Búsqueda de circuitos mínimos para computar el problema CLIQUE
Search for minimum circuits to compute the CLIQUE problem

Trabajo de Fin de Grado
Curso 2023–2024

Autor
Jaime Jacobo Romo González

Directores
Ismael Rodríguez Laguna
Narciso Martí Oliet

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

27 de mayo de 2024

Agradecimientos

A mis tutores, Ismael y Narciso, por ayudarme y guiarme para hacer posible el desarrollo de este trabajo. A Ignacio, por ayudarme con la literatura sobre circuitos. A todos los amigos que me han acompañado estos últimos años. A mi familia, por estar siempre conmigo y por el apoyo que me han brindado. Muchas gracias a todos.

Resumen

Búsqueda de circuitos mínimos para computar el problema CLIQUE

A lo largo de las últimas décadas se ha intentado abordar el problema $\mathcal{P}\ vs\ \mathcal{NP}$ de varias maneras. Una de las más interesantes es a través de funciones y circuitos booleanos. Con el fin de aumentar el conocimiento de este campo, presentamos en este texto un análisis sobre cómo se comportan los circuitos que computan el problema clique. Desarrollamos en detalle uno de los grandes resultados sobre circuitos monótonos: cualquier circuito monótono que compute clique necesita una cantidad superpolinómica de puertas. Después, estudiamos cómo se comportan dos métricas bajo el problema del clique. Analizamos una métrica semántica que se centra en la precisión de los resultados y una métrica sintáctica que se centra en las estructura de las funciones. Por último, realizamos experimentos y simulaciones con ambas métricas en grafos de 8 vértices y cliques de tamaño 4.

Palabras clave

Función booleana, circuito booleano, clique, métrica, $\mathcal{P}_{/poly}$, $\mathcal{P}\ vs\ \mathcal{NP}$, cota inferior, complejidad.

Abstract

Search for minimum circuits to compute the CLIQUE problem

Over the last decades, several attemps have been made to address the $\mathcal{P}$ *vs* $\mathcal{NP}$ problem. One of the most interesting ways is through boolean functions and circuits. In order to enhance the knowledge in this area, we present in this text an analysis about how the circuits that compute the clique problem behave. We elaborate on one of the great results in monotone circuits: every monotone circuit that computes the clique problem needs a superpolynomial amount of gates. Next, we study hoy two metrics behave under the clique problem. We analyze a semantic metric that focuses on the accuracy of results and a syntactic metric that focuses on the structure of functions. Finally, we perform experiments and simulations with both metrics on 8-vertex graphs and cliques of size 4.

Keywords

Boolean function, boolean circuit, clique, metric, $\mathcal{P}_{/poly}$, $\mathcal{P}$ *vs* $\mathcal{NP}$, lower bound, complexity.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación y objetivos

El problema $\mathcal{P}$ *vs* $\mathcal{NP}$ es uno de los problemas abiertos de la informática más importantes de nuestros días. Resolverlo supondría no solo solucionar un Problema del Milenio, sino también encontrar dónde está la frontera entre lo que se piensa que son problemas fáciles y difíciles, o demostrar que dicha frontera realmente no existe.

Durante las últimas décadas se han abordado diferentes caminos para intentar resolver este problema. Una forma común de razonar en informática es mediante argumentos diagonales (como por ejemplo el problema de la parada). Así que no parece descabellado intentar abordar $\mathcal{P}$ *vs* $\mathcal{NP}$ mediante argumentos diagonales. Por desgracia, expertos en el tema han observado que resolver $\mathcal{P}$ *vs* $\mathcal{NP}$ solo con un argumento diagonal es una tarea imposible.

Los circuitos y funciones booleanos permiten observar con detalle cómo es un problema de decisión, alejándose de estos argumentos diagonales. Esto los convierte en buenos candidatos para intentar abordar $\mathcal{P}$ *vs* $\mathcal{NP}$, pues dejamos de ver los problemas como oráculos que reciben una entrada y devuelven una salida. Gracias a la naturaleza de los circuitos, podemos examinar la estructura interna de los problemas e intentar entender qué los hace diferentes entre ellos.

Ligada al concepto de circuito booleano se encuentra la clase de complejidad $\mathcal{P}_{/poly}$. Esta clase agrupa aquellos problemas de decisión que pueden ser computados mediante circuitos de tamaño polinómico. A partir de la clase $\mathcal{P}_{/poly}$ se buscan cotas inferiores para circuitos que computan ciertos problemas. Si se encontrara un problema en $\mathcal{NP}$ que necesitara de circuitos de tamaño superpolinómico, se probaría que $\mathcal{P} \neq \mathcal{NP}$, pues se cumple $\mathcal{P} \subset \mathcal{P}_{/poly}$.

Para encontrar cotas inferiores sobre circuitos podemos utilizar el concepto de métrica. Si conseguimos acotar el crecimiento máximo que puede experimentar una función en un circuito, podremos encontrar una cota inferior para el tamaño del mismo.

En resumen, juntando funciones, circuitos y métricas, tenemos una estrategia

que nos permite abordar $\mathcal{P}$ *vs* $\mathcal{NP}$.

Sin embargo, aunque este enfoque parece prometedor, no ha habido grandes avances en este campo durante los últimos años. Es por ello que cualquier aportación se considera importante, pues puede ayudar a entender mejor qué ocurre realmente con los circuitos y a generar nuevas ideas.

Este texto se centra en el problema del clique, un problema clásico de computación. Se estudiará la estructura interna de los circuitos que calculan este problema, y cómo se comportan bajo ciertas métricas.

1.2. Plan de trabajo

Este trabajo comenzó con una idea: "El producto cartesiano que provocan las puertas AND se lleva mal con el problema del clique". Al hacer el AND de dos funciones en forma normal disyuntiva, el número de cláusulas puede llegar a crecer cuadráticamente con respecto al tamaño inicial de las funciones. Pero, si estamos computando clique, a medida que una función se vaya pareciendo más a la deseada, llegará un momento en el que la mayoría de las cláusulas producidas por cada AND serán inútiles o perniciosas, pues la mayoría de las cláusulas incluirán demasiados literales para identificar correctamente cliques. Dado que cada puerta OR hace crecer el número de cláusulas linealmente, solo las puertas AND podrían producir la cantidad exponencial de cláusulas necesarias para computar clique usando pocas puertas. Pero si no pueden por el efecto anterior, entonces podrían hacer falta muchísimas puertas para computar clique. Si en particular dicha cantidad fuera superpolinómica, entonces el problema clique no estaría en $\mathcal{P}_{/poly}$, por tanto tampoco en $\mathcal{P}$, y al ser clique un problema $\mathcal{NP}$-completo, trendríamos $\mathcal{P} \neq \mathcal{NP}$.

Bajo esta idea, se desarrolló el siguiente plan de trabajo:

- Estudio de la literatura relacionada con funciones y circuitos booleanos, en particular el capítulo 6 del libro *Computational Complexity: A Modern Approach* [2]. En este capítulo se desarrollan con detalle los conceptos de función booleana, circuito booleano y la clase de complejidad $\mathcal{P}_{/poly}$. En el capítulo 2 se explican los avances y conocimientos del campo hasta la fecha.
- En la sección 3.1 del capítulo 3 se explica y analiza más en detalle el concepto de métrica, sobre el que se centra el resto del texto.
- Estudio del resultado que obtuvo Razborov [11]. Razborov demostró que, si eliminamos las negaciones de un circuito, calcular clique requiere una cantidad superpolinómica de puertas. Como este trabajo se centra en el problema del clique, procedía estudiar en detalle este resultado. Tras una lectura del mismo, comprobamos que las ideas subyacentes eran muy similares a la idea de partida de este trabajo. En la sección 3.2 del capítulo 3 se cuenta en detalle este resultado.
- Explorar los aspectos en los que la idea inicial no coincidía con la de Razborov. En el capítulo 4 se estudia el comportamiento de clique sobre dos métricas: una

semántica frente a otra sintáctica. En ambas métricas se realizan experimentos con la puerta AND y se analizan los resultados observados. El código utilizado para generar los datos de las tablas y las gráficas se encuentra en el repositorio público [12].

- En el capítulo 5 se resumen los resultados obtenidos, así como su discusión y algunas ideas sobre cómo continuar con el análisis realizado.

Introduction

1.3. Motivation and Objectives

$\mathcal{P}$ *vs* $\mathcal{NP}$ is one of the most important open problems in computer science nowadays. Solving it would involve not only solving a Millennium Problem, but also finding where the boundary is between what are thought to be easy and hard problems, or proving that such a boundary does not exist.

During the last decades different paths have been explored to try to solve this problem. A common way of reasoning in computer science is by diagonal arguments (for example, the halting problem). Thus, it does not seem unreasonable trying to approach the $\mathcal{P}$ *vs* $\mathcal{NP}$ problem trough diagonal arguments. Unfortunately, experts in the area have noticed that solving the $\mathcal{P}$ *vs* $\mathcal{NP}$ problem by relying only on a diagonal argument is an impossible task.

Boolean circuits and functions allow us to closely observe how a decision problem is, avoiding these diagonal arguments. It is because of that why they are great candidates for trying to approach the $\mathcal{P}$ *vs* $\mathcal{NP}$ problem, as we no longer see the problems as oracles that receive an input and return an output. Thanks to the nature of circuits, we can examine the internal structure of problems and try to understand what makes them different from each other.

Related to the boolean circuit concept we find the complexity class $\mathcal{P}_{/poly}$. This class groups those decision problems that can be computed by polynomial-sized circuits. Starting from the $\mathcal{P}_{/poly}$ class, lower bounds are sought for circuits that compute certain problems. If only one $\mathcal{NP}$ problem that requires superpolynomial-sized circuits were found, it would be proved that $\mathcal{P} \neq \mathcal{NP}$, since $\mathcal{P} \subset \mathcal{P}_{/poly}$ stands.

To find lower bounds on boolean circuits we can use the metric concept. If we can bound the maximum increase a function can experiment through the gates of a circuit, we could find a lower bound for the size of the circuit.

In summary, by gathering boolean functions, circuits and metrics, we have a strategy that allows us to approach the $\mathcal{P}$ *vs* $\mathcal{NP}$ problem.

However, although this approach looks promising, there have been no major advances in this field over the last few years. That is why any contribution will be considered important, as it might help to better understand what really happens

with circuits and to brainstorm new ideas.

Thist text focuses on the clique problem, a classic computational problem. We will study the internal structure of circuits that compute this problem, as well as how they behave under certain metrics.

1.4. Work Plan

This work started with an idea: "The cartesian product caused by AND gates gets on badly with the clique problem". By making the conjunction of two functions in disjunctive normal form, the number of clauses can grow quadratically with respect to the initial size of the functions. But, if we are computing clique, as a function becomes more and more similar to the desired one, it will come a time when most of the clauses produced by each AND gate will be useless, since most clauses will contain too much literals to correctly identify cliques. Since each OR gate produces a linear increase in the number of clauses, only AND gates could produce the exponential number of clauses needed to compute clique using few gates. However, if they cannot becasue of the above effect, then it could take lots of gates to compute clique. If particularly this quantity were superpolynomial, then the clique problem would not be in $\mathcal{P}_{/poly}$, hence neither would it be in $\mathcal{P}$, and as the clique problem is a $\mathcal{NP}$-complete problem, one would have $\mathcal{P} \neq \mathcal{NP}$.

Based on this idea, the next work plan was developed:

- Study of the literature related to boolean functions and circuits, especially chapter 6 of the book: *Computational Complexity: A Modern Approach* [2]. In this chapter, boolean function, boolean circuit and $\mathcal{P}_{/poly}$ complexity class concepts are developed in detail. Chapter 2 explains the advances and knowledge of the field so far.
- Section 3.1 of chapter 3 explains and analyzes in more detail the metric concept, which the rest of the text focuses on.
- Study of the result achieved by Razborov [11]. Razborov proved that, if we do not consider NOT gates, computing clique requires a superpolynomial amount of gates. As this text focuses on the clique problem, it was appropiate to study this result in detail. After a reading of it, we found that the underlying ideas were very similar to the ones we had at the beginning of this work. Section 3.2 of chapter 3 presents a detailed explanation of this result.
- Explore the aspects in which the initial idea differed from Razborov's. In chapter 4 the behaviour of clique is studied on two metrics: a semantic versus a syntactic one. Experiments with the AND gate are performed in both metrics, as well as an analysis of the results obtained. The source code used to generate all the tables and graphics can be found at the public repository [12].
- Chapter 5 presents a summary of the results obtained, as well as their discussion and some ideas on how to continue the present text.

Capítulo 2

Estado de la cuestión

El estudio de la complejidad de los problemas es una de las ramas más estudiadas dentro de la informática teórica. Esta rama del conocimiento pretende clasificar los problemas en función de su dificultad, relacionándolos con lo que se conoce como clases de complejidad.

Existen una infinidad de clases de complejidad [1], cada una teniendo sus sutilezas e intereses. En particular, el problema $\mathcal{P}$ *vs* $\mathcal{NP}$ es de los más estudiados dentro de este campo.

2.1. Clases de complejidad $\mathcal{P}$ y $\mathcal{NP}$

Una forma de definir las clases de complejidad es a partir del concepto de problema de decisión.

Definición 2.1.1. *Decimos que $\mathcal{A}$ es un problema de decisión si los únicos valores posibles que devuelve sobre su entrada son 1 y 0 (verdadero y falso). Decimos que $\mathcal{A}$ acepta una entrada si devuelve 1 y la rechaza si su resultado es 0.*

Los problemas de decisión se pueden interpretar como preguntas cuya respuesta puede ser sí o no. En este sentido, un algoritmo que resuelve un problema de decisión es una fábrica de respuestas: dado unos datos de entrada, obtiene si los datos son aceptados por el problema de decisión. De este modo, es natural entender un problema de decisión $\mathcal{A}$ como un lenguaje formado por el conjunto de palabras (entradas) que acepta.

Definición 2.1.2. *Un problema de decisión $\mathcal{A}$ está en la clase de complejidad $\mathcal{P}$ si existe un algoritmo que resuelve $\mathcal{A}$ con una complejidad en tiempo polinómica.*

Definición 2.1.3. *Un problema de decisión $\mathcal{A}$ está en la clase de complejidad $\mathcal{NP}$ si, dada una posible solución al problema, se puede comprobar su veracidad en tiempo polinómico.*

Un ejemplo de problema de decisión que está en la clase $\mathcal{P}$ es determinar si un camino tiene coste mínimo en un grafo con pesos, pues el algoritmo de Dijkstra halla el camino mínimo en un grafo desde un vértice dado [7] en tiempo polinómico, y entonces podemos comparar su coste con el del camino proporcionado en tiempo lineal. Por otro lado, un ejemplo de problema que está en $\mathcal{NP}$ es determinar si un grafo es hamiltoniano, pues dado el ciclo o camino, la comprobación se realiza en tiempo polinómico.

La clase $\mathcal{P}$ está formada por aquellos problemas de decisión que pueden resolverse en tiempo polinómico. La clase $\mathcal{NP}$ está formada por aquellos problemas de decisión que pueden comprobarse en tiempo polinómico. Es claro que $\mathcal{P} \subseteq \mathcal{NP}$, pues para comprobar si una solución es válida basta con resolver el problema. El problema es la otra inclusión. Resolver si $\mathcal{NP} \subseteq \mathcal{P}$ es lo que se conoce como el problema $\mathcal{P}$ *vs* $\mathcal{NP}$. A día de hoy, nadie ha sido capaz de demostrar si esa inclusión es cierta o no. Aun así, hay ciertos resultados que dan ideas sobre cómo abordar el problema.

Definición 2.1.4. *Decimos que un problema de decisión $\mathcal{A}$ puede reducirse polinómicamente a un problema de decisión $\mathcal{B}$ si existe un algoritmo polinómico que transforma la entrada de $\mathcal{A}$ en una entrada para $\mathcal{B}$ y la salida del problema transformado es la misma que la del problema inicial. Se denota por $\mathcal{A} \leq_p \mathcal{B}$.*

Definición 2.1.5. *Decimos que un problema de decisión $\mathcal{B}$ es $\mathcal{NP}$-completo si cumple lo siguiente:*

1. $\mathcal{B} \in \mathcal{NP}$.
2. $\forall \mathcal{A} \in \mathcal{NP}$, $\mathcal{A} \leq_p \mathcal{B}$.

Equivalentemente se puede demostrar que un problema $\mathcal{B}$ es $\mathcal{NP}$-completo si existe otro problema $\mathcal{A}$ que sea $\mathcal{NP}$-completo y $\mathcal{A} \leq_p \mathcal{B}$. En particular, si encontramos un problema $\mathcal{NP}$-completo que se pueda resolver en tiempo polinómico, podríamos resolver todos los problemas de $\mathcal{NP}$ en tiempo polinómico y, por tanto, se tendría $\mathcal{NP} = \mathcal{P}$. Pero encontrar un primer problema que sea $\mathcal{NP}$-completo no es sencillo.

Teorema 2.1.1 [6]**.** *SAT-FNC es $\mathcal{NP}$-completo.*

A partir de *SAT-FNC* se pueden encontrar más problemas $\mathcal{NP}$-completos mediante reducciones polinómicas. Como cualquier problema en $\mathcal{NP}$ se puede reducir a un problema $\mathcal{NP}$-completo, estos son buenos candidatos para intentar probar que $\mathcal{P} \neq \mathcal{NP}$. En particular, este texto se centra en estudiar un problema $\mathcal{NP}$-completo, el problema del clique.

2.2. Problema del clique

Definición 2.2.1. *Se dice que un grafo no dirigido $G = (V, A)$ de $n = |V|$ vértices (nodos) tiene un clique de tamaño k si existe $V' \subseteq V$ tal que $|V'| = k$ y para cada par de vértices de V' existe una arista en A que los conecta. En otras palabras, V' es un subgrafo completo de k vértices.*

El problema de decisión asociado es, dado un grafo de n vértices y un número entero positivo k, determinar si el grafo contiene un clique de tamaño k. A partir de aquí nos referiremos a este problema de decisión por $Clique_{n,k}$.

El problema de decisión $Clique_{n,k}$ está en la clase $\mathcal{NP}$, pues se puede comprobar si un conjunto de vértices es un clique en tiempo polinómico.

Teorema 2.2.1. *$Clique_{n,k}$ es $\mathcal{NP}$-completo.*

Este resultado es consecuencia del teorema 2.1.1. Haciendo una reducción polinómica de *SAT-FNC* a *3-SAT* y de *3-SAT* a $Clique_{n,k}$ (véase [9]) se demuestra el teorema. Al ser $Clique_{n,k}$ un problema $\mathcal{NP}$-completo, es un buen candidato para ser estudiado y hacer experimentos con él.

Una simplificación de este problema es *Half-Clique*, que se define como la reformulación de $Clique_{n,k}$ tomando $k = n/2$. Haciendo una reducción sobre $Clique_{n,k}$ se puede demostrar que *Half-Clique* es también un problema $\mathcal{NP}$-completo (véase [5]). Usaremos este resultado en los experimentos, pues nos permite fijar el tamaño del clique y tener como parámetro variable únicamente el número de vértices del grafo.

2.3. Funciones y circuitos booleanos. Clase $\mathcal{P}_{/poly}$

Los otros grandes protagonistas de este texto son las funciones y los circuitos booleanos [2].

Definición 2.3.1. *Se dice que una función f es una función booleana si es de la forma $f : \{0,1\}^n \to \{0,1\}$, donde n es un entero no negativo que indica la aridad de la función, es decir, el número de variables que recibe como entrada. Si x es el conjunto de las n variables de una función booleana f, su salida la denotamos por $f(x)$. Decimos que f acepta x si $f(x) = 1$ y que rechaza x si $f(x) = 0$.*

Una función booleana de aridad n estará definida si conocemos la imagen de sus posibles 2^n valores de entrada. Es decir, si conocemos su tabla de verdad completa. Dos ejemplos de funciones booleanas de aridad 2 clásicas son f_{AND} y f_{OR}, que quedan definidas por las siguientes tablas de verdad.

x_1	x_2	$f_{AND}(x_1, x_2)$
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	$f_{OR}(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 2.1: Tablas de verdad de f_{AND} y f_{OR}

A partir de la tabla de verdad, nos damos cuenta de que si mantenemos fijo el orden de las entradas, podemos representar una función booleana a través de su salida. En otras palabras, para representar una función booleana de aridad n nos

basta con una cadena binaria de longitud 2^n. Para los ejemplos de f_{AND} y f_{OR} tendríamos

$$f_{AND} = 0001 \qquad f_{OR} = 0111$$

El interés de las funciones booleanas recae en que pueden expresar problemas de decisión de forma natural. Si para una función booleana f_n de aridad n denotamos $\mathcal{A}_{f_n} = \{x \in \{0,1\}^n : f_n(x) = 1\}$, un problema de decisión $\mathcal{A}$ tiene asociado una familia de funciones booleanas $\{f_n\}_{n\in\mathbb{N}}$ tales que se cumple

$$\mathcal{A} = \bigcup_{n\in\mathbb{N}} A_{f_n}$$

En particular, las clases de complejidad definidas sobre problemas de decisión también se pueden definir sobre funciones booleanas.

Definición 2.3.2. *Sea n un número entero positivo. Un circuito booleano C con n variables de entrada es un grafo dirigido acíclico con n fuentes (vértices que no tienen aristas incidentes) y un sumidero (vértice que no tiene aristas salientes). Cada vértice, que llamaremos puerta, está etiquetado por $\neg$, $\wedge$ o $\vee$, representando la correspondiente función booleana. Si $x \in \{0,1\}^n$ son variables de entrada, entonces la salida de C se representa por $C(x)$, que equivale al valor obtenido en el nodo sumidero del grafo. El tamaño de un circuito se denota por $|C|$ y corresponde al número de vértices del grafo.*

Las funciones con la etiqueta $\neg$ las denotaremos por NOT, las que tengan la etiqueta $\wedge$ por AND y las que tengan $\vee$ por OR. Diremos que un circuito C computa una función booleana f si $\forall x \in \{0,1\}^n$ se cumple $C(x) = f(x)$. En otras palabras, la salida de f coincide con la salida de C.

Gráficamente, un circuito booleano es un diagrama que muestra cómo obtener una salida a partir de una cadena binaria aplicando una secuencia de operaciones booleanas. Por ejemplo, la figura 2.1 muestra el circuito que computa la función f_{XOR}, definida como $f_{XOR}(x_1, x_2) = 1 \Leftrightarrow x_1 \neq x_2$.

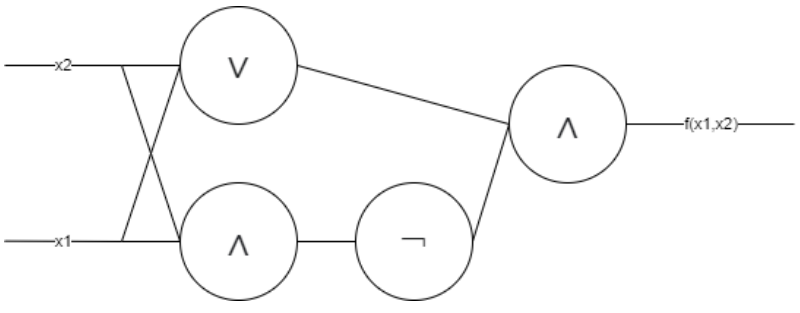


Figura 2.1: Circuito booleano que computa la función $f_{XOR}(x_1, x_2)$

Al igual que ocurre con las funciones booleanas, un circuito C que computa una función f está totalmente definido por la cadena binaria que representa a f. Además, como podemos traducir un circuito C en la función que computa, también podemos asociar un problema de decisión cualquiera con una familia de circuitos.

Definición 2.3.3. *Sea $T : \mathbb{N} \to \mathbb{N}$ una función. Una familia de circuitos de tamaño acotado por T es una secuencia de circuitos booleanos $\{C_n\}_{n\in\mathbb{N}}$ donde cada C_n tiene n variables de entrada y $|C_n| \leq T(n)$.*

Definición 2.3.4. *Dada una familia de funciones booleanas $\{f_n\}_{n\in\mathbb{N}}$, el problema de decisión $\mathcal{A} = \bigcup_{n\in\mathbb{N}} A_{f_n}$ está en la clase $\mathcal{P}_{/poly}$ si existe un polinomio p y una familia de circuitos $\{C_n\}_{n\in\mathbb{N}}$ de tamaño acotado por p tal que C_n computa f_n para todo n.*

La clase $\mathcal{P}_{/poly}$ pretende agrupar aquellos problemas de decisión que se pueden resolver con circuitos de tamaño polinómico. El siguiente resultado muestra la relación que hay entre $\mathcal{P}_{/poly}$ y las clases $\mathcal{P}$ y $\mathcal{NP}$.

Teorema 2.3.1. *$\mathcal{P} \subset \mathcal{P}_{/poly}$.*

La inclusión del teorema 2.3.1 nos proporciona un enfoque para abordar $\mathcal{P}$ *vs* $\mathcal{NP}$. Si encontramos un problema de decisión que esté en $\mathcal{NP}$ pero que no esté en $\mathcal{P}_{/poly}$, entonces tampoco puede estar en $\mathcal{P}$ y, por tanto, $\mathcal{P} \neq \mathcal{NP}$.

Pero, ¿por qué es razonable abordar $\mathcal{P}$ *vs* $\mathcal{NP}$ mediante funciones y circuitos booleanos?

Teorema 2.3.2 [14]**.** *Para n suficientemente grande, casi todas las funciones booleanas de aridad n necesitan de un circuito booleano de tamaño al menos $\frac{2^n}{n}$ para ser computadas.*

¿Qué nos dice este resultado? Que la mayoría de funciones booleanas requieren de circuitos de tamaño superpolinómico. Si la mayoría de funciones booleanas requieren un circuito de tamaño exponencial, no será difícil encontrar una familia de funciones cuyo problema de decisión asociado esté en $\mathcal{NP}$, ¿no?

Por desgracia, la demostración de este teorema no es constructiva, y todavía no se ha encontrado ningún problema que cumpla esta condición. Aun así, este resultado parece indicar que los circuitos y funciones booleanas son una buena manera de plantear $\mathcal{P}$ *vs* $\mathcal{NP}$.

Además, un circuito booleano recoge información no solo de la salida de una función, sino también de por qué ocurre lo que ocurre (transiciones entre vértices). En vez de tratar el problema como una especie de oráculo, los circuitos nos permiten estudiar las entrañas de un problema y revelarnos propiedades sobre su tamaño y comportamiento. Nos centraremos en funciones y circuitos que calculan $Clique_{n,k}$.

Capítulo 3

Cota inferior para circuitos monótonos que calculan $Clique_{n,k}$

En este capítulo explicaremos el concepto de métrica sobre un circuito y el potencial que tiene sobre $Clique_{n,k}$. Estudiaremos con detalle el comportamiento de las métricas en circuitos booleanos monótonos [13] que computan la función $Clique_{n,k}$. Un circuito monótono es aquel formado únicamente por puertas AND y OR, sin negaciones. En la sección 3.2 reproduciremos resultados parecidos a los que obtuvo Razborov [11] basándonos en las notas [4].

3.1. Concepto de métrica y crecimiento en un circuito

Supongamos por un momento que tenemos un circuito que computa cierta función f. El circuito va combinando funciones a través de sus puertas hasta llegar a f. Supongamos también que tenemos una función de asignación, llamemos μ, que mide cuánto se parece una función f_i a la función f. Cuanto más se parezca, más grande será el valor de μ, alcanzando su valor máximo en f. Digamos que este valor, $\mu(f)$, es suficientemente grande. Esta asignación funciona en cierta manera como un indicador que nos dice en qué parte del circuito estamos. Intuitivamente, si una función tiene un valor cercano a $\mu(f)$ es porque ha sido computada cerca del final del circuito, mientras que si tiene un valor lejano, es porque todavía le falta mucho camino por recorrer para llegar al final. De algún modo, μ tiene que ir creciendo según avanza por el circuito en cuestión.

Entendamos esto con un ejemplo. Creemos un circuito sencillo C que actúa sobre las variables $\{x_1, x_2, x_3, x_4\}$ y calcula la función $f = x_1 \vee x_2 \vee x_3 \vee x_4$. Una posible forma para C que cumpla estos requisitos es la que se muestra en la figura 3.1.

Tomemos μ de tal manera que $\mu(f_i)$ devuelva la cantidad de literales que contiene la disyunción que calcula f_i. Podemos ver cómo evoluciona μ por el camino inferior (el de x_1).

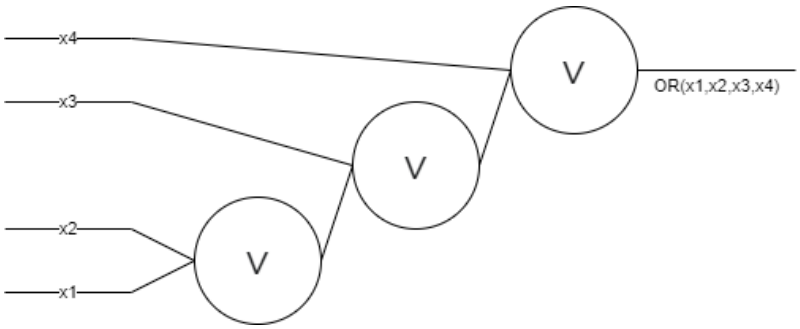


Figura 3.1: Circuito que computa OR de cuatro variables.

- Al inicio del circuito, $\mu(x_1) = 1$.
- Después de la primera puerta, $\mu(x_1 \vee x_2) = 2$.
- Después de la segunda puerta, $\mu(x_1 \vee x_2 \vee x_3) = 3$.
- Después de la última puerta, $\mu(x_1 \vee x_2 \vee x_3 \vee x_4) = 4$.

Aunque este circuito es muy sencillo, ejemplifica el comportamiento creciente de μ: cuanto más avanzamos en el circuito, más cerca estaremos de la función final y más alta será la puntuación.

Esto no quiere decir que μ tenga que ser monótona. En este circuito de ejemplo lo es, pero en otro caso podría ocurrir que fuera necesario retroceder para luego avanzar. Dependiendo de cómo sea μ, esta crecerá siempre, a veces, mucho, poco, decrecerá, contendrá más información o menos... Pero, ¿por qué interesa definir una métrica sobre un circuito?

Si encontramos una función que podemos acotar, podemos sacar información sobre el tamaño del circuito. ¿Cuánto va a aumentar como mucho la métrica después de atravesar una puerta? Si sabemos la puntuación inicial, la puntuación final y el aumento máximo que sufren las funciones a lo largo del circuito, podemos encontrar una cota inferior para el tamaño del circuito. En el ejemplo anterior (figura 3.1), podríamos haber argumentado lo siguiente. Como hacemos siempre OR con las variables de entrada, el número de variables aumenta como mucho en 1, es decir, $\mu(f \vee x_i) \leq \mu(f) + 1 \Rightarrow \mu(f \vee x_i) - \mu(f) \leq 1$. Al principio tenemos puntuación 1, al final tenemos puntuación 4 y en cada paso como mucho aumentamos en 1. Es decir, como poco hay que pasar por $\frac{4-1}{1} = 3$ puertas.

En resumen, la función μ nos da información sobre el número de puertas de un circuito y, por tanto, de la complejidad de un problema. Evidentemente, la elección de μ es esencial para que este proceso tenga sentido. Una función que pueda crecer mucho en cada paso no nos transmite prácticamente nada de información.

Para el problema del clique, una buena elección de μ podría demostrar que cualquier circuito que compute clique es de tamaño exponencial (y por tanto $\mathcal{P} \neq \mathcal{NP}$). Razborov, basándose en esta idea, demostró que cualquier circuito que computa clique utilizando puertas AND y OR tiene que tener tamaño exponencial.

3.2. Teorema de Razborov

Como hemos adelantado al principio de este capítulo, en esta sección vamos a desarrollar teóricamente el resultado al que llegó Razborov.

Un grafo cualquiera con n vértices podemos representarlo con $\binom{n}{2}$ variables binarias, donde cada variable indica la presencia de una arista.

Definición 3.2.1. *Dado un grafo G con n vértices, decimos que un indicador de clique $Ind(X)$ sobre un conjunto $X \subseteq \{1, ..., n\}$ es una función booleana*

$$Ind(X)(x_1, ..., x_{\binom{n}{2}}) = \begin{cases} 1 \text{ si } X \text{ es un clique en el grafo } G \\ 0 \text{ en caso contrario} \end{cases}$$

Si no hay ambigüedad, nos referiremos a los indicadores como $Ind(X)$, abusando notación. Con estos indicadores podemos construir la función que computa $Clique_{n,k}$, que es

$$K_{n,k} = \bigvee_{X \subseteq \{1,...,n\}, |X|=k} Ind(X) \tag{3.1}$$

Obsérvese que hay un total de $\binom{n}{k}$ subconjuntos X que cumplen $|X| = k$.

Para motivar las ideas detrás de las demostraciones que contaremos más adelante, limitémonos a un caso especial. Supongamos que tenemos circuitos que se pueden separar en dos niveles: un primer nivel de puertas AND y un segundo nivel de puertas OR. Este circuito computará cierta función

$$f = \bigvee_{i=1}^{r} \bigwedge_{j \in E_i} x_j$$

para ciertos conjuntos de variables $E_1, ..., E_r$. Estos conjuntos representan las variables sobre los que actúa cada puerta AND del circuito, y queremos demostrar que r necesariamente tiene que ser grande. Pensemos por un momento que los únicos grafos que tienen cliques de tamaño k son aquellos que tienen las aristas solo en el clique, y no tienen aristas fuera de él. Bajo esta suposición, podemos sustituir cada término $\wedge_{j \in E_i} x_j$ por el correspondiente indicador $Ind(E_i)$. Esto nos deja con una función

$$f = \bigvee_{i=1}^{r} Ind(E_i)$$

que se parece a la función objetivo $K_{n,k}$, para el que sabemos que necesitamos $\binom{n}{k}$ funciones indicadoras de clique. Además, sabemos en qué grafos puede fallar el circuito. Si existe algún E_i tal que $|E_i| < k$, podemos hacer fallar el circuito poniendo un clique de tamaño menor que k en los vértices de E_i, aceptando una entrada incorrecta. Del mismo modo, si todos los E_i cumplen que $|E_i| \geq k$ pero $r < \binom{n}{k}$, por el principio del palomar, existirá algún subconjunto de k elementos que no esté cubierto por los E_i, haciendo que el circuito rechace un grafo que debería ser aceptado.

Esta simplificación ilustra que si aproximamos el circuito con disyunciones de indicadores podemos controlar el comportamiento que tienen. El objetivo es, por tanto, encontrar una forma de aproximar el circuito con indicadores y encontrar una cota inferior en función del número de puertas que utiliza. Obsérvese que cualquier función monótona (formada por AND y OR) puede escribirse como una disyunción de conjunciones, lo que se conoce como forma normal disyuntiva (OR de AND). Esto se consigue aplicando las leyes distributivas sobre la función. No obstante, el tamaño puede descontrolarse drásticamente. Por ejemplo, la forma normal disyuntiva de $\bigwedge_{i=1}^{n}(x_i \wedge y_i)$ tiene un total de 2^n términos. Si queremos relacionar el tamaño del circuito con su aproximación, necesitamos tener el mayor control posible sobre el tamaño de las aproximaciones. ¿Cómo solucionamos este problema?

Definición 3.2.2. *Una colección de conjuntos $Z = \{z_1, ..., z_p\}$ se dice que es un girasol si las intersecciones dos a dos son constantes. Es decir, existe un conjunto z_0 tal que $\forall i, j \in \{1, ..., p\}$, $i \neq j$, $z_i \cap z_j = z_0$. z_0 se denomina el núcleo del girasol y p el número de pétalos.*

Definición 3.2.3. *Unificar un girasol $Z = \{z_1, ..., z_p\}$ con núcleo z_0 es sustituir todos los z_i por z_0.*

En nuestro caso, queremos usar girasoles para poder sustituir los indicadores en uno solo cuando su tamaño sea demasiado grande. El siguiente lema nos da una condición para poder formar un girasol.

Lema 3.2.1. *Sean $p \geq 2$, $l \geq 1$ dos números enteros y $Z = \{z_1, ..., z_n\}$ una colección de n conjuntos tal que $|z_i| \leq l$ $\forall i \in \{1, ..., n\}$. Entonces, si $|Z| > (p-1)^l l!$, existe un subconjunto $\{z_1, ..., z_p\} \subseteq Z$ que forma un girasol de p pétalos.*

Demostración. Procedemos por inducción sobre l, el tamaño máximo de cada conjunto.

Para $l = 1$ tenemos $|Z| > (p-1)^l l! = (p-1)$. Cada conjunto tiene un elemento, así que las intersecciones dos a dos dan como resultado el vacío. Como $|Z| > p-1$, podemos seleccionar p elementos distintos de Z, consiguiendo un girasol de p pétalos que tiene por núcleo al vacío.

Para $l > 1$, tomamos un subconjunto de Z de t pétalos que forme un girasol con el núcleo vacío, de tal manera que t sea lo más grande posible. Si $t \geq p$ hemos terminado, pues ya tenemos un girasol de p pétalos. Así que podemos suponer a partir de aquí que $t < p$. En este caso, vamos a probar que si Z no contiene ningún girasol de p pétalos, entonces $|Z| \leq (p-1)^l l!$. Llamemos $x_1, \ldots, x_t$ a los pétalos de este girasol de tamaño t y $X = \cup_{i=1}^{t} x_i$. Entonces

$$|X| = \left| \bigcup_{i=1}^{t} x_i \right| \leq \sum_{i=1}^{t} |x_i| \leq lt < lp$$

Como l y p son números enteros positivos, podemos expresar la cota anterior como $|X| \leq l(p-1)$.

Sea $Z_x = \{z \setminus \{x\} : x \in z \in Z\}$ para un $x \in X$. Cada Z_x es una colección de conjuntos donde cada elemento tiene a lo sumo tamaño $l-1$. Además, cualquier girasol de Z_x puede extenderse a un girasol en Z con el mismo número de pétalos: si $\{z_1, ..., z_m\}$ es un girasol de $Z_x \Rightarrow \{z_1 \cup \{x\}, ..., z_m \cup \{x\}\}$ es un girasol de Z. Como $t < p$, entonces no puede haber ningún girasol de tamaño p en Z_x y, por la hipótesis de inducción, $|Z_x| \leq (p-1)^{l-1}(l-1)!$.

Por otro lado, la maximalidad de t implica que todo $z \in Z$ interseca la unión X. Es decir, si $z \in Z \Rightarrow \exists x \in z \cap X$. Por tanto, cada z se corresponde con al menos un conjunto de Z_x, de donde se sigue

$$|Z| \leq \sum_{x \in X} |Z_x|$$

Por último, juntando todo

$$|Z| \leq \sum_{x \in X} |Z_x| \leq |X|(p-1)^{l-1}(l-1)! \leq$$
$$\leq l(p-1)(p-1)^{l-1}(l-1)! = (p-1)^l l!$$

En resumen, si Z no tiene girasoles de p pétalos, entonces $|Z| \leq (p-1)^l l!$, que es equivalente a decir que si $|Z| > (p-1)^l l!$, entonces Z tiene un girasol de p pétalos. □

Básicamente, este lema nos dice que si tenemos suficientes conjuntos, necesariamente existe un girasol de un tamaño p. Eligiendo p y l adecuadamente, podremos utilizar esto para controlar la cantidad de indicadores que usaremos para aproximar el circuito. Después de esto, ya estamos en condiciones de definir formalmente cómo vamos a aproximar un circuito.

Definición 3.2.4. *Dados m y l dos enteros positivos, decimos que A es un aproximador si A es de la forma*

$$A = \bigvee_{i=1}^{m'} Ind(X_i)$$

para cierto $m' \leq m$ y $|X_i| \leq l$ $\forall i \in \{1, ..., m'\}$.

Definición 3.2.5. *Sean m y l dos enteros positivos, C un circuito booleano formado por puertas AND y OR. Construimos una aproximación $\widetilde{C}$ de C inductivamente.*

- *C no tiene puertas. Entonces C no es más que una variable x_i. Es decir, C evalúa la existencia de una arista. Si x_i representa la arista entre el par de nodos u, v, tomamos $X = \{u, v\}$ y el aproximador del circuito es $\widetilde{C} = Ind(X)$.*
- *$C = C_1 \vee C_2$. Recursivamente, $\widetilde{C_1} = \bigvee_{i=1}^{m_1} Ind(X_i)$, $\widetilde{C_2} = \bigvee_{i=1}^{m_2} Ind(Y_i)$ y construimos $\widetilde{C}$ como*

$$\widetilde{C} = \widetilde{C_1} \vee \widetilde{C_2} = \bigvee_{i=1}^{m_1} Ind(X_i) \vee \bigvee_{i=1}^{m_2} Ind(Y_i)$$

Si $m_1 + m_2 \leq m$, $\widetilde{C}$ es el aproximador. En otro caso, unificamos los girasoles que se forman en la unión de los X_i con los Y_i hasta que no haya más de m indicadores de clique.

- *$C = C_1 \wedge C_2$. Del mismo modo, $\widetilde{C_1} = \bigvee_{i=1}^{m_1} Ind(X_i)$, $\widetilde{C_2} = \bigvee_{i=1}^{m_2} Ind(Y_i)$ y construimos $\widetilde{C}$. En este caso, no podemos tomar $\widetilde{C_1} \wedge \widetilde{C_2}$ como aproximador porque por definición un aproximador tiene que ser una disyunción de indicadores de clique. Para arreglar esto, hacemos los siguientes cambios. Partiendo de $\widetilde{C_1} \wedge \widetilde{C_2}$,*

$$\bigvee_{i=1}^{m_1} \bigvee_{j=1}^{m_2} Ind(X_i) \wedge Ind(Y_j)$$
$$\bigvee_{i=1}^{m_1} \bigvee_{j=1}^{m_2} Ind(X_i \cup Y_j)$$
$$\bigvee_{|X_i \cup Y_j| \leq l} Ind(X_i \cup Y_j)$$

  Si $m_1 \cdot m_2 \leq m$, tomamos esta última expresión como aproximador. En otro caso, al igual que con el caso anterior, unificamos girasoles hasta que no haya más de m indicadores de clique.

En los casos inductivos, es importante asegurar que siempre vamos a poder unificar girasoles hasta tener como mucho m elementos. Por eso es importante el lema 3.2.1, porque nos da una elección de m para poder tener siempre girasoles. Más adelante veremos cómo seleccionar el número de pétalos p de forma que tengan sentido las cuentas.

Por otro lado, ¿por qué sustituir indicadores por el núcleo del girasol que forman tiene sentido? En primer lugar, porque tenemos control de la cantidad de conjuntos que tenemos. En segundo lugar, sustituir por el núcleo mantiene cierta información sobre la existencia de cliques. Si $X_1, ..., X_p$ es un girasol de núcleo X_0,

- Si algún X_i es un clique, $Ind(X_i) = 1$ y cualquier subconjunto suyo es un clique. Luego X_0 va a ser un clique, es decir, $Ind(X_0) = 1$. Además, X_0 está contenido en todos los X_i, así que al cambiar los X_i por X_0 no perdemos esta información, esto es, $\bigvee_{i=1}^{p} Ind(X_i) = Ind(X_0)$.
- Si ningún X_i es un clique, entonces cambiar por X_0 no va a empeorar el conteo de cliques, esto es, $\bigvee_{i=1}^{p} Ind(X_i) \leq Ind(X_0)$.

Es decir, para un girasol $X_1, ..., X_p$ se cumple que $\bigvee_{i=1}^{p} Ind(X_i) \leq Ind(X_0)$. Esto nos dice que los girasoles y los indicadores de clique se llevan bien con la filosofía de las métricas, en el sentido de que al pasar por una puerta estamos aumentando la puntuación.

Definición 3.2.6. *Dado un grafo G de n vértices, y un tamaño para un clique $k \leq n$, decimos que*

- *G es de tipo I si G es un clique de tamaño k y no tiene más aristas que aquellas que forman el clique.*

- *G es de tipo II si G es un grafo $(k-1)$-coloreable que tiene aristas entre todo par de vértices de distinto color. En general, un grafo G es c-coloreable si podemos asignar un número entre 1 y c a cada vértice de G de tal forma que no haya dos vértices adyacentes con el mismo número asignado.*

Definición 3.2.7. *Sea f una función booleana que tiene por entrada $\binom{n}{2}$ variables correspondientes a las aristas de un grafo de n vértices. Las métricas μ_I y μ_{II} se definen como sigue*

- *$\mu_I(f)$ = cantidad de grafos de n vértices de tipo I que acepta f.*
- *$\mu_{II}(f)$ = cantidad de grafos de n vértices de tipo II que acepta f.*

Si nos referimos a grafos de n vértices y buscamos cliques de tamaño k, observamos lo siguiente.

- *Los grafos de tipo II no tienen cliques de tamaño k.* Si un grafo de tipo II tuviera un clique de tamaño k, existiría un vértice con al menos $k-1$ vértices adyacentes a él, que contradice el hecho de que el grafo sea $(k-1)$-coloreable. Por tanto, no hay colisiones entre los grafos de tipo I y tipo II.
- *Hay exactamente $\binom{n}{k}$ grafos de tipo I* (formas posibles de seleccionar k elementos sobre un conjunto de n elementos).
- *Hay exactamente $(k-1)^n$ grafos de tipo II.* Los grafos de tipo II dependen de qué color asignemos a cada nodo. Hay $k-1$ colores y n nodos, lo que nos deja con $(k-1)^n$ grafos de tipo II (grafos isomorfos con distinta asignación de colores los consideramos distintos).

En particular, los grafos de tipo I son aquellos que conforman la fórmula que computa $Clique_{n,k}$ (ecuación 3.1). Para computar $Clique_{n,k}$, todos los grafos de tipo I tienen que ser aceptados y todos los grafos de tipo II tienen que ser rechazados. Aunque ni los grafos de tipo I son los únicos que tienen que ser aceptados ni los grafos de tipo II son los únicos que tienen que ser rechazados, son una muestra suficiente para demostrar que un circuito que compute $Clique_{n,k}$ con puertas AND y OR necesita una cantidad exponencial de puertas.

En este punto, nos queda encontrar relaciones entre todos los conceptos: cómo varía la aproximación de un circuito respecto al original fijándonos en los grafos de tipo I y II y ver que estas aproximaciones nos llevan a concluir que los circuitos son demasiado grandes.

Lema 3.2.2. *Sea $A = \bigvee_{i=1}^{m'} Ind(X_i)$ un aproximador definido bajo los parámetros explicados en la definición 3.2.4. Entonces, una y solo una de las dos condiciones siguientes se cumple*

a) A rechaza todos los grafos.

b) $\mu_{II}(A) \geq \left(1 - \frac{\binom{l}{2}}{k-1}\right)(k-1)^n$

Demostración. Si $m' = 0$, entonces el aproximador A es una disyunción vacía, así que rechazará cualquier grafo que reciba por entrada, cumpliendo la primera condición.

Si $m' > 0$, tenemos que contar cuántos grafos de tipo II acepta el aproximador. Como sabemos que hay $(k-1)^n$ grafos de tipo II, una forma de realizar esta cuenta es calcular la probabilidad de aceptar un grafo de tipo II, que es complementaria a la probabilidad de rechazar un grafo de tipo II. Como A es una disyunción, que A acepte una entrada es lo mismo que decir que algún indicador suyo acepta la misma entrada. Así que podemos reducir a un caso más sencillo.

Tomamos un indicador cualquiera de la disyunción de A, digamos $Ind(X)$. ¿Cuál es la probabilidad de que $Ind(X)$ rechaze un grafo aleatorio de tipo II? Recordemos (véase la definición 3.2.1) que $Ind(X)$ acepta un grafo G si los nodos de X forman un clique en G. Como todos los vértices de distinto color de los grafos de tipo II están conectados entre sí, la única forma de que $Ind(X)$ rechaze un grafo de tipo II es que existan dos nodos $u, v \in X$ que tengan el mismo color asignado en el grafo. Dados $u, v \in X$, esta probabilidad es

$$P[color(u) = color(v)] = \sum_{c=1}^{k-1} P[color(u) = c \wedge color(v) = c] =$$

$$= \sum_{c=1}^{k-1} \big(P[color(u) = c] \cdot P[color(v) = c]\big) = \sum_{c=1}^{k-1} \left(\frac{1}{k-1} \cdot \frac{1}{k-1}\right) =$$

$$= \frac{1}{(k-1)^2} \cdot (k-1) = \frac{1}{k-1}$$

Con esto, podemos acotar la probabilidad de que un indicador cualquiera rechace un grafo de tipo II como sigue:

$$P[\exists u, v \in X \wedge color(u) = color(v)] = P\left[\bigcup_{u,v \in X, u \neq v} color(u) = color(v)\right] \leq$$

$$\leq \sum_{u,v \in X, u \neq v} P[color(u) = color(v)] = \sum_{u,v \in X, u \neq v} \frac{1}{k-1} \leq \frac{\binom{l}{2}}{k-1}$$

La última desigualdad ocurre porque $|X| \leq l$, así que como mucho hay $\binom{l}{2}$ pares distintos entre los elementos de X. Esta cuenta completa la prueba, pues la probabilidad de que $Ind(X)$ acepte un grafo de tipo II aleatorio es al menos $1 - \frac{\binom{l}{2}}{k-1}$. Como hay $(k-1)^n$ grafos de tipo II, la cantidad de grafos aceptados está acotado por esta probabilidad, que es

$$\left(1 - \frac{\binom{l}{2}}{k-1}\right)(k-1)^n$$

□

Definición 3.2.8. *Sean C y $\widetilde{C}$ un circuito y su aproximador, respectivamente, Definimos:*

- *$d_I(C, \widetilde{C})$ es la cantidad de grafos de tipo I que acepta C pero rechaza $\widetilde{C}$.*
- *$d_{II}(C, \widetilde{C})$ es la cantidad de grafos de tipo II que rechaza C pero acepta $\widetilde{C}$.*

Lema 3.2.3. *Dado un circuito C y su aproximador $\widetilde{C}$, entonces*

$$d_I(C, \widetilde{C}) \leq |C| m^2 \binom{n-l-1}{k-l-1}$$

Demostración. Buscamos la cantidad de grafos de tipo I que acepta C pero rechaza $\widetilde{C}$. Argumentamos por inducción estructural en la forma en la que se define $\widetilde{C}$ (véase la definición 3.2.5).

- C no tiene puertas. En este caso, $\widetilde{C}$ coincide con C, así que $d_I(C, \widetilde{C}) = 0$.
- $C = C_1 \vee C_2$. Queremos ver cuántos grafos de tipo I acepta $\widetilde{C_1} \vee \widetilde{C_2}$ que no son aceptados por $\widetilde{C}$. Si en este caso no unificamos ningún girasol, entonces $\widetilde{C}$ es precisamente esta unión, así que nada cambia. En el caso de unificar algún girasol, sea $Ind(X)$ un indicador que acepta un grafo de tipo I presente en $\widetilde{C_1} \cup \widetilde{C_2}$. Si al unificar girasoles no cambiamos este indicador, entonces $\widetilde{C}$ sigue aceptando el mismo grafo. Y si cambiamos $Ind(X)$ por un núcleo, ya hemos visto antes que el indicador del núcleo sigue aceptando el grafo. Por tanto, en todos los casos se acepta la misma cantidad de grafos de tipo I, es decir, que también se cumple $d_I(C, \widetilde{C}) = 0$.
- $C = C_1 \wedge C_2$. Queremos ver cuántos grafos de tipo I acepta $\widetilde{C_1} \wedge \widetilde{C_2}$ que no son aceptados por $\widetilde{C}$. Si para un grafo de tipo I $Ind(X_i) \wedge Ind(Y_i) = 1$, entonces tanto los vértices de X_i como los de Y_j forman un clique. Como los grafos de tipo I solo tienen aristas en el clique, $X_i \cup Y_j$ sigue siendo un clique, y por tanto $Ind(X_i \cup Y_j) = 1$. En otras palabras, al hacer este cambio no perdemos ningún grafo de tipo I. El problema surge cuando descartamos $X_i \cup Y_j$ porque $|X_i \cup Y_j| > l \Leftrightarrow |X_i \cup Y_j| \geq l+1$. Cualquier grafo de tipo I que acepte $Ind(X_i \cup Y_j)$ se pierde cuando descartamos esta unión. ¿Cuántos de estos grafos hay? Como poco hay $l+1$ elementos en $X_i \cup Y_j$, y completar a un clique de tamaño k supone seleccionar $k-(l+1)$ elementos sobre los restantes $n-(l+1)$ nodos. Es decir, como mucho descartamos $\binom{n-l-1}{k-l-1}$ grafos de tipo I. Por otro lado, como mucho se forman m^2 uniones, y como mucho hay $|C|$ conjunciones en el circuito, lo que nos lleva a $d_I(C, \widetilde{C}) \leq |C| m^2 \binom{n-l-1}{k-l-1}$.

□

Lema 3.2.4. *Dado un circuito C y su aproximador $\widetilde{C}$, entonces*

$$d_{II}(C, \widetilde{C}) \leq |C| m^2 \left(\frac{\binom{l}{2}}{k-1} \right)^p (k-1)^n$$

Demostración. Buscamos la cantidad de grafos de tipo II que rechaza C pero acepta $\widetilde{C}$. Argumentamos por inducción estructural en la forma en la que se define $\widetilde{C}$ (veáse la definición 3.2.5).

- C no tiene puertas. En este caso, $\widetilde{C}$ coincide con C, así que $d_{II}(C, \widetilde{C}) = 0$.

- $C = C_1 \vee C_2$. Queremos ver cuántos grafos de tipo II rechaza $\widetilde{C_1} \vee \widetilde{C_2}$ que son aceptados por $\widetilde{C}$. Si en este caso no unificamos ningún girasol, entonces $\widetilde{C}$ es precisamente esta unión, así que nada cambia. Luego el único caso que puede causar variaciones es al unificar girasoles. Dado un girasol $\{z_1, ..., z_p\}$ de núcleo z_0, ¿cuántos grafos como mucho rechaza $\bigvee_{i=1}^{p} Ind(z_i)$ pero acepta $Ind(z_0)$?

  Calculemos la probabilidad de que esto ocurra en un grafo aleatorio de tipo II. Digamos que un conjunto de vértices z_i está bien coloreado (b.c.) cuando todos los vértices tienen colores distintos. Entonces, un grafo de tipo II tiene un clique en los nodos de z_i si y solo si z_i está bien coloreado. Por tanto, queremos acotar esta probabilidad:

  $$P[z_1, ..., z_p \text{ no están b.c.} \wedge z_0 \text{ está b.c.}]$$

  Aplicando desigualdades y teniendo en cuenta que asignar colores son sucesos independientes,

  $$\begin{aligned} &P[z_1, ..., z_p \text{ no están b.c.} \wedge z_0 \text{ está b.c.}] \leq \\ &\leq P[z_1, ..., z_p \text{ no están b.c.} \mid z_0 \text{ está b.c.}] \leq \\ &\leq \prod_{i=1}^{p} P[z_i \text{ no está b.c.} \mid z_0 \text{ está b.c.}] \leq \\ &\leq \prod_{i=1}^{p} P[z_i \text{ no está b.c.}] \end{aligned}$$

  Que un conjunto de vértices z_i no esté bien coloreado es lo mismo que decir que existan dos vértices $u, v \in z_i$ que no tengan colores distintos. Y por tanto, $P[z_i \text{ no está b.c.}] = P[\exists u, v \in z_i \text{ tal que } color(u) = color(v)]$. Esta probabilidad ya la hemos calculado en la demostración del lema 3.2.2, así que podemos acotar todo por

  $$\leq \prod_{i=1}^{p} \frac{\binom{l}{2}}{k-1} = \left(\frac{\binom{l}{2}}{k-1}\right)^p$$

  Hay un total de $(k-1)^p$ grafos de tipo II, como mucho hacemos m unificaciones y como mucho hay $|C|$ disyunciones en todo el circuito, así que

  $$d_{II}(C, \widetilde{C}) \leq |C| m \left(\frac{\binom{l}{2}}{k-1}\right)^p (k-1)^n$$

- $C = C_1 \wedge C_2$. Queremos ver cuántos grafos de tipo II rechaza $\widetilde{C_1} \wedge \widetilde{C_2}$ y son aceptados por $\widetilde{C}$. Si $\widetilde{C_1} \wedge \widetilde{C_2}$ rechaza un grafo de tipo II, entonces todos los pares $Ind(X_i) \wedge Ind(Y_j)$ rechazan este grafo. Sin pérdida de generalidad, $Ind(X_i) = 0$, por lo que X_i no es un clique. Pero entonces $X_i \cup Y_j$ tampoco puede ser un clique (cualquier subconjunto de un clique es un clique), así que $Ind(X_i \cup Y_j) = 0$ y no estamos aceptando ningún grafo de tipo II nuevo. En

el siguiente paso, cuando descartamos los indicadores con $|X_i \cup Y_j| > l$ no aceptamos grafos nuevos, pues estamos quitando elementos de la disyunción, así que no podemos aceptar algo que no se aceptaba antes. En definitiva, nos reducimos al mismo caso que antes: podemos aceptar grafos al unificar girasoles. La cuenta es la misma, solo que en este caso podemos hacer un total de m^2 unificaciones, dejando

$$d_{II}(C, \widetilde{C}) \leq |C| m^2 \left(\frac{\binom{l}{2}}{k-1} \right)^p (k-1)^n$$

Como m es un entero positivo, $m^2 \geq m$, así que este caso cubre al primero, lo que termina la demostración.

□

Después de demostrar estos lemas, estamos en condiciones de demostrar el teorema de Razborov.

Teorema 3.2.1. *Sean n y k dos números enteros positivos tales que $3 \leq k \leq n^{1/4}$. Sea C un circuito monótono que computa $Clique_{n,k}$. Entonces,*

$$|C| \geq n^{\Omega(\sqrt{k})}$$

Tomando $k = n^{1/4}$ se tiene que $|C| \geq n^{\Omega(n^{1/8})} \geq 2^{\Omega(n^{1/8})}$, que es una cota inferior exponencial. Antes de mostrar la demostración de este teorema, hay que discutir sobre la elección de los parámetros p, l y m con los que hemos estado trabajando toda esta sección.

- En el lema 3.2.2 hemos llegado a acotar una probabilidad por $\dfrac{\binom{l}{2}}{k-1}$. Para que esto tenga algún sentido práctico, esta cota debería ser menor que 1, así que podemos encontrar un valor candidato para l.

$$\frac{\binom{l}{2}}{k-1} \leq 1 \Longrightarrow \frac{l^2 - l}{2} \leq k - 1 \leq k \Longrightarrow l^2 - l - 2k \leq 0$$
$$l \leq \frac{1 + \sqrt{1 + 8k}}{2} \leq \frac{1 + 1 + 2\sqrt{2k}}{2} = \sqrt{2k}$$

  Así que debería cumplirse $l \leq \sqrt{2k}$. Esto nos indica que $l = \sqrt{k}$ parece un buen candidato.

- Para m necesariamente tenemos que tomar la cota del lema 3.2.1 $(p-1)^l l!$. Esto es para que en la construcción de los aproximadores podamos unificar girasoles hasta quedarnos con m conjuntos.

- p es el número de pétalos de los girasoles y no es tan evidente cómo encontrar un candidato. En el lema 3.2.4 aparece p como exponente en una probabilidad y podríamos intentar hacer lo mismo que con l. El problema es que la elección

de l ya hace que esa probabilidad tenga sentido, dejando que p tome cualquier valor. El único otro lugar donde encontramos el parámetro p es precisamente en la elección de m, pero el valor de m tiene que depender de p para que todas las construcciones tengan sentido. Entonces, para elegir p necesitamos conocer m, pero para elegir m necesitamos conocer p ... En definitiva, no podemos usar m ni l para estimar el valor de p.

Entonces, ¿qué hacemos? Intentar intuir cómo debería ser p para obtener una cota exponencial. El valor de p tiene que ir en función de n y k, y ya hemos visto que en las expresiones de los lemas aparece como un exponente. La cota final que queremos lograr es $n^{\sqrt{k}}$, así que n tiene que desaparecer del exponente mientras que $\sqrt{k}$ se mantiene. Tomando $p = \sqrt{k}\log n$ logramos este efecto.

Ahora sí, vamos con la demostración del teorema.

Demostración. Sean $l = \lfloor\sqrt{k}\rfloor$, $p = \lceil\lambda\sqrt{k}\log n\rceil$ y $m = (p-1)^l l!$ para cierta constante λ que permita $p \geq 2$. Con estos parámetros estamos en condiciones de construir un aproximador $\widetilde{C}$ de C y, por tanto, podemos utilizar los lemas anteriores. El lema 3.2.2 nos dice que existen dos escenarios: o bien $\widetilde{C}$ rechaza todos los grafos, o bien acepta como poco una cantidad de grafos de tipo II.

Si estamos en el primer caso, como $\widetilde{C}$ rechaza todos los grafos se tiene que $\mu_I(\widetilde{C}) = 0$. Como C computa $Clique_{n,k}$, se tiene que $\mu_I(C) = \binom{n}{k} \Longrightarrow d_I(C,\widetilde{C}) = \binom{n}{k}$. Por el lema 3.2.3, $d_I(C,\widetilde{C}) \leq |C|m^2\binom{n-l-1}{k-l-1}$, así que se tiene la desigualdad

$$\binom{n}{k} \leq |C|m^2\binom{n-l-1}{k-l-1}$$

Haciendo algunas acotaciones y operaciones algebraicas,

$$m = (p-1)^l l! \leq (\lambda\sqrt{k}\log n)^{\sqrt{k}}(\sqrt{k})! \leq \lambda^{\sqrt{k}}\sqrt{k}^{\sqrt{k}}(\log n)^{\sqrt{k}}\sqrt{k}^{\sqrt{k}} = \lambda^{\sqrt{k}}k^{\sqrt{k}}(\log n)^{\sqrt{k}}$$

$$\frac{\binom{n}{k}}{\binom{n-l-1}{k-l-1}} = \frac{n}{k}\cdots\frac{n-l}{k-l} = \frac{n}{k}\cdots\frac{n-\sqrt{k}}{k-\sqrt{k}} \geq \left(\frac{n-\sqrt{k}}{k}\right)^{\sqrt{k}}$$

$$\binom{n}{k} \leq |C|m^2\binom{n-l-1}{k-l-1} \Rightarrow \frac{\binom{n}{k}}{\binom{n-l-1}{k-l-1}} \leq |C|m^2 \leq |C|\left(\lambda^{\sqrt{k}}k^{\sqrt{k}}(\log n)^{\sqrt{k}}\right)^2$$

$$\Rightarrow \left(\frac{n-\sqrt{k}}{k}\right)^{\sqrt{k}} \leq |C|\left(\lambda^{\sqrt{k}}k^{\sqrt{k}}(\log n)^{\sqrt{k}}\right)^2$$

$$\Rightarrow |C| \geq \frac{(n-\sqrt{k})^{\sqrt{k}}}{\left(\lambda^{\sqrt{k}}(\log n)^{\sqrt{k}}\right)^2 k^{3\sqrt{k}}} \geq \frac{(n-n^{1/4})^{\sqrt{k}}}{\left(\lambda^{\sqrt{k}}(\log n)^{\sqrt{k}}\right)^2 n^{3/4\sqrt{k}}} \Rightarrow |C| \geq n^{\Omega(\sqrt{k})}$$

Si estamos en el otro caso, el lema 3.2.2 nos proporciona la desigualdad

$$\mu_{II}(\widetilde{C}) \geq \left(1 - \frac{\binom{l}{2}}{k-1}\right)(k-1)^n$$

Como C computa $Clique_{n,k}$, C rechaza todos los grafos de tipo II, es decir, $\mu_{II}(C) = 0$. Esto quiere decir que $d_{II}(C, \widetilde{C}) = \mu_{II}(\widetilde{C}) - \mu_{II}(C)$ lo que lleva a la siguiente desigualdad

$$d_{II}(C, \widetilde{C}) = \mu_{II}(\widetilde{C}) - \mu_{II}(C) \geq \left(1 - \frac{\binom{l}{2}}{k-1}\right)(k-1)^n$$

Juntando esto con la cota del lema 3.2.4

$$|C|m^2 \left(\frac{\binom{l}{2}}{k-1}\right)^p (k-1)^n \geq d_{II}(C, \widetilde{C}) \geq \left(1 - \frac{\binom{l}{2}}{k-1}\right)(k-1)^n$$
$$\Rightarrow |C|m^2 \left(\frac{\binom{l}{2}}{k-1}\right)^p \geq \left(1 - \frac{\binom{l}{2}}{k-1}\right)$$

Por otro lado, como $l \leq \sqrt{k} \Rightarrow \frac{\binom{l}{2}}{k-1} \leq \frac{k-\sqrt{k}}{2(k-1)} \leq \frac{1}{2} \Rightarrow 1 - \frac{\binom{l}{2}}{k-1} \geq \frac{1}{2}$. Esto podemos utilizarlo para acotar ambos lados en la desigualdad anterior y obtenemos

$$|C|m^2 \frac{1}{2^p} \geq |C|m^2 \left(\frac{\binom{l}{2}}{k-1}\right)^p \geq \left(1 - \frac{\binom{l}{2}}{k-1}\right) \geq \frac{1}{2}$$
$$\Rightarrow |C| \geq \frac{2^{p-1}}{m^2} \geq \frac{2^{\lambda\sqrt{k}\log n - 1}}{\left(\lambda^{\sqrt{k}} k^{\sqrt{k}} (\log n)^{\sqrt{k}}\right)^2} = \frac{n^{\lambda\sqrt{k}}}{2\left(\lambda^{\sqrt{k}} k^{\sqrt{k}} (\log n)^{\sqrt{k}}\right)^2}$$
$$\Rightarrow |C| \geq \frac{n^{\lambda\sqrt{k}}}{2\left(\lambda^{\sqrt{k}} (\log n)^{\sqrt{k}}\right)^2 n^{\sqrt{k}/2}} \Rightarrow |C| \geq n^{\Omega(\sqrt{k})}$$

En ambos casos tenemos la misma cota inferior sobre C, lo que concluye la prueba. □

Este resultado encuentra una cota inferior exponencial para cualquier circuito monótono que calcule $Clique_{n,k}$. Pero esto no prueba que $Clique_{n,k} \notin \mathcal{P}/_{poly}$. Si en este resultado hubiéramos tenido en cuenta la puerta NOT, habríamos resuelto $\mathcal{P}$ *vs* $\mathcal{NP}$, pero no es el caso. Pero, ¿por qué este argumento no funciona si usamos la puerta NOT? Recordemos que gran parte de las cuentas se basan en tener control sobre la cantidad de indicadores y el tamaño de los conjuntos sobre los que trabajan. Cuando construimos un aproximador de un circuito, en el caso de la conjunción descartamos los conjuntos tales que $|X_i \cup Y_j| > l$. Si añadimos la negación sobre las variables, podría ocurrir que al hacer la conjunción de dos conjuntos se reduzca su tamaño. Por ejemplo, si $Ind(X_i) = x_1x_2x_3x_4x_5$ e $Ind(Y_i) = \bar{x}_1x_6x_7x_8x_9$, entonces $Ind(X_i) \wedge Ind(Y_j)$ es vacío, porque tiene tanto x_1 como $\bar{x}_1$. Este comportamiento destructivo de la negación hace que el argumento se venga abajo, pues ya no sabemos si un conjunto de tamaño grande se podrá reducir en un futuro, haciendo que el circuito pueda reducir su tamaño.

Capítulo 4

Resultados experimentales

Ya hemos visto que si no consideramos la negación entonces sabemos que cualquier circuito que computa $Clique_{n,k}$ es demasiado grande. También hemos visto que no podemos transformar directamente los mismos argumentos si queremos tener en cuenta la negación. Pero, ¿cómo deberían comportarse las puertas lógicas respecto a las métricas?

4.1. Aportaciones de cada puerta lógica

El objetivo de cualquier circuito que compute $Clique_{n,k}$ es acabar con una función booleana igual a la fórmula 3.1, que tiene $\binom{n}{k}$ disyunciones. Si nos fijamos, podemos ver que esta fórmula no tiene una sola negación en ella. Si lo que queremos es terminar con la fórmula de 3.1, llegará un punto en el que deje de haber negaciones. Ya hemos visto antes que, al introducir las negaciones, se rompe el argumento utilizado para demostrar el teorema 3.2.1. Pero esto no quiere decir que no podamos simplificar el circuito para controlar las negaciones.

Proposición 4.1.1 [8]**.** *Sea C un circuito booleano que computa una función f. Entonces, existe otro circuito C^* que computa f tal que $|C^*| = pol(|C|)$ y todas las puertas NOT se encuentran en el primer nivel del circuito, es decir, en las variables de entrada.*

Esquema de la demostración. La idea consiste en duplicar el circuito: una copia tendrá las variables positivas (sin negar) y otra las variables negativas (negadas). En la copia del circuito queremos que la función que llegue a cada puerta sea la negación de la función que se obtiene en el circuito original. Las leyes de de Morgan nos dan las siguientes identidades:

$$x \wedge y = \neg(\neg x \vee \neg y) \qquad , \qquad x \vee y = \neg(\neg x \wedge \neg y)$$

Negar se traduce en ir cambiando puertas AND por OR, y viceversa. Si $f = f_1 \wedge f_2$ está conectada a una negación g, añadimos una nueva puerta OR f' que tiene por

entrada $\neg f_1$ y $\neg f_2$. Después, si hay una conexión entre g y una puerta h (siendo g la entrada), la cambiamos por una conexión entre f' y h. Con la disyunción el caso es análogo, cambiando la puerta por una conjunción. Repitiendo este proceso conseguimos eliminar las puertas NOT, moviéndolas al primer nivel del circuito. □

Un corolario de esto es que si tenemos un circuito de tamaño polinómico, entonces existe un circuito donde las puertas NOT están en el primer nivel del circuito y que también tiene tamaño polinómico. Y no va a haber más puertas NOT que variables, así que el caracter polinómico del circuito depende del resto de puertas.

¿Qué nos queda? Las conjunciones y disyunciones: puertas AND y OR. Como las puertas AND y OR son las que extienden el circuito a partir del nivel inicial, si $Clique_{n,k}$ no está en $\mathcal{P}/_{poly}$ será porque la dinámica de evolución de la función computada bajo estas puertas lo impide.

Si vemos esto desde la perspectiva de las métricas, tiene que haber un punto en el que la puntuación aumente tan poco que se necesite una cantidad exponencial de puertas para llegar al final del circuito. Y este comportamiento podemos intentar simularlo bajo distintas métricas y observarlo empíricamente.

4.2. Limitaciones computacionales

A la hora de hacer experimentos, consideraremos el problema *Half-Clique*, que sabemos que también es $\mathcal{NP}$-completo. Así que el valor de k siempre será $n/2$.

Recordemos que un circuito booleano recibe como entrada una serie de variables $x \in \{0,1\}^n$ para cierto n natural. ¿Cuántas variables de entrada tendrá un circuito que compute $Clique_{n,k}$? ¿Cuántos grafos no dirigidos existen con n vértices?

Necesitamos conocer qué aristas están presentes y no lo están. Al igual que hicimos en la sección 3.2, podemos representar un grafo de n vértices con $\binom{n}{2}$ variables.

Si llamamos $m = \binom{n}{2}$, una función booleana que calcule $Clique_{n,k}$ será de la forma $f : \{0,1\}^m \to \{0,1\}$. Es decir, el tamaño de entrada de f crece cuadráticamente respecto al número de vértices del grafo. Pero este crecimiento no acaba aquí. Si queremos representar f como una cadena, vamos a necesitar 2^m bits. Y el número posible de cadenas (de funciones booleanas) de 2^m bits es 2^{2^m}. Por ejemplo, para $n = 10$ estaríamos hablando de funciones con 45 variables de entrada, con una representación que necesita $2^{45} \approx 3 \cdot 10^{13}$ bits, en un espacio de soluciones (posibles funciones) con un tamaño del orden de $2^{3 \cdot 10^{13}}$... Y aunque no representemos los grafos con cadenas de 2^m caracteres, el número de funciones booleanas posibles sigue siendo el mismo.

A la hora de hacer cálculos, esto nos limita enormemente el número de vértices. Para grafos con muchos vértices, este número se hace intratable. Por otro lado, grafos con muy pocos vértices (2, 3 o 4) en donde los límites todavía son pequeños, el problema pierde interés, pues estaríamos buscando cliques de tamaño 2 que se reduce a ver si existe alguna arista.

Un punto intermedio es tomar $n = 8$. Los cliques no son triviales, así que podemos observar comportamientos que podrían ocurrir en grafos más grandes. Además, los límites con $n = 8$ todavía son tratables con un poco de tiempo de cálculo.

4.3. Métrica semántica

A partir de aquí, mientras no se indique lo contrario, consideramos grafos con $n = 8$ vértices y cliques de tamaño $k = n/2 = 4$. Esto nos da un total de $\binom{8}{2} = 28$ posibles aristas y, por tanto, un total $2^{28} \approx 2 \cdot 10^8$ grafos de 8 vértices.

Definición 4.3.1. *Sea f una función booleana que recibe como entrada 28 variables, representando las aristas de un grafo de 8 vértices. Definimos la métrica semántica μ_s como la cantidad de grafos que acierta f. Formalmente, si f_{clique} es la función booleana para $Clique_{8,4}$:*

$$\mu_s(f) = \left|\{G : f(G) = f_{clique}(G), G \text{ es un grafo de 8 vértices}\}\right|$$

Calificamos esta métrica como semántica porque refleja el resultado de una función y no su estructura.

Por cómo está definida, maximizar μ_s requiere cumplir $f(G) = f_{clique}(G)$ para la mayor cantidad de grafos G posibles. En particular, esto ocurre para todos los grafos si $f = f_{clique}$. Por otro lado, si $f \neq f_{clique}$, es porque existe al menos un grafo G tal que $f(G) \neq f_{clique}(G)$. Así que cualquier función f que no sea f_{clique} cumple $\mu_s(f) < \mu_s(f_{clique})$.

Por otro lado, el comportamiento semántico de μ_s es idóneo para saber cuánto se parece una función f cualquiera a la función objetivo. Cuanto mayor sea $\mu_s(f)$, más grafos evalúa correctamente, así que más se parece a f_{clique}.

Además, $\mu_s(f_{clique}) = 2^{28}$, pues acepta todos los grafos posibles de 8 vértices. Podemos encontrar también una función cuya puntuación sea 0, la que menos se parece a f_{clique}, que es precisamente $\neg f_{clique}$. Luego el rango de valores posibles para μ_s está comprendido entre 0 y 2^{28}, que es un intervalo grande, pues tiene una longitud exponencial respecto al número de nodos.

En definitiva, μ_s recoge todos los comportamientos que buscamos para una métrica: tiene un amplio rango de valores, la puntuación máxima se alcanza en la función objetivo y cuanto más se parece una función f a la función objetivo, más grande es su puntuación y más se acerca a $\mu(f_{clique})$.

Para poder calcular μ_s, vamos a necesitar conocer f_{clique}, así que hay que computarla. La función estará representada por una lista de 0s y 1s, de tamaño 2^{28}, correspondiente con la cadena binaria que define la función. Esto nos obliga a fijar un orden en los grafos: cada $i \in \{0, ..., 2^{28} - 1\}$ representará a un solo grafo entre todos los 2^{28} existentes. Una vez fijada la representación de la función, necesitamos saber qué grafos cumplirán $f_{clique}[i] = 1$ y cuáles $f_{clique}[i] = 0$. Una forma de hacer esto es generar todos los grafos de 8 vértices y comprobar cuáles tienen cliques de tamaño

4. Así que necesitamos dos cosas: generar todos los grafos de 8 vértices y, dado un grafo, determinar si tiene un clique de tamaño 4.

Sabemos que hay 2^{28} grafos. Si queremos generar todos, no vamos a poder evitar este factor, que es bastante grande. Por ejemplo, si cada grafo ocupara 1 byte, estaríamos hablando de que necesitamos 2^{28} bytes = 256 Mbytes. Esto suponiendo que un grafo ocupa solamente 1 byte. Si no tenemos cuidado con su representación, estaríamos hablando de necesitar Gbytes de memoria para poder simplemente almacenar grafos de 8 vértices.

Existen dos representaciones típicas cuando se trabaja con grafos: listas y matrices de adyacencia. En las listas de adyacencia, para cada vértice se almacena una lista con los vértices adyacentes a él. En general, para un grafo $G = (V, A)$, esto supone un coste en memoria $O(|V| + |A|)$. Por su parte, las matrices de adyacencias son matrices cuadradas de tamaño $|V| \times |V|$ que almacenan en la posición i, j si los vértices v_i, v_j son adyacentes. Esto supone un coste en memoria $O(|V|^2)$. En ambos casos como poco tendríamos que utilizar una cantidad de memoria proporcional al número de vértices. Además, tenemos que lidiar con el tiempo necesario para construir estas estructuras. También tenemos que tener en cuenta que estamos generando los grafos con la idea de poder construir f_{clique}. Los grafos tienen que tener un orden determinado, y con ninguna de estas representaciones está claro cuál podría ser este orden.

Otra forma de representar un grafo es la que ya hemos usando en el texto: con $\binom{n}{2}$ variables binarias. Lo positivo de esta representación es que, si fijamos el orden de las variables, cada grafo es un número entre 0 y $2^{28} - 1$, que no solo nos proporciona el orden que estábamos buscando, sino que nos permite también almacenar cada grafo con un número entero, utilizando así menos memoria. En definitiva, generar todos los grafos es simplemente generar una lista con los números comprendidos entre 0 y $2^{28} - 1$, y luego trabajar con la representación binaria de cada número para saber qué aristas están presentes en el grafo.

Ahora solo nos queda calcular si un grafo dado tiene un clique de tamaño 4. Un grafo tendrá un clique de tamaño 4 si existe algún subconjunto de 4 vértices que forme un clique. De forma análoga a la representación binaria que hemos hecho con los grafos, los subconjuntos de 4 vértices son todas las cadenas de longitud 8 que tienen exactamente 4 bits activos (es decir, con valor 1). Estos subconjuntos podemos calcularlos recorriendo todos los números entre 0 y $2^8 - 1$, y comprobando cuáles tienen 4 bits activos. Dado un grafo y un subconjunto de 4 vértices, comprobar si existe un clique consiste en recorrer todos los pares de vértices del subconjunto, acceder al bit correspondiente a su arista y ver si está activa o no.

Con esto ya podemos generar f_{clique}. Para cada grafo $i \in \{0, ..., 2^{28} - 1\}$ recorremos los subconjuntos de tamaño 4 y si alguno es clique sabemos que $f_{clique}[i] = 1$. En otro caso, ningún subconjunto es un clique y, por tanto, $f_{clique}[i] = 0$. Para grafos de n vértices y cliques de tamaño k, este proceso lleva a un algoritmo con complejidad en tiempo $O\left(2^{\binom{n}{2}} \cdot \binom{n}{k} \cdot k^2\right)$. En nuestro caso, almacenar todos los grafos de tamaño 8 con cliques de tamaño 4 supone utilizar aproximadamente 0.5 Gbytes de memoria.

Una vez calculada la expresión de f_{clique}, obtener $\mu_s(f)$ para cualquier función

booleana f es bastante directo. El algoritmo 1 muestra cómo implementarlo, obteniendo una complejidad en tiempo del orden de $O(2^{\binom{n}{2}})$.

Algoritmo 1 Algoritmo para calcular $\mu_s(f)$

Entrada: f función booleana como una lista de 0s y 1s. f_{clique} función que computa $Clique_{8,4}$ como lista de 0s y 1s.
Salida: $\mu_s(f)$

```
1: result ← 0
2: for i ∈ {0, ..., μ_s(f_clique) − 1} do
3:     if f[i] = f_clique[i] then
4:         result ← result + 1
5:     end if
6: end for
7: return result
```

Como los grafos están numerados, basta recorrerlos en orden mientras llevamos la cuenta de qué grafos coinciden con la función objetivo. Una vez sabemos calcular μ_s, podemos generar funciones y experimentar con ellas.

Una ventaja de representar las funciones como una lista es que podemos generar funciones aleatorias de forma muy sencilla. Simplemente asignamos a cada posición de la lista el valor 0 o 1 aleatoriamente y habremos conseguido una función. Además, el número de posibles funciones distintas es enorme, $2^{2^{28}}$. En un espacio tan grande, la probabilidad de encontrar una excepción es prácticamente nula. Si tengo una caja con $2^{2^{28}}$ pelotas y, tras sacar unas pocas (aleatoriamente) veo que todas son de color rojo, puedo intuir que la mayoría de las pelotas serán de color rojo (es prácticamente imposible que justo haya sacado las pocas pelotas que hay de color rojo). En otras palabras, si con pocas muestras observamos comportamientos parecidos, podemos obtener conclusiones sobre cómo se comportan las funciones en un aspecto promedio.

Queremos observar la capacidad de las puertas AND y OR según nos vamos acercando a $\mu_s(f_{clique})$. Más concretamente, nos interesa analizar el máximo incremento que son capaces de producir. Aunque pueda darse el caso de que una métrica tenga que perder puntuación para luego ganarla, tiene que alcanzar la puntuación final para completar el circuito, así que en algún momento tendrá que aumentar la puntuación. Si el máximo incremento de puntuación que puede aportar cada puerta es muy pequeño (o al menos muy pequeño tras alcanzar ciertas puntuaciones), entonces harían falta muchísimas puertas para computar f_{clique}.

Veamos primero cómo se comporta μ_s con las puertas AND. La tabla 4.1 muestra cómo varían las puntuaciones al aplicar la puerta AND a funciones generadas aleatoriamente.

Las funciones f_1 y f_2 representan funciones generadas aleatoriamente. No es casualidad que los primeros dígitos de $\mu_s(f_1)$ y de $\mu_s(f_2)$ coincidan. Como las funciones están generadas aleatoriamente, la probabilidad de acertar un grafo i es precisamente $1/2$. Con esto podemos obtener la esperanza media de μ_s para una función generada

$\mu_s(f_1)$	$\mu_s(f_2)$	$\mu_s(f_1 \wedge f_2)$	$\mu_s(f_1 \wedge f_2) - \mu_s(f_1)$	$\mu_s(f_1 \wedge f_2) - \mu_s(f_2)$
134222863	134219686	140675398	6452535	6455712
134222188	134231090	140690381	6468193	6459291
134216855	134219624	140678503	6461648	6458879
134223077	134220635	140682323	6459246	6461688
134217541	134222583	140683754	6466213	6461171
134219580	134231447	140691594	6472014	6460147
134220448	134214842	140676926	6456478	6462084
134226123	134212791	140680982	6454859	6468191
134210199	134216261	140679104	6468905	6462843
134209245	134207574	140670356	6461111	6462782
134208605	134218554	140678085	6469480	6459531

Tabla 4.1: Comparación de μ_s sobre funciones aleatorias.

aleatoriamente.

$$\mathbb{E}[\mu_s(f)] = \sum_{i=0}^{2^{28}-1} P[f[i] = f_{clique}[i]] = \sum_{i=0}^{2^{28}-1} \frac{1}{2} = 2^{27} \tag{4.1}$$

El valor esperado es $2^{27} = 134217728$, que podemos observar que se aproxima a los valores obtenidos para las funciones aleatorias.

Lo siguiente que observamos es que al hacer el AND, la función resultante tiene una mayor puntuación que las dos anteriores. De hecho, parece que el aumento es aproximadamente el mismo en todos los casos. El porqué pasa esto no es tan simple como lo anterior, pero podemos intentar ver qué ocurre. Pensemos en un grafo i cualquiera.

- Si $f_1[i] = f_2[i]$ (ya sea un 0 un 1) $\Longrightarrow f_1[i] \wedge f_2[i] = f_1[i] = f_2[i]$. Por tanto, no va a cambiar el valor aportado por i al hacer la conjunción: si se tenía $f_1[i] = f_{clique}[i]$ (resp. $\neq$), se tendrá $(f_1 \wedge f_2)[i] = f_{clique}[i]$ (resp. $\neq$). Así que este caso no aporta nada al aumento de $\mu_s(f_1 \wedge f_2)$.

- Si $f_1[i] \neq f_2[i]$. Supongamos que se tiene $f_1[i] = 1$ y $f_2[i] = 0$. Entonces, $f_1[i] \wedge f_2[i] = 0$. Si el valor correcto para i es $f_{clique}[i] = 0$, entonces $f_1 \wedge f_2$ aumenta la puntuación sobre f_1 y se mantiene igual respecto a f_2. Si el valor correcto para i fuera $f_{clique}[i] = 1$, entonces $f_1 \wedge f_2$ mantiene la puntuación sobre f_2 pero pierde respecto a f_1. Sin embargo, los roles de f_1 y f_2 van cambiando según evaluamos distintos grafos. Como las funciones están generadas aleatoriamente, las veces en las que $f_1[i] = 1$ y $f_2[i] = 0$ serán aproximadamente las mismas que las veces en las que $f_1[i] = 0$ y $f_2[i] = 1$. Es decir, las veces que se pierde puntuación sobre f_1 (resp. f_2) se compensan con las veces que se gana puntuación sobre f_1 (resp. f_2). Esto nos dice que la distribución sobre los grafos que tienen cliques sobre los que no los tienen no es uniforme. El aumento de puntuación sobre μ_s nos dice que hay más grafos tales que $f_{clique}[i] = 0$ que los que cumplen $f_{clique}[i] = 1$.

El valor medio de los aumentos producidos (últimas dos columnas de la tabla de la figura 4.1) está aproximado por 6460000. Una cuenta sencilla nos dice por cuántas puertas más habría que pasar si tomamos esto como incremento medio.

$$2^{27} + 6460000 \cdot x = 2^{28} \Longrightarrow x = \frac{2^{27}}{6460000} \approx 21 \tag{4.2}$$

Es decir, con 20 puertas AND más, habríamos alcanzado la puntuación final. Esto parece estar lejos de indicar que se necesita una cantidad exponencial de puertas, aunque hay que tener en cuenta el valor de n y de k. De hecho, la cota encontrada por Razborov para circuitos monótonos era del orden de $2^{n/8}$. En nuestro caso, $n = 8$, y $2^{8/8} = 2$, que es mucho menor a 20. Aun así, la aproximación es lejana a la realidad, pues con funciones aleatorias no alcanzamos una puntuación cercana a la función objetivo. De hecho, las funciones se mantienen en un rango cercano a la mitad de esta puntuación.

Como hemos generado una expresión para f_{clique}, podemos construir funciones con cualquier puntuación. Si queremos encontrar una función f tal que $\mu_s(f) = t$, basta con seleccionar t grafos distintos y forzar que f acierte en ellos y falle en el resto. Con esta idea podemos generar funciones con puntuación cercana a $\mu_s(f_{clique})$, aunque podemos optimizar los cálculos si pensamos el problema a la inversa.

Si queremos conseguir funciones que se aproximen a la función final, es más eficiente marcar los grafos que difieren de f_{clique} que aquellos que se mantienen. Cuanto más cerca estemos de $\mu_s(f_{clique})$, la cantidad de grafos que fallan es mucho menor a la que aciertan, así que es más rápido pensar en quitar grafos que en añadirlos. Dicho de otra forma, computar una función f tal que $\mu_s(f) = t$ es lo mismo que computar f tal que $\mu_s(f) = \mu_s(f_{clique}) - t'$, con $t = \mu_s(f_{clique}) - t'$. Cuanto más nos acercamos a $\mu_s(f_{clique})$, menor será t'. Este proceso se queda reflejado en el algoritmo 2.

Algoritmo 2 Algoritmo para calcular una función f tal que $\mu_s(f) = \mu_s(f_{clique}) - t$.

Entrada: f_{clique} función que computa $Clique_{8,4}$ como lista de 0s y 1s. $t \in \{0, \ldots, \mu_s(f_{clique})\}$
Salida: f tal que $\mu_s(f) = \mu_s(f_{clique}) - t$.

```
1: S := {}
2: while |S| < t do
3:     x ← elemento aleatorio de {0, ..., μ_s(f_clique) − 1}
4:     S ← S ∪ {x}
5: end while
6: f := copia de μ_s(f_clique)
7: for i ∈ S do
8:     f[i] ← ¬f_clique[i]
9: end for
10: return f
```

Al igual que con las funciones aleatorias, podemos generar funciones de este tipo y compararlas con la puerta AND. Una forma de aproximarse progresivamente a la puntuación máxima es ir tomando porcentajes de esta y generar una función con esa puntuación.

Definición 4.3.2. *Sea $p \in [0, 100]$. Decimos que una función f es p-cercana a f_{clique} si cumple $\mu_s(f) \approx \frac{p}{100}\mu_s(f_{clique})$. Una función f que sea p-cercana a f_{clique} se denotará por f_p.*

El algoritmo 2 nos permite obtener funciones p-cercanas a f_{clique} para cualquier $p \in [0, 100]$. Bastaría tomar $t = \left(1 - \frac{p}{100}\right)\mu_s(f_{clique})$ para obtener la puntuación deseada. En particular, la expresión 4.1 muestra que las funciones generadas aleatoriamente son 50-cercanas a f_{clique}.

Las funciones p-cercanas nos permiten aproximar gradualmente f_{clique} y observar cómo se comporta un circuito en sus últimos niveles. La tabla 4.2 muestra los aumentos máximos obtenidos después de aplicar puertas AND sobre pares de funciones f_p del mismo tipo (mismo p) para $p \in \{85, \ldots, 99\}$. Cada casilla recoge el aumento máximo obtenido tras comparar funciones con el mismo p. Se puede apreciar cómo a medida que nos acercamos a f_{clique} cada vez el aumento obtenido es menor. Esto es esperable, pues cuanto más cerca se está de f_{clique} más grafos se aciertan y, por tanto, la cantidad de grafos que tienen que cambiar para aumentar la puntuación es menor. No obstante, podría pasar que dos funciones f_p del mismo tipo perdieran puntuación al pasar por una puerta AND. Por ejemplo, si $f_{clique} = 1111$ (puntuación 4) y $f_1 = 0111$, $f_2 = 1110$ (ambas con puntuación 3), se tendría $f_1 \wedge f_2 = 0110$ (puntuación 2). Que la tabla 4.2 muestre aumentos quiere decir que algún par ha generado este incremento en la puntuación, no que siempre sea este el caso. Pero, ¿cómo de grande es este aumento?

p	Aumento		p	Aumento
85	3297550		93	1687384
86	3117107		94	1459559
87	2925141		95	1231595
88	2728666		96	993914
89	2539117		97	753030
90	2323529		98	506557
91	2119207		99	256489
92	1908773			

Tabla 4.2: Aumento máximo obtenido para funciones p-cercanas a f_{clique}.

Una forma de estudiar el tamaño de este aumento es repitiendo la cuenta que hicimos en 4.2 con cada aumento obtenido en las distintas f_p. La gráfica de la figura 4.1 muestra la tendencia que siguen los números de puertas necesarias para terminar el circuito en cada f_p. La curva en color azul está formada por los puntos (p, x_p), donde x_p se obtiene para cada f_p como en 4.2. Podemos ver que esta curva tiene

una pendiente muy suave, indicando que según nos acercamos a f_{clique} el número de puertas necesarias para acabar el circuito se mantiene constante. Es decir, llega un punto en que las funciones no son capaces de crecer más y la única forma de alcanzar f_{clique} es aumentando la puntuación muy poco a poco.

La recta de color rojo representa el salto que hay de f_{50} a f_{85}. Esta recta es más pronunciada que la curva azul. De $p = 50$ a $p = 85$ conseguimos reducir 10 puertas, mientras que de $p = 85$ a $p = 99$ apenas conseguimos lograr una puerta de diferencia. La recta de color amarillo representa el desnivel existente que todavía hay que lograr para alcanzar f_{clique}. Prácticamente es un precipicio que no encaja ni sigue la tendencia descrita por la curva azul.

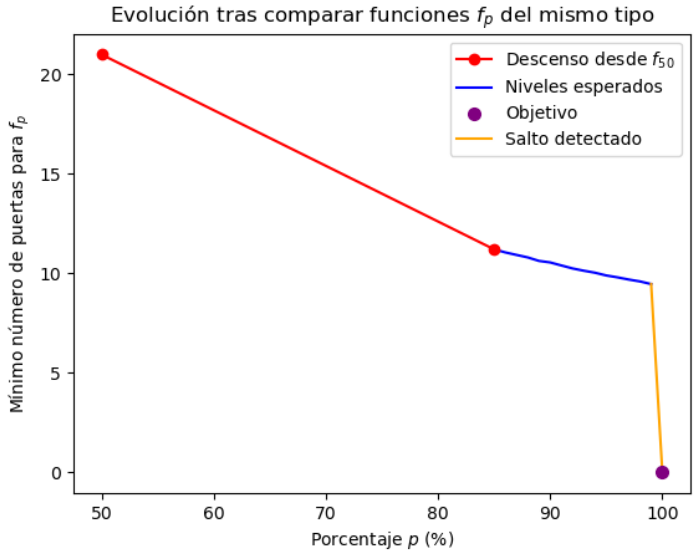


Figura 4.1: Evolución del mínimo número de puertas obtenido para f_p.

Esta comparación se ha hecho evaluando el crecimiento de μ_s sobre pares de funciones f_p del mismo tipo. Pero es posible que la realidad sea otra y que las funciones f_p se mezclen entre sí. Podemos comparar pues distintas funciones f_p y ver si se observa un comportamiento parecido.

La gráfica de la figura 4.2 muestra la puntuación obtenida tras comparar pares de funciones p-cercanas seleccionadas al azar para $p \in \{85, \ldots, 99\}$. Si se han comparado dos funciones f_{p_1} y g_{p_2}, la gráfica recoge los puntos $(p_1, \mu_s(f_{p_1} \wedge g_{p_2}))$ y $(p_2, \mu_s(f_{p_1} \wedge g_{p_2}))$.

La curva de color naranja muestra la tendencia de los valores máximos para cada p, mientras que la curva de color azul muestra la tendencia de los valores mínimos para cada p. Por otro lado, la recta de color verde muestra la recta formada por $\mu_s(f_p)$. Nótese que esta recta refleja un aumento constante de la puntuación, pues el aumento de $\mu_s(f_p)$ es lineal respecto a p.

Lo primero que observamos es que a partir de $p = 90$ ciertas comparaciones disminuyen la puntuación, comportamiento que no quedaba reflejado en la tabla 4.2. De hecho, cuanto más aumenta el valor de p, es más común disminuir la puntua-

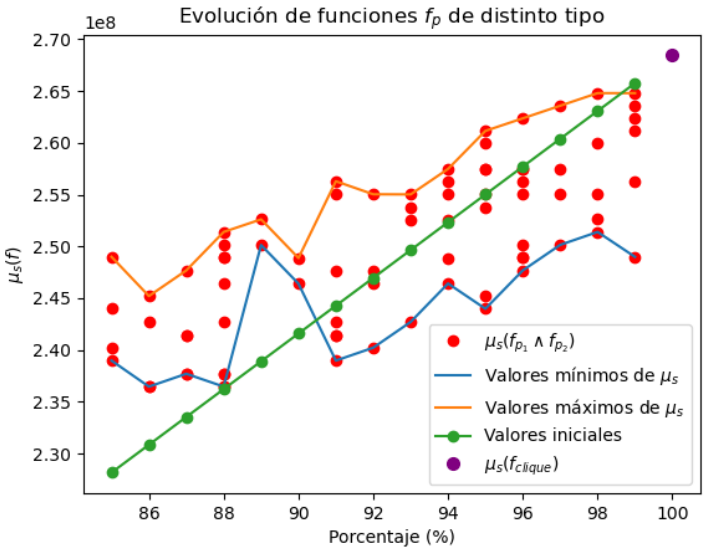


Figura 4.2: Evolución de las comparaciones entre pocas f_p de distinto tipo.

ción que aumentarla. Por otro lado, el aumento máximo que experimenta μ_s (curva naranja) parece acompañar la tendencia de la recta verde al principio, pero según avanza el parámetro p se va aplanando. De hecho, al final prácticamente se aplana del todo, llegando a estar por debajo de la recta verde. Al igual que ocurría en la gráfica de la Figura 4.1, cuanto más nos acercamos al final del circuito, más difícil es ganar puntuación, por poco que sea. De hecho, la curva azul también muestra cierto decrecimiento, indicando no solo una dificultad a la hora de crecer, sino también una tendencia a decrecer la puntuación. No obstante, en algunos puntos la curva de crecimiento máximo y mínimo tiene ciertos picos, que pueden deberse a la falta de datos.

Podemos repetir este experimento comparando más funciones y comprobar si estas tendencias se mantienen. Generando más funciones p-cercanas, obtenemos la gráfica de la figura 4.3.

Vemos que las curvas de valores máximos (naranja) y mínimos (azul) se han suavizado, pero se sigue observando esa frontera de $p = 90$ donde las funciones empiezan a perder puntuación en vez de ganarla. Además, se sigue cumpliendo que la curva de máximos se aplana cuando se aproxima a $\mu_s(f_{clique})$.

¿Qué ocurre con la puerta OR? Podríamos repetir todo este análisis con esta puerta, pero no hace falta si observamos lo siguiente. Al estar analizando una propiedad semántica, podríamos tomar otra función como función objetivo, en particular, con $\neg f_{clique}$. Acercarse a $\neg f_{clique}$ es lo mismo que alejarse de f_{clique}, y viceversa. Es decir, las funciones muy lejanas a f_{clique} son funciones muy cercanas a $\neg f_{clique}$. Como f_{clique} y $\neg f_{clique}$ son complementarias, presentarán comportamientos similares cuando las funciones se van aproximando a ellas.

Por otro lado, la disyunción de dos funciones $f \vee g$ puede expresarse a partir de las leyes de de Morgan como $\neg(\neg f \wedge \neg g)$. Si f y g son cercanas a f_{clique}, entonces $\neg f$

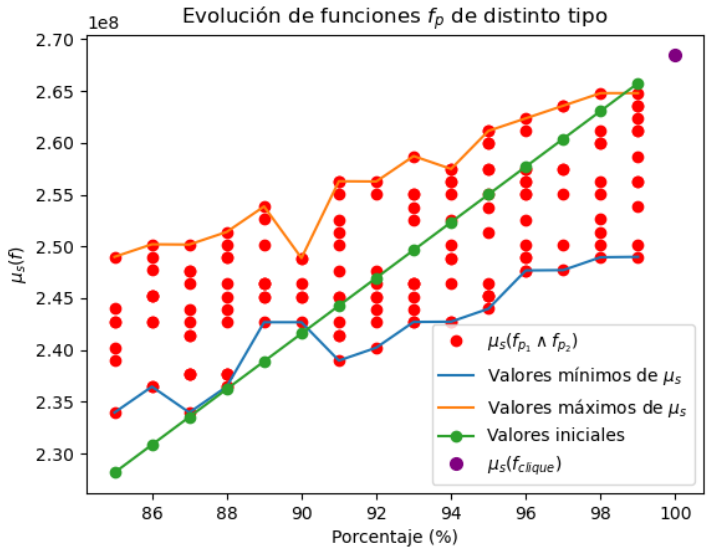


Figura 4.3: Evolución de las comparaciones entre más f_p de distinto tipo.

y $\neg g$ son cercanas a $\neg f_{clique}$. Hemos visto que la tendencia sobre las puertas AND es que cada vez es más difícil ganar puntuación, así que a $\neg f \wedge \neg g$ le costará acercarse a $\neg f_{clique}$ o, lo que es lo mismo, le costará alejarse de f_{clique}. Es decir, la disyunción de funciones cercanas a f_{clique} se traduce a la negación de una función que le cuesta acercarse a $\neg f_{clique}$, que es lo mismo que acercarse a f_{clique}.

En definitiva, tanto la conjunción como la disyunción de dos funciones cercanas a f_{clique} presentan esta tendencia en la que cada vez es más difícil ganar puntuación.

Pero esta tendencia ocurre con funciones aleatorias. Observar que las funciones aleatorias tienden a perder puntuación cuando se acercan a f_{clique}, no quiere decir que no existan funciones que sí que sean capaces de hacer crecer a μ_s. El problema que tiene μ_s es que no presta atención a cómo es $Clique_{n,k}$. Tiene en cuenta sus resultados, pero no cómo llega a ellos. Esta tendencia a decrecer cuando nos acercamos al objetivo podría ser engañosa. Una interpretación sería que no es que cada vez sea más difícil aumentar la puntuación, sino que cada vez se necesitan funciones más específicas para poder aumentar la puntuación. Para encontrar estas funciones habría que no solo explorar más exhaustivamente el espacio de búsqueda (que es gigantesco), sino también poder conocer qué funciones podrían aparecer en un circuito que compute f_{clique}. Podría ocurrir que justo las funciones que mejor funcionan bajo μ_s no pudieran aparecer en ningún circuito, o que para que aparecieran hicieran falta una gran cantidad de puertas. El carácter semántico de μ_s dificulta esta tarea, pues no tiene en cuenta cómo son las funciones. Esto motiva la siguiente sección, en la que se considera una métrica más sintáctica, que tiene en cuenta la estructura de las funciones y no tanto sus evaluaciones sobre los grafos de entrada.

4.4. Métrica sintáctica

¿Cómo explotamos el hecho de que estamos calculando $Clique_{n,k}$? ¿Qué diferencia a $Clique_{n,k}$ del resto de problemas? Recordemos un momento cómo era la fórmula 3.1.

$$K_{n,k} = \bigvee_{X \subseteq \{1,\dots,n\}, |X|=k} Ind(X)$$

donde los $Ind(X)$ son indicadores de clique. Si en vez de escribir la fórmula respecto a los indicadores lo hacemos respecto a las variables de entrada, es decir, las que representan las aristas del grafo, obtenemos la siguiente expresión.

$$f_{clique} = \bigvee_{\substack{X \subseteq \{1,\dots,n\} \\ |X|=k}} \bigwedge_{\substack{u,v \in X \\ u \neq v}} x_{\{u,v\}} \tag{4.3}$$

donde cada $x_{\{u,v\}}$ es la variable correspondiente a la arista entre el nodo u y v.

Básicamente esta expresión desarrolla cada indicador como una conjunción de aristas. Esta expresión nos proporciona una forma natural de comparar el parecido de una función f cualquiera con f_{clique}, Veámoslo con un ejemplo. Consideremos grafos de 4 vértices (numerados de 1 a 4) y cliques de tamaño 3. Entonces, la expresión 4.3 queda

$$f_{clique} = x_{\{1,2\}}x_{\{1,3\}}x_{\{2,3\}} \ \vee \ x_{\{1,3\}}x_{\{1,4\}}x_{\{3,4\}} \ \vee \ x_{\{2,3\}}x_{\{2,4\}}x_{\{3,4\}}$$

Si tomamos $f = x_{\{1,2\}}x_{\{1,3\}} \ \vee \ x_{\{2,3\}}x_{\{3,4\}}$, podemos ver que la primera cláusula de f comparte dos variables con la primera cláusula de f_{clique}, y que la segunda cláusula de f comparte dos variables con la tercera cláusula de f_{clique}. Si tomamos por ejemplo $g = x_{\{1,2\}}x_{\{1,3\}} \ \vee \ x_{\{1,3\}}x_{\{2,3\}}$, ambas cláusulas comparten dos variables con la primera cláusula de f_{clique}.

Comparar cláusulas de una función f con f_{clique} nos da información de cuánto se parecen ambas funciones. Si las cláusulas de f se parecen mucho a las cláusulas de f_{clique}, entonces f también se parecerá mucho a f_{clique}. Y viceversa. Si las cláusulas de f no tienen nada que ver con las de f_{clique}, quiere decir que f está lejos de parecerse a f_{clique}. Así que, para comparar una función f cualquiera con f_{clique}, basta con expresar f en forma normal disyuntiva y ver cómo de parecidas son sus cláusulas con las de f_{clique}.

Definición 4.4.1. *Sea f una función booleana en forma normal disyuntiva. Sea f_{clique} la función booleana que calcula $Clique_{n,k}$. Definimos la métrica sintáctica μ_x como la suma de los parecidos de cada cláusula de f con cada cláusula de f_{clique}. Formalmente, sean $f = \bigvee_{i=1}^{m} f_i$, $f_{clique} = \bigvee_{j=1}^{\binom{n}{k}} f_{clique}^{(j)}$. Sea $r = \max(m, \binom{n}{k})$. Extendemos si hace falta f o f_{clique} con cláusulas vacías para que ambas tengan exactamente r cláusulas. Sea S el conjunto formado por todas las biyecciones entre conjuntos de r elementos. Si $comp(a, b)$ es la cantidad de variables que tiene la cláusula a que están*

presentes en la cláusula b, entonces:

$$\mu_x(f) = \max_{\sigma \in S} \left\{ \sum_{i=1}^{r} comp(f_i, f_{clique}^{(\sigma(i))}) \right\}$$

Aunque la definición formal de μ_x es compleja, lo que calcula es una asignación entre cláusulas que maximice el parecido entre ellas. Para las fórmulas de ejemplo anteriores, la métrica obtendría los siguientes valores.

Para $f = x_{\{1,2\}}x_{\{1,3\}} \ \vee \ x_{\{2,3\}}x_{\{3,4\}}$,

- La primera cláusula de f, $x_{\{1,2\}}x_{\{1,3\}}$, se asignaría con $x_{\{1,2\}}x_{\{1,3\}}x_{\{2,3\}}$. La comparación resulta en

$$comp(x_{\{1,2\}}x_{\{1,3\}}, x_{\{1,2\}}x_{\{1,3\}}x_{\{2,3\}}) = 2$$

- La segunda cláusula de f, $x_{\{2,3\}}x_{\{3,4\}}$, se asignaría con $x_{\{2,3\}}x_{\{2,4\}}x_{\{3,4\}}$. La comparación resulta en

$$comp(x_{\{2,3\}}x_{\{3,4\}}, x_{\{2,3\}}x_{\{2,4\}}x_{\{3,4\}}) = 2$$

- Sumando las dos asignaciones, se llega a $\mu_x(f) = 2 + 2 = 4$.

Para $g = x_{\{1,2\}}x_{\{1,3\}} \ \vee \ x_{\{1,3\}}x_{\{2,3\}}$, ocurre algo distinto.

- La primera cláusula de g, $x_{\{1,2\}}x_{\{1,3\}}$, se asignaría con $x_{\{1,2\}}x_{\{1,3\}}x_{\{2,3\}}$. La comparación resulta en

$$comp(x_{\{1,2\}}x_{\{1,3\}}, x_{\{1,2\}}x_{\{1,3\}}x_{\{2,3\}}) = 2$$

- La segunda cláusula de g, $x_{\{1,3\}}x_{\{2,3\}}$, comparte más variables con la primera cláusula de f_{clique}. Pero esta cláusula ya está asignada, así que no podemos utilizarla otra vez. Esto implica que la segunda cláusula de g se asignaría con, por ejemplo, la segunda cláusula de f_{clique}, $x_{\{1,3\}}x_{\{1,4\}}x_{\{3,4\}}$. La comparación resulta en

$$comp(x_{\{1,3\}}x_{\{2,3\}}, x_{\{1,3\}}x_{\{1,4\}}x_{\{3,4\}}) = 1$$

- Sumando las dos asignaciones, se llega a $\mu_x(g) = 2 + 1 = 3$.

El ejemplo con la función g muestra por qué es importante que las asignaciones entre cláusulas sean independientes. En otro caso, podría ocurrir que dos cláusulas distintas se refieran a la misma, dando una puntación mayor a la real.

A diferencia de lo que ocurría con μ_s, calificamos a μ_x como una métrica sintáctica porque hace hincapié en la estructura interna de cada función y no en sus valores de salida.

Por cómo está definida, la manera de alcanzar el valor máximo para μ_x es obteniendo una asignación en la que cada cláusula contenga a las de f_{clique}. La única

forma de conseguir esto es que la función contenga todas las cláusulas de f_{clique}. En particular, el valor máximo se alcanza en $\mu_x(f_{clique})$, que es igual a $\binom{k}{2}\binom{n}{k}$. No obstante, no es la única función con esta puntuación. Cualquier función f que en forma normal disyuntiva se pueda expresar como $f = f_{clique} \vee f'$ para alguna función f' cumplirá $\mu_x(f) = \mu_x(f_{clique})$. Pero estamos de acuerdo con esto. Si una función en forma normal disyuntiva contiene a f_{clique}, quiere decir que se parece a ella, así que tiene sentido considerar que alcanza el valor máximo para μ_x. De este modo, alcanzar la puntuación máxima es condición necesaria para lograr f_{clique}. Si alcanzar esta puntuación requiere muchas puertas, entonces alcanzar f_{clique} también requerirá muchas puertas. Aun así, cualquier f cumple $\mu_x(f) \leq \mu_x(f_{clique})$.

Por otro lado, una función que contenga una sola variable, $f = x_i$, cumplirá $\mu_x(f) = 1$. Por tanto, el rango de valores de μ_x está comprendido entre 1 y $\binom{k}{2}\binom{n}{k}$.

En definitiva, μ_x también cumple los requisitos que buscamos para una métrica: ocupa un amplio rango de valores, el máximo se alcanza en f_{clique}, cuanto más se parece una función f a f_{clique} más grande es $\mu_x(f)$ y al inicio del circuito tiene el valor mínimo.

¿Cómo calculamos μ_x? Al igual que ocurría con μ_s, necesitamos conocer f_{clique} y representar las funciones de un modo adecuado que permita facilitar las cuentas. Como queremos que las funciones estén en forma normal disyuntiva, representaremos las funciones como listas de listas de variables. Las variables de entrada estarán numeradas de 1 a $\binom{n}{2}$ de manera única, cada número representando un único par de vértices. Por ejemplo, la función $f = x_1x_2 \vee x_4x_6x_3$ se representaría como `[[1,2], [4,6,3]]`. Con esta representación, para generar f_{clique} podemos generar todos los subconjuntos de k elementos y, para cada uno de estos, calcular las $\binom{k}{2}$ variables que representan ese conjunto. Este procedimiento se traduce en un algoritmo que construye la expresión de f_{clique} con una complejidad en tiempo del orden de $O\left(\binom{n}{k}\binom{k}{2}\right)$.

Habiendo construido ya f_{clique}, podemos obtener $\mu_x(f)$ para cualquier función f. Si intentamos aplicar directamente la fórmula de la definición 4.4.1 tendríamos que recorrer todas las permutaciones de las cláusulas de f_{clique} y quedarnos con la que maximice las comparaciones. Esto supone hacer $\binom{n}{k}!$ operaciones. Por ejemplo, para $n = 8$, $k = 4$, habría que hacer $\binom{8}{4}! = 70! \approx 10^{100}$ iteraciones, que es un número intratable de iteraciones. Así que no podemos recorrer todas las permutaciones posibles. Sin embargo, podemos transformar este cálculo en un problema de asignación [3, Capítulo 10]. Si definimos la matriz de costes $\mathcal{C}$ tal que

$$\mathcal{C}_{i,j} = comp(f_i, f^{(j)}_{clique}) \ \forall f_i \text{ cláusula de } f \ , \ \forall f^{(j)}_{clique} \text{ cláusula de } f_{clique}$$

calcular μ_x consiste en encontrar una asignación máxima para la matriz $\mathcal{C}$. El algoritmo húngaro [10] permite resolver este problema con una complejidad en tiempo $O(r^3)$, donde r es el orden de la matriz de costes $\mathcal{C}$.

La representación de las funciones booleanas como listas de listas nos permite calcular la matriz de costes $\mathcal{C}$ cómodamente. Una vez calculada esta matriz, utilizando el algoritmo húngaro podemos calcular el valor de μ_x para cualquier f. El algoritmo 3 refleja este proceso. Si m es la longitud máxima de cualquier cláusula entre f y f_{clique}, es decir, $m = \text{máx}\{|f_i|, |f^{(j)}_{clique}|\}$, podemos calcular $comp(f_i, f^{(j)}_{clique})$

con un algoritmo con complejidad en tiempo $O(m)$. Si $r = \text{máx}\{|f|, |f_{clique}|\}$, la matriz de costes $\mathcal{C}$ tendrá un tamaño acotado por $r \times r$, haciendo que el algoritmo 3 tenga una complejidad en tiempo acotada por $O(mr^2+r^3) = O(mr^3)$. Si suponemos que el caso límite de este algoritmo es tomar $f = f_{clique}$, se tendrá $r = \binom{n}{k}$ y $m = \binom{k}{2}$, obteniendo una complejidad $O\left(\binom{k}{2}\binom{n}{k}^3\right)$.

Algoritmo 3 Algoritmo para calcular $\mu_x(f)$

Entrada: f función booleana como lista de listas, f_{clique} función que computa $Clique_{n,k}$ como lista de listas.
Salida: $\mu_x(f)$.

```
1: l_1 ← |f|
2: l_2 ← |f_clique|
3: C := matriz de costes de tamaño l_1 × l_2
4: for i ∈ {0, ..., l_1 − 1} do
5:     for i ∈ {0, ..., l_2 − 1} do
6:         C_{i,j} = comp(f[i], f_clique[j])
7:     end for
8: end for
9: result ← algoritmo_húngaro(C)
10: return result
```

En comparación con el algoritmo 1 que computaba μ_s con complejidad $O\left(2^{\binom{n}{2}}\right)$, el algoritmo 3 es considerablemente más eficiente, lo que nos permite hacer experimentos con más datos.

Pero antes de hacer experimentos, ¿cómo esperamos que se comporten las puertas AND y OR bajo μ_x?

Proposición 4.4.1. *Sean f y g dos funciones booleanas expresadas en forma normal disyuntiva, es decir, de la forma $f = \bigvee_{i=1}^{m_1} f_i$, $g = \bigvee_{j=1}^{m_2} g_j$. Entonces,*

$$\mu_x(f \vee g) \leq \mu_x(f) + \mu_x(g)$$

Demostración. Nótese que $f \vee g = \bigvee_{i=1}^{m_1} f_i \ \vee \ \bigvee_{j=1}^{m_2} g_j$, así que las cláusulas de $f \vee g$ son la unión de las cláusulas de f con las de g. Distinguimos dos casos en función de las cláusulas de f_{clique} que se asignan al calcular $\mu_x(f)$ y $\mu_x(g)$.

- Si las cláusulas asignadas a f son distintas de las cláusulas asignadas a g. En el caso de que $f_i \neq g_j$, la maximalidad de μ_x asegura que esta asignación será óptima en $f \vee g$, es decir, $\mu_x(f \vee g) = \mu_x(f) + \mu_x(g)$. En el caso de que existan i, j tales que $f_i = g_j$, entonces $f \vee g$ tiene menos cláusulas que f y g por separado. La maximalidad de μ_x asegura que el resto de cláusulas no repetidas no pueden asignarse a mejores cláusulas de f_{clique} y, como en el conjunto tenemos menos sumandos, se tiene $\mu_x(f \vee g) \leq \mu_x(f) + \mu_x(g)$.

- Si existen cláusulas asignadas a f que son iguales a las cláusulas asignadas a g. Si existiera otra asignación con el mismo valor para $\mu_x(f)$ y $\mu_x(g)$ tal que no compartan cláusulas, estaríamos en el caso anterior. En otro caso, de nuevo la maximalidad de μ_x asegura el resultado. Al unir las cláusulas en $f \vee g$, alguna tendrá que ser asignada a una cláusula de f_{clique} que tenga menor valor, haciendo que $\mu_x(f \vee g) \leq \mu_x(f) + \mu_x(g)$.

□

Proposición 4.4.2. *Sea C un circuito booleano que computa una función f utilizando únicamente puertas OR y NOT, teniendo todas sus puertas NOT en el primer nivel. Entonces,*

$$|C| \geq \mu_x(f) - 1$$

Demostración. Procedemos por inducción estructural en C.

- Si $C = x_i$, entonces $f = x_i$ y $\mu_x(f) = 1$. C no tiene puertas, así que $|C| = 0 = \mu_x(f) - 1$.
- Si $C = \bar{x}_i$, entonces $f = \bar{x}_i$ y $\mu_x(f) = 0$, pues f_{clique} no tiene ninguna negación. C tiene una puerta, así que $|C| = 1 \geq \mu_x(f) - 1$.
- Si $C = C_1 \vee C_2$, entonces $f = f_1 \vee f_2$ para ciertas funciones f_1 y f_2 computadas por C_1 y C_2, respectivamente. Por inducción, $|C_1| \geq \mu_x(f_1) - 1$ y $|C_2| \geq \mu_x(f_2) - 1$. La proposición 4.4.1 nos dice que $\mu_x(f_1 \vee f_2) \leq \mu_x(f_1) + \mu_x(f_2)$. Por tanto,

$$\begin{aligned} |C| = |C_1| + |C_2| + 1 \geq \mu_x(f_1) - 1 + \mu_x(f_2) - 1 + 1 = \\ = \mu_x(f_1) + \mu_x(f_2) - 1 \geq \mu_x(f_1 \vee f_2) - 1 = \mu_x(f) - 1 \end{aligned}$$

□

La proposición 4.1.1 nos permite reducir cualquier circuito que calcule $Clique_{n,k}$ a otro circuito que tenga las puertas NOT en el primer nivel añadiendo un número polinómico de puertas. Por otro lado, $\mu_x(f_{clique}) = \binom{k}{2}\binom{n}{k}$. Si suponemos que un circuito C calcula $Clique_{n,k}$ sin puertas AND, la proposición 4.4.2 nos dice que

$$|C| \geq \mu_x(f_{clique}) - 1 = \binom{k}{2}\binom{n}{k} - 1$$

así que habríamos encontrado una cota inferior de tamaño exponencial. Aunque observando la fórmula 4.3 nos damos cuenta de que la única forma de computar $Clique_{n,k}$ sin usar la puerta AND y teniendo las puertas NOT en el primer nivel, es teniendo $k \leq 2$.

De todos modos, ¿qué ocurre si intentamos repetir estos cálculos teniendo en cuenta la puerta AND? Intentemos acotar el incremento máximo que se puede obtener. Si $f = \bigvee_{i=1}^{m_1} f_i$ y $g = \bigvee_{j=1}^{m_2} g_j$, entonces $f \wedge g = \bigvee_{i=1}^{m_1} \bigvee_{j=1}^{m_2} f_i \wedge g_j$. Si r es una cota superior del tamaño de las cláusulas de f y g, entonces el incremento máximo que se

puede obtener está acotado por $m_1 m_2 2r$, pues se forman $m_1 m_2$ cláusulas de tamaño como mucho $2r$. Por su parte, con estos parámetros se tiene que $\mu_x(f) \leq m_1 r$ y $\mu_x(g) \leq m_2 r$. En general, $m_1 m_2 2r \not\leq m_1 r + m_2 r$, así que no se cumpliría la versión con AND de la proposición 4.4.1. Esta condición es esencial en el paso inductivo $C = C_1 \vee C_2$ en la prueba de la proposición 4.4.2, así que no podríamos llegar a este resultado siguiendo el mismo camino.

La cota que se obtendría para la puerta AND sería $\mu_x(f \wedge g) \leq 2 \cdot \mu_x(f) \cdot \mu_x(f)$. Es precisamente este comportamiento cuadrático el que podría causar que la métrica creciera rápidamente combinando puertas. No obstante, el resultado de la proposición 4.4.2 sí que nos dice algo. Es precisamente la puerta AND la que podría causar que un circuito no tuviera tamaño exponencial. En el caso de μ_s pudimos deshacernos de la puerta NOT, pero con μ_x podemos centrarnos en estudiar cómo evolucionan las funciones a través de las puertas AND. Y este comportamiento podemos comprobarlo experimentalmente.

4.4.1. Análisis sin la puerta NOT

Supongamos por ahora que nuestro circuito no tiene puertas NOT (luego veremos cómo añadirlas de forma sencilla). Al igual que hicimos con μ_s, vamos a tomar grafos con $n = 8$ vértices y cliques de tamaño $k = n/2 = 4$. En el caso de μ_s, tanto generar funciones aleatorias como calcular el AND de dos funciones dadas era muy simple, pues estaban representadas como listas de 0s y 1s. En el caso de μ_x, el proceso no es tan directo. Para construir una función aleatoria f,

1. Seleccionamos una cantidad aleatoria de cláusulas de f_{clique}. A partir de estas cláusulas construiremos las cláusulas de f.
2. Para cada cláusula seleccionada, tomamos una cantidad aleatoria de variables de esa cláusula. Estas variables compondrán cada cláusula de f.
3. Juntamos todas las nuevas cláusulas (listas de enteros) en una lista de listas, obteniendo la función f.

Por otro lado, para construir $f \wedge g$ a partir de f y g,

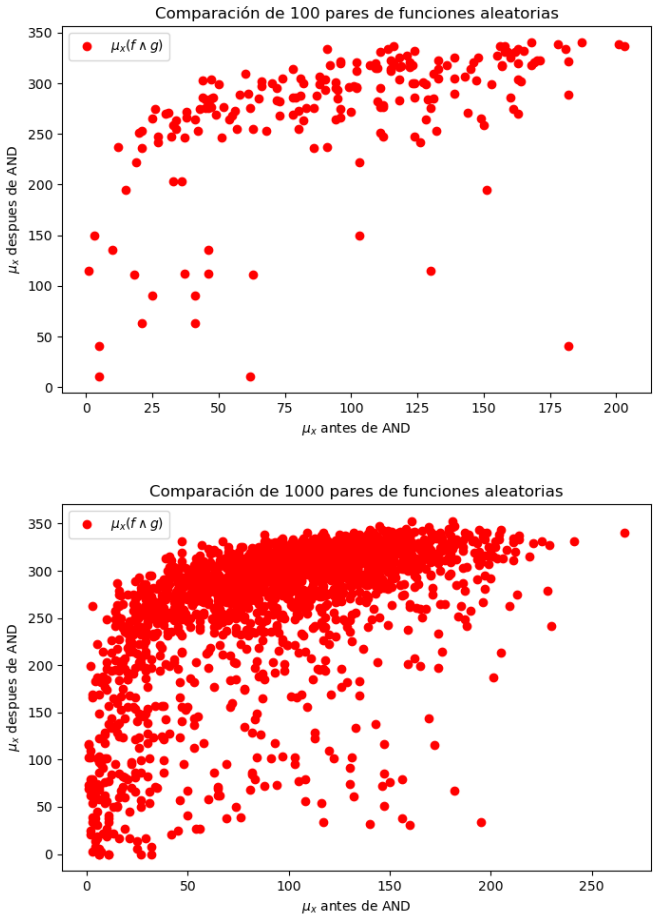
1. Convertimos cada par $\{f[i], g[j]\}$ en una nueva lista $[f[i] + g[j]]$ ($+$ es la concatenación de listas) obteniendo así todas las posibles cláusulas de $f \wedge g$.
2. Para cada una de estas nuevas listas eliminamos las variables repetidas.
3. Si alguna de las nuevas cláusulas tiene más de $\binom{k}{2}$ elementos podemos descartarla, pues al no haber puerta NOT nunca vamos a poder reducirla, siendo imposible llegar a f_{clique}.
4. Para cada lista restante, comprobamos si está contenida en alguna otra lista. Esto es lo que se conoce como absorción. En general, $A \vee AB = A$, así que si alguna cláusula contiene a otra, podemos descartarla.

5. Juntamos todas las cláusulas restantes en una lista de listas, obteniendo la función $f \wedge g$.

Técnicamente hablando, el paso 3 hace que la función obtenida no sea exactamente $f \wedge g$. Pero en el peor de los casos, estamos convirtiendo una función que sabemos que no puede estar en el circuito (no hay puertas NOT) en una que sí puede estarlo. Al no haber negaciones, la única forma de reducir una cláusula es mediante absorción.

Si en algún momento se llega a una cláusula de tamaño mayor a $\binom{k}{2}$, tendrá que ser convertida en irrelevante por una cláusula de tamaño más pequeño. Si mantenemos todas las cláusulas de tamaño superior, μ_x podría conseguir puntuación con cláusulas "sobredefinidas" que todavía no sabemos con qué otras cláusulas se van a reducir. Ignorar estas cláusulas sobredefinidas es una manera de hacer que una función f no obtenga un valor que todavía no ha alcanzado (esta corrección no hará falta al añadir la puerta NOT). Si los incrementos a través de la puerta AND son muy bajos a pesar de esta corrección, también lo serán sin ella.

Las gráficas de la figura 4.4 muestran cómo cambia μ_x tras comparar 100, 1000 y 10000 pares de funciones generadas aleatoriamente, respectivamente. Si f y g son dos funciones generadas aleatoriamente, cada gráfica contiene los puntos $(\mu_x(f), \mu_x(f \wedge g))$ y $(\mu_x(g), \mu_x(f \wedge g))$.



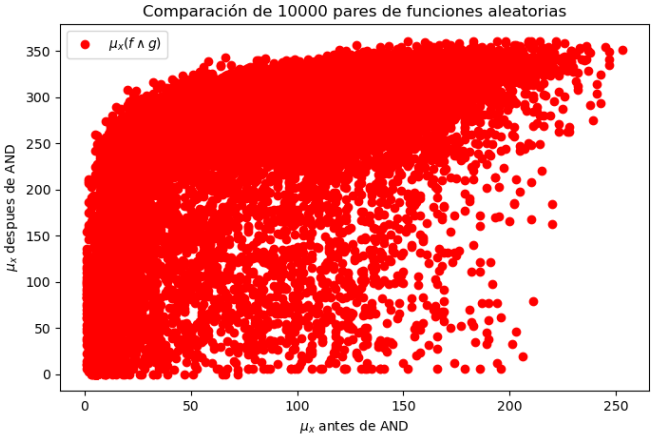


Figura 4.4: Comparación de la puerta AND sobre μ_x para funciones generadas aleatoriamente.

Se observa claramente cómo llega un momento en el que las funciones AND no son capaces de aumentar su puntuación, atascando su productividad. Con funciones aleatorias alcanzan una puntuación de 350, lejos de la puntuación máxima, que con $n = 8$ y $k = 4$ es 420. La tendencia parece ser que, una vez alcanzada esta frontera de 350 puntos, la puntuación en cada paso aumenta muy poco, haciendo necesaria una gran cantidad de puertas para alcanzar f_{clique}. Pero, ¿por qué ocurre esto?

Para que una función tenga una puntuación alta tiene que tener cláusulas muy parecidas a las de f_{clique}. En particular, tendrá que contener gran parte de las $\binom{n}{2}$ variables de entrada del circuito. Por otro lado, para obtener el incremento cuadrático del que hemos hablando antes, es necesario que la absorción no invalide prácticamente cláusulas, y que el tamaño de cada cláusula sea el máximo posible. ¿Cómo conseguimos esto? Haciendo la conjunción de dos funciones que compartan el mínimo número de variables posibles. Si dos cláusulas no comparten variables, su conjunción tendrá el tamaño máximo y la absorción apenas podrá eliminar cláusulas. Esto motiva la siguiente definición.

Definición 4.4.2. *Sean s y t dos enteros positivos. Decimos que una función $f = \bigvee_{i=1}^{m} f_i$ es endogámica sobre s y t si cumple las siguientes condiciones.*

1. *f tiene s cláusulas.*
2. *Cada cláusula de f tiene tamaño t, es decir, $|f_i| = t \ \forall i \in \{1, \ldots, m\}$.*
3. *El número de vértices a los que se refieren las variables de las cláusulas de f es el mínimo posible. Dicho de otra forma, no existe ninguna función g que cumpla 1 y 2 involucrando menos vértices que f.*

Denotaremos a este número de vértices por $\mathcal{V}$ y a una función endogámica sobre s y t por $f_{s,t}$.

Las funciones endogámicas son las que recogen la mayor cantidad de información con el menor número de vértices posibles. Al hacer la conjunción de dos funciones

endogámicas que se refieren a variables distintas es precisamente cuando tenemos que la conjunción mantiene el tamaño máximo posible. Las cláusulas de $f_{s,t}$ pretenden recoger trozos de cliques para que luego, al juntarse con otros trozos de cliques, se cree un trozo de clique más grande. En particular, este comportamiento debería ser más notable con puntuaciones pequeñas, donde el aumento cuadrático en la puntuación y en el número de cláusulas no alcanza el tamaño de f_{clique}.

Una forma de generar funciones endogámicas es ir generando sus cláusulas. Fijamos un orden sobre las variables del circuito, que servirá como criterio para saber qué variables usar en $f_{s,t}$. Generamos una cláusula de tamaño t (con las variables dadas en el orden tomado). Después, probamos a cambiar cada una de las variables de esta cláusula por la siguiente que no hayamos añadido todavía. Si esta nueva cláusula no la hemos generado aún, formará parte de $f_{s,t}$. El proceso se va repitiendo con todas las cláusulas que hayan sido aceptadas, hasta completar s cláusulas (o quedarse sin variables). De este modo, generamos $f_{s,t}$.

La gráfica de la figura 4.5 muestra el incremento obtenido tras aplicar AND con funciones $f_{s,t}$ (color rojo) referidas a distintas variables en comparación con el incremento obtenido con funciones aleatorias (color azul) en el mismo rango de puntuación. Las funciones $f_{s,t}$ se han generado aleatoriamente con $t \in \left\{1, \ldots, \binom{k}{2}/2\right\}$ y $s \in \left\{1, \ldots, \sqrt{\binom{n}{k}}\right\}$. De forma general, si se ha realizado la conjunción sobre dos funciones f y g, la gráfica recoge los puntos $(\mu_x(f), \mu_x(f \wedge g))$ y $(\mu_x(g), \mu_x(f \wedge g))$.

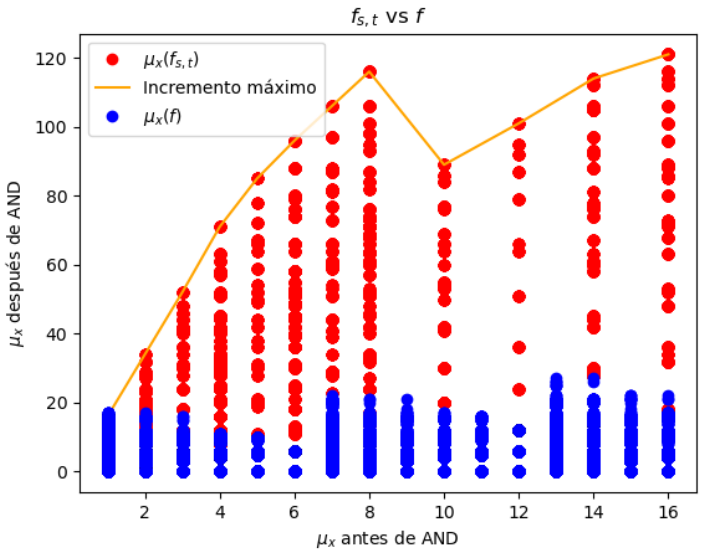


Figura 4.5: Comparación de $f_{s,t}$ con funciones aleatorias.

Podemos observar que el incremento máximo producido por las funciones endogámicas (curva naranja) es bastante superior al que generan las funciones aleatorias, al menos con valores pequeños.

Interesa acotar el incremento máximo que se produce en μ_x. En caso de que este incremento máximo sea pequeño, querrá decir que se necesitan muchas puertas AND

para poder alcanzar f_{clique}. Si las funciones $f_{s,t}$ son aquellas que explotan mejor el comportamiento cuadrático de μ_x bajo la puerta AND, podemos intentar estudiar cuándo las funciones $f_{s,t}$ empiezan a dejar de ser tan endogámicas.

Demos la vuelta a la definición un momento y supongamos que los parámetros que tenemos son $\mathcal{V}$ y t en vez de s y t. ¿Cuántas cláusulas podría tener $f_{s,t}$ como máximo? Si $f_{s,t}$ se refiere a $\mathcal{V}$ vértices, entonces como mucho puede contener $\binom{\mathcal{V}}{2}$ variables distintas (una por cada par de vértices). Si cada cláusula tiene t elementos, el problema se reduce a conocer la cantidad de formas en las que se pueden repartir $\binom{\mathcal{V}}{2}$ en conjuntos de t elementos. Este número es precisamente $\binom{\binom{\mathcal{V}}{2}}{t}$. Así que para $\mathcal{V}$ y t tenemos una cota para s.

Volviendo a s y t, saber a cuántos vértices se puede referir $f_{s,t}$ como poco es precisamente encontrar el menor $\mathcal{V}$ que cumpla la siguiente desigualdad:

$$s \leq \binom{\binom{\mathcal{V}}{2}}{t}$$

El caso límite es cuando s alcanza la cota. Pero queremos calcular f_{clique}, que sabemos que tiene un total de $\binom{n}{k}$ cláusulas, así que una estimación en el caso límite es tomar s como el número de cláusulas que tiene f_{clique}. Por otro lado, en f_{clique} cada cláusula tiene tamaño $\binom{k}{2}$. Al hacer una conjunción, el tamaño de las cláusulas aumenta como mucho el doble de su tamaño inicial. Así que podemos estimar el valor de t en el caso extremo como $t = \binom{k}{2}/2$. Es decir, queremos que se cumpla la siguiente desigualdad.

$$s = \binom{\binom{\mathcal{V}}{2}}{t} \leq \binom{n}{k} \Longrightarrow \binom{\binom{\mathcal{V}}{2}}{\binom{k}{2}/2} \leq \binom{n}{k} \tag{4.4}$$

Y aquí está el problema. El término de la izquierda crece más rápido que el de la derecha, haciendo que llegue antes a este límite de vértices. En general, obtener el valor exacto para $\mathcal{V}$ en la expresión 4.4 no es fácil. Una forma analítica de intentar abordar el problema es mediante la función Γ. Esta función es una forma de extender la función factorial a los números complejos. Está definida como

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt, \qquad Re(z) > 0$$

Si n es un entero positivo, se cumple que $\Gamma(n) = (n-1)!$. Esta propiedad permite expresar la desigualdad 4.4 en términos de Γ, que podría utilizarse para intentar buscar una solución general en función de n y k.

No obstante, computacionalmente podemos abordar el problema de forma más sencilla. El coeficiente binomial de la parte izquierda de la desigualdad es creciente en función de $\mathcal{V}$. Si fijamos n y k, podemos hacer una búsqueda binaria sobre el espacio de soluciones de $\mathcal{V}$ para encontrarlo. El algoritmo 4 muestra este proceso, con una complejidad en tiempo de $O(\log(cota_{n,k}) \cdot \tau(n,k))$, donde $\tau(n,k)$ es la complejidad de calcular un coeficiente binomial. El parámetro $cota_{n,k}$ es un valor suficientemente grande para acotar el rango de la búsqueda, en función de n y de k.

Algoritmo 4 Algoritmo para calcular $\mathcal{V}$

Entrada: n, k y $cota_{n,k}$ enteros positivos.
Salida: $\mathcal{V}$.

```
1: ini ← 1
2: fin ← cota_{n,k}
3: result ← fin
4: b_1 ← binom(n, k)
5: b_2 ← binom(k, 2)
6: while ini ≤ fin do
7:     mitad ← (ini+fin)/2
8:     b_3 ← binom(mitad, 2)
9:     b_4 ← binom(b_3, b_2)
10:    if b_4 ≤ b_1 then
11:        result ← mitad
12:        ini ← mitad + 1
13:    else
14:        fin ← mitad − 1
15:    end if
16: end while
17: return result
```

La gráfica de la figura 4.6 muestra el $\mathcal{V}$ obtenido para grafos de hasta 10 vértices y cliques de tamaño $k = n/2$. Podemos observar que con grafos con pocos nodos no hay problemas con las funciones $f_{s,t}$ y nunca pierden información. No obstante, a partir de $n = 7$ se empieza a ver un desajuste en el que las funciones $f_{s,t}$ alcanzan su máximo con menos nodos de los disponibles.

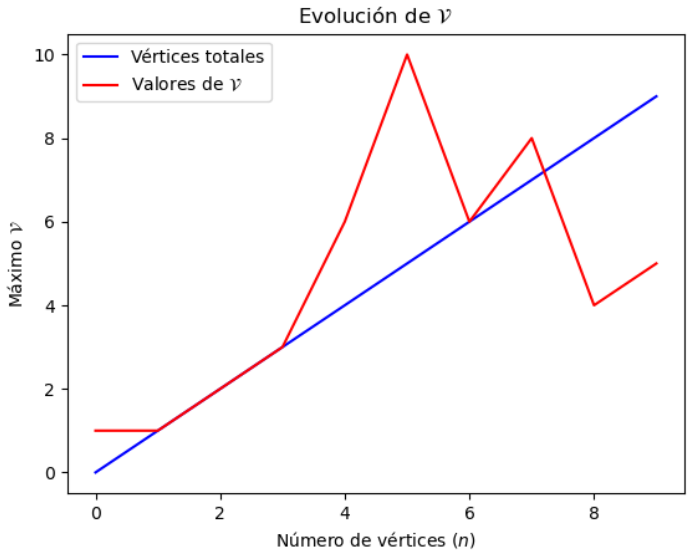


Figura 4.6: Evolución del máximo $\mathcal{V}$ en grafos con pocos vértices.

La gráfica de la figura 4.7 muestra la misma comparación pero con grafos de hasta $n = 100$ vértices. A medida que aumenta el número de vértices, esta diferencia es cada vez mayor, indicando que las funciones $f_{s,t}$ dejan de producir puntuación cada vez más pronto.

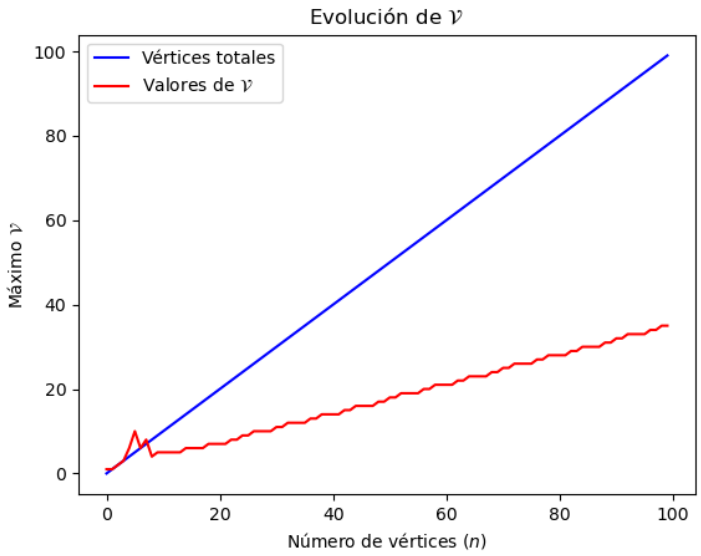


Figura 4.7: Evolución del máximo $\mathcal{V}$ en grafos con más vértices.

En resumen, las funciones $f_{s,t}$ parecen ser las que más puntuación producen antes de alcanzar el número de cláusulas que tiene f_{clique}, momento en el que todavía la puntuación puede crecer sin que se eliminen cláusulas por absorción o porque su tamaño es demasiado grande. Sin embargo, el número de vértices de un grafo limita el número de vértices al que puede referirse una función $f_{s,t}$. Según aumentamos el número de vértices totales, las funciones endogámicas alcanzan antes la frontera de $\binom{n}{k}$ cláusulas. Es decir, llenan sus cláusulas de variables antes de poder referirse a todos los vértices del grafo, haciendo que la puerta AND deje de aumentar la puntuación al mismo ritmo que con puntuaciones pequeñas.

4.4.2. Análisis con la puerta NOT

Una forma natural de añadir las negaciones en la representación es teniendo el cuenta el signo: los números positivos serán variables sin negar, mientras que los negativos serán variables negadas. Por ejemplo, $f = x_1x_2 \vee \overline{x}_4x_6\overline{x}_3 \vee \overline{x}_1x_8$ estaría representada por `[[1,2], [-4,6,-3], [-1, 8]]`. Si consideramos que las variables negadas son distintas a las variables sin negar, el algoritmo 3 sigue funcionando correctamente, pues una variable negada nunca va añadir puntuación sobre una cláusula de f_{clique}.

Por otro lado, crear una función aleatoria con variables negadas sigue el mismo proceso que con las variables sin negar, añadiendo un paso extra.

1. Seleccionamos una cantidad aleatoria de cláusulas de f_{clique}. A partir de estas cláusulas construiremos las cláusulas de f.

2. Para cada cláusula seleccionada, tomamos una cantidad aleatoria de variables de esa cláusula. Estas variables compondrán cada cláusula de f.

3. *(Hasta aquí igual que el caso sin NOT)* Para cada cláusula de f, tomamos una cantidad de variables aleatorias y les cambiamos el signo para convertirlas en negaciones.

4. Juntamos todas las nuevas cláusulas (listas de enteros) en una lista de listas, obteniendo la función f.

Por su parte, para construir $f \wedge g$ a partir de f y g, el proceso cambia ligeramente con respecto al caso sin NOT.

1. Convertimos cada par $\{f[i], g[j]\}$ en una nueva lista $[f[i] + g[j]]$ ($+$ es la concatenación de listas) obteniendo así todas las posibles cláusulas de $f \wedge g$.

2. Para cada una de estas nuevas listas eliminamos las variables repetidas.

3. *(Hasta aquí igual que el caso sin NOT)* Si alguna de las nuevas cláusulas contiene una variable negada y sin negar al mismo tiempo, esta se anula, así que podemos descartarla.

4. Para cada lista restante, comprobamos si está contenida en alguna otra lista (absorción).

5. Juntamos todas las cláusulas restantes en una lista de lista, obteniendo la función $f \wedge g$.

Recordemos que consideramos a las variables negadas como variables distintas a las que no lo están. Aunque x_i y $\overline{x}_i$ en verdad se refieren a la misma variable, en la práctica es más cómodo tratarlas como elementos distintos.

A diferencia de lo que ocurría con la conjunción en el caso sin la negación, ahora sí que estamos calculando la función exacta $f \wedge g$, pues las cláusulas descartadas son aquellas que se anulan por completo por culpa del NOT. En este caso, no hace falta eliminar aquellas cláusulas que tengan un tamaño mayor a $\binom{k}{2}$, pues la puerta NOT puede eliminarlas en un futuro.

Al igual que hicimos con el caso sin NOT, podemos comprobar qué ocurre si comparamos funciones generadas aleatoriamente, también para grafos con $n = 8$ y $k = n/2 = 4$. Las gráficas de la figura 4.8 muestran cómo cambian μ_x a través de la puerta AND tras comparar 100, 1000, 10000 funciones aleatorias, respectivamente.

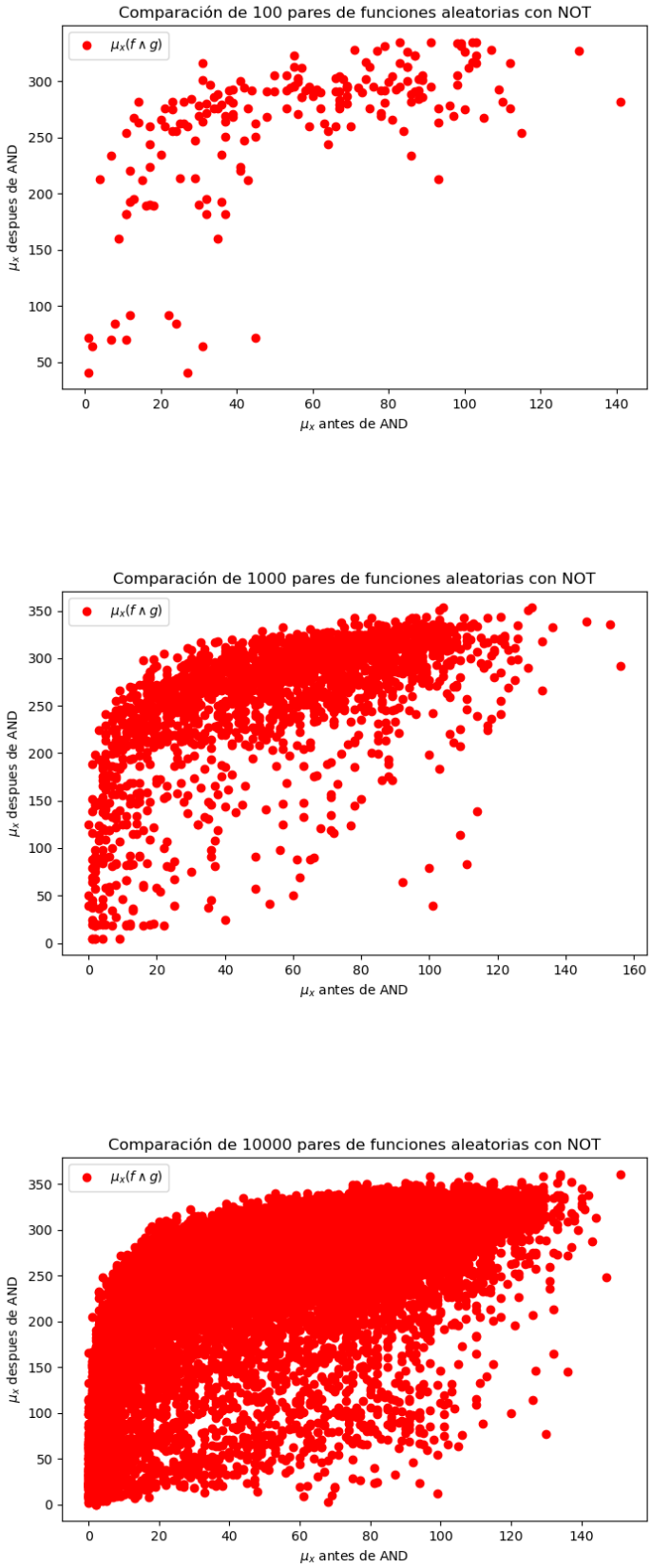


Figura 4.8: Comparación de la puerta AND sobre μ_x para funciones generadas aleatoriamente considerando negaciones.

Podemos observar que, al igual que ocurría sin tener en cuenta el NOT, las conjunciones empiezan a dejar de ser productivas alrededor de 350, lejos del 420 que se quiere alcanzar. El incremento máximo parece seguir una tendencia logarítmica. En comparación con los resultados observados en la figura 4.4, en el caso del NOT parece que se alcanza ligeramente antes esta frontera de 350 puntos, y también se observa que el incremento mínimo es mejor que cuando no considerábamos el NOT. Pero las gráficas en sí muestran resultados prácticamente idénticos.

En cuanto a las funciones $f_{s,t}$, podemos hacer un análisis parecido considerando la puerta NOT. Hay que modificar ligeramente la desigualdad 4.4 para aproximarnos mejor a la realidad.

Una función $f_{s,t}$ sin negaciones que se refiere a $\mathcal{V}$ nodos contiene a lo sumo $\binom{\mathcal{V}}{2}$ variables. En el caso de las negaciones, estamos duplicando este número, así que una función $f_{s,t}$ que se refiere a $\mathcal{V}$ nodos, contiene a lo sumo $2 \cdot \binom{\mathcal{V}}{2}$ variables. Por otro lado, al hacer la conjunción de dos cláusulas que tienen negaciones, es posible que se cancelen entre ellas. Pero esto lo que causaría es tener menos cláusulas. El crecimiento máximo del tamaño de cada una sigue siendo como mucho el doble, así que podemos seguir tomando $t = \binom{k}{2}/2$. Por último, que dos cláusulas se cancelen sí que afecta al número total de cláusulas que podemos esperar. Al hacer una conjunción, el número de cláusulas aumenta como mucho cuadráticamente respecto al número de cláusulas inicial, así que en vez de acotar con $\binom{n}{k}$, acotamos por $\binom{n}{k}^2$. Juntando todo, obtenemos la desigualdad 4.5.

$$\binom{2 \cdot \binom{\mathcal{V}}{2}}{\binom{k}{2}/2} \leq \binom{n}{k}^2 \tag{4.5}$$

Al igual que ocurría antes, fijados n y k, el término de la izquierda de la desigualdad es creciente en función de $\mathcal{V}$, así que podemos adaptar el algoritmo 4 para obtener $\mathcal{V}$ en este caso. La gráfica de la figura 4.6 muestra la tendencia de $\mathcal{V}$ para grafos de n vértices y cliques de tamaño $k = n/2$.

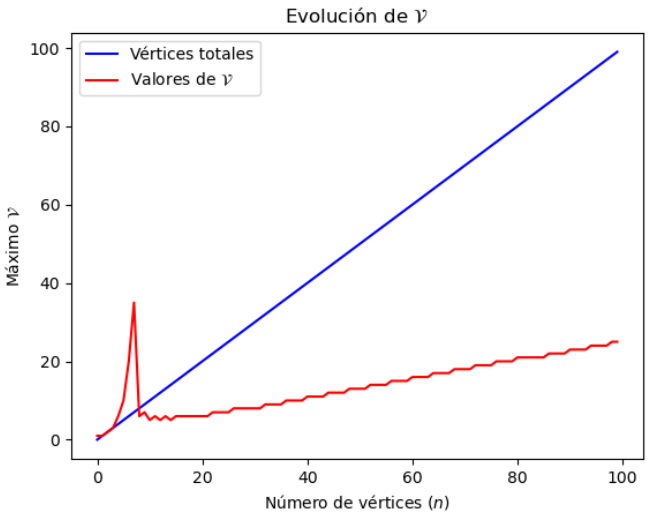


Figura 4.9: Comparación de la puerta AND sobre μ_x para funciones generadas aleatoriamente considerando negaciones.

Vemos que esta tendencia parece ser la misma que la que obteníamos en la figura 4.7. El pico que se observa con grafos de pocos nodos no debe interpretarse como el valor numérico mostrado. Superar la recta de color azul (número de vértices reales), significa que las funciones $f_{s,t}$ son capaces de mantener toda la información posible hasta llegar a cubrir todos los vértices. Es este pico precisamente el que nos dice que las funciones $f_{s,t}$ funcionan mejor con grafos con pocos nodos que con grafos con muchos nodos.

4.4.3. Simulación de un circuito que calcula $Clique_{n,k}$

Como último experimento, podemos intentar construir un circuito que compute $Clique_{n,k}$ y comprobar si se sigue observando esta tendencia a cada vez crecer menos cuando nos acercamos a la función objetivo. Para ello, vamos a basarnos en los conceptos que hay detrás de los algoritmos genéticos.

En términos generales, un algoritmo genético es una técnica de optimización que pretende encontrar el máximo o mínimo de una función dada. En ellos, se parte de unos datos iniciales (población) entre los que se selecciona una parte para que avancen (evolucionar) durante el proceso y se mezclen entre ellos (reproducir). Esto provoca que la población inicial cambie y haya nuevos datos potencialmente mejores que los que teníamos al inicio, en términos de la función que se quiere optimizar. Se descartan los peores datos de esta nueva población y se repite el proceso. Idealmente, cada población de datos es mejor que su predecesora, así que producirá mejores datos al mezclarlos, optimizando cada vez más la función objetivo (simulando un proceso de selección natural).

En general, los algoritmos genéticos consiguen buenos resultados cuando la función a optimizar tiene pocos máximos/mínimos locales o todos los máximos/mínimos son cercanos al valor óptimo de la función objetivo. No obstante, el número de iteraciones necesarias para alcanzar este punto óptimo depende del problema que se trate. La solución óptima encontrada es óptima en comparación con el resto de soluciones generadas durante la simulación, así que a veces resulta difícil saber cuándo parar.

En nuestro caso, queremos optimizar μ_x, pues ya hemos visto que en su valor máximo la función contiene a f_{clique}. La población inicial serán las variables de entrada a un circuito, negadas y sin negar, pues podemos suponer que las negaciones están en el primer nivel. Con la representación que teníamos hasta ahora, si $A = \binom{n}{2}$ (número de variables) la población inicial $\mathcal{I}$ será

$$\mathcal{I} = \Big\{ \texttt{[i]} : i \in \{1, \ldots, A\} \Big\} \cup \Big\{ \texttt{[-i]} : i \in \{1, \ldots, A\} \Big\}$$

Por otro lado, el proceso de mezclar será seleccionar dos funciones de la población actual y hacer o bien su conjunción o bien su disyunción (recordemos que las puertas NOT están ya en las variables, así que podemos olvidarlas). En cada iteración, seleccionaremos una cantidad aleatoria de funciones sobre las que realizar la disyunción y otra cantidad aleatoria sobre la que realizar la conjunción.

Como al principio del circuito las funciones tendrán una puntuación pequeña, no tiene sentido empezar a descartar funciones tan pronto. Dejaremos que la población alcance un tamaño máximo y, cuando lo sobrepase, eliminaremos de la población aquellas funciones que tengan menor μ_x. Además, impondremos un número máximo de cláusulas para cada función.

Realizamos la simulación sobre grafos de $n = 8$ vértices y cliques de tamaño $k = n/2 = 4$. Seleccionamos 15 pares de funciones para hacer disyunciones, 15 pares de funciones para hacer conjunciones, un tamaño máximo de 300 funciones para la población y un tamaño máximo de 150 cláusulas para cada función.

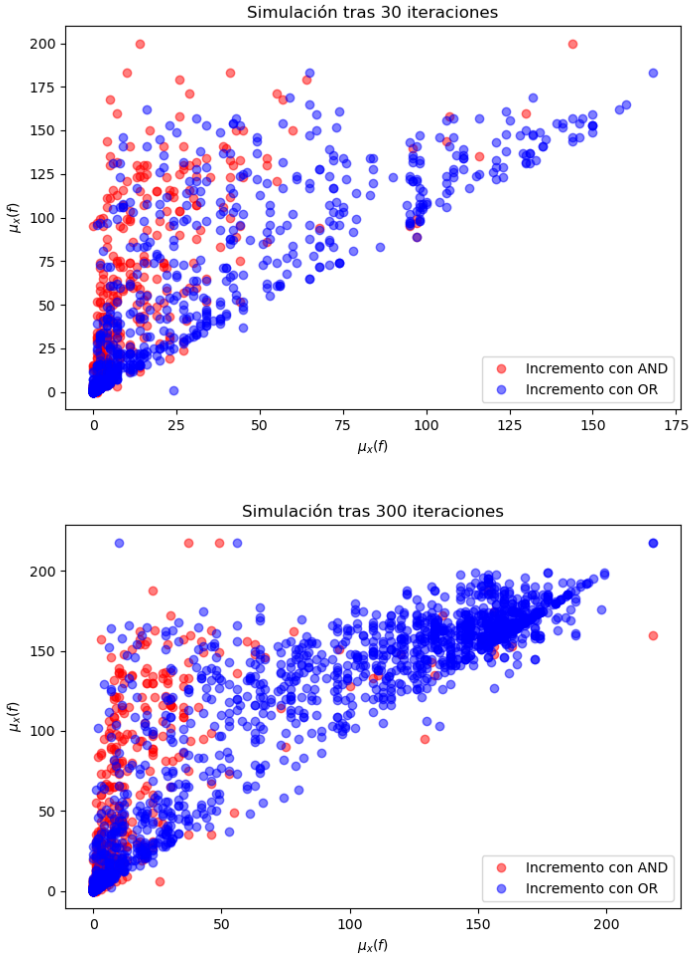


Figura 4.10: Simulación de un circuito que computa $Clique_{n,k}$

La primera gráfica de la figura 4.10 muestra cómo son los incrementos de μ_x tras 30 iteraciones. En azul están indicados los incrementos producidos por puertas OR, mientras que en rojo los producidos por puertas AND. Se observa cómo las puertas OR siguen la tendencia lineal predicha por la proposición 4.4.2. Por su parte, las puertas AND son capaces de aumentar más la puntuación, pero tras 30 iteraciones aún están lejos del valor final $\mu_x(f_{clique})$.

La segunda gráfica de la figura 4.10 muestra la evolución del circuito tras 300

iteraciones. Podemos ver que, aunque hayamos aumentado las iteraciones, el incremento sigue la misma tendencia. Las puertas AND no son capaces de producir más que las OR, y las OR siguen una tendencia lineal.

Chapter 5

Conclusiones y trabajo futuro

En este texto se ha estudiado cómo se comportan los circuitos que computan $Clique_{n,k}$. Se ha desarrollado con detalle el resultado al que llegó Razborov sobre circuitos monótonos y se han realizado experimentos sobre dos métricas, una semántica y otra sintáctica.

Los experimentos realizados sobre μ_s, la métrica semántica, nos proporcionan evidencias de que, según nos acercamos al final de un circuito, cada vez se necesitan funciones más específicas para poder seguir avanzando. No obstante, con μ_s no obtenemos información sobre cómo son las funciones que hay en el interior de un circuito que computa $Clique_{n,k}$.

Por otro lado, los experimentos realizados sobre μ_x, la métrica sintáctica, nos proporcionan resultados más específicos. Bajo μ_x, las puertas OR producen un aumento lineal en la puntuación. Las funciones con puntuación pequeña son capaces de aumentar más cuando atraviesan una puerta AND. Sin embargo, cuando alcanzan puntuaciones cercanas a f_{clique}, el aumento cada vez es menor, describiendo una curva logarítmica.

Estos resultados proporcionan nuevos indicios de $\mathcal{P} \neq \mathcal{NP}$. El incremento lineal de las puertas OR no es suficiente para conseguir un circuito de tamaño polinómico y, a partir de cierto punto, se necesitan muchas puertas AND para aumentar muy poco la puntuación. En definitiva, las observaciones realizadas son compatibles con que computar $Clique_{n,k}$ necesite un circuito de tamaño superpolinómico. Existen diferentes maneras de ahondar más en estas ideas, entre las que destacamos las siguientes.

Un estudio más detallado sobre qué funciones hacen falta para llegar a f_{clique} con μ_s podría proporcionar nuevos resultados. El carácter semántico de μ_s permite generar funciones con una salida deseada de manera cómoda. Por tanto, se podría intentar caracterizar y analizar qué funciones hacen falta para aumentar la puntuación cerca del final de un circuito.

La simulación realizada en la sección 4.4.3 está basada en el concepto de algoritmo genético. Utilizando otras técnicas que aproximen mejor el problema resultarían en una simulación más fiel y consistente. Una opción, por ejemplo, sería

entrenar una red neuronal que construya un circuito óptimo con respecto a μ_x y comprobar si se sigue observando esta tendencia a dejar de crecer al final del circuito.

A nivel teórico, tomando como referencia el teorema de Razoborov explicado en la sección 3.2, se podría intentar repetir los resultados con μ_x. Este teorema muestra el potencial que tienen las métricas sobre $Clique_{n,k}$. Teniendo en cuenta que los experimentos indican que μ_x está acotada al final del circuito, parece razonable intentar hacer un desarrollo teórico centrado en ella. Sobre todo con las funciones endogámicas, pues parecen ser las que mejor funcionan al inicio del circuito.

Por último, se podría hacer un análisis parecido sobre métricas diferentes o incluso mezclarlas entre ellas. Estudiar e intentar acotar nuevas métricas (ya sea para clique o para otros problemas) podría revelar conclusiones y resultados diferentes a nivel tanto teórico como práctico.

Conclusions and Future Work

In this text we have studied how circuits that compute $Clique_{n,k}$ behave. We have discussed in detail Razborov's result on monotone circuits and experiments on two metrics, one semantic and the other syntactic, have been done.

Experiments realized on μ_s, the semantic metric, give us evidence that, as we approach the end of a circuit, we need more and more specific functions to move forward. However, μ_s does not provide information about what the functions inside a circuit that compute $Clique_{n,k}$ look like.

On the other hand, experiments performed on μ_x, the syntactic metric, give us more concrete results. Under μ_x, OR gates produce a linear increase on the score. Small score functions are able to increase more when they pass through an AND gate. Nevertheless, when they reach scores close to f_{clique}, the increase become smaller and smaller, describing a logarithmic curve.

These results provide new evidence of $\mathcal{P} \neq \mathcal{NP}$. The linear increment of OR gates is not enough to reach a polynomial-sized circuit and, beyond a certain point, lots of AND gates are needed to increase the score only slightly. Definitely, the observations made are consistent with the fact that computing $Clique_{n,k}$ requires superpolynomial-sized circuits. There are different ways to dig deeper into these ideas, among we highlight the following.

A more detailed study of what functions are needed to reach f_{clique} with μ_s could provide new results. The semantic character of μ_s allows to generate functions with a desired output easily. Therefore, one could try to characterize and analyze what functions are needed to increase the score near the end of a circuit.

The simulation perfomed in section 4.4.3 is based on the genetic algorithm concept. Relying on other techniques that better approximate the problem would result in a more faithful and consistent simulation. One option, for example, would be to train a neural network that builds an optimal circuit relative to μ_x and check if this tendency to stop increasing score at the end of a circuit is still observed.

At a theoretical level, taking Razborov's theorem explained in section 3.2 as reference, one could try to repeat the results with μ_x. This theorem shows the potencial that metrics have on $Clique_{n,k}$. Considering that experiments indicate that μ_x is bounded at the end of a circuit, it seems reasonable indeed to try to make

a theoretical elaboration centered on it. Especially with endogamic functions, as they seem to be the ones that work best at the beginning of the circuit.

Finally, a similar analysis could be done on different metrics or even mixing them together. Studying and trying to bound new metrics (either for the clique problem or for other problems) could reveal different conclusions and results on both practical and theoretical levels.

Bibliografía

[1] Aaronson, S. *The Complexity Zoo*. Disponible en `https://complexityzoo.net/Complexity_Zoo`. Consultada el 27 de mayo de 2024.

[2] Arora, S. y Barak, B. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2006. ISBN 978-0-521-42426-4.

[3] Bazaraa, M., Jarvis, J. y Sherali, H. *Linear Programming and Network Flows*. Wiley, 2009. ISBN 9780470462720.

[4] Buss, S. y Aisenberg, J. *The Razborov monotone circuit lower bound*. Disponible en `https://mathweb.ucsd.edu/~sbuss/CourseWeb/Math262A_2013F`. Notas del curso *Combinatorics: Circuit Complexity*. Universidad de California en San Diego, 2013.

[5] Carro Garrido, E. y Cobían Fernández, J. R. *Analysis of Boolean functions using equanimity and entanglement*. Trabajo de fin de grado, Universidad Complutense, Madrid, 2023.

[6] Cook, S. A. The Complexity of Theorem-Proving Procedures. En *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA* (editado por M. A. Harrison, R. B. Banerji y J. D. Ullman), páginas 151–158. ACM, 1971.

[7] Dijkstra, E. W. *A note on two problems in connexion with graphs*. Numerische mathematik, vol. 1(1), páginas 269–271, 1959.

[8] Dunne, P. E. *On monotone simulations of nonmonotone networks*. Theoretical Computer Science, vol. 66(1), páginas 1–24, 1989. ISSN 0304-3975.

[9] Garey, M. R. y Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edición, 1979. ISBN 0716710455.

[10] Kuhn, H. W. *The Hungarian Method for the Assignment Problem*. Naval Research Logistics Quarterly, vol. 2(1–2), páginas 83–97, 1955.

[11] Razborov, A. A. *Lower bounds on the monotone complexity of some boolean functions*. Dokl. Akad. Nauk SSSR, vol. 281, páginas 798–801, 1985.

[12] Romo González, J. J. *Repositorio de este texto*. Disponible en `https://github.com/JaimeRG01/TFG`. 2024.

[13] Savage, J. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998. ISBN 9780201895391.

[14] Shannon, C. E. *The synthesis of two-terminal switching circuits*. The Bell System Technical Journal, vol. 28(1), páginas 59–98, 1949.