UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

GRADO EN INGENIERÍA INFORMÁTICA

OPTIMIZACIÓN DE ALGORITMOS DE FCA EN R USANDO EL LENGUAJE DE PROGRAMACIÓN C++

OPTIMIZATION OF FCA ALGORITHMS IN R USING C++ PROGRAMMING LANGUAGE

Realizado por
LORENZO REINA DELGADO

Tutorizado por
ÁNGEL MORA BONILLA
DOMINGO LÓPEZ RODRÍGUEZ

Departamento
MATEMÁTICA APLICADA
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre 2022

GRADO EN INGENIERÍA INFORMÁTICA

OPTIMIZACIÓN DE ALGORITMOS DE FCA EN R USANDO EL LENGUAJE DE PROGRAMACIÓN C++

OPTIMIZATION OF FCA ALGORITHMS IN R USING C++ PROGRAMMING LANGUAGE

Realizado por
LORENZO REINA DELGADO

Tutorizado por
ÁNGEL MORA BONILLA
DOMINGO LÓPEZ RODRÍGUEZ

Departamento
MATEMÁTICA APLICADA
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre 2022

Abstract

This final year dissertation focuses on the fcaR package, created for the R programming language, which implements the main methods and functions of Formal Concept Analysis. This is an algebraic-logical paradigm for the treatment of data in tabular form that allows the discovery of knowledge within a data set using the concept lattice and logical implications.

The problem to be solved is that the computation of its algorithms has a high computational complexity, requiring the use of specific libraries for profiling and performance measurement, virtualization and debugging and analysis tools to detect bottlenecks in current implementations.

With all this, there are proposed implementations that improve execution time by applying more efficient code writing techniques, optimized data structures and making use of the C++ programming language, focused on resource efficiency, as well as other libraries for the integration of both programming languages.

Keywords: *Formal Concept Analysis, Algorithm Optimization Techniques, R (Programming Language), C++ (Programming Language), fcaR*

Resumen

Este trabajo de fin de grado se centra en el paquete fcaR, creado para el lenguaje de programación R, el cual implementa los principales métodos y funciones del Análisis de Conceptos Formales. Este es un paradigma algebraico-lógico para el tratamiento de datos en forma de tabla que permite descubrir conocimiento dentro de un conjunto de datos usando el retículo de conceptos y las implicaciones lógicas.

El problema a solucionar reside en que el cáculo de sus algoritmos tiene una alta complejidad computacional, siendo necesario el uso de librerías específicas de perfilado y medición del rendimiento, virtualización y herramientas de depuración y análisis para detectar los cuellos de botella de las implementaciones actuales.

Con todo ello, se proponen implementaciones que mejoran el tiempo de ejecución aplicando técnicas de escritura de código más eficientes, estructuras de datos optimizadas y haciendo uso del lenguaje de programación C++, enfocado a la eficiencia en el uso de recursos, además de otras librerías para la integración de ambos lenguajes de programación.

Palabras clave: *Análisis de Conceptos Formales, Técnicas de Optimización de Algoritmos, R (Lenguaje de Programación), C++ (Lenguaje de Programación), fcaR*

Índice

Índice de figuras

Índice de cuadros

Código fuente

1 Introducción

1.1. Motivación

Inicialmente, el lenguaje de programación R fue creado con el propósito de ser una solución enfocada principalmente a la interactividad del análisis de ingentes cantidades de datos para un mejor y rápido entendimiento por parte de los seres humanos. Esto hizo que derivara desde su comienzo en un lenguaje de programación de ejecución lenta, el cual, no priorizaba la rapidez en la resolución de los algoritmos.

Este trabajo de fin de grado se centra en el paquete fcaR, escrito en el lenguaje de programación antes mencionado, el cual implementa los principales métodos y funciones del Análisis de Conceptos Formales (FCA), que es un paradigma algebraico-lógico usado para el procesamiento y tratamiento de datos en forma de una tabla. Este análisis de conceptos permite a su vez descubrir conocimiento dentro de un conjunto de datos usando dos representaciones, las implicaciones lógicas y el retículo de conceptos.

El inconveniente que presenta dicho paradigma reside en que existe una alta complejidad computacional, siendo exponencial en muchos de los casos, en lo que a sus algoritmos se refiere, por lo que en la mayoría de ocasiones, o la ejecución de alguno de ellos llega a tardar horas o días, dependiendo del conjunto de entrada, o bien su ejecución puede ser inviable en el tiempo con una potencia de procesamiento propia de cualquier ordenador personal actual.

Es debido a esto que surge la necesidad de reducir dicho tiempo de ejecución, y para cubrirla, en este trabajo se investigan herramientas que puedan analizar los cuellos de botella de las implementaciones actuales del paquete, se proponen implementaciones más eficientes

aplicando técnicas de escritura de código más eficiente en R, se utilizan estructuras de datos más optimizadas a los problemas a resolver y se hace uso del lenguaje de programación C++, que está enfocado a la eficiencia en el uso de recursos, integrándolo con el paquete escrito inicialmente en R.

1.2. Objetivos

El objetivo del desarrollo consiste en la aplicación de técnicas de depuración de código en el lenguaje de programación R y C++, la medición del rendimiento de los principales algoritmos del paquete fcaR y la mejora de la eficiencia y el rendimiento de los mismos, o bien la reescritura de código en C++, entre otras, con el fin último de acelerar la lenta ejecución provocada por la alta complejidad computacional intrínseca, minimizando el uso de recursos, tanto espaciales como temporales.

El objetivo principal, por tanto, consiste en:

- Mejorar la eficiencia y el consumo de recursos de los algoritmos de FCA implementados en fcaR.

Con todo ello, el trabajo realizado tiene como objetivos más específicos:

- Investigar técnicas de depuración de código, análisis y mejora de rendimiento de algoritmos en R.
- Depurar y analizar el rendimiento de los principales algoritmos del paquete fcaR.
- Utilizar las técnicas aprendidas para hallar algoritmos más eficientes haciendo uso de la reescritura óptima de código R o usando el lenguaje de programación C++ adaptado al entorno de RStudio.

1.3. Estructura del documento

La estructuración de este documento se define como:

- **Introducción:** en ella se define la motivación que impulsa la realización del trabajo de fin de carrera, así como los objetivos del mismo, tanto el principal como los objetivos más específicos, y finalmente la estructura de la memoria.

- **Estudio del arte y tecnologías a utilizar:** en este capítulo se detalla el entorno teórico sobre el que se basa el paquete fcaR, el Análisis de Conceptos Formales (FCA), además de las tecnologías necesarias para el desempeño de los procesos requeridos para alcanzar los objetivos dictaminados.

- **Presentación del paquete fcaR:** en ella se expone el paquete fcaR, enumerando sus principales métodos, al igual que sus funcionalidades con algunos ejemplos.

- **Metodología:** en ella se explica la metodología de trabajo por la cual se ha regido el desarrollo, y la metodología operativa en cuanto a los procesos que se han proseguido.

- **Hallazgos y propuestas de mejora:** en este apartado se muestra todo el conocimiento adquirido y hallazgos encontrados, así como las propuestas de mejora que hallan surgido tras la aplicación de las técnicas aprendidas.

- **Resultados experimentales:** en este capítulo se realizan las pruebas pertinentes y mediciones de rendimiento tras la aplicación de las propuestas de mejora, comparando el uso de recursos entre las implementaciones actuales y las propuestas.

- **Conclusiones y Líneas futuras:** en ellas se recogen todas las conclusiones que ha desencadenado el desarrollo del proyecto, ofreciendo una valoración general del mismo. Finalmente se expresan algunas ideas sobre la continuación del proyecto y mejorías futuras.

- **Referencias:** consiste en una lista que comprende todas las referencias bibliográficas que se han utilizado durante el desarrollo. Todas ellas están definidas siguiendo la normativa APA.

- **Glosario:** es un catálogo alfabetizado que recoge todas las palabras y acrónimos difíciles de comprender presentes en la memoria, así como su definición.
- **Apéndices:** como apéndice se incluye un manual de instalación en el que se explica al lector cómo preparar el entorno para aplicar las técnicas detalladas en esta memoria.

2

Estudio del arte y tecnologías a utilizar

2.1. Análisis de Conceptos Formales (AFC)

Como se explica en (Škopljanac-Mačina y Blašković, 2014), el Análisis de Conceptos Formales, en inglés, Formal Concept Analysis (FCA), es un método para la representación del conocimiento, gestión de la información y análisis de datos. Este método se encarga de encontrar y visualizar, a partir de un conjunto de datos tabulares de entrada, todos los conceptos junto con sus dependencias.

Este tipo de análisis se ha aplicado hasta día de hoy en numerosos campos de la ciencia, como las matemáticas, biología, medicina, economía, psicología o sociología. Sin embargo, las aplicaciones de mayor interés pueden ser en el campo de las ciencias de la computación; siendo posible su utilización en el análisis de datos, la minería de datos, la corrección de errores en el código fuente, en la recuperación de información, o incluso en el aprendizaje automático.

Como conjuntos de entrada en FCA se utilizan matrices hechas por filas y columnas, de modo que cada fila representa un objeto dentro del dominio de interés, y cada columna representa uno de los varios atributos que pueden estar definidos.

Esta matriz pasada como conjunto de entrada es definida como un contexto formal sobre el que se va a realizar el análisis. Aplicando FCA, se obtienen dos conjuntos de datos de salida; el primero de ellos ofrece una relación jerárquica en forma de diagrama lineal de todos los conceptos establecidos, llamado retículo de conceptos, y el segundo conjunto es un listado de

todas las dependencias internas entre atributos en el contexto formal.

Este trabajo de fin de grado se fundamenta en la teoría de FCA, trabajando con conjuntos difusos en el paquete fcaR. Esto significa que los elementos de la matriz de entrada pueden no tomar un valor booleano, sino tomar uno difuso.

Para conocer más sobre la teoría del Análisis de Conceptos Formales puede consultarse el libro (Ganter y Wille, 1999), donde se detallan los fundamentos matemáticos en los que se basa, definiciones, teoremas, etc.

2.2. Tecnologías a utilizar

Para el desarrollo del proyecto se han instalado y usado una serie de tecnologías que abarcan diferentes entornos de desarrollo, diferentes lenguajes de programación, y el uso de diferentes y diversas librerías que aportan las funcionalidades necesarias para poner en práctica las técnicas que derivan en el alcance de los objetivos finales.

2.2.1. Virtualización con VirtualBox

Dado que uno de los paquetes de R más importantes que se han utilizado como herramienta de perfilado para detectar cuellos de botella en los algoritmos del código está desarrollado para sistemas Linux y MacOS, se ha decidido utilizar la virtualización y establecer el entorno de desarrollo en una máquina virtual con Ubuntu instalado. Esto permite trabajar con todas las herramientas utilizadas en un mismo entorno compatible con ellas.

Para este proyecto se ha decidido utilizar la herramienta VirtualBox creada por Oracle, una solución completa de virtualización de propósito general para hardware de arquitectura x86, orientado al uso en servidores y ordenadores personales. (Oracle, s.f.). Además, se han asignado recursos como 2GB de memoria ram y un espacio en disco de 20 GB.

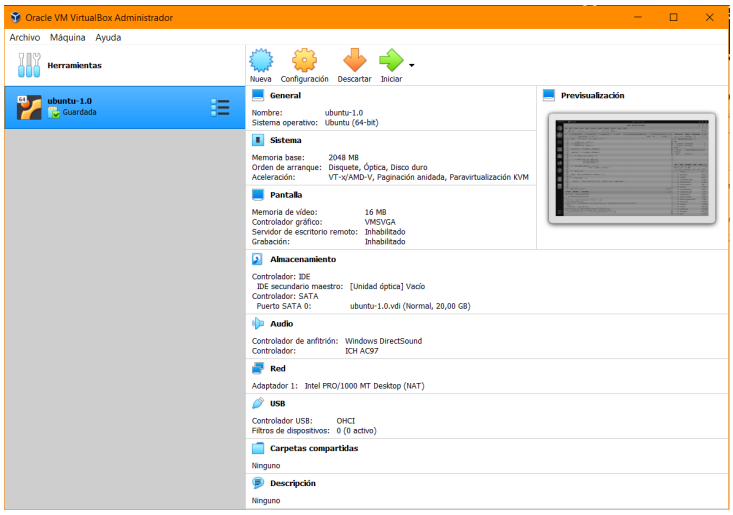

Figura 1: Configuración general de la máquina virtual con Ubuntu

La virtualización como concepto hace alusión a una tecnología capaz de ejecutar un número determinado de máquinas virtuales sobre una máquina física. Esto se hace principalmente para aprovechar los recursos de un sistema al máximo. A cada una de estas máquinas virtuales se les puede asignar unos recuros determinados y pueden trabajar sobre distintos sistemas operativos en un mismo servidor.

Para explicar el funcionamiento de esta tecnología, hay 2 conceptos fundamentales que se deben entender; la máquina virtual, la cual se crea a través de software y ejecuta las aplicaciones y el sistema operativo, quien hace uso de los recursos que le proporciona la máquina física, y el hipervisor, que se encarga de crear la capa de virtualización. Además, este asigna a cada máquina los recursos necesarios de forma dinámica, creyéndose así que dispone del hardware (Limones, 2021).

2.2.2. RStudio

RStudio es un IDE moderno (Entorno de Desarrollo Integrado) y completo de desarrollo, el cual tiene capacidad para integrarse con numerosas herramientas de gestión de proyectos. Entre las características que nos ofrece se encuentran el formateado de código, código autocompletado, coloreado de sintaxis, etc.

Fue creada para el lenguaje de programación R, y a parte, tiene integrado un sistema de control de versiones como Git, y otras herramientas como documentación del lenguaje que se está utilizando, visualización de datos y gestión de proyectos, depuradores con detección de errores, y un sistema de paquetes con los que se pueden ampliar las funcionalidades según son requeridas. Como se puede observar en la documentación de RStudio (RStudio, s.f.-a), y en la página oficial (RStudio, s.f.-b), existe una gran comunidad que respalda esta solución, con infinidad de guías y tutoriales en internet.

2.2.3. git SCM (Source Code Management)

Como se indica en su página oficial («Documentación oficial de git», s.f.), Git es un sistema distribuido y gratuito de control de versiones de código abierto enfocado y creado para administrar varios proyectos de manera rápida y eficiente. Git no degrada el rendimiento de las herramientas con las que se integra porque utiliza muy pocos recursos. Sus características principales incluyen ramificaciones locales de bajo coste y múltiples flujos de trabajo.

Durante el desarrollo se ha decidido utilizar Git integrado con el IDE RStudio, lo cual permite disponer de su funcionalidad a través de una interfaz gráfica sencilla e intuitiva. Además de la decisión tomada de usar esta herramienta, también se ha hecho uso de la aplicación web GitHub basada en Git, ya que el código fuente del paquete fcaR se encuentra en un repositorio público dentro de esta plataforma.

La metodología de trabajo, que posteriormente se explicará en su propio capítulo, con lo que a Git se refiere, ha consistido en la creación de una nueva rama sobre la rama master del repositorio original, en la cual se ha ido trabajando, añadiendo y modificando código fuente para acelerar los algoritmos. Este proceso ha sido escalonado debido a que se han realizado actualizaciones del repositorio a medida que se aplicaban cambios, y una vez finalizado el proyecto, el porpósito es mezclar la rama del repositorio ya actualizado con la original.

2.2.4. Overleaf y LaTeX

Overleaf consiste en una herramienta colaborativa en línea de escritura y publicación en LaTeX y texto enriquecido (Overleaf, s.f.). Overleaf consigue que el proceso de edición, escritura y publicación de documentos académicos o científicos sea más rápido e intuitivo. También incluye un entorno LaTex completo que se ejecuta en sus servidores y está listo para usar, el cual, engloba todo el proceso de documentación científica en una única plataforma, permitiendo la revisión y la publicación.

LaTeX ha sido la solución para la redacción de esta memoria, y para ello, se ha hecho uso de una plantilla proporcionada por la Universidad de Málaga, que incluye numerosos paquetes que aportan muchas funcionalidades a LaTeX. Además de esto, se ha decidido usar el paquete biblatex para las citas en formato APA, el paquete glossaries para la creación de un glosario de palabras díficiles de entender o muy específicas del campo de estudio, entre otros.

2.2.5. Lenguaje de programación R

R es un lenguaje de programación de código abierto ampliamente utilizado como herramienta de análisis de datos y software estadístico (Foundation, s.f.). R normalmente viene con una interfaz de línea de comandos, y se utiliza como una herramienta líder para la programación, el aprendizaje automático, las estadísticas y el análisis de datos.

R facilita también la creación de funciones, objetos y paquetes, ya que es un lenguaje libre y de código abierto, no considerándose solo un paquete estadístico, sino que también permite la integración con otros lenguajes (C, C++). De modo que puede interactuar fácilmente con muchas fuentes de datos y paquetes estadísticos.

El lenguaje de programación R tiene una gran base de usuarios y crece cada día. Además, incluye CRAN (Comprehensive R Archive Network) («Documentación del lenguaje de programación R», s.f.), un reporsitorio de más de 10.000 paquetes.

Sin embargo, el lenguaje de programación R es mucho más lento que otros lenguajes de programación debido a su enfoque interactivo de visualización de datos, y es ahí donde surge la necesidad de desarrollar este trabajo de fin de grado, con el fin de acelerar la ejecución de ciertas funciones y algoritmos pertenecientes al paquete fcaR del lenguaje.

2.2.6. Lenguaje de programación C++

El lenguaje de programación C++ está basado y hereda la sintaxis del lenguaje de programación C (CppReference, 2017). Este sigue principalmente el paradigma POO o programación orientada a objetos, usando en su caso estructuras de datos y tipos básicos como su predecesor. Entre sus principales características, que vienen detalladas en su documentación (cppreference.com, s.f.), podemos encontrarnos con:

- El uso de la herencia, la abstracción, el encapsulado, el polimorfismo.
- Es un lenguaje que acepta múltiples paradigmas.
- Es capaz de admitir funciones anónimas o Lambda funciones, el uso de punteros, sobrecarga del operador y puede manejar los errores y excepciones.

Las ventajas del uso de este lenguaje de programación frente a otros es que es un lenguaje compilado que se acerca mucho a los lenguajes de bajo nivel, y dispone de una enorme eficiencia con el hardware. Es por ello que en este proyecto se ha utilizado para optimizar ciertos algoritmos y reducir considerablemente su consumo de recursos. También se puede integrar muy fácilmente con su predecesor, el lenguaje C, aun siendo ambos lenguajes de alto nivel, y cualquier compilador de C++ es capaz de ejecutar y compilar código C, además de poder hacer uso de librerías en C sin modificarlas en demasía.

Sin embargo, es un lenguaje fuertemente tipado por lo que el programador debe conocer casi en su totalidad el lenguaje para conseguir hacer funcionar un algoritmo o función. Esto hace que sea un lenguaje muy complejo, que está pensado para programadores experimentados.

2.2.7. profvis

Profvis es un paquete o librería de perfilado de código que funciona únicamente con el lenguaje de programación R. Este tipo de herramientas de perfilado ayudan a identificar en qué parte del código se están gastando más recursos, tanto de tiempo como de memoria, por lo que gracias a ella se pueden identificar rápidamente cuellos de botella que ralentizan la ejecución.

Proporciona también una interfaz gráfica interactiva que permite visualizar los datos recogidos por Rprof, que forma parte de la distribución básica de R y es la herramienta de R por excelencia para la recopilación de perfiles («Documentación extra del paquete profvis», s.f.).

Profvis usa un intervalo de 10ms por cada detección al intérprete de R que hace el perfilador, mirando la pila de llamadas de la función que se ejecuta en el momento, y registrándola en un archivo. Cada vez que se hace una llamada a profvis, el resultado varía con ligereza, ya que la solución funciona por muestreo y el resultado, por lo tanto, no es determinista. Además, profvis ofrece una vista de datos y una vista del perfilado en una tabla.

El paquete pertenece a CRAN (Chang, 2020), y se ha recurrido a su documentación oficial para encontrar información de las diferentes funcionalidades que pueden ser finalmente aplicadas a este trabajo de fin de grado.

2.2.8. bench

Bench es un paquete para el cronometraje de alta precisión de expresiones de R, perteneciente también a CRAN (Hester, 2021). El objetivo del paquete es hacer un benchmark del código, siguiendo el tiempo de ejecución, las asignaciones de memoria y las recolecciones de basura (Hester, s.f.), es decir, medir el tiempo que tarda en ejecutarse una expresión o sección de código.

Algunas de sus más remarcables características que lo diferencian de sus alternativas son:

- Monitoriza todas las asignaciones de memoria correspondientes a cada expresión.
- Utiliza las APIs disponibles más precisas de cada sistema operativo.
- Hace un seguimiento de las recolecciones de basura de R en todas las expresiones.
- El resumen de estadísticas se computa tras la filtración de las iteraciones con recolección de basura, lo que aísla el rendimiento real de los efectos de esta recolección de basura durante la ejecución.

2.2.9. jointprof

Jointprof es un paquete de R cuyo propósito es muy similar al del paquete profvis mencionado anteriormente. Lo que le diferencia es la capacidad de poder perfilar código procedente de varios lenguajes de programación a la vez (Rconsortium, s.f.). Es por ello que se ha decidido usar para perfilar código en R integrado con código C++ en este proyecto.

En el caso de jointprof, se ejecuta el código y se detiene la ejecución cada cierto tiempo (por defecto 50 veces por segundo), y registra la pila de llamadas en cada ocurrencia (Muller, 2021).

Otra de las ventajas que tiene es la capacidad de mostrar los datos recogidos a través de diferentes interfaces gráficas, como por ejemplo un grafo o una gráfica de barras, lo que lo hace muy intuitivo.

El paquete solo se encuentra disponible para distribuciones Linux y MacOS, pero este paquete ha sido muy decisivo para el desarrollo de este trabajo de fin de estudios. Tanto es así que se ha decidido hacer uso de la virtualización para que el entorno de desarrollo del proyecto fuese compatible con jointprof.

2.2.10. Rcpp

Rcpp ha sido otro de los paquetes imprescindibles. Perteneciente a CRAN, Rcpp proporciona funciones de R y clases de C++ que brindan una integración perfecta entre R y C++ (Eddelbuettel, 2022). Muchos tipos y objetos de datos de R se pueden convertir a sus equivalentes de C++, lo que facilita la escritura de código nuevo y la integración de bibliotecas de terceros (Project, s.f.).

En el libro de Hadley Wickham (Wickham, 2019), en el capítulo ”High performance functions with Rcpp”, se detallan con precisión técnicas para la aceleración de la ejecución de código a través de la integración del paquete Rcpp y el uso del lenguaje C++ para ello, donde se describen los principales cuellos de botella que se pueden llegar a solucionar con esta técnica, como los bucles que no se pueden vectorizar fácilmente debido a la dependencia entre iteraciones, problemas que implican la llamada de millones de funciones, o problemas que requieran de estructuras de datos y algoritmos avanzados que proporciona C++ en su librería estándar debido a las características propias del lenguaje.

3

Presentación del paquete fcaR

fcaR es un paquete de R que se basa en el Análisis de Conceptos Formales Difusos y está disponible en CRAN. Este puede guardar y cargar Contextos Formales a través de funciones, además de extraer su red de implicaciones y conceptos. Estas implicaciones, a su vez, pueden usarse para el cálculo de cierres semánticos de los conjuntos difusos, construyendo así sistemas de recomendación (López y Mora, 2021).

3.1. Principales métodos

Para trabajar con contextos formales y conjuntos de implicaciones, el paquete tiene implementadas tres clases principales usando la programación orientada a objetos en R, de las que derivan numerosos algoritmos internamente. Estas clases son, como podemos ver en (López y Ángel, 2018):

- La clase FormalContext constituye el contexto formal (G, M, I), siendo G el conjunto de objetos, M el conjunto de atributos e I la matriz de relaciones (difusas), y dispone también de métodos para hacer operaciones usando las herramientas de FCA. En la figura 2 se pueden observar los métodos que implementa esta clase.

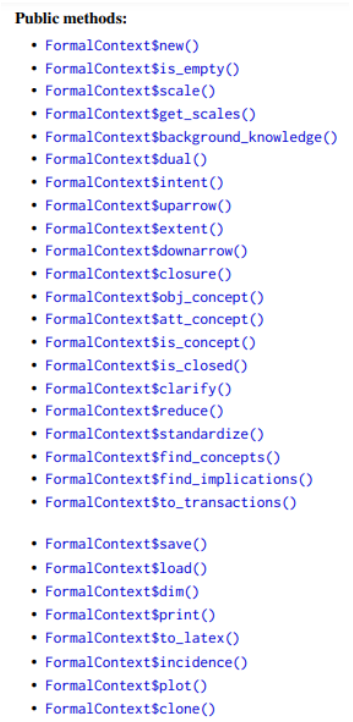

Figura 2: Métodos de la clase FormalContext (López y Mora, 2022)

- ImplicationSet representa un conjunto de implicaciones sobre un contexto formal específico. En la figura 3 se pueden observar los métodos que implementa esta clase.

Figura 3: Métodos de la clase ImplicationSet (López y Mora, 2022)

- ConceptLattice representa el conjunto de conceptos y sus relaciones, incluyendo métodos para operar sobre el retículo. En la figura 4 se pueden observar los métodos que implementa esta clase.

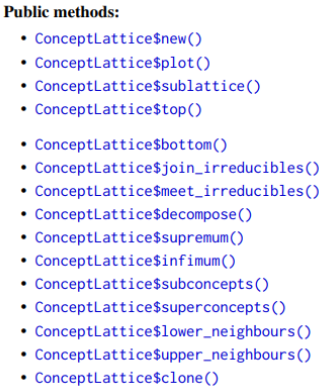

Figura 4: Métodos de la clase ConceptLattice (López y Mora, 2022)

Además, también se aportan las clases Set, que representa a los conjuntos matemáticos y cuyo fin es permitir la visualización de los mismos, y Concept, que encapsula la extensión y la intensión de un concepto formal como un Set.

fcaR es una extensión del modelo de datos del paquete arules, por lo que la implementación interopera con las principales clases S4 de arules.

3.2. Ejemplos

Aquí se muestra un ejemplo siguiendo las viñetas del paquete fcaR (López y Ángel, 2018).

```
library(fcaR)
###### Trabajando con contextos formales ######
fc_planets <- FormalContext$new(planets)
fc_I <- FormalContext$new(I)
###### Visualizando contextos formales ######
print(fc_planets)
```

```
print(fc_I)
fc_planets$plot()
fc_I$plot()
fc_planets$to_latex()
###### Importando contextos formales desde ficheros ######
filename <- system.file("contexts", "airlines.csv",
                        package = "fcaR")
fc1 <- FormalContext$new(filename)
filename <- system.file("contexts", "lives_in_water.cxt",
                        package = "fcaR")
fc2 <- FormalContext$new(filename)
###### Calculando la traspuesta de la matriz del contexto
   formal ######
fc_dual <- fc_planets$dual()
fc_dual
###### Calculando el cierre ######
S <- Set$new(attributes = fc_planets$objects)
S$assign(Earth = 1, Mars = 1)
fc_planets$intent(S)
S <- Set$new(attributes = fc_planets$attributes)
S$assign(moon = 1, large = 1)
fc_planets$extent(S)
Sc <- fc_planets$closure(S)
fc_planets$is_closed(S)
fc_planets$is_closed(Sc)
###### Clarificación y reducción ######
fc_planets$reduce(TRUE)
fc_I$clarify(TRUE)
###### Extrayendo implicaciones y conceptos ######
fc_planets$find_implications()
fc_I$find_implications()
```

```
fc_planets$concepts
fc_planets$implications
###### Estandarizando contextos ######
fc_planets$standardize()
fc_I$standardize()
###### Cargando y guardando ######
fc$save(filename = "./fc.rds")
fc2 <- FormalContext$new("./fc.rds")
###### Visualizando conceptos ######
fc_planets$concepts$plot()
fc_I$concepts$plot()
fc_planets$concepts
fc_planets$concepts$to_latex()
###### Obteniendo la extensión e intensión ######
fc_planets$concepts[2:3]
fc_planets$concepts$extents()
fc_planets$concepts$intents()
###### Soporte de los conceptos ######
fc_planets$concepts$support()
###### Subretículos ######
idx <- which(fc_I$concepts$support() > 0.2)
sublattice <- fc_I$concepts$sublattice(idx)
sublattice
sublattice$plot()
###### Subconceptos, superconceptos, ínfimo y supremo ######
C <- fc_planets$concepts$sub(5)
fc_planets$concepts$subconcepts(C)
fc_planets$concepts$superconcepts(C)
C <- fc_planets$concepts[5:7]
fc_planets$concepts$supremum(C)
fc_planets$concepts$infimum(C)
```

```
68 ###### Join-irreducibles y meet-irreducibles ######
69 fc_planets$concepts$join_irreducibles()
70 fc_planets$concepts$meet_irreducibles()
```

Código.- 1: Código de ejemplo usando el paquete fcaR (López y Ángel, 2018)

4 Metodología

4.1. Metodología de trabajo

Se ha utilizado una metodología Agile para el desarrollo del proyecto, siendo esta una de las metodologías más populares y usadas actualmente en el sector de la informática y el desarrollo software, la cual se basa principalmente en la consecución de hitos para llegar finalmente al objetivo marcado. Durante este proceso pueden aparecer cambios, por lo que se considera una metodología flexible que se adapta a la situación actual del proyecto y a lo que el mismo desarrollo demanda.

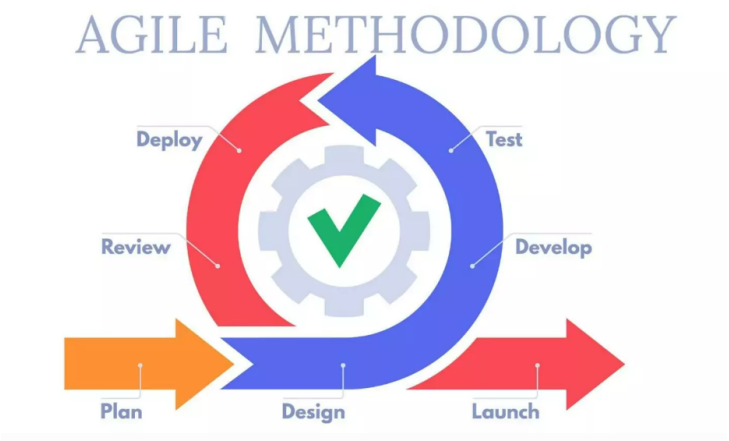

Figura 5: Pasos de la metodología Agile, imágen tomada de (Davies, 2018)

Gracias al seguimiento de Agile, se pueden resolver problemas más complejos de una forma escalonada que se vuelve más eficiente y sencilla. Esto es, un desarrollo a base de iteraciones que persiguen alcanzar unos hitos para progresar en el proyecto.

Se ha dividido el proyecto en varias etapas dentro de cada fase de trabajo, asignando un tiempo de varias semanas a cada una de ellas con el fin de cumplir los plazos y avanzar iterativamente, comprobando el funcionamiento y la corrección del trabajo realizado en cada etapa y acoplándolo a lo desarrollado en etapas anteriores.

4.2. Metodología operativa

La metodología operativa por la que se ha optado finalmente para el trabajo de fin de estudios ha seguido sprints de dos semanas de duración, en los cuales se marcaban hitos asequibles.

Inicialmente, los dos primeros sprints han tenido como finalidad la ampliación de conocimiento sobre algunas de las tecnologías y lenguajes que, a priori, ya se sabía que iban a ser utilizadas. Entre estas tecnologías y lenguajes se encontraban R y C++, y las herramientas RStudio, Git y algunos paquetes o librerías de integración de ambos lenguajes como Rcpp.

Los sprints siguientes han consistido en procesos de investigación y recolección de información sobre técnincas de análisis y mejora de rendimiento en algoritmos del lenguaje de programación R. Estos hitos marcados han tenido como objetivo hacer un estudio sobre las técnicas mencionadas para poder aplicarlas al paquete fcaR en posteriores sprints. Durante el proceso, se han tomado apuntes, se ha ampliado el conocimiento del campo, y se ha comenzado con la instalación de las herramientas para la creación del entorno de desarrollo sobre el que trabajar.

Ya en el ecuador del proyecto, se han dedicado cuatro sprints cuyos objetivos pasaban por el análisis de rendimiento de los algoritmos del paquete en cuestión, aplicándose las técnicas ya aprendidas, haciéndose uso de la virtualización para el entorno de desarrollo, y el uso de paquetes de depuración y perfilado de código en R para identificar aquellos algoritmos que necesitaban optimizarse para mejorar su consumo de recursos. Esto se ha realizado conjuntamente con el proceso de mejora de rendimiento de los mismos, en el que se han hecho propuestas de mejora que en el próximo capítulo se detallan, haciéndose uso del lenguaje C++ adaptado al entorno de RStudio con el paquete Rcpp.

En el último mes, equivalente a los dos últimos sprints, se ha procedido a la redacción de los entregables del proyecto, consistentes en la memoria y un manual de instalación. También se han ultimado y optimizado varios de los algoritmos esenciales del paquete fcaR. Con todo ello, se ha procedido a la correción y validación por los tutores de los entregables que aquí se describen.

5

Hallazgos y propuestas de mejora

5.1. Investigación sobre técnicas de análisis y mejora de rendimiento de algoritmos en R

Para conseguir mejorar el rendimiento de una sección de código fuente es necesario seguir una ruta que pasa primero por identificar errores, si los hubiera. Para ello existen técnicas y herramientas de depuración de errores que pueden aplicarse en el entorno de RStudio y el lenguaje R.

Según redacta Wickham en el capítulo 22 de su libro Advanced R(Wickham, 2019), existen varios métodos para abordar los problemas causados por errores de código.

Una aproximación general para resolver esta situación comienza con la búsqueda en Google del mensaje de error. Para estas búsquedas es recomendable eliminar cualquier nombre de variable o valor específico del problema. Si esto no ha tenido éxito, lo próximo para encontrar la causa es hacer un ejemplo del error que sea reproducible simplificando el problema y los datos que se tratan en él. Si esto no diese resultado, también es una opción recomendable seguir el método científico, e intentar generar una hipótesis, diseñar experimentos para probarlos y recoger los resultados. Una vez encontrado el error, ya solo queda intentar solventarlo, siendo muy útil disponer de sistemas de pruebas automatizadas.

Por otro lado, existen herramientas muy útiles que, una vez encontrado el error y se haya hecho repetible, nos pueden servir para solventarlo. Un ejemplo de ellas es traceback(), la cual muestra la secuencia de llamadas o la pila de llamadas que condujo al error.

Además de todo ello, existe la posibilidad de hacer uso de herramientas de depuración interactiva, como la herramienta de RStudio, Rerun with Debug, que pausa la ejecución justo en la parte del código donde se encuentra el error.

Otra opción es el uso de breakpoints o puntos de parada. Se pueden añadir manualmente en RStudio a las líneas de código dónde se quiera forzar la detención. Finalmente, si ninguna de las opciones anteriores es viable, siempre se puede recurrir a la depuración insertando líneas de código con funciones de escritura para evaluar hasta dónde llega la ejecución y en qué estado se encuentra.

El siguiente paso que marca la hoja de ruta para alcanzar una mejoría de rendimiento en el código pasa por la detección de partes del mismo que provocan cuellos de botella. Para conseguirlo, se suelen utilizar librerías que disponen de soluciones para medir el rendimiento en la ejecución de los algoritmos. Entre las que se han usado en este proyecto se encuentran profvis, la cual perfila código únicamente en R, y jointprof, que es una solución muy completa y elegante, ya que perfila código en R y además también analiza partes de código escritas en el lenguaje C++, y con ello medir el rendimiento total.

5.1.1. Herramientas de perfilado de código; profvis y jointprof

Existen numerosas variedades de generadores de perfiles, pero R utiliza uno muy simple denominado perfilador estadístico o de muestreo. Este tipo de perfiladores detienen la ejecución del código cada cierto tiempo, normalmente unos pocos milisegundos, y hace un registro de la pila de llamadas. Estos perfiladores tiene un pequeño inconveniente, el cual reside en que cada vez que se haga un perfil, se obtendrá una respuesta ligeramente distinta debido a que existe cierta variabilidad en la precisión del temporizador como en el tiempo que toma cada operación. La variabilidad, por otro lado, afecta sobre todo a las funciones más rápidas, que

son las que menos interesan. (Wickham, 2019)

Para usar profvis en el código, se puede hacer tanto desde el menú Perfil en RStudio, como con la función profvis::profvis(). Esta ejecución creará un panel en el que se podrán visualizar tanto la fuente de donde viene el código en la parte superior, como el gráfico de la pila de llamadas en la parte inferior. Este panel es interactivo, y arrastrando el ratón por las secciones del mismo aparece más información del resultado del perfilado. La información que nos brinda este perfilador corresponde al consumo de memoria de cada llamada de la pila de llamadas, además del tiempo de ejecución de esa llamada en particular. Un indicio de que el código necesita optimización puede ser la aparición de muchas llamadas al recolector de basura, el cual se ve reflejado en el panel como <GC>significando ”garbage collector”.

Como se ha comentado en varias ocasiones, profvis sirve exclusivamente para el perfilado de código en R, y aunque internamente este código llame a secciones escritas en otros lenguajes, la herramienta ignorará esas llamadas y, aunque se vean reflejadas en el perfilado, no se aporta información de la ejecución de las mismas.

Para solventar el problema que se acaba de mencionar se ha utilizado el paquete jointprof, disponible únicamente para los sistemas operativos de Linux y MacOS. Debido a esta incompatibilidad con Windows, y al ser este paquete tan importante para la detección de cuellos de botella en el código, se ha decidido montar el entorno de desarrollo en una máquina virtual con la solución de Linux Ubuntu, donde se han instalado todos los paquetes y herramientas necesarias para hacer el desarrollo del proyecto.

Jointprof, como ya se ha mencionado repetidamente, es un generador de perfiles muy similar a profvis, pero que en su caso tiene en cuenta las llamadas a código escrito en el lenguaje C++. Esta funcionalidad permite tanto la detección de secciones de código a optimizar, como la comprobación tras el proceso de optimización con el uso del lenguaje C++ de que los cuellos de botella detectados han desaparecido.

El uso de este paquete viene muy bien detallado en las viñetas creadas por el autor del paquete (Muller, 2021), y se puede acceder a ellas como el aprendizaje previo a utilizarlo.

Además de lo mencionado, este paquete exporta el resultado de la generación del perfil a una Web UI en http://localhost:8080, en el que aparece una interfaz interactiva donde se muestran diferentes modos de visualizar los datos recogidos, tanto en grafos como tablas o diagramas.

5.1.2. Técnicas de mejora de rendimiento del código; Rcpp

Se proponen una serie de técnicas o pautas que se deben contemplar para alcanzar la optimización deseada de una sección de código. Aunque son técnicas diferentes, es sabido que todas se complementan, y si se aplican todas al problema en cuestión, habrá más posibilidades de que el intento de aceleración en la ejecución del código sea fructífero.

La primera estrategia y una de las más conocidas, y que mejor se adaptan a la optimización de algoritmos, consiste en hacer que éste haga lo menos posible, es decir, la manera en que una función sea más rápida, reside en que esa función efectúe menos trabajo (Wickham, 2019). Por ejemplo en R:

- colMeans(), rowMeans(), rowSums() y colSums() son funciones vectorizadas y por ello son más rápidas que las invocaciones equivalentes que usan apply().
- vapply() especifica el tipo de dato del resultado y es por ello que es más rápido que sapply().
- Probar la igualdad entre un valor y los valores de un vector es siempre más rápido que comprobar la inclusión de conjuntos.

- Al hacer una llamada a un método, se debe revisar si acepta parámetros que puedan ser conocidos por el problema, que consigan acelerar mucho la ejecución de dicho método, sobre todo si se llama iterativamente. Por ejemplo, mean.default() será más rápido que mean() para vectores de poco tamaño (el cálculo de la media); sin embargo, hay que ser consciente de que la entrada debe ser un vector numérico, si no se mostrará un error. .Internal(mean()) es aún más rápido, ya que no hace verificaciones de los valores de entrada ni trabaja con NA.

- Reconocer qué estructura de datos se debe utilizar en cada momento puede ser un factor crucial para la optimización de un método. Existen ciertas estructuras que son mucho más eficientes con determinados tipos de datos de entrada que otras, pero esto sólo se aprende a través de la experiencia programando.

Otra de las estrategias más conocidas en R es la de priorizar el uso de vectores para los cálculos y dejar atrás los bucles que sean innecesarios. Esto es así porque, internamente, los bucles de los vectores en R están programados en lenguaje C, que es mucho más eficiente y tiene menos sobrecarga.

Hay que considerar también que hacer muchas copias de estructuras u objetos supone un coste enorme para la ejecución. Cuando se utilizan funciones como c(), paste(), cbind(), rbind() o append() hay que prestar especial atención, intentando no involucrarlas en bucles.

Hay veces que aún aplicando todas las técnicas descritas arriba, el código en el lenguaje de programación R no es capaz de optimizarse hasta el punto en el que el programador desea, y una posible solución para hacerlo puede pasar por hacer uso de la librería Rcpp. Esta hace que sea sencillo conectar C++ y R y ecribir el código en este lenguaje, el cual, está mucho más optimizado, ya que está fuertemente tipado, y aunque sea un lenguaje de alto nivel, se puede llegar a considerar de bajo nivel. Rcpp proporciona una API que permite a R escribir código de alto rendimiento.

Los problemas más comunes que se pueden abordar con este método son la optimización de cuellos de botella en ciertos bucles que no son fáciles de vectorizar y que las iteraciones posteriores dependen de las anteriores, las funciones recursivas o llamadas a millones de funciones, o problemas avanzados que requieren de estructuras de datos complejas que R no proporciona pero que C++ si lo hace a través de la biblioteca estándar.

El paquete Rcpp dispone de una función cppFunction() que permite escribir código C++ y pasárselo ahí mismo, pero esta no es una solución muy elegante y no se va a utilizar para este proyecto. En cambio, para el caso en cuestión, se usarán archivos C++ independientes para luego convertirlos a R usando sourceCpp(). Para ello, es necesario que este archivo tenga la extensión .cpp y que comience con:

```
#include <Rcpp.h>
using namespace Rcpp;
//////////////////////////////////////////////
# ADEMÁS PARA CADA FUNCIÓN QUE SE QUIERA EXPORTAR A R SE DEBE
   AÑADIR AL INICIO
//////////////////////////////////////////////
// [[Rcpp::export]]
```

En el libro de Wickham (Wickham, 2019) se puede ampliar más información del funcionamiento de este paquete y las interesantes aplicaciones que ofrece. En él se explica tanto la sintaxis como los tipos de datos, escalares como enteros, dobles, cadenas, booleanos, vectores, listas, tramas de datos o incluso funciones en R, que se pueden pasar como argumento a métodos de C++. Quizás las estructuras de datos más interesantes y más optimizadas son las más complejas, como los vectores ya mencionados, los conjuntos, o los mapas de datos.

En todo el proceso de uso del paquete Rcpp no se puede olvidar el especial cuidado que debe tenerse con los valores NA, ya que para vectores, C++ los toma como NA-REAL, NA-INTEGER, NA-LOGICAL, o NA-STRING, pero para el caso de los booleanos en R, sus valores son true, false y NA, pero en C++ un valor faltante se convertirá en TRUE.

Otra opción de la que dispone C++ es el uso del operador *, que permite no acceder a la referencia de un objeto, sino al valor en sí del mismo, por lo que se puede controlar el acceso a los datos como se desee.

Para la conversión de estructuras de datos de un lenguaje a otro se proporcionan varios métodos, as<T>(Rcpp) convierte un tipo de Rcpp a STD (librería estándar de C++), y wrap(Rcpp) convierte de la librería estándar STD al tipo equivalente de Rcpp.

5.1.3. Herramientas de comparación de rendimiento; bench

Es necesario utilizar herramientas para medir el grado de mejoría tras aplicar las técnicas de optimización aprendidas en este proyecto. Su uso nos permite comparar fragmentos de código para tareas específicas. Una solución que ofrece R es el paquete bench, el cual, hace uso de un temporizador de muy alta precisión que permite hacer mediciones que toman una cantidad de tiempo ínfima.

Este paquete internamente comprueba que cada ejecución de las que se van a comparar devuelven el mismo resultado o valor, que es lo que tiene realmente sentido en estas pruebas. Para comparar velocidades de expresiones que devuelven diferentes valores se puede configurar con el atributo check=FALSE. De manera predeterminada bench::mark() ejecuta las expresiones una vez, pero esto también puede personalizarse.

El resultado devuelto por esta herramienta es un tibble, en el cual, cada expresión de entrada ocupa una fila, y en las columnas aparecen una serie de medidores que se deben interpretar para sacar conclusiones de la comparación. Algunos de estos medidores indican los tiempos mínimo, máximo, la media o la mediana de la ejecución de cada expresión. Además, mem-alloc indica la memoria asignada a la primera ejecución, n-itr indica cuántas veces se evalúa la expresión y total-time indica el tiempo total que se tomó para evaluarlas.

Este resultado puede convertirse a un formato LaTex en R con ayuda del paquete knitr y la siguiente expresion, y para insertarlo en LaTeX se deberá hacer uso del paquete booktabs.

```
resultado %>% kable(format = 'latex', booktabs = TRUE)
```

5.2. Aplicación de técnicas de análisis y mejora de rendimiento de algoritmos del paquete fcaR

Se procede en este punto del trabajo de fin de grado al análisis y perfilado de cada uno de los principales algoritmos del paquete fcaR, buscando secciones de código que supongan cuellos de botella y evaluando si son candidatos a optimizarse, o, en cambio son lo suficientemente rápidos, o la posible mejoría que se pudiese alcanzar no resultara suficiente para justificar la inversión de tiempo en realizar dicha hazaña.

5.2.1. Método dual()

El algoritmo dual() del paquete fcaR toma un contexto formal y devuelve uno nuevo en el que los objetos y los atributos cambian sus roles, es decir, los objetos que antes estaban en las filas pasan a estar en las columnas y los atributos que estaban en las columnas se trasladan a las filas.

La eficiencia de dual() se ha analizado con un perfilador de código en R, que también toma en cuenta los algoritmos y funciones escritas en código C++, y se ha observado que la totalidad del tiempo de ejecución es consumido por la función que crea un nuevo contexto formal, como se refleja en la figura 6. Debido a la observación anterior, se ha analizado la inicialización de los contextos formales y se ha descubierto un gran cuello de botella en la ejecución interna de la función compute-grades(), la cual, está escrita en su totalidad en el lenguaje R y consume el 85 por ciento de los recursos. El culpable de esto recae sobre la función lapply, bastante ineficiente, por lo que se ha procedido a escribir el algoritmo en C++ para optimizarlo.

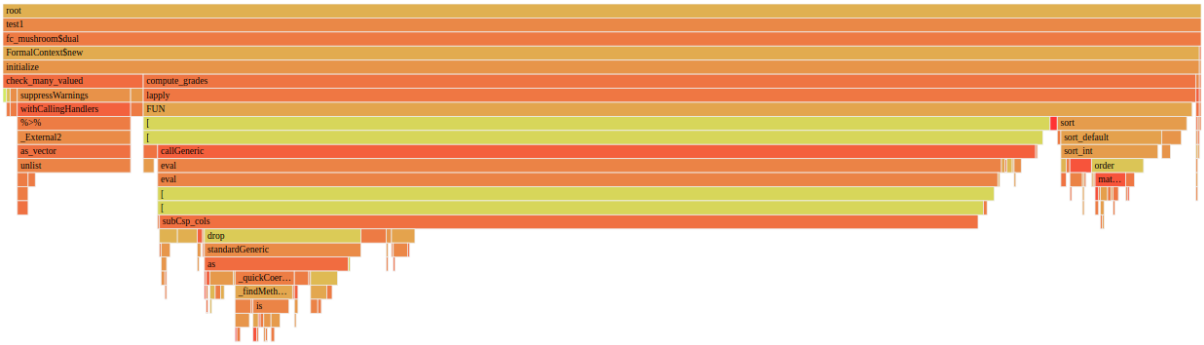

Figura 6: Perfilado del método dual() del paquete fcaR

El algoritmo que se propone como mejora, denominado compute-grades-C(), se ha introducido en el archivo lorenzo-optimization.cpp de la carpeta src del repositorio.

```
// [[Rcpp::export]]
List compute_grades_c(NumericMatrix mat) {

  List res;
  for(int i=0; i<mat.cols(); i++) {
    NumericVector v = mat(_,i);
    v.push_back(1);
    v.push_front(0);
    v = sort_unique(v);
    res.push_back(v);
  }
  return res;
}
```

Código.- 2: Algoritmo propio para acelerar la inicialización de contextos formales compute-grades-C

Una vez optimizado, se han realizado varias pruebas contabilizando el tiempo de ejecución. En dichas pruebas se ha medido un aumento en la rapidez del algoritmo dual(). La ventaja de haber optimizado este algoritmo es que muchas de los métodos del paquete hacen uso del mismo, por lo que, indirectamente, se está acelerando todo fcaR.

5.2.2. Método closure()

El algoritmo original para calcular el cierre de un contexto formal destina el 71,43 por ciento de los recursos temporales a la ejecución de la función Matrix::SparseMatrix() (en conjuntos de entrada de tamaño relativamente pequeño), que es el constructor de una matriz dispersa que posteriormente se pasa como parámetro en la creación del conjunto resultado.

Haciendo un profundo análisis del código, se ha llegado a la conclusión de que la función compute-closure() llamada anteriormente para el cálculo del cierre devuelve como resultado una matriz dispersa que puede ser enviada directamente a la función que crea el conjunto sin hacer uso de Matrix::sparseMatrix(). Con todo esto en mente se propone la sustracción de dicha función inservible del código del algoritmo que se está optimizando.

Si nos pasamos al marco de la realidad, en el que los casos de prueba son muchos más amplios y densos, compuestos por miles de objetos y cientos de atributos, la notable mejoría que se ha mencionado antes disminuye considerablemente. En este caso, la sección del código que provoca el cuello de botella está ligada a la función compute-closure, por lo que se ha hecho un intento de optimización sin un resultado aparente en este apartado.

La función que más recursos consume es Matrix::as.matrix(), de la cual se ha intentado prescindir, creando otras funciones equivalentes, pero a través del lenguaje de programación C++, que es mucho más eficiente y consume menos recursos.

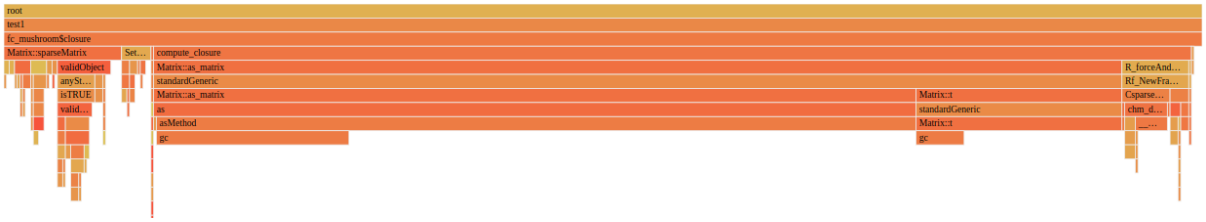

Figura 7: Perfilado del método closure() del paquete fcaR

Se han creado algoritmos cuyo objetivo era convertir un objeto S4 de R, consistente en una matriz dispersa del tipo dgCMatrix, a un objeto NumericMatrix del paquete Rcpp de una forma más eficiente que as.matrix(), para enviarlo como argumento después a las funciones que calculan el cierre. Sin embargo, ha resultado en un enlentecimiento en la ejecución. Esto es debido a que el algoritmo Matrix::as.matrix() ya está muy optimizado y hace uso de las mejores técnicas de programación internamente para consumir los mínimos recursos.

```
//Created by Lorenzo
//Converts S4 object into NumericMatrix, using vectors inside
NumericMatrix S4toNumericMatrix(S4 I) {

  std::vector<int> i = I.slot("i");
  std::vector<int> p = I.slot("p");
  std::vector<double> x = I.slot("x");
  std::vector<int> adims = I.slot("Dim");
  //NumericVector vect(adims[0]*adims[1]);
  std::vector<double> vect(adims[0]*adims[1]);

  int it = 0;
  for(long long unsigned int k=0; k<p.size()-1; k++) {
    int ant = p[k];
    int post = p[k+1];
    int aux = post - ant;
```

```
    int cont = 0;

    for(int j=0; j<adims[0]; j++) {
      if(cont<aux) {
        if(j==i[ant]) {
          vect[it] = x[ant];
          ant++;
          cont++;
        } else{
          vect[it] = 0;
        }
      } else {
        vect[it] = 0;
      }
      it++;
    }
  }
  //print(vect);
  //NumericVector s = wrap(vect);
  NumericMatrix res(adims[0], adims[1], vect.begin());

  return(res);
}
```

Código.- 3: Código propio para convertir un objeto S4 en un NumericMatrix

5.2.3. Métodos intent() y extent()

El cálculo de la intensión de un contexto formal se lleva a cabo a través del algoritmo intent() dentro del paquete fcaR, cuya ejecución destina el 15 por ciento de los recursos temporales en computar una conversión de tipos, que al igual que en el análisis de la eficiencia del algoritmo anterior (el cierre), siguiendo la filosofía de hacer lo mínimo posible, y respetando

el funcionamiento correcto, esta no es necesaria, por lo que se hace la propuesta de eliminarla del código.

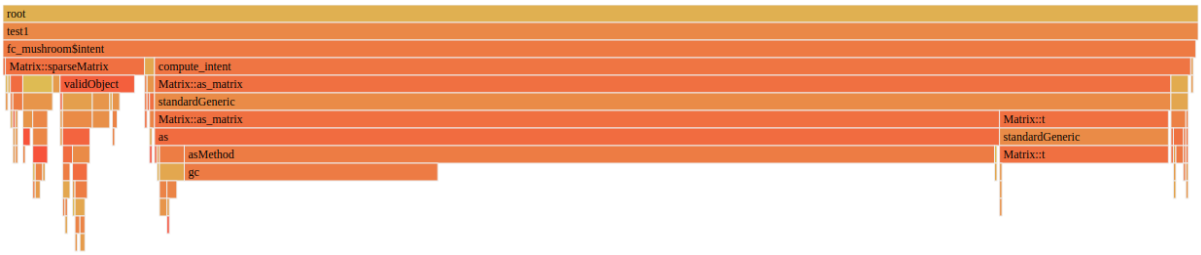

Figura 8: Perfilado del método intent() del paquete fcaR

Debido a esta pequeña optimización, cuando el algoritmo se ejecuta bajo entradas de datos relativamente pequeñas, la eficiencia consigue aumentar considerablemente a diferencia de cuando las entradas comienzan a crecer.

En cuanto al verdadero cuello de botella existente y analizado, ligado a la función del cálculo de la intensión en sí, compute-intent(), ocurre exactamente lo mismo que con el análisis del cierre, en el que la función Matrix::as.matrix() acapara todos los recursos (85 por ciento) y el intento de optimización a través de la función S4toNumericMatrix() en C++ no resulta exitoso.

Al mismo modo, la extensión de un contexto formal con la función extent() sigue un camino muy similar, en el que se han aplicado las mismas técnicas de optimización de recursos que con la función intent(), obteniéndose resultados parecidos.

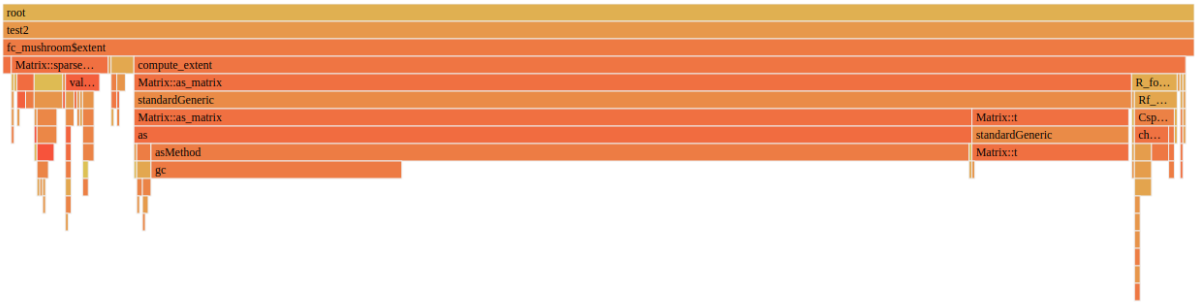

Figura 9: Perfilado del método extent() del paquete fcaR

5.2.4. Método clarify()

El algoritmo clarify actúa sobre un contexto formal para eliminar la redundancia dentro de los atributos y objetos del mismo. También recibe como parámetro un booleano, que en caso de ser TRUE se crea un nuevo contexto formal con el algoritmo ya aplicado, y en caso de ser FALSE, el contexto formal actual se sobreescribe sin redundancia.

En el perfilado de la ejecución se han percibido varios cuellos de botella, de los cuales destacan la creación de un nuevo contexto formal con FormalContext-new() y las llamadas internas que realiza, y la ejecución de la función clarify-matrix(), que consume el 55 por ciento de los recursos de tiempo.

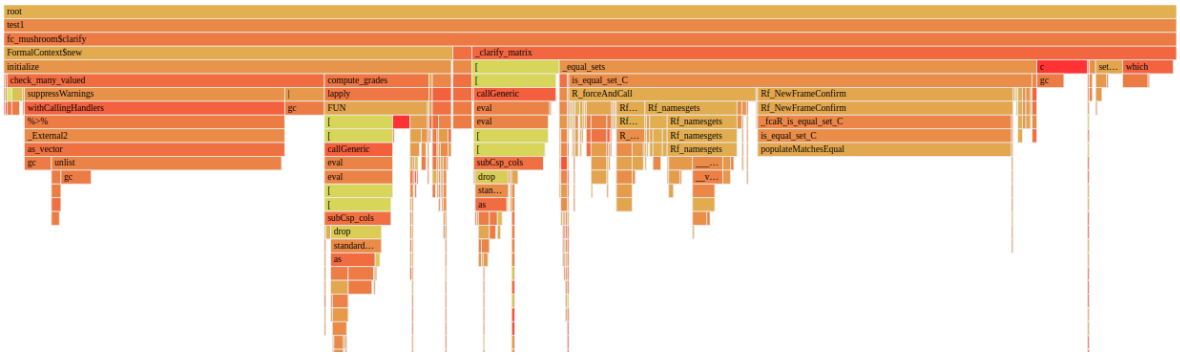

Figura 10: Perfilado del método clarify() del paquete fcaR

Entrando en el estudio del código que lleva a ejecutarse esta última función, se observa que populateMatchesEqual() representa un parte importante de dicha carga, y su posible aceleración conseguiría resultados notables. Sin embargo, dicha función está ampliamente optimizada, ya que sigue muy buenas prácticas de optimización de algoritmos como son la escritura del mismo en el lenguaje C++, el uso de estructuras de datos muy eficientes. También, se reserva y libera la memoria utilizada conforme se ha necesitado, se usan bucles en situaciones estrictamente necesarias o se usa la librería estándar, que se considera óptima y es ampliamente usada.

Sin embargo, se ha realizado un intento de optimización de la misma. El enfoque seguido para acelerar populateMatchesEqual ha consistido en la idea de crear una función igual, que únicamente funcione con matrices de entrada binarias. Al hacer esto, se suprimen del código varias comprobaciones extra que se debían hacer cuando la entrada era difusa, por lo que el rendimiento en el caso binario ha aumentado ligeramente, como se observará en el siguiente capítulo.

5.2.5. Método reduce()

Dentro del paquete se encuentra una función que reduce el contexto formal, eliminando las redundancias y dando como resultado un contexto formal equivalente al original, manteniendo el conocimiento. Esta función se denomina reduce(), quien internamente también llama a clarify(), analizada anteriormente. Es, por tanto, la ejecución de reduce(), la que devuelve un contexto formal sin repeticiones ni redundancia equivalente al original. Cabe destacar, que este método solo está implementado para funcionar con conjuntos de entrada binarios, o lo que es lo mismo, no difusos.

Haciendo un perfilado con un conjunto de pruebas que se aproxima al marco de la realidad (mushroom), se ha observado que el 67 por ciento de los recursos son consumidos por el cálculo de la extensión de contextos formales extent(), lo que se puede llegar a considerar un cuello de botella en el código. En la figura 11 se puede observar el comportamiento del código.

Es, por tanto, que se han realizado pruebas de rapidez a la función original comparada con

una versión en la que se utiliza el cálculo de las extensiones ya optimizado. Esta propuesta consigue mejorar el tiempo de procesamiento, como se comprobará en el siguiente capítulo, resultados experimentales.

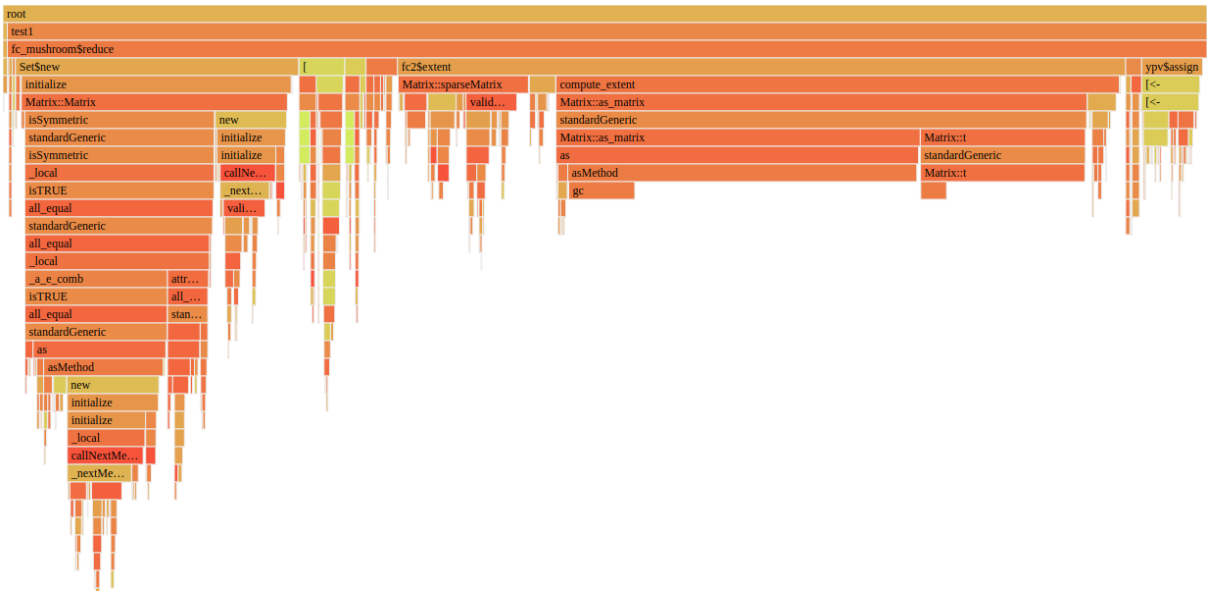

Figura 11: Perfilado del método reduce() del paquete fcaR

La optimización previa de funciones como clarify() o FormalContext-initialize() indirectamente también han contribuido al aumento de eficiencia de esta función en menor medida, y esto se ha de tener en cuenta para el proceso de optimización de próximas funciones.

5.2.6. Método find-implications()

La función encargada del cálculo de las implicaciones y del retículo de conceptos de un contexto formal es find-implications(). Esta hace uso de dos clases creadas dentro del paquete, ConceptLattice para el retículo de conceptos, e ImplicationSet para guardar el conjunto de implicaciones dentro del mismo contexto formal. Esta función puede recibir como parámetro un booleano que en caso de ser TRUE, la función guardará el retículo de conceptos mencionado antes, o FALSE, en cuyo caso no se guarda y simplemente se computan las implicaciones, lo que hace que la función se ejecute ligeramente más rápido.

Esta función es una de las más costosas computacionalmente hablando, y debido a su complejidad, los autores y creadores del paquete fcaR decidieron pasar a lenguaje C++ la parte del código con mayor carga de trabajo, y con ello, seguir las buenas prácticas de la escritura de código enfocada a la optimización y el ahorro de recursos.

Por su complejidad, solo se han podido hacer pruebas de perfilado para contextos formales de tamaño reducido, como es el contexto formal generado a través del conjunto de datos cobre32 que está incluido en el paquete. Las pruebas con contextos formales reales como lo es mushroom no han sido exitosas, ya que para el perfilado se ha utilizado una librería que solo está disponible para sistemas operativos basados en Linux y en MacOS, y es por ello que se ha hecho empleo de una máquina virtual con el sistema operativo Ubuntu instalado, cuya potencia computacional dependía de un único procesador y 2 GB de memoria RAM.

Sin embargo, a pesar de todo, se ha podido llegar a ver que todos los recursos son consumidos por la función interna next-closure-implications(), con casi un 100 por ciento del tiempo de ejecución total. Esta función realiza una cantidad ingente de operaciones internamente, ya que es un algoritmo de complejidad exponencial, y aun estando optimizado, es normal que consuma tantos recursos. Es por ello que aun siendo un cuello de botella, todas estas operaciones son necesarias para el funcionamiento correcto del algoritmo. Es por esta gran cantidad de operaciones que el perfilado no detecta nada más que esta función, resultando en un intento de optimización complicado.

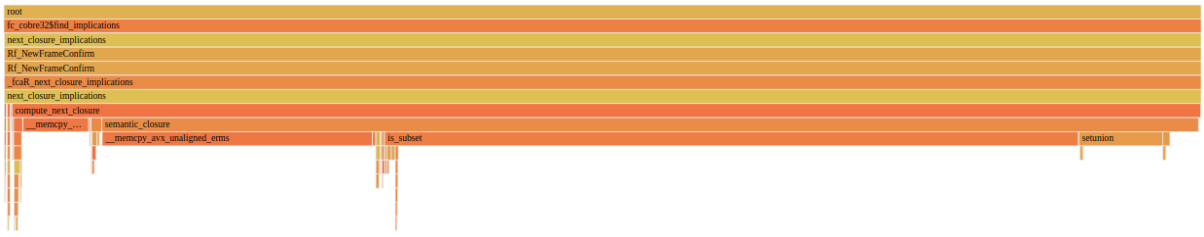

Figura 12: Perfilado del método find-implications() del paquete fcaR

Viendo detenidamente la figura 12, se comprueba que el problema computacional proviene del método is-subset(), el cual, es llamado por semantic-closure(), que a su vez es invocado por compute-next-closure(). Todos ellos están implementados en C++.

Estos algoritmos hacen un uso eficiente de la memoria, reservando la necesaria para realizar las operaciones, y liberándola cuando ya no es necesaria para realizar más cálculos. Además, se hacen uso de estructuras de datos internas muy eficientes, cuyos métodos de reserva de memoria dinámica hacen de estas estructuras las más rápidas. Entre estas estructuras se encuentran los arrays con reserva de memoria dinámica, y los árboles, los cuales son de las estructuras más eficientes actualmente en la programación.

5.2.7. Métodos infimum() y supremum()

infimum() es un método perteneciente a la clase ConceptLattice que calcula el ínfimo de los conceptos. Este recibe como parámetros de entrada una lista de índices de tipo entero o una lista con los conceptos separados por comas. Estos conceptos sirven para hacer el cálculo del ínfimo dentro del retículo.

Analizando la figura 13, se aprecia como el método is-subset-C() representa el código que mayor uso de recursos presenta. Se considera, pues, que se encuentra un cuello de botella en esta función que toma el 65 por ciento del tiempo de ejecución.

Internamente, is-subset-C() hace llamadas en bucle a la función populateMatches(), causa de la complejidad de este algoritmo, quien posee un doble bucle anidado que también realiza numerosas operaciones. Como se verá en los próximos algoritmos que procedemos a optimizar, populateMatches() representa un cuello de botella en todos ellos, por lo que su fructífera aceleración supondrá el aumento en la eficiencia de todos ellos.

Para acelerar el método populateMatches(), se ha investigado detenidamente el funcionamiento del mismo, tanto los parámetros de entrada como las salidas, además de las operaciones internas. Finalmente, se ha detectado una posible optimización en el código que consigue aumentar el rendimiento del algoritmo notablemente.

El método calcula realmente si un conjunto es subconjunto de otro, por lo que se ha añadido en el código una condición que, de ser falsa, se saldría del bucle y no seguiría consumiendo recursos innecesariamente, porque se rechazaría la hipótesis de ser subconjunto. Esta optimización permite al algoritmo ahorrarse muchas operaciones si se detecta que un elemento de un conjunto no es subconjunto del otro, cuando debería serlo para corroborar la hipótesis inicial.

Además, también se propone aplicar un enfoque únicamente binario que permita el ahorro de varias comprobaciones en el código, resultando en el aumento de rendimiento del método cuando las matrices de entrada son binarias y no difusas.

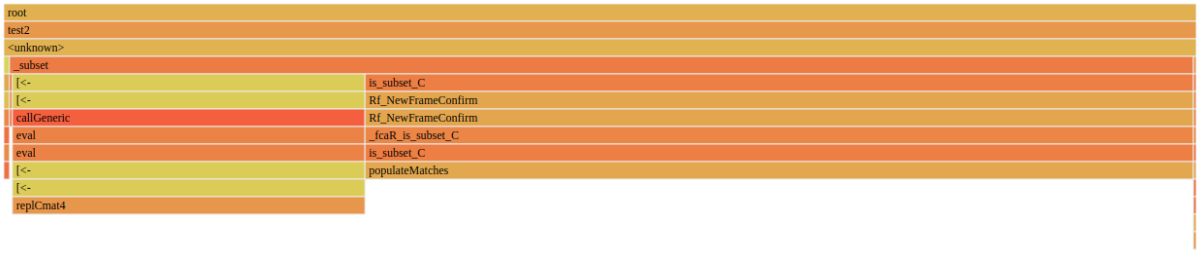

Figura 13: Perfilado del método infimum() del paquete fcaR

Del mismo modo, supremum() recibe como argumento los mismos parámetros que infimum(), pero en este caso para calcular el supremo entre los conceptos del retículo. Además, presenta el mismo problema de eficiencia que infimum(), por lo que el trato para su optimización pasa por el mismo procedimiento.

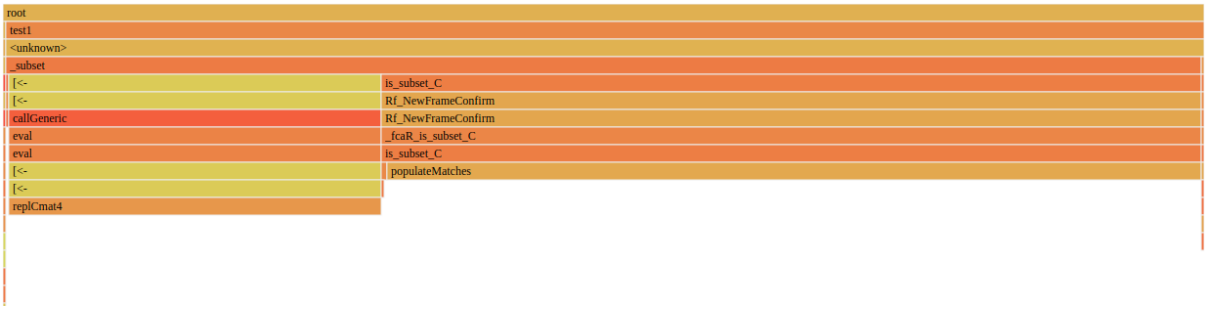

Figura 14: Perfilado del método supremum() del paquete fcaR

5.2.8. Métodos join-irreducibles() y meet-irreducibles()

Las funciones join-irreducibles() y meet-irreducibles(), dado un retículo de conceptos calculado previamente, devuelven los elementos irreducibles con respecto al supremo y al ínfimo, dos funciones que también se encuentran en el paquete como supremum() e infimum().

Tras analizar el código dentro de la clase ConceptLattice, se observa como ambas funciones hacen llamadas a dos métodos principalmente, un método subset(), quien supone la mayor carga computacional con un 80 por ciento de ocupación en la ejecución, y un método denominado reduce-transitivity() con un 20 por ciento. La mayor carga está implementada en lenguaje C++ con funciones como is-subset-C(). Esta última ya se ha analizado en métodos anteriores, por lo que la aceleración de estos dos algoritmos pasan por la optimización de populateMatches(), quien se ha conseguido optimizar con éxito.

Figura 15: Perfilado del método join-irreducibles() del paquete fcaR

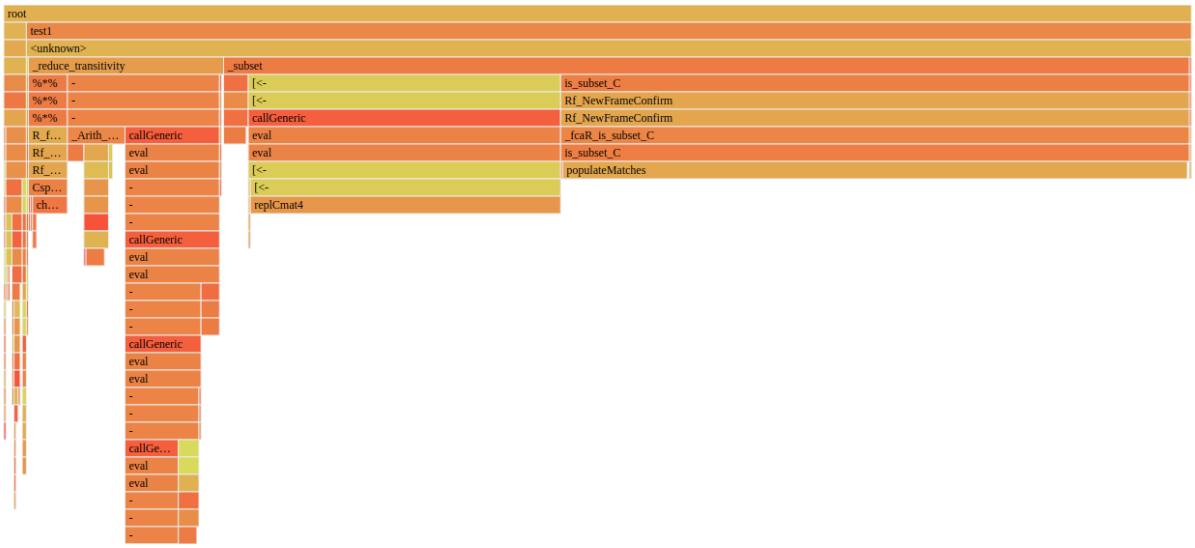

Figura 16: Perfilado del método meet-irreducibles() del paquete fcaR

Se ha observado que el código utilizado para calcular estos algoritmos es uno de los más veloces para el cálculo de subconjuntos. El autor del código es Ian Johnson y recoge todas las operaciones sobre subconjuntos de matrices dispersas.

5.2.9. Método standardize()

Este método construye el contexto estándar a través del contexto formal. Para su uso, se deben haber calculado previamente todos los conceptos del mismo, y para obtener el resultado utiliza los elementos de join-irreducibles() y meet-irreducibles().

En la figura 17 se muestra el perfilado de standardize(), donde se aprecia que el método definido para comparar conceptos <= consume hasta el 60 por ciento del tiempo de ejecución. Este método hace uso de un algoritmo de comparación de vectores ya de por sí eficiente, por lo que cualquier intento de mejora no es viable.

```
1 all(C1$get_vector() <= C2$get_vector())
```

Código.- 4: Comparación de vectores para comprobar la propiedad de la subpertenencia.

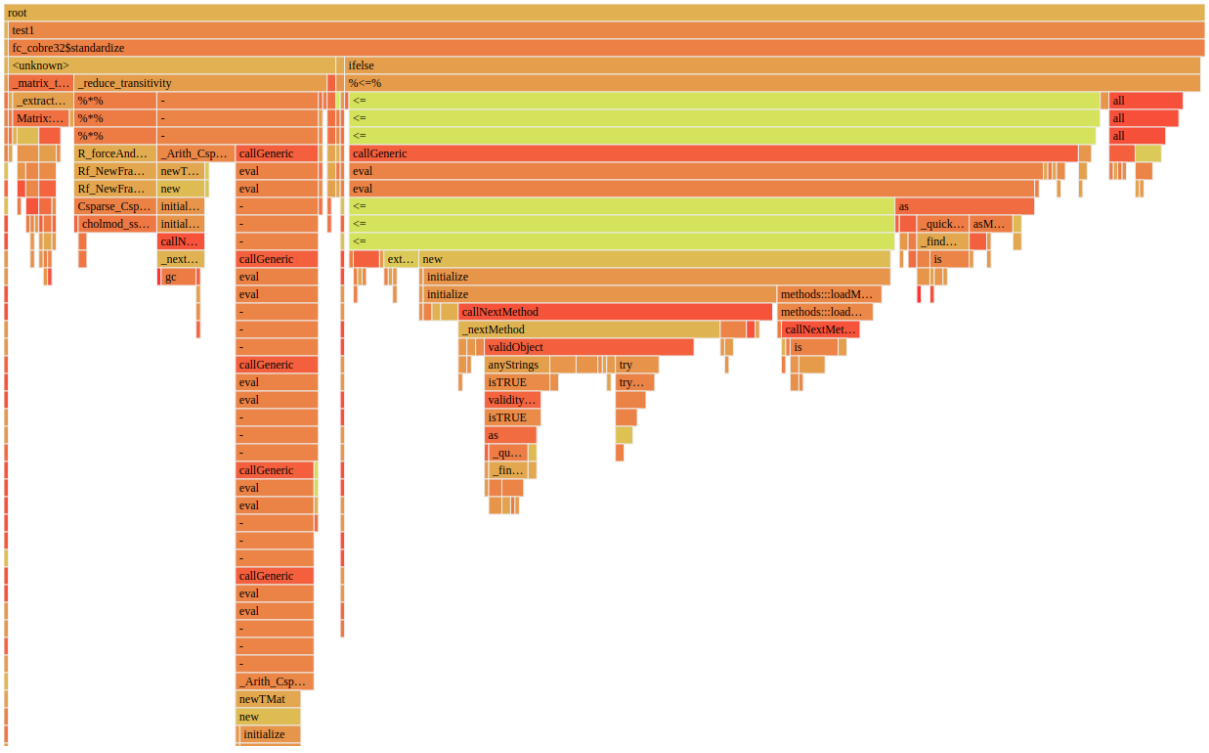

Figura 17: Perfilado del método standardize() del paquete fcaR

Sin embargo, como se observa, el algoritmo initialize(), que ha sido optimizado previamente, aparece en numerosas ocasiones en la pila de llamadas de este perfilado, por lo que, indirectamente se ha conseguido acelerar este método.

5.2.10. Métodos sublattice(), subconcepts() y superconcepts()

El método sublattice() toma un retículo de conceptos para calcular un sub retículo pasando como argumento, o bien, una lista de índices enteros, o una lista de conceptos, los cuales, serán usados para generar un sub retículo en forma de objeto ConceptLattice.

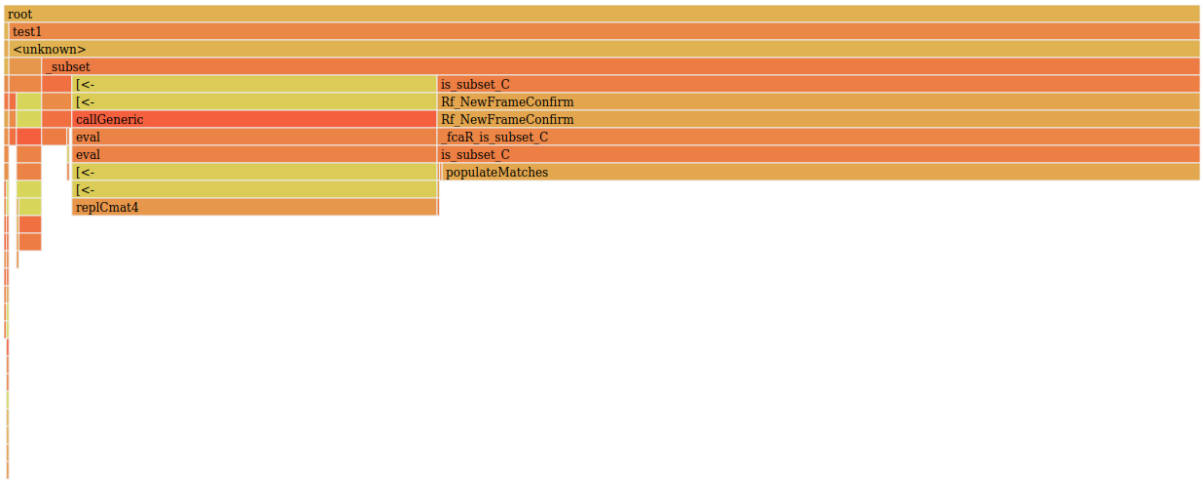

Figura 18: Perfilado del método sublattice() del paquete fcaR

Por otro lado, los métodos subconcepts() y superconcepts() calculan los subconceptos y los superconceptos de un concepto pasado como parámetro al método, y sobre un retículo de conceptos dado. El resultado devuelto consiste en una lista con los subconceptos y superconceptos respectivamente.

La razón de la inclusión de estos tres métodos en la misma sección de esta memoria reside en que el comportamiento de los tres es muy similar, en cuanto al perfilado se refiere. En estos casos, el algoritmo populateMatches() vuelve a ser el protagonista, por lo que, al igual que se ha comentado anteriormente, al optimizarlo, los tres métodos se acelerarán a la vez.

Figura 19: Perfilado del método subconcepts() del paquete fcaR

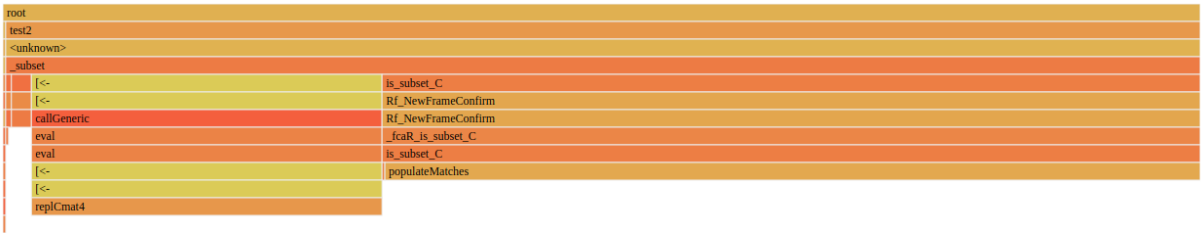

Figura 20: Perfilado del método superconcepts() del paquete fcaR

Como se verá en el capítulo 6 de esta memoria, gracias a la exitosa optimización de populateMatches(), estos 3 métodos aumentan su rendimiento considerablemente.

6

Resultados experimentales

Para comprobar los resultados obtenidos se han generado casos de prueba para todos los algoritmos que se han intentado optimizar, ubicados en la carpeta optimization-tests de la rama lorenrd del repositorio del paquete fcaR. En cada uno de ellos se han realizado pruebas de rendimiento con diferentes conjuntos de datos de entrada, un conjunto más reducido para la realización de pruebas que se ejecutan en poco tiempo y pueden probarse tantas veces se quiera debido a esto (cobre32, perteneciente al paquete fcaR), y un conjunto mucho más grande que se acerca a los casos reales, el cual consume mucho tiempo por iteración, y su reiterada ejecución no es viable (mushroom, perteneciente al paquete arules).

En estas pruebas se han comparado varias ejecuciones de diferentes implementaciones del código, siempre nombrando como test() con el número más bajo a la implementación original del paquete fcaR. En todas las tablas mostradas, la medida más importante por la que se ha guiado la optimización es el atributo median, que aproxima el valor medio del tiempo empleado para ejecutar cada una de las implementaciones. Además, los parámetros iterations y check=FALSE, se refieren al número de iteraciones de cada implementación para luego hacer la media, y a la comprobación de que la salida de todas las implementaciones es la misma. En ocasiones, este parámetro ha tomado el valor FALSE a causa de la creación de la clase FormalContext-opt, que al llamarse de forma diferente a la original, bench considera que las salidas de las implementaciones son distintas.

Cabe destacar que, a parte de optimizar el código de todos los algoritmos que aparecen a continuación, también se ha aplicado una corrección en un método del paquete, el cual, no estaba funcionando correctamente, o como se supone que debería hacerlo.

```
SparseVector S4toSparse(S4 A) {

  std::vector<int> ap = A.slot("p");
  std::vector<int> ai = A.slot("i");
  std::vector<double> ax = A.slot("x");
  IntegerVector adims = A.slot("Dim");
  SparseVector V;
  initVector(&V, adims[0]);
  for (size_t i = 0; i < ai.size(); i++) {
    insertArray(&(V.i), ai[i]);
    insertArray(&(V.x), ax[i]);
  }
  for (size_t i = 0; i < ap.size(); i++) {
    insertArray(&(V.p), ap[i]);
  }

  return V;
}
```

Código.- 5: Corrección al código erróneo del paquete fcaR en el método S4toSparse()

Además, para los casos en que la ejecución de los algoritmos de mayor complejidad toma demasiado tiempo, aproximándose al marco de las horas en el conjunto de entrada mushroom, se ha añadido al paquete una función que crea una matriz de pruebas según los parámetros que se elijan. A esta función se le indican el número de objetos, atributos, los valores que pueden tomar las celdas y la densidad de la misma, generando aleatoriamente el conjunto deseado. En las pruebas, se ha usado para generar matrices binarias de entrada para los algoritmos.

```
generate_context <- function(n_objects = 500,
                             n_attributes = 20,
                             grades = c(0, 1),
                             density = 0.4) {
  grades <- grades[grades > 0]
  v <- vector(mode = "numeric",
              length = n_objects * n_attributes)
  idx <- sample.int(n = n_objects * n_attributes,
                    size = round(density * n_objects * n_
   attributes))
  vals <- sample(grades,
                 size = round(density * n_objects * n_
   attributes),
                 replace = TRUE)
  v[idx] <- vals
  I <- matrix(v, nrow = n_objects, ncol = n_attributes)
  colnames(I) <- paste0("A", seq(n_attributes))
  rownames(I) <- paste0("O", seq(n_objects))
  return(I)
}
```

Código.- 6: Generador de contextos formales de prueba aleatorios.

6.1. Método dual()

Tras la aplicación de las propuestas de mejora, y habiendo optimizado la inicialización de nuevos contextos formales a través del lenguaje C++, se ha obtenido una mejoría muy notable, con un factor de incremento de velocidad de 4,7 puntos. Esto quiere decir que casi se ha quintuplicado la velocidad de ejecución para conjuntos de entradas relativamente pequeños como es cobre32. Esta comparación se ha realizado en 1000 ejecuciones distintas de las implementaciones y se ha calculado el valor de la media de tiempo de ejecución del total.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test2()	14.7ms	15.18ms	65.26875	112	13.61s	676KB
test3()	2.99ms	3.23ms	301.85255	11	3.28s	581KB

Cuadro 1: Resultado del benchmark de dual() para la entrada cobre32 (1000 iteraciones)

Por otro lado, para casos en los que los conjuntos de entrada crecen y se asemejan a entradas reales, como el conjunto mushroom, se ha observado una mejoría de un factor de 2,16, llegando a la conclusión de que, para estos casos, el tiempo de ejecución medio se ha reducido a menos de la mitad.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test2()	4.16s	4.37s	0.2242386	25	22.3s	157MB
test3()	1.83s	2.02s	0.4930268	46	10.1s	387MB

Cuadro 2: Resultado del benchmark de dual() para la entrada mushroom (5 iteraciones)

Con esto, el nuevo algoritmo a añadir al paquete, aumentaría la eficiencia de ciertas operaciones considerablemente. Además, el algoritmo de inicialización de contextos formales es llamado en numerosas ocasiones, por lo que, indirectamente, se mejorarán varios algoritmos más.

6.2. Método closure()

La función compute-closure() devuelve como resultado una matriz dispersa que puede ser enviada directamente a la función que crea el conjunto sin hacer uso de Matrix::sparseMatrix(), por lo que se sustrae dicha conversión de tipos del código y se consigue acelerar, siguiendo la filosofía de Wickham de hacer lo menos posible.

La expresión test2() corresponde con la implementación original del paquete, test3() es la implementación con la optimización de la iniciación del contexto formal aplicada, test4(),

test5() y test() son implementaciones propias basadas en vectores y en matrices. Como se observa en el cuadro 3, la implementación más rápida es el test5(), el cual utiliza vectores para calcular el cierre. La velocidad para pruebas con el conjunto de entrada cobre32 aumenta en un factor de 3,4 en la velocidad. Otro apunte interesante es que el uso de memoria tras la mejora del algoritmo se reduce casi hasta la mitad. Estas pruebas se han realizado bajo 10.000 iteraciones.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test2()	540µs	577µs	1620.665	43	6.14s	76.1KB
test3()	539µs	574µs	1643.558	43	6.06s	39.5KB
test4()	169µs	178µs	5022.204	12	1.99s	39.5KB
test5()	156µs	170µs	5412.966	10	1.85s	38.3KB
test6()	167µs	180µs	5216.916	9	1.92s	38.3KB

Cuadro 3: Resultado del benchmark de closure() para la entrada cobre32 (10000 iteraciones)

En casos reales, el test4() y el test5(), equivalentes a las versiones de las implementaciones del test5() y test6() anteriores, no llegan a ser más óptimos que la implementación original. Sin embargo, la implementación propuesta que elimina la conversión de tipos innecesarios si ve incrementada su velocidad un 11 por ciento con respecto al original.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test1()	5.49ms	6.6ms	146.57055	118	6.02s	9.3MB
test2()	5.52ms	6.66ms	146.75059	118	6.01s	9.3MB
test3()	4.94ms	5.91ms	169.01867	120	5.41s	9.3MB
test4()	11.45ms	12.65ms	77.11353	125	11.35s	9.24MB
test5()	13.22ms	14.28ms	68.41902	128	12.74s	9.24MB

Cuadro 4: Resultado del benchmark de closure() para la entrada mushroom (1000 iteraciones)

El algoritmo existente para el cálculo del cierre sigue muchas de las buenas prácticas de la programación enfocada a la optimización del código. Por otro lado, el uso de la función que

acelera el algoritmo un 10 por ciento podría usarse como reemplazo a la actual, puesto que se aprecia la mejoría aun siendo leve.

6.3. Métodos intent() y extent()

Debido a esta pequeña optimización, cuando el algoritmo se ejecuta bajo entradas de datos relativamente pequeñas, la eficiencia consigue aumentar considerablemente a diferencia de cuando las entradas comienzan a crecer. Ahí se asemejan a los casos reales de entradas de datos, en cuyo caso, esta optimización consigue aumentar la rapidez de ejecución en un 12 por ciento aproximadamente según las pruebas aportadas.

En estas pruebas para el cálculo de la extensión e intensión se sigue la misma filosofía aplicada al cálculo anterior del cierre. La optimización pasa por eliminar una sentencia del código inservible que consumía recursos correspondiente a una conversión de tipos.

Para el método intent(), con un parámetro de 10.000 iteraciones establecido, se observa que la optimización ha conseguido una mejoría del 335 por ciento con respecto a la implementación original para el conjunto de pruebas cobre32.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test3()	522µs	566µs	1667.097	43	5.97s	39.9KB
test4()	524µs	569µs	1658.064	43	6s	39.9KB
test5()	158µs	169µs	5502.753	12	1.81s	39.9KB

Cuadro 5: Resultado del benchmark de intent() para la entrada cobre32 (10000 iteraciones)

Por otra parte, para el conjunto de entrada mushroom y 1.000 ejecuciones, se ha calculado una mejoría media del 16,82 por ciento.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test3()	5.28ms	6.53ms	146.4737	121	6s	9.33MB
test4()	5.26ms	6.39ms	147.2512	118	5.99s	9.33MB
test5()	4.7ms	5.59ms	173.5661	119	5.08s	9.33MB

Cuadro 6: Resultado del benchmark de intent() para la entrada mushroom (1000 iteraciones)

Al mismo modo, la extensión de un contexto formal con la función extent() sigue un camino muy similar, en el que se han aplicado las mismas técnicas de optimización de recursos que con la función intent(), obteniéndose resultados parecidos, aunque en este caso la mejoría es del 339 por ciento en el consumo de recursos de tiempo para cobre32 y del 16,2 por ciento para el conjunto de entrada mushroom.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test6()	533µs	584µs	1594.243	44	6.25s	39.5KB
test7()	533µs	576µs	1611.409	43	6.18s	39.5KB
test8()	162µs	172µs	5515.720	12	1.81s	39.5KB

Cuadro 7: Resultado del benchmark de extent() para la entrada cobre32 (10000 iteraciones)

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test6()	5.5ms	6.59ms	147.1571	120	5.98s	9.3MB
test7()	5.49ms	6.62ms	148.3792	118	5.94s	9.3MB
test8()	4.72ms	5.67ms	169.7029	119	5.18s	9.3MB

Cuadro 8: Resultado del benchmark de extent() para la entrada mushroom (1000 iteraciones)

Se considera, pues, que el algoritmo en uso para el cálculo de la intensión y la extensión de contextos formales dentro del análisis formal de conceptos del paquete fcaR es adecuado; sin embargo, puede ser sustituido por la nueva versión desarrollada, la cual, hace un mejor uso de los recursos e incrementa la velocidad de ejecución de los mismos.

6.4. Método clarify()

Se ha mejorado el rendimiento para los casos binarios gracias a la aceleración de populateMatchesEqual() en estos casos, pero en la entrada cobre32 no se aprecia debido a que se trata de un conjunto de pruebas difuso. Además, se ha notado un incremento de velocidad en cuestión debido al incremento de rapidez de FormalContext-new(), el cual tomaba el 45 por ciento de la ejecución.

Para el conjunto de entrada cobre32, la mejoría tras haber optimizado la inicialización de contextos formales hace que el algoritmo clarify() incremente su rendimiento un 78,9 por ciento de media, comprobado con un respaldo de 1.000 iteraciones en la comparación de velocidad de ejecución entre los diferentes implementaciones. También se ha notado una mejora en el consumo de memoria, reduciéndose ligeramente.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test2()	7.42ms	8.18ms	119.4959	31	8.11s	1.07MB
test3()	4.24ms	4.57ms	214.2573	15	4.6s	430.37KB

Cuadro 9: Resultado del benchmark de clarify() para la entrada cobre32 (1000 iteraciones)

En el marco de la realidad, al pasar como conjunto de entrada mushroom, la implementación ha mejorado en un factor de 10,5 por ciento en la velocidad de ejecución, percibiéndose un pequeño ahorro en el uso de la memoria. Para los casos binarios mencionados anteriormente, este aumento supone un 15 por ciento de velocidad de ejecución frente al 10,5 del algoritmo no optimizado para conjuntos binarios.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test2()	1.07s	1.15s	0.8642372	35	11.6s	313MB
test3()	1.01s	1.04s	0.9299362	40	10.5s	296MB
test4()	950ms	1.00s	1.0023015	40	10.1s	296MB

Cuadro 10: Resultado del benchmark de clarify() para la entrada mushroom (10 iteraciones)

6.5. Método reduce()

La optimización de métodos como clarify(), FormalContext-initialize() o extent() han hecho que dual() aumente también su rendimiento, ya que, en la pila de llamadas, aparecen estas funciones, y extent() supone un cuello de botella como se ha mencionado en el capítulo anterior.

Podemos ver, en el cuadro 11, como el test3() con la implementación que utiliza los métodos mencionados ya optimizados consigue mejorar a su predecesor por un factor de 1,13, o lo que es lo mismo, una mejoría del 13 por ciento. Además, se observa un menor uso de memoria y un menor número de llamadas al recolector de basura.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test3()	2.10m	2.10m	0.0079890	763	2.13m	93.3GB
test2()	2.31m	2.31m	0.0071901	766	2.34m	94.3GB
test1()	2.37m	2.37m	0.0070230	770	2.37m	94.3GB

Cuadro 11: Resultado del benchmark de reduce() para la entrada mushroom (1 iteración)

6.6. Método find-implications()

El cálculo del método find-implications() tiene una complejidad exponencial, por lo que cualquier implementación que se utilice para su ejecución tendrá esta complejidad como mínimo. Actualmente, el algoritmo que provoca el cuello de botella no se ha podido optimizar, ya que es de los más eficientes que existen.

Tras hacer pruebas de comparación de tiempos de ejecución, el método original comparado con la implementación optimizada, tras mejorar la inicialización de los contextos formales y los algoritmos mencionados anteriores a este, ha mejorado ligeramente. En la siguiente tabla se muestra un incremento de rendimiento del 11,5 por ciento para una prueba con un conjunto de entrada difuso, cobre32.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test1()	1.37m	1.37m	0.0121325	1	1.37m	3.18MB
test2()	1.23m	1.23m	0.0135016	1	1.23m	3.12MB

Cuadro 12: Resultado del benchmark de find-implications() para la entrada cobre32 (1 iteración)

Si esta prueba se hace con el conjunto de pruebas binario mushroom, el tiempo que tarda el algoritmo original en ejecutarse es de 23,4 minutos frente a los 21,1 minutos del algoritmo optimizado. Esto supone una mejoría nada despreciable del 11 por ciento, que supone más de 2 minutos de ahorro para esta entrada.

expression	min	median	itr/sec	n_gc	total_time	mem_alloc
test1()	23.4m	23.4m	0.0007108	24	23.4m	281MB
test2()	21.1m	21.1m	0.0007805	24	21.1m	250MB

Cuadro 13: Resultado del benchmark de find-implications() para la entrada mushroom (1 iteración)

6.7. Métodos infimum() y supremum()

La optimización de los métodos infimum() y supremum() pasaba por la optimización de populateMatches(), que consumía casi el 60 por ciento de los recursos. Para estas pruebas se ha comparado la implementación original del paquete con una versión optimizada y otra ver-

sión optimizada igual que la anterior pero enfocada a las entradas binarias.

Debido a la naturaleza de estos métodos, se ha tenido que cambiar de herramienta de comparación de tiempo de ejecución. Internamente, de aquí a todos los algoritmos que quedan por analizar, se ha usado la función system.time(), la cual, recibe una función que ejecuta y mide el tiempo que ha tardado. Este método es algo más rudimentario que el uso de bench::mark(), pero es necesario.

Esto es debido a que los algoritmos, cuando se ejecutan la primera vez, almacenan el resultado en una variable, y cuando se llaman por segunda vez recuperan estos datos sin necesidad de ejecutar todas las operaciones. Es por ello que bench::mark() no sirve para estos casos, ya que finalmente muestra el tiempo que tarda la función en devolver una simple variable.

En el caso de supremum() para la entrada cobre32 se ha observado una mejoría del 35,7 por ciento frente a la implementación del paquete fcaR, mientras que para infimum(), la mejora ha sido del 37 por ciento.

expression	user	time	elapsed
test1()	11.24s	0.01s	11.26s
test2()	8.24s	0.06s	8.30s

Cuadro 14: Resultado del benchmark de supremum() para la entrada cobre32 (1 iteración)

expression	user	time	elapsed
test1()	11.59s	0.08s	11.67s
test2()	8.46s	0.02s	8.52s

Cuadro 15: Resultado del benchmark de infimum() para la entrada cobre32 (1 iteración)

En las dos siguientes tablas se observa la comparación entre el original, y las versiones con populateMatches() optimizado, estando la última enfocada a los conjuntos binarios de prueba.

La mejoría para supremum() es de un increíble 82,3 por ciento y un 85 para el caso binario. Para infimum() se observan mejorías de rendimiento del 73,8 por ciento y 76,5 respectivamente. Esto se traduce en que los algoritmos tardan en ejecutarse casi la mitad de tiempo que el original.

expression	user	time	elapsed
test1()	26.92s	0.02s	26.94s
test2()	14.76s	0.02s	14.78s
test3()	14.55s	0.04s	14.59s

Cuadro 16: Resultado del benchmark de supremum() para la entrada binaria aleatoria (1 iteración)

expression	user	time	elapsed
test1()	39.67s	0.07s	39.74s
test2()	22.84s	0.03s	22.87s
test3()	22.45s	0.06s	22.51s

Cuadro 17: Resultado del benchmark de infimum() para la entrada binaria aleatoria (1 iteración)

6.8. Métodos join-irreducibles() y meet-irreducibles()

Los métodos join-irreducibles() y meet-irreducibles() siguen el mismo camino del cálculo del ínfimo y el supremo, analizados en la sección anterior. Esto se debe a que el perfilado de estos dos métodos es muy similar a los anteriores, al igual que los cálculos que hacen internamente todos ellos, llamando a la función .subset(), quien representa el cuello de botella.

Con la entrada difusa cobre32, el método join-irreducibles() presenta una aceleración del 53,5 por ciento de media en las pruebas, mientras que meet-irreducibles() consigue optimizarse un 44 por ciento.

expression	user	time	elapsed
test1()	14.26	0.33s	14.63s
test2()	9.36	0.17s	9.53s

Cuadro 18: Resultado del benchmark de join-irreducibles() para la entrada cobre32 (1 iteración)

expression	user	time	elapsed
test1()	11.70s	0.19s	11.90s
test2()	8.10s	0.19s	8.30s

Cuadro 19: Resultado del benchmark de meet-irreducibles() para la entrada cobre32 (1 iteración)

Para entradas binarias y algo más grandes join-irreducibles() se acelera en un increíble factor de 1.79, casi un 80 por ciento. meet-irreducibles(), en su defecto, lo hace en un factor de 1.98, o lo que viene a ser lo mismo, tarda la mitad de tiempo en ejecutarse. Para los casos binarios las mejoras son del 82,4 por ciento y el 105 por ciento.

expression	user	time	elapsed
test1()	25.56s	0.08s	25.69s
test2()	14.15s	0.17s	14.35s
test3()	13.90s	0.09s	14.08s

Cuadro 20: Resultado del benchmark de join-irreducibles() para la entrada binaria aleatoria (1 iteración)

expression	user	time	elapsed
test1()	30.77s	0.10s	30.90s
test2()	15.45s	0.06s	15.53s
test3()	14.97s	0.06s	15.05s

Cuadro 21: Resultado del benchmark de meet-irreducibles() para la entrada binaria aleatoria (1 iteración)

6.9. Método standardize()

Como se ha indicado en el capítulo anterior, la aceleración de standardize() se ha realizado indirectamente por la optimización del algoritmo de inicialización de contextos formales, entre otros. Para el caso de una entrada difusa con cobre32, el método consigue aumentar en rendimiento un 22,6 por ciento. Sin embargo, para los casos binarios y más aproximados a la realidad, con un conjunto de entrada aleatorio de 500 objetos y 20 atributos, y una densidad de 0.4, la mejora es del 48,7 por ciento, y del 50,8 por ciento para la optimización enfocada a los conjuntos binarios.

expression	user	time	elapsed
test1()	17.64s	0.48s	18.22s
test2()	14.49s	0.35s	14.86s

Cuadro 22: Resultado del benchmark de standardize() para la entrada cobre32 (1 iteración)

expression	user	time	elapsed
test1()	49.69s	0.42s	50.27s
test2()	33.43s	0.36s	33.80s
test3()	32.86s	0.41s	33.32s

Cuadro 23: Resultado del benchmark de standardize() para la entrada binaria aleatoria (1 iteración)

6.10. Métodos sublattice(), subconcepts() y superconcepts()

Los casos de los métodos sublattice(), subconcepts() y superconcepts(), son muy parecidos, y su optimización pasa también por la aceleración en la ejecución de populateMatches(), ya que estas tres funciones hacen llamadas al algoritmo .subset(), quien representa en todos ellos el 60 por ciento del uso de recursos en la ejecución.

Comenzando con sublattice(), con la entrada difusa cobre32 la mejora representa un 27,7 por ciento de media. Sin embargo, esta aceleración se incrementa para un conjunto de prueba binario aleatorio más grande en la entrada, siendo la mejora del 84,4 por ciento, y de un 86,8 por ciento para la optimización binaria.

expression	user	time	elapsed
test1()	11.21s	0.03s	11.28s
test2()	8.75s	0.06s	8.83s

Cuadro 24: Resultado del benchmark de sublattice() para la entrada cobre32 (1 iteración)

expression	user	time	elapsed
test1()	29.11s	0.00s	29.14s
test2()	15.76s	0.01s	15.80s
test3()	15.60s	0.00s	15.60s

Cuadro 25: Resultado del benchmark de sublattice() para la entrada binaria aleatoria (1 iteración)

En el caso de subconcepts(), con la entrada cobre32 la mejora es del 36,21 por ciento. Para la entrada binaria, se consigue acelerar el método un 86,1 por ciento, y un 94,8 por ciento para la optimización del caso binario.

expression	user	time	elapsed
test1()	11.34s	0.01s	11.36s
test2()	8.33s	0.01s	8.34s

Cuadro 26: Resultado del benchmark de subconcepts() para la entrada cobre32 (1 iteración)

expression	user	time	elapsed
test1()	36.45s	0.09s	36.54s
test2()	19.54s	0.10s	19.64s
test3()	18.68s	0.07s	18.75s

Cuadro 27: Resultado del benchmark de subconcepts() para la entrada binaria aleatoria (1 iteración)

Por último, en la comparación de rendimiento para las diferentes implementaciones de superconcepts(), con la matriz dispersa de entrada se obtiene como resultado un aumento de velocidad del 34,8 por ciento. Por otro lado, para la entrada binaria esta mejora asciende al 80,6 por ciento, y al 86,5 por ciento respectivamente.

expression	user	time	elapsed
test1()	11.34s	0.01s	11.39s
test2()	8.39s	0.03s	8.45s

Cuadro 28: Resultado del benchmark de superconcepts() para la entrada cobre32 (1 iteración)

expression	user	time	elapsed
test1()	32.23s	0.08s	32.34s
test2()	17.89s	0.01s	17.91s
test3()	17.31s	0.02s	17.34s

Cuadro 29: Resultado del benchmark de superconcepts() para la entrada binaria aleatoria (1 iteración)

7

Conclusiones y Líneas Futuras

7.1. Dificultades y conclusiones finales

En el transcurrir de esta trabajo de fin de carrera han surgido numerosas dificultades, retrasando el proyecto en algún punto que, posteriormente, se ha podido recuperar.

En primera instancia, la información sobre técnicas para analizar y optimizar algoritmos escritos en R es relativamente escasa, debido a la naturaleza del lenguaje, enfocado a la interactividad y visualización de grandes cantidades de datos. Esto ha provocado que se hayan tenido que investigar numerosas fuentes, y aunarlas para recopilar la información necesaria para abordar el problema.

La primera complicación, y la más difícil de todas, surge con la necesidad de comprender y conocer con exactitud el paquete fcaR del lenguaje de programación R y todos los algoritmos y métodos de los que dispone internamente. Para abordar un problema de optimización, antes debe conocerse el algoritmo al milímetro, además de la complejidad computacional del mismo, para luego proponer ideas y mejoras.

Esta tarea ha resultado trabajosa debido a que el paquete ha sido creado por varios programadores, cada uno con un estilo de programación diferente, y que, además, se han usado algoritmos externos de otras fuentes, los cuales se han modificado para funcionar dentro de fcaR. Añadido a esto, la mayoría del código no dispone de comentarios explicativos sobre el funcionamiento de cada método, las entradas que recibe o las salidas.

La solución a este problema ha consistido en preguntar a alguno de los autores del paquete, que son los tutores de este proyecto, sobre el funcionamiento de los algoritmos complejos de entender.

Otra de las dificultades ha consistido en la integración de código escrito en dos lenguajes distintos, C++ y R. Para ello se ha utilizado el paquete Rcpp, quien facilita mucho el proceso, y cuya documentación si es accesible y relativamente sencilla de comprender.

En mitad del proyecto, otra situación dificultosa se presentó cuando se utilizaba la herramienta escogida para el perfilado de algoritmos, profvis. Esta herramienta no funcionaba correctamente cuando el código escrito en R estaba mezclado con código en C++ a través del paquete Rcpp, por lo que se tuvo que optar por emplear la librería jointprof para desempeñar esta función. Esto desencadenó otro problema, ya que esta librería solo funcionaría en los sistemas operativos Linux y MacOS.

Este problema se solucionó llevando el entorno de desarrollo a una máquina virtual con el sistema operativo Ubuntu instalado, el cual, era compatible con todas las herramientas que eran necesarias para la ejecución del proyecto.

Se considera que se ha aprendido muchísimo durante el desarrollo de este trabajo de fin de estudios. Durante el proceso se ha profundizado en lenguajes de programación como C++ y R, además de haber aprendido a manejar paquetes de estos lenguajes como Rcpp, profvis, knitr, bench o jointprof. También se ha utilizado la virtualización, una tecnología muy útil en muchos ámbitos, se ha usado una herramienta para el control de versiones como Git, y otras tecnologías y entornos como RStudio, Overleaf o LaTeX.

Finalmente, a pesar de todas las adversidades, se ha llegado a buen puerto y se considera que se han conseguido alcanzar todos los objetivos marcados en el anteproyecto. Se ha logrado mejorar la eficiencia y el consumo de recursos de los principales algoritmos de FCA implementados en el paquete fcaR, y para ello, se han investigado técnicas de depuración de

código, análisis y mejora de rendimiento de algoritmos en R, y estas han sido aplicadas al mismo, obteniendo resultados muy buenos.

7.2. Valoración general

La realización de este trabajo de fin de grado supuso un reto desde el principio. Esta línea de trabajo, inicialmente, se planteaba como un proceso arduo de investigación, autodidacta, y sin la existencia de alguna metodología aparente con la que abordar un problema de estas características.

Los profesores que tutorizan este proyecto, avisaron desde un comienzo que no iba a ser un camino fácil, y que se tendría que aportar mucho esfuerzo y motivación, ya que durante el transcurso del mismo se utilizarían numerosas tecnologías, herramientas y lenguajes de programación distintos.

Sin embargo, en las primeras semanas, los tutores guiaron el proyecto con varias lecturas en las que se abordaban problemas similares que, sinceramente, sirvieron de mucha ayuda para situarse y tener un punto de partida.

En lo que al transcurso del proyecto se refiere, han existido numerosas situaciones difíciles, como se ha comentado en la sección anterior que, de una forma u otra, se han superado, y han contribuido a que se incremente con creces el conocimiento en el campo en cuestión.

El proyecto ha obtenido resultados y ha conseguido los objetivos marcados inicialmente, pero lo más importante ha sido el camino de aprendizaje, que ha tenido como consecuencia lograr el objetivo final de optimizar los principales algoritmos del paquete fcaR de R.

Se cree que gracias a este proyecto, y al enfoque que se le ha dado, se han ampliado conocimientos vastamente en muchas tecnologías muy interesantes, y se anima a próximos estudiantes a seguir esta línea de trabajo, ampliándola en las ramas que en la siguiente sección se detallan.

7.3. Líneas Futuras

Aunque este proyecto se considere exitoso, no quiere decir que pueda considerarse cerrado. Conforme se han ido investigando y aplicando las técnicas de optimización han surgido nuevas oportunidades de mejora de algoritmos que, por problemas de tiempo y extensión, no han podido aplicarse en este trabajo y en esta memoria.

Esta línea de trabajo queda, pues, abierta para cualquier estudiante que desee ampliarla, proponiéndose aquí varios enfoques posibles que desarrollar y añadir, y enriquecer con ello también este proyecto.

Las técnicas que aquí se describen para la optimización de algoritmos no son, ni mucho menos, las únicas que existen, y es por ello, que pueden investigarse muchas más y agregarlas a la investigación.

Por otro lado, muchos algoritmos dentro del paquete fcaR se han quedado atrás y no han entrado dentro de este proyecto, el cual, se ha enfocado principalmente en los métodos de la clase FormalContext y ConceptLattice, y todos los algoritmos y funciones que cuelgan de ellos. Más específicamente, se ha centrado en los métodos de la principal viñeta del paquete fcaR relacionada con contextos formales y retículos de conceptos. (López y Ángel, 2018)

Finalmente, se anima al próximo estudiante que desee seguir esta línea de trabajo, a que considere la optimización de los algoritmos de las nuevas extensiones del paquete fcaR, tanto las ya existentes, que también se muestran en las viñetas, como a las extensiones que se añadirán pronto por parte de otros compañeros que también están añadiendo funcionalidades al paquete fcaR en sus respectivos trabajos de fin de grado.

Referencias

Chang, W. (2020). *Documentación del paquete profvis*. https://CRAN.R-project.org/package=profvis

CppReference. (2017). *Documentación de C++ en cppreference.com*. https://en.cppreference.com/w/cpp

cppreference.com. (s.f.). *Documentación del lenguaje C++ de cppreference.com*. https://devdocs.io/cpp/

Davies, A. (2018). *What is Agile Methodology?* https://www.devteam.space/blog/what-is-an-agile-methodology/

Documentación del lenguaje de programación R. (s.f.). https://cran.r-project.org/

Documentación extra del paquete profvis. (s.f.). http://rstudio.github.io/profvis/

Documentación oficial de git. (s.f.). https://git-scm.com/doc

Eddelbuettel, D. (2022). *Documentación del paquete Rcpp RDocumentation*. https://www.rdocumentation.org/packages/Rcpp/versions/1.0.9

Foundation, T. R. (s.f.). *Lenguaje de Programación R*. https://www.r-project.org/

Ganter, B. & Wille, R. (1999). *Formal concept analysis: mathematical foundations*. Springer.

Hester, J. (s.f.). *Documentación en viñetas del paquete bench*. https://bench.r-lib.org/index.html

Hester, J. (2021). *Documentación del paquete bench*. https://CRAN.R-project.org/package=bench

Limones, E. (2021). *Virtualización: Qué es, para qué sirve y ventajas*. https://openwebinars.net/blog/virtualizacion-que-es-para-que-sirve-y-ventajas/

López, D. & Ángel. (2018). *Documentación en viñetas del paquete fcaR*. https://malaga-fca-group.github.io/fcaR/

López, D. & Mora, Á. (2021). *Documentación del paquete fcaR*. https://CRAN.R-project.org/package=fcaR

López, D. & Mora, Á. (2022). *Documentación del paquete fcaR en pdf*. https://cran.r-project.org/web/packages/fcaR/fcaR.pdf

Muller, K. (2021). *Viñetas y documentación del paquete jointprof*. https://r-prof.github.io/jointprof/articles/jointprof.html

Oracle. (s.f.). *Web de virtualbox*. https://www.virtualbox.org/wiki/VirtualBox

Overleaf. (s.f.). *Web de Overleaf*. https://es.overleaf.com/learn

Project, C. (s.f.). *Documentación del paquete Rcpp CRAN*. https://CRAN.R-project.org/package=Rcpp

Rconsortium. (s.f.). *Repositorio github del paquete jointprof*. https://github.com/r-prof/jointprof

RStudio, P. (s.f.-a). *Documentación de RStudio*. https://docs.rstudio.com/

RStudio, P. (s.f.-b). *Web de RStudio*. https://www.rstudio.com/

Škopljanac-Mačina, F. & Blašković, B. (2014). Formal concept analysis–overview and applications. *Procedia Engineering*, *69*, 1258-1267.

Wickham, H. (2019). *Advanced r*. CRC press.

Glosario

Agile Metodología que permite adaptar la forma de trabajo a las condiciones del proyecto.. 35

APA American Psychological Association, organización científica y profesional de psicólogos estadounidenses.. 17, 23

API Application Programming Interfaces.. 26, 43

benchmark Prueba de rendimiento.. 25

booleanos El tipo de dato lógico, TRUE o FALSE.. 44

bucle Es una secuencia de instrucciones de código que se ejecuta repetidas veces.. 27, 43, 44, 53, 56

C++ Lenguaje de programación basado en C.. 2, 3, 5, 16, 23, 24, 26, 27, 36, 40, 41, 43–46, 48, 51, 53, 55, 56, 58, 65

conjuntos difusos Conjunto que permite valores intermedios de pertenencia.. 20, 29

CRAN Comprehensive R Archive Network, repositorio de R con más de 10.000 paquetes.. 23, 25, 27, 29

cuellos de botella Es una etapa de trabajo que recibe más solicitudes de las que puede procesar a su máxima capacidad.. 3, 15, 20, 25, 27, 40, 41, 44, 46, 52

código fuente Conjunto de líneas de texto con los pasos que debe seguir la computadora para ejecutar un cargador.. 19, 22, 39

depuración Quitar errores del código.. 16, 36, 39, 40

extent La extensión de conjuntos.. 51, 53, 69, 71

FCA Formal Concept Analysis.. 15–17, 19, 20, 29

fcaR Paquete del lenguaje de programación R basado en FCA.. 2, 3, 15–17, 20, 22, 24, 29, 31, 36, 37, 46, 48, 50, 55, 63, 69

Git Sistema de Control de Versiones.. 22, 36

hipervisor Proceso que crea y ejecuta máquinas virtuales.. 21

IDE Entorno de Desarrollo Integrado.. 21, 22

intent La intensión de conjuntos.. 50, 68, 69

iteraciones Repetir varias veces un proceso con la intención de alcanzar una meta.. 26, 27, 35, 63, 67, 68, 70

Lambda funciones Subrutina definida que no está enlazada a un identificador.. 24

LaTeX Sistema de composición de textos de alta calidad tipográfica, usado principalmente en la generación de artículos y libros científicos.. 5, 23, 46

matriz Es un conjunto bidimensional de números.. 19, 20, 29, 48, 49, 66

metodología Conjunto de métodos que se siguen en una investigación científica.. 17, 22, 35, 36

NA NA significa not available, o que ciertos datos no están disponibles.. 43, 44

Oracle Compañía especializada en el desarrollo de soluciones de nube y locales.. 20

Overleaf Editor colaborativo de LaTeX basado en la nube.. 23

paradigma Teoría o conjunto de teorías cuyo núcleo central se acepta sin cuestionar y que suministra la base y modelo para resolver problemas y avanzar en el conocimiento.. 15, 24

perfilado Recopilación de las características del programa durante su ejecución.. 20, 25, 36, 41, 46, 52, 53, 55, 60, 61

pila de llamadas Estructura de datos que almacena la información sobre las subrutinas activas de un programa de computadora.. 25, 26, 40, 41, 61, 71

R Lenguaje de programación con enfoque estadístico.. 2, 3, 15, 16, 20, 22–27, 29, 36, 39–44, 46

recolector de basura Es un mecanismo implícito de gestión de memoria.. 41, 71

repositorio Espacio centralizado donde se almacena, organiza, mantiene y difunde información digital.. 22, 47, 63

retículo Tejido en forma de red.. 15, 19, 31, 54, 56–58, 61

Rprof Función definida en el paquete básico de R para el perfilado de código.. 25

RStudio Entorno de desarrollo para el lenguaje de programación R.. 16, 21, 22, 36, 39–41

sistemas operativos Software que coordina y dirige todos los servicios y aplicaciones que utiliza el usuario en una computadora.. 21, 41, 55

sprints Periodo de tiempo en el que se debe entregar un incremento de valor de un producto.. 36, 37

tibble Es un tipo de DataFrame.. 45

Ubuntu Sistema operativo basado en Linux.. 9, 20, 21, 41, 55

VirtualBox Herramienta de virtualización de Oracle.. 20

Apéndice A
Manual de Instalación

En este apéndice se adjunta un manual de instalación, en el que se explica detalladamente cómo se prepara el entorno de desarrollo compatible con todas las herramientas necesarias para aplicar y probar todas las técnicas de análisis de rendimiento y benchmarking aplicadas en este proyecto.

El proceso pasa por la descarga de una herramienta para la virtualización, y la instalación de un sistema operativo basado en Linux para asegurar la compatibilidad de todos los paquetes. Dentro de esta máquina virtual que se crea, se comienza a instalar el IDE, el cual, se vincula a un repositorio Git. Luego se instalan los paquetes utilizados en este trabajo de fin de grado y todas sus dependencias, para finalmente explicar el uso del perfilador y la herramienta de comparación de rendimiento de algoritmos.

A.1. Virtualización y VirtualBox

Como herramienta de virtualización se ha usado Oracle VM VirtualBox para windows en la versión 6.1.38: Oracle VM VirtualBox

El sistema operativo instalado es Ubuntu en su versión 22.04.1 LTS. Se descarga la imagen .iso que pesa 3,6GB: SO Ubuntu

Una vez se hayan descargado ambos, e instalado VirtualBox, se abre la herramienta. En su interfaz, se debe pulsar en la opción **Nueva** para la creación de la máquina virtual, a la que se deberá dar un nombre, y elegir el sistema operativo que se desee instalar, el mismo que la imagen que se ha descargado anteriormente.

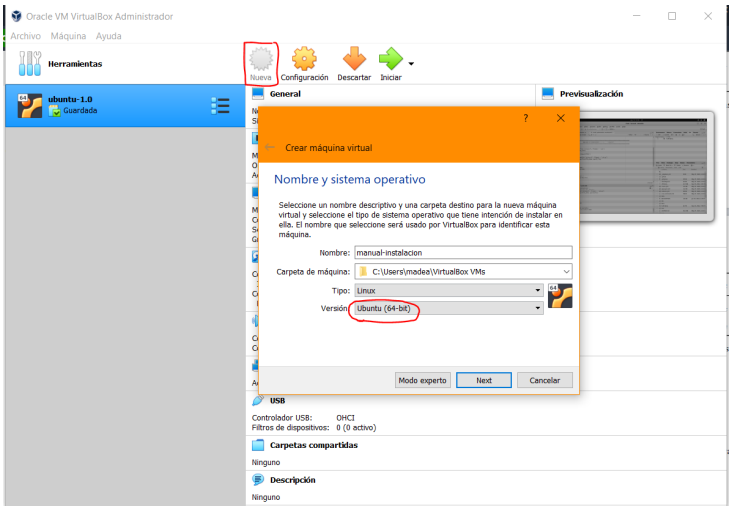

Figura 21: Creación de la máquina virtual con Ubuntu.

Se selecciona el tamaño de memoria RAM pertinente, siendo el mínimo recomendado 1 GB. En este caso se ha optado por 2 GB de memoria debido a que, en ocasiones, se van a realizar ingentes cantidades de operaciones que requerirán de ella.

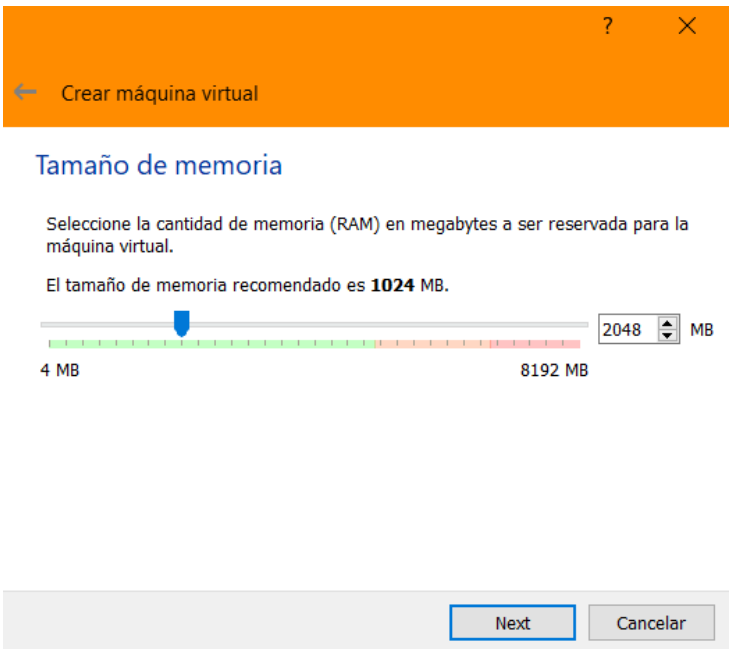

Figura 22: Cantidad de memoria RAM asignada a la máquina virtual.

Para el disco duro de la máquina, el tamaño mínimo recomendable es de 10 GB, pero se han asignado 20 GB para que haya margen para la instalación del entorno de desarrollo y todos los paquetes. El tipo de archivo de disco duro se seleccionará VDI (VirtualBox Disk Image).

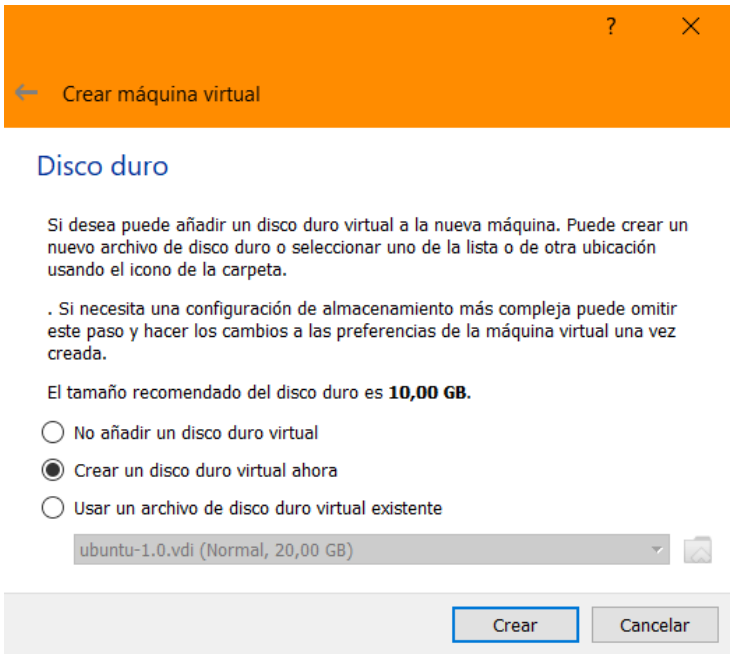

Figura 23: Tamaño del disco duro de la máquina virtual.

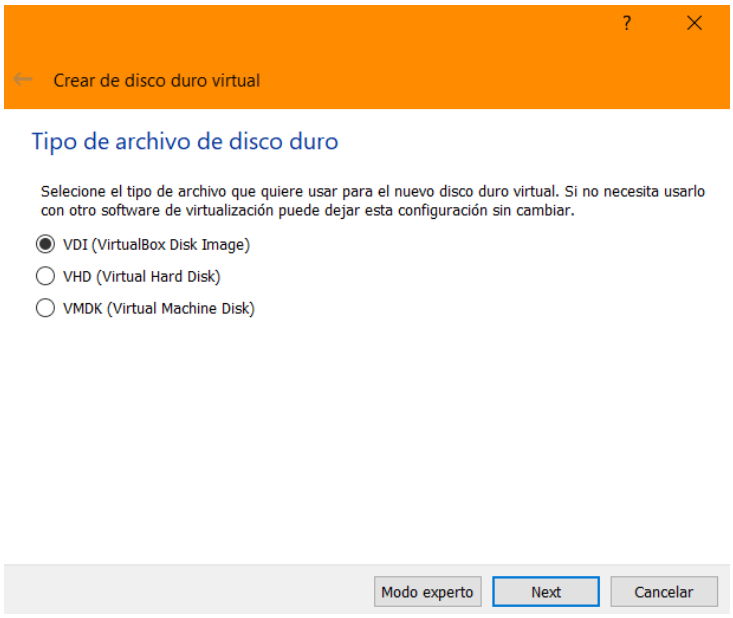

Figura 24: Tipo de archivo de disco duro de la máquina virtual.

Finalmente, se termina de instanciar la configuración de la máquina virtual, quedando como se muestra en la siguiente figura.

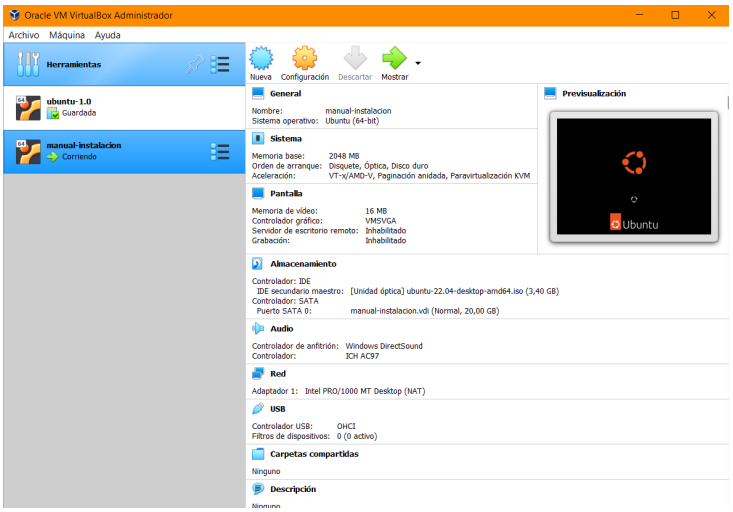

Figura 25: Configuración de la máquina virtual.

Dentro de esa máquina, se debe instalar el sistema operativo a partir de la imagen .iso previamente descargada. Cuando se inicia la máquina virtual por primera vez, aparece una pestaña que requiere la selección de dicha imagen, la cual detecta automáticamento en el equipo.

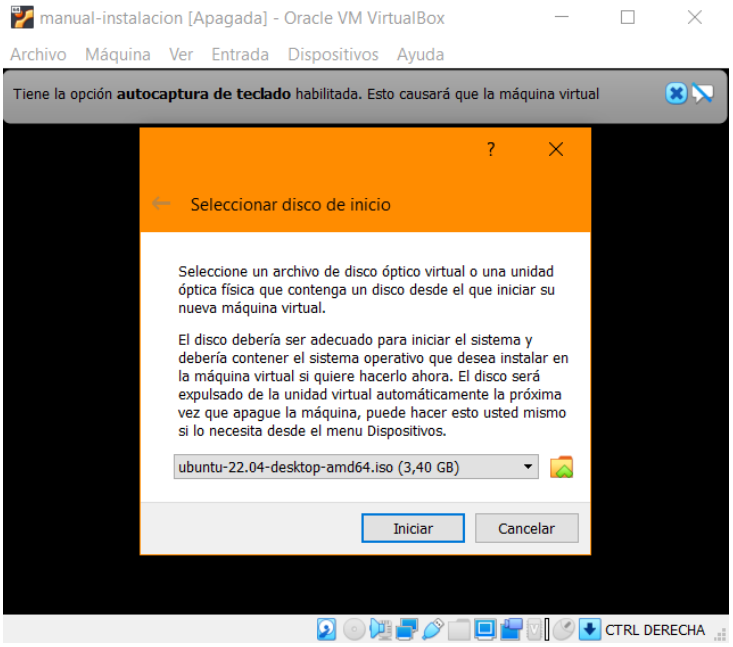

Figura 26: Selección de la imágen .iso con el sistema operativo.

Por último, se realiza la instalación del sistema operativo siguiendo los paneles de instalación y dejando todas las opciones que se ofrecen por defecto, seleccionando el idioma deseado, y creando un usuario en la máquina. Una vez terminado el proceso de instalación del SO, la máquina requerirá un reinicio.

Figura 27: Proceso de instalación del sistema operativo Ubuntu.

A.2. Instalación y configuración de RStudio y Git

Previo a la instalación del IDE RStudio, es requerida la instalación de R en la máquina. Esto se hace a través de los siguientes comandos en el terminal de Linux, según la web oficial:

```
# update indices
sudo apt update -qq
# install two helper packages we need
sudo apt install --no-install-recommends software-properties-
    common dirmngr
# add the signing key (by Michael Rutter) for these repos
# To verify key, run gpg --show-keys /etc/apt/trusted.gpg.d/
    cran_ubuntu_key.asc
# Fingerprint: E298A3A825C0D65DFD57CBB651716619E084DAB9
```

```
wget -qO- https://cloud.r-project.org/bin/linux/ubuntu/marutter_pubkey.asc | sudo tee -a /etc/apt/trusted.gpg.d/cran_ubuntu_key.asc
# add the R 4.0 repo from CRAN -- adjust 'focal' to 'groovy' or 'bionic' as needed
sudo add-apt-repository "deb https://cloud.r-project.org/bin/linux/ubuntu $(lsb_release -cs)-cran40/"

# Then run this to install R and its dependencies
sudo apt install --no-install-recommends r-base
```

Código.- 7: Instalación de R en la máquina virtual.

Luego hay que dirigirse a la página oficial de descargas de RStudio y elegir la versión que se adapta al sistema operativo Ubuntu 22: Descargar RStudio

Seguidamente, se instala Git a través de estos comandos:

```
sudo apt-get install git
sudo add-apt-repository ppa:git-core/ppa
sudo apt update; sudo apt install git
```

Código.- 8: Instalación de Git en la máquina virtual

Una vez abierto RStudio, creamos un nuevo proyecto con la opción de **Control de Versiones**, seleccionando Git y añadiendo la URL del repositorio sobre el que se quiere trabajar. Con todo esto, ya se está trabajando sobre el repositorio deseado al cambiar a la rama del mismo que interese.

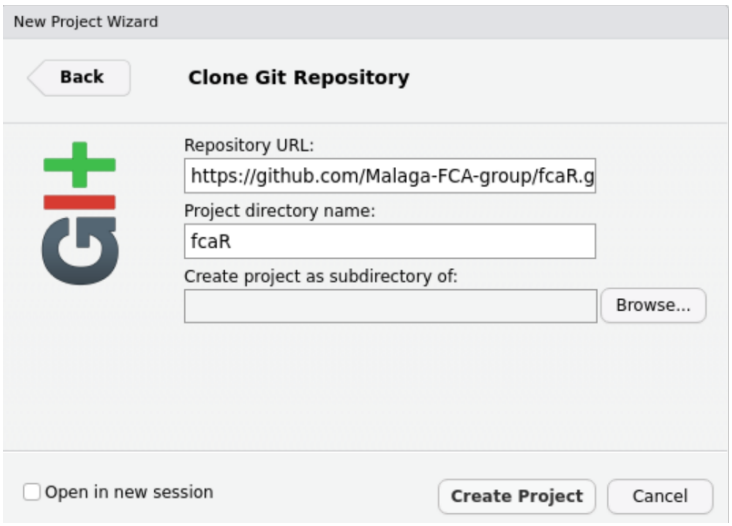

Figura 28: Creación de un nuevo proyecto conectado al repositorio de Git.

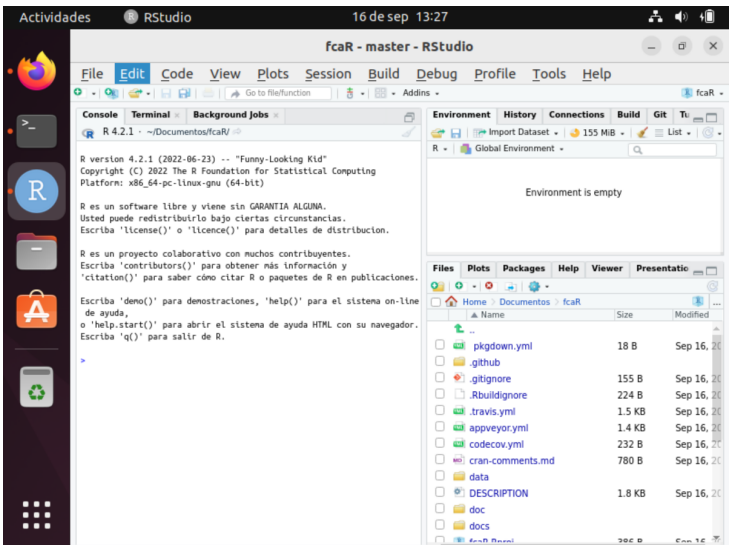

Figura 29: Interfaz gráfica de la herramienta RStudio ya configurada.

Ya en este punto se instalan paquetes necesarios como arules, tidyverse, knitr, fcaR, profvis y Rcpp con todas sus dependencias.

```
install.packages("tidyverse")
install.packages("arules")
install.packages("knitr")
install.packages("fcaR")
```

```
install.packages("profvis")
install.packages("Rcpp")
```

Código.- 9: Instalación de los diferentes paquetes.

A.3. Instalación y uso de jointprof

Para la instalación del paquete hay que dirigirse a la página web con las viñetas, en las que se muestra como es la instalación para los diferentes sistemas operativos, así como tutoriales para aprender a usarlo. Viñetas del paquete jointprof para la instalación y uso.

Para perfilar con el paquete jointprof, primero se crea un fichero donde se almacenará la salida de la función del perfilado en sí. Después se ejecuta la función start-profiler() pasándo como parámetro el fichero generado anteriormente. Esta función se ejecutará de manera ininterrumpida hasta que se pare con la función stop-profiler(), quien devuelve los datos del perfilado, por lo que hay que guardarlos en una variable. En medio de estas dos funciones comentadas se inserta el código que se quiera perfilar.

Posteriormente, los datos que se obtuvieron como resultado se publican en http://localhost:8080, hasta que se decida cortar la ejecución desde RStudio.

```
out_file <- tempfile("jointprof", fileext = ".out")
start_profiler(out_file)
test1()
profile_data <- stop_profiler()


pprof_file <- tempfile("jointprof", fileext = ".pb.gz")
profile::write_pprof(profile_data, pprof_file)
system2(
  find_pprof(),
  c(
```

```
    "-http",
    "localhost:8080",
    shQuote(pprof_file)
  )
)
```

Código.- 10: Uso del paquete jointprof para el perfilado de código.

Si se entra en la interfaz gráfica publicada en el localhost de la web, se obtiene el perfilado del código en cuestión. Este puede visualizarse interactivamente en diferentes formatos, gráficas, grafos, tablas, etc. En las vistas se indica el porcentaje de uso de recursos y alguna información más para identificar posibles cuellos de botella.

Figura 30: Visualizador de los datos obtenidos del perfilado en gráfica.

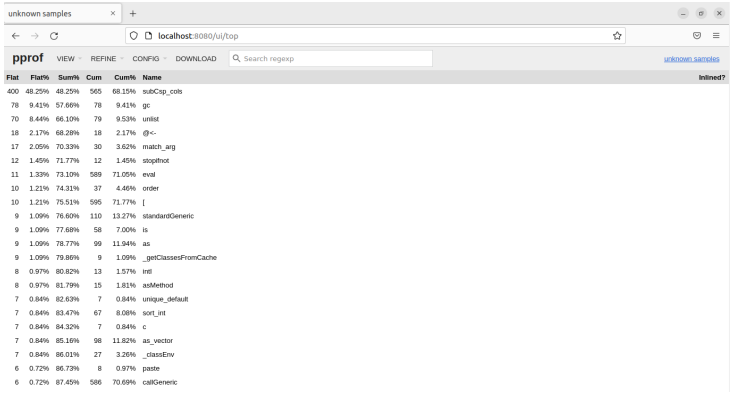

Figura 31: Visualizador de los datos obtenidos del perfilado en tabla.

A.4. Instalación y uso de bench

La instalación del paquete de R bench pasa por la ejecución del mismo comando utilizado para la instalación de cualquier paquete:

```
install.packages("bench")
```

Código.- 11: Instalación del paquete bench en la máquina virtual.

Para comparar dos implementaciones de dos algoritmos equivalentes se hace uso de la función mark() del paquete bench, es decir, bench::mark(). Como parámetro a esta función, se pasan todos los algoritmos que se quieran comparar separados por comas.

Además, se le pasan otros parámetros como el número de ejecuciones de cada algoritmo, para luego calcular la media de tiempo que emplean cada uno de ellos (iterations = <número de iteraciones>), o como check = <TRUE o FALSE>, dependiendo de si se quiere hacer una comprobación de que ambos algoritmos tienen la misma salida.

Finalmente, para exportar los datos, la salida de bench::mark() se almacena en una variable, que se puede pasar al formato LaTeX con el paquete knitr el método kable().

```
dual_results <- bench::mark(
  test2(),
  test3(),
  iterations = 5,
  check = FALSE
)[c("expression", "min", "median", "itr/sec", "n_gc", "total_
   time", "mem_alloc")]

dual_results
dual_results %>% kable(format = 'latex', booktabs = TRUE)
```

Código.- 12: Paquete bench para comparación de rendimiento y exportación del tibble a LaTeX.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA