Universidad Complutense de Madrid

Trabajo de fin de grado en Matemáticas

Doble Grado en Ingeniería Informática y Matemáticas

Generación exhaustiva de circuitos booleanos para el estudio de la complejidad de circuitos

Autor:
Jorge Villarrubia Elvira

Directores:
Ismael Rodríguez Laguna,
Narciso Martí Oliet

Julio 2021

Índice general

Resumen

Este trabajo busca, desde lo experimental, arrojar un poco de luz sobre cuáles son los factores que influyen para que una función booleana pueda ser computada mediante un circuito combinacional de tamaño "pequeño", o que, por contra, requiera de circuitos necesariamente "grandes". Como explicaremos, hay sobrados motivos para defender que estos factores pueden ser determinantes para entender qué problemas no están en la clase de complejidad P y marcar una línea de resolución del problema P vs NP.

En una primera parte del trabajo se ha generado una tabla, lo más grande posible, de funciones booleanas de 5 bits de entrada (y uno de salida), junto al tamaño del circuito mínimo que las computa (o al menos de un tamaño similar al mínimo). Para ello, se ha diseñado un algoritmo que recorre astutamente cientos de millones de circuitos booleanos de 5 bits de entrada, anotando las funciones que estos computan.

Como se explicará, el repertorio de puertas lógicas utilizado para los circuitos y la manera de recorrerlos han sido cuestiones fundamentales para poder garantizar que la tabla sea un buen reflejo de funciones booleanas computables con circuitos "pequeños". Sin embargo, esta tarea no ha estado exenta de inconvenientes propios de la dificultad de buscar "a ciegas" en un inmenso mar de 2^{32} funciones booleanas que, por supuesto, también serán comentados.

La tabla obtenida constituye el dataset con el que se ha trabajado en la segunda parte. En esta segunda parte se han obtenido dos conjeturas sobre los patrones o factores que pudieran compartir los datos de la tabla y que, por tanto, pudieran ser característicos de las funciones computables con circuitos "pequeños". La primera tiene que ver con la repetitividad de las funciones y la segunda con una dificultad que denominamos "distinción de pares cruzados".

Estas conjeturas, en general, han respondido a aspectos cualitativos derivados de la inspección visual de los datos o simplemente de la intuición y, por tanto, se han acompañado de distintas métricas que permitiesen cotejarlas con el dataset. Ahí, ha jugado un papel fundamental el hecho de que, por construcción del dataset, es razonable asumir que las funciones del complementario requieren circuitos "grandes". De esta forma, se han podido usar las métricas, no solo para comprobar que efectivamente las puntuaciones sobre el dataset eran como esperábamos, sino también para corroborar que el comportamiento en el complementario era claramente diferente.

Adicionalmente, se han utilizado diversas técnicas de Aprendizaje Automático con dos grandes objetivos: tratar de optimizar el uso de las métricas como elemento clasificador de funciones y tratar de extraer conjeturas a partir de los propios datos según lo aprendido por los modelos. Los resultados de estas técnicas han sido muy alentadores, pero también han estado acompañados de inconvenientes relacionados con la imposibilidad de entender el fondo de lo aprendido para darle una explicación lógica.

Finalmente, con lo aprendido en los experimentos anteriores, se han realizado pruebas a mayor escala sobre tablas de verdad mucho más grandes del problema *NP-completo Set Cover* y del problema de la clase P *Mayoría*. Con ello, se ha tratado de utilizar los modelos anteriores en una especie de test P vs NP. La teoría que da sentido a nuestro trabajo hace presagiar que los factores encontrados pueden ser determinantes para este problema, y parecía natural concluir nuestra labor tratando de comprobar en qué medida era así. Las dificultades que esto conlleva son evidentes pues esta tarea, al contrario que las anteriores, no es sistemática ni automatizable. Por ello, nos conformamos con probar un problema de cada clase con unos pocos millones de bits en sus tablas de verdad. También han sido necesarias simplificaciones tales como cierto muestreo en las tablas de verdad que fue sesgado astutamente para que resultase representativo.

Los resultados, tanto de este experimento como de los anteriores, han sido sorprendentemente buenos e intrigantes. Destacamos el buen quehacer de nuestras métricas y de una red neuronal que se comportó extremadamente bien para diferenciar nuestro dataset del complementario. Esta red también funcionó de manera muy interesante para las muestras de *Set Cover* y *Mayoría* en el experimento final.

Abstract

This work attempts to shed some light on the main factors that make a Boolean function computable using combinational circuits of "small" sizes, or "big" circuit instances. As we will cover, there are plenty of reasons that support that these factors might be the key to understanding problems that are not in the P complexity class, and therefore draw up a strategy to address the P vs NP problem.

On the first hand, we generated a table, as big as possible, of Boolean functions of 5 bits input (and 1 bit output) along with the minimum size of a circuit that can compute the function (or at least of similar size). In order to do this, we developed an algorithm that iterates wisely hundreds of millions of 5-bits input Boolean circuits, writing down the functions that compute them.

As we will explain, the set of logic gates used to generate the circuits, and the way of iterating over them, is essential to guarantee that the table portrays a good sample of the Boolean functions computable with "small" circuits. Nevertheless, this task has not been challenge free, since it is generally quite hard to blind search over a set of 2^{32} Boolean functions. We will also detail how this task was overcome.

On the other hand, the table generated will be used as our main dataset for data analysis in the second part of the project. We then establish two conjectures on the patterns and factors that might lead to a common ground. These elements that are in common represent a sample of the characteristics of the functions computable with "small" circuits. In particular, the first conjecture presented is related to the repetitiveness of the functions, and the second is linked to the problem that we named as "crossed pairs distinction".

These conjectures have explained most of the qualitative aspects observed in the visual inspection of the table, and hence they are presented along with different metrics that can be used to collate them with the dataset. In this process, a key role was played by the fact that, due to how the dataset was generated, it was reasonable to assume that the complementary functions of the ones analysed lead to "big" circuit instances. In this sense, we were able to use the metrics not only to effectively score the dataset, but also to analyse the behaviour of the complementary functions and check that their scores clearly differed.

Additionally, we used different Machine Learning techniques to try to optimize the usage of the metrics as a function classifier and to try to extract conjectures from the data reviewing what different models can learn from it. The results of applying these techniques were really appealing, but they were also impacted by the fact that it is rather difficult to grasp some knowledge on the inner workings of the models to get human-understandable explanations.

Finally, with all that we learnt in the previous experiments, we carried out some bigger instances of tests over bigger tables related to the *NP-complete* problem *Set Cover*, and the *Majority* problem that lies on the P class. By doing this, we tried to apply the previous models to come up with a P vs NP test. The theory that underlies our work suggests that the factors found might be decisive to understand the P vs NP problem. Therefore, it looks reasonable to conclude our work by checking to what extent this was the case. The main challenges that arise when trying to do this are clear, since there was no clear way to automatically or systematically address it. For this reason, we were satisfied with testing a problem of each complexity class with a few millions of bits in their truth tables. It was also required to apply some simplifications such as sampling over the truth tables to cleverly come up with a skewed distribution of representative cases.

All in all, the results of the experiments have been fortunately good and interesting. We want to highlight the good performance of the metrics and a neural network that was able to distinguish with great accuracy instances of circuits from the dataset of his complement. This neural network was particularly useful to analyse the samples of the *Set Cover* and the *Majority* problems in the final experiment.

Capítulo 1

Introducción

1.1. Motivación, antecedentes y objetivos

Uno de los problemas más candentes en el mundo de la informática consiste en determinar si las clases de complejidad P y NP coinciden o no. Probar su coincidencia supondría una revolución para la sociedad, pues abriría la puerta a resolver de forma mucho más eficiente problemas que actualmente son inasumibles desde el punto de vista computacional. Y probar lo contrario aportaría mucha clarividencia en el campo de la Complejidad Computacional cerrando definitivamente el debate.

Ciertas preguntas relativas a este problema pueden abordarse a través de la Complejidad de Circuitos. Un resultado importante (teorema 6.6 de [1]), que da pie a los distintos experimentos realizados en este trabajo, garantiza que cualquier problema de decisión que esté en la clase P también está en la clase $P_{/poly}$, es decir, que puede ser resuelto, para cada tamaño de entrada n, por un circuito combinacional que use a lo sumo $p(n)$ puertas del repertorio NOT-AND-OR, siendo p un polinomio de una variable.

De esta forma, una posible demostración de que $P \neq NP$ podría consistir en probar que cierto problema *NP-completo* no está en la clase $P_{/poly}$ y, por tanto, tampoco está en P. Además, esta parece una vía bastante razonable para demostrar la no coincidencia, ya que si algún problema *NP-completo* estuviera en $P_{/poly}$, entre otros fenómenos inesperados colapsaría la jerarquía polinómica a su segundo nivel (teorema 6.19 de [1]). Por tanto, no se espera que haya problemas *NP-completos* que estén en $P_{/poly}$ pero no en P.

En este sentido, es fundamental conocer los factores que influyen para que un problema de decisión pueda computarse, para cada n natural, con circuitos de tamaño polinómico; lo que permite que el problema esté en $P_{/poly}$. Esto nos permitiría entender las características de los problemas que están en P y sobre todo intuir qué problemas son susceptibles de no estar en P.

Eso es lo que se ha hecho a lo largo de este trabajo: buscar, mediante diversos experimentos, cuáles pueden ser estos factores. Algo bastante novedoso hasta donde sabemos, que goza como único y principal antecedente del Trabajo de Fin de Grado de Enrique Román Calvo [5], que ya trató la endogamia en circuitos booleanos como uno de estos posibles factores.

Prácticamente todo el trabajo gira en torno a circuitos y funciones booleanas, conceptos que definiremos en el capítulo 2. Estos conceptos responden a las ideas intuitivas de función que recibe n bits de entrada y devuelve un bit de salida, y de circuito combinacional que hace lo mismo mediante una implementación con puertas lógicas. Nótese que, en principio, una misma función será calculada por muchos circuitos diferentes y nuestro interés será por el circuito mínimo (el de tamaño más favorable que, en cierto modo, determina la complejidad real de la función).

Debido a que nuestro experimento tenía que ser finito, no podíamos abordar la pertenencia a $P_{/poly}$ como se define, pues exige una acotación polinómica para cada tamaño de entrada natural; es decir, para infinitos tamaños. Debido a que tenía que ser asumible computacionalmente, tampoco parecía factible trabajar con muchos tamaños de entrada. Por tanto, nos limitamos a abordar un único tamaño de entrada medianamente representativo, que no es poco teniendo en cuenta la cantidad de funciones booleanas que hay detrás.

La idea del trabajo es fijar un tamaño de entrada n sobre el que descubrir factores influyentes en el tamaño mínimo de circuito de una función, que resulten extrapolables a los demás tamaños. En este sentido es muy importante que, desde el punto de vista teórico, entendamos el porqué de la influencia de estos factores, cerciorándonos de su independencia con respecto al tamaño n escogido. De esa forma, sin llegar a estudiar tamaños superiores (computacionalmente habría muchas dificultades en ello), podríamos aventurar la influencia de los mismos factores en los demás tamaños. En cierto modo, lo que hacemos es fijar un caso "sencillo" en busca de patrones generales para caracterizar $P_{/poly}$ (o al menos aproximarse a una caracterización), a pesar de que esta clase de complejidad se defina a infinitos tamaños.

A la hora de escoger el tamaño, si este fuera muy pequeño ocultaría la dificultad real de muchos problemas y no permitiría distinguir bien lo que es un circuito "pequeño" de lo que es un circuito "grande", que es la simplificación que vamos a hacer nosotros, pues no podemos hablar de "polinómico" o "no polinómico" habiendo fijado n, ya que estas nociones son inherentemente asintóticas (a lo largo del trabajo se precisará y justificará lo que entenderemos por "pequeño" y "grande", pero anticipamos que sus fronteras tienen que ver con el límite alcanzado por el experimento de generación de circuitos inicial).

Por contra, un tamaño demasiado grande nos limitaría mucho computacionalmente, pues la cantidad de funciones booleanas diferentes para n bits de entrada es doblemente exponencial con ese número de bits, i.e. 2^{2^n}. Basta pensar en que hay 2^n posibles entradas para una función y que, para cada una de ellas, la función puede hacer 2 cosas: devolver 0 o devolver 1. Por tanto, si queremos saber algo sobre un número representativo de las funciones booleanas que estudiemos, no podemos excedernos con el valor de n pues, de lo contrario, las que nosotros consigamos calcular resultarían una cantidad absolutamente despreciable respecto del total. En tal caso, correríamos el riego de no descubrir nada o tan solo alcanzar aspectos absolutamente triviales.

Es por ello que se decidió trabajar con $n = 5$ donde tenemos hasta $2^5 = 32$ entradas, que pueden complicar mucho la vida a las 2^{32} (algo más de 4 mil millones) funciones distintas que hay. Aunque son demasiadas funciones para computar, parecía una cantidad asumible para extraer conclusiones decentes consiguiendo "entender" algunos millones de funciones que fuesen computables con un tamaño especialmente "pequeño".

La idea era comenzar ejecutando un generador sistemático que recorriese todos los circuitos con 5 entradas y 1 salida, en orden creciente de tamaños, proporcionándonos una tabla con las funciones booleanas y el tamaño mínimo de los circuitos que las computan. Como la cantidad de estos circuitos es infinita, en algún momento hay que parar el recorrido. En ningún caso será habiendo computado todas las funciones posibles, pues al menos necesitaríamos recorrer 2^{32} circuitos (en caso de que nuestro generador fuese muy hábil), lo cual no es factible computacionalmente.

Inicialmente se pretendía recorrer algunos miles de millones de circuitos, pero veremos que el generador que diseñamos desde cero no nos permitía tanto. Sin embargo, el objetivo no era acumular circuitos sino acumular funciones, y este generador fue escogido porque ofrecía otras ventajas: recorría crecientemente en tamaño y hacía aflorar bastantes más funciones que otras opciones como reutilizar la tabla de Enrique Román o simplemente modificar su algoritmo [6].

El siguiente reto era lanzar buenas conjeturas sobre los factores que influyen en el tamaño del circuito mínimo encontrado, acompañadas de métricas que permitiesen afianzarlas sobre los datos. Traíamos ya ciertos indicios del TFG de Enrique Román[5] en relación a la repetición de patrones en las funciones booleanas computables con circuitos "pequeños" que, por supuesto, exploramos en mayor profundidad.

Cabe decir que estos factores no eran un mero espejismo y que en nuestro dataset también se manifestaron de manera importante. Sin embargo, su medición, salvo que tergiversemos mucho la métrica, no es invariante bajo permutaciones de las entradas, un hecho que explicaremos con detalle cuando tratemos el tema. Desde el punto de vista teórico puede justificarse que no tienen mucho sentido los factores que no satisfagan esta invarianza. Es por ello que pensamos que la repetitividad no es causa, sino consecuencia, de un factor que influye en el tamaño mínimo de circuitos que sí sería invariante bajo permutaciones.

Asimismo, se han explorado otras vías más centradas en la relación entre las diferentes entradas de la función teniendo en cuenta sus salidas, que, como comentaremos, tiene que ser donde está el meollo del asunto. Se han utilizado técnicas tales como redes neuronales, árboles de decisión e incluso otra métrica relacionada con la distinción de pares de entradas. Para algunas de ellas se han obtenido resultados bastante interesantes que, aunque no prueban nada, suponen un indicio empírico de que si la tabla de verdad de un problema puntúa "mal" con cierta métrica, entonces es probable que requiera un circuito "grande" para ser computado, y si puntúa "bien", entonces es probable que pueda computarse con un circuito "pequeño".

Como colofón se ha tratado de comprobar si, efectivamente, sucedía lo esperado con la tabla de verdad de un problema *NP-completo* y otro de la clase P. Esto conlleva bastantes dificultades, tales como representar instancias donde estos problemas comiencen a manifestar su complejidad, y en particular con una cantidad de bits suficientemente pequeña para que la doble exponencialidad de la tabla de verdad no nos impida realizar los cómputos. Este es uno de los motivos por el que, desde el principio, se trataron de obtener

métricas computables de manera eficiente. Como no siempre se pudo, y además nuestros modelos automáticos funcionaban exclusivamente para el tamaño 32 con el que se entrenaron, fue necesario un muestreo de ese tamaño en las tablas.

1.2. Plan de trabajo

Inicialmente esbozamos la manera de proceder en el trabajo fijando como pilares básicos: la obtención de una tabla de funciones junto a su tamaño mínimo de circuito y el desarrollo de diversas conjeturas y métricas en relación a dicho tamaño mínimo. Sin embargo, pensamos que era mejor ir encauzando un trabajo tan empírico en base a los acontecimientos y, por tanto, evitamos concretar en exceso de antemano.

Esta hoja de ruta tan básica se tradujo, con el devenir de los acontecimientos, en los siguientes puntos ya más concretos:

- Estudio en profundidad del capítulo 6 del libro *Computational Complexity - A Modern Aproach* de Sanjeev Arora y Boaz Barak [1], cuyos resultados más útiles e importantes para el trabajo ya han sido presentados en los antecedentes.
- Lectura del TFG de Enrique Román Calvo sobre la endogamia en circuitos booleanos [5], con especial atención a su *algoritmo del índice* para generar circuitos.
- Diseño, implementación y ejecución de un nuevo algoritmo generador de circuitos que proporcionase funciones booleanas de 5 bits de entrada junto al tamaño de un circuito mínimo que las computase[1].
- Definición y comprensión de la invarianza bajo permutación de las entradas que resultó importante de cara al desarrollo de conjeturas y métricas. Ampliación de los datos obtenidos con el generador apoyándose en esta invarianza para completar clases de equivalencia.
- Inspección visual de los datos obtenidos con el generador y primeras conjeturas sobre qué factores pueden influir en el tamaño del circuito mínimo que computa una función.
- Definición de métricas para cuantificar las primeras conjeturas y adaptación de las mismas según su comportamiento sobre los datos.
- Uso de diversas técnicas de Aprendizaje Automático para optimizar el uso de las métricas y encontrar nuevos patrones sobre los datos.
- Cálculo de las métricas y de los modelos de aprendizaje sobre problemas *NP-completos* y de la clase P para tamaños más grandes.
- Análisis de resultados y conclusiones.
- Elaboración de la memoria final del trabajo.

De entre los que cabe destacar, por el tiempo invertido, el desarrollo del generador de circuitos que llevó casi todo el primer cuatrimestre. La calidad de su salida se presumía fundamental para que el resto del trabajo tuviera éxito.

[1] También existía la opción de reutilizar o mejorar la tabla y el generador de Enrique Román (en lugar de desarrollar uno nuevo desde cero). Fue desestimada debido a los múltiples beneficios que podía producir, y que de hecho produjo, darle un enfoque completamente distinto al recorrido de los circuitos. Hubo un periodo de reflexión para valorar opciones y ver si era posible un recorrido alternativo que satisficiese nuestras necesidades.

Capítulo 2

Conceptos básicos y terminología

En este capítulo vamos a definir formalmente algunos de los conceptos que ya se han utilizado en la introducción y que mencionaremos frecuentemente a lo largo del trabajo. Asimismo, enunciaremos con precisión los resultados troncales que dan sentido al trabajo, aunque no los demostraremos, ya que sus pruebas pueden encontrarse fácilmente en la bibliografía y requerirían de cierta teoría previa que nosotros no necesitaremos. El objetivo es simplemente acomodar al lector para que pueda seguir el trabajo con cierta base terminológica.

2.1. Conceptos de circuito, evaluación de un circuito y función booleana

Comenzamos por definir formalmente lo que entenderemos por *circuito booleano de n entradas*, al que nos referiremos muchas veces a secas como *circuito*, pues trabajaremos con un número de entradas fijo: $n = 5$.

Definición 2.1.1. *[**Circuito booleano**] Para cada $n \geq 1$, un circuito booleano con n entradas es un grafo dirigido acíclico con m vértices sin aristas de entrada, donde $1 \leq m \leq n$, que llamaremos vértices fuente, y un vértice sin aristas de salida, conocido como sumidero. Cada vértice fuente f_i con $i \in \{0, ..., m-1\}$ lo consideraremos etiquetado con cierto e_j, no repetible, con $j \in \{0, ..., n-1\}$, para fijar la entrada que recibirá cada fuente. Y los vértices que no son fuente, que se denominarán puertas, estarán etiquetados representando las operaciones lógicas AND, OR o NOT. Además, las aristas del grafo se conocerán como cables.*

Consideraremos que dos circuitos con grafos isomorfos, donde el isomorfismo entre ellos preserva las etiquetas, son el mismo circuito. Es decir, los circuitos son los grafos cocientados por los isomorfismos g entre vértices, tales que la etiqueta de $V_i' = g(V_i)$ coincide con la etiqueta de V_i para todo V_i vértice del grafo.

Ejemplo 2.1.2. [**Circuito booleano**] Incluimos a continuación un primer ejemplo de circuito que utilizaremos más adelante para explicar conceptos.

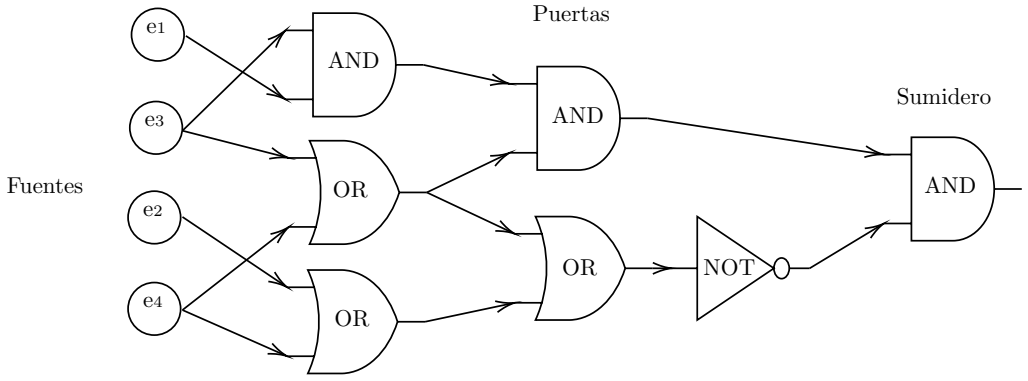
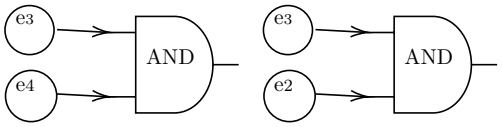


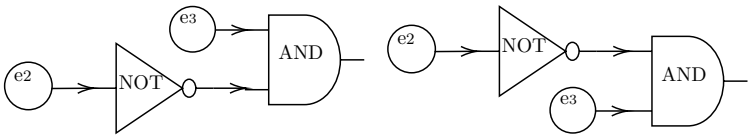
Figura 2.1.1: Ejemplo de circuito según la definición.

▲

Ejemplo 2.1.3. [**Circuitos iguales**] En cuanto a la igualdad de circuitos por isomorfismos, los siguientes circuitos son distintos debido a que sus fuentes etiquetadas son distintas y no puede existir un isomorfismo entre los grafos que las preserve:



Mientras que los siguientes circuitos son el mismo, ya que hay un isomorfismo evidente entre ellos que preserva las etiquetas:



▲

Notemos que la definición 2.1.1 recoge el caso degenerado de un grafo con un único vértice y sin aristas, donde ese vértice sería a la vez fuente y sumidero. Este caso responde a un circuito donde la salida es directamente una de las n entradas $e_1, ..., e_n$, y será el origen de nuestro algoritmo en el capítulo 3. Además, si bien se ha exigido etiquetar los vértices fuente de cara a que la evaluación del circuito sea única, no se ha pedido un orden en las entradas de las puertas, ya que, como veremos, al ser conmutativas las puertas recogidas en la definición 2.1.1, preservan la unicidad de evaluación.

La lógica de una puerta se corresponde con la tabla de verdad que conocemos, por tanto no entraremos a explicar lo que hace cada puerta, pues es lo usual y se da por supuesto. Y aunque en la definición 2.1.1 las puertas AND y OR puedan tener cualquier cantidad de aristas de entrada, no existe pérdida de generalidad si restringimos esta cantidad de cables a dos y nos limitamos a trabajar con sus *puertas binarias*, de forma que así lo haremos. Es trivial comprobar que cualquiera de estas puertas con e entradas puede construirse combinando $e - 1$ puertas binarias que representen la misma operación lógica. Por tanto, todas nuestra puertas tendrán uno o dos cables de entrada (uno si es una puerta NOT).

Definición 2.1.4. *[**Subcircuito**] Diremos que C' es un subcircuito de C si C' es un subgrafo del grafo C que cumple la definición de circuito 2.1.1. Lo denotaremos como $C' \subset_c C$*

Ejemplo 2.1.5. [**Subcircuito**] Por ejemplo, el siguiente circuito sería subcircuito del que vimos en el ejemplo 2.1.2, ya que es un subgrafo suyo que verifica la definición de circuito:

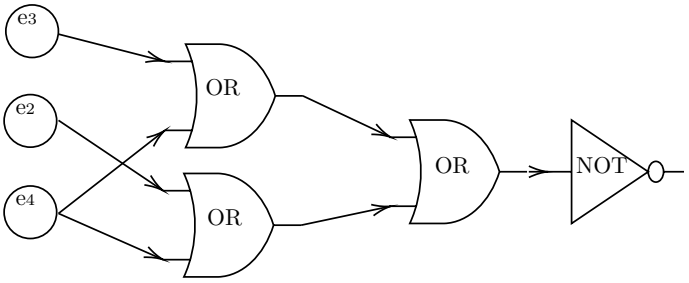


Figura 2.1.2: Subcircuito del mostrado en el ejemplo 2.1.2.

▲

El siguiente concepto será de vital importancia cuando expliquemos el generador de circuitos en el siguiente capítulo. Su cometido será el de evitar redundancias en los circuitos, que no están excluidas por la definición 2.1.1, y para nuestra labor será preferible eliminar.

Definición 2.1.6. *[**Subcircuito repetido**] Diremos que C' es un subcircuito repetido de C, si C' es subcircuito de C más de una vez. Es decir, si, considerando los subcircuitos de C, con repeticiones, C' aparece al menos dos veces. Diremos que un circuito que carece de subcircuitos tales es un circuito sin repeticiones.*

Veamos un ejemplo:

Ejemplo 2.1.7. [**Subcircuitos repetidos**] El siguiente circuito es el resultado de replicar la puerta OR que recibe e_3 y e_4 en el circuito del ejemplo 2.1.2, cuyo cable de salida iba a otras dos puertas:

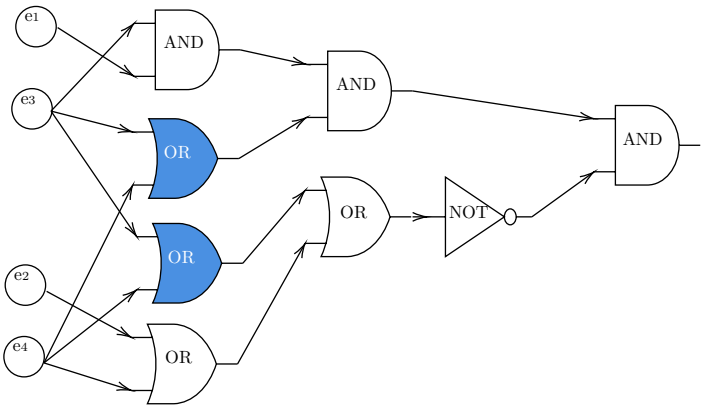
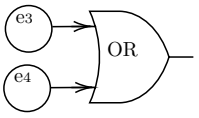


Figura 2.1.3: Circuito del ejemplo 2.1.2 con las puertas en azul replicadas.

Notemos que, intuitivamente, este circuito y el del ejemplo 2.1.2 tienen la misma función booleana asociada. Este circuito es igual que el del ejemplo inicial pero con una redundancia extra. Sin embargo, sus grafos no son isomorfos, ya que sus números de vértices son distintos (este tiene una puerta más), y, por tanto, debido a la definición 2.1.1, no son el mismo circuito.

Lo que sucede es que este circuito tiene el siguiente repetido:



Que, sin embargo, en el circuito del ejemplo 2.1.2 solo aparece como subcircuito una vez. De hecho, el circuito del ejemplo 2.1.2 no tiene ningún circuito repetido y es por tanto un circuito sin repeticiones. Este tipo de circuitos serán los que nos interesen a nosotros en el siguiente capítulo. ▲

Pasamos ahora a formalizar el concepto de evaluación de un circuito sobre cierta entrada o *input*.

Definición 2.1.8. *[**Evaluación de una fuente**] La evaluación del vértice fuente de un circuito con etiqueta e_i para cierto $i \in \{0, ..., n-1\}$ sobre la entrada $x = (x_0, ..., x_{n-1}) \in \{0,1\}^n$ es x_i y lo denotaremos como $ev(e_i, x) = x_i$.*

Definición 2.1.9. *[**Evaluación de una puerta**] La evaluación de la puerta P de $k \in \mathbb{N}^+$ aristas de entrada, sobre el input $x = (x_0, ..., x_{k-1}) \in \{0,1\}^k$ es el valor $ev \in \{0,1\}$, si en la tabla de verdad de la operación lógica asociada a la puerta P, la entrada $(x_0, ..., x_{k-1})$ tiene como salida ev. Lo denotaremos como $ev(P, x) = P(x) = P(x_0, ..., x_{k-1}) = ev$.*

Por ejemplo, como según la tabla de verdad del AND binario lógico, la operación AND entre *cierto* y *cierto* da *cierto*, según nuestra notación AND(1,1) = 1, y diremos que la evaluación de AND sobre (1,1) es 1. Implícitamente estamos haciendo una correspondencia entre el valor numérico 1 y el valor lógico cierto y entre el valor número 0 y el valor lógico falso.

Definición 2.1.10. *[**Evaluación de un circuito para una entrada**] La evaluación del circuito C para $x = (x_0, ..., x_{n-1}) \in \{0,1\}^n$, que denotaremos por $C(x) \in \{0,1\}$, es el valor ev_S, siendo S la puerta sumidero, según el siguiente esquema:*

- *$ev_V = ev(e_i, x_i) = x_i$ si V es un vértice fuente con etiqueta e_i,*

- $ev_V = ev(P, (ev_{V_0}, ..., ev_{V_{k-1}}))$ *si* V *es una puerta cuyos* k *cables de entrada tienen origen en los vértices* $V_0, ..., V_{k-1}$.

Esta definición es una manera de formalizar la idea de que se pone el valor de $(x_0, ..., x_{n-1})$ en las entradas $(e_0, ..., e_{n-1})$, según indican los subíndices, y se propaga hacia adelante siguiendo la dirección de las aristas del grafo, evaluando según la lógica de la puerta en cada caso. El resultado de la evaluación en el circuito es "lo que quede tras la última puerta". Nótese que, por generalidad en la definición, hemos hablado de puertas de k entradas, pero reiteramos que nosotros trabajaremos con puertas unarias y binarias nada mas.

Uno de los motivos por los que conjugan bien las definiciones de circuito 2.1.1 y de evaluación de circuito 2.1.10 es que las puertas AND y OR son conmutativas.

Definición 2.1.11. *[**Puerta conmutativa**] Diremos que una puerta P, que representa cierta operación lógica, es conmutativa si* $P(x_0, x_1) = P(x_1, x_0)\ \forall (x_0, x_1) \in \{0,1\}^2$.

Si alguna de las puertas no fuese conmutativa sería posible que, según cómo consideremos que recibe sus entradas dentro de un circuito, el circuito proporcionase evaluaciones distintas, lo cual no parece deseable. Es por ello que, más adelante, cuando consideremos puertas que no sean conmutativas, habrá que tenerlo en cuenta y fijar un orden en los cables que llegan a la puerta, cosa que un grafo no incorpora de manera implícita.

Pasamos ahora a medir cuantitativamente estos circuitos mediante el concepto de tamaño que está íntimamente ligado al resultado troncal que pretendemos presentar.

Definición 2.1.12. *[**Tamaño de un circuito**] El tamaño de un circuito C es el número de puertas que tiene. En ocasiones nos referiremos a él como* $|C|$.

El circuito de la figura 2.1.1 tiene tamaño 7 debido a que tiene 7 puertas: 3 puertas AND, 3 puertas OR y una puerta NOT.

Definición 2.1.13. *[**Función booleana computada por un circuito**] Diremos que una función* $f : \{0,1\}^n \to \{0,1\}$ *es una función booleana con n entradas. En particular, diremos que es la función computada por el circuito C si* $f(x) = C(x)\ \forall x \in \{0,1\}^n$. *Y lo denotaremos como* $C \to f$.

Notemos que, debido a la conmutatividad de las puertas AND y OR, la evaluación de un circuito C es única para cada entrada $x \in \{0,1\}^n$ y por tanto también lo es la función f que el circuito C computa. La conmutatividad de estas puertas puede comprobarse trivialmente en las correspondientes tablas de verdad.

Ahora que ya hemos formalizado el concepto de función, aprovechamos para hacer un inciso e introducir las representaciones que utilizaremos para ellas.

Definición 2.1.14. *[**Representaciones de una función**] Dada una función booleana* $f : \{0,1\}^n \to \{0,1\}$ *definimos:*

- *Su representación como cadena* $Rep(f)$ *dada por* $Rep(f) = (c_{2^n-1}, ..., c_0) \in \{0,1\}^{2^n}$ *donde* $f(x_0, ..., x_{n-1}) = c_i$ *si* $(x_{n-1}...x_0)_2 = (i)_{10}$. *Es decir, la cadena binaria formada por las salidas para las entradas ordenadas según su valor en decimal, considerando los bits de entrada en orden creciente de significación.*
- *Su representación decimal* $RepDec(f)$ *dada por* $(RepDec(f))_{10} = (Rep(f))_2$. *Es decir, el valor decimal para la cadena de la representación anterior considerando a esta como un número binario.*

Veámoslo con un ejemplo concreto, que seguramente se entienda mejor.

Ejemplo 2.1.15. [**Representaciones de una función**] Sea la función de 5 entradas $f : \{0,1\}^5 \to \{0,1\}$ que consiste en sacar el AND de las entradas e_0 y e_1, i.e. la función $f(e_0, e_1, e_2, e_3, e_4) = e_0$ AND e_1.

Para dicha función, $Rep(f) =$ `10001000100010001000100010001000`[1] $\in \{0,1\}^{32}$, ya que, si ordenamos las entradas por su valor binario de mayor a menor (primero la entrada `11111`, luego la `11110`, luego la `11101`, etc.), y calculamos el valor de la función para cada una de ellas, en este orden, obtenemos dicha secuencia de bits:

[1] Matemáticamente sería más correcto separar por comas las componentes aunque, para aligerar, frecuentemente utilizaremos el abuso de notación de colocarlas todas seguidas sin comas.

e_4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e_3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
e_2	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
e_1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
e_0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
f	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0

Nótese que estamos considerando que la significación de la entrada e_i la marca el subíndice i. Es decir, e_0 es la entrada menos significativa, e_1 la segunda menos significativa, etc. Por tanto, con otro orden de significación la representación como cadena será, en general, distinta. Esto tendrá su miga cuando trabajemos con métricas en el cuarto capítulo, aunque de momento no es relevante y se explicará con detalle en su debido momento.

La representación decimal de la función f es el número decimal asociado a la representación como cadena viéndola como número binario. Por tanto, $RepDec(f) = 2^3+2^7+2^{11}+2^{15}+2^{19}+2^{23}+2^{27}+2^{31} = 2290649224$, como valor decimal de $(10001000100010001000100010001000)_2$. ▲

Como ya comentamos en la introducción, buscamos trabajar con el *circuito mínimo* asociado a una función. Su tamaño cuantifica la verdadera dificultad de la función y será lo que utilizaremos más adelante. Además, en el siguiente capítulo usaremos el concepto de *circuitos equivalentes*, que es algo que trataremos de evitar, al menos de manera trivial, en nuestro recorrido sistemático. Vamos a concluir definiendo estos dos conceptos.

Definición 2.1.16. *[**Circuito mínimo que computa una función**] Dada la función $f : \{0,1\}^n \to \{0,1\}$, un circuito mínimo que la computa será cualquier circuito C de n entradas que cumpla que $C \to f$ y $|C| \leq |C'|$ $\forall C'$ tal que $C' \to f$.*

Nótese que, en general, una función puede tener varios circuitos mínimos. Nosotros no tendremos preferencia por ninguno, porque lo único que nos importa es el tamaño que todos ellos comparten trivialmente por la definición.

Definición 2.1.17. *[**Circuitos equivalentes**] Diremos que los circuitos C y C' son circuitos equivalentes si $C \to f$ y $C' \to f$ para cierta función booleana f, es decir, si computan la misma función.*

2.2. Clase $P_{/poly}$ y resultados fundamentales

Vamos ya a presentar el resultado que, como comentamos en la introducción, da sentido a nuestra labor en adelante. Para ello, hacemos una pequeña adaptación de lo que puede encontrarse en la página 108 de [1], cambiando el concepto de *lenguaje* por el de familia de funciones booleanas, ya que se ajusta mejor a lo que hemos tratado y, como vamos a justificar, no cambia en nada los resultados.

Un lenguaje L con alfabeto $\{0,1\}$ es un subconjunto de las posibles palabras formadas por estos dos símbolos. Para construir una familia de funciones $\{f_n\}_{n\in\mathbb{N}}$ que represente lo mismo, basta considerar que $\forall n \in \mathbb{N}$ y $\forall x \in \{0,1\}^n$, se cumple que $f_n(x) = 1$ si y solo si $x \in L$.

Definición 2.2.1. *[**Tamaño de una familia**] Consideremos cierta función $T : \mathbb{N} \to \mathbb{N}$. Se dice que la familia de circuitos $\{C_n\}_{n\in\mathbb{N}}$, donde el circuito C_n es de n entradas, es una familia de circuitos de tamaño T si $|C_n| \leq T(n)$ para cada $n \in \mathbb{N}$. En tal caso se dice que la familia de funciones computadas por dicha familia de circuitos, i.e. $\{f_n\}_{n\in\mathbb{N}}$ tal que $C_n \to f_n$ para cada $n \in N$, está en $SIZE(T(n))$.*

Definición 2.2.2. *[**$\mathbf{P}_{/\mathbf{poly}}$ para funciones**] La clase de complejidad $P_{/poly}$ es la formada por las familias de funciones $\{f_n\}_{n\in\mathbb{N}}$ asociadas a familias de circuitos $\{C_n\}_{n\in\mathbb{N}}$ de tamaño polinómico. Es decir:*

$$P_{/poly} = \bigcup_{c\in\mathbb{N}} SIZE(n^c)$$

Recordemos que la clase de complejidad P es la formada por los lenguajes que puede decidir una máquina de Turing en tiempo polinómico (puede verse con detalle en la página 25 de [1]). Equivalentemente, para nuestra adaptación, es la clase de complejidad formada por el conjunto de familias de funciones cuya salida puede computar una máquina de Turing en tiempo polinómico. Se tiene que:

Teorema 2.2.3. *[Condición necesaria para problemas en P]*
$P \subset P_{/poly}$ y, por tanto, una condición necesaria para que la salida de cierta familia de funciones pueda computarse por una máquina de Turing en tiempo polinómico es que exista una familia de circuitos polinómicos que computen esas funciones.

La idea es que P exige poder construir en tiempo polinómico, para cada tamaño de entrada n, el circuito C_n que computa la correspondiente función f_n, y además que este circuito tenga tamaño polinómico. Mientras que $P_{/poly}$ solo exige lo segundo, el tamaño polinómico. Es por ello que el contenido $P \subset P_{/poly}$ es estricto, pudiendo encontrarse en $P_{/poly}$ incluso problemas indecidibles (véase la versión como lenguaje unario del problema de parada de la observación 6.8 de [1]).

Sin embargo, no se espera que $NP \subseteq P_{/poly}$, pues, como demuestra el teorema de Karp-Lipton (teorema 6.19 de [1]), esto supondría el colapso de la jerarquía polinómica en su segundo nivel. Sin meternos en lo que esto significa, pues requeriría entrar en muchos asuntos que exceden nuestro cometido, es algo que los teóricos de la Complejidad Computacional ven poco probable. De esta forma, el resultado refuerza nuestro objetivo de buscar los factores que permiten a un problema estar en $P_{/poly}$ como una fuente de indicios de $P \neq NP$ que idealmente pudieran ayudar a una demostración formal: presumiblemente, habrá problemas en NP que no respondan bien a estos factores.

Además, es seguro que, para cada n, hay funciones que requieren tamaños no polinómicos para ser computadas. Es más, esto sucede para *casi todas* las funciones. Exponemos brevemente el argumento de cardinalidad utilizado por Shannon para demostrarlo, que puede encontrarse con todo detalle en el teorema 6.21 de [1].

Hay 2^{2^n} funciones diferentes de n entradas y una cantidad inferior, acotada superiormente por $2^{0.9 \cdot 2^n}$, de circuitos computables con tamaño $\frac{2^n}{10n}$, y por tanto también de funciones computables con este tamaño (la cantidad de funciones ha de ser $\leq$ que la de circuitos). Por ello, para cada n, existe al menos una función que requiere tamaño superior a $\frac{2^n}{10n}$ para computarse. Además, debido a esto, tomando una función al azar, la probabilidad de que sea computable con tamaño menor que $\frac{2^n}{10n}$ es $(2^{0.9 \cdot 2^n}/2^{2^n}) = 2^{-0.1 \cdot 2^n}$ que $\to 0$ si $n \to \infty$. Y por tanto la probabilidad del suceso complementario, es decir, de que requiera al menos ese tamaño, $\to 1$ si $n \to \infty$. Por tanto, podemos afirmar que lo cumplen *casi todas* las funciones.

Con ello sumamos un aliciente más a nuestro interés por estudiar los factores que hacen que un problema esté en $P_{/poly}$. El resultado deja entrever que, para un n "grande", son muy pocas las funciones que pueden computarse con circuitos pequeños (al menos en proporción con la cantidad de total de funciones) y por tanto su estudio es especialmente interesante, ya que dichas funciones constituyen un "pequeño reducto" que debe guardar algo especial que sea la clave para estar en $P_{/poly}$.

Capítulo 3

Generador sistemático de circuitos

Como ya hemos comentado, nuestro primer paso consiste en construir una tabla con un gran número de funciones booleanas acompañadas del tamaño de un circuito mínimo que las compute. Con ello, a través del estudio de esta tabla, después podremos tratar de determinar los factores que influyen en el tamaño de este circuito mínimo. Partimos con varias opciones: reutilizar la tabla que ya construyó Enrique Román para su TFG [5], reutilizar su *Algoritmo del Índice* [6] tratando de mejorarlo con alguna modificación, o desarrollar un algoritmo totalmente nuevo desde cero.

3.1. Discusión sobre características y objetivos del generador

Construir esta tabla astutamente es algo fundamental para que el resto del trabajo pueda tener éxito. Al fin y al cabo, y a pesar de haber simplificado el problema fijando $n = 5$ bits de entrada, nos enfrentamos a un inmenso mar de 2^{32} funciones, al que solo tenemos acceso generando circuitos y calculando la función que estos computan. Esta función será en muchos casos repetida, así que de ninguna forma podemos aspirar a cosechar tantas funciones diferentes como circuitos seamos capaces de generar. Y tampoco nos servirá de mucho obtener funciones dispersas, en el sentido de que tengan tamaños de circuito mínimo muy dispares. Dado que, por limitaciones computacionales, la cantidad de funciones que vamos a poder obtener es muy pequeña respecto del total, es fundamental que todas sean de tamaños mínimos similares para que sean representativas de cierta región de estos tamaños mínimos.

Como nuestro objetivo posterior es encontrar factores que diferencien funciones computables con circuitos "pequeños" de funciones que necesitan circuitos "grandes", es muy conveniente obtener funciones de uno de los dos polos: o bien las que necesitan los circuitos más grandes o bien las computables con los circuitos más pequeños. Dado que es más difícil lo primero, encontrar las que necesitan circuitos "grandes", pues no sabemos por dónde empezar "por arriba", surge la idea de **diseñar un algoritmo que recorra los circuitos crecientemente en tamaño**. Esto tiene múltiples beneficios. Por una parte, no nos da solo información de las funciones que encontremos, sino también de las que no consigamos encontrar. Si nuestro generador consigue llegar hasta los circuitos de tamaño m, incluidos los de este tamaño, entonces todas las funciones que no estén en nuestro *dataset* al menos necesitan tamaño $m+1$ para computarse. Por otra parte, tendremos la certeza de estar realmente encontrando el tamaño mínimo para computar la función, ya que, al recorrer crecientemente, es el tamaño del primer circuito que la compute.

El *Algoritmo del Índice* de Enrique Román [6] no cumple con estas pretensiones, ni parece fácil modificarlo para hacer que las cumpla. El algoritmo, a muy grosso modo, consiste en, fijada una profundidad, recorrer de manera exhaustiva todos los circuitos para cada posible anchura[1]. Pero, como es obvio, esto no garantiza que se pase de un tamaño al siguiente. Esta forma tan particular de recorrer los circuitos es donde reside la clave para no dejarse ninguno que no sea trivialmente equivalente a otro. Así, se puede afirmar que, potencialmente, el algoritmo puede llegar a computar todas las funciones. Es por ello que se optó por diseñar un nuevo algoritmo empezando desde cero.

El otro gran objetivo del algoritmo es **hacer aflorar funciones diferentes lo antes posible**. Todo algoritmo completo tiene la capacidad de generar todas las funciones, pero en la práctica se estará ejecutando durante un tiempo limitado y solo llegará a unas cuantas. Es preferible que, durante el tiempo que se esté ejecutando, el algoritmo recorra circuitos con funciones computadas que se repitan lo menos posible. Así, para un trabajo equivalente en cuanto a cantidad de circuitos recorridos, se obtiene un mejor resultado (más funciones en la tabla) para un algoritmo que visita "pocos" circuitos equivalentes. En esta línea, hay que tratar de evitar equivalencias obvias como grafos isomorfos, que constituyen una pérdida de tiempo al

[1] No hemos definido formalmente profundidad y anchura de circuito, porque no lo usaremos. La profundidad es el camino máximo desde una entrada a la salida del circuito y la anchura el máximo de puertas en un nivel. Puede leerse con detalle en [5].

recorrer. Y existe la intuición de que el repertorio de puertas lógicas utilizado (en principio AND, OR y NOT) puede tener que ver con la variabilidad de funciones computadas y, por tanto, debe tenerse en cuenta.

En relación a este interés por "no perder el tiempo" para maximizar la cantidad de funciones que podamos obtener, no solo es importante lo que hemos comentado de ir computando funciones muy variadas, sino también es fundamental que el cómputo de las mismas sea rápido. Un algoritmo que, de cada diez funciones que computa obtiene nueve repetidas, es preferible a uno que obtenga todas diferentes pero que explore menos de la décima parte de circuitos.

Como la generación de circuitos no parece una barrera temporal importante, porque debería consistir en una modificación constante o bastante "barata" de otro circuito, el problema podría ser la evaluación de la función que el circuito computa. Sería deseable que **el algoritmo obtuviese la evaluación de un circuito con un coste bajo**. Desde luego, sin necesidad de recorrerlo 32 veces, una por cada posible entrada, para computar la salida, que es la manera trivial de hacerlo.

3.2. Idea y explicación del algoritmo generador

La idea del algoritmo diseñado es muy simple, pero su formalización y el estudio de su corrección no tanto. Trataremos de explicarlo en adelante. El algoritmo gira en torno a dos ideas:

- Todo circuito de tamaño $n \geq 1$ sin repeticiones es el resultado de unir dos circuitos de tamaño $< n$ sin repeticiones.
- Un circuito mínimo que compute una determinada función no tiene subcircuitos repetidos.

En adelante vamos a tratar de darle formalidad a dichas ideas y a construir a partir de ello el algoritmo probando su corrección y completitud.

3.2.1. Subcircuitos padres y unión de circuitos

Vamos a empezar viendo cómo "descomponer" un circuito. Esto nos dará idea de cómo construirlos y nos resultará útil para justificar la completitud de nuestro algoritmo.

Definición 3.2.1. *[**Subcircuitos padres**] Dado un circuito C, a los subcircuitos que resultan de quitar su vértice sumidero V y sus cables de entrada, les llamaremos subcircuitos padres.*

Nótese que si C no es un circuito degenerado con tamaño 0, los padres existen y están bien llamados subcircuitos porque responden a la definición 2.1.4. Al quitar los cables que unen cierto vértice con el sumidero, dicho vértice se convierte en el nuevo sumidero. Obsérvese también que, si el sumidero es binario, habrá 2 subcircuitos padres, y si es unario, habrá solamente uno.

Ejemplo 3.2.2. [**Subcircuitos padres**] Por ejemplo, el circuito que vimos al comienzo, en el ejemplo 2.1.2, tiene los siguientes subcircuitos padres de tamaños 3 y 4 respectivamente, con una puerta AND como sumidero en el primero y NOT en el segundo:

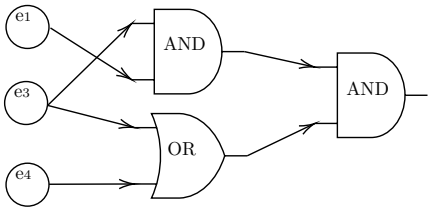


Figura 3.2.1: Subcircuito padre de tamaño 3.

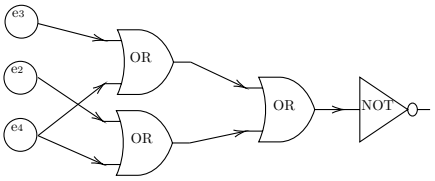


Figura 3.2.2: Subcircuito padre de tamaño 4.

Algo que llama la atención, con lo que hay que tener cuidado, es que, teniendo subcircuitos padres de tamaños

3 y 4, el tamaño del circuito unión no es $3 + 4 + 1 = 8$ debido a la puerta final extra, sino que el tamaño es 7. Esto es porque ambos subcircuitos padres tienen una puerta OR que recibe las entradas e_3 y e_4, y este subcircuito en común se comparte en la unión, con lo cual cuenta solo una vez para el tamaño.

Una "unión" ingenua de estos padres para formar un nuevo circuito, que justamente sería el que vimos en el ejemplo 2.1.7, redundaría en subcircuitos repetidos. Puesto que la repetición o no de subcircuitos influye en el tamaño, que es nuestro principal interés, reiteramos que debe tratarse con sumo cuidado. ▲

Ahora, deberíamos plantearnos si podemos "darle la vuelta" a esta descomposición de una manera deseable. Para empezar, acabamos de ver un ejemplo en el que el circuito unión no gana tamaño respecto a la suma de puertas de los padres. E incluso es posible que se pierda tamaño con respecto a esta suma: basta pensar en subcircuitos padres que puedan compartir al menos 2 puertas. Pero, ¿podría entonces un subcircuito padre tener al menos el mismo tamaño que el circuito unión? Veamos que esto no es posible.

Proposición 3.2.3. *Un circuito C de tamaño $n \geq 1$ tiene subcircuitos padres de tamaño $< n$.*

Demostración. Puesto que $n \geq 1$, el caso degenerado de circuito con tamaño 0 está excluido y se puede aplicar la observación hecha tras la definición 3.2.1 para asegurar la existencia de tales padres.

Si la puerta sumidero de C es unaria, quitándola obtenemos su único subcircuito padre C_1, que responde a un grafo con un vértice menos. Por tanto, $|C_1| = n - 1 < n$. Y si la puerta sumidero es binaria, suponiendo que uno de sus dos subcircuitos padres C_i con $i \in \{1, 2\}$ cumple que $|C_i| = m \geq n$, el circuito C tiene al menos las m puertas de C_i y la puerta sumidero, es decir, al menos $m + 1 > n$ puertas. Lo cual contradice que C tenga tamaño n. □

La visión que hemos dado hasta ahora de los circuitos es un tanto destructiva, en el sentido de que hemos explicado cómo descomponerlos en otros circuitos, que ahora sabemos que son estrictamente más pequeños. Sin embargo, lo que nosotros queremos es todo lo contrario, componer para construir circuitos que tengan estrictamente más tamaño. Para ello, vamos a definir la siguiente operación de unión de circuitos, que servirá, en cierto sentido, como inversa a la separación en padres.

Definición 3.2.4. ***[Unión de circuitos]*** *Diremos que el circuito C es la unión de los circuitos C_1 y C_2 mediante la puerta binaria P si es el grafo que resulta de juntar los grafos C_1 y C_2 eliminando los subcircuitos repetidos*[2] *y añadiendo aristas dirigidas desde los sumideros de C_1 y C_2 hacia una nueva puerta sumidero P. Lo denotaremos como $C = C_1 \sqcup_P C_2$.*

Diremos que el circuito C es la unión del circuito C_1 mediante la puerta unaria P si es el grafo que resulta de añadir una arista desde el sumidero de C_1 a una nueva puerta sumidero P. Lo denotaremos como $C = \sqcup_P C_1$.

Es evidente que esta unión de circuitos nos proporciona de nuevo un circuito, ya que se cumple trivialmente la definición 2.1.1. Veamos un ejemplo de esta unión.

Ejemplo 3.2.5. [**Unión de circuitos**] Si consideramos los siguientes circuitos:

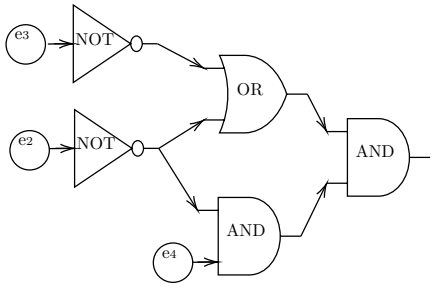


Figura 3.2.3: Circuito C_1.

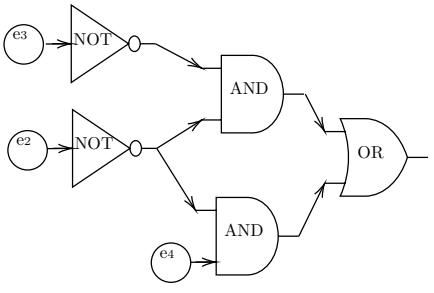


Figura 3.2.4: Circuito C_2.

[2]Reutilizando para ello la salida de la única copia que se deja de estos subcircuitos repetidos para que llegue a todos los vértices a los que llegaba habiendo repeticiones. Nótese que no se impone una metodología específica para esta eliminación. Cualquier forma de dejar una sola copia de cada subcircuito repetido, reutilizando las salidas de sus puertas como se ha indicado, será válida.

Su unión mediante una puerta AND, según la definición 3.2.4, es el circuito:

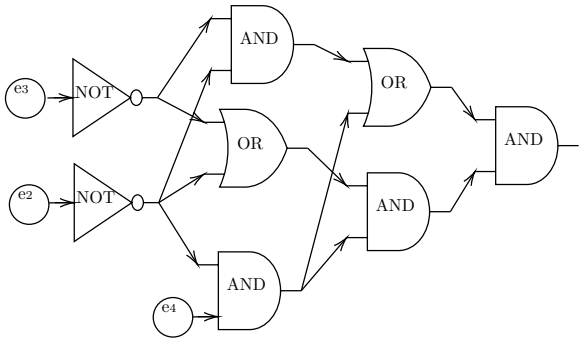
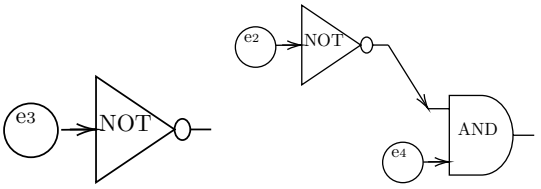


Figura 3.2.5: Circuito $C = C_1 \sqcup_{AND} C_2$.

donde solo se han considerado una vez los siguientes subcircuitos comunes tanto a C_1 como C_2:



▲

Si consideramos los subcircuitos padres del ejemplo 3.2.2 y los unimos con una AND, es trivial comprobar que obtenemos el circuito del ejemplo 2.1.2 que era el circuito original, antes de tomar subcircuitos padres. La unión que acabamos de definir resulta, de manera natural, una operación inversa a la separación en subcircuitos padres, que además elimina subcircuitos repetidos. En ese sentido va la siguiente proposición:

Proposición 3.2.6. *Todo circuito C de tamaño $n \geq 1$ sin repeticiones es la unión de circuitos de tamaño $< n$ sin repeticiones. Estos circuitos coinciden con sus subcircuitos padres.*

Demostración. Si la puerta sumidero de C es binaria, llamamos C_1 y C_2 a sus dos circuitos padres. Veamos que estos padres tienen el tamaño exigido, que no tienen repeticiones y que su unión produce C.

Por la proposición 3.2.3, dichos padres existen y tienen tamaño $< n$. Por ser subcircuitos de C, y porque C no tiene repeticiones, los padres tampoco pueden tenerlas. Si C_i, con $i = 1, 2$, tiene un subcircuito repetido C', es porque al quitar el sumidero y sus cables de entrada a C, el circuito C' aparece repetido. Por tanto, también aparecía repetido antes de hacerlo y C sí tendría subcircuitos repetidos, lo cual es contradictorio.

Y además, la unión de C_1 y C_2 proporciona C, ya que C_1 y C_2 con cables a la puerta sumidero de C es tal cual el propio C salvo porque los subcircuitos comunes a ambos aparecerán repetidos (al tomar padres los subcircuitos comunes aparecen en cada uno de ellos). Sin embargo, la unión que hemos definido en 3.2.4 se preocupa de eliminar repeticiones y, por tanto, estos subcircuitos solo aparecerán una vez.

Si la puerta sumidero de C es unaria, es trivial hacer la misma jugada. El padre existe y tiene tamaño $< n$ por la proposición 3.2.3, no puede tener repeticiones porque las tendría C y su unión con la puerta sumidero de C es literalmente inversa a tomar el subcircuito padre de C: reponer el sumidero y cable quitados al tomar el subcircuito padre. □

Un aspecto clave que se deriva de la proposición 3.2.6 es que, teniendo calculados todos los circuitos de tamaños $< n$ sin repeticiones, podríamos obtener todos los circuitos de tamaño n sin repeticiones, simplemente utilizando la operación de unión que hemos definido.

Pero a nosotros nos interesan los circuitos mínimos que computan las funciones. ¿Están estos circuitos mínimos entre los que no tienen repeticiones? Veamos que sí.

Proposición 3.2.7. *Dada la función booleana f, si C es un circuito mínimo que la computa, entonces C no tiene subcircuitos repetidos.*

Demostración. Si suponemos que C tiene un subcircuito repetido C', este no puede ser degenerado (de tamaño 0), pues sería una fuente y la definición 2.1.1 de circuito impone que no puede haber fuentes repetidas (no hay etiquetas e_i repetidas). Por tanto, $|C'| \geq 1$, y simplemente eliminándolo como repetido en C (reutilizando sus salidas), obtendríamos cierto circuito C' equivalente a C tal que $|C'| < |C|$, lo que contradice que C sea mínimo computando f. □

3.2.2. Generación de circuitos, propagación de hipótesis y cálculo del tamaño de la unión

De todo lo visto hasta el momento, se deduce una posible estrategia de recorrido. Asumiendo que tenemos calculados todos los circuitos de tamaño m sin subcircuitos repetidos para todo $m < n$, el algoritmo podría generar los circuitos de tamaño n sin repeticiones uniendo los anteriores en todas sus posibles combinaciones, con todas las posibles puertas sumidero. De esta forma, nos estaríamos quitando el trabajo absurdo de calcular circuitos que sí tuviesen repeticiones, y no estaríamos perdiendo con ello los circuitos que nos interesan, los circuitos mínimos.

Pero con esto, no solo aparecerán circuitos de tamaño n, sino también muchos circuitos de tamaños superiores: $n+1$, $n+2$, etc., que volveremos a obtener más tarde si iteramos el mismo procedimiento. Un algoritmo que, en su etapa n-ésima, generase los circuitos de tamaño n de esta forma, repetiría mucho trabajo innecesariamente, ya que en la siguiente etapa volvería a unir todos los circuitos de tamaño $< n$ entre ellos, cayendo en redundancias como las que pretendíamos evitar. Por tanto, debemos afinar un poco más el tiro.

El siguiente corolario de alternativa que se deduce trivialmente de la proposición 3.2.6 nos permite evitar la repetición de circuitos que hemos comentado:

Corolario 3.2.8. *[**Alternativa en la generación**] Todo circuito C de tamaño $n \geq 1$ sin repeticiones es la unión de circuitos de tamaño $< n$ sin repeticiones tales que:*

- *O bien todos tienen tamaño $< n - 1$,*
- *O bien alguno de ellos tiene tamaño $= n - 1$.*

Estos circuitos, además, coinciden con los subcircuitos padres de C.

Por tanto, asumiendo que además de tener todos los circuitos de tamaño $m < n$, disponemos también de todas las uniones entre circuitos de tamaño $< n-1$, por el corolario 3.2.8 basta calcular todas las uniones de circuitos de tamaño $n-1$ con circuitos de tamaño $m < n$ y añadirlas a las uniones anteriores para obtener todos los circuitos de tamaño n. Un algoritmo que, en su etapa n-ésima, haga esto para generar los circuitos de tamaño n, hace el trabajo justo, pues une cada circuito C con todos los demás exactamente una vez: cuando va por la etapa correspondiente a su tamaño.

Nuestras asunciones serán admisibles siempre y cuando tengamos un caso base en el que apoyarnos, pues, como veremos ahora, nuestro metodología las propaga etapa a etapa. Pero los circuitos degenerados de tamaño 0, que son el caso base, son pocos y bien conocidos, de manera que basta "meterlos a mano" para arrancar la generación de circuitos crecientemente en tamaño como se ha comentado. En nuestro caso, estos circuitos consisten en el grafo monovértice e_i con $i \in \{0, .., 4\}$, que es a la vez entrada y sumidero y, por tanto, carecen de repeticiones por definición.

Notación 3.2.9. Dados dos conjuntos de circuitos A y B, vamos a denotar por $A \sqcup_P B$ a la unión de todos los circuitos de A con todos los circuitos de B mediante la puerta P. Es decir, $A \sqcup_P B = \{C \sqcup_P C' \mid C \in A, C' \in B\}$. Y vamos a omitir la puerta cuando consideremos todas las puertas posibles en la unión, denotándolo como $A \sqcup B$.

Análogamente usaremos $\sqcup_P A$ para referirnos a la unión de todos los circuitos del conjunto A mediante la puerta unaria P, omitiendo también la puerta en el caso de consideremos todas las posibles: $\sqcup A$.

Vamos también a denotar por L_n al conjunto de los circuitos de tamaño $< n$ sin subcircuitos repetidos. Y por E_n al conjunto de los circuitos de tamaño exactamente n también sin repeticiones. ▲

Proposición 3.2.10. *[**Propagación de las hipótesis**] Dado $n >= 1$, se cumple lo siguiente:*

$$L_n \subset (L_{n-1} \sqcup L_{n-1}) \cup (E_{n-1} \sqcup E_{n-1}) \cup (E_{n-1} \sqcup L_{n-1}) \cup (\sqcup E_{n-1}) \cup L_{n-1} \tag{3.2.5}$$

$$(L_n \sqcup L_n) = (L_{n-1} \sqcup L_{n-1}) \cup (E_{n-1} \sqcup E_{n-1}) \cup (E_{n-1} \sqcup L_{n-1}) \tag{3.2.6}$$

Demostración. Para probar la ecuación 3.2.5, tomamos un circuito $C \in L_n$, es decir, un circuito de tamaño $< n$ que no tiene repeticiones, y vemos que pertenece a alguno de los conjuntos de la parte derecha de la inclusión.

Si su tamaño es $< n-1$, por definición $C \in L_{n-1}$.

Si su tamaño es exactamente $n-1$, debido al corolario 3.2.8:

- O bien C es la unión de circuitos de tamaño $< n-1$ sin repeticiones y, por tanto, $C \in (L_{n-1} \sqcup L_{n-1})$. (La puerta sumidero es binaria porque con unarias el padre tiene tamaño exactamente $n-1$).
- O bien es la unión de circuitos de tamaño $< n$ sin repeticiones, donde alguno de ellos tiene tamaño exactamente $n-1$. En este caso, estos circuitos son sus dos padres, que:
  - Si el sumidero de C es binario:
    - O tienen ambos tamaño $n-1$, en cuyo caso $C \in (E_{n-1} \sqcup E_{n-1})$.
    - O solo uno de ellos tiene tamaño $n-1$ y, por tanto, $C \in (E_{n-1} \sqcup L_{n-1})$.
  - Si el sumidero de C es unario, su único padre tiene tamaño $n-1$ y trivialmente $C \in \sqcup E_{n-1}$.

Para probar la ecuación 3.2.6 hacemos lo propio con uno de los contenidos ($\subset$). Tomamos $C \in (L_n \sqcup L_n)$, es decir, un circuito que es unión de dos circuitos de tamaño $< n$, y vemos que está en alguno de los conjuntos que se unen en el otro lado de la igualdad.

Suponiendo que $C = C_1 \sqcup_P C_2$ donde $C_1, C_2 \in L_n$:

- O bien $|C_1| < n-1$ y $|C_2| < n-1$, en cuyo caso $C \in (L_{n-1} \sqcup L_{n-1})$,
- O bien $|C_1| = n-1$ y $|C_2| = n-1$ en cuyo caso $C \in (E_{n-1} \sqcup E_{n-1})$,
- O bien $|C_i| = n-1$ con $i \in \{1,2\}$ y $|C_j| < n-1$ con $i \neq j \in \{1,2\}$, en cuyo caso $C \in (E_{n-1} \sqcup L_{n-1})$.

Mientras que el contenido contrario ($\supset$) sale trivialmente comprobando que con las definiciones dadas todos los conjuntos que se unen son subconjuntos de $L_n \sqcup L_n$: $(L_{n-1} \sqcup L_{n-1}), (E_{n-1} \sqcup E_{n-1}), (E_{n-1} \sqcup L_{n-1}) \subset (L_n \sqcup L_n)$.

□

La proposición 3.2.10 que acabamos de ver induce de manera natural el siguiente esquema de generación de circuitos:

Inicialmente tenemos L_0 (los 5 circuitos degenerados) y $(L_0 \sqcup L_0) = \emptyset$.

En bucle, para $n = 1, 2, 3, ...$ (hasta que se pare el algoritmo):

1. Unimos todos los circuitos de tamaño $n-1$ entre sí de todas las formas posibles con puertas binarias: $E_{n-1} \sqcup E_{n-1}$.
2. Unimos todos los circuitos de tamaño $n-1$ de todas las formas posibles con puertas unarias: $\sqcup E_{n-1}$.
3. Unimos todos los circuitos de tamaño $n-1$ de todas las formas posibles con todas las uniones entre circuitos de tamaño $< n-1$ mediante puertas binarias: $E_{n-1} \sqcup L_{n-1}$
4. Obtenemos $L_n \sqcup L_n = (L_{n-1} \sqcup L_{n-1}) \cup (E_{n-1} \sqcup E_{n-1}) \cup (E_{n-1} \sqcup L_{n-1})$
5. Obtenemos L_n, que está contenido en $(L_{n-1} \sqcup L_{n-1}) \cup (E_{n-1} \sqcup E_{n-1}) \cup (E_{n-1} \sqcup L_{n-1}) \cup (\sqcup E_{n-1}) \cup L_{n-1}$

Hay que notar que, en efecto, disponemos de los circuitos E_{n-1} que se usan en los pasos 1), 2) y 3) del esquema, porque la proposición 3.2.6 nos asegura que $E_{n-1} \subset L_{n-1} \sqcup L_{n-1}$, conjunto del que disponemos por hipótesis. Por tanto, solo falta saber cómo localizamos cada uno de los conjuntos que están contenidos en otros para ejecutar los pasos que se han descrito en el esquema. Como ya hemos comentado, las uniones del paso 1) producirán circuitos de E_n, pero también de E_{n+1}, E_{n+2}, etc. ¿Cómo sabemos cuáles son de E_n para la próxima etapa? ¿Cómo sabemos cuál es L_n dentro de lo calculado? Solo sabemos que todos sus circuitos los tenemos, pero necesitamos saber cuáles son...

Todos nuestros problemas se revuelven conociendo el tamaño de cada circuito que generemos, porque basta almacenar los circuitos indexados por tamaño para sacar fácilmente cada uno de los conjuntos que necesitamos. Todo circuito (salvo los degenerados del caso base) surge de una unión de circuitos y la siguiente proposición nos permite calcular el tamaño de una unión a partir del tamaño de los circuitos que unimos:

Proposición 3.2.11. *[**Tamaño del circuito unión**] Sean C_1 y C_2 circuitos sin repeticiones tales que $|C_1| = n_1$ y $|C_2| = n_2$. Sea $S = \{C \mid C \subset_c C_1, C \subset_c C_2, |C| \geq 1\}$[3], entonces se tiene que:*

$$|C_1 \sqcup_P C_2| = n_1 + n_2 + 1 - Card(S)$$

Demostración. El número de puertas si uniésemos C_1 y C_2 simplemente colocando cables desde sus respectivos sumideros a la nueva puerta sumidero P sería $n_1 + n_2 + 1$. Pero la unión definida en 3.2.4 adicionalmente elimina subcircuitos repetidos. Vamos a justificar que eso supone eliminar exactamente $Card(S)$ puertas.

La eliminación de repeticiones puede hacerse de cualquier forma que garantice que los subcircuitos repetidos, i.e. los circuitos de S y los repetidos de tamaño 0, aparezcan una sola vez. Denotemos al conjunto de circuitos en S de tamaño i por $S_i = \{C \in S \mid |C| = i\}$ con $i \geq 1$. Los conjuntos $S_1, S_2..., S_k$ para cierto $k \leq mín(n_1, n_2)$ constituyen una partición de S[4]. Vamos a ir eliminando repetidos en el orden que nos marcan los subíndices de la partición y contando cuántas puertas quitamos por cada circuito:

- Aunque no está en la partición, puede haber subcircuitos repetidos de tamaño 0, es decir, circuitos que son un simple vértice fuente. Estos circuitos no están en S, pero su eliminación como repetidos no supone eliminar ninguna puerta, ya que no tienen. Por tanto, no nos influyen en el contador de puertas eliminadas.
- La eliminación de los repetidos de S_i supone eliminar, para cada uno de sus circuitos, solamente su puerta sumidero. Dado $C \in S_i$ toda puerta que no sea su sumidero está en uno de sus subcircuitos

[3] Notemos que la notación $C' \subset_c C_1$, que no se había utilizado hasta ahora, se definió en 2.1.4 y significa que C es subcircuito de C_1. Y notemos que S es el conjunto de los subcircuitos tanto de C_1 como de C_2 que no son degenerados (tienen tamaño ≥ 1).

[4] Nótese que no puede haber un subcircuito tanto de C_1 como de C_2 de tamaño superior al tamaño de C_1 (n_1) o al tamaño de C_2 (n_2) Por tanto existe $k \leq mín(n_1, n_2)$ a partir del cual $S_i = \emptyset$ si $i \geq k$.

padres, que por la proposición 3.2.3 tiene tamaño $< i$. Por hipótesis, ya se han eliminado como repetidos los circuitos de C_j para todo $j < i$, entre los que estarán los padres ya que si $C \in S$ sus padres trivialmente también. Por tanto, ya se habrá eliminado dicha puerta que no es sumidero y tenemos que por cada circuito en S_i se elimina solo una puerta. En total, se eliminan $Card(S_i)$ al eliminar los repetidos de S_i según esta metodología.

Por tanto, se eliminan $\sum_{i=1}^{k} Card(S_i) = Card(S)$ puertas. □

Además, de manera trivial, conocemos el tamaño de una unión si es con una puerta unaria. Se tiene que $|\sqcup_P C| = |C| + 1$, pues la unión definida en 3.2.4 simplemente consiste en añadir a P como sumidero.

3.2.3. Aclaraciones finales sobre el algoritmo

Llegados a este punto, ya tendríamos la generación de circuitos crecientemente en tamaño que buscábamos. Sin embargo, conviene aclarar algunas cosas sobre cómo se puede implementar el esquema que hemos deducido. En especial, sobre cómo deben gestionarse la anotación de funciones encontradas y su cálculo.

Una ventaja clave de un generador que recorriera los circuitos crecientemente en tamaño era que el tamaño de los circuitos mínimos que computasen una función lo proporcionaría el primer circuito encontrado para ella.

Sin embargo, en nuestro caso no es literalmente así. Solo nos beneficiaremos de esta ventaja en parte. En la etapa n-ésima, donde, con total certeza, generamos L_n al completo, lo nuevo respecto a la etapa anterior es que tenemos E_n en su totalidad. Sin embargo, también estamos generando algunos circuitos de tamaños superiores: circuitos de E_{n+1}, E_{n+2}, ..., pero, en principio, no todos. Por tanto, es posible que, anotando que el tamaño mínimo para computar una función es $n+2$ debido a un circuito de E_{n+2} recién generado en esta etapa, luego resulte ser $n+1$ porque en la siguiente etapa descubramos otro circuito de tamaño $n+1$ que lo mejore. ¿Hay que estar sobrescribiendo tamaños todo el rato?

No, los circuitos que realmente debemos considerar como descubrimiento en la etapa n-ésima no son los que generamos: E_n, E_{n+1}, E_{n+2}..., sino los que recogemos: E_{n-1}.

Proposición 3.2.12. *Si el algoritmo en la etapa n-ésima anota exclusivamente las funciones nuevas que sean computadas por algún circuito de E_{n-1}, junto a este tamaño, entonces el algoritmo encuentra el tamaño de los circuitos mínimos para computar cualquier función.*

Demostración. Es trivial ver que toda función se llega a anotar, pues al menos se encontrará una vez: en la etapa n-ésima cuando se anoten los circuitos E_{n-1}, donde $n-1$ es el tamaño de los circuitos mínimos que la computen.

Por otra parte, hay que ver que el tamaño con el que se anota cada función es necesariamente el del sus circuitos mínimos. En la etapa n-ésima, si se anota una función, es porque aún no está anotada. Por tanto, no hay circuitos de E_i con $i < n-1$ que la computen (de lo contrario habría sido anotada en alguna etapa anterior). Y por consiguiente, el tamaño de sus circuitos mínimos es $\geq n-1$. Dado que, además, la función se anota en esta etapa, es computable por un circuito de E_{n-1} y efectivamente $n-1$ es su tamaño mínimo. □

Esto significa que, mientras se está ejecutando el generador, los circuitos de tamaño superior a la etapa por la que vamos se anotan más tarde. Pero, por supuesto, se utilizan. En la etapa n-ésima se anotan todos los circuitos de E_{n-1}, y se generan algunos circuitos de tamaño E_n, E_{n+1}, E_{n+2}, ..., que aunque no se anotan, serán recogidos en las etapas $n+1, n+2, n+3, ...$, respectivamente, estando ya todos.

Pero ¿qué pasa cuando se para el algoritmo? Como ya no recorreremos etapas superiores a la de parada, digamos la n-ésima etapa, ¿se "tiran" los circuitos de tamaños superiores a $n-1$ que fuesen generados? Sería absurdo, pues puede haber un montón de circuitos para los que el algoritmo haya invertido mucho tiempo, y descartarlos supondría que ese tiempo habría sido perdido. No, lo más lógico es anotarlos, aunque no podamos garantizar que el tamaño sea mínimo para las funciones nuevas que nos proporcionen.

Entonces, ¿podríamos tener funciones con un tamaño muy superior al de los circuitos mínimos que las computan? Si así fuese, sería incluso mejor opción no haberlas anotado, pues evitaríamos con ello contaminar nuestro dataset con datos muy alejados de la realidad. Por fortuna, no es así.

Proposición 3.2.13. *[**Acotación del tamaño generado**] Todos los circuitos generados por el algoritmo en la etapa n-ésima tienen tamaño* $\leq 2n-1$*. Además, todos los circuitos que lleve el algoritmo en la etapa n-ésima tienen tamaño* $\leq 2n-1$*.*

Demostración. Los circuitos que unimos en la etapa n-ésima tienen todos tamaño $\leq n-1$: en el paso 1) unimos $E_{n-1} \sqcup E_{n-1}$, en el paso 2) unimos $\sqcup E_{n-1}$ y en el paso 3) unimos $E_{n-1} \sqcup L_{n-1}$. Por ello:

Debido a la proposición 3.2.11 , $|C_1 \sqcup_P C_2| = |C_1| + |C_2| + 1 - Card(S) \leq n-1+n-1+1-0 = 2n-1$ para circuitos C_1 y C_2 de los que se unen en la etapa n-ésima.

Además $|\sqcup_P C_1| = |C_1| + 1 \leq n-1+1 = n \leq 2n-1$ para todo $n \geq 1$.

Por tanto, todos los circuitos que genera el algoritmo en esta etapa tienen tamaño $\leq 2n-1$. Puesto que todos los demás circuitos que lleve el algoritmo en esta etapa son generados en etapas anteriores y $2m-1 < 2n-1$ para todo $m < n$, se tiene que todos los circuitos que lleve el algoritmo en esta etapa tienen tamaño $\leq 2n-1$.

□

Con lo cual, los circuitos que anotamos una vez parado el generador no pueden llegar a apuntarse con el doble o más valor del tamaño que tienen sus circuitos mínimos. Si llegamos a la etapa n-ésima sin una función, al menos necesita tamaño n para computarse y como mucho la anotaremos con tamaño $2n-1$.

Puesto que nuestro propósito es obtener un dataset de funciones computables con circuitos "pequeños", no es un problema excesivamente grave. A efectos prácticos el tamaño de los circuitos mínimos de estas funciones descubiertas tras parar la generación se mueve entre n y $2n-1$, que no será un rango muy amplio pues probablemente no se alcanzarán valores muy grandes de n. Parece razonable entender que también son funciones computables con circuitos "pequeños". Además, ese es el caso límite, y también habrá casos en que encontremos el tamaño de los circuitos mínimos o la diferencia con el encontrado no llegue a ser casi el doble.

Finalmente, otra propiedad que, tal y como nos habíamos propuesto, se puede propagar con un coste bajo siguiendo esta metodología, es la evaluación del circuito. Aunque recorrer 32 veces, una por cada entrada, un circuito de "tamaño pequeño" (como los que llevará el algoritmo), no parece para tanto, pretendiendo generar cientos de millones de circuitos, podría ser un *hándicap* a tener en cuenta. Sin embargo, dada la evaluación de los subcircuitos de la etapa anterior en la representación como cadena, que vimos en el capítulo anterior (2.1.14), se puede calcular la representación como cadena de la función computada por el circuito unión con coste constante.

Proposición 3.2.14. *[**Evaluación de circuitos**] Para calcular la representación como cadena de un circuito unión C, basta aplicar bit a bit la operación lógica de la puerta sumidero a las representaciones 1 de los circuitos unidos.*

Demostración. El enunciado es cierto casi por definición. La representación como cadena definida en 2.1.14 consiste en una cadena de 32 bits donde el bit i-ésimo se corresponde con la evaluación del circuito para la entrada binaria $x = (i)_2 \in \{0,1\}^5$.

Tomando los valores binarios del bit i-ésimo de esta representación en los circuitos que se unen, que son sus evaluaciones para la entrada $x = (i)_2$, debido a la definición de evaluación de circuito para una entrada 2.1.10, basta evaluar la puerta sumidero sobre dichos valores (hacer la operación lógica como se definió en 2.1.9), y se obtiene el valor de evaluar C para la entrada x. Este valor es el bit i-ésimo de la representación como cadena de C. □

3.3. Implementación, ejecución y resultados del generador

3.3.1. Comentarios sobre la implementación

Por motivos de velocidad en los cálculos, el algoritmo generador fue implementado en C++. Su código fuente, para el repertorio completo de puertas (en breve se explicará el tema de los repertorios), puede encontrarse en [11].

Además, sin querer entrar en muchos detalles de implementación, cabe destacar que este lenguaje también fue escogido por su comodidad y versatilidad para trabajar con direcciones de memoria mediante punteros. Entre otras cosas, nos ha permitido una implementación muy fácil y directa de la operación de unión. Identificar subcircuitos repetidos ha consistido meramente en comparar punteros recorriendo un árbol binario, ya que cada circuito ha sido así representado, como una estructura en memoria dinámica con punteros a sus padres.

Otra ventaja es que, aunque desde el punto de vista teórico sea necesario explicar y entender la eliminación de repeticiones que hace nuestra unión definida en 3.2.4, en la práctica está implícita por el lenguaje. Un subcircuito repetido equivale a un puntero repetido (misma dirección de memoria) que apunta a un mismo objeto de memoria. Con lo cual la repetición no es tal, a efectos de gestión de memoria. Solamente es necesario identificar la repetición del puntero para calcular el tamaño de la unión según lo descrito en la proposición 3.2.11 restando tanto al tamaño como subcircuitos repetidos haya.

La versión final del algoritmo aprovecha la independencia en la generación de circuitos de cada etapa y cierta independencia para anotar funciones en la salida. Estas tareas se reparten entre distintos hilos de cómputo que se ejecutan en paralelo. Para paralelizar adecuadamente el algoritmo se utilizaron mecanismos de protección de variables compartidas tales como cerrojos y métodos de sincronización de hilos tales como barreras. Notemos que la generación de circuitos en paralelo es correcta exclusivamente etapa a etapa. Es fundamental que ningún hilo empiece a generar circuitos de la siguiente etapa sin que todos los demás hilos hayan acabado con la etapa actual. Para ello es necesario sincronizar los hilos al final de cada etapa.

3.3.2. Ejecución del generador

La ejecución del algoritmo ha estado muy condicionada por su **exigente uso de memoria**. Hasta el momento hemos destacado bastante las bondades del algoritmo, que nos marcamos como objetivos al principio del capítulo, pero no hemos puesto de manifiesto lo costoso que resulta ir "engordando una bolsa de circuitos" que no se va a vaciar hasta el final de la generación.

A pesar de que, para tratar de paliar nuestro problema, la representación mediante punteros de cada circuito fue lo más ajustada posible a las necesidades (apenas 10 bytes por circuito), no fue posible satisfacer los objetivos iniciales de moverse en los miles de millones de circuitos generados (al menos en las máquinas a las que tuvimos acceso para ejecutar el algoritmo).

Sin embargo, con apenas 500 millones de circuitos generados en la ejecución más larga, se consiguieron muy buenos resultados en cuanto a la cantidad de funciones distintas computadas, que es lo que nos importaba realmente. Para ello fue absolutamente clave probar con otros repertorios de puertas distintos al {AND,OR,NOT}.

La teoría que motivaba nuestra labor, vista en el capítulo anterior, es completamente cierta para cualquier repertorio de puertas universal; es decir, que nos permita generar cualquier función booleana. Las puertas AND, OR y NOT pueden construirse con cierto número de puertas mediante cualquiera de estos repertorios (por ser universales, estos repertorios pueden computar en particular las tablas de verdad de AND, OR y NOT). Al traducir un circuito con {AND,OR,NOT} a otro repertorio universal, hay un cambio de tamaño a lo sumo lineal debido a que la variación de tamaño de cada puerta es constante. Como un cambio lineal en el tamaño de cada circuito no altera que cierta familia de circuitos tenga o no tamaño polinómico, la clase $P_{/poly}$, definida en 2.2.2, contiene las mismas familias de funciones independientemente del repertorio utilizado.

De cara a probar con otros repertorios, nos parecía improbable que lo óptimo para que las funciones booleanas aflorasen lo antes posible fuese utilizar un repertorio "intermedio" en cuanto a su cantidad de puertas diferentes. Además de con el repertorio estándar {AND,OR,NOT}, se probó el generador con los dos extremos

posibles: el repertorio que consiste solo en la puerta NAND y un repertorio con todas las puertas binarias que se pueden concebir al que llamamos repertorio completo.

En principio, este repertorio completo estaría formado por las 16 puertas binarias definidas por todas las posibles tablas de verdad para 2 bits. Sin embargo, como se puede ver en la tabla 3.1, solo se escogieron 10 de esas puertas, puesto que las otras 6 se intuían contraproducentes por los motivos que vamos a comentar.

Puerta	Entrada 11	Entrada 10	Entrada 01	Entrada 00
~~ID0~~	0	0	0	0
NOR	0	0	0	1
RONLY	0	0	1	0
~~LNOT~~	0	0	1	1
LONLY	0	1	0	0
~~RNOT~~	0	1	0	1
XOR	0	1	1	0
NAND	0	1	1	1
AND	1	0	0	0
XNOR	1	0	0	1
~~RID~~	1	0	1	0
NLONLY	1	0	1	1
~~LID~~	1	1	0	0
NRONLY	1	1	0	1
OR	1	1	1	0
~~ID1~~	1	1	1	1

Tabla 3.1: Puertas lógicas del repertorio completo (eliminadas las puertas tachadas y señaladas en rojo)

Motivos para descartar puertas

- Las puertas ID0 e ID1 tienen salida constante, es decir, que su salida es la misma independientemente de la entrada que reciben (ID0 siempre devuelve '0' e ID1 siempre devuelve '1'). Es absurdo considerarlas como sumidero de nuestros circuitos durante la generación pues apenas permiten computar las funciones constantemente 0 y constantemente 1, que obtendremos de forma trivial en las primeras etapas (por ejemplo con el AND de una entrada y su negación). Considerar estas puertas iría en contra de nuestro objetivo de computar las máximas funciones posibles para el mismo número de circuitos generados.
- Las puertas RID y LID devuelven directamente uno de los bits de su entrada (el bit derecho o izquierdo respectivamente). Utilizarlas como sumidero en nuestros circuitos es también absurdo porque la función computada por un circuito tal sería la misma que la computada por uno de lo circuitos padres, que tienen tamaño estrictamente menor. Eliminar estas puertas nos evita perder tiempo con circuitos que no pueden ser mínimos para una función que ya se había encontrado.
- Las puertas RNOT y LNOT devuelven la negación de uno de los bits de su entrada (el bit derecho o izquierdo respectivamente). En este caso sí puede resultar interesante lo que hacen estas puertas, pero no cómo lo hacen. Para cierto circuito C, con nuestro algoritmo se calcularían los circuitos $C' \sqcup_{RNOT} C$ y $C \sqcup_{LNOT} C'$ para todo C', que son equivalentes y computan la función f tal que $\sqcup_{NOT} C \to f$. Resulta mejor sustituir estas dos puertas por la puerta unaria NOT de forma que solo se genere el circuito $\sqcup_{NOT} C$.

Por tanto, el repertorio completo quedó finalmente formado por 11 puertas, las 10 puertas binarias que aparecen en la tabla 3.1 y la puerta unaria NOT.

En nuestras pruebas apreciamos que **el repertorio NAND proporciona muchas más funciones diferentes para la misma cantidad de circuitos generados que el repertorio estándar y el repertorio completo**. En la figura 3.3.1 puede verse una comparativa de las funciones diferentes generadas por cada uno de estos repertorios para ejecuciones no demasiado exigentes del generador (una ejecución larga puede tardar horas, y suficientes ejecuciones para una gráfica decente habrían tardado demasiado).

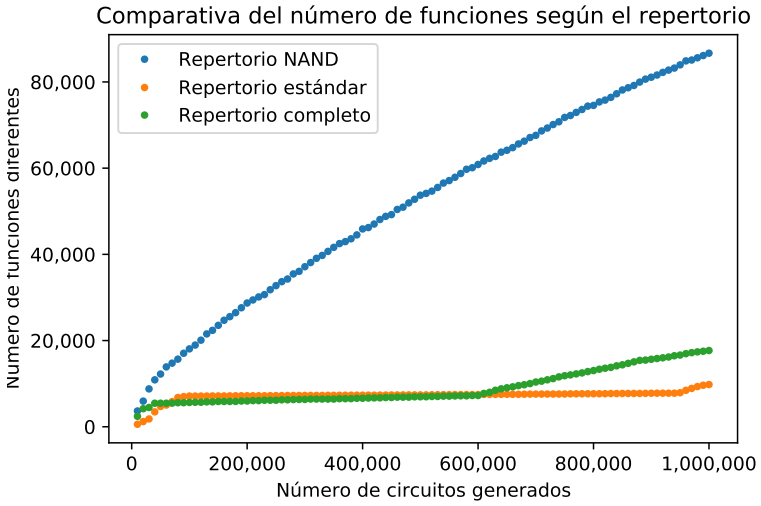


Figura 3.3.1: Cantidad de circuitos generados frente a la cantidad de funciones diferentes computadas.

Teníamos diversas intuiciones sobre por qué estos dos repertorios podían funcionar mejor que el estándar y que otros "intermedios" en cuánto al número de puertas. A la vista de los resultados tomaron fuerzas las intuiciones favorables al repertorio NAND y no tanto las favorables al repertorio completo.

La principal ventaja que veíamos en un repertorio monopuerta como el *NAND* era el hecho de explorar más estructuras de circuito diferentes para la misma cantidad de circuitos generados.

Definición 3.3.1. *[**Estructura de un circuito**] Una estructura de circuito E de n entradas es un grafo que cumple la definición de circuito 2.1.1 para n entradas, salvo por el hecho de que sus puertas no están etiquetadas. Se dice que un circuito C que coincide con E, al quitar las etiquetas de sus puertas, es un circuito con dicha estructura.*

Proposición 3.3.2. *[**Cantidad de circuitos con la misma estructura**] Sea E una estructura de circuito con conjunto de vértices puerta (vértices con alguna arista de entrada) P y sea R el conjunto de etiquetas que representan las puertas lógicas de un repertorio, donde $R_i \subset R$ denota a las que tienen aridad i. Entonces, la cantidad de circuitos con estructura E para el repertorio R es*

$$\prod_{v \in P} |R_{g^+(v)}|$$

donde $g^+(v)$ es el grado de entrada del vértice v.

Demostración. Basta observar que un vértice $v \in P$ de un circuito que tenga estructura E solo puede estar etiquetado con una puerta lógica de aridad $g^+(v)$ (de lo contrario el circuito estaría mal definido). Por tanto, ese vértice podría estar etiquetado de $|R_{g^+(v)}|$ formas diferentes. Como los vértices de P son los únicos que hay que etiquetar para concretar cualquier circuito con estructura E y cualquier combinación de etiquetas que respete la condición de aridad es válida, se sigue la fórmula del enunciado que multiplica las posibilidades para cada vértice de P entre sí. □

La proposición anterior deja entrever la enorme cantidad de circuitos con la misma estructura que pueden generarse con un repertorio con más de una puerta de alguna aridad (que aumenta exponencialmente con el número de vértices del circuito con ese grado de entrada). Mientras que con el repertorio NAND solo hay un circuito de cada posible estructura.

Ejemplo 3.3.3. [**Circuitos con una misma estructura para los repertorios estudiados.**]
En la figura 3.3.2 puede verse un ejemplo sencillo de los circuitos que tiene el repertorio NAND y el repertorio

estándar para una estructura de circuito pequeña (apenas 3 vértices con aristas de entrada). (Obsérvese que por claridad, en lugar de escribir la etiqueta en cada vértice se ha representado este con la forma habitual para estas puertas).

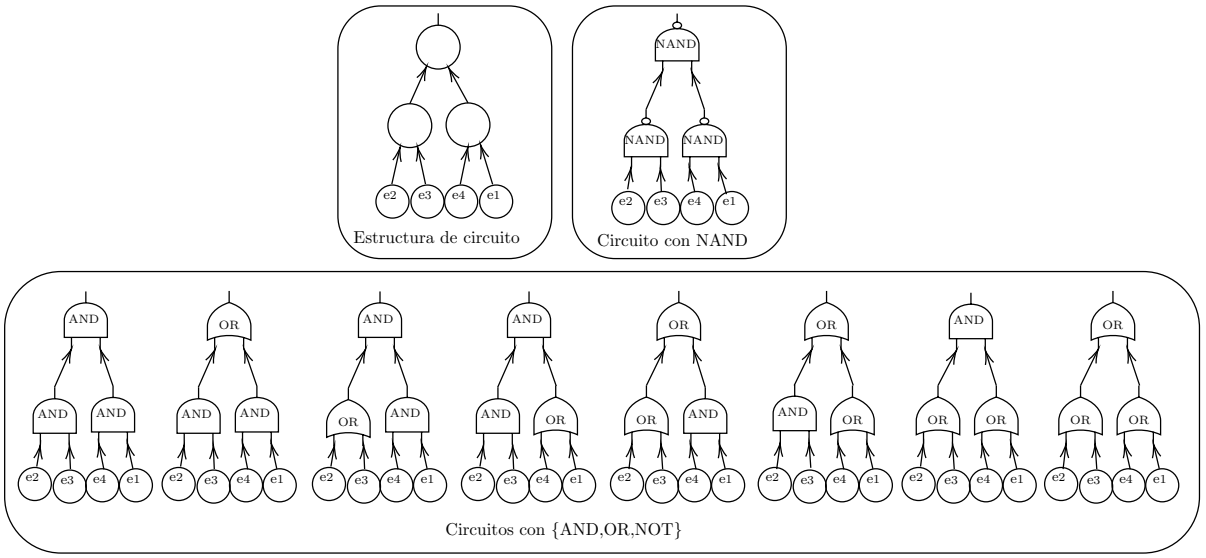


Figura 3.3.2: Circuitos con una misma estructura para el repertorio NAND y *estándar*.

Observamos que, al tener todos los vértices con aristas de entrada grado 2, hay $1^3 = 1$ circuitos posibles para el repertorio NAND (siempre sucede esto pues solo tienen sentido puertas binarias y solo hay una) y $2^3 = 8$ circuitos posibles para el repertorio estándar, debido a las puertas AND y OR (de aridad 2). Con el repertorio completo, según la proposición 3.3.2, habría hasta $10^3 = 1.000$ circuitos con esa misma estructura tan simple, debido a las 10 puertas binarias que tiene. ▲

Nuestro generador cambia de estructura de circuito constantemente con el repertorio NAND visitando exactamente una vez cada estructura, mientras que con un repertorio con muchas puertas, como el *completo*, visita muchísimas veces la misma estructura simplemente alterando la lógica de sus vértices.

Nuestra intuición era que una mayor diversidad en las estructuras de circuito podía redundar en una mayor diversidad en las funciones computadas pues, a pesar de que la alteración de puertas lógicas para una misma estructura puede proporcionar funciones diferentes, habría cierta tendencia a computar funciones "parecidas" que en muchos casos serían iguales.

Además, el hecho de generar cada estructura solamente una vez hace que aumente la profundidad de circuito antes (con menos circuitos generados). Hay motivos para afirmar que la profundidad de un circuito influye notablemente en su expresividad.

Hay ciertas familias de funciones $\{f_n\}_{n\in\mathbb{N}}$, como las asociadas a los problemas *Paridad* y *Mayoría* (que veremos más adelante), que pertenecen a $P_{/poly}$ pero no a AC^0 (teoremas 3 y 6 de [2]). La clase de complejidad AC^0 está formada por las familias de funciones computables con circuitos de tamaño polinómico pero profundidad constante y, por tanto, las mencionadas familias necesitan de una profundidad variable respecto del tamaño de la entrada para computarse con circuitos de tamaño polinómico. Esto sugiere la importancia de la profundidad para la expresividad de los circuitos.

El argumento que teníamos a favor de repertorios con muchas puertas, como el repertorio completo, es la mayor expresividad de cada vértice. Al haber muchas más posibilidades para cada vértice y poder colocarse puertas con una lógica más sofisticada (por ejemplo XOR, que necesita de hasta 4 NAND para implementarse) intuíamos una mayor expresividad local. Si bien no teníamos muy claro en qué medida esta expresividad local se traduciría en expresividad global, parecía razonable darles una oportunidad y ver que pasaba en el generador.

La mayor expresividad de las puertas permitiría a nuestro generador computar, por ejemplo, la función de *Paridad*[5] para $n = 5$, mediante el sencillo árbol de XOR de la figura 3.3.3. Esta función probablemente también llegaría a ser computada en una ejecución larga con los repertorios NAND o *completo* (con algunas puertas más), pero ejemplifica lo que podría pasar con bastantes funciones sofisticadas que solo llegaríamos a encontrar con el repertorio completo, ya que con los demás repertorios se detendría la generación antes de alcanzar su circuito mínimo.

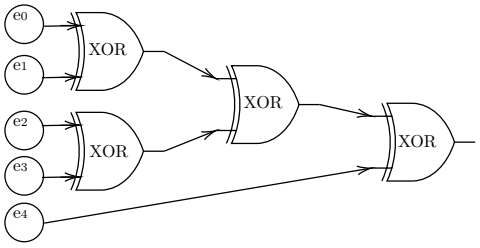


Figura 3.3.3: Árbol de XOR que computa la función *Paridad*.

Ante los resultados obtenidos cobran más sentido los argumentos que defienden el repertorio NAND, si bien es cierto que las ventajas de un repertorio con muchas puertas no deben ir muy desencaminadas pues en la figura 3.3.1 ya se aprecian más funciones computadas para el repertorio completo que para el *extándar*, lo cual se acentuó para cantidades de circuitos generados superiores.

3.3.3. Resultados finales del generador

La ejecución más larga que pudimos hacer con el generador (500 millones de circuitos) proporcionó:

- 1.655.365 funciones diferentes con el repertorio NAND.
- 486.685 funciones diferentes con el repertorio completo.
- Menos de 200.000 funciones diferentes con el repertorio estándar.

Las salidas con estas funciones, que pueden encontrarse en la carpeta [8], se mezclaron entre sí formando el dataset definitivo [10], que consta de 2.050.334 funciones diferentes. Una cantidad más que satisfactoria teniendo en cuenta que la tabla de Enrique, que podíamos reutilizar desde un principio, apenas tenía unas 600.000 funciones diferentes.

Esta mezcla supone una pequeña perturbación en los datos puesto que, para cada tamaño mínimo, hay funciones procedentes de circuitos con solo la puerta NAND y otras que proceden de circuitos que usan puertas muy sofisticadas. No es lo mismo una función con tamaño mínimo 7 porque pueda computarse con 7 puertas NAND, que una función con tamaño mínimo 7 porque pueda computarse con 7 puertas XOR.

Sin embargo, se consideró que la perturbación era asumible y que era más beneficiosa que perjudicial. Como hemos visto, suponía un aumento sustancial en el número de funciones diferentes (unas 400.000 funciones más) y, puesto que los tamaños máximos que llegó a generar nuestro algoritmo (tamaño 11 para el repertorio NAND y tamaño 7 para el repertorio completo) no eran muy grandes, la perturbación mencionada sería pequeña. La cantidad de funciones de nuestro dataset podía resultar determinante si finalmente utilizábamos técnicas de *machine learning* y, perturbaciones que no alterasen demasiado la calidad de los datos, podían resultar apropiadas para "engordar" el dataset.

Por el funcionamiento del algoritmo, nuestro dataset ya no podía ser perfecto pero tampoco era imprescindible que lo fuese. Como ya comentamos, podía haber funciones junto a un tamaño de circuito que no fuese el mínimo, lo que no era muy problemático pues la proposición 3.2.13 garantiza que el tamaño anotado en estos casos no excedía al mínimo en más del doble. Buscábamos datos representativos de funciones computables con circuitos "pequeños". Entonces ¿por qué no iba a ser legítima otra pequeña perturbación?

[5] *Paridad* consta de las funciones f_n tales que $f_n(x_0, ..., x_{n-1}) = 1$ si y solo si el cardinal de $\{x_i \mid x_i = 1, i \in \{0, ..., n-1\}\}$ es impar (es decir, determina si la entrada tiene una cantidad impar de bits a 1).

En nuestro análisis de los datos el tamaño mínimo concreto no se utilizaría pues supondría hilar demasiado fino con apenas un 0.048 % de las 2^{32} funciones que hay. ¿Qué factores influyentes en el tamaño mínimo íbamos a descubrir enfrentando entre sí un porcentaje tan pequeño de funciones cuyo tamaño mínimo apenas varía en unas unidades? Nuestro objetivo no era descubrir la diferencias entre las funciones que requieren tamaño 7 o tamaño 8, sino la diferencia entre ambos casos y las funciones que requiere tamaño mínimo 100. Por tanto, podíamos meter en el mismo saco a funciones de tamaños mínimos que fuesen razonablemente parecidos.

Para mezclar, también se utilizaron salidas de distintas ejecuciones con los mismos repertorios de puertas. Como la ejecución de cada etapa es paralela y la última etapa se detiene sin acabar, cada ejecución es no determinista y puede producir salidas diferentes (depende de cómo se haya repartido la CPU para los hilos de la última etapa). Aprovechando ese no determinismo, se obtuvieron salidas que aportaron una decenas de miles de funciones nuevas (para más detalles del aprovechamiento del no determinismo véase la sección 2.5.2 de [7]).

Capítulo 4

Análisis y estudio de los datos

El siguiente paso era utilizar los datos obtenidos con el generador para tratar de descubrir factores influyentes en el tamaño mínimo de circuito.

4.1. Consideraciones y planteamiento del estudio de los datos

Un aspecto de vital importancia para nosotros era entender, desde el punto de vista teórico (idealmente de una manera sencilla), el porqué de la influencia en el tamaño mínimo de circuito de los factores descubiertos. Mediante Aprendizaje Automático se podrían capturar fácilmente y de manera prácticamente óptima patrones en los datos, pero los modelos que suelen tener mejor acierto (fundamentalmente las redes neuronales) resultan habitualmente difíciles de descifrar.

Nuestra idea era proponer conjeturas que surgiesen de observar los datos y de distintos aspectos que nos resultaran intuitivos, utilizando el dataset para corroborarlas o desecharlas. Una vez descubiertos y entendidos ciertos factores mediante esta vía más rudimentaria, sí que utilizaríamos modelos automáticos más sofisticados.

El planteamiento de nuestro análisis de conjeturas ya se ha dejado entrever en el capítulo anterior donde se destaca el interés por recorrer crecientemente los circuitos y se justifica cierta permisividad ante perturbaciones en los datos que sean pequeñas. Hasta ahora no hemos determinado de manera precisa qué era para nosotros un circuito "pequeño" y un circuito "grande", pero hemos destacado que, por construcción, nuestro dataset resulta representativo de las funciones computables con circuitos "pequeños". Entonces, ¿por qué no considerar que las funciones del dataset son las computables con circuitos "pequeños" y las de su complementario las que requieren circuitos "grandes"?

Definición 4.1.1. *[**Funciones computables con circuitos "pequeños" y "grandes"**] El conjunto de funciones del dataset, que denotaremos como Data, es el conjunto de funciones computables con circuitos "pequeños". El conjunto de funciones complementarias* $Comp = (Fun(\{0,1\}^5, \{0,1\}) \setminus Data)$ *es el conjunto de funciones que requiere circuitos "grandes".*

A la definición anterior se le puede reprochar que en $Comp$ hay funciones con tamaño mínimo de circuito muy similar o incluso menor que para algunas funciones de $Data$. Eso es cierto, pero intuimos que es una cantidad de funciones ridícula en términos relativos, pues juega a nuestro favor la enorme cantidad de funciones totales ($|Fun(\{0,1\}^5, \{0,1\})| = 2^{32}$).

Tomando funciones aleatorias de $Comp$ debería ser extremadamente improbable que su tamaño mínimo de circuito esté cerca del tamaño máximo asociado a funciones de $Data$. En nuestro recorrido sistemático y creciente nos quedamos en apenas el 0.048 % de las funciones totales y, aunque el ritmo de aparición de funciones nuevas para cada tamaño crece a medida que el tamaño es mayor, la velocidad a la que lo hace parece insuficiente para que el 99.52 % de las funciones que faltan por descubrir aparezcan en apenas unos pocos tamaños consecutivos en torno a los tamaños 9, 10 u 11 que son los que marcan el límite de $Data$ (con el *repertorio NAND*)[1].

En la figura 4.2.3 podemos observar la cantidad de funciones que obtuvimos con el *repertorio NAND* para cada tamaño mínimo. Aunque para algunas no estamos seguros de que el tamaño encontrado sea el mínimo y solo tenemos certeza de tener todas las funciones de tamaño ≤ 6, la figura puede dar una idea de cómo es la evolución en cuanto a la cantidad de funciones diferentes. Aunque aparecen significativamente más funciones,

[1] Hay que notar que, aunque la cantidad de circuitos aumenta una barbaridad al aumentar el tamaño, cada vez son más los que resultan equivalentes a alguno anterior y computan una función repetida. Aun así, para cada tamaño serán más las funciones nuevas encontradas.

hasta completarse los más de 4 mil millones de funciones totales hay tanto margen que sospechamos que aún faltarían bastantes tamaños.

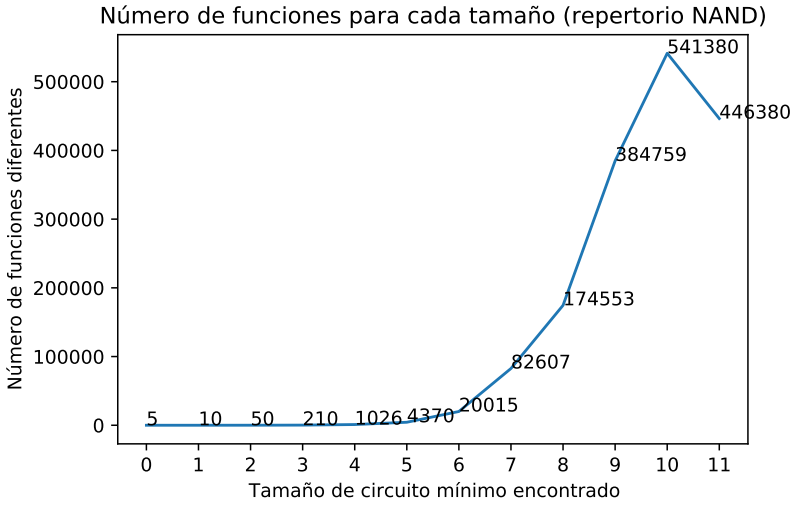


Figura 4.1.1: Cantidad de funciones encontradas según el tamaño mínimo con el *repertorio NAND*.

El principal aval para lo que estamos defendiendo son los propios resultados. Si fuese relativamente frecuente que el tamaño mínimo de las funciones de *Comp* fuese ≤ 11 o próximo a 11, nuestras métricas y modelos se comportarían necesariamente mal sobre los datos. ¿Cómo se pueden distinguir cosas que apenas se diferencian? En tal caso, las funciones de *Data* y *Comp* serían muy difíciles de distinguir y los resultados clasificando entre estos dos conjuntos a ciegas (sin que el modelo sepa la respuesta correcta) habrían de resultar malos. Ya anticipamos que no ha sido así y que se han logrado porcentajes de acierto muy altos en clasificaciones de este tipo, algunas de las cuales se han hecho mediante métricas elaboradas por nosotros que intentan fijarse en aspectos de las funciones que pensamos que pueden influir en el tamaño mínimo de circuito.

Por tanto, nuestra manera de plantear las cosas fue como problema de clasificación entre las clases *Data* y *Comp*, donde la primera significa tamaño de circuito mínimo "pequeño" y la segunda tamaño de circuito mínimo "grande".

4.1.1. Invarianza bajo permutaciones en las entradas

Hay un cuestión que, desde el punto de vista teórico, deberían cumplir los factores influyentes en el tamaño mínimo de circuito para una función. Esta cuestión es la invarianza bajo permutación de las entradas.

Dos funciones que simplemente se diferencien en las posiciones en que reciben los argumentos de entrada pero no en cómo los utilizan, son, a efectos de tamaño mínimo de circuito, totalmente equivalentes. Alterando esas posiciones apropiadamente se convertirían en la misma función y podría modificarse los circuitos que computan a una para computar a la otra sin alterar su número de puertas.

Definición 4.1.2. *[**Funciones iguales al permutar las entradas**] Diremos que dos funciones* $f_1, f_2 : \{0,1\}^5 \to \{0,1\}$ *para las que existe una permutación* $\sigma : \{0,1,2,3,4\} \to \{0,1,2,3,4\}$ *tal que*

$$f_1(x_0, x_1, x_2, x_3, x_4) = f_2(x_{\sigma(0)}, x_{\sigma(1)}, x_{\sigma(2)}, x_{\sigma(3)}, x_{\sigma(4)}) \text{ para todo } (x_0, x_1, x_2, x_3, x_4) \in \{0,1\}^5$$

son iguales al permutar entradas.

Proposición 4.1.3. *[**Mismo tamaño mínimo entre funciones iguales al permutar entradas**] Dadas dos funciones booleanas* $f_1, f_2 : \{0,1\}^5 \to \{0,1\}$ *iguales al permutar entradas, se cumple que si* C_1 *es un circuito mínimo que computa* f_1 *y* C_2 *es un circuito mínimo que computa* f_2*, entonces* $|C_1| = |C_2|$.

Demostración. Cambiando la etiqueta e_i con $i \in \{0, ..., 4\}$ de cada vértice fuente de C_1 por la etiqueta $e_{\sigma(i)}$ se obtiene un circuito C_1' que compute f_2 (puesto que $f_1(x_0, x_1, x_2, x_3, x_4) = f_2(x_{\sigma(0)}, x_{\sigma(1)}, x_{\sigma(2)}, x_{\sigma(3)}, x_{\sigma(4)})$ para cada $(x_0, x_1, x_2, x_3, x_4) \in \{0,1\}^5$). Esto supone que $|C_2| \leq |C_1'| = |C_1|$ pues, por hipótesis, C_2 es un circuito de tamaño mínimo que computa f_2.

Cambiando de forma análoga la etiqueta $\sigma(i)$ con $i \in \{0, ..., 4\}$ de cada vértice fuente de C_2 por i obtenemos un circuito C_2' que computa f_1 (notemos que por ser σ una biyección existe exactamente una preimagen i para cada $\sigma(i)$ y cualquier etiqueta de C_2 es de la forma $\sigma(i)$ para algún i). Como C_1 es un circuito de tamaño mínimo que computa f_2, se sigue que $|C_1| \leq |C_2'| = |C_2|$.

□

Debido a la proposición anterior, cabe pensar que los factores que influyan en el tamaño mínimo de circuito para computar una función deben ser independientes del orden de sus entradas.

Lo que subyace en una permutación de las entradas es simplemente un cambio de referencia para mirar a las funciones desde otra perspectiva, lo que no altera la naturaleza de las mismas. Por poner un símil matemático, es como un caso particular de un cambio de base en un espacio vectorial (solamente el caso de permutar los elementos de la base), que altera el aspecto de los elementos al representarlos aunque estos sigan siendo los mismos elementos.

En la tabla 4.1 podemos ver un ejemplo muy básico con funciones que directamente devuelven alguna de sus entradas pero que son diferentes entre sí. Como vemos, mediante un reordenamiento adecuado de la entrada podemos conseguir que todas ellas computen la misma salida (se conviertan en la misma función).

Función/Orden	$(e_4, e_3, e_2, e_1, e_0)$	$(e_3, e_2, e_1, e_0, e_4)$	$(e_2, e_1, e_0, e_4, e_3)$	$(e_1, e_0, e_4, e_3, e_2)$	$(e_0, e_4, e_3, e_2, e_1)$
$f(x, y, z, t, w) = w$	e_0	e_4	e_3	e_2	e_1
$f(x, y, z, t, w) = t$	e_1	e_0	e_4	e_3	e_2
$f(x, y, z, t, w) = z$	e_2	e_1	e_0	e_4	e_3
$f(x, y, z, t, w) = y$	e_3	e_2	e_1	e_0	e_4
$f(x, y, z, t, w) = x$	e_4	e_3	e_2	e_1	e_0

Tabla 4.1: Funciones iguales entre sí con alguna permutación de la entrada

Resultaría deseable para nuestra labor acabar con esas diferencias. Afortunadamente, al menos desde el punto de vista teórico, podemos hacerlo con herramientas matemáticas como el conjunto cociente. La igualdad entre funciones al permutar entradas es una relación de equivalencia que induce una partición sobre las funciones.

Proposición 4.1.4. ***[La relación de igualdad entre funciones es de equivalencia]*** *La relación $\mathcal{R}$ sobre $Fun(\{0,1\}^5, \{0,1\})$ dada por*

$$f_1 \mathcal{R} f_2 \iff f_1 \text{ y } f_2 \text{ son iguales al permutar entradas}$$

es una relación de equivalencia que induce una partición en clases. Diremos que dos funciones f_1 y f_2 tales que $f_1 \mathcal{R} f_2$ son equivalentes bajo permutación de las entradas.

Demostración. La propiedad reflexiva es inmediata con la permutación $\sigma = id$. La propiedad simétrica se comprueba fácilmente con la permutación inversa. La propiedad transitiva se sigue de considerar la permutación composición. □

Notemos que la clase de equivalencia $[f]$ de cierta función $f \in Fun(\{0,1\}^5, \{0,1\})$ puede contener hasta $5! = 120$ funciones diferentes (tantas como permutaciones de 5 elementos hay). Pero, ¿cómo podemos detectarlas o incluso generarlas con nuestra representación? Al fin y al cabo nosotros fijamos una referencia al representar funciones en la definición 2.1.14, pues establecimos cierta significación en los bits de entrada (e_0 el menos significativo, e_1 el segundo menos significativo, etc.). Esta significación se correspondía con el orden de las entradas (al i-ésimo empezando por la izquierda se le asignó significación i-ésima) Es precisamente permutando esa significación como pasamos de un orden de entrada a otro en nuestras representaciones.

Definición 4.1.5. *[**Orden de significación**] Un orden de significación es una tupla ordenada* $(e_{\sigma(0)}, e_{\sigma(1)}, e_{\sigma(2)}, e_{\sigma(3)}, e_{\sigma(4)})$ *donde* $\sigma : \{0,1,2,3,4\} \to \{0,1,2,3,4\}$ *es una permutación.*

Para un orden tal, la significación aumenta de izquierda a derecha según el orden en la tupla, es decir, $e_{\sigma(0)}$ *es el bit de entrada menos significativo,* $e_{\sigma(1)}$ *es el segundo bit menos significativo, etc.*

Ejemplo 4.1.6. [**Representación para distintos órdenes de significación**]

Las representaciones explicadas en el ejemplo 2.1.15 se adaptan correspondientemente a otros órdenes de significación. La función utilizada en aquel ejemplo tenía representación como cadena `1000100010001000 100010001000100` para el orden $(e_0, e_1, e_2, e_3, e_4)$, mientras con orden de significación $(e_4, e_1, e_2, e_3, e_0)$ se representaría como `11001100110011000000000000000000`.

N	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
e_4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e_3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
e_2	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
e_1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
e_0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
f	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0

N	31	15	29	13	27	11	25	9	23	7	21	5	19	3	17	1	30	14	28	12	26	10	24	8	22	6	20	4	18	2	16	0
e_0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e_3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
e_2	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
e_1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
e_4	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
f'	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabla 4.2: Representación con dos órdenes de significación distintos (la etiqueta N ayuda a seguir la imagen de cada entrada al cambiar el orden de significación)

Notemos que se ha producido la correspondiente alteración en el orden de la entrada al cambiar la significación para nuestra representación. Se tiene que:

$$f(x_0, x_1, x_2, x_3, x_4) = f'(x_4, x_1, x_2, x_3, x_0) \text{ para todo } (x_0, x_1, x_2, x_3, x_4) \in \{0,1\}^5$$

▲

Como ya hemos comentado, los factores que nosotros anhelamos descubrir deberían ser independientes del orden de las entradas y, por tanto, las métricas con las que tratemos de medirlos deberían ir en la misma linea. Estas métricas trabajarán sobre nuestra representación de funciones e, idealmente, devolverán la misma puntuación para representaciones de funciones que sean equivalentes al permutar la entrada.

Definición 4.1.7. *[**Métrica invariante bajo permutación en las entradas**] Una métrica es una función* $M : Fun(\{0,1\}^5, \{0,1\}) \to \mathbb{R}^+ \cup \{0\}$*. Decimos que una métrica es invariante bajo permutación en las entradas si*

$$M(f_1) = M(f_2) \text{ para todo } f_1, f_2 \text{ tal que } f_1 \mathcal{R} f_2$$

según la relación de equivalencia $\mathcal{R}$ *definida y probada en 4.1.4.*

Finalmente, antes de empezar con las conjeturas, se amplió *Data* añadiendo todas las funciones que resultasen de permutar la entrada de alguna que ya estuviera y empaquetándolas en sus respectivas clases de equivalencia. Notemos que debido a la proposición 4.1.3 es acertado incluir esas funciones en *Data*, ya que tienen el mismo tamaño mínimo que alguna que ya está. Aunque para tamaños mínimos donde ya tengamos todas las funciones asociadas es inútil este trabajo, para tamaños superiores a 6 es posible que falten algunas de estas funciones ya que la última etapa se detiene sin acabar.

Para ello, se consideraron todas las permutaciones posibles de la significación para todas las representaciones de funciones de *Data* (como en el ejemplo 4.1.6, pero de todas las formas posibles para todas las funciones).

El nuevo dataset ampliado, que puede encontrarse en [9], pasó de 2.050.334 funciones a 2.395.113, con un total de 25.210 clases de equivalencia.

Habría sido también muy interesante tratar de explotar este cómputo de funciones equivalentes en el propio generador de circuitos. Sin embargo, suponía cambios sustanciales que habrían requerido un tiempo que no parecía apropiado invertir. La cantidad y calidad de funciones de *Data* resultaba prometedora y no parecía necesario abordar esta cuestión. Sin embargo tenemos ideas de cómo aprovecharla, de cara a continuaciones del trabajo, evitando visitar muchos más circuitos en el recorrido sistemático. En principio, sin demasiado coste adicional por circuito, se podría visitar un solo representante canónico de cada clase.

4.2. Conjetura de repetitividad

A continuación vamos a explicar la primera conjetura que obtuvimos, que está relacionada con la repetición de patrones. Explicaremos cómo la descubrimos, en qué consiste, su fundamento teórico, la métrica que usamos para cotejarla y los resultados obtenidos.

4.2.1. Deducción de la conjetura

Comenzamos nuestro estudio con una inspección visual de los datos con la esperanza de que nos llamase la atención algo sobre ellos. Como no teníamos muy claro por dónde empezar a mirar, decidimos ordenar las funciones según el número de veces que fueron computadas por el generador. Parece razonable pensar que aquellas funciones que se repiten más en la generación resultan algo más representativas que las demás en cuanto a poder computarse con circuitos "pequeños".

Tras una rápida modificación del generador, que permitió obtener el dato de cuántas veces se computaba cada función (puede encontrarse ese dato en la primera celda del notebook [16]), se pudieron observar dos cosas muy interesantes.

La primera es una consecuencia directa de lo que ya sabíamos sobre invarianza bajo permutación en las entradas. Resultó muy tranquilizador comprobar que las funciones que más veces fueron generadas por nuestro algoritmo (las que devuelven directamente alguna de sus entradas), son equivalentes bajo permutación de las entradas según la relación definida en 4.1.4. Además, el número de veces que fueron computadas es muy similar en todas. Si seguimos bajando en el ranking de funciones más computadas, vamos encontrando más grupos de funciones consecutivas que también son equivalentes entre sí.

Posición	Función	Representación como cadena	Número de veces computada
1	$f(e_0, e_1, e_2, e_3, e_4) = e_3$	11111111000000001111111100000000	2.765.442
2	$f(e_0, e_1, e_2, e_3, e_4) = e_4$	11111111111111110000000000000000	2.689.237
3	$f(e_0, e_1, e_2, e_3, e_4) = e_2$	11110000111100001111000011110000	2.678.444
4	$f(e_0, e_1, e_2, e_3, e_4) = e_1$	11001100110011001100110011001100	2.662.373
5	$f(e_0, e_1, e_2, e_3, e_4) = e_0$	10101010101010101010101010101010	2.655.816
6	$f(e_0, e_1, e_2, e_3, e_4) = 1$	11111111111111111111111111111111	1.650.463
7	$f(e_0, e_1, e_2, e_3, e_4) = e_0$ NAND e_2	01011111010111110101111101011111	514.359
8	$f(e_0, e_1, e_2, e_3, e_4) = e_1$ NAND e_4	00110011001100111111111111111111	513.141
9	$f(e_0, e_1, e_2, e_3, e_4) = e_0$ NAND e_3	01010101111111110101010111111111	511.623
10	$f(e_0, e_1, e_2, e_3, e_4) = e_0$ NAND e_4	01010101010101011111111111111111	509.413
11	$f(e_0, e_1, e_2, e_3, e_4) = e_1$ NAND e_2	00111111001111110011111100111111	507.305
12	$f(e_0, e_1, e_2, e_3, e_4) = e_0$ NAND e_1	01110111011101110111011101110111	505.512
13	$f(e_0, e_1, e_2, e_3, e_4) = e_2$ NAND e_3	00110011111111110011001111111111	502.883
14	$f(e_0, e_1, e_2, e_3, e_4) = e_3$ NAND e_4	00000000111111111111111111111111	502.647
15	$f(e_0, e_1, e_2, e_3, e_4) = e_2$ NAND e_4	00001111000011111111111111111111	502.160

Tabla 4.3: Primeras posiciones del ranking de funciones más computadas. Señaladas con el mismo color aquellas que pertenecen a la misma clase de equivalencia bajo permutación de las entradas.

Nuestra ordenación reflejaba, de alguna forma, las distintas clases de equivalencia que dedujimos en la sección previa. Esto tenía mucho sentido pues, como ya comentamos, las funciones de una misma clase deberían tratarse igual desde el punto de vista de los circuitos.

La segunda cosa que observamos, más interesante si cabe, es que las funciones de *Data* parecían propensas a mostrar patrones de longitud potencia de 2 repetidos en sus representaciones como cadenas, de forma disjunta. Esta repetitividad tan particular parecía, a simple vista, superior a la que debiera existir en un subconjunto de cadenas aleatorias de *Comp*. Además, era más palpable a medida que subíamos en el ranking de funciones más computadas de *Data* que, en general, coincidían con las de menor tamaño de circuito mínimo.

Definición 4.2.1. *[**Patrón de repetición de longitud potencia de 2**] Dada una cadena* $c = (c_0, ..., c_{31}) \in \{0,1\}^{32}$*, decimos que* $(c_j, ..., c_{j+k})$ *con* $j, j+k \in \{0, ..., 31\}$ *y* $k = 2^m$ *para* $m \in \mathbb{N} \cup \{0\}$[2] *es un patrón de repetición de longitud potencia de 2 disjunto en* c *si existe* i *tal que* $i \in \{0, ..., j-1\} \cup \{j+k+1, ..., 31\}$ *de forma que* $(c_j, ..., c_{j+k}) = (c_i, ..., c_{i+k})$*.*

Conjetura 4.2.2. *[**Repetitividad de patrones de longitud potencia de 2 disjuntos**]*
Las funciones que requieren circuitos "pequeños" tienden a tener más repeticiones de longitud potencia de 2 en sus representaciones como cadena que las funciones que requieren circuitos "grandes".

Además, observamos tendencia a poder anidar estos patrones, expresando algunos de ellos a su vez como repetición de patrones. La tabla 4.4 muestra cómo las 15 primeras funciones más computadas (las de la tabla 4.3) pueden expresarse íntegramente mediante patrones de repetición, muchos de los cuales son anidados.

Posición	Representación como cadena	Expresión con patrones de repetición
1	11111111000000001111111100000000	$(1^8 0^8)^2$
2	11111111111111110000000000000000	$1^{16}0^{16}$
3	11110000111100001111000011110000	$(1^4 0^4)^4$
4	11001100110011001100110011001100	$(1^2 0^2)^8$
5	10101010101010101010101010101010	$(10)^{16}$
6	11111111111111111111111111111111	1^{32}
7	01011111010111110101111101011111	$((01)^2 1^4)^4$
8	00110011001100111111111111111111	$(0^2 1^2)^4 1^{16}$
9	01010101111111110101010111111111	$((01)^4 1^8)^2$
10	01010101010101011111111111111111	$(01)^8 1^{16}$
11	00111111001111110011111100111111	$(0^2 1^6)^4$
12	01110111011101110111011101110111	$(01^3)^8$
13	00110011111111110011001111111111	$((0^2 1^2)^2 1^4)^2$
14	00000000111111111111111111111111	$0^8 1^{24}$
15	00001111000011111111111111111111	$(0^4 1^4)^2 1^{16}$

Tabla 4.4: Muestra de las funciones más computadas como expresión de patrones de repetición y del anidamiento de los mismos.

Las expresiones de la tabla 4.4 deben tomarse simplemente como ejemplo de la enorme repetitividad de estas funciones del ranking y del anidamiento de sus patrones. Hay otras expresiones como repetición de patrones para estas funciones y no hay un criterio claro de cuáles son mejores o peores. De hecho, por motivos computacionales, este tipo de expresiones no tendrán nada que ver con la métrica desarrollada y, por los motivos que explicaremos, ni siquiera serán considerados los patrones de longitudes 1 y 2. Sin embargo, es interesante saber lo que sucede para las funciones de *Data*, aunque no seamos capaces de hilar tan fino al medir.

4.2.2. Discusión teórica de la repetitividad

Las expresiones de las funciones de la tabla 4.4 parecen especialmente repetitivas porque muestran muchos patrones consecutivos (periodos). Pero la consecutividad de estos patrones se intuye irrelevante para nosotros, debido a que se destruye (y a veces se construye) con una simple permutación de las entradas.

[2]Se permite que $k = 2^0 = 1$, aunque un patrón de longitud 1 ('0' o '1') se repite con total seguridad y resulta un poco absurdo considerar como repetición a uno de los dos únicos valores que puede tomar una componente de la cadena. Algo similar sucede con $k = 2^1 = 2$ (hay necesariamente una repetición de esa longitud), pero no con $k = 2^2 = 4$. Veremos esto con más detalle después.

El orden de la representación como cadena de una función, que es donde se manifiesta la repetitividad, está sujeto a la referencia (orden de significación) que utilicemos para mirarla. Como ya comentamos en la sección 4.1.1, considerando órdenes distintos de significación (que equivalen a permutaciones en las entradas) cambia también la representación de la función pero, a efectos de tamaño mínimo de circuito, todo permanece igual. Este cambio en las representaciones, en cierto modo, desplaza o incluso trocea los patrones repetidos, pudiendo hacer que repeticiones consecutivas dejen de serlo, que patrones repetidos que no eran consecutivos pasen a serlo o incluso que se ganen o pierdan repeticiones.

Es decir, que la repetitividad depende de la referencia con la que miremos a la función, ya que se modifica al cambiar de referencia. Por tanto, nuestra conjetura no es invariante, de forma natural, ante permutaciones de las entradas. Aunque esto pueda encender todas las alarmas, no significa que haya que descartarla porque esté mal. Simplemente no es un factor que influya directamente en el tamaño de circuito sino, muy probablemente, la manifestación de un factor tal. Tenemos la impresión de que hay un factor oculto que no depende de referencias y provoca la repetitividad, y esto último lo percibimos distorsionado de una u otra forma en función de la referencia escogida.

Nuestra teoría es que la repetitividad es algo así como el fenotipo de cierto factor influyente directamente en el tamaño mínimo de circuito. Por poner una analogía sencilla, es como si el factor desconocido fuese una determinada secuencia de genes que influyen para tener una nariz pequeña y favorecen el tener color de ojos azul. La repetitividad sería el color de ojos azul que nosotros percibimos y nos llama la atención, mientras que el tamaño de circuito mínimo sería el tamaño de la nariz, que se ve afectado directamente por estos genes. A nosotros nos llama la atención lo que se ve, que es lo que hemos descubierto, aunque quizás sería más interesante lo que está oculto.

Aunque la repetitividad no sea causa sino consecuencia, es igualmente útil e interesante. Para empezar, porque supone un hilo del que tirar para encontrar el factor oculto que influye directamente en el tamaño de circuito. Pero también porque puede permitirnos distinguir con altas probabilidades los problemas que se pueden computar con circuitos "pequeños" y los que no. Continuando con la analogía anterior, si la influencia de los genes es fuerte y sabemos que una persona tiene los ojos azules, entones afirmando que tiene una nariz pequeña es bastante probable que acertemos. Es decir, que puede existir una correlación implícita entre las consecuencias de la que nos podemos aprovechar.

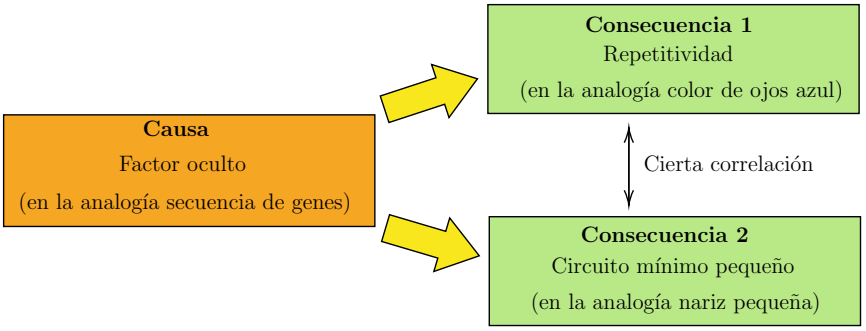


Figura 4.2.1: Diagrama que refleja la situación que intuimos para la conjetura de repetitividad.

Pero, ¿cómo de fuerte es la relación entre las consecuencias para poder predecir una en base a la otra? ¿Qué probabilidad hay de acertar? Eso es lo que vamos a tratar de intuir experimentalmente, utilizando la repetitividad para distinguir entre las funciones con tamaño de circuito mínimo "pequeño" o "grande".

Antes de pasar a ello, vale la pena dar una intuición de por qué puede haber tanta repetitividad de longitud potencia de 2 en las funciones computables con los circuitos más "pequeños". Los circuitos de tamaño 0 (los que devuelven directamente una de las entradas) computan funciones tremendamente repetitivas (las 5 primeras de la tabla 4.4) y las operaciones lógicas que, aplicadas sucesivamente sobre esas funciones nos proporcionan todas las demás, reproducen, en cierta forma, los patrones repetidos también en longitudes potencia de 2, aunque se van embarullando y perdiendo poco a poco.

La idea es que empezamos con repetitividad máxima y según vamos aplicando operaciones lógicas la vamos

perdiendo poco a poco. Cada una de estas operaciones equivale, en términos de circuito, a colocar una puerta sumidero combinando dos circuitos lo que incrementa su tamaño. Por tanto, a más puertas en los circuitos menos repetitividad queda.

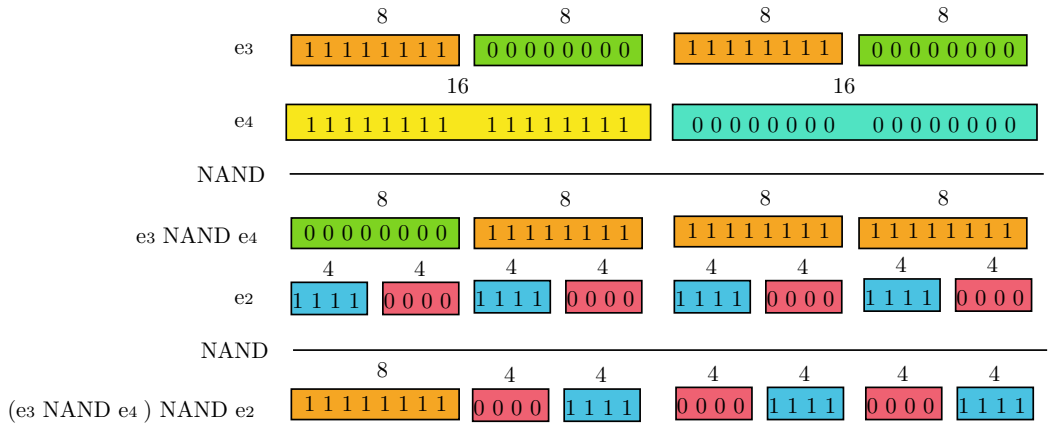


Figura 4.2.2: Diagrama que refleja la situación que intuimos para la conjetura de repetitividad.

A todo esto se le puede dar incluso una interpretación físico-matemática más próxima al Análisis de Fourier que está bien tener en mente, porque quizás más adelante podría revelarnos algo. Las funciones booleanas pueden entenderse como la representación de ondas cuadradas que pueden tomar dos valores (podemos establecer que el valor '1' significa "alto" en la onda y el valor '0' significa "bajo"). Las funciones computadas por los circuitos de tamaño 0 representan las distintas ondas periódicas para todos los posibles periodos potencia de 2 y una operación entre funciones consiste en una superposición de ondas donde análogamente se va reflejando la pérdida paulatina de periodicidad y patrones repetitivos cuanta más superposición se hace.

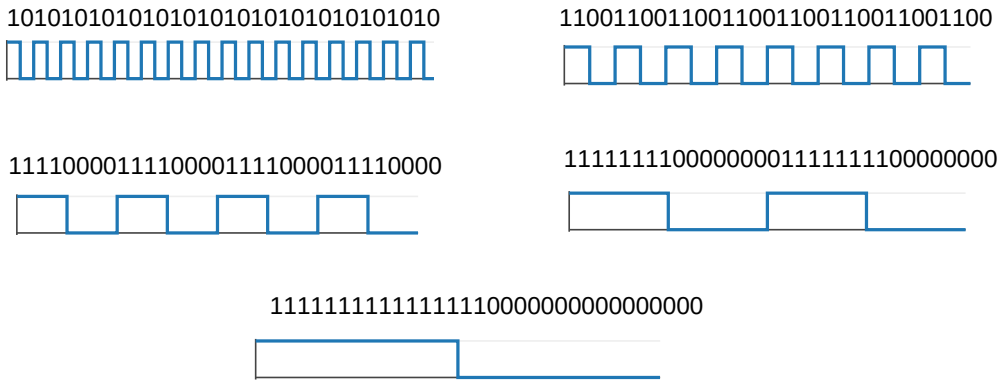


Figura 4.2.3: Ondas cuadradas periódicas asociadas a cada una de las funciones computables con tamaño 0.

4.2.3. Métrica de repetitividad

Pasamos ya a tratar de evaluar la repetitividad como elemento diferenciador entre funciones computables con un circuitos "pequeños" y funciones que requieren circuitos "grandes". Para ello, vamos a desarrollar una métrica cuyas puntuaciones nos permitan comparar el comportamiento de nuestra conjetura sobre *Data* y *Comp*. (Las pruebas y resultados que vamos a detallar en esta sección pueden encontrarse en [16]).

Lo que pretendíamos en un principio, que sería lo ideal, era obtener expresiones de repetitividad como las de la

tabla 4.3, tratando después de asignarles un valor numérico que midiese su calidad (por ejemplo, pensamos en contar la cantidad de símbolos '0' y '1' utilizados en la expresión ponderando de algún modo el anidamiento). Sin embargo, no encontramos una manera razonablemente eficiente de obtener buenas expresiones (nuestros intentos requerían demasiadas comparaciones y eran muy lentos para pequeños subconjuntos de *Data*). Para obtener expresiones que fuesen malas era mejor darle otro enfoque al asunto.

Aunque las funciones que más veces fueron computadas podían cubrirse íntegramente mediante patrones de repetición disjuntos como los definidos en 4.2.1, excluyendo los de tamaños $k = 1$ y $k = 2$, parecía no ser muy habitual en las funciones de *Data* y se intuía que mucho menos en *Comp*. Surgió la idea de medir la longitud de la cadena que se podía cubrir mediante este tipo de patrones de repetición disjuntos. Como estos patrones solían comenzar en posiciones de la cadena que fuesen suma de potencias de 2, nos limitaríamos a mirarlos desde esas posiciones y sumar sus longitudes. En cierto modo íbamos a buscar la mejor manera de partir la cadena en patrones potencia de 2 para sumar la máxima longitud con patrones periódicos, obteniendo así una puntuación que reflejase la repetitividad de la función.

Definición 4.2.3. *[**Longitud cubierta mediante repetición de patrones**] Dada una función $f : \{0,1\}^5 \to \{0,1\}$ y denotando por $c = (c_0, ..., c_{31}) = Rep(f)$ su representación como cadena, definimos su longitud cubierta mediante repeticiones como*

$$L_{Rep}(f) = \max_{p \in P}\{S_p\} \in \mathbb{N} \cup \{0\}$$

donde

- $P = \{(i_1, ..., i_n) \mid i_1 = 0 < i_2 < ... < i_n = 32$ *con* $(i_{j+1} - i_j) = 2^m \ \forall j \in \{1, ..., n\}$ *y* $m \geq 2$ $\}$ *es el conjunto de posibles particiones en potencias de 2 (expresadas como tuplas de índices).*
- $S_p = \sum_{l \in L_p} l$ *es la suma de las longitudes de las repeticiones para cierta partición p.*
- $L_{(i_1,...,i_n)} = \{(i_{j+1} - i_j) \mid (c_{i_j}, ..., c_{i_{j+1}-1})$[3] *es una repetición en c*$\}$ *es el conjunto de longitudes de patrones repetidos en c para un partición concreta.*

La longitud cubierta mediante repeticiones puede calcularse exclusivamente atendiendo a la partición cuyos patrones tienen todos longitud 4. Esta partición es la más fina posible y capta, al menos, la misma longitud con repeticiones que cualquier otra partición.

Proposición 4.2.4. *[**Simplificación a la partición más fina**] Dada una función $f : \{0,1\}^5 \to \{0,1\}$ y denotando por $c = (c_0, ..., c_{31}) = Rep(f)$ se tiene que*

$$L_{Rep}(f) = S_{(0,4,...,32)}$$

Demostración. Por la definición 4.2.3 de $L_{Rep}(f)$ como máximo de todos los S_p se tiene que $S_{(0,4,...,32)} \leq L_{Rep}(f)$.

Por otra parte, dada una partición $(i_1, ..., i_n)$ sea $r = (c_{i_j}, ..., c_{i_{j+1}-1})$ una repetición. Puesto que $(i_{k+1} - i_k) = 2^m \geq 2^2 = 4$ para todo $k \in \{1, ..., n\}$, se tiene que i_j es múltiplo de 4 ya que es una suma de múltiplos de 4 ($i_j = \sum_{k<j}(i_{k+1} - i_k)$) y, por tanto, es un índice de la tupla $(0, 4, ..., 32)$.

Como el patrón r además se puede subdividir en $2^m/2^2 = 2^{m-2}$ patrones de la forma $(c_{i_j}, ..., c_{i_{j+4h}})$ con $h \in \{1, ..., 2^{m-2}\}$ y r es una repetición, cada uno de estos patrones también lo es y sus longitudes se contabilizan en $S_{(0,4,...,32)}$.

Se deduce que $S_p \leq S_{(0,4,...,32)}$ para cualquier $p \in P$ y, por tanto, $L_{Rep}(f) = \max_{p \in P}\{S_p\} \leq S_{(0,4,...,32)}$. □

Nótese también que en la definición 4.2.3 se descartó la posibilidad de patrones de longitud $2^0 = 1$ pues, con total seguridad, alguno de ellos se repite y, de hecho, permite cubrir la cadena casi en su totalidad mediante repeticiones. El hecho de que un '0' o un '1' aparezca repetido en una cadena es consecuencia de que apenas

[3] Nótese que dos índices consecutivos de una partición nos marcan cierta subcadena, cerrada por la izquierda y abierta por la derecha. Por eso, su longitud es simplemente la resta de estos índices, habiendo uno más que intervalos en la partición.

hay 2 símbolos que utilizar 32 veces. Por tanto, no parece que debiera tenerse en cuenta como un verdadero patrón repetido. También se descartó la posibilidad de longitud $2^1 = 2$ por un motivo muy similar. Démosle formalidad a todo esto y mostremos cómo cambia el panorama al pasar a longitud $2^2 = 4$ donde muchas funciones ni siquiera tienen una repetición.

Proposición 4.2.5. *[**Longitud de los patrones de repetición**]*

1. *Si en el conjunto P de la definición 4.2.3 considerásemos $m \geq 0$, para toda $f : \{0,1\}^5 \to \{0,1\}$ se tendría que $31 \leq L_{Rep}(f) \leq 32$.*
2. *Si en el conjunto P de la definición 4.2.3 considerásemos $m \geq 1$, para toda $f : \{0,1\}^5 \to \{0,1\}$ se tendría que $26 \leq L_{Rep}(f) \leq 32$.*
3. *Con el conjunto P de la definición 4.2.3 (que considera $m \geq 2$) existen $16!/8!$ funciones diferentes $f : \{0,1\}^5 \to \{0,1\}$ tales que $L_{Rep}(f) = 0$.*

Demostración. En cualquier caso $L_{Rep}(f) \leq 32$ porque $S_{(i_1,\dots,i_n)} \leq \sum_{j=1}^{n-1}(i_{j+1} - i_j) = 32$ para toda $(i_1, ..., i_n) \in P$. Probemos ahora los 3 apartados.

1. En caso de considerar $m \geq 0$, para la partición $p = (i_1 = 0, i_2 = 1, ..., i_{33} = 32) \in P$ se cumple que (c_{i_j}) es una repetición para todo $j \in \{1, ..., 32\}$ excepto para, a lo sumo, uno. Suponiendo que existen j y j' con $j \neq j'$ tales que (c_{i_j}) y $(c_{i_{j'}})$ no son patrones de repetición, se tiene que $(c_{i_j}) = (0)$ y $(c_{i_{j'}}) = (1)$ o $(c_{i_j}) = (1)$ y $(c_{i_{j'}}) = (0)$ (de lo contrario coincidirían y sí serían patrones de repetición). Tomando j'' tal que $j'' \neq j$ y $j'' \neq j'$ se tiene que $(c_{i_{j''}}) = (0)$ o $(c_{i_{j''}}) = (1)$ y, por tanto, $(c_{i_{j''}}) = (c_{i_j})$ o $(c_{i_{j''}}) = (c_{i_{j'}})$, lo que contradice que ni (c_{i_j}) ni $(c_{i_{j'}})$ sean patrones de repetición. Se deduce que $L_{Rep}(f) \geq S_p \geq \left(\sum_{j=1}^{32} 1\right) - 1 = 32 - 1 = 31$.
2. En caso de considerar $m \geq 1$ sirve el mismo argumento pero con la partición $p = (i_1 = 0, i_2 = 2, ..., i_{17} = 32) \in P$. Al haber solo 4 posibles patrones $(0,0), (0,1), (1,0)$ y $(1,1)$, todos los (c_{i_j}, c_{i_j+1}) serán repeticiones salvo, a lo sumo, 3. Por tanto, como cada uno aporta longitud 2 al sumatorio, $L_{Rep}(f) \geq S_p \geq \left(\sum_{j=1}^{16} 2\right) - 3 \cdot 2 = 32 - 6 = 26$.
3. Debido a la proposición 4.2.4, podemos fijar la partición más fina $p = (0, 4, ..., 32)$. Tendremos tantas funciones f con $L_{Rep}(f) = 0$ como cadenas de tamaño 32 puedan formarse sin repetición de patrones para esta partición. Para ello, basta asegurar que los 8 patrones de longitud 4 $(c_{4h}, ..., c_{4h+4})$ con $h \in \{0, ..., 8\}$ sean diferentes dos a dos. Con los símbolos $\{0,1\}$ se pueden formar $2^4 = 16$ patrones de longitud 4 y tenemos $\binom{16}{8}$ formas diferentes de elegir 8. Para cada elección de 8 patrones existen 8! posibles permutaciones que proporcionan funciones distintas. Por tanto, hay $\binom{16}{8} \cdot 8! = (16! \cdot 8!)/(8! \cdot 8!) = 16!/8!$ funciones distintas sin repetición de patrones.

□

Justificado ya el porqué de la restricción $k \geq 2$ en la definición de repetición, estamos en condiciones de otorgar a la longitud cubierta mediante repeticiones el papel de métrica de repetititividad.

4.2.4. Resultados de la métrica de repetitividad

Para probar nuestras métricas sobre los datos escogimos funciones aleatorias de distintas clases. Como la cantidad de funciones en cada clase varía entre 1 y $5! = 120$, dar la misma posibilidad a todas las funciones para ser escogidas de manera aleatoria significaría sobreestimar a las clases que tengan más funciones (las que tengan cerca de 120).

Para muestras aleatorias de *Comp* habría resultado muy costosa la comprobación de que todas las funciones fuesen de clases distintas (podríamos haber utilizado siempre la misma muestra y haberlo calculado para ella, pero resultaba más fiable escoger funciones de *Comp* nuevas cada vez y en ese caso la comprobación ralentiza bastante cada prueba). Teniendo en cuenta que se escogen apenas unos pocos miles de funciones para las pruebas de entre las casi $2^{32} - 2.395.113$ funciones que hay en *Comp* (más de 4 mil millones) y que como mucho hay 120 en cada clase, la probabilidad de que muchas de ellas caigan en la misma clase es muy baja.

En las figuras 4.2.4 y 4.2.5 puede verse una representación de las puntuaciones de L_{Rep} para unos pocos miles de muestras aleatorias de ambas clases. Y en la figura 4.2.6 una representación simultánea de las distribuciones de puntuación. Se obtuvieron tendencias de puntuaciones muy diferentes que respondían más o menos a lo esperado. Las puntuaciones sobre *Data* eran frecuentemente altas, mientras que las puntuaciones sobre *Comp* tendían a ser bajas.

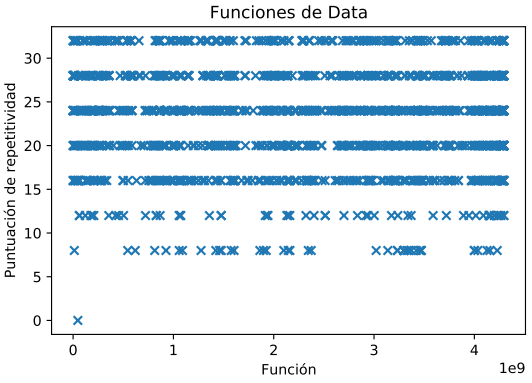


Figura 4.2.4: L_{Rep} sobre una muestra de *Data*.

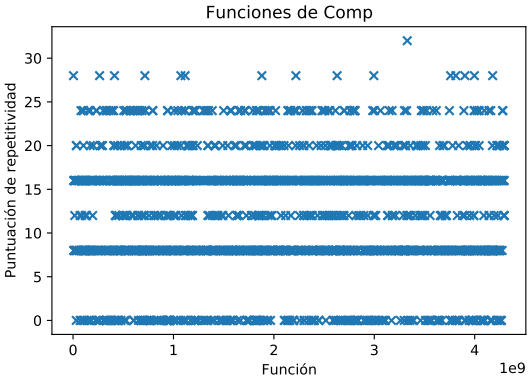


Figura 4.2.5: L_{Rep} sobre una muestra de *Comp*.

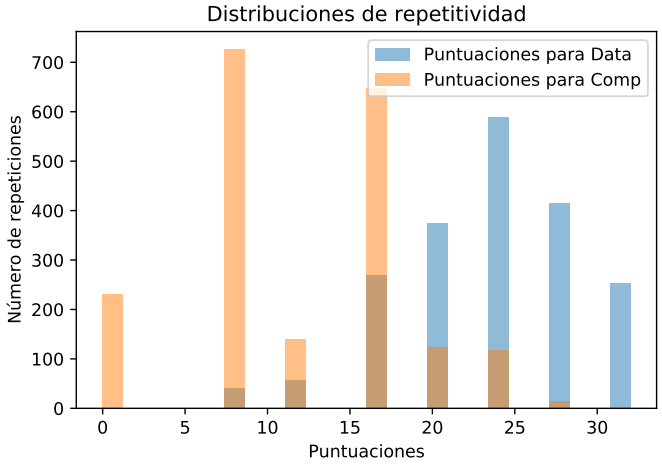


Figura 4.2.6: Distribuciones de puntuación de repetitividad para muestras aleatorias de *Data* y *Comp*.

Un aspecto bastante positivo de las distribuciones que se observan en 4.2.6 es la escasez (casi ausencia) de casos claramente contradictorios. La distribución de *Data* no llega hasta el extremo más bajo de puntuaciones donde están buena parte de los casos de *Comp*, y la distribución de *Comp* tampoco llega a puntuación máxima donde están buena parte de los casos de *Data*. Parece deducirse que una repetitividad extrema en uno de los dos sentidos es síntoma claro de un circuito mínimo "pequeño" o "grande".

Adicionalmente, sometimos los datos al siguiente test clasificatorio sencillo.

1. Generamos un subconjunto aleatorio $Data_{train} \subset Data$ y un subconjunto aleatorio $Comp_{train} \subset Comp$ con $|Data_{train}| = |Comp_{train}| = 20.000$ y $[f] \neq [f'] \; \forall f, f' \in Data_{train}$ tales que $f \neq f'$.
2. Calculamos las puntuaciones medias $Data_{mean}$ y $Comp_{mean}$ de L_{Rep} para sendos subconjuntos sobre nuestra métrica.
3. Generamos nuevos subconjuntos aleatorios $Data_{test} \subset (Data \setminus Data_{train})$ y $Comp_{test} \subset (Data \setminus Data_{test})$ de forma que $|Data_{test}| = |Comp_{test}| = 5.000$ y $[f] \neq [f'] \; \forall f, f' \in Data_{test}$ tales que $f \neq f'$.

4. Clasificamos cada $x \in Data_{test} \cup Comp_{test}$ eligiendo, según su puntuación en la métrica, el conjunto cuya media de puntuaciones resulte más próxima, es decir:

$$Predición(x) = \begin{cases} Data & \text{si } |L_{Rep}(f) - Data_{mean}| \leq |L_{Rep}(f) - Comp_{mean}| \\ Comp & \text{si } |L_{Rep}(f) - Comp_{mean}| < |L_{Rep}(f) - Data_{mean}| \end{cases}$$

5. Calculamos los porcentajes de acierto donde, como es obvio, un acierto consiste en que $x \in Predicción(x)$.

Tras varias pruebas con este test, se obtuvo en torno a un 82 % de acierto para los ejemplos de $Data_{test}$ y en torno a un 86 % para los ejemplos de $Comp_{test}$ sin que hubiese prácticamente variaciones entre ejecuciones distintas. Esto ratificó el buen funcionamiento de la métrica, reforzando la conjetura 4.3.3.

4.3. Conjetura de distinción de pares cruzados

Poco después desarrollamos otra conjetura que va más enfocada en la relación bit a bit que hay entre las entradas que devuelven salidas iguales y entre las entradas que devuelven salidas distintas. Intuimos que este tipo de relaciones debe ser la esencia de la computabilidad de una función mediante un circuito "pequeño" o "grande". A continuación vamos a desarrollar la idea de distinción de pares cruzados como fenómeno habitual de las funciones que requieren circuitos "grandes".

4.3.1. Discusión teórica de la distinción de pares cruzados

El hecho de que una función booleana devuelva salidas diferentes para entradas parecidas es intuitivamente una dificultad a considerar para su cómputo. En tal caso, será necesario fijarse en los detalles que hacen diferentes a esas entradas. Esos detalles, que básicamente consisten en los bits que toman distinto valor (bits de los focos), es lo único a lo que puede agarrarse el circuito para computar salida '0' en un caso y salida '1' en el otro.

Definición 4.3.1. *[**Distinción de pares de entradas, foco y contexto**] Dada una función $f : \{0,1\}^5 \to \{0,1\}$ y un par de entradas $x = (x_0, ..., x_4), x' = (x'_0, ..., x'_4) \in \{0,1\}^5$, decimos que f distingue el par de entradas x y x' si $f(x) \neq f(x')$.*

Sea $Dist(x, x') = \{i_1, ..., i_n\} = \{i \mid x_i \neq x'_i\}$ el conjunto de índices de los bits distintos de x y x' siendo $i_1 < i_2 < ... < i_n$. Llamamos foco de x, para la distinción anterior, a la tupla $Foco_x(x, x') = (x_{i_1}, ..., x_{i_n})$ y llamamos foco de x', para la distinción anterior, a la tupla $Foco_{x'}(x, x') = (x'_{i_1}, ..., x'_{i_n})$.

Sea $Eq(x, x') = \{j_1, ..., j_{5-n}\} = \{j \mid x_j = x'_j\}$ el conjunto de índices de los bits iguales de x y x' siendo $j_1 < j_2 < ... < j_{5-n}$. Llamamos contexto de la distinción a la tupla $Cont(x, x') = (x_{j_1}, ..., x_{j_{5-n}}) = (x'_{j_1}, ..., x'_{j_{5-n}})$.

En la figura 4.3.1 podemos ver un ejemplo de distinción de un par de entradas con focos 1 y 0 y contexto 1000[4]. Un circuito que compute esas salidas para esas entradas necesariamente se fija en los focos.

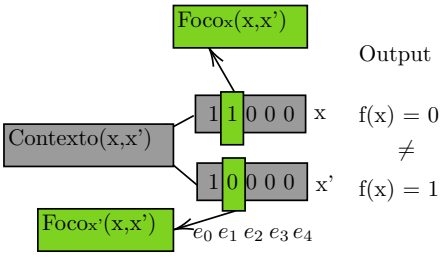


Figura 4.3.1: Distinción de pares con focos de longitud 1 y contextos de longitud 4.

Una primera idea es que, como un circuito debe tener la lógica suficiente para hacer estas distinciones, por lo general necesitará más puertas a mayor número de distinciones. Esto no supone un gran descubrimiento,

[4]De nuevo utilizamos esta notación sin comas para las cadenas, que formalmente son tuplas.

pues básicamente significa que las funciones que devuelven una cantidad equilibrada de salidas a '0' y a '1' son más difíciles y suelen requerir circuitos más grandes que las funciones que devuelven mucho más uno de los dos valores que el otro.

Sin embargo, la clave de los circuitos reside en la reutilización de algunas de sus zonas para atender a tareas muy diversas. En este sentido, la simple diferenciación de pares quedaría un poco pobre y sería necesario ir un paso más allá. Lo interesante sería descubrir qué tipo de diferenciaciones se "molestan" más entre sí y, por tanto, resultan más incompatibles a la hora de ser resueltas por la misma zona de puertas del circuito. En esa linea se nos ocurrió una diferenciación de pares que resultaría más sofisticada, la diferenciación de pares cruzados.

Supongamos una diferenciación de pares como la de la figura 4.3.1 y, adicionalmente, otra diferenciación de pares con los mismos focos, en las mismas posiciones, pero distintos contextos y las salidas justo al revés (para el foco donde con el primer par se devuelve '0' con el segundo se devuelve '1' y viceversa). En este caso no vale con fijarse en los focos para diferenciar ambas cosas porque, según los focos, para un par las salidas son de una forma y para el otro par son de forma opuesta. Un circuito que distinga esas situaciones debe fijarse también en los contextos combinándolos del algún modo con los focos, lo que ya es un motivo más razonable para presagiar la necesidad de puertas extra (sobre todo si esta situación sucede con frecuencia en la función).

Definición 4.3.2. *[**Distinción de pares cruzados**] Dada una función* $f : \{0,1\}^5 \to \{0,1\}$ *y dos pares de entradas* $(x, x') \in \{0,1\}^5 \times \{0,1\}^5$ *e* $(y, y') \in \{0,1\}^5 \times \{0,1\}^5$ *decimos que* f *los distingue como pares cruzados si*

$$Dist(x,x') = Dist(y,y'), Foco_x(x,x') = Foco_y(y,y'), Cont(x,x') = Cont(y,y'), f(x) = f(y') \neq f(x') = f(y)^5.$$

En la figura 4.3.2 podemos observar un distinción de pares cruzados donde interviene el par visto en la figura 4.3.1 con contexto 1000 y otro para distinguir con el mismo foco pero salidas cruzadas que tiene contexto 0010.

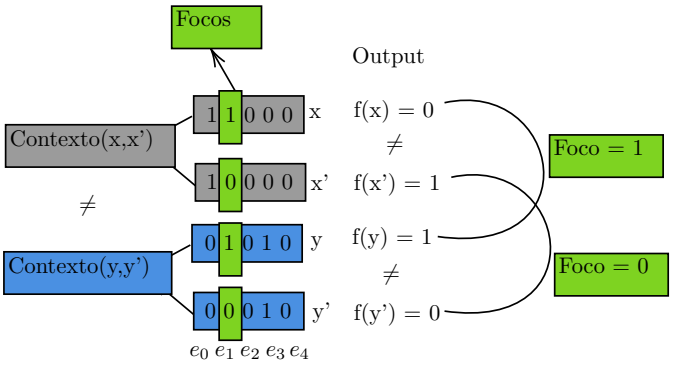


Figura 4.3.2: Ejemplo de distinción de pares cruzados.

Sobre esta idea base de distinción de pares cruzados se puede afinar un poco más tratando de ponderar de algún modo las longitudes de los focos y las partes coincidentes de los contextos. En principio, cuanto menor sea la longitud de los focos mayor es la dificultad en la distinción porque hay menos en lo que apoyarse para distinguir. De igual forma, cuanto más parecidos sean los contextos de los pares cruzados mayor es la dificultad porque solo son útiles para distinguir correctamente sus pocos bits distintos.

En la figura 4.3.3 se aprecia cómo en la distinción de pares que vimos antes solo son útiles 2 bits diferentes de los contextos (el bit e_0 y e_1 según la figura). Los demás bits toman el mismo valor en los contextos de los dos pares y resultan inútiles para esta distinción cruzada.

[5] Nótese que no hace falta pedir que $Foco_{x'}(x,x') = Foco_{y'}(y,y')$ porque viene implícito en el hecho de que $Foco_{x'}(x,x') = Foco_{y'}(y,y')$ (sus complementarios bit a bit coinciden luego ellos también). Tampoco es necesario pedir que $Eq(x,x') = Eq(y,y')$ pues, según lo exigido en la definición, sus complementarios coinciden ($Dist(x,x') = Dist(y,y')$).

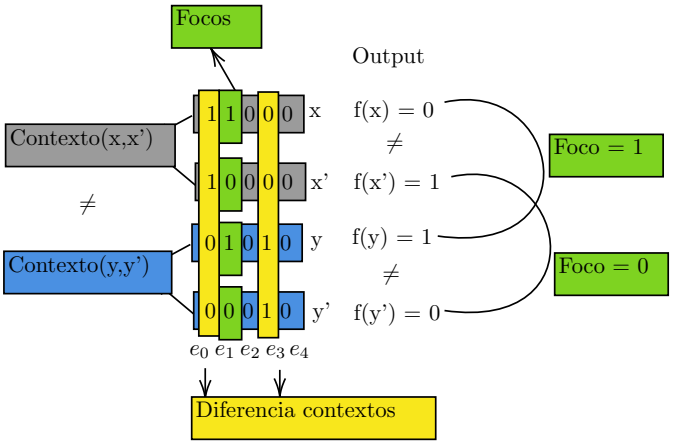


Figura 4.3.3: Ejemplo de distinción de pares cruzados con la parte diferente de los contextos en amarillo.

Con todo ello se estableció la siguiente conjetura que trataríamos de cotejar mediante un par de métricas posteriormente.

Conjetura 4.3.3. *[**Distinción de pares cruzados**] Las funciones que distinguen más pares cruzados, según lo definido en 4.3.2, suelen requerir un tamaño mínimo de circuito mayor. Además, las distinciones de pares cruzados con focos más pequeños y menos bits diferentes entre sus contextos suelen ser más influyentes para que el tamaño del circuito mínimo aumente.*

Esta conjetura, según lo que hemos venido discutiendo, se presta con facilidad a invarianza bajo permutación de las entradas, lo que resulta un indicio de que va en una buena dirección. No hay motivos para ligar este fenómeno de distinción de pares a las posiciones que ocupen los bits. Reordenando las entradas es posible que se reordenan también los focos y los contextos, pero los aspectos verdaderamente relevantes de ellos como las longitudes o la cantidad de bits coincidentes entre contextos no se ven alterados por la reordenación.

En la figura 4.3.4 puede verse la distinción de pares que mostramos con orden de significación $(e_0, e_1, e_2, e_3, e_4)$ en 4.3.3, pero cambiando la significación a $(e_1, e_2, e_4, e_3, e_0)$.

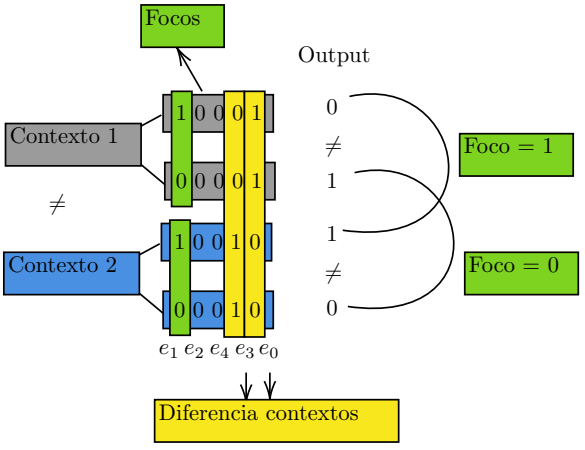


Figura 4.3.4: Distinción de pares cruzados con el orden de significación $(e_1, e_2, e_4, e_3, e_0)$.

Aunque los contextos ahora sean 0001 y 0010 (permutaciones respecto a los contextos del ejemplo que vimos un poco antes), lo relevante de ellos es la longitud 4 con apenas 2 bits diferentes, lo cual no ha cambiado al permutar entradas. Las mediciones para esta distinción resultan, de manera natural, iguales considerando un orden u otro.

4.3.2. Métricas de distinción de pares cruzados

A continuación definimos dos métricas para la distinción de pares cruzados explicada en la subsección anterior cuya implementación y resultados detallados pueden encontrarse en [15]. La primera es una versión muy simple que apenas cuenta el número de distinciones de pares cruzados de una función, mientras que la segunda es una versión mejorada que añade la ponderación por longitudes de los focos y el número de bits diferentes entre contextos. El interés por tener versiones separadas de las métricas es poder determinar si los focos pequeños y los contextos parecidos tienden a traducirse en circuitos mínimos más grandes como presuponíamos.

Métrica 4.3.4. *[**Distinción de pares sin ponderar**] Definimos como métrica de distinción de pares sin ponderar a la función* $DP : Fun(\{0,1\}^5, \{0,1\}) \to \mathbb{R}^+ \cup \{0\}$ *dada por*

$$DP(f) = \sum_{(x,x')\in(\{0,1\}^5)^2} \left(\sum_{(y,y')\in(\{0,1\}^5)^2} \frac{\mathcal{D}\big(f,(x,x'),(y,y')\big)}{2} \right)$$ [6]

donde la función indicatriz $\mathcal{D} : Fun(\{0,1\}^5, \{0,1\}) \times (\{0,1\}^5)^2 \times (\{0,1\}^5)^2 \to \{0,1\}$ *viene dada por*

$$\mathcal{D}(f,(x,x'),(y,y')) = \begin{cases} 1 & \text{si } f \text{ distingue como pares cruzados a } (x,x') \text{ e } (y,y') \text{ según 4.3.2.} \\ 0 & \text{en caso contrario.} \end{cases}$$

Métrica 4.3.5. *[**Distinción de pares ponderada**] Definimos como métrica de distinción de pares ponderada a la función* $DP_{Pond} : Fun(\{0,1\}^5, \{0,1\}) \to \mathbb{R}^+ \cup \{0\}$ *dada por*

$$DP_{Pond}(f) = \sum_{(x,x')\in(\{0,1\}^5)^2} \left(\sum_{(y,y')\in(\{0,1\}^5)^2} \frac{\mathcal{D}\big(f,(x,x'),(y,y')\big) \cdot Pond\big((x,x'),(y,y')\big)}{2} \right)$$

donde la función de ponderación $Pond : (\{0,1\}^5)^2 \times (\{0,1\}^5)^2 \to \mathbb{R}^+ \cup \{0\}$ *viene dada por*

$$Pond\big((x,x'),(y,y')\big) = \begin{cases} |Eq(x,x')| + \dfrac{|Eq\big(Cont(x,x'),Cont(y,y')\big)|}{|Cont(x,x')|}\ ^{[7]} & \text{si } |Cont(x,x')| \neq 0 \\ 0 & \text{si } |Cont(x,x')| = 0 \end{cases}$$

4.3.3. Resultados de las métricas de distinción de pares

Como las métricas diseñadas para distinción de pares cruzados son invariantes bajo permutaciones de las entradas, con más motivo debíamos aplicarlas solamente sobre un representante de cada clase de equivalencia. La puntuación para las demás funciones de la clase sería exactamente la misma que para el representante tomado.

El cálculo de la métrica sin ponderar 4.3.4 y de la métrica ponderada 4.3.5 sobre un representante de todas las clases de *Data* y el mismo número de funciones aleatorias de *Comp* proporcionó unas distribuciones de puntuación que presagiaban buenos resultados (figuras 4.3.5 y 4.3.6).

No obstante, en las distribuciones de puntuación de *Data* se apreció un fenómeno de "cola larga" que resulta poco deseable. La separación entre distribuciones parece buena (incluso algo mejor que con la métrica de repetitividad), pero los casos con mayor puntuación, que deberían corresponderse con funciones que requieren circuitos "grandes", pertenecen al conjunto *Data*. Se trata de una cantidad bastante pequeña de los casos pero contrasta con la métrica de repetitividad, donde destacamos la ausencia de "contraejemplos". Esto nos

[6] Nótese que cada par se suma 2 veces y por ello dividimos por 2.

[7] Al sumar $Eq(x,x')$ estamos sumando 5 menos la longitud del foco. De esta forma ponderan más focos más pequeños. El otro sumando aporta la proporción coincidente en los contextos. Si los contextos son muy similares sumará prácticamente uno y si son muy diferentes no sumará nada. El término $|Cont(x,x')|$ puede ser nulo si x y x' son cadenas complementarias y, para no dividir por 0, separamos ese caso otrogando ponderación nula.

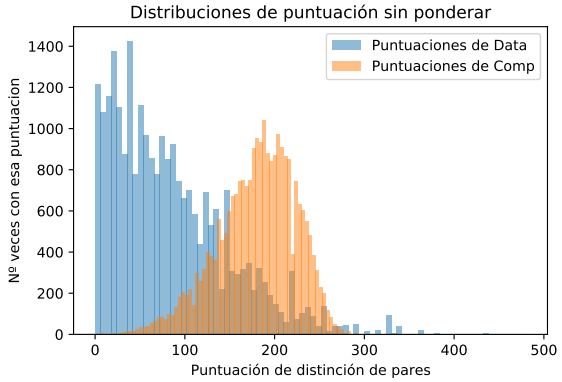


Figura 4.3.5: Distribución de DP sobre muestras aleatorias.

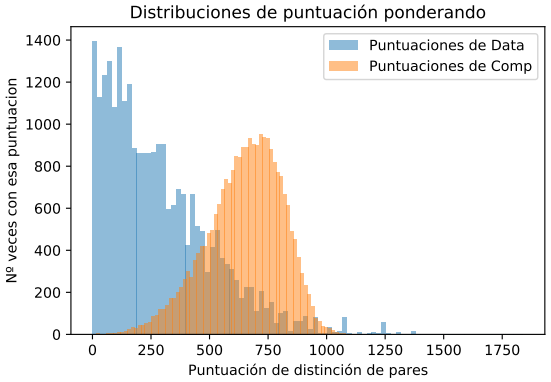


Figura 4.3.6: Distribución de DP_{Pond} sobre muestras aleatorias.

lleva a pensar que la repetitividad pueda ser más determinante que la distinción de pares para decidir la computabilidad o no de una función con un circuito "pequeño".

De todas formas, nuestros resultados están condicionados por el diseño de las métricas que, como ya hemos comentado, no son perfectas ni pretenden serlo. Son una herramienta para tener un *feedback* medianamente claro sobre nuestras conjeturas, tratando de guardar cierto equilibrio entre precisión para medir y eficiencia computacional. Por tanto, debemos ser muy cautelosos con las conclusiones que extraigamos de nuestros experimentos que, como mucho, pueden calificarse de indicios.

La nota positiva de los resultados es que, repitiendo el test clasificatorio que vimos para la métrica de repetitividad, tanto si ponderamos como si no, se mejoran los porcentajes de acierto respecto a aquella métrica. Además, los resultados de la métrica con ponderación mejoran a los de la métrica sin ponderar, reforzando las intuiciones que expusimos en la sección previa.

Con la métrica sin ponderar DP se obtuvo en torno a un 79 % de acierto para los ejemplos de $Data_{test}$ y en torno a un 85 % para los ejemplos de $Comp_{test}$. Con la métrica ponderada DP_{pond} el porcentaje de acierto subió al 83 % para $Data_{test}$ y al 87 % para $Comp_{test}$.

4.4. Modelos de Aprendizaje Automático

A continuación vamos a relatar nuestro breve paso por el Aprendizaje Automático, primeramente para optimizar el uso combinado de nuestras métricas, y después para tratar de descubrir nuevos patrones directamente sobre los datos obtenidos con el generador. Para ello, aprovechamos las distintas funcionalidades que ofrece la biblioteca de Python *sklearn* [4].

4.4.1. Combinación de métricas

Calculamos las puntuaciones de nuestra métrica de repetitividad L_{Rep} y de nuestra métrica de distinción de pares ponderada DP_{Pond} para todas las funciones de $Data$ y un subconjunto de funciones aleatorias de $Comp$ (de igual cardinal que $Data$). Repartiendo 2/3 de esas puntuaciones en conjuntos para entrenar y el 1/3 restante para evaluar, desarrollamos distintos modelos de Aprendizaje Automático para predecir la clase correcta de una función ($Data$ o $Comp$) en base a sus puntuaciones sobre las métricas. Estas pruebas pueden encontrarse en el *notebook* [14].

Observamos que la combinación más interesante de las métricas respondía a un patrón lineal y, de hecho, bastante sencillo. El uso de modelos más complejos tales como redes neuronales multicapa no mejoró los resultados que se obtenían con una simple regresión logística.

Con un modelo tan sencillo, que hace una combinación lineal del tipo $\theta_1 \cdot L_{Rep}(f) + \theta_2 \cdot DP_{Pond}(f)$, para predecir si f es una función de $Data$ o de $Comp$, se obtuvo un 91 % de acierto sobre el conjunto de test con

la matriz de confusión que puede verse normalizada y sin normalizar en las figuras 4.4.1 y 4.4.2.

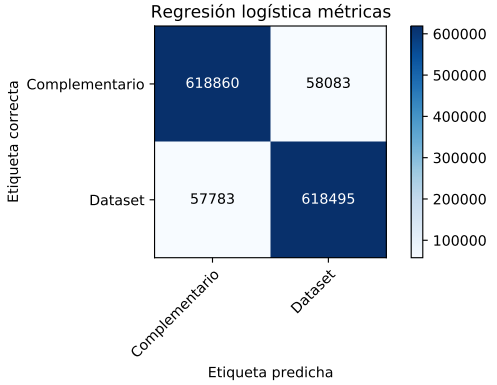


Figura 4.4.1: Matriz de confusión sin normalizar.

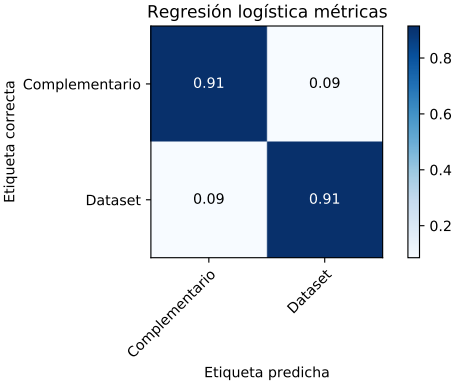


Figura 4.4.2: Matriz de confusión normalizada.

Los pesos de la red con los datos de puntuaciones estandarizados (tipificados para reescalarlos a una normal estándar) fueron $\theta_1 = 2.15$ y $\theta_2 = -1.98$. En este caso, al tener los datos rangos de valores similares, los pesos nos informan de la importancia que le da el modelo a cada variable. Vemos que las dos métricas tienen más o menos la misma importancia aunque, como es obvio, se utilizan de manera inversa ya que una mide "sencillez" para computar una función con tamaño "pequeño" mientras que la otra mide "dificultad" en este sentido.

4.4.2. Aprendizaje directamente sobre los datos

Hicimos un proceso similar, pero directamente sobre la representación de las funciones como cadenas, probando varios modelos que resolviesen un problema de clasificación entre *Data* y *Comp*. El *notebook* [13] contiene estas pruebas.

En este caso, las redes neuronales tuvieron unos resultados muy buenos, en particular mucho mejores que los modelos lineales (más de un 30 % de acierto de diferencia). Al probar con distintas topologías de red y diferentes valores para los hiperparámetros de aprendizaje, se llegó a la conclusión de que lo ideal era colocar una sola capa oculta con un gran número de neuronas. En la figura 4.4.3 podemos ver las curvas de aprendizaje (acierto en función del parámetro de aprendizaje de la red) para cantidades pequeñas de neuronas.

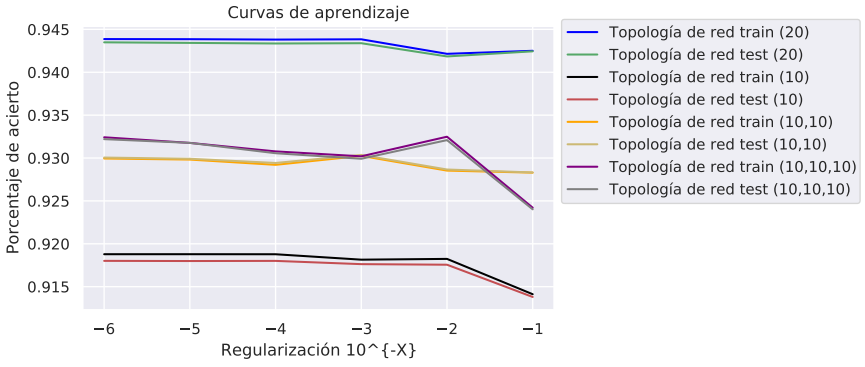


Figura 4.4.3: Curvas de aprendizaje para una capa oculta de 20 neuronas, una capa de 10, dos capas de 10 y tres capas de 10.

El hecho de que una sola capa con muchas neuronas funcione igual de bien que varias capas con el mismo número de neuronas en cada una, y que sí haya mejora al poner más neuronas en la primera capa nos

indica que es más importante fijarse en situaciones muy diversas que hacer combinaciones muy complejas. La primera capa oculta se encarga de tomar distintas perspectivas de los datos y las demás capas de combinar estas perspectivas, haciendo que el resultado sea cada vez menos lineal respecto a la entradas.

El acierto de esta red, para una sola capa oculta con 200 neuronas, fue del 97 % para los casos de *Data* y del 96 % para los casos de *Comp*. A continuación pueden verse su matriz de confusión.

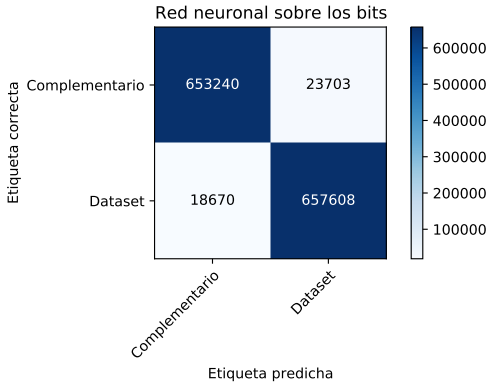


Figura 4.4.4: Matriz de confusión sin normalizar.

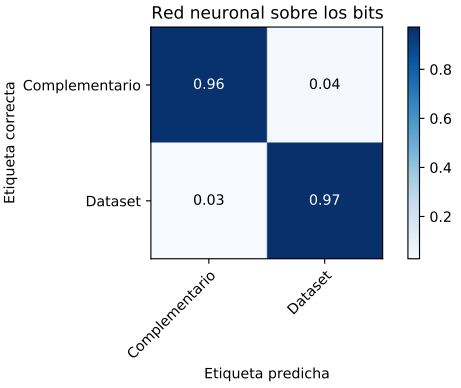


Figura 4.4.5: Matriz de confusión normalizada.

Existía un gran interés en entender lo aprendido por esta red, pero una simple visualización de los pesos no fue suficiente para descubrirlo o, al menos, para intuirlo. Debido a que un estudio profundo de los pesos habría requerido bastante tiempo (no solo para el estudio en sí sino también para documentarse sobre cómo interpretar las matrices de pesos adecuadamente), preferimos no hacerlo para evitar poner en riesgo la realización del experimento final.

Actualmente estamos tratando de encontrar simetrías en los pesos de la capa oculta y variando con cuidado el número de neuronas de esta capa para identificar qué perspectivas resultan irrenunciables para la red y qué relación hay entre ellas. Esperamos resolver pronto este enigma, que podría revelarnos algo muy interesante sobre complejidad de circuitos cuyo calado podría idealmente alcanzar incluso a P vs NP.

Por otra parte, tras varios modelos alternativos que funcionaban bastante mal, obtuvimos unos resultados moderadamente buenos con árboles de decisión. Sin embargo, sus porcentajes de acierto eran en el mejor de los casos del 85 % (véanse las matrices de confusión 4.4.6 y 4.4.7). Esto es casi lo que obtenían nuestras métricas por separado en el sencillo test clasificatorio con el que las probamos y un 6 % menos de lo que acertaban las métricas combinadas. Por tanto, aunque existe interés por estos árboles, no es tan desmesurado como el que nos suscita la red neuronal. Lo que nos pueden descubrir sobre complejidad de circuitos estos árboles no parece más potente que lo que hemos entendido ya de repetitividad y distinción de pares. En cualquier caso, podía ser algo nuevo y por ello también interesante para nosotros.

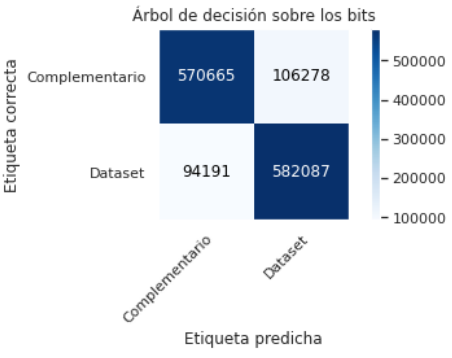


Figura 4.4.6: Matriz de confusión sin normalizar.

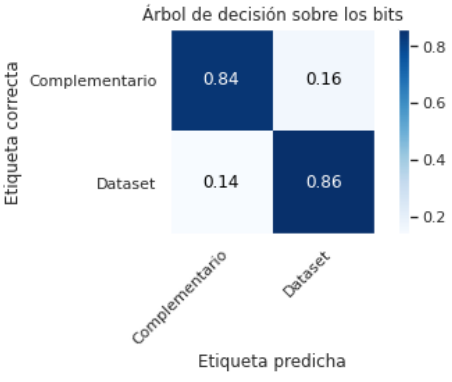


Figura 4.4.7: Matriz de confusión normalizada.

Los árboles son modelos más fáciles de interpretar que las redes neuronales y, por tanto, decidimos invertir algo de tiempo en observar su representación, sin llegar a entrar en un estudio muy meticuloso, tratando de extraer alguna conclusión.

Apreciamos ciertas "incompatibilidades" entre las entradas, como las que presagiamos cuando desarrollamos nuestra conjetura de distinción de pares cruzados. En muchos nodos del árbol, a los que se llega bajando por una rama (lo que supone fijar ciertos valores de salida para ciertas entradas) se percibía un cambio drástico en la decisión a tomar en función de la salida para otra entrada.

Es decir, fijado el valor de una serie de entradas $x_1, ..., x_m \in \{0,1\}^5$ a una serie de valores de salida $s_1, ..., s_m \in \{0,1\}$, se apreciaba que cierta elección $s_k \in \{0,1\}$ de la salida para cierta entrada $x_k\{0,1\}^5$ (diferentes de las anteriores) decantaba claramente los casos hacia *Data*, mientras que la elección opuesta $|1 - s_k| \in \{0,1\}$ decantaba claramente los casos hacia *Comp*. En el primer caso se estaría reflejando "compatibilidad" entre entradas y en el segundo "incompatibilidad".

En la figura 4.4.8 podemos ver un ejemplo de esto. Se alcanza el nodo que señala la flecha amarilla con una serie de valores de salida fijos para las entradas $0, 1, 2, 5, 8, 9, 12, 16, 17, 24$ y 25[8]. Esta situación parece ser bastante más común en funciones computables con un circuito "pequeño" pues lo cumplen 33.845 funciones de *Data* frente a tan solo 2.619 funciones de *Comp* (inicialmente están en igual cantidad). Lo interesante es que, si además de esto, se exige que la función devuelva '0' para la entrada 00100 (4 en binario), apenas 94 de las 33.845 funciones de *Data* lo cumplen. Es decir, mientras que la salida fijada para las entradas $0, 1, 2, 5, 8, 9, 12, 16, 17, 24$ y 25 resultaba "compatible" para computarse con un circuito "pequeño", solo lo es realmente si para la entrada 4 hay que sacar un 1.

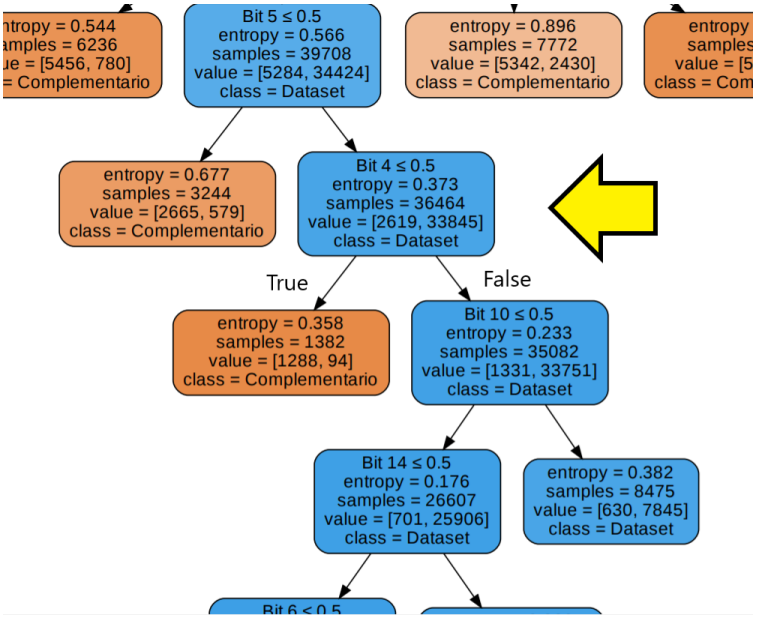


Figura 4.4.8: Nodo cuyo hijo izquierdo supone un cambio drástico de decisión.

El siguiente paso sería entender qué relación guardan las entradas que resultan "incompatibles". Como ya discutimos, el patrón que pueda haber entre ellas debería tener que ver con la igualdad entre sus bits como representación binaria, teniéndose en cuenta a la vez la igualdad o no de sus salidas. (Algo así como lo que intentamos modelar con nuestra distinción de pares cruzados).

Se comprobó visualmente que la distinción de pares cruzados no era exactamente lo que el árbol había aprendido, aunque sí se encontró alguna rama que, en cierto modo, parecía mostrarla. Sin embargo, los nodos más llamativos en el árbol (tales como el que hemos mostrado en la figura 4.4.8) escondían algo diferente.

[8]Las entradas están en la representación decimal asociada a su valor en binario.

No nos sobraba el tiempo y un análisis más exhaustivo del árbol habría requerido desarrollar bastante código para poder recopilar la información que nos interesara. La implementación de biblioteca que estábamos usando no proporciona un acceso tan selectivo como para poder recorrer el árbol y para recopilar las entradas (y salidas asociadas) que, por número de casos de cada clase, resultasen "incompatibles" por motivos sillares a los de la figura 4.4.8. No se nos ocurre otra manera de obtener estos grupos de entradas que no sea recorrer el árbol sistemáticamente. La única alternativa sería sacarlos nosotros a mano mirando el árbol e introducirlos, también a mano, en un programa que los utilice para revelarnos los patrones que hay en cada uno, pero en el árbol hay cientos de nodos y seguramente muchas decenas de grupos que sacar como para hacerlo de esta manera.

Tratar este asunto desde los posibles grupos de entradas y las posibles salidas asociadas para ellas conllevaría una combinatoria inasumibles (al haber 32 entradas hay 2^{32} posibles grupos de entradas). Sería interesante aprovechar la estrategia del árbol de ir eligiendo la variable que más reduce la entropía. Pero eso, en definitiva, nos obliga a implementar un árbol de decisión o entender una implementación que exista y nos dé acceso a él. Es algo interesante que seguramente abordaremos en el futuro, pero que queda relegado a cuando acabemos de descifrar los pesos de la red.

Capítulo 5

Prueba final: *Set Cover* vs *Mayoría*

Para ponerle el broche final al trabajo, reorientamos el foco hacia el problema que motivaba nuestra labor y experimentos, P vs NP. La complejidad de circuitos tiene interés por sí misma, pero el hecho de haber podido aportar algo, aunque solo sea un granito de arena, en uno de los problemas del milenio, resultaba especialmente excitante para nosotros. Nuestra idea era terminar el trabajo tratando de comprobar si las métricas y modelos automáticos desarrollados habían conseguido captar algo de este problema. Para ello, enfrentamos mediante nuestros modelos la tabla de verdad de un problema de la clase NP con la tabla de verdad de un problema de la clase P, teniendo la esperanza de obtener un comportamiento diferente en cada caso.

5.1. Elección de los problemas y codificación de instancias

Como nuestra prueba iría enfocada en la hipótesis $P \neq NP$, debíamos escoger un problema de NP especialmente susceptible de no pertenecer a P, de lo contrario podíamos fracasar simplemente por estar buscando donde no hay nada. En este sentido nos fijamos en los problemas *NP-completos*, que actualmente no se sabe si están o no en la clase P y se encuentran interrelacionados de tal forma que si se descubriera que uno de ellos pertenece a la clase P automáticamente también sucedería para los demás.

Para escoger este problema *NP-completo* hay que tener en cuenta que la tabla de verdad que íbamos a construir es una función booleana $f : \{0,1\}^n \to \{0,1\}$ (como las que hemos tratado a lo largo del trabajo) que recibe las instancias del problema representadas mediante n bits. Al haber 2^n posibles instancias que hay que resolver y, dado que para los problemas *NP-completos* no conocemos soluciones de tiempo polinómico, por motivos computacionales era preciso imponer que n no fuese demasiado grande. Sin embargo, n es la cantidad de bits que tenemos para representar las instancias, que pretendemos que sean lo suficientemente importantes como para que afloren las dificultades del problema. Por eso, era importante escoger un problema en el que pudiésemos representar instancias considerables con no demasiados bits.

Por este motivo, se descartaron opciones como el problema SAT donde era preciso desperdiciar muchos bits en representar los símbolos lógicos $\wedge, \vee$ y $\neg$. Su versión 3-SAT resultaba algo mejor porque solo exige decidir las variables en cada literal y el $\neg$ de cada una, pero con apenas 6 variables ya son necesarios 4 bits por cada literal (3 para la variable y 1 para el $\neg$). Como hay 3 literales por cláusula disyuntiva, son necesarios hasta 12 bits por cada una y, para que la cantidad de instancias fuese asumible, apenas nos alcanzaría para 2 cláusulas disyuntivas (con 3 cláusulas ya tendríamos 2^{36} instancias, que son demasiadas).

En cambio, el problema *Set Cover*, que enunciamos a continuación, se presta bastante bien a representaciones interesantes con pocos bits.

Definición 5.1.1. *[**Problema Set Cover**] Dado un universo de elementos $\mathcal{U}$, un conjunto de cobertura $\mathcal{S}$ tal que $\mathcal{S} \subset \mathcal{P}(\mathcal{U})$, $\mathcal{U} = \bigcup_{S \in \mathcal{S}} S$ y un valor $k \in \mathbb{N}$ tal que $1 \leq k \leq |\mathcal{S}|$, el problema Set Cover consiste en determinar si existe $\mathcal{C} \subset \mathcal{S}$ tal que $|\mathcal{C}| \leq k$ y $\mathcal{U} = \bigcup_{C \in \mathcal{C}} C$.*

Es decir, el problema consiste en determinar si dado un universo $\mathcal{U}$, un conjunto $\mathcal{S}$ de subconjuntos que cubren al universo y cierto entero k, es posible quedarse con una cantidad de conjuntos de $\mathcal{S} \leq k$ cubriendo aún a $\mathcal{U}$.

Con apenas 20 bits, que suponen la razonable cantidad de $2^{20} = 1.048.576$ instancias, pudimos representar todos los conjuntos con 4 subconjuntos de un universo con 5 elementos ($\mathcal{U} = \{e_1, e_2, e_3, e_4, e_5\}$ y $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$ con $S_i \subset \mathcal{U}$ para $i \in \{1, ..., 4\}$). La representación simplemente consistió en utilizar los bits de 5 en 5 para indicar la pertenencia o no de cada elemento del universo a cada uno de los subconjuntos.

Es decir, con nuestra representación una entrada $(x_{11}, ..., x_{15}, x_{21}, ..., x_{25}, x_{31}, ..., x_{35}, x_{41}, ..., x_{45}) \in \{0,1\}^{20}$ cumple que $x_{i,j} = 1$ si y solo si el i-ésimo subconjunto de $\mathcal{S}$ contiene al j-ésimo elemento de $\mathcal{U}$.

Puede parecer que falta por representar el entero $k \in \{1, ..., 4\}$ y que habría que añadir 2 bits a la representación anterior, pero no es así. Optamos por producir 4 tablas de verdad distintas, una para cada posible valor de k. Esta opción nos pareció más clara y ordenada, y podía desarrollarse fácilmente calculando cuál era el menor k' para que la respuesta a una instancia de *Set Cover* fuese afirmativa. Colocando un '1' en la correspondiente salida de todas las tablas con $k \leq k'$ y un '0' en la salida para todas las tablas con $k > k'$ se obtendría el resultado adecuado (en lugar de resolver el problema de decisión enunciado en 5.1.1 buscamos el menor tamaño de cobertura de una instancia).

Notemos también que, de esta forma, no hay ninguna garantía de que el conjunto $\mathcal{S}$ representado por una entrada consiga cubrir el universo $\mathcal{U}$. Como estas instancias no tienen sentido, ya que la definición 5.1.1 del problema exige que el conjunto $\mathcal{S}$ sea una cobertura, se decidió marcar de manera diferente sus salidas para no confundirlas con las instancias con salida '0'. Colocamos 'F' en lugar de '0' para las instancias donde no hay una subcobertura del tamaño requerido debido a que el conjunto $\mathcal{S}$ ni siquiera es cobertura del universo.

Por otra parte, tratamos de elegir un problema de la clase P cuya complejidad, al menos desde el punto de vista de los circuitos, pudiera parecer mayor de lo que realmente es. Nos estamos refiriendo a problemas como *Mayoría* o *Paridad* que, como ya comentamos anteriormente, están en la clase $P_{/poly}$ pero no en la clase AC^0, lo que significa que son computables con circuitos de tamaño polinómico solo cuando la profundidad de circuito depende del tamaño y no es una constante fija (de hecho, para ellos es suficiente con una profundidad logarítmica respecto al tamaño de su entrada).

Las tablas de verdad de estos problemas están equilibradas en cuanto la cantidad de valores a '0' y '1'. Por ese motivo, al menos en cuanto a su apariencia superficial, deberían resultar problemas tanto o más difíciles que *Set Cover* para nuestras métricas y modelos. Si utilizásemos un problema de P cuya salida fuese casi todo '0' o casi todo '1' evidentemente resultaría más "fácil" que *Set Cover* para nuestros modelos, sin embargo sería una prueba poco interesante.

Enunciamos formalmente el problema *Mayoría*, aunque debido a su nombre seguramente ya se pueda intuir en qué consiste.

Definición 5.1.2. ***[Problema Mayoría]*** *Dada una cadena* $(e_0, ..., e_{n-1}) \in \{0,1\}^n$ *el problema Mayoría consiste en determinar si* $\big|\{e_j \mid j \in \{0, ..., n-1\}, e_j = 1\}\big| > \big|\{e_j \mid j \in \{0, ..., n-1\}, e_j = 0\}\big|$.

Es decir, es el problema que consiste en determinar si la entrada tiene más valores a '1' que a '0' o no.

La codificación de instancias no es necesaria porque es un problema que se define directamente sobre entradas binarias. Por tanto, simplemente se resolvió para todas las posibles instancias con $n = 20$, generando así su tabla de verdad (en la carpeta [12] pueden encontrarse las tablas de verdad obtenidas para *Set Cover* y *Mayoría*, junto a los programas que las computan y demás recursos relacionados con este experimento).

5.2. Uso de las tablas de verdad y muestreo

Utilizar todas las tablas de verdad completas obtenidas para *Set Cover* era poco astuto e inasumible desde el punto de vista computacional.

Para empezar, no iba a ser posible porque, por ejemplo, nuestras métricas de distinción de pares tenían un coste cuadrático respecto al tamaño de la cadena a puntuar, que en este caso sería de millones de bits.

Y tampoco sería astuto, porque la dificultad de los problemas *NP-completos* se concentra en zonas relativamente pequeñas de sus tablas de verdad. Por ejemplo, en el caso de *SAT*, la zona difícil se encuentra donde la ratio entre el número de cláusulas y el de variables es de aproximadamente 4.26 (véase [3]). En otras zonas, estos problemas resultan incluso más triviales que los más sencillos de la clase P. Tener en cuenta también esas zonas "sencillas" (que son mayoritarias en las tablas) podría diluir la dificultad de las otras partes hasta el punto de hacerlas pasar desapercibidas.

En el caso de *Set Cover* primeramente hay que centrar la mira en las tablas para $k \simeq |\mathcal{S}|/2$. Las instancias de *Set Cover* con un valor de k muy pequeño suelen tener salida '0', ya que son bastante pocas las instancias que tienen los subconjuntos tan cargados de elementos como para que una cantidad tan pequeña de ellos permita mantener el cubrimiento. Y las instancias con un valor de k muy grande pecan de todo lo contrario, de una mayor tendencia a salida '1' debida a la facilidad para mantener el cubrimiento teniendo que quitar

pocos subconjuntos. Esto proporciona cierta intuición sobre cómo la dificultad de *Set Cover* aumenta según nos alejamos de los valores extremos para k, alcanzando su máxima plenitud en el valor intermedio $|\mathcal{S}|/2$.

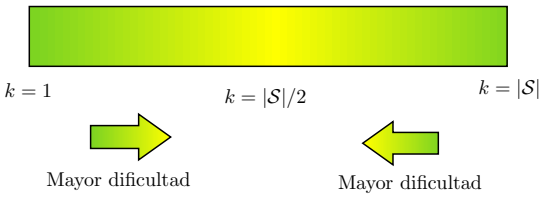


Figura 5.2.1: Representación sobre la intuición de cómo se reparte la dificultad de *Set Cover* según k.

Esta intuición se manifiesta de manera notable solo cuando $|\mathcal{S}|$ toma valores suficientemente grandes y, en cualquier caso, no es determinante pues hay muchos otros factores a tener en cuenta como, por ejemplo, la cantidad de elementos que tienen los subconjuntos de la cobertura. Sin embargo, da cierta intuición de por qué nosotros escogimos $k = 2$.

Además, dentro de esta inmensa tabla, también hay muchas zonas tremendamente triviales que sería deseable evitar (precisamente por el tema de si los subconjuntos traen o no muchos elementos). Por ejemplo, hay muchos miles de instancias para las que alguno de los 4 subconjuntos tiene los 5 elementos (es decir, es el propio $\mathcal{U}$). Para estas instancias la salida es trivialmente '1', pues es posible cubrir $\mathcal{U}$ con un subconjunto (el propio $\mathcal{U}$).

Tratamos de evitar estas instancias triviales tomando muestras de la tabla que pasasen algún tipo de filtro de "dificultad". Como nuestros modelos de Aprendizaje Automático funcionaban exclusivamente para cadenas de tamaño 32, decidimos tomar precisamente este tamaño para las muestras.

Para considerar que una muestra era suficientemente "complicada" para ser seleccionada decidimos exigir que su cantidad de valores a '0' y a '1' estuviese bastante equilibrada (como mucho hubiese 2 de diferencia). Sabemos que las cadenas con muchos más valores a '0' que a '1' (o viceversa) son mucho más "fáciles" desde el punto de vista de la complejidad de circuitos y, por tanto, resultaba razonable descartarlas.

También vimos pertinente descartar las muestras que trajesen algún valor 'F' (aquellas en que $\mathcal{S}$ ni siquiera es cobertura). Aunque habría sido legítimo tomarlas como valores '0', porque un problema análogo en el que no se asegure si $\mathcal{S}$ es o no cobertura tiene la misma complejidad que *Set Cover*, no eran muchísimos casos y habitualmente irían en las mismas muestras. Por tanto, no nos generaba demasiado conflicto ceñirnos al enunciado del problema 5.1.1.

$$(s_{31}, ..., s_0) \in \{0,1\}^{32} \text{ es seleccionada} \iff \Big|\big|\{s_i \mid s_i = 1\}\big| - \big|\{s_i \mid s_i = 0\}\big|\Big| \leq 2 \text{ y } s_i \neq F \ \forall i \in \{0, ..., 31\}.$$

De cara a escoger esas muestras de 32 salidas en las tablas de verdad, tuvimos cierto cuidado en no hacerlo variando siempre los mismos 5 bits. Esto podía redundar en cierto "privilegio" de bits que no parecía muy deseable, pues intuíamos que podía influir de algún modo en la "complejidad local" de la cadena. Por ejemplo, variando siempre los 5 últimos bits para escoger las muestras solo estaríamos alterando la cantidad de elementos del último subconjunto de $\mathcal{S}$.

Sucede lo mismo que cuando discutimos la permutación del orden en las entradas de una función, donde argumentamos que la posición que ocupe un bit en la representación es totalmente irrelevante. Por tanto, para ser justos, decidimos tomar las muestras sorteando los 5 bits que serían variados y eligiendo también de manera aleatoria los valores para los 15 bits restantes que mantendríamos fijos.

Por ejemplo, si al elegir 5 bits aleatorios entre los 20 bits $x_{i,j}$ con $i \in \{1, ..., 4\}, j \in \{1, ..., 5\}$ se obtienen $x_{1,3}, x_{2,1}, x_{2,2}, x_{3,5}$ y $x_{4,3}$ y, al elegir un valor binario aleatorio para los otros 15 bits se obtiene 011001011100110, la muestra estaría formada por la salida para las entradas que indica la tabla 5.1.

Para *Mayoría* hicimos el mismo muestreo, evitando con ello que alguno de los problemas partiese con ventaja por el simple hecho de tener muchas más salidas a '0' que a '1' o viceversa.

$x_{1,1}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{1,2}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$x_{1,3}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{1,4}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$x_{1,5}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{2,1}$	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
$x_{2,2}$	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
$x_{2,3}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{2,4}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$x_{2,5}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{3,1}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$x_{3,2}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$x_{3,3}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$x_{3,4}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{3,5}$	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
$x_{4,1}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{4,2}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$x_{4,3}$	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
$x_{4,4}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$x_{4,5}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabla 5.1: Ejemplo de entradas donde se varían los bits $x_{1,3}, x_{2,1}, x_{2,2}, x_{3,5}$ y $x_{4,3}$ y los demás se fijan a 011001011100110, cuyas salidas formarían una muestra.

5.3. Resultados del experimento

Los resultados que obtuvimos, tras probar nuestras métricas y modelos sobre decenas de miles de muestras fueron los siguientes:

- **Para la métrica de repetitividad:** una puntuación media de 22.44 para las muestras de *Set Cover* y de 19.11 para las muestras de *Mayoría*.
- **Para la métrica de distinción de pares ponderada:** una puntuación media de 629.44 para las muestras de *Set Cover* y de 625.46 para las muestras de *Mayoría*.
- **Para la regresión logística entrenada con las métricas:** un 56.42 % de respuesta "Complementario" en las muestras de *Set Cover* y un 43.58 % de respuesta "Complementario" en las muestras de *Mayoría*.
- **Para la red neuronal entrenada con los datos:** un 78,02 % de respuesta "Complementario" en las muestras de *Set Cover* y tan solo un 22,46 % de respuesta "Complementario" en las muestras de *Mayoría*.

La métrica de repetitividad, que supuestamente otorga puntuaciones más altas a problemas más "fáciles", funcionó de manera contraria a lo esperado, al otorgarle más puntuación de media a las muestras de *Set Cover* que a las de *Mayoría*. Sin embargo, la métrica de distinción de pares ponderada, que otorga puntuaciones más altas a problemas más "difíciles", proporcionó un resultado más coherente aunque igualmente poco claro, al darle ligeramente más puntuación a las muestras de *Set Cover*.

La regresión logística que combinaba las métricas para predecir logró, de algún modo, captar la mayor dificultad de *Set Cover*, aunque de nuevo de manera muy tenue (56.42 % de respuesta "Complementario" en las muestras de *Set Cover* vs 43.58 % de respuesta "Complementario" en *Mayoría*). Sin duda lo más sorprendente fue que la red que utiliza directamente la representación como cadena de las funciones distingue las muestras de ambos problemas bastante bien. Identifica como "complementario" (tamaño de circuito supuestamente "grande") al 78,02 % de las muestras de *Set Cover* frente a tan solo el 22,46 % de las muestras de *Mayoría*.

Es importante ser muy prudentes con estos resultados pues, debido a la complejidad del asunto que pretendíamos abordar (algo así como un test empírico de P vs NP), hay muchas simplificaciones de por medio que, aunque hemos tratado de hacer de manera razonable, evidentemente condicionan los resultados. La más importante de ellas es que estamos estudiando la complejidad de cada problema sobre minúsculas muestras de tamaño 32. Es difícil creer que la complejidad de los problemas *NP-completos* se pueda proyectar demasiado

sobre tamaños tan ridículamente pequeños en relación a los millones de bits que hay en las tablas de verdad que computamos. Sin embargo, sí que hay que reconocer que los resultados de la red neuronal entrenada con los datos son bastante llamativos aunque, por supuesto, no debemos sacar conclusiones precipitadas de ello.

Conclusiones

Terminamos el trabajo haciendo un breve balance de los resultados obtenidos, de cómo los interpretamos y de lo que podrían suponer. Recogemos también algunas ideas que podrían ser útiles para continuar en esta misma línea en el futuro.

Para empezar, estamos bastante satisfechos con las conjeturas obtenidas y las métricas elaboradas. Antes de ponernos con ello temíamos que nuestras intuiciones no fuesen correctas y correlacionasen mal con los datos. En tal caso solo nos habrían quedado los modelos automáticos que, aunque garantizan buenos resultados (siempre que haya patrones más o menos coherentes en los datos los descubren), en ocasiones son difíciles de descifrar.

Nuestra misión principal era entender y poder explicar aquello que descubriésemos en relación a factores influyentes en el tamaño mínimo de circuito para computar una función y, tanto la repetitividad de patrones disjuntos con longitud potencia de 2, como la distinción de pares cruzados, fueron comprendidos, explicados y ampliamente cotejados sobre los datos. La métrica de repetitividad permitió diferenciar con aproximadamente el 83 % de acierto las funciones de *Data* y *Comp*, mientras que la métrica ponderada de distinción de pares obtuvo, para la misma prueba clasificatoria, unos cuantos puntos más de acierto. La combinación de las métricas mediante un modelo automático elevó el acierto en la clasificación al 91 %.

Pensamos que uno de los principales motivos para estos buenos resultados es la calidad y cantidad de los datos obtenidos con el generador sistemático. Invertimos bastante tiempo y esfuerzo en diseñar un nuevo algoritmo generador de circuitos que los recorriese crecientemente en tamaño. También tratamos por todos los medios de aumentar la cantidad de funciones diferentes que formarían parte de nuestro dataset sin perturbar su calidad en exceso. Con lo primero nos garantizamos que las funciones obtenidas fuesen representativas de las computables con circuitos "pequeños". Y con lo segundo proveernos de la máxima cantidad de datos para que el funcionamiento de modelos de Aprendizaje Automático fuese lo mejor posible.

Logramos recorrer crecientemente los circuitos mediante un esquema de unión donde resulta absolutamente clave la eliminación de subcircuitos repetidos. Por su parte, la enorme cantidad de funciones obtenidas fue debida principalmente al cambio del repertorio de puertas y a la mezcla de salidas para distintas ejecuciones. Aun así, nos dimos cuenta algo después de que podíamos mejorar bastante el generador llevando estructuras de circuito en lugar de circuitos concretos. Esta es una de las cosas que quizás valdría la pena desarrollar en el futuro (la decisión de no hacerlo entonces tuvo que ver con que los datos ya eran suficientemente buenos y abundantes y no sobraba tiempo para otros menesteres).

Esta mejora para el generador se nos ocurrió al tratar el tema de la invarianza bajo permutaciones, que resulta un aspecto de absoluta relevancia desde el punto de vista teórico. Llegamos a la conclusión de que cualquier factor que influya en el tamaño mínimo de circuitos para computar una función debe ser independiente del orden en que esta reciba sus entradas. Puesto que la repetitividad de patrones mostrada por los datos no cumplía con esta invarianza, elucubramos la posibilidad de que no fuese causa del tamaño mínimo de circuito sino consecuencia de algún otro factor. Habría otro factor invariante bajo permutaciones que tendría incidencia tanto en el tamaño mínimo de circuito como en la repetitividad de una función representada en distintas referencias. De esta forma, existiría cierta correlación entre las dos consecuencias siendo útil una para predecir a la otra (en particular la repetitividad para predecir el tamaño mínimo de circuito).

Por otra parte, los modelos de Aprendizaje Automático entrenados directamente con los datos resultaron muy esperanzadores, aunque haya aún bastante trabajo por delante para descifrar lo que aprendieron. El más interesante de ellos fue una red neuronal con hasta un 97 % de acierto para distinguir "a ciegas" ejemplos de *Data* y *Comp*. Llegamos a descubrir que su éxito se debía a las múltiples perspectivas que tomaba de los datos y no tanto a la manera de combinarlas. Pensamos que entender lo aprendido por esta red podría ser fundamental de cara al tamaño mínimo de circuito para computar una función.

También descubrimos, al trabajar con árboles de decisión, ciertas incompatibilidades entre algunas entradas en función de sus salidas. La extracción de esos grupos de entradas del árbol para un nuevo análisis sistemático quedó pendiente, aunque pensamos que debería quedar en un segundo plano respecto al tratamiento de los pesos de la red neuronal. El acierto de los árboles era bastante inferior al de nuestras métricas (en torno a un 85 %), lo que nos lleva a pensar que no esconden algo tan suculento como la red.

El experimento final, cuyos resultados, debido a las múltiples simplificaciones que fue necesario hacer, hay que valorar con mucha cautela, nos reveló que la red neuronal parecía captar de algún modo una mayor dificultad en el problema *NP-completo Set Cover* que en el problema de la clase *P Mayoría*. Los resultados con esta red fueron bastante contundentes, pero pocas conclusiones se pueden extraer de un experimento tan condicionado por la simplificación. Estos resultados, que aún no sabemos muy bien cómo tomar, vendrían a decirnos que la dificultad extra de *Set Cover* sobre *Mayoría* es detectable con bastante precisión para trozos muy pequeños de sus tablas de verdad con instancias relativamente simples. Las combinación de métricas respondió en la misma linea pero de una manera mucho más ambigua, que quizás se aproxima más a lo que esperábamos obtener.

Bibliografía

[1] Sanjeev Arora y Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[2] Abdul Basit, Swastik Kopparty y Meng Li. *Lecture 3: AC0, the switching lemma*. URL: https://sites.math.rutgers.edu/~sk1233/courses/topics-S13/lec3.pdf.

[3] Alex Devkar, Kevin Leyton-Br, Eugene Nudelman y Yoav Shoham. *Understanding Random SAT: Beyond the Clauses-to-Variables Ratio*. URL: http://robotics.stanford.edu/users/shoham/wwwpapers/CP04randomsat.pdf.

[4] *Librería sklearn*. URL: https://scikit-learn.org/stable/.

[5] Enrique Román Calvo. *Trabajo de fin de grado: Estudio de la endogamia de los circuitos booleanos*. 2020. URL: https://eprints.ucm.es/id/eprint/61715/.

[6] Enrique Román Calvo (Galieve). *Algoritmo del índice (recorrido sistemático del espacio de circuitos booleanos)*. 2020. URL: https://github.com/Galieve/TFG-Informatica.

[7] Jorge Villarrubia Elvira. *Trabajo de fin de grado: Identificación experimental de las funciones booleanas que requieren circuitos extensos y aplicación al estudio de P vs NP*. Facultad de Informática, 2021.

[8] Jorge Villarrubia Elvira (Jorgitou98). *Carpeta con los ficheros de distintas ejecuciones mezclados*. URL: https://github.com/Jorgitou98/TFG/tree/main/Generador/ficherosParaMezclar.

[9] Jorge Villarrubia Elvira (Jorgitou98). *Dataset ampliado organizado en clases de equivalencia*. URL: https://github.com/Jorgitou98/TFG/blob/main/Generador/DatasetClases/dataSetEnClases.txt.

[10] Jorge Villarrubia Elvira (Jorgitou98). *Dataset ordenado sin funciones repetidas (quitados los circuitos mínimos pero no su tamaño)*. URL: https://github.com/Jorgitou98/TFG/blob/main/Generador/DatasetMezclado/funcionesSinRepeticion.txt.

[11] Jorge Villarrubia Elvira (Jorgitou98). *Implementación del algorimo generador para el repertorio completo*. 2020. URL: https://github.com/Jorgitou98/TFG/blob/main/Generador/GeneradorTodasPuertas.cpp.

[12] Jorge Villarrubia Elvira (Jorgitou98). *Programas y salidas de la resolución de Set Cover y Mayoría*. URL: https://github.com/Jorgitou98/TFG/tree/main/SetCoveryMayoria.

[13] Jorge Villarrubia Elvira (Jorgitou98). *Pruebas con modelos de Aprendizaje Automático directamente sobre los datos del generador*. URL: https://github.com/Jorgitou98/TFG/blob/main/ModelosAprendizajeAutomatico/modelosSobreDatos.ipynb.

[14] Jorge Villarrubia Elvira (Jorgitou98). *Pruebas con modelos de Aprendizaje Automático para las métricas*. URL: https://github.com/Jorgitou98/TFG/blob/main/ModelosAprendizajeAutomatico/modelosSobreLasMetricas.ipynb.

[15] Jorge Villarrubia Elvira (Jorgitou98). *Pruebas de la métrica de distinción de pares cruzados*. URL: https://github.com/Jorgitou98/TFG/blob/main/Metricas/distincionParesCruzados.ipynb.

[16] Jorge Villarrubia Elvira (Jorgitou98). *Pruebas de la métrica de repetitividad*. URL: https://github.com/Jorgitou98/TFG/blob/main/Metricas/pruebasRepetitividad.ipynb.