M.O.A.P
optimización de circuitos usando puertas NOR

Marian Iglesias y Aina Maull

ABSTRACT

The following project presents the implementation of an evolutionary algorithm to optimize logical circuits.
The goal is to minimize the fitness function that includes both correct outputs from the truth table and the cost of connections and NOR doors. To achieve the objective we have based our project on the biological principles that describe the evolution and selection of the fittest (through the operators presented in class).
We were asked to find the strategy that would allow us to find the result in the most fast and efficient way. Finally, we have obtained the four circuits by interpreting every time their graphics and making modifications of the values of different parameters.

INTRODUCCIÓN

Los algoritmos evolutivos se presentan como un método para encontrar soluciones óptimas a un problema lo suficientemente complejo como para no poder evaluar de forma exhaustiva todas las posibilidades. Estos algoritmos se inspiran en la mecánica de la selección natural y la genética para evolucionar una población inicial de puntos sucesivamente hacia mejores regiones del espacio de búsqueda.[2]
Hemos afrontado este proyecto como un algoritmo genético, en el cual cada individuo es una matriz de bits que codifica para un fenotipo en concreto (conexiones y puertas NOR que configuran el resultado). Para realizar una búsqueda eficiente y rápida hemos creado una población de individuos que estarán sometidos a diferentes procesos selectivos. El símil se plantea de la siguiente forma: Estos procesos de selección, recombinación, mutación e inmigración simulan un entorno en el que no todas las soluciones son aptas para la supervivencia y solo aquellas que logren sobrevivir perpetúan su código genético a las siguientes generaciones.
El propósito es encontrar las 4 matrices que representan los circuitos que dan como salida las soluciones de las diferentes tablas de la verdad solo implementando puertas NOR. Para conseguirlo, nuestro proyecto consta de un script general donde podemos encontrar todo el desarrollo de nuestro algoritmo evolutivo y la única función que hemos decidido separar ha sido la función de guía (fitness). Para crear nuestro algoritmo nos hemos marcado tres objetivos principales:

En primer lugar, hemos intentado promover la creación de variabilidad en nuestro algoritmo evolutivo, ya que así se promueve la ampliación del espacio de búsqueda y se evita caer en mínimos locales. Para conseguirlo hemos aplicado dos tipos de mutación distintos, la recombinación y una sección de población random a cada iteración.

Por otra parte, hemos intentado investigar en nuevas formas de función fitness para intentar cambiar el comportamiento lineal, ya que de esta forma se consigue guiar el algoritmo mejor sin explorar tantos espacios de búsqueda alejados del óptimo.

Finalmente, se ha intentado desde el inicio optimizar al máximo nuestro código ya que a menos tiempo para encontrar el óptimo, más pruebas puedes hacer hasta encontrar el equilibrio perfecto entre selección- mutación. Estos parámetros se deben ajustar finamente para que el entorno funcione de la mejor forma posible.

MATERIALES Y MÉTODOS

el código se compone de diferentes procesos a los que se someterà nuestra nube de soluciones.

Población inicial:

El número de individuos que se generan inicialmente es variable para cada circuito. Para optimizar las matrices de

adyacencia, hemos creado matrices de dimensiones variables que constan de tres filas más que columnas, ya que las tres primeras filas son los inputs de entrada de las tablas de verdad.
Las dimensiones iniciales y el número de conexiones (unos) son totalmente al azar. Finalmente se añade una limitación al introducir unos ya que estos no pueden estar por debajo la diagonal que existe justo después de las tres filas de inputs. Para conseguirlo configuramos la relación de coordenadas x>(y+2), ya que de este modo nos aseguramos de que es un circuito feedforward.

NUEVA GENERACIÓN:
la población que se crea como descendencia a cada iteración, constará de cuatro secciones bien definidas. Estas serán fracciones de individuos variables dependiendo de cada caso particular. En primer lugar, después de pasar por el proceso de selección, creamos un sub-vector que contiene una porción (variable según el circuito) de los individuos más adaptados de toda la población llamados "Fit". Después aplicamos el operador recombinación generando otro sub-vector llamado "Rec", al que después lo someteremos a cambios puntuales. Por otro lado, para los que se someterán a mutación serán siempre los 5 individuos más adaptados procedentes del primer vector "Fit". Finalmente, generamos otro llamado "Inm" que completará así la creación de toda la descendencia. Por lo que solo nos restará hacer la concatenación de todos estos sub-vectores creando así la generación de reemplazo.

Selección:

Para evaluar los circuitos generados se necesita de una función fitness. Esta será la que 'guiará' el algoritmo evolutivo hacia la solución, nuestro mapa, por lo tanto, acabará determinando si encontramos el óptimo de manera eficiente. Para hacer esta evaluación, diferenciamos dos partes de nuestra función: una que tiene en cuenta tanto la distancia de hamming como el número de elementos (conexiones y puertas), y otra en la cual le damos más peso a unas salidas que otras, haciendo así que esta función deje de ser lineal.

La distancia de hamming es el número de bits diferentes entre dos vectores. Para calcularla, hacemos el valor absoluto de la resta del output de la tabla de verdad con el del circuito para cada input. Por otro lado, el número de elementos será el número de columnas de la matriz más los 1 que tenga.

El circuito óptimo será aquel que tenga un número mínimo de elementos y que cumpla la tabla, siendo esta segunda condición la más importante, ya que, si un circuito es muy pequeño, pero no cumple la tabla de la verdad no nos sirve. Es por esto por lo que le damos mucho más peso a la distancia de hamming que al número de conexiones. Hasta aquí tendríamos una función lineal.

De forma intuitiva nos dimos cuenta de que la mejor manera de definir esta función era de forma no lineal ya que algunos outputs de la función le costaban más encontrarlos. Por esta razón, decidimos otorgarles más peso a ciertos outputs, de manera que si algún circuito cumplía esa salida tenía mejor fitness que si cumplía cualquier otra, redirigiendo así la búsqueda hacia el óptimo. El valor de estos pesos, los encontrábamos a prueba y error, viendo si encontraba el óptimo. En general no ha sido difícil de encontrarlos, simplemente íbamos aumentando este en caso que fuese necesario.

Nos lo podemos imaginar cómo ir navegando y que en algunas zonas haya corrientes que nos lleven a lugares que nos desvíen de la ruta adecuada. Si nuestra función de fitness no es lineal y prioriza ciertos caminos no nos arrastrará la corriente y exploraremos zonas cada vez más cercanas al óptimo.

Para intentar demostrar que realmente hay outputs que no se encuentran con la

misma facilidad, tal vez debido a que tienen 'menos grados de libertad', probamos a utilizar la función de fitness solo con la parte lineal y creamos un vector que representa cuánto le costaba encontrar cada salida. Lo que hicimos fue que, si no coincidía la salida con la esperada, íbamos sumando al peso correspondiente de dicha salida 0.0001 y si no restábamos (esto es para compensar y que no se vaya acumulando, porque al principio tendremos valores muy grandes para cualquier output, pero queremos ver cuáles se van ir cumpliendo). Esto solo lo hicimos para la tercera tabla.

De este modo se les vincula un valor escalar de fitness a cada individuo de forma que podrán ser ordenados de mayor adaptación a menor (siendo el mayor el de la fitness más baja). Entonces solo una fracción de la población, los mejore, se reproducirán copias exactas para la siguiente generación.

Recombinación:
La recombinación es un operador genético utilizado para generar variación en la población. Es análogo a la recombinación de cromosomas que se da en la reproducción sexual biológica y permite generar combinaciones de genes distintas a las de los padres produciendo así alelos quiméricos. Gracias a este proceso en nuestro código se generan nuevos individuos que tienen características propias de dos individuos explorando así nuevas opciones de búsqueda evitando caer en mínimos locales.

Concretamente en nuestro script, para no generar nuevos individuos demasiado aleatorios hemos optado por fusionar dos matrices. Una proviene de un individuo al azar de la sección de los más aptos, y la otra es una matriz aleatoria de la población restante.

El proceso de recombinación se basa inicialmente en asignar un punto de corte aleatorio (una columna). Este determinará el lugar de unión de nuestros dos individuos. Para fusionarlos hay que tener presente que tienen dimensiones distintas y por lo tanto la configuración resultante (el hijo) tendrá las dimensiones de la matriz mas pequeña. Esta limitación tendremos que tenerla en cuenta más tarde ya que se están efectuando mutaciones importantes.
El nuevo fenotipo lo configurará las primeras columnas del primer individuo, hasta el punto de recombinación, y las últimas que pertenecen al segundo. De ese modo al final obtenemos un nuevo vector que consta únicamente de todos esos individuos que se han recombinado.

Mutación:
Al igual que la recombinación, la mutación es un operador genético que permite incrementar la variabilidad de nuestra población. Se da de forma normal en la biología y consiste en modificaciones a pequeña escala que pueden ser modeladas mediante cambios puntuales en cadenas de bits.

Es importante remarcar que esta mutación se efectúa con una probabilidad preestablecida, llamada probabilidad de mutación que se tendrá que ir modificando y ajustando para cada uno de los circuitos. De hecho, en nuestro proyecto hemos configurado dos procesos de mutación distintos para poder alcanzar otros espacios de búsqueda muy próximos al óptimo que con otros operadores genéticos habría resultado casi imposible.

Nuestro primer proceso de mutación se ha aplicado a todos esos individuos que habían pasado previamente por la recombinación. Estos mutan con una probabilidad baja produciendo cambios puntuales de bits (la llamaremos mutación puntual). Estos permiten generar nuevas conexiones o quitarlas, por lo tanto, creamos otros fenotipos que nunca se habían generado en el código. Las modificaciones las conseguimos gracias a la creación de una matriz auxiliar con las mismas dimensiones e inicialmente llena de ceros. Después se le generan un número reducido de conexiones aleatorias y se procede a la

suma binaria de las dos matrices que producirá una matriz resultante, esta se comprueba que sea válida y se introduce de nuevo al vector que contiene, ahora, todos esos individuos recombinados y mutados.

Por otra parte, nuestro segundo proceso de mutación se lo aplicamos a una copia de los 5 individuos más adaptados. Esta mutación la añadimos posteriormente porque observamos que de las configuraciones más buenas les faltaban cambios muy pequeños para ser óptimas. Esta mutación consiste en hacer mutaciones puntuales de bits para crear o eliminar conexiones juntamente con otro tipo de mutación. En este caso también hay la posibilidad de eliminar una puerta NOR entera borrando columnas y sus filas respectivas enteras en las matrices.

Podríamos entender estas mutaciones como deleciones que se dan en el genoma. Cada tipo de mutación actúa con una probabilidad distinta que se va modificando para ajustarse a los diferentes circuitos.

Como hemos comentado en la recombinación, se generaba un cambio de dimensiones importante así que por esa razón no le aplicaremos el tipo de mutación que nos permite eliminar puertas lógicas, ya que estaríamos randomerizando demasiado nuestra población desajustando así el equilibrio selección-mutación.

Inmigración:
Finalmente, para poder completar toda la población que pasará a ser la nueva generación nos faltará una porción de individuos que serán el resultado de la creación de matrices totalmente aleatorias. Siguiendo el mismo procedimiento que hemos utilizado para la creación de la población inicial.

Parámetros:
Es necesario que exista un balance selección – mutación para que el algoritmo evolutivo funcione bien. Por un lado, si la selección es muy fuerte y no generamos suficiente diversidad podemos quedarnos atrapados en un mínimo local y no llegar al óptimo, pero si por el contrario tenemos demasiada diversidad estamos haciendo que nuestro algoritmo sea random. Para tener este equilibrio se tienen que ir ajustando los diferentes parámetros: la fracción de población que se recombinará, la que solo mutará, los inmigrantes, los que se mantienen y las tasas de mutación. Si hay equilibrio a medida que baja la fitness del mejor también debería bajar la media.

RESULTADOS

Al tratarse de un proceso que depende de ciertas probabilidades, puede suceder que aun introduciendo los parámetros correspondientes no salga en las iteraciones marcadas y requiera de más tiempo, pero por lo general sale en el rango propuesto. En todos los casos usamos los mismos pesos para la distancia de hamming (0.99) y para el número de elementos (0.01). En el caso de los que mutan exclusivamente, son solo los 5 mejores lo que pueden mutar.

Circuito 1:
Este circuito lo encontramos solo con mutación y selección de la población (usando una población de 500), y en este caso la fitness utilizada era lineal, es decir solo considerando distancia de hamming y número de elementos (no había salidas con más peso). Es la que encuentra más rápido.

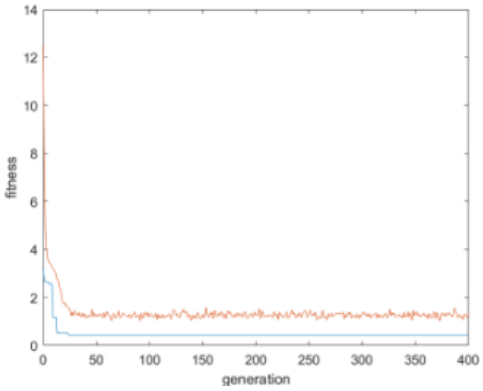


fig [1]-gráfica circuito 1 con mutación

Usando recombinación y mutación:

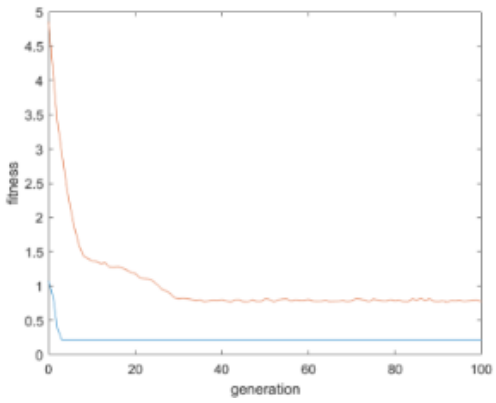


fig [2]-gráfica circuito 1 con todos los operadores

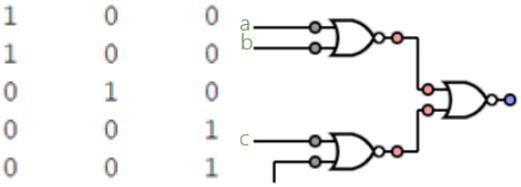


fig [3]-matriz y circuito 1

Circuito 2:

Para el circuito óptimo de la tabla dos, usamos una población de 2600 y 350 iteraciones. En la mayoría de los casos antes de las 100 iteraciones ya ha encontrado un circuito que funcione y en pocas iteraciones más encuentra el óptimo. La tasa de mutación de un bit (puntual) es de 0.1. La probabilidad de solo cambiar ciertos bits es de 0.4 y la probabilidad de solo sufrir deleción (una columna y la fila correspondiente a este nodo) es de 0.6. La estructura población queda de la siguiente manera: 547 hijos (20%, es decir el resultado de recombinación y mutacion), 211 que solo sufrirán mutación (8%, o deleción o cambiar algunos bits) y 23 inmigrantes (1%, matrices aleatorias) y el resto, 1819, sobreviven.

En este caso se le dio más peso a la última salida y a la segunda (2.5 a las dos). El circuito óptimo tiene 11 conexiones y 5 puertas NOR, por lo tanto, con un coste de 0.16

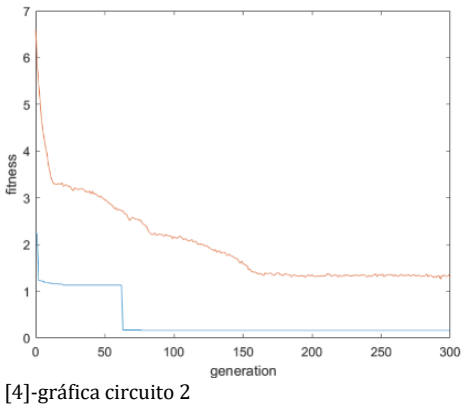


fig [4]-gráfica circuito 2

1	0	0	1	0
1	0	1	0	0
0	1	0	0	0
0	1	1	1	0
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1

fig [5]- matriz circuito 2

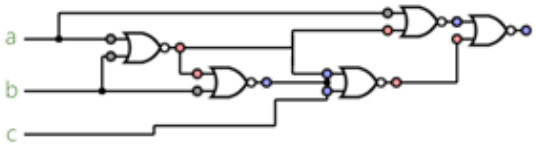


fig [6]- circuito 2

Circuito 3:

Este es el circuito que más ha costado encontrar debido a que fue más difícil saber que salidas darles más peso. En los otros circuitos (2 y 4) lo hicimos de forma más intuitiva y lo que íbamos modulando era el peso que le dábamos, pero en este caso nos guiamos por los pesos obtenidos haciendo la simulación solo con la parte lineal (explicado en el apartado de fitness). Tenían más peso la última, la segunda, la tercera y la cuarta salida (3, 2, 2.5 y 2.5 respectivamente). En este caso el 70% de la población se mantuvo (1750), 375 eran hijos, 338 solo mutaban y 37 eran inmigrantes (2500 de población en total). La tasa de mutación puntual es de 0.15 (en el caso de solo mutar tenemos 40% probabilidad de mutar algún bit) y la tasa de mutación de deleción 0.6.

```
w =                                          Salidas 1-4
    -1.9357      13.807      20.301      3.9768
                                             Salidas 5-8
-5.5695     -6.1844     -6.058      99.46
```

fig [7]-vector w sin implementación de pesos (lineal)

Una vez le ponemos los pesos adecuados:

```
w =                                          Salidas 1-4
    -26.789     -35.012     -46.876     -75.335
                                             Salidas 5-8
  -79.304     -65.325     -40.696     -59.581
```

fig [8]-vector w con implementación de pesos

El circuito óptimo obtenido consta de 6 puertas y 15 conexiones, teniendo así un coste de 0.21.

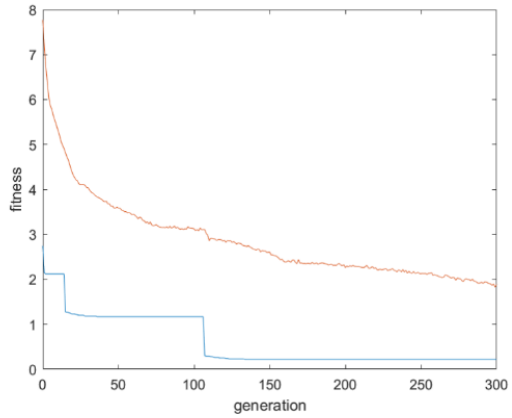


fig [9]- grafica circuito 3

1	0	1	0	1	0
0	0	1	1	0	0
1	1	0	0	0	0
0	1	1	0	1	0
0	0	0	0	0	1
0	0	0	1	1	0
0	0	0	0	0	1
0	0	0	0	0	1

fig [10]- matriz circuito 3

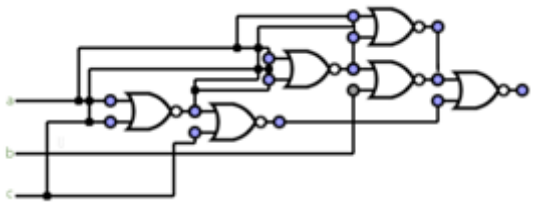


fig [11]- circuito 3

Circuito 4:

Este circuito es uno de los que encuentra más rápidamente el algoritmo, normalmente antes de las 50 iteraciones. En este caso también solo le damos más peso a una salida, a la última, un peso de 1.5. Igual que para los circuitos anteriores. Manteniendo 1819, 508 hijos, 246 que muten y solo 27 inmigrantes. La tasa de mutación puntual es de 0.15 (en el caso de solo mutar tenemos 40% probabilidad de mutar algún bit) y la tasa de mutación de deleción 0.6. En este caso la población total era también de 2600.

El circuito consta de 5 puertas y 10 conexiones, con un coste de 0.15. Lo curioso de esta tabla es que encontramos 3 circuitos óptimos diferentes con el mismo coste, siendo uno más común que los otros.

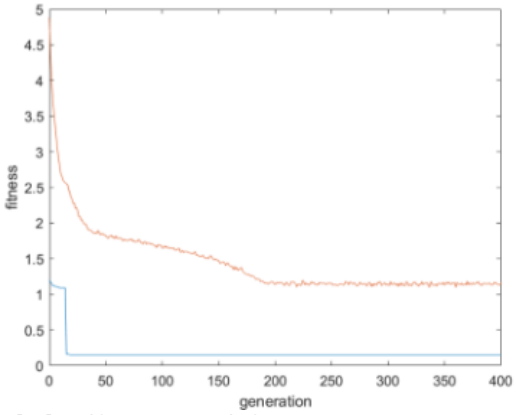


fig [12]- gráfica circuito más frecuente 4

1	0	0	0	0
0	1	0	1	0
0	0	1	1	0
0	1	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1

fig [13]-matriz circuito 4

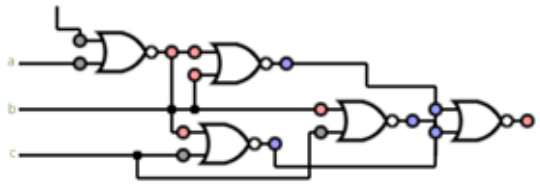
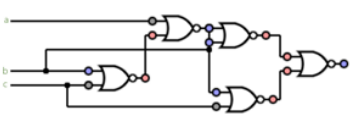


fig [14]-circuito más frecuente



0	1	0	0	0
1	0	1	0	0
1	0	0	1	0
0	1	0	0	0
0	0	1	1	0
0	0	0	0	1
0	0	0	0	1

fig [15]- segundo circuito más frecuente

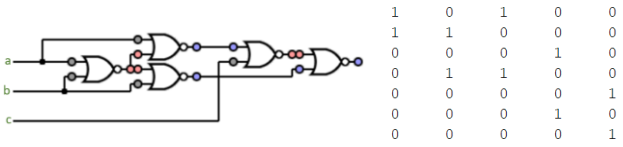


fig [16]-tercer circuito más frecuente

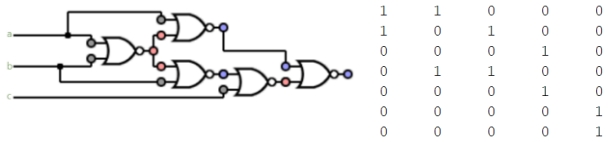


fig [15]-cuarto circuito más frecuente

DISCUSIÓN

Con la estructura poblacional propuesta y los operadores genéticos propuestos conseguimos obtener los 4 circuitos óptimos para cada tabla de la verdad. Además, algo positivo a destacar es que no solo encontraba la solución en pocas iteraciones la gran mayoría de veces sino que, el código se ejecutaba muy rápido (con iteraciones propuestas, entre 2-4 minutos dependiendo del circuito). Esto nos ha facilitado mucho la tarea de ajustar los parámetros para el equilibrio selección-mutación, es decir, fracción de población que serán hijos, que solo mutarán, tasa de mutación, etc.

Observando las gráficas que hemos obtenido de cada circuito vemos que el comportamiento entre fitness mediana y fitness del individuo más apto concuerdan con lo esperado. La línea que representa la solución final (color azul) es de forma escalonada alcanzando valores cada vez más pequeños hasta llegar al mínimo de la función. Si ahora nos fijamos en la representación de la fitness mediana vemos que entre iteraciones existe una oscilación muy pequeña. Esta es debida a que la generación descendiente no difiere tanto de la parental, consiguiendo así una búsqueda muy direccionada. Esta va bajando siguiendo el comportamiento de la fitness del más óptimo, aunque es importante destacar que siempre existirá una diferencia entre las dos líneas ya que una parte de nuestra población es totalmente aleatoria (la porción de inmigración).

Por lo que concierne a cómo elegimos los pesos de las salidas del tercer circuito podemos observar que, en el vector "w" que refleja las salidas se están cumpliendo y cuales no de las soluciones generadas, tenemos valores positivos exactamente en las salidas 2,3,4 y 8 siendo la más alta la última y las restantes con valores positivos. Lo que sugiere esto es que estas cuatro salidas positivas no se están cumpliendo y por lo tanto le otorgamos más peso de forma proporcional al valor obtenido en la prueba. Si volvemos a ejecutar el código con los pesos y sin modificar el valor del resto de parámetros, vemos que ahora todos los valores de las salidas han pasado a ser negativas por lo que concluimos que se han cumplido todas ellas. Finalmente, podemos asegurar que de alguna manera nuestra búsqueda se redirige hacia el óptimo gracias a estos pesos.

Hemos intentado crear el código más optimizado posible, aun así sabemos que hay ciertos aspectos que podríamos haber mejorado con algo más de tiempo.
Un ejemplo de ello, sería la representación del circuito, ya que al tratarse de un circuito feedforward tenemos la parte inferior a la diagonal a partir de los tres inputs, llena de ceros. Es por esta razón que pensamos en representarlos como vectores excluyendo esas partes que no nos aportan información, pero al final nos pareció mucho más intuitivo hacerlo con matrices [1].

Respecto a la recombinación, hemos planteado un proceso de fusión de dos matrices por bloques. Esto hace que que algunos de los sectores próximos van a ser mucho más difíciles de encontrar, debido a que no permite combinar, por ejemplo, la primera y la última columna de una matriz y el resto que se mantenga. Seguramente una recombinación que permita entremezclar columnas de ambas matrices sería más adecuada. Nos planteamos cambiarlo pero como ya habíamos conseguido tres de los cuatro

circuitos decidimos dejarlo como lo habíamos propuesto inicialmente.

Con respecto al operador mutacional de los que solo mutan, que incluye la posibilidad de sufrir una deleción de puertas en vez de cambiar unos pocos bits, creemos que afectó de manera positiva respecto al número de iteraciones reduciendo así la distancia mínima hasta encontrar el óptimo.

A pesar de haber utilizado una fitness no lineal, que consideramos un aspecto muy positivo de nuestro código, ya que creemos que la evaluación de cada circuito que será más real y permita una navegación más guiada entre las posibles soluciones, pensamos que existe una forma de plasmar nuestra idea de forma matemática con otra función de fitness, siendo esta también no lineal como por ejemplo de forma exponencial.

Nos hubiera gustado tener más tiempo para intentar buscar una manera de que el mismo código encuentre cuáles deberían las salidas con más peso y cuál debería ser su valor, usando una idea similar con la que encontramos los valores de los pesos del tercer circuito. El planteamiento general sería ejecutar el código una primera vez de manera lineal y hacer lo que hemos hecho para tercera salida, y saber cuáles son las salidas que más le ha costado encontrar. A partir de esta ejecución obtener unos valores proporcionales de los pesos y volver a ejecutar el código con estos pesos encontrados.

CONCLUSIÓN

el objetivo de este trabajo era encontrar el circuito Óptimo para cada una de las diferentes tablas de la verdad mediante un algoritmo evolutivo usando operadores genéticos. Para que este algoritmo funcione tiene que existir un balance entre selección y mutación que se consigue usando diferentes valores de los parámetros para cada caso. son los pesos de nuestra función fitness los que hacen que esta no sea lineal y los que permiten encontrar la solución óptima, ya que sin estos no funcionaría sin cambiar el valor de otros parámetros como los pesos otorgados a la distancia de hamming y a las conexiones.

"machines take me by surprise with great frequency"
-Alan Turing-

REFERENCIAS

[1] http://www.genetic/programming.com /published /eetimes060396.html

[2] Barry Shackleford *et al.* Synthesis of Minimum-Cost Multilevel Logic Networks via Genetic Algorithm. SASIMI 2000, Kyoto April 6-7, 2000