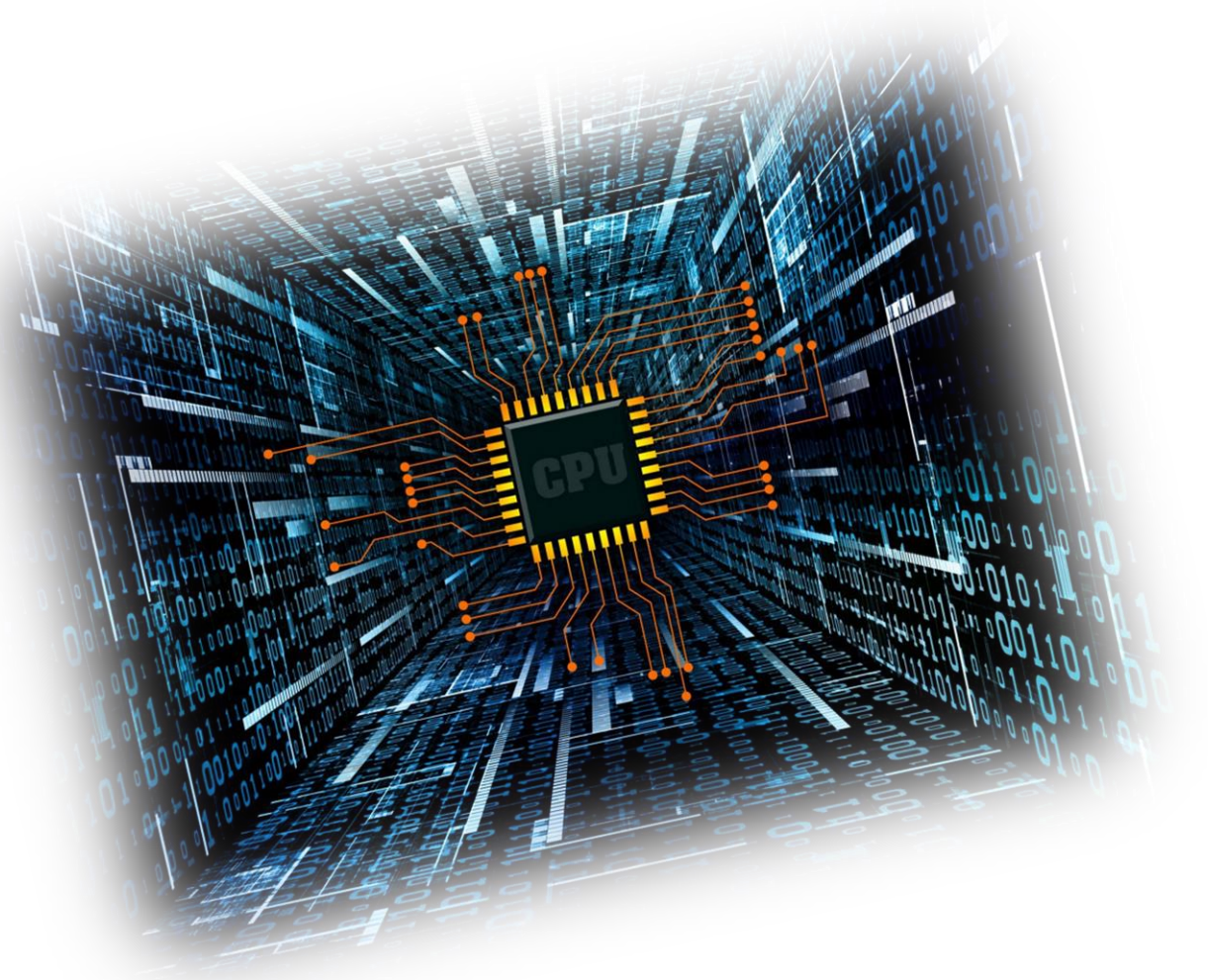
REPARACIÓN AUTOMÁTICA DE CIRCUITOS RECONFIGURABLES



Asen Rangelov Baykushev

dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Sevilla, España

aseranbay@alum.us.es

Javier Herraiz Olivas

dpto. Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Sevilla, España

javheroli@alum.us.es

RESUMEN

Este trabajo práctico trata de diseñar e implementar un sistema evolutivo de autoreparación. El sistema debe ser capaz de reparar circuitos digitales reconfigurables que están físicamente dañados. La reparación consiste en encontrar una configuración alternativa de las puertas lógicas del circuito y de las conexiones entre ellas de tal manera que el circuito resultante se comporte igual que el circuito original a nivel global. A lo largo del proceso de diseño del sistema evolutivo se implementan otros mecanismos. En concreto, un simulador de circuitos digitales capaz de calcular la salida de un circuito digital a partir de la entrada y un mecanismo de autodiagnóstico cuyo objetivo es detectar posibles fallos físicos en un circuito digital.

Una vez implementado el sistema, se ha procedido a analizar su rendimiento. Para ello, se han llevado a cabo varios experimentos. Se han usado varios circuitos digitales con distintos daños físicos. Se ha comprobado que el sistema repara el circuito dañado siempre que sea posible. Si el circuito no se puede reparar, el sistema realiza una reparación óptima.

Palabras Clave: Inteligencia Artificial, circuito reconfigurable, puertas lógicas, conexiones, vector de entrada, vector de salida, daños, comportamiento correcto, comportamiento incorrecto, autodiagnóstico, autoreparación, algoritmo genético, genes, genotipo, fenotipo, cromosoma, cruzamiento, mutación

CONTENIDO

I. INTRODUCCIÓN

En los últimos años el avance en la microelectrónica ha posibilitado el disponer de circuitos programables y reconfigurables con una gran variedad de recursos lógicos. En un circuito reconfigurable cada elemento lógico (puertas lógica y conexiones entre ellas) puede ser programado para comportarse de una determinada forma, consiguiendo de este modo que el circuito adopte un comportamiento específico a nivel global.

El objetivo fundamental de este trabajo es diseñar e implementar un mecanismo que detecte si un circuito configurado para actuar de cierta manera ha sufrido algún daño físico que provoque un comportamiento incorrecto del mismo. En caso afirmativo, dicho mecanismo ha de ser capaz de cambiar de manera automática la configuración de las puertas y las conexiones no dañadas de tal manera que el circuito vuelva a adoptar su comportamiento normal a nivel global, pese a la presencia de elementos físicamente dañados. En resumen, la idea de la propuesta de trabajo es dotar a un circuito reconfigurable de la posibilidad de autodiagnosticarse y autorepararse.

A continuación, se describen brevemente los métodos y las técnicas empleadas para realizar el trabajo. Después se procede a explicar las decisiones de diseño y la metodología seguida. Acto seguido, se detallan los experimentos llevados a cabo y se analizan los resultados. Por último, se incluyen las conclusiones extraídas del análisis.

II. PRELIMINARES

Métodos empleados: (se listan sólo los más importantes)

- **Estructura de datos**: en primer lugar es necesario disponer de una estructura de datos que represente un circuito digital con una determinada configuración de puertas lógicas y conexiones entre ellas.
- **salida_Puerta(TipoPuerta, entrada1, entrada2)** y **salida_Circuito(Circuito, Entrada, Puertas_defectuosas, Conexiones_defectuosas)**: Se ha de implementar un mecanismo que simule el comportamiento del circuito de manera que, partiendo de cualquier vector de entrada, se obtenga el vector de salida correspondiente.
- **construir_pares_referencia()** y **autodiagnostico(Circuito, Puertas_defectuosas, Conexiones_defectuosas)**: El paso siguiente será implementar un método de autodiagnóstico que sea capaz de detectar daños físicos en el circuito digital. Dicho método debe operar sin conocer cuáles son las

puertas y las conexiones dañadas (si los hubiera), por lo que es necesario el uso de datos de referencia que representen el comportamiento correcto del circuito.

- **generador_genes()**, **fenotipo(individuo)** y **evaluar_individuo(individuo)**: Una vez implementado el método de autodiagnóstico, se procederá a diseñar el mecanismo de autoreparación. Se hará uso de un algoritmo genético [1] cuyo objetivo será encontrar (si es posible) una configuración alternativa de las puertas y las conexiones que no resulten dañadas, de tal forma que el nuevo circuito actúe igual que el anterior. De nuevo, el algoritmo no conoce cuáles son los daños introducidos en el circuito.

III. METODOLOGÍA

A. ESTRUCTURA DE DATOS

Como se ha mencionado anteriormente, lo primero que hay que hacer es definir una estructura de datos que simule un circuito configurado de cierta manera. Para representar un circuito digital basta con definir cada una de las puertas que lo compongan, así como las conexiones entre ellas. Para este trabajo se ha asumido que el circuito es rectangular y está formado por **m.n** puertas lógicas, donde **m** es el número de capas y **n** es el número de puertas lógicas que componen cada capa. Intuitivamente, la entrada y la salida del circuito son vectores de **n** bits. Una restricción muy importante es que cada puerta puede estar conectada únicamente con puertas de las dos capas anteriores. Esto significa, por ejemplo, que las puertas de la capa **m=2** pueden recibir como entradas bits provenientes del vector de entrada, pero las de la capa **m=3** no lo pueden.

Para definir cada puerta del circuito se usará una tupla de 3 elementos. El primer elemento de la tupla indicará el tipo de la puerta (‘AND’, ’OR’, ’NOT’, ‘NAND’, ‘NOR’, ‘XOR’ o ‘XNOR’). El segundo elemento será un número que servirá para identificar cada puerta del circuito de manera unívoca. El tercer elemento será una lista de dos pares de números que indicarán las puertas de las capas anteriores (o los bits de entrada) con las que la puerta está conectada mediante sus dos entradas. A efectos de simplicidad, se ha supuesto que cada puerta tiene dos entradas (la segunda entrada de las puertas ‘NOT’ siempre está desconectada y nunca se toma en cuenta para calcular la salida).

La siguiente figura ilustra con detalle los elementos de una tupla:

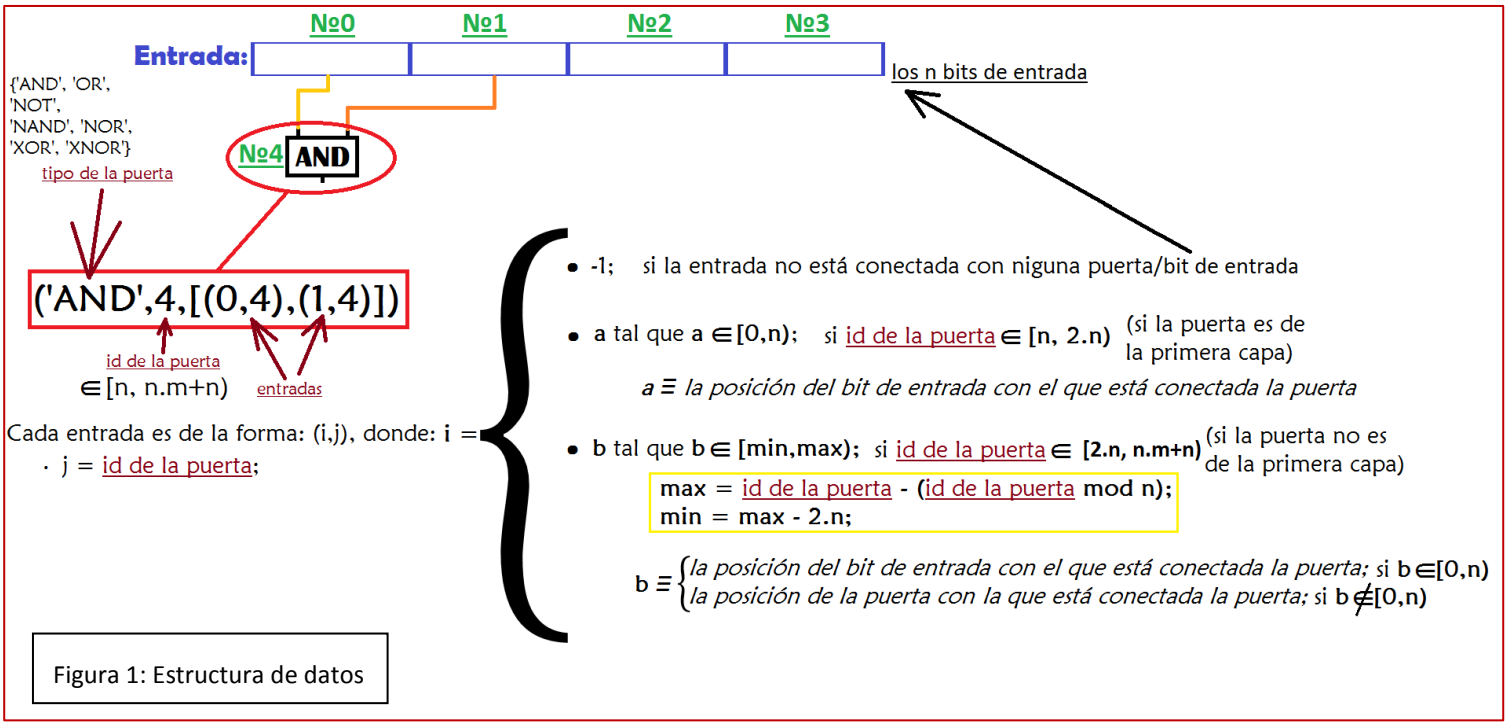


Figura 1: Estructura de datos

A continuación se muestran más ejemplos de tuplas:

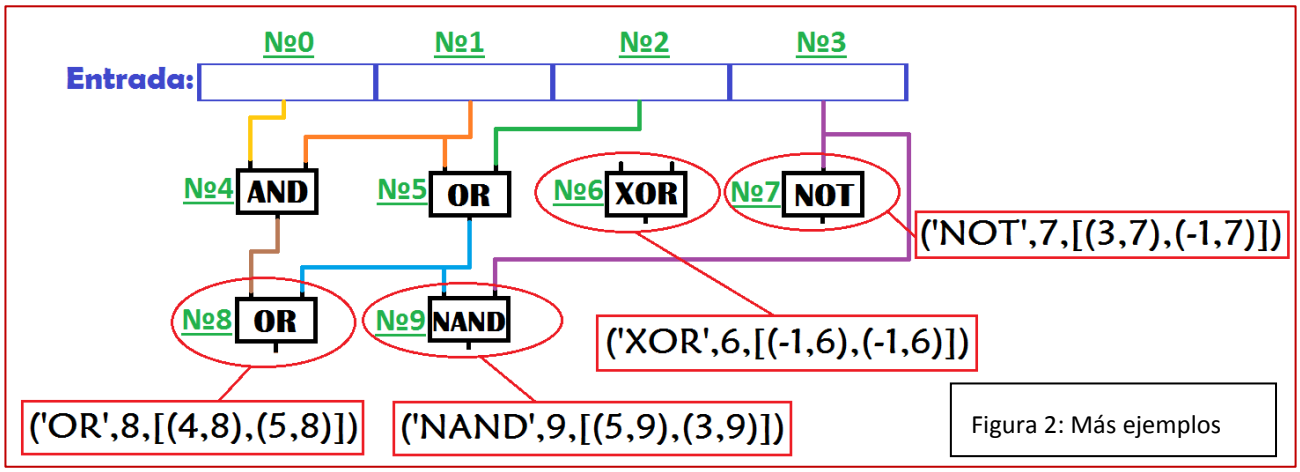


Figura 2: Más ejemplos

Por consiguiente, para representar un circuito digital se necesita un conjunto de **n.m** tuplas – una por cada puerta del circuito.

B. SIMULADOR DE CIRCUITOS

Una vez definida la estructura de datos, se procede a implementar un método que simule el comportamiento del circuito digital. Esto es, dado cualquier vector de entrada, dicho método ha de devolver la salida correspondiente. Antes que nada, se implementa un método auxiliar **salida_Puerta (TipoPuerta, entrada1, entrada2)**. Este método simplemente calcula la salida que debe devolver una puerta en función de su tipo y las entradas, usando para ello la expresión lógica correspondiente a la tabla de verdad de la puerta. Cabe mencionar que si la entrada de una puerta está desconectada (esto es, el número que indica la conexión es -1), se considera que dicha entrada recibe un 0. El método que actuará como simulador de circuito es **salida_Circuito (Circuito, Entrada, Puertas_defectuosas, Conexiones_defectuosas)**. Este método recibe el conjunto de tuplas que representa el circuito y un vector de entrada para devolver el vector de salida

correspondiente. El método recorre las puertas del circuito empezando por las de la primera capa. Conforme las puertas se van recorriendo, se van calculando las salidas correspondientes (haciendo uso del susodicho método auxiliar) y se van guardando los resultados en una lista. Esta lista se consulta para calcular las salidas de las puertas de la siguiente capa, y así sucesivamente, hasta llegar a la última capa del circuito cuando el recorrido termina y como resultado se devuelve las salidas de las puertas de la última capa.

Este método también permite simular daños en el circuito, puesto que incluye una opción de marcar ciertas puertas y/o conexiones como "defectuosas" y calcular la salida correspondiente a la entrada teniendo en cuenta los daños introducidos. Las puertas marcadas como "defectuosas" siempre devuelven un 0 como salida, independientemente de su tipo o sus entradas. Las conexiones marcadas como "defectuosas" hacen que la entrada correspondiente reciba siempre un 0 (como si la entrada estuviera desconectada). Para marcar una puerta como "defectuosa" basta con introducir su número identificador en la lista **Puertas_defectuosas** que el método recibe como parámetro de entrada. Asimismo, para marcar una conexión como "defectuosa" hay que introducir en la lista **Conexiones_defectuosas** el par de números que referencian los elementos conectados a los dos extremos de la conexión.

C. AUTODIAGNÓSTICO

El siguiente paso será implementar el método de autodiagnóstico. Tal y como se ha mencionado en las preliminares, se debe implementar un mecanismo que sea capaz de detectar posibles daños en el circuito que provoquen un comportamiento incorrecto del mismo. Para conseguir este objetivo es necesario encontrar alguna forma de representar el comportamiento correcto del circuito, comparar dicho comportamiento con el comportamiento actual que manifieste el circuito y comprobar si hay discrepancias.

Se procede de la siguiente manera: se implementa un método auxiliar **construir_pares_referencia()** cuyo objetivo es crear una lista de tuplas entrada-salida que servirá de referencia para representar el comportamiento correcto del circuito. Para ello, se hace uso de la función **product()** del módulo **itertools** [2] de Python para construir una lista de longitud 2^n con todas las posibles entradas del circuito (esto es, todas las combinaciones de **n** bits). Luego, la lista de entradas se recorre y a cada entrada se le asigna la salida correspondiente. Para calcular la salida correcta correspondiente a cada una de las entradas, se utiliza el método **salida_Circuito (Circuito, Entrada, Puertas_defectuosas, Conexiones_defectuosas)**. Es importante no marcar ninguna puerta o conexión como "defectuosa" ya que el objetivo de este método auxiliar es representar el comportamiento correcto del circuito. Después se implementa el método **autodiagnóstico (Circuito, Puertas_defectuosas, Conexiones_defectuosas)**. Dicho método va recorriendo la lista de tuplas entrada-salida devuelta por el método auxiliar y va calculando la salida correspondiente a cada una de las entradas. No obstante, esta vez se tienen en cuenta las listas de puertas y conexiones marcadas como "defectuosas" (las

cuales se pasan como parámetro de entrada). Según va calculando las salidas, el método va comprobando si la salida recién calculada coincide con la salida calculada por el método auxiliar. Si las dos salidas coinciden, la presencia de daños no influye en el comportamiento correcto del circuito (con respecto a la entrada que se esté procesando). Sin embargo, si las dos salidas son distintas, los daños han provocado que el circuito digital se comporte de forma incorrecta. En este último caso se ha de incrementar en 1 un contador previamente definido para indicar el número de discrepancias entre salidas esperadas y salidas devueltas, ya que para la implementación del mecanismo de autoreparación es necesario saber no sólo si el circuito está dañado, sino el impacto de los daños presentes en el mismo.

D. AUTOREPARACIÓN

Por último, se implementará un mecanismo que, partiendo de un circuito con daños en las puertas y/o las conexiones y haciendo uso de un algoritmo genético, cambie la configuración de las restantes puertas y conexiones (las no dañadas) con el fin de construir un circuito digital alternativo que "imite" completamente el comportamiento del circuito original (antes de ser dañado). Dicho mecanismo debe trabajar sin conocer qué puertas y conexiones son las dañadas.

Para implementar el algoritmo genético se ha usado el paquete **DEAP** de Python y se ha seguido la metodología proporcionada en la Práctica 4 del curso IA de Ingeniería del Software (curso 2017/18) [3]. A continuación se explicarán únicamente las decisiones de diseño más importantes que se han tomado.

Puesto que el objetivo del algoritmo genético es devolver un circuito alternativo, es fácil deducir que el tipo de cromosomas que se ha de usar debe ser capaz de representar la configuración completa de puertas y conexiones de un circuito alternativo. Es decir, los individuos de la población serán circuitos alternativos diferentes y el algoritmo genético se encargará de elegir el mejor de ellos. Para representar un individuo (esto es, un circuito digital) se utilizará un cromosoma de **3.n** genes. Se implementa el método **generador_genes()** para definir los posibles valores que puede tomar cada gen del cromosoma. Cada grupo de tres genes representará una puerta del circuito alternativo. El primer gen de cada grupo indicará el tipo de la puerta, mientras que los dos restantes genes referenciarán los elementos (bits de entrada u otras puertas) conectados con las dos entradas de la puerta. La siguiente figura ilustra detalladamente el genotipo.

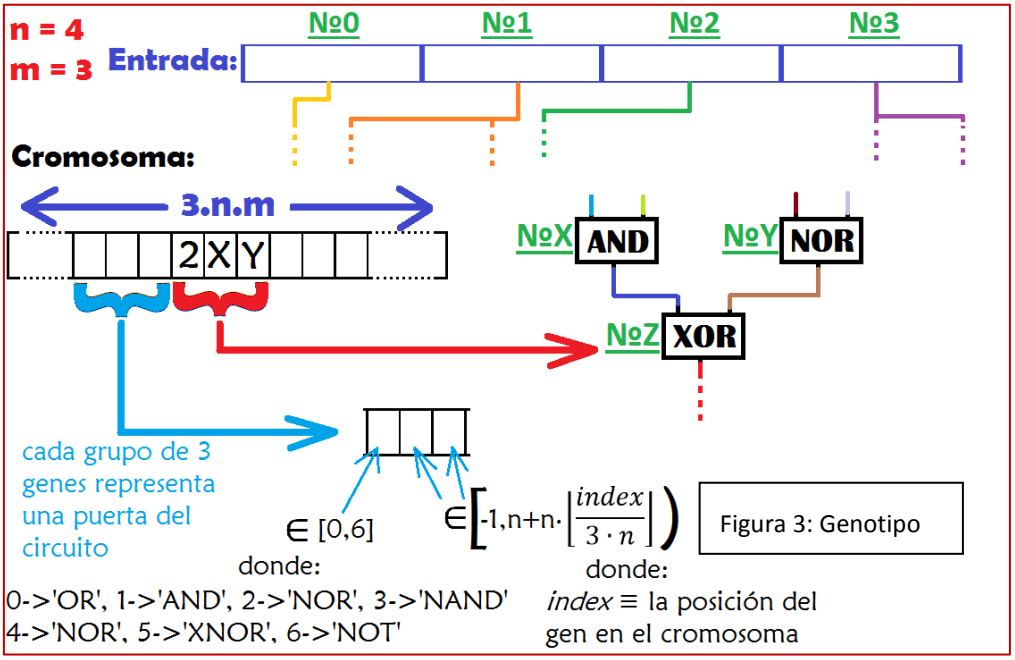


Figura 3: Genotipo

Cabe hacer notar que este diseño imposibilita que se generen cromosomas erróneos, en el sentido de que exista la posibilidad de que una puerta tenga como entrada la salida de otra puerta de la misma capa o una capa posterior. No obstante, aún es posible que se generen cromosomas erróneos en el sentido de que la entrada de una puerta provenga de un elemento no perteneciente a las 2 capas anteriores. Esta situación se penalizará en la función de aptitud.

El método **fenotipo (individuo)** que sirve para descodificar el genotipo y construir el fenotipo de un individuo es muy intuitivo. Simplemente se recorren todos los genes del cromosoma en grupos de tres en tres y se van construyendo las puertas del nuevo circuito alternativo.

La siguiente figura muestra el fenotipo (esto es, un circuito digital) correspondiente a un cromosoma generado por el algoritmo.

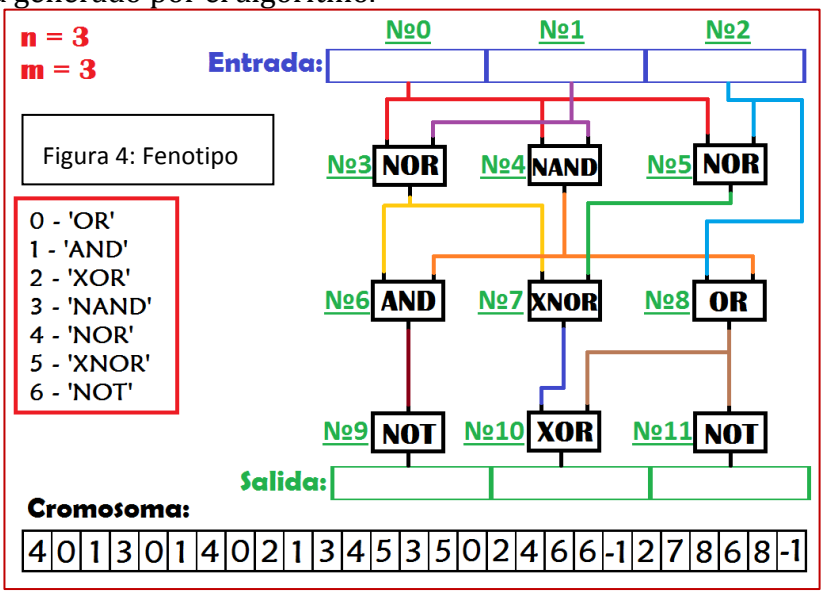


Figura 4: Fenotipo

Conviene mencionar que se ha posibilitado que el algoritmo genere puertas 'NOT' cuya segunda entrada no esté desconectada. Además, el algoritmo no sabe qué puertas y/o conexiones son las dañadas, por lo que crea una configuración aleatoria sin ningún tipo

de restricción acerca de qué puertas y conexiones se pueden usar y cuáles no. No obstante, ninguno de estos dos tipos de incoherencia influye en el cálculo correcto de la función de aptitud.

La función de aptitud la implementa el método **evaluar_individuo(individuo)** cuyo objetivo es evaluar cada individuo de la población calculando su valor de **fitness**. El **fitness** de un individuo se calcula en función del grado en el que el circuito alternativo representado por el individuo "imita" el comportamiento correcto del circuito original, así como en función del número de conexiones inválidas. Una conexión se considera inválida si conecta una de las entradas de una puerta perteneciente a una determinada capa con un elemento que no pertenezca a ninguna de las dos capas anteriores. El **fitness** de un individuo que "imita" perfectamente el comportamiento del circuito original y no tenga conexiones inválidas es 0.

A continuación se muestra la función de aptitud en pseudocódigo:

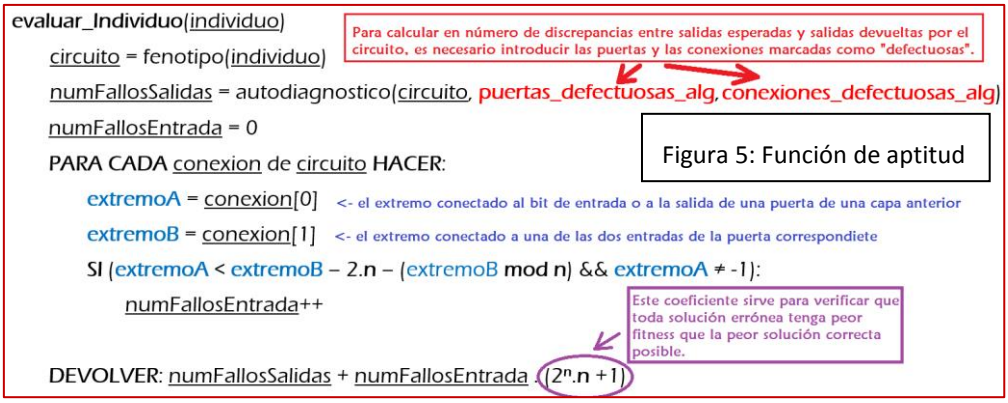


Figura 5: Función de aptitud

Finalmente, se debe especificar qué tipo de operador de cruzamiento y operador de mutación se utilizará en el algoritmo genético. Tal y como se ha diseñado la estructura del cromosoma, un operador de cruzamiento bueno es la recombinación uniforme. Al cruzarse dos cromosomas, habrá una posibilidad de 10% de que dos genes que están en la misma posición en los dos cromosomas se intercambien. El uso de este operador de cruzamiento imposibilitará que tras el cruzamiento aparezcan individuos con genes que tomen valores fuera del correspondiente rango de valores permitidos. En cuanto al operador de mutación, se utilizará la mutación uniforme (**mutUniformInt(individual, low, up, indpb)** [4]) . Habrá una posibilidad de 10% de que un gen del cromosoma cambie su valor por uno aleatorio. No obstante, es primordial establecer correctamente el intervalo dentro del cual dicho valor aleatorio puede variar. El intervalo de valores permitidos que cada gen pueda tomar se calcula en función de la posición del gen en el cromosoma. Estos intervalos deben ser los mismos que los que se han usado para diseñar el genotipo (ver Figura 3).

IV. RESULTADOS

En esta sección se exponen los resultados de varios experimentos. Se han llevado a cabo 4 experimentos usando 2 circuitos diferentes. En las tablas se han recogido las estadísticas más relevantes – el fitness del mejor individuo de cada generación y el valor medio del fitness de todos los individuos de la generación.

"Circuito1": (n=3, m=4)

```
[('OR', 3, [(0, 3), (1, 3)]), ('NOT', 4, [(1, 4), (-1, 4)]), ('XNOR', 5, [(1, 5),
(2, 5)]), ('NAND', 6, [(3, 6), (5, 6)]), ('OR', 7, [(3, 7), (4, 7)]), ('AND', 8,
[(-1, 8), (-1, 8)]), ('XOR', 9, [(3, 9), (6, 9)]), ('NOR', 10, [(7, 10), (8, 10)]),
('AND', 11, [(7, 11), (5, 11)]), ('XNOR', 12, [(9, 12), (6, 12)]), ('NOT', 13,
[(10, 13), (-1, 13)]), ('NOR', 14, [(7, 14), (11, 14)])]
```

"Circuito2": (n=3, m=3)

```
[('NOR', 3, [(0, 3), (1, 3)]), ('NAND', 4, [(0, 4), (1, 4)]), ('NOR', 5, [(0, 5), (
2, 5)]), ('AND', 6, [(3, 6), (4, 6)]), ('XNOR', 7, [(3, 7), (5, 7)]), ('OR', 8, [(2
, 8), (4, 8)]), ('NOT', 9, [(6, 9), (-1, 9)]), ('XOR', 10, [(7, 10), (8, 10)]), ('N
OT', 11, [(8, 11), (-1, 11)])]
```

Experimento 1		
Circuito: "Circuito1"		
Puertas dañadas: [4,11]		
Conexiones dañadas: [(3,6), (6,12)]		
gen.	mín.	media
0	8	127.6
1	6	92.826
2	4	68.477
3	4	50.603
4	2	35.676
5	2	24.383
6	2	16.393
7	0	12.782
8	0	10.96
9	0	9.438
10	0	8.7085
Rendimiento: 100%		

Experimento 2		
Circuito: "Circuito1"		
Puertas dañadas: [9,12]		
Conexiones dañadas: [(7,10), (5,11), (7,14)]		
gen.	mín.	media
0	4	124.768
1	2	91.114
2	2	66.9
3	2	49.018
4	2	33.93
5	2	22.39
6	2	14.212
7	2	10.108
8	2	8.328
9	2	7.2355
10	2	7.075
Rendimiento: 91.67%		

Experimento 3		
Circuito: "Circuito2"		
Puertas dañadas: [7]		
Conexiones dañadas: [(2,8), (4,6)]		
gen.	mín.	media
0	6	57.255
1	6	35.561
2	4	21.501
3	4	13.028
4	3	10.14
5	3	9.159
6	3	8.821
7	2	8.562
8	2	8.121
9	2	7.637
10	2	7.49
11	2	6.995
12	2	6.843
13	1	6.268
14	1	5.976
15	1	6.044
16	1	5.727
17	0	5.351
18	0	4.951
Rendimiento: 100%		

Experimento 4		
Circuito: "Circuito2"		
Puertas dañadas: [6,7]		
Conexiones dañadas: [(2,8), (4,8)]		
gen.	mín.	media
0	6	57.232
1	6	35.429
2	4	21.219
3	4	12.628
4	4	9.719
5	4	8.893
6	4	8.587
7	4	8.368
8	4	8.024
9	2	7.687
10	2	7.688
11	2	7.188
12	2	7.233
13	2	6.852
14	2	6.789
15	2	6.981
16	2	6.892
17	2	6.723
18	2	6.245
19	2	6.347
20	2	6.138
21	1	5.146
22	1	5.152
Rendimiento: 95.834%		

- En el “Experimento 1” ha devuelto como resultado el siguiente circuito alternativo:

```
[('XNOR', 3, [(2, 3), (0, 3)]), ('OR', 4, [(-1, 4), (0, 4)]), ('AND', 5, [(2, 5), (-1, 5)]), (
'NOT', 6, [(1, 6), (0, 6)]), ('AND', 7, [(0, 7), (1, 7)]), ('OR', 8, [(1, 8), (0, 8)]), ('NOT'
, 9, [(5, 9), (6, 9)]), ('XOR', 10, [(4, 10), (7, 10)]), ('OR', 11, [(-1, 11), (7, 11)]), ('XN
OR', 12, [(6, 12), (8, 12)]), ('NAND', 13, [(6, 13), (11, 13)]), ('XOR', 14, [(7, 14), (10, 14
)])]
```

- En el “Experimento 2” ha devuelto como resultado el siguiente circuito alternativo:

```
[('AND', 3, [(-1, 3), (-1, 3)]), ('NOR', 4, [(-1, 4), (-1, 4)]), ('NOT', 5, [(0, 5), (1, 5)]),
 ('NOT', 6, [(1, 6), (1, 6)]), ('NOR', 7, [(5, 7), (3, 7)]), ('XOR', 8, [(4, 8), (4, 8)]), ('N
OR', 9, [(7, 9), (4, 9)]), ('XNOR', 10, [(8, 10), (7, 10)]), ('XOR', 11, [(4, 11), (3, 11)]),
('NAND', 12, [(9, 12), (-1, 12)]), ('NOT', 13, [(9, 13), (10, 13)]), ('AND', 14, [(8, 14), (8,
 14)])]
```

- En el “Experimento 3” ha devuelto como resultado el siguiente circuito alternativo:

```
[('NAND', 3, [(0, 3), (2, 3)]), ('NOR', 4, [(1, 4), (0, 4)]), ('XOR', 5, [(2, 5), (1, 5)]), ('
NOT', 6, [(-1, 6), (3, 6)]), ('NOR', 7, [(3, 7), (5, 7)]), ('AND', 8, [(0, 8), (3, 8)]), ('NOR
', 9, [(4, 9), (4, 9)]), ('AND', 10, [(5, 10), (3, 10)]), ('AND', 11, [(8, 11), (5, 11)])]
```

- En el “Experimento 4” ha devuelto como resultado el siguiente circuito alternativo:

```
[('NAND', 3, [(2, 3), (2, 3)]), ('OR', 4, [(1, 4), (0, 4)]), ('AND', 5, [(1, 5), (0, 5)]), ('X
NOR', 6, [(4, 6), (0, 6)]), ('NOR', 7, [(0, 7), (0, 7)]), ('NOT', 8, [(3, 8), (-1, 8)]), ('OR'
, 9, [(7, 9), (4, 9)]), ('XOR', 10, [(8, 10), (4, 10)]), ('AND', 11, [(3, 11), (5, 11)])]
```

Se ha comprobado también que el sistema de autoreparación funciona correctamente con circuitos de tamaño **m.n = 4.4** y **m.n = 5.5**.

V. CONCLUSIONES

Es importante darse cuenta de que la autoreparación de un circuito reconfigurable no es un problema de maximización, ni de minimización, puesto que si hay dos individuos de la población cuyos fenotipos son circuitos alternativos diferentes, ambos “imitan” perfectamente el comportamiento del circuito original y ninguno de ellos contiene conexiones inválidas, dichos individuos son igual de “buenos” y cualquiera de ellos puede ser la solución al problema. Esto es así porque en realidad no existe ninguna función objetivo. El único “objetivo” de este problema es conseguir que la suma de las penalizaciones por incumplimiento de distintas restricciones sea 0. Esto significa que el objetivo principal del algoritmo genético es ir realizando el proceso evolutivo hasta encontrarse con el primer individuo cuyo valor de **fitness** es 0. Una vez que lo encuentra, el algoritmo puede parar el proceso evolutivo y devolver el fenotipo de dicho individuo como solución al problema. Observando los resultados de la sección anterior, se puede ver que si el individuo “perfecto” (es decir, el que tenga el **fitness** = 0) existe, el algoritmo genético es capaz de encontrarlo relativamente rápido. En las tablas correspondientes a los experimentos 1 y 3 se ve que a lo largo del proceso evolutivo el **fitness** del mejor individuo converge a 0. El individuo “perfecto” aparece en la séptima generación en Experimento 1 y en la decimoséptima generación en Experimento 3. En experimentos 2 y 4 dicho individuo no existe y, por tanto, el **fitness** del mejor individuo en cada generación va disminuyendo pero nunca alcanza el valor 0.

REFERENCIAS

[1] “Tema 2: Algoritmos genéticos” del curso IA de Ingeniería del Software. https://www.cs.us.es/cursos/iais-2017/temas/Geneticos.pdf

[2] El paquete Itertools https://python-para-impacientes.blogspot.com/2015/08/bucles-eficientes-con-itertools.html

[3] “Práctica 4 del curso IA de Ingeniería del Software (curso 2017/18)” https://www.cs.us.es/cursos/iais-2017/practicas/Pr%C3%A1ctica_04.zip

[4] Operador de mutación con un rango específico de valores permitidos
http://deap.readthedocs.io/en/1.0.x/api/tools.html#deap.tools.mutUniformInt