ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

Estudio de operadores discretos definidos sobre cadenas finitas

Study of discrete operators on finite chains

Realizado por
Franco Manuel García Dos Santos

Tutorizado por
Carlos Bejines López, Manuel Ojeda Hernández

Departamento
Matemática Aplicada

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2023

Resumen

En la lógica clásica, los predicados son binarios y toman valores de verdad de 0 o 1, que se corresponden con falso y verdadero, respectivamente. Sin embargo, en la mayoría de las veces, el razonamiento humano y muchos de los problemas en el mundo real donde suele tener su ámbito de aplicación no son deterministas y no pueden ser tratados con modelos probabilísticos. En estos casos, se necesita el uso de herramientas matemáticas capaces de manejar información no binaria. Precisamente, la Lógica Difusa es una disciplina que nace para recoger y trabajar con entornos que contienen incertidumbre, vaguedad, información imprecisa u incompleta. La Lógica Difusa está constituida y actúa con el rigor inherente de las matemáticas en entornos difusos.

En este Trabajo de Fin de Grado consideramos los operadores definidos sobre una escala discreta, es decir, una cadena finita de valores. En concreto, trabajaremos sobre los operadores que modelan la conjunción lógica en la lógica difusa.

El objetivo principal de este trabajo es el estudio de todos los operadores discretos definidos en una cadena finita y del cardinal del conjunto de dichos operadores mediante el desarrollo de una aplicación web interactiva con Shiny directamente desde R.

En la memoria se detallarán los conceptos necesarios que se han utilizado a lo largo del trabajo, así como las distintas vistas en la interfaz de usuario y el pseudocódigo de los algoritmos implementados. También se proporcionará una copia del código para su uso propio del interesado, así como un manual de instalación del software.

Palabras clave: **Lógica difusa, Operador discreto, Cardinal de conjuntos, R, Shiny.**

Abstract

In classical logic, predicates are binary and take truth values of 0 or 1, corresponding to true and false, respectively. However, most of the time, human reasoning and many of the real-world problems in which it is usually applied are non-deterministic and cannot be treated with probabilistic models. In these cases, the use of mathematical tools capable of handling non-binary information is needed. Fuzzy Logic is a discipline that was born to collect and work with environments that contain uncertainty, vagueness, imprecise or incomplete information. Fuzzy Logic is constituted and acts with the inherent rigour of mathematics in fuzzy environments.

In this End of Degree Project we consider operators defined on a discrete scale, i.e. a finite chain of values. In particular, we will work on the operators that model the logical conjunction in multivalued logic.

The main objective of this project is the study of all discrete operators defined on a finite chain and the cardinality of the set of such operators by developing an interactive web application with Shiny directly from R.

The memoir will detail the necessary concepts used throughout the work, as well as the different views in the user interface and the pseudocode of the implemented algorithms. A copy of the code will also be provided for own use of the person interested, as well as a software installation manual.

Keywords: Fuzzy logic, Discrete operator, Cardinal of sets, R, Shiny.

Índice

1. INTRODUCCIÓN **1**
1.1. Estado del arte 1
1.2. Objetivos 2
1.3. Metodología de trabajo 3
1.4. Tecnologías utilizadas 4
1.5. Estructura de la memoria 5
2. CONCEPTOS TEÓRICOS **7**
3. SHINY **10**
3.1. ¿Qué es Shiny? 10
3.2. Reactividad en Shiny 10
3.3. Funciones principales de Shiny 12
3.3.1. Funciones para la UI 12
3.3.2. Funciones para el servidor 16
3.3.3. Otras funciones 18
3.3. Módulos en Shiny 18
3.4. Paquetes utilizadas en Shiny 19
4. DESCRIPCIÓN DEL FUNCIONAMIENTO DE LA APLICACIÓN SHINY **23**
4.1. Análisis del operador 23
4.1.1. Matriz manual 24
4.1.2. Matriz por archivo 25
4.1.3. Matriz por fórmula 26
4.2. Calcular 28
4.3. Comparar dominancia 29
4.4. Filtrar matrices 31
5. ALGORITMOS PARA CALCULAR CARDINAL DE OPERADORES BINARIOS **33**
6. CONCLUSIONES Y LÍNEAS FUTURAS **59**
BIBLIOGRAFÍA **61**
APÉNDICE A. MANUAL DE INSTALACIÓN **65**
APÉNDICE B. FUNCIONES MÁS RELEVANTES DE PAQUETES QUE MEJORAN SHINY. **69**

1. Introducción

1.1. Estado del arte

En la Lógica Clásica, los predicados admiten dos valores de verdad: verdadero o falso, identificados con el valor 1 y 0, respectivamente. Sin embargo, en muchas situaciones, el razonamiento humano y el entorno en el que se aplican no son deterministas ni se pueden modelar de manera probabilística. En estos casos, se necesitan herramientas matemáticas capaces de manejar información más allá de lo binario. Precisamente, la Lógica Difusa (*Fuzzy Logic*) nace para recoger y trabajar con entornos que presentan incertidumbre, vaguedad, información imprecisa y, en ocasiones, incompleta. La Lógica Difusa está constituida y utiliza el rigor de las matemáticas para abordar estos entornos difusos.

Dado un conjunto X, un subconjunto $Y \subseteq X$ puede ser representado como una función de X a {0, 1}, donde se asigna el valor 1 a los elementos que pertenecen a Y y 0 a los que no. Esta aplicación se conoce como la función característica del subconjunto Y. Por otra parte, un subconjunto difuso de X se define como una función $X \longrightarrow [0, 1]$, donde el valor asignado a cada elemento del conjunto de referencia es el grado de pertenencia de dicho elemento. Esta noción amplía el concepto clásico de subconjunto y fue introducida por L. Zadeh en 1965 (ver [1]) quién inició la teoría de conjuntos difusos, que es el marco matemático en el que se basa la Lógica Difusa.

En determinadas situaciones prácticas, es común trabajar con un rango de valores finito, y en ocasiones es preciso hacerlo con elementos no comparables. Por ello, aunque inicialmente el codominio de los subconjuntos difusos fue el intervalo unidad, se han estudiado otros codominios, como los conjuntos parcialmente ordenados y, más específicamente, los retículos finitos.

El retículo finito más básico es la cadena discreta. El estudio de los operadores puede reducirse al estudio de operadores definidos sobre la cadena finita $C_n = \{0, 1, ..., n-1\}$ puesto que toda cadena finita con n elementos es isomorfa a C_n. En el caso de una cadena finita, el número de operadores que se pueden construir también es finito. Bajo determinadas

circunstancias, es posible determinar el número de dichos operadores. Este ha sido el punto de partida de varias líneas de investigación.

En 1999 (ver [2]), De Baets y Mesiar obtuvieron el número de t-normas definidas en cadenas discretas de hasta catorce elementos a través de métodos computacionales. Esto debía acercarnos a la obtención de una fórmula general de la sucesión de t-normas sobre cadenas discretas, pero no ha sido así. A día de hoy, la obtención de esta sucesión sigue siendo un problema abierto.

Por otro lado, la investigación en otros tipos de operadores discretos ha avanzado en los últimos años. Más concretamente, la obtención de fórmulas explícitas para cierto tipo de operadores es un campo de estudio activo en la matemática moderna. Son ejemplos: las t-normas arquimedianas maximales (ver [3]), los conjuntores conmutativos (ver [4]) o las semicópulas (ver [5]).

A partir de lo anterior, se deduce de manera evidente que el problema de determinar la cardinalidad de ciertos operadores discretos ha sido tratada en diversos estudios. Sin embargo, existen otras familias de operadores cuya cardinalidad o construcción no se conoce con certeza en la actualidad. De alguna forma, la visualización de los distintos operadores discretos puede ayudar a comprender qué comportamiento tienen y acelerar la obtención de una fórmula general.

Por tanto, la problemática que se quiere resolver es la de encontrar una forma eficiente de calcular operadores discretos dentro del ámbito de la lógica difusa. En concreto, vamos a trabajar sobre los operadores que modelan la conjunción lógica en la lógica difusa.

1.2. Objetivos

El objetivo principal de este Trabajo Fin de Grado es el estudio de todos los operadores binarios definidos sobre una escala discreta, es decir, una cadena finita de valores, y el cálculo del cardinal de cualquier conjunto de dichos operadores bajo ciertas propiedades.

Para llevar a cabo este objetivo principal mediante el desarrollo de una aplicación web interactiva con Shiny, se han realizado las siguientes tareas:

1. Desarrollo de interfaces y programas que permitan, a partir de cualquier operador discreto definido en una cadena finita, ser capaz de analizar y clasificar dicho operador en función de las propiedades que cumpla. Se implementan tres formas diferentes de introducir dicho operador.
2. Desarrollo de una interfaz y una serie de algoritmos que sean capaces de determinar el cardinal de conjuntos de operadores a partir de una serie de propiedades seleccionadas, así como obtener un número determinado de dichos operadores para su posterior descarga y visualización.
3. Desarrollo de una interfaz y programas que permita comparar la dominancia de dos maneras; entre dos operadores dados o, de la misma forma, entre un operador 'A' y una lista de operadores dada. La interfaz permite seleccionar tanto los operadores que dominan a A como los que son dominados por A y la descarga de los resultados en el segundo caso.
4. Desarrollo de una interfaz y programas que permitan filtrar una lista de matrices a partir de ciertas propiedades. De igual forma, la interfaz permite la descarga de los resultados.

1.3. Metodología de trabajo

En primer lugar, se habló del contexto general del trabajo, el objetivo principal a lograr, y se debatió sobre el software a utilizar para el desarrollo del trabajo. Dado que R, uno de los lenguajes de programación más utilizados en investigación científica, ofrece una variedad extensa de bibliotecas y paquetes, entre ellos Shiny, que facilita la creación de aplicaciones web interactivas sin ser versados en diseño web, se decidió escoger R como lenguaje de programación y se definieron los objetivos generales y funcionalidades a alto nivel de la aplicación.

Debido al alto grado de flexibilidad y en los que se esperan resultados intermedios a corto plazo, la metodología utilizada en el desarrollo del TFG ha sido la metodología Scrum, ya que resulta ser la más adecuada para proyectos de esta complejidad.

Scrum se define como un proceso para desarrollo ágil de software que se basa en aplicar de manera regular buenas prácticas para trabajar colaborativamente y realizar entregas parciales y regulares de manera eficiente y efectiva en entornos donde se necesitan resultados rápidos,

donde la flexibilidad y la adaptabilidad son fundamentales. Se trabaja en ciclos cortos denominados 'sprints' de duración variable entre una y cuatro semanas en el que al final cada sprint se tiene que proporcionar un resultado concreto y completo, donde se revisa el progreso y se planifica el trabajo futuro (ver [6]).

En el caso de este TFG, las reuniones con los tutores se han llevado a cabo a través de sprints de dos a tres semanas en el que se comunican los avances conseguidos y los problemas encontrados si los hubiera, planificando los objetivos futuros con la retroalimentación de los tutores para la siguiente reunión.

1.4. Tecnologías utilizadas

A continuación se enumeran las tecnologías empleadas en el desarrollo del TFG:

1. **R.** Es un lenguaje de programación de software libre y de código abierto que se utiliza ampliamente en el ámbito del análisis estadístico, la visualización de datos y la investigación matemática. Su sintaxis es sencilla e intuitiva, lo que permite a los investigadores escribir código de manera eficiente y rápida. R también tiene la posibilidad de cargar y utilizar una gran cantidad de paquetes disponibles con diversas funcionalidades y que se enfocan en diferentes áreas de estudio, como estadísticas, manipulación y visualización de datos, minería de datos, machine learning, optimización, entre otros. Además, permite la integración con otros lenguajes de programación, como Python y C++, lo que aumenta su capacidad de uso (ver [7]).
2. **Git.** Es un software de control de versiones libre y de código abierto que permite la rápida gestión del mantenimiento de diferentes versiones en el código fuente. Al mismo tiempo, Git almacena el historial completo del proyecto, lo que permite rastrear cambios en el código fuente y regresar a una versión anterior del código si es necesario. Otra ventaja es que, Git ofrece la posibilidad de crear ramas, que son versiones alternativas del proyecto que se pueden utilizar para experimentar nuevas características o realizar correcciones de errores de forma simultánea sin afectar la

rama principal, lo que aumenta la eficiencia y la seguridad en el desarrollo de software (ver [8]). El proyecto se encuentra alojado en Github[1].

3. **RStudio.** Es un entorno de desarrollo integrado de código abierto especialmente diseñado para trabajar con el lenguaje de programación R. Incluye un editor de código, una consola interactiva y herramientas de gráficos que facilitan la escritura, depuración[2] y ejecución de código, así como la visualización y manipulación de datos. Además, incorpora la capacidad de instalar y administrar paquetes de R, un navegador de archivos y directorios y herramientas para la importación y exportación de datos. Otras características destacables son la integración para el control de versiones con Git y Github, y la compatibilidad con RMarkdown[3] y LaTeX[4] para la generación de informes (ver [9]).

1.5. Estructura de la memoria

En esta sección se enumeran los capítulos en los que está dividido el presente documento, acompañado de una breve descripción de en qué consiste cada capítulo.

1. **Capítulo 1. Introducción:** En este capítulo se ofrece la contextualización y motivación del proyecto, así como el objetivo principal a conseguir, la metodología de trabajo seguida y las tecnologías empleadas.
2. **Capítulo 2. Conceptos teóricos:** En este capítulo se presentan una serie de fundamentos teóricos esenciales para comprender el desarrollo del proyecto.
3. **Capítulo 3. Shiny:** En este capítulo se abordan diferentes conceptos de Shiny y una serie de extensiones empleadas en el desarrollo de la aplicación.
4. **Capítulo 4. Descripción del funcionamiento de la aplicación Shiny:** En este capítulo se describe la fase de implementación del proyecto, especificando cada una de las

[1] Plataforma de alojamiento de proyectos utilizando Git.

[2] Proceso de detección y corrección de fallos en el código fuente de cualquier software.

[3] Lenguaje de marcado que combina texto normal con elementos de formato y código ejecutable en R.

[4] Sistema de composición tipográfica utilizado para crear documentos de alta calidad, especialmente para expresiones matemáticas.

funcionalidades desarrolladas en la aplicación y mostrando las diferentes interfaces de usuario.

5. **Capítulo 5. Algoritmos para calcular cardinal de operadores binarios:** En este capítulo se muestran una serie de algoritmos empleados para la generación y el cálculo del cardinal de conjuntos de operadores binarios bajo ciertas propiedades.
6. **Capítulo 6. Conclusiones y líneas futuras:** En este capítulo se presentan las conclusiones alcanzadas tras llevar a cabo el proyecto, así como se plantean posibles mejoras y ampliaciones futuras del proyecto.
7. **Apéndice A. Manual de Instalación:** En este apéndice se proporciona una serie de instrucciones detalladas que permiten al usuario instalar R, RStudio y los paquetes necesarios para poder trabajar con la aplicación software.
8. **Apéndice B. Funciones más relevantes de paquetes que mejoran Shiny:** En este apéndice se mencionan las funciones más importantes de los paquetes que extienden la funcionalidad de Shiny.

2. Conceptos teóricos

Un operador binario **S** definido sobre una cadena finita de longitud n, $\mathbf{C_n} = \{x_1, x_2, \dots, x_{n-1}, x_n\}$, es una operación que a cada par de valores de la cadena definida le corresponde un tercer valor en la misma cadena, es decir:

$$S : C_n \times C_n \longrightarrow C_n$$

Una forma común de representar un operador binario es mediante su matriz interior. La matriz interior **M** de un operador binario es una matriz cuadrada de tamaño nxn donde se asigna a cada valor de la cadena un índice, en el que cada celda representa el resultado de la operación en función de los índices asignados a los valores de entrada, es decir:

$$M[i, j] = S(x_i, x_j) \quad \forall i, j \in \{1, \dots, n-1, n\}$$

A continuación se enumeran las propiedades de interés utilizadas a lo largo del trabajo:

1. **Elemento neutro:** Un operador S se dice que tiene elemento neutro si existe un elemento e de la cadena, tal que para cualquier otro elemento x de la cadena, se cumple que
$$S(e, x) = S(x, e) = x.$$
2. **Monotonía creciente:** Un operador S se dice monótona creciente si para cada n1, n2, m en la cadena tal que $n1 \leq n2$, se cumple que
    1. $S(n1, m) \leq S(n2, m)$.
    2. $S(m, n1) \leq S(m, n2)$.
3. **Conmutatividad:** Un operador S cumple la conmutatividad si para cada x, y en la cadena se verifica que
$$S(x, y) = S(y, x).$$
4. **Asociatividad:** Un operador S cumple la asociatividad si para cada x, y, z en la cadena se verifica que
$$S(x, S(y, z)) = S(S(x, y), z).$$

5. **Propiedad arquimediana:** Un operador S es arquimediano si sus únicos elementos idempotentes son el primero y el último. Un elemento x en la cadena se dice idempotente si cumple que

$$S(x, x) = x.$$

6. **Divisibilidad:** Un operador S se dice que es divisible si para cada x, y en la cadena tal que $x \leq y$, existe otro elemento z tal que

$$S(y, z) = x.$$

7. **Maximalidad:** Dado un conjunto de operadores S definido en una cadena finita, decimos que un operador A_1 es mayor que A_2 si para todo x, y en la cadena, se cumple que

$$A_1(x, y) \geq A_2(x, y).$$

   Cuando no existe en S ningún operador mayor que A_1, entonces decimos que dicho operador es maximal en el conjunto S. Cabe mencionar que existen casos en los que dos operadores no son comparables. Esto significa que en algunos casos se cumple que $A_1(x, y) > A_2(x, y)$ y en otros casos, al revés.

8. **Dominancia:** Dados dos operadores S_1 y S_2, decimos que S_1 domina a S_2 ($A >> B$) si para todo x, y, z, t en la cadena, se cumple que

$$S_1(S_2(x,z), S_2(y,t)) \geq S_2(S_1(x,y), S_1(z,t)).$$

A partir de las cuatro primeras propiedades, se puede definir una clasificación de los operadores como sigue:

1. Los operadores que satisfacen la monotonía creciente se les denominan conjuntores.
2. Si un conjuntor tiene como elemento neutro el último elemento de la cadena, se le denomina una semicópula.
3. Si un operador tiene elemento neutro y satisface la monotonía creciente, la conmutatividad y la asociatividad, existen tres posibles casos:
   - **3.1.** Si el elemento neutro es el último elemento de la cadena, se le denomina una norma triangular, o t-norma.
   - **3.2.** Si el elemento neutro es el primer elemento de la cadena, se le denomina una conorma triangular, o t-conorma.
   - **3.3.** En otro caso, se le denomina una uninorma triangular, o simplemente uninorma.

Los tres operadores discretos básicos (son, de hecho, también las t-normas discretas básicas) sobre una cadena finita $C_n = \{0, \ldots, n\text{-}1\}$ son los siguientes:

1. **1.** La t-norma drástica $D : C_n \times C_n \longrightarrow C_n$ es la t-norma más pequeña y se define por

$$D(x, y) = \begin{cases} \min(x, y) & \text{si } \max(x,y) = n - 1, \\ 0 & \text{en otro caso} \end{cases}$$

Para n = 4, su matriz interna es:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 \\ 0 & 1 & 2 & 3 \end{pmatrix}$$

2. La t-norma de Lukasiewicz $T_L : C_n \times C_n \longrightarrow C_n$ se define por

$$T_L(x, y) = \max(0, x + y - (n - 1)).$$

Para n = 4, su matriz interna es:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 3 \end{pmatrix}$$

3. El operador mínimo $M : C_n \times C_n \longrightarrow C_n$ es la t-norma más grande y se define por

$$M(x, y) = \min(x, y).$$

Para n = 4, su matriz interna es:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 2 & 2 \\ 0 & 1 & 2 & 3 \end{pmatrix}$$

3. Shiny

En este capítulo se presenta una introducción al paquete Shiny de R, se abordan diferentes conceptos y funciones, junto con una serie de paquetes empleados para mejorar la experiencia del usuario, utilizados en el desarrollo de la aplicación.

3.1. ¿Qué es Shiny?

Shiny es un paquete de R que permite crear aplicaciones web interactivas y dinámicas directamente desde R. Con Shiny, los usuarios pueden crear interfaces de usuario (UI) personalizadas e implementar su propia lógica de programación en R para realizar cálculos, analizar y manipular los datos, y obtener información valiosa de forma dinámica. Entre otras ventajas y utilidades de Shiny se encuentran:

1. Flexibilidad: La alta personalización de Shiny permite a los usuarios adaptar la aplicación a los requerimientos específicos de cada proyecto.
2. Integración: Shiny se integra con otros paquetes y herramientas de R de manera sencilla, lo que facilita la creación de aplicaciones más complejas de manera eficiente.
3. Reutilización: Las aplicaciones Shiny pueden ser fácilmente reutilizadas y adaptadas para su uso en diferentes proyectos y contextos (ver [10]).

3.2. Reactividad en Shiny

La programación reactiva es un estilo de programación que se centra en valores que cambian con el tiempo, y en la propagación automática de cambios a través de un sistema conectado, de manera que los cambios en un componente se propagan automáticamente a los componentes conectados, reaccionando a ellos.

En este sentido, la programación reactiva es especialmente útil en aplicaciones en tiempo real como las de Shiny. Esta sección ha sido documentada fundamentalmente por las pautas del apartado web de shiny sobre reactividad (ver [11]).

Shiny se encarga de la programación reactiva en la aplicación web, permitiendo que los cambios en los datos de entrada del usuario se propaguen automáticamente a través de la aplicación y se actualicen las salidas correspondientes en consecuencia.

La programación reactiva en Shiny se basa en tres tipos de objetos: fuentes reactivas, conductores reactivos y extremos reactivos, representados respectivamente con los siguientes símbolos:

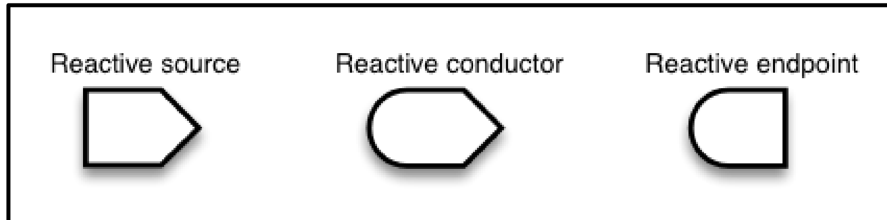

Figura 1. Objetos esenciales de la programación reactiva en Shiny.

Las fuentes reactivas suelen ser objetos que representan entradas del usuario a través de la interfaz de usuario. Por ejemplo, cuando el usuario selecciona un elemento dada una lista de opciones, introduce valores en un campo o hace clic en un botón, estas acciones crearán valores que son fuentes reactivas. Estos objetos son accesibles a través del objeto 'input' en el lado del servidor.

Los conductores reactivos son objetos reactivos útiles para encapsular cualquier tipo de operaciones con datos, en particular las que son lentas o costosas computacionalmente. Se actualizan automáticamente en función de las fuentes u otros conductores reactivos de los que dependa, y devuelven un valor que puede ser utilizado para actualizar salidas en la UI.

Los extremos reactivos suele ser objetos reactivos que generan algún tipo de efecto secundario o salida final en la interfaz de usuario de la aplicación, como un gráfico o una tabla de valores. Estos objetos son accesibles a través del objeto 'output' en el lado del servidor y se actualizan automáticamente en función de los cambios en fuentes reactivas o conductores reactivos de los que dependa.

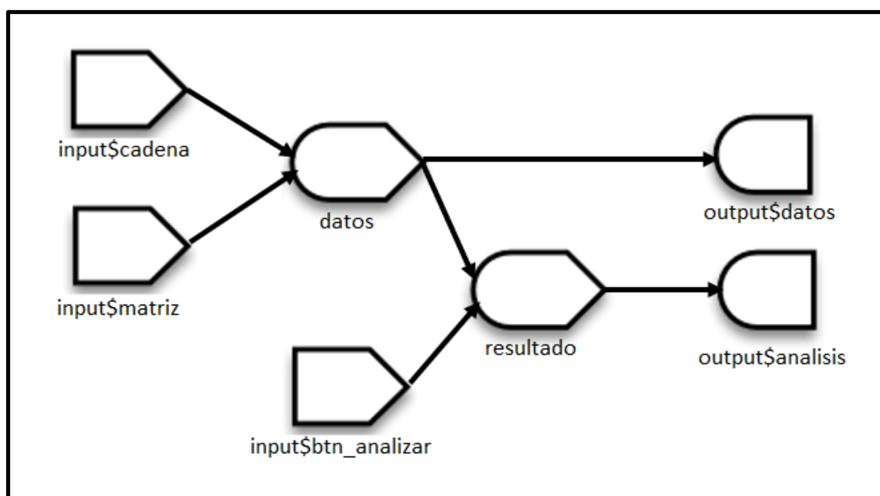

Figura 2. Ejemplo de grafo de reactividad en Shiny.

En este ejemplo, se tienen dos datos de entrada: una cadena y una matriz. Estos datos son procesados por una expresión reactiva llamada 'datos', cuyo resultado es utilizado para producir un output con el mismo nombre que muestra ambas entradas en la UI. Además, se cuenta con otra expresión reactiva llamada 'resultado', que se activa al presionar un botón y realiza un análisis basado en el valor de la expresión 'datos'. El resultado del análisis se muestra en otro output llamado 'análisis' de la UI.

Shiny tiene una clase de objetos que actúan como cada uno de los tres roles:

1. Los valores reactivos son una implementación de las fuentes reactivas
2. Las expresiones reactivas son una implementación de los conductores reactivos.
3. Los observers son una implementación de los extremos reactivos.

3.3. Funciones principales de Shiny

En la presente sección se introducen las principales funciones y objetos del paquete Shiny (ver [12]) que se han utilizado en la aplicación.

3.3.1. Funciones para la UI

En esta subsección se introducen una serie de funciones que se utilizan para crear y diseñar la UI de la aplicación con la que los usuarios interactuarán. Dichas funciones se han dividido en diferentes categorías según su función.

UI Layout

Estas funciones permiten definir la estructura y disposición visual de los elementos que componen la UI en una aplicación Shiny.

Función	Disposición
column()	Define una columna para organizar componentes en una disposición columnar, típicamente dentro de fluidRow() entre otras.
fluidPage() **fluidRow()**	Crea una página con un diseño fluido y adaptable a diferentes dispositivos. Esta página se compone de filas, que se crean con la función fluidRow(), y que a su vez incluyen columnas. Las filas sirven para asegurar de que los elementos que contiene (si caben) se muestran en la misma línea, en un espacio dividido en 12 unidades de ancho. Las columnas, por su parte, definen cuánto espacio ocupan dichos elementos dentro de una fila.
sidebarLayout(), **sidebarPanel(),** **mainPanel()**	Crea una estructura con una barra lateral creada por sidebarPanel(), la cual suele contener elementos de entrada, y un área principal creada por mainPanel(), la cual suele mostrar los resultados.
tabPanel()	Crea un panel que se muestra en pestañas utilizando navbarPage() o tabsetPanel().
navbarPage(), **navbarMenu()**	navbarPage() permite crear una página con una barra de navegación en la parte de arriba de la aplicación. Toma como argumentos el título de la aplicación y un conjunto de tabPanels que al darle click se muestra en la página el contenido de dicho tabPanel. navbarMenu() se utiliza para crear un menú desplegable dentro de la barra de navegación creada con navbarPage(). Permite agrupar diferentes elementos de la aplicación bajo un mismo encabezado. Cada elemento se define utilizando la función tabPanel().
tabsetPanel()	Crea un tabset que contiene elementos tabPanel() para dividir la salida en múltiples secciones de forma independiente. Puede utilizarse dentro de un mainPanel()
splitLayout()	Crea un contenedor que distribuye los elementos en una disposición horizontal, dividiendo el espacio disponible en partes iguales (por defecto) o el especificado.
verticalLayout()	Crea un contenedor que permite organizar los elementos de la UI en una disposición vertical.

conditionalPanel()	Crea un panel que muestra su contenido si la condición se evalúa como TRUE.
helpText()	Crea un texto de ayuda que puede incluirse en un input para proporcionar explicaciones adicionales o algún tipo de contexto en la UI.
Icon()	Crea un icono que puede utilizarse en varias partes de una aplicación

Tabla 1. Funciones para el diseño/disposición de la UI.

Tags

En Shiny, el objeto 'tags' proporciona una serie de funciones que permiten construir y personalizar elementos HTML utilizados para diseñar la interfaz de usuario de la aplicación web, como encabezados, párrafos, campos de entrada, entre otros.

Los elementos HTML creados con tags pueden ser personalizados con clases de estilo, atributos, estilos CSS y eventos para lograr una interfaz de usuario atractiva.

UI Inputs

Shiny viene con una familia de widgets[5] estándar, cada uno asociado con una función R que los crea. Los widgets de nuestro interés son:

Función	Widget
actionButton(), actionLink()	Botón o link de acción
checkboxInput()	Checkbox para (des)seleccionar una opción.
checkboxGroupInput()	Grupo de checkboxes para (des)seleccionar múltiples opciones de forma independientemente.
radioButtons()	Conjunto de radio buttons para elegir una única opcion de una lista de opciones.
numericInput():	Campo de entrada de valores numéricos
sliderInput()	Widget deslizante para elegir un valor dado un rango

[5] Elemento web que solicita al usuario un valor de entrada o algún tipo de interacción.

selectInput() selectizeInput()	Lista de selección para elegir uno o varios elementos de una lista de valores.
textInput()	Campo de entrada de texto simple
textAreaInput()	Campo de entrada de varias líneas para texto más amplio
fileInput()	Campo de entrada para cargar archivos

Tabla 2. Funciones para la creación de widgets de Shiny.

Es necesario asignar un nombre único a cada widget para poder acceder posteriormente a su valor desde la parte del servidor mediante "input$nombre_del_widget". Además, cada widget tiene una función 'update*()' para actualizar su valor desde el servidor.

UI outputs

Son funciones que, junto con las funciones render*() correspondientes en el servidor, crean y muestran diferentes tipos de salida en la UI de la aplicación.

Función	Output
htmlOutput() uiOutput()	Contenedor para la salida generada por renderUI().
plotOutput()	Contenedor para la salida generada por renderPlot().
tableOutput(), dataTableOutput()	Contenedor para la salida generada por renderTable() y renderDataTable(), respectivamente.
verbatimTextOutput(), textOutput()	Contenedor para la salida generada por renderPrint() y renderText(), respectivamente.
downloadButton(), downloadLink()	Crea un botón o link que permite la descarga de archivos de contenidos de la aplicación mediante la función downloadHandler() en el servidor.

Tabla 3. Funciones para la creación de outputs en Shiny.

Es necesario asignar un nombre único a cada output para poder renderizar posteriormente en ellos la salida correspondiente desde el servidor mediante "output$nombre_del_output".

3.3.2. Funciones para el servidor

En esta subsección se introducen una serie de funciones que se utilizan para definir la lógica y la funcionalidad de la aplicación, como procesar datos, realizar cálculos y generar resultados dinámicos en la UI.

Funciones render

Son funciones que, asignándolas a sus elementos de salidas que aparecen en la UI, renderizan diferentes tipos de datos. Permiten actualizar los elementos de la UI en función de los cambios en los elementos reactivos de los que depende.

Función	Render
renderUI()	Renderiza una variable de salida reactiva como HTML.
renderPlot()	Renderiza un gráfico reactivo.
renderTable(), renderDataTable()	Crea una tabla reactiva utilizando HTML estándar o la librería JavaScript 'DataTables', lo que permite lograr una tabla interactiva con más características.
renderPrint(), renderText()	Renderiza una variable de salida reactiva como texto. reactivo. renderPrint() imprime el resultado de la expresión y renderText() lo concatena en una sola cadena de texto.
downloadHandler()	Especifica cómo se debe generar y descargar el archivo.

Tabla 4. Funciones para renderizar outputs en Shiny.

Programación reactiva

Son funciones específicas que proporcionan facilidades para llevar a cabo la reactividad entre las diferentes partes de la aplicación Shiny.

Función	Render

reactive()	Función que se utiliza para crear expresiones reactivas, es decir, expresiones cuyos resultados cambian automáticamente cuando cambia alguna de sus dependencias (ya sean valores reactivos o llamadas a otras expresiones reactivas). Si una expresión reactiva se invalida, todas las expresiones que dependan de ella también se invalidarán. Los valores generados por las expresiones reactivas se pueden utilizar en otras partes de la aplicación.
Is.reactive()	Comprueba si su argumento es una expresión reactiva.
observe()	Función que se utiliza para definir expresiones 'observers' que se ejecutan automáticamente cada vez que alguna de sus dependencias cambie. La diferencia con reactive(), que devuelve un valor reactivo, es que observe() se utiliza para producir efectos secundarios (por ejemplo, realizar acciones como actualizar un input o actualizar un output).
Isolate()	Función que se utiliza para aislar una expresión de la reactividad de Shiny. Es útil cuando quieres evitar que ciertas entradas o expresiones reactivas afecten a la evaluación de una expresión.
observeEvent(), eventReactive()	Funciones que modifican una expresión reactiva o un 'observer' respectivamente para que solo responda a los eventos reactivos especificados.
reactiveVal()	Crea un único valor reactivo. Cuando lees su valor, la expresión que lo llama toma una dependencia reactiva de ese valor, y cuando escribes en él, notifica a toda expresión reactiva que dependa de dicho valor.
reactiveValues()	Crea un objeto (similar a una lista) que almacena valores reactivos. Cada valor se comporta de la misma forma que un 'reactiveVal'.
reactiveValuesToList()	Convierte un objeto 'reactivevalues' a una lista.
Is.reactivevalues()	Comprueba si su argumento es un objeto 'reactivevalues'.
bindEvent()	Función que modifica un objeto para que solo responda a los eventos reactivos especificados. Si se utiliza junto con reactive() y observe(), funciona igual que eventReactive() y observeEvent(). Sin embargo, bindEvent() es más flexible, ya que puede ser combinado con bindCache() y utilizar junto con funciones render.
bindCache()	Permite almacenar en la caché todos los valores anteriores de expresiones reactivas y funciones render.

Tabla 5. Funciones para la reactividad en Shiny.

3.3.3. Otras funciones

Esta subsección contiene funciones de uso general que no se incluyeron en las subsecciones anteriores.

withMathJax

MathJax es una biblioteca de JavaScript de código abierto que permite facilitar la visualización de fórmulas matemáticas en páginas web, mediante el uso de sintaxis principalmente en LaTeX, en un formato fácil de entender, utilizar y de alta calidad. Se integra con Shiny para mostrar información matemática en diferentes partes de la aplicación, como etiquetas HTML y textos, de manera clara y legible (ver [13]).

La integración entre MathJax y Shiny se lleva a cabo a través de la función withMathJax(), la cual funciona como un wrapper para cargar la librería MathJax dentro de una aplicación Shiny. Para el contenido HTML estático, sólo se necesita llamar a la función una vez. En cambio, para la UI dinámica a través de renderUI(), se debe envolver el contenido que contiene expresiones matemáticas en la función withMathJax (ver [14]).

Módulos

La función NS recibe un namespace[6] como argumento y devuelve una función que espera una cadena id como único argumento y devuelve ese id con el namespace añadido con "-" como separador. Su uso está pensado para la modularización en aplicaciones Shiny.

3.3. Módulos en Shiny

La modularización en Shiny hace referencia a la práctica de dividir una aplicación Shiny en módulos independientes y reutilizables, en lugar de escribir todo el código en un solo archivo, lo que puede ser especialmente útil a medida que la aplicación crece en tamaño y complejidad. Esta sección ha sido documentada principalmente de [15].

[6] Identificador único que se utiliza para evitar conflictos de nombres entre los diferentes componentes de una página.

Un módulo en Shiny consiste esencialmente en dos funciones: una función para definir los elementos de la UI y otra función para definir la propia lógica de reactividad del servidor. Los módulos permiten separar la aplicación en componentes más pequeños y manejables, lo que facilita el desarrollo y mantenimiento y reduce la redundancia de código y los errores.

Cada módulo tiene su propio archivo de definición, lo que proporciona una mejor organización del código y facilita su reutilización. Los módulos se pueden usar en varios lugares de una misma aplicación, lo que los convierte en una herramienta muy útil para ayudar a reducir la complejidad del proyecto y mejorar la eficiencia.

Los módulos permiten evitar conflictos de nombres entre diferentes partes de la aplicación al tener cada uno su propio namespace. Las funciones y variables dentro de un módulo pueden tener los mismos nombres que las de otro módulo, siempre y cuando los namespaces sean distintos para cada uno.

Por último, a los módulos de Shiny se les pueden pasar valores como argumentos, y estos valores pueden ser utilizados dentro del código del módulo para realizar diferentes tareas o cálculos. Además, un módulo también puede devolver valores en el lado del servidor que pueden ser utilizados en otras partes de la aplicación, permitiendo incluso la comunicación entre módulos en una aplicación Shiny.

3.4. Paquetes utilizadas en Shiny

A continuación se presentan una serie de paquetes que permiten añadir funcionalidades y elementos visuales a una aplicación Shiny de manera fácil y rápida, que no se encuentran disponibles de manera nativa en Shiny. Las funciones más relevantes de los paquetes que se mencionan a continuación se mostrarán en el Apéndice B.

shinydashboard

El paquete 'shinydashboard' ayuda a crear paneles de control interactivos (dashboards) en aplicaciones Shiny, proporcionando herramientas adicionales para su diseño y construcción. Se divide en tres partes principales: barra lateral, cuerpo y cabecera, lo que permite una mayor flexibilidad en el diseño y organización del contenido.

El paquete también proporciona plantillas y elementos visuales predefinidos que se pueden personalizar fácilmente, simplificando su uso a usuarios sin experiencia en diseño o programación (ver [16]).

shinyWidgets

El paquete 'shinyWidgets' ofrece una variedad de widgets personalizados y componentes de UI adicionales para mejorar y enriquecer las aplicaciones Shiny. Cada widget tiene una función de 'update*()' para cambiar el valor de entrada desde el servidor (ver [17]).

bsplus

El paquete 'bsplus' amplía las capacidades de Shiny al proporcionar una sintaxis simplificada para crear una colección de componentes de Bootstrap[7], que permiten añadir funcionalidad y estilo a la aplicación (ver [18]).

shinyMatrix

El paquete 'shinyMatrix' proporciona un campo de entrada para introducir cómodamente matrices en una aplicación Shiny. Estas matrices pueden ser editadas y visualizadas en tiempo real por los usuarios. El paquete también incluye una serie de opciones para su personalización y el formateo de las matrices (ver [19]).

shinycssloaders

Cuando un output de Shiny (como un gráfico, una tabla, etc.) se está recalculando, permanece atenuado en gris hasta que el nuevo valor haya sido calculado y se muestre por pantalla. El paquete 'shinycssloaders' permite añadir animaciones de carga a las salidas que se muestran/ocultan automáticamente cuando la salida se recalcula/haya terminado. Proporciona una serie de opciones para personalizarlo (ver [20]).

[7] marco de diseño web que facilita la creación de interfaces de usuario atractivas y responsivas.

shinyvalidate

El paquete 'shinyvalidate' permite mejorar la experiencia de usuario de las aplicaciones Shiny añadiendo validaciones en tiempo real que permiten indicar cuando faltan entradas que son requeridas o cuando los valores de entrada no son válidos mediante mensajes alrededor de los campo de entradas en la UI (ver [21]).

shinyalert

El paquete 'shinyalert' permite crear fácilmente mensajes emergentes (modales) atractivos en Shiny. Los modales pueden contener texto, imágenes, botones OK/Cancelar, entradas y salidas Shiny, siendo posible personalizarlo en otros aspectos, como establecer un temporizador para cerrarse automáticamente o especificar código personalizado para que se ejecute cuando se cierre el modal (ver [22]).

shiny.i18n

El paquete 'shiny.i18n' es una herramienta que facilita la internacionalización de las aplicaciones Shiny, lo que significa que permite adaptar el contenido de la aplicación a diferentes idiomas. Ofrece una forma sencilla y ordenada de administrar archivos de traducción, y proporciona funciones para la localización de textos, números, etc., (ver [23]).

shinyjs

El paquete 'shinyjs' permite realizar acciones comunes de JavaScript en aplicaciones Shiny que mejoran la experiencia de usuario sin necesidad de tener conocimientos de JavaScript. Proporciona una interfaz fácil de usar para estas acciones a través de funciones R que envuelven la lógica JavaScript subyacente. Además de estas acciones, 'shinyjs' también permite invocar tus propias funciones personalizadas de JavaScript desde R (ver [24]).

4. Descripción del funcionamiento de la aplicación Shiny

La aplicación permite al usuario seleccionar el idioma a su preferencia entre inglés (por defecto) y español, y presenta cuatro secciones principales en la barra lateral, cada una con diferentes funcionalidades. A continuación se enumeran cada una de estas secciones.

4.1. Análisis del operador

La sección 'Análisis del operador' de la aplicación se encarga de visualizar los datos introducidos por el usuario para la matriz de un operador discreto definido en una cadena finita de tamaño 'n' señalado por el usuario y determinar las propiedades que cumple dicho operador. En función de estas propiedades, se clasifica el operador y se muestra los resultados al usuario.

Esta sección consta de tres subsecciones que trabajan de forma independiente y que únicamente difieren en la forma de introducir y tratar los datos requeridos para el análisis.

Antes de explicar las diferencias entre cada una de ellas, vamos a explicar la parte de visualización y descarga que comparten todas ellas.

Cuando el usuario hace clic en cualquier botón de actualizar valores, se mostrará en pantalla:

1. La cadena introducida por el usuario en notación matemática como una matriz de una fila y tantas columnas como elementos tenga, en orden ascendente.
2. La matriz introducida por el usuario en notación matemática como una matriz cuadrada del mismo tamaño que la cadena.
3. Un botón para iniciar el análisis del operador que el usuario puede pulsar una vez que ha verificado que los datos introducidos son correctos.

Una vez pulsado el botón de análisis, se muestra:

1. Se visualiza para cada propiedad si se cumple o no dicha propiedad, y se muestra un contraejemplo si es necesario, excepto en el caso del elemento neutro, para el cual se muestra el valor correspondiente si cumple la propiedad.
2. Se muestra la clasificación del operador según el apartado 1.3. Si pertenece a alguna de ellas, se muestra al usuario dicha clasificación y las propiedades adicionales que cumple. En caso contrario, solamente se muestran las propiedades que cumple.

En la zona superior de la interfaz, se encuentran dos opciones:

1. Un menú desplegable para exportar/descargar los resultados en tres formatos posibles:
   - **1.1.** Texto plano/CSV: Se exporta la matriz como archivo de texto plano o CSV. Se debe introducir el nombre del archivo, elegir el separador de valores y decidir si se desea guardar la cadena como encabezado (primera línea del archivo).
   - **1.2.** PDF: Se descarga la cadena y la matriz como archivo PDF. Es posible agregar una descripción en un campo de texto y elegir si se incluyen las propiedades del operador en el documento.
2. Un checkbox para indicar si mostrar los índices de la matriz correspondiente con cada elemento de la cadena. Además, se mostrará el número de fila/columna al que pertenece cada elemento.

A continuación se presentarán las tres formas que tiene el usuario para introducir un operador binario definido en una cadena. Cada una de estas formas ofrece una forma diferente y eficiente de introducir los datos necesarios.

4.1.1. Matriz manual

Esta subsección incluye la introducción manual de la cadena y la matriz asociada al operador.

El usuario puede interactuar en tres partes diferentes:

1. Elegir un tamaño 'n' de la cadena.
2. Introducir la cadena verificando que tenga la longitud adecuada de 'n' elementos y que no esté vacía. Los valores válidos pueden ser números naturales, números decimales o fracciones. Si se modifica el valor de 'n', la cadena mantiene sus valores; en caso contrario,

se autocompleta con la cadena {0, ..., n-1}. Además, se han agregado funcionalidades para borrar, copiar y pegar la cadena.

3. Insertar los valores en la matriz nxn asegurándose de que todas las celdas estén llenas. La matriz se ajusta automáticamente cuando se modifica el valor de n y mantiene los valores que tenía insertados.

Las comprobaciones que se realizan para garantizar el buen funcionamiento son las siguientes:

1. Comprobar que la cadena introducida esté completa y sus valores no estén repetidos.
2. Comprobar que la matriz introducida esté completa y que sus valores sean válidos y estén presentes en la cadena introducida anteriormente.

Si alguna comprobación falla, se mostrará un mensaje en la interfaz de usuario o se abrirá un modal con información de la causa del problema.

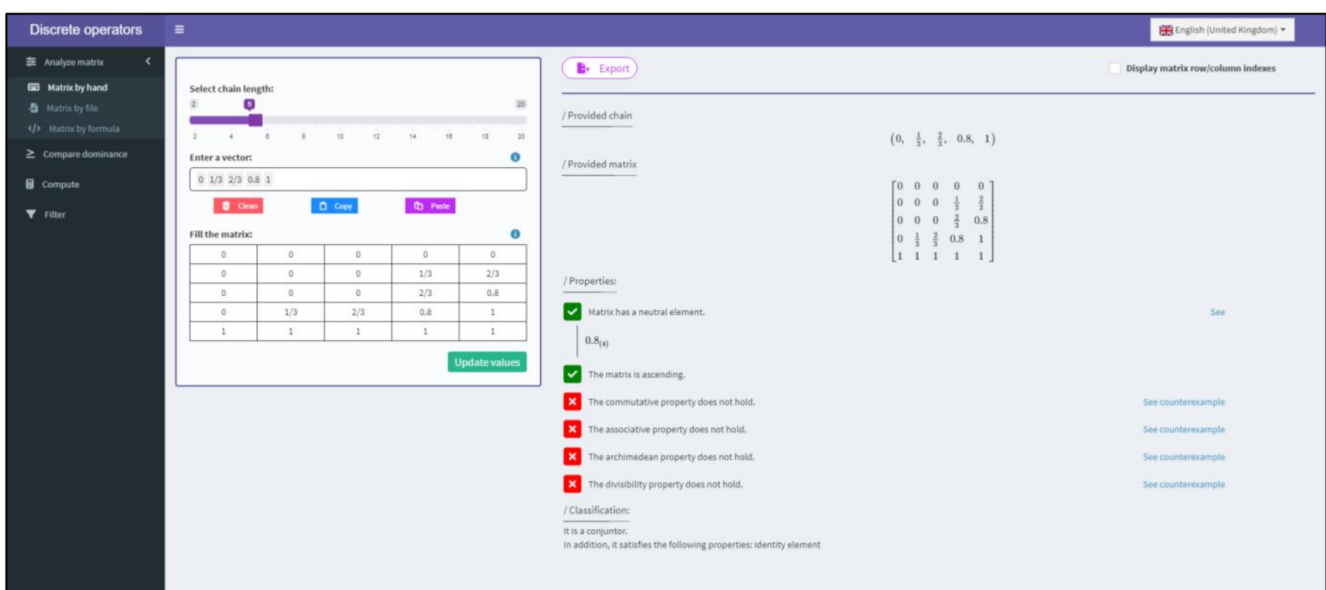

Figura 3. Interfaz de usuario de la subsección 'Matriz manual'.

4.1.2. Matriz por archivo

Esta subsección incluye la introducción por archivo de texto de la matriz asociada al operador junto con la cadena como cabecera del archivo o, en caso contrario, su introducción manual.

El usuario puede interactuar en tres partes diferentes:

1. Cargar un archivo de texto plano (.txt) o CSV (.csv) que contenga la matriz asociada al operador y, opcionalmente, la cadena como cabecera del archivo.
2. Un switch para indicar si el archivo contiene una cabecera.
3. En caso de que se desactive el switch, se mostrará un input para introducir la cadena con las mismas propiedades y condiciones explicadas en la subsección 4.1.1.

Las comprobaciones que se realizan para garantizar el buen funcionamiento son las siguientes:

1. Comprobar que la cadena esté completa y sus valores no estén repetidos. Si el archivo tiene cabecera, comprobar además que los valores sean válidos.
2. Comprobar que la matriz sea cuadrada y que la cadena coincida en tamaño con ella
3. Comprobar que la matriz esté completa y que sus valores sean válidos y estén presentes en la cadena.

Si alguna comprobación falla, se mostrará un mensaje en la interfaz de usuario o se abrirá un modal con información de la causa del problema.

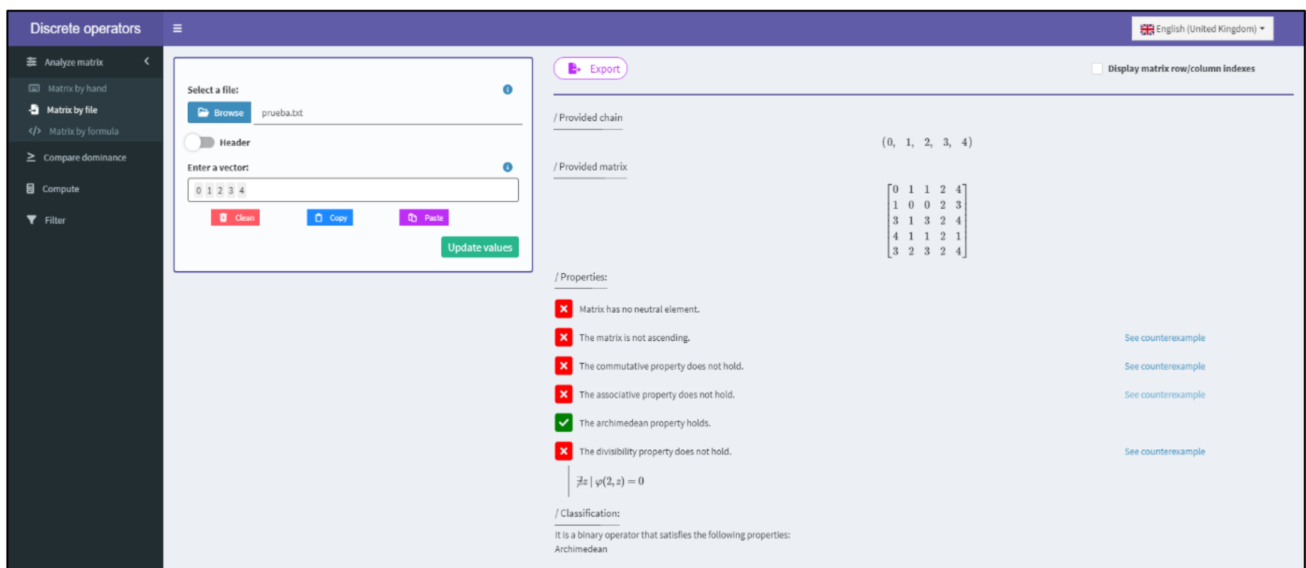

Figura 4. Interfaz de usuario de la subsección 'Matriz por archivo'.

4.1.3. Matriz por fórmula

Esta subsección incluye la introducción manual de la cadena y la fórmula asociada a la matriz que representa el operador binario.

El usuario puede interactuar en tres partes diferentes:

1. Elegir un tamaño 'n' de la cadena.
2. Introducir la cadena con las mismas propiedades y condiciones explicadas en la subsección 4.1.1.
3. Introducir la formula que representa al operador binario en la sintaxis del lenguaje R y que haga referencia a 'x' e 'y' como los operandos.

Las comprobaciones que se realizan para garantizar el buen funcionamiento son las siguientes:

1. Comprobar que la cadena introducida esté completa y sus valores no estén repetidos.
2. Comprobar que el número de paréntesis izquierdos y derechos coincida en la fórmula, y que solo se haga referencia a los operandos con las letras 'x' e 'y'.
3. Comprobar que la formula provista sea sintácticamente correcta en R.
4. Comprobar que los valores de la matriz construida a partir de la fórmula estén presentes en la cadena introducida anteriormente.

Si alguna comprobación falla, se mostrará un mensaje en la interfaz de usuario o se abrirá un modal con información de la causa del problema.

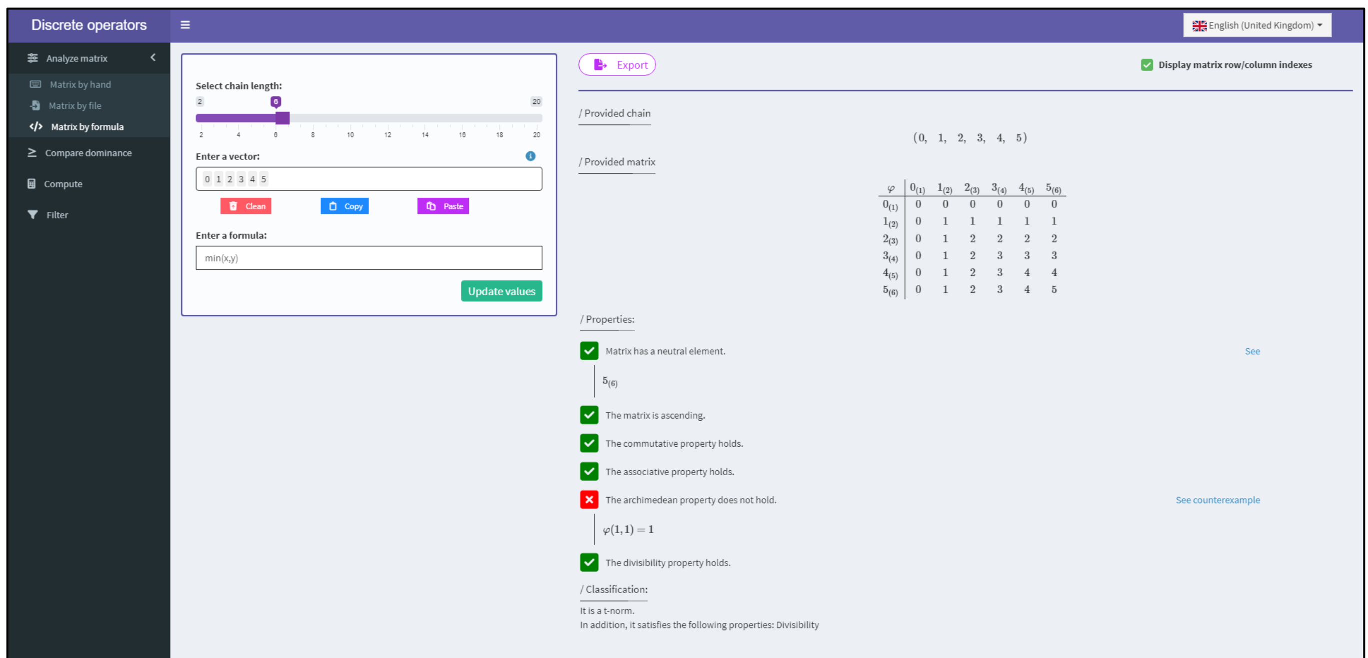

Figura 5. Interfaz de usuario de la subsección 'Matriz por formula'.

4.2. Calcular

La sección 'Calcular' de la aplicación se encarga de determinar el cardinal de conjuntos de operadores a partir de la selección entre las cuatro propiedades básicas (elemento neutro, monotonía ascendente, conmutatividad y asociatividad) y guardar un número determinado de matrices como muestra.

El usuario puede interactuar con tres partes diferentes:

1. Elegir el tamaño de la cadena. Si el usuario selecciona un tamaño 'n' de la cadena, las matrices generadas serán de tamaño nxn y estarán definidas en la cadena C = {0,…,n-1}.
2. Seleccionar las propiedades que las matrices deben cumplir. En caso de elegir la propiedad de elemento neutro, se debe especificar la posición de dicho elemento entre 1 y n.
3. Indicar el número de matrices a guardar para su posterior descarga en un rango de 0 a 3 millones. Se guardarán hasta la cantidad de matrices indicada, a menos que el número total de matrices sea menor.

Una vez pulsado el botón de calcular, se mostrará en pantalla:

1. El valor de n seleccionado.
2. La fórmula para calcular el cardinal del conjunto de operadores seleccionado en base a n, si existe.
3. El resultado del cálculo del cardinal del conjunto de operadores seleccionado.

4. Dos opciones para descargar las matrices: exportarlas en un documento PDF o descargarlas como un archivo de extensión R.

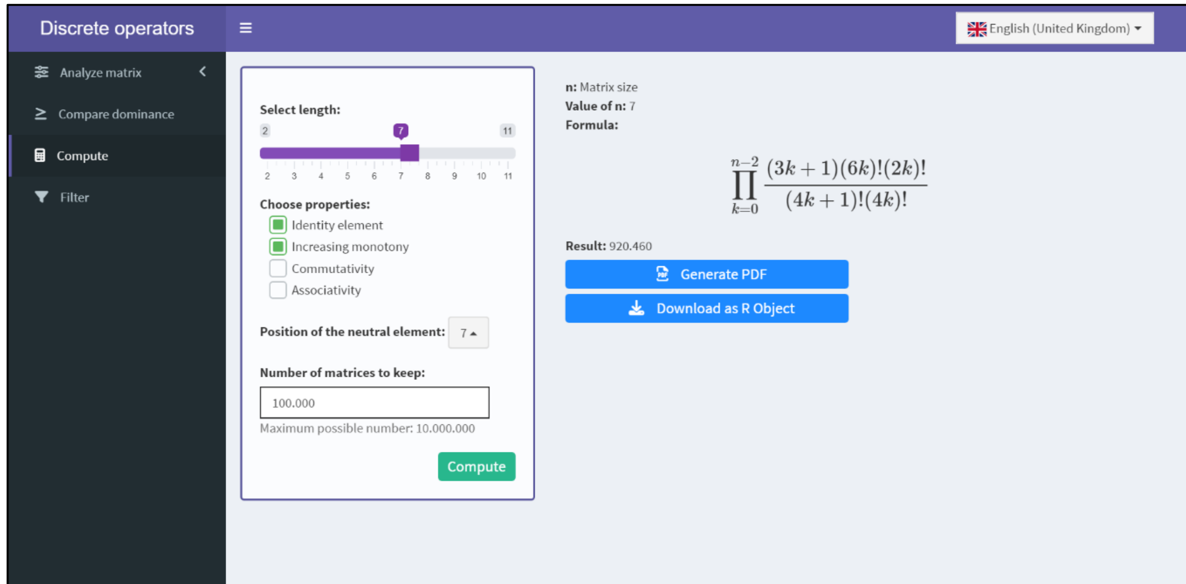

Figura 6. Interfaz de usuario de la sección 'Calcular'.

4.3. Comparar dominancia

La sección 'Comparar dominancia' de la aplicación permite aplicar la propiedad de dominación entre dos matrices diferentes o entre una matriz y una lista de matrices. El usuario puede cambiar en cualquier momento entre las dos opciones de aplicar la propiedad según sea necesario.

Ambas opciones comparten las siguientes partes:

1. Elegir el tamaño de la cadena y, por ende, el de las matrices.
2. Introducir la cadena con las mismas condiciones explicadas en la subsección 4.1.1.
3. Seleccionar un modo de introducir una matriz (A) entre las explicadas en la subsección 4.1 con las mismas condiciones. En el caso de introducir la matriz mediante archivo, se considerará la matriz sin encabezado.

Ahora se detallarán las diferencias entre cada opción.

<u>Matriz vs Matriz</u>

Si el usuario elige esta opción, deberá introducir una segunda matriz (B) con las mismas opciones que al introducir la matriz A.

Antes de proceder a la comparación de ambas matrices, es necesario hacer clic en el botón de comprobar para verificar los valores introducidos por el usuario. Si no hay ningún problema, al hacer clic en el botón de comparar, se mostrarán en pantalla los siguientes posibles casos:

1. La matriz A domina a la matriz B, o viceversa. En ambos casos se muestra un ejemplo con la combinación de x,y,z y t que hace que $A(B(x,z), B(y,t)) > B(A(x,y),A(z,t))$, o viceversa.
2. Ninguna de las dos matrices domina a la otra.

Matriz vs Lista de matrices

Si el usuario elige esta opción, deberá introducir un archivo de extensión R que debe contener una lista de matrices. Además, el usuario debe seleccionar si desea todas las matrices de M que dominen a A o sean dominadas A.

Antes de proceder a la comparación de ambas matrices, es necesario hacer clic en el botón de comprobar para verificar los valores introducidos por el usuario y comprobar que todas las matrices de la lista tengan el mismo tamaño, el cual debe ser igual al tamaño seleccionado por el usuario en la aplicación.

Si no hay ningún problema, al hacer clic en el botón de comparar, se mostrará en pantalla el número de matrices que dominan a A o son dominadas por A, junto con dos opciones para descargar las matrices: exportarlas en un documento PDF o descargarlas como un archivo con extensión R.

Cabe mencionar que para que funcione correctamente esta opción, todas las matrices deben estar definidas en la cadena C = {0, …, n}.

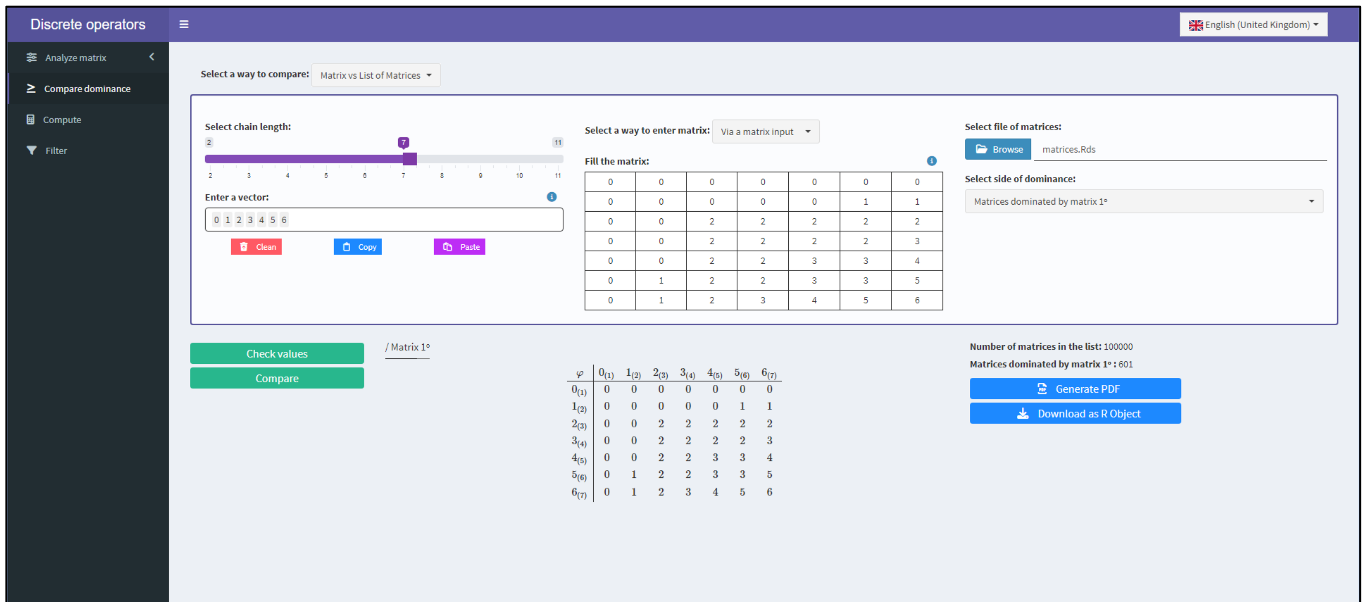

Figura 7. Interfaz de usuario de la sección 'Comparar dominancia'.

4.4. Filtrar matrices

La sección 'Filtrar matrices' de la aplicación permite al usuario filtrar una lista de matrices utilizando alguna propiedad. El usuario puede elegir entre varias opciones y seleccionar una sola propiedad a la vez.

El usuario deberá introducir un archivo de extensión R que contenga una lista de matrices y seleccionar una opción de filtrado. Las opciones de propiedades que el usuario puede elegir son la maximalidad, la divisibilidad y la propiedad arquimediana.

Al hacer clic en el botón de filtrar, se mostrará en pantalla el número de matrices original y el número de matrices resultantes después del filtrado. También se ofrecerán dos opciones para descargar las matrices: exportarlas en un documento PDF o descargarlas como un archivo de extensión R.

Cabe mencionar que para que funcione correctamente esta sección, todas las matrices de la lista deben tener el mismo tamaño y estar definidas en la cadena C = {0, …, n}.

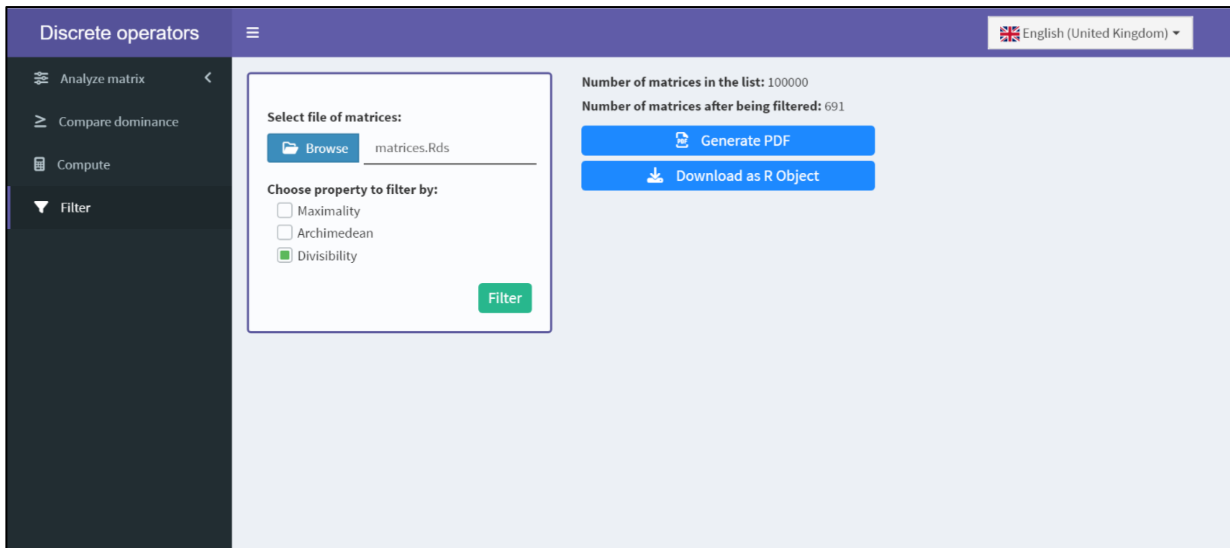

Figura 8. Interfaz de usuario de la sección 'Filtrar matrices'.

5. Algoritmos para calcular cardinal de operadores binarios

A continuación, se presentan una serie de algoritmos empleados en el apartado 4.1.3 para la generación y el cálculo del cardinal de conjuntos de operadores binarios dado ciertas propiedades, los cuales son una parte fundamental de este trabajo.

Antes de abordar la explicación de los algoritmos, es necesario mencionar una función que se utiliza en algunos de ellos posteriormente.

Comprobar si una matriz de tamaño nxn es asociativa conlleva una complejidad n^3. Esto puede ser computacionalmente costoso a la hora de generar una lista de matrices que cumplan la propiedad asociativa. Sin embargo, existe un algoritmo más eficiente que aprovecha propiedades algebraicas de las matrices conmutativas (ver [25]).

La siguiente tabla muestra el pseudocódigo que describe el algoritmo para verificar la asociatividad de un operador binario conmutativo utilizado por la función 'asociativo_conm()'.

asociativo_conm <- function(cadena, M)

cadena -> Cadena en la que está definida el operador

M -> Matriz del operador con la cadena como índices

1. Formar una matriz A con n columnas y n*(n + 1)/2 filas. Dividir A en submatrices $A_1, A_2, ..., A_n$, que contienen respectivamente n, n - 1, ..., 1 filas. Etiquetar las columnas de A exactamente como las columnas de M. Etiquetar las filas de A_1, A_2,.... ,A_n con los valores, empezando por la diagonal, de las filas 1,2,..., n, respectivamente, de M.
2. Insertar en cada fila de A los valores de la fila del mismo índice de M.
3. Hacer lo siguiente para cada una de las submatrices $A_1, A_2, ..., A_n$ de manera sucesiva:
   3.1. Comparar la fila superior con la columna de la izquierda. Si no son iguales, detente; en caso contrario, elimínalas.

3.2. Comparar la fila siguiente con la columna siguiente, así como con la columna izquierda de la submatriz siguiente. Si no son iguales, detente; si no, elimínalas. Repetir para las filas sucesivas hasta que toda la submatriz esté vacía.

Si, al final, todos los elementos de A (excepto el último elemento de A_n) han sido eliminados, entonces el operador es asociativo.

Tabla 6. Pseudocódigo de la función asociativo_conm().

La función 'neutro()' genera matrices cuadradas de tamaño nxn con elemento neutro en la posición especificada como argumento, y asociativas si se indica con un valor booleano también como argumento. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'neutro()':

```
neutro <- function(n, pos, limite, seguir, asoc)
//n -> Tamaño de la cadena y por ende de la matriz.
//pos -> Posición del elemento neutro de la matriz.
//limite -> Número máximo de matrices a guardar.
//seguir -> Valor booleano que indica si seguir contando cuando se supera el límite.
//asoc -> Valor booleano que indica si la matriz debe ser asociativa.

índices <- Índices de la matriz en orden descendiente, excepto 'pos'.
matriz <- Matriz de 0s con elemento neutro en la posición 'pos'.
bandera <- Valor para indicar si la matriz ha sido ya modificada y si es válida (inicial a 0)
lista <- Lista para guardar las matrices
cont <- Contador del número de matrices generadas
fin <- Valor para indicar si se debe salir del bucle (inicial a 0)

While fin = 0
    For Each i In índices
        For Each j In índices
            If bandera = 0
                If matriz[i, j] < n-1
                    matriz[i, j] <- matriz[i, j] + 1
```

```
                    bandera <- 1
                    If asoc = FALSE o asociativa(matriz) = TRUE
                        cont <- cont + 1
                        bandera <- 2
                    End If
                Else
                    matriz[i, j] <- 0
                End If
            End If
        End For
    End For

    If bandera = 2
        If cont <= limite
            lista.añadir(matriz)
        Else If seguir = FALSE
            Devolver lista.
        End If
    Else If bandera = 0
        fin <- 1
    End If
End While
Devolver lista y cont.
```

Tabla 7. Pseudocódigo de la función neutro().

Este método iterativo recorre todas las filas y columnas de la matriz de abajo hasta arriba y de derecha a izquierda, a excepción de la fila y columna correspondientes al elemento neutro.

Para cada celda en la matriz, si la matriz no ha sido modificada previamente y el valor de la celda es menor que el valor máximo permitido, se incrementa el valor en esa celda en 1. Si no lo es, se reinicia dicha celda a cero.

Si se ha modificado la matriz (y es asociativa si 'asoc' es verdadero) se aumenta el contador. Si no se ha alcanzado el límite máximo, la matriz se agrega a la lista de matrices generadas. En otro caso, si 'seguir' es falso se devuelve directamente la lista de matrices.

La función 'monotono()' genera matrices monótonas ascendentes cuadradas de tamaño nxn, y asociativas si se indica con un valor booleano como argumento. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'monotono()':

```
monotono <- function(n, limite, asoc)
//n -> Tamaño de la cadena y por ende de la matriz.
//limite -> Número máximo de matrices a guardar.
//asoc -> Valor booleano que indica si la matriz debe ser asociativa.

índices <- Índices de la matriz en orden descendiente.
matriz <- Matriz de 0s.
bandera <- Valor para indicar si la matriz ha sido ya modificada y si es válida (inicial a 0).
lista <- Lista para guardar las matrices.
cont <- Contador del número de matrices generadas.
fin <- Valor para indicar si se debe salir del bucle (inicial a 0).

While fin = 0
    For Each i In índices
        For Each j In índices
            If bandera = 0
                If matriz[i, j] < n-1
                    matriz[i, j] <- matriz[i, j] + 1
                    bandera <- 1
                    For col = {j, …, 1}
                        Asignar el valor matriz[i, col] a las filas posteriores.
                    End For
                    If j < n
                        If i > 1
```

```
                    For col = {j+1, …, n}
                        max_v <- max(mat[i, col-1], mat[i-1, col])
                        Asignar max_v a las filas posteriores.
                    End For
                Else If
                    Asignar mat[i, j] a las filas y columnas posteriores.
                End If
            End If
            If asoc = FALSE o asociativa(matriz) = TRUE
                cont <- cont + 1
                bandera <- 2
            End If
        End If
    End If
  End For
End For

If bandera = 2
    If cont <= limite
        lista.añadir(matriz)
    End If
Else If bandera = 0
    fin <- 1
End If
End While
Devolver lista y cont.
```

Tabla 8. Pseudocódigo de la funcion monotono().

Este método iterativo recorre todas las filas y columnas de la matriz de abajo hasta arriba y de derecha a izquierda.

Para cada celda en la matriz, si la matriz no ha sido modificada previamente y el valor de la celda es menor que el valor máximo permitido, se incrementa el valor en esa celda en 1 y se mantiene la monotonía izquierda y derecha de la matriz de la siguiente forma:

1. Para cada columna 'col' desde la actual hasta la primera, asignar el valor de la celda en la fila actual (i) y columna 'col', a todas las filas desde 'i' hasta la última.
2. Para cada una de las columnas posteriores, asignar el valor máximo entre la celda anterior de la fila actual y la celda superior de la columna actual, a todas las filas desde 'i' hasta la última. Si la fila actual es la primera fila de la matriz, simplemente se asigna a todas las filas y columnas siguientes el valor de la celda actual.

Si se ha modificado la matriz (y es asociativa si 'asoc' es verdadero) se aumenta el contador. Si no se ha alcanzado el límite máximo, la matriz se agrega a la lista de matrices generadas.

La función 'conmutativo()' genera matrices asociativas cuadradas de tamaño nxn, y asociativas si se indica con un valor booleano también como argumento. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'conmutativo()':

```
conmutativo <- function(n, limite, seguir, asoc)
//n -> Tamaño de la cadena y por ende de la matriz.
//limite -> Número máximo de matrices a guardar.
//seguir -> Valor booleano que indica si seguir contando cuando se supera el límite.
//asoc -> Valor booleano que indica si la matriz debe ser asociativa.

indices <- Índices de la matriz en orden descendiente.
matriz <- Matriz de 0s.
bandera <- Valor para indicar si la matriz ha sido ya modificada y si es válida (inicial a 0).
lista <- Lista para guardar las matrices.
cont <- Contador del número de matrices generadas.
fin <- Valor para indicar si se debe salir del bucle (inicial a 0).

While fin = 0
        For Each i In índices
```

```
            For j = {n, ..., i}
                If bandera = 0
                    If matriz[i, j] < n-1
                        matriz[i, j] <- matriz[i, j] + 1
                        matriz[j, i] <- matriz[i, j]
                        bandera <- 1
                        If asoc = FALSE o asociativa_conm(..., matriz) = TRUE
                            cont <- cont + 1
                            bandera <- 2
                        End If
                    Else
                        matriz[i, j] <- 0
                        matriz[j, i] <- 0
                    End If
                End If
            End For
        End For

        If bandera = 2
            If cont <= limite
                lista.añadir(matriz)
            Else If seguir = FALSE
                Devolver lista.
            End If
        Else If bandera = 0
            fin <- 1
        End If
End While
Devolver lista y cont.
```
Tabla 9. Pseudocódigo de la función conmutativo().

Este método iterativo recorre todas las filas de la matriz de abajo hacia arriba y todas las columnas de derecha a izquierda hasta llegar a la diagonal.

Para cada celda en la matriz, si la matriz no ha sido modificada previamente y el valor de la celda es menor que el valor máximo permitido, se incrementa el valor en esa celda y en su simétrica en 1. Si no lo es, se reinicia ambas celdas a cero.

Si se ha modificado la matriz (y es asociativa si 'asoc' es verdadero), se aumenta el contador. Si no se ha alcanzado el límite máximo, la matriz se agrega a la lista de matrices generadas. En otro caso, si 'seguir' es falso se devuelve directamente la lista de matrices.

La función 'asociativo()' genera matrices asociativas cuadradas de tamaño nxn. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'asociativo()':

```
asociativo <- function(n, limite)
//n -> Tamaño de la cadena y por ende de la matriz.
//limite -> Número máximo de matrices a guardar.

índices <- Índices de la matriz en orden descendiente.
matriz <- Matriz de 0s.
bandera <- Valor para indicar si la matriz ha sido ya modificada y si es válida (inicial a 0).
lista <- Lista para guardar las matrices.
cont <- Contador del número de matrices generadas.
fin <- Valor para indicar si se debe salir del bucle (inicial a 0).

Mientras fin = 0 Hacer
    For Each i In índices
        For Each j In índices
            If bandera = 0
                If matriz[i, j] < n-1
                    matriz[i, j] <- matriz[i, j] + 1
                    bandera <- 1
                    If asociativa(matriz) = TRUE
```

```
                        cont <- cont + 1
                        bandera <- 2
                    End If
                Else
                    matriz[i, j] <- 0
                End If
            End If
        End For
    End For

    If bandera = 2
        If cont <= limite
            lista.añadir(matriz)
        End If
    Else If bandera = 0
        fin <- 1
    End If
End While
Devolver lista y cont.
```
Tabla 10. Pseudocódigo de la función asociativo().

Este método iterativo recorre todas las filas y columnas de la matriz de derecha a izquierda y de abajo hasta arriba.

Para cada celda en la matriz, si la matriz no ha sido modificada previamente y el valor de la celda es menor que el valor máximo permitido, se incrementa el valor en esa celda y en 1. Si no lo es, se reinicia dicha celda a cero.

Si se ha modificado la matriz y es asociativa, se aumenta el contador. Si no se ha alcanzado el límite máximo, la matriz se agrega a la lista de matrices generadas.

La función 'neutro_conm()' genera matrices cuadradas de tamaño nxn, las cuales son conmutativas y tienen un elemento neutro en la posición especificada como argumento, y

asociativas si se indica con un valor booleano también como argumento. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'neutro_conm()':

```
neutro_conm <- function(n, pos, limite, seguir, asoc)
//n -> Tamaño de la cadena y por ende de la matriz.
//pos -> Posición del elemento neutro de la matriz.
//limite -> Número máximo de matrices a guardar.
//seguir -> Valor booleano que indica si seguir contando cuando se supera el límite.
//asoc -> Valor booleano que indica si la matriz debe ser asociativa.

índices <- Índices de la matriz en orden descendiente, excepto 'pos'.
matriz <- Matriz de 0s con elemento neutro en la posición 'pos'.
bandera <- Valor para indicar si la matriz ha sido ya modificada y si es válida (inicial a 0).
lista <- Lista para guardar las matrices.
cont <- Contador del número de matrices generadas.
fin <- Valor para indicar si se debe salir del bucle (inicial a 0).

Whle fin = 0
    For Each i en índices
        For  j = {n, ..., i}
            If bandera = 0
                If matriz[i, j] < n-1
                    matriz[i, j] <- matriz[i, j] + 1
                    matriz[j, i] <- matriz[i, j]
                    bandera <- 1
                    If asoc = FALSE o asociativa_conm(..., matriz) = TRUE
                        cont <- cont + 1
                        bandera <- 2
                    End If
                Else
                    matriz[i, j] <- 0
                    matriz[j, i] <- 0
```

```
                  End If
              End If
          End For
      End For

      If bandera = 2
          If cont <= limite
              lista.añadir(matriz)
          Else
              Devolver lista.
          End If
      Else If bandera = 0
          fin <- 1
      End If
  End While
  Devolver lista y cont.
```
Tabla 11. Pseudocódigo de la función neutro_conm().

Este método iterativo es similar a 'conmutativo()', excepto que no se recorre la fila y columna correspondientes al elemento neutro.

La función 'monotono_conm()' genera matrices cuadradas de tamaño nxn, las cuales son monótonas ascendentes y conmutativas, y asociativas si se indica con un valor booleano también como argumento. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'monotono_conm()':

```
monotono_conm <- function(n, limite, asoc)
//n -> Tamaño de la cadena y por ende de la matriz.
//limite -> Número máximo de matrices a guardar.
//asoc -> Valor booleano que indica si la matriz debe ser asociativa.

índices <- Índices de la matriz en orden descendiente.
matriz <- Matriz de 0s.
bandera <- Valor para indicar si la matriz ha sido ya modificada y si es válida (inicial a 0).
lista <- Lista para guardar las matrices.
cont <- Contador del número de matrices generadas.
fin <- Valor para indicar si se debe salir del bucle (inicial a 0).

While fin = 0
    For Each i en índices
        For Each j = {n, …, i}
            If bandera = 0
                If matriz[i, j] < n-1
                    bandera <- 1
                    v <- matriz[i, j] + 1
                    If j-i >= 2
                        For col = {i, …, j-1}
                            v2 <- P[i, col]
                            Para cada columna 'col', asignar a las filas desde 'i'
                            hasta 'col' con v2 y al contrario.
                        End For
                    End If
                    If i = 1
                        Asignar v a toda la matriz a partir de la fila y columna
                        actual.
                    Else
```

```
                                    Asignar 'v' a todas las filas desde la actual hasta la fila 'j'
                                    en la columna actual.
                                    For col = {j+1, ..., n}
                                        v_m <- max(v, matriz[i-1, col])
                                        Para cada columna 'col', asignar 'v_m' a todas las
                                        filas desde i hasta 'col'.
                                    End For
                                End If
                                If asoc = FALSE o asociativa_conm(..., matriz) = TRUE
                                    cont <- cont + 1
                                    bandera <- 2
                                End If
                            End If
                        End If
                    End For
                End For

                If bandera = 2
                    If cont <= limite
                        lista.añadir(matriz)
                    End If
                Else If bandera = 0
                    fin <- 1
                End If
End While
Devolver lista y cont.
```

Tabla 12. Pseudocódigo de la función monotono_conm().

Este método iterativo recorre todas las filas de la matriz de abajo hacia arriba y todas las columnas de derecha a izquierda hasta llegar a la diagonal.

Para cada celda en la matriz, si la matriz no ha sido modificada previamente y el valor de la celda es menor que el valor máximo permitido, se incrementa el valor en dicha celda en 1 y se mantiene la monotonía izquierda y derecha de la matriz de la siguiente forma:

1. Si la diferencia entre el índice de la columna actual (j) y el índice la fila actual (i) es mayor que 2, entonces para cada columna 'col' desde la diagonal hasta 'i' sin incluir, se asigna el valor de la celda en la fila 'i' y columna 'col', a todas las filas desde la fila 'i' hasta 'col'.
2. Si la fila actual es la primera fila de la matriz, directamente se asigna el valor de la celda actual (v) a las filas y columnas posteriores. En cualquier otro caso, se asigna 'v' a todas las filas desde 'i' hasta 'j' para la columna actual. Para el resto de columnas 'col' posteriores, asignar el valor máximo entre 'v' y la celda superior de cada columna, a todas las filas desde la fila 'i' hasta 'col'.

Si se ha modificado la matriz (y es asociativa si 'asoc' es verdadero), se aumenta el contador. Si no se ha alcanzado el límite máximo, la matriz se agrega a la lista de matrices generadas.

Para la siguiente función es necesario introducir una función auxiliar :

La función 'comboGrid()' del paquete "RcppAlgos" permite generar todas las posibles combinaciones de elementos de una lista de vectores, en la que el orden de estos elementos no importa (ver [26]).

Con esta función, es posible generar eficientemente todas las combinaciones de filas monótonas ascendentes pasando como argumento a la función una lista de vectores donde cada vector representa los posibles valores para cada columna.

La función 'monotono_neutro()' genera matrices cuadradas de tamaño nxn, las cuales son monótonas ascendentes y tienen un elemento neutro en la posición especificada como argumento, y asociativas si se indica con un valor booleano también como argumento. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'monotono_neutro()':

```
monotono_neutro <- function(n, pos, limite, seguir, asoc)
//n -> Tamaño de la cadena y por ende de la matriz.
//pos -> Posición del elemento neutro de la matriz.
//limite -> Número máximo de matrices a guardar.
//seguir -> Valor booleano que indica si seguir contando cuando se supera el límite.
//asoc -> Valor booleano que indica si la matriz debe ser asociativa.

índices <- Índices de la matriz, excepto 'pos'.
matriz <- Matriz con elemento neutro en la posición 'pos'.
lista <- Lista para guardar las matrices.
cont <- Contador del número de matrices generadas.
l <- Combinaciones para la primera fila.

recursive_function <- function(m, matriz, combinations, cont)
    //m -> Número de fila por la que voy.
    //combinations -> Matriz con todos los valores posibles de la fila correspondiente.
    row <- Índice de la fila actual de la matriz.
    If m = n-1
        For Each comb en combinations
            Asignar los valores de comb a la fila actual.
            If asoc = FALSE o asociativa(matriz) = TRUE
                cont <- cont + 1
                If cont <= limite
                    lista.añadir(matriz)
                Else If seguir = FALSE
                    Devolver cont.
                End If
            End If
        End For
    Else
        next_row <- Índice de la siguiente fila de la matriz.
```

```
            For Each comb en combinations
                  Asignar los valores de comb a la fila 'row' de la matriz.
                  If next_row < n
                        l <- Lista con valores de cada columna para la siguiente fila teniendo
                        en cuenta que el índice es menor que el índice del elemento neutro
                        de la matriz.
                  Else
                        l <- Lista con valores de cada columna para la siguiente fila teniendo
                        en cuenta que el índice es mayor que el índice del elemento neutro
                        de la matriz.
                  End If
                  cont <- recursive_function(m+1, mat, comboGrid(l), cont)
                  If cont >= limite y seguir = FALSE
                        Devolver cont.
                  End If
            End For
      End If
      Devolver cont.
End función recursiva

l <- Lista con valores de cada columna para la primera fila.
contador <- recursive_function(1, matriz, comboGrid(l), cont)

Devolver lista y cont.
```

Tabla 13. Pseudocódigo de la función monotono_neutro().

Este método llama recursivamente a una función que recorre todas las filas de la matriz de arriba hacia abajo, excepto la fila correspondiente al elemento neutro. Para cada llamada a esta función se proporciona el número de fila y todas las combinaciones de valores válidos para esa fila.

Si el índice de la última fila a recorrer es alcanzado, se asigna a la fila actual cada una de las combinación 'comb', y si además cumple la asociatividad cuando 'asoc' es verdadero, se incrementa el contador y se añade a la lista si no se supera el límite establecido.

Para cualquier otra fila, se calcula el índice de la siguiente fila correspondiente (next_row). Posteriormente, se asigna a la fila actual cada combinación 'comb' y se calculan las combinaciones para la siguiente fila, teniendo en cuenta dos casos:

1. Si el indice 'next_row' es menor que 'pos', sabemos que los valores para cada índice 'i' de las columnas izquierdas pueden ir desde el valor correspondiente en 'comb' hasta el valor de la cadena dependiendo del índice más pequeño entre 'next_row' y 'i'.

   Los valores para cada índice 'i' de las columnas derechas pueden ir desde un valor máximo entre el valor correspondiente en 'comb' y el valor de la cadena en 'next_row' hasta el valor de la cadena en 'i'.
2. Si el índice 'next_row' es mayor que 'pos', sabemos que los valores para cada índice 'i' de las columnas izquierdas pueden ir desde el valor correspondiente en la anterior fila hasta el valor de la cadena en 'next_row'.

   Los valores para cada índice 'i' de las columnas derechas pueden ir desde un valor máximo entre el valor correspondiente en la fila anterior y el valor de la cadena en 'next_row' hasta el valor máximo de la cadena.

La función 'monotono_neutro_conm()' genera matrices cuadradas de tamaño nxn, las cuales son monótonas ascendentes, conmutativas y tienen un elemento neutro en la posición especificada como argumento. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'monotono_neutro_conm()':

```
monotono_neutro_conm <- function(n, pos, limite)
//n -> Tamaño de la cadena y por ende de la matriz.
//pos -> Posición del elemento neutro de la matriz.
//limite -> Número máximo de matrices a guardar.

índices <- Índices de la matriz, excepto 'pos'.
matriz <- Matriz con elemento neutro en la posición 'pos'.
lista <- Lista para guardar las matrices.
cont <- Contador del número de matrices generadas.

recursive_function <- function(m, matriz, combinations, cont)
    //m -> Número de fila por la que voy.
    //combinations -> Matriz con los valores posibles para la fila correspondiente.
    row <- Índice de la fila actual de la matriz.
    If m = n-1
        For Each comb en combinations
            Asignar los valores de comb a la fila actual.
            cont <- cont + 1
            If cont <= límite
                lista.añadir(matriz)
            Else If seguir = FALSE
                Devolver cont.
            End If
        End For
    Else
        next_row <- Índice de la siguiente fila de la matriz
        For Each comb en combinations
            Asignar los valores de comb a la fila actual.
            If next_row < n
                l <- Lista con valores de cada columna para la siguiente fila teniendo
                en cuenta que el índice es menor que el índice del elemento neutro
```

```
                    de la matriz y las columnas con índice menor a next_row no pueden
                    cambiar.
                Else
                    l <- Lista con valores de cada columna para la siguiente fila teniendo
                    en cuenta que el índice es más alto que el índice del elemento
                    neutro de la matriz las columnas con índice menor a next_row no
                    pueden cambiar.
                End If
                cont <- recursive_function(m+1, mat, comboGrid(l), cont)
                If cont >= limite y seguir = FALSE
                    Devolver cont.
                End If
            End For
        End If
        Devolver cont.
End función recursiva

l <- Lista con valores de cada columna para la primera fila.
cont <- recursive_function(1, mat, l, cont)

Devolver lista y cont.
```
Tabla 14. Pseudocódigo de la función monotono_neutro_conm().

Esta función se asemeja a la función monotono_neutro(), con la diferencia de que no incluye el argumento 'asoc' y el cálculo de las combinaciones de las filas se simplifica, debido a que los valores de las columnas a la izquierda de la diagonal ya están fijados por la propiedad conmutativa.

La función 'tnormas()' genera matrices cuadradas de tamaño nxn, las cuales son monótonas ascendentes, conmutativas, asociativas y tienen un elemento neutro en la posición 'n'. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'tnormas()':

```
tnormas <- function(n)
//n -> Tamaño de la cadena y por ende de la matriz.

lista <- Lista de t-normas de tamaño (n-1)x(n-1) pero eliminando su primera y última fila/columna.
índices <-  Índices de la matriz en orden decreciente.
bandera <- Valor para indicar si la matriz ha sido ya modificada y si es válida (inicial a 0).
lista <- Lista para guardar las matrices.
cont <- Contador del número de matrices generadas.
fin <- Valor para indicar si se debe salir del bucle (inicial a 0).

For Each M en lista
    P <- Matriz M agregando una nueva fila y columna, que son iguales a la última fila y columna de M. El último valor de la diagonal de P es igual al de M decrementado en 1.
    fin <- 0
    While fin = 0
        bandera <- 0
        For Each i en índices
            If bandera = 0
                If P[i, n] < i
                    P[i, n] <- P[i, n] +1
                    For f = {i, …, n}
                        P[f, n] <- Máximo valor entre P[i, n] y P[f, n-1]
                        P[n, f] <- P[f, n]
                    End For
                    Bandera <- 2
                End If
            End If
        End For
        If bandera = 2
            For i = {1, …, n-1} Hacer
```

```
                    If bandera = 2
                        For k = {1, ..., n}
                            If P[k, n] ≠ 0 y P[i, k] ≠ 0 y P[P[i, k], n] ≠ P[i, P[k, n]]
                                Bandera <- 3
                                Terminar comprobación, P no es asociativa.
                            End If
                        End For
                    End If
                End For
            End If
            If bandera = 2
                cont <- cont + 1
                lista.añadir(P)
            Else If bandera = 0
                fin <- 1
            End If
        End While
End For
Devolver lista.
```

Tabla 15. Pseudocódigo de la función tnormas().

En caso de contar con una t-norma definida en la cadena {0,1,2,...,n-1}, al eliminar el elemento 'n-2' de dicha cadena (suprimiendo su fila y columna correspondientes), se puede observar que la matriz resultante también es una t-norma en la cadena {0,1,2,...,n-1} \ {n-2}.

De esta manera, se pueden calcular las t-normas nxn usando las t-normas (n-1)x(n-1) al completar únicamente la penúltima fila/columna con los valores correspondientes y comprobar que se cumpla la asociatividad.

Una t-norma de tamaño nxn tiene sus valores fijados en la primera y última fila/columna de su matriz, por lo que solo es necesario considerar la submatriz resultante de eliminar ambas filas/columnas, que tendrá un tamaño de (n-2)x(n-2).

Este método iterativo recorre la última columna de la matriz P de abajo hacia arriba a partir de cada t-norma de tamaño (n-1)x(n-1).

Para cada celda en la matriz, si la matriz no ha sido modificada previamente y el valor de la celda es menor que el valor máximo permitido en esa fila, se incrementa el valor en dicha celda (v) en 1.

Para mantener la monotonía, se asigna a cada una de las filas posteriores 'f' con el valor máximo entre 'v' y el valor de la celda en la fila 'f' y columna anterior, y simétricamente para garantizar la conmutatividad de la matriz.

Si se ha modificado la matriz y cumple la asociatividad, se aumenta el contador y se añade la matriz a la lista.

La función 'uninormas()' genera matrices cuadradas de tamaño nxn, las cuales son monótonas ascendentes, conmutativas, asociativas y tienen elemento neutro en una posición distinta a 1 y n. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'uninormas()':

```
uninormas <- function(n, pos)
//n -> Tamaño de la cadena y por ende de la matriz.
//pos -> Posición del elemento neutro de la matriz.

matriz <- Matriz con elemento neutro en la posición 'pos'.
lista1 <- Lista de t-normas correspondiente.
lista2 <- Lista de t-conormas correspondiente.
lista3 <- Lista de combinaciones de la esquina derecha superior.

For Each m1 en lista1
    Asignar m1 a la esquina izquierda superior de la matriz.
    For Each m2 en lista2
        Asignar m2 a la esquina derecha inferior de la matriz sumando un valor de
        umbral para que la matriz siga cumpliendo la monotonía.
        For Each m3 en lista3
```

```
                Asignar m3 a la esquina derecha superior de la matriz. Hacer lo mismo en
                la esquina inferior izquierda pero transponiendo m3.
                If asociativa_conm(..., matriz) = TRUE
                    cont <- cont +1
                    If cont <= limite
                        lista.añadir(matriz)
                    End If
                End If
            End For
        End For
End For
Devolver lista y cont.
```

Tabla 16. Pseudocódigo de la función uninormas().

Este método iterativo construye las uninormas de tamaño nxn haciendo uso de las t-normas de tamaño posxpos para la esquina superior izquierda, las t-conormas de tamaño (n-pos+1)x(n-pos+1) para la esquina inferior derecha y las combinaciones válidas de matrices para la esquina superior derecha de la matriz.

Se itera sobre cada t-norma en lista1, luego sobre cada t-conorma en lista2 y finalmente sobre cada matriz en lista3, transponiendo su valor para la esquina izquierda inferior de la matriz por la propiedad de conmutatividad. Si la matriz resultante es asociativa, se incrementa el contador en 1 y si este contador no supera el límite establecido, se añade la matriz a la lista.

Sin embargo, para generar todas las posibles combinaciones para la esquina superior derecha se hace uso de otra función implementada.

La función completar() devuelve todas las posibles combinaciones de la esquina superior derecha de matrices nxn monótonas y con índice del elemento neutro menor a n. La siguiente tabla muestra el pseudocódigo que describe el algoritmo que sigue la función 'completar()':

```
completar <- function(n, pos)
//n -> Tamaño de la cadena y por ende de la matriz.
//pos -> Índice del elemento neutro.
```

```
n_col <- Número de columnas.
n_fila <- Número de filas.
matriz <- Matriz de tamaño n_fila x n_col.
max_valores <- Lista con el máximo valor que se puede asignar a cada columna.
cont <- Contador del número de matrices.
lista <- Lista para guardar las matrices.

recursive_function <- function(fila, combinations, matriz, cont)
    //combinations -> Combinaciones para la fila indicada por 'fila'.
    If fila = n_fila
        For Each comb en combinations
            Asignar los valores de comb a la última fila.
            cont <- cont+1
            lista.añadir(matriz)
        End For
    Else If
        For Each comb en combinations
            Asignar los valores de comb a la fila actual.
            l <- Lista con valores de cada columna para la siguiente fila. Tener en
            cuenta el valor mínimo de la fila siguiente y el valor máximo de cada
            columna.
            cont <- recursivo(fila+1, comboGrid(l), matriz, cont)
        End For
    End If
    Devolver cont.
End función recursiva

l <- Lista con valores de cada columna para la primera fila.
cont <- recursivo(1, comboGrid(l), matriz, cont)
```

Devolver lista y cont.

Tabla 17. Pseudocódigo de la función completar().

Este método recursivo recorre todas las filas de la matriz de arriba hacia abajo. Para cada llamada a esta función se proporciona el índice de fila y todas las combinaciones de valores válidos para esa fila.

Si el índice de la fila a recorrer es el último, se asigna la fila actual con cada combinación 'comb', se incrementa el contador y se añade a la lista.

Para cualquier otra fila, se asigna la fila actual con cada combinación 'comb' y se calculan las combinaciones para la siguiente fila. En este caso, es importante tener en cuenta que el valor mínimo para cada columna 'j' debe ser el máximo entre 'fila' y la celda superior en la columna 'j', mientras que el valor máximo se puede obtener a partir de 'max_valores' y 'j'.

6. Conclusiones y líneas futuras

En este trabajo, se ha estudiado la problemática de encontrar una forma eficiente de calcular operadores discretos en el ámbito de la lógica difusa, centrándose en los operadores definidos sobre una escala discreta que modelan la conjunción lógica. Para abordar este problema, se ha desarrollado una aplicación Shiny que cuenta con las siguientes utilidades:

1. Es una herramienta para aquellos usuarios que no están muy familiarizados con la lógica difusa o la programación, ya que está diseñada para ser intuitiva y fácil de usar. Además, la aplicación es de utilidad para aquellos investigadores acostumbrados en el campo de la lógica difusa, ya que les permite trabajar con diferentes operadores de manera rápida y eficiente.
2. La aplicación es especialmente útil para determinar el número de operadores difusos de un conjunto que cumplan ciertas restricciones e incluso brinda la opción de descargar los operadores calculados para su posterior utilización o análisis visual.
3. La aplicación es de gran ayuda para comparar diferentes operadores difusos utilizando la propiedad de dominancia, la cual es relevante en el contexto de la lógica difusa. Del mismo modo, ofrece la posibilidad de comparar la dominancia de un conjunto de operadores en relación a un operador específico.
4. La aplicación también permite verificar propiedades de cualquier operador difuso que se introduzca, lo que agiliza la tarea de identificar y seleccionar el operador difuso más adecuado para un problema específico.

En resumen, se trata de una herramienta versátil y fácil de usar que puede ser de gran utilidad para generar, analizar, clasificar y comparar diferentes operadores en el ámbito de la lógica difusa.

Como líneas futuras, está prevista la elaboración de un artículo científico en la categoría de *Computer Sciences* donde se realice un estudio más detallado de la propiedad de dominancia sobre operadores difusos. Además, se pretende poner a disposición este software a las comunidades científicas de interés como la European Society For Fuzzy Logic And Technology (EUSFLAT), congreso Español Sobre Tecnologías Y Lógica Fuzzy (ESTYLF), entre otras. No

obstante, son sólo algunas de las posibles opciones que se pueden llevar a cabo, ya que existen diversas vías de investigación en las que este software podría intervenir, como aquellas donde encontrar operadores discretos sea difícil, la determinación de propiedades de los operadores o el cálculo del número de operadores de un conjunto con ciertas restricciones. Por otro lado, existe la posibilidad de extender la funcionalidad de la aplicación añadiendo, por ejemplo, nuevas propiedades a las ya indicadas en la memoria.

Bibliografía

[1] Zadeh, L.A. (1965). Fuzzy Sets. Information and Control, 8, 338-353. 10.1016/S0019-9958(65)90241-X.

[2] De Baets, B., & Mesiar, R. (1999). Triangular norms on product lattices. Fuzzy Sets and Systems. 104. 61-75. 10.1016/S0165-0114(98)00259-0.

[3] Bejines, C., & Navara, M. (2022). The Fibonacci sequence in the description of maximal discrete Archimedean t-norms. Fuzzy Sets and Systems. 451. 10.1016/j.fss.2022.08.012.

[4] Munar Covas, M., Massanet, S., Ruiz-Aguilera, D. (2022). On the cardinality of some families of discrete connectives. Information Sciences. 621. 10.1016/j.ins.2022.10.121.

[5] Bejines, C., & Ojeda-Hernández, M. (2022). Counting semicopulas on finite structures. Fuzzy Sets and Systems. 10.1016/j.fss.2022.09.011.

[6] Scrum.org. What is Scrum? Recuperado el 7 de abril de 2023, de https://www.scrum.org/learning-series/what-is-scrum.

[7] Wikipedia. R (lenguaje de programación). Recuperado el 6 de abril de 2023, de https://es.wikipedia.org/wiki/R_(lenguaje_de_programación).

[8] Chacon, S., & Straub, B. (2014). Pro Git (2nd ed.). Apress. Recuperado el 7 de abril de 2023, de https://git-scm.com/book/en/v2.

[9] RStudio Team. (2022). RStudio: Integrated Development Environment for R. Recuperado el 6 de abril de 2023, de https://rstudio.com/products/rstudio/.

[10] TeachDataScience. (2021). Shiny: An R package for building interactive web applications. Recuperado el 20 de abril de 2023, de https://teachdatascience.com/shiny1/.

[11] Shiny. (2017). Reactivity - An overview. Recuperado el 20 de abril de 2023, de https://shiny.posit.co/r/articles/build/reactivity-overview/.

[12] Chang, W., Cheng, J., Allaire, J., Sievert, C., Schloerke, B., Xie, Y., Allen, J., McPherson, J., Dipert, A., Borges, B. (2022). shiny: Web Application Framework for R. https://CRAN.R-project.org/package=shiny.

[13] MathJax Consortium. (s.f.). MathJax. Recuperado el 6 de abril de 2023, de https://www.mathjax.org/.

[14] RStudio. (2015). withMathJax function. Recuperado el 6 de abril de 2023, de https://shiny.rstudio.com/reference/shiny/0.11/withmathjax/.

[15] Chang, W. (2020). Modularizing Shiny app code. Recuperado el 20 de abril de 2023, de https://shiny.posit.co/r/articles/improve/modules/.

[16] Chang, W., Borges Ribeiro, B. (2021). shinydashboard: Create Dashboards with 'Shiny'. https://CRAN.R-project.org/package=shinydashboard.

[17] Perrier, V., Meyer, F., Granjon, D. (2022). shinyWidgets: Custom Inputs Widgets for Shiny. https://cran.r-project.org/package=shinyWidgets.

[18] Lyttle, I. (2022). bsplus: Adds Functionality to the R Markdown + Shiny Bootstrap Framework. https://CRAN.R-project.org/package=bsplus.

[19] Neudecker, A. (2021). shinyMatrix: Shiny Matrix Input Field. https://CRAN.R-project.org/package=shinyMatrix.

[20] Sali, A., Attali, D. (2020). shinycssloaders: Add Loading Animations to a 'shiny' Output While It's Recalculating. https://CRAN.R-project.org/package=shinycssloaders

[21] Iannone, R., Cheng, J. (2022). shinyvalidate: Input Validation for Shiny Apps. https://CRAN.R-project.org/package=shinyvalidate

[22] Attali, D., Edwards, T. (2021). shinyalert: Easily Create Pretty Popup Messages (Modals) in Shiny. https://CRAN.R-project.org/package=shinyalert

[23] Krzemiński, D., Igras, K. (2020). shiny.i18n: Shiny Applications Internationalization. https://CRAN.R-project.org/package=shiny.i18n

[24] Attali, D. (2021). shinyjs: Easily Improve the User Experience of Your Shiny Apps in Seconds. https://CRAN.R-project.org/package=shinyjs

[25] Abdali, S. K. (1970). Verification of Associativity of a Binary Operation. The Mathematical Gazette, 54(390), 372-374. 10.2307/3613856.

[26] Wood, J. (2022). comboGrid: Grid Generation for Combinatorial Problems. RcppAlgos. Recuperado el 30 de abril de 2023, de https://jwood000.github.io/RcppAlgos/-reference/comboGrid.html

Apéndice A. Manual de Instalación

Dentro de este apéndice se describirá el proceso de instalación de R, RStudio y las librerías externas necesarias para el correcto funcionamiento de la aplicación.

Para la realización de este proyecto, se ha utilizado la versión '4.2.2' (2022-10-31) de R y la versión '2022.7.2.576' de RStudio.

Descargar R

La descarga de R puede llevarse a cabo desde la página web oficial https://cran.rstudio.com/. A partir de allí, se debe selecciónar la versión correspondiente a nuestro sistema operativo (Linux, macOS, Windows) y realizar los pasos indicados para su correcta instalación.

Descargar RStudio

Para descargar R Studio, es necesario acceder a la página web oficial https://posit.co/download/rstudio-desktop/. A partir de allí, se debe descargar el ejecutable correspondiente a nuestro sistema operativo.

Cabe mencionar que, en función de la política de seguridad del sistema operativo, los usuarios de Linux pueden necesitar importar la clave pública de firma de código de Posit antes de proceder con la instalación. Para más infromación al respecto, puede visitar https://posit.co/code-signing/.

Una vez instalado y abierto RStudio, vemos como la pantalla está dividida en tres secciones:

1. A la izquierda está la consola, donde se ejecutan los comandos de R.
2. En la parte superior derecha, encontramos el panel que muestra nuestro entorno de trabajo.
3. En la parte inferior derecha, encontramos el panel de utilidades con dos pestañas de interés. La pestaña 'Files' muestra el contenido del directorio donde R se iniciará,

mientras que la pestaña 'Packages' permite ver qué librerías de R tenemos instaladas y permite instalar nuevas librerías.

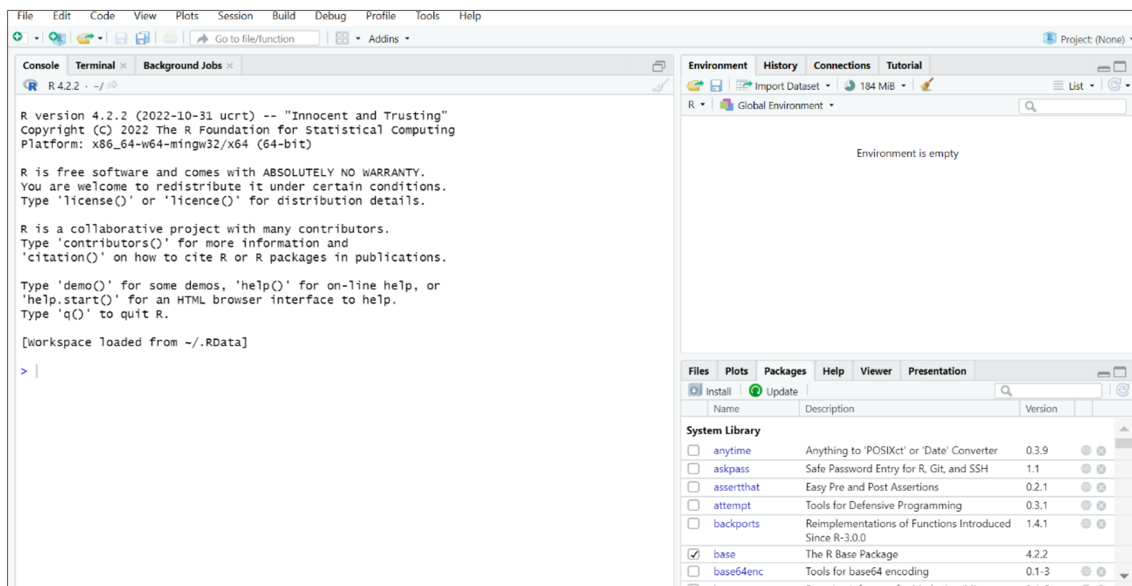

Figura A.1. Interfaz de RStudio.

<u>Descargar paquetes necesarias</u>

Para instalar un paquete en R, hay dos opciones; a través de la consola de R o mediante la interfaz de RStudio:

1. En la consola, solo es necesario escribir el comando "install.packages('nombre_del_paquete')". Por ejemplo, para instalar el paquete shiny debemos escribir "install.packages('shiny')" .
2. Desde el panel de utilidades tenemos que seguir los siguientes pasos:

   2.1. Seleccionar la pestaña "Packages".

   2.2. Hacer clic en "Install" para abrir la ventana de instalación de paquetes.

   2.3. Comprobar que en "Install from:" esté seleccionado "Repository (CRAN)".

   2.4. Escribir el nombre del paquete que se desea instalar.

   2.5. Hacer clic en "Install" para iniciar la instalación.

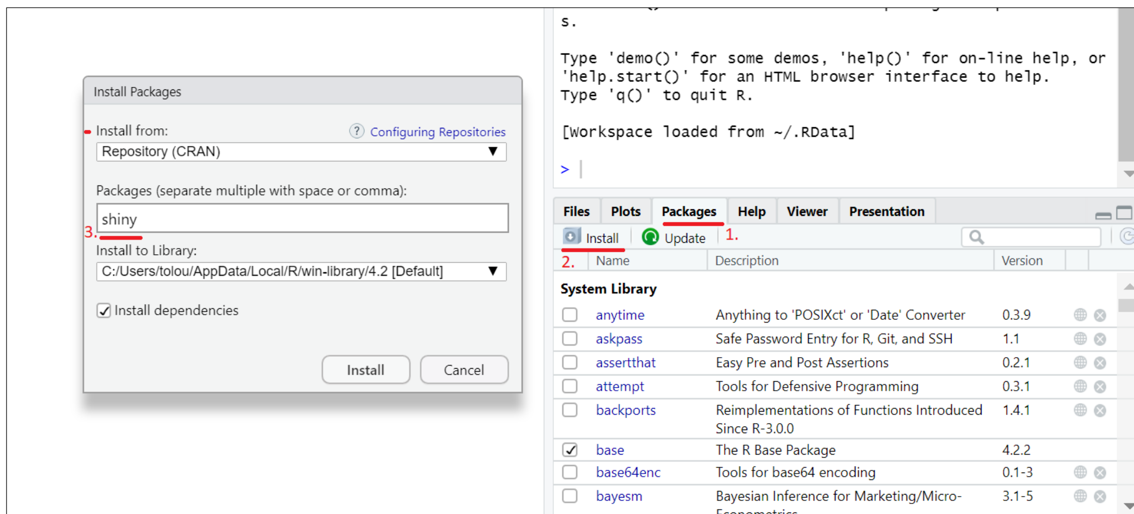

Figura A.2. Ejemplo de la segunda opción para descargar un paquete R.

Para poder cargar las librerías hay que utilizar el comando library("nombre_del_paquete") en la consola.

Abrir aplicación

Una vez instalado todo lo necesario y descargado el proyecto desde https://github.com/FrancoGarciaDosSantos/TFG-UMA2023.git, lo descomprimimos en la ubicación que prefiramos.

Luego, abrimos la carpeta 'App' y hacemos doble clic sobre el archivo 'app.R'. Para iniciar la aplicación en RStudio, tenemos dos opciones:

1. Hacer clic en "Run App" en la parte superior derecha de la ventana del editor de código. Se abrirá la ventana de la aplicación.

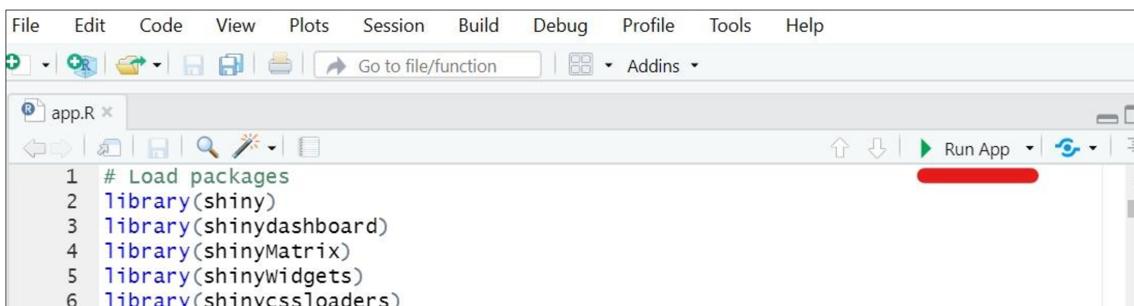

Figura A.3. Ejemplo de la primera opción para iniciar la aplicación Shiny.

2. Escribir el comando "shiny::runApp()" en la consola.

Apéndice B. Funciones más relevantes de paquetes que mejoran Shiny.

Dentro de este apéndice se presentan las funciones relevantes de los paquetes mencionados en la sección 3.4.

shinyWidgets

1. **actionBttn():** Es una alternativa al actionButton predeterminado de Shiny, que te permite personalizarlo con diferentes opciones de tamaño, color, estilo, entre otros.
2. **downloadBttn():** Permite crear botones de descarga utilizando la función actionBttn en lugar del downloadButton predeterminado de Shiny.
3. **dropdown(), dropdownButton(), dropMenu():** Permiten crear menús desplegables que muestran/ocultan los elementos de entrada en su interior a través de un botón personalizable. Además, dropdownButton y dropMenu te permiten usar Bootstrap/CSS para personalizar aún más la apariencia del menú.
4. **textInputIcon(), numericInputIcon():** Son una extensión de los inputs clásicos de Shiny, textInput() y numericInput(), permitiendo la adición de texto y/o iconos antes, después o a ambos lados del input.
5. **autonumericInput():** Es una función wrapper sobre la biblioteca 'AutoNumeric' de JavaScript que extiende la función numericInput() de Shiny. Esta agrega nuevas características como la capacidad de formatear automáticamente el número de entrada y personalizar el comportamiento de la entrada numérica.
6. **pickerInput():** Es una alternativa a selectInput que ofrece numerosas opciones de personalización. Permite la agrupación de valores en grupos, la asociación de iconos a cada uno de ellos, la inclusión de una barra de búsqueda y otras opciones de configuración.
7. **materialSwitch(), prettySwitch(), switchInput():** Permiten crear un toggle switch para activar o desactivar una selección, y reemplazan a un checkbox. Cada una de las funciones proporciona distintas formas de personalización.

8. **awesomeCheckbox(), prettyCheckBox():** Permiten crear un checkbox elegante con distintas formas de personalización. XXXGroup permite crear un conjunto de checkboxes (especificar valores lógicos).
9. **awesomeRadio(), prettyRadioButtons(), radioGroupButtons():** Son funciones que permiten crear conjuntos de radio buttons con un estilo atractivo para seleccionar un elemento de una lista. La diferencia con la función radioGroupButtons es que esta crea un conjunto de botones que actúan como radio buttons.

bsplus

1. **bs_embed_tooltip():** Esta función añade un 'tooltip' a un elemento HTML. Un tooltip es una forma útil y fácil de añadir información adicional al usuario que se muestra al pasar el cursor sobre el elemento.
2. **use_bs_tooltip():** Es necesario llamar a esta función en la UI para activar el uso de tooltips en la aplicación.
3. **bs_embed_popover():** Esta función añade un 'popover' a un elemento HTML. Un popover es una forma útil de añadir una explicación algo más extensa al usuario que se muestra como una ventana al hacer clic sobre el elemento.
4. **use_bs_popover():** Es necesario llamar a esta función en la UI para activar el uso de popovers en la aplicación.
5. **bs_collapse():** Esta función permite crear un panel colapsable en la interfaz de usuario. Es muy útil, ya que permite mostrar u ocultar contenido de forma dinámica interactuando con un botón o enlace que active el componente según las necesidades del usuario.
6. **bs_attach_collapse():** Esta función permite unir una etiqueta HTML, como un botón o un enlace, con un panel colapsable específico.
7. **shiny_iconlink():** Esta función permite crear un enlace alrededor de un icono de Shiny. Es muy útil para adjuntar en él un panel colapsable, un popover o un tooltip.
8. **shinyInput_label_embed():** Esta función incrusta un elemento en la etiqueta de un input de Shiny. El elemento se desplazará hacia el borde derecho de la etiqueta.

shinyMatrix

1. **matrixInput():** Esta función crea un campo de entrada de una matriz, típicamente utilizado en la parte de la UI. Se puede acceder a su valor a través de 'input$inputId' en la parte del servidor, cuya clase siempre será una matriz y contendrá valores de la clase que se especifique. Se pueden especificar otras opciones como mostrar y editar los nombres a las filas y columnas, paginar la matriz y personalizarla. La matriz soporta salto de línea y tabulación.
2. **updateMatrixInput():** Esta función actualiza la matriz de entrada creada con 'matrixInput' con otra matriz de la misma clase, sin necesidad de tener el mismo tamaño.

shinycssloaders

1. **withSpinner():** Añade un spinner que aparece cuando alguna salida envuelta por la función se está recalculando. Se puede elegir entre 8 tipos de spinner, permitiendo cambiar el color, el tamaño y el fondo del spinner, entre otras personalizaciones.

shinyvalidate

1. **InputValidator:** Un objeto 'InputValidator' que sirve para añadir validaciones en tiempo real a las aplicaciones Shiny. Están diseñados para ser creados como variables locales en la parte del servidor Shiny, pudiendo crear una o múltiples reglas de validación para cada campo de entrada en la UI. Los métodos públicos de nuestro interés son los siguientes:
   - **1.1. new():** Crea un nuevo objeto InputValidator.
   - **1.2. add_rule():** Añade una regla de validación a una sola entrada.
   - **1.3. add_validator():** Añade otro objeto InputValidator como hijo. Este objeto devolverá TRUE si sus reglas y sus hijos InputValidator son válidos.
   - **1.4. condition():** Permite establecer una condición que se ejecutará antes de realizar la validación. Si la condición devuelve TRUE, la validación continua con normalidad; si devuelve FALSE, las reglas de validación se omiten y se consideran que todas son correctas.
   - **1.5. enable():** Empieza a proporcionar las validaciones a todas las entradas en la UI. Si este objeto validador es hijo de otro objeto validador, enable() debe llamarse desde el objeto padre.

1.6. **disable():** Elimina y deja de proporcionar las validaciones existentes en la UI para todas las entradas que están presentes en el conjunto de reglas del objeto validador.

1.7. **is_valid():** Devuelve TRUE si todas las reglas de validación pasan, y FALSE en caso contrario.

2. **input_provided():** Esta función toma un valor de entrada y utiliza la heurística para adivinar si el valor representa una entrada 'vacía' frente a una que haya proporcionado el usuario, que variará según el tipo de entrada.
3. **sv_required():** Genera una función de validación que asegura que un valor de entrada está presente. Requiere un mensaje que muestra en caso de error de validación.
4. Otras funciones similares para reglas de validación: **sv_between()**, **sv_equal()**, **sv_gte()**, **sv_gt()**, **sv_in_set()**, **sv_integer()**, **sv_lte()**, **sv_lt()**, **sv_not_equal()**, **sv_numeric()**, **sv_regex()**.

shinyalert

1. **shinyAlert():** Función que permite crear alertas alertas que pueden ser personalizadas con diferentes tipos de botones, texto, colores, tamaño, etc.

shiny.i18n

El paquete shiny.i18n ofrece dos formatos de archivo de traducción, CSV y JSON. El utilizado en la aplicación es el formato JSON, que permite mantener cómodamente todas las traducciones en un solo lugar.

El archivo JSON 'translation.json' contiene los siguientes dos campos obligatorios: "languages" y "translation".

1. languages. Contiene la lista de códigos de los idiomas. Por ejemplo, es bien conocido el código "en" para referirse al idioma inglés (english). El primer idioma en la lista es llamado la 'key translation', utilizado para referenciar al consultar las traducciones de los textos.
2. translation. Contiene una lista de diccionarios. Cada diccionario asigna a cada código de idioma el texto en dicho idioma.

A continuación se presenta las funciones principal del paquete:

1. **Translator:** Un objeto Translator utilizado para las traducciones. Los métodos públicos de nuestro interés son los siguientes:
    1.1. **new():** Inicializa el objeto con los datos de traducción en la ruta especificada.
    1.2. **get_languages():** Devuelve todos los idiomas disponibles.
    1.3. **get_key_translation():** Devuelve la 'key translation' activa.
    1.4. **translate() / t():** Traduce una palabra o vector de palabras al idioma especificado por ’set_translation_language’.
    1.5. **set_translation_language():** Permite especificar el idioma de la traducción (debe existir en el campo 'languages' del archivo).
2. **update_lang():** Esta función envía un mensaje al objeto sesión del usuario para actualizar el idioma de los elementos de la UI al código de idioma especificado.
3. **usei18n():** Esta función es necesaria para indicar a la UI que monitorice las traducciones de idiomas en directo.

shinyjs

1. **click():** Esta función permite simular de manera programática un clic en un actionButton de Shiny.
2. **onclick(), onevent():** onclick ejecuta una expresión R, ya sea una función shinyjs o cualquier otro código, cuando un elemento es clicado. onevent funciona igual, pero puede usarse con cualquier otro evento en el elemento.
3. **delay():** Esta función permite ejecutar código R una vez transcurrido cierta cantidad de tiempo. Puede usarse en combinación con otras funciones de shinyjs.
4. **show(), hide(), toggle():** show hace visible un elemento, ocultar lo hace invisible, y toggle, dada una condición, si es TRUE muestra el elemento y si es FALSE lo oculta.
5. **enable(), disable(), toggleState():** enable habilita una elemento Shiny, disable lo deshabilita y toggleState, dada una condición, si es TRUE habilita el elemento si está deshabilitado y si es FALSE lo deshabilita si está habilitado.
6. **runjs():** Ejecuta código JavaScript.
7. **hidden(), disabled():** hidden/disabled crea un elemento Shiny que permanece oculto/deshabilitado cuando se inicia la aplicación.

8. **useShinyjs():** Esta función debe ser llamada desde la UI para que el resto de funciones de shinyjs funcionen.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática

Bulevar Louis Pasteur, 35

Campus de Teatinos