UNIVERSIDAD DE GRANADA

ESTUDIO EXPERIMENTAL DE ALGORITMOS DE CÁLCULO DE RETÍCULOS EN ANÁLISIS FORMAL DE CONCEPTOS

MIGUEL ÁNGEL CANTARERO LÓPEZ

Trabajo Fin de Grado

Doble Grado en Ingeniería Informática y Matemáticas

Tutores

Nicolás Marín Ruiz

Daniel Sánchez Fernández

FACULTAD DE CIENCIAS

E.T.S. INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, a 2 de diciembre de 2021

Estudio experimental de algoritmos de cálculo de retículos en Análisis Formal de Conceptos

Miguel Ángel Cantarero López

Miguel Ángel Cantarero López *Estudio experimental de algoritmos de cálculo de retículos en Análisis Formal de Conceptos*.

Trabajo de fin de Grado. Curso académico 2021-2022.

Responsables de tutorización

Nicolás Marín Ruiz
Ciencias de la Computación e Inteligencia Artificial

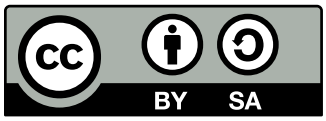
Daniel Sánchez Fernández
Ciencias de la Computación e Inteligencia Artificial

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Facultad de Ciencias

Doble Grado en Ingeniería Informática y Matemáticas

Universidad de Granada



El código fuente de este documento y programas desarrollados están disponibles en el repositorio de GitHub https://github.com/miguecl97/TFG-AlgoritmosFCA

RESUMEN

Palabras clave : análisis formal de conceptos, concepto formal, contexto formal, relación binaria, objeto, atributo, retículo, Teoría del Orden, algoritmo, eficiencia.

En este trabajo se expone una teoría matemática basada en realizar una representación matemática de la palabra concepto. Se utiliza para la extracción de información a partir de un conjunto de datos. Tiene múltiples aplicaciones, siendo una de las principales el agrupamiento jerárquico de datos. Realizaremos un estudio teórico de ella y posteriormente realizaremos un estudio experimental de la eficiencia de algunos algoritmos propuestos en la literatura que se utilizan para aplicarla.

En la primera parte hacemos un desarrollo teórico del Análisis Formal de Conceptos. Comenzamos realizando una introducción y motivación al tema (capítulo 1), que junto con una pequeña síntesis de los resultados obtenidos se puede considerar una de las partes más importantes del trabajo. En el siguiente capítulo (capítulo 2) se presentan los fundamentos teóricos de otras ramas matemáticas que se necesitan para comprender la parte teórica donde, dentro del álgebra, destacan la Teoría del Orden y los retículos. Una vez presentados los fundamentos, la Teoría del FCA es expuesta en el capítulo 3, donde se presentan las principales definiciones del FCA como pueden ser la definición de "contexto formal", cuyo objetivo es unir los "objetos" y "atributos" mediante una "relación" para así desembocar en la definición de "concepto formal". Para terminar este capítulo se anuncia el teorema más importante sobre el que se sustenta toda la teoría anterior: el "Teorema Básico del FCA".

A continuación, el capítulo 4 sirve para realizar la transición de la parte teórica a la práctica. En él se presentan todos los algoritmos del FCA que serán objeto de estudio, y se analiza desde un punto de vista teórico su funcionamiento y propiedades. También explicamos el pseudocódigo que hemos seguido para su posterior implementación.

Una vez finalizada la explicación teórica, la parte 2 se centra en la experimentación realizada para estudiar la eficiencia temporal de los algoritmos. Empieza por establecer cuáles son los espacios de búsqueda sobre los que se va a probar toda la teoría (capítulo 5). Seguidamente realizamos una explicación de cómo se ha diseñado el software (capítulo 6), así como la preparación de los conjuntos de datos en los ficheros que se utilizarán en la experimentación.

El capítulo 7 expone todos los resultados obtenidos en la experimentación, mediante gráficas obtenemos una imagen del desempeño de cada algoritmo según el tipo de datos que reciba como entrada. Los resultados de esta representación se concentran en

el último capítulo (capítulo 8), donde extraemos las conclusiones de haber realizado todo este trabajo y proponemos futuras vías para su continuación.

Todo el trabajo se encuentra disponible de forma pública para la consulta y el uso por parte de otros investigadores interesados en la materia en:
https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code.

SUMMARY

Key words : formal concept analysis, formal concept, formal context, binary relation, object, attribute, lattice, Order Theory, algorithm, efficiency.

This paper presents a mathematical theory used for the extraction of information from a set of data, which is based on the mathematical representation of the word concept. Also it has multiple applications, one of the most popular being hierarchical clustering of data. We will carry out a theoretical study of it and then, by means of an experimental study, we will analyse the efficiency of some of the most important algorithms used to apply it.

The paper is divided into two parts. In the first part we make a theoretical development of Formal Concept Analysis. We begin with an introduction and motivation to the subject (chapter 1) in which the problem is historically situated, the main applications that motivate the need to carry out this study are discussed and which, together with a small synthesis of the results obtained, can be considered one of the most important parts of the work. In the next chapter (chapter 2), the theoretical foundations of other mathematical branches are presented, which are necessary to understand the theoretical part, where within algebra, concepts related to Order Theory and lattices are explained, and on these the FCA Theory is built.

The FCA Theory is presented in chapter 3, where the main definitions of the FCA are presented, such as the definition of "formal context", which links the "objects" and "attributes" by means of a "relation" to present the definition of "formal concept". In addition, this chapter announces the most important theorem on which the whole theory is based, the "The Basic Theorem of the FCA". Finally, it explains how information can be extracted from the graphical representation of a concept lattice and reviews in more detail the applications of this theory.

All these applications have in common the need to build the concept lattice. This task is taken care of by different algorithms presented throughout the literature. Chapter 4 serves to make the transition from the theoretical part to the practical part. It introduces all the FCA algorithms that will be studied. In addition, the fundamentals on which their operation is based are explained one by one, as well as the pseudocode that has been followed for their implementation. They are divided into two main groups: batch algorithms, to which the NextClosure (section 4.4.1), Lindig (section 4.4.2), InClose (section 4.4.3), Berry (section 4.4.4), and Bordat (section 4.4.5) algorithms belong; and incremental algorithms, such as the Norris (section 4.5.1), Godin (section 4.5.2) and AddIntent (section 4.5.3) algorithms. These algorithms have a

certain computational complexity, so choosing the right algorithm for the intended application can bring great advantages in terms of application runtime.

For this reason, part 2 of the paper presents the experimental process followed to compare the efficiency of these algorithms. We begin by establishing the search spaces on which the whole theory will be tested (chapter 5), distinguishing between two types of data sets, artificial ones created explicitly for this experimentation and real ones. The aim is to establish a framework for comparison to clarify the differences between the algorithms and then to test whether these results can be extended to the real world. Chapter 6 explains how the software to be used for the experimentation has been designed. It describes how the main structures such as formal contexts, concept lattice and formal concepts are implemented.

Once all the requirements for experimentation have been explained, Chapter 7 presents all the results obtained in practice, using various graphs to obtain a picture of the performance of each algorithm according to the type of data it receives as input. The section shows that with real data sets the result can be quite different, and this behaviour is explained by the different distributions of the relationships between some sets and others. The last section summarises the results of this chapter.

Finally, chapter 8 reviews the objectives set at the beginning of the work, explaining to what extent these objectives have been achieved and drawing the conclusions that have been obtained in the course of the work. To conclude the paper, possible future avenues for follow-up of this study are indicated.

The entire work is available for consultation and use by any researcher interested in the subject at: https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code.

DECLARACIÓN DE ORIGINALIDAD

D. Miguel Ángel Cantarero López

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2021-2022, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 2 de diciembre de 2021

Fdo: Miguel Ángel Cantarero López

AGRADECIMIENTOS

A mis tutores Daniel y Nicolás por su incansable labor tutorizando este trabajo. A mis amigos por estar siempre ahí, en especial a Daniel. Y en mayor medida gracias a mi familia por apoyarme incondicionalmente cada día durante todos estos años que culminan en la elaboración de este trabajo.

Este Trabajo de Fin de Grado ha estado adherido al Programa Mentor de la Universidad de Granada. Por lo que me gustaría agradecer al estudiante de doctorando Gustavo Rivas Gervilla por todas las herramientas que me ha enseñado y he conseguido poner en práctica realizando este trabajo.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1

INTRODUCCIÓN, MOTIVACIÓN Y OBJETIVOS

1.1 MARCO HISTÓRICO

El Análisis Formal de Conceptos (FCA)[1] es un campo de la matemática aplicada que se basa en la matematización de la palabra concepto y la jerarquía conceptual. De este modo, tal y como explicó Bernhard Ganter [17] en uno de los primeros artículos que se publicaron sobre la materia, el FCA activa y utiliza el pensamiento matemático para el análisis de datos conceptuales y el procesamiento del conocimiento dando lugar a un análisis desde un punto de vista más realista y cercano para el usuario.

En matemáticas, desempeñó un papel especial durante la aparición de la lógica matemática en el siglo XIX. Sin embargo, posteriormente no tuvo prácticamente ninguna repercusión en el pensamiento matemático. No fue hasta 1977 cuando se retomó el tema y se profundizó en estudiar las propiedades de los retículos desde un punto de vista computacional [43] o las relaciones que se podían extraer a partir de su estructura [2]. Ya en 1981, Rudolf Wille [49] formalizó por primera vez toda esta rama de la Teoría del Orden conocida como el Análisis Formal de Conceptos. Desde entonces, a través de un gran número de contribuciones, el Análisis Formal de Conceptos ha obtenido tal amplitud que se utiliza en una gran variedad de campos y a día de hoy se siguen buscando mejoras para su rendimiento. Prueba de ello son las Conferencias Internacionales sobre el Análisis Formal de Conceptos (ICFCA), en las que desde el 2004 los principales investigadores sobre el tema se reúnen anualmente para compartir los mejores avances en la materia. Una lista con enlaces a las publicaciones de estas conferencias se puede encontrar en el siguiente enlace [2].

1 En inglés Formal Concept Analysis, cuyas siglas utilizaremos para referirnos al Análisis Formal de Conceptos en adelante.

2 https://link.springer.com/conference/icfca

1.2 INTRODUCCIÓN Y MOTIVACIÓN DEL TRABAJO

El principal objetivo que reside en el FCA es el de a partir de un conjunto de datos construir un retículo matemático que este formado por conceptos formales. Dichos conceptos no son más que agrupaciones convenientes de conjuntos de instancias y propiedades del conjunto de datos que comparten dichas instancias y viceversa. Este hecho provoca que se puedan utilizar tanto los objetos como los atributos a la hora de trabajar con los elementos del retículo, y esta característica es lo que hace esta teoría tan interesante. Y como principal consecuencia es la cantidad de aplicaciones que posee en la actualidad:

- En la recuperación de información [12], se construye un conjunto de datos que recoge los documentos y las características que poseen. Mediante este conjunto de datos se construye el retículo de conceptos y el buscador se encarga de recuperar los nodos del retículo según las búsquedas del usuario, facilitando la recuperación de elementos relacionados en la búsqueda ya que estos serán los vecinos superiores e inferiores en el retículo del elemento actual.
- En biología, los primeros usos que se le dieron al FCA fueron para analizar los datos de expresión genética, identificando grupos de genes con niveles de expresión similares [4], [30]. El retículo permite recuperar de manera sencilla el conjunto de genes que comparten ciertas expresiones.
- Para el diseño de software [20], el FCA puede utilizarse siempre que los conceptos desempeñen un papel importante en el proceso de software. Los trabajos realizados en este ámbito se centran en la ingeniería de requisitos, el análisis de casos de uso, el modelado orientado a objetos y el análisis de jerarquías de clases u objetos.

También existen aplicaciones más específicas y de gran utilidad:

- El FCA se utilizó para generar una herramienta que permitiese a la policía de Amsterdam identificar a los sospechosos y a las víctimas de la trata de personas a través de analizar 266157 informes de actividades sospechosas. Mediante el uso de retículos de conceptos, se revelaron numerosos sospechosos desconocidos relacionados con el tráfico de personas. Dichas sospechas fueron confirmadas por la policía hasta el punto de que hoy en día el sistema sigue vigente [35].

Todas estas aplicaciones tienen en común la necesidad de construir el retículo de conceptos, para su posterior uso específico. El retículo es la estructura matemática principal sobre la que se basa toda la Teoría del FCA, por ello su construcción es de suma importancia. A lo largo de la literatura, numerosos algoritmos han sido publicados para completar esta tarea, y mediante este trabajo vamos a realizar un estudio experimental de una selección de los más importantes. Estos algoritmos tienen cierta complejidad computacional y por ello hemos considerado interesante realizar un estudio previo para obtener una comparativa de los mismos.

1.3 OBJETIVOS Y RESULTADOS OBTENIDOS

Las principales metas que se plantearon al inicio del trabajo fueron:

1. Estudiar, comprender y explicar de manera clara y concisa las nociones del Análisis Formal de Conceptos, teoría matemática sobre las que se contruyen los algoritmos que se van a estudiar. En particular la noción de retículo de conceptos y la importancia de su cálculo.
2. Realizar una revisión bibliográfica sobre el tema y recopilar los algoritmos más importantes para el estudio.
3. Implementar los algoritmos indicados para la elaboración del retículo de conceptos, a partir del pseudocódigo que proporcionan sus autores en la literatura. Así como probar dichos algoritmos con diferentes conjuntos de datos y analizar su desempeño.
4. Publicar las herramientas utilizadas para el estudio experimental de tal manera que sea reproducible por cualquier investigador interesado en el trabajo.
5. De una manera transversal, comprobar que el uso combinado de las competencias sobre Matemáticas y sobre Ingeniería Informática obtenidas durante el transcurso de la carrera nos permitan analizar y comprender con rigor matemático el desempeño de los algoritmos ante unos tipos de datos determinados.

En general todas las resultados se han completado con éxito, únicamente atravesando por ciertos problemas en la implementación que se solucionaron con más tiempo y dedicación en esa parte.

Sin embargo, a la hora de comprobar el desempeño de los algoritmos, el resultado ha sido más inesperado a lo que se planeaba al inicio del trabajo. Tras implementar y experimentar con todos los algoritmos, los resultados han sido algo diferentes a los del principal artículo previo que existe sobre el tema [25], obteniendo en general mejores tiempos de ejecución. En buena medida esto se debe a la mayor capacidad computacional que tiene nuestra máquina con respecto a la que se usó en el artículo. También se ha diferido en el desempeño de algunos algoritmos, factor que se explica teniendo en cuenta que los autores no proporcionaban ninguna implementación de los algoritmos y quizás algunas estructuras o procesos son diferentes. Sin embargo, en el principal experimento también hemos compartido resultados con los autores, como que el algoritmo *NextClosure* es el que mejor eficiencia temporal tiene en conjuntos de datos de cierto tamaño.

1.4 ESTRUCTURA DEL TRABAJO

Todo este trabajo ha sido recogido en la presente documentación que está estructurada de la siguiente manera:

1. **PARTE I - Presentación del Análisis Formal de Conceptos**. En esta parte se desarrolla todo el marco teórico sobre el que se construye el trabajo. En el primer capítulo se presentan los fundamentos de la Teoría del Orden y del Álgebra. En el segundo capítulo se explica toda la teoría del Análisis Formal de Conceptos y por último en el tercer capítulo se explican los algoritmos que surgen a partir de estos fundamentos, junto con ciertas propiedades teóricas sobre las que se basa cada uno.
2. **PARTE II - Experimentación**. En esta segunda parte se abordan los problemas relacionados con aplicar los algoritmos expuestos anteriormente en diferentes conjuntos de datos. En primer lugar se seleccionan conjuntos de datos determinados que se aplicarán a los algoritmos, posteriormente se seleccionan las métricas para medir el rendimiento y seguidamente se realizan los experimentos. Una vez realizado este proceso, el trabajo analiza los resultados obtenidos. En el último capítulo comentamos las conclusiones que extraemos de todo este proceso.

El código, así como los ficheros con los datos y los retículos correspondientes como resultados producidos en la segunda parte del trabajo se encuentran disponibles en el repositorio de GitHub que se ha utilizado para el trabajo https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code. A su vez, a modo de entrega para la corrección por parte del tribunal se proporciona el siguiente enlace a Consigna con la entrega en la fecha estipulada para ello:

https://consigna.ugr.es/f/9elU5np9Fi2G7GpF/codigo.zip.

Parte I

PRESENTACIÓN DEL ANÁLISIS FORMAL DE CONCEPTOS

2

TEORÍA DEL ORDEN Y RETÍCULOS

Este capítulo trata de aportar al lector una introducción a los conceptos básicos de la Teoría del Orden en los que se basa la Teoría del Análisis Formal de Conceptos. Así como el concepto de retículo que, tal y como veremos a lo largo del trabajo, juega un papel crucial en toda esta teoría.

2.1 INTRODUCCIÓN

Como comentábamos en el primer capítulo, el FCA utiliza métodos matemáticos para desarrollar y analizar las relaciones entre diferentes conceptos y sus implicaciones jerárquicas partiendo de un conjunto de datos. Concretamente, su objetivo es agrupar los objetos jerárquicamente de acuerdo con sus atributos comunes. Se basa en una representación matemática de la palabra concepto, formalizada por primera vez por Wille [49] en 1980. Para ello desempeñará un importante papel la Teoría de Conjuntos.

Tal y como recogió Sarah Roscoe en uno de sus artículos, la idea principal del FCA se puede resumir de la siguiente manera [37]: *los datos se representan en un tipo de datos muy básico que se denomina "contexto formal "* (lo que en la práctica será una tabla cruzada). *Cada contexto formal se transforma luego en una estructura matemática denominada retículo de conceptos cuya representación matemática coincidirá con la de un retículo algebraico. Dicho retículo puede ser transformado de nuevo en un contexto formal y viceversa* [1]. En algunos casos será más apropiado representar los datos gráficamente para mejorar y facilitar su comprensión o alternativamente, investigar sus propiedades utilizando métodos más algebraicos tal y como veremos en próximas secciones. Por último, antes de presentar las nociones básicas del FCA junto con algunos ejemplos, necesitaremos presentar ciertos fundamentos matemáticos que sustenten todo nuestro trabajo.

1 Traducción propia del texto que se puede encontrar en [37]

2.1.1 *Fundamentos de la Teoría del Orden*

La teoría matemática sobre la que se fundamenta el FCA es la llamada Teoría del Orden, dentro del ámbito del Álgebra, en particular en la Teoría de los Retículos Completos. Presentamos a continuación algunas definiciones básicas que nos ayudarán a comprender mejor la base del problema.

Definición 1 (Relación binaria [8])**.** Una relación binaria R sobre dos conjuntos M y N es un subconjunto $R \subseteq M \times N$.

Esto es, $(m, n) \in R$ significa "m está relacionado con n por R" y normalmente se escribe como mRn. La relación binaria R contiene todas las parejas de elementos de $M \times N$ que están relacionados por R. Si $R \subseteq M \times M$, simplemente decimos que R es una relación binaria sobre M.

A partir de una relación binaria, si cumple ciertas propiedades, podemos definir el orden dentro de un conjunto.

Definición 2 (Relación de orden [8])**.** Una relación binaria R se denomina relación de orden sobre un conjunto M si se verifican las siguientes propiedades:

- Reflexiva. xRx para todo $x \in M$.
- Antisimétrica. Si xRy e yRx entonces $x = y$ para todo $x, y \in M$.
- Transitiva. Si xRy e yRz entonces xRz para todo $x, y, z \in M$.

La propiedad reflexiva es equivalente a decir que todo elemento de M está relacionado consigo mismo. La antisimétrica afirma que si dos elementos de M se relacionan mutuamente, entonces son el mismo elemento. Y, en tercer lugar, la propiedad transitiva dice que si un elemento de M está relacionado con otro, y ese otro a su vez se relaciona con un tercero, entonces el primero estará relacionado también con este último.

Una relación de orden usualmente se denota por el símbolo "$\leq$". En este caso $x \leq y$ se lee como "x es menor o igual que y". Para $x \leq y$ también escribimos $y \geq x$ y se dice que y es mayor o igual que x. Si $x \leq y$ y $x \neq y$, escribiremos $x < y$ o $y > x$ y diremos que x es menor que y o que y es mayor que x.

En el ámbito de esta definición, distinguimos dos tipos de relaciones de orden, teniendo en cuenta si se pueden comparar todos los elementos de un conjunto entre sí o no: dando lugar a las relaciones de orden parcial y total.

Definición 3 (Relación de orden parcial [8])**.** Sea M un conjunto dado, diremos que $\leq$ es una relación de orden parcial si y solo si la relación $\leq$ es de orden y al menos un par de elementos de M están relacionados entre sí, es decir, $\exists x, y \in M, (x \leq y) \vee (y \leq x)$.

Si imponemos que todo par de elementos esté relacionado entre sí obtenemos la definición de relación de orden total.

Definición 4 (Relación de orden total [8])**.** Sea M un conjunto dado, diremos que $\leq$ es una relación de orden total si y solo si la relación $\leq$ es de orden y todos los elementos de M están relacionados entre sí, es decir, $\forall x, y \in M, (x \leq y) \vee (y \leq x)$.

Una observación interesante es que en ambas definiciones anteriores, siempre tenemos en cuenta el conjunto M sobre el que definimos la relación de orden [2], por lo que realmente nos estamos refiriendo al par formado por un conjunto y una relación de orden, dando lugar a la siguiente definición:

Definición 5 (Conjunto ordenado [8])**.** Un conjunto totalmente ordenado (o parcialmente ordenado) es un par $\mathbb{M} = (M, \leq)$ que consiste en un conjunto M y una relación de orden total (o parcial) $\leq$ que actúa sobre M.

Un conjunto ordenado será finito si y solo si el conjunto M anterior es finito. En este documento solo trabajaremos con conjuntos ordenados finitos.

Algunos ejemplos de conjuntos ordenados son:

Ejemplo 1 (Ejemplos de relaciones de orden.).

- El conjunto de los números reales $\mathbb{R}$ con su usual relación de orden $\leq$. Es claramente una relación de orden total ya que para todo $x, y \in \mathbb{R}$ con $x \neq y$ se tiene que $y \leq x$ ó $x \leq y$.
- El espacio $\mathbb{R}^n$ con

  $$(x_1, x_2, ..., x_n) \leq (y_1, y_2, ..., y_n) \iff x_i \leq y_i, i = 1, 2, ..., n;$$

  Para el caso $n = 1$ ya hemos visto que es una relación de orden total, pero si $n > 1$ podemos encontrar n-uplas diferentes cuyas coordenadas sean mayores y menores simultáneamente, por ejemplo en $\mathbb{R}^2$, tomando $(1, 2)$, $(0, 3)$, tenemos que $1 > 0$ pero $2 < 3$, por lo que $(1, 2) \not\leq (0, 3)$, y concluimos que es una relación de orden parcial.
- El conjunto de los números naturales $\mathbb{N}$ con la relación de divisibilidad $|$, se trata claramente de una relación de orden parcial pero no total ya que podemos encontrar dos naturales que no sean divisibles entre sí. Por ejemplo 5 no divide a 12 ni 12 divide a 5.
- Sea $\mathcal{P}(X)$ el conjunto potencia de X, esto es el conjunto formado por todos los subconjuntos de X, junto con la relación de inclusión de conjuntos $\subseteq$, como por ejemplo el caso $(\mathcal{P}(X = \{1, 2, 3\}), \subseteq)$. Este conjunto potencia da lugar a una relación de orden parcial . Sean $A = \{1\}$, $B = \{3\}$, $C = \{1, 2\} \in \mathcal{P}(X)$, se tiene que $A \subseteq C$, pero $(A \nsubseteq B) \wedge (B \nsubseteq A)$. Deducimos que no es una relación de orden total.

A continuación presentaremos las nociones de supremo e ínfimo que juegan un papel clave en la teoría del Análisis Formal de Conceptos.

2 De aquí en adelante salvo que se precise de otra manera entenderemos que al decir relación de orden hacemos referencia a una relación de orden parcial.

Definición 6 (Cotas inferiores, cotas superiores [17])**.** Si $\mathbb{M} = (M, \leq)$ es un conjunto ordenado y A es un subconjunto de M, entonces la cota inferior de A es un elemento i de M tal que $i \leq a$ para todo $a \in A$. La cota superior de A se define análogamente como un elemento s de M tal que $s \geq a$ para todo $a \in A$.

Definición 7 (Supremo e ínfimo [17])**.** El mayor elemento de todas las cotas inferiores de A recibe el nombre de ínfimo de A y se denota por $\text{ínf}(A)$ o $\bigwedge A$. Análogamente, el menor elemento de todas las cotas superiores de A se denomina supremo de A y se denota por $\sup(A)$ o $\bigvee A$.

En el caso concreto de $A = \{x, y\}$, también escribimos $x \wedge y$ para el $\text{ínf}(A)$ y $x \vee y$ para el $\sup(A)$. A partir de estas nociones podemos presentar el concepto de retículo.

Definición 8 (Retículo [17])**.** Un conjunto ordenado $\mathbb{L} = (L, \leq)$ diremos que es un retículo si para todo par de elementos de L existen el supremo y el ínfimo. Esto es, existen $x \wedge y$ e $x \vee y$ para todo $x, y \in L$.

Definición 9 (Retículo completo [17])**.** Diremos que L es un retículo completo si para todo subconjunto de L existen supremo e ínfimo. Esto es, existen $\bigwedge S$ y $\bigvee S$ para todo $S \subseteq L$.

Cada retículo completo tiene un elemento mayor $\bigvee L$, al que llamaremos elemento unidad y denotaremos por 1_L. Así mismo , al elemento más pequeño $\bigwedge L$ lo llamaremos elemento nulo y lo denotaremos por 0_L .

La definición de retículo completo implica que exista el supremo y el ínfimo para cada subconjunto $X \subseteq L$. En particular, para $X = \emptyset$ tenemos que $\bigvee \emptyset = 1_L$ y $\bigwedge \emptyset = 0_L$. De donde deducimos el siguiente corolario:

Corolario 1. *[8] Cada retículo finito no vacío es un retículo completo.*

Si no se especifica lo contrario, siempre que aparezca el concepto de retículo estamos refiriéndonos a la definición de retículo completo. Algunas de sus propiedades más útiles son las indicadas por la siguiente proposición.

Proposición 1 (Propiedades de un retículo [32])**.** *Sea $(L, \leq)$ un retículo, las operaciones supremo $\vee$ e ínfimo $\wedge$ satisfacen las siguientes propiedades:*

- *Conmutativa* $\begin{cases} x \vee y = y \vee x \\ x \wedge y = y \wedge x \end{cases}$
- *Asociativa* $\begin{cases} x \vee (y \vee z) = (x \vee y) \vee z \\ x \wedge (y \wedge z) = (x \wedge y) \wedge z \end{cases}$
- *Absorción* $\begin{cases} x \vee (x \wedge y) = x \\ x \wedge (x \vee y) = x \end{cases}$

- *Idempotencia* $\begin{cases} x \vee x = x \\ x \wedge x = x \end{cases}$

Estas propiedades son consecuencia de la definición del conjunto ordenado $(L, \leq)$. Si $x, y \in L$, con $x \leq y$, por definición se cumple que $x \vee y = y$ e $x \wedge y = x$.

Cada conjunto ordenado, en concreto un retículo $\mathbb{L} = (L, \leq)$ se puede representar como un diagrama de líneas, más conocido como Diagrama de Hasse.

Definición 10. (Diagrama de Hasse) [32]. El diagrama de Hasse de un conjunto ordenado $(L, \leq)$ es un grafo dirigido cuyos vértices son los elementos de L, y existe un lado de x a y si $x < y$ y no existe $z \in L$ tal que $x < z < y$.

Siguiendo esta definición no siempre se puede construir el Diagrama de Hasse a partir de un conjunto ordenado. Por ejemplo, si tomamos el conjunto $\mathbb{R}$ con su relación de orden usual " $\leq$ " se trata de un retículo no completo ya que dado cualquier $x \in \mathbb{R}$ no podemos encontrar un $y \in \mathbb{R}$ que esté conectado a x por algún lado. Esto se debe a que $\mathbb{Q}$ es denso en $\mathbb{R}$, y siempre podemos encontrar un $z \in \mathbb{R}$, con $x, y \neq z$ tal que $x \leq z \leq y$.

Sin embargo, si el conjunto L es finito, como será nuestro caso, entonces dados $x, y \in L$ se tiene que $x \leq y$ si $x = y$ ó existe algún camino que parta de x y termine en y.

Una forma habitual de representar el diagrama de Hasse es dibujar los lados como líneas ascendentes, lo que implica colocar los vértices de forma apropiada.

Ejemplo 2. En el caso de los números naturales divisores de 60, como 20,12,30 dividen a 60 y no hay ningún natural mayor que ellos que divida a 60 y sea divisible por estos, todos están conectados a 60 y siguiendo este razonamiento vamos conectando todos los vértices. Como los naturales divisibles por 4 mayores que él son 20 y 12, trazamos una arista que los conecte, y así con el resto de divisores hasta completar el retículo.

Ejemplo 3. El diagrama de Hasse del conjunto $\mathcal{P}(\{1, 2, 3\})$ con la relación de inclusión de conjuntos $\subseteq$ se construye de forma análoga al anterior pero utilizando dicha inclusión de conjuntos como relación de orden.

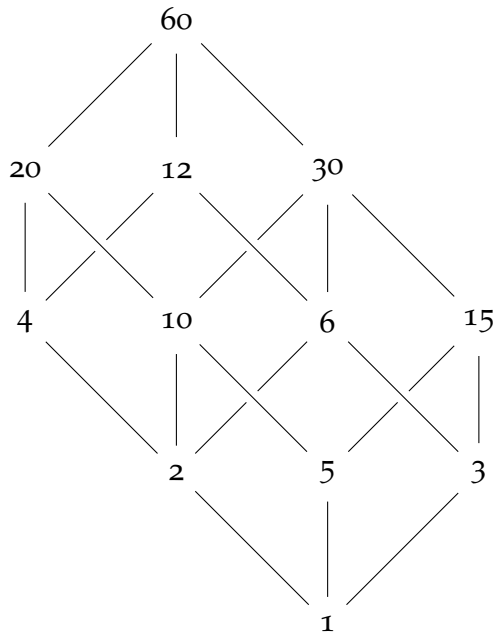


Figura 1: Diagrama de Hasse del retículo de los números naturales divisores de 60, con la relación de divisibilidad $|$.

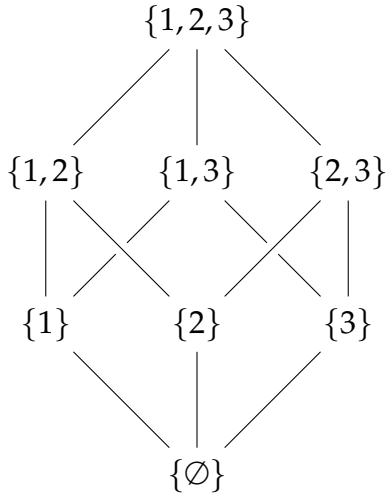


Figura 2: Diagrama de Hasse del retículo del conjunto de partes de $\mathcal{P}(\{1,2,3\})$ con la relación de inclusión $\subseteq$.

Por último, presentaremos unos conceptos que necesitaremos más adelante para elaborar el Teorema Fundamental del FCA.

Definición 11 (Supremo-denso , ínfimo-denso [17])**.** Sea $(L, \leq)$ un retículo completo. Un subconjunto $X \subseteq L$ es supremo-denso en L si todo elemento de L puede ser escrito como el supremo de un subconjunto de X, esto es $\forall\ y \in L$, $y = \bigvee\{x \in X | y \geq x\}$. Análogamente, un subconjunto $X \subseteq L$ es ínfimo-denso en L si todo elemento de L

puede ser escrito como el ínfimo de un subconjunto de X, es decir, si para todo $y \in L$ se puede escribir $y = \bigwedge\{x \in X | y \leq x\}$.

Ejemplo 4 (Ejemplo de un conjunto supremo-denso e ínfimo denso.). Si consideramos el retículo formado por el intervalo cerrado de números reales comprendidos entre el 0 y el 1, $L = \{x \in \mathbb{R} : 0 \leq x \leq 1\}$ con su relación de orden usual $\leq$, obtenemos el retículo completo $(L, \leq)$. Tomando el subconjunto formado por los racionales pertenecientes al intervalo $[0, 1]$ con la misma relación de orden usual $\leq$, este subconjunto $X = \{x \in \mathbb{Q} : 0 \leq x \leq 1\}$ es ínfimo-denso en L ya que $\forall y \in L \; x = \bigwedge\{x \in X : y \leq x\}$ debido a que $\mathbb{Q}$ es denso en $\mathbb{R}$. No es difícil comprobar que este mismo subconjunto es también supremo-denso en L.

Por otro lado, si nos remontamos al Ejemplo 2), tomando el subconjunto $X = \{1, 2, 3, 4, 5\}$ obtenemos un conjunto supremo-denso en el retículo $(D(60), |)$. Ya que cualquier elemento del retículo lo podemos expresar como el supremo de elementos de X. En el caso de 60 por ejemplo lo podemos expresar como $60 = 3 \vee 4 \vee 5$, o también $15 = 3 \vee 5$.

Como veremos a lo largo del documento, unos de los elementos más relevantes en el FCA serán funciones que conecten dos conjuntos ordenados. Necesitaremos ciertas características de estas funciones para trabajar con ellas. Las siguientes definiciones se pueden encontrar en [8].

Definición 12. Sean P y Q conjuntos ordenados. Una función $\varphi : P \to Q$ se dice que es :

1. Monótona, si $x \leq y$ en P implica $\varphi(x) \leq \varphi(y)$ en Q.
2. Un embebimiento de orden, si $x \leq y$ en P si y solo si $\varphi(x) \leq \varphi(y)$ en Q.
3. Un isomorfismo de orden, si es un embebimiento de orden de P en Q y además es biyectiva.

Ejemplo 5. Un ejemplo de isomorfismo de orden lo podemos encontrar tomando los conjuntos ordenados $(\mathbb{R}, \leq)$,$(\mathbb{R}, \geq)$ con la función $f : \mathbb{R} \to \mathbb{R}$ $f(x) = -x$, donde $\mathbb{R}$ es el conjunto de los números reales y $\leq, \geq$ las relaciones usuales de orden. Tenemos una función claramente biyectiva que además es un embebimiento de orden ya que sean $x, y \in \mathbb{R}$ con $x \leq y$ en $(\mathbb{R}, \leq)$, tenemos que $f(x) = -x \geq -y = f(y)$ en $(\mathbb{R}, \geq)$. Tomando $x \geq y$ en $(\mathbb{R}, \geq)$, $f^{-1}(x) = -x \leq -y = f^{-1}(y)$ en $(\mathbb{R}, \leq)$.

A partir de estas características definimos lo que será el ingrediente principal que da lugar a toda la Teoría del FCA, la Conexión de Galois.

Definición 13 (Conexión de Galois [18])**.** Sean $\varphi : P \to Q$ y $\phi : Q \to P$ dos mapas entre dos conjuntos ordenados $(P, \leq)$, $(Q, \leq)$. Entonces diremos que el par de funciones forman una conexión de Galois entre los conjuntos ordenados si:

$$p_1 \leq p_2 \Longrightarrow \varphi(p_1) \geq \varphi(p_2) \tag{1}$$

$$q_1 \leq q_2 \Longrightarrow \phi(q_1) \geq \phi(q_2) \tag{2}$$

$$p \leq \phi(\varphi(p)) \; y \; q \leq \varphi(\phi(q)) \tag{3}$$

Estas tres condiciones necesarias se pueden resumir con la siguiente proposición:

Proposición 2. *[18] Un par de funciones (φ, ϕ) es una conexión de Galois si y solo si:*

$$p \leq \phi(q) \Longleftrightarrow q \leq \varphi(p) \tag{4}$$

Demostración 1. [3]

$\boxed{\Longrightarrow}$ Partimos de $p \leq \phi(q)$. Por la ecuación (1) deducimos que $\varphi(p) \geq \varphi(\phi(q))$, y usando la ecuación (3) obtenemos que $\varphi(p) \geq q$.

$\boxed{\Longleftarrow}$ De $\varphi(p) \leq \varphi(p)$, por la ecuación (4) llegamos a que $p \leq \phi(\varphi(p))$, obteniendo así la ecuación (3). Llamando $p = p_1$ y $q = p_2$, suponiendo que $p_1 \leq p_2$ (en caso contrario intercambiamos los papeles de p_1 y p_2), deducimos que $p_1 \leq \phi(\varphi(p_2))$ y por la ecuación (4), se llega a que $\varphi(p_2) \leq \varphi(p_1)$. Obteniendo así la ecuación (1) y siguiendo el mismo razonamiento la (2).

3 Esta demostración es propia y alternativa a la que aparece en la referencia [18].

3

INTRODUCCIÓN A LA TEORÍA DEL ANÁLISIS FORMAL DE CONCEPTOS

De forma intuitiva, las nociones básicas del Análisis Formal de Conceptos son el contexto formal y concepto formal. Como bien indica Thomas Tilley en la introducción de su artículo [42], el adjetivo "formal" trata de resaltar que estamos trabajando con nociones matemáticas, lo que diferencia el significado de concepto y contexto de su uso normal. Siempre que escribamos concepto o contexto nos estaremos refiriendo a contexto formal o concepto formal.

Durante todo el recorrido de introducción nos ayudaremos de un ejemplo para obtener una visión aplicada de toda la nueva información que se presenta al lector. Comenzaremos definiendo las nociones de contexto formal y de concepto formal, formalizadass por primera vez por Rudolf Wille en 1982.

Definición 14 (Contexto formal ([49]))**.** Un contexto formal $\mathbb{K} := (\mathcal{G}, \mathcal{M}, \mathcal{I})$ está compuesto por un conjunto $\mathcal{G}$, cuyos elementos se llaman objetos, un conjunto $\mathcal{M}$, cuyos elementos se llaman atributos, y una relación binaria $\mathcal{I} \subseteq \mathcal{G} \times \mathcal{M}$. Llamaremos a $\mathcal{I}$ la relación de incidencia y leeremos $(g, m) \in \mathcal{I}$ como "el objeto g tiene el atributo m".

Un contexto formal $\mathbb{K}$ puede ser visto como una tabla 2-dimensional, o tabla cruzada, que usa cruces en sus casillas para indicar la relación entre dos elementos.

I	m_1	m_2	m_3	...
g_1		x		
g_2			x	
g_3	x		x	
...				...

Tabla 1: Tabla cruzada de atributos por objetos $g_1 ... g_n \in \mathcal{G} \times m_1 ... m_n \in \mathcal{M}$ relacionados por I

Utilizaremos un ejemplo para ilustrar las nociones que se acabamos de presentar. Se trata de un pequeño conjunto de datos con información sobre distintos tipos de monumentos de Granada y ciertas características sobre ellos. El conjunto de monumen-

tos será nuestro conjunto de objetos $\mathcal{G} = \{$Alhambra, Catedral, Corral del Carbón, Abadía del Sacromonte, Monasterio de la Cartuja, Madraza de Granada$\}$ todos ellos monumentos bien conocidos del patrimonio granadino. Y el conjunto de atributos será $\mathcal{M} = \{$Reino Nazarí, Interés Cultural, Visitas Guiadas, Descuento Estudiantes$\}$ los cuales indican ciertas propiedades sobre los monumentos. La tabla cruzada de la relación $\mathcal{I}$ entre $\mathcal{G}$ Y $\mathcal{M}$ es la tabla 2.

$\mathcal{I}$	Reino Nazarí	Interés Cultural	Visitas Guiadas	Descuento Estudiantes
Alhambra	x	x	x	
Catedral		x	x	x
Corral del Carbón	x	x		
Abadía del Sacromonte		x	x	
Monasterio de La Cartuja		x		x
Madraza de Granada	x			

Tabla 2: Tabla cruzada de atributos por objetos, de un conjunto de datos de monumentos de Granada.

Ejemplo 6 (Ejemplo de un contexto formal con monumentos de Granada).

En este ejemplo se aprecia de forma clara cómo se relacionan atributos y objetos mediante la relación $\mathcal{I}$, en este caso representada por la tabla cruzada. Se observan relaciones como que la Alhambra posee el atributo de pertenecer al Reino Nazarí o que la Catedral es un monumento con Descuento para Estudiantes (ya que está relacionada con dicho atributo). Por otro lado, podemos leer también que monumentos con Visitas Guiadas hay 3, la Alhambra, la Catedral y la Abadía del Sacromonte. Las cruces indican los atributos que tiene[1] cada objeto. Como veremos más adelante, la tabla cruzada es la herramienta que nos permite implementar el contexto formal a la hora de realizar la implementación. A nivel de diseño, esta tabla será una matriz de unos y ceros, que indicarán si un determinado objeto posee cierto atributo o no.

Una de las claves del FCA es esta relación que se establece entre el conjunto de objetos y el conjunto de atributos, la relación $\mathcal{I}$ dará lugar una conexión de Galois entre el conjunto de objetos y el de atributos que será uno de los principales ingredientes del FCA. Una evidencia puede ser observada en las relaciones entre los dos conjuntos, que en nuestro trabajo serán el conjunto de objetos $\mathcal{G}$ y el conjunto de atributos $\mathcal{M}$. Cuando el conjunto de objetos aumenta, el conjunto de atributos en común que poseen esos objetos decrementa en tamaño, esto es, a más objetos seleccionemos, menos atributos compartirán entre ellos y viceversa. Para definir este comportamiento, introducimos dos operadores de derivación definidos también por Rudolf Wille en 1981. Para un subconjunto $A \subseteq \mathcal{G}$ de objetos, definimos el conjunto de atributos comunes a todos los objetos de A como:

1 Los atributos que están relacionados con cada objeto.

Definición 15 (Operador de derivación para objetos [49]).

$$A' := \{m \in \mathcal{M} \mid g\mathcal{I}m \ \ \forall g \in A\} \ , \ \{\emptyset\}' = \mathcal{M} \tag{5}$$

Si aplicamos el operador de derivación al objeto Catedral, obtendremos todos los atributos relacionados con Catedral, en este caso {Catedral}′={Interés Cultural, Visitas Guiadas, Descuento Estudiantes} o si se lo aplicamos a el subconjunto {Catedral, Corral del Carbón} obtendremos los atributos relacionados con ambos objetos, {Catedral,Corral del Carbón}′={Interés Cultural}.

Respectivamente, para un subconjunto $B \subseteq \mathcal{M}$ de atributos definimos un conjunto de objetos que tienen todos los atributos de B como

Definición 16 (Operador de derivación para atributos [49]).

$$B' := \{g \in \mathcal{G} \mid g\mathcal{I}m \ \ \forall m \in B\} \ , \ \{\emptyset\}' = \mathcal{G} \tag{6}$$

En este caso {Reino Nazarí , Interés Cultural}′={Alhambra, Corral del Carbón} nos dice cuáles son los monumentos pertenecientes al Reino Nazarí y además son de Interés Cultural.

Cabe plantearse qué ocurre cuando aplicamos los dos operadores de derivación uno detrás del otro. Depende de con qué operador empecemos vamos a obtener dos operadores de doble derivación diferentes. El primero lleva subconjuntos de objetos en subconjuntos de objetos $'' : 2^{|\mathcal{G}|} \to 2^{|\mathcal{G}|}$ y el segundo llevará subconjuntos de atributos en subconjuntos de atributos $'' : 2^{|\mathcal{M}|} \to 2^{|\mathcal{M}|}$. En posteriores secciones veremos que estos operadores son *operadores de cierre*.

Ejemplo 7 (Operador doble prima $''$). Tomando el mismo ejemplo que antes, aplicando el operador de derivación a {Catedral,Corral del Carbón} obtuvimos {Catedral,Corral del Carbón}′={Interés Cultural}, aplicando ahora el operador de derivación de atributos obtendríamos el operador doble de derivación para objetos, esto es {Catedral,Corral del Carbón}″={Interés Cultural}′={Alhambra, Catedral, Corral del Carbón, Abadía del Sacromonte, Monasterio de la Cartuja}.

Este par de operadores de derivación se pueden ver como funciones que llevan $' : 2^{|\mathcal{G}|} \to 2^{|\mathcal{M}|}$ y viceversa $' : 2^{|\mathcal{M}|} \to 2^{|\mathcal{G}|}$ y utilizando la definición 13 concluimos que forman una Conexión de Galois. Esto propicia que las siguientes propiedades sean verdaderas para cualquier contexto formal $(\mathcal{G}, \mathcal{M}, \mathcal{I})$.

Proposición 3. *[17] Sean subconjuntos $A, A_1, A_2 \subseteq \mathcal{G}$ de objetos y subconjuntos $B, B_1, B_2 \subseteq \mathcal{M}$ de atributos. Se cumplen las siguientes propiedades :*

$$\textit{Monotonía:} \quad A_1 \subseteq A_2 \Longrightarrow A_2' \subseteq A_1' \ y \ B_1 \subseteq B_2 \Longrightarrow B_2' \subseteq B_1' \tag{7}$$

$$\textit{Extensividad:} \quad A \subseteq A'' \; y \; B \subseteq B'' \tag{8}$$

$$\textit{Idempotencia:} \quad A' = A''' \; y \; B' = B''' \tag{9}$$

$$A \subseteq B' \iff B \subseteq A' \iff A \times B \subseteq \mathcal{I} \tag{10}$$

Veamos una demostración que hemos desarrollado alternativa y con más detalle que la presentada por Bernhard Ganter [17] en su libro.

Demostración 2. Monotonía (7): En primer lugar, observamos que si $m \in A_2'$, entonces por definición m cumple $g\mathcal{I}m \; \forall g \in A_2$, en particular $g\mathcal{I}m \; \forall g \in A_1$. Si $A_1 \subseteq A_2$ se tiene que $m \in A_1'$ para todo $m \in A_2'$ por tanto $A_2' \subseteq A_1'$.

Extensividad (8): Sea $g \in A$ usando la monotonía(7) es claro que $A' \subseteq \{g\}' \iff \forall m \in A' : g\mathcal{I}m \iff g \in A''$.

Idempotencia (9): Por la extensividad (8) tenemos que $A'' \subseteq A''''$. Faltaría probar $A'''' \subseteq A''$. Sea $g \in A''''$ tenemos que $g\mathcal{I}m \; \forall m \in A'''$ y como $A' \subseteq A'''$ esto es lo mismo que decir que $g\mathcal{I}m \; \forall m \in A' \iff g \in A''$.

(10): Es consecuencia directa de la definición de los operadores e $\mathcal{I}$.

Ambos operadores de derivación y la mencionada Conexión de Galois que forman nos permiten definir lo que será un concepto formal.

Definición 17 (Concepto formal ([49]))**.** Un concepto formal de un contexto formal $\mathbb{K} := (\mathcal{G}, \mathcal{M}, \mathcal{I})$ se define como una pareja $(A, B) \subseteq \mathcal{I}$ con $A \subseteq \mathcal{G}$, $B \subseteq M$, que cumple $A' = B$ y $A = B'$.

Al subconjunto de objetos A que forma el concepto lo llamaremos *extensión del concepto formal*. Respectivamente, al subconjunto de atributos B que forma el concepto lo llamaremos *intensión del concepto*. En un sentido general, un concepto está hecho de dos partes: la extensión y la intensión[2]. La extensión cubre todos los objetos que pertenecen a este concepto y la intensión comprende todos los atributos válidos para todos esos objetos.

Si nos remitimos al ejemplo 6 anterior, podemos comenzar a extraer varios conceptos fijándonos en la tabla cruzada. Por ejemplo (A, B) =({Alhambra, Catedral, Corral del Carbón, Abadía del Sacromonte, Monasterio de la Cartuja},{Interés Cultural}) es el concepto de aquellos monumentos de Interés Cultural, cuya extensión es {Alhambra, Catedral, Corral del Carbón, Abadía del Sacromonte, Monasterio de la Cartuja}=A el conjunto de objetos que comparten ese atributo, y cuya intensión sera

2 Algunos autores difieren en el orden

{Interés Cultural}=B. Otro concepto que encontramos es ({Catedral},{Visitas Guiadas, Descuento Estudiantes, Interés Cultural}), este concepto nos indica que en nuestro contexto, el objeto *Catedral* posee los atributos *Visitas Guiadas, Descuento Estudiantes e Interés Cultural* y no existe otro objeto en $\mathcal{G}$ que posea estos 3 atributos.

Denotaremos el conjunto de todos los conceptos del contexto formal $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ como $\mathfrak{B}(\mathcal{G}, \mathcal{M}, \mathcal{I})$ y escribimos $\mathfrak{B}(\mathbb{K})$ de modo abreviado.

En general, no todos los subconjuntos de $\mathcal{G}$ forman una extensión, y no todos los subconjuntos de $\mathcal{M}$ forman una intensión de algún concepto.

Proposición 4. *[16]*

Dos subconjuntos $A \subseteq \mathcal{G}$ y $B \subseteq \mathcal{M}$ son una extensión e intensión de un concepto si y solo si $A'' = A$ y $B'' = B$ respectivamente.

Definamos ahora un orden entre conceptos que nos permita obtener la estructura de retículo en un contexto formal $\mathbb{K} := (\mathcal{G}, \mathcal{M}, \mathcal{I})$.

Definición 18 (Subconcepto [49])**.** (A_1, B_1) es un subconcepto de (A_2, B_2) si $A_1 \subseteq A_2$. Cuando esto ocurra escribiremos $(A_1, B_1) \leq (A_2, B_2)$. Y diremos que el concepto (A_1, B_1) es menos general que (A_2, B_2).

Tomando los dos conceptos anteriores y siguiendo esta definición ya podemos establecer un orden entre ellos y afirmar que el concepto ({Catedral},{Visitas Guiadas, Descuento Estudiantes, Interés Cultural}) es un subconcepto de ({Alhambra, Catedral, Corral del Carbón, Abadía del Sacromonte, Monasterio de la Cartuja},{Interés Cultural}) y podemos escribir ({Catedral},{Visitas Guiadas, Descuento Estudiantes, Interés Cultural}) $\subseteq$({Alhambra, Catedral, Corral del Carbón, Abadía del Sacromonte, Monasterio de la Cartuja},{Interés Cultural}).

Una observación inmediata de esta definición es que cuando $A_1 \subseteq A_2$ por tratarse de conceptos formales se tiene que:

$$B_1 = A_1' = \{m \in \mathcal{M} | \forall g \in A_1 : g\mathcal{I}m\}$$

$$B_2 = A_2' = \{m \in \mathcal{M} | \forall g \in A_2 : g\mathcal{I}m\}$$

de donde deducimos que $B_2 \subseteq B_1$.

De igual manera si $B_2 \subseteq B_1$, entonces:

$$A_1 = B_1' = \{g \in G | \forall m \in B_1 : g\mathcal{I}m\}$$

$$A_2 = B_2' = \{g \in G | \forall m \in B_2 : g\mathcal{I}m\}$$

y deducimos que $A_1 \subseteq A_2$.

Corolario 2. *[49]* $A_1 \subseteq A_2 \iff B_2 \subseteq B_1$

Así podemos enunciar también la definición anterior de la siguiente manera. (A_1, B_1) es un subconcepto de (A_2, B_2) si $B_2 \subseteq B_1$.

Hemos definido una relación binaria entre los conceptos. Veámos que con esta definición "$\leq$" es una relación de orden:

Proposición 5 (Propiedades de $\leq$: [49])**.**

1. *Reflexividad:* $(A, B) \leq (A, B)$
2. *Antisimetría:* $(A_1, B_1) \leq (A_2, B_2) \wedge (A_2, B_2) \leq (A_1, B_2) \Longrightarrow (A_1, B_1) = (A_2, B_2)$
3. *Transitividad:* $(A_1, B_1) \leq (A_2, B_2) \leq (A_3, B_3) \Longrightarrow (A_1, B_1) \leq (A_3, B_3)$

Ahora podemos definir el ínfimo de dos conceptos como el mayor de los subconceptos comunes [16]:

$$(A_1, B_1) \wedge (A_2, B_2) = (A_1 \cap A_2, (B_1 \cup B_2)'')$$

La intuición nos dice que si buscamos conceptos que sean menores que esos dos, necesitamos que su extensión esté contenida en la de ambos conceptos (por ello se toma la intersección), y este objeto tendrá todas las propiedades que tenían ambos objetos y se le aplica el operador de cierre $''$ para asegurarnos que sea una intensión. Aunque intuitivamente parezca sencillo la demostración técnica de que a través de esta definición obtenemos en efecto el ínfimo de dos conceptos es bastante tediosa y no aporta nada nuevo al lector. Para profundizar en ellas, tanto la definición como su demostración se pueden consultar en [16].

Similarmente se define el supremo de dos conceptos como el menor de los conceptos más generales [16]:

$$(A_1, B_1) \vee (A_2, B_2) = ((A_1 \cup A_2)'', B_1 \cap B_2)$$

3.0.1 *De los contextos formales a los conceptos formales*

El número de conceptos en un retículo de conceptos dependerá de la relación $\mathcal{I}$ en el contexto formal. Un contexto formal generado por $|G|$ objetos y $|M|$ atributos producirá, como mucho, un retículo de conceptos con $2^{min(|G|,|M|)}$ conceptos. En particular, un contexto formal $\mathbb{K} := (\{1, ..., n\}, \{1, ..., n\}, \neq)$ tendrá 2^n conceptos.

Presentaremos un método muy básico para extraer los conceptos a partir de la tabla cruzada, basado en uno de los primeros algoritmos presentados para el FCA, llamado *NEXT-CLOSURE* [15] pero mucho más simplificado.

Partiendo de la idea de que un subconjunto de objetos $A \subseteq \mathcal{G}$ es una extensión de un concepto si cumple que $A'' = B' = A$. Respectivamente con $B \in \mathcal{M}$, $B'' = A' = B$ El algoritmo simplificado es el siguiente [12]:

1. Inicializar conjunto de conceptos $\mathfrak{B} := \emptyset$
2. Seleccionar $A \in \mathcal{P}(\mathcal{G})$
3. Derivar A'
4. Derivar de nuevo $A'' = (A')'$
5. $\mathfrak{B} := \mathfrak{B} \cup \{(A'', A')\}$
6. Repetir de (2) a (5) para el siguiente $A \in \mathcal{P}(\mathcal{G})$ de los restantes
7. Devolver $\mathfrak{B}$, que es el conjunto de todos los conceptos

Vamos a aplicarlo a un contexto de menor tamaño que el del ejemplo anterior, para no obtener un resultado excesivamente grande.

$\mathcal{I}$	**Cine(C)**	**Ropa(R)**	**Alimentación(A)**	**Pantalla Gigante(P)**	**Electrónica(E)**
Nevada (NA)		x	x		x
Neptuno (NE)	x	x	x		
Serrallo (SE)	x	x	x	x	

Tabla 3: Tabla cruzada para un contexto de centros comerciales de Granada.

$2^{\lvert G\rvert}$	A′	A″	(A″,A′)	$\mathfrak{B}(\mathbb{K})$
∅	{C,R,A,P,E}	∅	(∅,{C,R,A,P,E})	(∅,{C,R,A,P,E})
{NA}	{R,A,E}	{NA}	({NA},{R,A,E})	({NA},{R,A,E})
{NE}	{C,R,A}	{NE,SE}	({NE,SE},{C,R,A})	({NE,SE},{C,R,A})
{SE}	{C,R,A,P}	{SE}	({SE},{C,R,A,P})	({SE},{C,R,A,P})
{NA,NE}	{R,A}	{NA,NE,SE}	({NA,NE,SE},{R,A})	({NA,NE,SE},{R,A})
{NA,SE}	{R,A}	{NA,NE,SE}	({NA,NE,SE},{R,A})	
{NE,SE}	{C,R,A}	{NE,SE}	({NE,SE},{C,R,A})	
{NA,NE,SE}	{R,A}	{NA,NE,SE}	({NA,NE,SE},{R,A})	

Tabla 4: Tabla esquemática del algoritmo paso a paso

Ejemplo 8 (Tomamos un conjunto de datos correspondientes a centros comerciales de la provincia de Granada. Los centros comerciales analizados son el Nevada Shopping, Centro Comercial Neptuno y Serrallo Plaza. Las características o propiedades a estudio son si disponen de ciertos servicios, como son Cine, tiendas de Ropa, tiendas de Alimentación, Pantalla Gigante para retransmitir eventos o tienda de Electrónica.). La lista de conceptos obtenida por el algoritmo es $\mathfrak{B}(\mathbb{K})$={(∅,{C,R,A,P,E}),({NA},{R,A,E}) , ({NE,SE},{C,R,A}) , ({SE},{C,R,A,P}) ,({NA,NE,SE},{R,A})}

3.0.2 *El teorema básico del FCA*

Sean (A_1, B_1) y (A_2, B_2) conceptos formales de un contexto formal $(\mathcal{G}, \mathcal{M}, \mathcal{I})$. Si se cumple que $A_1 \subseteq A_2$, o lo que es equivalente $B_2 \subseteq B_1$, entonces (A_1, B_1) se llama subconcepto de (A_2, B_2), y respectivamente (A_2, B_2) es un superconcepto de (A_1, B_1). En esta situación escribiremos $(A_1, B_1) \leq (A_2, B_2)$. Esto define una relación de orden sobre el conjunto de todos los conceptos formales.

Definición 19 (Retículo de conceptos [17])**.** El conjunto de todos los conceptos formales de $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ ordenados por la anterior relación de orden $\leq$ definida es un retículo de conceptos y se denota por

$$\underline{\mathfrak{B}}(\mathcal{G}, \mathcal{M}, \mathcal{I}) := (\mathfrak{B}(\mathcal{G}, \mathcal{M}, \mathcal{I}), \leq)$$

Estamos ahora en condiciones de presentar el Teorema Básico del FCA (Ganter y Wille 1999).

Teorema 1 (El Teorema Básico de los retículos de conceptos[17])**.** *El retículo de conceptos de cualquier contexto $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ es un retículo completo. Para cualquier subconjunto arbitrario*

$$\{(A_t, B_t) | t \in T\} \subseteq \mathfrak{B}(\mathcal{G}, \mathcal{M}, \mathcal{I})$$

de conceptos formales, el ínfimo viene dado por

$$\text{ínf}\{(A_t, B_t)_{t \in T}\} = \bigwedge_{t \in T} (A_t, B_t) = (\bigcap_{t \in T} A_t, (\bigcup_{t \in T} B_t)'') \tag{11}$$

y el supremo viene dado por

$$\sup\{(A_t, B_t)_{t \in T}\} = \bigvee_{t \in T} (A_t, B_t) = ((\bigcup_{t \in T} A_t)'', \bigcap_{t \in T} B_t) \tag{12}$$

Un retículo completo $\underline{L}$ es isomorfo a $\underline{\mathfrak{B}}(\mathcal{G}, \mathcal{M}, \mathcal{I})$ si y solo si existen funciones $\overline{\gamma} : \mathcal{G} \to L$ y $\overline{\mu} : \mathcal{M} \to \mathcal{L}$ tales que $\overline{\gamma}(\mathcal{G})$ es denso-supremo y $\overline{\mu}(\mathcal{M})$ es denso-ínfimo en $\underline{L}$ y

$$g \mathcal{I} m \iff \overline{\gamma}(g) \leq \overline{\mu}(m)$$

En particular $\underline{L} \cong \underline{\mathfrak{B}}(\mathcal{G}, \mathcal{M}, \mathcal{I})$

La primera parte del teorema precisa unas fórmulas determinadas para el cálculo del supremo y del ínfimo de un conjunto de conceptos arbitrario.

La segunda parte del teorema proporciona (además de otra información) fundamentos para saber si un retículo de conceptos tiene alguna propiedad que le haga diferir de toda la Teoría de Retículos que hemos visto anteriormente. La respuesta es negativa, ya que se concluye con que cada retículo completo es isomorfo a un retículo de conceptos [16]. Esto significa que para cada retículo completo debemos de ser capaces de encontrar un conjunto $\mathcal{G}$ de objetos, un conjunto $\mathcal{M}$ de atributos, y una relación de orden I, tal que el retículo que tomemos $(L, \subseteq)$ sea isomorfo a $\underline{\mathfrak{B}}(\mathcal{G}, \mathcal{M}, \mathcal{I})$. El teorema no solo dice que esto sea posible sino que da la forma para construirlo .

Para ver esto, en primer lugar expliquemos el papel que juegan las funciones γ y μ [16]:

Sea $(\mathcal{G}, \mathcal{M}, \mathcal{I})$

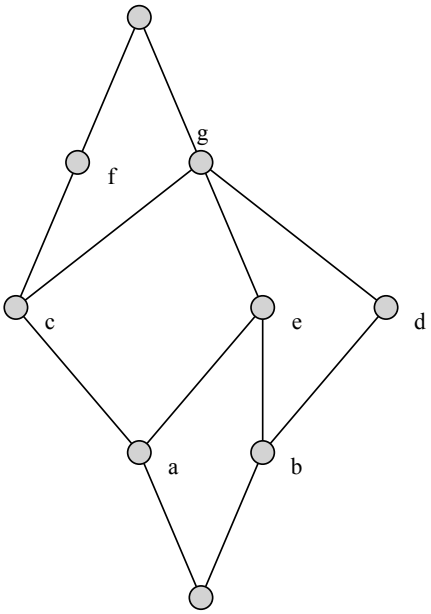
- para cada objeto $g \in G$ el correspondiente "*objeto concepto*" es: $\gamma(g) := (\{g\}'', \{g\}')$,
- y para cada atributo $m \in M$ el "*atributo concepto*" viene dado por $\mu(m) := (\{m\}', \{m\}'')$.

Haciendo uso de la Definición 17 y la Proposición 3, se comprueba que estas expresiones definen conceptos formales de $(\mathcal{G}, \mathcal{M}, \mathcal{I})$.

Tenemos que $\gamma(g) \leq (A, B) \iff g \in A$. El Teorema afirma que cada concepto formal es el supremo de todos los conceptos que hay por debajo suyo. Por tanto el conjunto $\gamma(\mathcal{G})$ es supremo-denso. De la misma manera el conjunto $\mu(\mathcal{M})$ es ínfimo-denso en $\underline{\mathfrak{B}}(\mathcal{G}, \mathcal{M}, \mathcal{I})$. Entonces cualquier conjunto supremo-denso en un retículo $\underline{L}$ puede ser tomado como el conjunto de objetos y cualquier conjunto ínfimo-denso puede ser tomado como el conjunto de atributos para obtener dicho isomorfismo.

Ilustremos esta idea con un ejemplo:

Sea $\underline{L}$ el retículo de la figura que presentamos a continuación. Además del elemento superior y el elemento unidad tiene 7 elementos. El conjunto $\mathcal{G} = \{a, b, c, d, f, g\}$ es supremo denso en L. El conjunto $\mathcal{M} = \{d, e, f, g\}$ es ínfimo denso. El Teorema Básico nos dice que el retículo de conceptos del contexto $(\mathcal{G}, \mathcal{M}, \mathcal{I})$, donde $g\mathcal{I}m$ cuando $g \leq m$ en $\underline{L}$ es isomorfo a L. Este contexto es el que se muestra en la tabla de la izquierda.



(b) Reticulo

	d	**e**	**f**	**g**
a	x	x	x	
b		x	x	x
c	x	x		
d		x	x	
f		x		x
g	x			

(a) Tabla

Figura 3: Tabla del contexto con un retículo de conceptos isomorfo al representado por el diagrama de la derecha.

Como veremos a continuación, justo este retículo coincide con el del ejemplo de los monumentos de Granada.

3.1 REPRESENTACIÓN GRÁFICA DE UN RETÍCULO DE CONCEPTOS

3.1.1 *Diagrama de líneas*

Según Ganter [17] un diagrama de líneas o diagrama de Hasse corresponde a un contexto (G, M, I) si y solo si se cumplen estas 4 cosas:

1. El diagrama corresponde al de un retículo bien definido.
2. Exactamente un nodo $\gamma(g)$ está etiquetado por cada $g \in \mathcal{G}$.
3. Exactamente un nodo $\mu(m)$ está etiquetado por cada atributo $m \in \mathcal{M}$.
4. $\forall g \in \mathcal{G},\ \forall m \in \mathcal{M}$ se tiene que $gIm \iff \gamma(g) \leq \mu(m)$.

Este diagrama de líneas se puede utilizar para leer la extensión o intensión de un concepto, se pueden seguir las aristas representadas por la jerarquía de sub y super-

conceptos en el diagrama de líneas. La extensión de un concepto puede obtenerse uniendo todos los objetos situados en el círculo respectivo y los círculos a los que se puede llegar por caminos descendentes desde este. Por otro lado, la intensión de un concepto puede obtenerse uniendo todos los objetos situados en el círculo respectivo y los círculos a los que se puede llegar por caminos ascendentes desde este.

En las figuras 4 y 5 se pueden observar los diagramas de líneas de los retículos correspondiente a nuestros ejemplos 8 y 6 respectivamente:

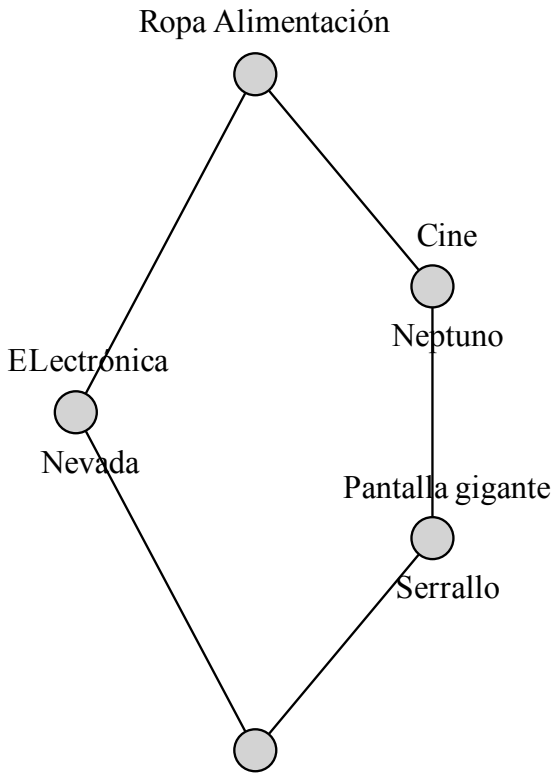


Figura 4: Diagrama de Hasse del retículo de conceptos asociado al ejemplo de centros comerciales de Granada.

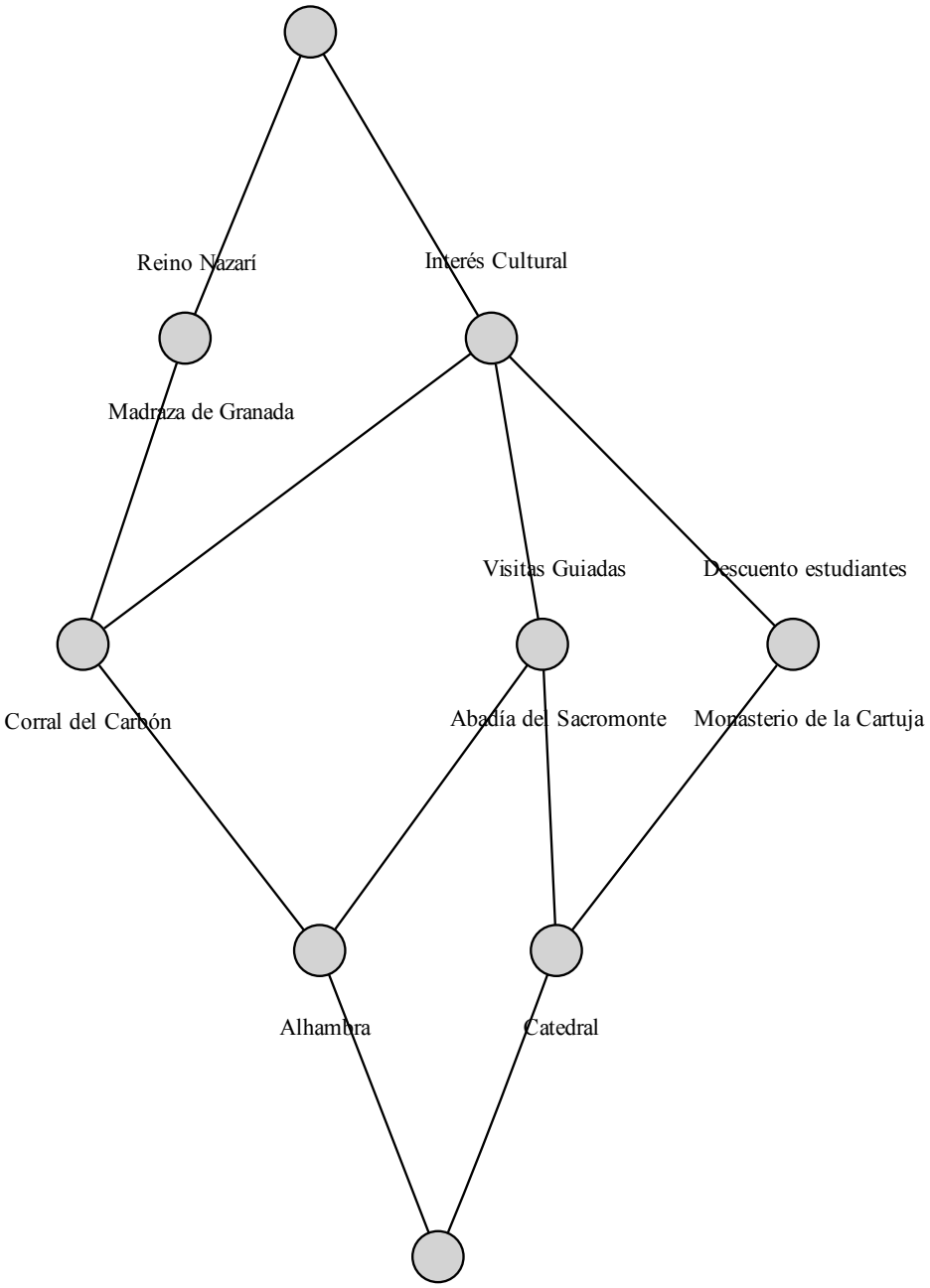


Figura 5: Diagrama de Hasse del retículo de conceptos asociado al ejemplo de monumentos de Granada.

Es decir para el concepto que representa los monumentos de con visitas guiadas nos iríamos en la figura 5 al nodo con etiqueta "Visitas Guiadas", para leer la extensión del concepto bajamos en los nodos descendientes y nos quedamos con las etiquetas correspondientes de cada uno de ellos. Por lo que la extensión sería {Abadía del Sacromonte, Alhambra, Catedral}, y la intensión se leería ascendiendo desde ese nodo y tomando las etiquetas de los atributos, en este caso la intensión sería {Interés Cul-

tural, Visitas Guiadas}, dando lugar al concepto ({Abadía del Sacromonte, Alhambra, Catedral},{Interés Cultural, Visitas Guiadas}).

En este retículo, según el diagrama, el elemento unidad o $0_{\mathfrak{B}}$ sería ({∅},{Reino Nazarí, Visitas Guiadas, Descuento Estudiantes,Interés Cultural}).

3.1.2 *Implicaciones*

Uno de los resultados adicionales que proporciona el FCA, además del retículo de conceptos, es un conjunto de implicaciones entre atributos. Estas implicaciones provocan que el FCA se use para ciertas aplicaciones que requieran estudiar las dependencias entre los atributos de los datos.

Definición 20 (Implicación [17])**.** Sea $\mathcal{M}$ un conjunto de atributos de un contexto formal $(\mathcal{G}, \mathcal{M}, \mathcal{I})$, y sean $C, D \subseteq M$. Diremos que $C \to D$ es una implicación que se cumple en $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ si cada objeto que tiene todos los atributos de C tiene también todos los atributos de D.

Proposición 6. *[17] Una implicación $C \to D$ se cumple en $(\mathcal{G}, \mathcal{M}, \mathcal{I}) \iff D \subseteq C''$.*

Para leer implicaciones a partir de un diagrama, se elige un conjunto de atributos y se toman los conceptos correspondientes a ellos, a partir de los cuales se construye el ínfimo de dichos conceptos. Los atributos que están por encima del concepto identificado por el ínfimo están implicados por los atributos previamente elegidos.

Así en nuestro ejemplo tendríamos implicaciones como {Visitas Guiadas, Descuento Estudiantes} → {Interés Cultural }. En efecto, {Interés Cultural} ⊆ {Visitas Guiadas, Descuento Estudiantes}={Catedral}′ ={Interés Cultural, Visitas Guiadas, Descuento Estudiantes}.

3.2 SOFTWARE PARA LA GENERACIÓN DE CONCEPTOS Y DIBUJO DEL RETÍCULO

Aunque posteriormente realizaremos una implementación propia de los principales algoritmos propuestos a lo largo de la historia para la generación de conceptos, ya hay algunos programas que implementan algunos algoritmos en concreto o que dibujan el retículo de conceptos. Estos son algunos de ellos:

- El primer programa desarrollado para trabajar con FCA fue *GLAD* [11]. Implementa diversos algoritmos para el minado de conceptos pero está muy limitado a conceptos con un número reducido de atributos.
- Una de las herramientas más usadas es el programa Concept Explorer (ConExp) [21], este posee las características esenciales para trabajar con el FCA. Proporciona minado de conceptos, construcción del retículo y su representación gráfica,

incluso minado de implicaciones. Es una herramienta de software libre implementada en Java. Su sitio web se puede consultar aquí ConExp [3].

- *TOSCANAJ* [48] es otra herramienta algo más avanzada para el procesamiento de conceptos. También implementada en Java. Su característica principal es que posee una interfaz para utilizar bases de datos mediante el FCA.
- Otro software libre implementado en Java es *GALICIA* [45]. Destaca por su interfaz gráfica de fácil acceso y la posibilidad de realizar test de nuevos algoritmos para la construcción de los retículos.
- Otra herramienta muy útil para trabajar con el FCA es la librería *CONCEPTS* para Python. La documentación de esta librería la podemos encontrar en su página web [4]. Es la que se ha utilizado en este trabajo para la representación gráfica de los retículos, está basada en el uso de *GRAPHVIZ* [13].

Una lista actualizada y mucho más extensa de todo el software disponible para trabajar con el FCA la podemos encontrar en su sitio web [5]. Según el objetivo del trabajo que tengamos nos será más conveniente el uso de un software u otro.

3.3 ESCENARIOS DE USO DEL FCA

Ya en la introducción comentábamos algunas de las aplicaciones que posee esta teoría, en esta sección las expondremos con mayor nivel de detalle.

Una de las principales ventajas del FCA, es que permite trabajar con conjuntos de datos proporcionando una visualización de la estructura que poseen, es por ello que puede ser utilizado de diversas maneras según qué objetivo deseemos conseguir. Al ser una herramienta para analizar datos, así como su estructura, su uso abarca gran variedad de campos relacionados con la clasificación y el análisis de datos.

En sistemas de recuperación de información y clasificación de documentos[6], [27] se utiliza el FCA para ordenar el conjunto de documentos en una estructura de retículo de conceptos. Definiendo ciertas características o atributos para los documentos, se puede elaborar una tabla cruzada y construir así dicho retículo. El problema suele estar en que el número de documentos a clasificar suele ser bastante grande, pero a partir de la petición del usuario se puede refinar la exploración del retículo para que no sea tan costoso. Es de gran utilidad ya que generalmente cuando un usuario se encuentra en un documento, los vecinos superiores e inferiores del documento en el retículo serán aquellos que más relación tengan con el documento actual.

En el ámbito de la compilación se puede usar tanto para identificar módulos dentro de un proyecto [38], como para tratar la herencia de clases dentro de un proyecto [39]. Se estructuran todas las cabeceras de los módulos o de las clases que heredan en

3 http://conexp.sourceforge.net/index.html
4 https://concepts.readthedocs.io/en/stable/
5 https://upriss.github.io/fca/fcasoftware.html

forma de retículo y se observa si el retículo tiene una estructura válida y coherente para complementar el proceso de compilación.

O su uso más habitual, en el análisis (o minería) de datos, donde se permite al usuario explorar el conjunto de datos, realizar búsquedas de diferentes tipos sobre el mismo, observar las relaciones que tienen ciertos atributos con sus elementos o refinar los resultados en secciones interesantes. Para este caso los requisitos de los algoritmos pueden ser muy diferentes. Se puede realizar primero la elaboración de la parte del retículo que consume mucho tiempo y luego permitir al usuario realizar otras consultas en la red, esto se conoce como *retículo iceberg* [41]. Por otro lado, se puede hacer una primera búsqueda en la que se almacenan los nodos más importantes y luego se van explorando los necesarios a petición del usuario.

Estos son unos de los usos más populares del FCA pero hay muchos más ejemplos como los que se pueden consultar en [15]. Todos ellos tienen en común la necesidad de encontrar un algoritmo que calcule el retículo de conceptos, y cualquier algoritmo que se requiera para esta tarea tiene cierta complejidad computacional. Por ello, dedicaremos las siguientes secciones a explorar y comparar los distintos algoritmos que existen para el cálculo del retículo de conceptos, obteniendo una instantánea global de todos ellos y de su desempeño según el ámbito en el que trabajemos.

4

ALGORITMOS PARA EL FCA

Uno de los componentes más importantes de cualquier herramienta que utilice el análisis formal de conceptos es el algoritmo que emplee para extraer todos los conceptos a partir de un determinado contexto. Su eficiencia tanto a nivel de espacio como de tiempo de ejecución, juega un papel crucial en las aplicaciones que se le pretendan dar.

Anteriormente, la mayoría de los algoritmos para el análisis de conceptos formales eran comparados y probados en pequeños conjuntos de datos. Pero con el paso de los años se han desarrollado algoritmos cada vez más sofisticados que, aprovechando ciertas propiedades teóricas del FCA, han mejorado considerablemente el desempeño de los algoritmos más clásicos. Por ello este capítulo está dedicado a la presentación de algunos algoritmos para el Análisis Formal de Conceptos.

4.1 PROPIEDADES DESEABLES DE UN ALGORITMO

A la hora de trabajar con un algoritmo para la exploración de conceptos, es importante reflexionar antes acerca de las propiedades óptimas que debe de tener este en referencia a su funcionamiento interno, para de esta manera aprovechar al máximo los recursos disponibles.

Como todo buen algoritmo que se precie, un algoritmo diseñado para esta tarea debe de cumplir con ciertas propiedades básicas como que sea finito, preciso, escalable y eficiente. Pero restringiéndonos a nuestro problema, la propiedad que más nos interesa, y en la que más nos vamos a centrar, es la eficiencia, y cuando hablamos de eficiencia nos referimos a dos tipos de eficiencia. Buscamos que sea eficiente en tiempo y eficiente en espacio (aunque este trabajo está enfocado en el análsis de la eficiencia en tiempo). El estudio comparativo que veremos en la próxima sección estará centrado en ella, ya que como comentábamos en capítulos anteriores el principal uso del FCA es aportar utilidad mediante el tratamiento y la comprensión de conjuntos de datos complejos, y para ello la eficiencia es una propiedad clave. Estará determinada en gran parte en cómo enfoca el algoritmo el cálculo de todos los conceptos del retículo. Sin olvidar, como hemos mencionado, que no se dejen de lado las propie-

dades básicas. Debe ser finito, es decir, que esté diseñado para que se termine en un objetivo determinado, en este caso el retículo de conceptos. También debe ser preciso, hay que dejar claro qué se calcula y cómo, sin posibilidad de errores. Y, por otro lado, la escalabilidad que en nuestro caso particular es una propiedad muy interesante ya que puede resultarnos muy beneficioso que el algoritmo se comporte de igual o mejor manera conforme el conjunto de datos que recibe como entrada crece.

Además, teniendo en cuenta que el proceso de generar un retículo es algo iterativo, y puede verse como un desarrollo o expansión de nodos hasta completar el grafo (retículo), lo que particularmente nos gustaría que cumpliese el algoritmo es que se calcule cada punto del espacio de búsqueda tan solo una sola vez. Esta idea es algo a lo que los primeros algoritmos no daban mucha importancia. Se repetía el mismo proceso iterativo sin tener en cuenta que se recalculaban los mismos conceptos la mayoría de las veces, ya que al expandir los conceptos de la base del grafo, muchos de los conceptos generados tienen en común varios sucesores. La repetición de conceptos es una de las características principales en las que se centraron los autores a la hora de desarrollar nuevos algoritmos, llegando a añadir atributos a los conceptos para marcar si ya se había explorado o no. Como es lógico el buen uso de esta técnica debe repercutir directamente en la eficiencia del algoritmo. Se necesitará menos espacio para su ejecución y se calcularán antes el resto de conceptos.

4.2 PRESENTACIÓN DE LOS ALGORITMOS

Desde los comienzos del FCA, se han publicado conocidos algoritmos para la tarea de la generación del retículo de conceptos; el primero fue el que Ganter[15] propuso en 1984, y a partir de ahí muchos otros surgieron intentando mejorar o modificar el trabajo que existía hasta la fecha.

A lo largo de la literatura, si recopilamos todos los algoritmos propuestos, se puede observar una principal diferencia entre ellos que va a dar lugar a dos grandes grupos: distinguiremos entre aquellos que ante un cambio en el conjunto de datos de entrada deben de volver a calcular todo el retículo de conceptos y los algoritmos que, por otro lado, solo deben modificar el retículo ya existente para añadir los nuevos cambios.

Generalmente esta caracterización establece una clasificación entre lo que conocemos como algoritmos por lotes[1] y algoritmos incrementales. Los algoritmos por lotes comienzan siempre por un conjunto de datos y a partir de ahí construyen de manera secuencial todo el retículo de principio a fin, sin posibilidad a interrumpir o modificar este proceso. Por otro lado los incrementales se caracterizan por manejar una estructura de base, que en este caso será un retículo, sobre la que se realizan actualizaciones para llegar al resultado final.

1 o algoritmos tipo *batch*

La principal desventaja de los algoritmos por lotes mencionados anteriormente es que, como ya hemos explicado, requieren la elaboración del retículo por completo desde el principio. Por ello, en el caso de que la base de datos cambie, es necesario que todo el retículo se construya nuevamente desde cero, lo que puede resultar muy ineficiente para ciertas aplicaciones como puede ser en la recuperación de información. Por ejemplo si disponemos de una base de datos de documentos, y usamos el FCA para recuperarlos, si es frecuente que se indexen nuevos documentos, necesitaremos estar construyendo todo el retículo cada poco tiempo. Dentro del enfoque por lotes los más destacables son el algoritmo de Lindig [27], el InClose [1], el algoritmo de Bordat [5] y el propuesto por Berry [26].

En cambio, los algoritmos incrementales resuelven este problema utilizando la estructura del retículo como entrada para ir actualizando el mismo. Los enfoques incrementales más populares son el de Norris[33], Dowling [9], Godin [19], Capineto y Romano [6] y AddIntent [47].

Es por ello que es muy importante que la elección del algoritmo que tomemos sea en función de la aplicación que se le va a dar y de la topología de los datos con los que se va a tratar. Ante aplicaciones con constantes cambios en los datos quizás nos interese más optar por un algoritmo incremental. Aunque como discutiremos en el análisis experimental, esto variará mucho en función de la distribución de los datos con los que estemos tratando y el algoritmo elegido.

4.3 REVISIÓN BIBLIOGRÁFICA DE ALGORITMOS

A lo largo de la literatura se han presentado muchos algoritmos dedicados a la tarea de construir el retículo de conceptos. En la próxima sección presentaremos los elegidos y analizados en este trabajo, pero antes es necesario realizar una reflexión sobre algunos enfoques que son bastante populares pero que no se han incluido en este estudio por diversos motivos.

Uno de los algoritmos más antiguos que se encuentran en la literatura es el de Dowling [9] pero no lo hemos incluido ya que el pseudocódigo que aporta el autor en el documento es prácticamente ilegible y el artículo original en el que se presenta está en francés, lo que ha dificultado aún más su comprensión.

Nourine y Raynaud [34] presentaron un algoritmo incremental para la construcción del retículo de conceptos. Este algoritmo en estudios previos sobre la materia obtuvo un rendimiento muy bajo, bastante peor que el del resto de algoritmos, por este motivo no hemos considerado necesario volver a implementarlo. Lo mismo ocurre con el algoritmo de Valtchev [44], el de Gajdos [14] y el propuesto en 1999 por Carpineto y Romano [6] (este último ya lo descartan algunos autores en sus comparativas).

Otro algoritmo con un desempeño muy malo en anteriores trabajos es el de Chein [7], que además fue uno de los primeros en la literatura. Este algoritmo se usa como

base para construir otro que es bastante popular como es el algoritmo Close-By-one [23], pero como veremos en la siguiente sección, se ha optado por implementar una mejora que surgió de ese algoritmo y fue publicada después de realizar las otras comparativas. El hecho de implementar un algoritmo que surge como la mejora de estos dos últimos algoritmos nos ha hecho descartar ambos para el estudio y quedarnos con la versión más óptima de ellos.

Por otro lado, durante la revisión bibliográfica también se recopilaron diversos algoritmos que construyen el retículo de conceptos utilizando técnicas de paralelización. Un ejemplo de estos son: el algoritmo propuesto por Olga Prokasheva [50] que utiliza el modelo MapReduce para la construcción del retículo, o el algoritmo de recursión paralelo propuesto por Krajca [22]. Su implementación se ha descartado debido a que no sería justo comparar algoritmos secuenciales con paralelizables para la misma tarea, pero sería una vía de trabajo futuro interesante el repetir este trabajo con algoritmos paralelizables.

4.4 ALGORITMOS POR LOTES

Los algoritmos por lotes para el Análisis Formal de Conceptos son aquellos que construyen el retículo de conceptos partiendo desde el inicio de manera secuencial, y que ante algún cambio en el conjunto de datos deben volver a construir el retículo por completo. En esta sección veremos los más populares y sobre qué principios se basa su funcionamiento.

Para obtener una idea más clara del funcionamiento de los algoritmos, en algunas ocasiones haremos referencia al ejemplo sencillo de los centros comerciales de Granada, cuyo contexto recordemos que era:

$\mathcal{I}$	**Cine(C)**	**Ropa(R)**	**Alimentación(A)**	**Pantalla Gigante(P)**	**Electrónica(E)**
Nevada (NA)		x	x		x
Neptuno (NE)	x	x	x		
Serrallo (SE)	x	x	x	x	

Tabla 5: Tabla cruzada para un contexto de centros comerciales de Granada.

Comenzaremos por uno de los primeros algoritmos que encontramos en la literatura, a partir del que a base de mejoras y modificaciones se fueron apareciendo el resto.

4.4.1 *Algoritmo de Ganter*

En 1984, Ganter [15] diseñó el primer algoritmo en la historia del FCA, llamado *NextClosure*, basado en la idea de que cada concepto está únicamente determinado por su extensión e intensión.

En primer lugar, Ganter en su estudio apreció que los operadores de derivación eran operadores de clausura:

Definición 21 (Operador de clausura[15])**.** Sea $\mathcal{P}(M)$ el conjunto de todas las partes de M. Llamamos operador de clausura [2] a una función:

$$\overline{*} : \mathcal{P}(M) \to \mathcal{P}(M)$$

que es extensiva, monótona e idempotente, esto es, para todo $A, B \subseteq M$:

1. $A \subseteq \overline{A}$
2. $A \subseteq B \Longrightarrow \overline{A} \subseteq \overline{B}$
3. $\overline{\overline{A}} = \overline{A}$

Definición 22 (Sistema de clausura[15])**.** A la colección de todos los conjuntos cerrados de un operador de clausura se le llama sistema de clausura.

Si nos remontamos a la Proposición 3, vemos que los operadores doble prima $<''>$ son operadores de clausura tales que $'' : 2^{|\mathcal{G}|} \to 2^{|\mathcal{G}|}$ y $'' : 2^{|\mathcal{M}|} \to 2^{|\mathcal{M}|}$.

Utilizando que por definición (A, B) es un concepto si y solo si $A = B'$ y $B = A'$, si aplicamos dichos operadores $A'' = (A')' = B' = A$ vemos que todo A que sea la extensión de un concepto formal cumple la definición de conjunto cerrado (análogo para $B'' = B$). Concluimos que las extensiones e intensiones que forman un concepto son los sistemas de clausura de los operadores dobles de derivación.

Con este argumento Ganter desarrolló un algoritmo cuyo objetivo era partir del conjunto de objetos más pequeño de todos, e ir calculando el cierre de cada conjunto para así obtener todas las extensiones de los conceptos existentes. Las intensiones vendrían unívocamente determinadas por dichas extensiones. Pero para ordenar los subconjuntos de objetos y de atributos necesitó establecer el orden lexicográfico.

Definición 23 (Orden lexicográfico[15])**.** Sea $M = \{1, 2, ..., m\}$ un conjunto ordenado linealmente. Sean $A, B \subseteq M$, con $A = \{a_1, ..., a_j\}$, $B = \{b_1, ..., b_k\}$ definimos:

$$A < B \Longleftrightarrow (a_1 < b_1) \vee (a_1 = b_1 \wedge \{a_2, ..., a_j\} < \{b_2, ..., b_k\})$$

2 También conocido como cierre de un conjunto

Así siendo $A = \{1,2,3\}$ y $B = \{2,3,5\}$, se tendría $A < B$.

Siguiendo este orden el algoritmo comienza examinando el conjunto formado por el mayor objeto de $\mathcal{G}$, y termina cuando el cierre generado canónicamente es igual a $\mathcal{G}$, esto es $A'' = \mathcal{G}$. Y en cada iteración calcula el cierre de la extensión formada A, y comprueba si es la extensión de un concepto. En cuyo caso añade el concepto (A'', A'). Para la implementación cogeremos la versión del algoritmo que implementó Sergei Obiedkov en su artículo, veamos su pseudocódigo [25]:

Algoritmo 1: Pseudocódico del algoritmo *NEXTCLOSURE*.

Entrada: Un contexto$(\mathcal{G}, \mathcal{M}, \mathcal{I})$
Salida: Un retículo de conceptos L

```
1  g := max(G)
2  L := ∅
3  A := ∅
4  mientras A ≠ G hacer
5  |  A := A ∪ {g} \ {h|h ∈ A&g < h}
6  |  si {h|h ∈ A'' \ A&h < g} == ∅ entonces
7  |  |  L := L ∪ {(A'', A')}
8  |  |  g := max({h|h ∈ G \ A''})
9  |  |  A := A''
10 |  fin
11 |  en otro caso
12 |  |  g := max({h|h ∈ G \ A&h < g})
13 |  fin
14 fin
15 return L;
```

Para mejorar su rendimiento, vemos que al principio (algoritmo 1 - línea 6) Ganter incluyó una condición para comprobar si el cierre que se iba a calcular estaba ya dentro de la lista de conceptos, y así evitar calcularlo nuevamente. Es lo que se conoce como test de canonicidad.

Definición 24 (Canonicidad [15])**.** Sea A un subconjunto de $\mathcal{G}$, y $g \in \mathcal{G}$. La generación de A'' se considera canónica si $A'' \setminus A$ no contiene a g.

Bajo estas condiciones, suponiendo que la generación de A'' es canónica (y A'' no es igual a $\mathcal{G}$), el siguiente conjunto a examinar se obtiene a partir de A'' como sigue:

$$A'' \cup \{g\} \setminus \{h|h \in A'' \ \& \ g < h\} \ \ con \ g = max(\{h|h \in \mathcal{G} \setminus A''\})$$

Si por el contrario no es canónica el siguiente conjunto a examinar se consigue de una manera similar, pero el objeto a añadir debe de ser menor que el mayor de A:

$$A \cup \{g\} \setminus \{h | h \in A \ \& \ g < h\} \ \ con \ g = max(\{h | h \in \mathcal{G} \setminus A \ \& \ h < max(A)\})$$

Este algoritmo propuesto por Ganter tiene una eficiencia teórica de $O(|\mathcal{G}|^2 \times |\mathcal{M}| \times |L(\mathcal{G}, \mathcal{M}, \mathcal{I})|)$ [25].

Aplicando el algoritmo a nuestro ejemplo 5, obtendríamos la siguiente traza y su correspondiente salida:

Iteración	A	{g}	(A",A')	¿Canónica?	L
0					add({∅},{C,R,A,P,E})
1	∅	{SE}	$(\{SE\}, \{C, R, A, P\})$	Sí, A={SE},g=NE	add($\{SE\}, \{C, R, A, P\}$)
2	{SE}	{NE}	$(\{NE, SE\}, \{C, R, A\})$	Sí, A={NE,SE}, g=NA	add(($\{NE, SE\}, \{C, R, A\}$))
3	{NE,SE}	{NA}	$(\{NA\}, \{R, A, E\})$	Sí, A={NA}, g=SE	add(($\{NA\}, \{R, A, E\}$))
4	{NA}	{SE}	$(\{NA, NE, SE\}, \{R, A\})$	No, A={NA}, g=NE	
5	{NA}	{NE}	$(\{NA, NE, SE\}, \{R, A\})$	Sí, A={NA,NE,SE}	add(($\{NA, NE, SE\}, \{R, A\}$))

Tabla 6: Tabla con el procedimiento del algoritmo NEXTCLOSURE paso a paso.

Vemos que es muy parecida a la traza del primer algoritmo básico que presentamos en la introducción 4, pero con la diferencia de que gracias al test de canonicidad y a la forma de construcción del siguiente conjunto A a examinar, nos ahorramos muchos cálculos innecesarios.

4.4.2 *Algoritmo de Lindig*

Unos años después Lindig [27] presentó otro algoritmo alternativo con una idea diferente. En 1999 presentó un algoritmo tipo "*de abajo hacia arriba*" cuya idea era generar el concepto unidad del retículo (que se encuentra más abajo) y recursivamente generar todos los vecinos superiores hasta generar el retículo completo. Para lograr esta tarea utilizó un árbol de conceptos que permite comprobar si un concepto ha sido ya generado anteriormente o no.

La descripción explícita de este árbol no se encuentra en su artículo original, por ello nuestra elección ha sido utilizar una lista de nodos en la que cada nodo posee a su vez otra lista con sus vecinos superiores e inferiores tal y como utilizó Obiedkov [25] para su implementación.

Para encontrar todos los vecinos superiores de un concepto e ir generando el retículo, Lindig se basó en la idea que recoge el principal teorema de su artículo [28]. Dado un concepto (A, B) que sea distinto del elemento unidad 1_L del retículo, el siguiente conjunto S contiene todos los conceptos mayores que (A, B).

$$S = \{((A \cup \{g\})'', (A \cup \{g\})') | g \notin A\}$$

Los conceptos del conjunto S son aquellos que son más grandes que (A, B) pero no necesariamente vecinos suyos. Los vecinos superiores los identificamos mediante el siguiente teorema.

Teorema 2 (Teorema de Lindig [27]). *Sea* $(A, B) \in \mathcal{B}(\mathcal{G}, \mathcal{M}, \mathcal{I})$ *y* $(A, B) \neq 1_{\mathcal{B}}$. *Entonces* $(A \cup \{g\})''$, *donde* $g \in \mathcal{G} \setminus A$, *es una extensión de un vecino superior de* (A, B) *si y solo si para todo* $y \in (A \cup \{g\})'' \setminus A$ *se cumple lo siguiente :* $(A \cup \{y\})'' = (A \cup \{g\})''$.

La prueba de este teorema se puede encontrar en la tesis original de Lindig [28].

La clave del teorema está en la monotonía del operador $''$, el cual nos asegura que al calcular la extensión de un nuevo concepto, esto es, $A \cup \{g\}$ y calcular su cierre, obtendremos la extensión de un concepto superior al anterior: $A \cup \{g\} \subseteq (A \cup \{g\})''$.

Este es el fundamento del algoritmo *NEIGHBORS* [27], desarrollado por Lindig:

Algoritmo 2: Pseudocódico del algoritmo *NEIGHBORS*.

Entrada: ((A,B), $(\mathcal{G}, \mathcal{M}, \mathcal{I})$)
Salida: Lista de vecinos del concepto

```
1  min ← G \ A
2  neighbors ← ∅
3  para g ∈ G \ A hacer
4  |  M1 ← (A ∪ {g})'
5  |  G1 ← M1'
6  |  si (min ⋂ (G1 \ A \ {g})) == ∅ entonces
7  |  |  neighbors ← neighbors ∪ {(G1, M1)}
8  |  fin
9  |  en otro caso
10 |  |  min ← min \ {g}
11 |  fin
12 |  Return neighbors;
13 fin
```

El conjunto *min* se usa para almacenar todos los elementos de $\mathcal{G} \setminus A$ que generan vecinos superiores. Inicialmente se asume que todos los elementos generan vecinos y posteriormente se eliminan aquellos que no generan vecinos.

Su eficiencia teórica es del orden $O(|\mathcal{G}|^2 \times |\mathcal{M}|)$ [27].

Consideremos el ejemplo de generar todos los vecinos superiores del concepto $(\{NE, SE\}, \{C, R, A\})$. Según el algoritmo 2:

- En primer lugar inicializamos $min = \mathcal{G} \setminus A = \{NA\}$ y $neighbors = \emptyset$ (líneas 1 y 2).
- ahora para $g \in \mathcal{G} \setminus A = g \in \{NA\}$ (línea 3-13)
  - calculamos $M_1 = (\{NE, SE\} \cup \{NA\})' = \{R, A\}$ y $G_1 = \{R, A\}' = \{NA, NE, SE\}$

- entraríamos en el primer condicional (línea 6): $(\min \bigcap(G_1 \setminus A \setminus \{g\})) = NA \cap (\emptyset) == \emptyset$
- por tanto $neighbors \leftarrow (\{NA, NE, SE\}, \{R, A\})$

- se devuelven los vecinos que serían $neighbors \leftarrow (\{NA, NE, SE\}, \{R, A\})$

Podemos comprobar que esto es correcto con el grafo del retículo 4.

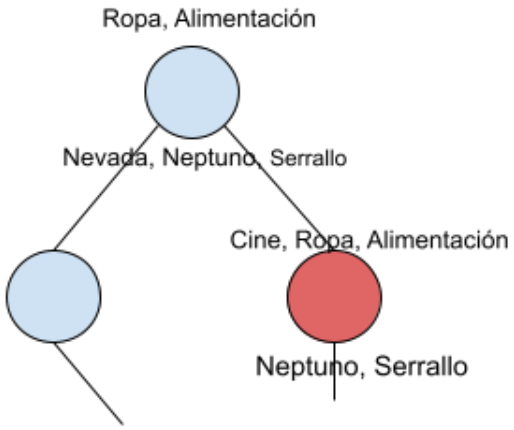


Figura 6: Como se comprueba en el grafo el vecino superior del nodo marcado en rojo es $(\{NA, NE, SE\}, \{R, A\})$.

El algoritmo *NEIGHBORS* se puede usar de forma recursiva para calcular todos los conceptos de un contexto empezando por el más pequeño $(\emptyset, \mathcal{M})$, dando lugar al algoritmo *LATTICE* [27] que nos da el retículo completo. Cada concepto c va a tener dos listas asociadas: la lista de sus vecinos inferiores x_* y la de los vecinos superiores c^*. La función *insertLookup* (algoritmo 3 -línea 4) se encarga de buscar el concepto x en la lista L de todos los conceptos, y *next* proporciona el siguiente concepto mayor a c, usando el orden léxico.

Algoritmo 3: Pseudocódigo del algoritmo *LATTICE*.

Entrada: $(\mathcal{G}, \mathcal{M}, \mathcal{I})$
Salida: Retículo de conceptos

```
c ← (∅'', ∅')
insert(c, L)
para x ∈ Neighbors(c, (G, M, I)) hacer
    intentar x ← insertLookup(x, L)
    en caso de No encontrar x → insert(x, L)
    x* ← x* ∪ {c}
    c* ← c* ∪ {x}
fin
intentar c ← next(c, L)
en caso de No encontrar c → exit
return L
```

Su eficiencia teórica en el peor de los casos es de $O(|\mathcal{L}| \times |\mathcal{G}|^2 \times |\mathcal{M}|)$ [27] ya que las operaciones de búsqueda en la lista no añaden complejidad a lo anterior. Apreciamos que comparte la misma complejidad asintótica que el algoritmo *NEXTCLOSURE* de Ganter.

4.4.3 *Algoritmo InClose*

Poco después del algoritmo *NEXTCLOSURE*, Kuznetsov implementó una modificación con el nombre de algoritmo *CLOSE-BY-ONE* [23]. Ambos tienen en común que en cada iteración trabajan con un concepto actual, a partir del cual van explorando sus vecindades en el grafo. El siguiente concepto generado es nuevo si su extensión no contiene objetos del concepto actual. Por ejemplo, supongamos que la extensión del objeto actual es {2}, si la extensión generada es {1,2,3}, el correspondiente objeto no es nuevo y puede ser descartado.

El algoritmo *IN-CLOSE* propuesto por Andrews [1] surge como una mejora del *CLOSE-BY-ONE*. Trata de formar combinaciones de atributos y comprobar si son nuevos o ya han sido generados.

El método general se puede describir de la siguiente manera [1]. Se añaden atributos, uno por uno, a la intensión actual. Cada vez que el nuevo atributo se añade, la correspondiente extensión se calcula. Esto se consigue intersecando la extensión actual A, junto con la extensión del atributo $\{j\}'$, esto es $A_{new} = A \cap \{j\}'$. Ahora con la nueva extensión hay 3 posibilidades :

1. Que sea vacía $A_{new} = \emptyset$.
2. Que sea no vacía y más pequeña que A, $A_{new} \neq \emptyset \subseteq A$.
3. Que sea el mismo conjunto $A_{new} = A$.

Cuando ocurre la opción 3, se finaliza la iteración de los atributos y se corta la recursión de esa rama, es decir, se poda esa rama y evitamos hacer cálculos innecesarios.

Un ejemplo generalizado de este proceso se muestra en [1]: para un concepto con intensión $B = \{0, 1, 3....\}$[3] ocurriría lo siguiente. Inicialmente se tiene el atributo $\{0\}$, el primer paso sería unir el atributo 1 al 0, como $\{1\} \in B$, $A \cap \{1\}' = A$, obtenemos el caso 3) y se poda esa rama de la recursión (figura 7.*(i)*). Ahora el siguiente es $j = 2$ (figura 7.*(ii)*, como $2 \notin B$ existen 2 posibilidades, la rama es válida (si A no es vacío), o la rama se poda (si A es vacío). En la figura 7.*(iii)* el atributo 3 se une al $\{0, 1\}$ y al igual que antes la rama se poda. Así se seguiría con todos los atributos hasta tener el cierre de la intensión $B = \{0, 1, 3, ...\}$.

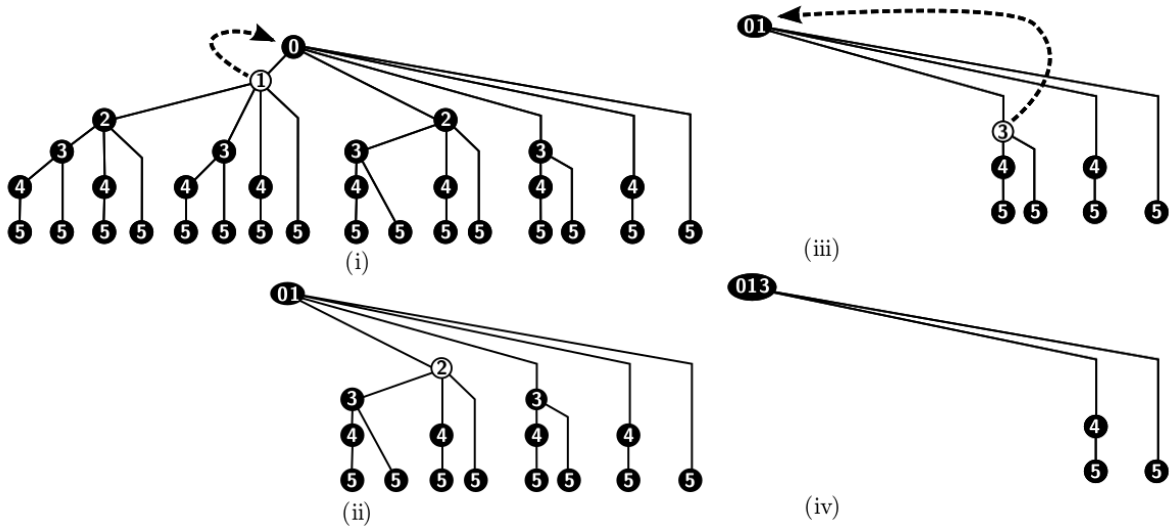


Figura 7: Proceso general de construcción de extensiones del algoritmo InClose (fuente: [1]).

Si nos ceñimos al contexto de la tabla 5, nuestra primera extensión sería $A[0] = \{\mathcal{G}\} = \{NA, NE, SE\}$ y entonces iríamos formando nuevas extensiones iterando sobre $j \in \{\mathcal{M}\} = \{C, R, A, P, E\}$.

- En el primer caso $\{j\} = \{C\} \Longrightarrow \{j\}' = \{NE, SE\}$
  $A[rnew] = A \cap \{j\}' = \{NA, NE, SE\} \cap \{NE, SE\} = \{NE, SE\}$, por lo que estaríamos en el caso 2 ya que hemos obtenido una extensión no vacía distinta de la anterior.
- Aplicaríamos el mismo proceso con cada $\{j\} \in \{C, R, A, P, E\}$ hasta terminar de construir todas las posibles nuevas extensiones de A. Si en algún momento obtuviésemos un $A[rnew] = \emptyset$ abandonaríamos esta rama y comenzaríamos la siguiente.

Este es el pseudocódigo que modela dicho comportamiento [1].

3 Nótese que no contiene el atributo 2.

Algoritmo 4: Pseudocódigo del algoritmo *INCLOSE(r,y)*.

Entrada: lista de extensiones $A_0, ...A_r$
lista de intensiones $B_0, ..., B_r$
número de concepto actual r
número de atributo actual y
un contexto formal $(\mathcal{G}, \mathcal{M}, \mathcal{I})$
Salida: Modifica los parámetros de entrada:
lista de extensiones $A_0, ...A_{rnew}$
lista de intensiones $B_0, ..., B_{rnew}$
número del siguiente concepto $r_{new} > r$
número del siguiente atributo $y_{new} > y$

```
1  r_new ← r_new + 1
2  para j = y ... |M| hacer
3  |  A[r_new] ← ∅;
4  |  para i ∈ A[r] hacer
5  |  |  si I[i][j] entonces
6  |  |  |  A[r_new] ← A[r_new] ∪ {i}
7  |  |  fin
8  |  fin
9  |  si |A[r_new]| > 0 entonces
10 |  |  si |A[r_new]| = |A[r]| entonces
11 |  |  |  B[r] ← B[r] ∪ {j}
12 |  |  fin
13 |  fin
14 |  en otro caso
15 |  |  si IsCannonical(r, j − 1) entonces
16 |  |  |  B[r_new] ← B[r] ∪ {j}
17 |  |  |  InClose(r_new, j + 1)
18 |  |  fin
19 |  fin
20 fin
```

La función *ISCANNONICAL* [1], comprueba si el cierre que hemos obtenido ha sido ya generado antes, para evitar hacer cálculos repetidos. Devuelve *true* si no encuentra $A[r_{new}]$ en las extensiones ya calculadas y *false* en caso de haberlo encontrado. La eficiencia teórica del algoritmo completo es del orden $O(|\mathcal{G}|^2|\mathcal{M}||L|)$ [1].

4.4.4 *Algoritmo de Berry*

Este algoritmo implementa una búsqueda en profundidad. En primer lugar se calculan los primeros sucesores del elemento unidad y, posteriormente, para cada sucesor,

se calcula su cobertura, esto es, todos los conceptos que son sucesores de ese concepto. Berry [26] realizó las siguientes apreciaciones antes de desarrollar su algoritmo:

Definición 25. Sea $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ un contexto formal, sean $x, y \in \mathcal{M}$. Diremos que x domina a y si $\{x\}' \subseteq \{y\}'$.

En la tabla 5 podemos observar que, por ejemplo, Pantalla Gigante domina a Cine, puesto que $\{P\}' \subseteq \{C\}'$.

Definición 26 (Rectángulo máximo [26])**.** Dado un contexto $(\mathcal{G}, \mathcal{M}, \mathcal{I})$, un rectángulo máximo de I es un conjunto X de atributos tales que $\forall x, y \in X$, $\{x\}' = \{y\}'$ y $\forall z \in (\mathcal{M} - X), \{z\}' \neq X'$. En este caso $X' = \{x\}'$ para cualquier $x \in X \subseteq \mathcal{M}$.

Un rectángulo máximo de la tabla 5 sería $\{R, A\}$. Tenemos que $\{R\}' = \{A\}'$ y $\forall z \in \{C, P, E\}$, con $Z' \neq X'$.

Definición 27 (Rectángulo máximo no dominante [26])**.** Dados dos rectángulo máximos, X e Y, diremos que X domina a Y si $X' \subset Y'$. Un rectángulo máximo X será no dominante si no hay otro rectángulo máximo al que domine.

En la tabla 5, el rectángulo máximo que hemos descrito anteriormente será no dominante puesto que no hay otro rectángulo máximo tal que $X' \subseteq Y'$.

Con esta información Berry desarrolló un teorema para calcular las intensiones del cierre de un concepto y así obtener un método para calcular todos sus sucesores.

Teorema 3. *[3] Dado un concepto (A, B), $B \cup X$ es la intensión de un concepto que pertenece al cierre de (A, B) si y solo si X es un rectángulo máximo no dominante de $I(A, \mathcal{M} - B)$. En ese caso la extensión de $B \cup X$ es $A \cap X'$.*

En nuestro ejemplo, como ya los hemos descrito anteriormente para calcular el cierre del concepto $(\{NE\}, \{C, R, A\})$, la intensión del cierre sería $B \cup X = \{C, R, A\} \cup \{P\}$ ya que $X = \{P\}$ es un rectángulo máximo no dominante de la tabla si le quitamos las columnas C, R, A tal y como dice el Teorema. De esta manera la extensión sería $A \cup \{P\}' = \{NE, SE\}$. Y concluimos que el sucesor del concepto $(\{NE\}, \{C, R, A\})$ sería $(\{NE, SE\}, \{C, R, A, P\})$.

El principal problema que tiene esta estrategia es que, en conceptos de mayor tamaño, dos o más conceptos pueden compartir sucesores. El principal objetivo de Berry era procesar cada concepto exactamente una vez, para mejorar notablemente la eficiencia del proceso. Para lograr esto utiliza la información de los rectángulos máximos. Cuando un concepto con un rectángulo máximo es procesado, y se generan todos sus sucesores, se almacena esta información para indicar que cualquier concepto que se vuelva a generar con dicho rectángulo máximo no se debe procesar de nuevo.

Para ello Berry propone el uso de dos estructuras auxiliares que mantengan esta información actualizada para todo el contexto. Estas estructuras son una tabla de do-

minancia T, la cual indica en cada casilla $T(x,y)$ el número de objetos que impiden que el atributo x domine al atributo y, en consecuencia el atributo x domina al atributo y si y solo si $T(x,y) = 0$. Y un vector D, tal que $D(x)$ que indica el número de atributos $y \in \mathcal{M}$ con $T(x,y) = 0$, esto es, el número de atributos que domina x. Un rectángulo máximo X será no dominante si y solo si $\forall x \in X$, $D(x) = |X|$. Dando así entonces respuesta a la pregunta: "¿Cuáles son los rectángulos-máximos no dominantes? " con una eficiencia del orden O(n).

Este es el proceso para crear dichas estructuras a partir de un contexto, lo recogemos en el algoritmo 5:

Algoritmo 5: Inicializar tablas T y D.

Entrada: Entrada
Salida: Salida

```
1  para x ∈ M hacer
2      D[x] ← n;
3      para y ∈ M hacer
4          para z ∈ G hacer
5              si (x,z) ∈ I and (y,z) ∉ I entonces
6                  si T[x,y] == 0 entonces
7                      D[x] ← D[x] − 1;
8                  fin
9                  T[x,y] ← T[x,y] + 1;
10             fin
11         fin
12     fin
13 fin
```

Para el ejemplo de la tabla 5, las tablas T y D serían:

	C	R	A	P	E
C	0	1	1	0	1
R	0	0	0	0	0
A	0	0	0	0	0
P	1	2	2	0	1
E	2	2	2	1	0

Tabla 7: Tabla T con la dominancia de los atributos entre sí aplicada al ejemplo de los centros comerciales.

C	R	A	P	E
3	2	2	4	3

Tabla 8: Tabla D con la información sobre la dominancia de cada atributo con respecto al resto aplicada al ejemplo de los centros comerciales.

Si analizamos estas tablas, la información más valiosa que obtenemos es que $T(x,y) = 0\ \forall x \in \{R,A\}$ lo cual nos dice que $X = \{R,A\}$ dominan al resto de atributos, y en

consecuencia es un rectángulo máximo. Ahora como $|X| = 2$ y $D(x) = 2\ \forall x \in X$ deducimos que se trata de un rectángulo máximo no dominante. Estas estructuras nos permiten identificar de forma rápida los rectángulos máximos no dominantes de una relación para poder aplicar el algoritmo diseñado por Berry.

Por último, el algoritmo *INHERIT-CONCEPTS* (6) [26] se encarga de calcular los descendientes de un concepto (A, B), e ir actualizando las tablas T y D con la información de la relación $\mathcal{I}$ para explorar únicamente conceptos nuevos.

Algoritmo 6: Algoritmo *INHERIT-CONCEPTS*

Entrada: Un contexto $(\mathcal{G}, \mathcal{M}, \mathcal{I})$
Un concepto $(A \times B)$
Un vector de conceptos marcados *marked*
Salida: Un retículo de conceptos $\mathcal{L}(\mathcal{G}, \mathcal{M}, \mathcal{I})$

```
1  Part = M − A;
2  para X ∈ Part hacer
3  |  x ∈ X;
4  |  si D[x] = |X| entonces
5  |  |  ND ← ND + X;
6  |  fin
7  fin
8  NEW ← ND para x ∈ NEW hacer
9  |  A' ← (A + X);
10 |  B' ← (B ∩ X');
11 |  L.add( A' × B');
12 |  PREUPDATE(A, X);
13 |  INHERIT − CONCEPTS(A' × B', Marked);
14 |  POSTUPDATE(A, X);
15 |  Y ← X
16 |  MARKED ← MARKED ∪ X ∪ Y;
17 fin
```

Con los algoritmos *PREUPDATE* [26] y *POSTUPDATE* [26], mantenemos la información de las tablas actualizadas. La eficiencia teórica global del algoritmo es del orden $O(|\mathcal{G}||\mathcal{M}|)$ [26].

4.4.5 *Algoritmo de Bordat*

Se trata de otro algoritmo tipo "*de arriba hacia abajo*", propuesto por Bordat [5]. Utiliza una estructura de árbol para almacenar y recuperar los conceptos. La versión del algoritmo que hemos implementado la podemos encontrar en [25] y utiliza una función *FIND* que busca si un concepto ha sido generado con una eficiencia de $O(|\mathcal{M}|)$. Bordat elaboró una proposición con la que fue capaz de describir los elementos que

existen alrededor de un concepto para así desarrollar el algoritmo que calcula los vecinos de un concepto [5]:

Proposición 7. *Si (S,T) es un predecesor de (E,F) en el árbol, (X,Y) es hermano de (S,T) en el árbol, y (V,W) es el padre de (S,T) y (X,Y) en el árbol, entonces $F \cap Y = W$.*

En el caso del ejemplo 5, esta proposición describe prácticamente todo el retículo. Sea $(S,T) = (\{NE\},\{C,R,A\})$ un predecesor de $(E,F) = (\{SE\},\{C,R,A,P\})$ tal y como se puede apreciar en el grafo 4. Tenemos que su hermano izquierdo es $(X,Y) = (\{NA\},\{R,A,E\})$, y su padre es $(V,W) = (\{NA,NE,SE\},\{R,A\})$, con $F \cap Y = \{C,R,A,P\} \cap \{R,A,E\} = \{R,A\} = W$ tal y como nos indica la proposición anterior.

En esto se basa el algoritmo *LOWER-NEIGHBORS* [25] para calcular los vecinos inferiores de un concepto.

Algoritmo 7: Algoritmo *LOWER-NEIGHBORS*

Entrada: Un concepto (A,B)
Salida: Lista de conceptos vecinos

```
1  Lattice := ∅;
2  C := B
3  g :=primer objeto en A tal que ¬(({g}' ⊆ C);
4  si no existe dicho primer objeto, g es el último elemento de A
5  mientras g! =último elemento de A hacer
6      E := {g}
7      F := {g}'
8      h := g
9      mientras h! =último elemento de A hacer
10         h.next()
11         si ¬(F ∩ {h}' ⊆ C) entonces
12             E := E ∪ {h}
13             F := F ∩ {h}'
14         fin
15         si F ∩ C = B entonces
16             Lattice := Lattice ∪ {(E,F)}
17         fin
18         C := C ∪ F
19     fin
20 fin
```

Y con esta idea, partiendo del concepto total $(\{\mathcal{G}\},\{\mathcal{G}\}')$, el algoritmo de *BORDAT* [25] calcula el retículo completo:

Algoritmo 8: Algoritmo *BORDAT*

Entrada: Un concepto (A, B) y una extensión C
Salida: Retículo de conceptos L

```
1 L = L ∪ {(A, B)}
2 LN := LowerNeightbors((A, B))
3 para (D, E) ∈ LN hacer
4     si C ∩ E = B entonces
5         C := C ∪ E
6         Bordat(((D, E), C)
7     fin
8 fin
```

Como cada concepto puede ser procesado solo una vez, la complejidad del algoritmo de Bordat es de $O(|\mathcal{G}||\mathcal{M}^2||L|)$, donde $|L|$ es el tamaño del retículo.

Usaremos el contexto de la tabla 5 para ilustrar como funciona el algoritmo 8. Partimos del objeto total que es $(\mathcal{G}, \{\mathcal{G}\}') = (\{NA, NE, SE\}, \{R, A\})$, $C_0 = \{R, A\}$ es la intensión actual. Los siguientes pasos son:

- $(\{NA, NE, SE\}, \{R, A\})$ es el concepto maximal
- $C_0 = \{R, A\}$
- Se llama a Bordat($\{NA, NE, SE\}, \{R, A\}$,$\{R, A\}$)
- $(\{NA\}, \{R, A, E\})$, $(\{NE, SE\}, \{C, R, A\})$ son los vecinos inferiores de $(\{NA, NE, SE\}, \{R, A\})$
- $C_0 \cap \{R, A, E\} = \emptyset \Longrightarrow C_1 = \{R, A, E\}$
- Se llama a Bordat($\{NA\}$,$\{R, A, E\}$,$\{R, A, E\}$)
- $(\{NA\}, \{R, A, E\})$ no tiene vecinos inferiores
- $C_1 \cap \{C, R, A\} = \{R, A\} \Longrightarrow C_2 = \{C, R, A, E\}$
- Se llama a Bordat($\{NE, SE\}, \{C, R, A\}$,$\{C, R, A, E\}$)
- $(\{SE\}, \{C, R, A, P\})$ es el vecino inferior de $(\{NA\}, \{R, A, E\})$
- $C_3 = \{C, R, A, P, E\}$
- Se llama a Bordat($\{SE\}, \{C, R, A, P\}$,$\{C, R, A, P, E\}$)
- $(\{SE\}, \{C, R, A, P\})$ no tiene vecinos inferiores
- Se añade el concepto unidad $(\{M\}', \{M\})$=$(\{\emptyset\}, \{C, R, A, P, E\})$
- Fin del proceso.

4.5 ALGORITMOS INCREMENTALES

Por otro lado tenemos los algoritmos incrementales que ante una inserción de objetos en el conjunto de datos solo deben procesar esa modificación en el retículo ya existente y no construir todo el retículo desde el inicio.

Es evidente que con este tipo de algoritmos también se puede construir el retículo completo, simplemente añadiendo objetos desde el inicio hasta tenerlos todos dentro del retículo. Este método será el que utilizaremos para establecer una comparación con los algoritmos por lotes anteriormente explicados. Tendremos una función diferente para cada algoritmo cuya sintaxis será de la forma $AddX(object, Lattice)$ cuyo cometido será el añadir el objeto "*object*" al retículo *Lattice*. Y así si llamamos a dicha función para todos los objetos de $\mathcal{G}$ de manera incremental podemos construir el retículo completo utilizando un procedimiento muy similar a este que mostramos a continuación.

Algoritmo 9: Algoritmo *CREATELATTICE*

Entrada: Un contexto $(\mathcal{G}, \mathcal{M}, \mathcal{I})$
Un retículo de conceptos L
Un objeto $g \in \mathcal{G}$ a añadir
Salida: Retículo de conceptos L.

```
para g ∈ G hacer
    Add(g, L);
fin
L es el retículo de conceptos
```

Para simplificar el proceso de creación incremental de los retículos, para todos ellos el concepto correspondiente al elemento unidad 0_L del retículo $(\{\mathcal{M}\}', \{\mathcal{M}\})$ se añade de manera manual al finalizar cada algoritmo.

4.5.1 *Algoritmo de Norris*

El algoritmo propuesto por Norris [33] se puede entender como una versión incremental del algoritmo *CLOSE-BY-ONE* [23]. El árbol de conceptos se puede construir de la siguiente manera: primero, solo está la raíz del árbol $(\{\emptyset\}, \{\emptyset\})$; después se examinan los objetos de $\mathcal{G}$ y para cada cada concepto del árbol se comprueba si el objeto en cuestión tiene todos los atributos de la intensión del concepto; si los tiene, se añade a la extensión; en caso contrario, se crea un nuevo nodo en el árbol y se marca como el hijo del nodo actual, la extensión será la extensión actual más el objeto que está seleccionado; y la intensión será la intersección de la intensión del objeto actual con la intensión del padre, después se pasa al siguiente objeto hasta recorrer $\mathcal{G}$.

Ilustremos esto con el contexto de la tabla 5. Partimos del árbol que solo contiene la raíz trivial, y queremos añadir el primer objeto $g \in \mathcal{G}, g = NA$. Tenemos que recorrer todos los conceptos del árbol pero en este primer caso no tenemos ningún concepto. Como el conjunto $\{h | h \in \mathcal{G}$ &h ya ha sido añadido al retículo &$\{g\}' \subseteq \{h\}'\}$ es vacío se añadiría el primer concepto $(\{NA\}, \{R, A, E\})$.

Figura 8: Retículo tras añadir el primer objeto $\{g\} = \{NA\}$.

Ya tendríamos la raíz del árbol que sería $(\{NA\}, \{R, A, E\})$. Ahora queremos añadir el objeto $\{NE\}$. Tenemos $(A, B) = (\{NA\}, \{R, A, E\}) \in L$. Como $\{R, A, E\} \nsubseteq \{C, R, A\} = \{NE\}'$ se crea un nuevo nodo en el árbol cuya extensión será $\{NA, NE\}$ y la intensión será $\{R, A\} = \{R, A, E\} \cap \{C, R, A\}$, además, al igual que antes se añade el nodo $(\{g\}, \{g\}') = (\{NE\}, \{C, R, A\})$ y se marca como hijo del anterior.

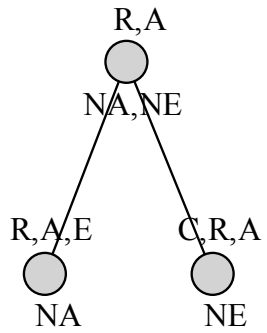


Figura 9: Retículo tras añadir el objeto $\{g\} = \{NE\}$.

Para añadir $\{SE\}$ se actuaría de la misma manera. Recorrer la lista de conceptos para comprobar si debemos modificar su extensión e intensión y añadir el nodo $(\{SE\}, \{C, R, A, P\})$.

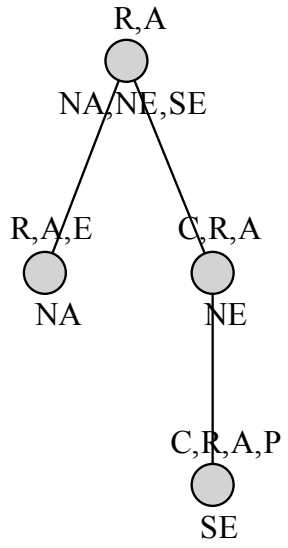


Figura 10: Retículo tras añadir el objeto $\{g\} = \{SE\}$.

Para terminar la construcción del retículo se añade el concepto correspondiente al elemento nulo del retículo $(\{\mathcal{M}\}', \{\mathcal{M}\}) = (\{\emptyset\}, \{\mathcal{M}\})$.

El pseudocódigo [25] que modela este comportamiento es el siguiente (algoritmo 10) :

Algoritmo 10: Algoritmo *AddNorris*

Entrada: Un contexto $(\mathcal{G}, \mathcal{M}, \mathcal{I})$
Un retículo de conceptos L
Un objeto $g \in \mathcal{G}$ a añadir
Salida: Retículo de conceptos L.

```
para (A, B) ∈ L hacer
    si B ⊂ {g}' entonces
        A := A ∪ {g}
    fin
    en otro caso
        D := B ∩ {g}'
        si {h|h ∈ G \ A &h added &D ⊆ {h}'}== ∅ entonces
            L := L ∪ {(A ∪ {g}, D)}
        fin
    fin
    si {h|h ∈ G &h added &{g}' ⊆ {h}'}== ∅ entonces
        L := L ∪ ({g}, {g}')
    fin
fin
```

Tiene una complejidad de $O(|\mathcal{G}|^2|\mathcal{M}||L|)$ [33] si se construye el retículo por completo.

4.5.2 *Algoritmo de Godin*

El algoritmo propuesto por Godin [19] utiliza una heurística basada en el tamaño de los conjuntos de atributos. A su vez, el algoritmo mantiene todo el tiempo actualizado el concepto unidad, al que llamaremos inf.

En su artículo original, Godin propuso un algoritmo y una mejora de dicho algoritmo para conjuntos de datos más grandes, la cual simplificaba los cálculos de los descendientes del retículo. Dicha versión mejorada es la que corresponde al siguiente pseudocódigo (algoritmo 11) [19]:

Algoritmo 11: Algoritmo *AddGodin*

Entrada: Un contexto $(\mathcal{G}, \mathcal{M}, \mathcal{I})$
Un retículo de conceptos L
El concepto unidad inf
Un objeto $g \in \mathcal{G}$ a añadir
Salida: Retículo de conceptos L.

```
si inf == (∅,∅) entonces
    inf := ({g}, {g}')
    L := inf
fin
en otro caso
    si ¬({g}' ⊆ la intension de inf) entonces
        si la extension de inf == ∅ entonces
            la intension de inf := la intension de inf ∪ {g}'
        fin
    fin
    en otro caso
        L := L ∪ (∅, la intension de inf ∪ {g}')
        inf := (∅, la intension de inf ∪ {g}')
    fin
    para i := 0 hasta máx({j|∃(A,B) ((A,B) ∈ L&|B| = j}) hacer
        C_i := {(A,B)|((A,B) ∈ L&|B| = i})
        C'_i := ∅
    fin
    para i := 0 hasta máx({j|∃(A,B) ((A,B) ∈ L&|B| = j}) hacer
        para (A,B) ∈ C_i hacer
            si B ⊆ {g}' entonces
                A := A ∪ {g}
                C'_i := C'_i ∪ {(A,B)}
                si B == {g}' entonces
                    return L;
                fin
            fin
            en otro caso
                Int := B ∩ {g}'
                si ¬∃(A1,B1)|((A1,B1) ∈ C'_|Int|&B1 == Int) entonces
                    L := L ∪ {(A ∪ {g}, Int)}
                    C'_|Int| := C'_|Int| ∪ {(A ∪ {g}, Int)}
                    si Int == {g}' entonces
                        return L;
                    fin
                fin
            fin
        fin
    fin
fin
```

La idea del algoritmo es muy sencilla: cuando se va a añadir un nuevo objeto, primero se comprueba si la intensión del objeto pertenece a la intensión del objeto unidad y, en caso afirmativo, se añade. En caso contrario se añade un nuevo nodo con la nueva intensión y a continuación, se revisan todos los nodos del árbol y se comprueba si la intensión contiene los atributos del objeto. En caso afirmativo se añade el objeto al nodo para actualizar el resto de nodos.

Veamos como comenzaría el algoritmo 10 para el contexto de la tabla 5:

- **Añadir el objeto** $g = NA$**:** $inf = \{\emptyset\}, \{\emptyset\}$
  Para este primer objeto entraríamos en el primer If (línea 1) ya que $inf == (\emptyset, \emptyset)$ y cambiaría a $inf = (\{NA\}, \{R, A, E\})$
- **Añadir el objeto** $g = NE$**:**, $inf = (\{NA\}, \{R, A, E\})$
  Como $\{g\}' = \{C, R, A\} \not\subseteq \{R, A, E\}$, añadiríamos el concepto $\{NE\}, \{C, R, A\}$ y modificaríamos el $inf = \{\emptyset\}, \{C, R, A, E\}$
  ahora calculamos los objetos que hay en el árbol (línea 16) $C_i = \{(\{NA\}, \{R, A, E\})$, $(\{\emptyset\},\{C, R, A, E\})$ y formamos las nuevas intensiones a partir de estos objetos.
  Con el C_1 se forma $intent = \{R, A\}$ y con dicha intensión formamos el concepto $(\{NA, NE\}, \{R, A\})$.
  Con el C_2 se forma $intent = \{C, R, A\}$ que dará lugar al concepto $(\{NE\}, \{C, R, A\})$.

Siguiendo este proceso iterativo se formaría el retículo completo.

Su eficiencia teórica es del orden de $O(|\mathcal{G}|^2)$ [19].

4.5.3 *Algoritmo AddIntent*

Este es uno de los algoritmos más recientes, diseñado por Sergei Obiedkov [47] y utiliza ideas simples e intuitivas para añadir nuevos objetos a un retículo.

Diremos que un concepto $(A, B) \in L$ es modificado si $B \subseteq \{g\}'$, puesto que debemos añadir $\{g\}$ a su extensión. De otra forma $B \cap g' = D \neq B$ para algún concepto $(C, D) \in L$. Si el concepto (C, D) es nuevo, entonces diremos que (A, B) es un generador de (C, D). Cada concepto tiene al menos un generador, pero puede tener varios [46].

El algoritmo AddIntent utiliza el retículo existente para encontrar el generador inmediato de un concepto, es decir, el que no tiene otro concepto entre sí mismo y el generado. Y una vez hecho esto, revisa todos los conceptos que están por encima de ese generador para añadir el objeto a su extensión.

Algoritmo 12: Algoritmo *AddIntent*

Entrada: Un contexto $(\mathcal{G}, \mathcal{M}, \mathcal{I})$
Un retículo de conceptos L
Una intensión *intent*
Un concepto generador *GeneratorConcept*
Salida: Un concepto generado.

```
1  si GeneratorConcept.Intent == intent entonces
2  |  return GeneratorConcept;
3  fin
4  GeneratorParents := GetParents(GeneratorConcept, L)
5  NewParents := ∅
6  para Candidate ∈ GeneratorParents hacer
7  |  si Candidate.Intent ⊈ intent entonces
8  |  |  Candidate := AddIntent(Candidate.Intent ∩ intent, Candidate, L)
9  |  fin
10 |  addParent := true
11 |  para Parent ∈ NewParents hacer
12 |  |  si Candidate.Intent ⊆ Parent.Intent entonces
13 |  |  |  addParent := false
14 |  |  |  salir del bucle;
15 |  |  fin
16 |  |  si Parent.Intent ⊆ Candidate.Intent entonces
17 |  |  |  Eliminar Parent de NewParents
18 |  |  fin
19 |  fin
20 |  si addParent == true entonces
21 |  |  Añadir Candidate a NewParents
22 |  fin
23 |  NewConcept := (GeneratorConcept.Extent, intent)
24 |  L := L ∪ {NewConcept}
25 |  return NewConcept;
26 fin
```

Por ejemplo, supongamos que estamos en el proceso de construcción del retículo para el ejemplo 5. Y que ya hemos añadido los objetos NA, NE. El estado del retículo sería el que se muestra en la imagen 11. Para añadir el objeto $\{g\} = \{SE\}$, se calcularía $\{g\}' = \{C, R, A, P\}$ y con esta intensión se calculan los conceptos que estarán por encima en el retículo que serán los nodos marcados en rojo. Y el concepto generador, que en este caso es el más inmediato cuya intensión contiene a $\{C, R, A, P\}$ (intersecándola con la del concepto unidad), que en este caso, mirando el grafo coincidirá con el concepto unidad $(\{\emptyset\}, \{C, R, A, P, E\})$.

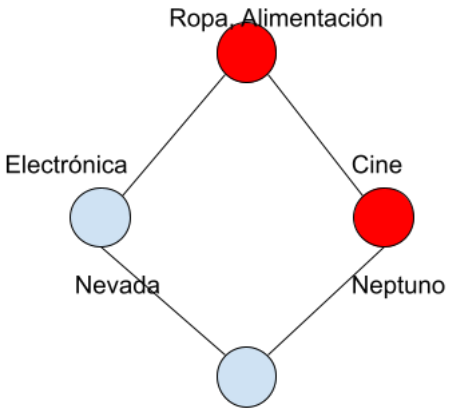


Figura 11: Retículo del contexto 5 antes de añadir el objeto $\{g\} = \{SE\}$.

Con esta información se va añadiendo el objeto g a todos los conceptos que están por encima suya, y se añaden los atributos $\{g\}'$ a todos los conceptos que están por debajo suya, empezando por el generador.

En el peor de los casos el algoritmo tiene una complejidad de $O(|\mathcal{G}|^3|\mathcal{M}||L|)$ [33].

4.6 PROPIEDADES

A modo de resumen de la sección, presentamos una tabla comparativa (tabla 9) con las características más importantes de cada algoritmo que nos ayudará a entender mejor su desempeño en diferentes tipos de escenarios. Esta tabla es una versión adaptada de la que se puede encontrar en [25], de la que hemos eliminado algunas propiedades poco útiles y a la que hemos añadido otras nuevas [4].

4 las propiedades p2,p3,p4,p6,p7 no se encontraban en la tabla del artículo.

Algoritmo	**p1**	**p2**	**p3**	**p4**	**p5**	**p6**	**p7**	**p8**
Next Closure (Ganter)		x	x	x		x		
Lindig		x		x	x		x	x
InClose		x	x	x		x		x
Inherit-Concepts (Berry)		x			x			
Bordat		x					x	
Norris	x				x			x
Godin	x				x			x
AddIntent	x				x		x	x

Tabla 9: Tabla comparativa de todos los algoritmos: p1 - incremental; p2 - por lotes; p3 - utiliza el orden léxico; p4 - en cada paso selecciona un concepto actual; p5 - utiliza alguna estructura auxiliar; p6 - emplea un test de canonicidad para evitar repetir cálculos; p7 - calcula vecinos de los conceptos; p8 - calcula las intersecciones de las intensiones que no son objetos y las intensiones de los objetos;

Parte II

EXPERIMENTACIÓN

5

ESPACIOS DE BÚSQUEDA

Cuando se trabaja con cualquier algoritmo, el comportamiento de este va a estar determinado en gran medida por el conjunto de datos que reciba como entrada. Para ver cómo se producen estos comportamientos, en este capítulo presentaremos unos conjuntos de datos sobre los que serán probados todos los algoritmos que hemos explicado en el capítulo anterior.

Usualmente, los conjuntos de datos sobre los que se prueban los algoritmos son artificiales, lo cual se debe a varias razones. La principal es que los datos más interesantes del mundo real suelen tener un alto valor comercial y las empresas los mantienen confidenciales. Y por otro lado, muchas veces queremos ver el comportamiento de los algoritmos sobre conjuntos de datos con ciertas características, y estos son difíciles de encontrar presentes en el mundo real. Para ello, los conjuntos de datos generados aleatoriamente permiten explorar fácilmente el comportamiento de escalado de los algoritmos.

Pero el hecho de solo probarlos en conjuntos de datos artificiales puede resultar contraproducente, ya que podemos provocar un estudio de los algoritmos que poco tenga que ver con su desempeño real. Así finalmente nos hemos decidido por utilizar ambos tipos de conjuntos de datos.

Por último, es necesario destacar que nos hemos preocupado de almacenar y dejar disponibles públicamente todos los conjuntos de datos utilizados para esta experimentación para su uso posterior por parte de otros investigadores interesados en la materia. Se encuentran en el repositorio de GitHub creado para el trabajo[1].

Antes de presentarlos es interesante saber en qué características nos hemos fijado para seleccionar unos u otros.

1 https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code/datasets

5.1 MÉTRICAS PARA LOS CONJUNTOS DE DATOS.

Recordamos que, tal y como vimos en capítulos anteriores, los contextos vienen representados por lo que se denominan "tablas cruzadas", que no dejan de ser tablas cuya primera fila contiene los atributos y cuya primera columna contiene los objetos, con una X en la casilla si el atributo y el objeto correspondientes están relacionados por $\mathcal{I}$. Por tanto lo que todos los conjuntos tienen en común es su estructura, una primera columna que comprenderán los objetos a estudio, y una primera fila que engloba los atributos del contexto. El factor diferencial y lo que los hace distintos entre ellos es la relación entre el conjunto de objetos y el conjunto de atributos. Esta relación, que los contextos la denotábamos por el símbolo $\mathcal{I}$, dará lugar a 3 métricas para poder comparar los distintos conjuntos.

La primera de ellas será el número de objetos, que determina el número de instancias que posee el conjunto de datos. A más objetos, más muestras tendrá nuestro conjunto de datos y más grande será en sentido "vertical". Si nos fijamos, en la mayoría de nuestros algoritmos la eficiencia teórica viene determinada de manera asintótica por $|\mathcal{G}|$.

En segundo lugar, el número de atributos $|\mathcal{M}|$ determinará también en parte la eficiencia del algoritmo. A mayor número de atributos más posibles relaciones pueden existir y por tanto más conceptos pueden existir. Pero la relación viene determinada por la cantidad de atributos que posea cada objeto, de donde obtenemos la tercera métrica.

Antes de conocer la tercera métrica, es interesante discutir una propiedad que posee la Teoría del Análisis Formal de Conceptos. Y es que la dualidad entre el conjunto de objetos $\mathcal{G}$ y el conjunto de atributos $\mathcal{M}$ nos permite intercambiarlos en sus papeles y poder aplicar todo lo visto en la primera parte del documento tomando un conjunto por el otro. Esto se traduce en que siempre que tengamos un conjunto de datos podríamos utilizar los atributos como objetos y viceversa, según nos convenga. Por ser más claros y precisos, en lo que en adelante sigue nunca alteraremos el orden entre los conjuntos y los mantendremos el conjunto de objetos como las filas del dataset y el conjunto de atributos como las columnas.

La tercera métrica que vamos a utilizar es el número medio de atributos que están relacionados con cada objeto. Denotaremos [2] por $|g'|$ al número de atributos que poseen los objetos de $\mathcal{G}$, por tanto lo podemos definir como : $|g'| = \frac{\sum_{g \in \mathcal{G}} |\{g\}'|}{|\mathcal{G}|}$. Para los conjuntos de datos artificiales este número será el mismo para todos los objetos del conjunto. Sin embargo para los conjuntos de datos reales, tomaremos la media de todos los $|\{g\}'|$ con $g \in \mathcal{G}$. Por lo general, a mayor cantidad de atributos por cada objeto, mayor número de conceptos obtendremos en nuestro retículo (salvo casos triviales).

2 Siguiendo la notación en artículos previos sobre la materia.

Si nos remontamos al ejemplo (6) de los monumentos de Granada cuya tabla cruzada (2) puede ser utilizada como un conjunto de datos: tendríamos que hay $|\mathcal{G}| = 6$ objetos, $|\mathcal{M}| = 4$ atributos y $|g'| = \frac{3+3+2+2+2+1}{6} = 2,1$ número medio de atributos por objeto.

5.2 CONJUNTOS DE DATOS SELECCIONADOS.

Para mejorar la estimación asintótica de la complejidad de los algoritmos, hemos construido contextos con conjuntos de datos artificiales con una complejidad creciente en cuanto a las 2 de las 3 métricas indicadas. Hemos decidido crear conjuntos de datos fijando en 100 el número de atributos $|\mathcal{M}|$ (ya que es el que se usa en estudios previos de la materia [25]), y variando el número de objetos $|\mathcal{G}|$, y el número de atributos por objeto $|g'|$. La manera de generar los atributos que posee cada objeto se realiza de forma aleatoria generando $|g'|$ números entre 1 y $|\mathcal{M}|$ siguiendo una distribución uniforme. Este hecho provocará que las diferencias entre las ejecuciones de un mismo algoritmo con dos conjuntos de datos con los mismos parámetros sean mínimas y obtengamos conjuntos de datos comparables conforme vayan escalando. El hecho de distribuir los atributos siguiendo una distribución uniforme se debe a que es una técnica estándar que se ha seguido en los trabajos anteriores en la materia, por lo que también nos es útil para comparar los resultados que obtengamos con los anteriores. En la figura 12 se puede observar al distribución que siguen los atributos para un conjunto de datos con $|\mathcal{M}| = 100$, $|g'| = 40$ y $|\mathcal{G}| = 100$.

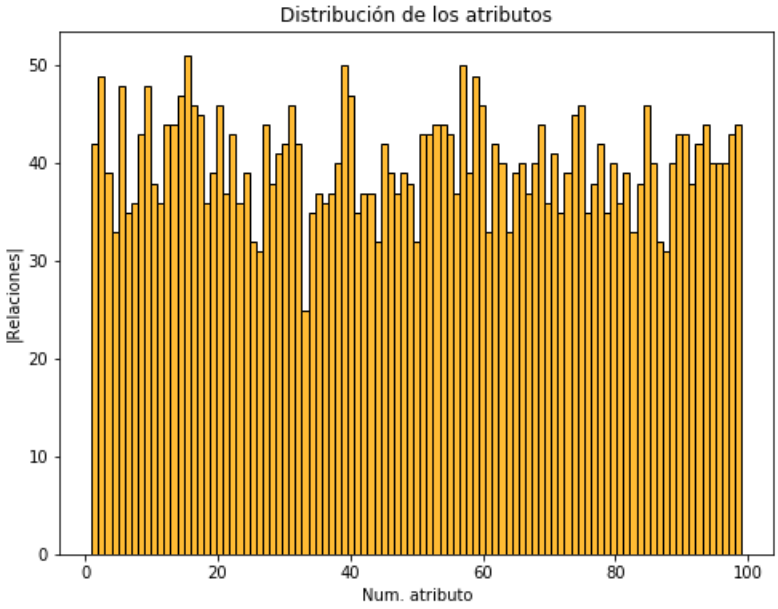


Figura 12: Histograma con el número de objetos con los que cada atributo está relacionado para el conjunto de datos artificial con $|\mathcal{M}| = 100$, $|g'| = 40$ y $|\mathcal{G}| = 100$.

Los conjuntos de datos se encuentran disponibles en https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code/datasets y están clasificados para los experimentos de la siguiente manera:

Parámetros para los conjuntos de datos artificiales (cada configuración es un conjunto de datos):

- $|\mathcal{M}| = 100$, $|g'| = 10$, $|\mathcal{G}| = \{20, 40, 60, 80, 100\}$.
- $|\mathcal{M}| = 100$, $|g'| = 20$, $|\mathcal{G}| = \{20, 40, 60, 80, 100\}$.
- $|\mathcal{M}| = 100$, $|g'| = 30$, $|\mathcal{G}| = \{20, 40, 60, 80, 100\}$.
- $|\mathcal{M}| = 100$, $|g'| = 40$, $|\mathcal{G}| = \{20, 40, 60, 80, 100\}$.

Para el primer caso, cuando $|g'| = 10$, se obtienen muy pocos conceptos para cada contexto,y el tiempo de ejecución es mínimo, por lo que crearemos conjuntos de datos de mayor tamaño para comprobar sus límites:

- $|\mathcal{M}| = 100$, $|g'| = 10$, $|\mathcal{G}| = \{150, 200, 250, 300, 350, 400, 450, 500\}$.

Lo que hace un total de 28 datasets con conjuntos de datos artificiales. Para comprobar que la generación aleatoria de la relación entre atributos y objetos no altera gravemente la forma del dataset (factor que desencadenaría en tiempos de ejecución muy diferentes para dos dataset con la misma configuración), se han generado por cada configuración 10 datasets diferentes sobre los que se ejecutarán los algoritmos. En consecuencia se han producido 280 ficheros ".csv" con conjuntos de datos aleatorios para la experimentación. Los ficheros artificiales se encuentran en la carpeta *datasets/M100*, y dentro de ella se dividen en 4 subcarpetas según el valor de $|g'|$. Dentro de ella los ficheros se llaman *G(númeroobjetos)dataset(i).csv*, donde i indica el número de dataset que tiene esa forma. Por ejemplo, los conjuntos de datos con $|\mathcal{M}| = 100$, $|g'| = 10$ y $|\mathcal{G}| = \{20\}$ se pueden encontrar en la carpeta *datasets/M100/g'10/* con el nombre *G20dataset(i).csv*, con $i \in \{1, 2, ..., 10\}$ para almacenar los 10 conjuntos diferentes con los mismo parámetros.

Por otro lado, los conjuntos de datos reales se han obtenido realizando una binarización [36] (o escalado) de los atributos no binarios para los conjuntos de datos disponibles en el repositorio de aprendizaje automático de la UCI [3] [10]. Los escogidos han sido:

- "**Breast Cancer**". Contiene información recopilada por el doctor Wolberg [29] durante sus casos clínicos en relación con el cáncer de mama. Inicialmente este conjunto tiene 699 objetos y 10 atributos. Tras preprocesarlo y binarizar los atributos se obtienen finalmente 99 atributos. Posteriormente para nuestro estudio recogimos una muestra de 500 objetos. El número medio de atributos por objeto es 10. El dataset original se encuentra disponible en https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/ y el fichero csv que

3 Universidad de California, Irvine

queda tras preprocesarlo se encuentra disponible en la carpeta datasets/real-datasets/breastcancer.csv [4] del repositorio de GitHub del trabajo.

- "**Sponges**". Contiene información relativa a un estudio realizado sobre esponjas marinas que se encuentran en la costa del mediterráneo, con atributos como el color, el tipo de espora, el número de papilas, su forma, etc. Originalmente este conjunto de datos posee 76 objetos y 45 atributos. Para adaptarlo a nuestro experimento se ha realizado una binarización y eliminación de algunos atributos para obtener finalmente un fichero con 76 objetos y 100 atributos. El número medio de atributos por objeto es 29. El dataset original se encuentra disponible en https://archive.ics.uci.edu/ml/datasets/sponge y el fichero csv tras haberlo preprocesado está en datasets/real-datasets/sponge.csv [5] del repositorio de GitHub del trabajo.

En resumen, para los conjuntos de datos reales:

- Breast-cancer.csv: $|\mathcal{M}| = 99$, $|g'| = 10$, $|\mathcal{G}| = 500$.
- Sponges.csv: $|\mathcal{M}| = 100$, $|g'| = 29$, $|\mathcal{G}| = 76$.

4 https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code/datasets/real-datasets/breastcancer.csv

5 https://github.com/miguecl97/TFG-AlgoritmosFCA/blob/master/code/datasets/real-datasets/sponge.csv

6

DISEÑO DEL SOFTWARE

En este capítulo explicaremos las pautas seguidas para implementar todo lo expuesto en las secciones previas. Comenzaremos explicando cómo hemos generado los conjuntos de datos y seguidamente los pasos seguidos para la implementación de los algoritmos. Todo el código relativo a esta implementación se encuentra disponible de manera pública en el repositorio de *GitHub* utilizado para el trabajo `https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code`.

6.1 IMPLEMENTACIÓN DE LOS ALGORITMOS

El lenguaje de programación elegido para la implementación del proyecto ha sido *C++*, ya que se trata de un lenguaje que conocemos bastante bien como producto de los años de estudio en el grado, se pueden reutilizar las funciones en el código, es fácilmente paralelizable en vista a un trabajo futuro, y la principal ventaja es que disponemos de librerías muy útiles como la *Standard Template Library (STL)* que proporcionan herramientas como la clase *'vector'* con una serie de funciones ya implementadas que nos han resultado de gran utilidad para el problema que abordamos. Aparte en artículos anteriores sobre la materia [25] también ha sido el lenguaje utilizado para la implementación por lo que podemos establecer una comparación más cercana con el trabajo que ya existía.

El código se distribuye en varios ficheros, con funciones que ayudan a la ejecución de la experimentación y donde principalmente destacan las clases *Lattice* y *Context*. En el fichero *main.cpp* se realiza la llamada a los algoritmos para realizar la experimentación del problema. En los ficheros *batchalgorithms.cpp* e *incrementalalgorithms.cpp* se encuentran varias funciones cada una correspondiente con el pseudocódigo del algoritmo implementado. Se ha optado por agrupar todos los algoritmos en los mismos ficheros para aumentar la consistencia del proyecto y reducir el número de clases empleadas. Por último el fichero *utilities.cpp* contiene varias funciones auxiliares que resuelven problemas muy específicos que han ido surgiendo a lo largo del proyecto.

Conceptos formales

La principal estructura con la que se trabaja en todo el programa es la de concepto formal, decidimos implementarla como un par de vectores de números. Dichos números corresponden al objeto o atributo que ocupa esa posición en los vectores de atributos u objetos.

```
typedef pair<vector<int>, vector<int>> formalConcept;
```

Clase Contexto

Esta clase implementa lo que es un contexto formal $(\mathcal{G}, \mathcal{M}, \mathcal{I})$ mediante un conjunto de datos (tabla) cuya primera columna corresponde a los objetos del contexto, mientras que la primera fila contiene el nombre de los atributos del mismo. El resto de casillas construyen la tabla cruzada que indica la relación $\mathcal{I}$ entre objetos y atributos, apareciendo un '1' en la casilla (i, j) si el objeto número i tiene el atributo número j.

Esto lo implementamos en nuestro código definiendo los siguientes atributos para la clase Contexto:

```
size_t nObj, nProp;
vector<string> attributes;
vector<string> objects;
vector<vector<bool>> table;
```

Además de eso implementamos una serie de funciones que nos permiten consultar estos atributos desde fuera de la clase. También en esta clase se implementan herramientas importantes para los algoritmos como los operadores de derivación para objetos y atributos mediante las funciones:

```
    void objectPrime(vector<int> &objset, vector<int> &objPrime);
    void attributePrime(vector<int> &attrset, vector<int> &attributePrime);
```

Clase Retículo

En la clase Retículo implementamos la estructura de retículo de conceptos mediante un vector de nodos, donde cada nodo contendrá la información relativa a un concepto. Por ello los atributos de la clase Lattice son:

```
int count;
bool was_inserted;
int last_position_index;
vector<Node> concepts;
```

Los nodos del retículo están definidos mediante la siguiente estructura. Contienen el concepto y información relativa a los vecinos del nodo tanto superiores como inferiores, así como el índice en el que se ha insertado en el retículo:

```
struct Node{
    int index;
    formalConcept c;
    pair<vector<int>,vector<int>> lowerUpperNeighbors;
}
```

Además contiene funciones para insertar, buscar y reemplazar elementos del retículo, así como imprimirlos por pantalla o en un archivo.

6.2 OBTENCIÓN DE LOS CONJUNTOS DE DATOS

Como se mencionó en la sección anterior los conjuntos de datos reales han sido obtenidos del repositorio de la Universidad de California para *'machine-learning'* disponible en su página web [1]. Una vez descargados encontramos un problema en ellos y es que la mayoría de sus variables son categóricas o numéricas, y para aplicar los algoritmos de FCA necesitamos que todas estas sean variables categóricas nominales con dos posibles valores, 'sí' o 'no'. Ya que es la forma en la que implementamos el contexto en al tabla cruzada.

Para realizar este preprocesamiento se ha utilizado el lenguaje de programación *PYTHON* (Python Software Foundation, https://www.python.org/) junto con la librería *PANDAS* [31]. Los conjuntos de datos reales son los disponibles ambos en la carpeta *'datasets'* del repositorio que hemos utilizado para el trabajo. Y los scripts de python utilizados para su obtención y preprocesamiento están en la carpeta *'utils'*.

Por otro lado, para los contextos artificiales, hemos implementado una función en el proyecto de $C++$ la cual construye un contexto formal con un número de objetos, atributos, y número de atributos que va a tener cada objeto como datos de entrada. Utilizando la clase *uniform_int_distribution* generamos un número aleatorio en el intervalo $[1, |\mathcal{M}|]$ siguiendo una distribución uniforme. Repetimos dicho proceso $|g'|$ veces por fila y obtenemos el conjunto de datos deseado.

```
Context generate(int nObj, int nProp, int nGPrime, int filenumber){
    vector<vector<bool>> mat(nObj, vector<bool>(nProp, false));
    const int range_from  = 1;
    const int range_to    = nProp;
    std::random_device                  rand_dev;
    std::mt19937                        generator(rand_dev());
    std::uniform_int_distribution<int>  distr(range_from, range_to);

    for(auto &row : mat){
        for(int i=0; i<nGPrime;i++){
```

1 https://archive.ics.uci.edu/ml/datasets.php

```
            int random =distr(generator);
            while(row[random]==1){
                random = distr(generator);
            }
            row[random]=1;
        }
    }

    //...Se omite el proceso de volcar el contexto en un .csv..

    return Context(mat);
}
```

6.3 LIBRERÍAS AUXILIARES

Estas son las librerías más destacables que hemos necesitado a la hora de realizar el trabajo:

- Para generar los atributos que posee cada objeto y que esta generación aleatoria siga una distribución uniforme, hemos utilizado la clase uniform-int-distribution [2] de la librería <random> [3] con la que podemos obtener valores aleatorios dentro de un intervalo $[a, b]$ y uniformemente distribuidos.
- Para la medición del tiempo de ejecución hemos utilizado la librería <chrono> [4] de la STL. Tomando tiempos al inicio y al final de cada algoritmo obtenemos el tiempo de ejecución que tarda en obtener el retículo de conceptos, con una precisión de milisegundos.
- La librería *pandas* [5] de *PYTHON* ha sido necesaria pra el preprocesamiento de los conjuntos de datos reales.
- Por último, la librería *concepts* [6]. Otra librería de *PYTHON* que nos permite trabajar con el FCA, la hemos utilizado para comprobar que los resultados de los algoritmos eran correctos y generar la representación gráfica de algunos retículos.

6.4 VALIDACIÓN DEL SOFTWARE

Las pruebas de unidad nos permiten comprobar el correcto funcionamiento de nuestros algoritmos. Dentro de estas se distinguen dos tipos según si nos fijamos en la

2 https://en.cppreference.com/w/cpp/numeric/random/uniform-int-distribution
3 https://www.cplusplus.com/reference/random
4 https://en.cppreference.com/w/cpp/header/chrono
5 https://pandas.pydata.org/docs/
6 https://concepts.readthedocs.io/en/stable

estructura interna del algoritmo (”pruebas de caja blanca”) o tratándolos como una caja negra para comprobar que con cierta entrada se produce cierta salida (”pruebas de caja negra”).

Durante el desarrollo del software todas las funciones programadas han sido probadas mediante pequeños tests manuales para comprobar que en los casos límite funcionan correctamente.

A modo de pruebas de caja negra, las más importantes para asegurarnos de que nuestro software está realizando su función correctamente, hemos ido recopilando pequeños conjuntos de datos que los autores proporcionan en la literatura, con la salida que estos mismos especifican. Y así hemos comprobado que tomando los mismos conjuntos de datos se obtienen exactamente los mismos conceptos que los autores indican en la literatura. A su vez, para los conjuntos de datos creados para ilustrar este documento, sus retículos los hemos obtenido utilizando la librería *concepts* de python, y posteriormente hemos comprobado que nuestros algoritmos nos proporcionan la misma salida. Los test de caja negra se encuentran disponibles en https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code/blackBoxTests.

Otra de las evidencias de que nuestro software es correcto es que en todas las ejecuciones de nuestra experimentación, todos los algoritmos proporcionan el mismo número de conceptos ante el mismo conjunto de datos de entrada .

7

RESULTADOS DEL EXPERIMENTO

Estamos ya en condiciones de comenzar la experimentación de los algoritmos con los conjuntos de datos escogidos. En primer lugar, comentaremos qué es lo que se desea conseguir con estos experimentos, seguidamente seleccionaremos las métricas que nos explicarán el desempeño de los algoritmos, en tercer lugar indicaremos los recursos de los que dispone el sistema utilizado, y por último comentaremos los resultados obtenido en conjunto.

7.1 OBJETIVOS DEL EXPERIMENTO

En un escenario ideal, la meta principal de esta experimentación es comprobar cómo se comportan los algoritmos según el conjunto de datos que se utiliza como entrada, y además verificar la eficiencia en tiempo a nivel asintótico que los autores proporcionan en cada uno de los artículos. Hay varios artículos que han tratado de replicar este procedimiento en la literatura [25], [40] o [24]. Quizás el más relevante de todos sea el primero de ellos, realizado en 2002 por Sergei O. Kuznetsov. Nuestro trabajo estará enfocado en comprobar la experimentación realizada por el autor y en el mejor de los casos obtener mejoras en la eficiencia provocadas por la mayor capacidad computacional que poseemos en ventaja a la que había al principio del milenio. Además nuestra experimentación incluye nuevos algoritmos como el de Berry o el InClose.

Por ello nuestro principal objetivo será qué algoritmos funcionan mejor según el conjunto de datos que reciban como entrada. Analizaremos la eficiencia en cuanto a tiempo de ejecución de los algoritmos, ya que en vistas a los primeros experimentos que hicimos los algoritmos no ocupaban apenas los recursos de almacenamiento del sistema. Aún en los datasets de mayor densidad había algoritmos que tardaban más de 20 minutos en ejecutarse y apenas ocupaban menos de $1GB$ de memoria RAM de almacenamiento.

Aparte de los tiempos de ejecución que se han analizado, los retículos minados por cada algoritmo en cada iteración se encuentran disponibles en la carpeta *results* del

repositorio de GitHub [1] del trabajo. Los retículos se muestran en ficheros de texto planos donde las primeras líneas de cada algoritmo incluye la medición de los tiempos de ejecución del algoritmo y seguidamente se muestra la lista de conceptos correspondiente al retículo. Siguen la misma estructura de almacenamiento en carpetas que los conjuntos de datos de entrada.

7.2 ENTORNO DE EJECUCIÓN

Los experimentos han sido ejecutados en un Intel® Core™ i5-8265U CPU @ 1.60GHz × 8 , con 7,6 Gigabytes de RAM bajo el sistema operativo Ubuntu 19.10. Los ficheros de código han sido compilados utilizando el compilador *g++* con la opción de optimización $-O2$ por ser la que mayor mejora de rendimiento en cuanto a tiempo de ejecución nos ha aportado tras realizar varias ejecuciones con diferentes conjuntos de datos de prueba.

7.3 RESULTADOS

A continuación presentaremos los resultados obtenidos midiendo el tiempo de ejecución de los algoritmos ante conjuntos de datos con diferentes parámetros. Utilizaremos gráficas para sintetizar y presentar toda la información obtenida. Las ejecuciones detalladas se pueden consultar en el apéndice B que se encuentra al final del documento.

7.3.1 *Conjuntos de datos artificiales*

- $|\mathcal{M}| = 100$ $|g'| = 10$ $|\mathcal{G}| = \{20, 40, 60, 80, 100\}$.

  En este primer experimento (gráfica 13) observamos que ante conjuntos con pocos atributos por objeto, la mayoría de algoritmos funcionan de manera similar, destacando de forma negativa el algoritmo AddIntent que, cuando el conjunto de objetos alcanza un tamaño de 100 objetos, empeora considerablemente su desempeño con respecto al resto de algoritmos. Pare hacer justicia, hemos de indicar que este comportamiento puede deberse a que la implementación del método para obtener los padres de un concepto que hemos realizado no sea la más eficiente de todas, ya que el autor no proporciona ninguna indicación para realizar dicho proceso. Por este motivo, y teniendo en cuenta que este rendimiento no mejora cuando el conjunto de datos escala, el algoritmo AddIntent será eliminado del resto de gráficas para obtener una comparación más visual.

1 https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code

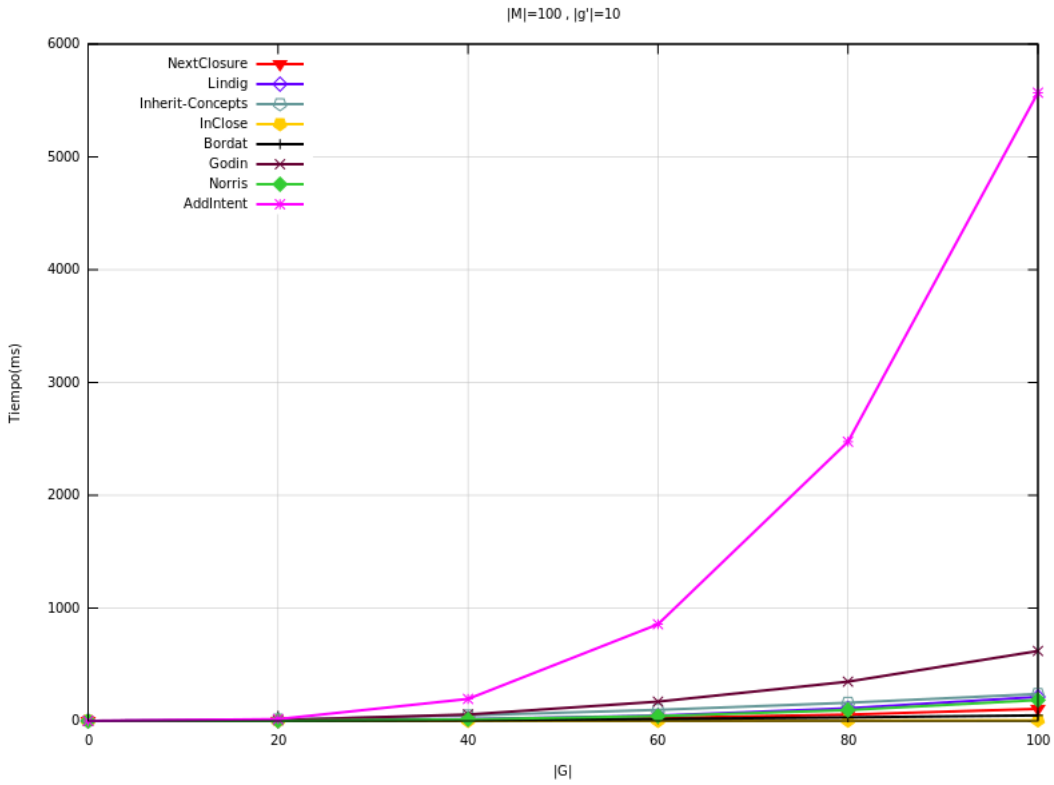


Figura 13: Gráfica comparativa con los tiempos de ejecución de cada algoritmo sobre conjuntos de datos con $|\mathcal{M}| = 100$ y $|g'| = 10$.

En la gráfica 14 se ha eliminado el algoritmo AddIntent. Vemos que para este tipo de conjuntos de datos el algoritmo de Godin tampoco tiene un buen rendimiento mientras que el algoritmo InClose es el más rápido, obteniendo tiempos realmente bajos y estables aunque aumente el número de objetos. En el siguiente experimento aumentaremos el número de objetos para ver cómo evoluciona su comportamiento.

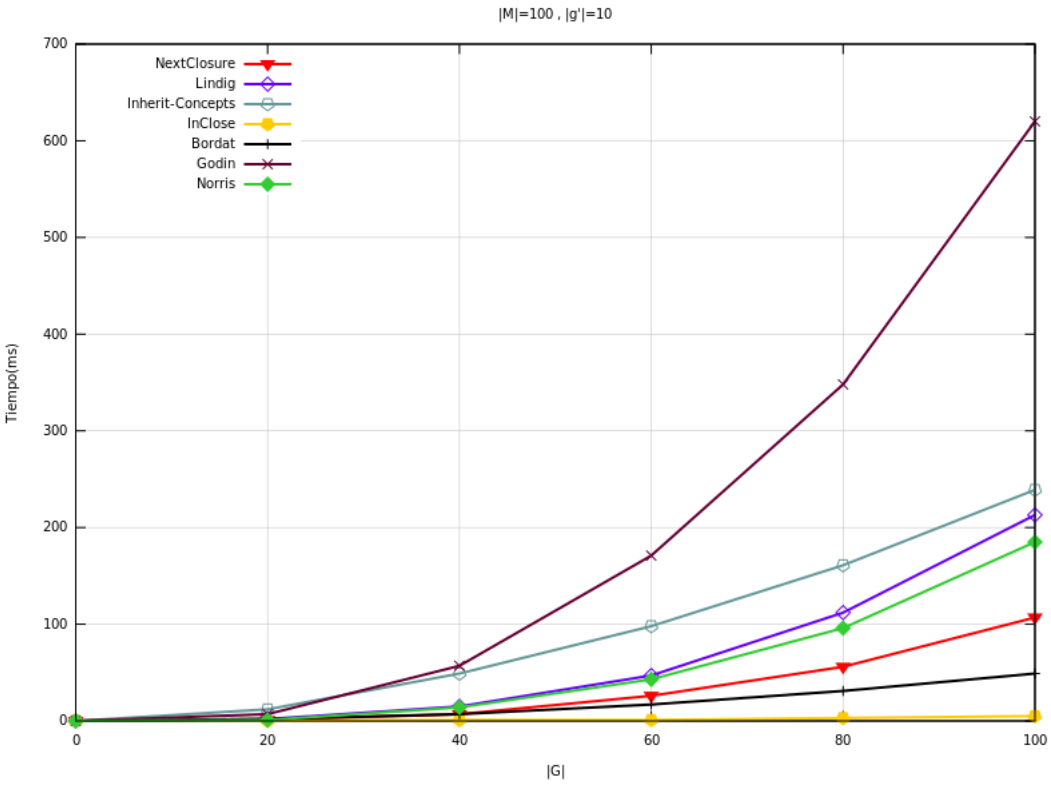


Figura 14: Gráfica comparativa con los tiempos de ejecución de cada algoritmo sobre conjuntos de datos con $|\mathcal{M}| = 100$ y $|g'| = 10$ sin AddIntent.

- $|\mathcal{M}| = 100$ $|g'| = 10$ $|\mathcal{G}| = \{100, 150, 200, 250, 300, 350, 400, 450, 500\}$.

  Bajo las mismas condiciones que el experimento anterior, manteniendo la baja densidad del dataset, cuando aumenta el tamaño del conjunto de datos las diferencias entre los algoritmos se van acentuando. Sin embargo los algoritmos InClose y Bordat se mantienen estables teniendo una eficiencia realmente buena cuando el conjunto de datos adopta un tamaño considerable. Este hecho lo podemos ver representado en la figura 15.

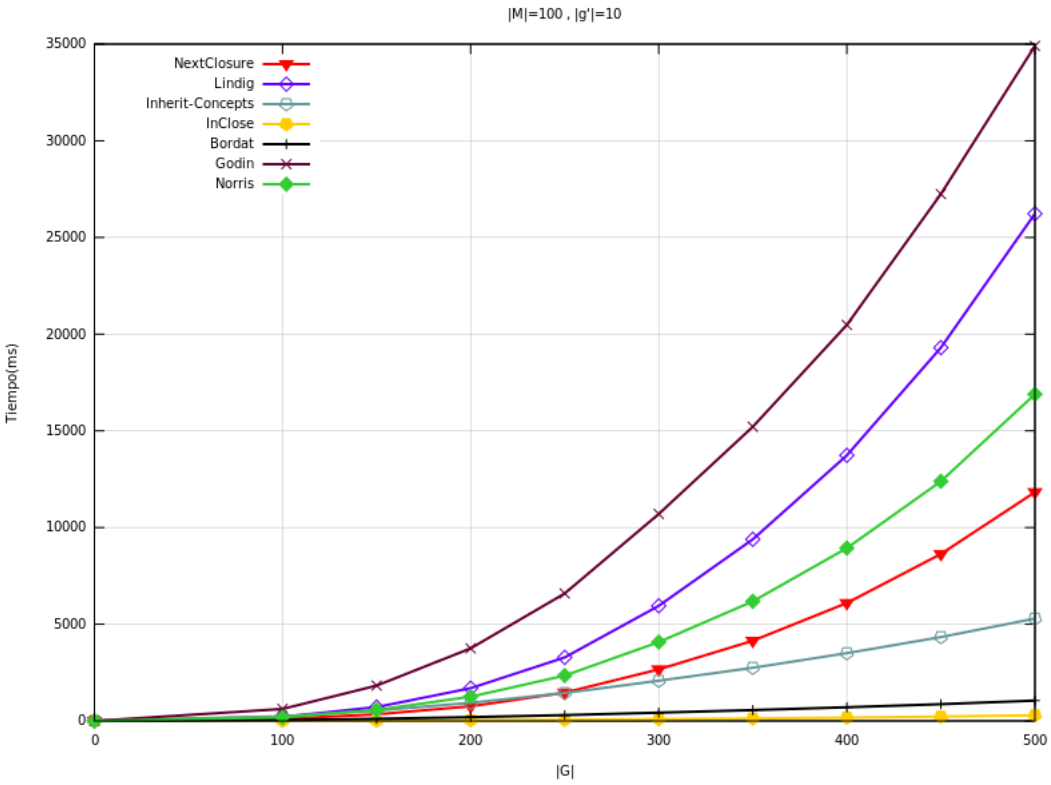


Figura 15: Gráfica comparativa con los tiempos de ejecución de cada algoritmo sobre conjuntos de datos con $|\mathcal{M}| = 100$ y $|g'| = 10$.

- $|\mathcal{M}| = 100$ $|g'| = 20$ $|\mathcal{G}| = \{20, 40, 60, 80, 100\}$.

  Para este experimento aumentamos a 20 el número de atributos que posee cada objeto. Con este cambio apreciamos cómo los algoritmos empeoran su rendimiento respecto al experimento anterior, hecho debido a que aumenta la densidad y se minan más conceptos por cada dataset. Como se puede apreciar en la gráfica 16, el algoritmo InClose sigue siendo el que mejor desempeño ofrece, pero por otro lado, el de Bordat ha empeorado su rendimiento, llegando a dar un peor tiempo de ejecución que el NextClosure. Otro hecho destacable es que el algoritmo de Lindig en el paso de 60 a 100 objetos pasa de ser el cuarto mejor algoritmo a emparejarse con el sexto en tiempos de ejecución.

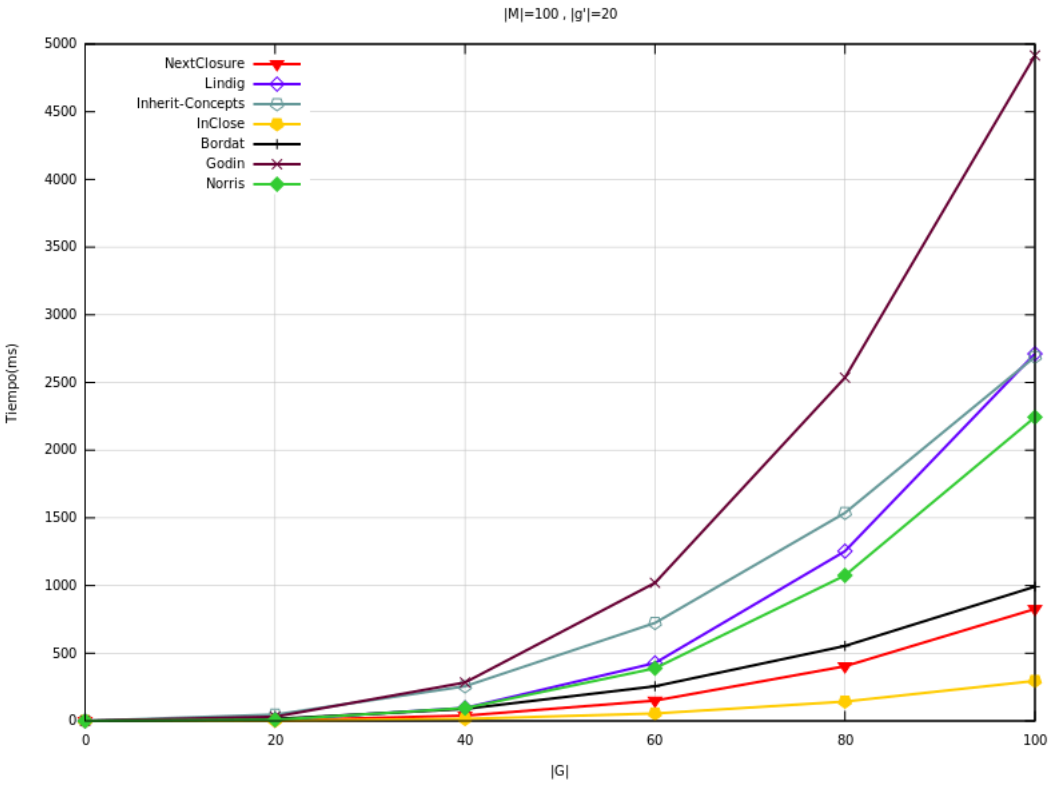


Figura 16: Gráfica comparativa con los tiempos de ejecución de cada algoritmo sobre conjuntos de datos con $|\mathcal{M}| = 100$ y $|g'| = 20$.

- $|\mathcal{M}| = 100$ $|g'| = 30$ $|\mathcal{G}| = \{20, 40, 60, 80, 100\}$.

  El siguiente caso a estudiar es cuando el número de atributos por objeto es de 30. Con este cambio el conjunto de datos aumenta considerablemente su densidad con respecto a los casos anteriores. Al aumentar su densidad se aprecian grandes cambios en los tiempos de ejecución. Como es el caso del algoritmo de Lindig, que al trabajar con un conjunto de datos de 100 objetos tiene una gran caída en su rendimiento. Por el contrario vemos que el algoritmo InClose deja de ser el que mejor rendimiento nos ofrece. En estos casos el algoritmo NextClosure se mantiene más estable en cuanto a términos de eficiencia en tiempo de ejecución. Este fenómeno puede deberse a que al aumentar el número de relaciones en la tabla, el método de poda del algoritmo InClose pierde utilidad, ya que aumentando el número de atributos se le impide podar antes la rama para pasar al siguiente concepto. Además los algoritmos de Bordat y de Norris intercambian posiciones al pasar de 80 a 100 objetos.

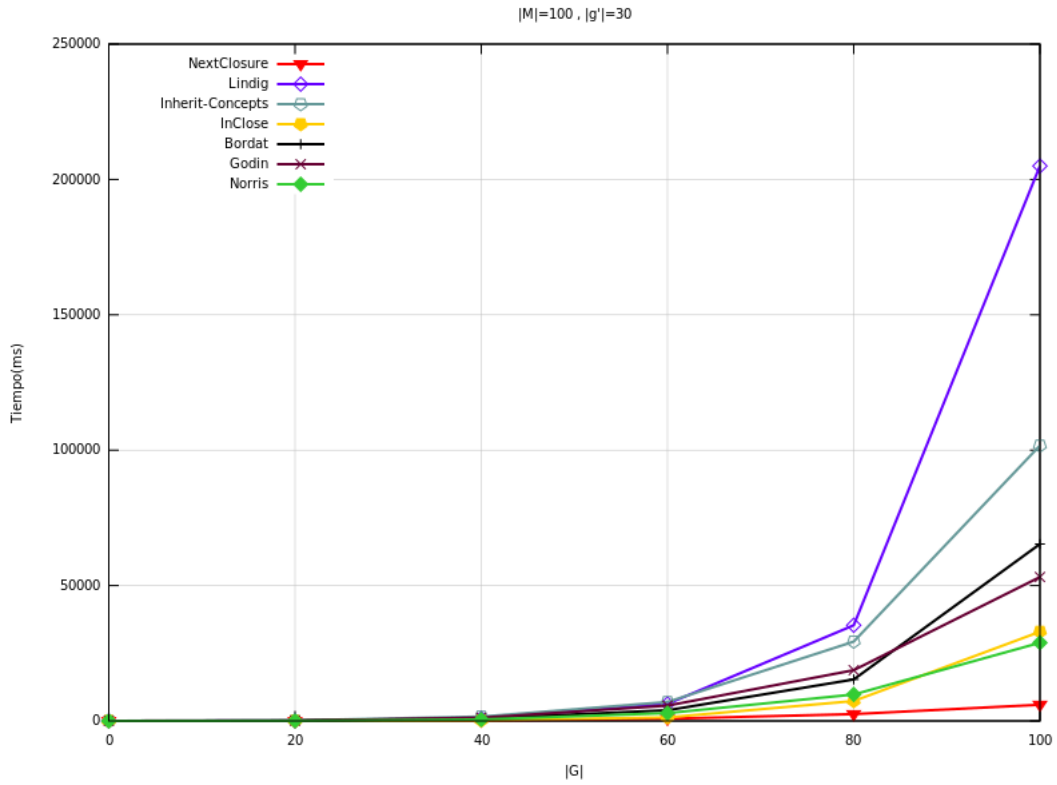


Figura 17: Gráfica comparativa con los tiempos de ejecución de cada algoritmo sobre conjuntos de datos con $|\mathcal{M}| = 100$ y $|g'| = 30$.

- $|\mathcal{M}| = 100$ $|g'| = 40$ $|\mathcal{G}| = \{10, 20, 30, 40, 50, 60, 70, 80\}$.
  Para este último experimento volvemos a aumentar en 10 el número de atributos que posee cada objeto de la tabla, terminando en 40. Este valor para el parámetro $|g'|$ provoca que se generen un gran número de conceptos para cada contexto, lo que se traduce en un aumento considerable en los tiempos de ejecución y en los recursos del sistema. Este es el motivo por el que algunos algoritmos no se siguen ejecutando para valores grandes de $|\mathcal{G}|$. El resultado de este experimento es que al llegar a la cifra de $|G| = 60$ varios algoritmos comienzan a empeorar su desempeño de manera notable. El algoritmo de Lindig que anteriormente vimos que perdía bastante eficiencia, confirma este suceso, uniéndose a él los algoritmos de Inherit-Concepts y Bordat. No nos sorprende que el algoritmo Inherit-Concepts sea el segundo que peor rendimiento ofrece ya que las tablas T y D que utiliza para mantener información alcanzan un tamaño considerable, y esto provoca que su actualización sea costosa. Por otro lado el algoritmo NextClosure se confirma como el mejor candidato a la hora de ejecutar contextos con una densidad elevada. Cuando pasamos a $|\mathcal{G}| = 80$, los algoritmos de Norris, InClose y Godin se disparan en el tiempo de ejecución, y el único que se mantiene estable es el de NextClosure. Este hecho es muy similar al que ocurrió en la experimentación realizada por Obiedkov [25].

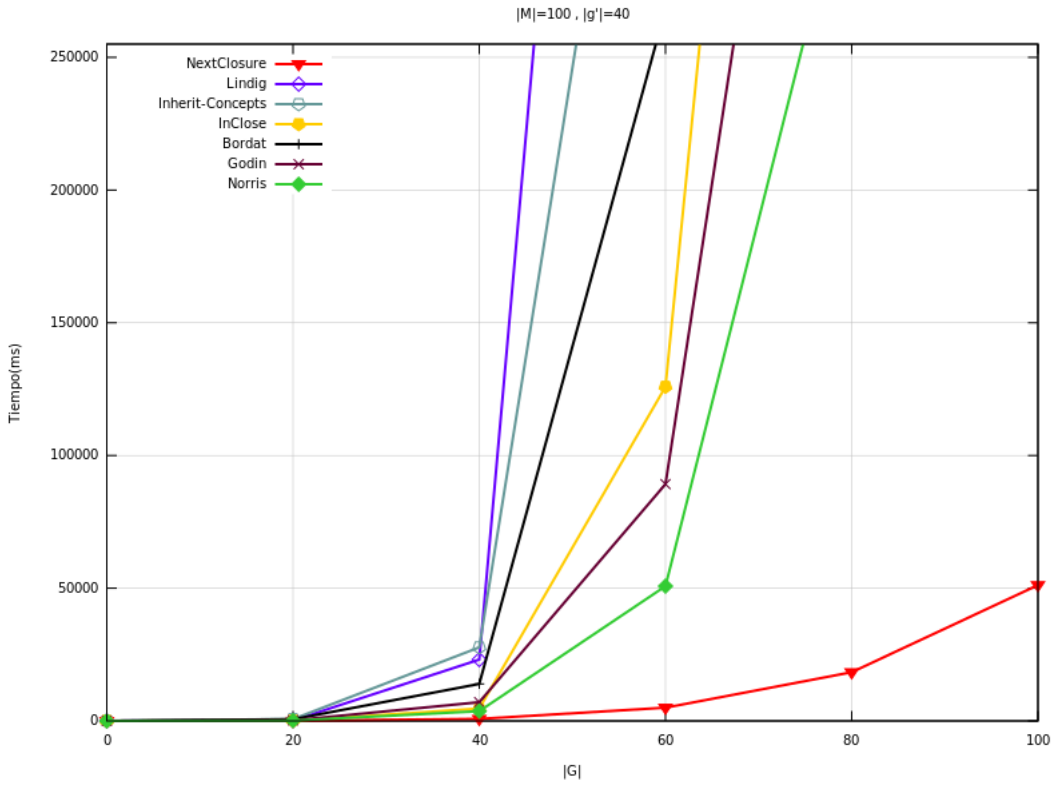


Figura 18: Gráfica de comparación de algoritmos con $|\mathcal{M}| = 100$ y $|g'| = 40$.

Ventaja en tiempo de ejecución de los algoritmos incrementales

Nos planteamos ahora que si utilizando la principal propiedad de los algoritmos incrementales podríamos obtener alguna ventaja para este último caso. Suponiendo que tengamos que añadir 20 nuevos objetos a nuestro retículo, vamos a simular que para nuestra aplicación en primer lugar construimos el retículo con los primeros 60 objetos que recibamos, y posteriormente queremos añadir 20 más, hasta llegar a 80. Para ello resumimos en una tabla el tiempo de ejecución que emplearía cada uno. En el caso del NextClosure, habría que construir primero el retículo con los primeros 60 objetos, y al añadir 20, deberíamos de volver a construir desde cero el retículo pero ahora con 80 objetos, por lo que habría que sumar ambos tiempos. Por el contrario, los incrementales van añadiendo esos 20 objetos al retículo ya construido por lo que simplemente tomamos el tiempo que tardan en construir el retículo de 80. Tal y como se observa en la tabla 10, sigue siendo más eficiente construir el retículo de nuevo por completo con el algoritmo NextClosure.

$\|g'\|$	$\|\mathcal{G}\|$	NextClosure	Norris	Godin
40	60	4990	50786	89212,9
40	80	18359,5	327035,3	542232,6
	Tiempo(ms) en añadir 20 objetos (de 60 a 80):	23349,5	327035,3	542232,6

Tabla 10: Tabla comparativa de tiempos de ejecución en ms para $|\mathcal{M}| = 100$, que tardaría cada algoritmo en añadir 20 objetos al retículo.

Para buscar un caso donde nos sea rentable la principal utilidad de los algoritmos incrementales nos debemos remontar a la figura 14. En este experimento recordamos que el algoritmo de Godin era el más lento. Pero el de Norris y el NextClosure tenían rendimientos similares, siendo el de NextClosure el más rápido a la hora de construir el retículo de 100 objetos por lotes. Veamos en la siguiente tabla 11 lo que tardaría cada algoritmo en construir de manera incremental el retículo añadiendo en cada ocasión 20 objetos al contexto.

$\|g'\|$	$\|\mathcal{G}\|$	NextClosure	Norris	NextClosure (incrementalmente)	Norris (incrementalmente)
10	20	1	1,8	1	1,8
10	40	7,6	7,88	8,6	7,88
10	60	26,7	43,88	35,3	43,88
10	80	56,2	95	91,5	95
10	100	107,1	185,1	198,6	185,1
	Tiempo total (ms):			197,1	185,1

Tabla 11: Tabla comparativa de tiempos de ejecución para una ejecución incremental con $|\mathcal{M}| = 100$ y $|g'| = 10$.

En este caso vemos como el algoritmo de Norris nos ofrece cierta mejoría en cuanto a tiempo de ejecución a la hora de realizar esta tarea, aunque no sea muy apreciable. Esto deja claro el hecho de que es necesario mejorar y actualizar los algoritmos incrementales que existen en la literatura.

7.3.2 *Conjuntos de datos reales*

- **Breast-cancer.csv**
  Para el conjunto de datos reales con información acerca del cáncer de mama los tiempos de ejecución (en milisegundos) por algoritmo que hemos obtenido han sido:

$\|g'\|$	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris
10	500	8788,2	14262,6	3193	42,7	1400,8	17671,8	8153,3

Tabla 12: Tabla de tiempos de ejecución en ms para el conjunto de datos de cáncer de pulmón.

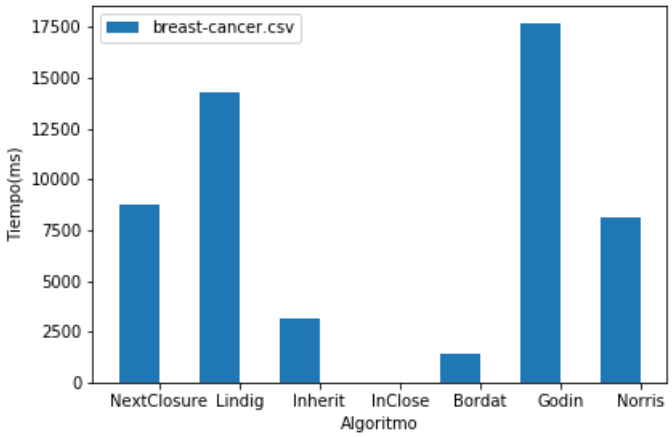


Figura 19: Gráfica de tiempos de ejecución de cada algoritmo para el conjunto de datos real

Apreciamos que el algoritmo InClose es el más rápido con diferencia con respecto a sus competidores, seguido por el algoritmo de Bordat y el InheritConcepts. Cabe preguntarnos si este comportamiento se corresponde con el estudiado en nuestros conjuntos de datos artificiales, por ello, compararemos el tiempo de ejecución obtenido con el del dataset artificial con las características más parecidas $|\mathcal{M}| = 100$, $|\mathcal{G}| = 500$ y $|g'| = 10$.

Dataset	$\lvert g' \rvert$	$\lvert \mathcal{G} \rvert$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris
breas-cancer.csv	10	500	8788,2	14262,6	3193	42,7	1400,8	17671,8	8153,3
artificial	10	500	11830,5	26228,4	5295,1	291,6	1052,8	34903,9	16893,1

Tabla 13: Tabla comparativa de tiempos de ejecución en ms para el conjunto de datos de cancer de pulmon y uno artificial.

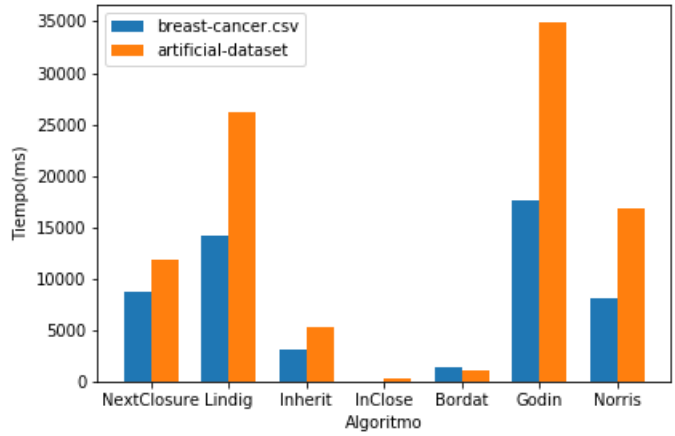


Figura 20: Gráfica comparativa de tiempos de ejecución de cada algoritmo entre el conjunto de datos real y artificial del mismo tamaño.

En la figura 20 podemos observar que el comportamiento es bastante parecido y, aunque se obtienen tiempos diferentes, los algoritmos siguen prácticamente el mismo orden. Esta diferencia de tiempos se debe en mayor medida a la distribución de los atributos para cada objeto. Ya que en nuestro dataset artificial se seguía una distribución uniforme y el conjunto breast-cancer.csv probablemente no la siga. En efecto, en la figura 21 se puede ver la distribución que siguen los atributos del dataset, que está lejos de seguir una distribución uniforme. Mientras que en la figura 22 vemos la distribución que siguen los atributos para el conjunto de datos artificial sobre el que habíamos realizado el estudio. Estas diferencias probablemente son las que provocan los cambios de rendimiento en los algoritmos.

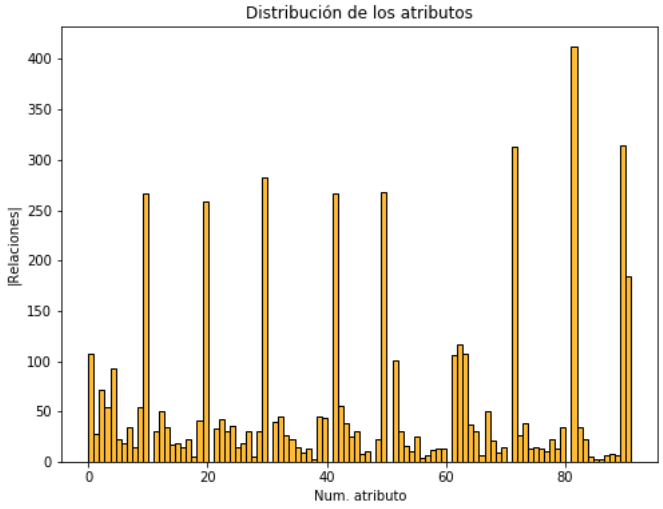


Figura 21: Histograma con el número de objetos con los que cada atributo está relacionado para el conjunto de datos breastcancer.csv .

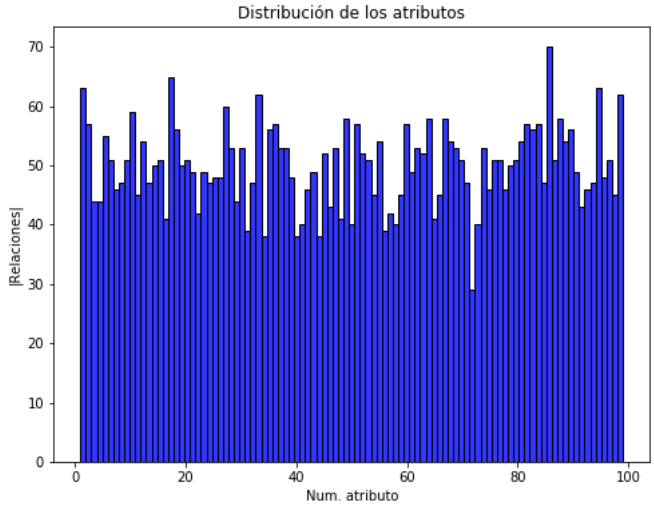


Figura 22: Histograma con el número de objetos con los que cada atributo está relacionado para el conjunto de datos artificial con $|\mathcal{M}| = 100$, $|g'| = 10$ y $|\mathcal{G}| = 500$.

- **Sponges.csv**
  Para el conjunto de datos con información de un estudio de esponjas marinas de la costa mediterránea los tiempos de ejecución obtenidos han sido los siguientes:

$\|g'\|$	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris
29	79	1328	3865,8	9590,1	309,4	7933,9	6651,4	3058,4

Tabla 14: Tabla de tiempos de ejecución en ms para el conjunto de datos de esponjas y uno artificial.

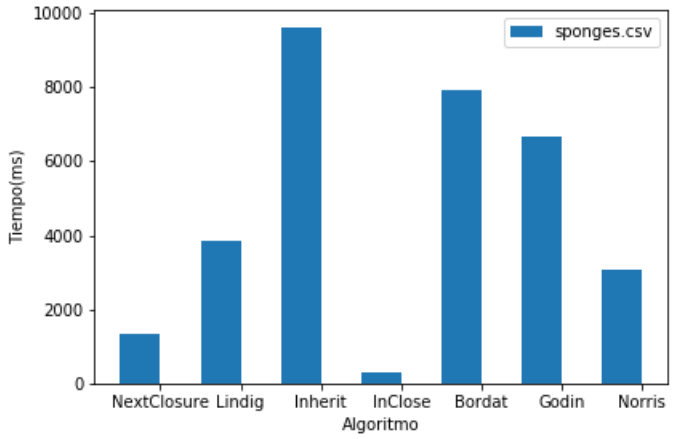


Figura 23: Gráfica de tiempos de ejecución de cada algoritmo para el conjunto de datos real

Tal y como se aprecia en la figura 23 el algorimo InClose vuelve a ser el más rápido, seguido del NextClosure. Veamos si se corresponde con las pruebas realizadas anterioremente:

Dataset	$\|g'\|$	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris
sponge.csv	29	76	1328	3865,8	9590,1	309,4	7933,9	6651,4	3058,4
artificial	30	80	2522,9	35337,1	29299,8	7346,3	15327,3	18749,6	9769,1

Tabla 15: Tabla comparativa de tiempos de ejecución en ms para el conjunto de datos de esponjas.

Ahora el resultado obtenido es bastante diferente, para el conjunto de datos artificial con la misma forma que el real el algoritmo NextClosure era el más rápido y el InClose era bastante más lento. Por otro lado destaca la diferencia de ejecución del algoritmo de Lindig que para el conjunto de datos real es 7 veces más rápido.

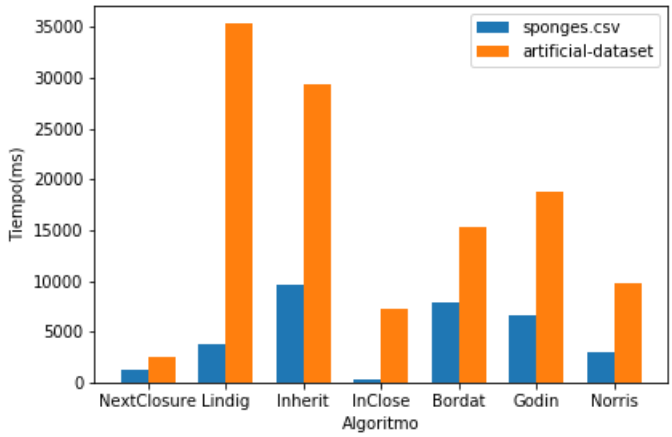


Figura 24: Gráfica comparativa de tiempos de ejecución de cada algoritmo entre el conjunto de datos real y artificial del mismo tamaño.

De nuevo estos cambios probablemente se deban a la diferencia en las distribuciones de los atributos entre los conjuntos de datos artificiales y real. Los conjuntos que hemos generado seguían una distribución uniforme (figura 26) en el número de relaciones que tiene cada atributo y el conjunto de datos de esponjas se distribuye de la siguiente manera (figura 25):

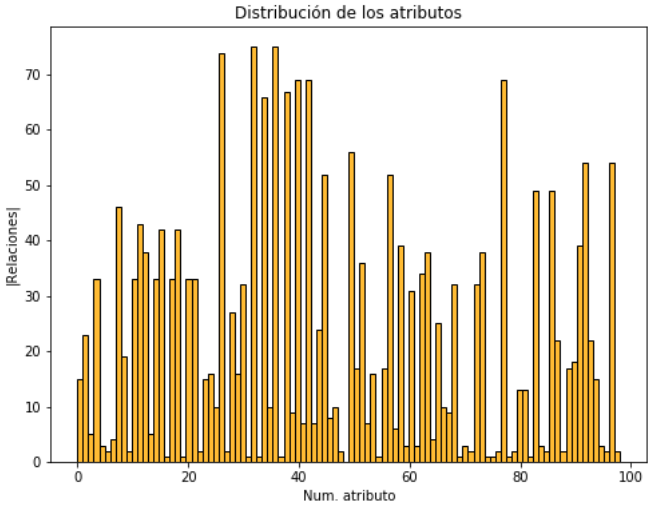


Figura 25: Histograma con el número de objetos con los que cada atributo está relacionado para el conjunto de datos sponges.csv.

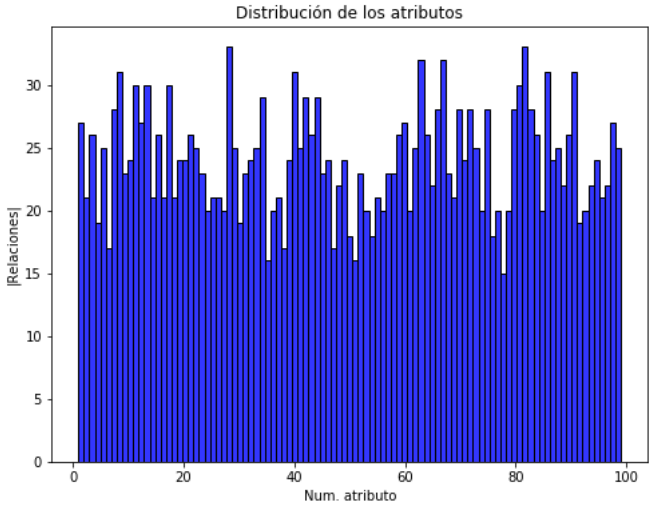


Figura 26: Histograma con el número de objetos con los que cada atributo está relacionado para el conjunto de datos artificial con $|\mathcal{M}| = 100$, $|g'| = 30$ y $|\mathcal{G}| = 80$.

En ambos conjuntos de datos reales obtenemos diferencias con respecto a la experimentación que se ha hecho anteriormente con los datos artificiales. La explicación que encontramos a este comportamiento es la diferencia en la distribución de los atributos del contexto. En efecto, si creamos dos contextos con las mismas características, por ejemplo $|\mathcal{M}| = 4$, $|g'| = 2$ y $|\mathcal{G}| = 3$, y en uno de ellos (tabla 16) distribuimos los atributos uniformemente, pero en el otro (tabla 17) no los distribuimos uniformemente, obtenemos dos retículos de conceptos totalmente diferentes con conceptos diferentes en tamaño y en forma.

	a1	a2	a3	a4
o1		X	X	
o2	X		X	
o3		X		X

Tabla 16: Ejemplo de un contexto uniforme.

Retículo de conceptos del contexto de la tabla 16:

- Concepto 1: ({}, {a1, a2, a3, a4})
- Concepto 2: ({o1}, {a2, a3})
- Concepto 3: ({o2}, {a1, a3})
- Concepto 4: ({o3}, {a2, a4})
- Concepto 5: ({o1, o2}, {a3})
- Concepto 6: ({o1, o3}, {a2})
- Concepto 7: ({o1, o2, o3}, {})

	a1	a2	a3	a4
o1	X	X		
o2	X			
o3	X		X	X

Tabla 17: Ejemplo de un contexto no uniforme.

Retículo de conceptos del contexto de la tabla 17:

- Concepto 1: ({}, {a1, a2, a3, a4})
- Concepto 2: ({o1}, {a1, a2})
- Concepto 3: ({o3}, {a1, a3, a4})
- Concepto 4: ({o1, o2, o3}, {a1})

7.4 DISCUSIÓN DE LOS RESULTADOS DE LA EXPERIMENTACIÓN

Una vez concluida la experimentación, a modo de síntesis, en esta sección nos centraremos en los resultados del estudio y la información que hemos extraído de ellos.

Para los conjuntos de datos artificiales, hemos podido apreciar cómo los algoritmos trabajan muy rápido ante conjuntos de datos con poca densidad, obteniendo tiempos de ejecución en general muy bajos con respecto a los conjuntos de datos con densidad elevada. Este fenómeno se debe en gran medida a que al aumentar el número de relaciones existentes en el contexto, el número de conceptos que posee también aumenta. En el mayor de los casos estudiados, para $|M| = 100$, $|\mathcal{G}| = 80$, $|g'| = 40$ se minaron 370683 conceptos por cada algoritmo. Este hecho hace que tengamos que ser muy cuidadosos a la hora de escoger el algoritmo a utilizar en este tipo de conjuntos de datos, teniendo como opciones válidas el NextClosure o el algoritmo de Norris, según si el conjunto de datos va a ser actualizado con frecuencia o no. Por otro lado, si nos encontramos ante un problema con un contexto de poca densidad la elección del algoritmo no penalizará tanto el rendimiento de la aplicación, aunque según nuestro estudio el algoritmo InClose, gracias a su mecanismo de poda, es el más rápido y el que mantiene una mayor estabilidad conforme aumenta el tamaño del contexto.

En resumen, las grandes conclusiones que podemos sacar de nuestra experimentación son:

- Si se trabaja con un conjunto de datos que siga una distribución uniforme en cuanto a los atributos que posee cada objeto, el resultado lo tenemos unívocamente estudiado en todas las casuísticas de la experimentación.
- En las condiciones anteriores, ante conjuntos de datos con densidad baja el algoritmo más rápido es el InClose, mientras que para conjuntos de densidad alta el más rápido con diferencia es el NextClosure.
- Si el conjunto de datos no sigue dicha distribución no podemos predecir qué algoritmo será más rápido para utilizarlo. Pero los algoritmos InClose o NextClosure son los principales candidatos a serlo.

- Los algoritmos incrementales no aportan la ventaja significativa que deberían dar a la hora de añadir conceptos al retículo, sobre todo cuando la densidad es alta, por lo que sería necesario replantearse su funcionamiento.
- Además de todo lo anterior, una de las grandes conclusiones que permanece de este trabajo es el estudio experimental en si mismo. Cualquier persona puede replicar el estudio con el mismo código y los mismos conjuntos de datos descargando el repositorio en su máquina. Esto parece algo poco relevante, pero ninguno de los estudios realizados hasta la fecha ha hecho público el código fuente de su experimentación, y tampoco es popular encontrar estos algoritmos programados en C++.

8

CONCLUSIONES Y TRABAJO FUTURO

A lo largo de este trabajo se ha presentado una implementación de diferentes algoritmos para el cálculo de retículos de conceptos formales dentro de la Teoría del FCA, así como un estudio experimental de su desempeño ante diferentes conjuntos de datos. El FCA ha sido ampliamente estudiado y ha sido objeto de numerosas publicaciones científicas. A su vez, se trata de una herramienta que posee una gran variedad de aplicaciones en el mundo actual.

Por ello nuestro trabajo se ha basado en establecer una conexión entre ambos mundos, y tratar de estudiar la eficiencia de los algoritmos de forma experimental sin perder de vista el marco teórico. Desde los primeros estudios que se realizaron, no se conoce ningún artículo que englobe un estudio experimental de los algoritmos seleccionados para este trabajo. Esto es lo que hace que nuestra aportación sea algo diferente en este ámbito.

Tras presentar el contexto del problema, nos centraremos en los objetivos que se plantearon al principio del trabajo y las conclusiones que hemos obtenido al cumplirlos:

1. *Estudio y comprensión de las nociones del Análisis Formal de Conceptos.*

   Hemos hecho un estudio del marco formal del FCA lo que nos ha llevado a conocer en profundidad una nueva herramienta de representación y extracción de conocimiento que no habíamos estudiado en el programa de estudios de ninguno de los dos grados.

2. *Revisión bibiliográfica sobre el tema y recopilación de los algoritmos más importantes para el estudio.*

   Hemos podido obtener una imagen del estado del arte de los principales algoritmos que se pueden encontrar en la literatura para el minado de los conceptos de un retículo. Consecuencia directa de esta revisión es el aprendizaje a la hora de utilizar buscadores de artículos como Scopus [1] o herramientas de gestión bibliográfica como Zotero [2]. Estas herramientas no las habíamos utilizado pre-

1 https://www.scopus.com/
2 https://www.zotero.org/

viamente y hemos visto el potencial que tienen a la hora de realizar trabajos de investigación.

3. *Implementación de algoritmos y experimentación.*

   El objetivo de la implementación y la experimentación se resumen en el capítulo anterior (capítulo 7. Hemos sido capaces de implementar la mayoría de algoritmos tal y como indicaban los autores en la literatura, salvo el algoritmo *AddIntent*, que aunque calculaba de manera correcta los retículos, su rendimiento era sumamente inferior al resto, lo que nos hace sospechar que alguna estructura no ha sido la óptima para la implementación. Esto nos ha hecho aprender que la selección de las estructuras de datos es muy importante en este tipo de algoritmos. Sin olvidar las conclusiones que obtuvimos de la experimentación: en conjuntos de datos que sigan una distribución uniforme en cuanto al número de relaciones de los atributos con una densidad baja el algoritmo InClose es el que mejor desempeño tiene; mientras que si aumenta la densidad del conjunto de datos el algoritmo más rápido pasa a ser el NextClosure.

4. *Publicación de todo el material utilizado en la experimentación.*

   Todo el trabajo práctico realizado se encuentra disponible en el repositorio [3] de GitHub creado para el TFG, lo que provoca que si cualquier investigador en un futuro tiene interés en replicar la experimentación, los conjuntos de datos, el código fuente de los algoritmos y los resultados para cada uno de ellos se encuentran disponibles de manera pública para su consulta y uso.

5. *Utilizar de manera combinada las competencias sobre Matemáticas y sobre Ingeniería Informática obtenidas durante el transcurso de la carrera para realizar el trabajo.*

   La conclusión de este quinto objetivo es que a lo largo de nuestra etapa estudiantil hemos conseguido formarnos en las dos ramas que hemos elegido como carreras universitarias, consiguiendo unas habilidades que hoy en día nos permiten comprender trabajos que mezclan ambos mundos, como es el que se ha presentado en este Trabajo de Fin de Grado.

Además, durante el desarrollo del trabajo hemos podido observar nuestra capacitación para ciertos ámbitos que pronto serán parte de nuestra vida profesional. Se nos ha requerido comprender una nueva teoría desde cero, y aprender cómo funcionan los algoritmos de minería de datos que hemos seleccionado para el estudio, así como implementarlos y desplegarlos para que funcionen según nuestras necesidades, tal y como puede ocurrir cuando comience nuestra etapa laboral. Por otro lado, a la hora de aprender y profundizar en el funcionamiento de cada algoritmo ha sido necesario hacer una profunda revisión bibliográfica e investigación sobre el tema similar a la

3 https://github.com/miguecl97/TFG-AlgoritmosFCA/tree/master/code

que se podría hacer en un trabajo de posgrado (salvando las distancias en dificultad y tiempo).

Por último, a raíz de los resultados de la experimentación en comparación con algunos trabajos previos, también hemos aprendido a indagar y no asumir toda la información que aparece en un artículo sin previamente reflexionarla, a realizar una experimentación metódica y técnica para obtener unos resultados representativos, y a mejorar nuestras técnicas de programación estructural para obtener un mejor rendimiento en los algoritmos.

Trabajo futuro

Teniendo todo lo anterior en cuenta, el estudio que hemos realizado se podría profundizar mucho más, y mientras realizábamos el trabajo se nos ocurrían ideas interesantes para continuar este proyecto en el tiempo y aportar mayor conocimiento. Las más destacables eran:

- Repetir la experimentación con conjuntos de datos con distintas distribuciones de asignación de atributos a objetos, no necesariamente distribuciones uniformes, y así tener más casuísticas que permitan predecir el comportamiento de los algoritmos ante otros tipos de conjuntos de datos.
- Modificar la implementación del algoritmo AddIntent con algún tipo de estructura auxiliar que permita recuperar los padres de un concepto de una forma más eficiente.
- Paralelizar los algoritmos que mejor rendimiento han tenido, como el InClose o el NextClosure, para obtener una versión aún más rápida de los mismos y comparar sus resultados con los anteriores.
- Realizar el análisis bibliográfico en profundidad de los algoritmos paralelizables que existen para la tarea de la construcción del retículo de conceptos y replicar el estudio realizado.

APÉNDICES

A

PLANIFICACIÓN TEMPORAL Y ESTIMACIÓN DE COSTE

En la tabla 18 se incluye la primera planificación temporal que se realizó al comienzo del proyecto, donde se estimaba su entrega para la convocatoria extraordinaria de septiembre de 2021. Debido a la concurrencia de clases y trabajo de fin de grado durante los primeros meses de 2021 a las primeras tareas se les asignó más tiempo, se esperaba que al finalizar el cuatrimestre el ritmo de trabajo aumentase y el tiempo dedicado a cada tarea fuese menor.

Año	2020	2021							
Tarea/Mes	Diciembre	Enero	Febrero	Marzo	Abril	Mayo	Junio	Julio	Agosto	Septiembre
Concretar el tema										
Recopilación de trabajos previos en la literatura										
Estudio de conceptos fundamentales										
Estudio de conceptos del FCA										
Recopilación y síntesis de algoritmos										
Diseño del software										
Programación de algoritmos por lotes										
Programación de algoritmos incrementales										
Diseño del experimento										
Preparación y ejecución del experimento										
Análisis de resultados										
Desarrollo de documentación										
Correcciones finales										

Tabla 18: Primera planificación temporal realizada para este proyecto.

Sin embargo, con el paso del tiempo tuvimos que ir remodelando dicha planificación conforme surgían nuevos imprevistos. El principal de ellos fue a la hora de programar los algoritmos, la implementación inicial de funciones auxiliares y estructuras nos hizo perder mucho tiempo en los primeros algoritmos, y también obtuvimos muchos errores lógicos que había que trazar minuciosamente y nos hicieron perder bastantes días. A todo esto se le sumó el periodo de exámenes donde el ritmo de trabajo disminuyó considerablemente. A favor le sumamos a la planificación anterior el desarrollo de la memoria mientras se trabajaba en los algoritmos, para explicarlos desde una experiencia más actual. Y reuniones semanales con los tutores y sesiones de tutorización con el mentor del TFG. Dichas reuniones se han ido manteniendo en el tiempo y han sido de gran utilidad a la hora de enfocar y corregir el trabajo realizado.

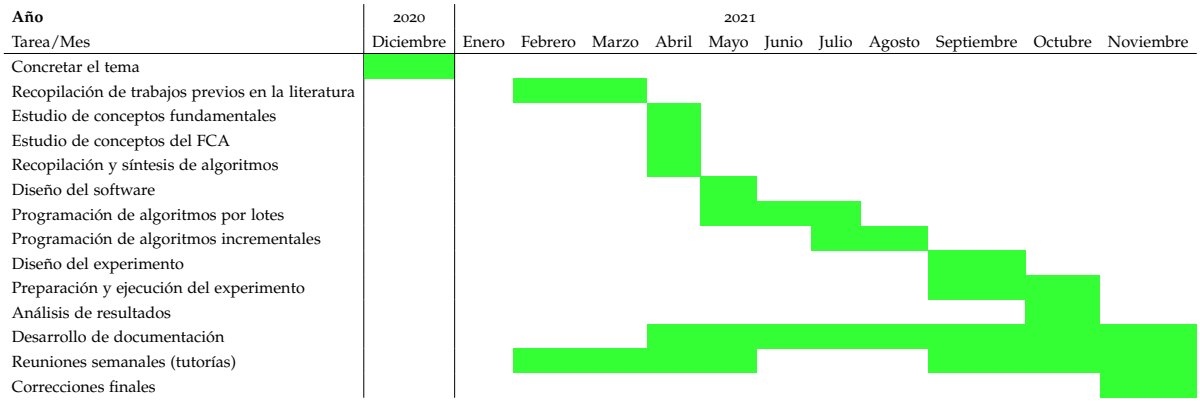


Tabla 19: Planificación temporal final que se ha seguido para este proyecto.

Coste estimado

Como el análisis experimental del trabajo conlleva la ejecución de varios experimentos, junto con un hardware y herramientas necesarias para ello, hemos preparado una estimación del coste monetario que podría incluir el replicar este trabajo por parte de cualquier agente externo teniendo en cuenta que la mano de obra que hemos aportado habrái que contratarla. Como todo el software utilizado para este trabajo es gratuito, tan solo se necesita una máquina para reproducirlo y personal que realice todo el trabajo acordado. Suponemos que un trabajador recibe 25 euros por cada hora de trabajo en el proyecto y esta es la estimación aproximada del coste del mismo:

Concepto	**Horas**	**Coste total en euros**
Entender y planificar el problema	20	500
Estudiar los conceptos básicos	30	750
Estudiar la teoría del FCA	30	750
Estudiar los trabajos anteriores	10	250
Programar los algoritmos	60	1500
Programar los experimentos	20	500
Ejecutar los experimentos	10	250
Analizar los resultados	10	250
Escribir la documentación	40	1000
Máquina para desarrollar el trabajo	-	700
Total	230	5750

Tabla 20: Estimación del coste para el desarrollo del proyecto

B

RESULTADOS DE LOS EXPERIMENTOS

En este apéndice se encuentran los resultados obtenidos en todas las iteraciones de los experimentos realizados. Por cada experimento se han generado 10 conjuntos de datos con los mismos parámetros y se ha medido el tiempo de ejecución de cada algoritmo para cada conjunto de datos en milisegundos. Las tablas recogen el tiempo de ejecución en milisegundos que cada algoritmo ha tardado en ejecutarse para cada conjunto de datos. La última fila es la media de las 10 iteraciones y el resultado que se toma para realizar las gráficas que contiene este documento. Si en alguna casilla no aparece ningún tiempo recogido "−−", esto indica que el algoritmo no se ha ejecutado ya que excedía los límites de tiempo y hacía la experimentación mucho más lenta.

Las ejecuciones están ordenadas del 1 al 10 con respecto al fichero de datos que se ha utilizado como entrada. Por ejemplo, en la tabla 21 la primera entrada se corresponde al fichero *datasets/M100/g'10/G20dataset1.csv*, la fila 3 (segunda entrada) corresponde al fichero *datasets/M100/g'10/G20dataset2.csv*,... así hasta la última entrada que es la del fichero *datasets/M100/g'10/G20dataset10.csv*. **Este es el orden que siguen todas tablas de las ejecuciones del experimento.**

B.1 TABLAS DEL EXPERIMENTO $|\mathcal{M}| = 100$ Y $|g'| = 10$.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	20	1	2	13	0	2	8	2	15
10	20	1	2	12	0	2	8	2	16
10	20	1	2	13	0	2	8	2	17
10	20	1	2	13	0	2	8	2	14
10	20	1	2	13	0	3	8	2	19
10	20	1	2	12	0	2	7	2	15
10	20	1	2	14	0	2	8	2	19
10	20	1	2	13	0	2	7	1	16
10	20	1	2	12	0	2	7	1	14
10	20	1	2	12	0	2	7	2	15
	MEDIA	1	2	12,7	0	2,1	7,6	1,8	16

Tabla 21: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=20.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	40	8	15	49	1	8	59	15	202
10	40	7	15	47	1	8	56	14	184
10	40	8	16	50	1	8	60	14	209
10	40	8	16	51	1	9	58	14	212
10	40	7	16	50	1	8	57	15	193
10	40	7	16	50	1	8	58	15	204
10	40	8	15	50	1	8	59	15	203
10	40	9	15	49	1	7	55	14	177
10	40	7	14	47	1	7	54	14	175
10	40	8	15	48	1	8	55	15	197
	MEDIA	7,66	15,33	49,22	1	7,88	57,33	14,44	195,44

Tabla 22: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=40.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	60	24	47	97	2	18	167	44	841
10	60	25	46	100	2	17	169	43	838
10	60	26	47	98	2	18	176	44	890
10	60	24	46	94	1	17	168	43	813
10	60	41	49	102	2	19	175	44	883
10	60	26	47	98	1	17	176	44	889
10	60	24	49	98	2	17	165	45	852
10	60	25	49	99	2	18	174	44	894
10	60	25	47	96	2	17	174	44	850
10	60	31	49	99	2	18	174	45	912
	MEDIA	26,75	47,5	98	1,88	17,63	171	43,88	857,63

Tabla 23: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=60.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	80	57	113	163	3	33	369	99	2640
10	80	54	108	154	4	30	332	93	2308
10	80	55	109	158	4	31	342	98	2416
10	80	61	112	161	3	31	360	100	2599
10	80	52	105	154	3	30	329	91	2236
10	80	56	110	158	3	31	358	94	2495
10	80	54	110	158	3	30	335	94	2392
10	80	56	112	159	3	31	346	93	2539
10	80	62	129	191	4	33	360	106	2568
10	80	55	113	161	3	32	349	96	2558
	MEDIA	56,2	112,1	161,7	3,3	31,2	348	95	2475,1

Tabla 24: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=80.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	100	138	215	242	5	50	628	185	5713
10	100	102	208	235	5	47	604	180	5372
10	100	106	219	239	5	49	608	185	5523
10	100	101	214	241	5	50	621	181	5478
10	100	104	212	239	5	50	629	189	5604
10	100	102	212	240	5	48	605	186	5462
10	100	106	215	246	5	50	633	191	5792
10	100	109	225	249	6	51	668	192	6096
10	100	98	198	225	5	46	582	177	4942
10	100	105	214	238	5	49	626	185	5709
	MEDIA	107,1	213,2	239,4	5,1	49	620,4	185,1	5569,1

Tabla 25: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=100.

g′	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	150	340	703	525	15	109	1725	542	–
10	150	349	697	517	15	106	1845	572	–
10	150	320	714	535	16	109	1731	552	–
10	150	362	699	531	16	107	1862	601	–
10	150	317	694	515	16	108	1695	544	–
10	150	319	680	515	15	104	1703	524	–
10	150	385	777	632	17	117	2012	619	–
10	150	354	776	576	17	117	2045	603	–
10	150	362	717	531	16	110	1862	570	–
10	150	330	700	518	16	106	1793	570	–
	MEDIA	343,8	715,7	539,5	15,9	109,3	1827,3	569,7	–

Tabla 26: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=150.

g′	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	200	765	1699	942	35	199	3789	1282	–
10	200	751	1663	927	33	194	3707	1272	–
10	200	760	1689	959	33	197	3760	1271	–
10	200	759	1678	935	33	192	3792	1222	–
10	200	746	1719	946	35	195	3644	1223	–
10	200	757	1695	931	33	197	3778	1245	–
10	200	754	1704	937	34	198	3743	1262	–
10	200	766	1709	946	35	197	3781	1291	–
10	200	749	1674	930	32	193	3726	1224	–
10	200	771	1705	931	34	197	3810	1257	–
	MEDIA	757,8	1693,5	938,4	33,7	195,9	3753	1254,9	–

Tabla 27: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=200.

g′	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	250	1464	3286	1449	60	299	6534	2356	–
10	250	1455	3279	1441	56	299	6513	2338	–
10	250	1462	3260	1438	58	298	6506	2378	–
10	250	1448	3265	1430	57	302	6651	2322	–
10	250	1480	3308	1456	58	301	6616	2401	–
10	250	1486	3295	1461	59	302	6680	2342	–
10	250	1489	3317	1454	59	306	6703	2366	–
10	250	1455	3271	1432	57	299	6569	2317	–
10	250	1436	3245	1431	57	297	6468	2282	–
10	250	1467	3326	1452	59	303	6675	2353	–
	MEDIA	1464,2	3285,2	1444,4	58	300,6	6591,5	2345,5	–

Tabla 28: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=250.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	300	2666	6005	2099	93	425	10661	4072	–
10	300	2653	5964	2075	91	429	10463	4170	–
10	300	2648	5875	2026	88	421	10723	4034	–
10	300	2595	5857	2040	86	418	10239	4048	–
10	300	2614	5887	2031	87	422	10488	3966	–
10	300	2641	5878	2042	89	425	10389	3997	–
10	300	2637	5956	2046	90	424	10641	3966	–
10	300	2600	5868	2026	87	419	10279	3974	–
10	300	2620	6013	2158	91	450	11390	4202	–
10	300	2986	6220	2247	100	450	11754	4358	–
	MEDIA	2666	5952,3	2079	90,2	428,3	10702,7	4078,7	–

Tabla 29: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=300.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	350	4212	9488	2790	130	569	15430	6217	–
10	350	4189	9435	2756	126	561	15250	6332	–
10	350	4135	9342	2746	131	554	15129	6442	–
10	350	4108	9302	2739	127	560	15126	6092	–
10	350	4138	9370	2740	126	557	15125	6164	–
10	350	4135	9423	2743	130	565	15277	6103	–
10	355	4135	9423	2743	130	565	15277	6103	–
10	350	4140	9461	2755	130	565	15380	6151	–
10	350	4145	9348	2731	128	560	15124	6070	–
10	350	4136	9327	2746	127	553	14991	6175	–
	MEDIA	4147,3	9391,9	2748,9	128,5	560,9	15210,9	6184,9	–

Tabla 30: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=350.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	400	6156	13654	3480	170	712	20370	9010	–
10	400	6150	13837	3528	173	708	20686	9058	–
10	400	6002	13593	3490	168	707	20144	8731	–
10	400	6109	13814	3520	173	705	20629	9094	–
10	400	6129	13778	3525	178	714	20672	9012	–
10	400	6069	13757	3495	173	700	20350	8879	–
10	400	6090	13654	3490	174	695	20563	8756	–
10	400	6112	13831	3513	174	702	20733	8987	–
10	400	6107	13763	3535	173	715	20176	8814	–
10	400	6057	13654	3480	168	706	20455	9016	–
	MEDIA	6098,1	13733,5	3505,6	172,4	706,4	20477,8	8935,7	–

Tabla 31: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=400.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	450	8604	19319	4359	221	865	27131	12224	–
10	450	8663	19270	4327	220	860	27442	12245	–
10	450	8659	19381	4376	227	869	27166	12516	–
10	450	8699	19425	4348	228	872	27411	12391	–
10	450	8690	19426	4374	230	868	27418	12672	–
10	450	8627	19372	4306	230	873	27314	11955	–
10	450	8641	19198	4347	222	872	27265	12432	–
10	450	8566	19089	4264	218	853	27098	12476	–
10	450	8614	19304	4347	228	868	27013	12266	–
10	450	8574	19243	4330	226	865	27192	12736	–
	MEDIA	8633,7	19302,7	4337,8	225	866,5	27245	12391,3	–

Tabla 32: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=450.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	500	11790	26222	5356	291	1078	34731	16797	–
10	500	11888	26268	5302	292	1049	35214	16727	–
10	500	11792	26325	5306	291	1045	34685	16770	–
10	500	11653	26102	5257	287	1053	34624	16915	–
10	500	11719	26202	5247	291	1044	34644	16360	–
10	500	12362	26335	5338	300	1074	35314	16992	–
10	500	11681	26145	5293	285	1045	34761	16780	–
10	500	11684	25947	5270	290	1031	34582	16978	–
10	500	11877	26438	5292	295	1065	35312	17582	–
10	500	11859	26300	5290	294	1044	35172	17030	–
	MEDIA	11830,5	26228,4	5295,1	291,6	1052,8	34903,9	16893,1	–

Tabla 33: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10, $|\mathcal{G}|$=500.

Tablas finales del experimento $|\mathcal{M}| = 100$, $|g'| = 10$

$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	1	2	12,7	0	2,1	7,6	1,8	16
40	7,67	15,33	49,22	1	7,88	57,33	14,44	195,44
60	26,75	47,5	98	1,88	17,63	171	43,88	857,63
80	56,2	112,1	161,7	3,3	31,2	348	96,4	2475,1
100	107,1	213,2	239,4	5,1	49	620,4	185,1	5569,1

Tabla 34: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10.

$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
100	107,1	213,2	239,4	5,1	49	620,4	185,1	5569,1
150	343,8	715,7	539,5	15,9	109,3	1827,3	569,7	–
200	757,8	1693,5	938,4	33,7	195,9	3753	1254,9	–
250	1464,2	3285,2	1444,4	58	300,6	6591,5	2345,5	–
300	2666	5952,3	2079	90,2	428,3	10702,7	4078,7	–
350	4147,3	9391,9	2748,9	128,5	560,9	15210,9	6184,9	–
400	6098,1	13733,5	3505,6	172,4	706,4	20477,8	8935,7	–
450	8633,7	19302,7	4337,8	225	866,5	27245	12391,3	–
500	11830,5	26228,4	5295,1	291,6	1052,8	34903,9	16893,1	–

Tabla 35: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=10.

B.2 TABLAS DEL EXPERIMENTO $|\mathcal{M}| = 100$ Y $|g'| = 20$.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	20	8	13	50	2	14	30	7	–
20	20	4	8	48	2	16	31	8	–
20	20	4	9	53	3	18	34	10	–
20	20	4	7	45	2	15	32	8	–
20	20	3	8	48	2	16	31	8	–
20	20	3	7	45	2	13	31	8	–
20	20	4	7	44	2	14	30	7	–
20	20	3	9	49	2	16	33	8	–
20	20	4	8	49	2	16	32	9	–
20	20	3	8	46	2	14	30	7	–
	MEDIA	4	8,4	47,7	2,1	15,2	31,4	8	–

Tabla 36: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=20, $|\mathcal{G}|$=20.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	40	37	92	239	13	84	271	90	–
20	40	39	95	257	16	91	285	94	–
20	40	41	100	261	17	92	296	100	–
20	40	42	100	257	16	91	299	97	–
20	40	40	98	253	15	89	277	93	–
20	40	38	94	258	15	86	279	94	–
20	40	39	94	252	15	85	282	94	–
20	40	40	104	277	16	99	290	102	–
20	40	40	96	258	15	88	284	95	–
20	40	38	93	249	14	85	279	92	–
	MEDIA	39,4	96,6	256,1	15,2	89	284,2	95,1	–

Tabla 37: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=20, $|\mathcal{G}|$=40.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	60	149	419	698	54	251	1004	377	–
20	60	148	415	726	55	250	1008	382	–
20	60	151	428	725	57	256	1027	389	–
20	60	150	422	721	55	255	1000	388	–
20	60	146	411	702	53	248	996	393	–
20	60	152	439	724	57	265	1027	397	–
20	60	151	431	715	56	254	1024	388	–
20	60	150	428	707	56	256	1013	386	–
20	60	156	444	741	58	269	1083	410	–
20	60	155	446	781	58	264	1023	380	–
	MEDIA	150,8	428,3	724	55,9	256,8	1020,5	389	–

Tabla 38: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=20, $|\mathcal{G}|$=60.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	80	382	1198	1539	136	531	2962	1105	–
20	80	483	1544	1884	181	680	2668	1342	–
20	80	397	1203	1478	136	535	2446	1016	–
20	80	406	1286	1553	153	569	2528	1084	–
20	80	392	1215	1461	137	535	2447	1006	–
20	80	385	1193	1476	134	522	2395	1021	–
20	80	401	1247	1519	144	551	2480	1060	–
20	80	415	1274	1515	148	567	2584	1076	–
20	80	391	1194	1476	137	532	2453	1013	–
20	80	388	1187	1474	133	524	2403	1032	–
	MEDIA	404	1254,1	1537,5	143,9	554,6	2536,6	1075,5	–

Tabla 39: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=20, $|\mathcal{G}|$=80.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	100	829	2751	2746	308	1009	4923	2267	–
20	100	810	2626	2615	284	977	4775	2191	–
20	100	844	2786	2735	301	1010	4997	2290	–
20	100	840	2742	2711	303	993	4969	2280	–
20	100	829	2692	2679	296	987	4884	2242	–
20	100	828	2692	2721	296	991	4928	2277	–
20	100	795	2617	2609	280	952	4919	2180	–
20	100	825	2725	2740	295	991	4897	2273	–
20	100	843	2720	2657	295	1006	4953	2182	–
20	100	832	2770	2696	304	1008	4920	2259	–
	MEDIA	827,5	2712,1	2690,9	296,2	992,4	4916,5	2244,1	–

Tabla 40: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=20, $|\mathcal{G}|$=100.

Tabla final del experimento $|\mathcal{M}| = 100$, $|g'| = 20$

$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	4	8,4	47,7	2,1	15,2	31,4	8	–
40	39,4	96,6	256,1	15,2	89	284,2	95,1	–
60	150,8	428,3	724	55,9	256,8	1020,5	389	–
80	404	1254,1	1537,5	143,9	554,6	2536,6	1075,5	–
100	827,5	2712,1	2690,9	296,2	992,4	4916,5	2244,1	–

Tabla 41: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=20.

B.3 TABLAS DEL EXPERIMENTO $|\mathcal{M}| = 100$ Y $|g'| = 30$.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
30	20	21	38	162	16	94	83	26	–
30	20	10	30	158	13	81	82	25	–
30	20	10	32	162	14	87	81	25	–
30	20	10	32	173	15	93	79	26	–
30	20	11	31	163	14	89	86	26	–
30	20	10	32	155	15	90	85	26	–
30	20	10	31	156	14	91	85	26	–
30	20	11	33	162	17	97	84	27	–
30	20	16	35	176	15	100	88	29	–
30	20	11	35	177	15	107	90	28	–
	MEDIA	12	32,9	164,4	14,8	92,9	84,3	26,4	–

Tabla 42: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=30, $|\mathcal{G}|$=20.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
30	40	179	983	1656	233	998	1240	546	–
30	40	158	830	1508	201	857	1121	496	–
30	40	178	927	1674	232	980	1229	542	–
30	40	161	806	1546	198	877	1134	501	–
30	40	161	810	1526	201	852	1112	495	–
30	40	182	962	1715	224	1012	1239	556	–
30	40	171	884	1590	219	934	1192	514	–
30	40	166	793	1483	189	840	1158	494	–
30	40	182	877	1644	220	913	1207	539	–
30	40	175	899	1657	224	956	1213	538	–
	MEDIA	171,3	877,1	1599,9	214,1	921,9	1184,5	522,1	–

Tabla 43: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=30, $|\mathcal{G}|$=40.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
30	60	830	6386	7100	1337	3981	5706	2989	–
30	60	776	5789	6672	1202	3677	5339	2726	–
30	60	787	5987	6812	1252	3773	5491	2758	–
30	60	794	6163	7181	1303	3940	5582	2897	–
30	60	800	6112	6603	1243	3816	5589	2820	–
30	60	817	6375	7073	1312	3954	5725	2903	–
30	60	822	6672	7208	1369	4085	5748	2942	–
30	60	819	6336	7133	1308	3958	5684	2906	–
30	60	811	6348	6974	1320	4064	5726	2921	–
30	60	845	6749	7256	1385	4159	5926	2975	–
	MEDIA	810,1	6291,7	7001,2	1303,1	3940,7	5651,6	2883,7	–

Tabla 44: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=30, $|\mathcal{G}|$=60.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
30	80	2487	38823	28832	6848	14589	18314	9548	–
30	80	2510	3995	27738	7045	16743	18149	10054	–
30	80	2535	41635	29765	7491	15492	18786	9766	–
30	80	2529	40440	30686	7532	15245	18847	10056	–
30	80	2676	36485	28261	7474	13803	20438	9548	–
30	80	2523	38631	28911	7551	15652	18666	9585	–
30	80	2449	37413	29364	7270	15133	18238	9718	–
30	80	2501	39580	29891	7723	16097	19011	9858	–
30	80	2486	36423	29397	7212	14586	18264	9626	–
30	80	2533	39946	30153	7317	15933	18783	9932	–
	MEDIA	2522,9	35337,1	29299,8	7346,3	15327,3	18749,6	9769,1	–

Tabla 45: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=30, $|\mathcal{G}|$=80.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
30	100	5970	209621	99975	31842	69105	52502	28861	–
30	100	5962	208100	98037	33465	63480	52129	29321	–
30	100	6130	214956	105935	36495	68855	54842	30223	–
30	100	6033	205390	104463	32429	64981	53583	28678	–
30	100	5897	201750	100228	33925	63325	53176	28333	–
30	100	5674	184309	91061	30089	55783	48705	26158	–
30	100	5952	203474	99057	31740	63356	52099	28488	–
30	100	6126	207541	104757	33397	66638	55295	29562	–
30	100	5992	202759	107681	32311	70480	54435	29758	–
30	100	6112	211849	105831	33844	67415	54811	29485	–
	MEDIA	5984,8	204974,9	101702,5	32953,7	65341,8	53157,7	28886,7	–

Tabla 46: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=30, $|\mathcal{G}|$=100.

Tabla final del experimento $|\mathcal{M}| = 100$, $|g'| = 30$

$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	12	32,9	164,4	14,8	92,9	84,3	26,4	–
40	171,3	877,1	1599,9	214,1	921,9	1184,5	522,1	–
60	810,1	6291,7	7001,2	1303,1	3940,7	5651,6	2883,7	–
80	2522,9	35337,1	29299,8	7346,3	15327,3	18749,6	9769,1	–
100	5984,8	204974,9	101702,5	32953,7	65341,8	53157,7	28886,7	–

Tabla 47: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=30.

B.4 TABLAS DEL EXPERIMENTO $|\mathcal{M}| = 100$ Y $|g'| = 40$.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
40	20	50	178	672	84	618	226	95	–
40	20	31	184	670	84	656	267	106	–
40	20	31	188	701	92	637	232	98	–
40	20	29	180	726	89	618	227	97	–
40	20	32	200	761	94	691	242	99	–
40	20	32	209	789	89	721	245	106	–
40	20	30	178	666	82	604	228	94	–
40	20	26	144	546	67	491	199	82	–
40	20	51	186	668	89	638	244	97	–
40	20	29	178	703	85	621	223	92	–
	MEDIA	34,1	182,5	690,2	85,5	629,5	233,3	96,6	–

Tabla 48: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=40, $|\mathcal{G}|$=20.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
40	40	756	23483	32619	4720	13462	7135	3688	–
40	40	763	20252	24605	4146	13047	6929	3596	–
40	40	783	24030	30059	5282	14987	7328	3828	–
40	40	802	24867	31284	5040	15379	7444	3903	–
40	40	815	27380	33141	5821	16249	7729	3986	–
40	40	815	24180	27345	5189	14564	7464	3780	–
40	40	700	18885	22424	3688	11938	6306	3346	–
40	40	747	22490	26385	4311	13045	6711	3563	–
40	40	793	24601	26188	4640	14244	7090	3577	–
40	40	729	20978	23206	3952	12529	6500	3450	–
	MEDIA	770,3	23114,6	27725,6	4678,9	13944,4	7063,6	3671,7	–

Tabla 49: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=40, $|\mathcal{G}|$=40.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
40	60	5002	795407	473634	130105	276549	87278	49211	–
40	60	5143	863900	545672	133526	289626	93501	54834	–
40	60	5005	795630	440780	124187	260289	88954	49978	–
40	60	4964	833246	438588	122477	272073	86361	49124	–
40	60	4795	748317	437090	123524	246653	81885	48800	–
40	60	4810	813081	451363	125717	252582	89594	48693	–
40	60	5053	806464	446798	124595	264513	90542	50207	–
40	60	4967	799655	455017	125870	261066	93702	52530	–
40	60	5052	843043	450141	123703	276894	87609	52150	–
40	60	5109	812486	502052	125350	280492	92703	52333	–
	MEDIA	4990	811122,9	464113,5	125905,4	268073,7	89212,9	50786	–

Tabla 50: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=40, $|\mathcal{G}|$=60.

g'	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
40	80	18497	–	–	807341	–	515324	308661	–
40	80	17988	–	–	810180	–	535683	324813	–
40	80	18825	–	–	862367	–	561152	337571	–
40	80	18607	–	–	827874	–	524214	316207	–
40	80	17693	–	–	793025	–	536753	325962	–
40	80	18353	–	–	817722	–	539284	328334	–
40	80	19394	–	–	903839	–	600792	358813	–
40	80	18251	–	–	826863	–	528261	315742	–
40	80	18077	–	–	820082	–	530676	318397	–
40	80	17910	–	–	805411	–	550187	335853	–
	MEDIA	18359,5	–	–	827470,4	–	542232,6	327035,3	–

Tabla 51: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=40, $|\mathcal{G}|$=80.

g′	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
40	100	51372	–	–	–	–	–	–	–
40	100	50268	–	–	–	–	–	–	–
40	100	53203	–	–	–	–	–	–	–
40	100	49626	–	–	–	–	–	–	–
40	100	47647	–	–	–	–	–	–	–
40	100	52609	–	–	–	–	–	–	–
40	100	51151	–	–	–	–	–	–	–
40	100	49667	–	–	–	–	–	–	–
40	100	54036	–	–	–	–	–	–	–
40	100	52179	–	–	–	–	–	–	–
	MEDIA	51175,8	–	–	–	–	–	–	–

Tabla 52: Tabla de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=40, $|\mathcal{G}|$=100.

Tabla final del experimento $|\mathcal{M}| = 100$, $|g'| = 40$

$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
20	34,1	182,5	690,2	85,5	629,5	233,3	96,6	–
40	770,3	23114,6	27725,6	4678,9	13944,4	7063,6	3671,7	–
60	4990	811122,9	464113,5	125905,4	268073,7	89212,9	50786	–
80	18359,5	–	–	827470,4	–	542232,6	327035,3	–
100	51175,8	–	–	–	–	–	–	–

Tabla 53: Tabla final de tiempos de ejecución (ms) para $|\mathcal{M}|$=100, $|g'|$=40.

B.5 TABLAS PARA LOS CONJUNTOS DE DATOS REALES

$\|g'\|$	$\|\mathcal{G}\|$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
10	500	8754	14227	3182	42	1399	18596	6848	–
10	500	8728	14211	3189	42	1401	18715	6847	–
10	500	8730	14233	3265	43	1398	18816	6844	–
10	500	8770	14314	3171	42	1397	18847	6842	–
10	500	8711	14202	3176	43	1395	18881	6839	–
10	500	8763	14212	3184	43	1402	18886	6850	–
10	500	8728	14219	3192	43	1404	19278	7346	–
10	500	9162	14334	3191	43	1404	18940	6870	–
10	500	8772	14304	3191	43	1405	18899	6863	–
10	500	8764	14370	3189	43	1403	6860	19384	–
	MEDIA	8788,2	14262,6	3193	42,7	1400,8	17671,8	8153,3	–

Tabla 54: Tabla con los tiempos de ejecución en ms para el conjunto de datos de cáncer de pulmón.

$\lvert g' \rvert$	$\lvert \mathcal{G} \rvert$	NextClosure	Lindig	Inherit-Concepts	InClose	Bordat	Godin	Norris	AddIntent
29	76	1370	3907	9459	318	7951	7124	3071	–
29	76	1344	4264	10902	318	7959	6609	3082	–
29	76	1321	3778	9283	300	7932	6543	3034	–
29	76	1326	3865	9277	317	7949	6647	3073	–
29	76	1323	3777	9275	301	7896	6524	3036	–
29	76	1322	3872	9468	317	7955	6610	3088	–
29	76	1319	3774	9793	301	7918	6660	3035	–
29	76	1318	3871	9506	318	7958	6631	3075	–
29	76	1319	3780	9442	303	7898	6622	3047	–
29	76	1318	3770	9496	301	7923	6544	3043	–
	MEDIA	1328	3865,8	9590,1	309,4	7933,9	6651,4	3058,4	–

Tabla 55: Tabla con los tiempos de ejecución en ms para el conjunto de datos de esponjas.

BIBLIOGRAFÍA

[1] S. Andrews. In-close, a fast algorithm for computing formal concepts. In *International Conference on Conceptual Structures (ICCS)*, January 2009. Final version of paper accepted (via peer review) for the International Conference on Conceptual Structures (ICCS) 2009, Moscow.

[2] Hans-J. Bandelt. Tolerance relations on lattices. *Bulletin of the Australian Mathematical Society*, page 367–381, 1981.

[3] Anne Berry and Alain Sigayret. Representing a concept lattice by a graph. *Discrete Applied Mathematics*, pages 27–42, November 2004.

[4] Jérémy Besson, Celine Robardet, Jean-Francois Boulicaut, and Sophie Rome. Constraint-based concept mining and its application to microarray data analysis. *Intell. Data Anal.*, 9:59–82, 03 2005.

[5] J. P. Bordat. Calcul pratique du treillis de galois d'une correspondance. *Mathématiques et Sciences Humaines*, 96:31–47, 1986.

[6] Claudio Carpineto and Giovanni Romano. A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning*, 24(2):95–122, August 1996.

[7] Michel Chein. Algorithme de recherche des sou-matrices premières d'une matrice. *Bulletin Mathématique de la Société des Sciences Mathématiques de la République Socialiste de Roumanie. Nouvelle Série*, 13, 01 1969.

[8] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, 2002.

[9] Cornelia E. Dowling. On the Irredundant Generation of Knowledge Spaces. *Journal of Mathematical Psychology*, 37(1):49–62, March 1993.

[10] Dheeru Dua and Casey Graff. UCI Machine Learning Repository, 2017.

[11] Vincent Duquenne. Glad (general lattice analysis & design): a fortran program for a glad user. *MSH-Maison Suger, Paris*, 30, 1992.

[12] Abderrahim El Qadi, Aboutajedine Driss, and Ennouary Yassine. Formal concept analysis for information retrieval. *International Journal of Computer Science and Information Security*, 7, 03 2010.

[13] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz— open source graph drawing tools. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing*, pages 483–484, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[14] Petr Gajdo and Vaclav Snasel. A new FCA algorithm enabling analyzing of complex and dynamic data sets. *Soft Computing*, 18, 2014.

[15] Bernhard Ganter. Two basic algorithms in concept analysis. In Léonard Kwuida and Barış Sertkaya, editors, *Formal Concept Analysis*, pages 312–340, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[16] Bernhard Ganter and Sergei Obiedkov. *Conceptual Exploration*. Springer Berlin Heidelberg, 2016.

[17] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[18] Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael W. Mislove, and Dana S. Scott. *A Compendium of Continuous Lattices*. Springer Berlin Heidelberg, 1980.

[19] Robert Godin, Rokia Missaoui, and Hassan Alaoui. Incremental concept formation algorithms based on galois concept lattices. *Computational Intelligence*, 11:246–267, May 1995.

[20] Wolfgang Hesse and Thomas Tilley. *Formal Concept Analysis Used for Software Analysis and Modelling*, pages 288–303. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[21] Katalin Tünde Jánosi-Rancz, Viorica Varga, and Janos Puskas. A software tool for data analysis based on Formal Concept Analysis (University Babes-Bolyai). 2, 2008.

[22] Petr Krajca, Jan Outrata, and Vilem Vychodil. V.: Parallel Recursive Algorithm for FCA. In *Palacky University, Olomouc*, pages 71–82, 2008.

[23] Sergei Kuznetsov. A fast algorithm for computing all intersections of objects in a finite semi-lattice. *Automatic Documentation and Mathematical Linguistics*, 27:11–21, 1993.

[24] Sergei Kuznetsov and Tatiana Makhalova. On interestingness measures of formal concepts. *Information Sciences*, 442, 11 2016.

[25] Sergei Kuznetsov and Sergei Obiedkov. Comparing performance of algorithms for generating concept lattices. *J. Exp. Theor. Artif. Intell.*, 14:189–216, 04 2002.

[26] Sergei O. Kuznetsov. Learning of simple conceptual graphs from positive and negative examples. In Jan M. Żytkow and Jan Rauch, editors, *Principles of Data Mining and Knowledge Discovery*, pages 384–391, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[27] Christian Lindig. Concept-based component retrieval. In *Working notes of the IJCAI-95 workshop: Formal Approaches To The Reuse Of Plans, Proofs, And Programs*, pages 21–25, 1995.

[28] Christian Lindig. *Algorithmen zur Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken*. PhD thesis, Technische Universität Braunschweig, November 1999.

[29] Olvi L. Mangasarian, W. Nick Street, and William H. Wolberg. Breast cancer diagnosis and prognosis via linear programming. *Operations Research*, 43(4):570–577, 1995.

[30] Greg R. Markby, Vicky E. Macrae, Kim M. Summers, and Brendan M. Corcoran. Disease severity-associated gene expression in canine myxomatous mitral valve disease is dominated by tgf signaling. *Frontiers in Genetics*, 11:372, 2020.

[31] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[32] María Burgos Navarro, Jesús García Miranda, Pedro A García Sánchez, and Jose Carlos Rosales. Apuntes de Matemática Discreta y Álgebra lineal (Universidad de Granada).

[33] Eugene Norris. An algorithm for computing the maximal rectangles in a binary relation. *Revue Roumaine de Mathématiques Pures et Appliquées*, 23, January 1978.

[34] Lhouari Nourine and Olivier Raynaud. A fast incremental algorithm for building lattices. *J. Exp. Theor. Artif. Intell.*, 14:217–227, 04 2002.

[35] Jonas Poelmans, Paul Elzinga, Dmitry I. Ignatov, and Sergei O. Kuznetsov. Semi-automated knowledge discovery: identifying and profiling human trafficking. *Int. J. General Systems*, 41(8):774–804, 2012.

[36] Olga Prokasheva, Alina Onishchenko, and Sergey Gurov. Classification Methods Based on Formal Concept Analysis. 2013.

[37] Sarah Roscoe. Formal Concept Analysis Applications in Bioinformatics (University of Nebraska). 2020.

[38] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, 1999.

[39] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *In ACM Trans. Programming Languages and Systems*, pages 99–110, 1998.

[40] Fedor Strok and Alexey Neznanov. Comparing and analyzing the computational complexity of FCA algorithms. In Paula Kotzé, Alta van der Merwe, and Aurona Gerber, editors, *Proceedings of the 2010 Annual Conference of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2010, Bela Bela, South Africa, October 11-13, 2010*, ACM International Conference Proceeding Series, pages 417–420. ACM, 2010.

[41] Gerd Stumme, Rafik Taouil, Yves Bastide, Nicolas Pasquier, and Lotfi Lakhal. Computing iceberg concept lattices with Titanic. *Data & Knowledge Engineering*, 42(2):189–222, August 2002.

[42] Thomas Tilley. Formal Concept Analysis applications to requirements engineering and design. (University of Queensland). 2003.

[43] Alasdair Urquhart. Raymond Balbes and Philip Dwinger. Distributive lattices. university of missouri press, columbia1974, xiii 294 pp. *Journal of Symbolic Logic*, 42(4):587–588, 1977.

[44] P. Valtchev, R. Missaoui, and P. Lebrun. A partition-based approach towards constructing Galois (concept) lattices. *Discrete Mathematics*, 256(3):801–829, October 2002.

[45] Petko Valtchev, David Grosser, Cyril Roume, and Mohamed Rouane Hacene. Galicia: An open platform for lattices. In *In Using Conceptual Structures: Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03*, pages 241–254. Shaker Verlag, 2003.

[46] Dean van der Merwe. *Constructing concept lattices and compressed pseudo-lattices by F.J. (Dean) van der Merwe Submitted in fulfilment of the requirements for the degree M.Sc. Computer Science in the Faculty Engineering, Built Environment and Information Technology University of Pretoria*. PhD thesis, 05 2003.

[47] Dean van der Merwe, Sergei Obiedkov, and Derrick Kourie. Addintent: A new incremental algorithm for constructing concept lattices. In Peter Eklund, editor, *Concept Lattices*, pages 372–385, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[48] Frank Vogt and Rudolf Wille. TOSCANA — A graphical tool for analyzing and exploring data. In Roberto Tamassia and Ioannis G. Tollis, editors, *Graph Drawing*, pages 226–233, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[49] Rudolf Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In Ivan Rival, editor, *Ordered Sets*, pages 445–470, Dordrecht, 1982. Springer Netherlands.

[50] Biao Xu, Ruairí de Fréin, Eric Robson, and Mícheál Ó Foghlú. Distributed Formal Concept Analysis Algorithms based on an Iterative MapReduce Framework. *Lecture Notes in Computer Science*, page 292–308, 2012.