Universidad Complutense de Madrid

Facultad de Informática

Trabajo de fin de grado en Ingeniería Informática

Identificación experimental de las funciones booleanas que requieren circuitos extensos y aplicación al estudio de P vs NP

Experimental identification of Boolean functions requiring large circuits and its application to the study of P vs NP

Autor:
Jorge Villarrubia Elvira

Directores:
Ismael Rodríguez Laguna,
Narciso Martí Oliet

Junio 2021

Índice

Resumen

En este trabajo se han tratado de explorar, mediante la vía empírica, aspectos relacionados con la complejidad computacional de circuitos booleanos que podrían resultar fundamentales en una hipotética resolución del problema P vs NP. Más concretamente, el trabajo se ha enfocado en determinar qué factores influyen para que una determinada función booleana pueda ser computada mediante un circuito "pequeño" (es decir, con pocas puertas lógicas) o que, por contra, requiera de un circuito "grande".

Si un problema está en P, entonces existen circuitos booleanos, uno por cada tamaño n de entrada, que resuelven el problema con un tamaño polinómico respecto a n. Por contrapositivo, si dichos circuitos no existen, el problema no está en la clase P. Por eso, averiguar qué funciones booleanas requieren circuitos "grandes" podría ser clave para demostrar la no pertenencia de ciertos problemas a la clase P (y por tanto, para tratar de demostrar que NP no está contenido en P).

Este trabajo comenzó con el desarrollo de un generador sistemático de circuitos que permitió obtener una tabla de funciones booleanas junto al tamaño de un circuito mínimo que las computaba. Dicha tabla sirvió como dataset para los experimentos posteriores, enfocados ya en determinar los factores que buscábamos. Para ello, fue fundamental una generación de circuitos suficientemente astuta, como para que la cantidad y calidad de los datos permitiera después obtener buenos resultados.

Posteriormente, se lanzaron diversas conjeturas fundamentadas tanto en una primera inspección visual de los datos, como en diversos aspectos intuitivos, cotejando todas nuestras hipótesis después sobre el dataset. Para valorar dichas conjeturas sobre los datos, se diseñaron varias métricas que permitieron cuantificar numéricamente cómo de ciertas eran nuestras hipótesis. De esta forma, se pudo decidir con cuáles de ellas era conveniente quedarse, debido a su buen comportamiento sobre los datos, y cuáles debían ser rechazadas.

Para el estudio de los datos resultó fundamental el hecho de que, por construcción del dataset, las funciones de nuestra tabla fueran exactamente las que podían computarse con los circuitos más pequeños. Esto supone que la tabla no solo contiene información sobre las funciones que aparecen anotadas en ella, sino también, en cierto modo, sobre todas las demás funciones, que al menos necesitarán el tamaño máximo que se ha llegado a anotar en la tabla para computarse.

Debido a ello, se pudo suponer que las funciones de nuestra tabla eran las computables con circuitos "pequeños" y las funciones del complementario eran las que necesitaban circuitos "grandes". Como teníamos que establecer cierta frontera entre "grande" y "pequeño" y, por construcción, las funciones complementarias a nuestro dataset requerían al menos tanto tamaño como el máximo del dataset, nos pareció razonable fijar esta frontera exactamente en el límite alcanzado por el experimento. No solo se pudo comprobar el buen funcionamiento de nuestras métricas sobre los datos, sino también corroborar un comportamiento opuesto sobre ejemplos aleatorios del complementario. Además, dicha suposición fue claramente refrendada por varios de nuestros experimentos, lo cual sustenta que es adecuado hacerla.

Seguidamente, ya con unas buenas métricas que aportaban cierto conocimiento sobre los datos, y que constituían algunos de los tan ansiados factores que buscábamos, se pasó a utilizar diversas técnicas de Aprendizaje Automático en busca de más factores. Los resultados fueron muy esperanzadores e interesantes en el sentido de que reforzaron enormemente el buen quehacer de nuestras métricas y dejaron entrever fuertes patrones de diferenciación entre nuestra tabla y el complementario. Pero, sin embargo, tuvieron la desventaja de no ser fácilmente comprensibles.

Se utilizaron redes neuronales que, aunque obtuvieron unos resultados excelentes, presentaban como *hándicap* la imposibilidad de entender lo aprendido más allá de un producto de matrices aplicado a las entradas. Nuestro propósito era descubrir propiedades generales con una explicación lógica detrás y desde el principio temíamos que la vía del Aprendizaje Automático diese problemas en ese sentido. Por eso sólo se exploró esta opción una vez obtenidas métricas que nos aseguraban haber cumplido ciertos objetivos.

Aunque sin mucho éxito, se trataron de analizar los pesos de las redes para descifrar los patrones que podían haber aprendido. Se consiguieron extraer ciertas conclusiones en base a los resultados para distintas topologías de red que, en cierto modo, reflejaban la naturaleza de lo aprendido sin terminar de concretarlo pero, simplemente observando los pesos, no se encontró un patrón concreto suficientemente claro. Un estudio

mucho más profundo y exhaustivo de los pesos requería una inversión de tiempo que, a esas alturas del trabajo, podía resultar excesiva poniendo en riesgo otros experimentos. Además, era una tarea que podía resultar poco fructífera al no haber ninguna garantía de que aquello que la red hubiese aprendido siguiera un patrón claro, sencillo y entendible para un ser humano. Por todo ello, se decidió pasar a otros asuntos dejando un hilo suelto del que seguir tirando en el futuro.

El broche final lo pusimos tratando de comprobar que, en efecto, las métricas y modelos obtenidos permitían, en cierto modo, diferenciar entre problemas de la clase P y problemas NP-*completos*. Esto es algo muy osado, pues constituiría un especie de test empírico de que $P \neq NP$, lo cual parecía difícil desde experimentos tan humildes como los nuestros. Por ello, no esperábamos grandes resultados, como mucho algún indicio o curiosidad.

Este experimento final presentaba varias dificultades. Para empezar, no era automatizable para una gran cantidad de problemas, pues la manera de resolver cada uno de ellos era particular. Se resolvió el problema NP-*completo Cliqué* (para grafos de 7 vértices) y el problema *Paridad* de la clase P. La elección de este problema NP-*completo* fue debida a la facilidad para codificar instancias medianamente interesantes con un número de bits no demasiado grande, de forma que el cálculo de la tabla de verdad resultase computable. El otro gran conflicto tenía que ver con la dificultad computacional de aplicar los modelos y métricas obtenidos sobre una tabla de verdad formada por millones de bits. Para solventar esto, se decidió hacer un muestreo de la tabla, aun a riesgo de que los resultados pudieran distorsionarse.

Para que el muestreo fuese interesante y permitiese cotejar si los modelos encerraban algo interesante, se sesgó dicho muestreo hacia las partes más complicadas de los problemas NP-*completos*. Las dificultades de los problemas NP-completos se concentran, en su tabla de verdad, dentro de una especie de frontera que separa instancias complicadas de otras que son bastante triviales. Por ello, la idea fue comparar los resultados de nuestros modelos para la parte complicada de estas tablas, frente al resultado de instancias de problemas de la clase P que, a priori, podrían ser igual de difíciles.

De esta forma, pudimos comprobar que algunos de nuestros modelos efectivamente habían conseguido captar las dificultades especiales que presentan los problemas NP-*completos*. Aunque los resultados no fueron todo lo claros que nos habría gustado, hay que tener en cuenta que están muy condicionados por el tamaño de las secuencias de bits analizadas, que eran muy pequeñas en relación a los millones de bits que formaban la secuencia total. Pero, teniendo en cuenta que el reto era muy grande y que no había muchas más opciones, podemos darnos por satisfechos con lo obtenido.

Palabras clave:

Circuito booleano. Función booleana. Complejidad de circuitos. Aproximación empírica a P vs NP. $P_{/poly}$. Generador de circuitos. Repetitividad de funciones. Distinción de pares cruzados.

Abstract

In this work, we empirically explore different properties of Boolean circuits related to their computational complexity, which can be of great interest when approaching the P vs NP problem. In particular, we focus on determining the factors influencing the size or depth of the Boolean circuits that compute a certain Boolean function (i. e., we study what makes the function computable via a small circuit, with few logic gates, or a big circuit).

It is important to notice that, for P problems, there exist Boolean circuits (one for each input size n) of polynomial size with respect to the input that can solve the problem. Conversely, the contrapositive states that whenever these circuits do not exist, then the problem is not in P. For this reason, studying what Boolean functions require big circuits to compute them could be a key to proving that there are certains problems in NP that do not belong to P (and therefore showing that $P \neq NP$).

Firstly, we developed a systematic circuit generator which allowed us to obtain the truth tables for different Boolean functions along with the size of the smallest circuits that compute it. The tables obtained were also used as datasets for later experimentation, which was focused on uncovering the determining factors we were looking for. With the intention of accomplishing this, we had to cunningly generate circuits to ensure that enough data was available, while we assured the quality of it.

Afterwards, we came up with several conjectures based on a first visual inspection of the data, as well as on different intuitive features detected. All of the assumptions were tested against the datasets. To assess all of the hypotheses, we designed several metrics to quantify the number of elements behaving as assumed. Thus, we could decide which ones were great fits for the data, and which ones should be discarded.

When studying the data it was essential to have the functions that could be computing using the smallest circuits. Hence, the truth tables do not only contain the information regarding the related functions, but they also shed light on the rest of the functions since, at least, they would require a circuit of size greater than or equal to the one depicted in the table.

In consideration of the foregoing, we could assume that the functions related to the tables were the ones computable via the smallest circuits, and the complementary ones were linked to big circuits. On the other hand, since we needed to establish a barrier between what we meant as small or big circuit sizes, we thought it was quite reasonable to set this limit as the one reached by the experiment itself (note that the complementary functions required a size no less than the one of the original datasets). Fortunately, we found out that not only the metrics reported great results on our dataset, but they also behaved appropriately for random complementary examples. Moreover, this behaviour was endorsed by several of our experiments on the data, supporting the methodology.

Subsequently, once we had great metrics to enlighten us with regard to the data, and which were themselves the factors we longed for, we started to apply different Machine Learning techniques. The results were really encouraging since they greatly supported the performance of our metrics, while they revealed contrastive patterns to differentiate between the tables generated and their complementary. However, the lack of interpretability in these techniques prevent us from gaining more insight about this.

We implemented neural network models that reported excellent results, but inner workings were not easy to grasp. Because our main goal was to figure out properties along with logical explanations on why they emerged, we were already concerned from the very first moment with the explainability and interpretability issue of Machine Learning approaches. Indeed, that is why we only applied these techniques after exploring the metrics which already met our initial objectives.

We studied the weights of the neural networks to try to unravel the implicit patterns learned, with modest success. We extracted several conclusions regarding the results obtained using different topologies, which in some sense reflect the nature of what was learned. Unfortunately, just by simple inspection on the weights it was not possible to conclude any key factor. Besides, since a deeper and comprehensive review of the model requires a great investment of time, we could not venture more into this. Further research in this matter is required, and it could potentially clear up the patterns learned by the models.

We put the finishing touches on this work by trying to verify that the metrics and models obtained allowed us, to some extent, to differentiate clearly between P and NP-*Complete* problems. This daring task could be conceived also as an empirical test to show that $P \neq NP$, a problem which at the beginning of this work looked unaddressable.

This final experiment presented some challenges. Firstly, it was not allowing automation in a wide range of problems, since the way every problem was solved was different. We solved the NP-*complete Clique* problem (for 7-vertices graphs), and the *Parity* problem of the P class. The choice of this NP-*complete* problem was based on the ease of coding quite interesting instances with a not too large number of bits, so that the computation of the truth table was feasible. The other vast challenge was related to the computational complexity of applying the models and metrics retrieved to a truth table composed of millions of bits. To solve this, we decided to sample the table, assuming that the results could be distorsioned as a risk.

For making the sampling more interesting, and for allowing us to check whether the models were hiding something crucial, we skewed it to take the hardest parts of the NP-*complete* problems. The difficulties of these problems are located, in their truth table, inside a frontier that separates the complex instances from the trivial ones. Therefore, we had the idea of comparing the results of our models for those complex locations, with the results of the instances of problems of P class that, a priori, seemed to have the same difficulty.

In this way, we were able to check that some of our models, indeed, had managed to capture the special difficulties that NP-*complete* problems present. Although the results were not as clear as we would have desired, we should take into consideration that they are very conditioned by the length of the analyzed bits sequences, which were really small in comparison to the millions of bits that are part of the complete sequence. However, considering the challenge was great and ambitious, and that there were no further options, we could be satisfied with the results obtained.

Keywords:

Boolean circuit. Boolean function. Circuit complexity. Empirical aproach to P vs NP. $P_{/poly}$. Circuit generator. Repeatability of functions. Distinction of crossed pairs.

Capítulo 1

Introducción

Para empezar, vamos a introducir los resultados fundamentales que dan sentido a nuestro labor en el trabajo. Asimismo, en adelante, trataremos de: por una parte justificar la importancia y el impacto que las conclusiones de nuestro estudio podrían tener sobre la resolución del problema P vs NP, y por otra parte comentar algunas de las simplificaciones que es necesario hacer para poder estudiar los distintos aspectos a tratar. El principal fin de este capítulo es contextualizar todo lo que vendrá después.

1.1. Antecedentes y motivación

Empezamos dando una explicación breve, básica y superficial sobre la cuestión que, en el fondo, motiva los experimentos realizados, justificando el porqué de su importancia. Comentamos también un trabajo previo que guarda similitudes con nuestra labor y que ha servido como referencia.

1.1.1. El problema P vs NP

En el ámbito de la Complejidad Computacional, uno de los aspectos más importantes para medir los recursos de la solución a un problema es el tiempo de cómputo invertido. En relación al mismo, se define la clase de complejidad P como el conjunto de lenguajes binarios decidibles por una máquina de Turing determinista en un número de pasos (tiempo de cómputo) polinómico respecto al tamaño de la entrada (definición 1.13 de [1]). Por su parte, se define la clase de complejidad NP como el conjunto de lenguajes binarios decidibles, en los mismos términos que los de P, pero para los cuales la máquina recibe, junto a la entrada, una "pista" apropiada, de tamaño polinómico respecto al que tiene la propia entrada (definición 2.1 de [1]).

Estas definiciones de las clases P y NP (existen otras equivalentes), son muy próximas al campo de la Complejidad de Circuitos, ya que, al fin y al cabo, consisten en una máquina que da una respuesta binaria ("pertenece" o "no pertenece") ante una entrada (la palabra a decidir) que también es un secuencia binaria (secuencia de unos y ceros). Por tanto, simplemente por definición, ya se puede entrever la posibilidad de estudiar estas clases desde los circuitos booleanos.

Pero, ¿por qué nos interesaría estudiar estas clases? Es evidente que muchos problemas de la clase NP sí pertenecen a la clase P. De hecho, atendiendo a la definición, todos los problemas de P están en NP, y se cumple la inclusión $P \subseteq NP$. Pero ¿realmente hay una dificultad extra, no subsanable con cálculos polinómicos, en encontrar esa "pista" que permite que un lenguaje de NP se decida como uno de P? Es decir, ¿se cumplirá también que $NP \subseteq P$? Si así fuera, la sociedad podría vivir una enorme revolución.

Las clases P y NP se pueden llevar equivalentemente a problemas de decisión donde las entradas no sean binarias, e incluso a problemas que no sean de decisión. Muchos de estos problemas tienen un enorme interés para los informáticos, y además, debido a su aplicabilidad, también para toda la sociedad. El problema del viajante o el problema de la mochila son algunos ejemplos cuya utilidad práctica es obvia. Pero, desgraciadamente, no existe una solución computacionalmente eficiente para ellos. Si ocurriese que $NP \subseteq P$, existiría una forma mucho mejor, al menos en términos asintóticos, de resolver instancias de estos problemas que actualmente resultan inasumibles. Con ello, muchas cosas del mundo que nos rodea cambiarían drásticamente (habría algoritmos de coste polinómico para resolver problemas muy relevantes).

Sin embargo, los entendidos en el tema confían poco, de hecho cada vez menos, en que esto pudiera ser cierto. Varias encuestas hechas en 2002, 2012 y 2019 a investigadores relacionados con la informática (algunos de ellos expertos en el tema) arrojan esa conclusión [3]. En la última, el 88 % se decantaba por decir que $P \neq NP$, mientras que el 12 % restante pensaba que $P = NP$. Restringiendo la pregunta solamente a expertos, la estadística era aún más clara, pues el 99 % afirmaba creer que $P \neq NP$. Detrás de estas opiniones, seguramente se encuentren aspectos, que mencionaremos más adelante, como el colapso de la jerarquía polinómica en el caso de que lo cierto fuese la coincidencia.

Probar que $P \neq NP$ no supondría la enorme revolución que comentábamos que podría causar la coincidencia, ya que simplemente cerraría un debate teórico, chafando falsas esperanzas y, seguramente, la mayoría de la sociedad ni se enteraría ni le daría importancia. Por tanto, es bastante menos excitante. Pero, como acabamos de comentar, parece lo más plausible y es la dirección en la que va enfocado nuestro trabajo, que trata de aportar su granito de arena para resolver el misterio. Nuestro trabajo trata de encontrar indicios que pueden reforzar la idea de que ambas clases no coinciden e incluso, idealmente, guiar una hipotética prueba formal al respecto. Para ello realizaremos un estudio empírico relacionado con la complejidad de circuitos booleanos, que, como ya hemos dejado intuir, está próxima a las clases de complejidad P y NP que generan todo este debate.

1.1.2. Complejidad de circuitos: clase $P_{/poly}$ y relación con P vs NP

El resultado más importante para nuestro trabajo tiene que ver con la clase de complejidad de circuitos $P_{/poly}$ que vamos a introducir en breve. Este resultado conecta la clase de complejidad $P_{/poly}$ con la clase P, que estábamos interesados en estudiar, mediante una inclusión que simplifica bastante la manera de abordar el estudio de los problemas de la clase P. Como veremos, el resultado da pie, de manera natural, a los experimentos desarrollados a lo largo del trabajo.

La clase $P_{/poly}$, de manera intuitiva, trata de englobar los lenguajes binarios decidibles mediante circuitos de tamaño "pequeño", al menos desde el punto de vista asintótico. A continuación vamos a dar algunas definiciones básicas sobre circuitos, recogidas en la sección 6.1 de [1], que son necesarias para presentar y entender la definición formal de $P_{/poly}$.

Estas definiciones son suficientes, como demuestra esa misma lectura, para deducir el resultado que pretendemos explicar. Sin embargo, más adelante necesitaremos ampliarlas (de manera legítima), introduciendo conceptos nuevos que exceden lo que recoge la mencionada lectura. Por tanto, constituyen simplemente la base de lo que vendrá después.

Definición 1.1.1. *[**Circuito booleano**] Un **circuito booleano** de n entradas (y una única salida) es un grafo dirigido acíclico, con n vértices sin aristas de entrada que llamaremos **vértices fuente** y un vértice sin aristas de salida que llamaremos **sumidero**.*

*Todos los vértices que no son fuente se conocen como **puertas**, las cuales están etiquetadas mediante diversos símbolos que representan las operaciones lógicas de AND, OR y NOT. Por su asociatividad, el número de entradas de AND y OR es variable pero no pasa lo mismo con NOT.*

*Los vértices fuente, a los que también llamaremos **entradas**, se consideran ordenados. Utilizaremos las etiquetas $e_0, ..., e_{n-1}$ para referirnos a ellos, reflejando dicho orden mediante el subíndice.*

Notemos que la definición 1.1.1 contempla la posibilidad de que el vértice sumidero sea a su vez un vértice fuente, lo que redunda en un circuito mono-vértice sin puertas. Estos circuitos serán especialmente importantes en el siguiente capítulo y nos referiremos a ellos como **circuitos degenerados**.

Dos conceptos muy simples, pero que resultan fundamentales, ya que los estaremos utilizando constantemente en adelante, son el tamaño y la evaluación de un circuito. Extendiendo la noción de tamaño a una familia de circuitos y también a un lenguaje, se puede definir directamente la clase $P_{/poly}$.

Definición 1.1.2. *[**Tamaño de un circuito**] Diremos que un circuito booleano C con $m \in \mathbb{N}$ puertas tiene tamaño m. Lo denotaremos como $|C| = m$.*

Definición 1.1.3. *[**Evaluación de un circuito**] Dado un circuito booleano C de n entradas y cierto **input** $x \in \{0,1\}^n$, la evaluación de C sobre x es la que surge, de manera natural, de propagar el valor de las entradas hacia adelante según indican las aristas del circuito, aplicando las operaciones lógicas de las diferentes puertas.*

Más formalmente, el valor de cada vértice v de C es:

- *El valor x_i de la i-ésima componente de la entrada si v es el i-ésimo vértice fuente.*

- *El valor que se obtiene al aplicar la operación lógica v sobre los valores de los vértices v_j que inciden sobre v, en caso de que v sea una puerta*[1].

Denotaremos por $C(x)$ al valor de la puerta sumidero según el esquema recursivo expuesto, que coincide con la evaluación del circuito C sobre el input x que pretendíamos definir.

Ejemplo 1.1.4. [**Circuito, tamaño y evaluación**] La siguiente figura muestra un primer ejemplo de circuito booleano C, de 4 entradas, que cumple la definición 1.1.1. Retomaremos este circuito en el capítulo 2, pues encierra aspectos que serán importantes para nosotros después. Por el momento solo queremos que queden claros los nombres introducidos en la definición: entradas en amarillo, puertas en naranja y sumidero (que en este caso también es una puerta) en verde:

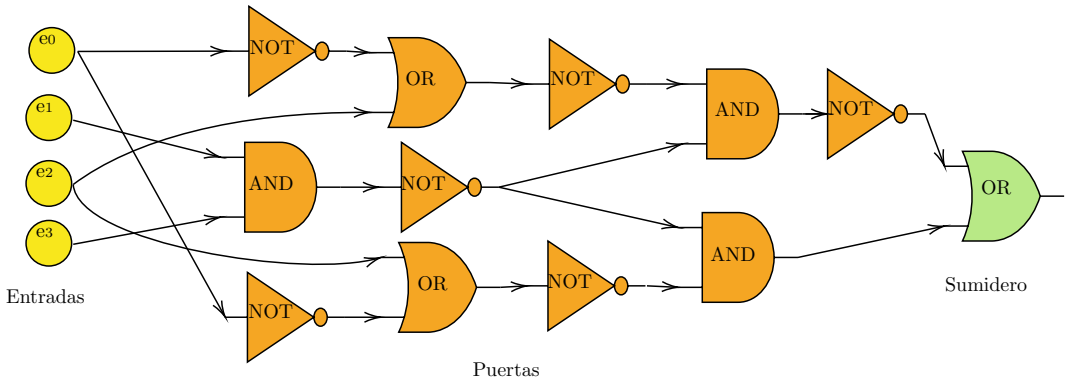


Figura 1.1.1: Ejemplo de circuito booleano con 4 entradas.

El circuito de la figura 1.1.1 tiene tamaño 12, las 11 puertas en naranja más la puerta sumidero.

Finalmente, su evaluación sobre el input $x = (x_0, x_1, x_2, x_3) = (1, 0, 1, 1) \in \{0,1\}^4$ (que habitualmente expresaremos sin las comas, es decir, $x = 1011$) es $C(x) = 1$, como se puede seguir en la siguiente figura:

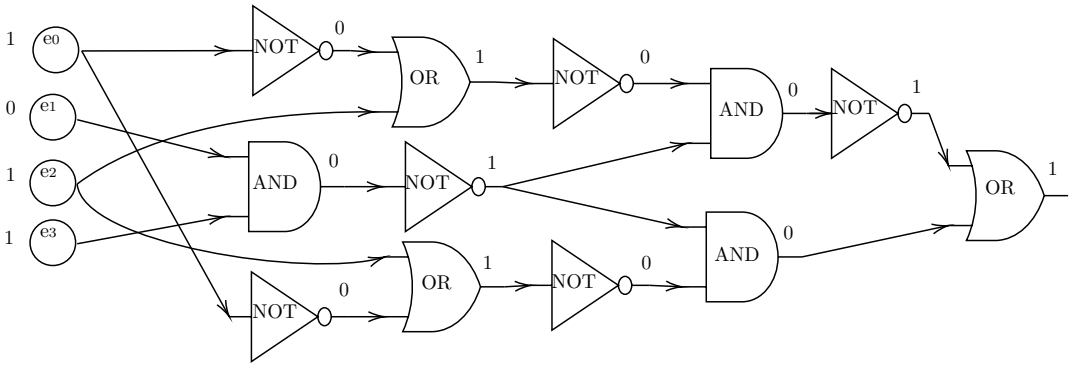


Figura 1.1.2: Evaluación $C(x)$ del circuito anterior para el input 1011.

▲

Definición 1.1.5. *[**Tamaño de una familia de circuitos**] Dada una función $T : \mathbb{N} \to \mathbb{N}$, decimos que el conjunto de circuitos $\{C_n\}_{n\in\mathbb{N}}$, donde, para cada n, el circuito booleano C_n tiene n entradas, es una familia de circuitos de tamaño $T(n)$, si $|C_n| \leq T(n)$ $\forall n \in \mathbb{N}$.*

[1] Nótese que la definición del valor de una puerta es, en general, recursiva, pues depende del valor de otras puertas. Pero en un número finito de pasos (a lo sumo tantos como vértices haya en el grafo) todas estas definiciones pueden expresarse exclusivamente en función del valor de vértices que son fuente, los cuáles actúan como caso base.

Definición 1.1.6. *[**Tamaño de un lenguaje**] Diremos que el lenguaje L está en $SIZE(T(n))$ si existe una familia de circuitos $\{C_n\}_{n\in\mathbb{N}}$ de tamaño $T(n)$ que decida el lenguaje, es decir, tal que $\forall x \in \{0,1\}^n$, $x \in L \Leftrightarrow C_n(x) = 1$.*

Definición 1.1.7. *[**Clase** $P_{/poly}$] $P_{/poly}$ es la clase de los lenguajes decidibles por una familia de circuitos de tamaño polinómico. Es decir:*

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

Por tanto, la clase $P_{/poly}$ está formada por aquellos lenguajes para los que es posible fijar un polinomio $p(n)$ que, para cada $n \in \mathbb{N}$, acote superiormente el número de puertas que se necesitan para decidir si una palabra con n símbolos binarios pertenece o no al lenguaje[2].

Notemos que la definición 1.1.1 no pone ninguna restricción sobre la aridad de las puertas *AND* y *OR*, que podrían ser binarias, como las utilizadas en el circuito del ejemplo 1.1.4, pero también podrían ser ternarias, cuaternarias, etc. No hay ninguna pérdida de generalidad en todos los conceptos introducidos si nos limitamos a solo puertas binarias, ya que, utilizando una cantidad polinómica de ellas, se puede construir una puerta de cualquier otra aridad anidándolas en un árbol como en la siguiente figura:

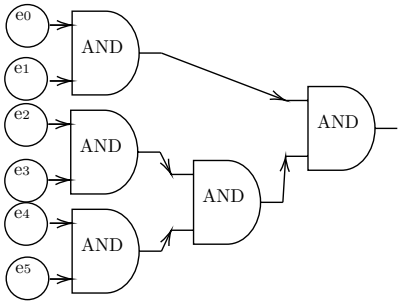


Figura 1.1.3: Ejemplo de árbol de *AND*'s binarias para $n = 6$ entradas.

En efecto, el número de puertas del primer nivel de este circuito para aridad n es $\leq \frac{n}{2}$, considerando la división real (2 entradas distintas van una puerta, pero si n es impar hay menos puertas que entradas). Puesto que, en cualquier otro nivel, la mecánica es la misma, pero las entradas son las salidas de puertas anteriores, nivel a nivel podemos acotar el número de puertas superiormente por la secuencia geométrica y finita de términos: $\frac{n}{2}, \frac{n}{4}, ..., \frac{n}{2^{\lceil log(n)\rceil}}$, cuya suma es:

$$\sum_{k=1}^{\lceil log(n)\rceil} \frac{n}{2^k} = \frac{n}{2}\left(\frac{1-(1/2)^{\lceil log(n)\rceil}}{1/2}\right) = n\left(\frac{2^{\lceil log(n)\rceil}-1}{2^{\lceil log(n)\rceil}}\right) \leq n$$

Por tanto, no hay problema con que en adelante trabajemos solo con puertas *AND* y *OR* de 2 entradas.

$P_{/poly}$ refleja, en cierto modo, poca complejidad para el reconocimiento de dichos lenguajes, al menos en términos asintóticos. El teorema 6.21 de [1] justifica que, para todo $n > 1$, existe alguna función $f : \{0,1\}^n \to \{0,1\}$ que requiere circuitos de tamaño superior a $2^n/10n$, lo cual crece bastante más rápido que cualquier polinomio. Revisando la demostración, no solo es que exista alguna función para la que suceda esto, sino que ocurre para casi todas, en el sentido de que la probabilidad de encontrar una función tal $\to 1$ si $n \to \infty$. Por tanto, lo que exige $P_{/poly}$, a medida que crece n, no es la norma sino que es la excepción.

Sin embargo, esto no debe ni mucho menos llevarnos a pensar que los problemas que, traducidos a lenguaje binario, estén en la clase $P_{/poly}$ son problemas "fáciles". En $P_{/poly}$ hay problemas incluso indecidibles, debido a que esta clase solo exige la existencia de circuitos de tamaño polinómico para decidir el lenguaje, sin fijarse de ninguna forma en cómo llegar a tales circuitos. Un ejemplo muy simple es el problema de parada:

[2] Aunque la definición de $P_{/poly}$ solo contemple polinomios de la forma n^c, cualquier polinomio se puede acotar por encima y por debajo, por uno que tenga esa forma, con lo cual es equivalente a considerarla con polinomios $p(n)$.

Ejemplo 1.1.8. [**El problema de parada está en** $P_{/poly}$]
Es fácil comprobar que cualquier lenguaje del alfabeto $\{0,1\}^*$, tal que $L \subseteq \{1^n \mid n \in \mathbb{N}\}$, es decir, cuyas palabras solo contienen el símbolo '1', está en $P_{/poly}$. Veámoslo.

- Si $n = 1$ basta un circuito que saque la propia entrada, que será de tamaño 0.
- Para cada $n > 1$:
  1. Si $1^n \notin L$, basta tomar C_n de tamaño constante que siempre saque '0' (por ejemplo, el *AND* entre cualquier entrada e_i y el *NOT* de dicha entrada $\overline{e_i}$).
  2. Si $1^n \in L$, basta coger como C_n un árbol de *AND*'s binarias anidadas como el de la figura 1.1.3, que ya justificamos que tiene tamaño $\leq n$.

Con lo cual se tiene una familia de circuitos $\{C_n\}_{n\in\mathbb{N}}$ con $C_n(x) = 1 \Leftrightarrow x = 1^n \in L$, de tamaño acotable por un mismo polinomio para todo n. Por tanto, $L \in P_{/poly}$.

Si consideramos el lenguaje:

$PARADA = \{1^n \mid n$ en binario codifica el par $< M, x >$ tal que la máquina de Turing M para con *input* $x\}$,

por lo visto anteriormente, $PARADA \in P_{/poly}$. Pero, como puede leerse con todo detalle en el teorema 1.11 de [1], a través de un argumento de diagonalidad, se prueba que el problema es indecidible, es decir, no puede computarse con una máquina de Turing. ▲

Esto pone de manifiesto que hay lenguajes cuya dificultad no está en los circuitos que hacen falta para decidirlos, sino en construir dichos circuitos. En el caso del ejemplo 1.1.8, saber montar esos circuitos para $PARADA$ supone resolver el propio problema, porque implica decidir si hay que construir la versión que acepta o la versión que rechaza, lo cual es equivalente a decidir si la máquina para o no. Por tanto, $P_{/poly}$ solo capta una parte de lo que hace "fácil" a un lenguaje y se olvida de la otra.

Intuitivamente, añadir la otra parte, la de poder construir los circuitos "fácilmente", es lo que forma la clase P. Las familias $\{C_n\}_{n\in\mathbb{N}}$ de circuitos *P-uniformes* son aquellas para las que una máquina de Turing, en un número de pasos polinómico, puede describir C_n a partir del *input* 1^n (la forma de que la máquina conozca el tamaño n). Y como la descripción que puede hacerse en un número de pasos polinómico tiene que tener tamaño polinómico, los lenguajes decidibles por familias *P-uniformes* son los que unen ambas cosas, la construcción en tiempo polinómico y el tamaño también polinómico. Y en efecto, se cumple que los lenguajes decidibles por estas familias coinciden con los lenguajes de la clase P.

Para probarlo, en una de las dos implicaciones hay que demostrar el resultado que damos a continuación, que es el más importante para nosotros.

Teorema 1.1.9. *[**Relación entre P y** $P_{/poly}$]*

$$P \subset P_{/poly}$$

Demostración. Ver teorema 6.6 de [1]. Intuitivamente, podemos representar en cada instante de ejecución el estado, la posición del cursor y el contenido completo de cinta de una máquina de Turing con una cadena binaria donde cada bit puede obtenerse mediante un circuito en función de algunos bits de la correspondiente cadena binaria del instante anterior. Como cualquier máquina de Turing que ejecute durante tiempo polinómico usará memoria polinómica, el tamaño del circuito completo resultante es también polinómico. □

Pero conocer la coincidencia entre los lenguajes decidibles por familias *P-uniformes* y los de la clase P (puede verse la prueba en el teorema 6.13 de [1]) nos proporciona la información adicional sobre qué falta para que $P_{/poly}$ sea igual a P: falta que los circuitos además se puedan contruir en tiempo polinómico.

Sin embargo, no se espera que, limitándonos a trabajar con el superconjunto $P_{/poly}$, esa parte en la que $P_{/poly}$ no se fija nos impida distinguir entre los lenguajes de P y NP, en el caso de que $P \neq NP$. Es cierto que en $P_{/poly}$ hay lenguajes que no están en P, pero, como se prueba en el teorema 6.19 de [1], si estuviesen todos los lenguajes de NP, colapsaría la jerarquía polinómica a su segundo nivel. Lo cual, sin entrar en mucho detalle de lo que significa, no es nada razonable según los expertos. Con lo cual, es más que improbable que

aquellos problemas de NP que desconocemos si están o no en P (por ejemplo los NP-*completos*) estén en $P_{/poly}$ sin estar en P, aunque el teorema no nos dé certeza de ello.

De lo que sí que da certeza el teorema es de que los factores que puedan influir para que un lenguaje esté en $P_{/poly}$ son, en particular, también factores que influyen para que el lenguaje esté en P. Y como hay motivos para pensar que esos factores no deben ser ajenos a las diferencias entre P y NP (en caso de que existan tales diferencias) hay un claro interés por tratar de encontrarlos. Al relajar el estudio de P a $P_{/poly}$ se puede experimentar de una manera más sistematizable y clara utilizando los circuitos y esa es la ventaja que aprovecharemos nosotros en adelante.

1.1.3. Antecedentes directos

Si bien toda esta teoría está ya muy estudiada y trabajada desde el punto de vista teórico, el hecho de tratar de aproximarse al problema P vs NP desde lo experimental, aprovechando que los resultados que hemos presentado permiten trasladarse al campo de la Complejidad de Circuitos, es algo bastante novedoso que tiene como principal antecedente en el trabajo de fin de grado que realizó Enrique Román Calvo en 2020 [5].

Enrique trató de estudiar la endogamia de los circuitos (circuitos diferentes dentro de un nivel que se definen a partir de los mismos circuitos en una o varias profundidades menos), en relación con el tamaño del circuito mínimo que permite computar una función booleana. Para ello, buscó las correlaciones entre los valores de distintas métricas de endogamia y el tamaño de los circuitos que obtuvo a partir de un generador sistemático.

Pese a que sus resultaron no fueron muy claros, su trabajo, que recomendamos enérgicamente leer, sienta las bases sobre la manera de proceder para sumergirse en este tipo de experimentos. Además, pone de relieve las cosas que vale la pena repetir porque dieron buenos frutos y las que quizás es mejor replantearse porque no funcionaron como se esperaba. Desde luego, es una buena referencia, que nos ha resultado muy útil por la cercanía y similitud con nuestra labor, pero de la que hemos tratado de desmarcarnos en los puntos donde parecía haber margen de mejora o, por lo menos, una alternativa clara.

1.2. Líneas generales sobre el desarrollo del trabajo

La idea de nuestro trabajo es construir un dataset que recoja funciones booleanas $f : \{0,1\}^n \to \{0,1\}$ junto al tamaño de un circuito mínimo que las compute. Dicho dataset se usaría después para identificar factores que caractericen a las funciones computables con un tamaño de circuito "pequeño" o "grande" como simplificación necesaria a "polinómico" y "no polinómico". Estos factores podrían extrapolarse para todo $n \in \mathbb{N}$, resultando así característicos de lenguajes de la clase $P_{/poly}$. Por la inclusión $P \subset P_{/poly}$ del teorema 1.2.1 y el colapso de la jerarquía polinómica si $NP \subset P_{/poly}$, estos factores podrían ser necesarios y particularmente relevantes para la pertenencia a la clase P. A continuación aclaramos un poco las simplificaciones.

Para empezar, cada problema en $P_{/poly}$ es un conjunto de palabras de cualquier tamaño n con el alfabeto $\{0,1\}$. Un circuito solo trata entradas de un tamaño n fijo y para aproximarse experimentalmente a esta clase de complejidad, desde los circuitos, es preciso relajar las condiciones de $P_{/poly}$ y rebajarse a un número finito de tamaños n. La definición 1.1.7 de $P_{/poly}$ implica trabajar con cualquier n natural y eso no es posible experimentalmente, de forma que es necesaria una simplificación.

En el momento que nos restringimos a una cantidad finita de tamaños n perdemos la capacidad de hablar de "polinómico" y "no polinómico", pues estas nociones son para todo $n \in \mathbb{N}$. Toda cantidad finita de tamaños de circuito $t_1 = |C_1|, ..., t_m = |C_m|$, es superable, para el tamaño de palabra $n \geq 1$ que reciba cada uno, por la evaluación de un polinomio "suficientemente grande" (basta coger $p(n) = C \cdot n$ con $C = máx(t_1, ..., t_m)$). Luego con un número finito de valores n, todo es polinómico. La simplificación obvia de estos conceptos consiste en referirnos a "pequeño" para "polinómico" y "grande" para "no polinómico", tratando de recoger la esencia de estos conceptos asintóticos mediante una instantánea finita.

Notemos también que, si fijamos un tamaño n para las palabras a decidir, sacar un '0' o sacar un '1' para cada posible palabra (en función de si está o no en el lenguaje) puede expresarse mediante una función $f : \{0,1\}^n \to \{0,1\}$. Una sucesión $\{f_n\}_{n\in\mathbb{N}}$ de tales funciones es, por tanto, equivalente a un lenguaje. Y es con este concepto, el de función booleana, con el que nosotros trabajaremos en adelante.

Una vez descubiertos los mencionados factores se trataría de corroborar que se manifiestan especialmente en la tabla (fijando algún n) de problemas de la clase P, siendo muy interesante compararlos con palabras de lenguajes *NP-completos*. Esto recabaría indicios sobre la posibilidad de diferenciar P de NP, explicitando el porqué de la no coincidencia, lo cual podría ser de gran ayuda en una hipotética demostración formal.

Nuestro primer paso será desarrollar y ejecutar un generador sistemático de circuitos cuyas evaluaciones (definidas en 1.1.3) sean las funciones computables con circuitos "pequeños".

Como hemos comentado, ningún experimento finito puede abordar un número infinito de tamaños y, para cada tamaño n, el número de funciones booleanas distintas $f : \{0,1\}^n \to \{0,1\}$ es doblemente exponencial con dicho tamaño:

Teorema 1.2.1. *[**Cantidad de funciones booleanas**] Hay exactamente 2^{2^n} funciones $f : \{0,1\}^n \to \{0,1\}$ diferentes.*

Demostración. Los *inputs* de la función f consisten en n valores binarios y, por tanto, hay 2^n *inputs* diferentes posibles para ella: $x_0, ..., x_{2^n-1}$. La función la determinan las salidas para dichos *inputs*: $s_0, ..., s_{2^n-1}$ donde $s_i = f(e_i)$. Y como cada salida es binaria ($s_i \in \{0,1\}$ para todo $i = 0, ..., 2^n - 1$), se pueden determinar 2^{2^n} funciones distintas. □

Con lo cual, fijando un tamaño n que no sea ridículamente pequeño, el número de funciones es ya inmensamente grande, y llegar a una cantidad medianamente representativa de las mismas es ya un reto enorme.

La capacidad de computar algunas funciones para circuitos "pequeños" debe basarse en poder explotar ciertas regularidades de dichas funciones, y al menos parte de esas regularidades desconocidas ya deben estar presentes en algunas funciones booleanas para muy pocos bits (correspondientemente ausentes para otras funciones con el mismo número de bits). Por tanto, la opción de fijar un solo valor para n no resultaba descabellada.

Además, si las regularidades misteriosas para computar las funciones con circuitos "pequeños" se hacen más difíciles de definir cuanto mayor es la entrada n, sería factible que se complicasen de forma gradual, por lo que la identificación de estos patrones para pocos bits podría servir de punto de partida para casos mayores.

En todo momento debemos tener presente que los factores que conjeturemos, que afecten a la necesidad de circuitos "grandes", idealmente deberían ser independientes del tamaño n fijado para nuestros experimentos. En el sentido de que, aunque los cuantifiquemos con métricas específicas para ese tamaño, el concepto que miden se pueda extrapolar de manera natural a otros tamaños distintos al elegido. De lo contrario, dichos factores no irán en la línea de distinguir entre necesitar circuitos "grandes" o no como primera aproximación a distinguir $P_{/poly}$ de su complementario, ya que responderán a propiedades concretas del n escogido y, sin embargo, $P_{/poly}$ involucra a todo $n \in \mathbb{N}$.

Se optó por fijar el tamaño de palabra $n = 5$, para el cual el número de funciones booleanas existentes es 2^{32} (algo más de 4 mil millones). Esta cantidad es excesiva para computarse por completo, sobre todo teniendo en cuenta que la manera de encontrar las funciones es recorrer circuitos y ver qué función computan. Lo cual, de manera obvia, provoca que muchas funciones se encuentren repetidas.

Sin embargo, con un tamaño de palabra n menor (con $n = 4$ solo hay unas 65.000 funciones), se corre el riego de confundir las funciones computables mediante circuitos "pequeños" con las que requieren circuitos "grandes". En algún lugar había que fijar la frontera entre lo "pequeño" y lo "grande" y, a efectos prácticos, era más fácil hacerlo en un ambiente donde los tamaños mínimos se extendiesen mucho más. Que una función requiera tamaño 3 para computarse o requiera tamaño 4, es algo demasiado sutil y meramente anecdótico. Lo que queremos es saber qué diferencia a ambas de otra función que requiere tamaño 100 para computarse. Esas diferencias tan notables en el tamaño de circuito mínimo se acentúan aumentando n.

Pensemos en casos límite como $n = 1$, donde solo hay dos funciones distintas, o $n = 2$, donde solo hay 16 funciones. Los circuitos para computarlas tienen tamaños tan parecidos, que por esa vía es prácticamente imposible distinguirlas. La idea es que, según se aumenta el tamaño n, hay más posibles *inputs* con los que poner más "dificultades" a las funciones, lo que obliga a meter más puertas en los circuitos que las computen. Una garantía de ello es el ya mencionado resultado sobre la existencia de funciones que requieren tamaño superior a $2^n/10n$ para computarse, que si n es grande además son casi todas (teorema 6.21 de [1]).

Evaluando para un n demasiado pequeño, un polinomio y una exponencial dividida por un polinomio pueden confundirse.

Nuestro propósito es identificar estas dificultades (o la ausencia de ellas según el caso) apoyándonos en el tamaño (número de puertas) de dichos circuitos, y por ello nos conviene que n sea grande, pero no demasiado, debido a la doble exponencialidad en la cantidad de funciones.

A partir de todo esto, queda clara la importancia de conseguir el máximo número posible de funciones computables con circuitos "pequeños". Dado que no vamos a poder llegar a todas, y llegando a un número insuficiente de ellas, podríamos quedarnos simplemente en las extremadamente triviales, corremos el riesgo de que nuestras conclusiones sean extremadamente obvias. Por tanto, surge la idea de **recorrer crecientemente en tamaño los circuitos**, para que las funciones halladas sean computables con tamaños "pequeños", y la idea de trabajar con **estrategias que permitan saltarse circuitos equivalentes o redundantes**, incrementando con ello la cantidad de funciones diferentes encontradas por el generador.

1.3. Plan de trabajo

Debido a las necesidades propias de un trabajo tan experimental, que debe ir encauzándose a partir de los resultados obtenidos, la hoja de ruta fijada antes de empezar simplemente era un esbozo de lo que se pretendía hacer. La idea era empezar con un estudio preliminar de aspectos teóricos sobre complejidad de circuitos, utilizar el resto del primer cuatrimestre para obtener la tabla de funciones booleanas acompañadas de su tamaño mínimo de circuito y acabar, en el segundo cuatrimestre, elaborando y contrastando con dicha tabla diversas conjeturas que se nos pudieran ocurrir.

Esta hoja de ruta 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[3].
- 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.

[3] También existía la opción de reutilizar o mejorar la tabla y el generador de Enrique (en lugar de desarrollar uno nuevo de 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.

1.4. Objetivos

En base a lo que hemos comentado en esta introducción los objetivos generales que teníamos antes del comenzar el trabajo eran los siguientes:

- Conseguir recorrer crecientemente los circuitos booleanos para 5 bits de entrada de manera sistemática, de forma que las funciones de nuestra tabla resultasen las computables con los circuitos más "pequeños".
- Que la cantidad de funciones diferentes computadas durante la generación de circuitos fuese lo más grande posible, saltando circuitos redundantes y obteniendo, idealmente, más funciones que en la tabla de Enrique a la que teníamos acceso desde el principio.
- Llegar a identificar y entender factores que verdaderamente fuesen muy influyentes en el tamaño mínimo del circuito para computar una función booleana. Y que dichos factores respondiesen a patrones claros fácilmente extrapolables a tamaños superiores.
- Que las métricas para esos factores capturasen adecuadamente su esencia y proporcionasen correlaciones claras con un tamaño "pequeño" o "grande" de circuito.
- Concluir, apoyándonos en la teoría expuesta en esta introducción y en diversos experimentos auxiliares, que los factores conjeturados y cotejados sobre la tabla suponen indicios empíricos (aunque obviamente muy limitados) de que $P \neq NP$.

Asimismo, según avanzábamos en el trabajo, fueron surgiendo nuevos objetivos mucho más concretos que serán comentados con todo detalle a lo largo de esta memoria. Por ejemplo, destacamos el deseo por diseñar métricas invariantes bajo permutaciones que explicaremos en la sección 3.2.

En las conclusiones del trabajo se comentará el balance final sobre los objetivos alcanzados. No obstante, para no dejar al lector en ascuas, anticipamos que el trabajo dio muy buenos resultados y todo lo que hemos listado lo damos por cumplido.

Capítulo 2

Generador de circuitos

En este capítulo vamos a explicar el algoritmo generador de circuitos que se desarrolló y ejecutó para obtener la tabla que ya mencionamos en la introducción. Justificaremos las ventajas de este algoritmo frente a otras alternativas y comentaremos también sus inconvenientes, que los tiene. Para acabar, daremos detalles de su implementación y haremos una pequeña valoración de los resultados obtenidos.

2.1. Objetivos del algoritmo

Como ya comentamos en las líneas generales del trabajo, era muy interesante que las funciones computadas por los circuitos recorridos en el algoritmo fuesen las que requerían menor tamaño. Por ello, resultaba especialmente importante **recorrer los circuitos crecientemente**. Otro interés que destacamos era el de llegar a alcanzar también funciones que no fuesen totalmente triviales, tratando de maximizar la cantidad a la podíamos llegar en un tiempo de ejecución razonable. Para ello, era importante conseguir que **aflorasen muchas funciones diferentes lo antes posible**, siendo capaces de **saltarnos circuitos trivialmente equivalentes** para no perder tiempo de cómputo en ellos.

A estos deseos, se unió la exigencia de que nuestro algoritmo fuese **completo**. Lo que significa que, para toda función, en algún momento **es capaz de encontrar un circuito de tamaño mínimo que la compute**. De esta forma, en una hipotética ejecución suficientemente larga, el algoritmo podría alcanzar las 2^{32} funciones que hay en total. Y además, de forma óptima, ya que, aunque deseablemente se dejaría muchos circuitos por el camino, existiría algún tipo de garantía de que estos no son mínimos para ninguna función.

Uniendo esta completitud a la forma creciente de recorrer los circuitos, podríamos afirmar que si el algoritmo se detuviera durante la generación de los circuitos de tamaño n, en nuestra tabla estarían **todas las funciones computables con tamaño** $< n$. Y además, tendríamos el dato de que todas aquellas funciones que no estuviesen en nuestra tabla requerirían tamaño $\geq n$.

El *algoritmo del índice* de Enrique [6] no satisfacía nuestros deseos. El recorrido no era creciente en tamaño, y la cantidad de funciones diferentes que lograba encontrar era paupérrima en relación a la de circuitos generados: unas 680.000 funciones para más de 13 mil millones de circuitos. Además, no parecía fácil modificarlo para acercarlo a nuestras pretensiones, de manera que se decidió desarrollar un algoritmo totalmente nuevo desde cero.

2.2. Idea del algoritmo

La idea de cómo obtener un algoritmo que recorra los circuitos crecientemente en tamaño es muy simple: **todo circuito de tamaño** $n > 1$ **es el resultado de combinar circuitos de tamaño** $< n$. Simplemente es necesario afinar un poco esta idea, para terminar de definirla y conseguir añadir el resto de requisitos que nos habíamos marcado.

Primero, para darnos cuenta de que la afirmación es cierta, basta pensar que, tomando un circuito C tal que $|C| > 1$, si le quitamos su puerta sumidero (y también las aristas que llegan a ella), obtenemos grafos que cumplen la definición 1.1.1 de circuito y que, como poco, tienen una puerta menos. Llamaremos a estos circuitos: *circuitos padres* del *circuito hijo* C.

Por tanto, supuesto que tenemos todos los circuitos de tamaño $< n$, si definimos las combinaciones de forma que con los padres podamos reconstruir a cierto hijo suyo, haciendo todas las combinaciones posibles entre ellos podemos obtener todos los circuitos de tamaño n. Por inducción, siguiendo esta mecánica, basta disponer de los circuitos degenerados de tamaño 0 para lograr obtener, en orden creciente de tamaños, todos

los circuitos posibles. Pero, estos circuitos degenerados simplemente consisten en los 5 circuitos que devuelven las 5 entradas posibles y, por tanto, se pueden meter "a mano" para arrancar el algoritmo.

Una de las ampliaciones de la definición de circuito 1.1.1, que avisábamos en el capítulo anterior, tiene que ver con considerar a los circuitos de m vértices fuente como circuitos de n entradas siempre que $m \leq n$. Según la definición 1.1.1, un circuito de n entradas tiene exactamente n vértices fuente, pero, ¿acaso no es el circuito de la figura 1.1.1 también de 5, 6 o 7 entradas? Simplemente no hace uso de todas ellas, pero nosotros, que trabajaremos con $n = 5$, necesitamos entender que el circuito de la figura 1.1.1, y muchos otros, son circuitos legítimos de 5 entradas que "desechan" algunas de ellas (el de la figura 1.1.1 no usa la entrada e_4).

2.2.1. Combinación de circuitos

Lo siguiente que deberíamos definir es cómo combinar circuitos entre sí para reconstruir a los hijos a partir de los padres. Ya hemos dicho que el padre es el resultado de quitar la puerta sumidero del hijo. Por tanto, combinar circuitos debería consistir en conectarlos a un puerta final que ejerza de tal sumidero. Para ello, debemos considerar todas las puertas posibles, ya que, por ese motivo, no tenemos razones para saltarnos circuitos hijos.

La puerta *NOT*, como es unaria, debe ponerse como sumidero de todos los circuitos de tamaño $< n$. Mientras que las puertas *AND* y *OR*, como son binarias, deben ponerse como sumidero de todos los pares de estos circuitos.

Sin embargo, algo fundamental, que no debemos pasar por alto, es que los circuitos padres C_1 y C_2 que resultan de quitarle el sumidero a cierto circuito C, puede tener subcircuitos en común que se reutilicen en el hijo. Esto sucede en el circuito C del ejemplo inicial 1.1.4, cuyo subcircuito marcado a continuación en verde, está repetido en los padres C_1 y C_2, pero se utiliza una sola vez en C:

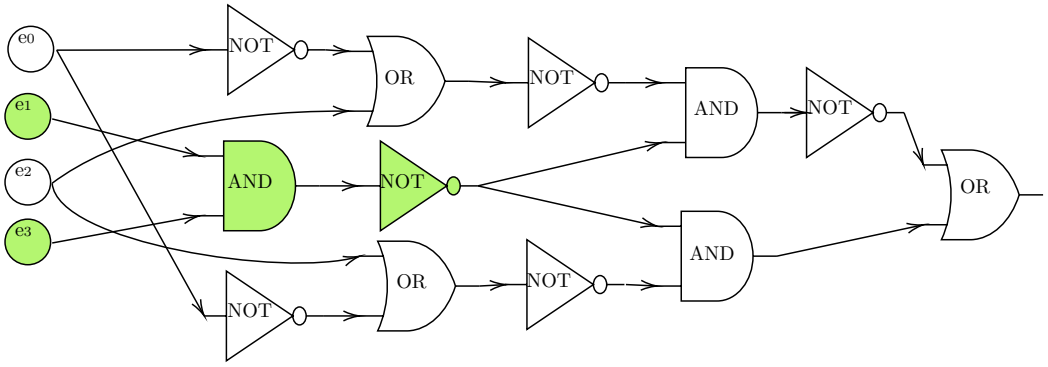


Figura 2.2.1: Circuito C destacando en verde un subcircuito repetido en los padres.

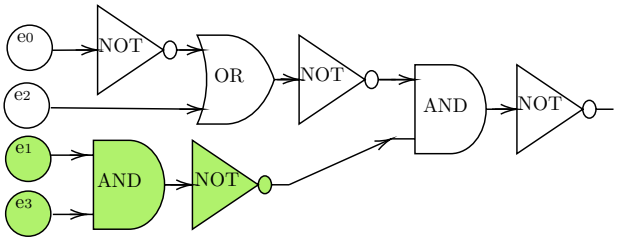


Figura 2.2.2: C_1 y un subcircuito común con C_2.

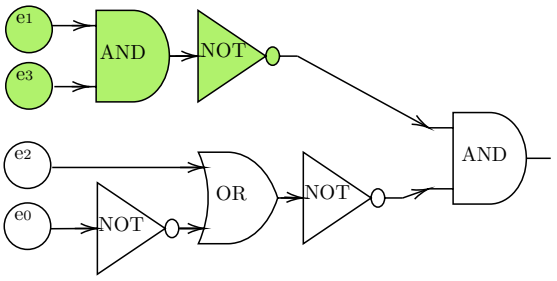


Figura 2.2.3: C_2 y un subcircuito común con C_1.

Y por tanto, la combinación de los padres C_1 y C_2 debería fusionar dicho circuito común para reconstruir al hijo. No obstante, en el mismo ejemplo que estábamos comentando, podemos apreciar que sendos circuitos

padres poseen otro subcircuito en común (el circuito C_2) que no aparecen fusionado en su totalidad en el hijo:

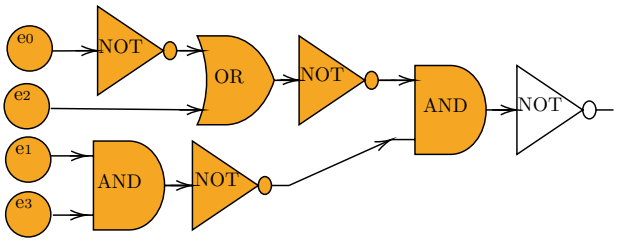


Figura 2.2.4: C_1 y otro subcircuito común con C_2.

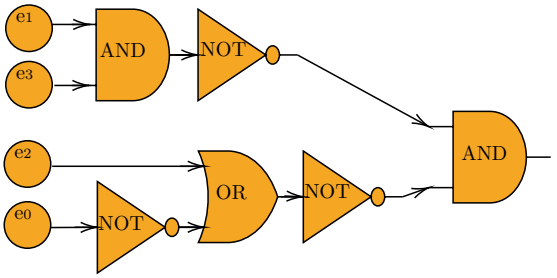


Figura 2.2.5: C_2 y otro subcircuito común con C_1.

Nótese que los subcircuitos señalados en naranja en las figuras 2.2.4 y 2.2.5 son el mismo. Aunque intencionadamente se han representado intercambiando el orden en que una puerta *OR* y una puerta *AND* reciben sus entrada, son el mismo grafo.

Esto pone de manifiesto que debemos ser capaces de identificar estos subcircuitos repetidos para tratarlos adecuadamente al combinar. Siendo para ello necesario que no existan circuitos iguales cuya detección requiera de averiguar si dos grafos son isomorfos o no. (El isomorfismo de grafos es un problema bastante complicado, y aunque nuestros circuitos van a ser pequeños, la operación de combinarlos debería ser poco costosa).

Para que no aparezcan estos subcircuitos, aparentemente diferentes, que en realidad son iguales, bastará con evitar combinaciones simétricas. Esto significa que, para cada par de circuitos C_1 y C_2 que combinemos con una puerta sumidero binaria P, si consideramos que C_1 es el argumento izquierdo de la puerta y C_2 es el argumento derecho (lo denotaremos como $P(C_1, C_2)$), no consideraremos la combinación cruzada $P(C_2, C_1)$. Lo cual, por construcción, garantiza que en una situación como la que resaltábamos anteriormente es trivial identificar la igualdad entre los subcircuitos, ya que la representación que tendremos de los mismos será literalmente idéntica.

Por otra parte, para gestionar la fusión de los subcircuitos repetidos, que ya somos capaces de identificar con facilidad, rescatamos la idea de que no queremos generar todos los circuitos posibles, sino solo aquellos que puedan ser de tamaño mínimo para alguna función. El circuito C de la figura 2.2.1 no puede ser mínimo para la función que computa, por el simple hecho siguiente: fusionando todas las puertas del subcircuito común que resaltamos en las figuras 2.2.4 y 2.2.5, se obtiene un circuito equivalente que tiene estrictamente menor tamaño.

Por tanto, surge de manera obvia la idea de que al combinar dos circuitos C_1 y C_2 hay que fusionar todos sus subcircuitos comunes. Por ejemplo, al combinar de esta forma los circuitos padres C_1 y C_2 de las figuras 2.2.4 y 2.2.5 con una puerta *OR* como sumidero, obtendríamos el circuito:

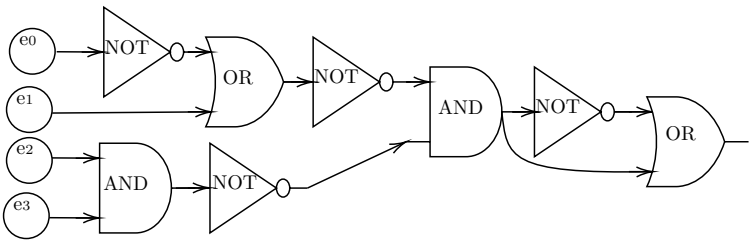


Figura 2.2.6: Circuito combinación de C_1 y C_2 fusionando subcircuitos.

Dicho circuito trivialmente computa la misma función que el circuito C de la figura 2.2.1, pero con tan solo 8 puertas, frente al tamaño 12 que vimos que tenía el circuito C en el ejemplo 1.1.4.

Además, es importante destacar que, simplemente considerando la mencionada fusión de subcircuitos comunes a los padres, se tiene la total certeza de que el circuito hijo carece de repetidos. Un repetido en el circuito hijo necesariamente implicaría un repetido en alguno de los padres. Todos los subcircuitos del hijo son también subcircuitos de alguno de los padres (a excepción del circuito hijo completo) y la repetición en el hijo no podría provenir de una sola aparición en cada padre debido a la fusión que realizamos. Por construcción, todos los circuitos, salvo los casos base de tamaño cero, han sido generados mediante combinaciones y, como estos casos base carecen de repetidos, por lo que acabamos de justificar, la ausencia de repetidos debe propagarse inductivamente a todos los demás (si no se propagara, tendríamos un primer caso con repetidos para algún hijo, sin repetidos en sus padres, lo cual es contradictorio).

Como los circuitos que tienen subcircuitos repetidos tampoco pueden ser mínimos para ninguna función, es bastante tranquilizador saber que nuestra manera de recorrer los elude por completo.

2.2.2. Tamaño y evaluación del circuito hijo

Una vez explicada la mecánica de combinación de circuitos, cuya única particularidad es la fusión de subcircuitos comunes a los padres, es fácil calcular el tamaño del circuito hijo a partir del tamaño de los padres. De nuevo, como el tamaño de los casos base lo conocemos (es 0), según generamos los circuitos, podemos ir calculando su tamaño, siendo esto, junto a la función que computan, lo que es imprescindible anotar en la tabla de salida.

Las puertas del circuito hijo son las de los padres junto al nuevo sumidero que se coloca "al final", teniendo cuidado de contar una sola vez las puertas de los padres que se fusionan. Si la puerta final es unaria (por ejemplo, una NOT), el tamaño del hijo es sumar uno al del padre. Y si es binaria, es la suma de los tamaños de los dos padres más uno (debido al sumidero), restando las puertas que se fusionan y se han contado dos veces[1].

No obstante, hay que tener cierto cuidado para que cada una de esas puertas se reste una sola vez, ya que, en general, aparecerán en varios subcircuitos comunes a los padres. Si pensamos en un subcircuito común a los padres con varios niveles de puertas, por ejemplo el señalado en naranja en las figuras 2.2.4 y 2.2.5, quitando su sumidero obtenemos subcircuitos también comunes a los padres que repiten puertas con él:

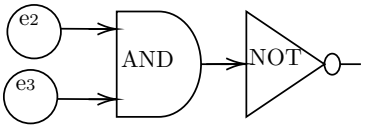


Figura 2.2.7: Subcircuito de subcircuito común de C_1 y C_2.

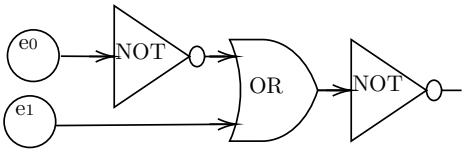


Figura 2.2.8: Subcircuito de subcircuito común de C_1 y C_2.

En caso de que, debido a los subcircuitos de las figuras 2.2.7 y 2.2.8, restásemos 5 puertas (las que tienen en total), estaríamos restando puertas de más. Debido al subcircuito que los contiene, y que es también común a los padres, esas puertas ya habrían sido restadas, a pesar de que solo se fusionen una vez.

Para controlar este problema surgen dos opciones. La primera es restar puertas solo debido a los *subcircuitos comunes maximales*: aquellos que no son a su vez subcircuitos de otro que sea común a los padres. Esto teóricamente es bastante claro (solo restamos cuando no hay nada "por encima" en común), pero implica invertir cierto tiempo de cómputo en calcular dichos maximales. Recordemos que la operación de combinación de circuitos se va a hacer, en principio, miles de millones de veces. Por tanto, un mejor opción es que, por cada subcircuito común, restemos solo uno, debido a la fusión de su puerta sumidero.

Es bastante claro que dos subcircuitos diferentes C_j y C_k, que sean comunes a los padres, no pueden tener la misma puerta sumidero. Por supuesto que el sumidero v de ambos puede ser la misma puerta lógica, en cuanto a tener la misma etiqueta, pero no la misma puerta del circuito, en cuanto a ser el mismo vértice. De lo contrario, simplemente recorriendo "hacia atrás" desde v en cualquiera de los padres, podríamos reconstruir C_j y C_k, en un proceso determinista, con lo cual deberían coincidir.

[1] Nótese que se trata de una adaptación a nuestro contexto de la *fórmula de inclusión-exclusión* para dos conjuntos.

También es claro que toda puerta v que se fusione es sumidero de algún subcircuito común a C_1 y C_2. Si se fusiona, es porque forma parte de algún subcircuito C' común a los padres. Y recorriendo desde ella "hacia atrás" en el grafo, obtendríamos un subcircuito de C' del que sería sumidero. Como un subcircuito de subcircuito común lo es también, la afirmación es cierta.

Por tanto, la cantidad de puertas que se fusionan y de sumideros de subcircuitos comunes coincide. Basta, por tanto, restar el número de subcircuitos comunes que tienen sumidero (los de tamaño 0 hay que excluirlos) para obtener la **fórmula del tamaño de una combinación**:

$$|P(C_1, C_2)| = |C_1| + |C_2| + 1 - Card(\{C \mid C \text{ es subcircuito de } C_1 \text{ y de } C_2 \text{ con } |C| \geq 1\}) \tag{2.2.6}$$

Por otra parte, hay que destacar que esta estrategia de combinar circuitos nos permite obtener la función que computa el hijo, a partir de las computadas por los padres, sin necesidad de recorrer los circuitos (la manera trivial sería aplicar la definición de evaluación 1.1.3 como ejemplificamos en 1.1.4 *input* a *input*, pero esto supone recorrer el circuito varias veces).

Notemos que la propagación hacia adelante de los valores según la lógica de las puertas, recogida en la definición 1.1.3, ya la conocemos, salvo por el último nivel (la puerta sumidero). La propagación anterior al sumidero es la correspondiente a los padres y, por tanto, basta aplicar *input* a *input* la operación lógica del sumidero sobre el valor que tenga en los padres. Se obtiene así la función que computa el circuito hijo.

Por ejemplo, para un *input* concreto $x = 11010$, suponiendo que $C_1(11010) = 1$ y que $C_2(11010) = 0$, se tiene que:

$$\text{OR}(C_1, C_2)(11010) = \text{OR}(C_1(11010), C_2(11010)) = \text{OR}(1, 0) = 1.$$

En general, para los 32 *inputs* posibles, la función computada por el hijo se calcula, uno a uno, como acabamos de comentar, es decir:

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_1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	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_3	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	1	0	1	0	1	1	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	1	0	0	1	0	1	0	1
f_2	1	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	0	0
OR(f_1,f_2)	1	1	1	1	0	1	1	0	0	0	0	1	0	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	1	1	0	1

Tabla 2.1: Función computada por el circuito hijo con sumidero OR cuyos padres computan f_1 y f_2

En esta tabla podemos apreciar los 32 posibles *inputs* ordenados decrecientemente de izquierda a derecha si consideramos que representan a números binarios: del $(11111)_2 = (31)_{10}$ al $(00000)_2 = (0)_{10}$. Y vemos proyectada la salida para cada uno de esos *inputs* de las funciones f_1 y f_2 que calculan los padres, así como de la función $OR(f_1, f_2)$ que calcula un hijo suyo cuyo sumidero sea una puerta OR. Esto forma una cadena de 32 bits con lo que trabajaremos habitualmente en adelante, que será una manera de representar a las funciones. Diremos, por tanto, que el circuito hijo computa la función 11110110000100110100110010111101, que es el resultado de leer lo que aparece en la tabla 2.1 de izquierda a derecha (de mayor a menor *input*).

Sin embargo, por motivos de eficiencia y sencillez, la representación interna de las funciones que utilizaremos en el algoritmo consistirá en el número decimal asociado a la cadena de 32 bits que acabamos de explicar, si la vemos a su vez como un número binario. En el ejemplo anterior la función 11110110000100110100110010111101 se codificaría como el entero 4128459965 ya que $(11110110000100110100110010111101)_2 = (4128459965)_{10}$.

2.2.3. Esquema de realización de las combinaciones

Otro aspecto que deberíamos cuidar tiene que ver con no realizar las mismas combinaciones más de una vez, haciéndolas exactamente en el momento adecuado. Lo ideal sería que en la etapa n-ésima, nuestro algoritmo generase los circuitos que tienen exactamente tamaño n. Así, etapa a etapa, podríamos hacer la generación de circuitos creciente en tamaño que nos habíamos propuesto al principio.

Desgraciadamente, esto no es posible debido a que la fusión de puertas entre subcircuitos comunes a los padres provoca que el tamaño del hijo sea impredecible usando exclusivamente el tamaño de los padres. Si bien la fórmula 2.2.6 nos permite calcular el tamaño del hijo a partir del tamaño de los padres, también interviene la cantidad de circuitos comunes a ellos, cuyo cálculo es el único cómputo necesario para la generación del circuito hijo. Por tanto, saber qué tamaño tendrá el hijo es literalmente equivalente a generarlo. Y esto impide que la generación se haga una sola vez obteniéndose nada más que los circuitos del tamaño n por el que vamos.

La idea es que, como tenemos que generar el circuito para saber su tamaño, aunque este no sea el deseado, almacenamos el circuito indexándolo por el tamaño obtenido, de forma que lo podamos recuperar después cuando vayamos por la etapa correspondiente a dicho tamaño. De esta forma, simplemente evitando volver a combinar los circuitos que ya se combinaron en etapa pretéritas, generaríamos cada circuito exactamente una vez.

En la etapa 0, el algoritmo arrancará con los 5 circuitos de tamaño 0 que le metemos "a mano", los combinará individualmente con la puerta sumidero NOT y por pares con las puertas sumidero OR y AND, y obtendrá así todos los circuitos de tamaño 1[2].

En la etapa 1 el algoritmo combinará los circuitos nuevos de tamaño 1, individualemente con una NOT y con puertas binarias con todos los generados hasta el momento (tamaños 0 y 1), obteniendo así todos los circuitos de tamaño 2 (todo circuito de tamaño 2 es combinación de otros de tamaño < 2 como ya justificamos). Pero también obtendrá algunos circuitos de tamaño 3, debido a combinaciones entre circuitos de tamaño 1 sin subcircuitos comunes, que según la fórmula 2.2.6 tienen tamaño $1+1+1=3$. Estos circuitos los guardará en una tabla indexados por el valor 3 para después utilizarlos.

En la etapa 2 ocurrirá algo similar a lo que sucedió en la etapa 1, pero al combinar los circuitos nuevos de tamaño 2 con los de tamaños menores (tamaños 0, 1 y 2), obtendrá otros circuitos de tamaño 3, que unidos a los que se almacenaron indexados con este tamaño en etapas anteriores, suponen todos los circuitos de tamaño 3. La justificación es que, llegados a este punto, se han combinado todos los circuitos de tamaño < 3, de todas las formas posibles, exactamente una vez. Adicionalmente, generará algunos circuitos de tamaños 4 y 5, que indexará correspondientemente en una tabla para futuras etapas.

Además, como ya justificamos, el algoritmo puede ir calculando las funciones que computan los circuitos hijos, a partir de la función de los padres. Sin embargo, de cara a anotar en la salida un tamaño de circuito como mínimo para una función, no es correcto anotar el primer tamaño de circuito que el algoritmo encuentre para ella, ya que, como acabamos de explicar, se generan circuitos de tamaños 5 antes de tener todos los de tamaño 4. En general, se tienen circuitos de tamaño m sin tener todos los de tamaño $n < m$, así que haber anotado el valor m como tamaño mínimo para una función nueva podría resultar incorrecto al descubrir después un circuito de tamaño n que la compute.

Lo que sí es seguro es que en la etapa n-ésima se tienen, combinando como hemos explicado, todos los circuitos de tamaño n. Por tanto, no hay riesgo alguno en anotar en la salida las funciones nuevas que estos aporten con tamaño n, ya que las etapas anteriores habrán hecho lo propio, y por ser la función necesariamente nueva, requiere de este tamaño mínimo. Es decir, que en cada etapa no se anotan en la salida las funciones nuevas de los circuitos generados, sino las funciones nuevas de los circuitos que se recogen por completo de la etapa anterior.

En general en la etapa n-ésima:

1. Se anotan en la salida las funciones nuevas computadas por circuitos de tamaño n.
2. Se combinan todos los circuitos indexados con tamaño n, con todos los circuitos indexados con tamaños $\leq n$, mediante puertas binarias (AND y OR). Para cada uno:
   - Se calcula y almacena la función que computa a partir de la función de los padres.
   - Se calcula su tamaño según la fórmula 2.2.6.
   - Se almacena indexado con el tamaño calculado.

[2]Cuando decimos "todos los circuitos" nos referimos a todos los circuitos sin subcircuitos repetidos, que son los que nos interesan. Ya comentamos que al fusionar puertas nos saltamos, de manera trivial, circuitos equivalentes a otro más pequeño.

3. Se combinan todos los circuitos indexados en la tabla con tamaño n utilizando puertas unarias como sumidero (NOT). Para cada uno:
   - Se calcula y almacena la función que computa a partir de la función de los padres.
   - Se almacena indexado con tamaño $n+1$.

2.2.4. Parada del algoritmo

El último escollo a evitar tiene que ver con cómo gestionar el momento en que el algoritmo pare de generar circuitos. En principio, para controlar la cantidad de circuitos generados, al comenzar se introducirá una valor que indique cuántos circuitos va a generar el algoritmo, llevando un contador internamente para saber exactamente cuándo parar (casi con certeza dejando una etapa a medias sin terminar).

El problema es qué hacer con todos los circuitos de tamaños superiores a la etapa de parada, que deberían ser recogidos y utilizados para la salida por etapas posteriores, que ya no se van a ejecutar. ¿Se tiran esos circuitos?

Esa cantidad de circuitos podría ser incluso superior a la de todas las etapas anteriores. Por tanto, no utilizarlos supondría que casi todo el trabajo de cómputo habría sido en vano. Con lo cual, una vez se detiene la generación del algoritmo deberíamos hacer un post-procesamiento anotando las funciones nuevas que estos circuitos aporten en la salida.

El principal inconveniente es que el tamaño asociado a estas funciones en la salida no tiene por qué ser mínimo debido a lo que comentamos anteriormente. Faltan circuitos de tamaños inferiores aún por explorar. Pero, ¿pueden ser muchos más grandes los que nosotros anotemos en la salida frente a los mínimos reales?

No, pues debido a la fórmula 2.2.6, el tamaño de los circuitos generados en la n-ésima etapa $\in \{n+1, n+2, ..., 2n+1\}$. Las puertas que se fusionan (que vimos que están en biyección con el conjunto de circuitos cuyo cardinal restábamos) no pueden superar el tamaño de ninguno de los circuitos que se combinan ($\leq n$) y hay un número no negativo de ellas (como poco no se fusiona ninguna). Como, además, según el esquema de combinaciones descrito anteriormente, al menos uno de los dos circuitos que se combinan tiene tamaño n (y el otro $\leq n$), podemos deducir lo siguiente:

Asumiendo, sin pérdida de generalidad, que el circuito de tamaño n es C_1, y denotando $S = Card(\{C \mid C$ es subcircuito de C_1 y de C_2 con $|C| \geq 1\})$, se tiene que:

$$n+1 = n+1+0 \leq n+1+(|C_2|-S) = |C_1|+|C_2|+1-S = n+|C_2|+1-S \leq n+n+1-0 = 2n+1$$

Con lo cual, aunque no es lo más deseable, el inconveniente que comentamos no es tan grave. El tamaño anotado no puede exceder al doble más uno del tamaño óptimo. Y dado que nuestro generador, para una cantidad realista de circuitos a generar, no alcanzará circuitos muy grandes (precisamente el generador pretende proporcionar una muestra significativa de lo computable con circuitos pequeños), la diferencia entre n y $2n+1$ (que podría darse en el caso peor) no debería de ser muy grande.

Es decir, si nuestro algoritmo se detiene en la etapa n quedarían anotadas:

- Todas las funciones computables con tamaño $\leq n$ junto al tamaño exacto de un circuito mínimo (y de hecho junto al tal circuito).
- Muchas funciones computables con tamaños mínimos de circuito $\in \{n+1, n+2, ..., 2n+1\}$ (que no todas) junto a un tamaño de circuito mínimo que puede o no ser el mínimo, pero que a lo sumo dista n del tamaño mínimo.

A todos los efectos, como motivamos en la introducción, que una función pueda computarse con tamaño 7 o tamaño 10 es hilar demasiado fino en nuestro estudio y nos da un poco igual. Lo importante es entender lo que diferencia a ambas de otra función que requiera tamaño mínimo 100. Por tanto, lo que hace nuestro algoritmo sobre estos circuitos una vez se detiene la generación, es asumible.

2.3. Implementación del generador

El generador fue implementado en C++ por dos motivos fundamentalmente. Por una parte, este lenguaje permite al programador gestionar de forma muy versátil el uso de la memoria mediante punteros, lo cual era especialmente interesante para representar los circuitos minimizando el espacio de memoria ocupado y liberando espacio nada más fuese posible. Cada circuito fue representado con una estructura que contenía punteros a sus circuitos padres, su tamaño, su evaluación y su puerta sumidero. En todo momento se usaron referencias a estas estructuras para evitar cualquier copia. Por otra parte, se escogió C++ por velocidad frente a otros lenguaje de alto nivel más lentos como Java o Python.

Para aprovechar aún más todas las capacidades de cómputo de la máquina donde se iba ejecutar el algoritmo, se desarrolló una versión concurrente que ejecuta de forma paralela la generación de circuitos de cada etapa y el volcado final de circuitos una vez detenida la generación.

Vamos a mostrar y comentar algunas partes clave de la implementación del generador. La implementación concreta para el repertorio completo de puertas puede encontrarse en [10]. Explicaremos después qué significa lo del repertorio completo de puertas, pero los conceptos generales de implementación son los mismos que con el repertorio {AND, OR, NOT} que hemos usado hasta ahora.

La función *main* que se ejecuta en el algoritmo es la siguiente:

```
int main() {
   std::cout << "Numero de circuitos a generar:\n";
   std::cin >> maxcircuitos;
   unsigned t0, t1;
   t0 = clock();
   std::string nombreFichero;
   std::cout << "Nombre del fichero de salida:\n";
   std::cin >> nombreFichero;
   fs = std::ofstream(nombreFichero);
   inicializa();
   int tamano = 0;
   while (!pararAlgoritmo) {
      generaCircuitos(tamano);
      anotaCircuitos(tamano);
      quitaCircuitosEspera(tamano);
      tamano++;
   }
   std::cout << "Circuitos generados exhaustivamente hasta size " << tamano - 1 << '\n';
   muestraSobrantes();
   std::cout << "\n";
   std::cout << "Procesando... \n";
   anotaSobrantes();
   std::cout << "Total de funciones distintas computadas " << almacen.size() +
    computadasSobrantes << '\n';
   fs << "Total de funciones distintas computadas: " << almacen.size() +
    computadasSobrantes << '\n';
   liberaAlmacen();
   t1 = clock();
   double time = (double(t1 - t0) / CLOCKS_PER_SEC);
   fs << "Tiempo de ejecucion: " << time << "\n";
   fs.close();
   return 0;
}
```

De ella, debemos destacar:

- Al empezar, la petición al usuario del máximo de circuitos que determina cuándo parar. Se lee en una variable global que se modificará al generar un nuevo circuito (debe ser global porque será compartida por varios procesos que se ejecutan en paralelo).
- La llamada a la función *inicializa* para introducir los circuitos de tamaño 0 y poder arrancar la generación.
- El bucle *while* que realiza las etapas de generación de circuitos, dentro del cual:
  - La función *generaCircuitos* genera los circuitos del tamaño que recibe.
  - La función *anotaCircuitos* escribe en la salida las funciones computadas por los circuitos del tamaño que recibe que no estuviesen ya computadas.
  - La función *quitaCircuitosEspera* simplemente libera espacio de memoria de una de las dos tablas donde se llevarán los circuitos indexados.
  - El proceso que genere el último circuito pondrá a *true* el valor de *parar Algoritmo* que también tiene que ser una variable global.
- La función *anotaSobrantes* se encarga de anotar en la salida aquellos circuitos que cuando se detuvo el algoritmo estaban preparados para considerarse en etapas posteriores.

La introducción de los circuitos degenerados de tamaño 0 que hace *inicializa* no tiene más misterio que saber cuáles son las funciones que estos computan. En la tabla 2.1 ya aparecían estas funciones (las de los circuitos $e_0, e_1, ..., e_4$) como cadena binaria de 32 bits. Basta pasar esas cadenas a decimal:

Circuito	Función en decimal	Función en decimal
e_0	10101010101010101010101010101010	2863311530
e_1	11001100110011001100110011001100	3435973836
e_2	11110000111100001111000011110000	4042322160
e_3	11111111000000001111111100000000	4278255360
e_4	11111111111111110000000000000000	4294901760

de donde surge la explicación a esta función ('C' tiene el significado de que estos circuitos no tienen puerta sumidero):

```
void inicializa() {
   espera[0] = std::vector<circuito*>();
// circuito(padre1, padre2, funcion, tamano, puerta)
   circuito* circ = new circuito(nullptr, nullptr, 2863311530, 0, 'C');
   espera[0].push_back(circ);
   circ = new circuito(nullptr, nullptr, 3435973836, 0, 'C');
   espera[0].push_back(circ);
   circ = new circuito(nullptr, nullptr, 4042322160, 0, 'C');
   espera[0].push_back(circ);
   circ = new circuito(nullptr, nullptr, 4278255360, 0, 'C');
   espera[0].push_back(circ);
   circ = new circuito(nullptr, nullptr, 4294901760, 0, 'C');
   espera[0].push_back(circ);
}
```

La función más importante, que genera los circuitos de una etapa n que recibe como parámetro es la siguiente:

```
void generaCircuitos(short n) {
   if (!espera.count(n)) return;
   size_t numProcesadores = std::thread::hardware_concurrency();
   size_t tamProceso = (espera[n].size() / numProcesadores);
   std::vector<std::thread> hilos(numProcesadores);
   for (size_t i = 0; i < numProcesadores; ++i) {
      if (i < numProcesadores - 1) hilos[i] = std::thread(tareaGeneraCircuitos, n, i *
    tamProceso, (i + 1) * tamProceso);
      else hilos[i] = std::thread(tareaGeneraCircuitos, n, i * tamProceso, espera[n].size
    ());
   }
   for (size_t i = 0; i < numProcesadores; ++i) hilos[i].join();
}
```

El procedimiento *generaCircuitos* divide la cantidad de circuitos de tamaño n, que están en la tabla de *espera*, en tantas partes como núcleos lógicos tenga la máquina de ejecución. Estos circuitos se deben combinar entre sí y con todos los de tamaños inferiores (los de la tabla *almacen*). Para ello, se lanza un hilo de ejecución encargado de hacer las combinaciones de cada parte, que trabaja en paralelo con los demás hilos cuya labor es independiente. Todos hilos se sincronizan, esperando al terminar su trabajo a que acaben todos los demás hilos, para evitar que empiece la siguiente etapa sin haberse acabado por completo la actual, lo que rompería el buen funcionamiento del algoritmo.

El trabajo de cada hilo consiste en una función que, al tener más líneas de código, dejamos que se vea en [10]. De esta función es importante destacar la existencia de 2 bucles con bucles anidados a su vez:

- Un bucle para combinar los circuitos del intervalo de la partición que le toca al correspondiente hilo, utilizando puertas binarias como sumidero, dentro del cual:
  - Hay un bucle anidado para combinar con todos los circuitos del mismo tamaño.
  - Hay otro bucle anidado para después hacer lo propio con todos los circuitos del *almacen* que tienen tamaños inferiores.
- Un bucle para combinar los circuitos del intervalo que le toque al correspondiente hilo, utilizando una puerta unaria NOT como sumidero.

También es importante apreciar que, debido a la concurrencia, el acceso y modificación de las tablas compartidas *espera* y *almacen*, así como algunas variables de sincronización (*parar Algoritmo* o *circuitosGenerados*) se tienen que hacer en exclusión mutua, lo que se ha controlado con un cerrojo de la biblioteca estándar de C++.

En esta tarea de generar circuitos se usan las operaciones de combinación y evaluación de circuitos. Como la evaluación es muy obvia se deja al lector que la vea en [10]. Por su parte, la combinación se implementó como sigue:

```
void introduceEnSet(std::set<circuito*>& set, circuito* circ) {
   if (circ != nullptr) {
      set.insert(circ);
      introduceEnSet(set, circ->padreIzq);
      introduceEnSet(set, circ->padreDer);
   }
}
int combinar(circuito* circuito1, circuito* circuito2) {
   int tam = circuito1->tam + circuito2->tam;
   std::set<circuito*> set1, set2;
   introduceEnSet(set1, circuito1);
   introduceEnSet(set2, circuito2);
```

```
   std::set<circuito*> intersect;
   std::set_intersection(set1.begin(), set1.end(), set2.begin(), set2.end(), std::
    inserter(intersect, intersect.begin()));
   for (circuito* circ : intersect) {
      if (circ->tam > 0) tam--;
   }
   return tam;
}
```

La función *combinar* se encarga de aplicar la fórmula 2.2.6. Inicializa la variable *tam* a la suma de los tamaños de *circuito*1 y *circuito*2 y le resta el cardinal de los subcircuitos comunes a los padres que tengan tamaño > 0. Para ello, se apoya en la función auxiliar *introduceEnSet* que, mediante recursión, acumula los subcircuitos de un circuito dado. Sobre ellos se hace después la intersección.

Finalmente, veamos la función *anotaSobrantes* que anota en la salida aquellos circuitos que, cuando se detuvo el algoritmo, estaban preparados para considerarse en etapas posteriores:

```
void tareaAnotaSobrantes(std::vector<circuito*> const& v, int ini, int fin) {
   for (int i = ini; i < fin; ++i) {
      circuito* circ = v[i];
      mtxGenerador.lock();
      if (!almacen.count(circ->eval)) {
         almacen[circ->eval] = std::vector<circuito*>();
         computadasSobrantes++;
         fs << circ;
      }
      mtxGenerador.unlock();
      delete circ;
   }
}

void anotaSobrantes() {
   std::vector<std::pair<int, std::vector<circuito*>>> sobrantes(espera.size());
   std::transform(espera.begin(), espera.end(), sobrantes.begin(), [](auto pair) {return
    std::move(pair); });
   espera.clear();
   sort(sobrantes.begin(), sobrantes.end(), [](auto pair1, auto pair2) {return pair1.
    first < pair2.first; });
   for (auto par : sobrantes) {

      int numProcesadores = std::thread::hardware_concurrency();
      int tamProceso = par.second.size() / numProcesadores;
      std::vector<std::thread> hilos(numProcesadores);
      for (int i = 0; i < numProcesadores; ++i) {
         if (i < numProcesadores - 1) hilos[i] = std::thread(tareaAnotaSobrantes, par.
    second, i * tamProceso, (i + 1) * tamProceso);
         else hilos[i] = std::thread(tareaAnotaSobrantes, par.second, i * tamProceso, par
    .second.size());
      }
      for (int i = 0; i < numProcesadores; ++i) {
         hilos[i].join();
      }
      std::cout << "Size " << par.first << " anotado\n";
   }
}
```

De nuevo, su cometido se realiza de manera concurrente aunque en este caso hay que tener un poco más de cuidado. Como ya de por sí el tamaño que tendrán estos circuitos para funciones nuevas en la salida no será el mínimo, por lo menos hay que tratar de anotar el tamaño más pequeño encontrado. Para eso, repartiendo el trabajo en hilos, se anotan en la salida las funciones nuevas en orden creciente de tamaño: $n+1, n+2, ..., 2n+1$ (se ordenan porque *espera* no los mantiene ordenados), sincronizando los hilos entre tamaño y el siguiente.

2.3.1. Coste de la generación de circuitos

Hay que destacar que el algoritmo es muy bueno desde el punto de vista de su coste en tiempo. La generación de un circuito apenas requiere de dos recorridos recursivos para calcular todos los subcircuitos de los padres y del cálculo de la intersección entre los mismos (eficiente en la librería estándar). Puesto que los circuitos tendrán tamaños pequeños durante la ejecución (como mucho algunas decenas de puertas), el algoritmo es bastante asequible en este sentido.

Sin embargo, **el coste en memoria del algoritmo es muy elevado**. Hay que notar que, hasta que no se detiene la generación, **se mantienen almacenados todos los circuitos obtenidos hasta el momento, que deberían ser muchos miles de millones**.

Esto supone un hándicap importante, en forma de muchos GB de datos almacenados, que, sin embargo, valía la pena asumir debido a las otras bondades que el algoritmo presentaba. De hecho, anticipando un poco acontecimientos, a pesar de no haber podido realizar ejecuciones tan largas como nos habría gustado, los resultados posteriores refrendaron el dataset obtenido, que se comportó razonablemente bien teniendo en cuenta nuestras pretensiones.

Una advertencia relacionada con este tema del consumo de memoria es que, para ejecuciones por encima de los 100 millones de circuitos (aproximadamente), se genera una excepción debida a que el programa se queda sin memoria para referenciar con punteros de 32 $bits = 4$ $bytes$, lo cual tan solo alcanza hasta 2^{32} posiciones de memoria (unos 4 GB). Por tanto, el programa debe compilarse como ejecutable de 64 bits, con punteros que ocupen 8 $bytes$, pudiendo direccionar hasta 2^{64} direcciones de memoria (unos 16 Teras).

2.4. Repertorios de puertas

Un aspecto que fue absolutamente clave para el objetivo que nos marcamos de **obtener la mayor cantidad de funciones diferentes posible** fue probar con otros repertorios de puertas diferentes al *repertorio estándar* {AND, OR, NOT}.

Otras puertas lógicas que pudiéramos utilizar son implementables con una cantidad constante de puertas del repertorio estándar. Por tanto, toda la teoría en relación a $P_{/poly}$ que vimos en el capítulo 1 se mantiene aunque cambiemos de repertorio (lo "polinómico" para todo n sigue siéndolo y lo "no polinómico" sigue sin serlo). Lo único con lo que debemos tener cuidado es que el repertorio escogido sea universal, en el sentido de permitirnos generar cualquier función booleana.

En relación a este tema, sobre el repertorio de puertas, nos pareció razonable estudiar qué pasaba en los casos extremos en cuanto a la cantidad de puertas: un repertorio universal que tuviera una sola puerta (como puede ser el formado por) y el repertorio completo consistente en todas las puertas lógicas binarias posibles. Si las puertas utilizadas influían de algún modo en la cantidad de funciones obtenidas, parecería extraño pensar que el óptimo, para la elección de tales puertas, fuera un término medio entre los anteriores.

En la siguiente tabla se han señalado con los mismos colores aquellas puertas que son simétricas entre sí, en el sentido de que, intercambiando el orden de las entradas de una de ellas, obtenemos la otra. Es decir, aquellas puertas P y P' tales que $P(x,y) = P'(y,x)$ para todo $x, y \in \{0,1\}$.

Puerta/Entrada	Entrada 11	Entrada 10	Entrada 01	Entrada 00
Puerta P_0	0	0	0	0
Puerta P_1	0	0	0	1
Puerta P_2	0	0	1	0
Puerta P_3	0	0	1	1

Puerta P_4	0	1	0	0
Puerta P_5	0	1	0	1
Puerta P_6	0	1	1	0
Puerta P_7	0	1	1	1
Puerta P_8	1	0	0	0
Puerta P_9	1	0	0	1
Puerta P_{10}	1	0	1	0
Puerta P_{11}	1	0	1	1
Puerta P_{12}	1	1	0	0
Puerta P_{13}	1	1	0	1
Puerta P_{14}	1	1	1	0
Puerta P_{15}	1	1	1	1

Tabla 2.3: Tabla de verdad de las 16 posibles puertas lógicas binarias

Como al implementar la combinación de circuitos con sumidero binario P, para cada par de circuitos C_1 y C_2, tuvimos cuidado de combinar $P(C_1, C_2)$ pero no $P(C_2, C_1)$ (ya que las puertas del repertorio estándar son simétricas y el circuito hijo sería el mismo), ahora debemos incluir las versiones simétricas de las puertas que no lo son. Si se hace la combinación de circuitos $P(C_1, C_2)$, pero no la combinación $P(C_2, C_1)$, cuyo resultado podría ser distinto, en nuestro repertorio debería estar también la puerta P', simétrica de P, que mediante la combinación $P'(C_1, C_2) = P(C_2, C_1)$ complete lo que nos falta.

Además, de las 16 puertas posibles, hay 2 que son tan inútiles que es mejor quitarlas del repertorio para no perder el tiempo en combinaciones absurdas con ellas. Las puertas P_0 y P_{15} no utilizan sus entradas y sacan siempre el valor '0' y '1' respectivamente. Cualquier circuito que utilice la puerta P_0 como sumidero computa la función idénticamente '0'. Por tanto, incluir esta puerta en el repertorio redunda en repetir muchas veces la misma función con diferentes circuitos, lo cual va totalmente en contra de lo que pretendemos conseguir. Análogamente sucede con la puerta P_{15} y la función idénticamente '1'.

Pasa algo parecido con las puertas P_{10} y P_{12}, que consisten en sacar una de las dos entradas que reciben. P_{10} es la identidad de la entrada derecha y P_{12} la identidad de la entrada izquierda. Cualquier combinación usándolas como sumidero computa la misma función que uno de los circuitos padres, pero con un tamaño estrictamente más grande. Por tanto es absurdo usarlas y las eliminamos también del repertorio.

Y finalmente P_3 y P_5 computan el NOT de una de las entradas, y por tanto se pueden sustituir por esta puerta unaria NOT. Con lo cual, de las 16 puertas que aparecían en la tabla 2.3, nos quedamos con la siguientes 10 puertas binarias, descartando, por los motivos comentados, las señaladas en rojo:

Puerta	Nombre	11	10	01	00	Char código	Descripción
P_0	ID0	0	0	0	0	-	Constantemente 0
P_1	NOR	0	0	0	1	O	OR seguido de NOT
P_2	RONLY	0	0	1	0	r	Detecta sólo entrada derecha a 1
P_3	LNOT	0	0	1	1	-	NOT de la entrada izquierda
P_4	LONLY	0	1	0	0	l	Detecta solo entrada izquierda a 1
P_5	RNOT	0	1	0	1	-	NOT de la entrada derecha
P_6	XOR	0	1	1	0	x	Una entrada sólo a 1
P_7	NAND	0	1	1	1	n	AND seguido de NOT
P_8	AND	1	0	0	0	a	Detecta ambas entradas a 1
P_9	XNOR	1	0	0	1	X	XOR seguido de NOT
P_{10}	RID	1	0	1	0	-	Saca la entrada derecha
P_{11}	NLONLY	1	0	1	1	L	LONLY seguido de NOT
P_{12}	LID	1	1	0	0	-	Saca la entrada izquierda
P_{13}	NRONLY	1	1	0	1	R	RONLY seguido de NOT
P_{14}	OR	1	1	1	0	o	Detecta alguna entrada a 1
P_{15}	ID1	1	1	1	1	-	Constantemente 1

El repertorio completo consta de esas 10 puertas binarias y de la puerta unaria NOT.

2.5. Resultados obtenidos

A continuación se resumen cuantitativamente los resultados obtenidos con las ejecuciones del algoritmo.

2.5.1. Resultados según el repertorio

Ejecutando el algoritmo con valores no muy grandes en cuanto al número de circuitos generados (hasta un millón), pudiendo así probar en un tiempo razonable con muchos valores y descubrir la tendencia de cada repertorio, se obtuvieron resultados radicalmente distintos en cuanto a la cantidad de funciones diferentes computadas (figura 2.5.1).

Puede apreciarse cómo, con muchísima diferencia, **el repertorio que mejor se comporta es el que solo tiene la puerta *NAND***. Los repertorios estándar y completo se comportan más o menos igual de mal, aunque parece que el repertorio completo empieza a diferenciarse en torno a los 600.000 circuitos generados. Sin embargo, no está claro si el estándar puede alcanzarle después con cantidades de circuitos mucho más grandes.

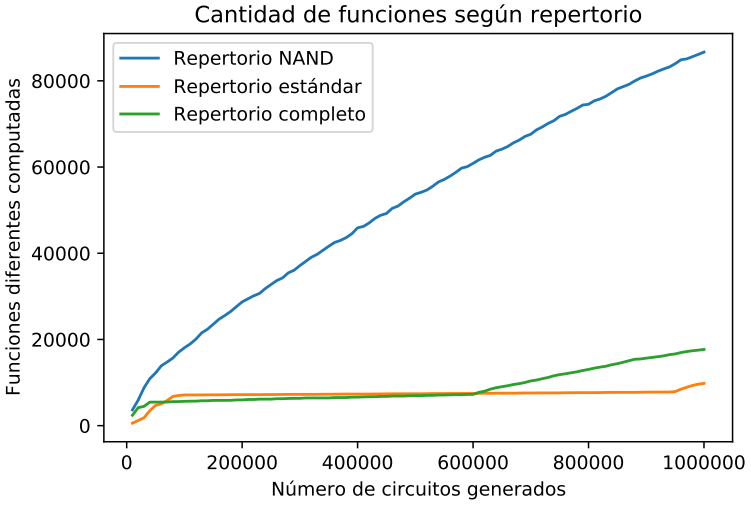


Figura 2.5.1: Circuitos generados frente al número de funciones diferentes computadas según el repertorio.

Al hacer ejecuciones mucho más largas del algoritmo, los resultados de la gráfica anterior se consolidan, solo que no podemos tener suficientes en un tiempo razonable como para representarlos como antes. La ejecución más larga que se ha podido realizar, debido al exigente uso de memoria del algoritmo, es de 500 millones de circuitos. El repertorio con *NAND* obtuvo, para una ejecucion de ese tamaño, un total de 1.655.365 funciones diferentes, que se pueden encontrar en el fichero [9]. El repertorio completo obtuvo 486.685 funciones, que se pueden encontrar en [8]. Y el repertorio estándar apenas superaba las 200.000 funciones y su fichero no se utilizó para el dataset (más tarde se explica el motivo).

Surgieron dos intuiciones al respecto de por qué funcionó mucho mejor el repertorio monopuerta. La primera tiene que ver con que la variedad de funciones no la aportan puertas muy distintas, sino mayores profundidades de circuito.

Cuando hay varias puertas, para una misma estructura de circuito se valoran las posibilidades de que, en cada vértice del circuito, se coloque cada una de las puertas del repertorio. Por tanto, se valoran, muy cerca en el tiempo, circuitos parecidos (al menos en su forma), lo que redunda en funciones parecidas a pesar de que las puertas hagan cosas distintas. Con un repertorio unitario, cada estructura de circuitos se visita una

sola vez, y esto hace que la variedad de estructuras de circuitos sea mucho mayor en menos tiempo, lo que redunda en mayor diversidad de funciones.

Nuestra segunda intuición es que alcanzar profundidades grandes más rápido aumenta antes la expresividad de los circuitos. Con un repertorio como el *NAND*, que alcanza profundidades grandes mucho antes, la capacidad de los circuitos para computar funciones medianamente complejas surge también antes. Esta expresividad más prematura redundaría en que funciones más diversas afloren antes.

Otra ventaja del repertorio *NAND*es que nos permitía ahorrar espacio, al no ser necesario representar internamente la puerta sumidero de cada circuito (no hay más que una). Almacenar esta puerta no era estrictamente necesario para obtener la función y tamaño, como pretendíamos en un principio, pero preferimos sacar también una representación del circuito mínimo encontrado por si podía resultarnos útil después.

En el caso de la ejecución para 500 millones con puertas *NAND*, se llegaron a generar circuitos de hasta tamaño 11, mientras que con el repertorio completo solo se llegó hasta tamaño 7.

2.5.2. Dataset definitivo

La cantidad de datos podía ser muy importante no solo de cara a cotejar después diversas conjeturas, sino sobre todo de cara a aplicar técnicas de Aprendizaje Automático. Los modelos de *machine learning* suelen funcionar mucho mejor con datasets muy grandes. Aunque el nuestro ya iba a gozar de millones de funciones, el afán por "engordarlo" sin introducir grandes distorsiones era más que evidente.

Por ello, entre otras cosas, se optó por mezclar las funciones obtenidas en las salidas de los distintos repertorios de puertas, ya que, aunque la mayoría iban a ser funciones repetidas, un número importante de ellas no, y era una manera fácil de aumentar el tamaño de nuestro dataset definitivo. Esto supone cierta perturbación en los datos, pues, como es evidente, no es lo mismo un circuito de tamaño 4 con el repertorio completo, que permite puertas tan sofisticadas como la *XOR* (implementable con 4 puertas *NAND*), que uno que tenga directamente tamaño 4 usando las *NAND*.

Nuestra decisión se justifica igual que la de incluir circuitos con un tamaño que pudiera no ser el mínimo (aunque sí parecido), tratando de no desperdiciar trabajo del generador. Debido a que posteriormente íbamos a simplificar entendiendo que los tamaños concretos son algo demasiado sutil y que lo importante es distiguir circuitos "pequeños" de "grandes" (no nos interesa la diferencia entre tamaño 7 vs 8 sino entre tamaño 7 vs 100), la perturbación por mezcla de repertorios resultaba ciertamente asumible.

En esta línea de mezclar salidas del generador, también se trabajó con las de distintas ejecuciones para el mismo repertorio. Al detenerse la generación en el algoritmo, los hilos de la última etapa que no terminan su tarea pueden haber avanzado de distinta forma en distintas ejecuciones (depende de cómo se hayan repartido las CPU's) y, por tanto, no hay determinismo en las funciones nuevas que estos hilos computan.

Esto se cuantificó para tres ejecuciones de 500 millones de circuitos con *NAND*, donde más de 400 millones se generan en la última etapa. La tabla 2.5 muestra la cantidad de circuitos generados por cada uno de los 8 hilos lanzados en la última etapa (la máquina usada tenía 8 núcleos). Podemos observar que, en la práctica, todos los hilos se comportan de manera similar para distintas ejecuciones (seguramente debido al planificador). No hay una ejecución en que un hilo genere 80 millones de circuitos y otro solo 20, sino que todos son muy estables en torno a los 50 millones. Es decir, hay un indeterminismo que, por el reparto de los procesadores, en la práctica es muy leve.

Traza/Hilo	Hilo 1	Hilo 2	Hilo 3	Hilo 4	Hilo 5	Hilo 6	Hilo 7	Hilo 8
Traza 1	50493842	51971215	50313064	50036364	51349448	50075003	50160540	51943369
Traza 2	51039129	50825861	50697700	50666388	49026247	51804210	51600415	50682895
Traza 3	50530730	50124053	51681642	50404832	51090513	50440588	51592009	50478478

Tabla 2.5: Numero de funciones computadas por hilos en 3 ejecuciones

Sin embargo, podemos explotar mucho más este no determinismo de la última etapa sin modificar el planificador. Los aproximadamente 50 millones de circuitos generados por cada hilos coinciden de una ejecución

para otra porque recorren su espacio de circuitos en el mismo sentido. Como el espacio de circuitos que cada hilo tiene capacidad de generar en la última etapa no se visitará entero (se trunca la etapa sin acabar), recorriendo los circuitos en sentido opuestos (basta recorrer los vectores al revés), podemos generar conjuntos de circuitos totalmente disjuntos para 2 ejecuciones distintas (figura 2.5.2).

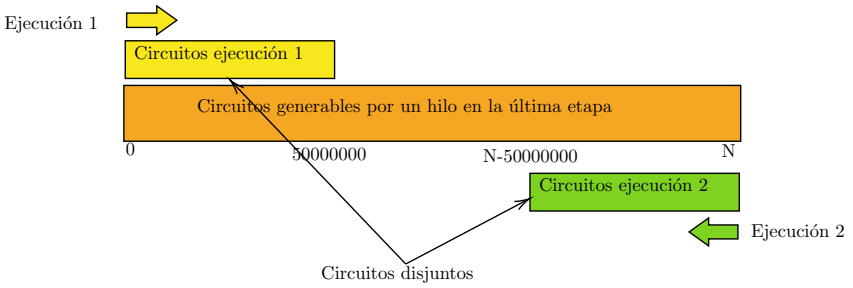


Figura 2.5.2: Generación de un hilo en la última etapa para dos ejecuciones recorriendo en distinto orden.

Basta que la cantidad de circuitos que cada hilo pudiera generar supere los 100 millones para que los 50 millones de le segunda ejecución recorriendo al revés sean totalmente nuevos respecto a los 50 millones de la primera. Se comprobó, que, en efecto, sucede así y la situación es la que refleja la figura 2.5.2. Esto, que sucederá para los 8 hilos, puede proporcionar muchas funciones distintas para ejecuciones igual de largas con el mismo repertorio.

Debido a la enorme diferencia en la cantidad de funciones entre repertorios, de cara a mezclar salidas parecía mejor invertir tiempo en ejecuciones largas con *NAND* con recorridos distintos, en lugar de en ejecuciones con el repertorio estándar. Sin embargo, sí se hizo mezcla con ejecuciones del repertorio completo porque sus resultados no fueron tan malos como en el estándar, y se intuía cierta variedad en sus funciones respecto a las obtenidas con *NAND*.

Mezclando salidas para el repertorio *NAND*, el repertorio el completo y entre distintas ejecuciones para ambos repertorios, se obtuvo un dataset (que se puede ver, ordenado por función en [7]) con un total de 2.050.334 funciones distintas (un aumento de 400.000 debido a la mezcla), que suponen un 0.048 % de las 2^{32} funciones totales. Lo cuál es más que satisfactorio teniendo en cuenta que la tabla de Enrique, que teníamos opción de reutilizar antes de sumergirnos en el diseño de nuestro generador, constaba tan solo de 600.000 funciones, sin ninguna garantía de que fuesen computables con circuitos “pequeños”.

Capítulo 3

Conjeturas y métricas

El siguiente paso en el trabajo era estudiar los factores que pueden influir en el tamaño de los circuitos mínimos para las funciones, apoyándonos para ello en nuestro dataset como muestra de funciones que se pueden computar con circuitos pequeños. Para ello, cambiamos el lenguaje de programación con el que seguir trabajando. Nuestra labor iba a ser más fácil y productiva en Python, debido a la inmensa cantidad de utilidades que proporcionan sus librerías para el análisis de datos.

Sin embargo, el cálculo de las métricas que diseñásemos para cotejar nuestras conjeturas podía complicarse debido a la lentitud de este lenguaje en comparación con C++ y la utilización de un IDE para facilitar nuestra labor programadora, que ralentiza la ejecución frente al trabajo por consola (se usaron notebooks de Jupyter). En esta fase del trabajo era más importante poder programar muchas cosas rápido para poder explorar muchas hipótesis diferentes, que la eficiencia y completitud de las pruebas. Por tanto, la mayoría de estas pruebas se hicieron solo sobre algunas partes del dataset, de forma que solo aquellas que dieron buenos frutos se programaron y ejecutaron en C++ sobre todos nuestros datos.

3.1. Inspección visual y primera conjetura: repetición de patrones

Para empezar, decidimos hacer una primera aproximación a nuestro dataset buscando a ojo patrones o características que nos llamasen la atención de los datos. Lo hicimos de esta forma y no con *machine learning* directamente, porque temíamos que un modelo automático, por simple que fuese, aprendiera cosas ya muy complicadas de explicar y entender para un ser humano. Esa vía se iba a tratar, pero más adelante. Por el momento solo buscábamos ideas claras y sencillas.

Como teníamos más de 2 millones de funciones, decidimos observar unas pocas ordenando por su cantidad de apariciones durante la generación (el dataset no anotaba esto, pero se hizo una pequeña modificación y una nueva ejecución larga para poder tener esos datos). De esa forma, podríamos directamente fijarnos en las funciones que, de algún modo, pueden ser más representativas de las computables con tamaño "pequeño", en el sentido de que han aparecido más veces construyendo circuitos "pequeños". Estas funciones, junto a su número de repeticiones al generarse, pueden encontrarse en la primera celda del notebook [15].

A continuación, en la tabla 3.1, podemos ver el top 10 de funciones más computadas por los circuitos generados. Están representadas como cadenas de 32 bits de la forma que vimos en la tabla 2.1.

Top	Función	Número de apariciones
1	11111111000000001111111100000000	2765442
2	11111111111111110000000000000000	2689237
3	11110000111100001111000011110000	2678444
4	11001100110011001100110011001100	2662373
5	10101010101010101010101010101010	2655816
6	11111111111111111111111111111111	1650463
7	01011111010111110101111101011111	514359
8	00110011001100111111111111111111	513141
9	01010101111111110101010111111111	511623
10	01010101010101011111111111111111	509413

Tabla 3.1: Top 10 de funciones según la cantidad de veces generadas

Algo que, de entrada, llama poderosamente la atención es la enorme repetitividad de patrones en la representación como cadena de bits de estas funciones.

La función top 1 de la tabla 3.1 es 2 veces la cadena con 8 unos y 8 ceros, que podríamos expresar como 2*(8*1's+8*0's). La función top 2 es la cadena formada por 16 unos y 16 ceros que podría expresarse como 16*1's+16*0's. La función top 3 es la cadena que consiste en repetir 4 veces el patrón 4 unos y 4 ceros, que podría expresarse como 4*(4*1's+4*0's). Y aunque menos marcados, hay patrones de ese estilo en las demás.

Pero, ¿estos patrones son cosa solo de las primeras funciones del top o sucede lo mismo en general? Si miramos que pasa mucho más abajo en el ranking, por ejemplo, en el top 300.000 observamos lo que recoge la tabla 3.2.

Top	Función	Número de apariciones
300000	00010101101111110111010111111111	110
300001	00110111001111110011111100110111	110
300002	00000010111011101010101011111111	110
300003	00010111000001110101011111111111	110
300004	10101011001000001111111100100000	110
300005	10011011010111111100111111001111	110
300006	10110011101111110000000010110011	110
300007	10001111100011101010101010101010	110
300008	11111111110000001111101011111010	110
300009	00010001011101110111001101110111	110
300010	01110011011100111111101101010001	110

Tabla 3.2: Funciones a partir de la top 300.000 según la cantidad de veces generadas

Conforme bajamos en la tabla hay bastante menos claridad en la repetición y periodicidad de patrones, pero igualmente parece percibirse algo más que la que debiera tener, a priori, la cadena promedio. Por ejemplo, la función top 300.001 empieza por 00110111, pero también acaba así. Y en medio tiene la repetición de 00111111 dos veces. Es decir, que puede expresarse como AB^2A donde $A = 00110111$ y $B = 00111111$.

Dedicando bastante tiempo a seguir viendo ejemplos de este estilo y tratando de encontrar patrones a ojo, se llegó a la conclusión de que un conjunto de cadenas aleatorias no debía tener tanta repetitividad como las de nuestro dataset. Por tanto, era muy probable que estuviésemos ante una característica de las funciones computables con circuitos "pequeños". Es más, después de mucho observar, se llegó también a la conclusión de que estos patrones solían tener longitud potencia de 2. Prácticamente todos los patrones encontrados a ojo tenían longitud 2, 4, 8, 16 o 32. En el ejemplo anterior, tanto A como B tienen longitud 8 y simplemente con ellos somos capaces de expresar la cadena entera que constituye la función.

Por tanto, obtuvimos una conjetura que seguidamente tratamos de cotejar sobre el dataset: **que una función tenga muchos patrones de repetición de longitud potencia de 2 favorece su computabilidad con un circuito "pequeño"**.

3.1.1. Intuición sobre la conjetura de repetitividad

Más allá de que nos pareció percibir en el dataset una mayor repetitividad de patrones de longitud potencia de 2, existe una explicación lógica que le da sentido a nuestra conjetura.

La idea es que las funciones de los circuitos iniciales de tamaño 0 ya son tremendamente repetitivas para longitudes potencia de 2. Al combinar estos circuitos entre sí, los patrones tienden a reproducirse en los hijos perdiéndose poco a poco y desapareciendo a la larga. Al fin y al cabo, aplicar una operación lógica a un par de estas funciones consiste en utilizar una de ellas como máscara sobre la otra.

Con esa idea de la máscara se ve muy claro que, al aplicar una operación entre patrones potencia de 2, obtenemos de nuevo patrones potencia de 2.

En la tabla 3.3 pueden verse las funciones computadas por los circuitos degenerados de tamaño 0, junto a una expresión que muestra su enorme repetitividad.

Circuito	Función	Expresión repetitiva
e_0	10101010101010101010101010101010	16*(‘1’+‘0’)
e_1	11001100110011001100110011001100	8*(2*‘1’+2*‘0’)
e_2	11110000111100001111000011110000	4*(4*‘1’+4*‘0’)
e_3	11111111000000001111111100000000	2*(8*‘1’+8*‘0’)
e_4	11111111111111110000000000000000	16*‘1’+16*‘0’

Tabla 3.3: Funciones computadas por los circuitos degenerados y expresión de repetitividad de patrones

Al aplicar la operación lógica *NAND* sobre las funciones computadas por e_3 y e_4, se obtiene una función que de nuevo presenta una gran repetición de patrones que siguen teniendo longitud potencia de 2 (figura 3.1.1).

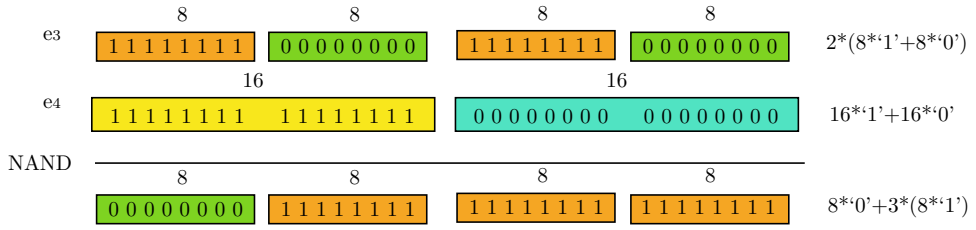


Figura 3.1.1: Combinación mediante *NAND* de las funciones e_3 y e_4.

Sin embargo, según vamos añadiendo profundidad al circuito (al meterle un sumidero más), los patrones repetitivos de longitud potencia de 2 se van degradando hasta ser prácticamente inexistentes. Al aplicar muchas máscaras sobre una de estas funciones, esos patrones de longitud potencia de 2 van poco a poco difuminándose.

Las funciones que estamos viendo son las que tienen el tamaño asociado más pequeño (las degeneradas o su combinación con *NAND*, computable con apenas tamaño 1). Estas funciones ocupaban las primeras posiciones del ranking de más veces computadas en la tabla 3.1 y fueron las que percibimos como más repetitivas. De seguir combinándolas entre sí, acabaríamos obteniendo todas las demás, incluyendo las que están más abajo en ese mismo ranking que, como ya comentamos, mostraban una perdida de repetitividad más que evidente respecto a las primeras. Esto, en cierto modo, pone de manifiesto el degradado de repetitividad que mencionábamos anteriormente.

3.2. Invarianza de las métrica bajo permutaciones de la significación

De cara a diseñar métricas que nos permitieran cotejar nuestras conjeturas sobre las funciones, pensamos que sería deseable que no dependieran del orden de significación de la representación escogida.

Las expresiones de repetitividad que hemos mostrado en la tabla 3.3 son para la representación que expusimos en la tabla 2.1, en la que hay fijado un orden de significación para las componentes de entrada: $(e_4, e_3, e_2, e_1, e_0)$ (yendo de más a menos significativo). Esto, implícitamente, supone fijar una referencia para mirar a las funciones, siendo cualquier otra igual de válida. Dándole otra significación a las componentes se puede apreciar que las funciones de la tabla 3.5 son, en el fondo, la misma (aquella que solo utiliza uno de los bits). Solo hay que mirarlas desde distintas perspectivas para darse cuenta de ello.

Por poner una analogía, cambiar el orden de significación es como reordenar los vectores de la base de un espacio (caso particular de un cambio de base). El aspecto de los elementos cambia, pero siguen siendo los mismos en el mismo espacio vectorial.

En la tabla 3.4 puede verse la componente de la entrada que computaría cada función correspondiente a circuitos de 0 puertas, para 5 órdenes de significación distintos.

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 3.4: Funciones computadas por circuitos de tamaño 0 para 5 órdenes de significación

La tabla anterior muestra que las 5 funciones son la misma cambiando referencias. Esto despierta cierto interés por tratar de desarrollar métricas invariantes bajo permutaciones en la significación, evitando así cierta "miopía" a la hora de mirar a una función y motiva la siguiente definición.

Definición 3.2.1. *Diremos que dos funciones booleanas* $f_1, f_2 : \{0,1\}^5 \to \{0,1\}$ *son* ***equivalentes bajo permutaciones de sus componentes*** *si existe cierta permutación* $\sigma : \{0,1,2,3,4\} \to \{0,1,2,3,4\}$*, para la cual:*

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

Lo denotaremos como $f_1 \sim f_2$*.*

La definición 3.2.1 refleja que, cambiando el sistema de referencia (orden de las componentes) con el que miramos a la primera función, es posible obtener exactamente la segunda función y viceversa[1]. Es decir, dos funciones equivalentes en este sentido son, en el fondo, la misma función vista de dos formas distintas. Cada una de estas funciones puede ser equivalente con hasta otras 119, ya que hay $5! = 120$ posibles permutaciones para el orden de significación y parece lógico proporcionar el mismo resultado para todas ellas en la métrica.

Definición 3.2.2. *Diremos que una métrica* $M : Func(\{0,1\}^5 \to \{0,1\}) \to \mathbb{R}$*, que asigna una puntuación* $M(f)$ *a cada función* $f : \{0,1\}^5 \to \{0,1\}$ *es* ***invariante bajo permutaciones en la significación*** *si:*

$$M(f_1) = M(f_2)\ \forall f_1, f_2 \text{ tal que } f_1 \sim f_2$$

En la tabla 3.5 puede verse un ejemplo menos trivial de esta equivalencia. Se muestran dos funciones que, por representación, puede parecer que no tienen nada que ver y que sin embargo son equivalentes cambiando el orden de significación $(e_2, e_0, e_3, e_1, e_4)$ por el orden $(e_4, e_3, e_2, e_1, e_0)$ y viceversa. (Las etiquetas de entrada mostradas en N permiten seguir a dónde va cada *input* tras cambiar el orden de las componentes).

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

N	31	15	27	11	30	14	26	10	23	7	19	3	22	6	18	2	21	5	25	9	28	12	24	8	29	13	17	1	20	4	16	0
e_2	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_0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
e_3	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	0	1	1	1	1	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	1	1	1	1

Tabla 3.5: Misma función f_1 considerando órdenes distintos para las componentes obtiene distinta representación

3.3. Métrica de repetición de patrones

Que nuestras métricas no sean invariantes bajo permutaciones según la definición 3.2.2 supone cierta incoherencia con los fundamentos teóricos que acabamos de exponer. Sin embargo, no invalida completamente su

[1] El "viceversa" es debido a que toda permutación σ tiene inversa σ^{-1}, que es también una permutación.

utilidad práctica. Las métricas son una herramienta para testear cierta conjetura sobre los datos y pueden tener imperfecciones al igual los propios datos las tienen.

En el caso de la repetitividad no se encontró una manera sencilla de calcular puntuaciones que, de forma natural, resultase invariante bajo permutaciones. El objetivo era comprobar si nuestros presagios de mayor repetitividad en el dataset eran ciertos, y para eso una métrica que no cumpliese la invarianza también podía servirnos.

3.3.1. Idea básica de la métrica de repetitividad

Una primera idea sobre cómo medir esta repetitividad tenía que ver con elaborar un lenguaje que proporcionase una descripción de las cadenas como repetición de patrones. Es decir, algo similar a lo que hicimos anteriormente al decir que cierta función podía expresarse como 2*(8*1's+8*0's). A partir de esa descripción, luego se cuantificaría su calidad, midiendo, por ejemplo, la cantidad de símbolos que tiene (a menos símbolos en principio mayor repetitividad), lo que determinaría cuánto de repetitiva es.

El problema de ir en esa línea, es que calcular una expresión óptima en ese sentido tiene costes demasiado elevados (hay demasiadas cosas que comprobar en la cadena). Y aunque renunciáramos a la óptimalidad para una descripción razonable de la cadena, los costes seguirían siendo demasiado altos para trabajar con todo nuestro dataset o con una parte grande más adelante.

Por tanto, como el principal objetivo de la métrica es sacarnos de dudas sobre si nuestra conjetura tiene o no sentido, optamos por mediciones muchísimo más simples y baratas computacionalmente.

Una idea fácil, que tiene mucho que ver con cómo buscaríamos nosotros a simple vista estos patrones de longitud potencia de 2, es trocear la cadena en subcadenas estudiando después la repetitividad entre los trozos. Si nos dicen que busquemos repeticiones en una palabra con patrones que sean potencia de 2, la partiríamos en dos mitades de 16, luego en tres partes donde las 2 primeras tengan tamaño 8 y la última tamaño 16, luego en otras 3 partes donde la primera tenga tamaño 8, la segunda 16 y la última 8, etc.

En la figura 3.3.1 se pueden ver algunos ejemplos de posibilidades para tales particiones sobre la función que salió top 300.001 en la tabla 3.2:

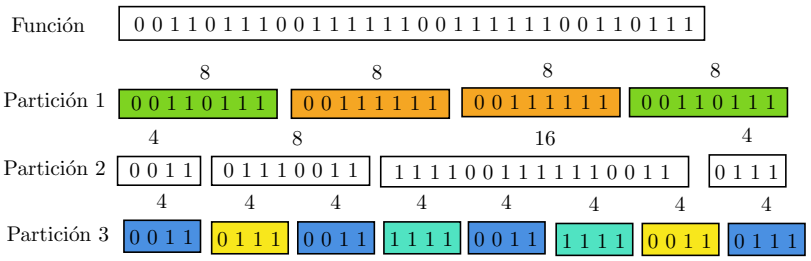


Figura 3.3.1: Tres posibles particiones en potencias de 2 de la cadena que vimos anteriormente.

Debemos darnos cuenta de varias cosas. La primera es que, si en la partición 1 de la figura 3.3.1 detectamos que los trozos 1 y 4, y los trozos 2 y 3 (señalados con el mismo color), son respectivamente iguales, habríamos descubierto la forma de expresar la palabra como AB^2A, que apuntábamos previamente como justificación de la enorme repetitividad de esta cadena.

La segunda cosa que debemos observar es que hay particiones, que, a priori, son menos interesantes para la repetitividad. En la partición 2 de la figura 3.3.1, al comparar trozos entre sí, solo podría obtenerse repetición en los dos trozos de tamaño 4, ya que los otros dos trozos tienen longitudes distintas entre sí (longitud 8 y 16). Parece que son más útiles las particiones con trozos de la misma longitud.

Y también debemos caer en la cuenta de que la partición 3 es una subpartición de la partición 1 y de la partición 2 (basta segmentar los trozos de longitud > 4 en trozos de longitud 4 para obtenerla).

3.3.2. Puntuación de repetitividad

La cantidad de particiones a considerar utilizando la idea anterior para puntuar surge de las formas de sumar 32 con elementos del conjunto $\{2, 4, 8, 16, 32\}$, siendo importante el orden en que se usan. Haciendo un sencillo programa para calcular cuáles eran, se comprobó que hay más de 5.000 particiones posibles, y por tanto resultaba poco asumible recorrerlas todas para cada función del dataset realizando cierto cálculo no trivial. La explosión combinatoria es debida a la cantidad de posibilidades en las que interviene el valor 2. Un valor que, por otra parte, es conflictivo para medir repetitividad entre trozos, ya que: ¿hasta qué punto una repetición de apenas tamaño 2 en zonas dispersas de la cadena no es una mera casualidad? Considerando patrones tan pequeños corríamos el riesgo de que todas las cadenas nos parecieran muy repetitivas.

Parecía lógico eliminar el valor 2 y trabajar solo con los valores del conjunto $\{4, 8, 16, 32\}$. Esto reducía la cantidad de posibles particiones a las apenas 56.

Tras unas cuantas pruebas buscando una idea sobre cómo puntuar de una manera interesante las repeticiones de trozos iguales de una función para una partición, se llegó a la conclusión de que podía bastar con medir la longitud cubierta por dichas repeticiones.

Aunque los ejemplos que hemos ido destacando se pueden cubrir íntegramente como repetición de patrones (cada patrón aparece al menos 2 veces), en general no es así y lo normal es que la cadena de una función no pueda cubrirse entera con repeticiones para ninguna de las 56 particiones posibles. Por ello, decidimos puntuar una función, para una determinada partición, con la longitud que logra cubrir mediante patrones repetidos, otorgándole a la función el máximo de sus puntuaciones para las particiones.

Más tarde, caímos en la cuenta de que no es necesario considerar las 56 particiones, sino que basta limitarse a la partición más fina posible: $p = (4, 4, 4, 4, 4, 4, 4, 4)$.

Si varios trozos se repiten para una de las 56 particiones, debido a que su longitud ≥ 4 es una potencia de 2, se pueden a su vez dividir en trozos de longitud 4, que igualmente se repiten. (Si dos trozos coinciden, en particular coinciden sus subtrozos). Por tanto, con la partición $p = (4, 4, 4, 4, 4, 4, 4, 4)$ se detectan al menos todas las repeticiones de cualquier otra partición p' (y puede que incluso alguna más), lográndose cubrir con repeticiones al menos la misma longitud de la cadena.

3.3.3. Evaluación y resultados de la métrica de repetitividad

Para evaluar las métricas sobre nuestros datos fue fundamental comparar sus comportamientos sobre el dataset frente a sus comportamientos para funciones aleatorias del complementario. Las distintas pruebas que vamos a relatar pueden encontrarse en el notebook [15].

Como ya comentamos, la manera creciente de generar circuitos garantiza que las funciones de nuestro dataset eran las computables con los circuitos más "pequeños". Teniendo en cuenta el inmenso mar de funciones que no llegamos a encontrar (más del 99 % de las 2^{32} funciones totales), cuyos tamaños mínimos de circuito en general no deberían parecerse a los del dataset, tomando funciones aleatorias del complementario podíamos obtener muestras para contrastar las métricas.

Las figuras 3.3.2 y 3.3.3 reflejan las puntuaciones de la métrica de repetividad sobre aproximadamente la milésima parte de las funciones del dataset y del complementario (2.000 funciones de cada) escogidas al azar.

Visualmente ya se aprecia una tónica de puntuaciones radicalmente distinta entre el dataset y el complementario, que augura buenos resultados. Las funciones del dataset obtienen, por lo general, puntuaciones de repetitividad mucho más altas que las del complementario, lo que parece corroborar nuestra conjetura.

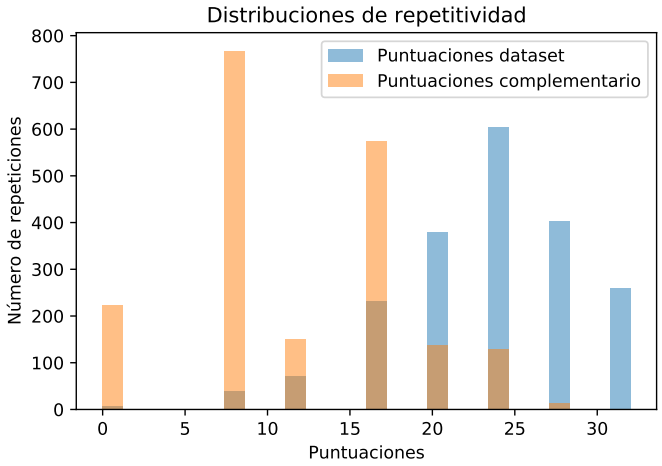


Figura 3.3.4: Distribuciones de puntuación de la métrica de repetitividad para muestras de tamaño 2.000.

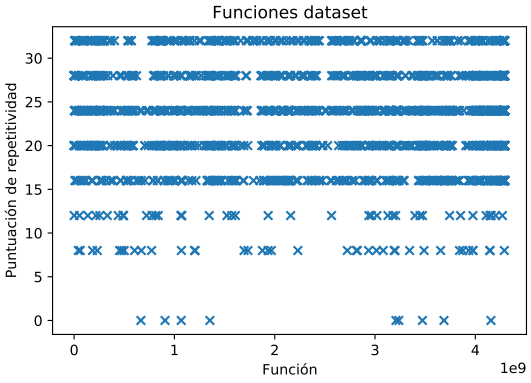


Figura 3.3.2: Puntuaciones en el dataset.

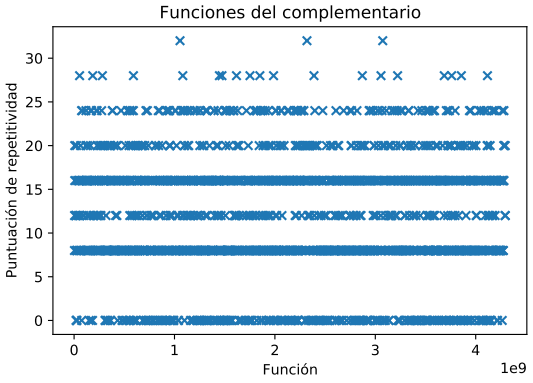


Figura 3.3.3: Puntuaciones en el complementario.

Cabe destacar que el azar que influye para determinar las funciones del dataset y del complementario escogidas para las muestras es casi irrelevante en los resultados. Dichos resultados permanecieron prácticamente estables para distintas ejecuciones y para distintos tamaños de muestra. Para la mayoría de pruebas se utilizó solamente una pequeña parte del dataset por motivos de visualización (y también computacionales).

Obtuvimos distribuciones de puntuaciones que pueden verse en el histograma de la figura 3.3.4. El histograma permite visualizar cómo de distinguibles son las clases "dataset" y "complementario" mediante la métrica, fijándose fundamentalmente en la cantidad de casos cuya clasificación sería muy clara y la cantidad de casos para los que no lo sería (los que reciben puntuaciones muy intermedias).

La métrica parecía ser suficiente para una buena distinción entre las dos clases, siendo pocos los casos donde proporcionaba una puntuación "ambigua" que no permitiese una clasificación clara entre el dataset y complementario. Una manera muy simple de comprobarlo era predecir la clase de un ejemplo (dataset o complementario) calculando su puntuación y elegir la clase cuya puntuación media fuese más próxima[2].

La media de puntuación del dataset salía de unos 23.6 y la del complementario rondaba los 11.6. Si un ejemplo obtuviese puntuación de 20 se clasificaría como ejemplo del dataset ($|23.6-20| < |11.6-20|$), siendo un acierto para el dataset si realmente lo era o un fallo para el complementario si no lo era.

Haciendo esto sobre otros 2.000 ejemplos aleatorios del dataset y del complementario (casi todos serían

[2]Más adelante ibamos a utilizar Aprendizaje Automático para hacer clasificaciones mucho mejores; por el momento solo queríamos hacernos una idea de la calidad de la métrica.

nuevos), se obtuvo un 82.15 % de acierto para las funciones del dataset y un 86.05 % para las funciones del complementario, lo que ratifica las buenas percepciones que teníamos de la métrica.

Reiteramos que, aumentando los tamaños de muestra, estos porcentajes permanecieron muy similares y los resultados no estaban condicionados por nuestro pequeño muestreo. Para muestras con 500.000 funciones, que ya suponen la cuarta parte de nuestro dataset los porcentajes de acierto fueron 82.11 % sobre el dataset y 86.27 % sobre el complementario.

Un aspecto muy positivo para la conjetura es la escasez (casi ausencia) de casos extremos indistinguibles con la métrica, que podríamos calificar como casos "contraejemplo". Parece indicar que la repetitividad no solo es un factor influyente en el tamaño del circuito mínimo de una función, sino también un factor determinante.

Siendo nuestra conjetura cierta, la influencia de la repetitividad podría, en muchos casos, quedar empequeñecida por otro factor que atenuase o prácticamente ocultase su influencia. En principio, podría pensarse que por muy repetitiva que sea una función, si cumpliera cierta otra dificultad ajena a nuestra conjetura, igualmente necesitaría un circuito "grande" para computarse. Pero no parece ser así, porque tal otra dificultad debería estar más presente en nuestros ejemplos aleatorios y no hubo un número considerable de casos muy discordantes (3.3.4).

Un pequeño *benchmark* que guardábamos en la recámara para darle el visto bueno final a la conjetura y a la métrica era comprobar su comportamiento con la función Paridad: la función que saca un '1' cuando la cantidad de símbolos '1' del *input* es par y saca '0' en caso contrario.

Esta función es muy interesante desde el punto de vista de la complejidad de circuitos porque es una función "falsamente difícil". Puede computarse fácilmente con un árbol binario de puertas *XOR* como el que mostramos en la figura 1.1.3 del primer capítulo para puertas *AND*'s, cuya cantidad de puertas ya justificamos que era polinómica. Sin embargo, si se restringe la profundidad de los circuitos para computarla con una profundidad constante (profundidad $\mathcal{O}(1)$) es imposible hacerlo con una cantidad polinómica de puertas.

En la página 288 de [1] se prueba que esta función no está en la clase de complejidad AC^0 que consiste en las familias de circuitos con profundidad $\mathcal{O}(1)$ y tamaño polinómico.

La representación de Paridad para 5 bits de entrada es 0110 1001 1001 0110 1001 0110 0110 1001, que puede expresarse como EF con E=CD, F=DC, C=AB y D=BA, donde A=1001 y B=0110. Deshaciendo esa anidación de patrones, Paridad para 5 bits puede expresarse como BAABABBA, lo que presenta patrones de repetición de longitud potencia de 2 como los que conjeturamos, que nuestra métrica es capaz de detectar, otorgándole puntuación máxima a la función (se cubre la palabra entera con repeticiones). De hecho, se comprobó que Paridad era una de las funciones del dataset.

En general, la tabla de verdad de paridad sigue una repetitividad recursiva en forma de fractal; para 5 bits puede expresarse como EF; para 6 bits consiste en GH con G=EF, H=FE. Para 7 bits como IJ donde I=GH, J=HG; y así sucesivamente. Nuestra métrica también sería capaz de detectar los patrones de Paridad a tamaños superiores, aunque siendo totalmente ajena a esta construcción fractal. Sin significar esto gran cosa, fue una pequeña comprobación de que, dentro de su sencillez, la métrica no iba mal encaminada.

3.4. Métrica de distinción de pares cruzados

3.4.1. Conjetura sobre distinción de pares cruzados

Una idea que manejábamos sobre por qué una función puede requerir un circuito grande para computarse tiene que ver con la necesidad de hacer muchas distinciones de casos que inevitablemente suponen el uso de muchas puertas.

Dada una función booleana y dos *inputs* distintos, si la función tiene salidas diferentes para ellos, un circuito que la compute necesariamente debe tener la lógica suficiente para diferenciarlos. En este sentido, las funciones que sacan mucho más el valor '0' que el valor '1', o viceversa, tienden a ser más fáciles de computar ya que sus circuitos tienen que distinguir menos casos.

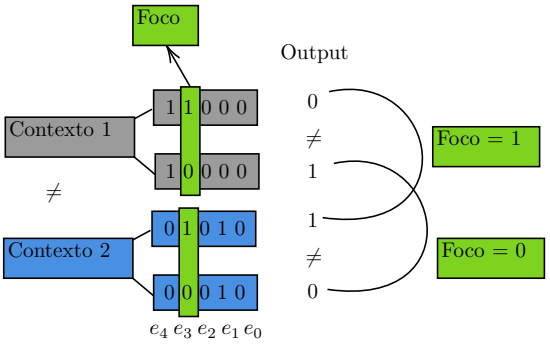


Figura 3.4.2: Dos pares de *inputs* a distinguir con igual foco y con salida "cruzada".

La magia de los circuitos reside en la capacidad de atender a muchos de estos casos con las mismas puertas. Cada distinción no requiere de un grupo propio de puertas en el circuito, sino que hay zonas del mismo que permiten diferenciar entre varios pares de *inputs* a la vez. Sin embargo, no todas las distinciones se molestan entre sí de la misma forma. Determinar qué distinciones de pares son más "compatibles" y cuáles más "incompatibles" podría ser fundamental para saber si una función puede computarse con un circuito "pequeño" o no.

En ese sentido, nos fijamos en parejas de *inputs* que fuesen especialmente difíciles de diferenciar por tener algunos bits iguales. En la figura 3.4.1 podemos ver un ejemplo de una función con distinta salida para los *inputs* 11000 y 10000.

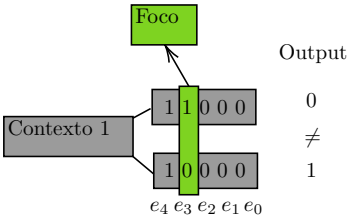


Figura 3.4.1: Par de *inputs* con salidas distintas que comparten 4 de sus 5 bits.

Esta diferenciación obliga a un circuito que compute la función a fijarse en el bit que hemos señalado como foco, ya que el resto de bits, que forman lo que hemos llamado contexto, son exactamente iguales. Para ello el circuito tiene que hilar más fino de lo normal fijándose en un bit en particular, aunque también son interesantes los casos en que debe fijarse en 2 o 3 bits.

Pero esta diferenciación, que ya es ligeramente más difícil de lo normal, se complica aún más cuando fijarse solo en el foco no es suficiente para que el circuito salga de dudas y es necesario utilizar también el contexto. Esta situación se da cuando hay una diferenciación con el mismo foco, distinto contexto y las salidas justo al revés. En la figura 3.4.2 podemos ver un ejemplo con dos distinciones de pares en esas condiciones.

Un circuito que compute una función con esas salidas, para esos *inputs*, no solo debe fijarse en el foco, que es común a las dos diferenciaciones de pares que debe hacer, sino que también tiene que utilizar los contextos de cada diferenciación. En la primera, el valor '1' del foco supone que el circuito debe sacar '0', mientras que en la segunda supone que debe sacar '1', con lo cual el foco no es suficiente para decidir. Estas diferenciaciones cruzadas se intuyen especialmente complicadas y tendría bastante sentido que una cantidad grande de las mismas supusiera la necesidad de más puertas en los circuitos.

Aunque, por simplificar la explicación, el ejemplo utilizado tiene un foco de 1 bit y un contexto de 4 bits, la idea es extrapolable a "focos" de 2 y 3 bits que también tienen interés[3]. Surge por tanto la siguiente

[3]El fenómeno de las distinciones cruzadas no es posible con "focos" de 4 bits porque no hay más que un par de *inputs* con tal foco.

conjetura: **las funciones en que se produce más el fenómeno de distinción de pares cruzados son más proclives a necesitar circuitos grandes para computarse**.

3.4.2. Desarrollo de la métrica y resultados

Un aspecto positivo sobre nuestra conjetura es que, de manera natural, se puede medir con invarianza bajo permutaciones de la significación. Anteriormente justificamos que, desde el punto de vista teórico, esta propiedad debería reflejarse en el estudio que hiciéramos de las funciones, pues tan válido era un orden escogido para representarlas como cualquier otro. En el caso de la distinción de pares cruzados, una permutación de la significación simplemente supone encontrar el foco y el contexto en posiciones diferentes de los bits. Pero esto no debería influir en el tratamiento de los pares a distinguir con salida cruzada, pues estas posiciones no deberían tener influencia en una métrica.

La distinción de pares cruzados de la figura 3.4.2 se detectaría con el orden de significación $(e_2, e_0, e_4, e_1, e_3)$ como puede verse en la figura 3.4.3.

Para tener invarianza bajo permutaciones de la significación, basta con tratar estas distinciones de pares de forma independiente de las posiciones que ocupen los bits del foco y del contexto, lo cual resulta natural debido a que estas posiciones no son importantes para computar la función. Que el foco ocupe la primera o la cuarta posición según nuestro orden es indiferente, lo importante es que cuando recibe un '1' o un '0' en ella, bajo el mismo contexto, tiene que hacer cosas diferentes. El circuito no entiende qué bit está recibiendo según nuestra representación, solo entiende si ese bit está a un '1' o un '0'.

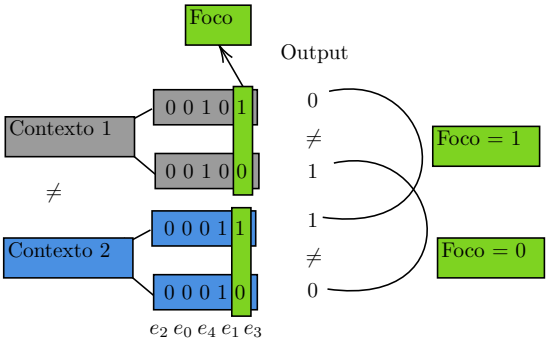


Figura 3.4.3: Dos pares de *inputs* a distinguir con igual foco y con salida "cruzada".

Se desarrollaron varias versiones de la métrica que pueden encontrarse en el notebook [14]. Una primera versión consistió en contar cuántas veces aparecía esta distinción de pares cruzados para cierta función, asignándole dicho valor como puntuación. En la figura 3.4.4 se reflejan sus distribuciones de puntuación.

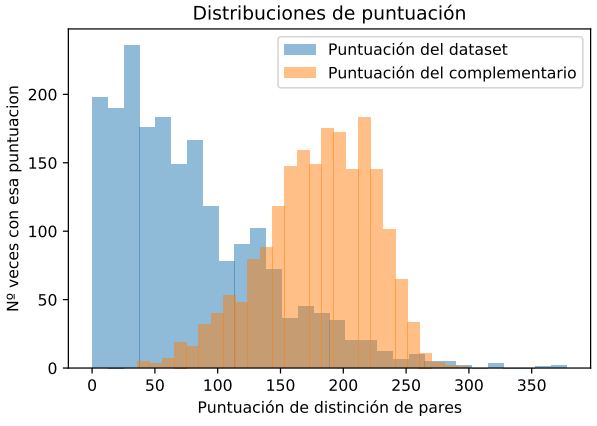


Figura 3.4.4: Métrica de distinción de pares para muestras de tamaño 2.000.

En base a la figura anterior, parece intuirse un buen funcionamiento de la métrica que justifica que la conjetura, en efecto, tiene sentido. Las funciones del dataset obtienen generalmente puntuaciones bajas y las del complementario puntuaciones más altas. Lo cuál es lógico, dado que esta métrica mide dificultad y debe puntuar más a funciones que requieran circuitos más "grandes".

Sin embargo, a diferencia de lo que ocurrió con la repetitividad, parecen existir bastante casos que son claramente "contraejemplo". Con la repetitividad apenas unos pocos casos del dataset lograban igualar la tónica habitual del complementario (y viceversa). Aquí los casos con mayor puntuación no pertenecen al complementario, como podría esperarse, sino al dataset y existe una cantidad considerable de ejemplos del dataset que llegan a igualar las puntuaciones habituales del complementario, aunque aparentemente sigue siendo minoritaria.

Las funciones del complementario que, en el fondo, representan la función promedio, tienen puntuaciones muy estables dentro de una franja de valores. Por su parte, las funciones del dataset tienen una clara tendencia a puntuaciones muy bajas.

Parece que el factor de distinción de pares cruzados no es tan determinante como el de repetitividad, pero que igualmente influye en el tamaño del circuito mínimo de una función. Una gran ausencia de esta distinción de pares cruzados parece implicar que la función puede computarse con un circuito "pequeño", pero una gran cantidad de tales distinciones no parece implicar la necesidad de un circuito "grande". A priori, esta métrica podría ser buena para complementar la que teníamos de repetitividad.

Repitiendo el mismo test clasificatorio que hicimos con la repetitividad (para cada función asignarle la clase cuya media de puntuaciones fuese más cercana), se obtuvo, sobre 2.000 nuevos ejemplos aleatorios, un 79.7 % de acierto para el dataset y un 84.8 % para el complementario. El peor comportamiento para el dataset tiene su explicación en esta mayor tendencia a casos "contraejemplo" que hemos indicado.

Sobre esta métrica se nos ocurrieron dos posibles mejoras. La primera consistía en ponderar más aquellas distinciones de pares con un foco de menor longitud. El ejemplo que vimos en la figura 3.4.1 requería fijarse únicamente en 1 bit e intuitivamente suponía hilar más fino que casos en los que el foco tuviese 2 o 3 bits. Para ello, se optó por sumar la longitud del contexto, en lugar de sumar 1 por cada distinción de pares cruzados. Así, una distinción con foco de 1 bit sumaría 4, con foco de 2 bits sumaría 3 y con foco de 3 bits sumaría 2.

La segunda mejora tenía que ver con la dificultad extra que puede haber en los contextos de la distinción de pares cruzados. Ya justificamos que para hacer las dos distinciones de pares de la figura 3.4.1 era necesario recurrir a los contextos, pero estos contextos a su vez tienen muchos bits en común que no se pueden usar para distinguir nada. En el caso de esa distinción de pares, como puede verse en la figura 3.4.5, necesariamente se usan los bits e_4 y e_1 de los contextos, pues los otros 3 bits coinciden en ambos. A más bits coincidentes en dichos contextos, en principio, mayor es la dificultad para diferenciar.

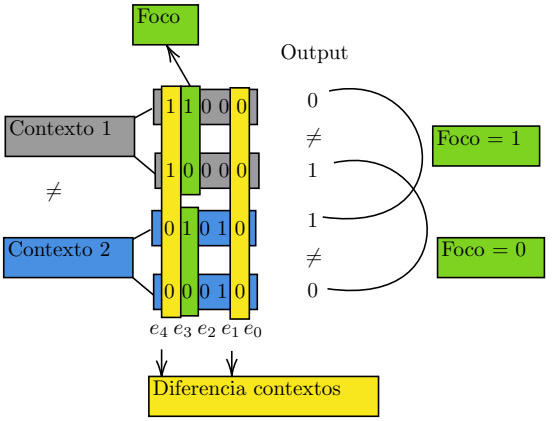


Figura 3.4.5: Diferencia de los contextos para distinguir los dos pares que con salida cruzada de la figura 3.4.1

Se probaron distintas combinaciones introduciendo las dos mejoras o solo una de ellas. Las mejoras tuvieron el efecto esperado y, tras varias ejecuciones con distintos tamaños de muestra, se llegó a la conclusión de que lo mejor era considerar ambas. Aunque la distribución de puntuaciones con ellas que puede verse en la figura 3.4.6 visualmente parece similar a la que obtuvimos sin mejoras, los porcentajes de acierto en nuestro test fueron mejores: 83.05 % para el dataset y 87.35 % para el complementario (siendo más o menos estables independientemente del tamaño de las muestras).

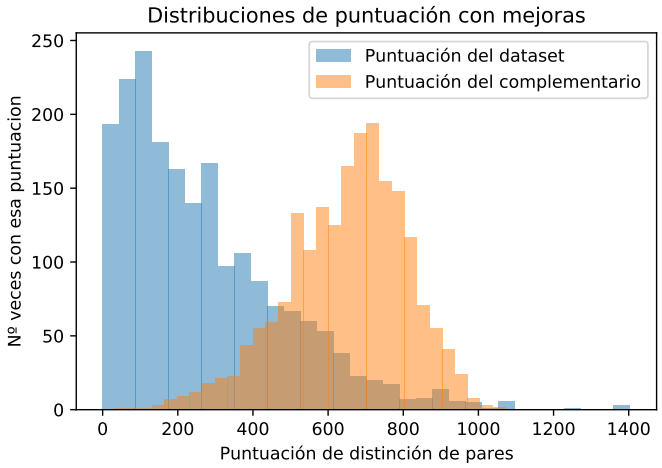


Figura 3.4.6: Métrica de distinción de pares con las dos mejoras para muestras de tamaño 2.000.

3.5. Primera combinación de métricas

Como ya comentamos, nuestra idea era hacer una combinación de las dos métricas anteriores formando una tercera métrica "combinación" que las mejorase. Un modelo automático podría hacer esta combinación de manera prácticamente óptima, pero antes de eso preferimos probar combinaciones más sencillas que pudiéramos entender fácilmente. En particular, estas combinaciones nos ayudarían a entender el peso de cada métrica y si nuestras conjeturas se fijan en aspectos distintas de las funciones tal y como pretendíamos al desarrollarlas (podría ocurrir que la repetitividad y la distinción de pares en el fondo reflejasen lo mismo y la combinación de métricas resultase similar a cada una por separado).

Como buscábamos algo sencillo, pensamos en una combinación lineal del tipo

$$\alpha \cdot \textit{puntuación de repetitividad} + \beta \cdot \textit{puntuación de distincion de pares} \tag{3.5.1}$$

para tratar de encontrar los valores de α y β que mejor funcionasen para diferenciar entre el dataset y el complementario. Las pruebas que vamos a comentar pueden encontrarse en [16].

Puesto que los rangos donde se movían las puntuaciones de repetitividad y distinción de pares son muy distintos, para no tener que buscar valores de α y β demasiado grandes o demasiados pequeños, se decidió reescalar los datos a un intervalo común antes de combinar. Puesto que las distribuciones visualizadas anteriormente no daban la sensación de responder a una campana de Gauss, se optó por normalizar las puntuaciones en lugar de estandarizarlas.

Para ello se llevaron, por separado, las puntuaciones de repetitividad y distinción de pares al intervalo $[0, 1]$ mediante la expresión:

$$\frac{X - X_{mín}}{X_{máx} - X_{mín}}$$

donde X es el conjunto de puntuaciones obtenidas para muestras del dataset y del complementario juntas[4].

Tras esta transformación de los datos sobre muestras aleatorias de 2.000 elementos del dataset, se obtuvieron las puntuaciones, para cada métrica, que pueden verse en las figuras 3.5.2 y 3.5.3.

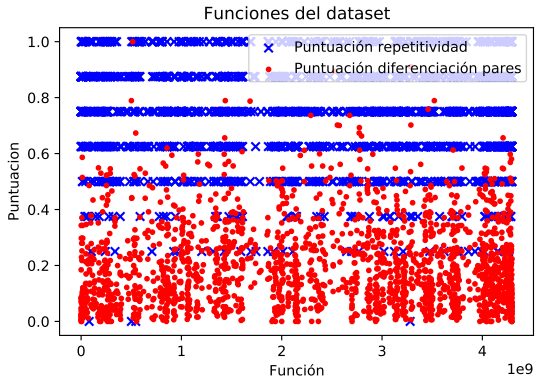


Figura 3.5.2: Puntuaciones del dataset reescaladas a $[0, 1]$.

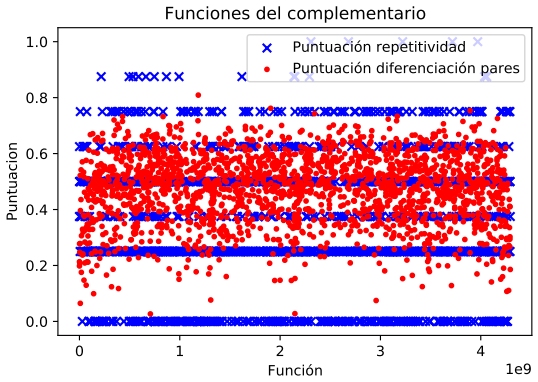


Figura 3.5.3: Puntuaciones del complementario reescaladas a $[0, 1]$.

Se aprecia un reescalado correcto, que respeta las proporciones para las distribuciones originales de las puntuaciones, como resultaba deseable.

A partir de ahí, se probó la combinación lineal descrita en la ecuación 3.5.1 para el valor $\alpha = 1$ con los valores $\beta \in \{0, -0.1, -0.2, -0.5, -1, -2, -5, -10\}$ y para el valor $\alpha = 0$ con $\beta = 1$. Con ello, se consideró la posibilidad de usar solo una de las métricas o combinarlas en las proporciones 1:-1, 1:-2, 2:-1, 1:-5, 5:-1, 1:-10 y 10:-1[5]. El signo negativo de la β se debe a que la repetitividad es mayor a mayor "sencillez" de la función y la distinción de pares es mayor a mayor "dificultad" de la función. Para que la combinación mida "sencillez" basta cambiar de signo del valor de la distinción de pares.

Los porcentajes de acierto para la métrica combinación, en función de los valores de α y β, con el test clasificatorio que usamos ya en las dos métricas por separado, pueden verse en la tabla 3.6.

Valor de α	**Valor de β**	**Acierto en dataset (%)**	**Acierto en complementario (%)**
0	-1	82.90	86.60
1	0	82.70	86.15
1	-0.1	82.70	86.15
1	-0.2	85.55	86.1
1	-0.5	89.80	87.30
1	-1	90.35	89.10
1	-2	88.40	90.10
1	-5	85.65	89.40
1	-10	84.65	88.30

Tabla 3.6: Porcentajes de acierto para el test clasificatorio con distintas combinaciones de las métricas.

Esta tabla recoge una mejora que parece más que suficiente para justificar los beneficios de la combinación de métricas. Con los valores $\alpha = 1$ y $\beta = -1$, el acierto en el dataset asciende al 90.35 %, frente al 82.90 %

[4]Si para calcular $X_{mín}$ y $X_{máx}$ lo hiciesemos por una parte con el dataset y por otra con el complementario, los valores mínimos para el dataset en la distribución de la figura 3.4.6 (en torno a 200) recibirían prácticamente la misma puntuación reescalada que los valores mínimos para el dataset en esa misma figura (en torno a 0). Estaríamos desplazando hacia la izquierda la distribución de color naranja de la figura 3.4.6 y mezclándola con la distribución azul.

[5]Nótese que lo único importante es la proporción al combinar. Con un nuevo reescalado a $[0, 1]$ obtendríamos lo mismo utilizando $\alpha = 10$ y $\beta = -2$ que usando $\alpha = 1$ y $\beta = -0.2$, ya que $\frac{10}{-2} = \frac{1}{-0.2} = -5$. Por tanto, nos da igual usar una que otra.

y 84.65 % de acierto utilizando solo repetitividad o distinción de pares, respectivamente. Esta importante diferencia se mantuvo bastante constante cambiando las muestras y su tamaño (unas décimas arriba o abajo se mantuvieron los porcentajes de la tabla 3.6).

Entre las distintas combinaciones, la mejor parecía ser la que usaba $\alpha = 1$ y $\beta = -1$ pues es la que más acertaba sobre el dataset (de hecho la clase más problemática de las dos al tener porcentajes de acierto, en general, bastante inferiores al complementario). Estos valores de α y β mantenían también un gran acierto para el complementario (un 89.10 %, bastante por encima del 86.60 % solo con distinción de pares). Esto nos dio cierta intuición de que lo óptimo para combinar linealmente las métricas era ponderarlas por igual.

Capítulo 4

Modelos de Aprendizaje Automático

Teniendo ya dos conjeturas medianamente sólidas que podíamos entender de forma clara (para ambas encontramos una explicación teórica sencilla a los buenos resultados de las métricas), parecía una buena idea pasar a utilizar Aprendizaje Automático. Por una parte podríamos optimizar la combinación de nuestras métricas y por otra trabajar directamente sobre los datos obtenidos con el generador en busca de más información sobre qué factores influyen en el tamaño mínimo asociado a una función. Para ello, se utilizaron diversos modelos de clasificación supervisada de la biblioteca *sklearn* de Python [4].

4.1. Combinación óptima de las métricas

Tras calcular las puntuaciones de las métricas de repetitividad y distinción de pares para el dataset completo y para la misma cantidad de funciones aleatorias del complementario (2.050.334 funciones), se probaron distintos modelos que combinasen las métricas. Estos modelos pueden encontrarse en [13].

Se empezó probando con redes neuronales. Los datos utilizados en las redes estaban etiquetados correspondientemente con las clases "dataset" y "complementario" (fichero [17]). Las redes fueron entrenadas con esa información (aprendizaje supervisado) pero los datos del fichero se dividieron en una parte de entrenamiento y una parte de test (67 % para entrenar y 33 % para testear), evitando así evaluar la red con los mismos datos de los que aprendió. Existía el riesgo de obtener puntuaciones demasiado buenas exclusivamente para los datos conocidos debido al sobreaprendizaje de los mismos.

Los resultados con varias redes, alterando el valor de los hiperparámetros de aprendizaje, el número de capas y el número de neuronas en cada capa fueron, a lo sumo, de un 92 % de acierto para el conjunto de test (tanto para el dataset como para el complementario). Esto no mejoraba demasiado los resultados que obtuvimos en la combinación lineal del capítulo anterior (donde ya alcanzamos en torno a un 90 % de acierto), y nos hacía intuir que la no linealidad de las combinaciones no resultaba demasiado útil (a más capas mayor es la no linealidad que puede expresar la red y no se apreciaba mejora al añadirle muchas capas).

Probando simplemente con una regresión logística (modelo lineal) obtuvimos casi lo mismo, un 91 % de acierto al predecir sobre el conjunto de test, con las matrices de confusión de las figuras 4.1.1 y 4.1.2.

Comprobamos los pesos ajustados por la regresión logística para esta predicción y apreciamos su proximidad a la proporción 1:-1 que vimos como destacada (al variar los valores de α y β) en el capítulo anterior. Los pesos ajustados por la regresión fueron 0.173 para la repetitividad y -0.168 para la distinción de pares.

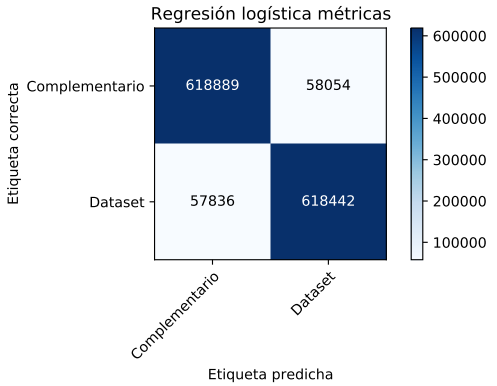


Figura 4.1.1: Matriz de confusión sin normalizar.

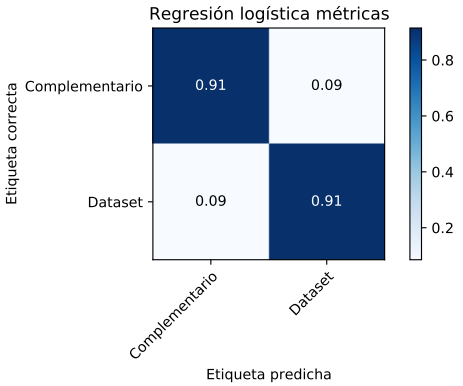


Figura 4.1.2: Matriz de confusión normalizada.

4.2. Aprendizaje directamente sobre los datos

También exploramos la opción de entrenar redes neuronales con los datos directamente salidos del generador. En este caso nuestro problema de clasificación tenía 32 variables binarias (el valor de la salida de la función para cada uno de los 32 posibles *inputs*), siendo las clases a predecir las mismas. De nuevo, se utilizaron 2.050.334 funciones aleatorias del complementario junto a todas las de nuestro dataset para realizar varias pruebas que pueden encontrarse en el notebook [12].

No teníamos mucha intuición sobre cómo debía ser la red porque, más allá de las métricas, tampoco teníamos mucho entendimiento de los datos. Tras probar con muchas configuraciones diferentes, apreciamos que funcionaban muy bien las redes con muchas neuronas en la primera capa y pocas capas neuronales (añadir capas como es lógico no empeoraba pero tampoco mejoraba). La primera capa es la encargada de fijarse en distintas situaciones de las variables de entrada y las demás son las que combinan esas situaciones para construir cosas cada vez menos lineales. Esto suscita que la complejidad de lo encerrado por nuestros datos no venía de la no linealidad sino de la cantidad de cosas a tener en cuenta.

En la figura 4.2.1 pueden verse las curvas de aprendizaje según el porcentaje de precisión obtenido para distintos valores del parámetro de regularización con 4 topologías de red distintas. Se aprecia que el aumento de neuronas en la primera capa es más útil que el aumento de capas. (Se pone este ejemplo con pocas neuronas para ver la tendencia debido a que, aunque quitemos parte del dataset, el entrenamiento es bastante lento).

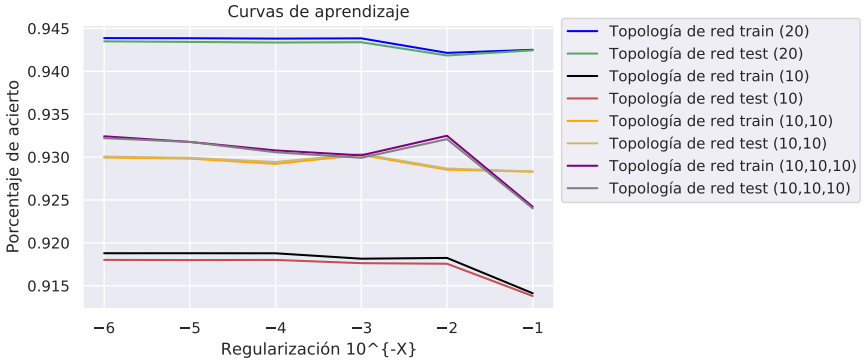


Figura 4.2.1: Curvas de aprendizaje para una capa oculta de 20 neuronas, una capa de 10, dos capas de 10 y tres capas de 10.

Una red con una sola capa intermedia de 200 neuronas alcanza una precisión del 97 % para el dataset y del 96 % para el complementario sobre el conjunto de test (partición con el 33 % de los datos para test y el 67 % para entrenamiento), con las matrices de confusión de las figuras 4.2.2 y 4.2.3.

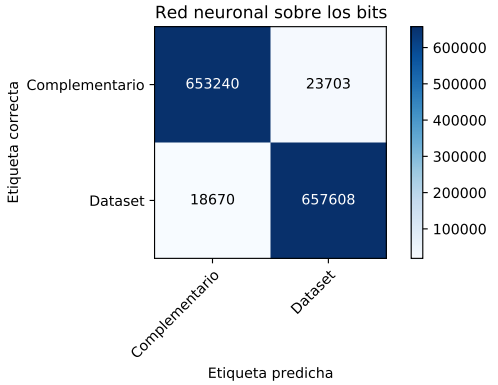


Figura 4.2.2: Matriz de confusión sin normalizar.

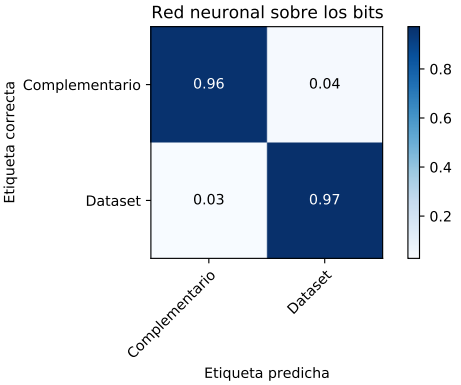


Figura 4.2.3: Matriz de confusión normalizada.

Esta red resultaba muy interesante al haber conseguido captar aspectos más sofisticados que nuestras métricas (a pesar del buen quehacer de las mismas), que podrían ser clave para determinar si una función requiere de un circuito "grande" para computarse o no. Si lo que esta red hubiese aprendido pudiera extrapolarse a otros tamaños de entrada (más allá de los 32 bits), podría, en un siguiente paso, utilizarse para probar algún tipo de resultado sobre el tamaño mínimo para computar una función que, idealmente, se podría utilizar para resolver el problema P vs NP. Una vez detectado el patrón aprendido por la red se podrían tratar de enunciar resultados del tipo: si una función de n entradas cumple cierta condición entonces son necesarias al menos $1.5^n/(n-1)$ puertas para desarrollar un circuito que la compute. Un resultado como ese supondría que una sucesión de tales funciones no está en $P_{/poly}$ y por tanto tampoco en P.

Sin embargo, no fuimos capaces de entender lo que la red había aprendido. La forma de acercarse a ese conocimiento era tratar de comprender los pesos que la red ajustaba para las distintas capas. Con la interpretación que dimos anteriormente, sobre que la primera capa mira cosas muy diversas y las demás aportan cada vez más no linealidad, tenía sentido fijarnos especialmente en los pesos de la primera[1]. Pero, por más que observamos los pesos de la matriz 32×200 de la primera capa, no fuimos capaces de sacar nada en claro. Los mapas de calor tanto de los pesos de la capa oculta y de la capa de salida, con los que se trató de entender la red, pueden encontrarse en el Anexo 1: mapas de calor de pesos de la red neuronal entrenada con el dataset.

Nos quedó la sensación de que descifrar lo que esa red había conseguido aprender podía ser fundamental, pero éramos incapaces de hacerlo. Para nada nos consideramos expertos en redes neuronales y, a las alturas del trabajo en que estábamos, tampoco teníamos mucho tiempo para convertirnos en ello. Tras un par de charlas con personas más entendidas en la materia, nos quedó claro que no había más forma de abordar el asunto que analizando los pesos de forma más sofisticada. Es una tarea interesante sobre la que se podría trabajar para tomar el relevo de este trabajo. No obstante, existe el riesgo de que aquello que la red encierra no siga ningún patrón claro, o que siga un patrón inentendible para un ser humano.

Tratando de obtener nuevas conjeturas con aprendizaje directamente sobre los datos, se probó con otros modelos que pudieran resultar más entendibles: regresión lineal, K-NN, árboles de decisión, etc. Los resultados fueron, en general, paupérrimos. Lo único que se salvó fueron los árboles de decisión, que obtuvieron en torno a un 85 % de acierto, lo que sigue estando por debajo del acierto de nuestras métricas.

[1] Que haya una capa oculta no significa que haya una sola capa en total, ya que siempre tenemos la capa de salida. Nosotros teníamos 2 matrices de pesos, una 32×200 para conectar las entradas con la neuronas intermedias y otra 200×1 para conectar las neuronas intermedias con la salida

Capítulo 5

Pruebas con problemas de *P* y *NP*

Para ponerle la guinda al trabajo nos pareció muy buena idea tratar de comprobar la escalabilidad de nuestras conjeturas sobre funciones con un tamaño de entrada n más grande. Como ya explicamos en la introducción, en el caso de que $P \neq NP$ las conjeturas obtenidas con nuestro trabajo podrían suponer factores de diferenciación entre las dos clases de complejidad. Por este motivo, era especialmente interesante trabajar sobre funciones asociadas a la tabla de verdad de problemas particularmente susceptibles de no estar en P que sí estuviesen en la clase NP.

Los problemas *NP-completos* nos parecieron una muy buena opción para ello. Por el momento no se conoce un algoritmo de coste polinómico para resolverlos y, por tanto, no se sabe si están en la clase P o no. De existir tal algoritmo para alguno de ellos, automáticamente habría una solución polinómica para todos los demás; es decir, si uno perteneciese a la clase P, todos los demás también. Parece intuirse que estos problemas no están en P pues, después de tanto tiempo pensando en ellos, ya se nos debería haber ocurrido una solución polinómica para resolver alguno. Por ello, su pertenencia a la clase P se antoja especialmente complicada y parecen problemas interesantes para tratar de cotejar posibles diferencias entre P y NP.

Nuestra idea era resolver algún problema *NP-completo* calculando, para cierto tamaño, la salida para cada posible instancia. Si codificamos las instancias de forma binaria con n bits y consideramos la versión de decisión del problema (aunque existan maneras de plantear el problema que no son de decisión, todo problema *NP-completo* se enuncia formalmente de manera que su respuesta sea "cierto" o "falso"), obtendríamos una tabla de verdad con 2^n valores binarios que supondría una función booleana $f : \{0,1\}^n \to \{0,1\}$ como las que venimos utilizando. Sobre esta función podríamos tratar de estudiar nuestras conjeturas aprovechando nuestras métricas y los modelos automáticos que construimos anteriormente.

El principal problema de todo esto es, de nuevo, computacional. Un n suficientemente grande como para poder representar de forma binaria instancias que no sean excesivamente triviales (para llegar a reflejar la dificultad del problema *NP-completo*) complica la resolución de las 2^n instancias posibles, pues el algoritmo conocido para ello es de coste superior a cualquier polinomio respecto de n. Además, supone un problema muy serio de cara a aplicar las métricas de forma exhaustiva sobre la función resultante, que será una cadena binaria de longitud 2^n. La métrica de distinción de pares, por ejemplo, tiene un coste en $\mathcal{O}(N^2)$ respecto al tamaño N de la función como cadena. Para un tamaño de cadena $N = 2^n$, como tenemos en este caso, resulta intratable si n no es ridículamente pequeño.

Por este motivo, el problema a resolver se escogió atendiendo a la capacidad de representar instancias medianamente grandes sin utilizar demasiados bits[1] y las métricas se aplicaron sobre muchas pequeñas muestras de las funciones. Esto se entenderá mejor en cuanto expliquemos la manera de actuar sobre *Cliqué*, el problema *NP-completo* que finalmente resolvimos.

5.1. Planteamiento de las tablas de verdad de *Cliqué* y *Paridad*

Cliqué es un problema *NP-completo* cuya versión de decisión consiste en, dado un grafo no dirigido de v vértices y un número natural positivo $k \leq v$, determinar si existe un subgrafo con exactamente k vértices que sea cliqué (cada vértice es adyacente a todos los demás).

En la figura 5.1.1 puede verse un ejemplo de un grafo con 5 vértices que contiene dos subgrafos cliqué de 3 vértices. En particular, quedándonos con solo uno o dos vértices de alguno de estos 3-cliqués[2] obtenemos

[1] SAT, que es quizás el problema *NP-completo* más famoso, no se escogió porque las representaciones que barajamos desperdiciaban muchos bits en representar los símbolos lógicos $\wedge, \vee$ y $\neg$. 3-SAT era mejor porque solo exige decidir las variables en cada literal y el $\neg$ de cada una. Pero, aún así, con 6 variables requiere de 3 bits para codificarlas y 1 bit para el $\neg$ que hacen un total de 4 bits por literal. Como hay 3 literales por cláusula son hasta 12 bits por cada cláusula disyuntiva.

[2] Un cliqué de k vértices es también llamado k-cliqué.

otro cliqué. Por tanto, para el grafo del ejemplo la respuesta del problema es cierta con $k \in \{1, 2, 3\}$. Además es falsa con $k \in \{4, 5\}$ ya que, por mucho que busquemos subgrafos, no encontraremos ninguno con al menos 4 vértices que sea cliqué.

El argumento anterior es cierto en general. Si un grafo contiene un k-cliqué entonces contiene un l-cliqué para todo $l \leq k$ (basta considerar subgrafos del k-cliqué). Para generar una tabla de verdad para cada $l \in \{1, 2, ..., v\}$ basta, para cada posible grafo g con v vértices, encontrar el k-cliqué más grande posible que sea subgrafo g. Así, la salida para g en la tabla de verdad con $l \leq k$ es un '1' y para la tabla de verdad con $l > k$ es un '0'.

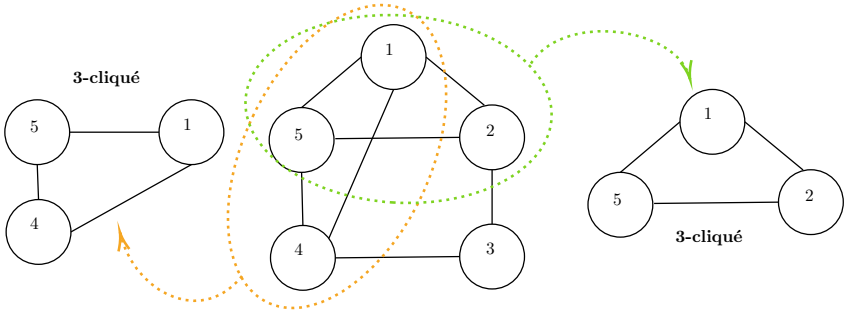


Figura 5.1.1: Dos subgrafos que son 3-cliqué de un grafo de 5 vértices.

Al llevar un tabla de verdad para cada posible tamaño de cliqué $l \in \{1, 2, ..., v\}$, las instancias a representar son simplemente los grafos. Un grafo viene dado por su matriz de adyacencia y, en caso de ser no dirigido como los de este problema, simplemente es necesaria su parte triangular inferior (la matriz es simétrica y la diagonal principal es innecesaria porque todo vértice es trivialmente adyacente a sí mismo). Por ejemplo, la parte relevante de la matriz de adyacencia para el grafo de la figura 5.1.1 sería:

$$\begin{pmatrix} \dots & \dots & \dots & \dots & \dots \\ a_{21} = 1 & \dots & \dots & \dots & \dots \\ a_{31} = 0 & a_{32} = 1 & \dots & \dots & \dots \\ a_{41} = 1 & a_{42} = 0 & a_{43} = 1 & \dots & \dots \\ a_{51} = 1 & a_{52} = 1 & a_{53} = 0 & a_{54} = 1 & \dots \end{pmatrix} \tag{5.1.2}$$

donde a_{ij} vale '1' si existe una arista que une al vértice i con el vértice j y '0' en caso contrario.

Si leemos por filas los valores '1' y '0' de la matriz obtenemos una cadena de $\{0, 1\}^{10}$ que nos sirve como representación binaria del grafo. En el caso de nuestro ejemplo tenemos que $(a_{21}a_{31}a_{32}a_{41}a_{42}a_{43}a_{51}a_{52}a_{53}a_{54})_2 = (1011011101)_2$ es la representación. Por tanto, en la posición $(733)_{10} = (1011011101)_2$ de cada tabla de verdad para $l \in \{0, 1, 2, 3\}$ se colocaría un '1' por lo que vimos en la figura 5.1.1 y en la misma posición $(733)_{10}$ de las tablas de verdad para $l \in \{4, 5\}$ se colocaría un '0'.

La idea es extrapolable a cualquier cantidad v de vértices aunque el ejemplo, por sencillez y claridad, sea solo con 5. Con $v = 5$ vértices acabamos de ver que necesitamos $10 = 5 \cdot 4/2$ bits para representar los grafos, habiendo tan solo $2^{10} = 1024$ instancias posibles. Nuestra capacidad para resolver instancias de *Cliqué* con un algoritmo de fuerza bruta de coste exponencial no era muy grande, pero sí podía alcanzar para algunos millones de instancias con $v > 5$. En general, para grafos de v vértices hacen falta $v \cdot (v - 1)/2$ bits para la representación binaria[3]. Con $v = 7$ son $21 = 7 \cdot 6/2$ bits que representan hasta $2^{21} = 2.097.152$ instancias, que es más o menos lo que aspirábamos a computar en un tiempo razonable. Decidimos, por tanto, calcular las tablas de verdad para *Cliqué* con grafos de 7 vértices.

Para comparar resultados calculamos también la tabla de verdad de un problema de la clase P. Escogimos el problema *Paridad* que explicamos al final de la sección 3.3.3 pues, como ya comentamos, era un buen *benchmark* de la clase P. Este problema, que consiste en determinar si el número de valores a '1' del *input* es

[3]Número de aristas posibles dividido por 2 ya que al ser no dirigidas es redundante contarlas en los 2 sentidos.

par o no, no está en la clase de complejidad AC^0. Esto significa que, con profundidad constante de circuito, necesita una cantidad de puertas superior a cualquier polinomio para resolverse, pero si no se limita la profundidad puede resolverse con una cantidad polinómica; por ello, es un problema falsamente difícil desde el punto de vista de los circuitos.

Se resolvió *Paridad* para las mismas instancias binarias $0, ..., 2^{21} - 1$ que *Cliqué*. Los programas en $C++$ que hacen los cálculos para resolver ambos problemas y sus ficheros de salida pueden encontrarse en [11].

5.1.1. Muestreo en las tablas de verdad

Como ya hemos comentado, resultaba imposible calcular la métrica de distinción de pares de manera exhaustiva sobre las funciones completas de nuestros problemas. Aunque la métrica de repetitividad, a priori, sí podía calcularse para cadenas de tamaño 2^{21}, era poco interesante hacerlo.

Los problemas *NP-completos* concentran su "dificultad" en zonas reducidas de su tabla de verdad, siendo en otras partes de la tabla incluso más triviales que algunos de los problemas más "fáciles" de la clase P. Por ejemplo, para el problema SAT esta zona "difícil" se encuentra donde la ratio entre cláusulas y variables es aproximadamente 4.26 (consultar [2]), siendo más "fácil" el problema cuanto más nos alejamos de ahí.

Considerar las tablas de verdad completas para el cálculo de las métricas sobre *Cliqué* no parecía la mejor idea. La "dificultad" de este problema podía quedar prácticamente oculta entre la trivialidad predominante en sus tablas. Si de verdad queríamos que saliesen a la luz las dificultades de los problemas *NP-completos* teníamos que fijarnos exclusivamente en sus zonas "difíciles". En esas zonas es donde debería residir la clave de una posible no pertenencia de *Cliqué* a la clase P. Para ser justos en nuestro estudio estas zonas "difíciles" de *Cliqué* se compararon con las zonas más "difíciles" de *Paridad*.

Es decir, si sabemos que, en caso de existir diferencia entre *Cliqué* y *Paridad*, estas residen en zonas concretas de especial "dificultad" ¿por qué no fijarnos solamente en ellas evitando que pasen desapercibidas en un estudio más global de las tablas de verdad? En el caso de *Cliqué* para v vértices, estas "dificultades" se encuentran con un tamaño de cliqué exigido $k \simeq v/2$.

Si el tamaño k de cliqué exigido es muy próximo a v la respuesta es casi siempre "falsa" puesto que apenas los grafos más densos pueden contener un cliqué tan grande. La tabla de verdad con $k = 7$ para nuestra resolución de *Cliqué* con $v = 7$ era todo '0' salvo la instancia 1^{21} correspondiente al grafo con todas las aristas posibles. Con $k = 6$ y $k = 5$ la cantidad de valores a '0' también era inmensamente mayor que la de valores a '1' (comprobarlo en los ficheros de salida de la carpeta [11]). Por contra, si el tamaño k exigido es muy próximo a 1 la respuesta es casi siempre "cierta" ya que prácticamente cualquier grafo tiene cliqués tan pequeños. Con $k = 1$ la tabla de verdad estaba íntegramente formada por el valor '1' (todo vértice es un 1-cliqué) y con $k = 2$ la cantidad de valores a '1' era inmensamente mayor que la de valores a '0'. La idea es que la dificultad dada por la k crece según nos acercamos por uno u otro lado al valor intermedio $k = v/2$.

Quedándonos con $k = 3$ o $k = 4$, que es la zona más difícil según el valor de k, seguimos teniendo instancias absolutamente triviales. Por ejemplo, el primer grafo de la figura 5.1.3 no contiene un 4-cliqué, porque tal cliqué contendría $4 * 3/2 = 6$ aristas y el grafo solo contiene 5. Mientras que el segundo grafo de la misma figura contiene un 4-cliqué porque apenas le faltan 3 aristas (a_{62}, a_{73} y a_{74}) para tener todas las posibles, lo que deja fuera de un cliqué como mucho a 3 vértices de los 7 que hay en total.

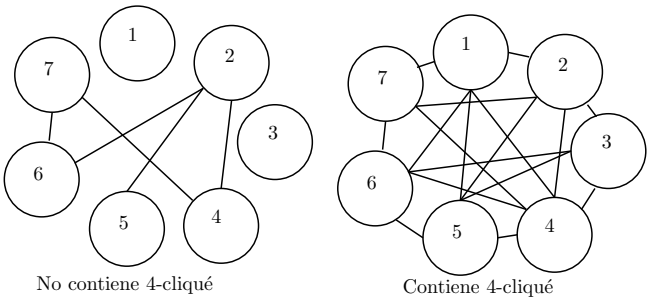


Figura 5.1.3: Grafos triviales para el posible *Cliqué* con $k = 4$.

Esto nos llevó a pensar que era mejor filtrar aún más en busca de la "dificultad" más pura del problema tomando como muestras ciertas subcadenas de las tablas de verdad. Una manera de determinar la "dificultad" de una subcadena era mediante la proporción de valores a '0' y a '1'. Al igual que razonamos para las cadenas completas con cada posible valor de k, podíamos razonar para un subcadena con un k concreto. En tal subcadena, muchos más valores a '0' que a '1' significan una muestra para la que casi ningún grafo contiene un k-cliqué y muchos más valores a '1' que '0' indica lo contrario, que casi todos los grafos de la muestra tienen un k-cliqué. Ambos casos resultan "fáciles" y nuestra intención era evitarlos tomando muestras con la cantidad de valores a '1' y a '0' suficientemente balanceada.

Para que estas muestras resultasen computables en un tiempo razonable con la métrica de distinción de pares y para poder utilizarlas en los modelos automáticos del capítulo 4, se tomó tamaño 32 para las subcadenas (los modelos se entrenaron para entradas de ese tamaño, que es el que tenían las funciones del dataset, y solo son aplicables para él).

Para que no hubiese bits "privilegiados" en esas muestras evitamos tomar las 32 salidas "difíciles" de manera siempre consecutiva en nuestra tabla de verdad. Para nuestra representación de los grafos el bit más significativo (correspondiente a a_{21}) nos dice si hay una arista entre los vértices 2 y 1, los bits segundo y tercero más significativos nos dicen si hay una arista entre los vértices 2 y 3 con el vértice 1, etc. (véase la matriz 5.1.2). Los 6 bits menos significativos (correspondientes a $a_{71}, ..., a_{76}$) solo nos indican si hay aristas que inciden en el vértice 7 y, por tanto, una muestra consecutiva de la tabla de verdad de tamaño $32 = 2^5$ solo altera la adyacencia de este vértice. Esta menor variedad en las parejas que representan estos últimos bits podría repercutir en la "complejidad local" haciendo que las muestras consecutivas resultasen falsamente "difíciles" o "fáciles".

Este privilegio de bits, debido a la significación al representar, puede evitarse fácilmente escogiendo 5 bits aleatorios entre los 21 que tienen los *inputs*. Estos 5 bits son los que se varían para considerar las $2^5 = 32$ posibilidades correspondientes a incluir o no las 5 aristas a las que representan. Al fijar un valor también aleatorio para los otros 16 bits correspondientes a las demás aristas se obtiene una muestra de tamaño 32 como la que necesitábamos. Con este procedimiento todas las aristas tienen la misma probabilidad de ser variadas evitando en general que tengan en común un mismo vértice.

Por ejemplo, si al elegir 5 bits aleatorios entre los 21 bits dados por a_{ij} con $i, j \in \{1, ..., 7\}$ e $i > j$, se obtienen $a_{41}, a_{43}, a_{54}, a_{73}$ y a_{76} y al elegir un valor binario aleatorio para los otros 16 bits se obtiene 1100101101000110 quedaría la muestra de tamaño 32 correspondiente a las salidas de los *inputs* que refleja la tabla 5.1.

a_{21}	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
a_{31}	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
a_{32}	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
a_{41}	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
a_{42}	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
a_{43}	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
a_{51}	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
a_{52}	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
a_{53}	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
a_{54}	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
a_{61}	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
a_{62}	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
a_{63}	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
a_{64}	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
a_{65}	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
a_{71}	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
a_{72}	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
a_{73}	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
a_{74}	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
a_{75}	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
a_{76}	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

Tabla 5.1: Ejemplo de muestra con los bits fijados en color naranja y los bits que varían en color verde.

Para que las muestras de *Cliqué* fuesen balanceadas y, por tanto, "difíciles" como pretendíamos, establecimos

el criterio 5.1.4.

$$\text{La muestra es útil} \iff |\text{El número de valores a 0} - \text{El número de valores a 1}| \leq 2 \tag{5.1.4}$$

Muestreando de la misma forma en *Paridad* teníamos incluso mejor balanceo al cumplirse para toda muestra que la cantidad de valores a '0' coincidía con la cantidad de valores a '1'[4]. Esto, en cierto modo, hacía que *Paridad* tuviese una ligerísima ventaja para resultar más "difícil" que *Cliqué* (lo contrario de lo que presumiblemente es cierto), al tener muestras algo más balanceadas.

5.1.2. Resultados para las métricas y modelos de aprendizaje

Tras calcular las métricas y aplicar los modelos sobre 90.000 muestras de *Cliqué* que con $k = 4$ cumplían el criterio 5.1.4 (había bastantes más muestran que cumplían el criterio con $k = 4$ que con $k = 3$) y hacer los propio con 100.000 muestras de *Paridad* se obtuvieron los siguientes resultados:

- **Para la métrica de repetitividad:** una puntuación media de 12.22 para las muestras de *Cliqué* y de 10.46 para las muestras de *Paridad*.

  Para la métrica de distinción de pares: una puntuación media de 629.26 para las muestras de *Cliqué* y de 600.67 para las muestras de *Paridad*.

- **Para la regresión logística entrenada con las métricas:** un 52.42 % de respuesta "Complementario" en las muestras de *Cliqué* y un 46.7 % de respuesta "Complementario" en las muestras de *Paridad*.

- **Para la red neuronal entrenada con los datos:** un 64.43 % de respuesta "Complementario" en las muestras de *Cliqué* y tan solo un 29.10 % de respuesta "Complementario" en las muestras de *Paridad*.

Recordemos que la métrica de repetitividad supuestamente otorga puntuaciones más altas a problemas más "fáciles" computables con circuitos más "pequeños". En este caso obtuvimos un resultado opuesto a lo esperado al tener mayor puntuación el problema *NP-completo* que el de clase P. La métrica de distinción de pares, que funciona al revés (mayor puntuación a mayor "dificultad"), sí parecía responder a lo esperado dándole algo más de puntuación al problema *NP-completo*.

El modelo logístico que utilizaba las métricas combinadas linealmente lograba, en cierta forma, captar la mayor "dificultad" de *Cliqué*, aunque de forma poco clara (52.42 % de respuesta de "complementario" en *Cliqué* frente a 46.42 % en *Paridad*). Esto parece achacable al mal funcionamiento de la métrica de repetitividad.

Sin duda lo más interesante es que la red que utiliza directamente la representación binaria de las funciones distingue las muestras de ambos problemas bastante bien. Identifica como "complementario" (tamaño de circuito supuestamente "grande") al 64.43 % de las de muestras de *Cliqué* frente a tan solo el 29.10 % de las muestras de *Paridad*.

[4]Nótese que el número de *inputs* con cantidad par de valores a '1' en los bits en verde de la tabla 5.1 coincide con el número de *inputs* con cantidad impar. Como los demás bits están fijos, este equilibrio perfecto se preserva considerándolos también.

Conclusiones

Terminamos haciendo un balance del trabajo y recogiendo algunas recomendaciones para posibles continuaciones del mismo. Unos resultados tan esperanzadores en relación a un problema del calibre de P vs NP, que en el fondo es lo que trata de abordar nuestro trabajo, bien justifican continuar a partir de él.

Cabe mencionar que los objetivos que nos marcamos al comienzo, que pueden verse en la sección 1.4, se vieron satisfechos con solvencia en su totalidad. Repasémoslos a la vez que explicamos cómo y por qué logramos cumplirlos.

Diseñamos un algoritmo generador que recorría crecientemente los circuitos de 5 entradas basándose en la idea de que todo circuito de tamaño n se obtiene al combinar circuitos de tamaño $< n$. A través de diversas sutilezas para evitar el cómputo de muchos circuitos inútiles y mediante la paralelización del generador se logró un algoritmo eficiente que proporcionó un dataset bastante grande. El no determinismo de la última etapa y la posibilidad de utilizar otros repertorios de puertas, como el que solo contiene la puerta *NAND*, se utilizaron astutamente para obtener un dataset con más de 2 millones de funciones booleanas computables con circuitos "pequeños". Esta cantidad de funciones, que prácticamente cuadriplicaba la del dataset que teníamos opción de reutilizar desde un principio, fue muy ventajosa al aplicar modelos de Aprendizaje Automático.

Utilizando este dataset, encontramos y cotejamos dos conjeturas sobre factores influyentes en el tamaño mínimo del circuito para computar una función. En base a los resultados obtenidos, prácticamente podemos concluirlas como ciertas.

Por una parte, comprobamos con una métrica muy simple la mayor tendencia de las funciones computables con circuitos "pequeños" (representadas por nuestro dataset) a mostrar patrones de longitud potencia de 2 repetidos en sus representaciones binarias. A este fenómeno, que denominamos "repetitividad", conseguimos darle una explicación teórica sencilla basada en una distorsión paulatina de los patrones potencia de 2 al aplicar cada vez más operaciones lógicas.

Por otra parte, conjeturamos la necesidad de incluir más puertas lógicas en los circuitos encargados de computar funciones que hacen distinciones "complejas" entre sus *inputs*. Nos centramos en el caso particular de una función con distintas salidas para *inputs* que comparten algunos de sus bits (bits "contexto"). Esta situación obliga a cualquier circuito que compute a la función a fijarse en los bits complementarios al "contexto" (bits "foco"). Yendo más allá y considerando que la función, en la situación de distinción anterior, tuviese salidas distintas para un mismo "foco" con diferentes "contextos", surge el fenómeno de "distinción de pares cruzados" que obliga a los circuitos a combinar el "foco" y el "contexto". Intuitivamente esto aumenta la complejidad del circuito, que tiende a necesitar más puertas para resolver esas situaciones. Formalizando la hipótesis de "distinción de pares cruzados" con diversas métricas, logramos cotejarla con éxito sobre el dataset.

La "repetitividad" ofreció un 82.15 % de acierto sobre el dataset y un 85.05 % sobre su complementario con un test clasificatorio muy simple. Y la "distinción de pares cruzados" un 83.05 % sobre el dataset y un 87.35 % sobre el complementario. Con ello, quedaron bastante ratificadas ambas conjeturas. Una posterior combinación de sus métricas optimizada mediante una regresión logística elevó el acierto de esta clasificación hasta un 91 % en ambas clases.

El entrenamiento de varias redes neuronales con los datos obtenidos directamente del generador mostró la existencia de factores influyentes para el tamaño mínimo de una función más allá de la "repetitividad" y la "distinción de pares cruzados". Una de esas redes alcanzó un 97 % de acierto diferenciando el dataset de su complementario, lo que nos daba a entender que había factores que se nos habían escapado. Por los resultados de la red para diversas topologías, llegamos también a la conclusión de que la dificultad de lo aprendido por la red no venía dada por una no linealidad excesiva sino por una cantidad enorme de aspectos en los que fijarse.

El altísimo acierto de la red, junto a nuestra incapacidad para entender lo aprendido por la misma (que achacamos fundamentalmente a la falta de tiempo), hacen muy interesante trabajar sobre ella para continuar la labor de este trabajo. Nos quedó la sensación de que descifrar lo que esa red había aprendido podría ser

fundamental para determinar cuándo una función necesita un circuito "grande" para computarse y cuándo puede computarse con uno "pequeño". Indirectamente, en esa red podría incluso estar la respuesta a P vs NP.

La aplicación final de nuestras métricas sobre un problema NP-*completo* (*Cliqué*) y un problema de la clase P (*Paridad*), con una tabla de verdad mas grande, nos dejó más luces que sombras. La dificultad computacional del experimento y el diseño *ad-hoc* de nuestras métricas y modelos para funciones con 32 *inputs* nos hacía presagiar una mala escalabilidad hacia problemas más grandes. Sin embargo, la obtención de resultados razonables al muestrear estas tablas por zonas "dificiles" nos dejó entrever que la dificultad global de los problemas NP-*completos* se refleja, al menos parcialmente, incluso en trozos muy pequeños de su tabla de verdad.

Aunque la métrica de "repetitividad" funcionó al revés de lo esperado (dio como problema más fácil al NP-*completo*), la distinción de pares lo corrigió, pues, al combinar métricas, casi un 6 % más de veces se clasificaron como "complementario" las muestras de *Cliqué* que las de *Paridad*. Lo más llamativo, que hace aún más tentadora la red neuronal, es que obtuvo resultados mucho más claros, respondiendo "complementario" para el 64.43 % de las muestras de *Cliqué* y tan solo para el 29.10 % de las muestras de *Paridad*. No solo es que la red suscite interés por el buen funcionamiento sobre nuestro dataset, sino que hay indicios para pensar que podría haber captado una diferencia entre los problemas en P y los problemas NP-*completos* (y por tanto, entre P y NP). Por supuesto, para poder llegar a una conclusión así, haría falta seguir observando esas diferencias entre muchos más problemas de ambos tipos.

Finalmente, queremos destacar como un buen complemento para esta lectura la memoria del trabajo de fin de grado en Matemáticas que nosotros mismos estamos desarrollando actualmente. Esta memoria incluirá mucho más rigor, recogiendo pruebas formales de algunos de los aspectos tratados en la presente lectura. Podrán encontrarse pruebas de la corrección y completitud del algoritmo generador, formalizaciones matemáticas de la invarianza bajo permutaciones y de las métricas, un pequeño inciso sobre unos árboles de decisión que se trataron en la parte de Aprendizaje Automático y la resolución y estudio de los problemas *Set Cover* y *Mayoría*. Sin embargo, no estarán presentes ni los detalles de implementación del generador, ni algunos experimentos como el realizado al final para *Cliqué*, ni muchos de los detalles e intuiciones (sobre todo relativos a las métricas) que hemos desarrollado a lo largo de la presente lectura. La otra memoria es una versión más directa y formal, pero menos explicativa, de algunos de los aspectos que aquí se han dado, que aporta cosas nuevas, pero omite algunas otras.

Conclusions

We now conclude by taking stock of the presented project and by compiling some recommendations for future work. Such encouraging results, in relation to such a problem as P vs NP, which actually is what our work is about, justify continuing the work from them.

It is worth mentioning that the objectives we set in the introduction, which can be seen in section 1.4, were completely and successfully satisfied. In the following, we review them while we explain how and why we fulfilled them.

We designed a generator algorithm that goes increasingly through the five-entries circuits, based on the idea that every circuit of size n is obtained by combining circuits of a lower size than n. Through many different subtleties to avoid computing useless circuits, and by parallelizing the generator, we obtained an efficient algorithm that provided us with a quite large dataset. The non-determinism of the last step and the possibility of using other gates sets (e.g. a set with only the *NAND* gate), were wisely used to obtain a dataset with more than two million of computable Boolean functions with *small* circuits. Such large amount of functions, that almost was four times larger than the dataset we were initially proposed to reuse, was really advantageous when applying Machine Learning models.

By using this dataset, we found and collated two conjectures about influencing factors in the minimum size of the circuit that computes a function. Based on results obtained, we can practically conclude them as true.

On the one hand, we checked, with a simple metric, the greatest tendency of the computable functions with *small* circuits (represented by our dataset) to show patterns with powers-of-two length repeated in their binary representation. We achieved the development of a simple theoretical explanation to this phenomenon, that we named as *repeatability*, based on a gradual distortion of the powers-of-two patterns since we are applying more and more logical operations.

On the other hand, we conjectured the need of including more logic gates in the circuits that are in charge of computing functions that perform *complex* filterings in their inputs. We focused on the particular case of a function with distinct outputs for the inputs that share some of their bits (*context* bits). This situation imposes that every circuit that computes the function must check the complementary bits of the *context* bits (*focus* bits). Moreover, if the function considered before had distinct outputs for the same *focus* but different *contexts*, the phenomenon of *distinction of crossed pairs* arises, that forces the circuits to combine the *focus* and the *contexts*. Intuitively, this increases the complexity of the circuit, which would need more gates in order to solve those situations. Formalizing the *distinction of crossed pairs* hypothesis with several metrics, we successfully collated the hypothesis against our dataset.

The *repeatability* resulted in a 82.15 % of accuracy against the dataset, and a 85.05 % against its complements, with a very simple classifying test. Furthermore, the *distinction of crossed pairs* resulted in 83.05 % of accuracy against the dataset, and 87.35 % against its complements. Thanks to that, both hypotheses were quite endorsed. A later combination of their metrics, optimized with a logistic regression, raised the classification hit to a 91 % in both classes.

The training over the neural networks with the data directly gathered from the generator showed the existence of influential factors for the minimum size of a function, apart from *repeatability* and *distinction of crossed pairs*. One of those neural networks reached a 97 % of accuracy distinguishing between the dataset and its complements, which suggested that we had missed some factors. Due to the outputs of the net for different topologies, we concluded that the difficulty of what the net had learned was not related to an exceeded non-linear aspect, but for a wide range of aspects to check.

The high accuracy of the neural network, along with our lack of understanding of its learning (that we mainly blame for the lack of time), makes this project interesting and worthy to continue investigating it. We felt that understanding that learning would have been fundamental to determine when a function needed a *big* circuit to compute, and when a *small* one. Indirectly, the answer to P versus NP could be hidden in that net.

The final application of our metrics to an *NP-complete* problem (*Clique*) and a P problem (*Parity*), together with a greater truth table, let us more lights than shadows. The computational complexity of the experiment,

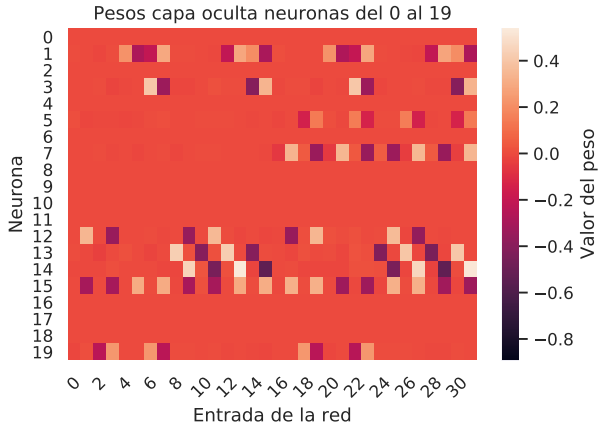
and the ad-hoc design of the metrics and models for functions with 32 inputs, led us to think that we would encounter scalability problems. However, the reasonable results we obtained when we sampled the tables through *hard* zones, show that the global difficulty of *NP-complete* problems is reflected, at least partially, even in tiny regions of their truth tables.

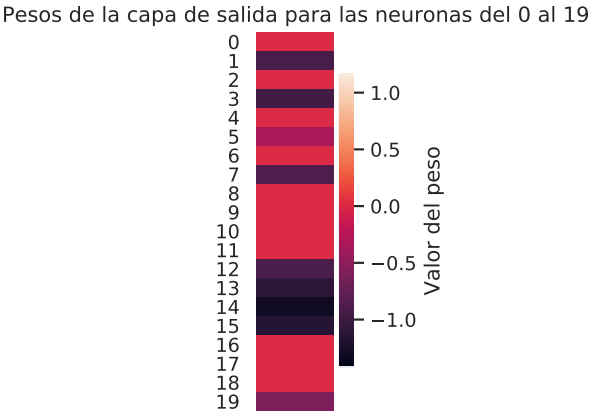
Although the metric for *repeatability* worked the other way around than expected (concluded that the easier problem was the *NP-complete*), the *distinction of crossed pairs* fixed it, since, when combining the metrics, the samples of *Clique* were classified as *complementary* almost more than a 6 % of the times the *Parity* samples were. The most remarkable thing, that makes this neural network more appealing, is that it obtained much clearer results by answering *complementary* for the 64.43 % of the *Clique* samples, and only the 29.10 % for the *Parity* samples. Not only does the net raise interest because of its excellent performance against our dataset, but also we realize that it could have noticed a difference between the P and *NP-complete* problems (and, therefore, between P and NP). Of course, to conclude and prove such claim, we should keep observing and investigating those differences in many more problems of both types.

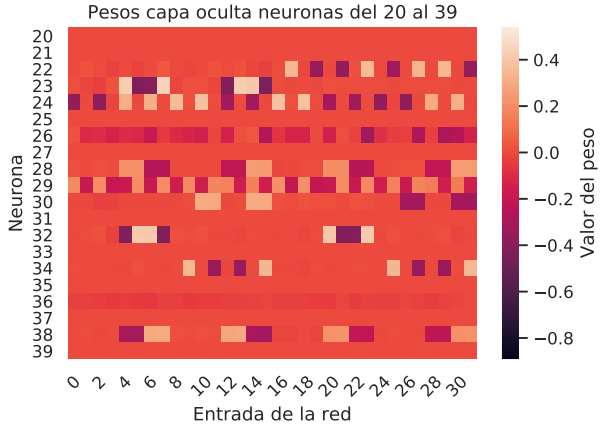
Finally, we would like to highlight the Final Degree Project in Mathematics, which we are also writing, serves as a supplementary document to this one. That research includes rigorously-stated formal proofs of many of the statements introduced in this work. For instance, it will prove completeness and correctness of the generator algorithm, a mathematical formalization of both the invariant property under permutations and the metrics, a brief research on the decision trees used for the Machine Learning approach, and the review and resolution of the *Set Cover* y *Majority* problems. However, we will not include the implementation details of the generator, nor some experiments such as the one carried out for the *Clique* problem, nor any intuition, related to the metrics, which we have developed throughout this work. The other Final Degree Project will address formally and directly some of the aspects that we have introduced here, but that we did not fully cover.

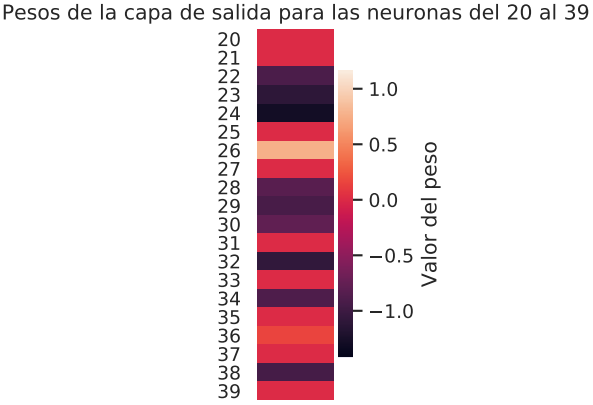
Anexos

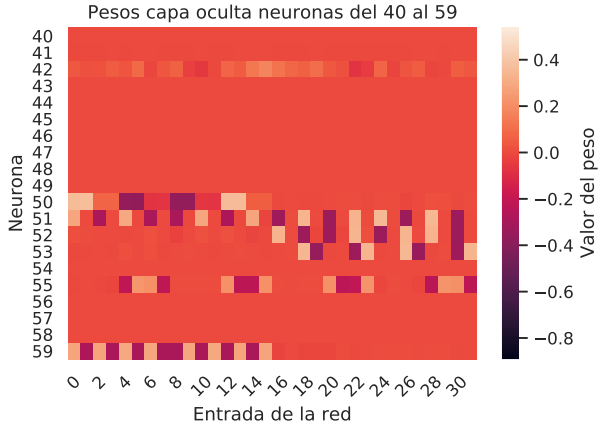
Anexo 1: mapas de calor de pesos de la red neuronal entrenada con el dataset

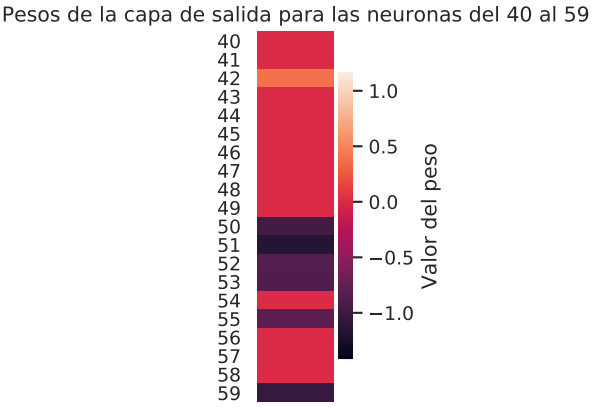


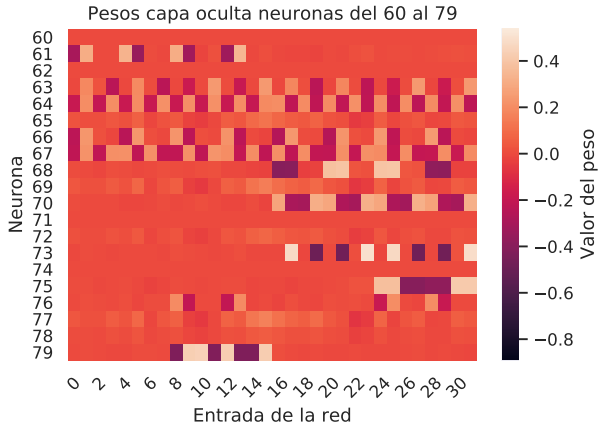


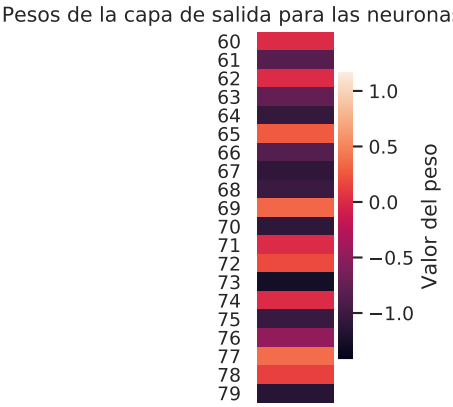


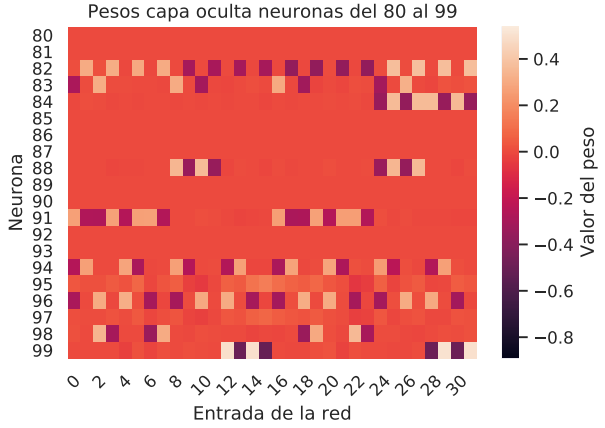


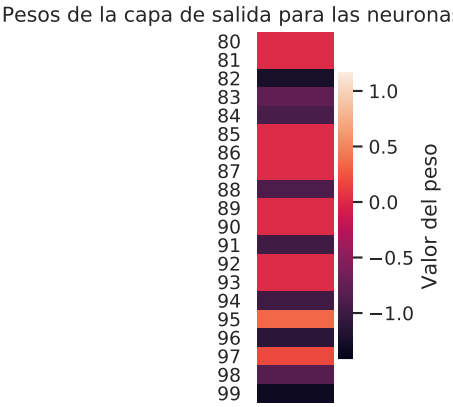


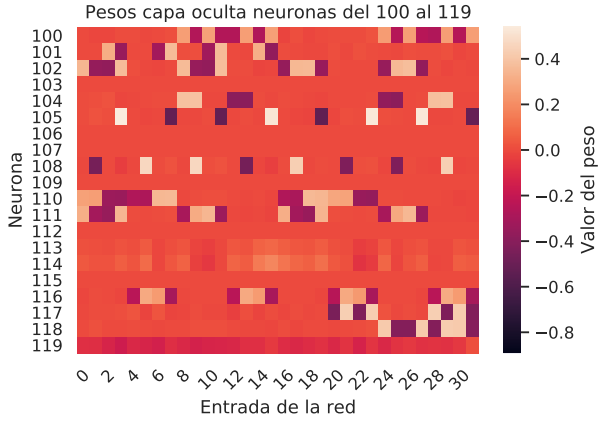


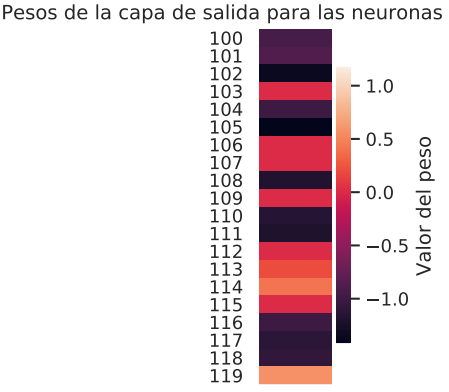

Pesos capa oculta neuronas del 60 al 79
Neurona
Entrada de la red
Valor del peso

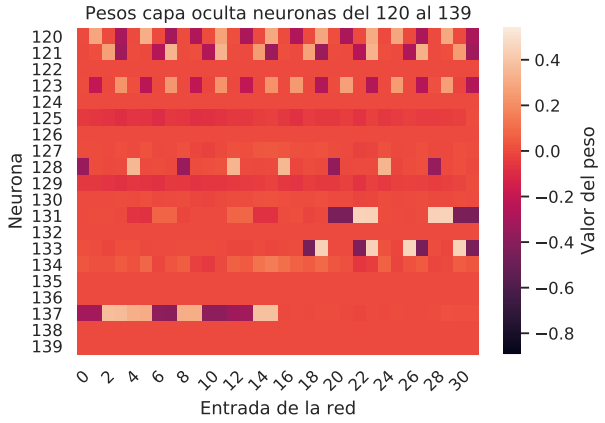

Pesos de la capa de salida para las neuronas del 60 al 79
Valor del peso

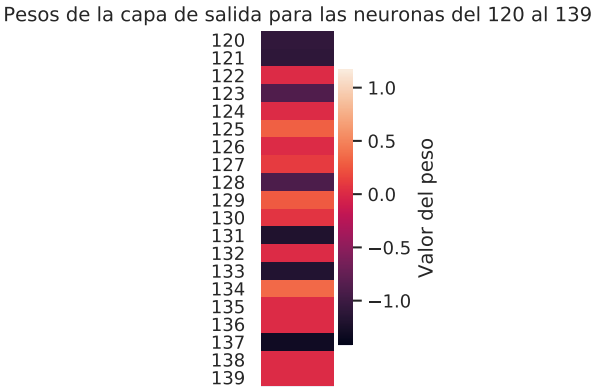

Pesos capa oculta neuronas del 80 al 99
Neurona
Entrada de la red
Valor del peso

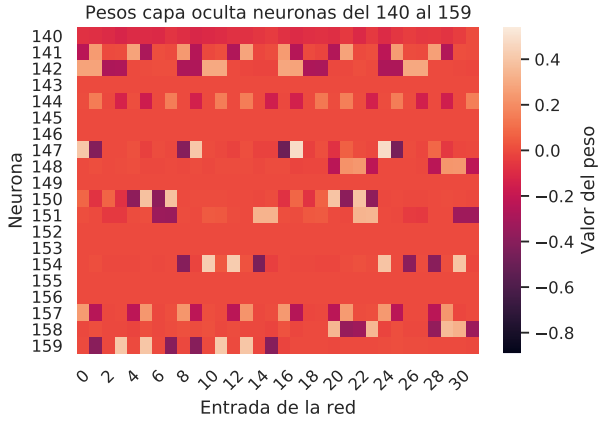

Pesos de la capa de salida para las neuronas del 80 al 99
Valor del peso

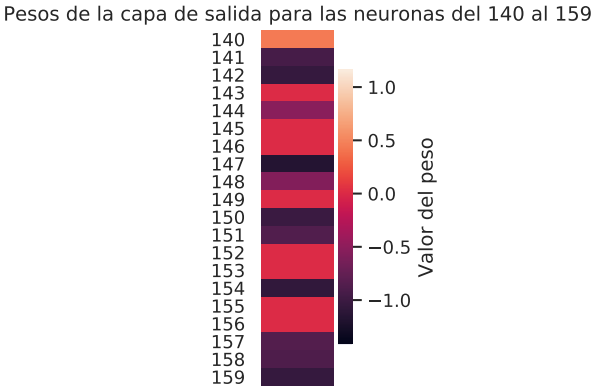

Pesos capa oculta neuronas del 100 al 119
Neurona
Entrada de la red
Valor del peso

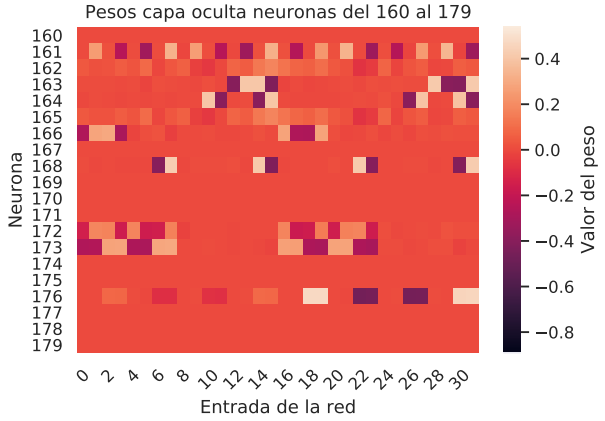

Pesos de la capa de salida para las neuronas del 100 al 119
Valor del peso

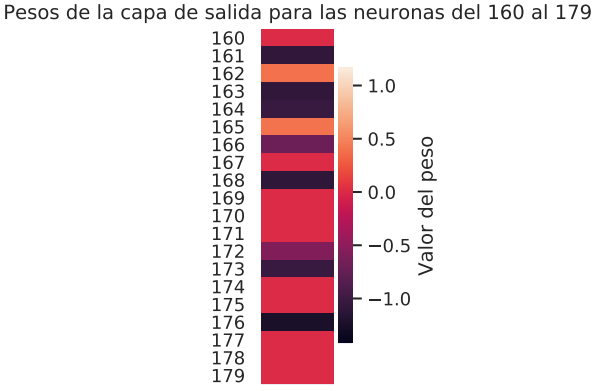

Pesos capa oculta neuronas del 120 al 139
Neurona
Entrada de la red
Valor del peso

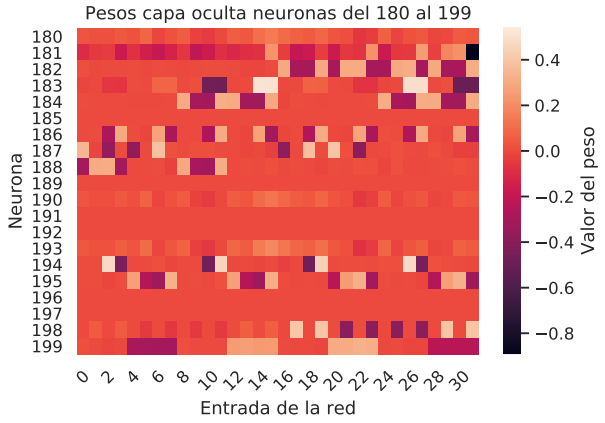

Pesos de la capa de salida para las neuronas del 120 al 139
Valor del peso

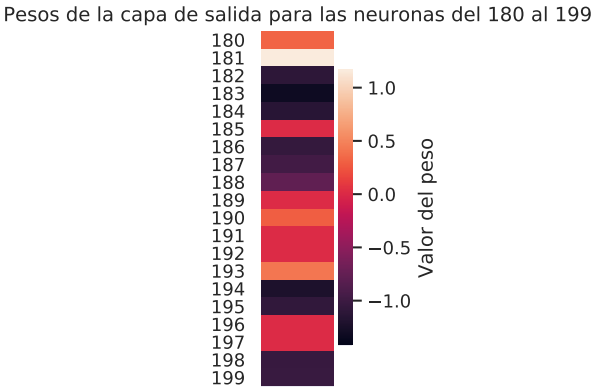

Pesos capa oculta neuronas del 140 al 159
Neurona
Entrada de la red
Valor del peso


Pesos de la capa de salida para las neuronas del 140 al 159
Valor del peso


Pesos capa oculta neuronas del 160 al 179
Neurona
Entrada de la red
Valor del peso


Pesos de la capa de salida para las neuronas del 160 al 179
Valor del peso


Pesos capa oculta neuronas del 180 al 199
Neurona
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
Entrada de la red
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
Valor del peso


Pesos de la capa de salida para las neuronas del 180 al 199
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
1.0
0.5
0.0
−0.5
−1.0
Valor del peso

Volver

Bibliografía

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

[2] 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.

[3] William Gasarch. "The P=?NP poll". En: *SIGACT News* 33 (ene. de 2002), págs. 34-47. URL: https://www.researchgate.net/publication/292393040_The_PNP_poll.

[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 (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.

[8] Jorge Villarrubia Elvira (Jorgitou98). *Fichero de salida del generador completo para 500 millones de circuitos*. URL: https://github.com/Jorgitou98/TFG/blob/main/Generador/ficherosParaMezclar/500millonesCompleto.txt.

[9] Jorge Villarrubia Elvira (Jorgitou98). *Fichero de salida del generador con NAND para 500 millones de circuitos*. URL: https://github.com/Jorgitou98/TFG/blob/main/Generador/ficherosParaMezclar/ficheroSalida500MNAND.txt.

[10] Jorge Villarrubia Elvira (Jorgitou98). *Generador de circuitos para el repertorio completo de puertas*. 2020. URL: https://github.com/Jorgitou98/TFG/blob/main/Generador/GeneradorTodasPuertas.cpp.

[11] Jorge Villarrubia Elvira (Jorgitou98). *Programas y salidas de la resolución de Cliqué y Paridad*. URL: https://github.com/Jorgitou98/TFG/tree/main/CliqueYParidad.

[12] 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.

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

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

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

[16] Jorge Villarrubia Elvira (Jorgitou98). *Pruebas para la combinación de métricas*. URL: https://github.com/Jorgitou98/TFG/blob/main/Metricas/combinacionMetricas.ipynb.

[17] Jorge Villarrubia Elvira (Jorgitou98). *Resultados de las métricas para el dataset y millones de casos alteaorios del complementario*. URL: https://github.com/Jorgitou98/TFG/blob/main/Metricas/salidaMetricas.csv.