ALGORITMOS PARA DECIDIR DEFINIBILIDAD EN FRAGMENTOS DE LÓGICA DE PRIMER ORDEN

POR

PABLO VENTURA

PRESENTADO ANTE LA FACULTAD DE MATEMÁTICA, ASTRONOMÍA, FÍSICA Y COMPUTACIÓN COMO PARTE DE LOS REQUERIMIENTOS PARA LA OBTENCIÓN DEL GRADO DE

DOCTOR EN CIENCIAS DE LA COMPUTACIÓN

DE LA

UNIVERSIDAD NACIONAL DE CÓRDOBA

Junio de 2023

DIRECTOR: DR. MIGUEL CAMPERCHOLI

TRIBUNAL ESPECIAL:
DR. JOSÉ PATRICIO DIAZ VARELA (DEPTO. DE MATEMÁTICA - UNS)
DR. MIGUEL MARÍA PAGANO (FAMAF - UNC)
DR. RAÚL ALBERTO FERVARI (FAMAF - UNC)
DR. DIEGO NICOLÁS CASTAÑO (DEPTO. DE MATEMÁTICA - UNS)
DRA. MARIANA VANESA BADANO (FAMAF - UNC)

A Inés y a Leonardo

RESUMEN

Dada una estructura $\boldsymbol{A}$, decimos que un conjunto T es definible en primer orden sin cuantificadores si y solo si existe una fórmula sin cuantificadores φ tal que $\boldsymbol{A} \models \varphi(x)$ para todo $x \in T$, mientras que $\boldsymbol{A} \not\models \varphi(x)$ para todo $x \in A \setminus T$.

Esta tesis doctoral aborda la definibilidad de primer orden sin cuantificadores desde un enfoque computacional. Se han desarrollado algoritmos eficientes basados en propiedades semánticas de los modelos y se han implementado para su uso como asistentes de investigación. También se presenta un manual de uso para las herramientas y un conjunto de pruebas para comprobar su eficiencia empíricamente. Finalmente, se ha estudiado la complejidad de estos problemas de definibilidad y se han presentado resultados teóricos que establecen los límites de su eficiencia computacional.

CLASIFICACIÓN: F.4.1 - Model theory.

PALABRAS CLAVE: Teoría de modelos finitos, definibilidad, relaciones, algoritmos, morfismos, fragmentos de primer orden.

ABSTRACT

Given a structure $\boldsymbol{A}$, we say that a set T is definable in first-order logic without quantifiers if and only if there exists a first-order formula φ without quantifiers such that $\boldsymbol{A} \models \varphi(x)$ for all $x \in T$, while $\boldsymbol{A} \not\models \varphi(x)$ for all $x \in A \backslash T$.

This doctoral thesis addresses first-order definability without quantifiers from a computational approach. Efficient algorithms based on semantic properties of models have been developed and implemented for use as research assistants. A usage manual for the tools and a set of tests to empirically verify their efficiency are also presented. Finally, the complexity of these definability problems has been studied and theoretical results establishing the limits of their computational efficiency have been presented.

CLASSIFICATION: F.4.1 - Model theory.

KEYWORDS: Finite model theory, definability, relations, algorithms, morphisms, first-order fragments.

PUBLICACIONES

Esta tesis está basada en las siguientes publicaciones:

[1] Carlos Areces, Miguel Campercholi, Daniel Penazzi y Pablo Ventura. «The complexity of definability by open first-order formulas». En: *Logic Journal of the IGPL* 28.6 (mayo de 2020), págs. 1093-1105. DOI: 10.1093/jigpal/jzaa008. URL: https://doi.org/10.1093/jigpal/jzaa008.

[2] Carlos Areces, Miguel Campercholi y Pablo Ventura. «Deciding Open Definability via Subisomorphisms». En: *Logic, Language, Information, and Computation*. Ed. por Lawrence S. Moss, Ruy de Queiroz y Maricarmen Martinez. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, págs. 91-105.

[3] Miguel Campercholi, Mauricio Tellechea y Pablo Ventura. «Deciding Quantifier-free Definability in Finite Algebraic Structures». En: *Electronic Notes in Theoretical Computer Science* 348 (2020). 14th International Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2019), págs. 23-41. DOI: https://doi.org/10.1016/j.entcs.2020.02.003. URL: https://www.sciencedirect.com/science/article/pii/S1571066120300037.

[4] Miguel Campercholi, Mauricio Tellechea y Pablo Ventura. *Two algorithms to decide Quantifier-free Definability in Finite Algebraic Structures*. 2023. arXiv: 2303.17017 [cs.LO].

AGRADECIMIENTOS

En primer lugar quiero, agradecer a Camper, mi director desde hace tantos años. Muchas gracias por las charlas. Por la paciencia con mis sustos. Por el acompañamiento y por hacerme probar tantas comidas nuevas.

Muchas gracias a todo el grupo de Semántica Algebraica (o Lógica, si Diego no se ofende :-P) por las charlas y los aprendizajes de todo tipo.

Gracias, también, a Diego Castaño y a Patricio Díaz Varela por las ideas compartidas en nuestras idas a Bahía Blanca.

A Carlos Areces por las reuniones de trabajo y la escritura de papers acompañados con tazas de té matcha.

A Daniel Penazzi, por sus ideas siempre presentes.

A Mauricio Tellechea, mi hermano mayor de doctorado, muchas gracias por los consejos, las risas y los sándwiches de miga.

También gracias a mi familia, mis padres y mis hermanas, que me dieron la fuerza para seguir con la tesis en los peores momentos.

Finalmente, gracias a Inés y a Leonardo, por ser la motivación última de toda mi vida.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

Parte I

INTRODUCCIÓN

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

Al definir un objeto entre otros es fundamental la capacidad expresiva del lenguaje con el que contamos. Para ilustrar la idea, veamos el siguiente esquema con figuras geométricas. Cada figura tiene una forma y una ubicación propia.

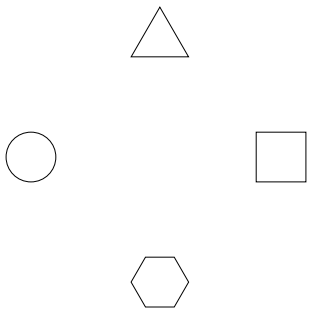


Figura 1.1: Figuras geométricas.

Supongamos que queremos definir al círculo. Para lograrlo necesitamos una propiedad que solo se cumpla en él. Un ejemplo trivial podría ser «Soy un círculo». Es claro que esta oración solo se cumple cuando ponemos como hablante al círculo. Esto sería haber *definido* al círculo en nuestro dibujo.

El problema se vuelve más interesante si restringimos un poco el lenguaje. Supongamos ahora que queremos definir al círculo, pero ya no podemos usar los nombres de las figuras geométricas. Tenemos la alternativa de definirlo como «Soy lo que está a la izquierda de todos los demás» y todavía podemos afirmar que, efectivamente, la oración es solo verdad cuando el círculo es quien la dice.

Ahora restrinjamos aun más el lenguaje, esta vez utilizando solo expresiones del tipo «está arriba o a la misma altura que» y «está abajo o a la misma altura que». Es fácil definir al triángulo como «Tengo a todos abajo o a la misma altura que yo» y también al hexágono como «Tengo a todos arriba o a la misma altura que yo».

Pero si intentamos definir al círculo, esta vez nuestro lenguaje ha perdido la capacidad de distinguir entre derecha e izquierda. Todo lo que podemos decir sobre él es que «Tengo alguien arriba o a la misma altura que yo, que no soy yo y tengo alguien abajo o a la misma altura que yo, que no soy yo». Finalmente, estamos ante un problema, ya que el cuadrado cumple esa misma propiedad. Ahora ya no podemos definir al círculo, ya que todo lo que podemos decir de él en nuestro nuevo lenguaje lo cumple también el cuadrado.

Esto se debe a que reflejar nuestro dibujo como en la figura siguiente mantiene el orden respecto de quién estaba abajo de quién (i.e preserva las propiedades expresables en nuestro lenguaje), pero invierte derecha e izquierda. Esto representa

un *automorfismo* de nuestro dibujo que, en particular, no *preserva* la ubicación del círculo.

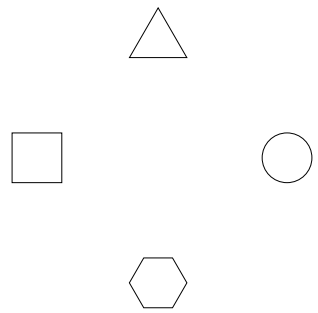


Figura 1.2: Figuras geométricas reflejadas.

Es interesante notar que, en cambio, para aquellas figuras geométricas sobre las que sí se preserva la ubicación, pudimos encontrar definiciones a pesar de lo restringido de nuestro lenguaje.

1.2 EL PROBLEMA

El problema sobre el que trabajamos no trata sobre decidir definibilidad en el lenguaje natural sino en el lenguaje formal llamado Lógica de Primer Orden, en particular sobre su fragmento abierto (i.e. fórmulas sin cuantificadores).

Sean $\mathcal{L}$ un lenguaje de primer orden y $\boldsymbol{A}$ una $\mathcal{L}$-estructura. Dado un conjunto n-ario $R \subseteq A^n$ diremos que la fórmula $\varphi(x_1, \ldots, x_n)$ *define* a R en $\boldsymbol{A}$ si

$$R = \{\overline{a} \in A^n | \boldsymbol{A} \vDash \varphi[\overline{a}]\}.$$

En este trabajo, estudiaremos el siguiente problema computacional.

Problema 1. Dados:

- $\mathcal{L}$ un lenguaje de primer orden finito,
- $\boldsymbol{A}$ una estructura finita de $\mathcal{L}$, y
- $R \subseteq A^n$ un conjunto,

decidir si hay una $\mathcal{L}$-fórmula abierta que define a R en $\boldsymbol{A}$.

En la Parte iii mostraremos que este problema resulta ser coNP-completo. En particular, una solución por «fuerza bruta» requiere la construcción de todas las relaciones definibles en $\boldsymbol{A}$ para poder dar una respuesta negativa. Nuestro enfoque es evitar este costo al explotar propiedades de preservación que pueda tener el conjunto de las fórmulas abiertas.

En el trabajo [3] se presentan condiciones semánticas equivalentes a la existencia de una $\mathcal{L}$-fórmula $\varphi(\bar{x})$ que define a R en $\boldsymbol{A}$, donde φ pertenece a un fragmento de primer orden específico, como las fórmulas abiertas, las abiertas positivas, las existenciales, etc.

En particular, para el caso de la definibilidad por una fórmula abierta se establece que (cf. Teorema 11) dado $\mathcal{L}$ un lenguaje de primer orden, $R \subseteq A^n$ un conjunto y $\mathbf{A}$ una $\mathcal{L}$-estructura finita, los siguientes son equivalentes:

1. Hay una $\mathcal{L}$-fórmula abierta que define R en $\mathbf{A}$.
2. Todo isomorfismo $\sigma : \mathbf{A}_0 \to \mathbf{B}_0$ con $\mathbf{A}_0$ y $\mathbf{B}_0$ $\mathcal{L}$-subestructuras de $\mathbf{A}$, preserva R (i.e. si $(a_1, \ldots, a_n) \in R$, entonces $(\sigma(a_1), \ldots, \sigma(a_n)) \in R$).

Al utilizar este resultado teórico, decidir definibilidad se convierte en un problema de búsqueda y chequeo de morfismos entre estructuras, potencialmente más tratable en su costo computacional.

1.2.1 *Ejemplo*

Consideremos en el siguiente reticulado $\mathbf{2} \times \mathbf{2}$ la relación unaria $P = \{u, u'\}$ representada en el siguiente diagrama. ¿Es P definible por una fórmula abierta en $\mathbf{2} \times \mathbf{2}$?

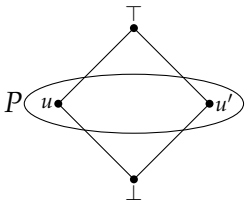


Figura 1.3: Reticulado $\mathbf{2} \times \mathbf{2}$ con una relación unaria P.

Notar que entre el subreticulado con universo $\{\top\}$ y el subreticulado con universo $\{u\}$ hay un isomorfismo σ que no preserva R, ya que $u \in R$ pero $\sigma(\top) = u \notin R$. A la luz del resultado mencionado anteriormente, esto nos permite asegurar que no existe una fórmula abierta (del lenguaje de los reticulados) que defina la relación P en $\mathbf{2} \times \mathbf{2}$.

1.3 ALGORITMO PARA ESTRUCTURAS RELACIONALES

En vista de lo expuesto, nuestro enfoque consiste en cambiar el costo computacional de construir las relaciones definibles sobre una clase de estructuras por el costo de computar una familia adecuada de morfismos y verificar que estos preservan la relación a definir. Sin dudas, el mayor peso computacional de este enfoque está en encontrar la familia de morfismos. Por esta razón, nuestra investigación se centra en estudiar el problema de cómo reducir el número de morfismos a construir. Los resultados obtenidos en esta dirección, como el Teorema 25, junto con el Corolario 26 y el Lema 27, se basan en encontrar un subconjunto *generador* de la familia de morfismos a calcular.

Otra característica importante del algoritmo desarrollado, en este caso en relación a su implementación, es que la construcción de isomorfismos se lleva a cabo como una instancia del *Constraint Satisfaction Problem* (CSP) (ver, e.g., [15, Capítulo 6]). Esto nos permite usufructuar la eficiencia alcanzada por los resolvedores de CSP.

1.4 ALGORITMOS PARA ÁLGEBRAS

Por otro lado, el Lema 30 y el Corolario 33 se basan en construir a demanda una representación del tipo de isomorfismo de una tupla en la estructura sin calcular explícitamente los isomorfismos involucrados. Utilizando esta representación del tipo de isomorfismo de una tupla, desarrollamos dos estrategias para resolver definibilidad.

La primera, utilizando el tipo de las tuplas para unirlas en clases de equivalencia por isomorfismos una vez que son calculadas; y, la segunda, refinando las clases de equivalencia cada vez que se calcula un paso más del tipo de las tuplas. En el caso de nuestra segunda estrategia basada en tipos de isomorfismos de tuplas nos permite, además, generar las fórmulas definidoras para los casos positivos.

1.5 ESTRUCTURA DEL TRABAJO

El trabajo se encuentra organizado de la siguiente forma:

En la Parte ii, se introducen algunos lemas fundamentales y definiciones para, luego, probar los teoremas de definibilidad de relaciones por fórmulas abiertas.

En la Parte iii, se hace un análisis de complejidad sobre la definibilidad abierta y también sobre el largo de las fórmulas.

En la Parte iv, se encuentran los algoritmos para chequeo de definibilidad y los resultados teóricos que fundamentan sobre el subconjunto de morfismos que genera por composición a todo el conjunto de morfismos a chequear. Luego de los algoritmos, mostramos cómo modelizar la búsqueda de homomorfismos mediante un CSP y, luego, discutimos el algoritmo que hace uso del isomorfismo de tuplas.

En la Parte v, presentamos nuestras implementaciones de los algoritmos documentando su instalación y su uso. Finalizamos esta parte con un análisis de los resultados empíricos de nuestras implementaciones.

En la Parte vi, se exponen nuestras conclusiones y las posibles continuaciones a este trabajo.

Parte II

PRELIMINARES

2

PRELIMINARES

En este capítulo ofrecemos algunas definiciones básicas y fijaremos notaciones. Si bien la intención ha sido la de escribir un trabajo autocontenido, suponemos que el lector está familiarizado con los conceptos básicos de la lógica de primer orden y su teoría de modelos elemental. Textos clásicos de referencia en estos temas son por ejemplo [9] y [14].

Luego presentamos resultados que caracterizan la definibilidad abierta. Estas caracterizaciones reducen la pregunta de si una relación es abierta definible a la pregunta de si dicha relación es preservada por una colección de morfismos que depende del fragmento de primer orden considerado. En particular, (cf. Teorema 11) dados $\mathcal{L}$ un lenguaje de primer orden, $R \subseteq A^n$ un conjunto y $\mathbf{A}$ una $\mathcal{L}$-estructura, entonces hay una fórmula abierta que define R en $\mathbf{A}$, si y solo si todo isomorfismo $\sigma : \mathbf{A}_0 \to \mathbf{B}_0$ donde $\mathbf{A}_0$ y $\mathbf{B}_0$ son subestructuras en $\mathcal{L} - \{R\}$ de $\mathbf{A}$, preserva R (i.e. si $(a_1, \ldots, a_n) \in R$, entonces $(\sigma(a_1), \ldots, \sigma(a_n)) \in R$).

Estos resultados, expuestos originalmente en [3], constituyen el punto de partida para nuestro estudio del chequeo computacional de la definibilidad.

2.1 DEFINICIONES PRELIMINARES

Para cada símbolo relacional o funcional s en un lenguaje τ, usaremos $\mathrm{ar}(s)$ para denotar la aridad de s. A continuación, supondremos todos los lenguajes como finitos. Supondremos que el lenguaje contiene variables de un conjunto infinito numerable $\mathsf{VAR} = \{x_1, x_2, \ldots, x_n, \ldots\}$. Dado un lenguaje τ de primer orden y $k \in \omega$ sea $\mathcal{T}_k$ el conjunto de τ-términos de profundidad k con variables en VAR. Escribiremos $\mathcal{T} := \bigcup_{k\in\omega} \mathcal{T}_k$ para el conjunto de todos los τ-términos sobre VAR.

Las *fórmulas atómicas* son ya sea de la forma $t = t'$, $R(\bar{t})$ o donde $t, t' \in \mathcal{T}$, $\bar{t}$ es una secuencia de términos en $\mathcal{T}$ de largo k y R es un símbolo de relación de aridad k. Las *fórmulas abiertas* son combinaciones booleanas de fórmulas atómicas (no tienen ocurrencias de $\forall$, $\exists$). A veces, escribiremos solo *fórmula* en lugar de *fórmula abierta*. Escribiremos $\varphi(v_1, \ldots, v_k)$ para una fórmula φ cuyas variables libres están incluidas en $\{v_1, \ldots, v_k\}$.

Sea τ un lenguaje, una *τ-estructura* (o modelo) es un par $\boldsymbol{A} = \langle A, \cdot^{\boldsymbol{A}} \rangle$ donde A es un conjunto no vacío (el dominio o el universo), y $\cdot^{\boldsymbol{A}}$ es una función de interpretación que asigna a cada símbolo de relación k-ario R en τ, un subconjunto $R^{\boldsymbol{A}}$ de A^k y a cada símbolo funcional k-ario f, una función $f^A : A^k \to A$. Si $\boldsymbol{A}$ es una estructura escribiremos A para su dominio y $\cdot^{\boldsymbol{A}}$ para su función de interpretación. Dada una fórmula $\varphi(v_1, \ldots, v_k)$, y una secuencia de elementos $\bar{a} = \langle a_1, \ldots, a_k \rangle \in A^k$ escribiremos $\boldsymbol{A} \models \varphi[\bar{a}]$ si φ se satisface en $\boldsymbol{A}$ mediante la asignación que mapea v_i a a_i.

A continuación, introducimos dos nociones centrales del presente trabajo: «definibilidad» y «preservación».

Definición 2. Sean $\mathcal{L}$ un lenguaje de primer orden y $\mathbf{A}$ una $\mathcal{L}$-estructura. Si $R \subseteq A^n$ es un conjunto, diremos que la fórmula $\varphi(x_1, \ldots, x_n)$ *define a* R *en* $\mathbf{A}$ si para todo $a_1, \ldots, a_n \in A$ vale que

$$(a_1, \ldots, a_n) \in R \iff \mathbf{A} \vDash \varphi(a_1, \ldots, a_n).$$

En particular, diremos que un subconjunto $T \subseteq A^k$ es *abierto definible* en $\boldsymbol{A}$ si hay una fórmula de primer orden $\varphi(x_1, \ldots, x_k)$ en el lenguaje de $\boldsymbol{A}$ tal que $T = \{\bar{a} \in A^k : \boldsymbol{A} \models \varphi[\bar{a}]\}$.

Definición 3. Sean $\mathcal{L}$ un lenguaje de primer orden, $R \in \mathcal{L}$ un símbolo de relación n-ario y $\mathbf{A}$, $\mathbf{B}$ dos $\mathcal{L}$-estructuras. Diremos que una función $f : A \to B$ *preserva* a R si para toda tupla $(a_1, \ldots, a_n) \in A$ tenemos que si $(a_1, \ldots, a_n) \in R^{\mathbf{A}}$ implica que $(f(a_1), \ldots, f(a_n)) \in R^{\mathbf{B}}$. Es decir, f es un homomorfismo de $\langle A, R^{\mathbf{A}} \rangle$ en $\langle B, R^{\mathbf{B}} \rangle$.

Dado un conjunto de fórmulas Δ, escribiremos $\Delta(\bar{x})$ para anunciar que cada una de las fórmulas en Δ tiene sus variables libres contenidas en la tupla $\bar{x}$. Si $\mathbf{A}$ es una estructura y $\bar{a}$ es una tupla de elementos de A, escribiremos $\mathbf{A} \vDash \Delta[\bar{a}]$ cuando $\mathbf{A} \vDash \delta[\bar{a}]$ para cada $\delta \in \Delta(\bar{x})$.

Dados $\mathcal{L} \subseteq \mathcal{L}'$ lenguajes de primer orden y $\mathbf{A}$ una $\mathcal{L}'$-estructura, usaremos $\mathbf{A}_{\mathcal{L}}$ para indicar el reducto de $\mathbf{A}$ al lenguaje $\mathcal{L}$. Si $\mathbf{A}, \mathbf{B}$ son $\mathcal{L}$-estructuras, escribimos $\mathbf{A} \leq \mathbf{B}$ para expresar que $\mathbf{A}$ es subestructura de $\mathbf{B}$. Dada una estructura $\mathbf{A}$ y $\bar{a} = a_1, \ldots, a_n \in A$ utilizaremos $\langle \bar{a} \rangle^{\mathbf{A}}$ para denotar la subestructura de $\mathbf{A}$ generada por $\bar{a}$.

Definición 4. Dos fórmulas $\alpha(\bar{x})$ y $\beta(\bar{x})$ se dicen *equivalentes* sobre una estructura $\mathbf{A}$, si para cada $\bar{a}$ de A vale que

$$\mathbf{A} \vDash \alpha[\bar{a}] \iff \mathbf{A} \vDash \beta[\bar{a}].$$

El siguiente teorema será utilizado para demostrar todas las caracterizaciones de definibilidad; es la herramienta que nos permitirá transformar un diagrama infinitario (i.e el conjunto infinito de fórmulas que se satisfacen para una tupla de elementos, que se definirá formalmente más adelante) en una fórmula.

Teorema 5. *Sea* $\mathbf{A}$ *una estructura finita. Hay un conjunto finito de fórmulas* $\Sigma(x_1, \ldots, x_n)$ *tal que para toda fórmula* $\varphi(\overline{x})$ *hay* $\sigma(\overline{x}) \in \Sigma(\overline{x})$ *tal que* $\varphi(\bar{x})$ *y* $\sigma(\overline{x})$ *son equivalentes sobre* $\mathbf{A}$.

Demostración. Para un fórmula $\varphi(\bar{x})$ sea

$$[\varphi(\bar{x})]^{\mathbf{A}} = \{\bar{a} \in A^n \mid \mathbf{A} \vDash \varphi[\bar{a}]\}.$$

Notar que $\varphi(\bar{x})$ es equivalente $\psi(\bar{x})$ en $\mathbf{A}$ sii $[\varphi(\bar{x})]^{\mathbf{A}} = [\psi(\bar{x})]^{\mathbf{A}}$. Luego, el número de clases de equivalencia es finito ya que está acotado por $|\mathcal{P}(A^n)|$. □

Recordamos que una función $\gamma : A \to B$ es un *embedding* de $\mathbf{A}$ en $\mathbf{B}$ si γ es un isomorfismo de $\mathbf{A}$ en $\mathbf{Im}(\gamma)$.

Lema 6 (Los embeddings preservan fórmulas abiertas)**.** *Sean* **A**,**B** *estructuras y* $\gamma : A \to B$ *una función. Son equivalentes:*

1. γ *es un embedding de* **A** *en* **B***.*
2. *Para toda fórmula abierta* $\varphi(\bar{x})$ *y para cada* $\bar{a}$ *de* A *vale que*

$$\mathbf{A} \vDash \varphi[\bar{a}] \Leftrightarrow \mathbf{B} \vDash \varphi[\gamma(\bar{a})].$$

Demostración. 1⇒2) Sea γ un embedding de **A** en **B**, sea $\varphi(x_1, ..., x_n)$ una fórmula abierta y $\bar{a} \in A^n$. Lo probaremos por inducción en las fórmulas. El caso base es directo ya que los homomorfismos preservan términos y relaciones. Veamos los casos inductivos:

Sea $\varphi(\bar{x}) = \neg\varphi_1(\bar{x})$:

$\mathbf{A} \vDash \neg\varphi_1[\bar{a}]$ sii $\mathbf{A} \nvDash \varphi_1[\bar{a}]$ sii $\mathbf{B} \nvDash \varphi_1[\gamma(\bar{a})]$ sii $\mathbf{B} \vDash \neg\varphi_1[\gamma(\bar{a})]$.

Sea $\varphi(\bar{x}) = (\varphi_1 \eta \varphi_2)(\bar{x})$:

$\mathbf{A} \vDash (\varphi_1 \eta \varphi_2)[\bar{a}]$ sii $\mathbf{A} \vDash \varphi_1[\bar{a}]$ "η" $\mathbf{A} \vDash \varphi_2[\bar{a}]$

sii $\mathbf{A} \vDash \varphi_1[\gamma(\bar{a})]$ "η" $\mathbf{B} \vDash \varphi_2[\gamma(\bar{a})]$ sii $\mathbf{B} \vDash (\varphi_1 \eta \varphi_2)[\gamma(\bar{a})]$.

2⇒1) Supongamos que para toda fórmula abierta $\varphi(\bar{x})$ y cada $\bar{a} \in A^m$ vale que:

$\mathbf{A} \vDash \varphi[\bar{a}] \Leftrightarrow \mathbf{B} \vDash \varphi[\gamma(\bar{a})]$.

Es de rutina ver que γ es homomorfismo.

- Veamos que γ es inyectiva:

$$\begin{aligned} &\gamma(a) = \gamma(a') \\ &\text{sii } \mathbf{B} \vDash (x_1 = x_2)[\gamma(a), \gamma(a')] \\ &\text{sii } \mathbf{A} \vDash (x_1 = x_2)[a, a'] \\ &\text{sii } a = a'. \end{aligned}$$

- Veamos que γ es embedding:

  Sea $r \in \mathcal{R}_n$

$$\begin{aligned} (\gamma(a_1), \ldots, \gamma(a_n)) \in r^{\mathbf{B}} &\\ \text{sii } \mathbf{B} \vDash r(x_1, \ldots, x_n)[\gamma(a_1), \ldots, \gamma(a_n)] &\\ \text{sii } \mathbf{A} \vDash r(x_1, \ldots, x_n)[a_1, \ldots, a_n] &\\ \text{sii } (a_1, \ldots, a_n) \in r^{\mathbf{A}}. & \end{aligned}$$

□

El siguiente lema muestra que las subestructuras preservan fórmulas abiertas.

Lema 7. *Si* $\mathbf{A}$ *es una subestructura de* $\mathbf{B}$ *y* $\varphi(\bar{x})$ *es una fórmula abierta, entonces para cada* $\bar{a} \in A^n$ *vale que*

$$\mathbf{A} \vDash \varphi[\bar{a}] \Leftrightarrow \mathbf{B} \vDash \varphi[\bar{a}].$$

Demostración. Notar que la inclusión $\iota : A \to B$, definida por $\iota(a) = a$, es un embedding de $\mathbf{A}$ en $\mathbf{B}$. Luego, una aplicación directa del Lema 6 produce el resultado deseado. □

Definimos, a continuación, el *diagrama abierto* que será utilizado en la construcción de fórmulas abiertas que definen relaciones.

Definición 8. Sea $\mathbf{A}$ una estructura y sean $a_1, \ldots, a_n \in A$. Definimos el *diagrama abierto* para $a_1, \ldots, a_n$ en $\mathbf{A}$ como:

$$\Delta_{\mathbf{A},\bar{a}}(x_1, \ldots, x_n) := \{\alpha(\bar{x}) \mid \alpha \text{ una } \mathcal{L}\text{-fórmula abierta y } \mathbf{A} \vDash \alpha[\bar{a}]\}.$$

El siguiente lema establece una relación muy interesante entre elementos de dos estructuras al mostrar que si dos secuencias de elementos tienen las mismas propiedades abiertas (i.e., el mismo diagrama abierto) significa que generan subestructuras isomorfas. Usaremos $\langle \bar{a} \rangle^{\mathbf{A}}$ para denotar el subuniverso de $\boldsymbol{A}$ generado con los elementos de la tupla $\bar{a}$.

Lema 9. *Sea* $\mathbf{A}$ *una estructura y* $b_1, \ldots, b_n \in B$*, son equivalentes:*

1. $\mathbf{B} \vDash \Delta_{\mathbf{A},\bar{a}}[\bar{b}]$,
2. Hay un isomorfismo γ de $\langle \bar{a} \rangle^{\mathbf{A}}$ en $\langle \bar{b} \rangle^{\mathbf{B}}$ tal que $\gamma(\bar{a}) = \bar{b}$.

Demostración. 1⇒2) Supongamos que $\mathbf{B} \vDash \Delta_{\mathbf{A},\bar{a}}[\bar{b}]$. Es decir que si $\alpha(\bar{x})$ es una fórmula abierta y $\mathbf{A} \vDash \alpha[\bar{a}]$, luego $\mathbf{B} \vDash \alpha[\bar{b}]$. Notar que

$$\gamma : \langle \bar{a} \rangle^{\mathbf{A}} \to \langle \bar{b} \rangle^{\mathbf{A}}$$
$$\gamma(t^{\mathbf{A}}[a_1, \ldots, a_n]) = t^{\mathbf{B}}[b_1, \ldots, b_n]$$

es un función bien definida que cumple $\gamma(\bar{a}) = \bar{b}$. Además, por hipótesis, preserva fórmulas abiertas. Así, por el Lema 6, sabemos que γ es un embedding. Por último, es evidente a partir de la definición que γ es sobreyectiva.
2⇒1) Consecuencia directa del Lema 6. □

En el lema siguiente se determina que los homomorfismos preservan fórmulas atómicas y también fórmulas abiertas positivas.

2.2 DEFINIBILIDAD POR FÓRMULAS ABIERTAS

Sea $f : \operatorname{dom} f \subseteq A \to A$ una función parcial. Dado $T \subseteq \operatorname{dom} f$, la restricción de f al conjunto T es la función $f|_T : T \to A$ donde $f|_T(a) = f(a)$ para todo $a \in T$. La función f es un *subisomorfismo* (subiso para acortar) de $\boldsymbol{A}$ siempre que f sea inyectiva, y tanto f como f^{-1} preserven $R^{\boldsymbol{A}}$ para cada relación $R \in \tau$ y γ los gráficos de $g^{\mathbf{A}}$ para cada función $g \in \tau$. (Notar que si $\boldsymbol{A}$ es puramente relacional, entonces un subiso de $\boldsymbol{A}$ es exactamente un isomorfismo entre subestructuras de $\boldsymbol{A}$). Denotaremos el conjunto de todos los subisomorfismos de $\boldsymbol{A}$ con subIso $\boldsymbol{A}$.

Presentamos a continuación el teorema de definibilidad abierta. Antes necesitaremos el siguiente resultado.

Lema 10. *Sean* $\mathbf{A}, \mathbf{B}$ *estructuras finita y* $a_1, \ldots, a_n \in A$. *Existe una fórmula abierta* $\varphi(\bar{x})$ *tal que para todo* $\bar{b} \in B^n$ *son equivalentes:*

1. $\mathbf{B} \vDash \varphi[\bar{b}]$.
2. *Hay un isomorfismo* γ *de* $\langle \bar{a} \rangle^{\mathbf{A}}$ *en* $\langle \bar{b} \rangle^{\mathbf{B}}$ *tal que* $\gamma(\bar{a}) = \bar{b}$.

Demostración. Por el Teorema 5 hay una fórmula $\varphi(\bar{x})$ equivalente sobre $\{\mathbf{A}, \mathbf{B}\}$ al diagrama abierto $\Delta_{\mathbf{A},\bar{a}}(\bar{x})$. Luego, por el Lema 9, es inmediata la equivalencia del enunciado. □

Estamos listos para demostrar el resultado de definibilidad.

Teorema 11 (cf. [3, Thm. 3.1])**.** *Sean* $\mathcal{L} \subseteq \mathcal{L}'$ *lenguajes de primer orden, sea* $R \in \mathcal{L}' - \mathcal{L}$ *un símbolo de relación n-ario. Para una* $\mathcal{L}'$*-estructura finita* $\mathbf{A}$, *los siguientes son equivalentes:*

1. *Hay una* $\mathcal{L}$*-fórmula abierta que define a* R *en* $\mathbf{A}$.
2. *Para todas* $\mathbf{A}_0 \leq \mathbf{A}_{\mathcal{L}}$ *y* $\mathbf{A}'_0 \leq \mathbf{A}_{\mathcal{L}}$ *se tiene que todo isomorfismo* $\sigma : \mathbf{A}_0 \to \mathbf{A}'_0$ *preserva* R.

Demostración. 1⇒2) Sea $\varphi(\bar{x})$ la fórmula que define R en $\mathbf{A}$. Sean $\mathbf{A}_0 \leq \mathbf{A}_{\mathcal{L}}$, $\mathbf{A}'_0 \leq \mathbf{A}_{\mathcal{L}}$ y $\sigma : \mathbf{A}_0 \to \mathbf{A}'_0$ un isomorfismo. Fijamos $\bar{a} \in R^{\mathbf{A}_0}$; veremos que $\sigma(\bar{a}) \in R^{\mathbf{A}'_0}$. Como $\bar{a} \in R^{\mathbf{A}_0} \Rightarrow \bar{a} \in R^{\mathbf{A}} \Leftrightarrow \mathbf{A} \vDash \varphi[\bar{a}]$, por Lema 7, $\mathbf{A}_0 \vDash \varphi[\bar{a}]$. Además, como $\mathbf{A}_0$ y $\mathbf{A}'_0$ son isomorfos por σ, tenemos que $\mathbf{A}'_0 \vDash \varphi[\sigma(\bar{a})]$, luego $\sigma(\bar{a}) \in R^{\mathbf{A}'_0}$.

2⇒1) Por el Teorema 5, dada $\mathbf{A}$, para cada $\bar{a} \in R^{\mathbf{A}}$ hay una fórmula $\delta_{\mathbf{A},\bar{a}}(\bar{x})$ equivalente sobre $\mathbf{A}$ al diagrama abierto $\Delta_{\mathbf{A},\bar{a}}(\bar{x})$. Definimos

$$\varphi(\bar{x}) = \bigvee_{\bar{a} \in R^{\mathbf{A}}} \delta_{\mathbf{A},\bar{a}}(\bar{x}).$$

Probamos ahora que $\varphi(\bar{x})$ define a R en $\mathbf{A}$. Fijamos $\mathbf{A}_0$ y $\bar{a}_0 \in A^n$. Supongamos primero que $\mathbf{A_0} \vDash \varphi[\bar{a}_0]$. Entonces hay $\bar{a} \in R^{\mathbf{A}}$ tales que

$$\mathbf{A}_0 \vDash \delta_{\mathbf{A},\bar{a}}[\bar{a}_0].$$

Por Lema 10, hay un isomorfismo $\gamma : \langle \bar{a} \rangle^{\mathbf{A}} \to \langle \bar{a_0} \rangle^{\mathbf{A_0}}$ tal que $\gamma(\bar{a}) = \bar{a_0}$. Ya que por hipótesis γ preserva R y $\bar{a} \in R^{\mathbf{A}}$, se sigue que $\bar{a_0} = \gamma(\bar{a}) \in R^{\mathbf{A_0}}$. Para probar la implicación restante, supongamos que $\bar{a_0} \in R^{\mathbf{A_0}}$. Como $\mathbf{A_0} \models \delta_{\mathbf{A}_0, \bar{a_0}}[\bar{a_0}]$, es evidente que $\mathbf{A_0} \models \varphi[\bar{a_0}]$. □

Un lenguaje es *algebraico* cuando no tiene símbolos de relación. Un *álgebra* es una estructura de un lenguaje algebraico. En este contexto, las únicas *fórmulas atómicas* son de la forma $t = t'$ donde t y t' son términos.

Sea $\mathbf{A}$ un álgebra. Un subconjunto $S \subseteq A$ es un *subuniverso* de $\mathbf{A}$ si S no es vacío y es cerrado bajo las operaciones fundamentales de $\mathbf{A}$. Para una tupla $\bar{a} = (a_1, \ldots, a_k) \in A^k$, escribiremos $\mathrm{Sg}(\bar{a})$ para denotar $\langle \bar{a} \rangle^{\mathbf{A}}$ (el subuniverso de $\mathbf{A}$ generado por $\{a_1, \ldots, a_k\}$) con el objetivo de evitar excesiva notación cuando por contexto se entienda el álgebra a la que nos referimos.

2.3 PROBLEMAS COMPUTACIONALES

En este trabajo estudiaremos los siguientes problemas de decisión computacional:

OpenDef

Instancia:	Una estructura relacional finita $\boldsymbol{A}$ y una relación T sobre el dominio de $\boldsymbol{A}$.
Pregunta:	¿Es T abierta definible en $\boldsymbol{A}$?

OpenDefAlg

Instancia:	Una estructura algebraica finita $\boldsymbol{A}$ y una relación T sobre el dominio de $\boldsymbol{A}$.
Pregunta:	¿Es T abierta definible en $\boldsymbol{A}$?

Llamaremos a las relaciones $R^{\boldsymbol{A}}$ con $R \in \boldsymbol{\mathcal{L}}$ *relaciones base*, mientras que T será la *relación target*. Usaremos $\mathrm{subIso}\,\boldsymbol{A}$ para denotar el conjunto de todos los subisomorfismos de $\boldsymbol{A}$.

La siguiente caracterización de definibilidad abierta es clave para nuestro estudio.

Teorema 12. *[3]Sea $\boldsymbol{A}$ una estructura relacional finita y $T \subseteq A^m$. Los siguientes son equivalentes:*

1. *T es abierta definible en $\boldsymbol{A}$.*
2. *T es preservada por todos los subisomorfismos γ de $\boldsymbol{A}$.*
3. *T es preservada por todos los subisomorfismos γ de $\boldsymbol{A}$ con $|\mathrm{dom}\,\gamma| \leq m$.*

Demostración. La equivalencia de (1) y (2) es probada en [3, Thm 3.1]. Claramente (2) implica (3), por lo que mostraremos que (3) implica (2). Sea γ un subiso de $\mathbf{A}$ y sea $\langle a_1, \ldots, a_m \rangle \in T \cap (\mathrm{dom}\,\gamma)^m$. Notar que la restricción $\gamma|_{\{a_1,\ldots,a_m\}}$ es un subiso de $\mathbf{A}$. Por lo tanto, $\gamma(\bar{a}) = \gamma|_{\{a_1,\ldots,a_m\}}(\bar{a}) \in T$. □

Parte III

COMPLEJIDAD

3

COMPLEJIDAD

3.1 PRELIMINARES

En lo que sigue, todos los lenguajes se suponen como finitos y puramente relacionales. En esta sección, estudiaremos la complejidad del el problema de decisión computacional OpenDef, presentado anteriormente. Esta parte se basa en [1].

3.1.1 *Codificaciones y tamaños*

Como es habitual cuando se consideran cuestiones de complejidad, el tamaño de un objeto es la longitud de la cadena sobre un alfabeto finito que codifica el objeto. En particular, los números naturales están codificados en binario y suponemos codificaciones fijas para lenguajes, relaciones, estructuras y fórmulas. Definimos el tamaño de estos objetos de acuerdo con estas codificaciones. Para un conjunto S, denotaremos $|S|$ para el número de elementos en S y, para un lenguaje relacional τ, usaremos $|\tau|$ para el número de símbolos relacionales en τ.

Escribiremos $\text{size}(ob)$ para denotar el tamaño de un objeto ob. Aunque no especificamos las codificaciones, supongamos las siguientes igualdades a lo largo de nuestro análisis. Sea τ un lenguaje relacional, $\mathbf{A}$ una τ-estructura, $T \subseteq A^m$ y φ una fórmula de primer orden.

- $\text{size}(\tau) = (|\tau| + \sum_{R\in\tau} \text{ar}(R)) \log |\tau|$,[1]
- $\text{size}(\mathbf{A}) = \text{size}(\tau) + (|A| + \sum_{R\in\tau} \text{ar}(R) \left|R^{\mathbf{A}}\right|) \log |A|$,
- $\text{size}(\mathbf{A}, T) = \text{size}(\mathbf{A}) + m\,|T| \log |A|$,
- $\text{size}_\tau(\varphi) = \mathit{relcount}(\varphi) \log |\tau| + \mathit{varcount}(\varphi) \log(\mathit{var}^{\#}(\varphi))$.

Aquí $\mathit{relcount}(\varphi)$ [$\mathit{varcount}(\varphi)$] devuelve el número de ocurrencias de símbolos de relación [variables] en φ, y $\mathit{var}^{\#}(\varphi)$ es el número de variables distintas que ocurren en φ. Como una fórmula φ es una τ-fórmula para cualquier τ que contenga los símbolos de relación de φ, la codificación de φ (y por lo tanto su tamaño) dependen de cuál lenguaje estamos pensando para φ. Otra suposición que hacemos sobre la codificación es que determinar si $\bar{a} \in R^{\mathbf{A}}$ puede ser computado en tiempo $O(\text{size}(\mathbf{A}))$.

1 Cuando una expresión que use $\log x$ no tenga sentido, debe leerse como $\lfloor \log x \rfloor$.

3.2 COMPLEJIDAD CLÁSICA DE OpenDef

En lo que sigue, un grafo es un modelo $\boldsymbol{G}$ de un lenguaje $\tau_{\text{GRAPH}} = \{E\}$, con E binaria, tal que $E^{\boldsymbol{G}}$ es simétrica e irreflexiva. Daremos una reducción desde el siguiente problema para demostrar nuestro resultado de hardness.

InducedPath	
Instancia:	Un grafo finito $\boldsymbol{G}$ y un entero positivo k.
Pregunta:	¿Hay un camino de largo k en $\boldsymbol{G}$ como un grafo inducido (i.e., como submodelo)?

InducedPath es conocido por ser NP-completo (ver, e.g., [12]).

Teorema 13. OpenDef *es* coNP*-completo.*

Demostración. Primero probaremos hardness. Fijemos un grafo de entrada $\boldsymbol{G}$ y un entero positivo k. Podemos suponer que G es disjunto con el conjunto de enteros. Sea $\boldsymbol{G}'$ el grafo con universo $G' := G \cup \{1, \dots, k\}$ y con

$$E^{\boldsymbol{G}'} := E^{\boldsymbol{G}} \cup \{\langle a, b\rangle \in \{1, \dots, k\}^2 : |a - b| = 1\}.$$

O sea, $\boldsymbol{G}'$ es la unión disjunta de $\boldsymbol{G}$ y un camino de largo k. Definimos

$$T := \{\langle 1, \dots, k\rangle, \langle k, \dots, 1\rangle\}.$$

Ahora, observar que por el Teorema 12 tenemos que OpenDef devuelve FALSE con entrada $(\boldsymbol{G}', T)$ si y solo si InducedPath devuelve TRUE para la entrada $(\boldsymbol{G}, k)$.

Mostrar que OpenDef está en coNP es una aplicación directa del Teorema 12. Dada una estructura relacional finita $\boldsymbol{A}$ y $T \subseteq A^k$, el hecho de que OpenDef devuelva FALSE para la entrada $(\boldsymbol{A}, T)$ es atestiguado por una biyección γ entre subconjuntos de A satisfaciendo condiciones fácilmente comprobables en tiempo polinomial con respecto al tamaño de $(\boldsymbol{A}, T)$. □

Dado un lenguaje relacional τ sea OpenDef$[\tau]$ la restricción de OpenDef a estructuras de entrada del lenguaje τ. Teniendo en cuenta la prueba del Teorema 13 tenemos lo siguiente.

Corolario 14. OpenDef$[\tau_{\text{GRAPH}}]$ *es* coNP*-completo.*

3.3 COMPLEJIDAD PARAMETRIZADA DE OPENDEF

La *complejidad parametrizada* es un marco matemático que permite hacer un análisis más fino del costo computacional de un problema que la complejidad clásica. En la parametrización de un problema (clásico), elegimos una parte específica de la entrada del problema y tratamos de entender cómo afecta esa parte al costo computacional. Por ejemplo, una parametrización del problema SAT proposicional

(i.e., el problema de satisfacibilidad para la lógica proposicional) podría ser el número de variables en la fórmula de entrada.

Comenzamos con las definiciones básicas que nos incumben. Hay ligeras discrepancias para estas definiciones en la literatura, pero nuestros resultados siguen siendo válidos independientemente de cuál versión se utilice. Usamos las definiciones de [11]. Como es habitual, los problemas de decisión (clásicos) se formalizan como lenguajes sobre alfabetos finitos no vacíos. Sea $\Sigma \neq \emptyset$ un alfabeto finito.

- Una *parametrización* de Σ^* es un mapeo $\kappa : \Sigma^* \to \mathbb{N}$ que es computable en tiempo polinomial.
- Un *problema parametrizado (o paramétrico)* (sobre Σ) es un par (Q, κ) consistente de un conjunto $Q \subseteq \Sigma^*$ de cadenas de Σ y una parametrización κ de Σ^*.

Consideraremos la siguiente parametrización[2] de OpenDef:

p-OpenDef	
Instancia:	Una estructura relacional finita $\boldsymbol{A}$ y $T \subseteq A^m$.
Parámetro:	$m\,\lvert T\rvert$.
Pregunta:	¿Es T abierta definible en $\boldsymbol{A}$?

Para un entero positivo k, la *fracción k-ésima* de un problema parametrizado (Q, κ) es la restricción del problema a todas las instancias $x \in \Sigma^*$ tales que $\kappa(x) = k$. Usaremos p-OpenDef$_k$ para denotar la fracción k-ésima de p-OpenDef.

Proposición 15. *p-OpenDef$_k$ es computable en tiempo $k!n^{2k}p(n)$ donde n es el tamaño de la entrada y $p(X)$ es un polinomio.*

Demostración. Fijemos una estructura relacional finita $\boldsymbol{A}$ y $T \subseteq A^m$. Sea n el tamaño de $(\boldsymbol{A}, T)$ y $k = m\,|T|$. Dado un entero positivo l, definimos

$$\mathrm{subIso}_l \boldsymbol{A} := \{\gamma \in \mathrm{subIso}\,\boldsymbol{A} : |\mathrm{dom}\,\gamma| = l\}.$$

Notar que el Corolario 12 implica que T es abierta definible en $\boldsymbol{A}$ si y solo si cada $\gamma \in (\bigcup_{l \leq m} \mathrm{subIso}_l \boldsymbol{A})$ preserva T. Mostramos que esta equivalencia hacia la derecha puede ser comprobada en tiempo polinomial. Sea

$$\mathcal{I}_l := \{\gamma : \text{hay } B, B' \subseteq A \text{ tales que } |B| = l \text{ y } \gamma : B \to B' \text{ es biyectiva}\}.$$

Observar que

$$|\mathcal{I}_l| = l! \cdot \binom{|A|}{l}^2,$$

2 Este problema parametrizado (y otros que aparecen después) es presentado de una manera informal, pero debería quedar claro cómo ajustarlos en la forma de una definición formal.

ya que hay $l!$ biyecciones entre cualesquiera dos subconjuntos de tamaño l de A , y así

$$|\mathcal{I}_l| \leq l! \cdot |A|^{2l} \leq k! \cdot n^{2k}.$$

Ahora, para cada $\gamma \in \mathcal{I}_l$ tenemos que revisar:

1. si $\gamma \in \text{subIso}\, \boldsymbol{A}$,
2. y, en ese caso, si γ preserva T.

No es difícil ver que hay un polinomio $p(X)$, tal que para cada $l \leq m$ y cada $\gamma \in \mathcal{I}_l$ ambas comprobaciones pueden llevarse a cabo en $p(n)$ pasos. Luego, revisar si cada miembro de $\text{subIso}_l\, \boldsymbol{A}$ preserva T toma como máximo $|\mathcal{I}_l| \cdot p(n)$ pasos, y la computación para p-OpenDef_k con entrada $(\boldsymbol{A}, T)$ puede ser realizada en como máximo

$$k! n^{2k} p(n)$$

pasos. □

Un problema parametrizado (Q, κ) sobre el alfabeto Σ es *tratable con parámetro fijo* (FPT) si hay un algoritmo $\mathcal{A}$ junto con un polinomio $p(X)$ y una función computable $f : \mathbb{N} \to \mathbb{N}$ tal que $\mathcal{A}$ decide si $x \in Q$ en tiempo $f(\kappa(x))p(|x|)$ para todo $x \in \Sigma^*$. La clase FPT de todos los problemas tratables con parámetro fijo juega el rol que P juega en la complejidad clásica.

Aunque cada fracción de p-OpenDef pueda ser computada en tiempo polinomial, la cota dada por la Proposición 15 no implica que p-OpenDef $\in$ FPT, ya que el parámetro aparece como un exponente del tamaño del input. De hecho, como veremos a continuación es poco probable que p-OpenDef esté en FPT, ya que es hard para la clase coW[1], una clase de problemas parametrizados que se cree que es estrictamente más grande que FPT. Pero, antes de poder discutir hardness en problemas parametrizados, necesitamos una noción adecuada de reducción.

Definición 16. Sean (Q, κ) y (Q', κ') problemas parametrizados sobre los alfabetos Σ y Σ', respectivamente. Una *fpt-reducción* de (Q, κ) a (Q', κ') es un mapeo $R : \Sigma^* \to (\Sigma')^*$ tal que:

1. Para todo $x \in \Sigma^*$ tenemos que $(x \in Q \Leftrightarrow R(x) \in Q')$.
2. Hay una función computable f y un polinomio $p(X)$ tal que $R(x)$ es computable en tiempo $f(\kappa(x)) \cdot p(x)$.
3. Hay una función computable $g : \mathbb{N} \to \mathbb{N}$ tal que $\kappa'(R(x)) \leq g(\kappa(x))$ para todo $x \in \Sigma^*$.

Si P y P' son problemas parametrizados, escribiremos $P \leq^{\text{fpt}} P'$ si hay una fpt-reducción de P a P'. Observar que $\leq^{\text{fpt}}$ es una relación transitiva. Escribiremos $P \equiv^{\text{fpt}} P'$ si hay fpt-reducciones en ambas direcciones.

Hay otras nociones de fpt-reducciones, como las fpt-reducciones de Turing [11], que incluyen oráculos. Todas las fpt-reducciones que aquí presentamos satisfacen la Definición 16, por lo que utilizaremos el nombre fpt-reducciones para ellas.

Para analizar la complejidad de los problemas paramétricos que parecen no ser tratables, Downey y Fellows introdujeron la jerarquía W [6]. Las clases

$$\mathrm{FPT} \subseteq \mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots \subseteq \mathrm{W}[P],$$

en esta jerarquía son cerradas bajo fpt-reducciones y se cree que son todas distintas. Tienen muchos problemas completos que son naturales (ver, e.g., [8, 11]). En lo que sigue, solo consideraremos las clases $\mathrm{W}[1]$ y $\mathrm{W}[P]$. Sus definiciones formales no son necesarias en nuestros argumentos, por lo que no las incluimos aquí. (El lector interesado puede encontrarlas en [11]). Necesitaremos la siguiente caracterización de $\mathrm{W}[P]$, análoga a la caracterización de NP en términos de «certificados».

Lema 17. *[11, Lem 3.8] Un problema parametrizado (Q, κ) sobre el alfabeto Σ está en $\mathrm{W}[P]$ si y solo si hay funciones computables $f, h : \mathbb{N} \to \mathbb{N}$, un polinomio $p(X)$, y un conjunto $Y \subseteq \Sigma^* \times \{0,1\}^*$ tales que:*

1. *Para todo par $(x, y) \in \Sigma^* \times \{0,1\}^*$ es decidible en tiempo $f(\kappa(x))p(|x|)$ si $(x, y) \in Y$.*
2. *Para todo par $(x, y) \in \Sigma^* \times \{0,1\}^*$, si $(x, y) \in Y$ entonces $|y| = h(\kappa(x))\lfloor \log |x| \rfloor$.*
3. *Para cada $x \in \Sigma^*$ tenemos que $x \in Q$ si y solo si existe $y \in \{0,1\}^*$ tal que $(x, y) \in Y$.*

El complemento $(Q, \kappa)^{\mathsf{C}}$ de un problema paramétrico (Q, κ) es el problema paramétrico $(\Sigma^* \setminus Q, \kappa)$. A partir de las definiciones se sigue que $P_1 \leq^{\mathrm{fpt}} P_2$ si y solo si $P_1^{\mathsf{C}} \leq^{\mathrm{fpt}} P_2^{\mathsf{C}}$. Para una clase K de problemas paramétricos, definiremos coK como la clase de todos los problemas paramétricos cuyo complemento está en K.

Proposición 18. p-OpenDef $\in$ co$\mathrm{W}[P]$.

Demostración. Por el Teorema 12 sabemos que $T \subseteq A^m$ no es abierta definible en $\mathbf{A}$ si y solo hay una biyección γ entre subconjuntos de A de cardinalidad como máximo m, donde γ es un subisomorfismo de $\mathbf{A}$ y no preserva T. Esta biyección puede ser codificada en una cadena binaria de largo $O(m^2 \log |A|)$, y podemos computar en tiempo polinomial en size$(\mathbf{A}, T)$ si γ es un subisomorfismo de $\mathbf{A}$ que no preserva T. Por lo tanto, usando (las codificaciones de) los subisomorfismos como certificados, nuestra proposición se sigue del Lema 17. (Notar que esto es un refinamiento del argumento que usamos en el Teorema 13 para probar OpenDef $\in$ coNP.) □

A continuación establecemos una cota inferior para la complejidad de p-OpenDef a través de una reducción de la siguiente versión parametrizada del problema Clique. Recordemos que una clique es un subgrafo completo.

p-Clique	
Instancia:	Un grafo finito $\boldsymbol{G}$ y un entero positivo k.
Parámetro:	k.
Pregunta:	¿Tiene $\boldsymbol{G}$ una clique de tamaño k?

En [7, Cor 3.2] se prueba que p-Clique es completo (bajo fpt-reducciones) para la clase W[1].

Lema 19. *p-Clique $\leq^{fpt}$ p-OpenDef$^{\mathsf{C}}$; luego p-OpenDef es hard para* coW[1].

Demostración. La idea es la misma que en la prueba del Teorema 13. Dada una entrada $\mathbf{G}, k$ para p-Clique la reducción computa la entrada $\mathbf{G} \sqcup \mathbf{K}_k, T_k$ para p-OpenDef$^{\mathsf{C}}$, donde $\mathbf{G} \sqcup \mathbf{K}_k$ es la unión disjunta de $\mathbf{G}$ con el grafo completo con vértices $\{1, \ldots, k\}$, y $T_k = \{(\sigma(1), \ldots, \sigma(k)) : \sigma$ es una permutación de $\{1, \ldots, k\}\}$. Es fácil ver que es una fpt-reducción, y que $\mathbf{G}$ tiene una clique de tamaño k si y solo si T_k no es abierta definible en $\mathbf{G} \sqcup \mathbf{K}_k$. (Notar que *no* es una reducción polinomial). □

En contraste con nuestro análisis de la complejidad clásica de OpenDef, no fuimos capaces de mostrar que p-OpenDef está en coW[1]. Sin embargo, cuando fijamos el lenguaje podemos establecer una cota superior. Para un lenguaje τ usaremos p-OpenDef$[\tau]$ para denotar la restricción de p-OpenDef a estructuras de entrada con lenguaje τ, y usaremos p-OpenDef$^{\mathsf{C}}[\tau]$ para denotar el complemento de este problema. Antes de empezar con las cotas superiores, tenemos la siguiente consecuencia del Lema 19.

Corolario 20. *Para cada lenguaje τ con al menos una relación de aridad como mínimo binaria tenemos que p-Clique $\leq^{fpt}$ p-OpenDef$^{\mathsf{C}}[\tau]$; y, por lo tanto, p-OpenDef$[\tau]$ es hard para* coW[1].

Demostración. Si τ tiene al menos una relación de aridad como mínimo binaria, es fácil de ver que

$$p\text{-OpenDef}^{\mathsf{C}}[\tau_{\text{GRAPH}}] \leq^{\text{fpt}} p\text{-OpenDef}^{\mathsf{C}}[\tau].$$

A partir de la prueba del Lema 19, se sigue que p-Clique $\leq^{\text{fpt}}$ p-OpenDef$^{\mathsf{C}}[\tau_{\text{GRAPH}}]$. □

Ahora pasaremos a establecer una cota superior para p-OpenDef$[\tau]$. Recordar que una sentencia es existencial si tiene la forma $\exists v_1 \ldots \exists v_l \, \alpha(v_1, \ldots, v_l)$ donde α es abierta. Sea $\Sigma_1[\tau]$ el conjunto de todas las sentencias existenciales sobre un lenguaje τ. Para un lenguaje dado τ, consideremos el siguiente problema de model checking parametrizado:

p-MC($\Sigma_1[\tau]$)	
Instancia:	Una τ-estructura finita $\mathbf{A}$ y una τ-sentencia existencial φ.
Parámetro:	$\text{size}_\tau(\varphi)$.
Pregunta:	¿$\mathbf{A}$ satisface φ?

Recordar que $\text{size}_\tau(\varphi) = \mathit{relcount}(\varphi)\log|\tau| + \mathit{varcount}(\varphi)\log(\mathit{var}^{\#}(\varphi))$. En [10, Thm. 8.4] se prueba que p-MC($\Sigma_1[\tau]$) está en W[1] para todo τ.

Teorema 21. *Para cada lenguaje τ, p-OpenDef$^{\mathsf{C}}[\tau] \leq^{fpt}$ p-MC($\Sigma_1[\tau]$).*
Así, p-OpenDef$[\tau] \in$ coW[1].

Demostración. Sea $\mathbf{A}$ una τ-estructura y $T \subseteq A^m$. Para cada tupla $\bar{a} = \langle a_1, \ldots, a_m\rangle \in A^m$, y cada $R \in \tau$ sea $\Delta_{\bar{a},R}(x_1, \ldots, x_m)$ la conjunción del siguiente conjunto de fórmulas atómicas:

$$\{R(x_{i_1}, \ldots, x_{i_r}) : (a_{i_1}, \ldots, a_{i_r}) \in R^{\mathbf{A}}\} \cup \{\neg R(x_{i_1}, \ldots, x_{i_r}) : (a_{i_1}, \ldots, a_{i_r}) \notin R^{\mathbf{A}}\},$$

donde $1 \leq i_j \leq m$ y r es la aridad de R. Observe que

$$\begin{aligned}\text{size}(\Delta_{\bar{a},R}) &= m^r \log|\tau| + rm^r \log m \\ &\leq q_r(m)\end{aligned}$$

para un polinomio adecuado $q_r(X)$. Notar que, como τ es fijo, $|\tau|$ es una constante y r está acotado superiormente por una constante. Además, observe que $\Delta_{\bar{a},R}$ puede ser computado en tiempo $O(m^r\,\text{size}(\mathbf{A}) + \text{size}(\Delta_{\bar{a},R}))$, por lo que hay un polinomio $p_r(X)$ tal que la computación de $\Delta_{\bar{a},R}$ puede ser realizada en, como máximo, $p_r(m)\,\text{size}(\mathbf{A})$ pasos. A continuación, definimos

$$\Delta_{\bar{a}}(x_1, \ldots, x_m) := \bigwedge\{\Delta_{\bar{a},R}(x_1, \ldots, x_m) : R \in \tau\}.$$

Sea ρ la mayor de las aridades de las relaciones en τ. Luego, $\text{size}(\Delta_{\bar{a}}) \leq |\tau|\,p_\rho(m)$, y $\Delta_{\bar{a}}$ es computable en tiempo acotado por $|\tau|\,p_\rho(m)\,\text{size}(\mathbf{A})$. Notar que $\Delta_{\bar{a}}$ caracteriza el tipo de isomorfismo de $\bar{a}$ en $\mathbf{A}$, i.e., para todo $\bar{b} \in A^m$ tenemos

$$\mathbf{A} \vDash \Delta_{\bar{a}}[b_1, \ldots, b_m] \iff \bar{a} \mapsto \bar{b} \text{ es un subisomorfismo de } \mathbf{A}.$$

Ahora, sea

$$\Delta_T(x_1, \ldots, x_m) := \bigvee\{\Delta_{\bar{a}}(x_1, \ldots, x_m) : \bar{a} \in T\},$$

y tomemos $\varphi_{\mathbf{A},T}$ como la sentencia

$$\exists\bar{x}_1 \ldots \exists\bar{x}_{t+1}\ \bigwedge\{\bar{x}_i \neq \bar{x}_j : 1 \leq i < j \leq t+1\} \wedge \bigwedge\{\Delta_T(\bar{x}_i) : 1 \leq i \leq t+1\},$$

donde t es el número de tuplas en T. Es directo probar que existen polinomios $p(X)$ y $q(X)$ tales que:

1. $\varphi_{\mathbf{A},T}$ puede ser computado en tiempo $p(m\,|T|)\,\mathrm{size}(\mathbf{A})$,

y

2. $\mathrm{size}(\varphi_{\mathbf{A},T}) \leq q(m\,|T|)$.

Ahora, notar que $\mathbf{A} \models \Delta_T[\bar{b}]$ si y solo si $\bar{b}$ tiene el mismo tipo de isomorfismo que alguna tupla en T; así $\varphi_{\mathbf{A},T}$ afirma que hay $t+1$ m-tuplas distintas, tal que cada una tiene el mismo tipo de isomorfismo que alguna tupla en T. Por lo tanto, $\mathbf{A} \models \varphi_{\mathbf{A},T}$ si y solo si hay $\bar{a} \in T$ y $\bar{b} \in A^m \setminus T$ tales que $\bar{a} \mapsto \bar{b}$ es un subisomorfismo de $\mathbf{A}$. Por el Teorema 12, esto nos dice que:

3. $\mathbf{A} \models \varphi_{\mathbf{A},T}$ si y solo si T no es abierta definible en $\mathbf{A}$.

Para concluir, observar que (1-3), garantizan que la transformación $\mathbf{A}, T \rightsquigarrow \mathbf{A}, \varphi_{\mathbf{A},T}$ es una fpt-reduccion de p-OpenDef$^{\mathsf{C}}[\tau]$ a p-MC$(\Sigma_1[\tau])$. (Notablemente, también es una reducción polinomial many-one.) □

Vale la pena señalar que, por el Corolario 20, el Teorema 21 en realidad vale con $\equiv^{\mathrm{fpt}}$ en lugar de $\leq^{\mathrm{fpt}}$. Esto es,

$$p\text{-OpenDef}^{\mathsf{C}}[\tau] \equiv^{\mathrm{fpt}} p\text{-MC}(\Sigma_1[\tau])$$

para cualquier lenguaje τ con al menos una relación de aridad como mínimo binaria.

Combinando las cotas superior e inferior que encontramos antes obtenemos el resultado principal de esta sección.

Teorema 22. *p-OpenDef$[\tau]$ es* coW$[1]$*-completo para cualquier lenguaje τ con al menos una relación de aridad como mínimo binaria. En particular, p-OpenDef$[\tau_{\mathrm{GRAPH}}]$ es* coW$[1]$*-completo.*

Demostración. Combinando el Corolario 20 y el Teorema 21. □

3.4 EL LARGO DE LAS FÓRMULAS ABIERTAS

En secciones previas mostramos que el problema de definibilidad para fórmulas abiertas no es tratable. Ahora discutiremos el tamaño de las fórmulas requeridas al definir. En particular, mostraremos que no es posible acotar polinomialmente el tamaño de la fórmula abierta requerida para definir una relación de una estructura dada. Mas formalmente, construiremos una secuencia de estructuras $\{(\mathbf{A}_n, T_n) : n \in \mathbb{N}\}$ cuyo tamaño crece polinomialmente en n, y tal que la definición más chica de T_n en $\mathbf{A}_n$ sea exponencialmente larga en n. Esto no es una sorpresa, ya que una cota polinomial al tamaño de las fórmulas abiertas que definen implicaría que OpenDef está en NP y, como sabemos que OpenDef es coNP-completo (Teorema 13), tendríamos que coNP $\subseteq$ NP.

Teorema 23. *Para cada $n \geq 3$ hay una estructura relacional finita $\mathbf{A}_n$ y una relación n^2-aria T_n sobre A_n tales que:*

- $\text{size}(\mathbf{A}_n, T_n) \in O(n^3 \log n)$,
- *T_n es abierta definible en $\mathbf{A}_n$, y*
- *Si φ es una fórmula abierta que define T_n en $\mathbf{A}_n$, entonces como mínimo ocurren $(n-1)^n$ literales diferentes en φ y, por lo tanto,* $\text{size}(\varphi) \geq 2^n$.

Demostración. Sea $n \geq 3$ y sea

$$\begin{aligned} A_n \;:=\; & \{a_1, \ldots, a_n\} \cup \{b_1, \ldots, b_n\} \cup \{c_1, \ldots, c_n\} \cup \\ & \{*_{11}, *_{12} \ldots, *_{1n}, \ldots, *_{n1}, \ldots, *_{nn}\} \end{aligned}$$

donde los 4 conjuntos del lado derecho son disjuntos de a pares. Por lo tanto, $|A_n| = 3n + n^2$. Sea M_1^n la matriz $n \times n$ representada a continuación

$$M_1^n := \begin{bmatrix} a_1 & *_{12} & \ldots & *_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_n & *_{n2} & \ldots & *_{nn} \end{bmatrix},$$

donde la primera columna de M_1^n es $\bar{a}$ y el resto es rellenado con $*$. Para cada $j \in \{2, \ldots, n\}$ tomamos la matriz $n \times n$ M_j^n tal que la primera columna de M_j^n sea $\bar{b}$, la j-ésima columna de M_j^n sea $\bar{c}$, y el resto de las entradas sea completado con $*$. Aquí hay un diagrama

$$M_j^n := \begin{bmatrix} b_1 & *_{12} & \ldots & *_{1j-1} & c_1 & *_{1j+1} & \ldots & *_{1n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ b_n & *_{n2} & \ldots & *_{nj-1} & c_n & *_{nj+1} & \ldots & *_{nn} \end{bmatrix}.$$

Luego, definimos

$$R_n := \{\bar{a}, \bar{b}, \bar{c}\} \cup \{\text{filas de } M_1^n\} \cup \{\text{filas de } M_2^n\} \cup \cdots \cup \{\text{filas de } M_n^n\}.$$

En lo que sigue, identificaremos a las n^2-tuplas con matrices $n \times n$ (de la manera obvia). Ajustaremos los índices de nuestras variables en consecuencia, i.e., escribiremos

$$\varphi\begin{pmatrix} x_{11} \ldots x_{1n} \\ \ddots \\ x_{n1} \ldots x_{nn} \end{pmatrix}$$

en vez de $\varphi(x_1, \ldots, x_{n^2})$.

Sea $T_n = \{M_1^n\}$; probaremos que $\mathbf{A}_n := (A_n, R_n)$ y T_n satisfacen las condiciones del teorema. Primero, observemos que, de acuerdo a nuestra definición de tamaño (ver Sección 3.1.1), tenemos

$$\begin{aligned} \text{size}(\mathbf{A}_n, T_n) &= (\text{size}(\mathbf{A}_n) + n^2) \log 4n \\ &= (1+n) + (4n + n(3+n^2) + n^2) \log 4n \\ &\in O(n^3 \log n). \end{aligned}$$

Fijaremos un número natural $n \geq 3$ para el resto de la prueba. Teniendo fijo n, omitiremos los subíndices y los subíndices respecto a n, escribiendo $\mathbf{A}$ en lugar de $\mathbf{A}_n$, M_j en lugar de M_j^n, y así sucesivamente. Mostraremos primero que $T = \{M_1\}$ es abierto definible en $\mathbf{A}$. Considerar la fórmula abierta

$$\alpha\begin{pmatrix} x_{11} \dots x_{1n} \\ \ddots \\ x_{n1} \dots x_{nn} \end{pmatrix} := x_{12} \neq x_{21} \wedge R(x_{11}, \dots, x_{n1}) \wedge \bigwedge_{j=1}^{n} R(x_{j1}, \dots, x_{jn}).$$

Supongamos que M es una matriz $n \times n$ con entradas en A tal que $\mathbf{A} \vDash \alpha(M)$. Entonces, la primera columna de M, que llamaremos $\bar{m} = (m_1, \dots, m_n)$, y cada una de sus filas deben estar en R. Como cada m_i debe ser la primera coordenada de alguna tupla en R, tenemos que $\bar{m} \in \{\bar{a}, \bar{b}\}$. Supongamos, primero, que $\bar{m} = \bar{a}$. Ya que para cada $i \in \{2, \dots, n\}$ hay exactamente una tupla en R que comienza con a_i, se sigue que M y M_1 coinciden desde la segunda fila para abajo. Además, observar que la primera fila de M debe ser o $\bar{a}$, o coincidir con la primer fila de M_1. Como el primer caso está excluido por el literal $x_{12} \neq x_{21}$ en α, concluimos que $M = M_1$. Supongamos a continuación que $\bar{m} = \bar{b}$. Una vez más, la condición $x_{12} \neq x_{21}$ en α nos dice que la primera fila de M no puede ser $\bar{b}$. Por lo tanto, tenemos que para cada $i \in \{1, \dots, n\}$ la i-ésima fila de M coincide con la i-ésima fila de M_{j_i} para algún $j_i \in \{2, \dots, n\}$. Dado $\bar{j} \in \{2, \dots, n\}^n$ sea $M^{\bar{j}}$ la matriz $n \times n$ tal que la i-ésima fila es la i-ésima fila de M_{j_i} para $i \in \{1, \dots, n\}$. Se sigue que

$$\{M : \mathbf{A} \vDash \alpha(M)\} = \{M_1\} \cup \{M^{\bar{j}} : \bar{j} \in \{2, \dots, n\}^n\}.$$

Para cada $\bar{j} \in \{2, \dots, n\}^n$ definimos

$$\lambda_{\bar{j}}\begin{pmatrix} x_{11} \dots x_{1n} \\ \ddots \\ x_{n1} \dots x_{nn} \end{pmatrix} := R(x_{1j_1}, \dots, x_{nj_n}),$$

y observar que $\mathbf{A} \vDash \lambda_{\bar{j}}(M^{\bar{j}})$ y $\mathbf{A} \vDash \neg\lambda_{\bar{j}}(M_1)$. Por lo tanto, si tomamos

$$\beta\begin{pmatrix} x_{11} \dots x_{1n} \\ \ddots \\ x_{n1} \dots x_{nn} \end{pmatrix} := \bigwedge_{\bar{j} \in \{2,\dots,n\}^n} \neg\lambda_{\bar{j}},$$

tenemos $\mathbf{A} \vDash (\alpha \wedge \beta)(M) \iff M = M_1$. Luego, T es abierta definible en $\mathbf{A}$.

Para concluir, probaremos que cada fórmula abierta que define T en $\mathbf{A}$ tiene como mínimo $(n-1)^n$ literales. Sea $\bar{x} := \langle x_{11}, x_{12} \dots, x_{1n}, \dots, x_{n1}, \dots, x_{nn} \rangle$, y sea At el conjunto de fórmulas atómicas con variables de $\bar{x}$. Todas las fórmulas de At tienen una de las siguientes dos formas:

- $v = w$, o
- $R(v_1, \dots, v_n)$,

donde $v, w, v_1, \ldots, v_n$ son de $\bar{x}$. Para una matriz $n \times n$ M con elementos de A tomamos

$$\mathrm{At}_M := \{\delta(\bar{x}) \in \mathrm{At} : \mathbf{A} \vDash \delta(M)\}.$$

Usaremos Δ para denotar el conjunto de todas las fórmulas en At de la forma$v = v$. Es inmediato que las únicas n-tuplas con entradas de M_1 que pertenecen a R son las filas de M_1 y su primera columna. Esto, junto con el hecho de que todas las entradas de M_1 son diferentes, nos dice que

$$\begin{aligned}\mathrm{At}_{M_1} \quad = \quad & \{R(x_{11}, \ldots, x_{1n}), R(x_{21}, \ldots, x_{2n}), \ldots, R(x_{n1}, \ldots, x_{nn})\} \cup \\ & \{R(x_{11}, \ldots, x_{n1})\} \cup \Delta.\end{aligned}$$

Un análisis similar muestra que para $\bar{j} = \langle j_1, \ldots, j_n \rangle \in \{2, \ldots, n\}^n$

$$\begin{aligned}\mathrm{At}_{M_{\bar{j}}} \quad = \quad & \{R(x_{11}, \ldots, x_{1n}), R(x_{21}, \ldots, x_{2n}), \ldots, R(x_{n1}, \ldots, x_{nn})\} \cup \\ & \{R(x_{11}, \ldots, x_{n1}), R(x_{1j_1}, \ldots, x_{nj_n})\} \cup \Delta.\end{aligned}$$

Esto es,

$$\mathrm{At}_{M_{\bar{j}}} = \mathrm{At}_{M_1} \cup \{\lambda_{\bar{j}}\}.$$

Sea $\varphi(\bar{x})$ una fórmula abierta que define T en $\mathbf{A}$. Notar que, ya que T es un singulete, podemos suponer sin pérdida de generalidad que φ es una conjunción de literales. Ahora, para cada $\bar{j} \in \{2, \ldots, n\}^n$ el único literal que distingue M_1 de $M^{\bar{j}}$ es $\neg\lambda_{\bar{j}}$, por lo tanto, debe ocurrir en φ. Luego, tenemos por lo menos $(n-1)^n$ literales en φ, y la prueba queda terminada. □

3.5 OpenDefAlg es coNP-completo

En lo que sigue un *grafo* es un modelo $\mathbf{G}$ del lenguaje con una única relación binaria E, tal que $E^{\mathbf{G}}$ es simétrica e irreflexiva. Recordemos que en el Teorema 13 demostramos que el siguiente problema es completo para coNP.

OpenDef[Graphs]	
Instancia:	Un grafo finito $\mathbf{G}$ y una relación target $R \subseteq G^k$.
Pregunta:	¿Es R abierta definible en $\mathbf{G}$?

Un *subisomorfismo* de un grafo $\mathbf{G} = (G, E)$ es una función inyectiva $\gamma : \mathrm{dom}\,\gamma \subseteq G \to G$ tal que para todo $a, b \in \mathrm{dom}\,\gamma$ se da que $(a, b) \in E$ si y solo si $(\gamma a, \gamma b) \in E$.

Teorema 24. OpenDefAlg *es* coNP*-completo.*

Demostración. Dado un grafo finito $\mathbf{G} = (G, E)$, sean $\hat{0}$ y $\hat{1}$ los primeros dos enteros positivos que no están en G. Definimos el álgebra $\mathbf{G}_* := (G \cup \{\hat{0}, \hat{1}\}, f)$ donde f es la operación binaria

$$f(a,b) = \begin{cases} \hat{1} & \text{si } (a,b) \in E \text{ o } a = b \in \{\hat{0}, \hat{1}\} \\ \hat{0} & \text{si no.} \end{cases}$$

Es directo comprobar que:

1. Si γ es un subisomorfismo de $\mathbf{G}$, entonces la extensión γ_* de γ dada por $\gamma_*(\hat{0}) = \hat{0}$ y $\gamma_*(\hat{1}) = \hat{1}$ es un subisomorfismo de $\mathbf{G}_*$.
2. Si δ es un subisomorfismo de $\mathbf{G}_*$, entonces $\delta[G] \subseteq G$ y $\delta|_G$ es un subisomorfismo de $\mathbf{G}$.

Probaremos a continuación que $\langle \mathbf{G}, R \rangle \longmapsto \langle \mathbf{G}_*, R \rangle$ es una reducción polinomial (de Karp) desde OpenDef[Graphs] a OpenDefAlg. Claramente, $\mathbf{G}_*$ puede ser computado a partir de $\mathbf{G}$ en tiempo polinomial, por lo que falta mostrar que R es abierta definible en $\mathbf{G}$ si y solo si R es abierta definible en $\mathbf{G}_*$. Fijemos un grafo finito $\mathbf{G}$ y $R \subseteq G^k$. Supongamos que R no es abierta definible en $\mathbf{G}$. Entonces, por el Teorema 12, hay un subisomorfismo γ de $\mathbf{G}$ que no preserva R. Ahora 1 dice que γ_* es un subisomorfismo de $\mathbf{G}_*$ y, como no preserva R, se sigue que R no es abierta definible en $\mathbf{G}_*$. Para la dirección restante, supongamos que R no es abierta definible en $\mathbf{G}_*$. Otra vez el Teorema 12 produce un subisomorfismo δ de $\mathbf{G}_*$ que no preserva R. Se sigue de (2), que $\delta|_G$ es un subismorfismo de $\mathbf{G}$ que no preserva R; por lo tanto R, no es abierta definible en $\mathbf{G}$.

Mostrar que OpenDefAlg está en coNP es una aplicación directa del Teorema 12. De hecho, cada instancia negativa $\langle \mathbf{A}, R \rangle$ de OpenDefAlg es atestiguada por una biyección γ entre subconjuntos de A que satisface condiciones fácilmente comprobables en tiempo polinomial respecto al tamaño de $\langle \mathbf{A}, R \rangle$. □

Se sigue de la prueba del Teorema 24 que la restricción de OpenDefAlg a estructuras con una única operación conmutativa es también completa para coNP.

Parte IV

ALGORITMOS

4

ESTRUCTURAS RELACIONALES

Nos centramos en la definibilidad mediante fórmulas abiertas de primer orden en un lenguaje de primer orden puramente relacional, es decir, sin símbolos para funciones ni constantes. Este capítulo se basa en el algoritmo presentando en [2].

4.1 DECIDIENDO OpenDef EFICIENTEMENTE

A la luz del Teorema 12 una posible estrategia para decidir $\mathrm{OpenDef}(\boldsymbol{A}, T)$ es comprobar que cada subiso de $\boldsymbol{A}$ preserva T. Los siguientes resultados muestran que, en muchos casos, no es necesario revisar todos los miembros de $\mathrm{subIso}\,\boldsymbol{A}$. Sea $\mathbb{S}(\boldsymbol{A})$ el conjunto de subestructuras de $\boldsymbol{A}$.

Teorema 25. *Sea $\boldsymbol{A}$ una estructura relacional finita. Supongamos $\mathcal{S} \subseteq \mathbb{S}(\boldsymbol{A})$ y $\mathcal{F} \subseteq \mathrm{subIso}\,\boldsymbol{A}$ tales que:*

1. *No hay dos miembros isomorfos en $\mathcal{S}$,*
2. *$\mathrm{aut}\,(\boldsymbol{B}) \subseteq \mathcal{F}$ para todo $\boldsymbol{B} \in \mathcal{S}$, y*
3. *para cada $\boldsymbol{C} \in \mathbb{S}(\boldsymbol{A}) \setminus \mathcal{S}$ hay un $\boldsymbol{C}' \in \mathcal{S}$ y $\rho, \rho^{-1} \in \mathcal{F}$ tal que $\rho : \boldsymbol{C} \to \boldsymbol{C}'$ es un isomorfismo.*

Luego, cada $\gamma \in \mathrm{subIso}\,\boldsymbol{A}$ es una composición de funciones en $\mathcal{F}$.

Demostración. Sea $\gamma : \boldsymbol{C} \to \boldsymbol{C}'$ un isomorfismo con $\boldsymbol{C}, \boldsymbol{C}' \in \mathbb{S}(\boldsymbol{A})$. Si ambos $\boldsymbol{C}$ y $\boldsymbol{C}'$ están en $\mathcal{S}$, entonces $\boldsymbol{C} = \boldsymbol{C}'$ y $\gamma \in \mathrm{aut}\,(\boldsymbol{C}) \subseteq \mathcal{F}$. Supongamos ahora que $\boldsymbol{C} \notin \mathcal{S}$ y $\boldsymbol{C}' \in \mathcal{S}$. Entonces, hay $\rho, \rho^{-1} \in \mathcal{F}$ tales que $\rho : \boldsymbol{C} \to \boldsymbol{C}'$ es un isomorfismo. Sea $\alpha := \gamma\rho^{-1}$, y observamos que $\alpha \in \mathrm{aut}\,(\boldsymbol{C})' \subseteq \mathcal{F}$. Por lo tanto, $\gamma = \alpha\rho$ es una composición de funciones en $\mathcal{F}$. Para concluir, supongamos que ni $\boldsymbol{C}$ ni $\boldsymbol{C}'$ están en $\mathcal{S}$. Entonces, hay $\boldsymbol{B} \in \mathcal{S}$ y $\rho, \rho^{-1}, \delta, \delta^{-1} \in \mathcal{F}$ tales que $\rho : \boldsymbol{C} \to \boldsymbol{B}$ y $\delta : \boldsymbol{C}' \to \boldsymbol{B}$ son isomorfismos. Notar que $\alpha := \delta\gamma\rho^{-1}$ es un automorfismo de $\boldsymbol{B}$, de esta forma $\alpha \in \mathcal{F}$. Luego, $\gamma = \delta^{-1}\alpha\rho$ es una composición de funciones en $\mathcal{F}$. □

Sea T un conjunto de tuplas, definimos el *espectro de T* de la siguiente manera

$$\mathrm{spec}\,T := \{|\{a_1, \ldots, a_n\}| : \langle a_1, \ldots, a_n\rangle \in T\}.$$

Combinando los Teoremas 12 y 25 obtenemos la siguiente caracterización de la definibilidad abierta, que es esencial para nuestra propuesta de algoritmo.

Corolario 26. *Sea $\boldsymbol{A}$ una estructura relacional finita y $T \subseteq A^n$. Supongamos que $\mathcal{S}$ y $\mathcal{F}$ son como en el Teorema 25, y sea $\mathcal{F}' := \{\gamma \in \mathcal{F} : |\mathrm{dom}\,\gamma| \in \mathrm{spec}(T)\}$. Entonces T es abierta definible en $\boldsymbol{A}$ si y solo si T es preservada por todos los subisos en $\mathcal{F}'$.*

Demostración. Si T es abierta definible en $\boldsymbol{A}$, entonces debe ser preservada por todos los subiso de $\boldsymbol{A}$; por lo que la vuelta es clara. Supongamos T preservada por todos los $\gamma \in \mathcal{F}'$. Fijemos $\delta \in \text{subIso}\, \boldsymbol{A}$ y sea $\langle a_1, \dots, a_n \rangle \in T$ tal que $\{a_1, \dots, a_n\} \subseteq \text{dom}\, \delta$. Notar que $|\{a_1, \dots, a_n\}| \in \text{spec}\, T$. Sea $\delta' := \delta|_{\{a_1,\dots,a_n\}}$; por el Teorema 25 sabemos que hay $\delta_1, \dots, \delta_m \in \mathcal{F}$ tal que $\delta' = \delta_1 \circ \cdots \circ \delta_m$. Ya que cada δ_j es una biyección, tenemos $|\text{dom}\, \delta_j| = |\text{dom}\, \delta| \in \text{spec}\, T$ y, por lo tanto, $\delta_j \in \mathcal{F}'$ para $j \in \{1, \dots, m\}$. Luego, como δ' es una composición de funciones que preservan T, tenemos que δ' preserva T. Finalmente, observamos que $\langle \delta a_1, \dots, \delta a_n \rangle = \langle \delta' a_1, \dots, \delta' a_n \rangle \in T$. □

A continuación, mostraremos que es posible quitar redundancias en las relaciones de la estructura sin afectar la definibilidad.

Sea $\bar{a} = \langle a_1, \dots, a_n \rangle$ una tupla. Definimos el *patrón de* $\bar{a}$, denotado con $\text{pattern}\, \bar{a}$, como una partición de $\{1, \dots, n\}$ tal que i, j están en el mismo bloque si y solo si $a_i = a_j$. Por ejemplo, $\text{pattern}\, \langle a, a, b, c, b, c \rangle = \{\{1, 2\}, \{3, 5\}, \{4, 6\}\}$. Denotaremos con $\lfloor \bar{a} \rfloor$ la tupla obtenida a partir de $\bar{a}$ borrando toda elemento igual a uno anterior. E.g., $\lfloor \langle a, a, b, c, b, c \rangle \rfloor = \langle a, b, c \rangle$. Dado un conjunto de tuplas S y un patrón θ, definimos $\lfloor S \rfloor := \{\lfloor \bar{s} \rfloor : \bar{s} \in S\}$ y $S_\theta := \{\bar{s} \in S : \text{pattern}\, \bar{s} = \theta\}$. Para una estructura $\mathbf{A}$ y un entero $k \geq 1$ definimos

- $\mathbf{A}_{\leq k}$ como la estructura obtenida a partir de $\mathbf{A}$ eliminando toda relación de aridad mayor que k, y
- $\lfloor \mathbf{A} \rfloor$ para la estructura obtenida a partir de $\mathbf{A}$ reemplazando cada relación R de $\mathbf{A}$ con las relaciones $\{\lfloor R_{\text{pattern}\, \bar{a}} \rfloor : \bar{a} \in R\}$.

Lema 27. *Sea* $\mathbf{A}$ *una estructura,* $T \subseteq A^m$*, y* $k := \text{máx}(\text{spec}(T))$*. Entonces,* T *es abierta definible en* $\mathbf{A}$ *si y solo si* T *es abierta definible en* $\lfloor \mathbf{A} \rfloor_{\leq k}$*.*

Demostración. Primero notar que si $\gamma : D \subseteq A \to A$ es inyectiva, entonces para cada relación R en $\mathbf{A}$ tenemos que: γ preserva R si y solo si γ preserva $\lfloor R_{\text{pattern}\, \bar{a}} \rfloor$ para cada $\bar{a} \in R$. Por lo que $\text{subIso}\, \mathbf{A} = \text{subIso}\, \lfloor \mathbf{A} \rfloor$. Luego, por el Teorema 12, tenemos que T es abierta definible en $\mathbf{A}$ si y solo si T es abierta definible en $\lfloor \mathbf{A} \rfloor$. Notar que el Corolario 26 implica que T es abierta definible en $\lfloor \mathbf{A} \rfloor$ si y solo si T es preservada por cada subisomorfismo con dominio de tamaño como máximo $k = \text{máx}(\text{spec}(T))$. Ahora observamos que si S es una relación de $\lfloor \mathbf{A} \rfloor$ con aridad mayor que k y $D \subseteq A$ tiene un tamaño como máximo k, entonces la restricción de S a D es vacía (ya que las tuplas en S no tienen repeticiones). En consecuencia, las relaciones de $\lfloor \mathbf{A} \rfloor$ con aridad máxima k determinan los subisos de $\lfloor \mathbf{A} \rfloor$ que deben preservar T para hacer T abierta definible. □

4.2 IMPLEMENTACIÓN

En esta sección presentamos un algoritmo para decidir definibilidad, un pseudo-código de la implementación es presentado en el Algoritmo 4.1.

El Algoritmo 4.1 decide si T es abierta definible en el modelo $\boldsymbol{A}$. Suponemos implementadas las siguientes funciones:

Algoritmo 4.1 Algoritmo para definibilidad abierta

```
1: function isOpenDef(A, T)
2:     spectrum = computeSpectrum(T)
3:     A = modelThinning(A, max(spectrum))
       {* spectrum es una lista de números naturales ordenada de mayor a menor*}
4:     global S = ∅
5:     if spectrum = [ ] then
6:         return True
7:     for B ∈ submodels(A, head(spectrum)) do
8:         if not isOpenDefR(B, T, tail(spectrum)) then
9:             return False
10:    return True
11: function isOpenDefR(A, T, spectrum)
12:    for S ∈ S do
13:        if cardinalityCheck(A,S) then
14:            if hay un iso γ : A → S then
15:                if γ y γ^{-1} preservan T then
16:                    return True
17:                else
18:                    return False
19:    for γ ∈ aut(A) do
20:        if γ no preserva T then
21:            return False
22:    S = S ∪ {A}
23:    if spectrum = [ ] then
24:        return True
25:    for B ∈ submodels(A, head(spectrum)) do
26:        if not isOpenDefR(B, tail(spectrum)) then
27:            return False
28:    return True
```

MODELTHINNING($\boldsymbol{A}$,k): Donde $\boldsymbol{A}$ es un modelo y $k > 0$, devuelve $\lfloor \boldsymbol{A} \rfloor_{\leq k}$ como en el Lema 27.

COMPUTESPECTRUM(t): Donde T es un conjunto de tuplas, devuelve una lista ordenada de mayor a menor conteniendo spec T.

CARDINALITYCHECK($\boldsymbol{A}$,$\boldsymbol{S}$): Donde $\boldsymbol{A}$ y $\boldsymbol{S}$ son modelos, devuelve True si para cada $R \in \mathcal{L}$, $|R^{\boldsymbol{A}}| = |R^{\boldsymbol{S}}|$ (ver Lema 28 a continuación).

SUBMODELS($\boldsymbol{A}$,n): Donde $\boldsymbol{A}$ es un modelo y n es un número natural, genera todos los submodelos de $\boldsymbol{A}$ con cardinalidad n.

Para describir mejor la forma en que funciona nuestro algoritmo, definimos un árbol $\mathrm{Tr}^{\boldsymbol{A}}$. La raíz de $\mathrm{Tr}^{\boldsymbol{A}}$ es $\boldsymbol{A}$. Dado un nodo $\boldsymbol{B}$ de $\mathrm{Tr}^{\boldsymbol{A}}$, sus hijos son los submodelos de $\boldsymbol{B}$ con l elementos, donde l es el mayor número en spec T estrictamente menor que $|B|$. Por lo tanto, si los números en $spec(T)$ son $l_1 > l_2 > \ldots > l_r$, el nivel j de $Tr^{\boldsymbol{A}}$ (para $j \geq 1$) contiene todos los submodelos de $\boldsymbol{A}$ con l_j elementos. El nivel 0 contiene solo a $\boldsymbol{A}$. Como el Algoritmo 4.1 recorre $\mathrm{Tr}^{\boldsymbol{A}}$ primero en profundidad,

comienza por el nivel 1, a menos que $|A| \in \operatorname{spec} T$ cuando comienza por el nivel 0. Para cada nodo $\boldsymbol{B}$ realiza lo siguiente:

- Comprobar si hay un isomorfismo $\gamma : \boldsymbol{B} \to \boldsymbol{S}$ para algún $\boldsymbol{S} \in \mathcal{S}$;
  1. si tal isomorfismo no existe, revisar los automorfismos de $\boldsymbol{B}$. Si algún miembro de $\operatorname{aut}(\boldsymbol{B})$ no preserva T, devolver False. De otra manera, agregar $\boldsymbol{B}$ a $\mathcal{S}$ y proceder a procesar los hijos de $\boldsymbol{B}$,
  2. si se encuentra un isomorfismo γ, comprobar que tanto γ como γ^{-1} preserven T. Si alguna preservación falla, devolver False. De otra manera, moverse a un hermano de $\boldsymbol{B}$ sin procesar.

Cuando el algoritmo ha terminado de procesar todos los nodos devuelve True. Un ejemplo de la ejecución del algoritmo es presentado en la Sección 4.4.

En nuestra implementación del Algoritmo 4.1, los isomorfismos se buscan como CSP (constraint satisfaction problem) que son resueltos utilizando MINION (versión 1.8) [13]. Cuando revisamos la existencia de un isomorfismo entre estructuras $\boldsymbol{A}$ y $\boldsymbol{B}$ primero verificamos que sus dominios tengan el mismo tamaño y que $|R^{\boldsymbol{A}}| = |R^{\boldsymbol{B}}|$ para cada R en el lenguaje. Después de esto, es suficiente verificar si hay un homomorfismo inyectivo entre las estructuras. Esto se hace en la Línea 14.

Lema 28. *Sean $\boldsymbol{A}$ y $\boldsymbol{B}$ estructuras relacionales finitas tales que $|R^{\boldsymbol{A}}| = |R^{\boldsymbol{B}}|$. Entonces, cualquier R-homomorfismo biyectivo de $\boldsymbol{A}$ a $\boldsymbol{B}$ es un R-isomorfismo.*

Demostración. Supongamos que γ^{-1} no es un isomorfismo. Entonces hay $\bar{b}$ tal que $\bar{b} \in R^{\boldsymbol{B}}$ pero $\gamma^{-1}(\bar{b}) \notin R^{\boldsymbol{A}}$, luego $|R^{\boldsymbol{A}}| \neq |R^{\boldsymbol{B}}|$. □

Sabiendo que CSP es difícil, vale la pena tratar de limitar las llamadas a MINION. En particular, las comprobaciones de cardinalidad descritas antes son maneras baratas de descubrir la inexistencia de isomorfismos. Notar que es más probable que las comprobaciones fallen en lenguajes grandes. En este aspecto, observe que cambiar $\boldsymbol{A}$ por $\lfloor \boldsymbol{A} \rfloor$ (Lema 27) aumenta el tamaño del lenguaje subyacente sin incrementar el tamaño del dominio de los isomorfismos involucrados.

4.3 DETECCIÓN DE HOMOMORFISMOS

Para la detección de homomorfismos, modelizamos el problema como un CSP y utilizamos Minion [13] como solucionador.

4.3.1 *CSP*

Un *CSP* (Constraint Satisfaction Problem) (ver, e.g., [15, Capítulo 6]) es definido como una terna $\langle X, D, C \rangle$, donde

$X = \{X_1, \ldots, X_n\}$ es un conjunto de variables,

$D = \{D_1, \ldots, D_n\}$ es un conjunto con los respectivos dominios de los valores,

$C = \{C_1, \ldots, C_m\}$ es un conjunto de restricciones.

Cada variable X_i se mueve en los valores del respectivo dominio no vacío D_i. Cada restricción $C_j \in C$ es un par $\langle t_j, R_j \rangle$, donde t_j es una k-upla de variables y R_j es una relación k-aria en el correspondiente dominio de cada variable. Una valuación sobre las variables es una función desde un subconjunto de las variables a un particular conjunto de valores en los correspondientes dominios de valores. Una valuación v satisface la restricción $\langle t_j, R_j \rangle$ si los valores asignados a las variables t_j satisfacen la relación R_j.

Una valuación es consistente si no viola ninguna de las restricciones. Una valuación es completa si incluye todas las variables. Una valuación es solución si es consistente y completa. En este último caso, diremos que la valuación *resuelve* el CSP.

4.3.2 *CSP para calcular homomorfismos*

Sean $\mathbf{A}$, $\mathbf{B}$ $\mathcal{L}$-estructuras, queremos encontrar homomorfismos de $\mathbf{A}$ en $\mathbf{B}$ resolviendo una instancia de CSP. Tomamos

$$X = \{X_i : i \in A\}$$

$$D_i = B$$

$$C = C^R \cup C^f$$

donde C^R es tal que para cada $R \in \mathcal{L}$ n-aria y para cada $(a_1, \ldots, a_n) \in R^{\mathbf{A}}$, se da que $\left((X_{a_1}, \ldots, X_{a_n}), R^{\mathbf{B}}\right) \in C^R$. Mientras que C^f es tal que, para cada $f \in \mathcal{L}$ n-aria, para cada $(a_1, \ldots, a_n, a') \in \text{graph}\left(f^{\mathbf{A}}\right)$, se da que $\left((X_{a_1}, \ldots, X_{a_n}, X_{a'}), \text{graph}\left(f^{\mathbf{B}}\right)\right) \in C^f$.

Supongamos que $V : X \to B$ es una valuación solución de (X, D, C). Entonces el homomorfismo γ determinado por V será $\gamma(a) = V(X_a)$.

Si, además, quisiéramos que el homomorfismo fuera inyectivo, bastaría con agregar una restricción:

$$\left((X_1, \ldots, X_m), \text{para todo } (i, j) \text{ con } i \neq j \text{ y } i, j \in \{1, \ldots, m\} \text{ se da } X_i \neq X_j\right)$$

donde $X_1, \ldots, X_m$ son todas las variables en X.

Para que fuera sobreyectivo, bastaría agregar la siguiente restricción:

$$\left((X_1, \ldots, X_m), \bigcup_{i=1}^{m} \{X_i\} = B\right).$$

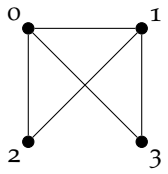
En el caso particular de la búsqueda de automorfismos, el lema 29 nos permite simplificar la búsqueda, al encontrar endomorfismos (homomorfismos de $\mathbf{A}$ en $\mathbf{A}$) inyectivos.

Lema 29. *Sea* **A** *una estructura finita, entonces todo endomorfismo inyectivo de* **A** *es un automorfismo de* **A**.

Demostración. Sea γ un endomorfismo inyectivo de **A**. Como A es finito tenemos que γ es sobre y además $\gamma^{-1} = \gamma^k$ para algún $k \geq 1$. □

4.4 UNA EJECUCIÓN DEL ALGORITMO 4.1

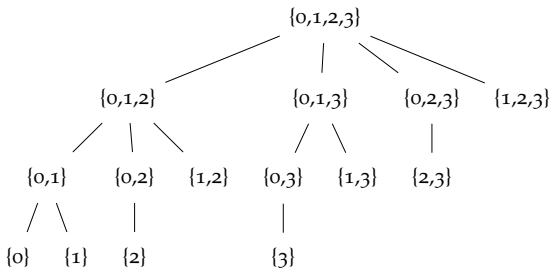
Para ilustrar como funciona el Algoritmo 4.1, mostramos una ejecución para la siguiente instancia. La estructura de entrada es el grafo $\boldsymbol{G} = (\{0,1,2,3\}, E)$



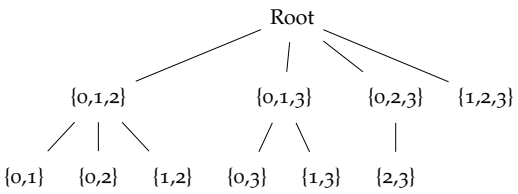
y la relación target es

$$T = \{(a,b,c,d) : E(b,c) \wedge (a = b \vee c = d)\}.$$

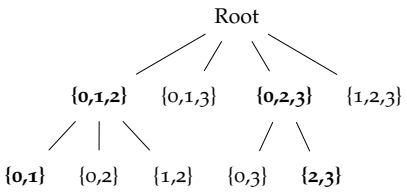
Un algoritmo de fuerza bruta basado en el Teorema 12 debería comprobar que todo subisomorfismo de $\boldsymbol{G}$ preserve T. Esto es, para cada subuniverso S de $\boldsymbol{G}$, computar todos los subisomorfismos con dominio S y revisarlos por preservación. Esto significa encontrar cada subuniverso listado en el árbol de abajo y calcular los automorfismos de todos estos subuniversos y comprobarlos para su preservación.



La primera mejora del Algoritmo 4.1 nos permite podar niveles del árbol cuyos nodos no tengan cardinalidad en spec(T) (esto es correcto debido al Lema 12). En el corriente ejemplo, únicamente se procesan subuniversos con tamaño en spec$(T) = \{2,3\}$. Esto equivale a procesar solo los siguientes subuniversos. (El nodo raíz no es un subuniverso a procesar, lo añadimos para visualizar mejor el árbol recorrido por DFS).



La segunda mejora es debida al orden en el cual el Algoritmo 4.1 computa los subisomorfismos con un dominio dado S. Primero, intenta encontrar un isomorfismo de S hacia algún subuniverso ya procesado. En el caso de que tal isomorfismo exista, no se computan más isomorfismos desde S, y los subuniversos contenidos en S son podados (esto es correcto por el Corolario 26). Finalmente, los subuniversos de $\boldsymbol{G}$ procesados por el Algoritmo 4.1 son los siguientes. Los nodos en negrita están en $\mathcal{S}$ y, por lo tanto, son los únicos que se comprueban en busca de automorfismos y subuniversos.



En vista de las estrategias que el Algoritmo 4.1 emplea tenemos una idea clara de lo que vuelve bien condicionada a una instancia $\boldsymbol{A}, T$:

- Relaciones target con un espectro pequeño comparado con el conjunto potencia de A.
- Estructuras con pocos tipos de isomorfismo entre sus subestructuras cuya cardinalidad esté en el espectro.

5

ESTRUCTURAS ALGEBRAICAS

5.1 COMPUTANDO EL TIPO DE ISOMORFISMO DE UNA TUPLA

En esta sección, basada en [4], presentamos un algoritmo para computar el tipo de isomorfismo de una tupla $\bar{a}$ en una estructura finita algebraica $\mathbf{A}$, y probamos que es correcto. Comenzaremos con algunas definiciones y resultados preliminares necesarios.

Sea $\mathbf{A}$ un álgebra finita, para cada $k \in \omega$ definimos una relación de equivalencia $\approx_k$ sobre A^n tal que $\bar{a} \approx_k \bar{b}$ si y solo si para todo par de términos $t, s \in \mathcal{T}_k(x_0, \ldots, x_{n-1})$ tenemos que

$$t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a}) \iff t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b}).$$

Definimos la relación de equivalencia $\approx$ sobre A^n tal que $\bar{a} \approx \bar{b}$ si y solo si $\bar{a} \approx_k \bar{b}$ para todo $k \in \omega$. Para $\bar{a} \in A^n$ definimos

$$\mathrm{K}_{\bar{a}} := \text{mín}\{k \in \omega : \text{ para todo } t \in \mathcal{T}_k(\bar{x}) \text{ hay un } \hat{t} \in \mathcal{T}_{k-1}(\bar{x}) \text{ tal que } t^{\mathbf{A}}(\bar{a}) = \hat{t}^{\mathbf{A}}(\bar{a})\}.$$

Notar que, como $\mathbf{A}$ es finito, este mínimo siempre existe.

Lema 30. *Sean $\mathbf{A}$ un álgebra finita y $\bar{a}, \bar{b} \in A^n$.*

1. *Si $\bar{a} \approx_{\mathrm{K}_{\bar{a}}} \bar{b}$, entonces para cada término $t(\bar{x})$ hay un término $\hat{t}(\bar{x}) \in \mathcal{T}_l$ con $l < \mathrm{K}_{\bar{a}}$ tal que $t^{\mathbf{A}}(\bar{a}) = \hat{t}^{\mathbf{A}}(\bar{a})$ y $t^{\mathbf{A}}(\bar{b}) = \hat{t}^{\mathbf{A}}(\bar{b})$.*

2. *Si $\bar{a} \approx_{\mathrm{K}_{\bar{a}}} \bar{b}$, entonces $\bar{a} \approx \bar{b}$.*

Demostración. (1) Observemos primero que de la definición se sigue que $\mathrm{K}_{\bar{a}} = \mathrm{K}_{\bar{b}}$, por lo que escribiremos K para $\mathrm{K}_{\bar{a}}$. Fijemos un término $t(\bar{x})$. Si $t(\bar{x}) \in \mathcal{T}_{K-1}$ podemos tomar $t = \hat{t}$, luego supongamos que $t \in \mathcal{T}_{K+j}$ con $j \geq 0$. Mostraremos que existe $\hat{t}$ por inducción en j. Supongamos $j = 0$; por la definición de $\mathrm{K}_{\bar{a}}$, hay un $\hat{t} \in \mathcal{T}_{K-1}$ tal que $t^{\mathbf{A}}(\bar{a}) = \hat{t}^{\mathbf{A}}(\bar{a})$ y, como $\bar{a} \approx_{\mathrm{K}} \bar{b}$, se sigue que $t^{\mathbf{A}}(\bar{b}) = \hat{t}^{\mathbf{A}}(\bar{b})$. Supongamos ahora que $j > 0$ y $t = f(t_1, ..., t_r)$, con cada $t_i \in \mathcal{T}_{K+j-1}$. Por hipótesis inductiva, hay términos $\hat{t}_1, ..., \hat{t}_r \in \mathcal{T}_u$ con $u < K$, tales que cada uno satisface $t_i^{\mathbf{A}}(\bar{a}) = \hat{t}_i{}^{\mathbf{A}}(\bar{a})$ y $t_i^{\mathbf{A}}(\bar{b}) = \hat{t}_i{}^{\mathbf{A}}(\bar{b})$. Esto significa que $\tilde{t} := f(\hat{t}_1, ..., \hat{t}_r) \in \mathcal{T}_{u+1}$ satisface $\tilde{t}^{\mathbf{A}}(\bar{a}) = t^{\mathbf{A}}(\bar{a})$ y $\check{t}^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b})$. Como $u + 1 \leq K$, por el caso $j = 0$ tenemos un término $\hat{t} \in \mathcal{T}_l$ con $l < K$ para el cual $\hat{t}^{\mathbf{A}}(\bar{a}) = \tilde{t}^{\mathbf{A}}(\bar{a}) = t^{\mathbf{A}}(\bar{a})$ y $\hat{t}^{\mathbf{A}}(\bar{b}) = \tilde{t}^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b})$.

(2) Sea $t, s \in \mathcal{T}$ tal que $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$. Tomemos $\hat{t}, \hat{s}$ como los términos dados por (1). Como $\hat{t}^{\mathbf{A}}(\bar{a}) = \hat{s}^{\mathbf{A}}(\bar{a})$, por hipótesis tenemos que $\hat{t}^{\mathbf{A}}(\bar{b}) = \hat{s}^{\mathbf{A}}(\bar{b})$ y, por lo tanto, $t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b})$. □

A continuación, introducimos, en la forma de pseudocódigo, el algoritmo que computa la función isoType (ver Algoritmo 5.1 abajo). Esta función toma un álgebra

finita $\mathbf{A}$ y una tupla $\bar{a}$ de A y devuelve una representación del tipo de isomorfismo de $\bar{a}$ y el subuniverso de $\mathbf{A}$ generado por $\bar{a}$ listado en un orden específico. El tipo de isomorfismo es obtenido recorriendo los términos en un orden específico y evaluándolos en $\bar{a}$. Como veremos (Corolario 32), es suficiente con guardar qué términos devuelven el mismo valor para capturar el tipo de isomorfismo de $\bar{a}$.

Describamos cómo funciona el Algoritmo 5.1. Fijemos un álgebra de entrada $\mathbf{A}$ con lenguaje τ (supondremos que los símbolos de función en τ están listados en un orden específico). Dada una tupla $\bar{a} \in A^n$, al comienzo del algoritmo, la variable `V` es inicializada como $[a_0, \ldots, a_{n-1}]$. Entonces, comienza el bucle while y los elementos añadidos a `V` son obtenidos por la aplicación de operaciones fundamentales de $\mathbf{A}$ a valores anteriores de `V`. Luego, todos los elementos en `V` pertenecen a $\mathrm{Sg}(\bar{a})$. La variable `P` guarda las repeticiones en las apariciones de los elementos a medida que son computados, mientras que la variable `H` guarda los índices donde cada elemento de $\mathrm{Sg}(\bar{a})$ aparece por primera vez en `V`. Al final de cada iteración del bucle while, la variable `N` se asigna con los índices que fueron añadidos a `H` en esa iteración. Como A es finito, eventualmente no se generan nuevos elementos, y `N` es asignada con la lista vacía. Esto garantiza terminación y el hecho de que todos los elementos de $\mathrm{Sg}(\bar{a})$ son listados en `V` cuando el algoritmo termina. Observemos que el bucle while es ejecutado exactamente $\mathrm{K}_{\bar{a}}$ veces para la entrada $\bar{a}$. Debemos remarcar que las líneas 3 y 16 - 17 en el Algoritmo 5.1, cuyo propósito es inicializar y actualizar la variable `V'`, no son necesarias para computar la función isoType, pero son incluidas para hacer nuestras pruebas más fáciles de seguir.

En cada pasada, para computar elementos de `V` y dada una aridad k de alguna operación fundamental, construimos una lista `T` de k-uplas de índices de elementos donde las operaciones fundamentales de aridad k serán aplicadas. Requerimos que estas tuplas tengan al menos un elemento que no haya aparecido antes para evitar computaciones innecesarias.

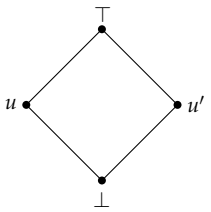


Figura 5.1: Reticulado $\mathbf{D}$.

Para ilustrar cómo funciona este algoritmo mostraremos un ejemplo. Sea $\mathbf{D} = \langle \{\top, \bot, u, u'\}, \wedge, \vee \rangle$ el reticulado de la Figura 5.1. Suponemos que las operaciones fundamentales están en el orden: $[\wedge, \vee]$. Para la entrada $\mathbf{D}$, $\bar{a} = (u, u', \bot)$, las variables son inicializadas del siguiente modo:

$$\mathtt{V} = [u, u', \bot] \qquad \mathtt{P} = [[0], [1], [2]] \qquad \mathtt{H} = [0, 1, 2] \qquad \mathtt{N} = [0, 1, 2].$$

Después de la primera corrida a través del bucle while, el estado es:

$$\mathtt{V} = [u, u', \bot, \underbrace{u}_{u\wedge u}, \underbrace{\bot}_{u\wedge u'}, \underbrace{\bot}_{u\wedge\bot}, \underbrace{\bot}_{u'\wedge u}, \underbrace{u'}_{u'\wedge u'}, \underbrace{\bot}_{u'\wedge\bot}, \underbrace{\bot}_{\bot\wedge u}, \underbrace{\bot}_{\bot\wedge u'}, \underbrace{\bot}_{\bot\wedge\bot}, \underbrace{u}_{u\vee u}, \underbrace{\top}_{u\vee u'}, \underbrace{u}_{u\vee\bot}, \underbrace{\top}_{u'\vee u}, \underbrace{u'}_{u'\vee u'}, \underbrace{u'}_{u'\vee\bot}, \underbrace{u}_{\bot\vee u}, \underbrace{u'}_{\bot\vee u'}, \underbrace{\bot}_{\bot\vee\bot}]$$

$$\mathtt{P} = [[0, 3, 12, 14, 18], [1, 7, 16, 17, 19], [2, 4, 5, 6, 8, 9, 10, 11, 20], [13, 15]]$$
$$\mathtt{H} = [0, 1, 2, 13]$$
$$\mathtt{N} = [13].$$

Ahora, el elemento $\top$, que no estaba antes en V, hace su primera aparición en la posición 13. Así, en la siguiente pasada las operaciones fundamentales serán aplicadas a todos los pares con elementos de $[u, u', \top, \bot]$ en los cuales $\top$ aparezca al menos una vez. Luego, la siguiente ronda produce:

$$\mathtt{V} = \mathtt{V} + [\underbrace{u}_{u\wedge\top}, \underbrace{u'}_{u'\wedge\top}, \underbrace{\bot}_{\bot\wedge\top}, \underbrace{u}_{\top\wedge u}, \underbrace{u'}_{\top\wedge u'}, \underbrace{\bot}_{\top\wedge\bot}, \underbrace{\top}_{\top\wedge\top}, \underbrace{\top}_{u\vee\top}, \underbrace{\top}_{u'\vee\top}, \underbrace{\top}_{\bot\vee\top}, \underbrace{\top}_{\top\vee u}, \underbrace{\top}_{\top\vee u'}, \underbrace{\top}_{\top\vee\bot}, \underbrace{\top}_{\top\vee\top}]$$

$$\mathtt{P} = [[0, 3, 12, 14, 18, 21, 24], [1, 7, 16, 17, 19, 22, 25], [2, 4, 5, 6, 8, 9, 10, 11, 20, 23, 26,], [13, 15, 27, 28, 29, 30, 31, 32, 33, 34]]$$
$$\mathtt{H} = [0, 1, 2, 13]$$
$$\mathtt{N} = [\,].$$

La variable N queda ahora vacía debido al hecho de que no se produjeron nuevos elementos, por lo que el Algoritmo 5.1 termina retornando la partición P y $\mathtt{U} = [u, u', \bot, \top] = \mathrm{Sg}(\bar{a})$. Al ejecutar el algoritmo para la tupla $\bar{b} = (u', u, \bot)$, obtenemos:

$$\mathtt{V} = [u', u, \bot, u', \bot, \bot, \bot, u, \bot, \bot, \bot, \bot, u', \top, u', \top, u, u, u', u, \bot, u', u, \bot, u', u, \bot, \top, \top, \top, \top, \top, \top, \top, \top]$$

$$\mathtt{H} = [0, 1, 2, 13]$$
$$\mathtt{P} = [[0, 3, 12, 14, 18, 21, 24], [1, 7, 16, 17, 19, 22, 25], [2, 4, 5, 6, 8, 9, 10, 11, 20, 23, 26,], [13, 15, 27, 28, 29, 30, 31, 32, 33, 34]]$$
$$\mathtt{U} = [u', u, \bot, \top].$$

Como las particiones finales son las mismas para $\bar{a}$ y $\bar{b}$, por el Corolario 32, el mapeo γ de $[u, u', \bot, \top]$ a $[u', u, \bot, \top]$ es un subisomorfismo que satisface $\gamma : \bar{a} \mapsto \bar{b}$.

5.1.1 *Correctitud y completitud*

Para un nombre de variable X en $\{\mathtt{V}, \mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$ y $k \in \{0, \ldots, \mathrm{K}_{\bar{a}}\}$, usaremos

- $\mathtt{X}_{\bar{a},k}$ para el valor de la variable X en una ejecución del Algoritmo 5.1 con entrada $\mathbf{A}, \bar{a}$, después de k ejecuciones del bucle while.

Además, dada una aridad r de un símbolo de función de $\mathbf{A}$, usaremos

- $\mathtt{T}^r_{\bar{a},k}$ para el valor asignado a la variable T en una ejecución del Algoritmo 5.1 con entrada $\mathbf{A}, \bar{a}$, una vez que se ha fijado una aridad r para el bucle (línea 10), durante la k-ésima ejecución del bucle while.

Algoritmo 5.1 Tipo de isomorfismo de una tupla

1: **function** IsoType($\mathbf{A}$, $\bar{a}$)
▷ $\mathbf{A}$ es un álgebra y $\bar{a}$ es una tupla de A
2: $\mathtt{V} = [a_0, \ldots, a_{n-1}]$
3: $\mathtt{V}' = [\text{“}x_0\text{”}, \ldots, \text{“}x_{n-1}\text{”}]$ ▷ esta es una lista de variables (con los términos representados como strings)
4: $\mathtt{P} = [B_1, \ldots, B_k]$, la partición de $\{0, \ldots, n-1\}$ donde i, j están en el mismo bloque sii $a_i = a_j$
5: $\mathtt{H} = \mathtt{sorted}([\text{mín}(B_j) : B_j \in \mathtt{P}])$ ▷ ordenados crecientemente
6: $\mathtt{N} = \mathtt{H}$
7: $\mathtt{arities} = [r_1, ..., r_k]$ ▷ donde $r_1 < ... < r_k$ son todas las aridades de las operaciones en $\mathbf{A}$
8: **while** $\mathtt{N} \neq \emptyset$ **do**
9: $\mathtt{H\_old} = \mathtt{copy}(\mathtt{H})$
10: **for** $r \in \mathtt{arities}$ **do**
11: $\mathtt{T} = \mathtt{sorted}([\bar{l} \in (\mathtt{H\_old})^r : l_j \in \mathtt{N} \text{ para algun } j])$ ▷ ordenados lexicográficamente
12: **for** $f \in \mathtt{op}(r)$ **do** ▷ $\mathtt{op}(r)$es la lista de símbolos de operaciones de aridad r en el orden dado por τ
13: **for** $\bar{l} \in \mathtt{T}$ **do**
14: $\mathtt{value} = f^{\mathbf{A}}(\mathtt{V}[l_0], ..., \mathtt{V}[l_{r-1}])$
15: agregar value a V
16: $\mathtt{term} = f \mathbin{+\!\!+} \text{“(”} \mathbin{+\!\!+} \mathtt{V}'[l_0] \mathbin{+\!\!+} \text{“,”} \mathbin{+\!\!+} ... \mathbin{+\!\!+} \text{“,”} \mathbin{+\!\!+} \mathtt{V}'[l_{r-1}] \mathbin{+\!\!+} \text{“)”}$
17: agregar term a $\mathtt{V}'$
▷ actualizar la partición P
18: **if** hay un $j \in \mathtt{H}$ tal que $\mathtt{value} = \mathtt{V}[j]$ **then** ▷ value ya estaba en V
19: agregar $|\mathtt{V}| - 1$ al bloque de P que contiene j
20: **else** ▷ value es nuevo
21: agregar el bloque $\{|\mathtt{V}| - 1\}$ a P
22: agregar $|\mathtt{V}| - 1$ a H
▷ actualización de N
23: $\mathtt{N} = [l : l \in \mathtt{H} \text{ and } l \notin \mathtt{H\_old}]$
24: $\mathtt{U} = [\mathtt{V}[j] : j \in \mathtt{H}]$ ▷ U lista los elementos de $\mathrm{Sg}(\bar{a})$ en el orden en que aparecieron en V
25: **return** P, U

Escribiremos $\mathtt{X}_{\bar{a}}$ para $\mathtt{X}_{\bar{a},\mathrm{K}_{\bar{a}}}$ y, para $k > \mathrm{K}_{\bar{a}}$, definimos $\mathtt{X}_{\bar{a},k}$ como $\mathtt{X}_{\bar{a}}$. El subíndice $\bar{a}$ se omite cuando no puede haber confusión.

Dada una tupla $\bar{a} \in A^n$ y $k \in \omega$ definimos $\rho_{\bar{a},k}$ como la relación de equivalencia sobre $\mathcal{T}_k(x_0, \ldots, x_{n-1})$ tal que $t \ \rho_{\bar{a},k} \ s \iff t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$. Además, $Eq(Y)$ denotará el conjunto de todas las relaciones de equivalencia sobre el conjunto Y. El siguiente lema provee una serie de resultados técnicos necesarios para establecer la correctitud del Algoritmo 5.1, lo que hacemos en el Corolario 32 abajo.

Lema 31. *Para todo $\bar{a} \in A^n$ y $k \in \omega$ tenemos que:*

1. *Cada miembro de $\mathtt{V}'_k$ es un término en las variables $x_0, \ldots, x_{n-1}$ de grado como máximo k.*

2. *El invariante $\mathtt{V}_k[i] = \mathtt{V}'_k[i]^{\mathbf{A}}(\bar{a})$ se mantiene a través de todo el bucle for de la línea 13.*

3. *Si* $\bar{a} \approx_k \bar{b}$ *entonces* $\mathtt{X}_{\bar{a},k} = \mathtt{X}_{\bar{b},k}$ *para cualquier variable* $\mathtt{X}$ *en* $\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$.
4. *Para todo* $1 \leq k \leq \mathrm{K}_{\bar{a}}$ *hay una función*

$$F_k : \mathcal{T}_k(x_0, \ldots, x_{n-1}) \times Eq(\mathcal{T}_{k-1}(x_0, \ldots, x_{n-1})) \to \omega,$$

tal que para cada $t \in \mathcal{T}_k(x_0, \ldots, x_{n-1})$, *el índice* $i = F_k(t, \rho_{\bar{a},k-1})$ *satisface* $i \leq |\mathtt{V}_k| - 1$ *y* $t^{\mathbf{A}}(\bar{a}) = \mathtt{V}_k[i]$.

5. *Para todo* $k \leq \mathrm{K}_{\bar{a}}$ *tenemos que* $\mathtt{P}_k$ *determina* $\mathtt{X}_j$ *para todo* $j < k$ *y cualquier nombre de variable* $\mathtt{X}$ *en* $\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$.
6. *Si* $\mathtt{P}_{\bar{a}} = \mathtt{P}_{\bar{b}}$, *entonces* $\bar{a} \approx \bar{b}$.

Demostración. (1) es fácilmente probado por inducción en k. (2) es claro a partir de la líneas 14 a 17.

(3) Haremos inducción en k. El caso $k = 0$ es directo. Supongamos ahora $k + 1 \leq \text{mín}(\mathrm{K}_{\bar{a}}, \mathrm{K}_{\bar{b}})$, y supongamos $\bar{a} \approx_{k+1} \bar{b}$. Como $\bar{a} \approx_k \bar{b}$, por nuestra hipótesis inductiva tenemos que

- $\mathtt{X}_{\bar{a},k} = \mathtt{X}_{\bar{b},k}$ para cualquier variable $\mathtt{X}$ en $\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$.

Notar que en la primera línea ejecutada cuando empezamos la $k+1$-ésima ejecución del bucle while se establece el valor de `H_old` al valor de $\mathtt{H}_k$. Ahora, observemos que el valor asignado a `T` para cada aridad r solo depende de `H_old` y $\mathtt{N}_k$. Luego, $\mathtt{T}^r_{\bar{a},k+1} = \mathtt{T}^r_{\bar{b},k+1}$ para toda aridad r. Esto, junto con el hecho de que los símbolos de función en el bucle for que comienza en la Línea 12 aparecen siempre en el mismo orden, implica que $\mathtt{V}'_{\bar{a},k+1} = \mathtt{V}'_{\bar{b},k+1}$. Probaremos ahora que $\mathtt{P}_{\bar{a},k+1} = \mathtt{P}_{\bar{b},k+1}$. Supongamos que los índices i y j están en el mismo bloque de $\mathtt{P}_{\bar{a},k+1}$. Esto es, $\mathtt{V}_{k+1}(\bar{a})[i] = \mathtt{V}_{k+1}(\bar{a})[j]$, por lo tanto, usando (2), tenemos que $\mathtt{V}'_{\bar{a},k+1}[i]^{\mathbf{A}}(\bar{a}) = \mathtt{V}'_{\bar{a},k+1}[j]^{\mathbf{A}}(\bar{a})$. Ahora, (1) nos dice que $\mathtt{V}'_{k+1}(\bar{a})[i]$ y $\mathtt{V}'_{k+1}(\bar{a})[j]$ son términos de grado como máximo $k+1$, y que coinciden en $\bar{a}$. Por lo tanto, coinciden en $\bar{b}$, ya que $\bar{a} \approx_{k+1} \bar{b}$. Invocando (2) otra vez, nos dice que i y j están en el mismo bloque de $\mathtt{P}_{\bar{b},k+1}$. A partir del hecho de que $\mathtt{P}_{\bar{a},k+1} = \mathtt{P}_{\bar{b},k+1}$, es fácil de ver que $\mathtt{H}_{\bar{a},k+1} = \mathtt{H}_{\bar{b},k+1}$ y $\mathtt{N}_{\bar{a},k+1} = \mathtt{N}_{\bar{b},k+1}$. En particular, para cualquier $l \leq \text{mín}(\mathrm{K}_{\bar{a}}, \mathrm{K}_{\bar{b}})$ tenemos que $\mathtt{N}_{\bar{a},l} = \emptyset$ si y solo si $\mathtt{N}_{\bar{b},l} = \emptyset$. Por lo tanto, $\mathrm{K}_{\bar{a}} = \mathrm{K}_{\bar{b}}$, de lo que se sigue de inmediato que (3) vale para $k + 1 > \text{mín}(\mathrm{K}_{\bar{a}}, \mathrm{K}_{\bar{b}})$.

(4) Tomemos $t = f(t_0, \ldots, t_{r-1}) \in \mathcal{T}_{k+1}$ y supongamos $k + 1 \leq \mathrm{K}_{\bar{a}}$. Para $k = 0$, definimos $F_0(x_j) = j$. Si $k \geq 1$, por hipótesis inductiva tenemos la función F_k, y así podemos obtener los índices $i_0 := F_k(t_0, \rho_{k-1}), \ldots, i_{r-1} := F_k(t_{r-1}, \rho_{k-1})$ funcionalmente de t y ρ_k (obviamente, $t_0, \ldots, t_{r-1}$ pueden ser obtenidos como funciones de t y $\rho_{k-1} = \rho_k \cap \mathcal{T}^2_{k-1}$). Si $k = 0$, solo tomamos $i_0 := F_0(t_0), \ldots, i_{r-1} := F_0(t_{r-1})$. Luego, definimos

$$\begin{aligned} \tilde{t}_0 &:= \mathtt{V}'_k[i_0] \\ &\vdots \\ \tilde{t}_{r-1} &:= \mathtt{V}'_k[i_{r-1}], \end{aligned}$$

y, otra vez, notamos que $\tilde{t}_l$ puede ser obtenido funcionalmente ya que $\mathtt{V}'_k$ puede ser obtenido funcionalmente desde ρ_k. Además, por nuestra hipótesis inductiva, tenemos que $t^{\mathbf{A}}(\bar{a}) = f(\tilde{t}_0, \ldots, \tilde{t}_{r-1})^{\mathbf{A}}(\bar{a})$.

Suponemos primero que hay $l \in \{0, \ldots, r-1\}$ tal que $i_l \in \mathtt{H}_k \setminus \mathtt{H}_{k-1}$. Entonces $i_l \in \mathtt{N}_k$, luego $\langle i_0, \ldots, i_{r-1} \rangle \in \mathtt{T}^r_{k+1}$, y tenemos que el término $f(\tilde{t}_0, \ldots, \tilde{t}_{r-1})$ es añadido a $\mathtt{V}'$, con índice j, durante la $(k+1)$-ésimo pasada del bucle while. Observemos que j solo depende de la signatura de $\mathbf{A}$ y de los números $\left|\mathtt{T}^u_{k+1}\right|$ para u una aridad menor o igual que r. Además $\mathtt{T}^u_{k+1}$ depende solo de $\mathtt{H}_k$ y $\mathtt{N}_k$, que a su vez están determinados por ρ_k. Así, j puede ser obtenido como una función de t y ρ_k, y podemos definir $F_{k+1}(t, \rho_k) := j$, en el caso de que haya $l \in \{0, \ldots, r-1\}$ tal que $i_l \in \mathtt{H}_k \setminus \mathtt{H}_{k-1}$. Supongamos, por otro lado, que $i_0, \ldots, i_{r-1} \in \mathtt{H}_{k-1}$. Entonces, por (1), tenemos que $\tilde{t}_0, \ldots, \tilde{t}_{r-1} \in \mathcal{T}_{k-1}$ y, así, $f(\tilde{t}_0, \ldots, \tilde{t}_{r-1}) \in \mathcal{T}_k$. Por lo tanto, en este caso podemos definir $F_{k+1}(t, \rho_k) := F_k(f(\tilde{t}_0, \ldots, \tilde{t}_{r-1}), \rho_{k-1})$.

(5) Notar primero que, para $j \leq k$, tenemos que $\mathtt{P}_j$ es la restricción de $\mathtt{P}_k$ al conjunto $\{0, ..., \left|\mathtt{V}_j\right| - 1\}$. Además, $\mathtt{H}_j = [\text{mín}(B) : B \in \mathtt{P}_j]$, por lo que $\mathtt{P}_j$ determina $\mathtt{H}_j$. Por otro lado, $\mathtt{N}_0$ es definido como $\mathtt{H}_0$, mientras $\mathtt{N}_{j+1} = \mathtt{H}_{j+1} \setminus \mathtt{H}_j$. Sumado a esto, observemos que $\mathtt{V}'_0$ está también determinado solo por la longitud de $\bar{a}$, y que $\mathtt{V}'_{j+1}$ es obtenido a partir de $\mathtt{H}_j$ y $\mathtt{N}_j$, por lo que es determinado por $\mathtt{H}_j$ y $\mathtt{H}_{j-1}$. Utilizando estas observaciones, la conclusión se deriva fácilmente mediante un argumento inductivo.

(6) Sea $K = \mathrm{K}_{\bar{a}}$. Por (5), para cada $k \leq K$ tenemos que $\mathtt{P}_{\bar{a},k} = \mathtt{P}_{\bar{b},k}$ y $\mathtt{V}'_{\bar{a},k} = \mathtt{V}'_{\bar{b},k}$. Sean $t, s \in \mathcal{T}_K$ tales que $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$. Definimos $i = F_K(t, \rho_{\bar{a},K-1})$ y $j = F_K(s, \rho_{\bar{a},K-1})$. Definimos, además, $\hat{t} = \mathtt{V}'_{\bar{a},k}[i]$ y $\hat{s} = \mathtt{V}'_{\bar{a},k}[j]$. Esto significa que $\hat{t}^{\mathbf{A}}(\bar{a}) = \hat{s}^{\mathbf{A}}(\bar{a})$ y, como $\mathtt{V}'_{\bar{a},K} = \mathtt{V}'_{\bar{b},K}$, tenemos que $\hat{t}^{\mathbf{A}}(\bar{b}) = \hat{s}^{\mathbf{A}}(\bar{b})$, lo que a su vez implica que $t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b})$. Esto muestra que $\bar{a} \approx_K \bar{b}$, por lo tanto, invocando el Lema 30.(2) tenemos que $\bar{a} \approx \bar{b}$. □

Dada un álgebra finita $\mathbf{A}$ y $\bar{a} \in A^n$, definimos $\mathrm{iType}(\bar{a})$ e $\mathrm{iUniv}(\bar{a})$ tales que $\mathrm{isoType}(\mathbf{A}, \bar{a}) = (\mathrm{iType}(\bar{a}), \mathrm{iUniv}(\bar{a}))$. Notar que $\mathrm{iUniv}(\bar{a})$ es una lista sin repeticiones de todos los elementos en $\mathrm{Sg}(\bar{a})$ en el orden en que fueron generados.

El siguiente corolario resume los resultados reflejados en el Lema 31 usando la terminología recién introducida. Esto muestra que la salida de nuestro algoritmo captura el tipo de isomorfismo de una tupla $\bar{a}$ en una estructura algebraica finita.

Corolario 32. *Sea* $\mathbf{A}$ *un álgebra finita. Entonces para cualesquiera* $\bar{a}, \bar{b} \in A^n$*, los siguientes son equivalentes:*

1. $\bar{a} \approx \bar{b}$,
2. $\mathrm{iType}(\bar{a}) = \mathrm{iType}(\bar{b})$.

Además, si se da cualquiera de estas condiciones, entonces el mapeo $\gamma : \mathrm{iUniv}(\bar{a})[j] \mapsto \mathrm{iUniv}(\bar{b})[j]$ *para* $j \in \{0, ..., |\mathrm{iUniv}(\bar{a})| - 1\}$ *es un isomorfismo de* $\mathbf{Sg}(\bar{a})$ *a* $\mathbf{Sg}(\bar{b})$ *que mapea* $\bar{a}$ *a* $\bar{b}$*.*

Demostración. Si $\bar{a} \approx \bar{b}$, a la luz del Lema 31.(3), tenemos que $\mathrm{iType}(\bar{a}) = \mathrm{iType}(\bar{b})$. A la inversa, si $\mathrm{iType}(\bar{a}) = \mathrm{iType}(\bar{b})$, entonces, por el Lema (31).(6), tenemos

que $\bar{a} \approx \bar{b}$. En cualquier caso, por el Lema (31).(3), obtenemos $\mathtt{X}_{\bar{a}} = \mathtt{X}_{\bar{b}}$ para $\mathtt{X} \in \{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$. En particular, $\mathtt{H}_{\bar{a}} = \mathtt{H}_{\bar{b}}$, de lo cual se sigue que $|\mathrm{iUniv}(\bar{a})| = |\mathrm{iUniv}(\bar{b})|$. Como todos los elementos en $\mathrm{iUniv}(\bar{a})$ (y también en $\mathrm{iUniv}(\bar{b})$) son diferentes, tenemos que γ es una biyección bien definida. Además, como $\mathtt{P}_{\bar{a},0} = \mathtt{P}_{\bar{b},0}$, tenemos que γ mapea cada a_j a b_j. Para probar que γ es un homomorfismo, sea t un término. Sea $i \in \mathtt{H}_{\bar{a}}$ tal que $t^{\mathbf{A}}(\bar{a}) = \mathtt{V}_{\bar{a}}[i]$, y definimos que $\hat{t} = \mathtt{V}'_{\bar{a}}[i]$. Por (2), t y $\hat{t}$ coinciden en $\bar{a}$, por lo que también coinciden en $\bar{b}$. Por lo que tenemos $\gamma(t^{\mathbf{A}}(\bar{a})) = \gamma(\hat{t}^{\mathbf{A}}(\bar{a})) = \mathtt{V}_{\bar{b}}[i] = \mathtt{V}'_{\bar{b}}[i]^{\mathbf{A}}(\bar{b})$ y, como $\mathtt{V}'_{\bar{a}} = \mathtt{V}'_{\bar{b}}$, esto equivale a $\hat{t}^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\gamma(\bar{a}))$. □

5.2 COMPUTANDO OPENDEFALG CON UNA ESTRATEGIA MERGING

En esta sección, presentamos nuestro algoritmo para decidir la definibilidad abierta en una estructura algebraica finita utilizando una estrategia de merging. El algoritmo está basado en una pequeña mejora del Teorema 12, que presentamos a continuación como Corolario 33 y se encuentra publicado en [4]. Antes de ejecutar el algoritmo, es conveniente preprocesar el input como explicamos en la siguiente subsección.

5.3 PREPROCESAMIENTO DE LA RELACIÓN TARGET

Sea $\bar{a} = (a_1, \ldots, a_k)$ una tupla. Definimos el *patrón* de $\bar{a}$, denotado por $\mathrm{pattern}\,\bar{a}$, como la partición de $\{1, \ldots, n\}$ tal que i, j están el mismo bloque si y solo si $a_i = a_j$. Por ejemplo, $\mathrm{pattern}(a, a, b, c, b, c) = \{\{1, 2\}, \{3, 5\}, \{4, 6\}\}$. Escribiremos $\lfloor \bar{a} \rfloor$ para denotar la tupla obtenida tomando $\bar{a}$ y eliminando cada entrada igual a una anterior. E.g., $\lfloor (a, a, b, c, b, c) \rfloor = (a, b, c)$. Sea R una relación y θ un patrón, definimos:

- $\lfloor R \rfloor := \{\lfloor \bar{a} \rfloor : \bar{a} \in R\}$,
- $\mathrm{spec}(R) := \{|\lfloor \bar{a} \rfloor| : \bar{a} \in R\}$,
- $R_\theta := \{\bar{a} \in R : \mathrm{pattern}\,\bar{a} = \theta\}$.

Dada una tupla $\bar{a}$ de A y relaciones $R_1, \ldots, R_l$ sobre A, sea $\mathrm{relType}(\bar{a}, R_1, \ldots, R_l) \in \{\mathrm{True}, \mathrm{False}\}^l$ la tupla $(R_1(\bar{a}), \ldots, R_l(\bar{a}))$. Para un entero positivo k, definimos $A^{(k)} := \{(a_1, \ldots, a_k) \in A^k : a_i = a_j \Leftrightarrow i = j\}$.

Corolario 33. *Sea* $\mathbf{A}$ *un álgebra finita,* $R \subseteq A^m$ *y sean* $R_1, \ldots, R_l$ *todas las relaciones de la forma* $\lfloor R_{\mathrm{pattern}\,\bar{a}} \rfloor$ *para* $\bar{a} \in R$*. Los siguientes son equivalentes:*

1. *R es abierta definible en $\mathbf{A}$.*
2. *$R_1, \ldots, R_l$ son abiertas definibles en $\mathbf{A}$.*
3. *Para todo $k \in \mathrm{spec}(R)$ y para todo $\bar{a}, \bar{b} \in A^{(k)}$ tenemos que $\bar{a} \approx \bar{b}$, implica que $\mathrm{relType}(\bar{a}, R_1, \ldots, R_l) = \mathrm{relType}(\bar{b}, R_1, \ldots, R_l)$.*

Demostración. La equivalencia entre (i)⇔(ii) es un ejercicio fácil, y (ii)⇔(iii) es solo una reafirmación del Teorema 12. □

5.3.1 *Análisis detallado del Algoritmo 5.2*

Sea $\mathbf{A}$ un álgebra finita y sea $R \subseteq A^n$. Nuestra estrategia para decidir si R es abierta definible en $\mathbf{A}$ se puede resumir como sigue.

1. Computar todas las relaciones de la forma $\lfloor R_{\text{pattern}\,\bar{a}} \rfloor$ para cada $\bar{a} \in R$. Supongamos que son $R_1, \dots, R_l$.
2. Para cada $k \in \text{spec}(R)$, computar la partición inducida por la relación de equivalencia $\approx$ en $A^{(k)}$.
3. Si para algún $k \in \text{spec}(R)$ hay $\bar{a}, \bar{b} \in A^{(k)}$ tales que $\text{relType}(\bar{a}, R_1, \dots, R_l) \neq \text{relType}(\bar{b}, R_1, \dots, R_l)$ y $\bar{a} \approx \bar{b}$, retornamos False. En otro caso, devolvemos True.

El primer paso descompone la relación en relaciones sin información superflua. Esto puede implicar una gran mejora en el rendimiento, dado que el espacio de búsqueda depende exponencialmente de la aridad del target.

Un método obvio para llevar a cabo los pasos 2 y 3 es usar la función isoType para etiquetar cada tupla en $\bigcup_{k \in \text{spec}(R)} A^{(k)}$ con su tipo de isomorfismo, y cada vez que dos tuplas tengan el mismo isoType comprobar que tengan el mismo relType respecto a $R_1, \dots, R_l$. Claramente, es conveniente llevar a cabo esta comprobación tan pronto como las tuplas de un mismo tipo de isomorfismo son descubiertas, ya que, si esta comprobación falla, el algoritmo puede terminar (y retornar False). Otra observación es que, cada vez que encontramos dos tuplas $\bar{a}, \bar{b}$ con el mismo tipo de isomorfismo, tenemos acceso (esencialmente sin ningún costo computacional aparte) a un isomorfismo γ entre $\text{Sg}(\bar{a})$ y $\text{Sg}(\bar{b})$. Si dos tuplas son conectadas por γ, tienen el mismo tipo de isomorfismo y, por lo tanto, es suficiente computar el tipo de isomorfismo de una de ellas. Estas observaciones, junto con una estrategia deliberada para recorrer las tuplas, constituyen las ideas principales de nuestro algoritmo. A continuación, explicamos cómo se implementan.

En cada punto durante la ejecución, los datos guardados en $\texttt{orbits}_k$ son una partición de $A^{(k)}$donde cada bloque es etiquetado con alguna información adicional. Llamamos a estos bloques etiquetados *órbitas*, los cuales tienen la forma (B, RT, T, U) donde $B \subseteq A^{(k)}$ es el bloque actual, RT es el relType de todas las tuplas en B, T guarda el iType de las tuplas en B (cuando ya es conocido), y U guarda el iUniv de una tupla en B (cuando es conocido). Cuando el algoritmo comienza, cada bloque es un singulete anotado con su relType, y T,U se establecen en Null. El algoritmo recorre y procesa las tuplas en $\bigcup_{k \in \text{spec}(R)} A^{(k)}$ a la manera de DFS. El *árbol* que es recorrido está compuesto por los subuniversos de $\mathbf{A}$ y tiene a A como su raíz. En cada punto de una ejecución, el algoritmo está trabajando en un nodo S de este árbol procesando las tuplas en $\bigcup_{k \in \text{spec}(R)} S^{(k)}$. Para llevar la cuenta del nodo actual y los previamente visitados, y también las tuplas ya procesadas en cada nodo, se utiliza una pila. Las entradas en la pila tienen la forma (S, L, G), donde S es un subuniverso, L es la lista de tuplas de S que todavía quedan por procesar, y $G \subseteq \bigcup_{k \in \text{spec}(R)} S^{(k)}$ es un conjunto de tuplas tal que cada tupla en G genera S y no hay dos tuplas en G que tengan el mismo tipo de isomorfismo. (El punto de llevar

el conjunto G se volverá claro después.) La pila es inicializada con $(A, L_0, \emptyset)$, donde L_0 es una lista de tuplas en $\bigcup_{k \in \text{spec}(R)} A^{(k)}$ ordenadas crecientemente por la aridad. La primera tupla procesada durante una ejecución es el primer elemento en L_0, y cada vez que el algoritmo está listo para procesar una nueva tupla, hay un triplete (S, L, G) en el tope de la pila y el primer elemento de L es poppeado y procesado. Para ver cómo se procesa una tupla, supongamos que en el tope de la pila está (S, L, G) y una tupla $\bar{a}$ ha sido poppeada de L. Si el tipo de isomorfismo de $\bar{a}$ es conocido (i.e., si $\bar{a}$ pertenece a una órbita con un tipo conocido), poppearemos la tupla siguiente en L (si no hay más tuplas en L, la entrada (S, L, G) es removida de la pila). Si, por otro lado, el tipo de $\bar{a}$ es desconocido, la función isoType es llamada para computarlo, y el paso siguiente es buscar a través de las órbitas para ver si alguna está etiquetada con $\text{iType}(\bar{a})$. Sin embargo, qué órbitas inspeccionamos (y el comportamiento del algoritmo después) depende de si $\text{Sg}(\bar{a})$ es más pequeño que S.

Caso $|\text{Sg}(\bar{a})| = |S|$. Aquí solo inspeccionamos las órbitas de las tuplas en G en búsqueda de una etiquetada con $\text{iType}(\bar{a})$ (para ver por qué esto es suficiente ver el Lema 35). Si ninguna de las órbitas está etiquetada con $\text{iType}(\bar{a})$, llamamos a la función `tag_orbit` para etiquetar la órbita de $\bar{a}$ con $\text{iType}(\bar{a})$ y $\text{iUniv}(\bar{a})$, y agregamos $\bar{a}$ a G. Si, en cambio, hay una órbita (B, RT, T, U) de un elemento en G con $T = \text{iType}(\bar{a})$, entonces, por el Corolario 32, la función $\gamma : \text{iUniv}(\bar{a})[i] \mapsto U[i]$ es un subisomorfismo de $\mathbf{A}$. Luego, procedemos a fusionar las órbitas conectadas por γ. En particular, la órbita que contiene a $\bar{a}$ es fusionada con (B, RT, T, U) y, así, $\bar{a}$ termina en una órbita etiquetada con $\text{iType}(\bar{a})$. La fusión se hace mediante la función `try_merge_orbits` (ver Algoritmo 5.3), la cual también comprueba que, cuando dos órbitas deban ser fusionadas, tengan el mismo relType (si esta comprobación falla, el Algoritmo 4.1 para y retorna False).

Caso $|\text{Sg}(\bar{a})| < |S|$. En este caso inspeccionamos todas las órbitas en $\texttt{orbits}_{|\bar{a}|}$ en la búsqueda de una con tipo de isomorfismo $\text{iType}(\bar{a})$. Si no existe tal órbita, entonces la órbita que contiene a $\bar{a}$ es etiquetada con $\text{iType}(\bar{a})$ y $\text{iUniv}(\bar{a})$. Además ponemos $(\text{Sg}(\bar{a}), L', \{\bar{a}\})$ en el tope de la pila (i.e. nos movemos a un nuevo nodo del árbol de subuniversos). Si hay una órbita en $\texttt{orbits}_{|\bar{a}|}$ etiquetada con $\text{iType}(\bar{a})$, procedemos de la misma manera que antes, llamando a `try_merge_orbits`.

Eventualmente, si ninguna llamada a `try_merge_orbits` retorna False, cada tupla en $\bigcup_{k \in \text{spec}(R)} A^{(k)}$ es procesada y, cuando el algoritmo finaliza, tenemos computada la partición inducida por $\approx$ en $A^{(k)}$ para cada $k \in \text{spec}(R)$. Como nos aseguramos de que todas las tuplas en la misma órbita tienen el mismo relType, el Corolario 33 garantiza que R es abierta definible en $\mathbf{A}$.

5.3.2 *Prueba de correctitud y completitud*

Para probar correctitud y completitud necesitamos algunos lemas auxiliares.

Lema 34. *Las siguientes propiedades se cumplen durante una ejecución del Algoritmo 4.1 para cada $k \in \text{spec}(R)$:*

1. *El conjunto $\{B : (B, RT, T, U) \in \texttt{orbits}_k\}$ es una partición de $A^{(k)}$.*

Algoritmo 5.2 Algoritmo de definibilidad abierta algebraica

function `OpenDefAlg`($\mathbf{A}$, R)
▷ $\mathbf{A}$ es un álgebra y R es una relación sobre A
    `targets` $= (R_1, \ldots, R_l)$ donde cada R_j son todas las relaciones distintas de la forma $\lfloor R_{\text{pattern}\,\bar{a}} \rfloor$ para $\bar{a} \in R$
    `spec` $=$ `sorted`(spec(R)) ▷ ordenadas crecientemente
    for $k \in$ `spec` **do**
        $\texttt{orbits}_k = \{(\{\bar{a}\}, \text{relType}(\bar{a}, \texttt{targets}), \texttt{Null}, \texttt{Null}) : \bar{a} \in A^{(k)}\}$ ▷ inicialización de las órbitas
    `orbits` $= \{\texttt{orbits}_k : k \in \texttt{spec}\}$
    `tuples_to_process` $=$ `sorted`$(\bigcup_{k \in \texttt{spec}} A^{(k)})$ ▷ ordenadas decrecientemente por aridad
    `stack` $= [(A, \texttt{tuples\_to\_process}, \emptyset)]$ ▷ inicialización de la pila
    while `stack` no sea vacía **do**
        Sea (`current_sub`, `tuples_to_process`, `generators`) la referencia al tope de la pila
        while `tuples_to_process` $\neq []$ **do**
            $\bar{a}$ primer elemento poppeado de `tuples_to_process`
            if `Type`$(\bar{a}) =$ `Null` **then** ▷ el tipo de $\bar{a}$ es conocido
                (`type_a`, `universe_a`) $=$ `IsoType`$(\boldsymbol{A}, \bar{a})$
                if $|$`universe_a`$| =$ $|$`current_sub`$|$ **then** ▷ Sg($\bar{a}$) no es más chico
                    if hay (`B`, `RT`, `T`, `U`) $\in$ {órbitas de las tuplas en `generators`} tal que `type_a` $=$ `T` **then**
                        $\gamma =$ el isomorfismo de `universe_a` a `U`
                        if not `try_merge_orbits`$(\gamma,$ `orbits`$)$ **then** ▷ ver Algoritmo 5.3
                            return `False`
                    else ▷ el tipo de $\bar{a}$ es nuevo
                        `tag_orbit`$(\bar{a},$ `type_a`, `universe_a`$)$
                        agregar $\bar{a}$ a `generators`
                else ▷ Sg($\bar{a}$) es más chico
                    if hay (`B`, `RT`, `T`, `U`) $\in$ `orbits`$_{|\bar{a}|}$ tal que `type_a` $=$ `T` **then** ▷ el tipo de $\bar{a}$ no es nuevo
                        $\gamma =$ isomorfismo de `universe_a` a `U`
                        if not `try_merge_orbits`$(\gamma,$ `orbits`$)$ **then** ▷ ver Algoritmo 5.3
                            return `False`
                    else ▷ el tipo de $\bar{a}$ es nuevo
                        `tag_orbit`$(\bar{a},$ `type_a`, `universe_a`$)$
                        `new_current_sub` $=$ `universe_a`
                        `new_tuples_to_process` $=$ `sorted`$(\bigcup_{k \in \texttt{spec}} \texttt{universe\_a}^{(k)})$
                        `new_generators` $= \{\bar{a}\}$
                        ▷ agregamos un nuevo nodo al tope de la pila
                        (`new_current_sub`, `new_tuples_to_process`, `new_generators`) se agrega en la pila
                        `tag_orbit`$(\bar{a},$ `type`, `universe`$)$
                        break
        if `tuples_to_process` $= []$ **then**
            eliminar el tope de la pila
    return `True`

Algoritmo 5.3 Fusionando órbitas

```
1: function try_merge_orbits(γ, orbits)
2:     for k ∈ spec do
3:         for ā ∈ (Dom(γ))^(k) do
4:             if orbit(ā) ≠ orbit(γ(ā)) then
5:                 (B, RT, T, U) = orbit(ā)
6:                 (B′, RT′, T′, U′) = orbit(γ(ā))
7:                 if RT ≠ RT′ then
8:                     return False                          ▷ los RelTypes son diferentes
9:                 B_merge = B ∪ B′                          ▷ los bloques son fusionados
10:                RT_merge = RT
11:                if T ≠ Null then                          ▷ el primer bloque fue etiquetado
12:                    T_merge = T
13:                    U_merge = U
14:                else if T′ ≠ Null then                    ▷ el segundo bloque fue etiquetado
15:                    T_merge = T′
16:                    U_merge = U′
17:                else                                      ▷ no hay bloques etiquetados
18:                    T_merge = Null
19:                    U_merge = Null
20:                eliminar (B, RT, T, U) y (B′, RT′, T′, U′) de orbits_k
21:                agregar (B_merge, RT_merge, T_merge, U_merge) a orbits_k]
22:    return True
```

2. *Para cada* $(B, RT, T, U) \in \texttt{orbits}_k$ *tenemos que:*
   a) *Para todo* $\bar{a}$ *en* B *tenemos que* $\text{relType}(\bar{a}, R_1, \ldots, R_l) = RT$.
   b) *Para todos* $\bar{a}, \bar{b}$ *en* B *tenemos que* $\bar{a} \approx \bar{b}$.
   c) *Si* $U \neq \texttt{Null}$, *entonces hay* $\bar{a}_0 \in B$ *tal que* $\text{iUniv}(\bar{a}_0) = U$.
   d) *Si* $T \neq \texttt{Null}$, *entonces* $T = \text{iType}(\bar{a})$ *para cada* $\bar{a}$ *en* B.

Demostración. Las propiedades son obviamente verdaderas en la inicialización de $\texttt{orbits}_k$. Notar que las únicas instrucciones del Algoritmo 4.1 que pueden cambiar $\texttt{orbits}_k$ son llamadas a las funciones `try_merge_orbits` y `tag_orbit`, y es fácil de ver que estas funciones preservan (1) y (2). □

Para hacer las pruebas que siguen más concisas, diremos que (en un punto dado de una ejecución del Algoritmo 4.1):

- una órbita (B, RT, T, U) está *etiquetada* si $T \neq \texttt{Null}$,
- la tupla $\bar{a}$ tiene *tipo conocido* si la (única) órbita que contiene a $\bar{a}$ está etiquetada.

Lema 35. *Las siguientes propiedades se cumplen durante una ejecución del Algoritmo 4.1:*

1. *Si la pila contiene* $[(S_m, L_m, G_m), (S_{m-1}, L_{m-1}, G_{m-1}), \ldots, (S_0, L_0, G_0)]$, *entonces* $|S_m| < \ldots < |S_0|$.
2. *Si una entrada* (S, L, G) *es eliminada de la pila, entonces cada tupla en* $\bigcup_{k \in \text{spec}(R)} S^{(k)}$ *tiene tipo conocido.*

3. *Supongamos que el tope de la pila es* (S, L, G). *Si* $\bar{a}$ *es una tupla con tipo conocido tal que* $|\mathrm{Sg}(\bar{a})| < |S|$, *entonces cada tupla en* $\bigcup_{k \in \mathrm{spec}(R)} \mathrm{Sg}(\bar{a})^{(k)}$ *tiene tipo conocido.*

4. *Supongamos que el tope de la pila es* (S, L, G). *Si* $\bar{a}$ *es una tupla con tipo conocido tal que* $|\mathrm{Sg}(\bar{a})| = |S|$, *se da que:*

   a) *hay un* $\bar{g} \in G$ *tal que* $\bar{g}$ *está en la misma órbita de* $\bar{a}$, *o*

   b) *cada tupla en* $\bigcup_{k \in \mathrm{spec}(R)} \mathrm{Sg}(\bar{a})^{(k)}$ *tiene tipo conocido.*

5. *Para cada* $k \in \mathrm{spec}(R)$, *y para todo* $O = (B, RT, T, U)$ *y* $O' = (B', RT', T', U')$ *en* `orbits`$_k$ *tal que* $T \neq$ `Null`, *tenemos que*

$$T = T' \iff O = O'.$$

Demostración. Es fácil de ver que el ítem (1) se cumple. Probaremos (2). Cuando un elemento (S, L, G) es agregado al tope de la pila, entonces L es una lista que contiene todas las tuplas en $\bigcup_{k \in \mathrm{spec}(R)} S^{(k)}$, y (S, L, G) es solo eliminada de la pila cuando todas las tuplas en T han sido procesadas. No es difícil de ver que cuando una tupla $\bar{a}$ es procesada, encontramos que:

- la órbita que la contiene ya está etiquetada, o
- es etiquetada por una llamada a `tag_orbit` o `try_merge_orbits`, a menos que la llamada a `try_merge_orbits` retorne False (haciendo que el Algoritmo 4.1 termine y retorne False).

Probemos ahora (3). Supongamos que en el tope de la pila está (S, L, G), y sea $\bar{a}$ como en el enunciado. Notemos que hay dos maneras por las cuales la órbita que contiene $\bar{a}$ pudo haber sido etiquetada: por una llamada a `tag_orbit` o por la fusión de la órbita de $\bar{a}$ con otra que ya estaba etiquetada. Supongamos, primero, que en algún punto del algoritmo se hace la llamada `tag_orbit`$(\bar{a}, \mathrm{iType}(\bar{a}), \mathrm{iUniv}(\bar{a}))$. La observación clave es que entonces $(\mathrm{Sg}(\bar{a}), L', G')$ ha estado en la pila. Ahora, como $|\mathrm{Sg}(\bar{a})| < |S|$, el ítem (1) dice que $(\mathrm{Sg}(\bar{a}), L', G')$ ya no está en la pila y, por (2), se sigue que cada tupla de $\mathrm{Sg}(\bar{a})$ tiene tipo conocido. Luego, supongamos que la órbita de $\bar{a}$ no estaba etiquetada por una llamada a `tag_orbit`.Como la única forma de introducir un nuevo tipo en `orbits` es con una llamada a `tag_orbit`, debe existir una tupla $\bar{b}$ tal que:

- $\mathrm{iType}(\bar{b}) = \mathrm{iType}(\bar{a})$, y `tag_orbit`$(\bar{b}, \mathrm{iType}(\bar{a}), \mathrm{iUniv}(\bar{b}))$ ha sido ejecutado en un tiempo anterior, y
- la órbita de $\bar{a}$ fue eventualmente etiquetada al fusionarse con la órbita de $\bar{b}$.

Usando el mismo razonamiento que antes, podemos concluir que cada tupla de $\mathrm{Sg}(\bar{b})$ tiene tipo conocido. Finalmente, observemos que, durante el proceso de fusión, cada tupla de $\mathrm{Sg}(\bar{a})$ termina en una órbita con una tupla de $\mathrm{Sg}(\bar{b})$ y, por lo tanto, tiene tipo conocido.

Dejamos la prueba de (4) al lector, ya que es similar a (3).

Para concluir probaremos (5). Notar que es suficiente comprobar que las llamadas a `tag_orbit` y `try_merge_orbits` preservan (5). Esto es claro en `try_merge_orbits`, ya que esta función no etiqueta órbitas con nuevos tipos. Por esto, nos enfocaremos en las llamadas a `tag_orbit`; comencemos con la de la línea 34 del Algoritmo 4.1. Notar que, inmediatamente antes de que esta llamada sea ejecutada comprobamos que el tipo a ser introducido no ocurra en ningún lugar de `orbits` y, por lo tanto, (5) es preservado. Finalmente, veamos la llamada a `tag_orbit`$(\bar{a}, \text{iType}(\bar{a}), \text{iUniv}(\bar{a}))$ en la línea 21. Debido a las comprobaciones hechas antes de esta llamada, sabemos que (al momento de esta llamada) el tope de la pila es $(\text{Sg}(\bar{a}), L, G)$ y que no hay una tupla en G con el mismo tipo que $\bar{a}$. Queremos mostrar que no hay ninguna órbita etiquetada con $\text{iType}(\bar{a})$. Para mostrar la contradicción, supongamos que hay $\bar{b}$ tal que su órbita, que llamaremos O', tiene tipo $\text{iType}(\bar{a})$. Por lo tanto, como no hay tupla de G en O', el ítem (4) dice que cada tupla de $\text{Sg}(\bar{b})$ tiene tipo conocido. Se sigue que $\text{Sg}(\bar{b}) \neq A$, ya que de otra forma $\bar{a}$ tendría tipo conocido. Por lo tanto, $(\text{Sg}(\bar{a}), L, G)$ es puesto en el tope de la pila al procesar la tupla $\bar{g}$ que se encontró que tenía un nuevo tipo. Pero esto es una contradicción, ya que $\text{Sg}(\bar{b})$ contiene una copia isomorfa de $\bar{g}$, y habría tenido tipo conocido. □

Proposición 36. *El Algoritmo 4.1 es correcto y completo.*

Demostración. Notar que el algoritmo termina si y solo si se da alguno de los siguientes:

- una llamada a `try_merge_orbits` retorna false y el Algoritmo 4.1 retorna false, o
- la pila está vacía y el Algoritmo 4.1 retorna true.

En el primer caso, es claro de una inspección al Algoritmo 5.3, que `try_merge_orbits` retorna false si y solo si hay dos órbitas etiquetadas con el mismo tipo de isomorfismo y diferentes relTypes. Luego, por el Lema 34, hay tuplas $\bar{a}, \bar{b}$ tales que $\bar{a} \approx \bar{b}$ y $\text{relType}(\bar{a}, R_1, \ldots, R_l) \neq \text{relType}(\bar{b}, R_1, \ldots, R_l)$. Luego, el Corolario 33 dice que R no es abierta definible en $\mathbf{A}$.

Supongamos, a continuación, que el algoritmo termina debido a que la pila queda vacía. Recordemos que la pila es inicializada con $(A, L_0, \emptyset)$; por lo tanto, por el Lema 35.(2), cada tupla en $\bigcup_{k \in \text{spec}(R)} A^{(k)}$ tiene tipo conocido cuando el algoritmo termina. Probaremos que el ítem (3) del Corolario 33 vale. Fijemos $k \in \text{spec}(R)$ y tomemos $\bar{a}, \bar{b} \in A^{(k)}$ tales que $\bar{a} \approx \bar{b}$. Ahora, el Lema 34 junto con el Lema 35.(5) garantizan que $\bar{a}$ y $\bar{b}$ están en la misma órbita y, así, el ítem (2*a*) del Lema 34 dice que $\text{relType}(\bar{a}, R_1, \ldots, R_l) = \text{relType}(\bar{b}, R_1, \ldots, R_l)$. □

5.4 COMPUTANDO OPENDEFALG USANDO UNA ESTRATEGIA DE SPLITTING

Ahora presentamos nuestro segundo algoritmo para decidir la definibilidad abierta, también basado en el Corolario 33. Lo llamamos el Algoritmo de *Splitting* y se encuentra en [5]. Comienza de la misma manera que el Algoritmo de Merging: preprocesando la relación target (ver subsección 5.3), y descomponiéndola en

Algoritmo 5.4 Estrategia de Splitting

```
1: function SplittingAlgorithm(A, target)
2:     formula = ⊥                                  ▷ Inicialización de la fórmula candidata
3:     full_blocks = []
4:     k = ar(target)
5:     tuples = {t for t in A^(k)}                  ▷ Tuplas sin elementos repetidos
6:     terms_to_process = [x_i for i in {0, ..., ar(target) − 1}]
7:     witnesses = []
8:     new_witnesses = []
9:     local_fórmula = ⊤
10:    step = 0
                                  ▷ Ahora creamos la estructura con campos para el Bloque
11:    B_0 = Block(tuples, witnesses, new_witnesses, terms_to_process,
                   local_fórmula, step)
12:    blocks_to_process = [B_0]                    ▷ Primer bloque en la stack
13:    while blocks_to_process ≠ [] do
14:        B = pop(blocks_to_process)
15:        if B.tuples ⊆ target then                ▷ B es un bloque lleno
16:            formula = formula ∨ B.formula
17:            full_blocks.append(B)
18:            continue
19:        else if B.tuples ∩ target = ∅ then       ▷ B es un bloque desechable
20:            continue
21:        else                                     ▷ B es un bloque mixto
22:            if B.terms_to_process == [] and B.new_witnesses == [] then
23:                return (False, B)   ▷ B es un un bloque mixto terminal, luego B es un
   contraejemplo
24:            blocks_to_process = process_mixed_block(B) + blocks_to_process
25:    return (True,formula)
```

varias relaciones target nuevas de la forma $\lfloor R_{\text{pattern}\,\bar{a}} \rfloor$ para alguna $\bar{a}$ en la relación target original (donde sus tuplas no tienen entradas repetidas). El siguiente paso es procesar estos nuevos targets agrupados por aridad. Si alguna de estas partes resulta ser no definible, el algoritmo devuelve `False` (junto con un contraejemplo). Si, por el contrario, todas las partes para todas las aridades son definibles, entonces el algoritmo devuelve `True`, junto con una fórmula definidora.

En aras de la simplicidad, la presentación que sigue describe el algoritmo que procesa una sola parte del target original (en lugar de todas las partes de una misma aridad dada). El pseudocódigo para ello se muestra en el Algoritmo 5.4. El lector no debería tener problemas para entender cómo modificar este algoritmo para obtener uno que funcione en el caso general. Tal vez, la única advertencia es cómo se produce la fórmula final de lo que hablaremos más adelante.

5.4.1 *Análisis detallado del Algoritmo de Splitting*

A diferencia de la estrategia de merging, donde computamos por completo el tipo de isomorfismo de cada tupla al principio y solo después particionamos el conjunto de tuplas correspondientemente, este nuevo algoritmo empieza con el conjunto entero de k-tuplas (las tuplas no tienen elementos repetidos), donde k es la aridad de la relación target. Estas tuplas comienzan formando un mismo bloque, y mientras el algoritmo avanza esta partición es refinada. La idea central es que, en cualquier momento dado, las tuplas en un bloque todavía no han sido descubiertas como no isomorfas. Un paso del algoritmo consiste en evaluar un término en todas las tuplas de un bloque y comparar los valores resultantes con los valores de los términos evaluados previamente. Entonces, las tuplas en el bloque son particionadas en bloques sucesores, de acuerdo a qué valor devolvieron. Para hacer esto, un bloque lleva la información de los términos ya evaluados y la información necesaria para generar los nuevos términos a ser evaluados. Para guardar estos bloques anotados, el algoritmo utiliza el tipo de dato `Block`, que puede ser pensado como un diccionario con los siguientes cinco campos:

- un conjunto `tuples` que contiene las tuplas del bloque,
- tres listas de términos `witnesses`, `new_witnesses`, `terms_to_process` y
- `formula`, que contiene una fórmula abierta.

Las listas de términos `witnesses` y `new_witnesses` tienen un propósito análogo a las variables `H` y `N`, respectivamente, en el Algoritmo 5.1. Los términos a ser evaluados, en el bloque actual y sus sucesores, son generados en un único fragmento por cada profundidad dada d. Para evitar generar términos redundantes, solo construimos términos que incluyan al menos un término de profundidad $d-1$ que produjo un nuevo valor en la última ronda. Estos son guardados en el campo `new_witnesses` cuya única función es evitar términos redundantes. Para entender el rol del campo `witnesses`, veamos qué pasa cuando un bloque B es procesado. Decimos que un bloque es *puro* si está contenido o es disjunto con el target. Como el Algoritmo 5.4 trata esencialmente de encontrar un contraejemplo, o sea, un par de

tuplas isomorfas donde una pertenece al target mientras que la otra no, los bloques puros ya no necesitan ser inspeccionados más en la búsqueda de este contraejemplo. Los bloques que son disjuntos con el target, que llamamos *desechables*, no necesitan ningún procesamiento más. Sin embargo, los bloques contenidos en el target, que llamamos *llenos*, contribuyen con sus fórmulas como una disyunción más a la fórmula del target. Así, el Algoritmo 5.4 (en las líneas 15 a 20) empieza el procesamiento de un bloque revisando si está lleno o es desechable. Ahora, analicemos cómo son procesados los bloques que no son puros, que llamamos *mixtos*. Sea B un bloque mixto. Primero debemos considerar el caso donde B.`terms_to_process` y B.`new_witnesses` sean vacíos. Llamaremos a estos bloques *terminales*. Esto pasa exactamente (Lema 39, (2)) cuando todas las tuplas en el bloque son isomorfas y, como B es mixto, hemos encontrado un contraejemplo y el algoritmo termina (línea 23). Supongamos ahora que B.`terms_to_process` o B.`new_witnesses` son no vacíos. Aquí, el Algoritmo 5.4 llama a la función `process_mixed_block` en B. En su ejecución, esta función primero revisa si hay términos a procesar en B. Si este no es el caso, se genera una nueva lista de términos (de profundidad aumentada) y se asigna a B.`terms_to_process`. Por otro lado, si B.`terms_to_process` es no vacío, tomamos su primer elemento t. Ahora, las tuplas en B son distribuidas entre bloques sucesores de acuerdo a su valor vía t; para cada $s \in B$.`witnesses` se crea un nuevo bloque con todas las tuplas $\bar{a} \in B$ tales que $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$, suponiendo que hay al menos una tupla que cumpla esta condición. En adición, se crea otro bloque más que contenga todas aquellas tuplas cuyo valor vía t no coincide con ninguno de los valores producidos por witnesses. En el pseudocódigo se ve claro cómo se instancian el resto de los campos en los bloques sucesores. La función retorna la lista de bloques sucesores de B que, a su vez, se coloca en la stack de bloques a procesar por el algoritmo principal. Eventualmente, encontramos un bloque mixto terminal o nos quedamos sin bloques para procesar, caso en el que el algoritmo se detiene retornando la fórmula definidora.

5.4.2 *Generación de fórmulas*

Concluimos el análisis del algoritmo de Splitting con una breve explicación de cómo construir la fórmula para el target original en el caso positivo. Recordemos que, para un input $(\mathbf{A}, R)$, comenzamos desarmando R en relaciones de la forma $\lfloor R_\theta \rfloor$ para algún patrón θ presente en R. En el caso de que R sea definible en $\mathbf{A}$, ejecutar el algoritmo de arriba produce una fórmula φ_θ para cada R_θ. Ahora, φ_θ utiliza las variables $x_0, \ldots, x_{|\theta|-1}$. Notar que $|\theta|$ es menor que $k := \mathrm{ar}(R)$ a menos que $\theta = \{\{0\}, \ldots, \{k-1\}\}$; por lo tanto, necesitamos sustituir las variables en φ_θ por las correspondientes variables de $\{x_0, \ldots, x_{k-1}\}$ y agregar las (des)igualdades entre variables que describe el pattern θ. Como ejemplo, supongamos $k = 5$ y $\theta = \{\{0,1\},\{2,3\},\{4\}\}$. En este caso la fórmula resultante es:

$$\tilde{\varphi}_\theta(x_0, x_1, x_2, x_3, x_4) := \varphi_\theta(x_0, x_2, x_4) \wedge x_0 = x_1 \wedge x_2 = x_3 \wedge x_0 \neq x_2 \wedge x_0 \neq x_4 \wedge x_2 \neq x_4.$$

Ahora, si Θ es el conjunto de todos los patrones presentes en R, tenemos que la fórmula abierta que define R en $\mathbf{A}$ es

$$\varphi(x_0, \ldots, x_{k-1}) := \bigvee_{\theta \in \Theta} \tilde{\varphi}_\theta(x_0, \ldots, x_{k-1}),$$

donde $\tilde{\varphi}_\theta$ es la fórmula modificada para φ_θ para cada $\theta \in \Theta$.

Algoritmo 5.5 Procesamiento de un bloque mixto

```
1:  function process_mixed_block(B)
2:      successors = []
3:      if B.terms_to_process == [] then        ▷ Debemos computar un nuevo grupo de términos
4:          Generar términos usando B.witnesses y B.new_witnesses en B.terms_to_process
5:          B.new_witnesses = []
6:      t = pop(B.terms_to_process)
7:      B.step = B.step + 1
8:      complement_block = B.tuples                    ▷ Inicializamos con todas las tuplas
9:      complement_formula = ⊤
10:     for s ∈ B.witnesses do
11:         eq_tuples = [v ∈ B.tuples : t(v) == s(v)]      ▷ Tuplas para un nuevo bloque
12:         if eq_tuples != [] then
13:             successor := Block(eq_tuples, B.witnesses, B.new_witnesses,
                                   B.terms_to_process, B.formula ∧ t = s, B.step)
14:             successors.append(successor)
                                ▷ Borramos las tuplas que aparecen en los nuevos bloques
15:             complement_block = complement_block - eq_tuples
                                                ▷ Agregamos las fórmulas con negaciones
16:             complement_formula = B.formula ∧ complement_formula ∧ (t ≠ s)
17:     if complement_block != [] then
18:         successor = Block(complement_block, B.witnesses + [t],
                              B.new_witnesses + [t], B.terms_to_process,
                              complement_formula, B.step)
19:         successors.append(successor)
20:     if hay una sola entrada en successors then
                                                    ▷ formula no necesita crecer
21:         successors[0].formula = B.formula
22:     return successors
```

Las variables `step` y `full_blocks` no son necesarias para el algoritmo en sí, pero son incluidas para facilitar las demostración que haremos después.

5.4.3 *Correctitud y completitud*

En esta sección demostramos la corrección del algoritmo de splitting. Comenzamos con algunas propiedades preliminares.

Nuestra prueba se basa en la siguiente consecuencia del Teorema 12. Recordemos que, dadas $\bar{a}, \bar{b} \in A^k$, diremos que $\bar{a} \approx \bar{b}$ cuando $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a}) \iff t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b})$ para todo par de términos t, s.

Corolario 37. *Sea* $\mathbf{A}$ *un álgebra finita y* R *una relación k-aria sobre* $\mathbf{A}$*. Entonces son equivalentes:*

R es no definible en $\mathbf{A}$.

Hay tuplas $\bar{a}, \bar{b} \in \mathbf{A}^k$ tales que $\bar{a} \in R$, $\bar{b} \notin R$ y $\bar{a} \approx \bar{b}$.

Dado un término $t(x_0, \ldots, x_{k-1})$, definimos $\mathrm{depth}(t)$ como el menor número natural i tal que $t \in \mathcal{T}_i(x_0, \ldots, x_{k-1})$. Además, para un bloque B, definimos $\mathrm{depth}(B) = \max\{\mathrm{depth}(t) : t \in B.\texttt{witnesses} +\!\!+ B.\texttt{terms\_to\_process}\}$, donde $+\!\!+$ es el operador de concatenación de listas. Recordemos que, dada una fórmula φ y un modelo $\mathbf{A}$, la extensión de φ en $\mathbf{A}$ es denotada por $[\varphi]^{\mathbf{A}}$.

Lema 38. *Sea* $\mathbf{A}$ *un álgebra finita y* R *una relación k-aria sobre* $\mathbf{A}$*. Entonces, los siguientes son invariantes durante la ejecución del Algoritmo 5.4 con la entrada* $(\mathbf{A}, R)$*:*

1. *Para todo bloque B, $t, s \in B.\texttt{witnesses}$ y $\bar{a} \in B.\texttt{tuples}$, tenemos que $t(\bar{a}) = s(\bar{a})$ implica $t = s$.*

2. *Los conjuntos de tuplas de los bloques en* `blocks_to_process` $+\!\!+$ `full_blocks` *son disjuntos por pares y su unión contiene a R.*

3. *Para todo bloque B, tenemos que $[B.\texttt{formula}]^{\mathbf{A}} = B.\texttt{tuples}$.*

4. *Para todo bloque B y para cada término t en $\mathcal{T}_d(x_0, \ldots, x_{k-1})$, con $d = \mathrm{depth}(B)$, hay un término $\hat{t}$ en $B.\texttt{witnesses} +\!\!+ B.\texttt{terms\_to\_process}$ tal que $t(\bar{a}) = \hat{t}(\bar{a})$ para cada $\bar{a}$ en $B.\texttt{tuples}$.*

5. *Para todo bloque B y cada término t en $\mathcal{T}_d(x_0, \ldots, x_{k-1}) \setminus \mathcal{T}_{d-1}(x_0, \ldots, x_{k-1})$ con $d = \mathrm{depth}(B)$, hay un término $\hat{t}$ en $B.\texttt{new\_witnesses} +\!\!+ B.\texttt{terms\_to\_process}$ tal que $t(\bar{a}) = \hat{t}(\bar{a})$ para cada $\bar{a}$ en $B.\texttt{tuples}$.*

6. *Para todos $\bar{a}, \bar{b} \in A^{(k)}$ tales que $\bar{a} \approx \bar{b}$, hay un bloque B tal que $\bar{a}, \bar{b} \in B.\texttt{tuples}$.*

Demostración. (1) Esto se sigue de la simple observación en el pseudocódigo de que un término poppeado t es agregado a la lista de términos de un bloque B en construcción solo si $t(\bar{a}) \neq s(\bar{a})$ para todo $s \in B.\texttt{witnesses}$, $\bar{a} \in B.\texttt{tuples}$.

(2) Observar que `process_mixed_block` no particiona el bloque cuando es movido desde `blocks_to_process` a `full_blocks` (Algoritmo 5.4, líneas 15 a 18) o cuando $\mathrm{depth}(B)$ es aumentada (Algoritmo 5.5, líneas 3 a 5), por lo que solo necesitamos probar el invariante cuando `step` es incrementada (Algoritmo 5.5, líneas 7 a 22). Como solo los bloques desechables son eliminados, R siempre está contenida en la unión de los bloques en `blocks_to_process` $+\!\!+$ `full_blocks`; desde la línea 15 en el Algoritmo 5.5, es directo que estos bloques siguen siendo disjuntos de a pares.

(3) La afirmación es claramente válida para el bloque inicial y en los siguientes casos:

- cuando un bloque lleno es movido desde `blocks_to_process` a `full_blocks` (Algoritmo 5.4, líneas 15 a 18),

- cuando un bloque desechable es eliminado de `blocks_to_process` (Algoritmo 5.4, líneas 20 a 20),

- cuando depth(B) es incrementado (Algoritmo 5.5, línea 4),
- después de que un término t es tomado de `terms_to_process`, incrementando `step` pero dejando B sin particionar (si se da la condición en el Algoritmo 5.5, línea 20).

Consideremos ahora el caso donde el bloque B es particionado después de tomar el término t (Algoritmo 5.5, líneas 10 a 19). Supongamos inductivamente que $[B.\texttt{formula}]^A = B.\texttt{tuples}$. Por construcción, en cada uno de los nuevos bloques la afirmación se convierte en $[(B.\texttt{formula}) \wedge (t = s)]^A = [t = s]^A \cap B.\texttt{tuples}$ para algún s en $B.\texttt{witnesses}$, que se deduce inmediatamente de la hipótesis inductiva.

(4) y (5) Haremos inducción en d. En el caso $d = 0$ y $\texttt{step} = 0$ tenemos que $B.\texttt{witnesses} = [\,]$ y $B.\texttt{terms\_to\_process} = [x_0, \ldots, x_{k-1}]$. Notar que podemos suponer que no hay símbolos de constantes en el lenguaje, ya que pueden ser sustituidos por funciones constantes de aridad 0, lo que significa que la afirmación es verdadera en este caso.

Supongamos, ahora, que $d > 0$ y $\texttt{step} > 0$. Notemos que ninguna de las instrucciones después de incrementar `step` rompe los invariantes: los bloques que surgen del proceso siguiente a tomar un término de $B.\texttt{terms\_to\_process}$ heredan el contenido de $B.\texttt{witnesses}$ y $B.\texttt{terms\_to\_process}$ con, quizá, la excepción del término t si es que coincide con algún otro término en todas las tuplas del bloque que se va a crear.

Así, podemos suponer $\texttt{step} = 0$. En este punto, el algoritmo alcanzó el estado actual tras construir una secuencia de términos en $B.\texttt{terms\_to\_process}$ desde un bloque mixto previo donde `terms_to_process` fue vaciado, incrementando d en 1. Sea $t = f(t_0, ..., t_{r-1}) \in \mathcal{T}_{d+1}$. Por hipótesis inductiva, hay términos $\hat{t}_0, ..., \hat{t}_{r-1} \in \mathcal{T}_d$, que satisfacen $t_i^{\mathbf{A}}(\bar{a}) = \hat{t}_i{}^{\mathbf{A}}(\bar{a})$ y podemos suponer que hay un i_0 tal que $t_{i_0} \in \mathcal{T}_d(x_0, \ldots, x_{k-1}) \setminus \mathcal{T}_{d-1}(x_0, \ldots, x_{k-1})$. Pero esto significa que $\hat{t} = f(\hat{t}_0, ..., \hat{t}_{r-1}) \in B.\texttt{terms\_to\_process}$.

(6) Al comenzar el algoritmo, cada par de tuplas está en el bloque original que contiene todas las tuplas; cuando un bloque es particionado, claramente las líneas 10 a 19 colocan el par en el mismo bloque. □

Lema 39. *Sea B un bloque con $B.\texttt{new\_witnesses} = B.\texttt{terms\_to\_process} = [\,]$. Entonces:*

1. *Para todo término t hay un término $\hat{t} \in B.\texttt{witnesses}$ tal que $t(\bar{a}) = \hat{t}(\bar{a})$ para cada $\bar{a} \in B.\texttt{tuples}$.*
2. *Si $\bar{a}, \bar{b} \in B.\texttt{tuples}$ y t, s son términos, entonces $t(\bar{a}) = s(\bar{a})$ implica $t(\bar{b}) = s(\bar{b})$.*

Demostración. (1) Definamos $d = \text{depth}(B)$ y sea $t \in \mathcal{T}_d$ un término. Si $\hat{t}$ es el término dado por (4), entonces de (5), junto con la hipótesis, se sigue que $\hat{t} \in \mathcal{T}_{d-1}$. El resultado, ahora, se sigue de esta observación con un simple argumento inductivo.

(2) Sean $\bar{a}, \bar{b} \in B.\texttt{tuples}$. Para todo término t, s, sean $\hat{t}, \hat{s}$ los términos dados por (1). Entonces $\hat{t}(\bar{a}) = \hat{s}(\bar{a})$, lo que significa que por el Lema 38 (1) se da que $\hat{t} = \hat{s}$, luego $t(\bar{b}) = \hat{t}(\bar{b}) = \hat{s}(\bar{b}) = s(\bar{b})$. □

Proposición 40. *El Algoritmo 5.4 es correcto y completo. Más precisamente, con la entrada* $(\mathbf{A}, R)$*, el Algoritmo 5.4 eventualmente se detiene y devuelve ya sea una fórmula* φ *que satisface* $R = [\varphi]^{\mathbf{A}}$*, o un contraejemplo, si* R *no es abierta definible.*

Demostración. Consideremos, primero, el caso en que R es abierta definible. Esto significa que nunca un bloque mixto es terminal, lo que implica que cada bloque mixto, eventualmente, se particiona en bloques llenos o desechables. A partir de que los bloques son removidos de `blocks_to_process` y A es finita, se puede garantizar terminación. Observemos, además, que `full_blocks` solo contiene bloques llenos. Como el algoritmo se detiene vaciando `blocks_to_process`, el conjunto dado por la unión descrita en (2) del Lema (38) coincide exactamente con R y el algoritmo devuelve una fórmula φ, la cual, por construcción, es escrita en forma normal disyuntiva. Cada uno de sus disyuntos corresponde a un bloque lleno puro añadido a `full_blocks` y, por (3) en el Lema (38), las tuplas en cada uno de esos bloques forman la extensión del correspondiente disyunto. Por las consideraciones de arriba tenemos que $R = [\varphi]^{\mathbf{A}}$.

Si el algoritmo no retorna una fórmula φ, significa que la terminación se dio a partir de encontrar un contraejemplo; por el Corolario 37, esto sucede exactamente cuando R no es abierta definible. □

Parte V

IMPLEMENTACIONES

6 UTILIZACIÓN DE LAS HERRAMIENTAS

En esta sección presentamos nuestras implementaciones de los algoritmos y facilitamos una versión para ser ejecutada desde la nube.

6.1 INSTALACIÓN DE LAS HERRAMIENTAS

El código fuente está disponible en https://github.com/pablogventura/OpenDef para el Algoritmo 4.1, para estructuras relacionales. La implementación del Algoritmo 5.2 que hace uso de la estrategia de merging para estructuras algebraicas se encuentra en https://github.com/pablogventura/OpenDefAlgMerging mientras que para para el Algoritmo 5.4, que hace uso de la estrategia de splitting para estructuras algebraicas, en https://github.com/pablogventura/OpenDefAlgSplitting.

6.2 FORMATO DEL MODELO DE ENTRADA

Las líneas de comentarios deben comenzar con «#». Las líneas vacías son ignoradas. Por ejemplo:

```
# Este archivo contiene el modelo malvado
```

La primera línea debe contener cada elemento del universo separado por un espacio. Por ejemplo:

```
0 1 2
```

Una relación debe comenzar con una línea de declaración con el nombre, luego un espacio, el número de tuplas, otro espacio y, finalmente, la aridad. Las siguientes líneas deben contener cada tupla de la relación con los valores separados por un espacio. Por ejemplo:

```
E 2 3
0 1 2
2 1 0
```

Una operación debe comenzar con una línea declarando el nombre y la aridad, separados por un espacio. Las siguientes líneas deben ser una por cada tupla de la relación dada por el gráfico de la operación separando los valores por un espacio. Por ejemplo:

```
+ 2
0 0 0
0 1 1
0 2 2
1 0 1
```

```
1 1 2
1 2 0
2 0 2
2 1 0
2 2 1
```

Una constante debe comenzar con una línea declarando el nombre y 0 (ya que es una operación 0-aria) por un espacio. La siguiente línea debe ser el valor. Por ejemplo:

```
Zero 0
0
```

6.3 EJECUCIÓN DEL CHEQUEO DE DEFINIBILIDAD

Una vez descargado el código y dentro de su directorio se ejecuta:

```
python3 main.py modelo.model
```

Donde «modelo.model» es un archivo que contiene la estructura donde se va a correr la prueba de definibilidad. Las relaciones a chequear serán todas aquellas con nombres que comiencen en «T». La salida será la respuesta al chequeo de definibilidad para todos los algoritmos y la fórmula en el caso del Algoritmo 5.4.

6.4 EJECUCIÓN DE LAS HERRAMIENTAS EN LA NUBE

Para facilitar el uso de las herramientas proporcionamos un IPython Notebook para su ejecución en Google Colab en cada uno de los repositorios de las herramientas. Las implementaciones se pueden encontrar en el siguiente enlace https://github.com/pablogventura/. Cada una de ellas en los siguientes directorios:

- Para el Algoritmo 4.1 para estructuras relacionales, se encuentra en:
  - OpenDef/OpenDef_Running_Example.ipynb.
- Para el Algoritmo 5.2 para estructuras algebraicas con una estrategia de merging, en:
  - OpenDefAlgMerging/OpenDefAlgMerging_Running_Example.ipynb.
- Para el Algoritmo 5.4 para estructuras algebraicas con una estrategia de splitting, en:
  - OpenDefAlgSplitting/OpenDefAlgSplitting_Running_Example.ipynb.

7

RESULTADOS EMPÍRICOS

En esta sección presentamos una evaluación empírica de nuestras implementaciones de los Algoritmos 4.1, 5.2 y 5.4. Nuestras herramientas están desarrolladas en Python 3, bajo la licencia GPL 3.

El objetivo principal de estos experimentos es dar una idea de los tiempos que nuestras herramientas requieren para computar diferentes tipos de instancias y de cómo los parámetros de las instancias impactan en los tiempos de ejecución.

7.1 ALGORITMO 4.1 PARA ESTRUCTURAS RELACIONALES

Todos los tests fueron desarrollados usando un procesador Intel Xeon E5-2620v3 con 12 núcleos (de todas maneras, nuestro algoritmo no hace uso de paralización) a 2.40GHz con 128 GiB DDR4 RAM 2133MHz. La memoria nunca fue un problema en las pruebas que ejecutamos. Como no hay un conjunto estándar de ejemplos para las pruebas, trabajamos con instancias aleatorias. Teniendo en cuenta las estrategias que nuestro algoritmo implementa, decidimos considerar los siguientes parámetros:

1. El tamaño del universo del modelo de entrada.
2. Número de relaciones en el modelo de entrada.
3. Tamaño de las relaciones en el modelo de entrada (ver el párrafo a continuación).
4. Aridad de la relación target.

Los parámetros 2 y 3 tienen el objetivo de probar nuestra herramienta en modelos con diferentes tipos de estructuras. Cuando consideramos cómo el número de tuplas en las relaciones base afecta la estructura, es fácil ver que depende del tamaño del universo (agregar una arista a un grafo con 3 nodos tiene mucho más impacto que agregar una arista a un grafo con 50 nodos). Como medida para una relación base R de $\boldsymbol{A}$, usaremos su densidad, definida como sigue:

$$\text{density}(R) = \frac{|R|}{|A|^{\text{ar}(R)}}.$$

Como intercambiar una relación por su complemento no afecta la definibilidad abierta, tiene sentido considerar solo relaciones base con densidad como máximo 0,5, ya que en este algoritmo no se aprovecha la estructura de target al momento de recorrer el modelo.

Para cada experimento, computamos la mediana del tiempo de ejecución (en segundos), sobre 300 instancias en cada configuración.

7.1.1 *Sobre el número de tipos de isomorfismo de los submodelos*

Una característica fundamental de nuestro algoritmo es que toma ventaja de los modelos con una pequeña cantidad de tipos de isomorfismo entre sus subestructuras (ya que en ese caso el conjunto $\mathcal{S}$ en el Algoritmo 4.1 permanece pequeño a lo largo del cómputo y cada vez que procesamos un submodelo lo comparamos con todos los miembros de $\mathcal{S}$). Se puede entender a un modelo que posee pocos tipos de isomorfismo en sus subestructuras como que tiene una estructura *regular* o *simétrica*. Por otro lado, si hay una gran diversidad de tipos de isomorfismo de los submodelos, el modelo tiene menos regularidades que nuestro algoritmo pueda aprovechar. Definimos la *k-diversidad* de una estructura $\boldsymbol{A}$ como el número de tipos de isomorfismo entre sus subestructuras de tamaño k. En este punto, es importante notar que la k-diversidad que tiende a tener el mayor impacto en los tiempos de ejecución es cuando $k = \text{máx}(\text{spec}\, T)$. Esto se debe al hecho de que los submodelos de tamaño mayor que k no son procesados y el número de submodelos de tamaño menor que k es sustancialmente menor (suponiendo $k \leq |A|/2$) y es mucho más fácil revisarlos por isomorfismos. Debemos señalar que, para pensar que la k-diversidad es grande o pequeña, debemos compararla con el número total de subconjuntos de tamaño k de A.

Para ser capaces de evaluar el impacto de la diversidad en el tiempo de ejecución, necesitamos generar instancias de distintas diversidades, para lo cual debemos notar dos cosas. Primero, los modelos con relaciones dispersas (de baja densidad) tienen baja diversidad ya que la mayoría de los submodelos no contienen ninguna arista de la relación. Segundo, un pequeño número de relaciones base de aridad baja implican una menor cantidad de tipos de isomorfismo posibles (e.g., hay solo 10 grafos dirigidos con dos elementos). Así, variando la densidad y el número de relaciones base en la configuración de nuestros experimentos, somos capaces de obtener un rango de diversidades. Como la diversidad crece rápidamente en función de la densidad de las relaciones base, utilizamos un incremento logarítmico para las densidades.

En todas las configuraciones de experimentos, salvo una (Figura 7.2), utilizamos solo relaciones binarias como base, debido a estas dos razones:

- Todas las relaciones target tienen como máximo aridad 3, por lo tanto, la función **modelThinning** eliminará las relaciones base de aridad mayor que 3.
- El punto de variar las relaciones base es para generar un rango de diversidades (como explicamos arriba) y esto puede ser conseguido agregando relaciones base, ya sean binarias o ternarias. Sin embargo, usando relaciones binarias tenemos un control mas preciso sobre los incrementos en la diversidad.

7.1.2 *Experimentos con target definible*

Cuando la relación target es definible, nuestro algoritmo tiene que encontrar y revisar por preservación el máximo posible número de subisos de $\boldsymbol{A}$. Por lo

tanto, para tener una idea de los tiempos del peor caso, ejecutamos la mayoría de los experimentos para el caso definible. Las instancias son generadas fijando el universo junto con la cantidad y densidad de las relaciones base, para luego construir cada una tomando tuplas aleatoriamente. En todas las instancias, la relación target contiene todas las tuplas del largo dado, por lo que son obviamente definibles.

En los resultados mostrados en la primera línea de la Figura 7.1, el target es binario. El tamaño de los dominios es 30, 40 y 50, y el número de relaciones base es 1, 5 y 10. Todas las relaciones base tienen aridad 2, y sus densidades están en $\frac{0{,}5}{2^4}, \frac{0{,}5}{2^3}, \frac{0{,}5}{2^2}, \frac{0{,}5}{2^1}, \frac{0{,}5}{2^0}$. Para los tests mostrados en la segunda línea, la aridad del target es 3.

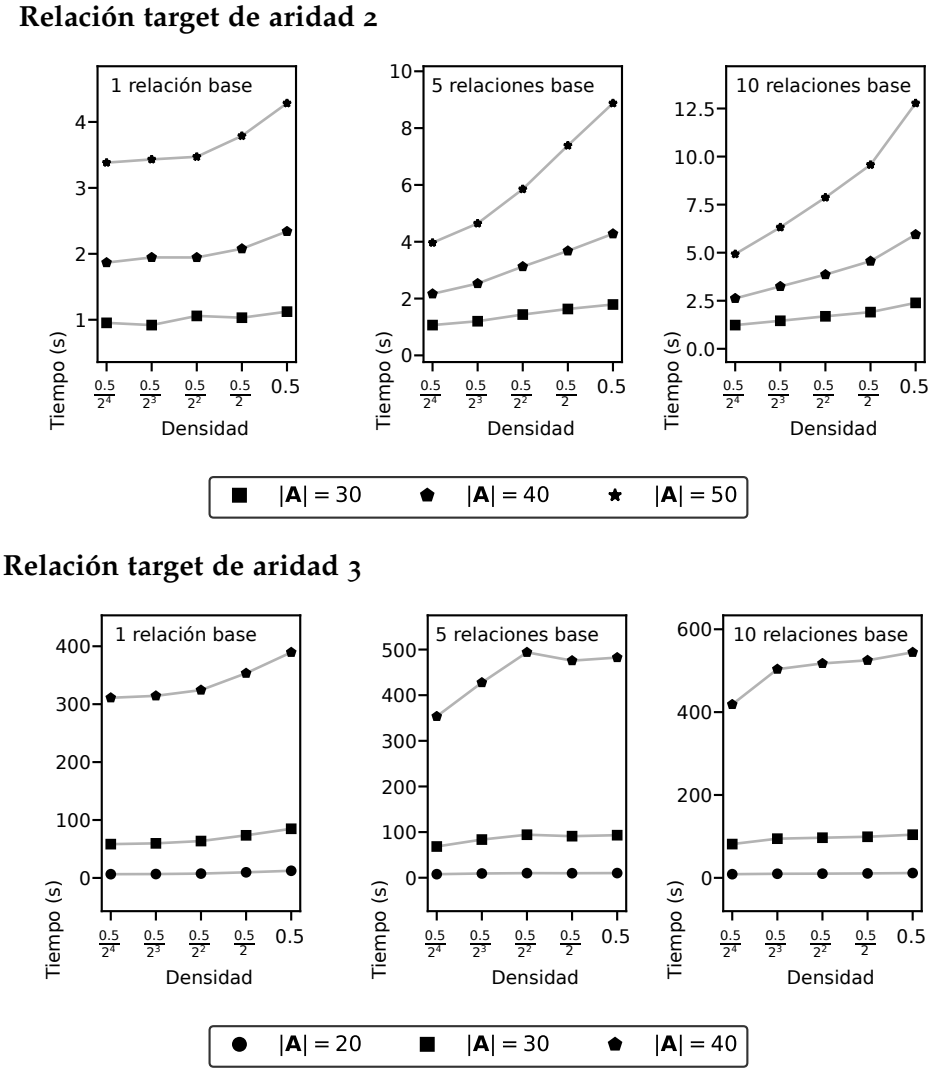


Figura 7.1: Múltiples relaciones base de aridad 2 con targets de aridad 2 y 3.

Lo primero que notamos es la considerable diferencia en cuanto a tiempos entre las instancias con targets binarios y ternarios. Esto es fácilmente comprensible si

consideramos que cada uno de los $\binom{|A|}{k}$ submodelos de tamaño $k = \text{máx}\,\text{spec}(T)$ tiene que ser procesado, y en estos ejemplos k coincide con la aridad del target. En cuanto al impacto del tamaño de los universos, podemos observar claramente un comportamiento polinómico; cuadrático, para el target binario, y cúbico, para el target ternario. De nuevo, esto está en correspondencia directa con el número de submodelos procesados en cada caso. Finalmente, analicemos el efecto del aumento del número y la densidad de las relaciones base. Observar que estos parámetros no afectan al número de submodelos a procesar, sino a la cantidad de procesamiento que requiere cada submodelo (como se explica en la Sección 7.1.1). Aumentar la densidad y la cantidad de relaciones base dificulta el problema, ya que implica un aumento en la diversidad. Sin embargo, es interesante notar que la sobrecarga es bastante pequeña. Por ejemplo, comparando los tiempos en la segunda línea de la Figura 7.1 entre instancias con universos de 40 elementos, podemos ver en el gráfico de la izquierda que, para una relación base de la menor densidad, el tiempo es de alrededor de 300 segundos, mientras que, como se puede ver en el gráfico de la derecha, el tiempo para los mismos tamaños de universo y diez relaciones base de aridad 0.5 es de alrededor de 550 segundos, lo que es un incremento modesto considerando la diferencia en el tamaño de las instancias comparadas (aquí, con el tamaño, nos referimos al número de bits necesarios para codificar la instancia). Este comportamiento se puede explicar al notar que aunque existe un gran diferencia de diversidad entre las dos instancias, la mayoría de las comprobaciones de isomorfismos hechas en el caso de alta diversidad son rápidamente respondidas con la función **cardinalityCheck**. A pesar de que aún deben realizarse muchas más comprobaciones por isomorfismos, se evitan las llamadas a la subrutina de comprobación de isomorfismos en el caso de mayor diversidad (recuerde que la prueba de estructuras relacionales para isomorfismo es GI-completa [16]). Debe tenerse en cuenta que este comportamiento puede no trasladarse a situaciones en las que las relaciones base no se generen de forma aleatoria.

7.1.2.1 *Efecto del modelThinning*

La subrutina **modelThinning** puede tener un gran impacto en el tiempo de ejecución. La Figura 7.2 muestra los tiempos con y sin la aplicación de la función **modelThinning** al modelo de entrada (línea 3 en el Algoritmo 4.1). Todas las instancias tienen dominios de 20 elementos y una relación base ternaria cuya densidad varía, como en los experimentos anteriores. El target es el conjunto de todas las 3-uplas y, por lo tanto, es definible.

La diferencia en los tiempos es explicada al notar que **modelThinning** se comporta rompiendo la relación base ternaria en una relación ternaria y varias binarias y unarias. Por lo tanto, la posibilidad de que la función **cardinalityCheck** detecte la inexistencia de isomorfismos entre submodelos aumenta enormemente, ya que para que dos submodelos $\boldsymbol{B}$ y $\boldsymbol{C}$ puedan ser isomorfos se debe dar que $\left|R^{\boldsymbol{B}}\right| = \left|R^{\boldsymbol{C}}\right|$ para cada R en el lenguaje. Nuevamente, este análisis podría no aplicar a relaciones base que no fueran generadas aleatoriamente.

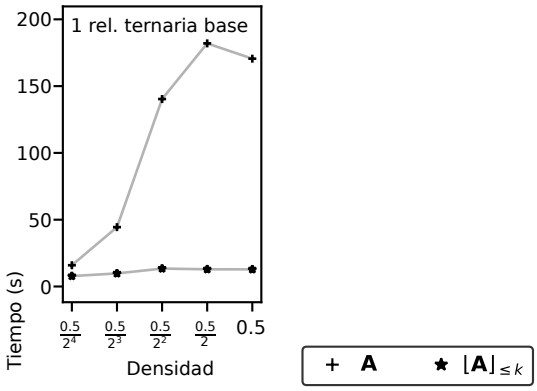
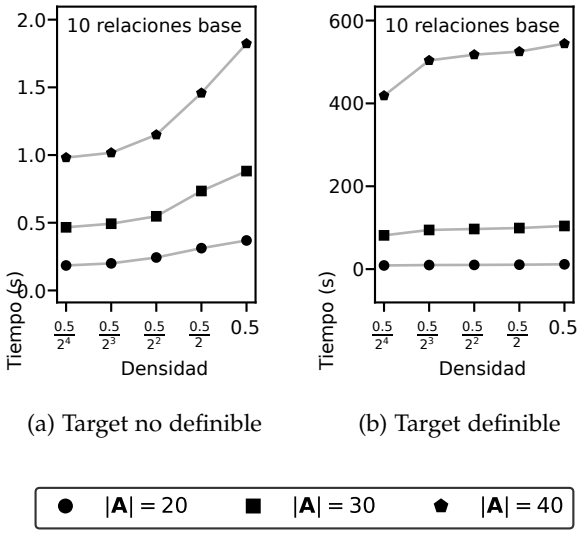


Figura 7.2: $|A| = 20$, una relación base ternaria y $T = A^3$.



(a) Target no definible (b) Target definible

Figura 7.3: Comparación entre los casos definibles y no definibles.

7.1.3 *Experimentos con target no definible*

Los casos donde la relación target T no es definible son potencialmente mucho más simples para nuestro algoritmo, ya que se puede frenar la computación al momento de encontrar un subisomorfismo que no preserva T. A la izquierda de la Figura 7.3, podemos ver los tiempos de los casos no definibles, donde las estructuras de entrada tienen los mismos parámetros que en el gráfico de la derecha en la segunda línea de la Figura 7.1 (la cual repetimos a la derecha de la Figura 7.3). Para generar targets no definibles para cada instancia, generamos aleatoriamente relaciones ternarias de densidad 0.5 hasta que encontramos una no definible.

7.2 ALGORITMOS 5.2 Y 5.4 PARA ESTRUCTURAS ALGEBRAICAS

En esta sección, presentamos una comprobación empírica del rendimiento del algoritmo IsoType (Algoritmo 5.1) versus la búsqueda de isomorfismos usando CSP, así como experimentos con OpenDefAlg para diferentes álgebras de entrada.

7.2.1 *Comparando* isoType *con un solver CSP*

En el capítulo previo, usamos el solver CSP Minion [13] para computar isomorfismos entre subestructuras en nuestro algoritmo con el objetivo de decidir definibilidad abierta en una estructura relacional. En esta nueva estrategia, para resolver el problema, usamos la función IsoType para computar los isomorfismos. Para comparar nuestra estrategia previa con la que presentamos ahora, diseñamos el siguiente experimento.

Dada un álgebra $\boldsymbol{A}$, el experimento consiste en particionar un conjunto aleatorio R de A^k; primero, usando IsoType; y, luego, usando CSP para encontrar isomorfismos relaciones de la siguiente manera: para cada $\bar{a} \in R$, computamos el subuniverso generado $\mathrm{Sg}(a)$ y, luego, construimos una estructura relacional $\boldsymbol{A}^{\mathrm{Rel}}_{\mathrm{Sg}(a)}$ con universo $\mathrm{Sg}(a)$ y, para cada símbolo de operación f en $\boldsymbol{A}$, la relación $F = \mathrm{graph}(f^{\boldsymbol{A}}|_{\mathrm{Sg}(a)})$ (el gráfico de la restricción de $f^{\boldsymbol{A}}$ a $\mathrm{Sg}(a)$). Notar que $\boldsymbol{A} \simeq \boldsymbol{B}$ si y solo si $\boldsymbol{A}^{\mathrm{Rel}} \simeq \boldsymbol{B}^{\mathrm{Rel}}$. Esto nos permite usar CSP para buscar isomorfismos entre $\boldsymbol{A}^{\mathrm{Rel}}_{\mathrm{Sg}(a)}$ y $\boldsymbol{A}^{\mathrm{Rel}}_{\mathrm{Sg}(b)}$ que extiendan el mapeo $\bar{a} \to \bar{b}$. Comparamos los tiempos de ejecución de esta última estrategia con los de la primera.

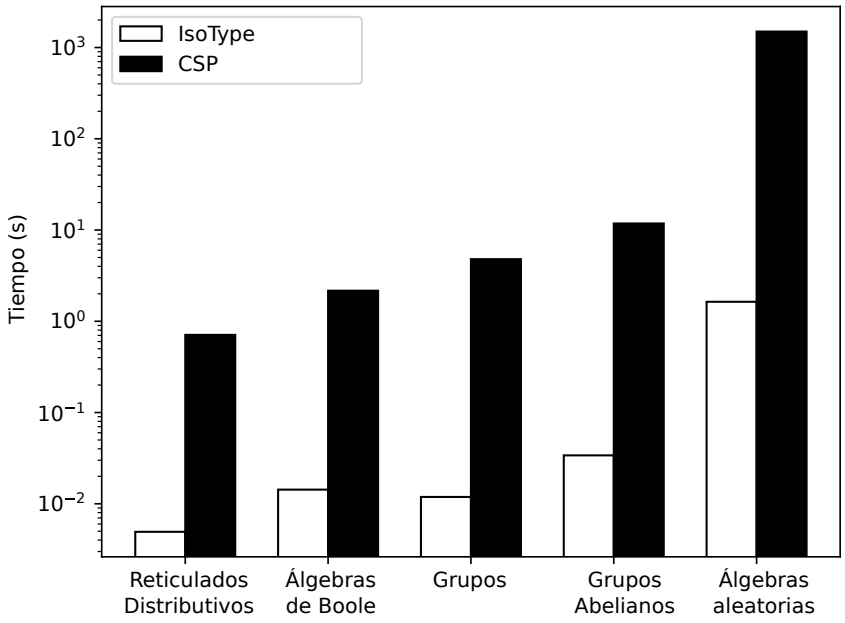


Figura 7.4: Tiempo para computar los tipos de isomorfismo.

Como no hay un conjunto de experimentos estándar para estos problemas, utilizamos cinco familias de álgebras: reticulados distributivos, álgebras de Boole, grupos, grupos abelianos y, finalmente, álgebras aleatorias. Estas últimas fueron generadas eligiendo aleatoriamente operaciones sobre un universo fijo. Para cada familia ejecutamos el experimento en 300 álgebras y tomamos la mediana del tiempo de ejecución. En la figura 7.4 podemos ver cómo la estrategia de IsoType es más rápida; la razón detrás de esto es que, contrario a CSP, IsoType guarda los tipos de isomorfismo en vez de computarlos cada vez. Es interesante notar como el tiempo incrementa mientras el número de subisomorfismos decrece, donde nuestra estrategia basada en simetrías internas es menos eficiente.

7.3 COMPARANDO LOS ALGORITMOS DE MERGING Y SPLITTING PARA ESTRUCTURAS ALGEBRAICAS

Teniendo dos enfoques algorítmicos diferentes para resolver el problema de definibilidad abierta, es natural preguntarse si alguno supera al otro. Tal comparación, a nivel teórico, resulta ser extremadamente difícil. Por lo tanto, nuestro enfoque fue ejecutar (implementaciones de) estos algoritmos en una serie de escenarios de prueba. Además, las pruebas nos permiten obtener una mejor comprensión en cuanto a las limitaciones prácticas de nuestras implementaciones en términos de tamaño de entrada.

Ejecutamos nuestros algoritmos en tres tipos de estructuras algebraicas: álgebras booleanas, grupos abelianos y álgebras generadas aleatoriamente. Estas últimas se generaron eligiendo operaciones aleatorias sobre un universo fijo. Para cada familia ejecutamos la prueba sobre 300 álgebras y tomamos la mediana de las muestras de tiempo de pared.

Inspeccionando ambas estrategias algorítmicas no es difícil concluir que el peor escenario es aquel en el que la relación target es definible. Esto se confirmó, además, durante las pruebas, ya que descubrimos que ambos algoritmos tardan muy poco en decidir que una relación target dada no es abierta definible. Por lo tanto, todas las pruebas presentadas son para entradas en las que el target es definible.

7.3.1 *Evaluación del rendimiento del algoritmo de merging*

Nuestro primer conjunto de datos empíricos se refiere al rendimiento del Algoritmo 4.1 en varios tipos de entradas. Las cardinalidades de las álgebras pasan por 4, 8, 16 y 32. En todos los casos, el target era la relación binaria formada por todos los pares del dominio del álgebra. Las álgebras aleatorias están dotadas de una operación binaria y otra ternaria. Los grupos abelianos se obtienen como productos de grupos cíclicos $\mathbb{Z}_k$.

Los resultados se exponen en la figura 7.5. Podemos ver cómo el algoritmo aprovecha las simetrías internas de las álgebras: el tiempo de ejecución aumenta cuando la cantidad de subisomorfismos disminuye. En particular, es interesante

observar el comportamiento de la curva de los grupos abelianos. Dado que todos los grupos de la prueba son productos directos con factores en $\{\mathbb{Z}_2, \mathbb{Z}_4\}$ (por ejemplo, el de 16 elementos es $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_4$ y el de 32 elementos es $\mathbb{Z}_2 \times \mathbb{Z}_4 \times \mathbb{Z}_4$), el hecho de que los grupos más grandes tengan factores repetidos aumenta el número de subisomorfismos, y, por lo tanto, nuestro algoritmo tiene un mejor rendimiento.

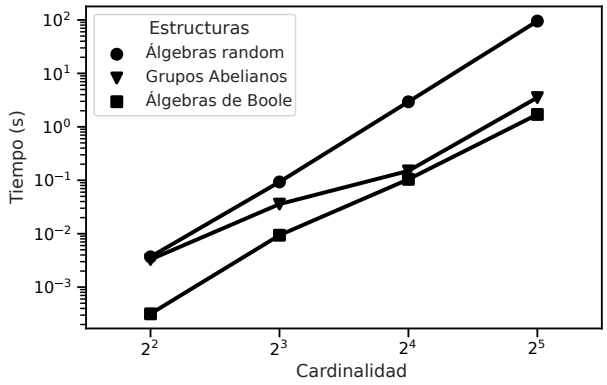


Figura 7.5: Tiempo para decidir definibilidad mediante el algoritmo de merging.

7.3.2 *Evaluación del rendimiento del algoritmo de splitting*

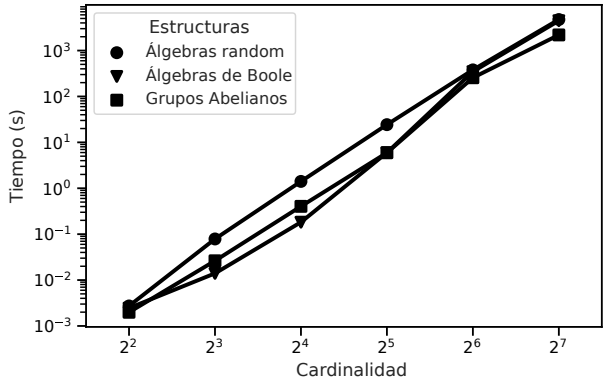


Figura 7.6: Tiempo para decidir definibilidad mediante el algoritmo de splitting.

Ahora pasamos al análisis de rendimiento del Algoritmo 5.4. Este algoritmo fue capaz de manejar álgebras de entrada de hasta 128 elementos. Más llamativo aún es el hecho de que las relaciones target en estas pruebas son ternarias, dado que el espacio de búsqueda depende exponencialmente de la aridad del target, como se vio en la Sección 3.3). También cabe mencionar que, dado que el algoritmo de splitting produce una respuesta inmediata sobre una relación target que contiene

todas las tuplas de una longitud determinada, la relación de entrada para estas pruebas se obtuvo tomando la extensión de una fórmula construida al azar.

En la Figura 7.6, vemos que, aunque las estructuras con más simetrías internas siguen tardando menos en ser procesadas, la diferencia ya no es tan notable. Esto puede explicarse por el hecho de que el algoritmo de splitting no necesita determinar completamente los tipos de isomorfismo de cada tupla en el espacio de búsqueda.

7.3.3 *Comparando ambas estrategias*

Nuestro último conjunto de datos empíricos expone el rendimiento de ambos algoritmos en la misma muestra de entradas. Aquí, al igual que en las pruebas del algoritmo de splitting, la relación de entrada se obtiene como la extensión de una fórmula libre de cuantificadores generada aleatoriamente. Los resultados se presentan en la figura 7.7. Tal y como se preveía en nuestras pruebas anteriores, el algoritmo de splitting supera ampliamente al de merging. De hecho, podemos ver que el algoritmo merging cae fuera de la ventana de tiempo considerada en modelos de tamaño 32 y superiores (las proyecciones prevén tiempos de ejecución superiores a 25 horas). Una razón obvia que explica esta sorprendente diferencia de rendimiento es el hecho de que, a diferencia del algoritmo de merging, el algoritmo de splitting no calcula el tipo de isomorfismo de cada tupla en el espacio de búsqueda. Además, para muchas de las tuplas procesadas por el algoritmo de splitting, el tipo de isomorfismo solo se calcula parcialmente. Teniendo en cuenta que el algoritmo de splitting también proporciona una fórmula en el caso positivo, podemos concluir que el algoritmo de splitting es claramente una estrategia más eficiente.

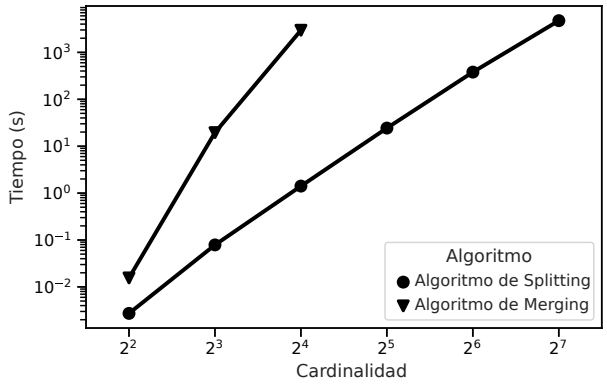


Figura 7.7: Tiempos utilizando el algoritmo de merging frente al de splitting.

Parte VI

CONCLUSIONES

8

CONCLUSIONES

En este trabajo diseñamos e implementamos un algoritmo que decide el problema de definibilidad para fórmulas abiertas de primer orden con igualdad sobre un lenguaje puramente relacional y dos algoritmos con estrategias diferentes para el caso puramente funcional. Los algoritmos se basan en la caracterización semántica de la definibilidad abierta dada en [3].

Para el caso puramente relacional, en el Teorema 13 se prueba que este problema es coNP-completo y que las fórmulas más cortas que logran definir la relación pueden crecer exponencialmente para secuencias de instancias que crecen polinomialmente en tamaño. Por lo tanto, el enfoque directo para resolver la definibilidad abierta mediante la búsqueda de una fórmula definidora no se ve como una alternativa tentadora. (Además, hay que notar que cualquier tipo de testigo con cota polinomial a la definibilidad implicaría coNP $\subseteq$ NP.) En cambio, nuestro algoritmo lleva a cabo una búsqueda sobre los submodelos de tamaño como máximo $k = \text{máx}(\text{spec}(T))$ de la estructura de entrada, y tiene que agotar el espacio de búsqueda para dar una respuesta positiva. Por lo tanto, el tiempo de ejecución de nuestro algoritmo depende exponencialmente del parámetro k (para un análisis detallado de la complejidad paramétrica de OpenDef ver la sección 3.3). Asimismo, reunimos estos hechos para hacer un refinamiento de la caracterización semántica original de la definibilidad abierta [3] en el Corolario 26, que puede tener un impacto significativo en casos adecuados. Los experimentos empíricos confirman la dependencia exponencial en k, y nos dan una idea del impacto de los otros parámetros considerados. Notablemente, aprendimos de los experimentos que, para estructuras generadas aleatoriamente, variar el número y densidad de las relaciones base no tiene un gran impacto en el tiempo de ejecución. Como se discutió en la Sección 7.1.2, esto se debe al hecho de que, a medida que la diversidad crece y es necesario hacer mas comprobaciones por isomorfismo, el número de las que pueden ser decididas fácilmente comparando cardinalidades también crece. Otra conclusión interesante que podemos sacar de los experimentos es que reemplazar una estructura de entrada $\boldsymbol{A}$ por la equivalente $\lfloor \boldsymbol{A} \rfloor_{\leq k}$ puede generar importantes mejoras en los tiempos de computación. Finalmente, los experimentos muestran que nuestra estrategia está mejor condicionada para instancias negativas del problema (por obvias razones).

También, hemos presentado dos algoritmos que deciden el problema de definibilidad mediante fórmulas de primer orden libres de cuantificadores sobre un lenguaje puramente funcional. Estos algoritmos se basan en la caracterización semántica de la definibilidad libre de cuantificadores dada en [3]. Demostramos que este problema es coNP-completo, que es la misma complejidad que obtuvimos en el caso relacional. Previamente, desarrollamos un algoritmo que da una

caracterización del tipo de isomorfismo de una tupla $\bar{a}$ en la estructura dada (en particular, se obtiene el subuniverso que genera).

Nuestro primer procedimiento de decisión de definibilidad (Algoritmo de merging) particiona el conjunto de todas las tuplas k del universo con $k \in \text{spec}(R)$, y tiene que agotar el espacio de búsqueda para dar una respuesta positiva. Por lo tanto, el tiempo de ejecución de nuestro algoritmo depende exponencialmente del parámetro k. Las pruebas empíricas confirman la dependencia exponencial de k y nos dan una idea del impacto de la cardinalidad de la estructura de entrada y de la familia de álgebras sobre la que realizamos las pruebas. También confirman la conjetura de que esta estrategia, que se basa en la existencia de simetrías, se comporta mejor en modelos con una gran cantidad de subisomorfismos, como vimos en la estrategia usada para el caso relacional.

El algoritmo de splitting comienza con un único bloque de tuplas y va partiendo los bloques a medida que avanza la determinación de los tipos de isomorfismo de sus tuplas. Esto difiere de la estrategia anterior, en la que empezamos con tuplas aisladas y fusionamos aquellos bloques con el mismo tipo de isomorfismo. Como era de esperar, esta nueva estrategia resultó ser mucho más eficiente, ya que se pueden descartar bloques enteros sin determinar completamente el tipo de isomorfismo de sus tuplas. El rendimiento superior fue claramente confirmado por las pruebas empíricas. Además, el hecho de que este enfoque permita obtener la fórmula definitoria en el caso positivo constituye una ventaja fundamental.

8.1 TRABAJO FUTURO

Hay muchas direcciones para trabajo futuro. Por un lado, los experimentos empíricos presentados son muy preliminares, por lo que sería necesario llevar a cabo más pruebas. No hay un conjunto estándar de experimentos para definibilidad y hay demasiadas variables que podrían, a priori, impactar en el rendimiento empírico de cualquier algoritmo al testear definibilidad. Encontramos particularmente difícil definir conjuntos de experimentos que nos permitirían explorar la transición de los casos definibles a los no definibles con suavidad. A su vez, sería interesante intentar extender nuestros resultados (y nuestra herramienta) a otros fragmentos de la lógica de primer orden. Ya hemos llevado a cabo un trabajo preliminar investigando el problema de la definibilidad con fórmulas abiertas positivas. Por otro lado, sería natural investigar la definibilidad sobre clases finitas de modelos en lugar de sobre un modelo único. En el caso particular de la definibilidad abierta, los dos problemas coinciden, ya que bastaría con considerar la unión disjunta de todos los modelos de la clase como entrada, pero no es el caso de otras lógicas. La fórmula producida por el algoritmo de splitting (en el caso positivo) resultó ser muy larga en las pruebas. Por lo tanto, una incógnita interesante que se plantea es entender cómo simplificar (si es posible) la fórmula producida. Esto parece ser un problema difícil, y sabemos que en algunos casos las fórmulas de definición son necesariamente grandes (véase Parte iii). Por último, sería interesante identificar clases de álgebras en las que el problema esté bien condicionado y para las que existan algoritmos de tiempo polinómico.

BIBLIOGRAFÍA

[1] Carlos Areces, Miguel Campercholi, Daniel Penazzi y Pablo Ventura. «The complexity of definability by open first-order formulas». En: *Logic Journal of the IGPL* 28.6 (mayo de 2020), págs. 1093-1105. DOI: 10.1093/jigpal/jzaa008. URL: https://doi.org/10.1093/jigpal/jzaa008 (vid. pág. 19).

[2] Carlos Areces, Miguel Campercholi y Pablo Ventura. «Deciding Open Definability via Subisomorphisms». En: *Logic, Language, Information, and Computation*. Ed. por Lawrence S. Moss, Ruy de Queiroz y Maricarmen Martinez. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, págs. 91-105 (vid. pág. 33).

[3] M. Campercholi y D. Vaggione. «Semantical conditions for the definability of functions and relations». En: *Algebra universalis* 76.1 (sep. de 2016), págs. 71-98. DOI: 10.1007/s00012-016-0384-1. URL: https://doi.org/10.1007/s00012-016-0384-1 (vid. págs. 4, 9, 13, 14, 77).

[4] Miguel Campercholi, Mauricio Tellechea y Pablo Ventura. «Deciding Quantifier-free Definability in Finite Algebraic Structures». En: *Electronic Notes in Theoretical Computer Science* 348 (2020). 14th International Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2019), págs. 23-41. DOI: https://doi.org/10.1016/j.entcs.2020.02.003. URL: https://www.sciencedirect.com/science/article/pii/S1571066120300037 (vid. págs. 41, 47).

[5] Miguel Campercholi, Mauricio Tellechea y Pablo Ventura. *Two algorithms to decide Quantifier-free Definability in Finite Algebraic Structures*. 2023. arXiv: 2303.17017 [cs.LO] (vid. pág. 53).

[6] R. Downey y M. Fellows. «Fixed-Parameter Tractability and Completeness I: Basic Results». En: *SIAM Journal on Computing* 24.4 (1995), págs. 873-921. DOI: 10.1137/S0097539792228228. eprint: https://doi.org/10.1137/S0097539792228228. URL: https://doi.org/10.1137/S0097539792228228 (vid. pág. 23).

[7] R. Downey y M. Fellows. «Fixed-parameter tractability and completeness II: On completeness for W[1]». En: *Theoretical Computer Science* 141.1 (1995), págs. 109-131. DOI: https://doi.org/10.1016/0304-3975(94)00097-3. URL: http://www.sciencedirect.com/science/article/pii/0304397594000973 (vid. pág. 24).

[8] R. Downey y M. Fellows. *Fundamentals of Parameterized Complexity*. Springer Publishing Company, Incorporated, 2013 (vid. pág. 23).

[9] H. Ebbinghaus, J. Flum y W. Thomas. *Mathematical Logic*. Springer-Verlag, 1994, pág. 291 (vid. pág. 9).

[10] J. Flum y M. Grohe. «Fixed-Parameter Tractability, Definability, and Model-Checking». En: *SIAM Journal on Computing* 31.1 (2001), págs. 113-145. DOI: 10.1137/S0097539799360768. eprint: https://doi.org/10.1137/S0097539799360768. URL: https://doi.org/10.1137/S0097539799360768 (vid. pág. 25).

[11] J. Flum y M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006 (vid. págs. 21, 23).

[12] M. Garey y D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman y Co., 1979 (vid. pág. 20).

[13] I. Gent, Ch. Jefferson e I. Miguel. «MINION: A Fast, Scalable, Constraint Solver». En: *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*. IOS Press, 2006, págs. 98-102 (vid. págs. 36, 70).

[14] Wilfrid Hodges. *Model theory*. Cambridge University Press, 1993 (vid. pág. 9).

[15] S. Russell y P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010 (vid. págs. 6, 36).

[16] V. Zemlyachenko, N. Korneenko y R. Tyshkevich. «Graph isomorphism problem». En: *Journal of Soviet Mathematics* 29.4 (mayo de 1985), págs. 1426-1481. DOI: 10.1007/BF02104746. URL: https://doi.org/10.1007/BF02104746 (vid. pág. 68).