# *Inmersión en el hardware evolutivo con Python y AG*

Luis Morán Cordón
*dpto. Ciencias de la Computación e Inteligencia Artificial*
*Universidad de Sevilla*
Sevilla, España
luimorcor@alum.us.es

Ignacio Oubiña Caballero
*dpto. Ciencias de la Computación e Inteligencia Artificial*
*Universidad de Sevilla*
Sevilla, España
ignoubcab@alum.us.es

**Resumen— El objetivo principal es que el circuito que vamos a implementar sea capaz de funcionar correctamente mediante auto-reparación en caso de fallos de hardware de algunas de sus puertas físicas (que sufran daño físico) o conexiones dañadas. Es decir, funcionará el circuito en su conjunto a nivel global como si no se hubiera estropeado nada y para ello aplicaremos el uso de hardware evolutivo.**
**Las conclusiones que hemos obtenidos se resumen en que el hardware evolutivo funciona satisfactoriamente para reparar circuitos reconfigurables, aunque habrá ocasiones en el que no se podrá reparar completamente, siendo el cruzamiento utilizado por el algoritmo genético más efectivo el uniforme.**

***Palabras Clave—Inteligencia Artificial, Algoritmos Genéticos, Computación Evolutiva, Circuito Reconfigurable, Hardware Evolutivo, EWH, DEAP.***

## I. INTRODUCCIÓN

Este trabajo se desarrolla en el contexto de los sistemas evolutivos en la informática actual.

Los sistemas evolutivos son aquellos capaces de transformarse permanentemente. No cuentan con reglas que le digan cómo resolver un problema dado, sino que cuentan con la capacidad de construir su propia imagen de la realidad y con mecanismos para percibirla y actuar dentro de ella[1].

El problema que estamos tratando de resolver, al tener que transformarse para realizar una auto-reparación, entraría dentro del conjunto de sistemas evolutivos.

Una importante rama de la inteligencia artificial, que surge directamente a través de los sistemas evolutivos, es la Computación Evolutiva (CE) [2]. La CE pues; comprende métodos de búsqueda y aprendizaje automatizado inspirados en los principios de la evolución natural (adaptación y supervivencia del más adaptado). Es la metodología aplicable para resolver problemas de sistemas evolutivos como el que estamos tratando.

Dentro de la CE, se engloban diferentes estrategias para la resolución de problemas de optimización, denominados de manera colectiva como algoritmos evolutivos. Uno de los más importantes son los algoritmos genéticos (AG) [3],uno de los miembros más representativos de las técnicas heurísticas modernas.

El trabajo como hemos expuesto anteriormente consiste en un circuito de hardware reconfigurable.
En la actualidad es muy común que se produzcan fallos o errores en un circuito, y la probabilidad suele ser mayor mientras más complejo sea este. Para poder seguir funcionando a pesar de producirse fallos, los sistemas deben incorporar algún mecanismo para afrontarlos. Normalmente se consigue una tolerancia a fallos mediante redundancia hardware, es decir, si tenemos varios circuitos que realicen la misma función y uno falla el sistema seguirá funcionando correctamente.

La existencia de un hardware capaz de reconfigurarse evitaría a priori la necesidad de disponer de tanto hardware realizando funciones similares, ya que si falla una parte del circuito otra podría reconfigurarse para solucionar el error adaptándose a una nueva funcionalidad requerida. Sin embargo, surge un problema, ¿cómo es capaz el sistema de saber que parte del circuito ha fallado y que parte debe reconfigurarse para solventar el problema?

El hardware evolutivo [4] (en inglés EWH) une el hardware reconfigurable junto a la inteligencia artificial (concretamente a algoritmos evolutivos) para solucionar el problema expresado en el párrafo anterior. En el enunciado de este mismo trabajo podemos encontrar la frase “Las plataformas de hardware reconfigurable ofrecen un mundo de posibilidades al combinarlas con sistemas evolutivos como los algoritmos genéticos.” Pues bien, esto es lo que llamamos hardware evolutivo, la combinación de estos dos conceptos explicados anteriormente.

La estructura del contenido será la siguiente:

## II. PRELIMINARES

### A. *Métodos empleados*

Concretamente vamos a trabajar con algoritmos genéticos, que como mencionamos anteriormente es una de las estrategias que conforman la computación evolutiva.

Para solucionar el problema de la reparación automática de circuitos reconfigurables, debemos conseguir que el circuito pueda seguir comportándose igual a nivel global. Con este objetivo aplicaremos un algoritmo genético al circuito para ir seleccionando iterativamente los mejores candidatos hasta conseguir "buenos" circuitos alternativos.
Los circuitos alternativos considerados como "buenos" serán aquellos cuyo rendimiento sea muy alto en lo que respecta a la auto-reparación del circuito original, es decir, que dado diferentes valores de entradas y sin saber que puertas o conexiones puedan ser las que están dañadas, funcione el circuito lo más igual posible en su conjunto global. Debemos tener en cuenta que según [5], en el marco del hardware evolutivo no podemos hablar de un circuito óptimo sino de un circuito que se trabaja correctamente.

Vamos a utilizar la librería DEAP [6].de Python (Distributed Evolutionary Algorithms in Python). DEAP es una librería para algoritmos evolutivos (esto incluye los algoritmos genéticos) e incluye un conjunto de módulos y herramientas para poder construir nuestras propias evoluciones.

DEAP se utilizó para la práctica 4 de laboratorio (algoritmos genéticos) de la asignatura Inteligencia Artificial del grado en Ingeniería del Software por la Universidad de Sevilla en el curso académico 2017-2018 [7].

### B. *Trabajo Relacionado*

Primero vamos a comenzar mencionando la frase del primer capítulo del libro *Evolvable Hardware [8]*, "… this book describes a new hardware paradigm called Evolvable Hardware" para destacar (aunque el libro se publicó en 2006) que el hardware evolutivo es un concepto relativamente moderno. Por ello no hemos encontrado una grandísima cantidad de trabajos relacionados, y además debemos señalar también que la mayoría de estos se encontraban escritos en inglés.

Ha sido importante para el trabajo la tesis doctoral de la PhD Tatiana G. Kalganova [9], ya que es una obra muy completa sobre las EWH y de ella hemos obtenido ideas para la presentación de los resultados.
También cabe destacar el proyecto de fin de carrera realizado por Javier Mora de Sambricio (2011) [10], que implementa una EWH para generar de forma automática filtros digitales para el procesamiento de imágenes con ruido desconocido en tiempo de diseño y es interesante para ver el alcance y la aplicación del hardware evolutivo.

## III. DECISIONES DE DISEÑO

Al abordar el problema en cuestión, la reparación automática de circuitos reconfigurables, comenzamos con una primera idea muy básica a la hora de realizar el primer esbozo del problema. Teníamos un circuito de tamaño m x n siendo m el número de capas y n el número de puertas lógicas de cada capa. Sin embargo, para facilitar la representación lo dibujaremos en un primer momento como una malla dividida en celdas (de tamaño m x n) como podemos ver en la siguiente imagen original.

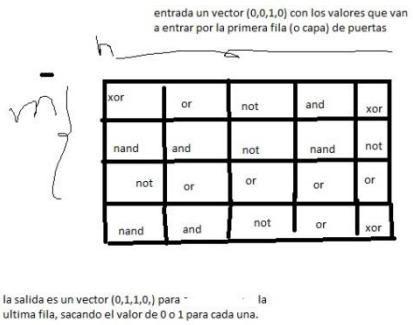



*Ilustración 1. Primer esbozo original*

La malla será un objeto de tipo Circuito, y cada celda de la malla un objeto de tipo puerta. Es decir, creamos dos clases específicas para resolver este problema, la clase Circuito y la clase Puerta.
Circuito a su vez será una clase que constará de un array lleno de puertas.

La clase puerta tiene 4 atributos. Estos son el tipo de puerta lógica que tiene programado en ese momento, la puerta asociada al primer input y la puerta asociada al segundo input y un último valor que es la salida de la puerta, que solo se utiliza cuando la puerta simula una entrada El constructor viene definido por 6 valores de entrada que son: [Tipo de puerta lógica, valor X entrada 1, valor Y entrada 1, valor X entrada 2, valor Y entrada 2 y salida].

-Tipo de la puerta lógica: puede tomar cualquiera de los siguientes valores numéricos (entre paréntesis está el tipo de puerta lógica a la que está asociado realmente cada número): 1(OR), 2 (AND), 3 (NOT), 4 (NAND), 5 (XOR), 0 o -1. En caso de ser -1 significaría que la puerta lógica sería una entrada y no una puerta per se. Si el valor es 0 la puerta está estropeada.

-Valor X entrada 1, valor Y entrada 1: sirven para identificar la puerta asociada al primer input. Si vemos el circuito de m x n casillas y cada casilla viene representada por un valor

(x,y). Sabemos que el primer input es aquel proveniente de la puerta situada en la celda x1, y1

-Valor X entrada 2, valor Y entrada 2: Puerta asociada al segundo input.

Anotación*: Es importante la observación de que en caso de que tome solo 1 input o ningún input (puerta desactivada), los valores x,y para ese input serían -1. Por ejemplo, la puerta [2, -1, -1, 3, 2] está programada para actuar como una AND y solo tendría un input que sería el proveniente de la segunda puerta situada en la tercera capa del circuito. El valor -1 simula el hecho de que a la puerta no le entre nada.

Clase circuito: como hemos expuesto anteriormente, la clase circuito tiene como atributos sus dimensiones; número de capas m y número de puertas por cada capa n. También tiene como atributo un array con los objetos de tipo puerta para cada casilla del circuito.

El constructor es (m,n,vector), siendo m y n los valores que tomarán los atributos explicados anteriormente y vector es un array de enteros de longitud n que inicializa las entradas.

La solución al problema está diseñada de manera que para todas las operaciones el circuito tiene m x n dimensiones, pero cuando se va a calcular el valor se le añade otra capa que pasa a ser la capa 0. Esta capa es la capa de entradas, que son n puertas sin **entradas y de tipo -1, cuya salida es el valor de la entrada a la que simulan.** Tal y como podemos apreciar en la siguiente imagen.

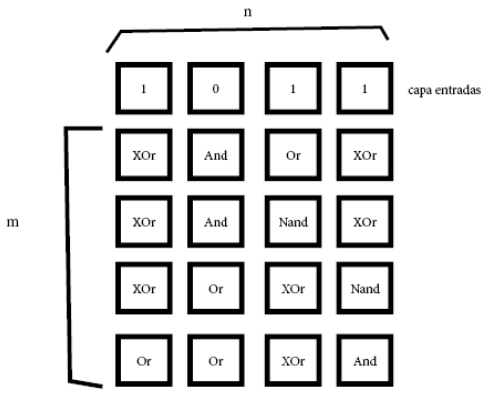



*Ilustración 2. Representación circuito con la capa de entradas*

Hay que tener esto en cuenta a la hora de poner los datos del circuito. Por ejemplo, si queremos que a una puerta le entren datos de la segunda entrada tendremos que poner que su entrada es x=0 e y=1, ya que la capa x=0 es la capa correspondiente a las entradas tal como explicamos anteriormente.

## IV. METODOLOGÍA

Una vez realizadas las clases sobre las que vamos a basar el problema decidimos que la configuración del circuito base no nos interesaba en nuestro problema. Solo no sirve los pares de entradas, salidas. En realidad, si el circuito tiene muchas puertas defectuosas el circuito base puede estar tan mal como cualquier otro.

Las funciones están anidadas en una función ejecutaPrograma que utiliza las difentes funciones para calcular el resultado del algoritmo.

Los datos de inicio son dos arrays de vectores. Uno de ellos indica cada entrada y otro cada salida siendo salida[i] la salida ideal de entrada[i]. Los vectores serán de longitud n. Para especificar las puertas se inicia un array con los valores x,y en los que se encuentran las puerta rotas dentro del array del circuito. Es decir, si el array incluye el par 2,1 significaría que la puerta 1 de la capa 2 estaría rota. Estos valores se usan antes de añadir la capa 0 de entradas por lo que la capa 0 sería la primera capa de puertas iniciales del circuito y no la correspondiente a la capa de entradas. Para las entradas el vector sería igual pero añadiendo un tercer elemento indicando cual de las dos entradas es la que es defectuosa.

Para asegurarse de si el circuito funciona correctamente usamos la función fitness y comprobamos si tiene ya valor máximo para el circuito que nos dan. Si el valor es máximo significaría que no necesitaríamos aplicar ningún cambio porque el circuito ya funciona bien. En caso contrario empezamos a configurar el algoritmo genético.

Los individuos son cromosomas de longitud m*n*5. Esto es

**Comprobación del funcionamiento correcto del circuito.**
**Entrada**:
- Un vector circuito

**Salidas**:
- Un int 1 o 0 dependiendo de si es correcto o no.

**Algoritmo**:
1. D=fitness(circuito)
2. if D==n*long(circuito)
   a.devolver 1
3.else
   a.delvolver 0

Pseudocódigo 1: Comprobación circuito

debido a que para representar un circuito cogemos conjuntos de 5 enteros que representan cada atributo de una puerta (hay que tener en cuenta que en este caso no se incluye el valor de salida de cada puerta). Siendo el primer entero el atributo tipo, los dos siguientes la x y la y de la primera entrada y los dos siguientes la x y la y de la segunda entrada. Como dijimos antes si la x y o la y de alguna vale -1 significa que por esa entrada no entra nada. Esto crearía un array bastante largo donde cada grupo de 5 enteros representaría una puerta del circuito y dependiendo de en que puesto se encuentre estaría en un lugar determinado del

array del circuito. Siendo i el primer entero de cada conjunto:

$$x=\frac{\frac{i}{5}}{n} \qquad y=\frac{i}{5} mod n$$

**Generación de individuos**
**Entrada**:
- int m
- int n

**Salidas**:
- array res de longitud n*m*5

**Algoritmo**:
1. for i de 0 a m*n*5
    a. ih=i/n
    b. res[i]=aleatorio (1 a 5)
    c. min=ih-1
    d. if min<0
        1.min=0
    e. d=aleatorio de min-1 a ih (si es min -1 la entrada está desconectada)
    f. if d==min-1
        1. d=-1
    g.res[i+1]=d
    h.res[i+2]=aleatorio de -1 a n-1
    i. d=aleatorio de min-1 a ih
    j. if d==min-1
        1. d=-1
    l.res[i+3]=d
    m.res[i+4]=aleatorio de -1 a n-1
devolver res

Pseudocódigo 2: Generación de individuos

P.ej. si el valor de i es 20 y el valor de las variables m,n es 4, x representaría la quinta puerta del circuito que estaría en la segunda fila (primera fila en el código) y sería la primera puerta de dicha fila (puerta número 0 en el código).

La evaluación del fenotipo sería la cantidad de bits de cada salida que coinciden con la salida que tenía el circuito antes de resultar dañado.

Para crear a cada individuo del problema creamos el método gen() que crea un cromosoma que cumple todas las restricciones del problema. Para ello se crea un array de m*n*5 de longitud y se le van añadiendo cada entero según se quiera poner el tipo o una entrada de una puerta. Los enteros se eligen de manera aleatoria entre los rangos posibles.

Ya que una puerta solo se puede conectar a puertas que pertenezcan a las dos capas anteriores, ponemos el rango de la x de las entradas de ih a min-1, teniendo en cuenta que min su mínimo valor es 0 (punto 1.d del pseudocódigo). También hay que tener en cuenta que hay algunas conexiones que están desconectadas, por lo que los intervalos son hasta -1 que implica que está desconectada.

Una vez hecho el generador de individuos hay que crear los parámetros para el algoritmo genético de DEAP. La función población() crea un conjunto de individuos sobre los que se realiza el algoritmo. Para este caso simplemente se ejecuta gen() 10 veces dando lugar a 10 circuitos aleatorios sobre los que empezar el algoritmo.

También hay que especificar los operadores usados para la mutación y el cruzamiento. Este apartado lo explicaremos mejor en Resultados.

Para hacer la mutación posible hay que incluir un apartado que elimine aquellos cromosomas que no sean correctos dentro de las restricciones del problema. Habría que comprobar:

- Que los tipos sean de 1 a 5
- Que las puertas no intenten conectarse a puertas que no estén en dos capas anteriores o que sean ellas mismas (lo cual crearía un bucle infinito al intentar obtener la salida del circuito).

Para el fitness de cada individuo tenemos una función fitness que calcula la salida del circuito dado solo el array individuo. Para eso tiene que hacer todas las comprobaciones que hemos comentado antes además de convertir el conjunto de enteros a un objeto de tipo circuito. Para ello se ayuda de varios métodos de la clase Puerta y la clase Circuito.

El método defArray(puertas) de la clase Circuito se ejecuta una vez tengamos un array con todas las puertas del circuito en las cuales las entradas son pares de enteros que apuntan a puertas del circuito o indican que la entrada está desconectada si cualquier valor del par es -1.

Primero añade al array de dimensiones (m+1)*n las puertas del circuito, teniendo en cuenta que la capa 0 son las entradas, que son enteros. A continuación recorre todo el array y va configurando primero las entradas para que se conviertan en puertas con salida el valor del entero y tipo -1, **para que directamente obtenga el atributo salida, ya que como veremos más tarde la función getSalida devuelve el atributo salida de la puerta si comprueba que el tipo de la puerta vale -1.** Después configura una a una cada puerta del circuito cambiando los atributos entradas, que son pares de enteros, por puertas del circuito. Para eso usa el método de la clase Puertas setCircuito(circuito,i,j).

**defArray**
**Entrada**:
- Un array puertas
- array del circuito array

**Salidas**:Ninguna.
**Algoritmo**:
1. array[capas de 1 a m+1]=puertas
2. for i en rango 0 a m+1
    a. for j en rango n
        1. if array[i][j] es una puerta
            a.setCircuito de esa puerta
        2. if array[i][j] es un entero
            a.array[i][j]=puerta con salida array[i][j] y entradas -1.

Pseudocódigo 3: Definición del circuito

setCircuito comprueba si las entradas son -1 o cualquier otro valor del array. En caso de que x o y sean -1 sustituye esa entrada por -1. Si no sustituye la entrada por la puerta que esté en x e y en el array del circuito.

**setCircuito**
**Entrada**:
- Un circuito
- La puerta en la que está

**Salidas**:Nada
**Algoritmo**:
1.if entrada0[0] y entrada0[1] !=-1
    a.entrada0=circuito.array[entrada0 x e y]
2. else
    a.entrada0=-1
3..if entrada1[0] y entrada1[1] !=-1
    a.entrada1=circuito.array[entrada1 x e y]
4. else
    a.entrada1=-1

Pseudocódigo 4: Cambio de int a tipo Puerta de entradas

Para poder realizar esta función primero tenemos que conseguir crear el array puertas a partir de los individuos. Esto se realiza en la misma función fitness después de haber comprobado que el individuo es válido. Para ello se recorre el array de individuo igual que en la comprobación y se crea una puerta con individuo [i] como tipo y los siguientes elementos i+1 i+2,i+3 e i+4 como los x e y de cada entrada.

**Crear array Puertas**
**Entrada**:
- Un array y vacío de dimensions m*n
- array x que es el individuo

**Salidas**:
- array y con las puertas.

**Algoritmo**:
1. for i de 0 a m*n*5
    a.puertax=Puerta(x[i],x[i+1],x[i+2],x[i+3],x[i+4],0)
    b. y[(i/5)/m][(i/5) mod m]=puertax

Pseudocódigo 5: Creación puertas a partir de individuo

La salida se pone a 0 porque al no ser una entrada no tiene relevancia. Después esa puerta se coloca en la posición x e y que especificamos en las ecuaciones del pseudocódigo 2 (generación de individuos).

También cuando realizamos esta parte del problema comprobamos que el array individuo es válido asegurándonos de todas las restricciones que mencionamos anteriormente. Se deben de comprobar otra vez porque cuando los cromosomas van mutando no podemos controlar que los individuos generados por el algoritmo genético cumplan las restricciones que sí cumplen los cromosomas generados por el método gen(). Si se detecta algún error la variable **error** pasa de 0 a 1. Más tarde explicaremos para que sirve esta variable.

Una vez tengamos el array con las puertas hay que cambiar aquellas que tengan conexiones o puertas dañadas. Para ello hay dos métodos en la clase Puerta para cambiar los atributos de la puerta y que esta refleje que está rota.

romperEntrada(i) rompe la entrada i(0 o 1) de la puerta en la que se ejecuta. Lo hace haciendo que esta conexión tenga entradas [-1,-1]. Ya que el valor de una conexión rota es el mismo que si no estuviera conectada.

setTipo(i) cambia el tipo de puerta a i. En el caso de una rota se cambiaría el tipo a 0. Una puerta que sea de tipo 0 siempre devuelve de salida 0.

El problema define dos arrays antes de la ejecución, uno para indicar cuales son las entradas defectuosas o rotas y otro que indica las puertas consideradas como dañadas. El array entradasRotas es un array de dos dimensiones en la que cada elemento es un array que indica las coordenadas de la entrada y la entrada (0 ó 1) que está rota. P. ej. [2,1,0] implicaría que la entrada 0 de la puerta que está en 2,1 está rota. El array puertas rotas es igual solo que no tiene el elemento que indica cual de las dos entradas es la que está rota. Así el elemento [1,1] indicaría que la puerta 1,1 estaría rota.
Teniendo en cuenta estos dos arrays el algoritmo que cambia los valores de las puertas dentro de la función fitness consiste en recorrer ambos arrays e ir usando la función romperEntrada y setTipo según sea un array u otro.

**cambiar valores puertas y entradas rotas**
**Entrada**:
- array puertasRotas
- array entradasRotas
- array Puertas

**Salidas**:
- array Puertas con puertas y entradas rotas modificadas

**Algoritmo**:
1. for k en rango 0-longitud de entradasRotas-1
    a.x=entradasRotas[k]
    b.si valores de x están en el rango del array
        1.array[x[0]][x[1]].romperEntrada[x[2]]
1. for l en rango 0-longitud de puertasRotas-1
    a.x=puertasRotas[k]
    b.si valores de x están en el rango del array
        1.array[x[0]][x[1]].setTipo(0).

Pseudocódigo 6: Cambiar valores puertas y entradas rotas

Una vez tenemos el array Puertas modificado tenemos que definirlo (defArray) y calcular su salida para compararla con la salida teórica del circuito. Pero primero comprobamos que el entero error no valga 1 sino 0. Si el entero error vale 1 significa que el circuito que genera ese individuo tiene un problema y por tanto su fitness vale 0 y no hace falta calcularlo. Si vale 0 tenemos que calcular su fitness y salida.

Primero definiremos un circuito creado anteriormente con el array Puertas que se obtiene tras modificar las puertas y entradas rotas. Después simplemente recorremos el arrayEntradas que se define al principio del problema y calculamos la salida del circuito para cada una de las entradas que se dan, para más tarde comparar la real con la salida teórica. El fitness es el número de valores que coinciden con la salida teórica, que están en vectorSalidas.

Para este último paso usamos el método setEntrada de la clase Circuito, que sirve para cambiar la entrada de un circuito. Este método simplemente recorre un array de longitud n y va cambiando los valores de salida del array del circuito en la capa para m=0 (la capa de las entradas). Como explicamos al principio la salida en las puertas de la capa de entradas son los valores de entrada del circuito.

**setEntrada**
**Entrada**:
- array entrada
- array **array** del circuito.

**Salidas**:
**Algoritmo**:
1. for k en rango 0-n
    a. array[0][k].setSalida(entrada[k])

Pseudocódigo 7: Cambiar entrada circuito

También se usa el método getSalida de la clase circuito que se encarga de calcular la salida del circuito.

**getSalida**
**Entrada**:
- array **array** del circuito.

**Salidas**:
- array **vector** de longitud n con salida del circuito

**Algoritmo**:
1. for k en rango 0-n
    a. añadir a **vector** array[m][k].getSalida()
2. devolver **vector**

Pseudocódigo 8: Obtener salida del circuito

Dentro del método getSalida de Circuito se usa el método **getSalida** de la clase Puerta. Este método se encarga de calcular la salida de la puerta que lo invoca. Es un método recursivo ya que cuando invocamos el método sobre una puerta este hace que las puertas que son entradas de dicha puerta lo invoquen a su vez. Creando así un bucle que termina cuando una puerta no tiene entradas, sus entradas están rotas o sus entradas son entradas del circuito. El método **getSalida** es el método más importante de la clase Puerta ya que es la base de la simulación del circuito.

El método crea dos parámetros **x** e **y** que representan el valor de las dos entradas con la que se hace distintas operaciones dependiendo del tipo de puerta. Primero se asegura de que las entradas no están rotas. En caso de que lo estén x e y, o las dos se convierten en 0. Si no están rotas alguna o las dos de ellas el valor de x e y sería la salida de la puerta que es entrada de esta puerta (aquí es donde entra en juego la recursividad) . Después dependiendo del tipo de puerta realiza una operación sobre x e y. Si la puerta vale 0 significa que la puerta está defectuosa por lo que no usa ni x ni y sino que simplemente devuelve 0. Si vale -1 significa que es una entrada del circuito por lo que devuelve directamente el valor del atributo salida de la puerta. Si vale de 1 a 5 el método realiza las operaciones AND, OR, NOR, etc. sobre x e y y devuelve el resultado.

**getSalida de Puerta**
**Entrada**:
- Puerta **puerta** del método

**Salidas**:
- entero **res** salida de la puerta

**Algoritmo**:
1.x=0
2.y=0
3. if puerta.entrada0 no es -1
    a.x=puerta.entrada0.getSalida
4. if puerta.entrada0 no es -1
    a.x=puerta.entrada0.getSalida
5.if puerta.tipo es 0
    a.devolver 0
6.if puerta.tipo es 1
    a.devolver x|y
7.if puerta.tipo es 2
    a.devolver x&y
8.if puerta.tipo es 3
    a.devolver ¬x
9.if puerta.tipo es 4
    a.devolver ¬(x & y)
10.if puerta.tipo es 5
    a.devolver (x&¬y)|(¬x&y)
11.if puerta.tipo es -1
    a.devolver puerta.salida

Pseudocódigo 9: Obtener salida puerta

Añadir que para este problema hemos tenido que crear el método bit_not(n) para las puertas que tenían que negar una entrada ya que en Python el operador not hace la negación dando por hecho que la entrada tiene 16 bits y no 1 como en nuestro caso.

Visto todo esto podemos hacer un resumen del método Fitness:

1. Creamos circuito de dimensiones m x n.
2. Generamos array Puertas a partir de individuo detectando errores.
3. Cambiamos puertas y entradas rotas
4. Si hay error devolvemos 0, sino seguimos.
5. Comparamos salida para cada vector entrada con las salidas teóricas.
6. Devolvemos número de coincidencias.

**cálculo Fitness**
**Entrada**:
- array **array** del circuito.
- Circuito **cir**
- array **vectorEntradas**
- array **vectorSalidas**

**Salidas**:
- entero **res** con valor de fitness

**Algoritmo**:
1. for k en rango 0-longitud(vectorEntradas)-1
    a. setEntrada de cir con vectorEntradas[k]
    b.a=getSalida de cir
    c. res=0
    d.for i en rango 0-n-1
        1.if a[i] es igual que vectorSalidas[k][i]
            a.**res**++
2.devolver **res**

Pseudocódigo 10: calcular fitness

Está función de fitness también debe añadirse a los parámetros de un algoritmo genético perteneciente a la librería DEAP. El parámetro correspondiente al fitness es "evaluate". Añadiendo los demás parámetros (mate, mutate y select) podemos realizar los experimentos.

Mate indica el operador de cruce del algoritmo genético. En este caso hemos usado el cruzamiento de un punto con el operador de DEAP cxOnePoint, cruzamiento de dos puntos con cxTwoPoint y cruzamiento uniforme con cxUniform.

Mutate indica el operador de mutación del algoritmo. Para la mutación hemos usado mutación uniforme con la mutUniformInt. Para ajustar los mínimos y los máximos de la mutación uniforme hemos creado la función generateLowAndUp. También hemos usado un mínimo fijo de 1 máximo 5. Esta función crea dos arrays de longitud mxnx5, en uno inserta los valores mínimos de cada puerta siguiendo un array modelo de individuo, y en otro los máximos.

El select indica el método de selección de los individuos. En nuestro caso hemos usado selTournament del módulo tools. Este selecciona al azar una cierta cantidad de individuos y selecciona el más apto.

Para la implementación del algoritmo genético usamos la función eaSimple del módulo algorithms. Esta función coge una población inicial, la muta y la cruza y la evalúa dando lugar al mejor individuo. Realiza esto varias veces (según el número de generaciones que indiquemos) y compara el resultado de cada generación. También hemos usado el eaMuPlusLambda donde μ es la dimensión la población padre y λ la de la hija.

Para la representación gráfica del circuito tenemos la función representa_Circuito que imprime en la consola de la pantalla el circuito para verlo de forma rápida, sencilla e intuitiva.

Para realizar el archivo ejecutable y el instalador hemos usado la librería PyInstaller y el programa NSIS respectivamente.

**Representación del circuito**
**Entrada**:
- Vector individuo
- Vector entradas_iniciales

**Salidas**:
- Salida por pantalla

**Algoritmo**:
1.Imprimir n
2. Para p en rango (n)
    a. Imprimir |  p  |
3. Imprimir m
4. Para k en rango (n)
    a. Imprimir | entradas_iniciales(k) |
4. Para i en rango (0,n*m*5) con i aumentando de 5 en 5
    a. Si i%(5*n) ==0
        i. Imprimir salto de línea
        ii.Imprimir pantalla ----------
        iii. Imprimir salto de línea
        iiii.Imprimir i/(5*n) +1
    b. Si individuo(i)= 1
        i. Imprimir OR
    c. Si individuo(i)= 2
        i. Imprimir AND
    d. Si individuo(i)= 3
        i. Imprimir NOT
    e. Si individuo(i)= 4
        i. Imprimir NAND
    f. Si individuo(i)= 5
        i. Imprimir XOR
    g. Imprimir (individuo(i+1), individuo(i+2)) , (individuo(i+3), individuo(i+4)) ||
5. Imprimir salto de línea
6. Para b en rango (n)
    a. Imprimir pantalla ----------

Pseudocódigo 11: Representar circuito por pantalla

## V. RESULTADOS

### *A. Tabla experimentación*

| | Circuitos | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **3x3** | | **3x3 (puerta salida dañada)** | | **4x4** | | | | | **5x6** | | | | | | | | | | | | | | | | |
| **Tipo de EA** | eaSimple | | | | | | | | | | | | | | | | | | | | | | eaMuPlusLambda(λ=15, μ=15) | | eaMuPlusLambda(λ=15, μ=5) | |
| **Tipo de mutación** | Ninguna | | | | Uniforme (sin ajustar) | | | Uniforme(indpb=0.2) | | Ninguna | | | | Uniiforme (sin ajustar ,indpb=0.2) | | | | Uniiforme (indpb=0.2) | | | | ,indpb=0.2 | | | |
| **% de mutación** | 0% | | | | 10% | | 20% | | | 0% | | | | 20% | 10% | 30% | 10% | 30% | | | | | | | | |
| **Tipo de cruzamiento** | Ninguno | | Uniforme | | | | Un punto | | | Dos puntos | Uniforme (indpb=0.2) | | | | | | | | | | | | | | | |
| **% de cruzamiento** | 0% | | 10% | | 30% | 10% | 30% | | | 20% | | | | | | | | | | | | | | | | |
| **Número de generaciones** | 20 | | | | | 20 | 2000 | 20 | 2000 | 20 | | | | | | | 200 | 20 | 200 | 2000 | | | 20 | | | |
| **Rendimiento máximo** | 100% | | 66,67% | 81,25% | 87,5% | 81,25% | 87,5% | 87,5% | 87,5% | 100% | 77,78% | | 83,33% | 77,78% | 88,89% | | | 94,44% | 97,22% | 100% | 97,22% | 88,89% | 83,3% | | | |
| **Media rendimiento 5 mejores** | 86,67% | | 66,67% | 81,25% | 83,75% | 81,25% | 87,5% | 83,75% | 85% | 100% | 77,78% | 74,44% | 76,67% | 72,22% | 88,89% | 75,56% | 88,89% | 90,00% | 95% | 100% | 97,22% | 81,11% | 73,3% | | |
| **Tiempo ejecución (seg)** | 0.055991 | 0.06599 | 0.064981 | 0.09596 | 0.117981 | 0.0729680 | 9.116575 | 0.134979 | 0.095986 | 11.14116 | 0.122981 | 0.123966 | 0.12197 | 0.174472 | 0.129981 | 0.157999 | 1.602202 | 0.234936 | 2.111678 | 29.176057 | 27.218878 | 0.186966 | 0.070990 | | | |

*B. Análisis*

Los experimentos consistieron en probar diferentes operadores de mutación y cruzamiento para ver con que configuración se realizaba la mejor reconfiguración. Para ello usamos varios circuitos y la misma configuración de puertas y entradas rotas para dichos circuitos. Probamos distintos circuitos ya que la dificultad de crear un circuito que se comporte igual al objetivo no es la misma si el circuito es más grande. En primer lugar, usamos un circuito 3x3 con puertas rotas en (1,1) y (0,2). Y las entradas (1,2,0) y (2,2,0) también dañadas.

El circuito se puede ver en la siguiente imagen capturada de la consola de eclipse.

| n->m | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | OR (0,0) , (0,1) | NOT (0,0) , (X,2) | NOT (X,X) , (0,1) |
| 2 | AND (1,1) , (X,0) | NAND (1,1) , (0,0) | OR (X,2) , (1,2) |
| 3 | XOR (1,2) , (2,0) | NAND (1,2) , (X,2) | OR (1,0) , (2,1) |

Cada puerta está representada por su tipo y las coordenadas de cada entrada entre paréntesis separadas por una coma. Si una de las coordenadas es X, significa que esa entrada no está conectada.

Si el circuito no tuviera puertas ni entradas dañadas, la salida a las entradas 010, 101 y 100 sería siempre 111. Con las puertas dañadas la salida sería 010.

En el caso de puerta de salida dañada simulamos que una entrada de la capa de salida está dañada para que de otros resultados.

Para el circuito 4x4 utilizamos el que viene representado en la imagen a continuación.

| n->m | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 1 | OR (0,2) , (X,1) | OR (0,1) , (0,0) | XOR (0,1) , (0,2) | NOT (0,3) , (0,2) |
| 2 | OR (0,2) , (0,X) | XOR (1,0) , (1,0) | XOR (X,2) , (1,2) | OR (0,1) , (1,1) |
| 3 | NAND (1,2) , (2,3) | AND (2,0) , (2,0) | NOT (2,0) , (2,1) | NAND (1,3) , (1,3) |
| 4 | OR (2,0) , (2,2) | AND (3,3) , (3,2) | NAND (3,0) , (3,X) | AND (2,1) , (3,1) |

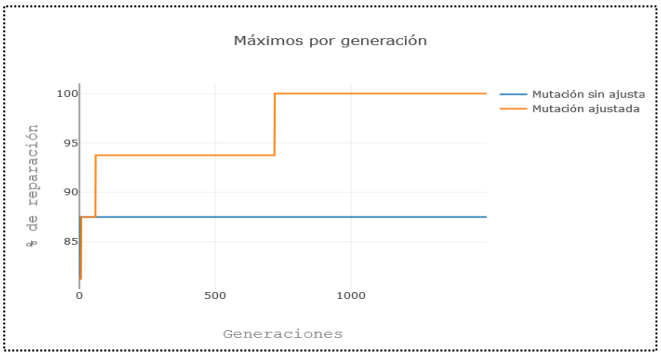
La tabla obtenida tras la experimentación muestra los porcentajes máximos de efectividad del algoritmo, la media de las 5 mejores soluciones y el tiempo de ejecución para cada caso mostrado.

Los porcentajes se obtienen dividendo el fenotipo del individuo por el fenotipo máximo, que sería n multiplicado por la cantidad de salidas que conocemos del circuito (dado en vectorSalidas).
La mutación no ajustada implica que el límite mínimo y el máximo de la mutación uniforme no cambia según el elemento, sino que es siempre 1 para el mínimo y 5 para el máximo.

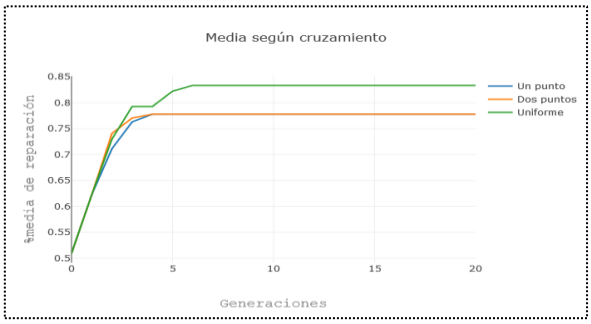
Como podemos observar para un circuito pequeño con unas pocas entradas y puertas rotas el algoritmo es capaz de resolver el problema sin ni siquiera usar mutación. Pero qué pasaría si hiciéramos que la tercera puerta de salida estuviera rota y por lo tanto siempre devolviera 0. Entonces el rendimiento caería al 66,67% como vemos ahora. Esto ocurre porque por mucho que cambiemos el circuito la tercera puerta de la última capa siempre devuelve 0 porque está dañada por lo que no puede corregir completamente el circuito.

Como podemos observar para cualquier otro caso en un circuito 3X3 como el que hemos usado el algoritmo si es capaz de reparar completamente el circuito. Incluso usando solo la población inicial el programa puede encontrar tres combinaciones que cumplen las restricciones del circuito

Para un circuito de 4x4 los resultados pueden llegar hasta el 81,25% de acierto sin mutación. Añadiendo mutación con límites 1 y 5(lo cual da lugar a algunos individuos no válidos) es capaz de llegar hasta el 87,5%. Pero aún así si subimos el número de generaciones nos damos cuenta de que el rendimiento no sube de 87,5% a no ser que usemos mutación con los límites ajustados. No obstante, con un rango de 20 generaciones la mutación ajustada no mejora los resultados de la mutación sin ajustar, incluso la no ajustada consigue llegar antes al 87,5%. Es cuando subimos el número de generaciones cuando conseguimos llegar al 100% de reparación. En la siguiente gráfica podemos ver el máximo obtenido por generación de la mutación ajustada y la no ajustada (para realizar las gráficas hemos usado la librería Plotly usando los resultados de las experimentaciones).

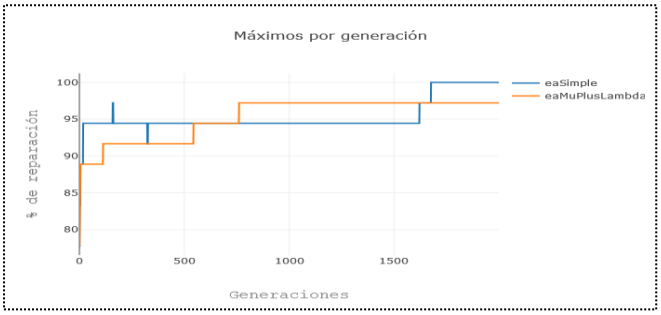



Con el circuito de tamaño 5x6 observamos que los máximos rendimientos de los experimentos con solo cruzamiento son más bajos, como es normal al ser el circuito más grande. De todos los tipos de cruzamiento el que mejor funciona es el uniforme que consigue llegar al 83,3%.




En esta gráfica analizamos la media a lo largo de 20 generaciones de reparación según cada cruzamiento.

Aunque la mutación uniforme sin ajustar empeora los resultados en uno de los casos, en los demás consigue subir el rendimiento hasta un 88,9%.




También hemos comparado los resultados con algoritmo μ+λ y los valores son muy similares para un μ y λ de 15. La desigualdad se obtiene cuando se sube el número de generaciones a 2000 y usamos mutación uniforme ajustando los límites. Para este caso podemos ver que el algoritmo μ+λ no consigue llegar al 100% a diferencia del eaSimple que llega en la generación 1700 como podemos ver en la siguiente gráfica.

## V. *CONCLUSIONES*

El problema se puede resolver incluso para circuitos grandes como el último que hemos comprobado siempre que se utilicen un número de generaciones alto. Aún así hay casos en los que el problema es imposible de resolver completamente, es decir, el algoritmo es incapaz de reparar completamente el circuito. Este el caso de circuitos que tiene una de las puertas de la última capa dañadas. Si por esa puerta debe salir 1 para alguna entrada el circuito es irreparable ya que las puertas dañadas siempre devuelven 0.

Hemos comprobado que entre todos los tipos de cruzamiento usados el uniforme es el más efectivo.

La mutación aun poniendo unos valores mínimos y máximos que dan lugar a que algunos individuos no sean válidos puede mejorar los resultados del algoritmo, incluso dando mejores resultados que ajustando los mínimos y máximos en las primeras generaciones de uno de los casos. Aún así es cuando ajustamos los valores mínimos y máximos de la mutación cuando conseguimos los resultados más altos de acierto llegando a reparar completamente el circuito si aumentamos el número de generaciones.

Después de probar dos algoritmos de la librería DEAP, el eaSimple y el eaMuPlusLambda hemos comprobado que en pocas generaciones su rendimiento es similar, incluso mejor en muPlusLambda pero que el eaSimple en un escenario de más generaciones es más efectivo llegando a 100% de efectividad para número de generaciones altos.

Por último, en gran parte de la documentación sobre EWH se hace referencia a un problema de escalabilidad (para circuitos mayores y más complejos) que hay que tener muy en cuenta, aunque nosotros no hayamos podido detectar dichos problemas en nuestro trabajo ya que los circuitos testeados no tenían una gran complejidad ni dimensión. [11]

## VI. BIBLIOGRAFÍA

### *A.* REFERENCIAS


[1] Página web introduciendo los sistemas evolutivos. http://fgalindosoria.com/eac/evolucion/evolucion_sistemas_evolutivos/evolucion_sistemas_evolutivos.htm

[2] Página web sobre la computación evolutiva. https://jarroba.com/computacion-evolutiva-ejemplo-con-un-algoritmo-genetico/

[3] Página web acerca de algoritmos genéticos. http://www.aic.uniovi.es/ssii/Tutorial/AGs.htm

[4] Alberto Urbón Aguado, "Estudio e implementación de reconocedores de secuencias mediante hardware evolutivo", p.15 https://eprints.ucm.es/9872/1/Hardware_Evolutivo_Alberto_Urbon.pdf

[5] Página web sobre EHW. http://19librasoccer.wixsite.com/inteligenciauatartif

[6] Documentación de DEAP. http://deap.readthedocs.io/en/1.0.x/index.html

[7] Página web del curso IA de Ingeniería del Software con las prácticas de la asignatura. https://www.cs.us.es/cursos/iais-2017/?contenido=practicas.php

[8] Xin Yao and Tetsuya Higuchi, "Introduction to Evolvable Hardware".

[9] Tatiana G. Kalganova "Evolvable hardware design of combinational logic circuits". https://www.napier.ac.uk/~/media/worktribe/output-278021/kalganovapdf.pdf

[10] Javier Mora de Sambricio, "Implementación de hardware evolutivo en un array sistólico mediante reconfiguración parcial". http://oa.upm.es/48786/1/PFC_JAVIER_MORA_DE_SAMBRICIO.pdf

[11] James Hereford, David Gwaltney, "Scalability, timing, and system design issues for intrinsic evolvable hardware". https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20040129638.pdf


### *B.* OTROS


- https://plot.ly/python/ Librería Plotly para gráficas
- https://www.pyinstaller.org/ Página de librería PyInstaller
- Xin Yao and Tetsuya Higuchi, "Promises and Challenges of Evolvable Hardware". http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.3665&rep=rep1&type=pdf
- http://nsis.sourceforge.net/Main_Page , página del programa NSIS
- Artículo "Making a Stand Alone Executable from a Python Script using PyInstaller" https://medium.com/dreamcatcher-its-blog/making-an-stand-alone-executable-from-a-python-script-using-pyinstaller-d1df9170e263