas si se hace en una estación con las mismas características. Es importante notar que el usar 21 máquinas no indica que el computo se realiza 21 veces más rápido (de hecho, se realizó 14 veces más rápido), debido a los costos adicionales en los que se incurre al hacer el cómputo distribuido (transferencia en red y sincronización por ejemplo).

En las siguientes corridas del algoritmo genético, con autómatas celulares unidimensionales, en el año 2005, se hicieron corridas independientes en PCs con procesadores Pentium IV (3.0HGz). El cómputo de la función objetivo para el caso unidimensional es mucho menos exigente computacionalmente, y puede ser implementado de manera óptima en un PC con registros de 32 bits.

Sin duda, cómputo puede ser distribuido de nuevo con un cluster o en una grilla, y la idea es hacerlo, pero antes de eso, se propone una exploración exhaustiva del espacio de todos los autómatas celulares de radio dos, que hoy es posible en un tiempo razonable (especulando, pocos meses) si se dispone de un cluster ó una grilla. Las estadísticas calculadas en este proyecto buscan optimizar el cómputo de la función objetivo para hacer más eficiente este cálculo y proveer una cota superior para el tiempo que se necesitaría en un cluster en particular.

4. Pruebas estadísticas de aleatoriedad

En la práctica se usan pruebas estadísticas para determinar si un generador de números pseudo-aleatorios es apropiado o no, y para compararlo con otros generadores.

Si una secuencia generada pasa un numero de pruebas estadísticas cuantitativas de aleatoriedad entonces se puede afirmar que la secuencia es aleatoria, por lo menos para propósitos prácticos.

En este trabajo se utilizó el software Diehard, que fue usado en todos los trabajos anteriores referenciados. Este programa, escrito por George Marsaglia, provee un conjunto de tests para evaluar los generadores de números pseudo-aleatorios y es reconocido como uno de los más exhaustivos ([4], p75). La documentación –incluyendo la descripción de las pruebas–se distribuye con el software Diehard. Este software fue escogido entre los disponibles con el objetivo de poder realizar comparaciones con los trabajos previos en el área.

Las pruebas aplicadas con el programa Diehard retornan un p-value. De acuerdo al autor, un generador falla una prueba si el p-value está muy cerca de cero o muy cerca de uno, por seis o más decimales (se usó el intervalo de aceptación $[0{,}001, 0{,}999]$). Estos valores son obtenidos al hacer $p = F(X)$, en donde F es la distribución asumida (con frecuencia, distribución normal) para la variable aleatoria X de la muestra. Debido a que F es una aproximación asintótica de la distribución, no se debe interpretar que un p-value menor que $0{,}025$ o mayor que $0{,}975$ indica que se ha fallado la prueba con un nivel de confianza de $0{,}05$, como se podría hacer en estadística. Los p-values son un intento de normalizar los resultados obtenidos por las diferentes pruebas y deberían estar distribuidos uniformemente si los números proceden de una fuente aleatoria.

Para resumir los resultados, se utiliza un programa que analiza la salida del programa Diehard y entrega un valor numérico, entre 0 y 23, dependiendo del número de pruebas que se consideren aprobadas usando el criterio antes expuesto (Ver parse_diehardc.pl).

5. Reglas obtenidas con un algoritmo genético

Para mantener la discusión relevante al alcance de este documento, no se describen los resultados de la exploración preliminar con autómatas celulares tridimensionales que permitió conseguir la función objetivo apropiada. El lector interesado puede consultar el documento, que presenta resultados detallados de esta experiencia[1].

Se usó un algoritmo genético estándar, con una población de 100 individuos. El genoma es binario, de 32 bits. Los individuos son inicializados con probabilidad uniforme.

Se obtuvieron mejores resultados con 2 etapas, una de ellas busca la exploración inicial del espacio de búsqueda, y se usa una probabilidad de cruce del $100\,\%$, una probabilidad de mutación alta del $50\,\%$, durante 200 iteraciones. La segunta etapa, de 600 iteraciones, se hace con una probabilidad de cruce del $0\,\%$, y una de mutación del $0{,}1\,\%$. Se implementa un caché de la función objetivo con un hash, con el fin de evitar los cálculos repetidos que pueden resultar cuando la diversidad de los individuos del algoritmo genético disminuye (cuando convergen, sobre todo en la segunda etapa).

Para calcular la función objetivo de cada individuo, se usa la que ya ha sido descrita en este documento, que es la entropía mínima de tres autómatas, de tamaños 255, 256 y 257.

Al terminar el algoritmo genético, se obtiene una regla para ser evaluada. Con ella, se crea un autómata celular de 512 celdas. Las celdas del autómata son inicializadas con el generador de números del sistema, *rand(3)*.

En cada iteración se contatenan todos los bits del autómata y se escriben a un archivo. Se itera hasta que se logre tener un archivo de por lo menos $1024 * 1024 * 10$

bytes. Este es el archivo que se usa como entrada al programa diehard.

Con 200 corridas del algoritmo genético, se obtuvieron las siguentes reglas que pasaron las 19 pruebas básicas y las pruebas adicionales del del test diehard. Las reglas se representan convirtiendo el genoma, que es un vector de 32 bits, a su representación entera. Las reglas son: 1164622485, 1771656805, 1767287397, 2510924438, 2589284954, 1512416730, 1448711850, 2509875798, 2774111590, 2526455145, 1217967975, 2527484330, 2790893910, 2773854809, 1718195561, 2859111829, 1452632469, 1515612521, 2779339174, 2510662230, 1788438873, 2523240854, 1653841260, 2527499610, 1768265369, 1318433130, 2577033578, 2577033573, 2572839510, 2778028634, 1771395498, 1722459733, 2523555174, 2510973529, 1687722855, 1432005285 y 2526434661.

Se muestra la forma en la que se puede probar el desempeño de una de las reglas con el código del proyecto, en este caso, la regla 1722459733.

```
$ ./rng1 1722459733
$ ./diehardc out.rnd out.txt \
   1111111111111111
```

OVERLAPPING SUMS : 0.953179 : P
RUNS UP 1 : 0.601011 : P
RUNS DOWN 1 : 0.018514 : P
RUNS UP 2 : 0.294392 : P
RUNS DOWN 2 : 0.156919 : P
3DSPHERES : 0.456263 : P
PARKING LOT : 0.625643 : P
BIRTHDAY SPACINGS TEST : 0.209988 : P
BINARY RANK 6x8 MATRICES : 0.677364 : P
BINARY RANK 31x31 MATRICES : 0.760132 : P
BINARY RANK 32x32 MATRICES : 0.819011 : P
COUNT-THE-1's IN SUCCESSIVE BYTES 1 : 0.791849 : P
COUNT-THE-1's IN SUCCESSIVE BYTES 2 : 0.387364 : P
CRAPS No. OF WINS : 0.837590 : P
CRAPS THROWS/GAME : 0.689475 : P
MINIMUM DISTANCE : 0.713164 : P
OVERLAPPING PERMUTATION 1 : 0.007082 : P
OVERLAPPING PERMUTATION 2 : 0.618620 : P
SQUEEZE : 0.646737 : P

Passed : 19/19
BITSTREAM : 20/20
OPSO : 23/23
OQSO : 28/28
DNA : 31/31

Score=23.000
KS=0.621875

Las últimas líneas totalizan el resultado, mostrando que se pasaron con éxito todas las pruebas, y que la prueba Kolmogorov-Smirnov que se hizo con los p-values generados se acerca a una distribución aleatoria (cercano a 0.5).

Note que la escogencia de el número 512 como tamaño para el autómata celular podría ser muy grande para una implementación en hardware, y colateralmente, –con toda intención– mejora el resultado de las pruebas presentadas. Por ejemplo, la misma regla del ejemplo obtuvo un score de 15 y un Kolmogorov-Smirnov igual a 1 al usar $N = 50$.

Para medir con más objetividad los resultados obtenidos, es conveniente conocer más el espacio de búsqueda, que es tratable con los recursos computacionales actuales.

6. Hacia la exploración exhaustiva del espacio de búsqueda

Debido a los múltiples parámetros a tener en cuenta, y a su impacto en las estadísticas que más importan –cuántos tests aprueba un generador– se propone no usar un algoritmo genético para explorar este mismo espacio de búsqueda, sino realizar una exploración exhaustiva y cuidadosa para tener más conocimiento del comportamiento de la función objetivo. Estos resultados pueden ser de gran utilidad para usar un algoritmo genético en autómatas celulares más grandes, de radio 3, que tiene 2^{128} posibles reglas. En las pruebas preliminares [1] se obtuvieron buenos resultados aplicando un AG con un genoma de 128 bits (2^{128} estados), en el caso del AC tridimensional, por lo que este espacio de búsqueda es todavía apropiado para un AG.

La exploración del espacio de los autómatas celulares de radio 2 (2^{32}, inicialmente) permitirá comprender que tan abundantes son los buenos generadores en el espacio de búsqueda del algoritmo genético. Así, se podrá poner en contexto los resultados obtenidos, que parecen buenos con las pruebas, que indudablemente, están particularmente "optimizadas" para que los resultados se vean bien, lo que se hizo evidente al reducir el tamaño del autómata celular.

Antes de pensar en modificar todas las pruebas y aplicar una estrategia de prueba y error para obtener una función objetivo que de mejores resultados con autómatas celulares pequeños (consideremos pequeños los de menos de 64 celdas), es conveniente tratar de responder primero algunas preguntas importantes.

- ¿Cuáles son los mejores parámetros para una función objetivo?
- ¿cuál es el máximo valor que esta función puede obtener?
- ¿Es posible acelerar el cómputo de la función objetivo tomando en cuenta otros datos para medir en el AC?
- ¿Qué tan fuerte es la correlación de esta función objetivo y las pruebas de diehard para las reglas que tienen mejor desempeño?
- ¿Cuál es el menor tamaño que se puede usar en un AC para seguir obteniedo buenos resultados en las pruebas de pseudoaleatoriedad?
- ¿Cómo se pueden combinar las reglas obtenidas para formar mejores generadores?

No es fácil responder *todas* estas preguntas, y como primer paso para comenzar a responder preguntas, encontrar nuevas, y

descartar las que no conviene considerar, se propone un primer paso exploratorio, que es hacer un muestreo estadístico del espacio de búsqueda considerado.

Se presentan algunas heurísticas que se consideran para disminuir el espacio de búsqueda a considerar para las pruebas preliminares, y algunas que serán implementadas en una exploración posterior.

6.1. Evitar las reglas que cambian el fondo

En primer lugar, para estas estadísticas se considerarán sólo las reglas que conservan el fondo. Es decir, el vecindario 00000 indica que después de una iteración, el estado de la celda central será 0. Esto disminuye a la mitad (2^{31}) el espacio a considerar. Esta consideración es útil para hacer posible la siguiente heurística.

6.2. Sintonizar las reglas caóticas y evitar simetrías

Langton [3] propuso un parámetro λ, que permite "sintonizar" reglas caóticas de forma probabilística. De acuerdo al rango del parámetro, es posible obtener autómatas que corresponden a una de las 4 clasificaciones propuestas por Wolfram [11]. De las clases propuestas, la que nos interesa es la III, que es la que produce comportamientos caóticos.

El parámetro λ fácil de calcular. Se toma el número de combinaciones en la tabla de transición que llevan al estado 1, y se divide este número entre el número de vecindarios posibles. Los autómatas celulares de clase III se encuentran cercanos al valor $\lambda = 0{,}5$. Así que si se quieren buscar reglas aleatorias, conviene empezar por $0{,}5$ e ir disminuyendo el valor de lambda.

En la práctica, esto se logra cambiando el número de unos que se permiten en la regla. Así, si la tabla de transición tiene 32 estados, podemos comenzar a considerar todas las reglas con 16 unos inicialmente. Luego, todas las reglas con 15 unos. Luego, todas las reglas con 14 unos, hasta que ya se note que no se consiguen suficientes autómatas celulares apropiados.

Note que consideramos sólo el rango $0 <= \lambda <= 1/2$, porque todo AC con $\lambda > 0{,}5$ corresponde a otro AC con $\lambda < 0{,}5$, si se cambia la convención para celdas "vivas" de 1 a 0. De esta forma se evita considerar configuraciones simétricas. En el caso que tratamos ahora, falta decidir si se harán pruebas con 16 unos en la regla o no.

Lo bueno es que ya con estas dos heurísticas, podemos saber cuantos ACs debemos evaluar, para diferentes valores de lambda, para un tamaño de AC (N) en particular, dependiendo del número de unos presentes en la regla del AC. Se consideran las combinaciones (no permutaciones), teniendo en cuenta que hay 31 posibles casillas para ubicar los números considerados.

En este ejemplo, se hace ver que el total de ACs a evaluar cuando hay 15 de 31 unos es 265182525, un poco más de 265 millones de reglas. De acuerdo a los cómputos preliminares, para los que se hizo una estadística con 10 millones de reglas para estas configuraciones, esta cantidad de reglas es tratable, por lo que es viable hacer una exploración exhaustiva del espacio de búsqueda que nos interesa. Si variamos el número de unos en la regla de 8 a 16, debemos evaluar 1072799175, un poco más de mil millones de reglas,

Unos	*# reglas*
1	1
2	31
3	465
4	4495
5	31465
6	169911
7	736281
8	2629575
9	7888725
10	20160075
11	44352165
12	84672315
13	141120525
14	206253075
15	265182525
16	300540195

Cuadro 1. Cantidad de reglas a evaluar de acuerdo al número de unos en la regla

un espacio menor menor al tamaño total del espacio considerado (2^{32}).

6.3. Estadísticas preliminares

Para comparar el algoritmo genético con la exploración exhaustiva de reglas, se hizo un muestreo del espacio de búsqueda. Para esto, se tomó un entero sin signo de 32 bits, que cumplía la función de contador, inicializado en uno. Una regla para los autómatas tratados en este trabajo requiere 32 bits. El contador se incrementaba y en cada momento se contaba el número de bits en uno que se requerían para representar el entero correspondiente. La idea es poder listar en un orden fácil de reproducir las reglas que tienen B bits en 1. Así, sólo se consideraban los números que cumplen este último criterio.

Esta aproximación no usa almacenamiento externo, y es fácil de implementar. (Ver *http://svn.arhuaco.org/svn/src/carnd/trunk/src/check-many.c*).

Se evaluó la entropía de las primeras 10 millones de reglas generadas de esta forma, con diferentes tamaños de retícula, siendo relevantes en este análisis 255, 256 y 257, por ser los mismos valores empleados en la función objetivo.

Esto se hizo con valores de 13, 14 y 15 para B, debido a que esto hace que el parámetro λ sea cercano a 0.5, y de acuerdo a la teoría existente la probabilidad de sintonizar reglas caóticas es mayor cuando λ se acerca a $0{,}5$. Se evaluaron 30 millones de reglas, cada una de ellas con una 3 diferentes tamaños de retícula, para un total de 90 millones de evaluaciones.

De la misma forma que se hace con la función objetivo, se tomó la mínima entropía de las tres medidas. Los 10 millones de valores se ordenaron ascendentemente, y se tomaron los mejores 2000. Estas reglas se sometieron a las pruebas estadísticas de aleatoriedad antes descritas, y el resultado -inesperado- fue que

no se encontraron reglas que pasaran la mayoría de las pruebas. De hecho, las 10 mejores reglas de las 10 millones tuvieron un pobre desempeño en las pruebas estadísticas. A continuación se muestran las 10 reglas con valor más alto de entropía. La primera columna corresponde a la regla, la segunda al valor de entropía medido de la misma forma que se hace en la función objetivo del AG, y la tercera al número de pruebas estadísticas aprobadas por la regla.

```
151649768  3.99008 0
131134734  3.98875 0
125096222  3.9823 0
151645688  3.97471 0
104031674  3.9744 4
39306166  3.95035 0
14523004  3.94862 0
14141334  3.94722 0
39307158  3.94707 0
46647190  3.94703 1
```

Un problema de la metodología usada para este análisis es que la mayoría de unos en las reglas consideradas están en los bits menos significativos, y esta restricción no existe para el algoritmo genético. Por esto, es necesario realizar otra prueba en la que se consideren todas las reglas disponibles para los mismos valores de B, y no sólo las primeras 10 millones.
En cuanto al tiempo de ejecución, con $B = 15$, una grilla de 257 celdas, la evaluación de 10 millones de reglas tomó 17,15 Horas. Esta medida se tomó en un Intel Pentium 4 con una velocidad de reloj de 3.20GHz (3192,322 MHz) y memoria caché de 1MB. Evaluar todas las reglas que se obtienen con $B = 15$ tomaría aproximadamente 445.9 Horas en un PC similar, tiempo que se debe multiplicar por 3 para considerar los diferentes tamaños de la grilla, igual que en la función objetivo. Afortunadamente, este proceso es fácilmente paralelizable.

7. Ciclos y Secuencias

Los resultados actuales del trabajo pueden ser usados para construir generadores de secuencias pseudo-aleatorias en hardware. Una característica importante de los generadores de números pseudo-aleatorios es el período de los mismos. Es posible diseñar autómatas celulares con período muy grande, pero estos no son buenos para la generación de secuencias pseudo-aleatorias. [7]
Si se consideran todos los estados de un AC y las transiciones entre estados, se puede tratar el problema de hallar el máximo periodo en un AC como un problema de grafos y ciclos en grafos. Conseguir un generador apropiado para tareas simples se reduce a encontrar el ciclo más grande en un autómata, y tomar el valor de uno de los nodos en el ciclo como semilla para la generación.
Es posible hacer una implementación eficiente del algoritmo, ya que los ciclos en autómatas celulares deterministas cumplen dos propiedades particulares: cada nodo tiene un sólo sucesor, y existe una función eficiente que permite hallar el sucesor para un estado dado. No es necesario tener cada estado (el valor de un nodo en el grafo) en RAM porque los estados son ennumerables, van de 0 a 2^N - 1 (N es el tamaño de la grilla). Por esto fue posible escribir un programa que necesita sólo 2 bits por estado para encontrar todos los ciclos en un AC (Ver *http://svn.arhuaco.org/svn/src/cagraph/*). Un bit se utiliza para marcar un estado ha sido visitado, y el otro para indicar si el estado hace parte del ciclo que se está buscando. El ciclo más grande encontrado en las pruebas preliminares es de 1.323.700 estados, corresponde al AC con regla 1516874342, y una grilla de 31 celdas. Con esta información sabemos que se puede generar una secuencia de 1.323.700 números de 31 bits. El grafo tiene 2.147.483.648 estados, y se requieren 512MB de RAM para realizar este cómputo. Se muestra un ejemplo de la ejecución del programa:

```
$  ./cafindcycles 1516874342 31
rule 1516874342
N=31
Memory = 536870912B 512MB
1
1323700
15593
124
31
124
1147
12214
6975
31
294192
(salida omitida)
```

En las pruebas iniciales no se espera al que programa termine, ya que se ha observado empíricamente que la probabilidad de encontrar el ciclo más grande disminuye a medida que se encuentran ciclos (no se ha examinado el fundamento teórico de esta observación, que muy probablemente ya está documentada).
Para la generación de secuencias usando este método existen dos mejoras posibles. La primera, es usar un contador de iteraciones para hacer un cambio de regla cuando se complete un ciclo, para así generar una secuencia más grande. La segunda es no considerar sólo el ciclo, sino el camino más largo que conduzca al ciclo. El algoritmo para la segunda mejora no ha sido explorado todavía en este trabajo.
Este problema de encontrar el ciclo más grande no parece fácilmente paralelizable, ya que si se usa más de un procesador habría que utilizar exclusión mutua para actualizar los datos que se necesitan para el cómputo, lo que iría en detrimento del desempeño. Esta paralelización no se ha considerado hasta el momento y no se ha probado en este trabajo. Por otro lado, el algoritmo no es nada apropiado para beneficiarse del uso del caché del procesador, ya que requiere uso de gran cantidad de RAM que es accedida de forma pseudo-aleatoria (¡a propósito!), y no se aprovecha el caché como sí lo hacen los problemas que acceden a la memoria con patrones menos caóticos.

8. Conclusión

En este trabajo se presenta una función objetivo nueva que permite encontrar autómatas celulares unidimensionales binarios de $r = 2$ apropiados para la generación de números pseudo-aleatorios de calidad. Con el análisis estadístico hecho al espacio de búsqueda del algoritmo genético, se logró confrontar los

mejores resultados disponibles en el espacio de búsqueda para contextualizar el desempeño del algoritmo genético, y además, se obtiene el conocimiento necesario para comenzar a explorar autómatas celulares con reglas más grandes. El siguiente paso lógico es explorar el espacio de los autómatas celulares de $r = 3$.

9. Trabajo Futuro

Como parte del trabajo futuro se ha considerado evaluar la literatura existente para intentar construir un generador de números aleatorios, combinando las mejores reglas obtenidas, obteniendo una cota inferior para el tamaño del período del generador. Es importante tener esta garantía para aplicaciones prácticas.

Actualmente en emQbit hemos desarrollado una board que tiene un procesador y un FPGA (ECB_AT91 V2). Ahí se pueden programar autómatas celulares como los que se describen en este artículo, explotando el paralelismo implícito que proveen los FPGAs. Se hará una implementación de prueba para evaluar las posibilidades y la aceleración que se puede obtener con una implementación razonablemente eficiente.

10. Agradecimientos

Para comenzar, este trabajo no habría sido posible sin la contínua ayuda de la comunidad del grupo *comp.theory.cell-automata* de USENET. Las discusiones con Idanis Díaz fueron muy importantes para el desarrollo de la etapa inicial de este trabajo. Rosario Madera, profesora de la Universidad del Magdalena, fue muy amigable al prestar asesoría estadística y lógica en la etapa inicial de este proyecto. El profesor Samuel Prieto brindó apoyo logístico. Se usaron recursos de la Universidad del Magdalena, especialmente computadores personales, para armar un cluster para cómputo intensivo durante las noches. Christian Trefftz –quien prefiere que no lo llame doctor ni profesor–, fue especialmente influyente al ser una guía permanente en los primeros años al ofrecerme tiempo de cómputo en máquinas a las que tenía acceso y al orientarme en las etapas iniciales, con el trabajo con clusters. Jaime Abella y Johnny Gomez prestaron apoyo logístico en la etapa de pruebas del cluster. Andrés Calderón brindó asesoría técnica para el desarrollo del proyecto. Por último, un agradecimiento a los autores de los programas (Octave, PVM, GALib, GCC, Perl, Autoconf, Python, GNUPLOT, etc) y sistema operativo (GNU/Linux) utilizados en este trabajo, por publicar sus códigos fuentes, en la mayoría de los casos con licencias permisivas.

11. Información del Autor

Nelson Castillo es ingeniero de sistemas, egresado de la Universidad del Magdalena. Es socio fundador de la empresa emQbit, en la que actualmente trabaja tiempo completo. Sus intereses incluyen la programación en paralelo, protocolos de red, optimización, aprendizaje de máquina, Linux embebido y lenguajes funcionales.

Referencias

[1] Castillo-Izquierdo, Nelson. *Generación de Números Pseudo-aleatorios con Autómatas Celulares Tridimensionales*. 54p. Trabajo de grado. 2001. Disponible en la dirección *http://svn.arhuaco.org/svn/bin/trabajo_grado.tar.gz*.

[2] Guan, Sheng-Uei; Zhang, Shu. *Incremental Evolution of Cellular Automata for Random Number Generation*. International Journal of Modern Physics C, Volume 14, Issue 07, pp. 881-896 (2003).

[3] Gutowitz H., Langton C. *Methods for Designing 'Interesting' Cellular automata*, CNLS News Letter ,1988.

[4] Knuth, Donald E. *The Art of Computer Programming volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 3rd edition, 1998. 762 p.

[5] Sheng-Uei Guan; Tan S, K. *Pseudorandom number generation with self-programmable cellular automata*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume: 23, Issue: 7. pp. 1095-1101. July 2004.

[6] Sipper M. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer-Verlag, Heidelberg, 1997. 199 p.

[7] Tomassini M., Sipper M., Zolla M., Perrenoud M, *Generating high-quality random numbers in parallel by cellular automata*, Future Generation Computer Systems 16, pp. 291-205, 1999.

[8] Tomassini M., Sipper M., Perrenoud M., *On the generation of high-quality random numbers by two-dimensional cellular automata*, IEEE Transactions on Computers, vol. 49, no. 10, pp. 1146-1151, 2000.

[9] Wolfram S. *Origins of Randomness in Physical Systems*. Physical Review Letters 55 pp. 449-452. 1985.

[10] Wolfram S. *Random Sequence Generation by Cellular Automata*. Advances in Applied Mathematics, 7 pp. 123-169. 1986.

[11] Wolfram S. *Universality and Complexity in Cellular Automata*. Physica D, 10. pp. 1-35. 1984. s