Reconfigurando circuitos lógicos

Adrián Cantón Fernández
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
adrcanfer@alum.us.es

Raúl Rodríguez Méndez
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
raurodmen@alum.us.es

***Resumen*— Este trabajo tiene como objetivo el de encontrar circuitos lógicos equivalentes a uno dado mediante la aplicación de Algoritmos Genéticos.**

La principal aplicación para la que se podría usar nuestra implementación sería la de detectar un fallo en un circuito lógico, generar un circuito equivalente para reconfigurarlo y detectar si se ha solucionado el fallo.

Palabras Clave—Inteligencia Artificial, Algoritmos Genéticos, Circuitos Reconfigurables, Puertas Lógicas.

I. INTRODUCCIÓN

En la década de los 70, surgieron de la mano de John Henry los algoritmos genéticos [1], una de las líneas más prometedoras de la inteligencia artificial. Reciben ese nombre porque se inspiran en la evolución biológica y en su base genético-molecular. En ellos se mantiene un conjunto de entidades, que reciben el nombre de cromosomas, las cuales se cruzan entre ellas o mutan y compiten entre sí de modo que las más aptas prevalecen, evolucionando hacia mejores soluciones cada vez.

Internet ha permitido que se extienda el uso remoto de plataformas reconfigurables, permitiendo reconfigurarlas sin que un técnico esté presente. Es necesario un método que detecte sin presencia física de un técnico, si un circuito funciona de manera correcta. Estas plataformas se usan en lugares de difícil acceso a personas, puesto que, si algún componente del circuito no funciona de manera correcta, sería posible mandarle una nueva configuración de forma remota.

En este trabajo hemos desarrollado un programa en Python que recibe el circuito implantado en la plataforma reconfigurable, y mediante una implementación de un Algoritmo Genético puede encontrar una configuración equivalente. Debido a la aleatoriedad de estos, no podemos garantizar que el método funcione siempre y en un tiempo razonable.

II. PRELIMINARES

A. Métodos empleados

Para el desarrollo de este proyecto hemos implementado un Algoritmo Genético. Este algoritmo precisa de una semilla para iniciar, es por ello por lo que hemos decidido hacer una implementación iterativa para esta elección. Esto podremos verlo más adelante en detalle.

III. METODOLOGÍA

A continuación, vamos a detallar el programa desarrollado para la resolución del problema presentado.

En primer lugar, hemos definido una estructura que permita representar el circuito y que pueda ser procesado por el ordenador. Recibe como parámetros el número de capas, el número de puertas, una lista con la programación de las puertas en orden y una lista con las conexiones. La programación de las puertas puede ser:

- ‘AND’
- ‘OR’
- ‘NOT’
- ‘NAND’
- ‘XOR’
- ‘—‘(No programada)

Estructura del circuito
Entrada:
- Número de capas del circuito (ncapas)
- Número de puertas de cada capa (npuertas)
- Una lista con la programación de las puertas (programacionPuertas)
- Una lista con las conexiones de cada puerta (conexiones)

Salidas:
- Circuito

Algoritmo:
1. Estructura = lista vacía
2. i = 0
3. Mientas i<ncapas
   a. Puertas = lista vacía
   b. j = 0
   c. Mientras j< npuertas
      i. Añadir a Puertas [programacionPuertas(i*npuertas)+j, conexiones(i*npuertas)+j]
   d. Añadir Puertas a Estructura

Pseudocódigo 1. Estructura del circuito

Ejemplo de definición de un circuito

```
programacionPuertas = ['OR', 'OR', 'NOT', '--']
conexiones = [[0,1],
              [0,1],
              [[0,0],[0,1]],
              [[0,0],[0,1]]]
ncapas = 2
npuertas = 2
```

Ejemplo 2. Definición circuito

En el ejemplo anterior podemos observar como espera recibir los datos la función que genera el circuito.

En primer lugar, la programación de las puertas viene dada por una lista de cadenas en orden. La primera programación sería la correspondiente a la primera puerta de la primera capa y así sucesivamente.

En segundo lugar, recibe las conexiones en otra lista. La primera puerta (OR) recibiría el primer y segundo bit del vector de entrada. La tercera puerta (NOT), situada en la segunda capa, recibiría los datos de salida de la primera y segunda puerta de la primera capa. Cabe destacar que las puertas con programación 'NOT' sólo leen la primera entrada.

Finalmente es necesario indicar el número de capas y de puertas para generar su estructura.

Este método no comprueba que la asignación de las conexiones sea válida. Este hecho se comprueba en la función fitness del algoritmo genético. Lo podremos ver en detalle más adelante. Nuestra interpretación del problema no permite que la capa 2 se comunique con el vector de entrada, puesto que, aunque una capa se puede comunicar con sus dos capas anteriores, el vector de entrada no es una capa y por tanto no hemos decidido que su comunicación sea posible.

A continuación, se ha implementado una función que actúa como simulador de circuitos. En primer lugar, tenemos una función que va calculando la salida de cada capa. Recibe como parámetros los vectores de salida de las 2 capas anteriores, el circuito a evaluar, la capa en la que estamos, y el número de puertas de cada capa. Y devuelve el vector de salida de dicha capa. El método itera por cada puerta de la capa, analiza las conexiones de dicha puerta, lee el bit del vector de entrada correspondiente y calcula la salida en función del tipo de puerta.

Resultado Circuito Capa
Entrada:

- Vector de salida de la capa i-2 (V1)
- Vector de salida de la capa i-1 (V2)
- Circuito a evaluar (Circuito)
- Capa a evaluar (i)
- Número de puertas (n) en la capa

Salidas:

- Vector de salida de la capa i (VS)

Algoritmo:

2. Vector de salida = Lista vacía
3. Si i == 0
   a. j = 0
   b. Mientras j < n
      i. Puerta = Circuito[i][j][0]
      ii. Conexiones = Circuito[i][j][1]
      i. Si Puerta == 'AND'
         1. Si V2[Conexiones[0]] + V2[Conexiones[1]] == 2
            a. Añadir 1 a VS
         2. Si no
            a. Añadir 0 a VS
      ii. Si Puerta == 'OR'
         1. Si V2[Conexiones[0]] + V2[Conexiones[1]] >=1
            a. Añadir 1 a VS
         2. Si no
            a. Añadir 0 a VS
      iii. Si Puerta == 'NOT'
         1. Si V2[Conexiones[0]] == 0
            a. Añadir 1 a VS
         2. Si no
            a. Añadir 0 a VS
      iv. Si Puerta == 'NAND'
         1. Si V2[Conexiones[0]] + V2[Conexiones[1]] == 2
            a. Añadir 0 a VS
         2. Si no
            a. Añadir 1 a VS
      v. Si Puerta == 'XOR'
         1. Si V2[Conexiones[0]] + V2[Conexiones[1]] == 1
            a. Añadir 1 a VS
         2. Si no
            a. Añadir 0 a VS
      vi. Si Puerta == '--'
         1. Añadir a 0 a VS
      vii. Incrementar j una unidad
3. Si i > 0
   a. j = 0
   b. Mientras j < n
      i. Puerta = Circuito[i][j][0]
      ii. Conexiones = Circuito[i][j][1]
      iii. Valor1 = 0
      iv. Valor2 = 0
      v. Si Conexiones[0][0] == i-2
         1. Valor1 = V1[Conexiones[0][1]]
      vi. Si no
         1. Valor1 = V2[Conexiones[0][1]]
      vii. Si Conexiones[1][0] == i-2
         1. Valor2 = V1[Conexiones[1][1]]
      viii. Si no
         1. Valor2 = V2[Conexiones[1][1]]
      ix. Si Puerta == 'AND'
         1. Si Valor1 + Valor2 == 2
            a. Añadir 1 a VS
         2. Si no
            a. Añadir 0 a VS
      x. Si Puerta == 'OR'
         1. Si Valor1 + Valor2 >=1
            a. Añadir 1 a VS
         2. Si no
            a. Añadir 0 a VS
      xi. Si Puerta == 'NOT'
         1. Si Valor1 + Valor2 == 0
            a. Añadir 1 a VS
         2. Si no
            a. Añadir 0 a VS
      xii. Si Puerta == 'NAND'
         1. Si Valor1 + Valor2 == 2
            a. Añadir 0 a VS
         2. Si no
            a. Añadir 1 a VS

iv. Si Puerta == 'XOR'
    1. Si Valor1 + Valor2 == 1
        a. Añadir 1 a VS
    2. Si no
        a. Añadir 0 a VS
v. Si Puerta == '--'
    1. Añadir a 0 a VS
        a. Añadir 1 a VS
vi. Si Puerta == 'XOR'
    1. Si V2[Conexiones[0]] + V2[Conexiones[1]] == 1
        a. Añadir 1 a VS
    2. Si no
        a. Añadir 0 a VS
vii. Si Puerta == '--'
    1. Añadir a 0 a VS
viii. Incrementar j una unidad
4. Devolver el vector de salida calculado

Pseudocódigo 2. Resultado de una capa del circuito

Puesto que las conexiones de la primera capa se introducen de forma distinta, debemos hacer una distinción en el método en función de si estamos en la primera capa o no, pero en líneas generales, es una función con una idea muy simple.

Después hemos implementado un método que calcula el resultado de un circuito completo usando el método anterior. Recibe los dos vectores de salida de las capas anteriores (un poco más adelante se explicará cómo deben ser estos vectores), el circuito, el número de capas y el número de puertas.

Resultado Circuito
Entrada:
- Vector1
- Vector2 (Vector de entrada del circuito)
- Circuito
- Número de capas (m)
- Número de puertas en cada capa (n)

Salidas:
- Vector de salida (VS)

Algoritmo:
1. VS = Lista vacía
2. i = 0
3. Mientras i<m
    a. VS = resultado_circuito_capas(Vector1, Vector2, circuito, i, n)
    b. Vector1 = Vector2
    c. Vector2 = VS
4. Devolver Vector2

Pseudocódigo 3. Resultado del circuito completo

Vamos a ver un ejemplo de cómo usar este ejemplo teniendo en cuenta el circuito anterior:

Ejemplo de cálculo del resultado de un circuito
Vector1 = [0,0]
Vector2 = [1,0]
Vector_Salida = resultado_circuito (Vector1, Vector2, circuito, ncapas, npuertas)

Ejemplo 2. Cálculo del resultado de un circuito

El valor del vector1 es indiferente, puesto que para la primera capa no se lee y luego se sobrescribe, pero es necesario para las siguientes iteraciones. El valor del vector 2, coincide con el valor del vector de entrada.

También se ha definido un método que calcula un circuito contemplando las puertas defectuosas. Este método es necesario para poder calcular la salida del circuito con defectos. Recibe como parámetros el número de capas del circuito, el número de puertas en cada capa del circuito, una lista con la programación de las puertas, otra con las conexiones de cada puerta (ambas en el formato establecido anteriormente) y una lista con las posiciones de las puertas que no funcionan correctamente. Calcula el circuito y cambia la programación de las puertas defectuosas por '—'.

Circuito defectuoso
Entrada:
- Número de capas del circuito (ncapas)
- Número de puertas de cada capa (npuertas)
- Una lista con la programación de las puertas (programacionPuertas)
- Una lista con las conexiones de cada puerta (conexiones)
- Una lista con las posiciones de las puertas defectuosas

Salidas:
- CircuitoDefectuoso

Algoritmo:
1. Calculamos el circuito llamando internamente a la función Estructura Circuito
2. i = 0
3. Mientras i<len(puertasDefectuosas)
    a. Cambiamos la puerta del circuito situada en la posición i de la lista por '—'

Pseudocódigo 4. Circuito defectuoso

Tras esto, vamos a definir una función que nos permita diagnosticar si un circuito está funcionando de manera correcta. Esta función va a recibir una lista con los vectores de entradas que se van a utilizar para evaluar el circuito, un vector de entrada v1 que es similar al vector1 visto en el método Resultado Circuito, es indiferente el valor, pero necesario para el algoritmo, el circuito base, es decir el que funciona como queremos, el circuito a analizar, el número de capas y el número de puertas.

Autodiagnóstico
Entrada:
- Vectores de entradas
- Vector1
- Circuito base
- Circuito a comparar
- Número de capas (ncapas)
- Número de puertas de cada capa (npuertas)

Salidas:
- Rendimiento

Algoritmo:
1. Calculamos los vectores de entrada (se realiza un ajuste en función del número de entradas del circuito y los vectores de entrada pasados como parámetros)
2. Calculamos los vectores de salida del circuito base
3. Calculamos los vectores de salida del circuito a comprobar
    b. Iguales = 0

4. i = 0
5. Mientras i < número de vectores de entrada
   a. Si Vector_Salida[i] == Vector_Salida_Circuito_Comprobar[i]
      i. Incrementar iguales
6. Devolvemos iguales/número de vectores de entrada

Pseudocódigo 5. Autodiagnóstico

Cuando calculamos los vectores de entrada, realmente nos queremos referir a que se ajustan los vectores de tal forma que los tamaños de los vectores coincidan con el tamaño del vector de entrada de los circuitos, ya que este método es genérico para cualquier circuito. Este cálculo y el de los vectores de salida, lo hemos implementado en métodos separados para simplificar la función de autodiagnóstico.

Finalmente, ya tenemos todo lo necesario para la aplicación de un Algoritmo Genético y calcular un circuito equivalente a otro dado. Para ello hemos usado el paquete DEAP [3], que es un marco de trabajo (comúnmente conocido como framework) de computación evolucional. En primer lugar, creamos la clase Fitness e indicamos que nuestro algoritmo va a ser de maximización. También creamos la clase Individuo y es necesario crear una caja de herramientas en la que vamos a ir registrando todas las funciones necesarias para que el algoritmo opere. Registramos nuestro gen, que variará en función del tamaño del circuito.

Registro del gen
1. Si ncapas*npuertas <6
   a. Registramos un gen de 0 a 5
2. Si no
   a. Registramos un gen de 0 a ncapas*npuertas-1

Pseudocódigo 6. Registro del gen en la caja de herramientas

Puesto que tenemos seis tipos de puertas, son necesarios al menos un cromosoma que vaya de 0 a 5 para contemplar todas las puertas. Para las conexiones hay que contemplar el número de puertas totales, es por ello que, si el circuito es muy pequeño, con menos de seis puertas, el cromosoma no contemplaría todos los tipos de puertas. Si definiéramos el cromosoma en función de los tipos de puertas, en circuitos grandes no se contemplarían todas las conexiones, por lo que debemos hacer esta distinción. Si en lugar de representar en un mismo cromosoma las puertas y conexiones, lo hiciéramos en dos y lo integráramos, no tendríamos este problema. Inicialmente esa fue nuestra idea, pero el rendimiento del algoritmo se veía afectado.

Después hay que definir el tamaño del cromosoma y registrarlo en nuestra caja de herramientas. El tamaño sería tres veces el producto del número de capas por el número de puertas.

Los primeros ncapas x npuertas bits representan la programación de las puertas, y los demás bits representan las conexiones y los bits de este último bloque deben ser interpretados de dos en dos.

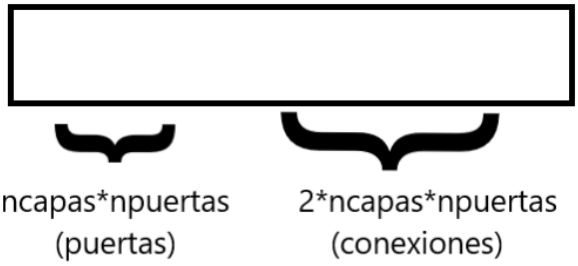

Imagen 1. Representación de un cromosoma

Tras eso, queda registrar la población, que representa el número de individuos que habrá en cada iteración del algoritmo genético, en nuestro caso lo hemos definido a 25 para que el tiempo de ejecución no sea excesivo, aunque para obtener mejores resultados sería conveniente modificar este valor.

A continuación, vamos a añadir varias funciones que nos permitan evaluar a un individuo. En primer lugar, hemos definido la función fenotipo que nos traduce los cromosomas o individuos a programaciones de puertas y conexiones, y con eso ya podremos representar un circuito. Para ello en primer lugar definiremos dos funciones que dado un gen nos devuelva la puerta o la conexión correspondiente y que serán usados por la función fenotipo. La función que decodifica una puerta es muy sencilla, recibe el gen y transforma ese gen en la programación correspondiente, o si el valor del gen no corresponde con ningún valor válido, se devuelve IAE (IllegalArgumentException).

Decodificación de puertas
Entrada:
- Gen a decodificar

Salidas:
- Programación de la puerta

Algoritmo:
progP = 'IAE'
1. Si gen = 0
   a. progP = '—'
2. Si gen = 1
   a. progP = 'AND'
3. Si gen = 2
   a. progP = 'OR'
4. Si gen = 3
   a. progP = 'NOT'
5. Si gen = 4
   a. progP = 'XOR'
6. Si gen = 5
   a. progP = 'NAND'

Pseudocódigo 7. Decodificación de una puerta

La función que decodifica las conexiones recibe el gen a decodificar, si el gen vale entre 0 y el número de puertas menos 1, significa que la puerta está conectada con uno de los bits del vector de entrada. Si el valor es mayor, entonces recibe la

salida de una de las puertas de otra capa y por tanto debemos especificar la capa y la puerta de la capa con la que se conecta.

El siguiente pseudocódigo explica lo anterior:

Decodificación de conexiones
Entrada:
- Gen a decodificar

Salidas:
- Lista con las conexiones de una puerta

Algoritmo:
1. conexión = Lista vacía
2. Si gen < número de puertas
   a. conexión = gen
3. Si no
   a. i = 0
   b. Mientras i < número de capas
      i. Si gen – (i+1)*número de puertas < número de puertas y gen – (i+1)*número de puertas > 0
         1. Añadir a conexión (i)
         2. Añadir a conexión (gen – (i+1)*número de puertas)
      ii. Aumentar i una unidad

Pseudocódigo 8. Decodificación de una conexión

Con estas dos funciones ya podemos decodificar un individuo completo. La función fenotipo recibe una lista con los bits del cromosoma a decodificar y devuelve una lista con la programación de las puertas y otra con la programación de las conexiones ambas en el formato establecido anteriormente. Lee los primeros ncapas*npuertas bits, que son los correspondientes a las puertas, los decodifica usando la función Decodificación de Puertas y los añade a la lista con la programación de las puertas. A continuación, lee los bits siguientes de dos en dos, los decodifica con la función Decodificación de Conexiones y finalmente los añade a la lista con las conexiones de cada puerta. Los bits del cromosoma son interpretados de dos en dos puesto que cada puerta tiene dos conexiones. Con los datos devueltos ya podemos representar el circuito.

Fenotipo
Entrada:
- Lista con los bits de un cromosoma (Individuo)

Salidas:
- Lista con la programación de las puertas
- Lista con las conexiones de cada puerta

Algoritmo:
1. progPuertas = Lista vacía
2. conexiones = Lista vacía
3. cuentaPuerta = ncapas*npuertas
4. i = 0
5. Mientras i < 2*ncapas*npuertas
   a. Si i < ncapas*npuertas
      i. Añadimos a progPuertas lo que nos devuelve el método decodePuerta(Individuo[i])
   b. Si no
      i. conexionPuerta = Lista vacía
      ii. Añadimos a la lista anterior lo que nos devuelve el método decodeConexion(Individuo[cuentaPuerta])
      iii. Incrementamos cuentaPuerta
      iv. Añadimos a conexionPuerta lo que nos devuelve el método decodeConexion(Individuo[cuentaPuerta])
      v. Incrementamos cuentaPuerta
      vi. Añadimos conexionPuerta a conexiones
   c. Aumentamos i en una unidad

Pseudocódigo 9. Función fenotipo

Ahora vamos a definir una función que nos sirva para penalizar los genes que representen a puertas y que sean inválidas, puesto que se pueden salir de rango debido a la representación de nuestro cromosoma. Esta función junto con otra que penalice las conexiones inválidas que definiremos más adelante serán usadas para evaluar como de bueno es un cromosoma. La función penaliza puertas analiza la lista con la programación de las puertas en busca de puertas con programación 'IAE', que es la programación asignada a puertas cuyo gen no están en el rango válido (0 – 5) y por tanto no son válidas.

Penaliza Puertas
Entrada:
- Lista con la programación de las puertas(progP)

Salidas:
- Número de puertas inválidas

Algoritmo:
1. Inicializamos un contador (penalizacionPuertas) a 0
2. i = 0
3. Mientras i sea menor que el tamaño de la lista de entradas
   a. Si progP[i] == 'IAE'
      i. Aumentamos una unidad el contador penalizacionPuertas
   b. Incrementamos i una unidad

Pseudocódigo 10. Función Penaliza Puertas

La función encargada de penalizar las conexiones recibe el circuito completo y analiza, primero si las puertas son las de la primera capa, entonces la conexión debe ser un entero que representa el bit del vector de entrada que lee, y por tanto si no es así penaliza la conexión, además también se penaliza que las dos conexiones sean la misma para una puerta. Por otro lado, si la puerta a analizar es de una capa superior, se comprueba que la conexión venga dada por una lista con dos valores, uno que representa la capa en la que se sitúa la puerta cuya salida va a ser leída y otro valor que representa la puerta en cuestión. Si no es un par de valores, significaría que la puerta está conectada con el vector de entrada y se penalizaría puesto que sólo la primera capa es la que puede hacer esto. Además, se penaliza que la puerta esté conectada con una puerta que no pertenezca a la capa i-1 ni a la i-2. Todas las conexiones inválidas que se detecten se van sumando en un contador y se devuelve dicho contador. A continuación, podemos ver el pseudocódigo de este método:

Penaliza Conexiones
Entrada:
- Circuito

Salidas:
- Número de conexiones inválidas

Algoritmo:
1. Inicializamos un contador (penalizacionConexiones) a 0
2. i = 0
3. Mientras i < ncapas
    a. Si i = 0
        i. j = 0
        ii. Mientras j < npuertas
            1. Si circuito[i][j][1][0] no es un entero
                a. Incrementamos el contador
            2. Si circuito[i][j][1][1] no es un entero
                a. Incrementamos el contador
            3. Si circuito[i][j][1][0] = circuito[i][j][1][1]
                a. Incrementamos el contador
            4. Aumentamos j una unidad
    b. Si i <0
        i. j = 0
        ii. Mientras j < npuertas
            1. Si circuito[i][j][1][0] es un entero
                a. Incrementamos el contador
            2. Si no
                a. Si circuito[i][j][1][0][0] >= i o < i-2
                    i. Incrementamos el contador
            3. Si circuito[i][j][1][1] es un entero
                a. Incrementamos el contador
            4. Si no
                a. Si circuito[i][j][1][1][0] >= i o < i-2
                    i. Incrementamos el contador
            5. Si circuito[i][j][1][0] = circuito[i][j][1][1]
                a. Incrementamos el contador
            6. Aumentamos j una unidad
    c. Aumentar i una unidad

Pseudocódigo 11. Función Penaliza conexiones

Con todo esto vamos a definir la función que evalúa un individuo. Esta función recibe un cromosoma, obtenemos la programación de las puertas y las conexiones con la función fenotipo, calcularemos el circuito que representa dicho individuo, calcularemos sus penalizaciones y en caso de que sea un circuito válido, es decir, cuyo valor de penalización sea 0, calcularemos su rendimiento usando la función de autodiagnóstico definida anteriormente. Si la penalización mayor que 0, entonces el método devuelve la penalización negativa, si es 0, devuelve el rendimiento devuelto por el autodiagnóstico. Este valor es el usado por el algoritmo genético para encontrar a los mejores individuos.

Evaluar Individuo
Entrada:
- Individuo

Salidas:
- Puntuación del individuo

Algoritmo:
1. Inicializamos penalización a 0
2. Obtenemos progP y conex llamando a la función fenotipo y pasándole el parámetro de entrada Individuo
3. Calculamos el circuitoAG llamando a la función estructura circuito y pasándole los parámetros que nos devuelve la función fenotipo
4. Calculamos la penalización debido a las conexiones llamando a la función penaliza conexiones y pasándole el circuitoAG calculado en el paso anterior.
5. Calculamos la penalización debido a las puertas llamando a la función penaliza puertas y pasándole la lista con la programación obtenida en el paso 2 y se la sumamos a penalización junto con el valor obtenido en el paso anterior.
6. Si penalización = 0
    a. Calculamos el porcentaje de acierto llamando al método autodiagnóstico que recibiría entre otros el circuito modelo y el circuitoAG calculado.
7. Si no
    a. Acierto = - penalización
8. Devolvemos una tupla con (acierto,)

Pseudocódigo 12. Función Evaluar Individuo

Tras esto, nos quedaría registrar esta última función en la caja de herramientas ya que como hemos dicho antes, esta función es la que se usa para evaluar los individuos y que el algoritmo se quede con los mejores.

A continuación, es necesario registrar los operadores de mutación y cruce y algún método de selección de individuos, en nuestro caso se ha usado el método de selección por torneo.

Ya tenemos todas las herramientas necesarias para la puesta en funcionamiento del algoritmo. Hemos comentado anteriormente que el algoritmo necesita de una semilla para que funcione. El uso de una buena semilla puede verse reflejado en el rendimiento. En el ejemplo con el que llevamos trabajando en todo el documento, en función de la semilla puede o no encontrar una solución. Es cierto que afectan muchos otros factores, como la probabilidad de mutación o de cruce, el número de individuos que se contemplan en una población o el número de individuos que se seleccionan aleatoriamente en nuestro método de selección entre otros. Es por ello que repetimos el algoritmo genético con 50 generaciones varias veces cambiando dicha semilla para encontrar un resultado de manera más rápida, aunque depende del circuito a evaluar esto puede o no funcionar de manera eficiente.

Resolver circuito
Algoritmo:

1. Inicializamos cuentaCV (variable que cuenta los circuitos válidos) a 0
2. MejorValorInd = 0
3. MejorValorIndReal = 0 (Este valor es considerando las puertas defectuosas)
4. i = 0
5. Mientras i < 50000
   a. Generamos la semilla con el valor de i
   b. Ejecutamos el algoritmo genético con dicha semilla y nos devuelve la población final y un registro
   c. Por cada individuo de la población devuelta por el AG
      i. Si el valor del individuo >= MejorValorInd
         1. Incrementamos cuentaCV
         2. Calculamos el circuitoDefectuoso
         3. Calculamos el % acierto
         4. Si acierto >= MejorValorIndReal
            a. MejorValorInd = valor del individuo
            b. MejorValorIndReal = acierto
            c. MejorIndividuo = Individuo
            d. MejorRegistro = Registro
      ii. Si MejorValorIndReal == 1
         1. Salimos del bucle de los individuos
   d. Si MejorValorIndReal == 1
      i. Salimos del bucle de las semillas
6. Si cuentaCV == 0
   a. Mostramos('No se ha encontrado solución')
7. Si no
   a. Se muestra el circuito equivalente y su rendimiento

Pseudocódigo 12. Ejecución del algoritmo genético

Cabe destacar que todas las funciones establecidas para la ejecución del algoritmo genético hacen uso del circuito a sustituir y por tanto antes de cargar dichas funciones debemos cargar en memoria todos los datos relacionados con el circuito. Además, si el circuito cambia de dimensiones, el algoritmo genético debe ser cargado de nuevo en memoria puesto que las dimensiones de su cromosoma y el rango de valores que puede tomar deben ser modificados.

IV. Experimentos

A continuación, vamos a mostrar las pruebas que hemos hecho para analizar el rendimiento del algoritmo implementado. Cabe destacar que para todas las pruebas los valores de los parámetros que podemos variar han sido:

- Número de individuos en cada población: 25
- Número de elementos seleccionados aleatoriamente en el método de selección por torneo: 3
- Rango de la semilla: 0 – 50.000
- Probabilidad de cruzamiento: 0,5
- Probabilidad de mutación: 0,3
- Número de generaciones: 50

El primer experimento realizado ha sido con el circuito de ejemplo 2x2 con el que se ha desarrollado este documento. La siguiente imagen muestra dicho circuito y marcada con una estrella la puerta defectuosa. Puesto que la puerta NOT lee la primera conexión, una solución sería simplemente cambiar la conexión de la puerta NOT y que la primera sea la de la segunda OR que es la que lee.

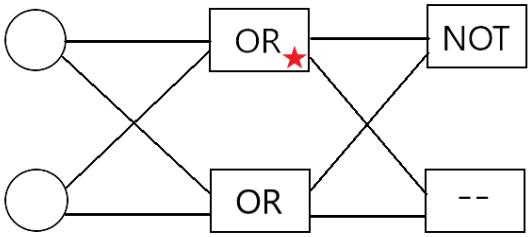

Imagen 2. Circuito 1 a reparar

El algoritmo ha tardado 14 segundos en encontrar una solución válida con rendimiento 1, es decir, equivalente al circuito anterior y que considerando las puertas defectuosas también tiene rendimiento 1 por lo podríamos cargar esta nueva configuración y el dispositivo podría seguir funcionando correctamente sin necesidad de reparación. El circuito equivalente es el siguiente:

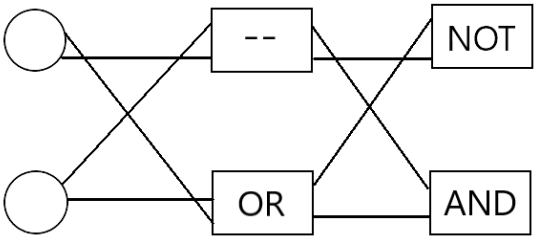

Imagen 3. Circuito equivalente al 1

Cabe destacar que los tiempos de ejecución podrían variar dependiendo del ordenador en el que se ejecute.

El segundo experimento ha tenido lugar con un circuito de 2x3, es decir dos capas con 3 puertas cada capa. Al igual que antes la primera puerta de la primera capa es la defectuosa.

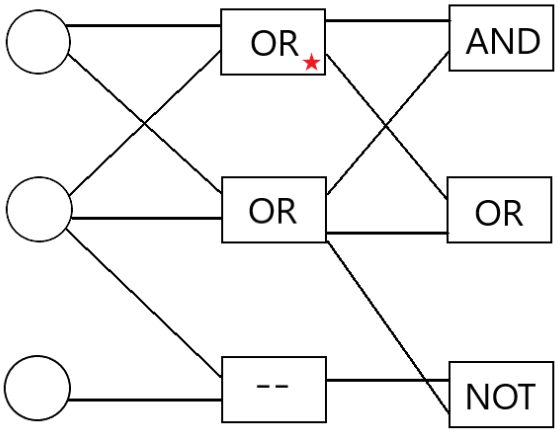

Imagen 4. Circuito 2 a reparar

El algoritmo ha tardado tan sólo 1 segundo en resolver este circuito. Existe una solución bastante más sencilla que la devuelta por el algoritmo que es cambiar la programación de la última puerta de la primera capa por una programación 'OR' e intercambiar las conexiones de la primera puerta con las de la última capa. Aun así, el algoritmo nos ha proporcionado otra solución:

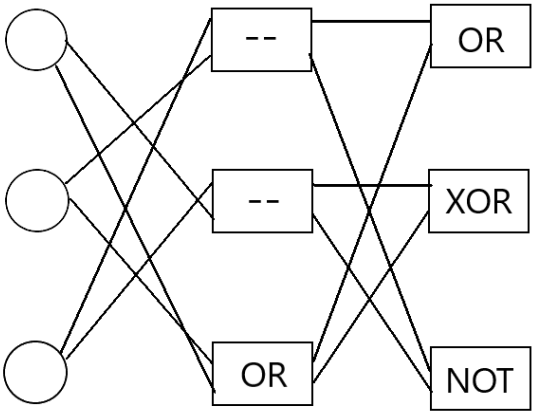

Imagen 5. Circuito equivalente al circuito 2

Esta solución también tiene rendimiento 1 sin considerar y considerando las puertas defectuosas, por los que nos podría valer como sustituto.

La siguiente prueba ha sido con un circuito de 3x3, al igual que en los anteriores con la primera puerta de la primera capa defectuosa. El algoritmo esta vez ha tardado 959 segundos en encontrar una solución válida.

Finalmente se ha hecho una prueba de un circuito de 3x5 que tiene solución, pero el algoritmo no la ha encontrado. Ha estado ejecutándose durante 5868 segundos, es decir, más de una hora y media y el mejor circuito válido encontrado es uno en el que el rendimiento del circuito devuelto tanto sin considerar las puertas defectuosas, como considerándolas, ha sido de 0.75. Cabe destacar que el rendimiento del circuito a sustituir era de 0.25, por lo que, aunque no actúa de manera correcta en todos los casos, el nuevo circuito obtenido se acerca más al funcionamiento inicial del circuito en buen estado.

V. CONCLUSIONES

Hemos implementado un algoritmo genético que encuentra soluciones para el problema presentado. Es cierto que, en circuitos muy grandes, donde las penalizaciones son muy fuertes y hay mayor probabilidad de generar circuitos inválidos el tiempo de ejecución de éste aumenta considerablemente. Pero en líneas generales, el método suele devolver un circuito que actúe mejor que el existente.

Sería conveniente en primer lugar implementar un algoritmo voraz [4] para la selección de la semilla, ya que así reduciríamos considerablemente el tiempo de ejecución. Otras mejoras que podrían hacerse sería la de contemplar más puertas lógicas, aunque deberíamos verificar que tipos de puertas se pueden implementar en las plataformas hardware a aplicar nuestro programa. También se podría incluir en un futuro una interfaz gráfica de la mano del paquete TkInter que permita introducir los parámetros necesarios para representar un circuito de manera más intuitiva.

REFERENCIAS

[1] Wikipedia – Algoritmos Evolutivos: https://es.wikipedia.org/wiki/Algoritmo_evolutivo
[2] Wikipedia – Algoritmos Genéticos : https://es.wikipedia.org/wiki/Algoritmo_gen%C3%A9tico
[3] Paquete DEAP: http://deap.readthedocs.io/en/1.0.x/
[4] Wikipedia - Algoritmo Voraz: https://es.wikipedia.org/wiki/Algoritmo_voraz