Algoritmos genéticos para el aprendizaje de autómatas celulares

María Castro Bonilla
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
marcasbon@alum.us.es, mariacastrobonilla@gmail.com

Alejandro Ruiz Jurado
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
aleruijur@alum.us.es, aleruizjurado@gmail.com

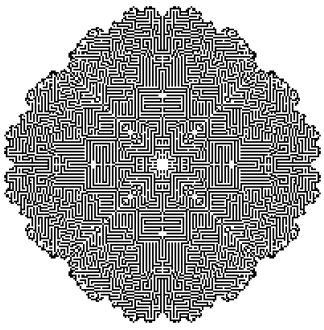
***Resumen*—Los autómatas celulares son modelos de computación con un gran potencial definidos por una serie de reglas que indican cómo evoluciona un estado según sus vecinos. Para obtener una regla útil para determinados problemas, en este trabajo se propone el uso de algoritmos genéticos para conseguir un autómata celular capaz de ofrecer buenas soluciones.**

Tras la realización de este trabajo, nos hemos percatado de la gran utilidad de estos conceptos y nos ha fascinado toda la investigación que hay su alrededor.

***Palabras clave*—Autómata celular, algoritmo genético, regla de transición, aprendizaje automático, configuración inicial, configuración objetivo...**

I. Introducción

Un autómata celular es un *modelo matemático* para un sistema dinámico compuesto por un conjunto de celdas o células que adquieren distintos estados o valores cuyos estados internos varían con el tiempo, teniendo tanto los posibles estados en los que pueden encontrarse las células como el propio tiempo naturaleza discreta. De esta manera este conjunto de células logran una evolución según una determinada expresión matemática, que es sensible a los estados de las células vecinas, y que se conoce como regla de transición local, la cual explica que cada célula está asociada a otras que conforman su vecindad, de modo que el estado de una de ellas en el instante n + 1 depende de los estados en los que estaban sus vecinas justo en el instante anterior n.



Los algoritmos genéticos es un modelo de búsqueda basado en la probabilidad que converge al elemento más óptimo. Basándose en fenómenos como la mutación, el cruce y el elitismo sabe conducir a una población de cromosomas, que no son más que listas de 0s y 1s, hasta un objetivo.

En este trabajo el principal objetivo será encontrar la regla más óptima para los ejercicios propuestos obteniéndolas a partir de un algoritmo genético.

Este trabajo esta estructurado en distintas secciones, en la primera de ellas, llamada Preliminares detallamos los autómatas celulares, las reglas y los algoritmos genéticos, siguiendo por la metodología donde la dedicamos a la descripción del método implementado en el trabajo. A continuación, se tienen los resultados, donde se detallarán los experimentos realizados y resultado de los mismos. Por último tenemos las conclusiones, dedicada a las reflexiones y aprendizajes obtenidos del trabajo, todo ellos acompañado del último apartado, referencias, el cual es una bibliografía de la información utilizada.

II. Preliminares

En esta sección se detalla el concepto de autómata celular y su funcionamiento seguido de la técnica empleada para obtener sus reglas, que en este caso son los algoritmos genéticos. Además, veremos los distintos trabajos que hemos encontrado relacionados con el que se nos solicita hacer.

A. Métodos empleados

1) Autómatas Celulares: Los *autómatas celulares* (AC) son sistemas descubiertos dentro del campo de la física computacional por John von Neumann en la década de 1950, el cual intentaba modelar una máquina que fuera capaz de autoreplicarse, llegando así a un modelo matemático de dicha maquina con reglas complicadas sobre una red rectangular. Inicialmente fueron interpretados como conjunto de células que crecían, se reproducían y morían a medida que pasaba el tiempo. Su nombre se debe a esta similitud con el crecimiento de las células. [5]

Muchos sistemas biológicos naturales, desde poblaciones de organismos al sistema respiratorio, están formados por la asociación de infinitos elementos homogéneos y simples que funcionan en paralelo, sin ningún tipo de control central y con un mecanismo de comunicación limitado, sin embargo, son capaces de exhibir una conducta extremadamente compleja que parece brotar de las intercomunicación entre los mismos. Los autómatas celulares comparten muchas de estas características con los sistemas biológicos: un gran número

de componentes homogéneos, extendidos en el espacio, sin control central y con un mecanismo limitado de comunicación entre los elementos.

Puesto que el cerebro es también un conjunto de células procesadoras o neuronas interconectadas, pronto surgió la idea de si estos formalismos serían capaces de servir como modelos formales para el estudio de la actividad cerebral. McCulloch y Pitts demostraron que siempre se podría diseñar un circuito compuesto por ciertas neuronas formales capaces de emular cualquier procesador digital binario, La neurona de McCulloch-Pitts es una unidad de cálculo que intenta modelar el comportamiento de una neurona "natural", similares a las que constituyen del cerebro humano. Ella es la unidad esencial con la cual se construye una red neuronal artificial [8]. Así nacieron las primeras Redes Neuronales Artificiales.

Los *autómatas celulares* constituyen un modelo matemático y computacional, de interacción extremadamente simple, discreto en el espacio y en el tiempo, que emula el comportamiento de muchos sistemas naturales. Además, otros sistemas dinámicos de extremada complejidad pueden ser modelados mediante autómatas celulares como el caso de corrientes circulatorias de tráfico o sistemas que exhiben autoorganización. Ciertos autómatas celulares son universales, es decir son capaces de representar cualquier algoritmo. Estos son maquinas abstractas capaces de construir nuevos autómatas que a su vez pueden generar otros. En otras palabras, son capaces de procesar cualquier cosa computable.

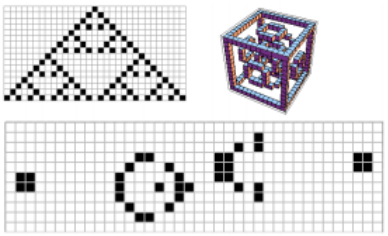


Fig. 1. Regla 90 de Wolfram (arriba izq.), AC en 3 dimensiones (arriba derecha) y Game of life de John Conway (abajo). [5]

El aspecto que más caracteriza a los AC es su capacidad de lograr una serie de propiedades que surgen de la propia dinámica local a través del paso del tiempo y no desde un inicio, aplicándose a todo el sistema en general. Por lo tanto no es fácil analizar las propiedades globales de un AC desde su comienzo, complejo por naturaleza, a no ser por vía de la simulación, partiendo de un estado o configuración inicial de células y cambiando en cada instante los estados de todas ellas de forma síncrona.

Los autómatas celulares pueden ser considerados como un tipo especial de máquinas de Turing, sin memoria interna, siendo algunos modelos capaces de llevar a cabo cualquier tipo de cálculo, es decir, computación universal. Un autómata celular está formado por elementos simples o células, que cambian de estado como consecuencia de un conjunto de reglas preestablecidas, a intervalos de tiempo discretos. A cada ciclo de tiempo, el estado de una célula depende del estado de las células que la rodean y cambia, por lo tanto, de la manera prescrita mediante la tabla de reglas que rige el comportamiento del autómata, conformando, así, su diagrama de transición de estados.

Un autómata celular bidimensional está constituido por un tablero o una rejilla, siendo cada una de las casillas una célula que puede adoptar uno de entre k estados, dependiendo de la configuración del entorno en el que se encuentra, es decir, de su estado actual y del estado de cierto número v de casillas o células vecinas que se denominan vecindad del autómata celular.

Se han estudiado 2 tipos de vecindad en los autómatas celulares bidimensionales (Se puede ver en Fig. 2):

- Vecindad de Moore: se consideran vecinos de una célula las 8 células que la rodean.
- Vecindad de Von Neumann: se consideran vecinos de una célula las 4 células que tocan ortogonalmente a la celda que se estudia. Por lo tanto, no se incluyen las diagonales.

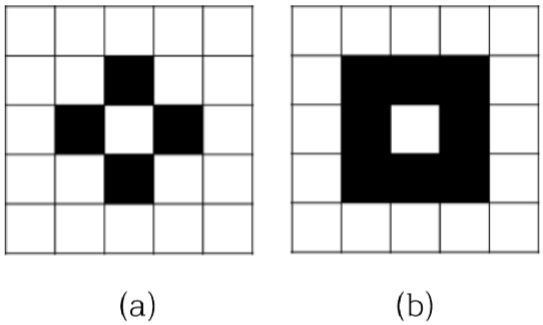


Fig. 2. Tipos de vecindad en un autómata celular bidimensional con r = 1: Vecindad de Von Neumann (a) y Vecindad de Moore (b).

Se ha demostrado que en general la vecindad de Moore obtiene mejores resultados y permite operaciones más complejas, aunque la de Von Neumann al tratar con menos vecinos es más ligera.

Por otra parte, el AC más simple es el unidimensional, se trata de una secuencia lineal de casillas con un radio de vecindad r = 1 (también llamado autómata celular elemental), está compuesto de células que dependen en cada momento del estado de sus dos células vecinas. Si se toma el radio r = 2, el estado de cada célula dependerá del estado de sus cuatro células vecinas, dos en un sentido y dos en el otro sentido.

Fig. 3. Ejemplo de autómata celular con r = 1 y r = 2. [5]

Los estados de un AC son alterados de un instante a otro en unidades de tiempo discreto, es decir, que se puede cuantificar con valores enteros a intervalos regulares. De esta manera este conjunto de células logran una evolución según

una determinada expresión matemática, que es sensible a los estados de las células vecinas, la cual se le conoce como *regla de transición local.*

En todo autómata celular existe una tabla de reglas que define el comportamiento del mismo a lo largo del tiempo, dependiendo de las dimensiones, el radio de vecindad y el número de estados.

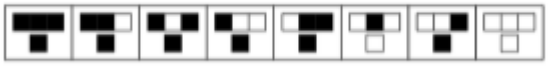


Fig. 4. Esta es la regla 250 del autómata celular [7]

Se pueden representar por colores como Fig. 4, siendo negro el valor 1 y blanco el valor 0 ó usando directamente 0s y 1s. Para un autómata celular unidimensional con r = 1 y número de estados k = 2 un ejemplo de tablas posibles sería:

000	001	010	011	100	101	110	111
0	1	1	1	0	0	1	0

Fig. 5. Tabla de un regla de ejemplo para un autómata unidimensional.

Estudiemos como ejemplo la regla de la Fig. 5: la célula central cambia de estado dependiendo de las situadas a su derecha e izquierda, las vecinas. Analizando esta tabla de reglas es posible derivar que una célula cambia de estado bajo las siguientes condiciones:

1) Cuando una célula se encuentra inactiva, se activa en el siguiente ciclo de tiempo si la de su derecha esta activa y la de su izquierda no lo está.
2) Cuando una célula de encuentra activa, se desactiva en el siguiente ciclo de tiempo si ambas células adyacentes se encuentran activas.
3) En cualquier otra condición una célula conserva su estado.

Cada regla de un autómata elemental se codifica por su representación decimal de la secuencia de bits del próximo estado. Por lo tanto, existen 256 reglas elementales. Si se incrementa el r ó el número de estados (3 colores, por ejemplo) este número aumenta considerablemente.

A continuación, se definen los elementos que conforman un autómata celular [5]:

- Un espacio regular.
  Ya sea una línea, un plano de 2 dimensiones o un espacio n-dimensional. Cada división homogénea del espacio es llamada célula.
- Conjunto de Estados.
  Es finito y cada elemento o célula del espacio toma un valor de este conjunto de estados. También se denomina alfabeto. Puede ser expresado en valores o colores.
- Configuración Inicial.
  Es la asignación inicial de un estado a cada una de las células del espacio.
- Vecindades.
  Define el conjunto de células que se consideran adyacentes a una dada, así como la posición relativa respecto a ella. Cuando el espacio es uniforme, la vecindad de cada célula es isomorfa (es decir, que tiene el mismo aspecto).
- Función de Transición Local.
  Es la regla de evolución que determina el comportamiento del autómata. Se calcula a partir del estado de la célula y su vecindad. Define cómo debe cambiar de estado cada célula dependiendo su estado anterior y de los estados anteriores de su vecindad. Suele darse como una expresión algebraica o un grupo de ecuaciones.

El aspecto que más caracteriza a los AC es su capacidad de lograr una serie de propiedades que surgen de la propia dinámica local a través del paso del tiempo y no desde un inicio, aplicándose a todo el sistema en general. Por lo tanto no es fácil analizar las propiedades globales de un AC desde su comienzo, complejo por naturaleza, a no ser por vía de la simulación, partiendo de un estado o configuración inicial de células y cambiando en cada instante los estados de todas ellas de forma síncrona.

2) Algoritmos Genéticos: Los algoritmos genéticos son métodos basados en la rama de la biología dedicada a la genética y en la evolución darwiniana. Fueron creados por John Holland en la década de los 60's y desarrollados junto a sus alumnos de la universidad de Mitchigan [1]. El algoritmo se basa en ver cómo evoluciona una población de cromosomas (arrays de 0s y 1s) a partir de una especie de selección natural y conceptos genéticos como la mutación y el cruce. Cada cromosoma está formado por varios genes (un bit) y la población va avanzando en cada iteración donde pueden ocurrir varios fenómenos para alterar el valor de cada gen.

El algoritmo de forma simplificada funciona de la siguiente manera:

1) Se genera de forma aleatoria una población inicial de un tamaño dado.
2) Se calcula el valor de cada cromosoma a través de la función de fitness.
3) Se seleccionan un par de cromosomas como padres usando el valor anterior como una probabilidad. Si se tiene un mejor valor, es más probable que sea un padre en esta etapa.
4) Se crean un par de nuevos cromosomas a partir de los padres anteriormente seleccionados. Pueden ser el resultado de un cruce entre sus padres, combinando sus genes de distintas formas (se detallarán más adelante) ó simplemente una copia de los padres.
5) Se mutan los nuevos cromosomas con una probabilidad de mutación dada como parámetro del algoritmo y se añaden a la nueva población.
6) Se vuelve al paso 2 usando la nueva población y se repite este proceso un número definido de veces.

La versión más usada suele incluir un detalle que mejora notablemente la eficacia del algoritmo, pasándose a llamarse algoritmos genéticos elitistas. El elitismo se basa en quedarse

con los mejores cromosomas de la población y pasarlos a la siguiente generación sin modificarlos.Además, estos serán los que se combinen para crear el resto de los integrantes de la nueva población. El número de cromosomas que se escogen viene dado por el ratio de élite.

A continuación, veremos los distintos parámetros que tiene el algoritmo escogido (en este caso usaremos el de la biblioteca geniticalgorithm [6]) y cuál es su significado:

- function: indica la función objetivo (fitness) que el algoritmo genético busca minimizar. Para cada cromosoma calculará su valor a través de esta función para luego ordenarlos por este valor y reemplazar los menos útiles por nuevos cromosomas.
- dimension: aquí se especifica la longitud de los cromosomas que se usaran en el algoritmo.
- variable_type: este será el tipo de valor que puede tomar cada gen, para los autómatas unidimensionales será 'bool'.
- variable_boundaries: este parámetro limita los valores que puede tomar el tipo antes especificado por medio de un rango de valores. Para el caso de un autómata unidimensional, debido a que se usa una variable booleana, no hay que especificar este parámetro.

Los siguientes parámetros viene en un diccionario llamado algorithm_parameters donde se indican los parámetros que son internos del algoritmo genético. Estos son:

- max_num_iterations: indica el número máximo de iteraciones que el algoritmo ejecutará. Se puede dejar por defecto y se calcula automáticamente pero no es recomendable.
- population_size: el número de cromosomas que tendrá la población inicial, por defecto es 100.
- mutation_probability: la mutación es un fenómeno que hace que un gen de forma aleatoria varíe de valor. Este parámetro indica la probabilidad de que esto ocurra, por defecto a 0.1 (10%)
- elit_ration: el elitsmo en un algoritmo genético es el concepto que dice que una porción de los cromosomas que obtengan un mejor valor de fitness serán clasificados como élite y serán conservados hasta el final. Con este parámetro definimos el porcentaje de élite, 0.01 por defecto. Si lo dejamos a 0 el algoritmo genético será uno estándar en lugar de elitista.
- crossover_probability: indica la probabilidad de cruce, 0.5 por defecto. Esto significa la probabilidad que un cromosoma tiene de ser cruzado con otro cromosoma para formar uno nuevo en la siguiente generación.
- parent_portion: este valor determina el porcentaje de cromosomas que pasarán a la siguiente generación sin modificaciones. Por defecto es 0.3.
- crossover_type: podrá tomar los valores one_point, two_point ó uniform, esta última es la que viene por defecto. Esto especifica cómo se realizará el cruce entre cromosomas. Si es one_point, se tomará un punto al azar para crear un nuevo cromosoma a partir de una porción de cada padre. Por otra parte, si es two_point se dividirán en tres partes para luego crear el nuevo cromosoma. Por último, el uniform escoge aleatoriamente cada gen de un padre o de otro. Ver Fig. 6 para una definición gráfica.
- max_iteration_without_improv: si se especifica este parámetro, indica las iteraciones que hará antes de pararse si el valor del mejor fitness sigue siendo el mismo. Por defecto no hay límite.

Hay más parámetros como function_timeout o variable_type_mixed que no especificaremos debido a que no será necesario su uso para los problemas que nos competen.

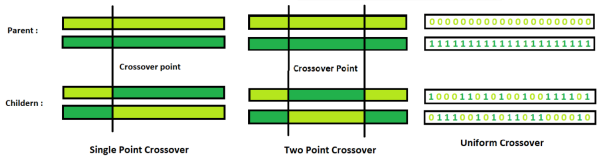


Fig. 6. Tipos de cruce en un algoritmo genético [2]

B. Trabajo Relacionado

Hemos encontrado varios trabajos relacionados con algoritmos genéticos y autómatas celulares de los cuales hemos leído en detenimiento para realizar nuestro trabajo. El primero es un proyecto muy similar al cual nos piden desarrollar llamado *Evolución de AC por AG* realizado por dos estudiantes de la Universidad de Deusto, Bilbao [3]. Este nos ha sido de gran utilidad para comprender mejor los objetivos del trabajo y sobre todo nos ha ayudado a relacionar los autómatas celulares con los algoritmos genéticos.

El segundo trabajo encontrado se titula *Evolving Transition Rules for Multi Dimensional Cellular Automata* y sus autores son un estudiante neerlandés y otro alemán [4]. Estos tratan los autómatas bidimensionales y nos han servido para el segundo problema propuesto, pudiendo entender mejor como tratar este tipo de autómatas con los algoritmos genéticos.

El trabajo de David Alejandro Reyes titulado *Descripción y Aplicaciones de los Autómatas Celulares* [5] ha complementado nuestra información y sobre todo nos ha aportado buenas imágenes para nuestro proyecto.

III. Metodología

En esta sección se detallarán los distintos problemas realizados y cómo se han enfocado para su resolución, explicando la utilidad de cada función desarrollada en todo momento.

Para usar los algoritmos genéticos nos hemos decantado por la librería geneticalgorithm [6]. Analizamos las tres librerías recomendadas por el profesor y esta fue la que vimos más sencilla de manejar y entender. Además, hemos usado la librería numpy para generar una semilla aleatoria como se indicada en el enunciado.

Hemos desarrollado dos métodos que realizan una iteración de un autómata celular, una para unidimensionales y otra para bidimensionales. Los métodos reciben una secuencia de 0s y 1s llamada CI y la regla que se aplicará a la CI. Se recorre

la CI y para cada célula se obtienen sus vecinos para luego analizar como quedará esta celda en el siguiente estado del autómata.

Para la versión unidimensional, hemos codificado la regla como un secuencia de 8 bits, cada uno representa el valor de la celda en el siguiente estado siguiendo el orden establecido por Wolfram (el que se puede ver en Fig. 4).

En el caso de los bidimensionales, la regla se codifica como una secuencia de cuatro enteros que representan el número de vecinos de la célula, representándose textualmente como BxySzw siendo x, y, z, w números entre 0 y 8 (ambos incluidos). Las dos primeras casillas representan el rango de número de vecinos que tienen que estar vivos para que la célula nazca (valga 1). Las dos siguientes indican el rango de número de vecinos que tienen que estar vivos para que la célula sobreviva (se mantenga a 1). En cualquier otro caso, la célula morirá (valdrá 0). Por ejemplo, la regla B2S13 se representa como [2,2,1,3], como se puede ver si se repiten dos números significa que debe ser exactamente ese número para ser 1. Por lo tanto, realmente no se tendrán en cuenta la posición los vecinos, sino del número de 1s que hay para determinar el próximo estado.

Se coge a los 8 vecinos usando las nomenclaturas que se pueden ver en Fig. 7:

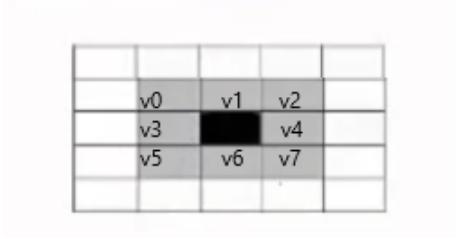


Fig. 7. Nomenclatura para los vecinos en autómatas bidimensionales

También se han implementado dos métodos para generar un determinado número de CIs de forma aleatoria. Cada CI tendrá un tamaño aleatorio y se usará la versión de una dimensión o 2D según lo que se necesite en cada problema.

Cada ejercicio sigue una estructura similar:

- Un método que define cuál será la CO dada una CI.
- Otro método devuelve True si llega a la CO dada una CI y un autómata definido por una regla en un máximo de Tmax iteraciones. Se aborta la ejecución del autómata si se encuentran 2 secuencias iguales seguidas
- Una función que valora la regla del autómata celular. La puntuación se basa es el porcentaje de CIs que sabe llevar a su CO. En algunos casos, si una regla sabe resolver CIs que su COs sean distintas, se le aplicará una bonificación para que el algoritmo genético sepa valorar esta característica.
- La inicialización del algoritmo genético indicando los parámetros necesarios y la función de fitness. El valor de fitness será la valoración ofrecida por el método anterior en negativo, esto es necesario ya que el algoritmo busca minimizar el fitness y nosotros queremos maximizar ese valor. Además, al algoritmo genético le decimos con dos parámetros que no nos muestre ni la gráfica ni la barra de progreso.
- Un bloque for donde se realizan las iteraciones del algoritmo genético. El modelo se inicializó para que cada ejecución del método run realice una sola iteración. Gracias a esto, nos permitió cambiar las CIs con las que se trabaja en cada iteración.
- Finalmente se encuentra una celda de código dedicada a probar las reglas encontradas con múltiples CIs y comprobar lo óptima que es.

Para cada actividad se han realizado los comentarios oportunos en el código además de instrucciones print para poder ver en detalle la traza del código, solo será necesario descomentar para poder ver lo que se necesite en consola.

A continuación veremos en detalle los distintos ejercicios propuestos:

1) *Ejercicio 1: Problema de la mayoría.* Dada una CI, llevarla a una configuración objetivo (CO) determinada por un valor p que será el porcentaje de 1s de la CI:
   - Si $p < 0.5$, la configuración objetivo será una secuencia de 0s.
   - Si $p > 0.5$, la configuración objetivo será una secuencia de 1s.
   - Si $p = 0.5$, la configuración objetivo será cualquiera de las anteriores.

   Usando un autómata celular elemental y algoritmos genéticos se pide obtener una regla capaz de llevar el mayor número de CI posibles a la CO.
   Lo primero que implementamos fue el método *porcentaje1s()*, el cual nos servirá tanto en un inicio para saber cuál es la CO como en cada iteración del autómata para saber si ya se a llegado a esta CO.
   Por otra parte, tenemos la función que intenta llegar a la CO, en este caso usando el método *porcentaje1s()* y *iteracionAutomataUnidimensional()*. Para la valoración usaremos el método anterior y haremos uso de una bonificación de resolución de casos distintos. Para los demás métodos no hay cambios sustanciales con respecto a lo comentado anteriormente.
2) *Ejercicio 2: Problema de la paridad.* Dada una CI, llevarla a una CO determinada por un valor n que será el número de 1s de la CI (Se usará para este apartado un autómata celular bidimensional):
   - Si n es un número par, entonces la configuración objetivo (final) será una cuyo estado de todas las celdas sea 1.
   - Si n es un número impar, entonces la configuración objetivo (final) será una cuyo estado de todas las celdas sea 0

   El método *num1sPar()* nos determinará cual el la CO. Crearemos una versión bidimensional de *porcentaje1s()* para poder saber si hemos llegado a la CO. El método para llegar al objetivo usará las funciones para autómatas bidimensionales.

Para el método que puntúa la regla no será necesario usar la bonificación ya que encuentra buenos resultados sin usarla. Sin embargo, ya que el algoritmo genético genera las reglas de forma aleatoria hay combinaciones que no son válidas según la definición anteriormente aportada Por ejemplo, el algoritmo puede generar la regla [3,1,4,2]. En un principio, pensamos en valorar esa regla como 0 para solo usar las reglas bien formada. Esta idea fue rechazada debido a la frecuencia con la que aparecía reglas mal formadas, por lo que se desaprovechaba el potencial del algoritmo. Finalmente decidimos que si nos encontramos una regla incorrecta, le daremos la vuelta de la siguiente forma: [3,1,4,2] será tratada como [1,3,2,4]. Si en algún momento vemos una regla de este tipo solo tenemos que pensar que se refiere a la equivalente según lo expuesto.

El único cambio que se realiza respecto a los demás ejercicios para lo que queda de código, a parte de usar los métodos adecuados, será el uso de variable_boundaries para limitar los valores de los genes entre 0 y 8 a la hora de inicializa el algoritmo genético.

3) *Ejercicio 3: Problema de un solo 1.* Dada una CI, llevarla a una CO determinada por:
   - Si la CI es una secuencia de 0s con un solo 1, la configuración objetivo será una secuencia de 1s.
   - En cualquier otro caso, la configuración objetivo será una secuencia de 0s.

   La función que determinará la CO será *tieneUnSolo1()*, la cual devolverá True si la CI tiene un solo 1. Aparte de esto todos los demás métodos seguirán la estructura mencionada haciendo uso de los métodos dedicados a este problema.

   Un detalle a comentar es la CI que lleva a todo 1s, esta es muy improbable generarla aleatoriamente. Por ello, hemos generado varias aleatoriamente pero luego hemos añadido a mano CI que tengan un solo 1 para que el problema pueda estudiarse correctamente.

IV. Resultados

A continuación, veremos los resultados obtenidos tras la ejecución de los distintos ejercicios y cuáles son nuestras reflexiones.

Para todos los ejercicios, los parámetros del algoritmo genético han sido:

- Número de iteraciones = 1, para que cada vez que usemos el método run solo haga una iteración y podamos cambiar de CIs en cada una. En cada ejercicio se ejecutará el método run un determinado número de veces. No hay un número máximo de iteraciones sin mejora.
- 20 individuos en la población inicial, 1 de ellos será de élite debido a que el ratio de elitismo es de 0.05.
- Probabilidad de cruce de 0.5 y de mutación de 0.3. Un 0.3 de parent_portion.
- El cruce será del tipo uniform.

Estos parámetros fueron ajustados para el primer ejercicio y se probaron para los demás y también dieron buen resultado, por lo que no los hemos cambiado de uno a otro.

Cabe recalcar que hemos desarrollados estos problemas en la herramienta de Google Colaboratory, por lo que los tiempos de ejecución los hemos sacado de la parte inferior de la página. Si se mantiene el ratón encima tras ejecutar una celda, muestra el tiempo que ha tardado en ejecutarla.

1) *Ejercicio 1: Problema de la mayoría.* Realizamos 10 iteraciones que conllevan 30s de ejecución. Para este problema las reglas que el algoritmo nos devuelve están centradas en llevar las CIs a todo 1s o a todo 0s. Reglas como la 254 o la 238 son resultados frecuentes que llevan las CI a 1s, por otro lado, la regla 128 lleva todo a 0s. Este tipo de reglas no son muy útiles ya que solo son capaces de resolver un tipo de caso y nunca lo harán mejor que la regla 255 y la regla 0. Obtenían una puntuación de en torno al 0.55 que no era más que el porcentaje de reglas con $p > 0.5$ ó $p < 0.5$ más las de $p = 0.5$.

   Para arreglar este problema, decidimos otorgar una bonificación a las reglas que sean capaces de llevar CIs a ambos objetivos. Gracias a esta técnica obtuvimos las reglas 127 y 7, las cuales son capaces de resolver ambos casos y consiguen resolver alrededor de un 20% de CI correctamente. Particularmente, la regla 127 parece darnos mejores resultados que la 7.

   Investigando sobre estas dos reglas hemos visto que tienen una característica que las hacen buscar líneas de 1s y 0s con más frecuencia (se puede ver en la Fig. 8).

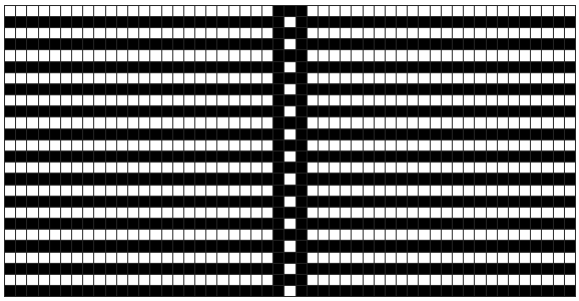


Fig. 8. Evolución de la regla 127.

Por lo que hemos visto, este problema no es posible resolverlo con este tipo de autómata. El problema de la mayoría ha sido muy estudiado a lo largo de la historia [9]. Gács, Kurdyumov, and Levin encontraron un autómata de radio 3 que el valor dependía de sí mismo, de su vecino y de el vecino 3 posiciones hacia el lado. Se mirarán los de la derecha si la celda vale 1 y los de la izquierda si vale 0. Este autómata tiene una precisión de 78%.

Incluso se han propuesto soluciones que combinan la regla 184 y la 232, pero esto y no sería considerado un autómata celular. Land and Belew demostraron que no es posible hallar un autómata que sea una solución

perfecta de este problema solo usando 2 estados. Esto se debe a que siempre habrá una CI lo suficientemente grande que no podrá ser resuelta de esta forma.

2) *Ejercicio 2: Problema de la paridad.* Realizaremos 3 iteraciones y generaremos 30 CI en cada iteración, suficientes para encontrar una buena solución. El tiempo de ejecución será aproximadamente de 3 minutos, algo normal debido a que para este problema trabajamos con AC bidimensional.

   Las mejores reglas encontradas son B07S15 y la B02S0, ambas con un noventa y nueve por ciento de éxito. Estas reglas nos resultan muy rentables ya que son capaces de resolver la mayoría de los casos y creemos que son realmente soluciones del problema.

   Analicemos la evolución de la regla B02S0:

```
0 0 0 0 0   1 1 0 1 1   0 0 0 0 0   1 1 1 1 1   0 0 0 0 0
0 1 1 1 0   1 0 0 0 1   0 0 1 0 0   1 1 1 1 1   0 0 0 0 0
0 1 1 0 0   1 0 0 0 1   0 0 0 0 0   1 1 1 1 1   0 0 0 0 0
0 0 0 0 0   1 1 1 1 1   0 0 0 0 0   1 1 1 1 1   0 0 0 0 0
```

Fig. 9. Evolución de la regla B02S0.

   Como se puede ver en la Fig. 9, la estrategia de estas reglas es ir erosionando los grupos de 1s para llegar a una configuración del mismo valor, si esa no es su CO sabe llegar en la siguiente iteración a ella. Debido a esta estrategia pensamos que estas reglas son solución ya que si se encuentran muchos unos juntos necesitará más iteraciones para poder llegar a la CO. Como tenemos límite de iteraciones para evitar bucles infinitos, muy pocas veces no soluciona el problema por este motivo.

3) *Ejercicio 3: Problema de un solo 1.* Para 5 iteraciones de este problema con 200 CIs (100 aleatorios y 100 casos de un solo 1) el algoritmo tarda 34s en ejecutarse. La mejor regla encontradas vuelve a ser la 127 con un 0.6 de puntuación debido al mismo comportamiento del ejercicio 1. Las reglas 31, 87 y 85 también funcionan bien con un 0.58 de valoración.

   Este problema es muy curioso, el caso que su CO es todo 1s es el que tiene un solo 1 en su CI. Pero realmente, esta CI es solo una, ya que no importa donde este ese 1 que el comportamiento será el mismo. Solo puede tener otro si es una CI con muchas casillas, debido al límite de iteraciones establecido.

   Analizando este caso podemos ver que para poder llegar a la CO necesitamos consultar las posiciones 3, 5, 6 y 7 de la regla. Estas son las que hay o un solo 1 o todo a 0 (100, 010, 001 y 000). De aquí sacamos en claro que al menos debe haber un 1 en esas posiciones para que el autómata pueda avanzar. No nos vale solo la posición 6 ya que la deja como estaba, pero sí será necesaria si usamos la posición 3 o la 5 sin usar la 7. El tener un 1 en la posición 7 nos facilitará las cosas pudiendo llenar de 1s rápidamente la secuencia y resolviendo las CI de muchas casillas sin problemas con el máximo de iteraciones

   Pensamos que con estas limitaciones en las posibles reglas reducimos el número de reglas que pueden ser solución del problema, y si existiera, habrían salido como solución del algoritmo genético. Como no ha sido así, no creemos que se pueda resolver este ejercicio con un autómata elemental.

V. Conclusiones

En el presente artículo hemos definido y señalado algunas características y propiedades de los autómatas celulares. Es sorprendente la capacidad que tienes los autómatas para simular desde fenómenos naturales, sistemas biológicos, hasta como también contribuye en las actividades humanas, al lograr que muchas operaciones matemáticas y de cómputo sean más rápidas y permitan un manejo de datos superior a otros modelos abstractos, ayudando así a dar una interpretación de los resultados de estudios de investigaciones aún más completa y favoreciendo la prevención de situaciones no deseadas.

La metodología de programación convencional es, naturalmente, de poca utilidad para un sistema de computación basado en un autómata celular. El desarrollo de una nueva metodología es difícil, pero un desafío importante, y los algoritmos genéticos pueden ser una herramienta fundamental en la búsqueda de la misma.

Es notoria cada vez más la importancia y la difusión que se le ha dado a los autómatas celulares, vemos como se fomenta el estudio de esto por parte de escuelas, universidades, etc. También invita a reflexionar en lo que se puede lograr profundizando en su entendimiento. Tal vez abra las puertas a nuevos paradigmas científicos que permitan, como los propios autómatas, pequeños avances, pero a pasos enormes.

Referencias

[1] Melanie Mitchell, “An introduction to genetic algorithms”, MIT Press, Cambridge, MA, USA, 1998.
[2] Avik_Dutta - GeekForGeek, Crossover in Genetic Algorithm
[3] Juan Ignacio Vázquez y Javier Oliver, “Evolución de AC por AG”
[4] Ron Breukelaar y Thomas Bäck, “Evolving Transition Rules for Multi Dimensional Cellular Automata”
[5] David Alejandro Reyes Gómez, “Descripción y Aplicaciones de los Autómatas Celulares”
[6] geneticalgorithm documentation
[7] Stephen Wolfram, “A new Kind of Science”, Cap. 2 y 3
[8] McCulloch-Pitts, “Neurona de McCulloch-Pitts”
[9] “Majority problem (cellular automation)”