Circuitos autónomos; con la reparación en los genes

Rafael Giráldez Liébana
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
rafgirlie@alum.us.es

Jose Luis Caro Bozzino
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
joscarboz@alum.us.es

El objetivo será demostrar, mediante la aplicación de un algoritmo genético, que es posible implementar un método de autoreparación para circuitos reconfigurables.

Respecto a la conclusión extraída podemos decir que es relativamente posible, con la ayuda de este tipo de algoritmo, obtener una reparación aceptable del hardware que presente daños físicos, de manera rápida y eficiente.

I. INTRODUCCIÓN

Un circuito reconfigurable es aquel al que se le puede modificar su programación inicial y que, debido a su densidad de compuertas, es capaz de implementar casi cualquier sistema digital. No obstante, este tipo de circuito es costoso en recursos debido a los circuitos adicionales que requiere. [1]

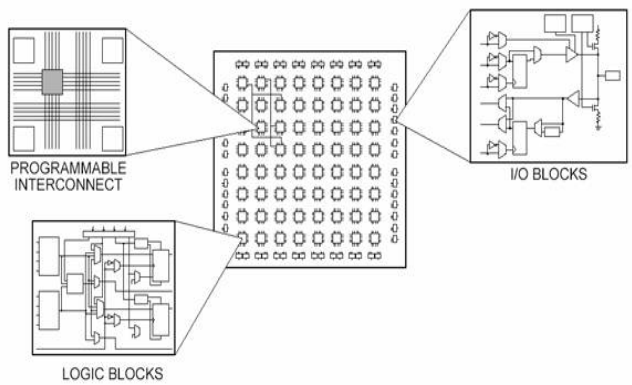


Ilustración 1. Ejemplo de circuito reconfigurable.

A priori, una reconfiguración de un circuito no parece un problema que ataña precisamente al campo de la Inteligencia artificial, sino más bien al campo de la Mecánica o la Electrónica. Pese a ello, a la hora de conseguir un hardware autónomo que administre su propia reparación encontramos que el desarrollo de algoritmos genéticos [2] tiene un papel fundamental.

Una vez realizado el estudio de diferentes algoritmos genéticos, encontramos que la forma en que se gestiona esta autoreparación por parte del hardware es muy similar a la que se realiza en un algoritmo genético a la hora de obtener los mejores individuos de una población, y que con una aplicación correcta podríamos conseguir que dicho hardware se sirviese de este algoritmo para cambiar compuertas dañadas, sustituyéndolas por otra opción más viable y que permita el correcto funcionamiento del sistema.

La problemática que recae sobre este problema es el tamaño de dichos circuitos a la hora de proponer las estrategias evolutivas de cada generación, entendiendo la palabra "generación" como cualquier cambio de compuertas que se cometa el hardware. Así, sabiendo la alta densidad de compuertas que puede llegar a alcanzar un circuito reconfigurable, lo más óptimo sería simplificar el modelo sobre el que vamos a trabajar para ver con claridad los resultados obtenidos gracias a la aplicación de un algoritmo genético.

II. MATERIAL Y MÉTODO

A. Método/s usados

Mutación y cruce[2]

Nuestro algoritmo genético se basará en dos operadores:

- **Cruce**: Será la forma de calcular los genes del nuevo inidviduo a partir de los genes de sus antecesores. En nuestro problema la primera generación de individuos tendrá n*m genes, asignados de forma aleatoria y realizarán un cruce básico. Como condición, solo podrán cruzarse los individuos que sean contiguous, y no necesariamente debe ocurrir; hemos indicado una probabilidad del 50% de que surja dicho cruce.
- **Mutación**: Será un cambio en el valor de uno o varios genes del individuo debido a un factor ajeno al propio algoritmo genético, y que conlleva aleatoriedad. En nuestro problema hemos asignado una probabilidad del 80% de que nuestros individuos muten, pues lo que se busca también es dinamismo en cada generación de la población para encontrar distintas opciones de reparación del sistema; a menor varianza, menor será el número de individuos distintos entre sí.

Inicialmente el problema contará con varios elementos que servirán para su correcto desarrollo:

- La variable de entrada (*input)* es un array de n bits con valores [0-1] que serán los que procesará la primera capa de compuertas de nuestro circuito.
- La variable de salida (*output)* es un array de n bits con valores [0-1] que representará la salida obtenida una vez que el *input* haya pasado por las compuertas correspondientes a todas las capas que hayamos asignado a nuestro circuito.

- Una matriz de tamaño n*m cuyos valores van en el rango [1-6], representando el circuito a estudiar y las puertas situadas en cada una de sus capas.
- A partir del número de compuertas (n) y el número de capas que se le asigne al circuito (m) se obtendrá un *gen* de tamaño n*m con valores [0-1] aleatorios que representa las compuertas que se le indican al algoritmo que deberá de cambiar (pues no sabe realmente que compuerta está rota).
- Como métodos auxiliares se definirán *simula_circuito*, que generará unas conexiones pseudoaleatorias para un circuito dado y nos devolverá una salida en función de ese circuito y un output , *compara_salida*, que comparará la salida real obtenida con la salida ideal, *averiar* que permitirá averiar la compuerta que le indiquemos, *fenotipo* que indicará al algoritmo cuando debe cambiar una compuerta, y por último *evaluar_fenotipo* que comparará la salida real obtenida inicialmente con la salida del fenotipo.
- Dentro de la *Toolbox*, los métodos más importantes que indicarán el proceder del algoritmo serán métodos de mutación, cruce y selección por torneo de los individuos y otro que recopile los mejores individuos de cada generación.

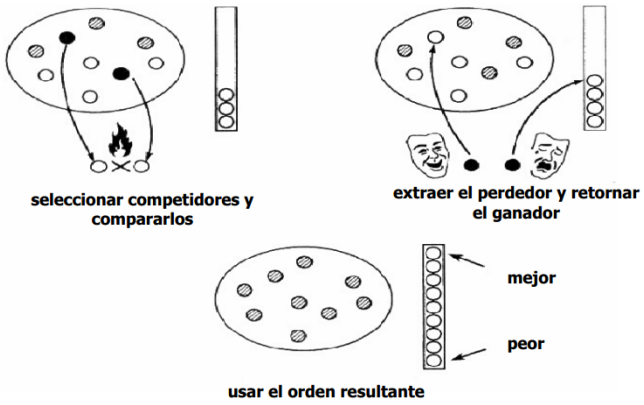


Ilustración 3. Ejemplo de elección por torneo.

B. Otras referencias y/o casos relacionados

Existen varios estudios y artículos científicos donde el uso de la Inteligencia Artificial permite tal grado de autonomía por parte de un elemento que no solo abarca el campo de la circuitería sino también el de estructuras más complejas. Casualmente uno de estos grandes avances está desarrollado gracias al uso de algoritmos genéticos, concretamente en la técnica de ensayo-error, donde un robot logra "sobrevivir" adaptando su nueva estructura, originada por daños físicos, a cualquier medio, logrando así que solo los individuos que consigan una mutación óptima logren dicho objetivo [4].

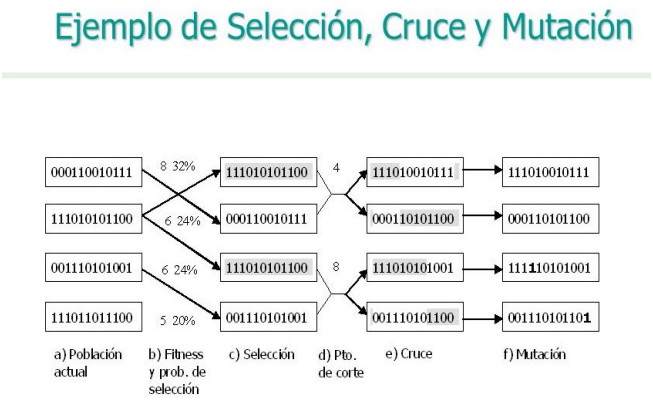


Ilustración 2. Ejemplo de mutación y cruce.

Selección por torneo [3]

Una vez que se realice la mutación y cruce de los individuos, se pasará a la elección de los más válidos por torneo.

Esta técnica consiste en seleccionar dicho individuo una vez que se compare de forma directa con otro individuo de esa misma población. En nuestro caso buscamos un individuo que provoque en el circuito solución los cambios necesarios en las puertas lógicas para obtener la solución ideal, o en su defecto acercarse lo máximo posible.

La versión que se utilizará en este caso será la determinística, tomando todos los individuos que component la población y eligiendo el más apto de todos ellos.

III. METODOLOGÍA

El problema a resolver es tal que, dado unos valores de entrada y un circuito aleatorio, el valor de salida que debe devolver este circuito debe ser el mismo ya sea si el circuito es ideal o si ha sufrido una reconfiguración por tener alguna puerta dañada. En resumen, se busca implementar un algoritmo que consiga una reparación que permita actuar de la misma manera al circuito.

Para su realización, usaremos el material y la metodología anteriormente expuesta:

- Primeramente, partimos de una o varias variables de entrada *input,* que constarán de n bits y que serán los valores de entrada para nuestro circuito.
- Introduciremos ya sea de forma manual o aleatoria (en nuestro caso se tratará de circuitos fijos o aleatorios pero con conexiones siempre pseudo-aleatorias) en forma de una matriz de n*m valores en la que cada valor asignado a una posición representará un tipo de puerta.

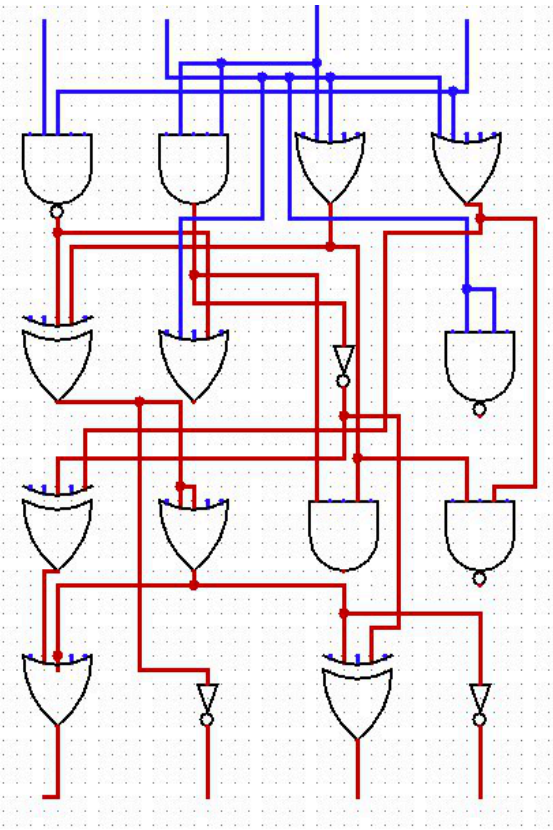


Ilustración 4. Representación gráfica del circuito solución.

$$\begin{pmatrix} 4 & 1 & 3 & 3 \\ 2 & 3 & 5 & 4 \\ 2 & 3 & 1 & 4 \\ 3 & 5 & 2 & 5 \end{pmatrix}$$

Ilustración 5 Representación numérica del circuito solución.

- A partir de una lista de inputs y ese circuito obtendremos otra lista que recogerá los outputs esperados para ese circuito tras simularlo.
- Procederemos entonces a averiar una o varias puertas del circuito original en una copia que crearemos de este. (Se sustituyen posiciones en la matriz circuito por 6s que representan puertas rotas).
- Definiremos un conjunto de individuos en forma de array que contengan n*m bits [0-1].
- Mediante el algoritmo genético obtenemos el fenotipo de cada individuo, que consistirá en convertir su array de bits en matrices de tamaño n*m de bits [1-0]. El método recorrerá esta matriz y por cada posición [i][j] que contenga un 1 cambiará en nuestro circuito averiado la puerta que se encuentre en la posición [i][j] por una aleatoria con conexiones aleatorias.
- Para evaluar este fenotipo se simularán de nuevo todos los inputs que se introdujeron al comienzo, esta vez para el circuito obtenido mediante el fenotipo, y se almacenarán en otra lista de outputs. Bit a bit se compararán los outputs del circuito original intacto con los de este nuevo circuito, calculando la calidad de ese individuo y estudiando si se ha producido una reparación completa o hasta qué punto coinciden las salidas.
- Mediante este mecanismo, mutación de individuos y técnicas de cruzamiento se procederá a un mecanismo de selección por torneo entre un número determinado de generaciones de individuos, que culminará en un hall of fame que muestre a los 3 mejores individuos.
- Para cada individuo en el hall of fame, se mostrará el circuito completo, sus conexiones y el número de salidas que coinciden en su totalidad con las del circuito original; mostrando un mensaje de reparación completa en dicho caso.

Teniendo en cuenta todo lo anteriormente expuesto, hemos procedido a su implementación.

Para realizar dicha implementación se ha utilizado Python [5] como lenguaje de programación y las librerías Numpy [6] y Deap [7]. La primera de ellas nos permitirá disponer de un mayor rango de funciones en el uso de matrices, mientras que la segunda nos permitirá implementar el algoritmo genético y los métodos auxiliares de cruce y mutación gracias a las herramientas y estructuras de datos de las que dispone.

Por otro lado utilizaremos el entorno de trabajo interactivo Jupyter Notebook [8] para el desarrollo del código del problema.

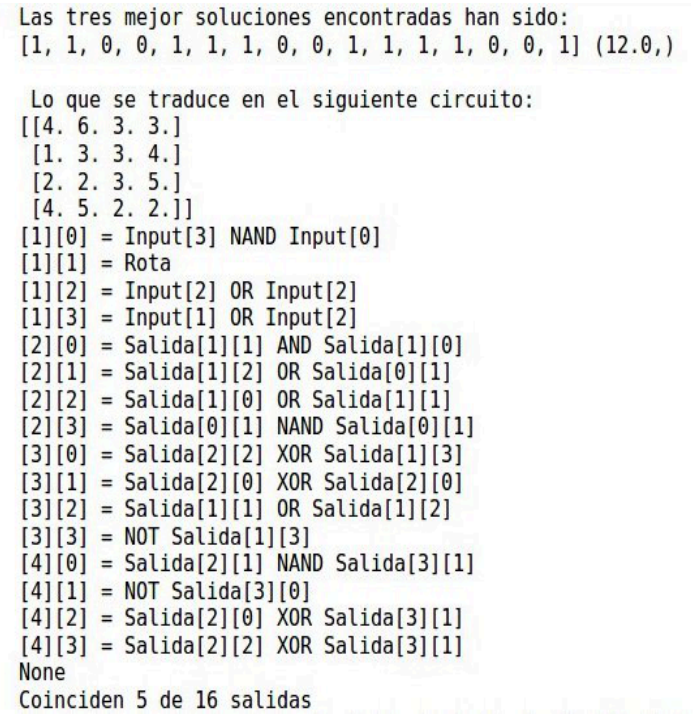
IV. RESULTADOS

Para exponer los pasos que realiza el algoritmo y su posterior solución usaremos el entorno anteriormente citado.

Puesto que el problema requiere de un circuito solución se ha decido variar el número de bits que se utilizarán en el *input* para registrar los distintos valores de calidad. Estos resultados se obtienen una vez que se ha averiado de forma intencionada varias puertas para que el uso del algoritmo genético sea efectivo.

Para la resolución que queremos mostrar se han utilizado 16 bits en el *input,* mostrando una calidad de 12,0. Este valor, dividido entre el número total de *inputs* nos dará el

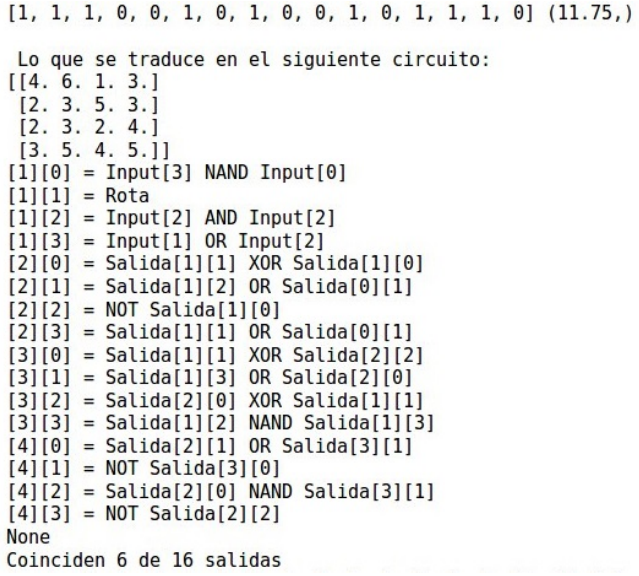
porcentaje de calidad. Este valor será mostrado junto a la primera reparación realizada.



```
Las tres mejor soluciones encontradas han sido:
[1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1] (12.0,)

 Lo que se traduce en el siguiente circuito:
[[4. 6. 3. 3.]
 [1. 3. 3. 4.]
 [2. 2. 3. 5.]
 [4. 5. 2. 2.]]
[1][0] = Input[3] NAND Input[0]
[1][1] = Rota
[1][2] = Input[2] OR Input[2]
[1][3] = Input[1] OR Input[2]
[2][0] = Salida[1][1] AND Salida[1][0]
[2][1] = Salida[1][2] OR Salida[0][1]
[2][2] = Salida[1][0] OR Salida[1][1]
[2][3] = Salida[0][1] NAND Salida[0][1]
[3][0] = Salida[2][2] XOR Salida[1][3]
[3][1] = Salida[2][0] XOR Salida[2][0]
[3][2] = Salida[1][1] OR Salida[1][2]
[3][3] = NOT Salida[1][3]
[4][0] = Salida[2][1] NAND Salida[3][1]
[4][1] = NOT Salida[3][0]
[4][2] = Salida[2][0] XOR Salida[3][1]
[4][3] = Salida[2][2] XOR Salida[3][1]
None
Coinciden 5 de 16 salidas
```

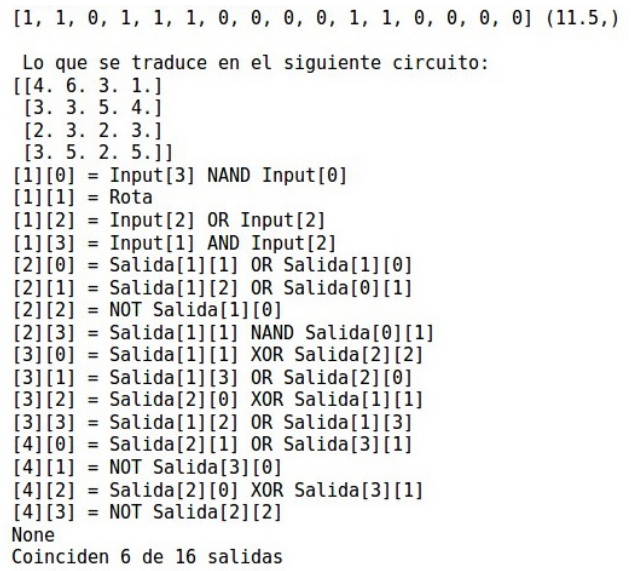
Ilustración 6. Primera reparación realizada por el algoritmo.



```
[1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0] (11.75,)

 Lo que se traduce en el siguiente circuito:
[[4. 6. 1. 3.]
 [2. 3. 5. 3.]
 [2. 3. 2. 4.]
 [3. 5. 4. 5.]]
[1][0] = Input[3] NAND Input[0]
[1][1] = Rota
[1][2] = Input[2] AND Input[2]
[1][3] = Input[1] OR Input[2]
[2][0] = Salida[1][1] XOR Salida[1][0]
[2][1] = Salida[1][2] OR Salida[0][1]
[2][2] = NOT Salida[1][0]
[2][3] = Salida[1][1] OR Salida[0][1]
[3][0] = Salida[1][1] XOR Salida[2][2]
[3][1] = Salida[1][3] OR Salida[2][0]
[3][2] = Salida[2][0] XOR Salida[1][1]
[3][3] = Salida[1][2] NAND Salida[1][3]
[4][0] = Salida[2][1] OR Salida[3][1]
[4][1] = NOT Salida[3][0]
[4][2] = Salida[2][0] NAND Salida[3][1]
[4][3] = NOT Salida[2][2]
None
Coinciden 6 de 16 salidas
```

Ilustración 7. Segunda reparación realizada por el algoritmo.



```
[1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0] (11.5,)

 Lo que se traduce en el siguiente circuito:
[[4. 6. 3. 1.]
 [3. 3. 5. 4.]
 [2. 3. 2. 3.]
 [3. 5. 2. 5.]]
[1][0] = Input[3] NAND Input[0]
[1][1] = Rota
[1][2] = Input[2] OR Input[2]
[1][3] = Input[1] AND Input[2]
[2][0] = Salida[1][1] OR Salida[1][0]
[2][1] = Salida[1][2] OR Salida[0][1]
[2][2] = NOT Salida[1][0]
[2][3] = Salida[1][1] NAND Salida[0][1]
[3][0] = Salida[1][1] XOR Salida[2][2]
[3][1] = Salida[1][3] OR Salida[2][0]
[3][2] = Salida[2][0] XOR Salida[1][1]
[3][3] = Salida[1][2] OR Salida[1][3]
[4][0] = Salida[2][1] OR Salida[3][1]
[4][1] = NOT Salida[3][0]
[4][2] = Salida[2][0] XOR Salida[3][1]
[4][3] = NOT Salida[2][2]
None
Coinciden 6 de 16 salidas
```

ación 8. Tercera reparación realizada por el algoritmo.

os resultados del hall of fame observamos que a pesar e el primero tiene una mayor calidad (12 puntos sobre ólo tiene una coincidencia absoluta en 5 de las 16 s para las que se ha programado la prueba, mientras anto el segundo como el tercer circuito sugerido tienen untuación menor (11.75-11.5), pero sin embargo tienen una mayor coincidencia absoluta de salidas (6 de 16 contra 5 que devuelve el circuito elegido como mejor).

Esto se debe a que según nuestra forma de entender el problema, la calidad no debía medir el número de salidas totalmente iguales que las devueltas por el original, sino el número de bits en el que coincide cada salida.

Es por esto mismo que ocurre este caso en el que el mejor circuito alternativo devuelve mayor puntuación pero menor número de salidas similares, implicando que aunque solo 5 de 16 salidas sean iguales a las esperadas en el circuito original, el resto de salidas deben ser extremadamente parecidas aunque no totalmente idénticas.

Esta calidad ha sido la elegida entre todas las calidades generadas en cada generación. Para este ejemplo se han utilizado 20 generaciones, siendo en la generación 6 donde se obtiene la calidad máxima más baja y en la generación 7 la obtenida anteriormente.

gen	nevals	mínimo	media	máximo
0	10	6.75	8.4	10.25
1	9	6	8.325	10
2	9	5.5	8.25	11.5
3	9	5.5	8.075	10
4	9	6.25	8.25	10.25
5	9	6.25	8.075	10.25
6	10	7	8.325	9.5
7	10	7	8.825	12
8	10	6	8.05	10.5
9	10	6.75	8.1	11
10	10	6	7.85	10
11	9	7.25	8.7	10
12	9	6.75	8.25	10.5
13	9	6	8.725	10.5
14	10	5.5	7.75	10.5
15	10	5.5	7.8	11
16	10	5.5	8.4	11.5
17	10	5.5	7.825	11.75
18	10	6.5	8.25	10
19	9	6	8.075	10
20	10	7	8.55	10.25

Ilustración 9. Calidades obtenidas en cada generación del proceso.

Nº de inputs	Calidad media
1	0'725
2	0'85
3	1'8
4	1'95
5	2'8
6	2'8
7	3'675
8	4'15
9	4'825
10	5'075
11	6'25
12	7'75
13	6'95
14	6'675
15	8'675
16	8'55

Ilustración 10. Calidades media en función de los inputs.

Mediante esta tabla podemos obtener el descenso en la calidad media de la reparación (es decir la similitud entre los *outputs* que estamos comparando), pasando de un 72,5% con un solo *input* a un 53,43% con 16. Esto no hace más que confirmar la dificultad ascendente que tendrá que sufrir el algoritmo a la hora de darnos la reparación más óptim

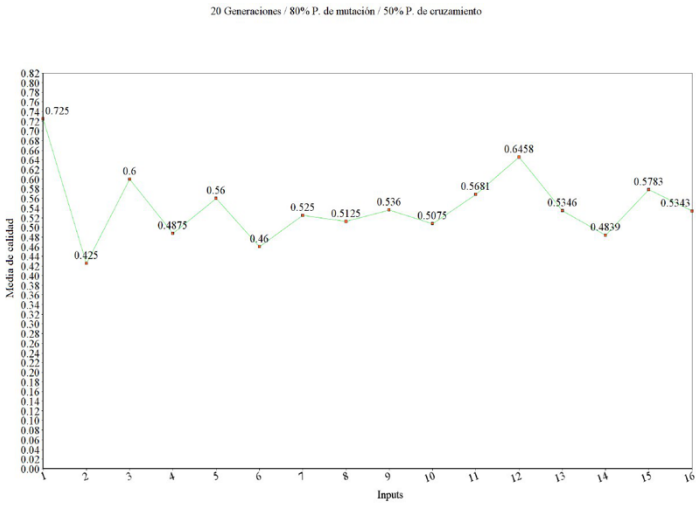


Ilustración 11. Calidades media en tanto por uno obtenidas.

Esta gráfica representa la tabla anteriormente mostrada pero con valores en tanto por uno. Muestra que a mayor número de input menor es la media de calidad tras 20 generaciones; esto debido a que mientras más inputs sean introducidos inicialmente, más exacto e individual será el funcionamiento del circuito, y por tanto mayor complicación encontrará el AG para hallar circuitos alternativos que funcionen totalmente igual. Es por esto que el proyecto sería útil para pequeños circuitos que se encarguen de una o dos funciones, pero no para replicar circuitos que tengan que tener un funcionamiento exacto para cualquier input.

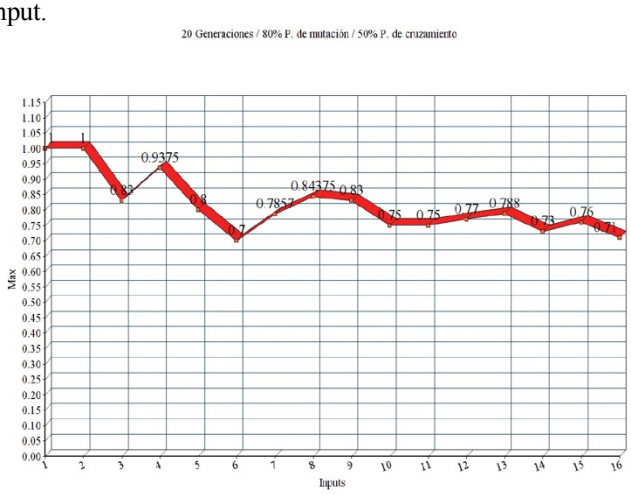


Ilustración 12. Calidades máxima en tanto por uno obtenidas.

En esta última gráfica se representa lo mismo que en la anterior, pero esta vez con la mayor calidad que se encuentra entre todas las 20 generaciones según aumenta el número de inputs. La tendencia es descendente aunque tiene varios repuntes, lo cual es normal puesto que un algoritmo genético no deja de tener un componente aleatorio bastante importante.

Nos ha parecido interesante que, pese a intentar subir la dificultad del problema añadiendo más puertas dañadas al circuito, no hemos conseguido un descenso de la calidad de las soluciones. Intuímos que esto sucede debido a la componente aleatoria antes citada y que, al encontrarse tal número de puertas dañadas, más que una reparación lo que se termina obteniendo es un nuevo circuito alternativo que aproxima su salida a la del original.

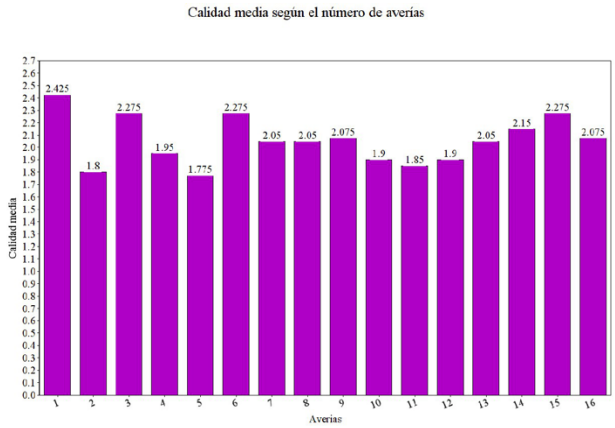


Ilustración 13. Calidades media en función de averías halladas.

Concretamente, la calidad media tiene su máximo en un 60,62% y su mínimo en un 44,37% pero no sigue un descenso progresivo sino que aumenta y disminuye en función de la generación en la que se halle, y como dijimos anteriormente, suponemos que se debe a la componente aleatoria a la que está sujeta el algoritmo genético.

V.CONCLUSIÓN

Tras haber realizado múltiples pruebas con distintos valores tanto en el número de averias como en el número de generaciones y el número de *inputs*, podemos decir que el algoritmo genético que hemos implementado consigue efectivamente una reparación semi-óptima para nuestro circuito solución. Pese a ello, su función no era reparar al completo y de forma perfecta el circuito sino aproximar su salida lo máximo posible a la salida ideal. Con pocos *inputs* y pocas averías el algoritmo sería capaz de alcanzar una solución perfecta con relativamente pocas generaciones.

Sin embargo, ya hemos visto que ha medida que aumenta la complejidad del problema, más dificil será obtener dicha similitudes entre salidas.
Como futuras mejoras podría implementarse una interfaz gráfica que permitiese ver representadas las puertas y conexiones del circuito, marcando las averiadas a base de clicks en vez de introducirlas a mano, a la vez que pudiésemos ver cómo el algoritmo cambia puertas dañadas y realiza la elección del mejor gen para cambiarlas.
A efectos prácticos, con ejemplos más realistas, nuestro algoritmo no tiene la potencia computacional necesaria para la resolución de circuitos a gran escala pero sí que nos permite ver, de forma muy simplificada, cómo realizaría dichas operaciones de reparación, sirviendo así de ayuda para comprender major los algoritmos genéticos.

REFERENCES

[1] Tema 1 – Introducción a la Lógica Reconfigurable – Instituto de Electrónica y Mecatrónica – Universidad Tecnológica de la Mixteca
http://www.utm.mx/~fsantiag/Cir_Dig_Rec/01_Introduccion.pdf

[2] Wikipedia – Algoritmo genético
https://es.wikipedia.org/wiki/Algoritmo_genético#Algoritmos_evolutivos

[3] Wikipedia – Selección por Torneos
https://es.wikipedia.org/wiki/Selección_por_torneos

[4] Evolving a Behavioral Repertoire for a Walking Robot – By A.Cully and J.B. Mouret
https://www.mitpressjournals.org/doi/abs/10.1162/EVCO_a_00143

[5] Página oficial del lenguaje de progrmación Python
https://www.python.org

[6] Página oficial de la librería Numpy
http://www.numpy.org

[7] Página oficial de la librería Deap
https://deap.readthedocs.io/en/master/

[8] Página oficial de Jupyter Notebook
http://jupyter.org