REGLAS Y AUTOMÁTAS CELULARES

Jesús Antonio Rivero Gallardo
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
jesrivgal@alum.us.es | rivero.gallardo.jesus@gmail.com

Francisco Villazán Torres
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
fraviltor@alum.us.es | fvillazan1999@gmail.com

En este trabajo se pretende diseñar dos autómatas celulares independientes, capaces de resolver cada uno el problema de la mayoría y el problema de la paridad, respectivamente. Además de este concepto, se hace uso también de los algoritmos genéticos, con el fin de optimizar los resultados obtenidos.

Al combinar estos dos conceptos satisfactoriamente se ha podido realizar un estudio exhaustivo de estos problemas y poder llegar a unas conclusiones claras y concisas sobre sus respectivas resoluciones.

Palabras clave - Inteligencia Artificial, Autómatas celulares, Algoritmos genéticos, Optimización, Células, Reglas, Generaciones, Números binarios, Mayoría, Paridad.…

I. INTRODUCCIÓN

No tendría sentido empezar a hablar sobre aplicaciones prácticas de los autómatas celulares o de algoritmos genéticos, sin antes dar una definición clara ni unas breves pinceladas sobre los aspectos fundamentales de estos. No obstante, una vez presentados los conceptos que, a nuestro parecer, son los requeridos para comprender la magnitud y resolución del problema, se tratarán en profundidad cada uno de los problemas planteados y su correspondiente resultado.

Los autómatas celulares constituyen un modelo matemático y computacional que, dado un sistema, evoluciona de manera discreta, es decir, que no puede tomar ningún valor entre dos consecutivos. Son apropiados para modelar sistemas naturales, ya que las reglas que siguen los autómatas se asemejan mucho a estas, aunque también son aptos para resolver problemas cuya descripción se pudiera relacionar con una muy amplia colección de objetos simples y homogéneos que interactúen entre ellos mismos.

Aunque sea complicado dar una definición precisa y formalmente correcta de un autómata celular, estos se pueden describir cómo un conjunto de elementos simples, denominados células, las cuales se disponen siguiendo la forma de un tablero o rejilla (se podría definir cómo que cada elemento del denominado tablero es una célula) y van cambiando de estado cómo consecuencia de una serie de reglas predefinidas. Cómo se ha mencionado antes, las células evolucionan en intervalos de tiempo discretos, esta evolución viene dada por las células que la rodean. Dentro de los autómatas celulares encontramos tanto los unidimensionales como los bidimensionales.

En primer lugar, los unidimensionales, se corresponden con una secuencia lineal de casillas. En estos casos podemos definir un radio de vecindad r, que viene dado cómo r=1 en el caso de que el estado de la célula en cuestión viene dado por el de las dos células que están situadas a su izquierda y derecha. En el caso de r=2 el funcionamiento sería el mismo que el anterior pero teniendo en cuenta el estado de las dos celdas situadas inmediatamente a su derecha y a su izquierda. Se define k cómo el número de estados que puede adoptar una célula, para los casos de estudio de este trabajo se tomará siempre k=2 ya que las células sólo van a poder tomar 2 valores, siendo estos 0 o 1.

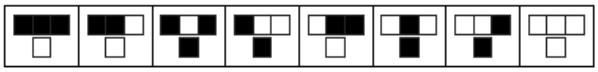

Fig. 1. Tabla con regla de autómata celular unidimensional con r=1 y k=2

En la Figura 1 se puede ver el cambio que sufre la célula central, debido a los estados de sus células colindantes. A partir de este ejemplo se podrían extraer una serie de reglas, obtenidas a partir de identificar los patrones de evolución de las células. Todos los autómatas con r=1 y k=2, con los que se ha trabajado en el primer ejercicio propuesto el cual luego será analizado, vienen definidos por un total de 8 reglas. Cada regla queda definida por el estado final de la célula afectada por la regla, en otras palabras, cada regla se corresponde con cada una de las posibles combinaciones de 0s y 1s. Teniendo en cuenta que, en estos casos, siempre va a haber 8 secuencias posibles, se puede afirmar que existen un total de 256 reglas para este tipo de autómatas celulares.

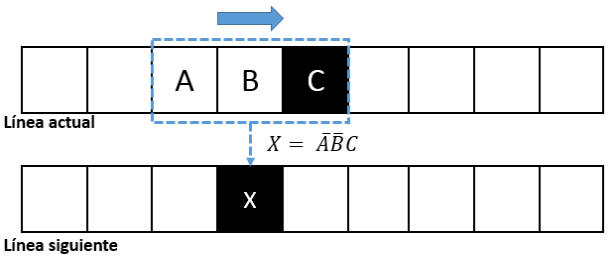

Fig. 2. Ejemplo de evolución de una célula de un estado al siguiente

En segundo lugar están los autómatas bidimensionales, en estos es donde el concepto de rejilla, previamente dado, cobra más sentido. En este caso el estado de una célula viene dado por el de todas las células colindantes a ella, sus vecinas. Este tipo de autómata celular es el empleado en el segundo ejercicio

propuesto, de cara a ejemplificar la estructura de estos autómatas se va a tratar el popular Juego de la vida del matemático británico John Horton.

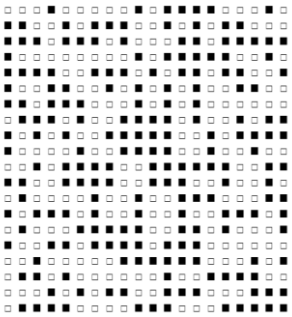

Fig. 3. Ejemplo de autómata celular bidimensional

La premisa de este problema es simple: el tablero viene dado por un autómata celular bidimensional que se extiende infinitamente en el espacio, lo que implica que cada célula tiene 8 células vecinas. Cada una de estas células tiene dos estados, "viva" o "muerta". Una célula "muerta" con exactamente 3 células vecinas vivas "nace". Una célula "viva" con 2 o 3 células "vivas" a su alrededor sigue viva, en otro caso muere.

A raíz de este problema se pueden sacar varias conclusiones sobre los autómatas celulares bidimensionales, en el caso concreto que a nosotros nos concierne de cara a abordar el problema práctico propuesto. Quizás una de las mayores diferencias con los autómatas celulares unidimensionales es el hecho de que, en este caso particular, no importa la posición de las células vecinas, simplemente tiene relevancia la cantidad de células cuyo estado sea "viva" y la cantidad de células cuyo estado sea "muerta".

Por último, cabe destacar que el comportamiento de los autómatas celulares puede llegar a ser clasificado, esto es debido a que observando su comportamiento se puede discernir un cierto comportamiento, el cual podemos clasificar en alguna de las siguientes categorías:

- Comportamiento fijo: todos los estados iniciales, tras un cierto número de generaciones, siempre van a converger a un mismo estado final.
- Comportamiento periódico: todos los estados iniciales convergen al mismo estado final o algún ciclo periódico.
- Comportamiento caótico: cuando desde un cierto estado inicial se converge hacia un estado caótico, el cual no es posible de predecir.
- Comportamiento complejo: aquí clasificamos cualquier estado inicial el cual converge en estructuras complejas, las cuales son conservadas durante el tiempo.

Para el desarrollo de este trabajo, tanto el estudio previo necesario cómo el desarrollo en sí de los problemas se han usado distintas fuentes, siendo algunas de estas proporcionadas por el profesorado [1] [2] [3]. Además de esta, durante nuestro trabajo de investigación hemos encontrado otras fuentes que nos han sido de gran utilidad y en las cuales nos hemos basado parcialmente, [4][5][6][7][8][9][10].

II. PRELIMINARES

Muchos problemas se reducen a la búsqueda de la solución óptima de una función concreta, pudiendo obtenerse esta solución por medio de la minimización o maximización del problema.

A. Métodos empleados

Un algoritmo genético no es más que un algoritmo matemático que transforma una población dada en otra, denominada siguiente generación, que se adapte mejor a un problema planteado. Esta transición se asemeja mucho con el "Principio de supervivencia del más apto", ya que esta selección se realiza en base a una valoración que se le da a cada población, persistiendo siempre la que más convenga según el problema. Esta valoración, también llamada adaptación o "fitness", no es más que una ponderación numérica, resultante de una serie de cálculos sobre la población. Estos cálculos se basan en premiar o penalizar a la población en cuestión, siguiendo una serie de criterios no previamente definidos, es decir, estos criterios van a depender en su totalidad del enunciado que se proponga y del resultado que se busque obtener.

Una vez se ha asignado un fitness a cada población entran en juego los denominados operadores genéticos:

- Reproducción: aquellos individuos con mejor adaptación se escogen, generando una descendencia cuya adaptación asociada será proporcional al de sus progenitores.
- Cruzamiento: nuevos individuos son creados a partir de los ya existentes, generando nuevas posibilidades por el espacio de soluciones del problema.
- Mutación: se altera algún elemento de la configuración de un individuo, creando así nuevos individuos los cuales abren nuevas posibilidades en el espacio de soluciones del problema.

Estas operaciones que se realizan sobre los individuos se realizan un número determinado de veces. Un algoritmo genético ejecuta tantas iteraciones cómo se le impone, este valor puede venir definido de distinta manera, ya sea parar de generar poblaciones en el momento que se incumpla alguna restricción o dejarlo operar libremente hasta que consiga un resultado que se considere satisfactorio.

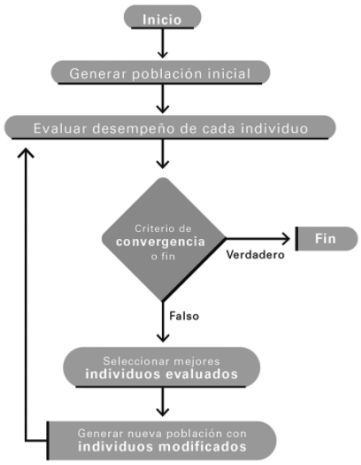

Fig. 4. Representación del funcionamiento de un algoritmo genético

Vamos a tratar ahora los principales parámetros de los algoritmos genéticos, particularmente los que usa la librería de Python DEAP que es de la que se ha hecho uso para resolver los problemas planteados. Estos son los siguientes:

- Fitness: corresponde con el fitness, del que ya se ha hablado con anterioridad, en este caso es una clase abstracta proporcionada por la librería. Necesita ciertos atributos "weights" para poder funcionar correctamente. Al igual que en un algoritmo genético cualquiera de forma estándar tiende a la maximización, esto puede invertirse usando "weights" negativos, consiguiendo así que el algoritmo tienda a la minimización.
- Individual: en el caso que se está tratando, los individuos corresponden con las distintas reglas. El individuo es el objeto de evaluación del problema, ya que el individuo mejor evaluado, es decir, el que tenga mejor fitness, corresponderá con la regla que resuelve nuestro problema de forma más efectiva.
- Population: el funcionamiento de las poblaciones es muy similar al de los individuos, con la salvedad de que se las poblaciones se inicializan con individuos o estrategias en lugar de con atributos.

B. Trabajo Relacionado

Para comprender mejor, desde un punto de vista teórico, la aplicación de algoritmos genéticos al desarrollo de estos problemas que usan autómatas celulares nos hemos basado en ciertos documentos de índole similar, donde se aplican estas dos técnicas de forma conjunta [4][5]. Ambos trabajos de investigación están enfocados desde un punto de vista ajeno totalmente a la implementación del código, aun así nos han sido de tremenda utilidad de cara a comprender cómo se combinan estas dos técnicas.

III. METODOLOGÍA

Esta sección se dedica a la descripción del método implementado en el trabajo:

A. Problema 1

El primero de los dos problemas que resolver era el siguiente: diseñar un autómata celular capaz de resolver el problema de la mayoría, usando un autómata celular elemental.

Una particularidad de este problema es el hecho de que no está definido en el caso de haber una total igualdad en la relación de 1s con respecto a 0s que aparecen en la configuración inicial. Para solventar esto, lo cual podría llegar a ser un problema, decidimos implantar desde un inicio que el número de células fuese siempre impar, evitando así este caso.

Según se ha visto en la teoría de los autómatas celulares unidimensionales, existen 8 secuencias y un total de 256 reglas. En base a esto, decidimos almacenar estas 8 secuencias en una lista de tuplas, donde cada tupla estuviese conformada por la codificación de las 3 células que definirían el estado de la siguiente.

La lista de tuplas es la siguiente: [(1,1,1),(1,1,0),(1,0,1),(1,0,0),(0,1,1),(0,1,0),(0,0,1),(0,0,0)].

A raíz de esto decidimos diseñar una función que generase una lista de listas, donde cada una de estas sublistas contuviese una regla, expresada de la siguiente forma:

[[1,0,1,1,0,0,1,1],[0,0,0,1,1,0,0,0],[0,1,1,1,0,0,1,0], ...].

En el caso de la primera lista, la regla que esta representa sería la regla 179:

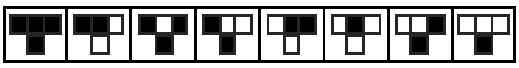

Fig. 5. Regla 179 de los autómatas celulares unidimensionales

Para el correcto funcionamiento del programa es necesario implementar distintas funciones de la librería DEAP. Encontramos un amplio abanico de funciones, encargadas de diversas tareas. Algunas de estas son: generar e inicializar los atributos, definir las probabilidades de cruce y mutación de individuos, definir una serie de estadísticas que nos ayuden a comprender los resultados obtenidos, etc. Todas estas funciones se han extraído del manual de instrucciones de la librería, siendo fragmentos de código independientes pero que al ejecutarlos de forma conjunta consiguen definir y ejecutar todos los parámetros necesarios para la buena aplicación de nuestras funciones, cómo la de fitness, y poder así obtener una solución óptima para el problema.

Una vez planteado todo el diseño del problema sólo queda diseñar una función de fitness que evalúe correctamente cada una de las reglas que se han generado. Por la definición del problema el objetivo es llegar a un estado, denominado como estado final, en el que el valor de todas las celdas sea 1 o 0, dependiendo del valor que predomina en el estado inicial. Es por esto que lo que le da valor a una regla es llegar al estado final correcto, con lo cual, conseguir esto es lo que le suma un valor positivo al fitness de una regla. Por el contrario, si la regla lleva el estado inicial a un estado final que no corresponde será penalizada. Además de esto, se contabilizan las generaciones

que tarda en llegar, poniendo el máximo de generaciones en 800, y suponiendo una penalización por cada generación que tome, obviando la primera.

Pseudocódigo correspondiente a la función fitness del primer problema:

FitnessFunctionEj1
Entrada:
- Una lista L (la regla a evaluar)
- Un conjunto de estados iniciales, en este caso lista de listas con valores binarios (la muestra)
- Lista de tuplas con las 8 posibles secuencias de 3 dígitos

Salida:
- Un valor que indica el fitness de la regla (lo buena que es)

Algoritmo:
1. Para cada estado, mientras la cantidad de 1s o 0s no sea la deseada:
    a. Recorrer el estado elemento a elemento, seleccionando para cada uno los elementos de sus lados, para así evaluar con la regla el valor a asignar en el estadoSiguiente
    b. Si el estado obtenido es igual al anterior, habremos entrado en bucle:
        i. Salimos y penalizamos
    c. estadoInicial=estadoSiguiente para la siguiente iteración
    d. Incrementamos en 1 las generaciones para luego primar la regla que demore menos en obtener el objetivo
    e. Si generaciones>800:
        i. Consideramos que no alcanzaremos el objetivo y salimos del bucle
    f. Si objetivoAlcanzado:
        i. Valoracion+1000
    g. Si conseguimos el estado contrario a objetivo:
        i. Penalizacion+1000
2. valoracionTotal=valoracionTotal+valoracion penalizaciones=penalizaciones+penal+generaciones
3. Devolvemos el fitness=valoracionTotal-penalizaciones

B. Problema 2

En este caso se pedía diseñar un autómata celular capaz de resolver el problema de la paridad, haciendo uso de autómatas celulares bidimensionales. El primer cambio significativo con respecto al problema anterior, más allá del propio problema a resolver, es la implementación de un autómata celular bidimensional, con las diferencias que supone respecto del unidimensional.

En primer lugar, había que diseñar una función capaz de generar diferentes rejillas, de tamaño NxN, en lugar de simplemente una línea, cómo es en el caso de los unidimensionales. Esta parte del diseño no se limita únicamente a la generación de las rejillas, sino también, por ejemplo, la creación de una función capaz de calcular el estado de las células según el estado de sus vecinas. Cabe destacar que aparte de estas funciones que sí hemos tenido que diseñar, y la función para el fitness de la cual se hablará a continuación, todo el código relacionado con la librería DEAP es el mismo que el usado en el anterior ejercicio, con la salvedad de algún que otro detalle puntual.

En contraste con el primer problema, en el cual se podía identificar muy intuitivamente el número exacto de reglas existentes y se podía trabajar en base a ello, en este nos encontramos con la existencia de un número mucho mayor de reglas posibles, ya que estas dependen en función del número de células adyacentes "vivas" y además, existen dos premisas a partir de las cuales puede cambiar el estado de una célula. Para enfrentar este dilema nos surgió la idea de plantear las reglas de tal manera que se generasen aleatoriamente dos números binarios de 9 cifras, uno para cada una de las dos premisas del problema, en los cuales cada posición correspondiese al número de células vecinas que se encuentran "vivas", representadas por un 1, o "muertas", representadas por un 0. Estos dos números se almacenan en una lista. Un ejemplo de la codificación usada es el siguiente:

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

En este caso el número 101100001_2 representaría que una célula "nace" en el caso de que tenga 8,6,5 o 0 células vecinas "vivas".

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

En este caso el número 011000110_2 representaría que una célula "sobrevive" en el caso de que tenga 7,6,2 o 1 células vecinas "vivas". Formalmente esta regla se expresaría de la forma B0568S1267.

Una vez implementada nuestra particular forma de generar aleatoriamente posibles reglas, sólo quedaba crear la función que calculara un fitness para cada una de ellas y las clasificara en concordancia con dicho valor asociado. Siguiendo con una tónica muy similar a la del primer problema se ha tenido en cuenta principalmente si la regla es capaz de obtener un tablero completo de células "vivas" o "muertas" según su estado inicial, siendo esto el mayor mérito y, por tanto, lo que da valor a una regla. Por otro lado, si se llega al estado opuesto al deseado la penalización será igualmente severa. En este caso, la bonificación por alcanzar el objetivo será de 1000, mientras la penalización por lograr el opuesto le da a la regla un valor de 100, siendo un valor negativo en caso de la penalización.

Además de esto, también se penalizará a la regla en función de las generaciones que necesite para llegar al estado deseado, siendo el valor de esta penalización cómo máximo de 400, ya que en caso de llegar a este número de generaciones el

algoritmo parará, asumiendo que la regla nunca conseguirá su objetivo.

Pseudocódigo correspondiente a la función fitness del segundo problema:

FitnessFunctionEj2
Entrada:
- Una lista L (la regla a evaluar)
- Un conjunto de estados iniciales, en este caso arrays de arrays con valores binarios (la muestra)

Salida:
- Un valor que indica el fitness de la regla (lo buena que es)

Algoritmo:
1. Para cada estado, mientras la cantidad de 1s o 0s no sea la deseada:
   a. Obtenemos los vecinos
   b. Actualizamos el estado tantas veces como necesitemos hasta alcanzar el estadoObjetivo:
      i. Si pos[i,j]==0 y el num(vecinos) está en L:
         1. pos[i,j]==1
      ii. Si pos[i,j]==1 y el num(vecinos) no está en L:
         1. pos[i,j]==0
      iii. Cada vez, incrementamos en 1 las generaciones
   c. Si estadoObjetivo:
      i. Valoracion+1000
   d. Si conseguimos el estado contrario a estadoObjetivo:
      i. Penalizacion+100
2. valoracionTotal=valoracionTotal+valoracion penalizaciones=penalizaciones+penal+generaciones
3. Devolvemos el fitness=valoracionTotal-penalizaciones

IV. RESULTADOS

En esta sección se detallarán tanto los experimentos realizados como los resultados conseguidos:

A. Problema 1

En un primer momento, cuando estábamos planteando el problema valoramos la posibilidad de que existiera una regla la cual resolviera una gran mayoría de estos casos, alrededor del 70-80% de casos. Las primeras reglas que se nos vinieron a la mente fueron las soluciones más triviales, es decir, las reglas 0 y 255 para los casos en los que existiera una mayoría de estados iniciales con mayoría de 0s y para los casos en los que existiera una mayoría de estados iniciales con mayoría de 1s. Estos casos fueron descartados en un inicio ya que, además de ser los triviales, su % de acierto depende totalmente de la aleatoriedad ya que sólo solucionan los casos en los que exista una mayoría de 0s o 1s. Después de esto la siguiente regla en la que pensamos fue la regla 232, la cual se relaciona directamente con la mayoría, ya que cambia el estado de una célula según el estado más repetido en sus células adyacentes.

Fig. 6. Regla 232 de los autómatas celulares unidimensionales

Una vez implementado el código empezamos a hacer pruebas y observamos que las reglas que mejor fitness obtenían eran reglas muy similares a la 0 o la 255, es decir, reglas que en muy pocas generaciones convertían un estado inicial en uno final de todo 0s o todo 1s. Esto es un arma de doble filo ya que estas reglas resuelven correctamente y muy rápidamente todos los estados que empiezan con una mayoría de células cuyo valor es favorable pero asimismo, resuelven de forma incorrecta todos los demás casos. Tras muchos intentos, y ajustando el código, siempre se llegaba a resultados muy similares.

Teniendo todo esto en cuenta llegamos a la siguiente conclusión, no existe una regla de autómatas celulares unidimensionales capaz de resolver de forma efectiva el problema de la mayoría. Si bien es cierto que hay reglas mejores y peores para cada estado, estas siempre tienden a ser las triviales o muy similares a estas y, cómo ya se ha mencionado anteriormente, la efectividad de dichas reglas se basa en su totalidad en sí existe mayoría de 1s o 0s en el estado inicial el estado del que partamos.

Una vez llegamos a la conclusión de que no había una regla fiable para resolver un alto % de estados decidimos implementar el uso de las semillas, para intentar mejorar la fiabilidad de los resultados obtenidos. Una semilla se genera mediante un método de Python que nos permite controlar la aleatoriedad en la generación de estados iniciales. En nuestro caso, las usamos para imponer una mayoría de 1s o 0s en la mayoría de estados iniciales y poder así estudiar los dos casos por separado. Gracias a esto pudimos confirmar nuestras conclusiones ya que, imponiendo una semilla que genera una mayoría de 1s, siempre se repetían los mismos resultados y de igual forma con una para 0s.

Para ejemplificar un poco todo esto, estas son algunas de las reglas cuyo comportamiento es muy similar al de la 0 o la 255:

- En el caso de tener un estado inicial con mayoría de ceros encontramos por ejemplo las reglas 32, 40, 64...

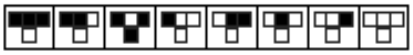

Fig. 7. Regla 32 de los autómatas celulares unidimensionales, una solución habitual para los estados con mayoría inicial de 0s.

- En el caso de tener un estado inicial con mayoría de unos encontramos por ejemplo las reglas 239, 235, 251...

Fig. 8. Regla 239 de los autómatas celulares unidimensionales, una solución habitual para los estados con mayoría inicial de 1s.

B. *Problema 2*

Según la premisa del segundo problema, y una vez con conocimiento del tremendo número de posibles respuestas, lo encaramos con la idea de poder encontrar una respuesta clara.

El primer inconveniente con el que nos encontramos fue el hecho de que en este problema se trabaja con un conjunto de datos mucho más amplio que en el problema anterior, esto repercute directamente en tiempos de ejecución mucho más elevados, sobre todo en los casos en los cuales el estado inicial es muy complejo y extenso. Debido a esto decidimos centrar nuestro objetivo de estudio principalmente en los estados iniciales de 10x10 células, aunque para contrastar nuestras conclusiones probamos con estados mucho mayores.

De igual forma que en el ejercicio anterior encontramos oportuno el uso de semillas, principalmente para diferenciar los dos claros casos de estudio y poder contrastar los resultados obtenidos entre ellos.

Tras muchas pruebas y ajustes, sobre todo en el fitness, dimos con la clave y empezamos a sacar nuestras propias conclusiones a raíz de los resultados que íbamos obteniendo. En este problema, a diferencia del anterior, si existen reglas que solucionan una inmensa mayoría de estados iniciales, incluso resolviendo todos si no se limita el número de generaciones. Hemos obtenido muchas reglas distintas capaces de solucionar muchos de los estados iniciales, incluso limitando severamente el número de generaciones. Algunos ejemplos de estas reglas son: B58S5678, B012368S01345, B012358S0235...

Teniendo en consideración el inmenso número de reglas posibles y que con diferentes pruebas se han obtenido muchas de ellas distintas, podemos llegar a la conclusión de que en total debería de existir un elevado número de reglas válidas.

Un dato que merece la pena destacar es el hecho de que en un principio obtuvimos muchos mejores resultados, en cuanto al fitness, ya que no restringimos que, por ejemplo, para llegar a un estado final de todo células “muertas” pasara previamente por un estado intermedio de todo células “vivas”. Algunas de las reglas mencionadas anteriormente solucionaban el 100% de los estados iniciales cuando no se había implementado esta restricción, pasando tras su implementación a resolver alrededor del 60-70% de los casos.

Para ejemplificar esto último supongamos una regla B015S136 y un tablero inicial con un número par de células “vivas”. Si esta regla alcanzara en primer lugar un estado donde todas sus células están “muertas”, en la iteración inmediatamente posterior alcanzaría el estado deseado, es decir, con todas las células “vivas”. Esto es debido a que cómo se define en la regla, una célula requiere de 0 células “vivas” vecinas para nacer.

Algunos de los resultados anteriormente señalados, se han obtenido utilizando los siguientes parámetros, logrando el siguiente fitness:

Nº Tableros y Dimensión	**CxPB, MUPB**	**Tiempo**	**Solución**	**Fitness**
20, 10x10	0.5, 0.2	305.208 s	B58S5678	10042
40, 10x10	0.5, 0.2	309.505 s	B58S5678	10042
20, 10x10	0.5, 0.2	370.572 s	B012368 S01345	10397
20, 10x10	0.8, 0.4	543.695 s	B0123678 S0124	10589

Como se puede observar, a pesar de variar los parámetros de cruce, mutación, número de generaciones o tamaño y dimensión de la muestra, los fitness obtenidos son bastante similares. Esto concuerda con la existencia de múltiples reglas que solucionan el problema. El tiempo de ejecución, en cambio, sí se ve afectado si incrementamos las probabilidades de cruce y/o mutación.

V. CONCLUSIONES

Finalmente, tras todo el estudio teórico y experimental, hemos podido llegar varias conclusiones, empezando por la afirmación de que los algoritmos genéticos son un método totalmente válido y acertado para poder ponderar el grado de adaptación de las reglas de los autómatas celulares. Centrándonos más en la resolución de los problemas en sí, llegamos a la conclusión de que no existe ninguna regla para autómatas celulares unidimensionales capaz de resolver eficientemente el planteado problema de la mayoría, en cambio, si existe una gran variedad de reglas de autómatas celulares bidimensionales que solucionan de forma muy rentable el problema de la paridad.

Si bien es cierto que la complejidad de los estados generados y el gran número de reglas posibles, sobre todo en el segundo problema, ha desembocado en tiempos de ejecución elevados, lo cual ha podido complicar la obtención de resultados, también es cierto que los resultados han sido claros y coherentes en las distintas pruebas.

Cómo bien hemos visto en este proyecto todo puede ser optimizado, y el código no iba a ser una excepción, muy posiblemente la función fitness que hemos diseñado no sea perfecta y se podría pulir en el futuro, obteniendo resultados más precisos y pudiendo evaluar problemas más complejos.

REFERENCIAS

[1] DEAP documentation — DEAP 1.3.1 documentation

[2] Melanie Mitchell, An introduction to genetic algorithms, MIT Press, Cambridge, MA, USA, 1998.

[3] Stephen Wolfram, A new kind of science, Wolfram Media Inc., Champaign, Ilinois, USA, 2002

[4] Evolución de autómatas celulares usando algoritmos genéticos, Juan Ignacio Vázquez, Javier Oliver

[5] Generador de autómatas celulares, César H. Rodríguez G., José A. Tumialan B

[6] The Wolfram Atlas

[7] Autómatas celulares, Fernando Sancho Caparrini

[8] Autómatas celulares - Biomates

[9] Autómata celular en Python

[10] Autómata celular - Wikipedia, la enciclopedia libre