Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Licenciatura en ciencias de la computación

Tesina

Optimización de algoritmos para grafos basados en conjuntos

Cavagna, Lucas Gastón (Estudiante)

Kofman, Ernesto (Director)

Marzorati, Denise (Codirectora)

Septiembre 2025

Agradecimientos

Antes de adentrarme en el desarrollo del contenido de esta tesina, deseo expresar mis más sinceros y profundos agradecimientos a mi director, Ernesto Kofman, y a mi codirectora, Denise Marsorti. Ambos me brindaron un apoyo invaluable, no solo desde lo académico, sino también desde lo humano, acompañándome de manera constante a lo largo de todo el proceso de realización de este trabajo.

Asimismo, deseo expresar mi gratitud a todos los profesores que, a lo largo de mi formación en esta licenciatura, me han transmitido sus conocimientos con dedicación y vocación, orientándome en cada etapa de este recorrido académico. Del mismo modo, agradezco profundamente tanto a mi familia como a todas aquellas personas que, de una u otra manera, me han brindado su apoyo incondicional durante estos años, acompañándome tanto en lo académico como en lo personal.

Índice general

Capítulo 1

Introducción

1.1. Estado del arte

1.1.1. Motivación

Con el objetivo de unificar los distintos lenguajes empleados en herramientas de modelado y simulación, un consorcio integrado por compañías de software y grupos de investigación propuso un lenguaje de modelado abierto, unificado y orientado a objetos: **Modelica** [5] [4].

Este lenguaje permite la representación de sistemas de tiempo continuo, tiempo discreto, eventos discretos y modelos híbridos. Los modelos elementales en Modelica se expresan mediante conjuntos de ecuaciones diferenciales y algebraicas. A su vez, estos modelos pueden interconectarse, lo que facilita el desarrollo de modelos más complejos y reutilizables.

Los compiladores del lenguaje Modelica tienen la tarea de transformar un modelo orientado a objetos en código de simulación. Este proceso implica varias etapas, en las cuales se produce la identificación y el ordenamiento de las ecuaciones que definen el comportamiento del modelo.

Hacia el año 2015, las principales herramientas de compilación de modelos Modelica existentes [6] [3] [10] no aprovechaban adecuadamente ciertas características relacionadas con los arreglos y las ecuaciones for. En particular, estas ecuaciones eran expandidas y la vectorización era eliminada desde las primeras etapas del proceso de compilación, lo cual afectaba significativamente el rendimiento del compilador. Para abordar este problema, se desarrolló un algoritmo de aplanado [1] que preservaba la vectorización de los modelos, manteniendo tanto los arreglos como las ecuaciones for. Dicho algoritmo calcula las componentes conexas en un grafo vectorizado, con el objetivo de reemplazar correctamente las conexiones presentes en los modelos por sus correspondientes ecuaciones.

En 2019, se presentaron nuevos algoritmos diseñados para convertir de forma eficiente grandes sistemas de Ecuaciones Diferenciales Algebraicas (DAEs) en Ecuaciones Diferenciales Ordinarias (ODEs) [11]. Estos algoritmos se basan en procedimientos ya conocidos tales como el de Edmonds-Karp para maximum matching, o el de Tarjan para componentes fuertemente conexas, apoyándose en un concepto innovador denominado Grafo Basado en Conjuntos (Set-Based Graph o SBG). Una de sus principales ventajas es que permiten representar los sistemas de ecuaciones sin necesidad de expandir los arreglos de incógnitas y las ecuaciones tipo for, lo que hace que la complejidad del algoritmo sea independiente del tamaño de los arreglos. Es en este contexto que se introduce por primera vez la biblioteca SBG en ModelicaCC.

Finalmente, en el año 2020, se rediseño la librería SBG para ModelicaCC [9], incorporando una amplia variedad de definiciones, tales como intervalos, multi-intervalos, conjuntos y mapas lineales. Estos avances estuvieron orientados al desarrollo e implementación de un algoritmo para la detección de componentes conexas, que forma parte de la primera etapa de conversión de modelos orientados a objetos en sistemas de ecuaciones.

Posteriormente, en 2022, la librería fue refinada en el marco del desarrollo de un nuevo trabajo, donde se convirtió en un nuevo proyecto independiente de ModelicaCC [7]. Así, finalmente, se llega al estado actual de

la misma con los últimos cambios propuestos en un escrito de 2024 [8].

1.1.2. Objetivo general

Actualmente, dentro de la librería SBG se emplean conjuntos *compactos*. El término *compacto* hace referencia a la forma en que estos conjuntos representan los elementos que contienen: mediante un número acotado de valores es posible describir una colección mucho más amplia. Los conjuntos compactos están conformados por una colección de multi-intervalos de igual dimensión. Cada multi-intervalo se compone de una colección de intervalos individuales, donde dicha cantidad define la dimensión del multi-intervalo. Estos intervalos representan subconjuntos de números naturales mediante una tripleta que especifica sus elementos: valor de inicio, paso y valor final. Mientras, los elementos de cada multi-intervalo son los que pertenecen al producto cartesiano de los intervalos que lo componen.

Bajo estas definiciones, existen actualmente dos implementaciones disponibles para conjuntos: *unordered sets* (conjuntos desordenados) y *ordered dense sets* (conjuntos ordenados densos). Esta última implementación contempla el orden únicamente en aquellos casos en los que los conjuntos están constituidos por multi-intervalos unidimensionales, compuestos por intervalos con paso igual a uno.

Por otro lado, se encuentran los *piecewise maps*, que se definen como una colección de mapas. Cada mapa está compuesto por un conjunto y una expresión lineal multi-dimensional. En este contexto, cada expresión lineal se asocia a una dimensión del conjunto. De esta manera, el dominio de cada expresión queda definido por la dimensión correspondiente del conjunto.

En el caso de los *piecewise maps*, actualmente solo se encuentra disponible una única implementación: *unordered piecewise maps* (*piecewise maps* desordenados). Esto significa que, hasta el momento, no existe una versión que aproveche el orden en ningún nivel dentro de los *piecewise maps*.

Uno de los objetivos de este trabajo, desde una perspectiva más general, es el desarrollo de una implementación eficiente de conjuntos ordenados que pueda manejar mas de una dimensión y un paso igual o distinto a uno. Esta implementación tiene como meta que las operaciones definidas sobre dichos conjuntos alcancen un rendimiento comparable al de la implementación existente para conjuntos ordenados densos, y que superen el desempeño observado en los conjuntos desordenados. A su vez, se persigue que dichas operaciones escalen, en la medida de lo posible, de manera lineal o lo mas cercano posible respecto a la cantidad de multi-intervalos que componen cada conjunto.

Adicionalmente, otro objetivo relevante consiste en diseñar una versión ordenada de los *piecewise maps*, que permita mejorar el rendimiento en comparación con la implementación desordenada disponible actualmente.

1.2. Organización de la tesina

Este escrito se divide en varios capítulos, cada uno con la tarea de explicar y exponer diferentes conceptos. En particular, los mismos son:

1. **Introducción:** El capítulo actual introduce los objetivos de la tesina y desde dónde se empezó a componer la misma.

2. **Conceptos Preliminares:** Capítulo que expondrá todos los conceptos base necesarios para entender por completo lo analizado en esta tesina.

3. **Optimizaciones para el desorden:** Este capítulo se centra únicamente en algunas mejoras algorítmicas aplicadas a ciertas funciones que no dependen del orden, pero que resulta importante mencionar ya que favorecieron significativamente el rendimiento de las operaciones en general.

4. **Conjuntos ordenados:** En este capitulo se describirá la estructura interna de los conjuntos ordenados, junto con el criterio de orden que se les asigna. Además, se presentarán de manera detallada los criterios de optimización y de ordenamiento aplicables a las operaciones de dichos conjuntos, explicando cada uno en profundidad. También se incluirán las funciones necesarias para el correcto funcionamiento de los

conjuntos ordenados, que si bien no forman parte del núcleo principal de sus métodos, resultan esenciales para su implementación. Finalmente, se expondrán los métodos específicos de los conjuntos ordenados, en base a las optimizaciones y a su representación interna.

5. ***Piecewise maps* ordenados:** En este capítulo se describirá la estructura interna de los *piecewise maps* ordenados, junto con el criterio de orden que se les asigna. Asimismo, se presentarán en detalle los criterios de optimización y de ordenamiento aplicables a las operaciones de dichos *piecewise maps*, explicando cada uno en profundidad. También se incluirán las funciones necesarias para su correcto funcionamiento, que si bien no forman parte del núcleo principal de sus métodos, resultan esenciales para su implementación. Finalmente, se expondrán los métodos específicos de los *piecewise maps* ordenados, fundamentados en las optimizaciones y en su representación interna.

6. **Resultados:** Aquí se presentarán múltiples casos de prueba y casos de prueba sintéticas para evidenciar la mejora que aportan las dos implementaciones basadas en el orden propuestas en este escrito.

7. **Conclusiones finales y trabajos futuros:** En este último capítulo se presentan las conclusiones finales de la tesina, teniendo en cuenta los resultados obtenidos en el capítulo anterior. Así como tambien posibles trabajos futuros en base a lo visto durante el desarrollo de este escrito.

Capítulo 2

Conceptos Preliminares

En este capítulo se presentarán varios conceptos preliminares antes de abordar la parte principal de este escrito. Se explicarán en detalle, y con ejemplos, los conceptos y operaciones de **intervalos**, **multi-intervalos**, **conjuntos**, **mapas**, y ***piecewise maps*** [9] [2]; con el objetivo de proporcionar una comprensión completa del contexto en el que se desarrolla este trabajo.

Adicionalmente, toda la notación presentada en este capítulo, así como en los capítulos siguientes, será recopilada en una tabla de notación que se incluye al final de este escrito, en el **Apéndice A**.

2.1. Intervalos

Un intervalo modela un conjunto unidimensional de valores naturales consecutivos, los cuales están separados por un salto o paso específico.

Un intervalo está compuesto por tres valores naturales: **inicio**, **salto** o **paso** y **fin**. Un intervalo se define como:

$$i = [k : l : m] = \{c \mid \exists d \in \mathbb{N}_0 \mid c = k + l \cdot d \wedge c \leq m\}$$

donde k es el inicio, l el salto o paso y m el fin.

A lo largo de esta tesina, entonces los intervalos serán representados utilizando la notación antes mencionada: $[inicio : salto : fin]$.

A modo ilustrativo, se presentan algunos ejemplos que ayudan a afianzar el concepto de intervalo: los números pares entre 2 y 10 se representan como $[2 : 2 : 10]$; los números impares entre 1 y 5, como $[1 : 2 : 5]$; y los múltiplos de 5 comprendidos entre 5 y 25, como $[5 : 5 : 25]$.

Para simplificar el código y mejorar la legibilidad, se definió el alias `NAT` para representar a los naturales como sinónimo de `unsigned long long` en C++. Además, se definió como `Inf` a el valor máximo representable de este tipo, es decir, el máximo número natural.

La estructura empleada para representar los intervalos se denomina `Interval`, y está compuesta por tres valores naturales internamente, tal como en la definición de los intervalos: `begin_`, que indica el valor de *inicio* del intervalo; `step_`, que define el *salto* o *paso* entre elementos consecutivos; y `end_`, que representa el valor *final* o *fin* del intervalo.

Pseudocódigo y Notación: Con el objetivo de simplificar y generalizar al máximo, en este capítulo y en todos los siguientes todas las operaciones descritas tendrán como primer argumento la instancia llamante del método en C++. Es decir:

```
a.método(b,c) == método(a,b,c)
```

En el caso de los operadores, se utilizaran como estos tendrían que usarse en código, de manera *inorder*. Por ejemplo con un operador `==` tal que a==b.

Algunas de las operaciones definidas e implementadas para los **intervalos**:

- `bool operator==(const Interval &i) const:` Verifica si dos intervalos son iguales, lo que ocurre si sus valores de inicio, paso y fin son exactamente iguales.

  Por ejemplo: $[0:1:10]$==$[0:1:10] = true$.

- `bool operator!=(const Interval &i) const:` Verifica si dos intervalos son distintos, es decir, si al menos uno de los valores de inicio, paso o fin es diferente entre ambos intervalos.

  Por ejemplo: $[0:1:10]$`!=`$[4:1:10] =$ `true`.

- `bool operator<(const Interval &i) const:` Verifica si un intervalo es estrictamente menor que otro, lo que ocurre si el valor de inicio del primer intervalo es menor estricto que el valor de inicio del segundo intervalo.

  Por ejemplo: $[0:1:10]$<$[4:1:10] =$ `true`.

- `unsigned int cardinal() const:` Devuelve la cantidad de elementos contenidos en el intervalo. Si el intervalo está vacío, devuelve cero.

  Por ejemplo: $cardinal([0:1:10]) = 11$.

- `bool isEmpty() const:` Verifica si un intervalo es vacío, es decir, si es igual a $[1:1:0]$.

  Por ejemplo: $isEmpty([0:1:10]) =$ `false`.

- `bool isMember(NAT x) const:` Verifica si el valor natural x pertenece al intervalo.

  Por ejemplo: $isMember([0:1:10], 5) =$ `true`.

- `Interval intersection(const Interval &i2) const:` Realiza la intersección entre dos intervalos, el que llama al método y el que viene como argumento $i2$, y devuelve el intervalo resultante.

  Por ejemplo: $intersection([0:1:10], [5:1:10]) = [1:1:0]$.

- `MaybeInterval compact(const Interval &i2) const:` Devuelve la concatenación del intervalo que llama al método y $i2$ si es posible unirlos en un único intervalo continuo. Si la concatenación no es posible, devuelve vacío. En particular `MaybeInterval` es sinónimo de `std::optional<Interval>`.

  Por ejemplo: $compact([0:1:10], [5:1:15]) = [0:1:15]$.

Claramente aquí se hace mención únicamente a las operaciones más relevantes con el objetivo también de afianzar conceptos claves, pero no es la intención de este documento proporcionar detalles explícitos sobre la implementación de dichas funciones. Si se desea consultar la implementación de las funciones de intervalos, la misma se encuentra en los archivos *interval.cpp* y *interval.hpp* en la carpeta *sbg* del repositorio de GitHub del CIFASIS.

2.2. Multi-intervalos

Los multi-intervalos modelan un conjunto multi-dimensional de valores consecutivos. Se definen como el producto cartesiano de una colección ordenada de intervalos. Dados k intervalos $i_0, i_2, \ldots, i_{k-1}$, se puede definir un multi-intervalo de dimensión k como:

$$mi = i_0 \times i_2 \times \ldots \times i_{k-1}$$

Una restricción importante sobre los intervalos que componen un multi-intervalo es que ninguno de ellos debe ser vacío. Esto se debe a que el producto cartesiano entre los elementos de los intervalos que conforman el multi-intervalo resulta en un conjunto vacío de n-uplas si al menos uno de los intervalos involucrados es vacío.

A continuación se presentan varios ejemplos para ilustrar y afianzar el concepto de multi-intervalo:

- **Unidimensional**

  $$[0:1:6]$$

  Este caso representa un intervalo simple que contiene todos los valores naturales desde 0 hasta 6.

- **Bidimensional**

  $$[0:1:2] \times [5:2:9]$$

  El multi-inventarlo contiene pares ordenados generados como producto cartesiano de ambos intervalos. Algunos ejemplos de tuplas incluidas son: $(0,5)$, $(0,7)$, $(0,9)$, $(1,5)$, $(1,7)$, $\ldots$, $(2,9)$.

- **Tridimensional**

  $$[0:1:1] \times [0:2:4] \times [1:1:2]$$

  Combinando estos valores se obtienen las triuplas como: $(0,0,1)$, $(0,2,2)$, $(1,4,1)$, $(1,0,2)$, entre otras.

La estructura utilizada para representar los multi-intervalos se denomina `MultiDimInter`, que luego tiene el sinónimo de tipos `SetPiece`. Esta estructura está compuesta por un vector de intervalos, de tipo `vector<Interval>`, nombrado `intervals_`, y donde el tipo ha sido renombrado como `InterVector` para facilitar su comprensión.

Dado que un multi-intervalo sin elementos representa un producto cartesiano sin n-uplas, se utilizará la notación $\emptyset_{mdi}$ para denotar un multi-intervalo vacío.

Se han definido las siguientes operaciones relevantes sobre los **multi-intervalos**:

- `Interval &operator[](std::size_t n)` o `const Interval &operator[](std::size_t n) const`: Devuelve el intervalo en la n-ésima dimensión.

  Por ejemplo: $[0:1:10] \times [11:1:20] \times [21:1:21][2] = [21:1:21]$.

- `void emplaceBack(Interval i)`: Agrega un intervalo i al multi-intervalo.

  Por ejemplo:
  $emplaceBack([0:1:10] \times [11:1:20] \times [21:1:21]) = [0:1:10] \times [11:1:20] \times [21:1:21] \times [1:1:20]$.

- `unsigned int cardinal() const`: Devuelve la cantidad de elementos contenidos en el multi-intervalo. Si el multi-intervalo es vacío, devuelve 1.

  Por ejemplo: $cardinal([0:1:10] \times [0:1:10]) = 121$.

- `bool isEmpty() const`: Verifica si un multi-intervalo es vacío.

  Por ejemplo: $isEmpty([0:1:10] \times [1:1:11])$`false`.

- `MD_NAT minElem() const:` Devuelve el mínimo elemento del multi-intervalo, representado como un natural multidimensional (`MD_NAT`), donde cada componente corresponde al valor `begin_` de cada uno de los intervalos que componen el multi-intervalo.

  El tipo `MD_NAT` se utiliza para representar naturales multi-dimensionales. Este tipo corresponde simplemente a un vector de valores del tipo `NAT`, es decir, se trata de una estructura del tipo `vector<NAT>`.

  Por ejemplo: $minElem([0:1:10] \times [1:1:11]) = (0, 1)$.

- `MD_NAT maxElem() const:` Devuelve el máximo elemento del multi-intervalo, representado como un natural multidimensional (`MD_NAT`), donde cada componente corresponde al valor `end_` de cada uno de los intervalos que componen el multi-intervalo.

  Por ejemplo: $minElem([0:1:10] \times [1:1:11]) = (10, 11)$.

- `MultiDimInter intersection(const MultiDimInter &other) const:` Realiza la intersección entre dos multi-intervalos. Si estos no tienen elementos en común, devuelve el multi-intervalo vacío,$\emptyset_{mdi}$.

  Por ejemplo: $intersection([0:1:10] \times [0:1:10], [5:1:10] \times [0:1:11]) = [5:1:10] \times [0:1:10]$.

- `bool operator==(const MultiDimInter &other) const:` Verifica si dos multi-intervalos son iguales, lo que ocurre si en cada dimensión los intervalos de ambos son exactamente iguales.

  Por ejemplo: $[0:1:10] \times [11:1:20] \times [21:1:21]$`==`$[0:1:10] \times [11:1:20] \times [21:1:21] =$ `true`.

- `bool operator!=(const MultiDimInter &other) const:` Verifica si dos multi-intervalos son distintos, lo que ocurre si en al menos una dimensión los intervalos de ambos son exactamente distintos.

  Por ejemplo: $[0:1:10] \times [11:1:20] \times [21:1:21]$`==`$[0:1:10] \times [11:1:20] \times [21:1:21] =$ `false`.

- `bool operator<(const MultiDimInter &other) const:` Verifica si un multi-intervalo es estrictamente menor que otro, lo que ocurre si el mínimo elemento de uno es menor que el del otro, en base al operador `<` definido para los naturales multi-dimensionales.

  Por ejemplo: $[0:1:10] \times [11:1:20] \times [21:1:21]$`<`$[0:1:10] \times [11:1:20] \times [21:1:21] =$ `false`.

- `std::size_t arity() const:` Devuelve la cantidad de intervalos que tiene el multi-intervalo, es decir, la cantidad de dimensiones que posee el multi-intervalo.

  Por ejemplo: $arity([0:1:10] \times [0:1:10]) = 2$.

- `MaybeMDI compact(const MultiDimInter &other) const:` Devuelve la compactación de dos multi-intervalos si es posible unirlos en un único multi-intervalo. Si la compactación no es posible, devuelve vacío. En particular `MaybeMDI` es sinónimo de `std::optional<MultiDimInter>`.

  Por ejemplo: $compact([0:1:10] \times [0:1:10], [5:1:15] \times [0:1:10]) = [0:1:15] \times [0:1:10]$.

Nuevamente, aquí se hace mención únicamente a las funciones mas relevantes y a su propósito, pero no es la intención de este documento proporcionar detalles explícitos sobre la implementación de dichas operaciones. Si se desea consultar la implementación, esta se encuentran en la carpeta *sbg* del repositorio, en el archivo *multidim_inter.cpp*.

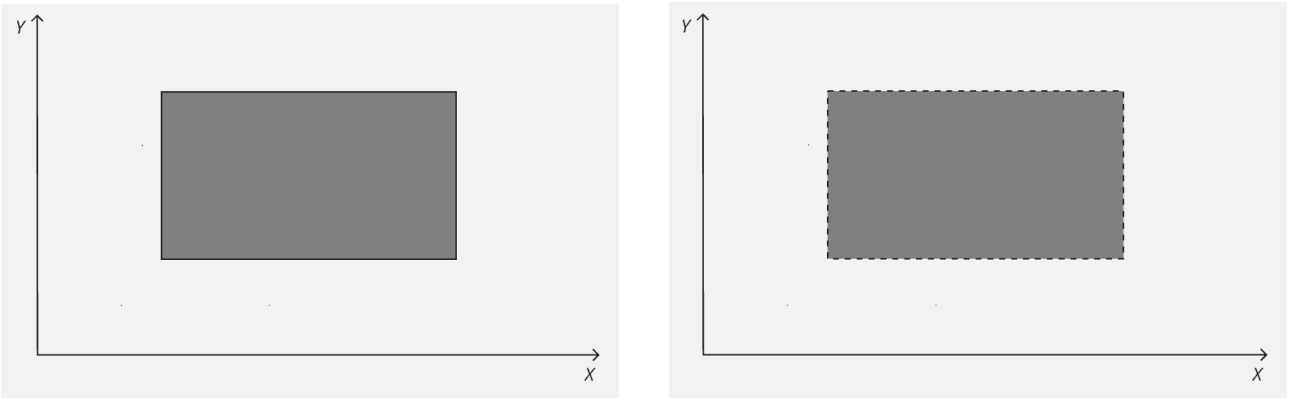
Antes de proseguir con la sección destinada a los conjuntos, será útil explicar cómo se representarán gráficamente los multi-intervalos. Cabe destacar que, a lo largo de esta tesina, se trabajará exclusivamente con ejemplos en los que los multi-intervalos son bidimensionales, ya que esta cantidad de dimensiones resulta adecuada para realizar las explicaciones sin que se vuelvan complejas de visualizar. Las representaciones gráficas se aplicarán, por lo tanto, únicamente a este caso.

Se utilizarán dos tipos de representación distintas:

- Para los multi-intervalos **densos** se usará la representación mostrada en la subfigura (a) de la Figura 2.1. En particular la definición de un multi-intervalo denso será la siguiente:

  Un multi-intervalo m se denomina denso ***si y solo si*** *para todas las dimensiones los intervalos tienen paso igual a 1.*

- Para los multi-intervalos **no densos** se usará la representación de la subfigura (b) de la Figura 2.1. Estos son los que no cumplen con la definición anterior, es decir, los que contienen al menos un intervalo con paso distinto de 1.



(a) Multi-intervalo denso — línea continua (b) Multi-intervalo no denso — línea discontinua

Figura 2.1: Representación de multi-intervalos bidimensionales

Adicionalmente, se incluirá una representación gráfica que resalta el elemento mínimo y máximo de un multi-intervalo. Para ello, se utilizará un punto verde para indicar el mínimo y un punto rojo para el máximo. En la Figura 2.2 se muestra esta representación de manera ilustrativa.

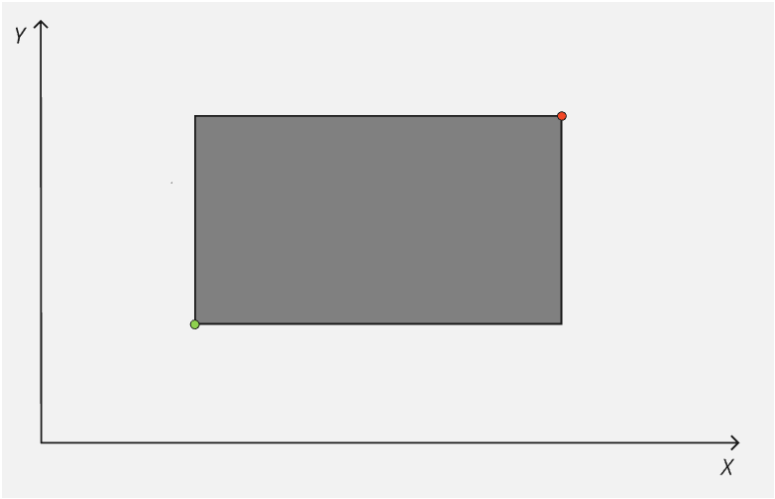


Figura 2.2: Representación de un multi-intervalo con su mínimo (verde) y máximo (rojo).

2.3. Conjuntos

Se modela un conjunto como una familia o colección de multi-intervalos no vacíos de igual cantidad de dimensiones y disjuntos dos a dos, es decir, no comparten ningún valor multi-dimensional d entre ellos.

En consecuencia, la dimensión de un conjunto no está determinada por la cantidad total de multi-intervalos que lo componen, sino que corresponde a la dimensión de cualquiera de ellos.

La notación que se le dará a los conjuntos será la siguiente:

$$\{i_{00} \times i_{01} \ \dots \ \times i_{0(k-1)},\ i_{10} \times i_{11} \ \dots \ \times i_{1(k-1)} \ \dots,\ i_{(i-1)0} \times i_{(i-1)1} \ \dots \ \times i_{(i-1)(k-1)}\}$$

Omitiendo los multi-intervalos:

$$\{mdi_0,\ mdi_1 \ \dots,\ mdi_{i-1}\}$$

donde i es la cantidad de multi-intervalos del conjunto y k es la cantidad de dimensiones que tendrán los mismos.

En particular, el conjunto vacío se representará de la siguiente manera: {}. Y en particular se utilizara la notación $\kappa(A)$ para representar la cantidad de multi-intervalos del conjunto A.

En primera instancia se verán todas las funciones relevantes que los conjuntos tienen disponibles en términos generales, ya que en el proyecto se disponen de múltiples implementaciones de conjuntos.

2.3.1. Conjuntos generales

La estructura `SetDelegate` representa un tipo base abstracto que define una interfaz común para manipulación de conjuntos de multi-intervalos. Este tipo se define como parte del conocido patrón de diseño *delegate*, utilizado en este contexto para encapsular distintas implementaciones concretas de conjuntos dentro de una estructura delegadora. Esta estructura delegadora corresponde al tipo `Set`. De este modo, siempre que se requiera trabajar con conjuntos fuera de su implementación, se empleará una instancia de `Set`, la cual contendrá una implementación concreta de conjuntos del tipo `SetDelegate`.

Entre estas implementaciones concretas se encuentran: `UnorderedSet`, que representa conjuntos desordenados; y `OrderedDenseSet`, que modela conjuntos ordenados densos, es decir, con una única dimensión y con paso igual a uno. A partir de aquí.

Se han definido las siguientes operaciones sobre los conjuntos en la interfaz propuesta por `SetDelegate` :

- `virtual bool operator==(const SetDelegate &other) const = 0`: Verifica que los dos conjuntos tengan exactamente los mismos elementos.

  Por ejemplo: $\{[0:1:10] \times [11:1:20] \times [21:1:21]\}$`==`$\{[0:1:10] \times [11:1:20] \times [21:1:21]\} =$ `true`.

- `virtual bool operator!=(const SetDelegate &other) const = 0`: Verifica que los dos conjuntos no ten

  Por ejemplo: $\{[0:1:10] \times [11:1:20] \times [21:1:21]\}$`!=`$\{\} =$ `true`.

- `virtual std::size_t size() const = 0`: Devuelve la cantidad de multi-intervalos en el conjunto.

  Por ejemplo: $size(\{[0:1:10] \times [11:1:20] \times [21:1:21]\}) = 1$.

- `virtual void emplaceBack(const SetPiece &mdi) = 0`: Agrega un multi-intervalo *mdi* más al final del conjunto.

  Por ejemplo: $emplaceBack(\{[0:1:10] \times [11:1:20] \times [21:1:21]\}, [11:1:21] \times [11:1:20] \times [21:1:21]) = \{[0:1:10] \times [11:1:20] \times [21:1:21], [11:1:21] \times [11:1:20] \times [21:1:21]\}$.

- `virtual void emplace(const SetPiece &mdi) = 0`: Agrega un multi-intervalo *mdi* en algún lugar, donde corresponda, del conjunto.

  Por ejemplo: $emplace(\{[0:1:10] \times [11:1:20] \times [21:1:21]\},$
  $[11:1:21] \times [11:1:20] \times [21:1:21]) =$
  $\{[0:1:10] \times [11:1:20] \times [21:1:21], [11:1:21] \times [11:1:20] \times [21:1:21]\}$.

- `virtual unsigned int cardinal() const = 0`: Devuelve la cantidad de elementos contenidos en el conjunto. En escancia la suma del cardinal de todos los multi-intervalos que lo conforman.

  Por ejemplo: $cardinal(\{[0:1:10]\}) = 11$.

- `virtual bool isEmpty() const = 0`: Verifica si un conjunto es vacío, es decir, no contiene multi-intervalos.

  Por ejemplo: $isEmpty(\{[0:1:10]\}) =$ `false`.

- `virtual MD_NAT minElem() const = 0`: Devuelve el mínimo elemento del conjunto.

  Por ejemplo: $minElem(\{[0:1:10],[15:1:100]\}) = 0$.

- `virtual MD_NAT maxElem() const = 0`: Devuelve el máximo elemento del conjunto.

  Por ejemplo: $maxElem(\{[0:1:10],[15:1:100]\}) = 100$.

- `virtual SetDelegPtr intersection(const SetDelegate &other) const = 0)`: Realiza la intersección entre el conjunto que invoca el método y el conjunto *other*. Si estos no tienen elementos en común, devuelve el conjunto vació.

  Por ejemplo: $intersection(\{[0:1:100],[500:1:1000]\},\{[1000:1:10000]\}) = \{[1000:1:1000]\}$.

- `virtual SetDelegPtr cup(const SetDelegate &other) const = 0`: Realiza la union entre el conjunto que invoca el método y el conjunto *other*.

  Por ejemplo: $cup(\{[0:1:100],[500:1:1000]\},\{[500:1:1000],[1400:1:10000]\}) = \{[0:1:100],[500:1:1000],[1400:1:10000]\}$.

- `virtual SetDelegPtr complement() const = 0`: Obtiene el complemento del conjunto que invoca el método.

  Por ejemplo: $complement(\{[10:1:100]\}) = \{[0:1:9],[101:1:$ `Inf`$]\}$.

- `virtual SetDelegPtr difference(const SetDelegate &other) const = 0`: Realiza la diferencia entre el conjunto que invoca el método y el conjunto *other*, es decir, al invocante se le resta *other*.

  Por ejemplo: $difference(\{[1:1:2],[3:1:100]\},\{[50:1:100]\}) = \{[1:1:2],[3:1:49]\}$.

- `virtual std::size_t arity() const = 0`: Devuelve la cantidad de dimensiones que tiene el conjunto, es decir, la aridad de los multi-intervalos que lo componen.

  Por ejemplo: $arity(\{[0:1:10]\times[11:1:20]\times[21:1:21]\}) = 3$.

- `virtual SetDelegPtr disjointCup(const SetDelegate &other) const = 0`: Realiza la unión disjunta entre el conjunto que invoca el método y el conjunto *other*. Se asume que los argumentos son conjuntos disjuntos.

  Por ejemplo: $disjointCup(\{[1:1:2],[3:1:100]\},\{[0:1:0],[1000:1:3000]\}) = \{[1:1:2],[3:1:100],[0:1:0],[1000:1:3000]\}$.

- `virtual SetDelegPtr compact() const = 0`: Devuelve el conjunto resultante de intentar compactar los distintos multi-intervalos del conjunto que invoca el método con el resto de los multi-intervalos.

Por ejemplo: $compact(\{[0:1:10] \times [11:1:20] \times [21:1:21]\}) = \{[0:1:21]\}$.

Para cumplir con el objetivo de proporcionar una implementación concreta de conjuntos ordenados, se decidió tomar como punto de partida la implementación ya existente de conjuntos desordenados.

Aunque, como se indicó anteriormente, no es el propósito de este documento detallar dichas implementaciones previas, se explicará con mayor profundidad el funcionamiento de la operación `complement` en el caso de conjuntos desordenados, ya que se considera útil especialmente para facilitar y modularizar la comprensión de las optimizaciones propuestas.

Asimismo, se incluirá una descripción más breve de otras operaciones relevantes, tales como `intersection` y `disjointCup`.

2.3.2. Conjuntos desordenados

Cabe destacar que la estructura `UnorderedSet` se encuentra compuesta exclusivamente por una colección de multi-intervalos, almacenada en una variable denominada `pieces_`. Esta colección se implementa mediante un contenedor de tipo `vector<SetPiece>`.

Pseudocódigo/Notación: En este capítulo los diferentes multi-intervalos de un conjunto desordenado se indexarán mediante subíndices. En particular, A_i representará el i-ésimo multi-intervalo del conjunto desordenado A, lo cual corresponde a `pieces_[i]` en C++, con $i \in \{0, \ldots, \kappa(A) - 1\}$.

2.3.2.1. Intersección - `intersection`

Como se puede observar en el Algoritmo 1 el pseudocódigo para la operación `intersection`, la implementación de la intersección entre conjuntos desordenados resulta, en esencia, relativamente simple y directa, si se excluyen los casos especiales que optimizan su ejecución.

El enfoque central de esta operación consiste en iterar sobre todos los multi-intervalos que conforman el conjunto desordenado A y, para cada uno de ellos, computar su intersección con todos los multi-intervalos del conjunto desordenado B. De esta forma, se construye un nuevo conjunto cuyos elementos corresponden a las intersecciones no vacías entre los pares de multi-intervalos provenientes de A y B respectivamente.

2.3.2.2. Complemento - `complement`

El cálculo del complemento de conjuntos desordenados se define a partir de dos funciones principales: `complement` y `complementAtom`.

La función `complement` es la encargada de realizar el cálculo general del complemento para un conjunto arbitrario con ayuda de la operación `intersection`, mientras que `complementAtom` se especializa en calcular el complemento de un conjunto que contiene únicamente un multi-intervalo.

A continuación, se detalla el funcionamiento específico de ambas funciones.

Complement

La operación de complemento, denominada `complement`, permite calcular el complemento de un conjunto desordenado compuesto por una cantidad variable de multi-intervalos. En el pseudocódigo, Algoritmo 2, se presenta la operación `complement`, donde se ilustra su funcionamiento.

Desde una perspectiva teórica de conjuntos, la operación busca calcular la siguiente expresión:

$$\bigcap_{i=0}^{\kappa(A)-1} \overline{\{A_i\}}$$

La cual se fundamenta en que:

Algorithm 1 Intersección entre conjuntos desordenados

Require: A, B son conjuntos desordenados
Ensure: C es un conjunto desordenado que representa $A \cap B$

```
1:  function INTERSECTION(A, B)
2:      C := {}
3:      if ISEMPTY(A) or ISEMPTY(B) then
4:          return C
5:      end if
6:      if MAXELEM(A) < MINELEM(B) or MAXELEM(B) < MINELEM(A) then
7:          return C
8:      end if
9:      if MAXELEM(A) == MINELEM(B) then
10:         C := {MAXELEM(A)}
11:         return C
12:     end if
13:     if MAXELEM(B) == MINELEM(A) then
14:         C := {MINELEM(A)}
15:         return C
16:     end if
17:     if A == B then
18:         return A
19:     end if

20:     for all a ∈ A do
21:         for all b ∈ B do
22:             mdi := INTERSECTION(a, b)
23:             if i ≠ ∅ then
24:                 EMPLACEBACK(C, mdi)
25:             end if
26:         end for
27:     end for
28:     return C
29: end function
```

$$A = \{A_0, A_1, A_2, \ldots, A_{\kappa(A)-1}\} = \bigcup_{i=0}^{\kappa(A)-1} \{A_i\}$$

y, por lo tanto,

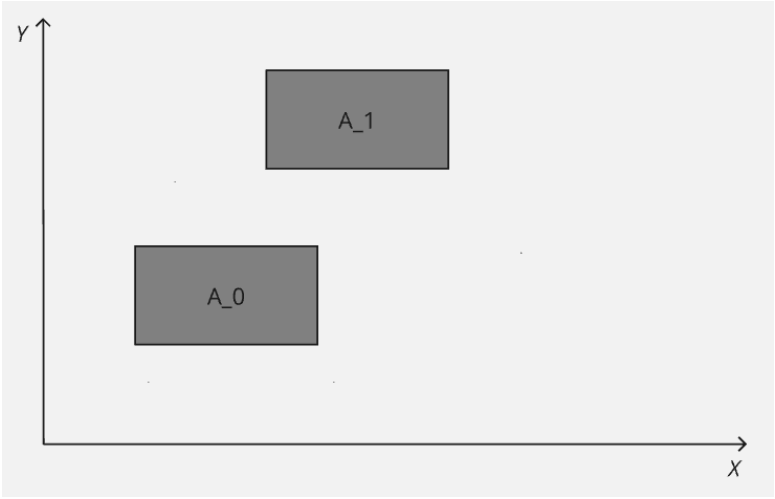
$$\overline{A} = \overline{\bigcup_{i=0}^{\kappa(A)-1} \{A_i\}} = \bigcap_{i=0}^{\kappa(A)-1} \overline{\{A_i\}}.$$

Algorithm 2 Complemento de un conjunto desordenado

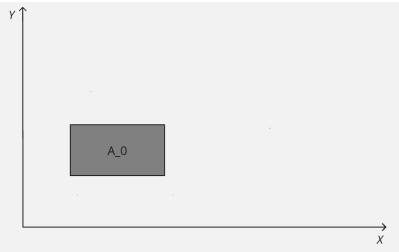
Require: A es un conjunto desordenado
Ensure: C es un conjunto desordenado que representa el complemento de A

```
1: function COMPLEMENT(A)
2:     C := {}
3:     first_mdi := A_0
4:     C := COMPLEMENTATOM(first_mdi)
5:     for i := 1; i ≠ size(A); i++ do
6:         mdi := A_i
7:         atomic_set := COMPLEMENTATOM(mdi)
8:         C := INTERSECTION(C,atomic_set)
9:     end for
10:    return C
11: end function
```

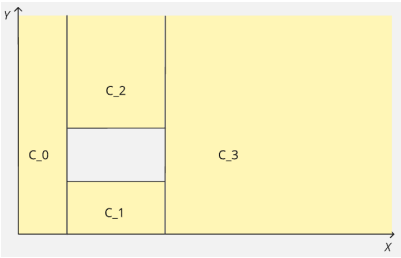
En la Figura 2.4 se muestra gráficamente cómo se va calculando el complemento de un conjunto desordenado. En particular en este ejemplo el conjunto A tiene solo multi-intervalos densos. Se eligieron multi-intervalos densos para que la representación gráfica sea mas sencilla de comprender.



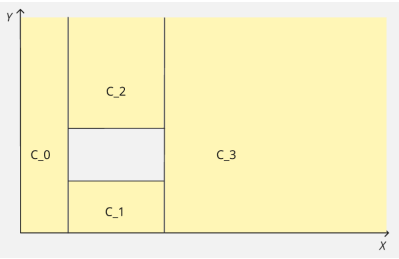
(a) Conjunto desordenado A



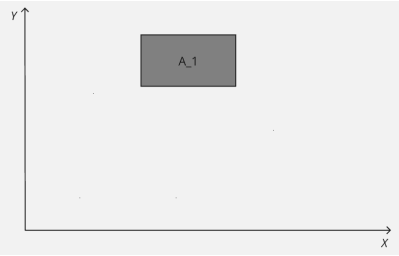
(b1) Conjunto atómico con A_0



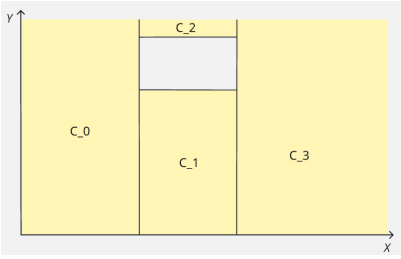
(b2) Complemento del conjunto atómico



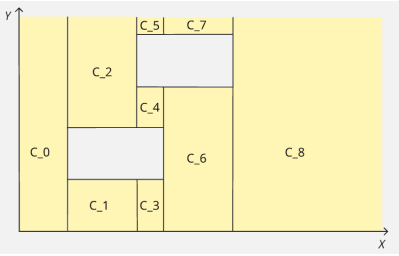
(b3) Conjunto C



(b4) Conjunto atómico con A_1



(b5) Complemento del conjunto atómico



(b6) Conjunto C

Figura 2.3: Visualización del cálculo del complemento de un conjunto desordenado con multi-intervalos densos A.

ComplementAtom

Una vez descrita la operación `complement` de conjuntos desordenados, lo único que resta por detallar es cómo se calcula el complemento de un conjunto desordenado que contiene un único multi-intervalo. Esta tarea es realizada por la operación *complementAtom*. El procedimiento que dicha operación lleva a cabo se encuentra plasmado en el pseudocódigo de la operación en el Algoritmo 3, y puede explicarse de la siguiente manera:

1. Se inicia el proceso con tres multi-intervalos fundamentales: *dense_mdi*, que tiene la misma disposición de intervalos que el del conjunto original, pero con paso 1 en todos ellos(representando al multi-intervalo en su versión completamente densa); *during_mdi*, que inicialmente es una copia de *dense_mdi* y *all*, que representa el multi-intervalo universo, con la misma aridad que el multi-intervalo del conjunto.

2. Ahora bien, para calcular el complemento del conjunto atómico, se tendrán que recorrer una a una las dimensiones del multi-intervalo interno con la siguiente lógica:

   a) **Selección de la dimensión.** Se empieza por la dimensión $d = 0$. Y se extrae en base a ella el

intervalo original del multi-intervalo del cual se quiere obtener el complemento:

$$d = [d_{\text{begin}} : d_{\text{step}} : d_{\text{end}}].$$

b) **Región "antes" del intervalo.** Si $d_{\text{begin}} > 0$, existe un rango no cubierto entre 0 y $d_{\text{begin}} - 1$. Entonces se toma el multi-intervalo universo *all* y se lo restringe en la dimensión d a

$$[0 : 1 : d_{\text{begin}} - 1].$$

dejando los intervalos de las demás dimensiones sin cambios. Esa variación de *all* se añade al conjunto parcial de resultados C y luego deshace el cambio.

c) **Huecos "durante" el intervalo.** Si el paso $d_{\text{step}} > 1$, el intervalo original salta posiciones; entre cada salto quedan "huecos" que también forman parte del complemento. Para cada $j = 0, \ldots, d_{\text{step}} - 2$ construimos

$$[d_{\text{begin}} + j + 1 : d_{\text{step}} : d_{\text{end}}],$$

y lo insertamos en una copia de nuestro multi-intervalo base, *during_mdi*, y lo metemos en el conjunto desordenado resultante C.

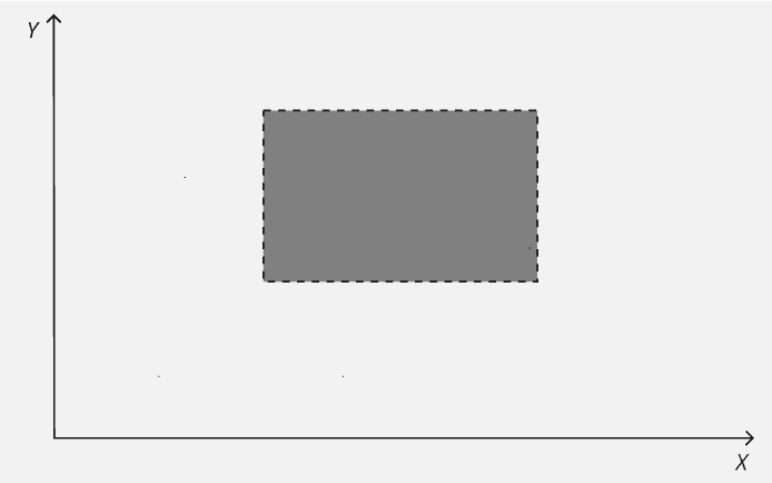
d) **Región "después" del intervalo.** Si $d_{\text{end}} <$ `Inf`, hay un rango desde $d_{\text{end}} + 1$ hasta `Inf` de valores no abarcados por i. De nuevo se usa *all* restringido en d a

$$[i_{\text{end}} + 1 : 1 : \texttt{Inf}],$$

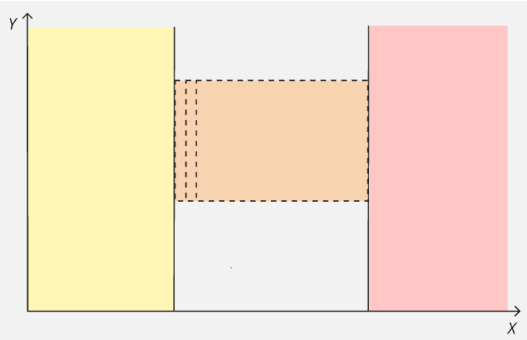
se lo añade al conjunto de resultados y se revierte el cambio de *all*.

e) **Restauración y avance.** Tras procesar el "antes", "durante" y "después" en la dimensión d, se actualizan *all* y *during_mdi*, colocándoles los valores densos y originales de la dimensión d del multi-intervalo del cual estamos sacando el complemento respectivamente. Luego se pasa a la siguiente dimensión $d + 1$, y se repite todo el procedimiento.

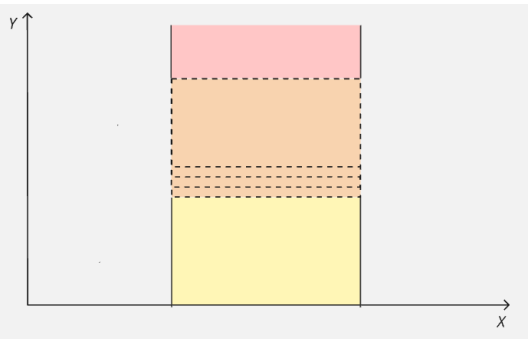
En la Figura a siguiente se muestra gráficamente cómo se va calculando el complemento atómico de un conjunto desordenado con un único multi-intervalo. En particular en este ejemplo el conjunto tiene un solo multi-intervalo no denso.



(a) Conjunto desordenado atómico



(b1) Multi-intervalos resultantes del complemento atómico sobre la primera dimensión x



(b2) Multi-intervalos resultantes del complemento atómico sobre la segunda dimensión y

Figura 2.4: Visualización del cálculo del complemento atómico sobre un conjunto desordenado atómico no denso.

2.3.2.3. Unión disjunta - `disjointCup`

En lo que respecta a la operación correspondiente a la *unión disjunta*(`disjointCup`), al igual que la intersección, se trata de una operación conceptualmente sencilla. Esta simplicidad se ve reflejada en el pseudocódigo del Algoritmo 4. Dado que los conjuntos involucrados son desordenados y, por hipótesis, disjuntos entre sí, la operación no requiere verificaciones adicionales de solapamiento o duplicación de elementos. Basta simplemente con reunir todos los multi-intervalos de ambos conjuntos en una único conjunto desordenado para obtener el resultado deseado.

2.3.3. Conjuntos ordenados densos

A continuación, se presenta una breve descripción de cómo los conjuntos ordenados densos organizan los multi-intervalos de manera interna. Esta explicación resulta relevante dado que, por razones de conveniencia, se ha adoptado el mismo criterio de orden para los multi-intervalos contenidos en un conjunto ordenado.

En particular, los conjuntos ordenados densos emplean una variable miembro llamada `pieces_`, de tipo `MDIOrdSet`, que es un sinónimo de `vector<SetPiece>`. Esta estructura almacena los multi-intervalos, los cuales se ordenan utilizando la operación $<$ definida para los multi-intervalos (`SetPiece`).

Internamente, el operador $<$ entre multi-intervalos evalúa si el mínimo del primer multi-intervalo es menor que el del segundo. Dado que los mínimos son de naturales multi-dimensionales(`MD_NAT`), esta comparación se realiza mediante el operador $<$ definido para `MD_NAT`. En si, el operador $<$ definido para el tipo `MD_NAT` compara

Algorithm 3 Complemento atómico de conjunto ordenado atómico

Require: A es un conjunto ordenado atómico
Ensure: Un conjunto desordenado C, el complemento de A

```
1: function COMPLEMENTATOM(A)
2:     C := ∅
3:     mdi := A_0
4:     dense_mdi := ||
5:     for all interval ∈ mdi do
6:         i :=[BEGIN(interval) : 1 : END(interval)]
7:         EMPLACEBACK(dense_mdi, i)
8:     end for
9:     during_mdi := dense_mdi
10:    univ := [0 : 1 : Inf]
11:    all := |univ|^ARITY(A)                ▷ Representa el universo completo de aridad ARITY(A)
12:    dim := 0
13:    for all i ∈ mdi do
14:        if BEGIN(i) ≠ 0 then
15:            i_res := [0 : 1 : BEGIN(i)−1]
16:            if ¬ISEMPTY(i_res) then
17:                all[dim] := i_res
18:                C := C ⌢ {all}
19:                all[dim] := univ
20:            end if
21:        end if
22:        if BEGIN(i) < Inf then
23:            if STEP(i) > 1 then
24:                for j = 0; i < size(A); j++ do
25:                    h := [BEGIN(i)+j + 1 :STEP(i) : END(i)]
26:                    if ¬ISEMPTY(h) then
27:                        during_mdi[dim] := h
28:                        C := C ⌢ during_mdi
29:                    end if
30:                end for
31:            end if
32:        end if
33:        if END(i) < Inf then
34:            i_res := [END(i)+1 : 1 : Inf]
35:            if ¬ISEMPTY(i_res) then
36:                all[dim] := i_res
37:                C := C ⌢ {all}
38:                all[dim] := univ
39:            end if
40:        end if
41:        all[dim] := dense_mdi[dim]
42:        during_mdi[dim] := i
43:        dim++
44:    end for
45:    return C
46: end function
```

Algorithm 4 Unión disjunta de dos conjuntos desordenados

Require: A, B son conjuntos desordenados
Ensure: R es un conjunto desordenado que representa la unión disjunta de A y B

```
1: function DISJOINTCUP(A, B)
2:     if ISEMPTY(A) then
3:         return B
4:     end if
5:     if ISEMPTY(B) or A == B then
6:         return A
7:     end if
8:     R := A
9:     for all b ∈ B do
10:        EMPLACEBACK(R, b)
11:    end for
12:    return R
13: end function
```

componente por componente dos `MD_NAT`, siguiendo el siguiente criterio que luego se aplica a los mínimos de los multi-intervalos:

Sea $x = (x_0, x_1, \ldots, x_{n-1})$ y $y = (y_0, y_1, \ldots, y_{n-1})$ dos elementos de tipo `MD_NAT`. Se define que $x < y$ si y sólo si existe un índice $j \in \{0, \ldots, n-1\}$ tal que se cumplen simultáneamente las siguientes condiciones:

- $x_j < y_j$, es decir, en la componente j, x es menor que y;
- Para todo i tal que $0 \leq i < j$, se cumple que $x_i = y_i$.

Ahora bien al trabajar con multi-intervalos de solo una dimensión, este operador de menor queda relegado a solo ser el $<$ tradicional de los naturales. Por ende dados los siguientes multi-intervalos de una dimensión:

$$mdi_1 = |[0:3:9]|, \text{ con mínimo } 0.$$
$$mdi_2 = |[4:1:10]|, \text{ con mínimo } 4.$$
$$mdi_3 = |[2:2:4]|, \text{ con mínimo } 2.$$

Se tiene que el conjunto ordenado denso queda tal que así:

$$\{mdi_1,\ mdi_3,\ mdi_2\}$$

ya que:

$$0 < 2 < 4$$

2.3.4. *Abstract factory* para conjuntos

Adicionalmente, se incorpora el patrón de diseño *abstract factory*, aplicado a la creación de instancias del tipo `SetDelegate`. Para ello, se define la clase abstracta `SetAF`, que actúa como una interfaz común para diferentes fábricas concretas, como `UnordAF`, para conjuntos desordenados y `OrdDenseAF`, para conjuntos ordenados densos, cada una responsable de construir conjuntos con una representación interna específica.

Esta abstracción cobra especial relevancia en la sección de mapas. Allí, la fábrica de mapas (`MapAF`) extiende a `SetAF`, y reutiliza internamente una fábrica concreta de conjuntos para construir tanto dominios como imágenes de los mapas.

2.4. Mapas lineales

Un mapa lineal puede entenderse como una asociación entre un conjunto de multi-intervalos y una colección de expresiones lineales. Formalmente, un mapa lineal de aridad $j \in \mathbb{N}$ se define como:

$$map = \{mdi_0,\ mdi_2,\ \ldots,\ mdi_{k-1}\} \mapsto [e_0,\ e_2,\ \ldots,\ e_{j-1}]$$

donde cada mdi_l es un multi-intervalo de aridad j, con l entre 0 y k y con $k \in \mathbb{N}$, y cada e_i, con i entre 0 y j, es una expresión lineal de la forma $m * x + h$, con $m, h \in \mathbb{Q}$.

El dominio del mapa está conformado multi-intervalos, mdi, y cada expresión e_i se evalúa sobre la i-ésima dimensión de dichos multi-intervalos. En otras palabras, el dominio de e_i está compuesto por la colección de todos los intervalos que aparecen en la dimensión i de cada mdi presente en el dominio.

Este concepto de mapa lineal coincide con el de función multidimensional cabe recalcar. Para afianzar la idea, véanse ejemplos en una, dos y tres dimensiones:

Ejemplo 1. Unidimensional

$$map = \{[0, 1, 2], [5, 1, 7]\} \mapsto [x + 1].$$

Por ejemplo: $(1) \mapsto [2]$ $(6) \mapsto [7]$

Ejemplo 2. Bidimensional

$$map = \{[0, 1, 1] \times [0, 1, 1], [3, 1, 4] \times [3, 1, 4]\} \mapsto [x_0 + 2,\ 2 * x_1 - 1].$$

Por ejemplo: $(0, 1) \mapsto [2, 1],\ (3, 4) \mapsto [5, 7]$.

Ejemplo 3. Tridimensional

$$map = \{[0, 1, 1] \times [0, 1, 1] \times [0, 1, 1], [2, 1, 3] \times [2, 1, 3] \times [2, 1, 3]\} \mapsto [x_0,\ x_1 + 1,\ 2 * x_2 - 1].$$

Por ejemplo:

$$(1, 0, 1)\ \mapsto\ [1, 1, 1], \quad (3, 2, 3)\ \mapsto\ [3, 3, 5].$$

La estructura **Map** se utilizara para representar a los mapas lineales y esta constituida por lo que sigue:

- `dom` (de tipo **Set**): representa el dominio del mapa.
- `exp` (de tipo **Exp**): contiene la(s) expresión(es) lineal(es) asociada(s).
- `fact` (de tipo **SetAF**): contiene una fábrica concreta utilizada para construir cualquier tipo de conjunto relacionado con el mapa

Cabe destacar que el tipo `Exp` representará una colección de expresiones lineales, es un sinónimo para `vector<LExp>`, donde `LExp` es tipo de estructura que representa a una única expresión lineal.

A continuación se presentan las operaciones que tienen a disposición los mapas:

- `bool operator==(const Map &other) const`: Verifica si dos mapas son iguales, es decir, si sus dominios y expresiones son idénticos, o si sus dominios e imágenes son iguales en caso de que sea un mapa con una único expresión.

  Por Ejemplo: $\{[1 : 1 : 3]\} \mapsto [x]$ `==` $\{[1 : 1 : 3]\} \mapsto [x] = true.$

- `bool operator!=(const Map &other) const`: Verifica si dos mapas son distintos.

  Por Ejemplo: $\{[1 : 1 : 3]\} \mapsto [x]$ `==` $\{[1 : 1 : 3]\} \mapsto [x + 1] = true.$

- `Map operator+(const Map &other) const`: Suma dos mapas realizando la intersección de sus dominios y sumando sus expresiones.

  Por Ejemplo: $\{[1:1:3]\} \mapsto [x] + \{|[2:1:4]|\} \mapsto [2*x] = \{[2:1:3]\} \mapsto [3*x]$.

- `std::size_t arity() const`: Devuelve la aridad del dominio del mapa.

  Por Ejemplo: $arity(\{[1:1:5] \times [0:1:2]\} \mapsto [x,x]) = 2$.

- `bool isEmpty() const`: Devuelve `true` si el mapa no tiene elementos en su dominio.

  Por Ejemplo: $isEmpty(\{\} \mapsto [x])$ = `true`.

- `Map restrict(const Set &subdom) const`: Restringe el dominio del mapa en base *subdom*.

  Por Ejemplo: $restrict(\{[0:1:10]\} \mapsto [x], \{[5:1:8]\}) = \{[5:1:8]\} \mapsto [x]$.

- `Set image() const`: Devuelve el conjunto imagen del mapa.

  Por Ejemplo: $image(\{[0:1:3]\} \mapsto [2*x]) = \{[0:2:6]\}$.

- `Set image(const Set &subdom) const`: Imagen del mapa restringido por *subdom*.

  Por Ejemplo: $image(\{[0:1:3]\} \mapsto [2*x], \{[1:1:1]\}) = \{[2:1:2]\}$.

- `Set preImage(const Set &subcodom) const`: Preimagen del subconjunto de la imagen *subcodom*.

  Por Ejemplo: $preImage(\{[0:1:3]\} \mapsto [2*x], \{[4:1:4]\}) = \{[2:1:2]\}$.

- `Map composition(const Map &other) const`: Composición de mapas.

  Por Ejemplo: $composition(\{[0:1:3]\} \mapsto [x+1], \{[0:1:3]\} \mapsto [2*x]) = \{[0:1:3]\} \mapsto [2*x+1]$.

- `Map minInv() const`: Calcula la inversa o pseudo-inversa de un mapa.

  Por Ejemplo:$composition(\{[5:1:10]\} \mapsto [x+1]) = \{[6:1:11]\} \mapsto [x-1]$.

- `MaybeMap compact(const Map &other) const`: Compacta dominios si tienen la misma ley.

  Por Ejemplo: $composition(\{[0:1:3]\} \mapsto [x], \{[4:1:5]\} \mapsto [x]) = \{[0:1:5]\} \mapsto [x]$.

Nuevamente, si se desea consultar todo el código relacionado con mapas, este se encuentra disponible en la carpeta *sbg* del repositorio, específicamente en los archivos *map.cpp* y *map.hpp*.

2.4.1. *Abstract factory* de mapas

En este caso, el patrón *abstract factory* se aplica nuevamente, pero orientado a la creación de mapas. A diferencia de su uso tradicional, donde suele aplicarse por la existencia de múltiples implementaciones concretas, aquí se emplea con el objetivo de encadenar la elección de la implementación de conjuntos a la de mapas.

Para ello, se define la estructura `MapAF`, la cual hereda de `SetAF`, y que contiene una fábrica concreta de conjuntos (`SetAF`). De esta manera, una vez elegida la implementación deseada para los conjuntos (por ejemplo, `UnordAF`, `OrdAF` o `OrdDenseAF`), basta con crear una instancia de `MapAF` utilizando dicha fábrica como parámetro. Esto permite que los mapas generados por una instancia de `MapAF` utilicen internamente conjuntos construidos con la misma implementación seleccionada, asegurando coherencia y compatibilidad estructural. Adicionalmente se sobrescriben los métodos de `SetAF` para poder también construir conjuntos a través de la fabrica concreta seleccionada.

2.5. Piecewise maps

Por último, se introducen los *piecewise maps*, los cuales pueden describirse de la siguiente manera:

Un piecewise map se modela como una familia o colección de mapas disjuntos dos a dos, es decir, cuyos dominios son disjuntos entre sí.

Al igual que ocurre con los conjuntos, todos los mapas que componen un *piecewise map* deben tener la misma aridad; esto significa que comparten la misma cantidad de dimensiones, tanto en su dominio como en las expresiones lineales que los definen.

Se denotaran a los *piecewise maps* de la siguiente forma:

$$\ll mp_0,\ mp_2,\ \ldots,\ mp_{k-1} \gg$$

donde cada mp_i representa un mapa individual con la misma aridad, para $i = 0, \ldots, k-1$.

Una notación expandida sería la siguiente:

$$\ll \{[0,1,2],\ [5,1,7]\} \mapsto [x+1], \quad \{[0,1,1],\ [3,1,4]\} \mapsto [x], \quad \ldots, \quad \{[1,1,2],\ [5,1,7]\} \mapsto [x-1] \gg$$

2.5.1. Piecewise maps generales

Nuevamente, se emplea el patrón *delegator* con el objetivo de permitir múltiples implementaciones de *piecewise maps*. Para ello, se define la estructura o tipo abstracto denominado `PWMapDelegate`, la cual especifica la interfaz común que deben seguir las implementaciones concretas, junto con una estructura delegadora llamada `PWMap`.

Al momento de la realización de esta tesina, solo se contaba con una implementación concreta: los *piecewise maps* desordenados, representados como `UnordPWMap`.

Cabe destacar que, aunque cada implementación concreta de *piecewise maps* puede definir internamente su comportamiento de manera distinta, todas heredan de `PWMapDelegate` una variable miembro denominada `fact_`, de tipo `MapAF`. Esta fábrica es utilizada para construir los mapas necesarios durante la ejecución de las operaciones, garantizando que dichos mapas sean coherentes con la implementación concreta de conjuntos seleccionada originalmente.

De este modo, se asegura que todos los mapas del *piecewise map* se construyan utilizando la misma estrategia de representación de conjuntos, preservando la consistencia estructural.

Y al igual que se hizo con conjuntos, se enunciaran las principales operaciones de las cuales van a disponer a través de la interfaz propuesta por `PWMapDelegate`:

- `virtual void emplaceBack(const Map &m) = 0:`
  Agrega un nuevo mapa m a la colección del *piecewise map*.

  Por ejemplo:
  $$emplaceBac(\ll \{[0,1,2],\ [5,1,7]\} \mapsto [x+1], \{[0,1,1],\ [3,1,4]\} \mapsto [x] \gg, \{[1,1,2],\ [5,1,7]\} \mapsto [x-1]) = \ll \{[0,1,2],\ [5,1,7]\} \mapsto [x+1], \{[0,1,1],\ [3,1,4]\} \mapsto [x], \{[1,1,2],\ [5,1,7]\} \mapsto [x-1] \gg.$$

- `virtual bool operator==(const PWMapDelegate &other) const = 0`,
  `virtual bool operator!=(const PWMapDelegate &other) const = 0`: Compara dos *piecewise maps* en base a dos criterios.

  Por ejemplo:
  $$\ll \{[0,1,2],\ [5,1,7]\} \mapsto [x+1], \{[0,1,1],\ [3,1,4]\} \mapsto [x] \gg \texttt{ == } \ll \{[0,1,2],\ [5,1,7]\} \mapsto [x+1], \{[0,1,1],\ [3,1,4]\} \mapsto [x] \gg = \texttt{ true}.$$

- `virtual PWMapDelegPtr operator+(const PWMapDelegate &other) const = 0:`
  Suma dos *piecewise maps*.

  Por ejemplo:

  $$\ll \{[0:1:10],\ [20:1:30]\} \mapsto [1*x+0],\ \{[40:1:50]\} \mapsto [1*x+1] \gg \texttt{ + } \ll \{[25:1:50]\} \mapsto [2*x+0] \gg = \ll \{[25:1:30]\} \mapsto [3*x+0],\ \{[40:1:50]\} \mapsto [3*x+1] \gg .$$

- `virtual PWMapDelegPtr operator-(const PWMapDelegate &other) const = 0:`
  Resta acotada de dos *piecewise maps*.

  Por ejemplo:

  $$\ll \{[0:1:10],\ [20:1:30]\} \mapsto [1*x+0],\ \{[40:1:50]\} \mapsto [4*x+1] \gg \texttt{ - } \ll \{[25:1:50]\} \mapsto [2*x-1] \gg = \ll \{[25:1:30]\} \mapsto [0],\ \{[40:1:50]\} \mapsto [2*x-1] \gg .$$

- `virtual bool isEmpty() const = 0:`
  Verifica si el *piecewise map* es vacío, es decir, $\ll\gg$.

  Por ejemplo: $isEmpty(\ll\gg) =$ `true`.

- `virtual Set dom() const = 0:`
  Devuelve el dominio del *piecewise map*, es decir, la unión de los dominios de todos sus componentes.

  Por ejemplo: $dom(\ll \{[0:1:10],\ [20:1:30]\} \mapsto [1*x+0],\ \{[40:1:50]\} \mapsto [4*x+1] \gg) = \{[0:1:10],\ [20:1:30],\ [40:1:50]\}$.

- `virtual PWMapDelegPtr restrict(const Set &subdom) const = 0:`
  Restringe el dominio del *piecewise map* al subconjunto *subdom*.

  Por ejemplo: $restrict(\ll \{[0:1:10],\ [20:1:30]\} \mapsto [1*x+0],\ \{[40:1:50]\} \mapsto [1*x+1] \gg, \{[25:1:50]\}) = \ll \{[25:1:30]\} \mapsto [1*x+0],\ \{[40:1:50]\} \mapsto [1*x+1] \gg$.

- `virtual Set image() const = 0`,

  `virtual Set image(const Set &subdom) const = 0`:
  Calcula la imagen del *piecewise map*, total o restringida a un dominio.

  Por ejemplo: $image(\ll \{[0:1:10],\ [20:1:30]\} \mapsto [1*x+0],\ \{[40:1:50]\} \mapsto [1*x+1] \gg, \{[0:1:10],\ [20:1:30],\ [40:1:50]\}) = \{[0:1:10],\ [20:1:30],\ [41:1:51]\}$.

- `virtual Set preImage(const Set &subcodom) const = 0:`
  Calcula la preimagen de un subconjunto de la imagen.

  Por ejemplo: $image(\ll \{[0:1:10],\ [20:1:30]\} \mapsto [1*x+0],\ \{[40:1:50]\} \mapsto [1*x+1] \gg, \{[0:1:10],\ [20:1:30],\ [41:1:51]\}) = \{[0:1:10],\ [20:1:30],\ [40:1:50]\}$.

- `virtual PWMapDelegPtr inverse() const = 0:`
  Devuelve la inversa de los mapas del *piecewise map*, si este es biyectivo.

  Por ejemplo: $inverse(\ll \{[0:1:10],\ [20:1:30]\} \mapsto [1*x+10],\ \{[40:1:50]\} \mapsto [1*x+15] \gg) = \ll \{[10:1:20],\ [30:1:40]\} \mapsto [1*x-10],\ \{[55:1:65]\} \mapsto [1*x-15] \gg$.

- `virtual PWMapDelegPtr composition(const PWMapDelegate &pw2) const = 0:`
  Devuelve la composición del *piecewise map* actual con otro *piecewise map*.

Por ejemplo: $composition(\ll \{[1:1:10],\ [20:2:30]\} \mapsto [2*x+1],$
$\{[15:3:18]\} \mapsto [0*x+0] \gg, \ll \{[1:1:30]\} \mapsto [1*x+1] \gg) = \ll \{[1:1:9],\ [19:2:29]\} \mapsto$
$[2*x+3],\ \{[14:3:17]\} \mapsto [0*x+0] \gg .$

- `virtual PWMapDelegPtr concatenation(const PWMapDelegate &other) const = 0:`
  Concatena dos *piecewise maps* con dominio disjuntos.

Por ejemplo: $concatenation(\ll \{[1:1:10],\ [20:2:30]\} \mapsto [2*x+1],$
$\{[15:3:18]\} \mapsto [0*x+0] \gg, \ll \{[50:1:90]\} \mapsto [1*x+1] \gg) =$
$\ll \{[1:1:10],\ [20:2:30]\} \mapsto [2*x+1],\ \{[15:3:18]\} \mapsto [0*x+0],\ \{[50:1:90]\} \mapsto [1*x+1] \gg .$

- `virtual PWMapDelegPtr combine(const PWMapDelegate &other) const = 0:`
  Extiende el *piecewise map* con los elementos exclusivos del dominio de *other*.

Por ejemplo: $combine(\ll \{[1:1:10],\ [20:2:30]\} \mapsto [2*x+1],\ \{[40:3:49]\} \mapsto [0*x+0] \gg$
$, \ll \{[1:1:50]\} \mapsto [1*x+1] \gg =$
$\ll \{[1:1:10],\ [20:2:30]\} \mapsto [2*x+1],\ \{[40:3:49]\} \mapsto [0*x+0],\ \{[11:19],\ [21:2:29],$
$[31:39],\ [41:3:47],\ [42:3:48],\ [50:50]\} \mapsto [1*x+1] \gg .$

- `virtual PWMapDelegPtr reduce(const Interval &i, const LExp &e) const = 0`,
  `virtual PWMapDelegPtr reduce(const Map &sbgmap) const = 0`,
  `virtual PWMapDelegPtr reduce() const = 0:`
  Se encargan de converger un *piecewise map*, sin iterar con respecto a la carnalidad de los conjuntos involucrados.

Por ejemplo: $reduce(\ll \{[100:1:200]\} \mapsto [1*x-1] \gg) =$
$\ll \{[100:1:200]\} \mapsto [0*x+99] \gg .$

- `virtual PWMapDelegPtr minAdjMap(const PWMapDelegate &other) const = 0:`
  Realiza un calculo de mínimos en base a los dos *piecewise maps* que recibe como argumentos.

Por ejemplo: $minAdjMap(\ll \{[1:1:4]\} \mapsto [1*x] \gg,$
$\ll \{[1:1:2]\} \mapsto [0*x+6],\ \{[3:1:4]\} \mapsto [0*x+7] \gg) =$
$ll\{[1:1:2]\} \mapsto [0*x+6],\ \{[3:1:4]\} \mapsto [0*x+7] \gg$

- `virtual PWMapDelegPtr firstInv(const Set &subdom) const = 0`,
  `virtual PWMapDelegPtr firstInv() const = 0:`
  Devuelve la inversa de los mapas del *piecewise map*, restringidas o no en base a un conjunto subdominio.

Por ejemplo: $firstInv(\ll \{[1:1:10]\} \mapsto [0*x+10] \gg, \{[1:1:10]\}) =$
$\ll \{[10:1:10]\} \mapsto [0*x+1] \gg .$

- `virtual Set equalImage(const PWMapDelegate &other) const = 0:`
  Retorna los elementos presentes en ambos dominios que tienen la misma imagen en ambos *piecewise maps*.

Por ejemplo:
$equalImage(\ll \{[1:1:10]\} \mapsto [x] \gg, \ll \{[5:1:15]\} \mapsto [x],\ \{[35:1:45]\} \mapsto [2*x+10] \gg) =$
$\ll \{[5:1:10]\} \mapsto [x] \gg .$

- `virtual PWMapDelegPtr offsetDom(const MD_NAT &off) const = 0`,
  `virtual PWMapDelegPtr offsetDom(const PWMapDelegate &off) const = 0:`
  Desplazan todos los elementos del dominio en una constante o los cambia a través de un *piecewise map*, manteniendo la ley.

Por ejemplo: $offsetDom(\ll \{[1:1:10]\} \mapsto [x+10] \gg, \ll \{[1:1:10]\} \mapsto [x+1] \gg) = \ll \{[2:1:11]\} \mapsto [x+10] \gg .$

- `virtual PWMapDelegPtr compact() const = 0:`
  Compacta en un solo mapa todos los mapas que tengan la misma expresión dentro de un *piecewise map*.

Por ejemplo: $compact(\ll \{[1:1:19],\ [20:1:30]\} \mapsto [x], \{[100:3:200]\} \mapsto [x] \gg) = \ll \{[1:1:30],\ [100:3:200]\} \mapsto [x] \gg .$

Nuevamente se expusieron las operaciones mas relevantes de los *piecewise maps*, en caso de que se desee ver a fondo las demás operaciones, estas se encuentran en el archivo *pw_map.hpp* en la carpeta *sbg*

2.5.2. *Abstract factory* de *piecewise maps*

Al igual que en el caso de los conjuntos, para poder crear una instancia de un *piecewise map* correspondiente a una implementación concreta, la cual será posteriormente compuesta con una instancia del tipo delegador `PWMap`, se hace uso del patrón de diseño *Abstract Factory*. En particular, dicho patrón se encuentra implementado a través de los archivos `af_pwmap.cpp` y `af_pwmap.hpp`, dentro de la biblioteca SBG, disponible en el repositorio [2].

2.5.3. *Piecewise maps* desordenados

A continuación, se presenta la implementación concreta de *piecewise maps* desordenados, que constituye la única implementación disponible definida al momento de la elaboración de esta tesina.

Desde el punto de vista estructural, los *piecewise maps* desordenados están compuestos exclusivamente por una colección de mapas, almacenadas en una variable miembro denominada `pieces_`, la cual es de tipo `vector<Map>`.

Al igual que en el caso de los conjuntos desordenados, a continuación se describen en mayor detalle varias de las operaciones definidas para los *piecewise maps* desordenados. Esto se debe a que, para implementar una versión concreta de *piecewise maps* ordenados, se tomó como punto de partida la implementación ya existente desordenada. Y, detallar dichas implementaciones previas facilita la exposición de las distintas optimizaciones que se proponen más adelante.

2.5.3.1. Igualdad - ==

Esta operación tiene como objetivo verificar si dos *piecewise maps* desordenados son iguales. Para ello, se realiza una comparación exhaustiva entre todos los pares de mapas, tomando uno de cada *piecewise map*.

La verificación se basa en dos posibles criterios: por un lado, si los mapas son iguales por el principio de extensionalidad; y por otro, si ambos disponen de la misma colección de expresiones lineales.

El pseudocódigo correspondiente a esta operación se presenta a continuación.

Algorithm 5 Igualdad de *piecewise maps* desordenados

Require: A y B son dos *piecewise maps* desordenados
Ensure: Devuelve **true** si los mapas son iguales, **false** en caso contrario

```
1:  function ==(A, B)
2:      if DOM(A) ! = DOM(B) then
3:          return false
4:      end if
5:      if A == B then
6:          return true
7:      end if
8:      for all a ∈ A do
9:          for all b ∈ B do
10:             dom_a := DOM(a)
11:             dom_b := DOM(b)
12:             cap_dom := INTERSECTION(dom_a, dom_b)
13:             if ¬ ISEMPTY(cap_dom) then
14:                 if CARDINAL(cap_dom) = 1 then
15:                     exp_a := EXP(a)
16:                     exp_b := EXP(b)
17:                     map_a := cap_dom ↦ exp_a
18:                     map_b := cap_dom ↦ exp_b
19:                     if IMAGE(map_b) ≠ IMAGE(map_b) then
20:                         return false
21:                     end if
22:                 else
23:                     if EXP(a) ≠ EXP(b) then
24:                         return false
25:                     end if
26:                 end if
27:             end if
28:         end for
29:     end for
30:     return true
31: end function
```

2.5.3.2. Suma - +

En lo que respecta a la operación de suma para *piecewise maps* desordenados, esta resulta conceptualmente sencilla.

El enfoque principal consiste en iterar sobre todos los mapas que componen un *piecewise map* A y, para cada uno de ellos, calcular su suma con todos los mapas de otro *piecewise map* B. Como resultado, se construye un nuevo *piecewise map* cuyos elementos corresponden a todas las sumas no vacías entre los pares de mapas tomados de A y B, respectivamente.

Este procedimiento se encuentra exhibido en el pseudocódigo mostrado a continuación en el Algoritmo 6.

Algorithm 6 Suma de *piecewise maps* desordenados

Require: A, B son *piecewise maps* desordenados
Ensure: Devuelve un nuevo *piecewise maps* C con la suma mapa a mapa de A y B

```
1: function +(A, B)
2:     C :=≪≫
3:     for all a ∈ A do
4:         for all b ∈ B do
5:             s := a + b
6:             emplaceBack(C, s)
7:         end for
8:     end for
9:     return C
10: end function
```

2.5.3.3. Resta - –

La operación de resta para *piecewise maps* desordenados presenta una complejidad considerablemente mayor en comparación con la operación de suma. En este caso, la operación realiza una resta acotada entre los mapas que componen los dos *piecewise maps* recibidos como argumentos.

Dicha resta acotada implica que, si el resultado de la operación entre dos mapas es negativo en alguna región de su dominio, se interviene para ajustar el valor de salida en esa sección. En particular, la operación fuerza a que el resultado sea igual a cero en las zonas donde la diferencia es negativa, empleando expresiones del tipo $0 * x + 0$.

El pseudocódigo correspondiente a esta operación se encuentra dividido en tres partes, debido a su extensión y complejidad. Puede consultarse en el Algoritmo 7, 8 y 9. Esta fragmentación responde a la necesidad de cubrir múltiples casos particulares, así como al tratamiento cuidadoso que debe aplicarse al momento de forzar expresiones a cero dentro de los dominios afectados.

Algorithm 7 Resta de *piecewise maps* desordenados: Parte 1: Preparación

Require: A y B son dos *piecewise maps* desordenados
Ensure: Devuelve un *piecewise map* desordenado que representa $A - B$

```
1:  function −(A, B)
2:      C :=≪≫
3:      dom_a := DOM(A)
4:      dom_b := DOM(B)
5:      if ISEMPTY(dom_a) or ISEMPTY(dom_b) then
6:          return C
7:      end if
8:      d := ARITY(A)
9:      all := [0 : 1 : Inf]
10:     univ := {|all|^d}
11:     for all a ∈ A do
12:         for all b ∈ B do
13:             dom_a := DOM(a)
14:             dom_b := DOM(b)
15:             dom := INTERSECTION(dom_a, dom_b)
16:             if ¬ ISEMPTY(dom) then
17:                 e_a := EXP(a)
18:                 e_b := EXP(b)
19:                 minus_exp := MINUS(e_a, e_b)
20:                 ith :=≪ univ ↦ [1 ∗ x + 0]^d
21:                 Parte 2...
22:                 restricted := RESTRICT(ith, dom)
23:                 C := CONCATENATION(C, restricted)
24:             end if
25:         end for
26:     end for
27:     return C
28: end function
```

Algorithm 8 Resta de *piecewise maps* desordenados: Parte 2: Procesamiento

```
1: function −
2:     for j = 0; j < d; j ++ do
3:         idx := minus_exp_j
4:         m := SLOPE(idx)
5:         h := OFFSET(idx)
6:         (begin_neg, end_neg, begin_pos, end_pos) := (0, Inf, 0, Inf)
7:         if m = 0 then
8:             if h < 0 then
9:                 (begin_pos, end_pos) := (1, 0)
10:            else
11:                (begin_neg, end_neg) := (1, 0)
12:            end if
13:        else if m > 0 then
14:            cross := −h/m
15:            if cross ≥ 0 then
16:                bp := TONAT(cross)                ▷ toNat trunca el valor racional
17:                if bp > 0 then
18:                    end_neg := bp − 1
19:                else
20:                    (begin_neg, begin_neg) := (1, 0)
21:                end if
22:            end if
23:        else
24:            cross := −h/m
25:            if cross ≥ 0 then
26:                ep := TONAT(cross)                ▷ toNat trunca el valor racional
27:                if ep > 0 then
28:                    begin_neg := ep + 1
29:                end if
30:            else
31:                (begin_pos, end_pos) := (1, 0)
32:            end if
33:        end if
34:        Parte 3...
35:    end for
36: end function
```

Algorithm 9 Resta de *piecewise maps* desordenados: Parte 3: Análisis y guardado

```
1: function −
2:     jth :=≪≫
3:     for all m ∈ ith do
4:         dset := DOM(m)
5:         mdi := dset_0
6:         e := EXP(m)
7:         negInterval := [begin_neg : 1 : end_neg]
8:         posInterval := [begin_pos : 1 : end_pos]
9:         if ¬ ISEMPTY(negInterval) then
10:            mdi_j := negInterval
11:            e[j] := 0 ∗ x + 0
12:            map := mdi ↦ e
13:            EMPLACEBACK(jth, map)
14:        end if
15:        if ¬ ISEMPTY(posInterval) then
16:            mdi_j := posInterval
17:            e[j] := minus_exp_j
18:            map := mdi ↦ e
19:            EMPLACEBACK(jth, map)
20:        end if
21:    end for
22:    ith := jth
23: end function
```

2.5.3.4. Restricción de dominio - `Restrict`

El funcionamiento de la operación `Restrict` es sencillo: consiste únicamente en restringir el dominio de cada uno de los mapas que conforman un *piecewise map* desordenado A a partir de un conjunto S.

Como resultado, se conservan únicamente aquellos mapas cuyo dominio tiene una intersección no vacía con S.

El funcionamiento puede observarse con claridad en el pseudocódigo mostrado en el Algoritmo 10.

Algorithm 10 Restricción de dominio de *piecewise maps* desordenados

Require: A es un *piecewise map* desordenado y S es un conjunto.
Ensure: Devuelve un nuevo *piecewise map* desordenado con cada mapa de A restringido en su domino por S.

```
1: function RESTRICT(A, S)
2:     C :=≪≫
3:     for all a ∈ A do
4:         ar := RESTRICTMAP(a, S)
5:         EMPLACEBACK(C,ar)
6:     end for
7:     return C
8: end function
```

2.5.3.5. Composición - `composition`

Esta función implementa la operación de **composición** entre dos *piecewise maps* desordenados. Su objetivo es construir un nuevo *piecewise map* que represente la composición punto a punto de todos los mapas individuales de ambos operandos, restringiendo previamente el dominio de uno de ellos.

Este procedimiento puede observarse con claridad en el pseudocódigo del Algoritmo 11, con los *piecewise maps* desordenados A y B como argumentos.

Algorithm 11 Composición de `piecewise maps` desordenados

Require: A y B son *piecewise maps* desordenados.
Ensure: Un nuevo *piecewise map* desordenado C que representa la composición de `A` con `B`

```
1: function COMPOSITION(A, B)
2:     C :=≪≫
3:     im := IMAGE(B)
4:     dom_a := DOM(A)
5:     inter := INTERSECTION(im, dom_a)
6:     new_dom := PREIMAGE(B, inter)
7:     B_restricted := RESTRICT(B, new_dom)
8:     for all a ∈ A do
9:         for all a ∈ B_restricted do
10:            c := COMPOSITION(a, b)
11:            EMPLACEBACK(C, c)
12:        end for
13:    end for
14:    return C
15: end function
```

2.5.3.6. Concatenacion - `concatenation`

Esta operación sobre *piecewise maps* es conceptualmente muy similar a la unión disjunta de conjuntos desordenados, ya que, en esencia, realiza la misma tarea. El objetivo consiste simplemente en unir dos *piecewise maps* disjuntos, es decir, aquellos en los que el dominio de cada mapa de uno no tiene intersección con los dominios de los mapas del otro.

Dado que no es necesario aplicar ninguna función sobre los elementos de los conjuntos, la implementación de esta operación resulta particularmente sencilla, como puede observarse en el pseudocódigo del Algoritmo 12.

Algorithm 12 Concatenación de `piecewise maps` desordenados

Require: A y B son *piecewise maps* desordenados.
Ensure: Un nuevo *piecewise map* desordenado C que representa la concatenación de A con B

```
1: function CONCATENATION(A, B)
2:     C := A
3:     for all b ∈ B do
4:         EMPLACEBACK(C, b)
5:     end for
6:     return C
7: end function
```

2.5.3.7. Combinacion - `combine`

La operación `combine` permite unir dos *piecewise maps* desordenados, A y B. A diferencia de una concatenación directa, esta operación elimina de B cualquier parte de los dominios de sus mapas que intersequen con el dominio de A, garantizando así que el resultado tenga mapas disjuntos dos a dos.

Esta operación es relativamente sencilla, y se puede ver su pseudocódigo en el Algoritmo 13

2.5.3.8. Mínimo adyacente - `minAdjMap`

Ahora bien, se ha llegado a una de las operaciones más abstractas de la librería, dado que su funcionamiento no presenta una relación tan directa con la estructura que representan los *piecewise maps*, como sí ocurre con operaciones como la restricción de dominio o la suma.

En este caso, la operación de mínimo adyacente o `minAdjMap` para *piecewise maps* desordenados se diseño para realizar cálculos en gafos, cuyas aristas se representan mediante dos *piecewise maps*.

Algorithm 13 Combinación de de `piecewise maps` desordenados

Require: A y B son *piecewise maps* desordenados.
Ensure: Devuelve un nuevo *piecewise map* desordenado con los mapas restringidos de B cuyo dominio no interseccione con el dominio de A

```
1:  function COMBINE(A, B)
2:      if ISEMPTY(A) then
3:          return B
4:      end if
5:      if ISEMPTY(B) then
6:          return A
7:      end if
8:      C := A
9:      dom_a := DOM(A)
10:     for all b ∈ B do
11:         dom_b := DOM(b)
12:         new_dom := DIFFERENCE(dom_b, dom_a)
13:         exp_b := EXP(b)
14:         r := new_dom ↦ exp_b
15:         EMPLACEBACK(C,r)
16:     end for
17:     return C
18: end function
```

Como el contexto y el uso de la operación se escapan al objetivo de este escrito, no se tratará de explicar su funcionamiento en detalle. No obstante, para quienes deseen indagar más profundamente en su funcionamiento, se recomienda consultar el repositorio [2], mas concretamente el archivo *pwmap.cpp*; y la tesina de Denise Marzorati [2].

2.5.3.9. Pseudoinversa - `firstInv`

La operación `firstInv`, que recibe un argumento se encarga de calculo de una pseudoinversa del *piecewise map*, restringido a un conjunto dado. En concreto, esta operación computa la pseudoinversa únicamente sobre el mapa cuya imagen sobre el conjunto argumento no sea vacía al quitarle todos los elementos que hayan formado parte de la imagen de los mapas anteriores.

El pseudocódigo correspondiente puede consultarse en el Algoritmo 15.

2.5.3.10. Igualdad de imágenes - `equalImage`

La operación de igualdad de imágenes es relativamente sencilla, y su pseudocódigo puede consultarse en el Algoritmo 16. En él se observa que, para cada par de mapas pertenecientes a los *piecewise maps* desordenados que se reciben como argumento, se calcula la intersección de sus dominios. Si, al considerar dicha intersección como dominio, ambos mapas resultan iguales, entonces esa intersección se conserva como parte del resultado.

2.5.3.11. Desplazamiento de dominio - `offsetDom`

La operación `offsetDom`, en su variante que recibe dos *piecewise maps* desordenados, realiza un desplazamiento del dominio de los mapas del primero en función del segundo.

En términos concretos, dados dos *piecewise maps* desordenados A y B, la operación aplica la imagen de B como desplazamiento sobre el dominio de cada uno de los mapas en A. Es decir, cada dominio de los mapas de A es trasladado según los valores definidos por B, produciendo un nuevo *piecewise map* de salida donde cada uno de los mapas es un mapa de A cuyo domino fue desplazado y no resultó vacío.

Este comportamiento se ilustra en el pseudocódigo del Algoritmo 17, el cual resulta relativamente simple.

Algorithm 14 Minimo adyacente de *piecewise maps* desordenados

Require: A y B son *piecewise maps* desordenados.
Ensure: Devuelve un *piecewise map* desordenado producto de la búsqueda de mínimos

```
1: function minAdjMap(A, B)
2:     C := ≪≫
3:     visited := {}
4:     for all a ∈ A do
5:         for all b ∈ B do
6:             dom_a := dom(a)
7:             dom_b := dom(b)
8:             ith_dom := intersection(dom_a, dom_b)
9:             if ¬ isEmpty(ith_dom) then
10:                e_res                          ▷ Una colección de expresiones lineales
11:                dom_res := image(a, ith_dom)
12:                e_a := exp(a)
13:                im_b := image(b, ith_dom)
14:                if ¬ isConstant(e_a) then
15:                    e_b := exp(b)
16:                    inv := inverse(e_a)
17:                    e_res := composition(e_b, inv)
18:                else
19:                    min := minElem(im_b)
20:                    e_res := [0 * x + min_0, 0 * x + min_1, ..., 0 * x + min_{arity(min)-1}]
21:                end if
22:                if ¬ isEmpty(dom_res) then
23:                    ith := dom_res ↦ e_res
24:                    pw := ≪≫
25:                    again := intersection(dom_res, visited)
26:                    if ¬ isEmpty(again) then
27:                        aux := restrict(res, dom_res)
28:                        min_map := minMap(aux, pw)
29:                        temp := combine(min_map, pw)
30:                        new_res := combine(temp, res)
31:                        C := new_res
32:                        visited := cup(visited, dom_res)
33:                    else
34:                        emplaceBack(C, ith)
35:                        visited := disjointCup(visited, dom_res)
36:                    end if
37:                end if
38:            end if
39:        end for
40:    end for
41:    return C
42: end function
```

Algorithm 15 Pseudoinversa - *piecewise maps* desordenados

Require: A es un *piecewise map* desordenado, S es un conjunto
Ensure: Devuelve un *piecewise map* desordenado cuyos mapas son pseudoinversas de los mapas restringidos de A en base a S

```
1: function firstInv(A, S)
2:     C := ≪≫
3:     visited := {}
4:     for all a ∈ A do
5:         img := image(a, S)
6:         res_dom := difference(img, visited)
7:         if ¬ isEmpty(res_dom) then
8:             pre := preImage(a, res_dom)
9:             exp := exp(a)
10:            map := pre ↦ exp
11:            inv_map := minInv(map)
12:            emplaceBack(C, inv_map)
13:            img_s := image(a, S)
14:            visited := cup(visited, img_s)
15:        end if
16:    end for
17:    return C
18: end function
```

Algorithm 16 Imagen igual de mapas pieza a pieza desordenados

Require: A y B son dos *piecewise maps* desordenados
Ensure: Devuelve un conjunto con las regiones donde A y B tienen la misma imagen en base a su dominio

```
1: function equalImage(A, B)
2:     f := fact(A)
3:     C := {}_⟨f⟩
4:     for all a ∈ A do
5:         for all b ∈ B do
6:             dom_a := dom(a)
7:             dom_b := dom(b)
8:             cap_dom := intersection(dom_a, dom_b)
9:             if ¬ isEmpty(cap_dom) then
10:                m_a_cap := cap_dom ↦ exp(a)
11:                m_b_cap := cap_dom ↦ exp(b)
12:                if m_a_cap = m_b_cap then
13:                    C := disjointCup(C, cap_dom)
14:                end if
15:            end if
16:        end for
17:    end for
18:    return C
19: end function
```

Algorithm 17 Desplazamiento de dominio de *piecewise maps* desordenados

Require: A y O son dos *piecewise maps* desordenados
Ensure: Devuelve un *piecewise map* desordenado con cada mapa de A cuyo dominio ha sido desplazado por O

```
1: function offsetDom(A, O)
2:     C := ≪≫
3:     for all a ∈ A do
4:         dom_a := dom(a)
5:         ith_dom := image(O, dom_a)
6:         exp_m := exp(a)
7:         map := ith_dom ↦ exp_a
8:         emplaceBack(C, map)
9:     end for
10:    return C
11: end function
```

2.5.3.12. Reducción - `reduce`

El caso de la operación `reduce` se mencionará principalmente por una optimización que se analizará en detalle más adelante, pero que no forma parte de las optimizaciones planteadas para las operaciones de los *piecewise maps* ordenados. En esencia, `reduce` está compuesta por tres funciones, cada una con variaciones en sus argumentos, las cuales se llaman en cadena. En este caso, la versión sin argumentos llama a la versión que recibe un mapa como argumento, la cual a su vez llama a la tercera versión que recibe un intervalo y una expresión lineal.

Sin embargo, para este escrito es de interés únicamente la versión sin argumentos, que se presenta en el pseudocódigo del Algoritmo 18.

El objetivo de esta operación es "componer" hasta converger, aunque no mediante la operación de composición de mapas o de *piecewise maps* desordenados.

Algorithm 18 Reducción de *piecewise maps* desordenados

Require: A es un *piecewise map* desordenado
Ensure: Devuelve un nuevo *piecewise map* desordenado reducido
1: **function** REDUCE(A)
2: $f :=$ FACT(A)
3: $C :=\ll\gg$
4: **for all** $a \in A$ **do**
5: $ith :=$ REDUCE(A, a)
6: $C :=$ CONCATENATION(C, ith)
7: **end for**
8: **return** C
9: **end function**

2.5.3.13. Compactación - `compact`

La operación `compact` tiene un propósito bastante claro: actuar como un mecanismo de reducción en la cantidad de mapas dentro de un *piecewise map* desordenado. Esta reducción se logra unificando los dominios de aquellos mapas que comparten exactamente la misma colección de expresiones lineales. El pseudocódigo correspondiente puede verse en el Algoritmo 19.

Algorithm 19 Compactación de *piecewise maps* desordenados

Require: A es un *piecewise map* desordenado
Ensure: Devuelve un *piecewise maps* desordenados con los mapas compactados

```
1: function COMPACT(A)
2:     C := ≪≫
3:     dom := DOM(A)
4:     if ISEMPTY(dom) then
5:         return C
6:     end if
7:     compacted := {}
8:     n := SIZE(A)
9:     for i := 0; i < n; i := i + 1 do
10:        m_i := A_i
11:        i := i + 1
12:        dom_i := DOM(m_i)
13:        ith_compacted := INTERSECTION(compacted, dom_i)
14:        if ISEMPTY(ith_compacted) then
15:            comp_dom := COMPACT(dom_i)
16:            exp_i := EXP(m_i)
17:            new_ith := comp_dom ↦ exp_i
18:            for j := i + 1; j < n; j := j + 1 do
19:                m_j := A_j
20:                dom_j := DOM(m_j)
21:                next_compacted := INTERSECTION(compacted, dom_j)
22:                if ISEMPTY(next_compacted) then
23:                    maybe := COMPACT(new_ith, m_j)
24:                    if maybe then
25:                        new_ith := VALUE(maybe)
26:                        compacted := CUP(compacted, dom_j)
27:                    end if
28:                end if
29:            end for
30:            EMPLACEBACK(C, new_ith)
31:        end if
32:    end for
33:    return C
34: end function
```

Capítulo 3

Optimizaciones para el desorden

Dado que tanto la implementación de los *piecewise maps* como la de los conjuntos ordenados se basan en sus respectivas versiones desordenadas, ambas fueron revisadas y analizadas previamente y durante el desarrollo de las implementaciones ordenadas. Como resultado de este análisis, en la implementación de los *piecewise maps* desordenados se identificaron varias funciones con un alto potencial de optimización, entre ellas: `composition`, `firstInv`,-, `reduce` y `compact`. En contraste, otras funciones solo admitieron mejoras mínimas. Cabe destacar que las optimizaciones mencionadas anteriormente fueron posteriormente incorporadas en la implementación ordenada de los *piecewise maps*, dado que, como se señaló, los *piecewise maps* desordenados constituyeron el punto de partida para su desarrollo.

Nuevamente, toda la notación empleada a lo largo de este capítulo, así como en el pseudocódigo, puede consultarse en el **Apéndice A**, donde se recopila la notación utilizada en esta tesina junto con su correspondiente significado.

3.1. Optimización de la inserción de mapas no vacíos

En muchas de las operaciones disponibles para los *piecewise maps* desordenados se realizan diversos cálculos para obtener los mapas que deben incorporarse en el resultado final. En la mayoría de los casos, dentro de dichas operaciones es necesario verificar que el mapa no sea vacío (es decir, que su dominio no sea vacío). Sin embargo, en un pequeño porcentaje de las operaciones esta verificación resulta innecesaria, ya que, en función de los procesos que lleva a cabo, se puede asegurar que el mapa no será vacío.

A pesar de ello, esta situación no se aprovecha para omitir la verificación, dado que siempre que se inserta un mapa se emplea la operación `emplaceBack`, la cual comprueba que el dominio del mapa no sea vacío.

Con el fin de mejorar este aspecto, se implementó una operación adicional, `pushBack`, que inserta mapas sin realizar dicha verificación. Las operaciones modificadas para utilizar `pushBack` son: `reduce` en todas sus variantes, `firstInv` que recibe un conjunto, `offsetDom` que recibe un natural multi-dimensional offset, `compact` y `offsetImage` en todas sus versiones.

Es importante señalar que este cambio introduce una optimización prácticamente nula de manera aislada; no obstante, mientras más optimizadas estén las operaciones en su conjunto, mayor será el beneficio acumulado. Finalmente, no se presentará el pseudocódigo de estas operaciones, dado que la modificación es mínima. Las implementaciones correspondientes pueden consultarse en el repositorio de GitHub, en el archivo *pwmap.cpp*, dentro de la carpeta *sbg* [2].

3.2. Optimizando `reduce`

Esta operación, en particular la variante sin argumentos, fue incluida entre las operaciones que debían ser implementadas por las diferentes versiones de *piecewise maps*. La versión inicial sin optimizaciones de la función puede verse en el pseudocódigo del Algoritmo 18, mientras que su versión modificada y optimizada se presenta

en el Algoritmo 20.

En particular, se omitió la operación `concatenation`, ya que, si bien tiene un costo lineal en función de la suma de los tamaños de las *piecewise maps* involucrados, su diseño obliga a copiar todos los elementos constantemente. Esto se debe a que la operación no modifica los *piecewise maps* que recibe como argumentos. Como consecuencia, al repetirse múltiples veces, su costo acumulado resulta muy elevado. Por esta razón, se decidió prescindir de su uso, como puede observarse en el pseudocódigo del Algoritmo 20.

Algorithm 20 Reducción de *piecewise maps* desordenados optimizada

Require: A es un *piecewise map* desordenado
Ensure: Devuelve un nuevo *piecewise map* desordenado reducido

```
1:  function REDUCE(A)
2:      C := ≪≫
3:      for all a ∈ A do
4:          ith := REDUCE(A, a)
5:          for all i ∈ ith do
6:              pushBack(C, i)
7:          end for
8:      end for
9:      return C
10: end function
```

3.3. Optimizando -

De manera análoga a la operación `reduce`, el operador `-` para los *piecewise maps* desordenados puede optimizarse omitiendo la operación de `concatenation`. En el pseudocódigo del Algoritmo 21 se muestra esta modificación. Cabe aclarar que únicamente se presenta la parte del pseudocódigo que efectivamente cambió, mientras que todas las demás partes permanecen iguales.

3.4. Optimizando `composition`

Esta operación ya fue tratada en la sub-sección correspondiente a los *piecewise maps* desordenados, y su pseudocódigo puede verse en el Algoritmo 11. El problema principal con esta función es que repite dos veces el mismo proceso: calcula una restricción sobre el dominio del segundo argumento, basada en la intersección entre la imagen del segundo argumento y el dominio del primero. Sin embargo, este procedimiento ya se lleva a cabo durante la composición entre los mapas. Esto puede observarse en el archivo `map.cpp`, dentro del módulo *sbg*, disponible en el repositorio.

Como conclusión, la restricción previa a la composición de mapas resulta innecesaria. Para mejorar la legibilidad y evitar duplicación de procedimientos, se decidió dejar esta responsabilidad en manos de la implementación de mapas, eliminando así la restricción explícita presente en el Algoritmo 11, quedando la función tal como se muestra en el Algoritmo 22.

3.5. Optimizando `compact`

Nuevamente se trata de una operación que se desarrolló en el Capítulo 2. El pseudocódigo correspondiente se muestra en el Algoritmo 19. La operación, en esencia, busca combinar en un solo mapa todos aquellos que comparten la misma expresión.

Sin embargo, como puede observarse, los mapas de A vuelven a ser recorridos incluso después de haber sido compactados. Esto, sumado a las operaciones de conjuntos necesarias para evitar rehacer compactaciones (por ejemplo mediante el conjunto `compacted`), convierte a la operación en una de alto costo computacional.

Como consecuencia, se desarrolló una versión optimizada, presentada en el pseudocódigo del Algoritmo 23.

Algorithm 21 Resta de *piecewise maps* desordenados optimizada: Parte 1: Preparación

Require: A y B son dos *piecewise maps* desordenados
Ensure: Devuelve un *piecewise map* desordenado que representa $A - B$

```
 1: function −(A, B)
 2:     C :=≪≫
 3:     dom_a := DOM(A)
 4:     dom_b := DOM(B)
 5:     if ISEMPTY(dom_a) or ISEMPTY(dom_b) then
 6:         return C
 7:     end if
 8:     d := ARITY(A)
 9:     all := [0 : 1 : Inf]
10:     univ := {|all|^d}
11:     for all a ∈ A do
12:         for all b ∈ B do
13:             dom_a := DOM(a)
14:             dom_b := DOM(b)
15:             dom := INTERSECTION(dom_a, dom_b)
16:             if ¬ ISEMPTY(dom) then
17:                 e_a := EXP(a)
18:                 e_b := EXP(b)
19:                 minus_exp := MINUS(e_a, e_b)
20:                 ith :=≪ univ ↦ [1 ∗ x + 0]^d
21:                 Parte 2...
22:                 restricted := RESTRICT(ith, dom)
23:                 C := C ⌢ restricted
24:             end if
25:         end for
26:     end for
27:     return C
28: end function
```

Algorithm 22 Composición de `piecewise maps` desordenados optimizada

Require: A y B son *piecewise maps* desordenados.
Ensure: Un nuevo *piecewise map* desordenado C que representa la composición de `A` con `B`

```
 1: function COMPOSITION(A, B)
 2:     C :=≪≫
 3:     for all a ∈ A do
 4:         for all b ∈ B do
 5:             c := COMPOSITION(a, b)
 6:             EMPLACEBACK(C, c)
 7:         end for
 8:     end for
 9:     return C
10: end function
```

En esta versión se utiliza una lista simplemente enlazada, lo que permite evitar múltiples recorridos sobre los elementos ya compactados, así como descartar de forma eficiente aquellos que ya han sido procesados. Adicionalmente, se elimina una parte significativa de las operaciones de conjuntos, reduciendo considerablemente el costo de la operación.

Algorithm 23 Compactación de *piecewise maps* desordenados optimizada

Require: A es un *piecewise map* desordenado
Ensure: Devuelve un *piecewise maps* desordenados con los mapas compactados

```
 1: function COMPACT(A)
 2:     C := ≪≫
 3:     dom := DOM(A)
 4:     if ISEMPTY(dom) then
 5:         return C
 6:     end if
 7:     size := SIZE(A)
 8:     indices := []                          ▷ Es una lista simplemente enlazada
 9:     for i = 0; i < size; i := i + 1 do
10:         indices := indices ++ [i]
11:     end for
12:     i := 0
13:     while i ≠ LENGTH(indices) do
14:         m := A_indices[i]
15:         dom_m := DOM(m)
16:         dom_comp := COMPACT(dom_m)
17:         exp_val := EXP(m)
18:         new_map := dom_comp → exp_val
19:         indices := indices ◁ i
20:         j := i + 1
21:         while j ≠ LENGTH(indices) do
22:             next := A_indices[j]
23:             optional := COMPACT(new_map, next)
24:             if optional then
25:                 new_map := value(optional)
26:                 indices := indices ◁ j
27:                 continue
28:             end if
29:             j ++
30:         end while
31:         pushBack(C, new_map)
32:         i ++
33:     end while
34:     return C
35: end function
```

3.6. Optimizando `firstInv`

Ahora es el turno de la operación `firstInv`. Esta se encarga de mantener un conjunto de valores ya visitados, denominado *visited*, como se muestra en el pseudocódigo del Algoritmo 15. Sin embargo, la forma en que se almacenan los elementos dentro de este conjunto es un aspecto que podría optimizarse.

En la versión original, se vuelve a calcular la imagen del mapa a a partir del conjunto de entrada y luego se combina con *visited* mediante la operación de unión de conjuntos. Este procedimiento recalcula un conjunto que ya está disponible y obliga a utilizar la operación `cup`, ya que los conjuntos no son disjuntos.

Una optimización posible consiste en emplear directamente *res_dom*, que contiene todos los elementos de la imagen que, además, no forman parte de *visited*. De este modo, es posible utilizar la operación `disjointCup` para unir los conjuntos, aprovechando que son disjuntos.

El código modificado y optimizado se presenta en el pseudocódigo del Algoritmo 24.

Algorithm 24 Pseudoinversa - *piecewise maps* desordenados

Require: A es un *piecewise map* desordenado, S es un conjunto
Ensure: Devuelve un *piecewise map* desordenado cuyos mapas son pseudoinversas de los mapas restringidos de A en base a S

```
1: function FIRSTINV(A, S)
2:     C := ≪≫
3:     visited := {}
4:     for all a ∈ A do
5:         img := IMAGE(a, S)
6:         res_dom := DIFFERENCE(img, visited)
7:         if ¬ ISEMPTY(res_dom) then
8:             pre := PREIMAGE(a, res_dom)
9:             exp := EXP(a)
10:            map := pre → exp
11:            inv_map := MININV(map)
12:            PUSHBACK(C, inv_map)
13:            visited := DISJOINTCUP(visited, res_dom)
14:        end if
15:    end for
16:    return C
17: end function
```

Capítulo 4

Conjuntos ordenados

Una vez establecidos los conceptos necesarios, se introducen a continuación los **conjuntos ordenados**. El propósito de este capítulo es explicar cómo estos conjuntos almacenan a los multi-intervalos, presentar las distintas optimizaciones que se obtuvieron para estos, denominadas *criterios de optimización* y *criterios de ordenamiento* y, finalmente, detallar cómo dichas optimizaciones fueron incorporadas en las operaciones de los conjuntos ordenados.

4.1. Estructura y orden

Como se mencionó previamente en la sub-sección dedicada a los conjuntos ordenados densos, estos utilizaban el operador $<$ definido para los multi-intervalos para dotarse de orden. Los **conjuntos ordenados** utilizan el mismo criterio. Por lo tanto, la estructura de los conjuntos ordenados, denominados `OrderedSet`, incluye la siguiente variable miembro:

- `pieces_` (de tipo **MDIOrdSet**): Representa el conjunto en sí mismo.

Ahora bien, a diferencia de los conjuntos ordenados densos, que operaban sobre una única dimensión, los conjuntos ordenados trabajan sobre múltiples dimensiones. En consecuencia, el operador $<$ definido para el tipo `MD_NAT` ya no se limita a una comparación tradicional entre valores numéricos, sino que ya trabaja a nivel dimensional como se mostró en la Sub-sección 2.3.3.

Supóngase que se trabaja con multi-intervalos bidimensionales, donde cada uno posee su correspondiente mínimo. Sean entonces por ejemplo los siguientes tres multi-intervalos:

- $mdi_1 = [2:1:2] \times [3:2:9]$, con mínimo $(2,\ 3)$
- $mdi_2 = [1:1:2] \times [5:2:9]$, con mínimo $(1,\ 5)$
- $mdi_3 = [2:1:2] \times [1:2:9]$, con mínimo $(2,\ 1)$

Aplicando el criterio de orden basado en los mínimos, con el operador $<$ definido para los multi-intervalos, se obtiene el siguiente ordenamiento:

$$mdi_2 < mdi_3 < mdi_1$$

ya que:

$$(1,\ 5) < (2,\ 1) < (2,\ 3)$$

Por lo tanto, el conjunto ordenado resultante será:

$$\{mdi_2,\ mdi_3,\ mdi_1\}$$

4.2. *Abstarct factory*

Al igual que se hizo con los conjuntos desordenados y ordenados densos, también se definió la fábrica concreta para los conjuntos ordenados, para poder aplicar el patrón *Abstarct factory*.

En particular, la fábrica concreta de los conjuntos ordenados se denominó `OrdAF`. Esta puede encontrarse en los archivos *af_set*, tanto en su versión *.cpp* como *.hpp*, dentro de la carpeta `sbg` del repositorio.

4.3. Criterios de optimización y ordenamiento

En esta sección se describen en detalle los diferentes **criterios de optimización**, así como los **criterios de ordenamiento** empleados en las operaciones para conjuntos ordenados. Ambos conjuntos de criterios se aplicarán posteriormente con el objetivo de mejorar la eficiencia general de las operaciones y reducir significativamente los tiempos de ejecución.

Ahora bien, para que todo este claro, un *criterio de optimización* es, en esencia, un predicado que permite mejorar el rendimiento de una operación, aprovechando una o varias propiedades de una estructura de datos y de los elementos que esta contiene. Mientras un *criterio de ordenamiento* es una regla que define cómo organizar de manera eficiente los elementos dentro de una estructura de datos para que se cumpla un cierto orden.

Pseudocódigo y notación: A partir de este momento, en este capítulo, al igual que se hizo para los conjuntos desordenados, se emplearán subíndices para referirse a los multi-intervalos de un conjunto ordenado. Dado un conjunto ordenado A, el elemento ubicado en la posición i, lo cual corresponde a `pieces_[i]` en C++, se denotará como A_i, donde i es un número natural que satisface $0 \leq i < \kappa(A)$, siendo $\kappa(A)$ la aridad del conjunto A, es decir, la cantidad total de elementos que contiene. Cabe destacar que, al tratarse de un conjunto ordenado, siempre se cumple que $A_i < A_j$ si y sólo si $i < j$, para todo par de i y j tal que $0 \leq i, j < \kappa(A)$. Adicionalmente se dirá que, en el caso anterior, A_i está **antes** de A_j en A, mientras A_j está **después** de A_i. Toda esta notación, adicionalmente, se puede encontrar dentro del **Apéndice A**.

4.3.1. Intersección - `intersection`

La operación `intersection` constituye un componente central dentro de la batería de operaciones fundamentales para conjuntos. Antes de presentar las optimizaciones específicas y su correspondiente criterio de ordenamiento, es conveniente recordar el funcionamiento general de la intersección de conjuntos:

Para intersecar dos conjuntos se debe considerar que cada conjunto está compuesto por multi-intervalos. Por lo tanto, la operación consiste en evaluar todas las posibles intersecciones de los multi-intervalos de uno de los conjuntos con los del otro conjunto participante .

Esto se puede ver en el pseudocódigo de la operación `intersección` para conjuntos desordenados, en el Algoritmo 1.

4.3.1.1. Criterios de optimización

Criterio de parada

Supóngase que se lleva a cabo la intersección entre A y B, dos conjuntos ordenados, y se están considerando las posibles intersecciones del multi-intervalo i-ésimo de A, A_i, con todos los multi-intervalos de B, cuya representación grafica se puede observar en la Figura 4.1.

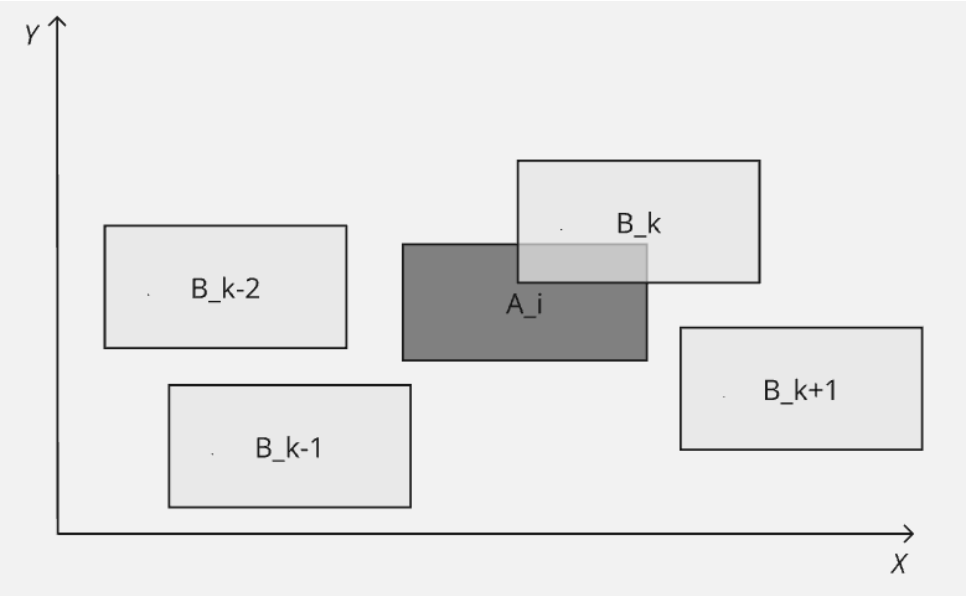


Figura 4.1: Ejemplificación de una iteración del proceso de intersección sobre dos conjuntos A y B.

En algún punto, puede que se alcance un valor k, con $0 \leq k < \kappa(B)$, tal que el elemento mínimo de B_k sea estrictamente mayor, en su primera dimensión (dimensión 0), que el elemento máximo de A_i, tal como se muestra en la Figura 4.2. Por ejemplo, si el mínimo de B_k es $(3, 5)$ y el máximo de A_i es $(2, 3)$. En este caso, se pueden derivar las siguientes conclusiones:

- La intersección entre A_i y B_k resulta vacía, ya que todos los valores de A_i tienen un valor menor o igual que el máximo de A_i en la primera dimensión y todos los valores de B_k tienen un valor mayor o igual que el mínimo de B_k en la primera dimensión.
- Debido al orden creciente de los conjuntos por el operador $<$ de multi-intervalos, cualquier multi-intervalo $B_{k'}$ con $k < k' < \kappa(B)$ tendrá en la primera dimensión de su mínimo un valor mayor o igual que el mínimo de B_k en dicha dimensión, lo cual garantiza que también será vacía su intersección con A_i.

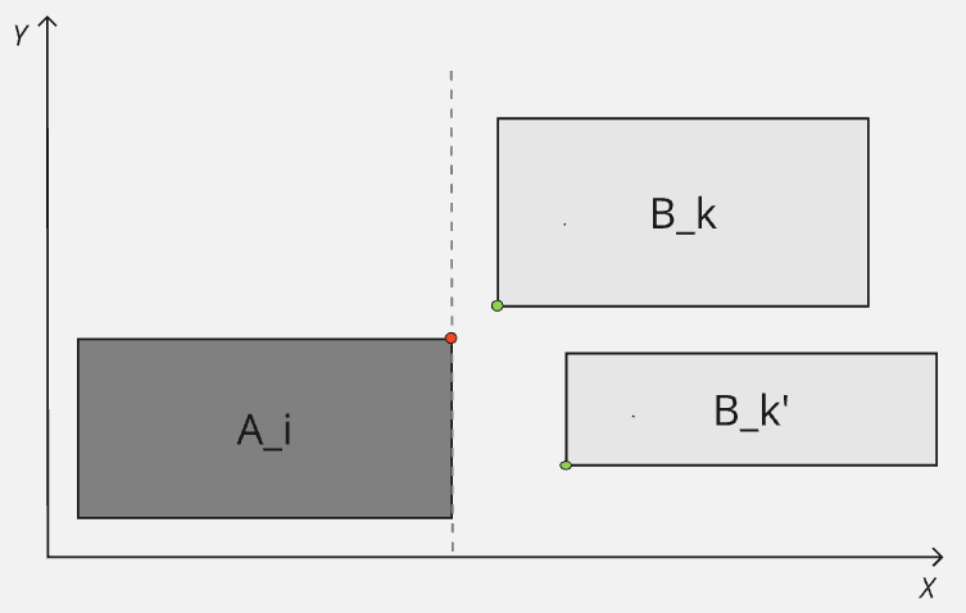


Figura 4.2: Criterio de parada.

En consecuencia, se puede establecer el siguiente criterio fundamental para optimizar la intersección entre conjuntos ordenados:

Criterio de parada

Sean A y B dos conjuntos ordenados. Supóngase que se está evaluando la intersección entre ambos, y en particular se consideran las posibles intersecciones entre un multi-intervalo A_i de A, con $0 \leq i < \kappa(A)$, y los multi-intervalos de B. Dado un índice k tal que $0 \leq k < \kappa(B)$, vale lo siguiente:

Si el máximo de A_i es estrictamente menor que el mínimo de B_k en la primera dimensión, **entonces**:

- la intersección entre A_i y $B_{k'}$ resulta vacía, $\forall k' \mid k \leq k' < \kappa(B)$,
- **y, entonces** puede continuarse directamente con la evaluación de las intersecciones entre A_{i+1}(si existe) y los elementos de B.

Criterio de eliminación

Nuevamente, supóngase que se lleva a cabo la intersección entre A y B, dos conjuntos ordenados, y se están considerando las posibles intersecciones del multi-intervalo i-ésimo de A, A_i, con todos los multi-intervalos de B.

En algún punto, puede que se alcance un valor k, con $0 \leq k < \kappa(B)$, tal que el elemento máximo de B_k sea estrictamente menor, en su primera dimensión (dimensión 0), que el elemento mínimo de A_i, tal como se muestra en la Figura 4.3. Por ejemplo, si el mínimo de A_i es $(3, 5)$ y el máximo de B_k es $(2, 3)$. En este caso, se pueden derivar las siguientes conclusiones:

- La intersección entre A_i y B_k resulta vacía, ya que todos los valores de A_i tienen un valor mayor o igual que el mínimo de A_i en la primera dimensión y todos los valores de B_k tienen un valor menor o igual que el máximo de B_k en la primera dimensión.
- Debido al orden creciente de los conjuntos por el operador $<$ de multi-intervalos, cualquier multi-intervalo $A_{i'}$ con $i < i' < \kappa(A)$ tendrá en la primera dimensión de su mínimo un valor mayor o igual que el mínimo de A_i en dicha dimensión, lo cual garantiza que también será vacía su intersección con B_k.

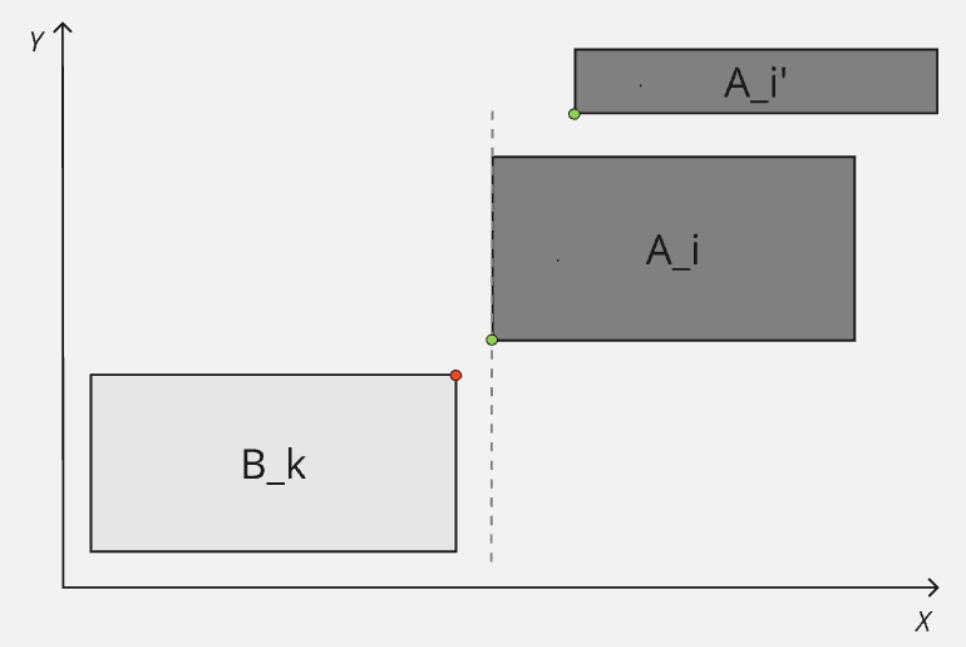


Figura 4.3: Criterio de eliminación.

En consecuencia, se puede establecer el siguiente criterio fundamental para optimizar la intersección entre conjuntos ordenados:

Criterio de eliminación

Sean A y B dos conjuntos ordenados. Supóngase que se está evaluando la intersección entre ambos, y en particular se consideran las posibles intersecciones entre un multi-intervalo A_i de A, con $0 \leq i < \kappa(A)$, y los multi-intervalos de B. Dado un índice k tal que $0 \leq k < \kappa(B)$, vale lo siguiente:

Si el máximo de B_k es estrictamente menor que el mínimo de A_i en la primera dimensión, **entonces**:

- la intersección entre $A_{i'}$ y B_k resulta vacía, $\forall i' \mid i \leq i' < \kappa(A)$,
- **y, entonces** puede descartarse B_k para los cálculos de las posibles intersecciones de los multi-intervalos posteriores a A_i.

Cabe señalar que, tanto para este criterio como para el anterior, no se especifica el tipo de multi-intervalo (si es denso o no), ya que dicha característica no afecta directamente la validez de los mismos.

Criterio de selección

Tal como se analizó en el criterio de eliminación, el procedimiento de intersección considera cada multi-intervalo del conjunto A e intenta intersectarlo con todos los multi-intervalos del conjunto B. Y donde la idea central del criterio es que, a medida que se avanza en este proceso, ciertos elementos de B pueden ser descartados progresivamente si se determina que ya no podrán intersectar con los siguientes elementos de A.

Naturalmente, se puede llegar al punto en el cual todos los elementos de B habrán sido descartados, lo que implica que la operación `intersection` podrá finalizar directamente ya que no quedan intersecciones posibles a verificar.

Ahora bien, existe una observación importante respecto al tamaño de B: cuanto menor sea la cantidad de multi-intervalos en B, más rápido podrán descartarse todos ellos posiblemente, y por tanto, terminar la operación.

En base a esta idea, se introduce el **criterio de selección**, el cual establece lo siguiente:

Criterio de selección

Sean dos conjuntos ordenados involucrados en la operación de intersección. Se establece lo siguiente:

Se define como B a aquel conjunto que contiene la menor cantidad de multi-intervalos, mientras que se denota como A al conjunto restante.

Criterio de solapamiento

Se introduce ahora el **criterio de solapamiento**, el cual establece lo siguiente:

Criterio de solapamiento

Sean A y B dos conjuntos, A_i un multi-intervalo del conjunto A y B_k un multi-intervalo del conjunto B, con índices tales que:

$$0 \leq i < \kappa(A), \quad 0 \leq k < \kappa(B).$$

Se establece entonces que, en el caso de realizar la intersección entre A y B:

- **Si** existe solapamiento entre A_i y B_k, **entonces**:
  - **Si** ambos multi-intervalos son *densos*, la intersección entre ellos es necesariamente no vacía y debe proceder.
  - **Si** al menos uno de ellos no es denso, la intersección *puede* ser no vacía, pero no se garantiza de modo que se debe proceder de igual manera.
- **Si no** existe solapamiento entre A_i y B_k, **entonces** la intersección entre ellos es necesariamente vacía, independientemente de si son densos o no, y por ende puede obviarse su cálculo.

Ahora bien, seguiría explicar que es el solapamiento en si:

*Sean dos multi-intervalos mdi_1 y mdi_2. Los multi-intervalos mdi_1 y mdi_2 se **solapan si, y sólo si**, todos sus intervalos componentes en cada dimensión se solapan.*

*Sean $a = [a_{begin} : a_{step} : a_{end}]$ y $b = [b_{begin} : b_{step} : b_{end}]$ dos intervalos. Se dice que se **solapan si, y sólo si**, se cumple la siguiente condición:*

$$\neg(b_{\text{end}} < a_{\text{begin}} \vee a_{\text{end}} < b_{\text{begin}})$$

o bien las siguientes dos condiciones:

$$\bullet\ b_{\text{end}} \geq a_{\text{begin}} \text{ y } \bullet\ a_{\text{end}} \geq b_{\text{begin}}$$

Para ejemplificar mejor la noción de solapamiento en intervalos se proponen los siguientes casos en base al predicado presentado por la definición:

- **Caso $b_{\text{end}} < a_{\text{begin}}$ y $a_{\text{end}} \geq b_{\text{begin}}$:**

  En este caso, no hay solapamiento. Por ejemplo, si $b = [1 : 1 : 5]$ y $a = [6 : 2 : 10]$, entonces $5 < 6$, por lo tanto, no se solapan. Dicho ejemplo se puede ver en la Figura 4.4.

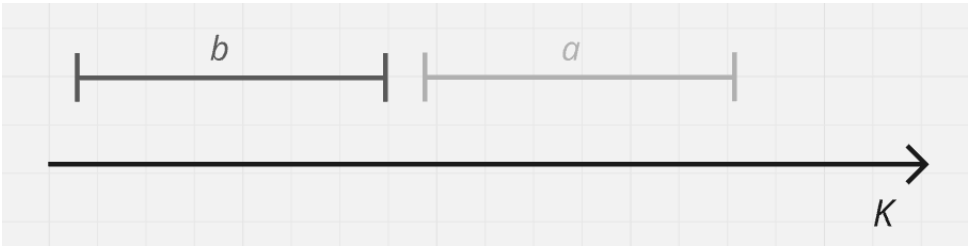


Figura 4.4: Solapamiento sobre intervalos: *segunda condición verdadera.*

- **Caso $b_{\text{end}} \geq a_{\text{begin}}$ y $a_{\text{end}} < b_{\text{begin}}$:**

  Nuevamente, no hay solapamiento. Por ejemplo, si $b = [8 : 2 : 10]$ y $a = [4 : 2 : 6]$, entonces $6 < 8$. Este caso se puede observar en la Figura 4.5.

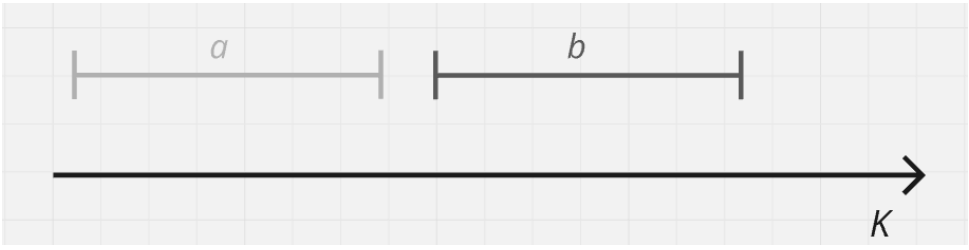
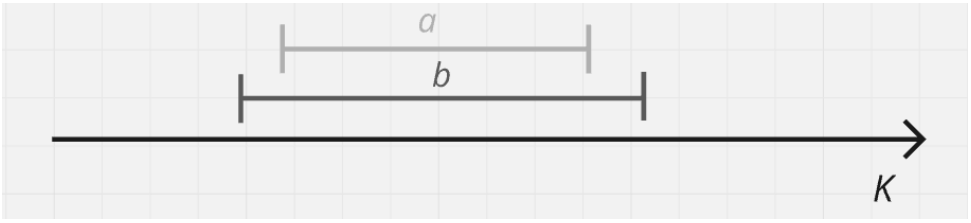


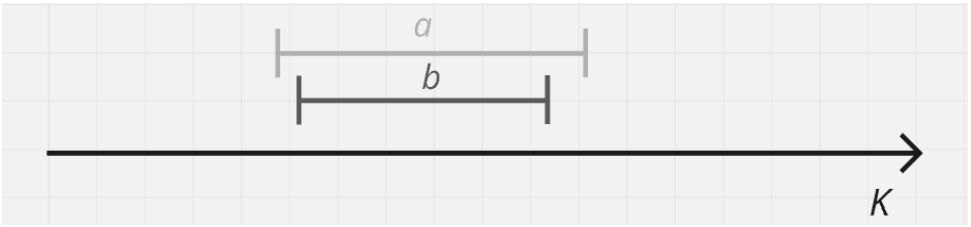
Figura 4.5: Solapamiento sobre intervalos: *primera condición verdadera.*

- **Caso ambas condiciones se cumplen:**

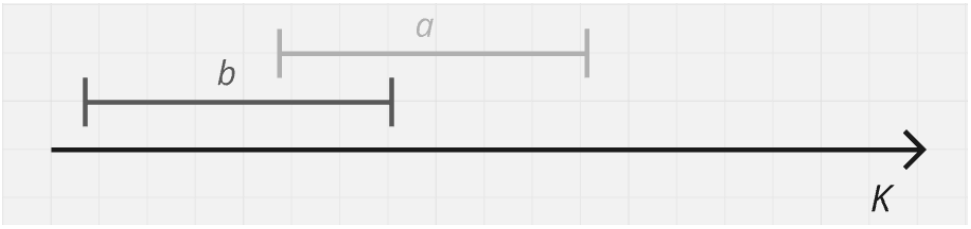
  En este caso resulta que se pueden dar dos situaciones unicamente: b_{begin} o b_{end} se encuentran entre a_{begin} y a_{end} inclusive (Figura 4.6b, Figura 4.6c y Figura 4.6d), ó $b_{begin} < a_{begin}$ y $b_{end} > a_{end}$ (Figura 4.6a).



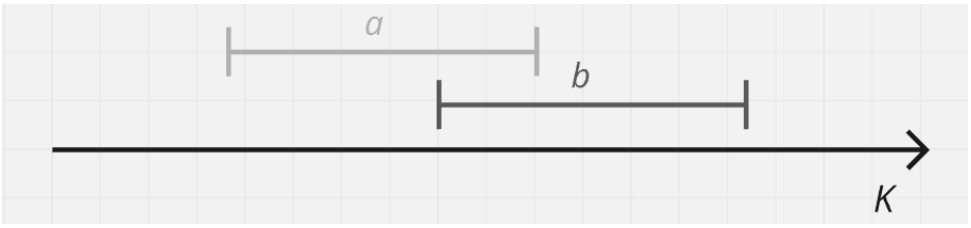
(a)



(b)



(c)



(d)

Figura 4.6: Solapamiento sobre intervalos: *ambas condiciones se cumplen.*

- **Caso ambas condiciones son falsas:**

  Cabe destacar que el caso en el cual **ambas condiciones** sean falsas simultáneamente es imposible. En efecto, si se cumple que:

  $$b_{\text{end}} < a_{\text{begin}} \quad \text{y} \quad a_{\text{end}} < b_{\text{begin}},$$

  entonces, por $a_{\text{begin}} < a_{\text{end}}$ y transitividad de la desigualdad, se deduce que:

  $$b_{\text{end}} < b_{\text{begin}},$$

  lo cual implicaría que el intervalo b está mal definido, ya que su extremo final es menor que su extremo inicial. Esto contradice la definición válida de intervalo, por lo tanto, dicho caso **no** puede ocurrir.

Por un lado, se puede observar que si, en al menos una dimensión, los intervalos correspondientes de dos multi-intervalos mdi y mdi', ya sean densos o no, no se solapan, entonces no existe ninguna n-upla $(mdi[0], \ldots, mdi[n-1]) \in mdi$ y $(mdi'[0], \ldots, mdi'[n-1]) \in mdi'$ tal que coincidan en esa dimensión, con n siendo la aridad de mdi y mdi'. En consecuencia, esto implica que la intersección entre mdi y mdi' será vacía, independientemente de que en las demás dimensiones sí exista solapamiento.

Por otro lado, si en todas las dimensiones los intervalos correspondientes se solapan y ambos multi-intervalos son densos, entonces se garantiza la existencia de al menos una n-upla $(mdi[0], \ldots, mdi[n-1])$ perteneciente tanto a mdi como a mdi'.

Sin embargo, si uno o ambos multi-intervalos no son densos, esta garantía desaparece. Aunque exista solapamiento en todas las dimensiones, podría no haber ninguna n-upla común, ya que en alguna dimensión podría ocurrir que, debido al paso distinto de 1, el intervalo solapamiento puede no tener valores comunes. Por ejemplo, en una dimensión se pueden contar con los intervalos $[5:5:10]$ y $[6:1:9]$, los cuales se solapan, pero cuya intersección es vacía.

4.3.1.2. Criterio de ordenamiento

Uno de los aspectos fundamentales a considerar al implementar la operación de intersección entre conjuntos ordenados, es cómo construir el conjunto resultado de manera que preserve el orden. En este contexto, el desafío consiste en insertar las intersecciones de los multi-intervalos en el conjunto resultado de la forma más eficiente posible, garantizando siempre que se mantenga la invariante del orden.

Criterio de ordenamiento

Supóngase que se realiza la intersección entre dos conjuntos ordenados, A y B, y que se están evaluando las posibles intersecciones del i-ésimo multi-intervalo de A, denotado por A_i, con todos los multi-intervalos de B. Ademas se cuenta con un conjunto resultado C.

Todas las intersecciones no vacías generadas con A_i deben insertarse en C **después** de aquellas intersecciones generadas por los multi-intervalos $A_0, A_1, \ldots, A_{i-1}$, cuyos mínimos sean estrictamente menores al de A_i, bajo el operador $<$ de naturales multi-dimensionales.

Este criterio se fundamenta en la observación de que la intersección entre dos multi-intervalos está contenida en ambos. Esto implica que el elemento mínimo de la intersección debe ser, necesariamente, un valor que pertenezca a ambos operandos, y por lo tanto debe coincidir con el mínimo de alguno de ellos, o bien ser un valor contenido en ambos.

Al fijar el multi-intervalo A_i como decreta el criterio, se observa que cualquier intersección no vacía con un multi-intervalo B_k del conjunto B tendrá su mínimo dentro de A_i, o coincidirá con el mínimo de este. Como resultado, cualquier intersección no vacía generada tendrá un mínimo mayor, bajo el operador $<$ de naturales multi-dimensionales, o igual que el mínimo de A_i.

Esto garantiza que tales intersecciones deben insertarse en el conjunto resultado después de aquellas cuyo mínimo sea estrictamente menor, bajo el operador $<$ de naturales multi-dimensionales, al de A_i.

La Figura 4.7 ilustra gráficamente esta situación. Como se observa, todas las intersecciones producidas a partir de A_i tienen un mínimo mayor o igual que el de A_i, y se insertan a continuación de las intersecciones previamente procesadas cuyo mínimo sea menor al de A_i.

Adicionalmente, no es necesario comenzar a verificar la posición de inserción de las intersecciones en el conjunto resultado desde el principio en cada iteración de los elementos de A. Dado el orden intrínseco de los conjuntos involucrados, se cumple que:

El mínimo de A_i es menor, bajo el operador $<$ de naturales multi-dimensionales, al mínimo de A_{i+1}

Esto implica, en conjunto con lo establecido previamente sobre los mínimos de las intersecciones generadas, que todas las intersecciones obtenidas a partir de A_{i+1} con los elementos de B deben insertarse en el conjunto resultado a partir de una posición **igual o posterior** a aquella en la que comenzaron a insertarse las intersecciones de A_i con B.

Por lo tanto, el criterio de ordenamiento puede refinarse del siguiente modo:

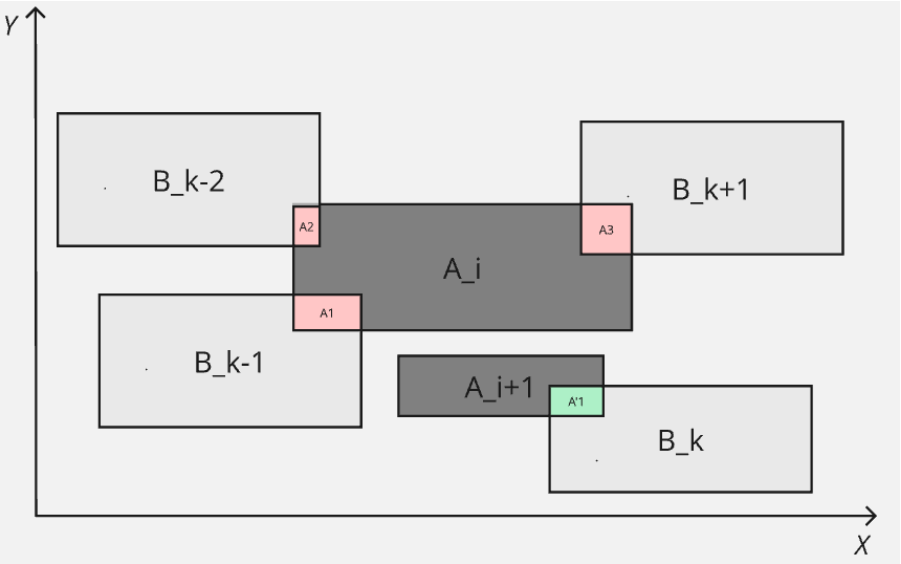


Figura 4.7: Criterio de ordenamiento en la intersección de conjuntos ordenados.

Criterio de ordenamiento

Supóngase que se realiza la intersección entre dos conjuntos ordenados, A y B, y que se están evaluando las posibles intersecciones del i-ésimo multi-intervalo de A, denotado por A_i, con todos los multi-intervalos de B. Ademas se cuenta con un conjunto resultado C.

Todas las intersecciones no vacías generadas con A_i deben insertarse en C **después** de aquellas intersecciones generadas por los multi-intervalos $A_0, A_1, \ldots, A_{i-1}$, cuyos mínimos sean estrictamente menores al de A_i, bajo el operador $<$ de naturales multi-dimensionales.

Adicionalmente **si** la posición a partir de la cual se colocan las intersecciones de A_i en el conjunto resultante es k, con $0 \leq k < \kappa(C)$, **entonces** aquellas generadas por A_{i+1} se insertaran a partir de k' tal que $k \leq k' < \kappa(C)$.

Esta optimización reduce significativamente la cantidad de comparaciones necesarias para ubicar cada intersección, manteniendo la eficiencia del proceso y evitando retrocesos innecesarios dentro de la estructura del conjunto ordenado resultante.

4.3.2. Unión disjunta - `disjointCup`

La operación `disjointCup` se encarga de realizar la unión disjunta entre dos conjuntos de multi-intervalos que, por hipótesis, son disjuntos. En particular, esta operación simplemente coloca los elementos de ambos conjuntos en uno solo, sin aplicar ningún tipo de transformación adicional, ni sobre los argumentos, ni sobre el contenido de sus elementos. Esto se puede ver muy bien particularmente en el pseudocódigo del Algoritmo 4 de esta para conjuntos desordenados.

Ahora bien, al trabajar con conjuntos ordenados, resulta necesario introducir un criterio de ordenamiento que garantice que el conjunto resultante mantenga el orden. Este criterio de ordenamiento se presenta a continuación.

4.3.2.1. Criterio de ordenamiento

Criterio de ordenamiento

Sean $A = \{A_0, A_1, \ldots, A_{n-1}\}$ y $B = \{B_0, B_1, \ldots, B_{m-1}\}$ dos conjuntos no vacíos de multi-intervalos, disjuntos y ordenados. Sea C un conjunto ordenado que contendrá el resultado de la fusión de A y B. Y sean A_i un multi-intervalo del conjunto A y B_k un multi-intervalo del conjunto B, con índices tales que:

$$0 \leq i < \kappa(A), \quad 0 \leq k < \kappa(B).$$

Entonces, al realizar la unión disjunta entre A y B vale que:

- **Si** $A_i < B_k$, entonces A_i debe insertarse en C **antes** que $B_{k'}$ y $A_{i'}$, $\forall i', k' \mid i < i' < \kappa(A) \wedge k \leq k' < \kappa(B)$.
- **Si** $B_k < A_i$, entonces B_k debe insertarse en C **antes** que que $B_{k'}$ y $A_{i'}$, $\forall i', k' \mid i \leq i' < \kappa(A) \wedge k < k' < \kappa(B)$.

Este criterio se fundamente por el orden que disponen los conjuntos ordenados y a su vez por la propiedad transitiva que verifica el operador $<$ de multi-intervalos.

4.3.3. Complemento - `complement / complementAtom`

Esta sub-sección se dividirá en dos partes, ya que la operación de complemento para conjuntos ordenados, al igual que su contraparte de conjuntos desordenados, se compone de dos etapas: el *complemento atómico* (`complementAtom`), que calcula el complemento de un conjunto ordenado representado por un único multi-intervalo; y el *complemento general* (`complement`), la operación principal del complemento, que extiende esta operación tal que sea posible calcular el complemento de conjuntos ordenados compuestos por múltiples multi-intervalos sacando provecho de la operación `intersection`.

4.3.3.1. Complemento atómico(`complementAtom`)

Dentro de la Sub-sección 2.3.2 se presentó el funcionamiento y la lógica de la construcción del conjunto complemento de un conjunto atómico, un conjunto compuesto por un único multi-intervalo, mediante la operación `complementAtom`, en el contexto de los conjuntos desordenados.

Tal como se mencionó, la estructura general de la implementación de las operaciones para conjuntos ordenados se basa en la utilizada para conjuntos desordenados. Por esta razón, correspondería ahora optimizar dicha operación teniendo en cuenta el orden y adaptarla para conjuntos ordenados.

Sin embargo, dado que el conjunto base contiene únicamente un multi-intervalo y solo se cuenta con un único conjunto como argumento de la operación, no hay margen para aplicar ninguna optimización basada en el orden de los argumentos de entrada.

Por lo tanto, la única tarea pendiente es mantener el orden en el resultado, lo cual implica determinar el **criterio de ordenamiento**.

Criterio de ordenamiento

En `complementAtom` se trabaja con dos tipos de multi-intervalos: *all* y *during*. Las transformaciones sucesivas de estos intervalos conforman el complemento del conjunto atómico. El uso de cada uno es el siguiente:

Sea i un número natural que hace referencia una dimensión entre 0 y la cantidad de dimensiones del conjunto atómico procesado.

- ***all***: Construye el multi-intervalo ($all_{i,b}$) que va desde 0 hasta el inicio del multi-intervalo del conjunto en la dimensión i, siempre que exista.

- ***during***: Cuando el multi-intervalo del conjunto no es denso en la dimensión i, es decir, el paso de la i-esima dimensión (paso_i) es distinto de 1, se generan una serie de multi-intervalos intermedios ($during_{i,c}$). Cada uno arranca en

$$\big(\text{begin}_i\big) + c, \quad c = 1, 2, \ldots, (\text{paso}_i - 1),$$

  donde begin_i es el inicio en la dimensión i del multi-intervalo del conjunto.

- ***all***: Construye el multi-intervalo ($all_{i,e}$) que va desde el final del multi-intervalo del conjunto en la dimensión i hasta `Inf`, siempre que sea posible.

Sin embargo, al realizar el procesamiento secuencial de cada dimensión, es necesario tener en cuenta cómo los inicios de los intervalos correspondientes a las dimensiones ya recorridas del multi-intervalo se propagan hacia *all* y *during*.

Una vez completados los cálculos en la dimensión i (creación de $all_{i,b}$, $all_{i,e}$ y todos los $during_{i,c}$), tanto *during* como *all* se modifican en dicha dimensión. A *all* se le remplaza el intervalo en esa dimensión por el del multi-intervalo del conjunto en la misma dimensión pero con paso 1, y a *during* se le hace lo mismo pero con el intervalo original.

Como consecuencia, todos los multi-intervalos que se generen en la siguiente dimensión $i+1$ tienen el mismo valor de inicio en la dimensión i que tiene el multi-intervalo del conjunto. Por lo tanto los inicios de $all_{i,b}$, $all_{i,e}$ y de cada $during_{i,c}$ coinciden, en las dimensiones $0, 1, \ldots, i-1$, con los del multi-intervalo atómico original. Lo que conlleva a que el siguiente conjunto este ordenado:

$$\{all_{i,b},\ during_{i,1},\ during_{i,2},\ \ldots,\ during_{i,k},\ all_{i,e}\}$$

Pero ahora queda ver que interacción hay entre los conjuntos de cada dimensión para ver como ordenar la totalidad de los multi-intervalos de todas las dimensiones en un único conjunto.

Al estar en una dimensión i y al haber terminado de generar los multi-intervalos de esa dimensión a través de *during* y *all*, por como son las modificaciones a estos dos multi-intervalos, resulta que todos los multi-intervalos generados en la dimensión $i+1$ van inmediatamente después de $all_{i,b}$(si existe), ya que en la dimensión i tiene un valor de inicio 0. Ademas, se los debe incluir antes de los $during_{i,c}$, cuyo inicio en la dimensión i es el mismo que el de los multi-intervalos generados en $i+1$, adicionando una constante c. Como resultado tenemos que se da el siguiente orden:

$$\begin{aligned}
&\{all_{x,b},\ all_{y,b},\ all_{z,b},\\
&\ during_{z,1},\ during_{z,2},\ \ldots,\ during_{z,k},\\
&\ all_{z,e},\ during_{y,1},\ during_{y,2},\ \ldots,\ during_{y,j},\\
&\ all_{y,e},\ during_{x,1},\ during_{x,2},\ \ldots,\ during_{x,j},\\
&\ all_{x,e}\}
\end{aligned}$$

trabajando con un conjunto atómico de tres dimensiones.

A partir de todo lo analizado previamente, se arriba a la siguiente formulación del criterio de ordenamiento para `complementAtom`:

Criterio de ordenamiento

El patrón general de inserción en las estructuras auxiliares *all* y *during*, al recorrer las dimensiones en orden creciente, responde a la siguiente disposición:

$$
\begin{aligned}
\{&all_{0,b},\ all_{1,b},\ \ldots,\ all_{k-1,b},\\
&during_{k-1,1},\ \ldots,\ during_{k-1,u_{i-1}},\\
&all_{k-1,e},\ during_{k-2,1},\ \ldots,\ during_{k-2,u_{i-2}},\\
&\quad\vdots\\
&all_{1,e},\ during_{0,1},\ \ldots,\ during_{0,u_0},\\
&all_{0,e}\}
\end{aligned}
$$

donde k representa la aridad del conjunto (es decir, la cantidad de dimensiones), y u_i indica la cantidad de pasos definidos en la dimensión i.

4.3.3.2. Complemento general

La operación `intersection` se utiliza como herramienta para calcular el complemento entre dos conjuntos desordenados en la operación `complement`: uno que representa el complemento de un conjunto de multi-intervalos desordenados, y otro que representa el complemento de un conjunto con un único multi-intervalo.

Teóricamente, podría utilizarse una única implementación general de `intersection`, como se hace en el caso de conjuntos desordenados y no modificar `complement` para el caso de conjuntos ordenados. Sin embargo, al tratarse de conjuntos ordenados, es posible aplicar ciertas optimizaciones adicionales, además de algunas de las ya detalladas en la sub-sección correspondiente a `intersection`, que permiten mejorar el rendimiento de la operación. Es decir, se utilizará una versión adaptada de la intersección de conjuntos ordenados para el complemento.

Intersección complementaria

En esta modificación de la operación de intersección, denominada `intersectionComp`, se preservan los criterios de **solapamiento** y **ordenamiento**. Además de suponer que los conjuntos recibidos están ordenados, esta variante introduce una hipótesis adicional: los argumentos deben respetar un orden específico. El primer conjunto debe representar el complemento de un conjunto de multi-intervalos (posiblemente varios), mientras que el segundo conjunto corresponde al complemento de un único multi-intervalo.

En otras palabras:

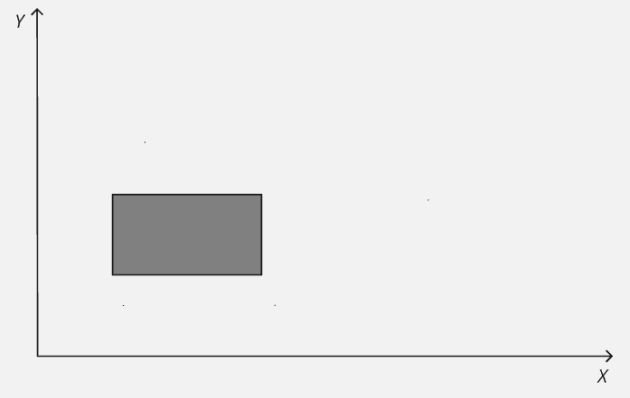
- El primer conjunto, A, actúa como un *complemento acumulado*.
- El segundo conjunto, denotado B, funciona como un *complemento unitario*.

Importante: Para facilitar la siguiente explicación, se asume que los multi-intervalos involucrados son siempre densos.

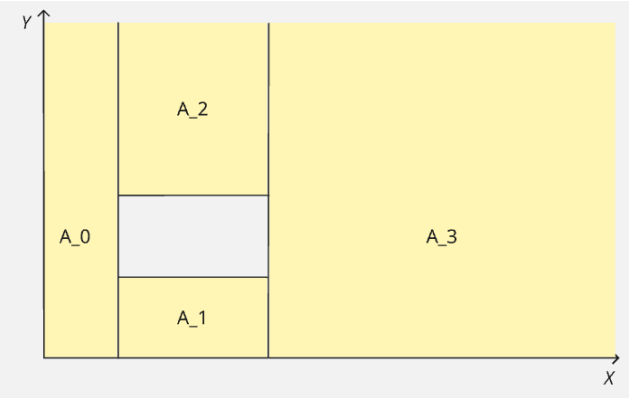
Supongase ahora que el conjunto A representa el complemento de un único multi-intervalo, al igual que B. Esta situación se ilustra en la Figura 4.8.

Si se aplicara directamente la operación `intersection`, utilizando su versión optimizada para conjuntos ordenados, el resultado obtenido sería el ilustrado en la Figura 4.9.

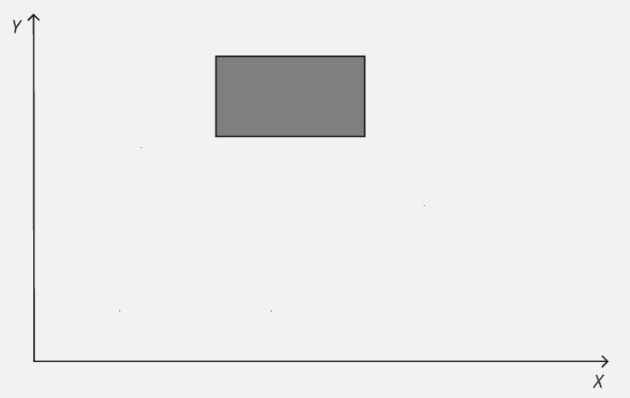
Sin embargo, puede observarse que se produce una partición adicional innecesaria: los multi-intervalos C_1 y C_3 podrían haberse mantenido como un único multi-intervalo. Esta fragmentación/particionamiento es consecuencia del funcionamiento de la operación intersección. No obstante, ambos multi-intervalos no comparten ningún elemento con el multi-intervalo generador de B, por lo que no deberían verse afectados ni sufrir ninguna sustracción de elementos al quitar de A los elementos del multi-intervalo generador de B a través de la intersección de A y B.



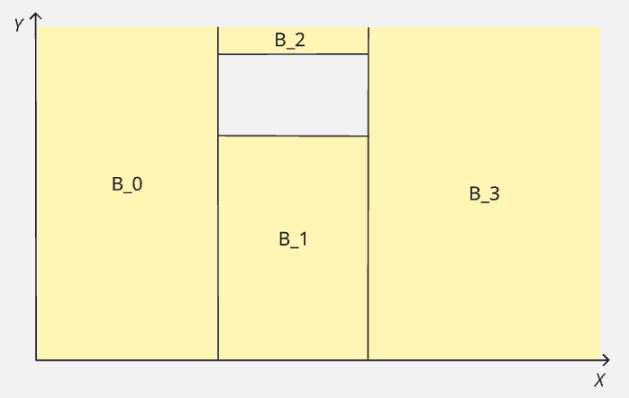
Multi-intervalo generador de A



Conjunto complemento A



Multi-intervalo generador de B



Conjunto complemento B

Figura 4.8: Ejemplo gráfico de los conjuntos complemento de A y B.

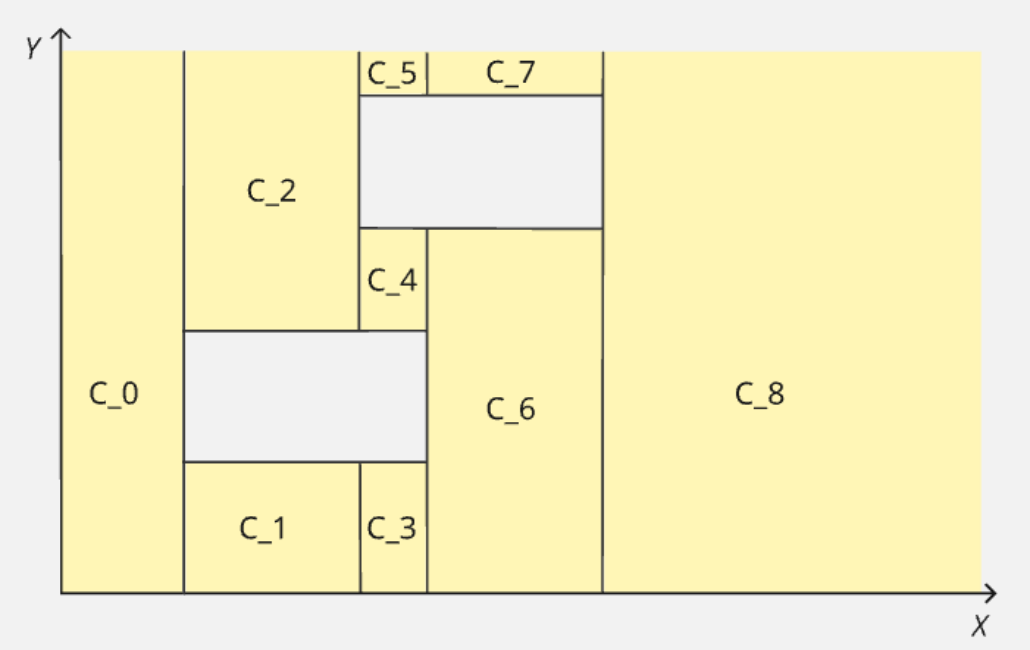


Figura 4.9: Conjunto complemento resultante de la intersección entre A y B, sin aplicar optimización.

Este tipo de fragmentación innecesaria puede evitarse mediante el criterio de optimización denominado **criterio de anti-particionamiento**, que establece lo siguiente:

Criterio de anti-particionamiento

Sea A un conjunto complemento y B el complemento de un conjunto compuesto por un único multi-intervalo mdi. Al realizar la intersección entre A y B durante el cálculo del complemento de un conjunto, se cumple que:

Todo multi-intervalo de A que no se solape con el multi-intervalo mdi generado por B no requiere ser intersectado con los elementos de B, y debe copiarse directamente al conjunto resultado.

Aplicando este criterio, el conjunto complemento resultante se muestra en la Figura 4.10, donde se observa una representación más compacta: se evita una partición innecesaria y se obtiene un conjunto más chico.

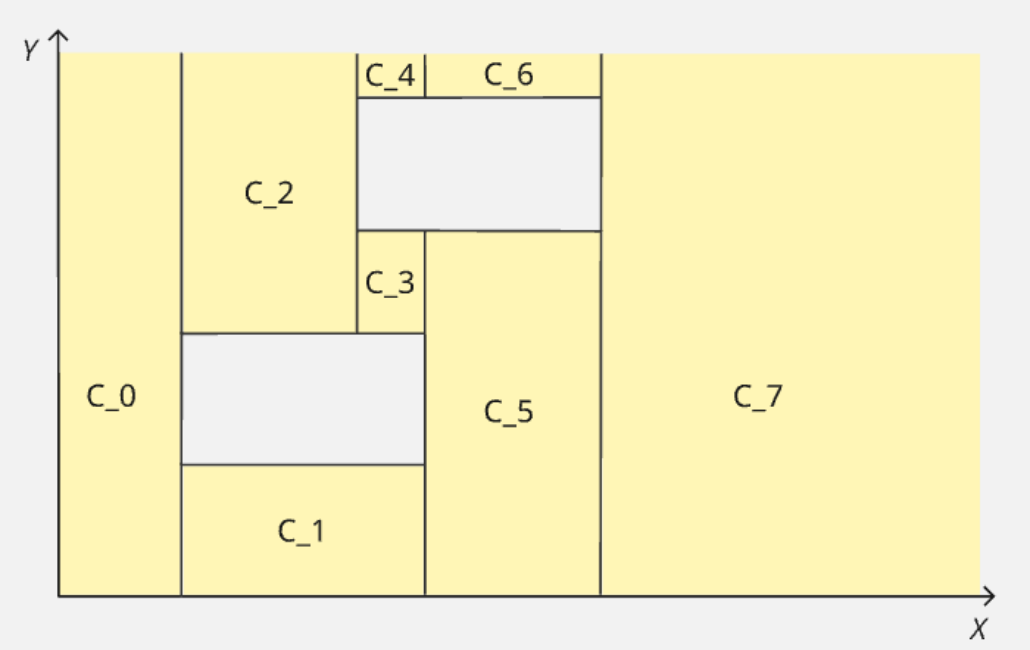


Figura 4.10: Conjunto complemento resultante tras aplicar el criterio de anti-particionamiento.

Cabe destacar que este criterio también es válido cuando se trabaja con multi-intervalos no densos, y resulta especialmente útil en esos casos, ya que evita múltiples operaciones de posicionamiento y comparación innecesarias.

Si bien la reducción en el número de multi-intervalos resultantes puede parecer ínfima, es importante tener en cuenta que en la operación `complement` se aplica de forma acumulativa esta reducción. Por lo tanto, reducir la cantidad de particiones en cada iteración tiene un efecto significativo en el rendimiento total, ya que disminuye la cantidad de elementos a recorrer y comparar en las iteraciones posteriores.

Existe además una mejora adicional que puede aplicarse a la operación de intersección en el contexto del complemento. Esta surge por el modo en que se construye el complemento ordenado de un conjunto atómico, tal como se explicó en la subsección correspondiente del complemento atómico.

En dicha construcción, el primer multi-intervalo del complemento ordenado es $all_{x,b}$. Este multi-intervalo es denso y se extiende hasta el infinito en todas sus dimensiones, con excepción de la primera, donde finaliza una posición antes del comienzo del multi-intervalo original que dio lugar al complemento.

Adicionalmente se cumple que la intersección entre cualquier multi-intervalo y uno denso, que lo contiene por completo, es simplemente el multi-intervalo contenido. Es decir, si un multi-intervalo mdi está completamente contenido en otro multi-intervalo denso mdi', entonces la intersección entre mdi y mdi' es igual a mdi.

A partir de las observaciones anteriores, puede establecerse el criterio de optimización denominado **criterio de obviedad**, el cual permite evitar cálculos innecesarios durante la intersección del complemento, y estipula lo siguiente:

Criterio de obviedad

Sea A un conjunto complemento y B el complemento de un conjunto compuesto por un único multi-intervalo mdi. Al realizar la intersección entre A y B durante el cálculo del complemento de un conjunto, se cumple que:

Todo multi-intervalo del conjunto A cuyo máximo sea estrictamente menor en la primera dimensión que el mínimo multi-intervalo generador de B, mdi, puede incorporarse directamente al conjunto resultado de la intersección, sin necesidad de ser intersectado con los elementos de B.

Tal como fue planteado, el **criterio de obviedad** permite evitar las comparaciones e intersecciones entre determinados multi-intervalos del conjunto complemento A y los elementos del conjunto complemento B, pero su alcance inicial es limitado: únicamente evita el procesamiento de los multi-intervalos de A que sean estrictamente menores (en la primera dimensión) que el multi-intervalo generador de B en una única invocación de la operación `intersectionComplement`.

Sin embargo, este criterio puede extenderse de forma significativa durante el proceso iterativo de la operación `complement`. Esto es posible gracias a la siguiente observación: los multi-intervalos que generan cada conjunto B a lo largo de las distintas iteraciones provienen de un conjunto ordenado, por lo tanto, sus valores iniciales en la primera dimensión son no decrecientes. Dado que estos multi-intervalos se recorren en orden, cada nuevo multi-intervalo generador B tendrá en la primera dimensión un valor de inicio igual o mayor que el anterior. Y, por ende cada nuevo $all_{x,b}$ del conjunto B generado por un multi-intervalo contendrá a todos los anteriores $all_{x,b}$.

Como consecuencia, una vez que un multi-intervalo de A ha sido descartado por el criterio de obviedad en una iteración, puede descartarse definitivamente para todas las iteraciones subsiguientes, ya que siempre será obviado por el criterio obviedad.

Esto permite trasladar dichos multi-intervalos directamente al resultado final de la operación `complement`, en lugar de seguirlos incorporando los en el cálculo de las posteriores iteraciones.

Se procede entonces a declarar el **criterio de obviedad extendido**:

Criterio de obviedad extendido

Sea A un conjunto ordenado de multi-intervalos que representa el complemento acumulado en una iteración de la operación `complement` y B el complemento de un conjunto compuesto por un único multi-intervalo de dicha iteración también.

Entonces, A puede *descomponerse* en dos subconjuntos disjuntos:

- A_o: el subconjunto de multi-intervalos que son descartados por el criterio de obviedad en base a B en dicha iteración, y que, por lo tanto, pueden ser incorporados directamente al resultado final sin necesidad de ser procesados en iteraciones posteriores;
- A_r: el subconjunto restante, compuesto por los multi-intervalos que aún pueden ser particionados en iteraciones futuras, es decir, aquellos que no son descartados por el criterio de obviedad en base a B. Luego, la intersección se dará entre este conjunto y B, y su resultado será el nuevo A.

Este criterio permite ahorrar el recorrer los mismos elementos múltiples veces durante el calculo de la intersección e el complemento. Esto permite aumentar en gran medida la eficiencia de la operación.

4.4. Implementaciones

Para llevar a cabo la implementación de los conjuntos ordenados, fue necesario desarrollar una serie de funciones adicionales. A continuación, se describen dichas implementaciones complementarias junto con su

correspondiente pseudocódigo.

4.4.1. Calculo de solapamiento - `doInt`

La operación `doInt` se utiliza en el contexto de conjuntos ordenados, aunque no forma parte de su interfaz de operaciones. Esta tiene el objetivo de calcular si existe solapamiento entre dos multi-intervalos. Esta operación resulta fundamental tanto para la aplicación del **criterio de solapamiento** como para el **criterio de anti-particionamiento**.

El pseudocódigo correspondiente a esta operación se presenta a continuación, y resulta relativamente breve:

Algorithm 25 Cálculo de solapamiento para conjuntos ordenados

Require: Mdi_1 y Mdi_2 son dos multi-intervalos
Ensure: Devuelve `true` si estos se solapan, `false` en caso contrario.

```
1:  function doInt(Mdi_1, Mdi_2)
2:      max_1 := maxElem(Mdi_1)
3:      min_1 := minElem(Mdi_1)
4:      max_2 := maxElem(Mdi_2)
5:      min_2 := minElem(Mdi_2)
6:      a := arity(Mdi_1)
7:      for i = 0; i < a − 1 ; i++ do
8:          if max_1[i] < min_2[i] or max_2[i] < min_1[i] then
9:              return false
10:         end if
11:     end for
12:     return true
13: end function
```

4.4.2. Inserción con pista - `emplaceHint`

La función `emplaceHint` se propone también como una herramienta auxiliar externa a la estructura principal de conjuntos ordenados. Su tarea consiste en insertar un multi-intervalo en un conjunto ordenado, siguiendo el mismo criterio de ordenamiento utilizado por los mismos. Para hacerlo, utiliza una posición pista (*hint*), es un número natural y una posición sugerida y que sirve como punto de inicio para encontrar eficientemente la ubicación correcta del multi-intervalo.

El pseudocódigo de esta función es corto, ya que su implementación es simple, y se encuentra en el Algoritmo 26.

4.4.3. Avance con pista - `advanceHint`

La función `advanceHint` se propone también como una herramienta auxiliar externa a la estructura principal de conjuntos ordenados. El objetivo de esta función es avanzar la posición de la pista (hint) que viene como argumento en función de los elementos presentes en un conjunto ordenado, utilizando un multi-intervalo como criterio de parada.

El pseudocódigo de esta función auxiliar se presenta a continuación.

4.5. Implementaciones

Llegado este punto, es momento de abordar las implementaciones concretas de las distintas operaciones sobre conjuntos ordenados.

Algorithm 26 Inserción con pista para conjuntos ordenados

Require: A es una conjunto ordenado, Mdi en un multi-intervalo y $Hint$ es un numero natural y una posición sugerida.

Ensure: Inserta el Mdi dentro del conjunto ordenado, manteniendo el orden y empezando a buscar la posición de inserción a partir de $Hint$.

```
1: function EMPLACEHINT(A, Mdi, Hint)
2:     n := SIZE(A)
3:     it := Hint
4:     while it < n do
5:         if A_it < mdi then
6:             it++
7:         else
8:             break
9:         end if
10:    end while
11:    A := A_{0:it-1} ⌢ {mdi} ⌢ A_{it:SIZE(A)-1}
12: end function
```

Algorithm 27 Avance con pista para conjuntos ordenados

Require: A es un conjunto ordenado, Mdi en un multi-intervalo y $Hint$ es un número natural.

Ensure: Avanza la pista $Hint$ en base a los elementos del conjunto y el Mdi.

```
1: function ADVANCEHINT(A, Mdi, Hint)
2:     n := SIZE(A)
3:     while Hint < n do
4:         if A_Hint < Mdi then
5:             Hint++
6:         else
7:             break
8:         end if
9:     end while
10: end function
```

Esta seccion se organiza en tres sub-secciones principales: una dedicada a aquellas operaciones cuya implementación no requirió modificaciones, una sub-sección que aborda las operaciones adaptadas para poder funcionar correctamente en el contexto de conjuntos ordenados, y finalmente otra correspondiente a las operaciones que fueron optimizadas.

4.5.1. Operaciones sin cambios

Al implementar los conjuntos ordenados utilizando como base la versión desordenada, se observó que ciertas operaciones no podían beneficiarse del orden para su optimización o no se podían optimizar en si, pero tampoco alteraban dicho orden. Esto se debe, principalmente, a dos razones: o bien son operaciones que no devuelven un conjunto ordenado, o bien el conjunto resultante ya se encuentra ordenado.

Dentro del primer conjunto de operaciones se incluyen, por ejemplo: `cardinal`, que devuelve el número de elementos que contiene el conjunto; `arity`, que informa la aridad del mismo; entre otras.

Dado que estas operaciones son numerosas y su comportamiento no resulta central para los objetivos de esta tesina, además de que sus implementaciones se mantuvieron sin modificaciones, no se detallarán aquí. Todas ellas pueden consultarse en el archivo *set.cpp*, disponible en el repositorio dentro de la carpeta *sbg*.

En el segundo conjunto se incluyen operaciones como `compact`, cuyo algoritmo no elimina el orden del conjunto resultante, o `cup`, cuya lógica interna se basa exclusivamente en operaciones que preservan dicho orden.

Al igual que en el caso anterior, se omite un desarrollo mas amplio de estas operaciones. Si se desea consultarlas en profundidad puede dirigirse al archivo *set.cpp*, disponible en el repositorio dentro de la carpeta *sbg*.

4.5.2. Operaciones adaptadas al orden

Este grupo de operaciones fueron adaptadas para funcionar en el contexto de conjuntos ordenados, pero no pudieron ser optimizadas. Dentro de estas operaciones se encuentran:

- `emplace`: Que se encarga de ubicar un multi-intervalo dentro de un conjunto.
- `emplaceBack`: Que se encarga de ubicar un multi-intervalo al final de un conjunto, como su último elemento, aquel con mayor mínimo.
- `minElem`: Busca el mínimo mínimo de todos los multi-intervalos de un conjunto

Si se desea ver cuales fueron sus modificaciones con respecto a sus versiones de conjuntos desordenados, puede verifica el archivo *set.cpp* dentro de la carpeta *sbg*, disponible en el repositorio.

4.5.3. Operaciones optimizadas

En esta sub-sección se presentan aquellas operaciones que pudieron ser optimizadas aprovechando el orden de los conjuntos. Cada una de ellas será descripta minuciosamente y explicada en detalle, destacando los criterios de optimización y ordenamiento en cada caso.

4.5.3.1. Intersección - `intersection`

Como se puede observar a continuación, la operación `intersection` resultó ser considerablemente más extensa que su homóloga para conjuntos desordenados.

No obstante, teniendo en cuenta lo expuesto en la Sub-sección 4.3.1, se tiene lo siguiente:

- Para implementar el **criterio de selección**, se realiza la verificación correspondiente según lo establecido por dicho criterio: se analiza la cantidad de multi-intervalos presentes en los conjuntos C y D. A partir de

esta comparación, se determina cuál es el conjunto largo, A, y cuál el corto, B, tal como se muestra en el Algoritmo 28, líneas 12 a 17.

- En relación con el **criterio de eliminación**, dado que su cumplimiento implica la remoción de un elemento de uno de los conjuntos, en este caso del conjunto B, y dado que no es posible eliminar elementos directamente del mismo, se utiliza una lista enlazada que contiene los índices que referencian a los multi-intervalos de B. Esto permite llevar a cabo la eliminación de los índices en vez de los multi-intervalos. La preparación de dicha lista se observa en el Algoritmo 28, líneas 18 a 21, y la implementación del criterio se detalla en el Algoritmo 29, líneas 7 a 10.
- En cuanto el **criterio de parada** este se implementa de la linea 11 a 13 en el Algoritmo 29, donde lo único que se hace es forzar la salida del segundo bucle a través de un `break`.
- Para la aplicación del **criterio de solapamiento**, se emplea la función auxiliar `doInt`, encargada de verificar si dos multi-intervalos se solapan. Esto se encuentra en el Algoritmo 29, línea 14.
- Finalmente, el **criterio de ordenamiento** se implementa mediante las funciones auxiliares `advanceHint` y `emplaceHint`, permitiendo que al aplicar el criterio, el conjunto resultado se mantenga ordenado a lo largo de la ejecución. Esto puede observarse en el Algoritmo 29, líneas 3 y 17.

Algorithm 28 Intersección de conjuntos ordenados — Parte 1: Preparación

Require: C y D son conjuntos ordenados
Ensure: R es la intersección ordenada de C y D

```
 1: function INTERSECTION(C, D)
 2:     R := {}
 3:     if ISEMPTY(C) or ISEMPTY(D) then
 4:         return R
 5:     end if
 6:     if C == D then
 7:         return C
 8:     end if
 9:     if MAXELEM(C) < MINELEM(D) or MAXELEM(C) < MINELEM(D) then
10:         return R
11:     end if
12:     B := C
13:     A := D
14:     if SIZE(D) < SIZE(C) then
15:         B := D
16:         A := C
17:     end if
18:     indices := []                          ▷ Es una lista simplemente enlazada
19:     for i = 0; i < SIZE(B); i++ do
20:         indices := indices ++ [i]          ▷ Se concatenan las listas enlazadas
21:     end for
22:     r_pos := 0
23: end function
```

Algorithm 29 Intersección de conjuntos ordenados — Parte 2: Construcción del resultado

```
1: function INTERSECTION
2:     for all a ∈ A do
3:         ADVANCEHINT(R, a, r_pos)
4:         i := 0
5:         while i ≠ LENGTH(indices) do
6:             b := B_indices[i]
7:             if MAXELEM(a)[0] < MINELEM(b)[0] then
8:                 indices := indices ◁ i
9:                 continue
10:            end if
11:            if MAXELEM(a)[0] < MINELEM(b)[0] then
12:                break
13:            end if
14:            if DOINT(b,a) then
15:                inter := INTERSECTION(a, b)
16:                if ¬ISEMPTY(inter) then
17:                    EMPLACEHINT(R, inter, r_pos)
18:                end if
19:            end if
20:            i ++
21:        end while
22:        if indices == [] then
23:            break
24:        end if
25:    end for
26:    return R
27: end function
```

4.5.3.2. Unión disjunta - *disjointCup*

A continuación se presenta el pseudocódigo correspondiente a la implementación de la operación `disjointCup`, encargada de llevar a cabo la unión disjunta de conjuntos ordenados.

El algoritmo de fusión que implementa la unión disjunta debe recorre entonces ambos conjuntos en paralelo: tomando el indice i_a para A y i_b para B, comenzando ambos en 0. En cada iteración de la operación se aplica el **criterio de ordenamiento**. El procedimiento distingue los siguientes casos:

- **Si** $A_{i\_a} < B_{i\_b}$: se agrega A_{i_a} al final de C y se incrementa el índice i_a (i.e., i_a++).
- **En caso contrario** (es decir, $A_i \geq B_{i\_b}$): se agrega B_{i_b} al final de C y se incrementa el índice i_b (i.e., i_b++).

Este proceso se repite hasta que al menos uno de los dos conjuntos (A o B) haya sido recorrido completamente. Una vez alcanzado ese punto, se agregan al final de C todos los elementos restantes del conjunto que aún no se haya agotado, preservando su orden original.

La Figura 4.11 ejemplifica el proceso de ejecución de la función `disjointCup`. En ella, los multi-intervalos de los conjuntos A y B están representados respectivamente en rojo y verde cuando son sometidos al criterio de ordenamiento. Estos elementos pasan a través del criterio de ordenamiento definido, determinando su incorporación al conjunto resultante C, cuyos elementos se muestran en amarillo a medida que se insertan.

(a) (e)

(b) (f)

(c) (g)

(d) (h)

Figura 4.11: Ilustración del criterio de ordenamiento aplicado en la fusión `disjointCup` dado un conjunto A y B ordenados y disjuntos.

Algorithm 30 Unión de conjuntos ordenados disjuntos

Require: A y B son conjuntos ordenados
Ensure: C es la unión disjunta ordenada de A y B

```
1: function DISJOINTCUP(A, B)
2:     if ISEMPTY(A) then
3:         return B
4:     end if
5:     if ISEMPTY(B) then
6:         return A
7:     end if
8:     C := {}
9:     i_a := 0
10:    i_b := 0
11:    end_a := SIZE(A)
12:    end_b := SIZE(B)
13:    if MINELEM(A_{end_a−1}) < MINELEM(B_0) then
14:        return A ⌢ B
15:    end if
16:    if MINELEM(B_{end_b−1}) < MINELEM(A_0) then
17:        return B ⌢ A
18:    end if
19:    for ;i_a ≠ end_a and i_b ≠ end_b; do
20:        if A_{i_a} < B_{i_b} then
21:            EMPLACEBACK(C, A_{i_a})
22:            i_a ++
23:        else
24:            EMPLACEBACK(C, B_{i_b})
25:            i_b ++
26:        end if
27:    end for
28:    C := C ⌢ A_{i_a:end_a−1}
29:    C := C ⌢ B_{i_b:end_b−1}
30:    return C
31: end function
```

4.5.3.3. Complemento - `complement`

A continuación se presenta el pseudocódigo correspondiente a la implementación de la operación `complement` en el Algoritmo 31, la cual es similar a su homóloga para conjuntos desordenados.

La única diferencia significativa radica en la incorporación parcial del **criterio de obviedad extendido**, generando un conjunto ordenado de multi-intervalos obviados R, que se pasa como argumento a la operación `intersectionComp`. En esta es donde se realizará la división de A en los dos subconjuntos disjuntos que se mencionan en el criterio.

Este proceso puede observarse en las líneas 6 y 16, respectivamente.

Algorithm 31 Complemento de conjuntos ordenados

Require: A es un conjunto ordenado
Ensure: El resultado es el complemento ordenado de A
1: **function** COMPLEMENT(C)
2: **if** ISEMPTY(C) **then**
3: **return** $\{\}$
4: **end if**
5: $A := \{\}$
6: $R := \{\}$
7: $first\_mdi := A_0$
8: $A :=$ COMPLEMENTATOM($\{first\_mdi\}$)
9: **for** $i = 1$; $i \neq$ SIZE(C) ; i ++ **do**
10: $mdi := C_i$
11: $atomic\_set :=$ COMPLEMENTATOM($\{mdi\}$)
12: $A :=$ INTERSECTIONCOMP($A, atomic\_set, R, mdi$)
13: **end for**
14: **return** DISJOINTCUP(R, A)
15: **end function**

4.5.3.4. Complemento atómico - `complementAtom`

En lo que respecta a la operación `complementAtom`, esta debía mantenerse equivalente a su contraparte para conjuntos desordenados, pero incorporando el **criterio de ordenamiento** definido específicamente para esta operación.

Como se puede observar en el pseudocódigo a continuación, dicho criterio se implementa mediante el uso de las variables numéricas *pos_global* y *pos_local* (línea 15 en el Algoritmo 32 y línea 3 en el Algoritmo 33), así como a través de la colocación ordenada de los elementos dentro del conjunto de salida C, en las líneas 10, 24 y 35 en el Algoritmo 33.

Algorithm 32 Complemento atómico para conjuntos ordenados — Parte 1: Preparación

Require: A es un conjunto ordenado atómico
Ensure: D es el complemento ordenado de A
1: **function** COMPLEMENTATOM(A)
2: $C := \{\}$
3: $mdi := A_0$
4: $dense\_mdi := []$
5: **for all** $interval \in mdi$ **do**
6: $b :=$ BEGIN($interval$)
7: $e :=$ END($interval$)
8: EMPLACEBACK($dense\_mdi, [b : 1 : e]$)
9: **end for**
10: $during\_mdi := dense\_mdi$
11: $univ := [0 : 1 : \texttt{Inf}]$
12: $a :=$ ARITY(A)
13: $all := univ^a$ ▷ Universo de dimensión a
14: $dim := 0$
15: $pos\_global := 0$
16: **end function**

Algorithm 33 Complemento atómico para conjuntos ordenados— Parte 2: Construcción y retorno

```
1: function COMPLEMENTATOM
2:     for all i ∈ mdi do
3:         pos_local := pos_global
4:         b := BEGIN(i)
5:         if BEGIN(i) ≠ 0 then
6:             i_res := [0 : 1 : BEGIN(i)−1]
7:             if ¬ISEMPTY(i_res) then
8:                 all[dim] := i_res
9:                 n := SIZE(C)
10:                C := C_{0:pos_local−1} ⌢ {all} ⌢ C_{pos_local:n−1}
11:                pos_local ++,  pos_global ++
12:                all[dim] := univ
13:            end if
14:        end if
15:        if BEGIN(i) < Inf then
16:            if STEP(i) > 1 then
17:                for j = 0; j < size(A); j++ do
18:                    h := [BEGIN(i)+j + 1 :STEP(i) : END(i)]
19:                    if ¬ISEMPTY(h) then
20:                        during_mdi[dim] := h
21:                        n := SIZE(C)
22:                        C := C_{0:pos_local−1} ⌢ {during_mdi} ⌢ C_{pos_local:n−1}
23:                        pos_local ++
24:                    end if
25:                end for
26:            end if
27:        end if
28:        if END(i) < Inf then
29:            i_res := [END(i)+1 : 1 : inf]
30:            if ¬ISEMPTY(i_res) then
31:                all[dim] := i_res
32:                n := SIZE(C)
33:                C := C_{0:pos_local−1} ⌢ {all} ⌢ C_{pos_local:n−1}
34:                pos_local++
35:                all_dim := univ
36:            end if
37:        end if
38:        all[dim] := dense_mdi[dim]
39:        during_mdi[dim] := i
40:        dim ++
41:    end for
42:    return C
43: end function
```

4.5.3.5. Intersección complementaria - `intersectionComp`

Por último, se presenta la operación adicional `intersectionComp`, la cual permite calcular una intersección optimizada cuando se ejecuta la operación `complement`.

Cabe recordar que esta operación es una variante de la operación `intersection` presentada anteriormente. Sin embargo, a diferencia de aquella, `intersectionComp` solo implementa el **criterio de solapamiento** y el **criterio de ordenamiento**, y, ademas, incorpora los tres criterios de optimización adicionales vistos con anterioridad:

- Se implementa el **criterio de obviedad** y la parte faltante del **criterio de obviedad extendido**, ambos utilizando el multi-intervalo Mdi . Esto puede observarse en el Algoritmo 34, líneas 10 a 13 del pseudocódigo, donde se subdivide a A en $Remnant$ y $A_{pos:\text{SIZE}(A)}$, que tienen los elementos que ya forman parte del resultado final del complemento y los elementos restantes a interseccionar, respectivamente.
- En cuanto al **criterio de anti-particionamiento** en el Algoritmo 35, se lleva a cabo mediante el uso de la función auxiliar `doInt`, que permite realizar un chequeo de solapamiento entre los multi-intervalos de A, y los de B. En función de este resultado, se decide si realizar el recorrido por los elementos de B, o bien guardar directamente los multi-intervalos de A en el conjunto resultado.

Algorithm 34 Intersección complementaria para conjuntos ordenados — Parte 1: Preparación

Require: A y B son conjuntos ordenados, $Remnant$ un conjunto ordenado, y Mdi un multi-intervalo
Ensure: C es la intersección ordenada de los elementos necesarios A y B

```
1:  function INTERSECTIONCOMP(A, B, Remnant, Mdi)
2:      C := {}
3:      if ISEMPTY(A) or ISEMPTY(B) then
4:          return C
5:      end if
6:      if A == B then
7:          return A
8:      end if

9:      pos := 0
10:     while pos < SIZE(A) and MAXELEM(A_pos)[0] < MINELEM(Mdi)[0] do
11:         EMPLACEBACK(Remnant, A_pos)
12:         pos ++
13:     end while

14:     c_pos := 0
15: end function
```

Algorithm 35 Intersección complementaria para conjuntos ordenados — Parte 2: Construcción del resultado

```
1:  function intersectionComp
2:      endA := size(A)
3:      for all a ∈ A_{pos:endA−1} do
4:          do_int := doInt(a, mdi)
5:          advanceHint(C, a, c_pos)

6:          if do_int then
7:              for all b ∈ B do
8:                  if doInt(b,a) then
9:                      inter := intersection(a, b)
10:                     if ¬ isEmpty(inter) then
11:                         emplaceHint(C,inter,c_pos)
12:                     end if
13:                 end if
14:             end for
15:         else
16:             emplaceHint(C, a, c_pos)
17:         end if

18:     end for

19:     return C
20: end function
```

Capítulo 5

Piecewise maps ordenados

De manera análoga a lo realizado con los **conjuntos ordenados**, se introducen ahora los ***Piecewise maps* ordenados**. El objetivo de este capítulo es explicar cómo estos almacenan los mapas, presentar las optimizaciones obtenidas, *criterios de optimización* y criterios de ordenamiento, y finalmente, detallar la forma en que dichas optimizaciones fueron incorporadas en las operaciones de los *Piecewise maps* ordenados.

5.1. Estructura y orden

Ahora es el turno de los ***piecewise maps* ordenados**. A diferencia de los conjuntos ordenados, no se disponía de una implementación previa que ofreciera una estructura ordenada de ninguna índole, capaz de brindar pistas de cómo desarrollar el criterio de orden. En consecuencia, fue necesario desarrollar dicho criterio desde cero.

Inicialmente, los piecewise maps ordenados deben contar con una variable miembro que contenga los mapas que se desean almacenar. Esta variable se denominará `pieces_`, aunque por el momento no se le asignará un tipo concreto, ya que primero es necesario definir el criterio de orden.

El objetivo es construir un criterio de orden que permita reutilizar y aprovechar los conceptos desarrollados para optimizar los conjuntos ordenados, aun cuando ahora se trabaja con mapas. Recordemos que un `Map` está definido por las siguientes dos componentes:

- Un conjunto dominio;
- Una colección de expresiones lineales.

Se decidió tomar como base del criterio de orden a los conjuntos dominio, ya que estos están definidos en base a los multi-intervalos, y pueden ayudar a cumplir el objetivo planteado anteriormente. En cambio, las expresiones lineales no aportan ninguna ventaja aparente, por lo cual son descartadas para gestionar el orden dentro de los *piecewise maps* ordenados.

Definido el elemento sobre el cual se va a basar el orden, corresponde establecer cuándo un mapa se considera estrictamente menor que otro:

Sean $m_1 = dom_1 \longmapsto exps_1$ y $m_2 = dom_2 \longmapsto exps_2$ dos mapas arbitrarios, es decir, objetos de tipo `Map`.
Entonces, se define que $m_1 < m_2$ si y solo si $minPer_1 < minPer_2$,
donde la comparación se realiza con el operador $<$ de los naturales multi-dimensionales o `MD_NAT`, y $minPer_1$ y $minPer_2$ son los mínimos perimetrales de dom_1 y dom_2, respectivamente.

Ahora bien, el mínimo perimetral se define de la siguiente manera:

Sea $x = (x_0, x_1, \ldots, x_{n-1})$ un natural multi-dimensional o `MD_NAT`, y sea C un conjunto.

Entonces, m es el **mínimo perimetral** de C si para todo $j \in \{0, \ldots, k-1\}$ se cumple que x_j es el mínimo valor en la j-ésima dimensión entre todos los elementos de C.

Por ejemplo: Considerando el conjunto $C = \{mdi_1, mdi_2, mdi_3\}$, siendo $mdi_1 = [2:1:2] \times [3:2:9]$, $mdi_2 = [1:1:2] \times [5:2:9]$ y $mdi_3 = [2:1:2] \times [1:2:9]$, el mínimo perimetral seria $(1, 1)$.

Teniendo todo esto presente, y dado que los conjuntos no implementan ningún método que permita calcular el mínimo perimetral, la idea será generar una operación adicional para calcular dicho mínimo, realizar esa operación una única vez (ya que implica recorrer todos los mínimos de un conjunto), y almacenarlo junto con su mapa correspondiente.

En consecuencia, la variable `pieces_` tendrá la siguiente forma:

$$\texttt{pieces\_} = \ll (m_0,\ minPer_0),\ (m_1,\ minPer_1),\ \ldots,\ (m_{k-1},\ minPer_{k-1}) \gg$$

donde para todo $i, j \in \{0, \ldots, k-1\}$ con $i < j$, se cumple que $m_i < m_j$, es decir, $minPer_i < minPer_j$.

Con esto, el orden dentro de un *piecewise map* ordenado queda definido. Sin embargo, esto no es suficiente para aprovechar por completo los conceptos utilizados para optimizar los conjuntos ordenados. Para ello, es necesario definir también el concepto de **máximo perimetral**:

Sea $x = (x_0, x_1, \ldots, x_{n-1})$ un natural multi-dimensional o `MD_NAT`, y sea C un conjunto.
Entonces, m es el **máximo perimetral** de C si para todo $j \in \{0, \ldots, k-1\}$ se cumple que x_j es el máximo valor en la j-ésima dimensión entre todos los elementos de C.

Por ejemplo: Considerando el conjunto $C = \{mdi_1, mdi_2, mdi_3\}$, siendo mdi_1, mdi_2 y mdi_3 los dados en el ejemplo anterior, el máximo perimetral seria $(2, 9)$.

Esta información también debe ser almacenada, tal como se hizo con el mínimo perimetral. Por lo tanto, la variable `pieces_` tendrá ahora la siguiente estructura extendida:

$$\begin{aligned}\texttt{pieces} = \ll\ & (m_0,\ (\texttt{minPer}_0,\ \texttt{maxPer}_0)),\\ & (m_1,\ (\texttt{minPer}_1,\ \texttt{maxPer}_1)),\\ & \ldots,\\ & (m_{k-1},\ (\texttt{minPer}_{k-1},\ \texttt{maxPer}_{k-1}))\ \gg\end{aligned}$$

Este concepto funciona como contraparte del mínimo perimetral y permite interpretar el conjunto dominio de un mapa como si fuera un único multi-intervalo definido por su mínimo y su máximo perimetral. En la Figura 5.1 se muestra un ejemplo de esta interpretación de manera gráfica, donde dicho multi-intervalo está marcado en línea punteada.

Con toda esta información ya se puede declarar formalmente cómo está constituida la variable `pieces_` para los *piecewise maps* ordenados, denominados como `OrdPWMap`:

- `pieces_` (de tipo **OrdMapCollection**): representa la colección ordenada de mapas junto con la información perimetral de sus dominios. En este contexto:
  - `OrdMapCollection` es sinónimo de `vector<MapEntry>`.
  - `MapEntry`, cuyas instancias denominaremos en adelante *entradas de mapa*, es sinónimo de `pair<Map, SetPerimeter>`.
  - `SetPerimeter`, cuyas instancias denominaremos en adelante *perímetros*, es sinónimo de `pair<MD_NAT, MD_NAT>`.

Y por último se presenta el siguiente ejemplo. Sean se los siguientes conjuntos dominio de los mapas m_1, m_2, y m_3 respectivamente:

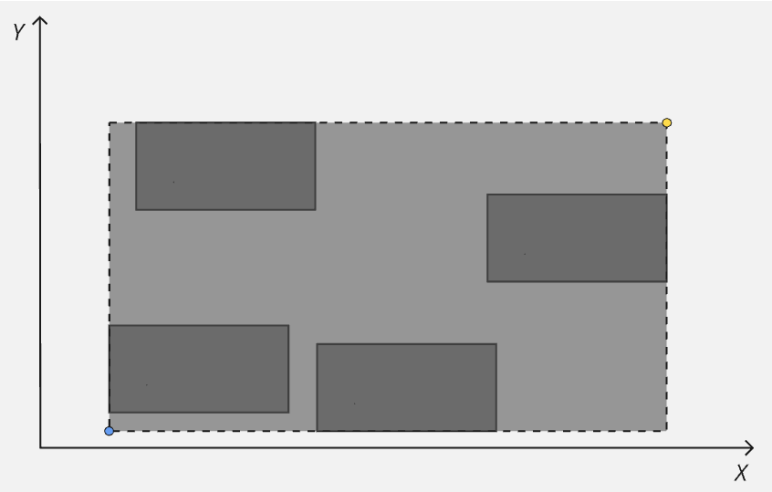


Figura 5.1: Representación gráfica del conjunto dominio de un mapa con 4 multi-intervalos, más el marcado del mínimo y máximo perimetral en azul y amarillo respectivamente.

- $c_1 = \{[10:1:15] \times [10:1:11], [16:1:20] \times [15:1:20]\}$, con mínimo perimetral $(10, 10)$
- $c_2 = \{[15:1:20] \times [15:1:25], [21:1:25] \times [15:1:20]\}$, con mínimo perimetral $(15, 15)$
- $c_3 = \{[30:1:35] \times [15:1:20], [36:1:40] \times [10:1:11]\}$, con mínimo perimetral $(30, 10)$

Aplicando el criterio de orden basado en los mínimos perimetrales, junto con el operador $<$ definido para la naturales multi-dimensionales, se obtiene el siguiente ordenamiento:

$$c_1 < c_2 < c_3$$

ya que:

$$(10, 10) < (15, 15) < (30, 10)$$

Por lo tanto, el conjunto ordenado resultante será:

$$\ll (m_1, (minPer_1, maxPer1)),\ (m_2, (minPer_2, maxPer2)),\ (m_3, (minPer_3, maxPer3)) \gg$$

5.2. *Abstarct factory*

Al igual que se hizo con los *piecewise maps* desordenados, también se definió la fábrica concreta para los *Piecewise maps* ordenados, para poder aplicar el patrón *Abstarct factory*.

En particular, la fábrica concreta de los *Piecewise maps* ordenados se denominó `PWMapAF`. Esta puede encontrarse en los archivos *af_pwmap*, tanto en su versión *.cpp* como *.hpp*, dentro de la carpeta `sbg` del repositorio.

5.3. Criterios de optimización y ordenamiento

En esta sección se describen en detalle los distintos **criterios de optimización** y **ordenamiento** utilizados en las operaciones sobre *piecewise maps* ordenados. Ambos conjuntos de criterios serán aplicados con el propósito de mejorar la eficiencia general de dichas operaciones y reducir significativamente los tiempos de ejecución.

Pseudocódigo y notación: En esta sección, al igual que se hizo para los *piecewise maps* desordenados, se emplearán subíndices para referirse a los mapas de un *piecewise map* ordenado con el fin de aliviar la notación. Dado un *piecewise map* ordenado A, el elemento ubicado en la posición i, lo cual corresponde a `pieces_[i].first` en C++, se denotará como A_i, donde i es un número natural que satisface $0 \leq i < \kappa(A)$, siendo $\kappa(A)$ la cantidad de mapas de A. Cabe destacar que, al tratarse de un *piecewise map* ordenado, siempre se cumple que $A_i < A_j$ si y sólo si $i < j$, para todo par de i y j tal que $0 \leq i, j < \kappa(A)$. Adicionalmente se dirá que, en el caso anterior, A_i está **antes** de A_j en A, mientras A_j está **después** de A_i.

5.3.1. Operaciones estructuralmente similares

Como se puede ver en la Sub-sección 2.5.3 existen muchas operaciones que presentan una estructura muy similar a la de la intersección entre conjuntos desordenados. Entre estas operaciones se encuentran: la **igualdad**, la **suma**, la i**gualdad de imágenes**, la **resta** y el **mínimo adyacente**.

Todas estas operaciones, junto con la intersección de conjuntos ordenados, comparten una estructura común que se puede dividir en dos partes:

1. Una **fase de comparación**, donde cada elemento de uno de los argumentos es comparado con todos los del otro argumento.

2. Un **núcleo de la operación**, donde se ejecuta la lógica específica de la operación sobre aquellos pares de elementos cuya comparación fue satisfactoria.

Esta estructura puede observarse gráficamente en el pseudocódigo presentado en 36.

En el caso de la intersección de conjuntos desordenados, la comparación consiste en verificar si la intersección entre dos multi-intervalos, uno de cada conjunto, es vacía, y el núcleo de la operación se encarga de guardar dicha intersección en el conjunto resultante.

En cambio, para las otras operaciones mencionadas, la comparación consiste en verificar si la intersección entre los dominios de dos mapas, uno de cada *piecewise map* desordenado argumento, es no vacía. El núcleo depende de la operación específica: por ejemplo, en el caso de la suma, se calcula la suma de los mapas y se almacena el resultado en el *piecewise map* desordenado de salida.

Algorithm 36 Estructura de las operaciones similares a la intersección de conjuntos desordenados

Require: A, B son conjuntos desordenados o dos *piecewise maps* desordenados

```
1:  function OPERACIÓN(A, B)
2:      ...                                  ▷ Casos base y definiciones necesarias de la operación
3:      for all a ∈ A do                     ▷ Fase de comparación de la operación.
4:          for all b ∈ B do
5:              I        ▷ Elemento a comparar proveniente de una o varias operaciones entre los elementos a y b
6:              if isEmpty(I) then
7:                  ...                      ▷ Núcleo de la operación.
8:              end if
9:          end for
10:     end for
11:     return ...
12: end function
```

Por lo tanto, dado que las operaciones sobre *piecewise maps* desordenados mencionadas anteriormente comparten esta misma estructura, y considerando todas las definiciones introducidas para los *piecewise maps* ordenados, es posible adaptar los criterios de optimización desarrollados para la intersección de conjuntos ordenados a las operaciones mencionadas de *piecewise maps* ordenados.

5.3.1.1. Criterios de optimización

Criterio de parada

Sean A y B dos *piecewise maps* ordenados. Supóngase que se está evaluando una de las operaciones mencionadas entre ambos, y en particular se consideran los posibles cálculos del núcleo de la operación entre mapa A_i de A, con $0 \leq i < \kappa(A)$, y los mapas de B. Dado un índice j tal que $0 \leq j < \kappa(B)$, vale lo siguiente:

Si el máximo perimetral del dominio de A_i es estrictamente menor que el mínimo perimetral del dominio de B_j en la primera dimensión, **entonces**:

- la intersección entre el dominio de A_i y el de $B_{j'}$ resulta vacía **y** no se calcula el núcleo de la operación para A_i y $B_{j'}$, $\forall j' \mid j \leq j' < |B|$;
- **y, entonces,** puede continuarse directamente con la fase de comparación entre A_{i+1}(si existe) y los mapas de B para validar el calculo del núcleo de la operación.

Criterio de eliminación

Sean A y B dos *piecewise maps* ordenados. Supóngase que se está evaluando una de las operaciones mencionadas entre ambos, y en particular se consideran los posibles cálculos del núcleo de la operación entre mapa A_i de A, con $0 \leq i < \kappa(A)$, y los mapas de B. Dado un índice j tal que $0 \leq j < \kappa(B)$, vale lo siguiente:

Si el máximo perimetral del dominio de (B_j es estrictamente menor que al mínimo perimetral del conjunto A_i en la primera dimensión, **entonces**:

- la intersección entre $A_{i'}$ y B_j resulta vacía **y** no se calcula el núcleo de la operación para $A_{i'}$ y B_j, $\forall i' \mid i \leq i' < \kappa(A)$;
- **y, entonces,** puede descartarse B_j para las fases de comparación de los mapas posteriores a A_i y así descartar completamente el calculo del núcleo de la operación B_j con estos.

Criterio de selección

Sean dos piecewise maps ordenados involucrados en alguna de las operaciones mencionadas. Se establece lo siguiente:

Se define como B a aquel piecewise map que contiene la menor cantidad de mapas, mientras que se denota como A al piecewise map restante.

Criterio de solapamiento

Sean A y B dos *piecewise maps*, A_i un mapa de A y B_j uno de B, con índices tales que:

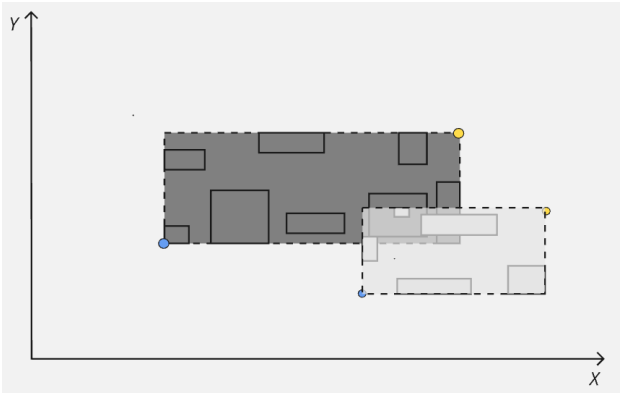
$$0 \leq i < \kappa(A), \quad 0 \leq j < \kappa(B).$$

Se establece entonces que, en el caso de realizar una de las operaciones entre A y B, y querer calcular el núcleo de la operación entre A_i y B_j:

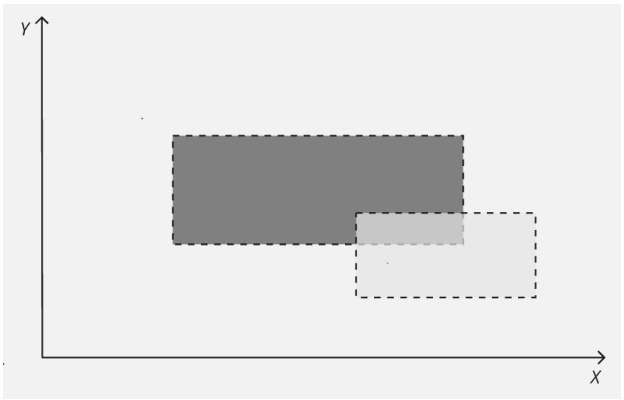
- **Si** existe solapamiento entre los conjuntos dominio de A_i y B_j, **entonces**:
  - **Si** ambos conjuntos son *densos*, la intersección entre ellos es necesariamente no vacía y se puede proceder con el calculo del núcleo de la operación.
  - **Si** al menos uno de ellos no es denso, la intersección *puede* ser no vacía, pero no se garantiza de modo que se debe proceder de igual manera.
- **Si no** existe solapamiento entre los conjuntos dominio de A_i y B_j, **entonces** la intersección entre ellos es necesariamente vacía, independientemente de si son densos o no, y por ende puede obviarse el calculo del núcleo de la operación.

En este último caso faltaría definir qué es el solapamiento entre conjuntos y cuándo un conjunto es denso para poder completar el criterio anterior:

Sean C y D dos conjuntos. Entonces se dice que C y D se solapan o están solapados si y solo si los multi-intervalos definidos a través de sus mínimos y máximos perimetrales se solapan bajo la definición de solapamiento de multi-intervalos ya vista.



(a) Conjuntos con sus multi-intervalos definidos con sus máximos y mínimos perimetrales.



(b) Solapamiento entre los multi-intervalos definidos.

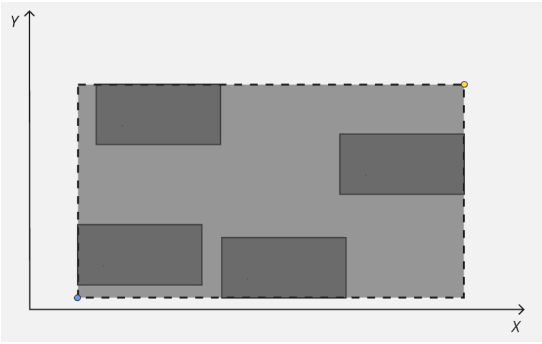
Figura 5.2: Criterios de solapamiento para *piecewise maps* ordenados.

Sea C el conjunto, y sean $minPer_C$ y $maxPer_C$ el mínimo y máximo perimetral, respectivamente. Entonces se dice que el conjunto es **denso** si y solo si, si se define un multi-intervalo denso mdi donde su mínimo y máximo sean $minPer_C$ y $maxPer_C$ respectivamente, entonces:

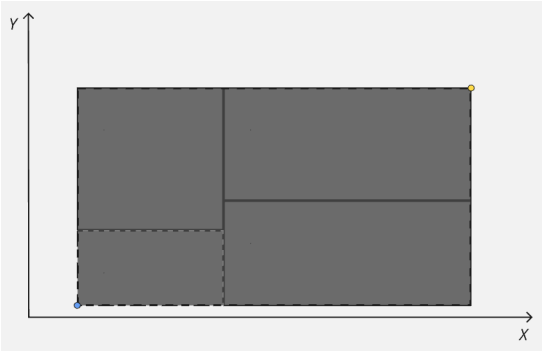
$$\{mdi\} - C = C - \{mdi\} = \emptyset$$

5.3.1.2. Criterios validos por operación

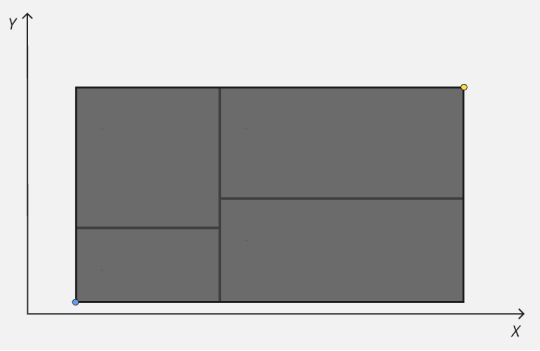
Ahora bien, el núcleo de la operación puede variar significativamente dependiendo de la naturaleza de la operación considerada. En consecuencia, el criterio de ordenamiento utilizado también puede diferir considerablemente entre una operación y otra.



(a) Conjunto no denso.



(b) Conjunto no denso.



(c) Conjunto denso.

Figura 5.3: Densidad de conjuntos.

Por esta razón, los criterios específicos serán mencionados y analizados individualmente, operación por operación.

Además, no todas las operaciones admitirán todos los criterios de optimización previamente definidos. La aplicabilidad de cada criterio dependerá de las particularidades del núcleo de la operación y del comportamiento esperado.

Igualdad - ==

Para el caso de la igualdad, se pueden aplicar todos los criterios de optimización previamente definidos.

En cuanto a su **criterio de ordenamiento**, esta operación no requiere de uno, ya que no produce como resultado un *piecewise map* ordenado. En consecuencia, no dispone de un criterio de ordenamiento asociado.

suma - +

En el caso de la suma, vuelven a aplicarse todos los criterios de optimización mencionados. Y, en cuanto a su criterio de ordenamiento, este resulta muy similar al propuesto para la intersección de conjuntos ordenados, con las modificaciones y adaptaciones necesarias claro esta.

Criterio de ordenamiento

Supóngase que se realiza la suma entre dos *piecewise maps* ordenados, A y B, y que se están evaluando las posibles sumas del i-ésimo mapa de A, denotado por A_i, con todos los mapas de B. Ademas se cuenta con un *piecewise map* resultado C.

Todas las sumas no vacías generadas con A_i deben insertarse en C **después** de aquellas sumas generadas por los mapas $A_0, A_1, \ldots, A_{i-1}$, cuyos mínimos perimetrales sean estrictamente menores al mínimo perimetral de A_i, bajo el operador $<$ de naturales multi-dimensionales.

Este criterio se basa nuevamente en la observación de que, al realizar la *suma*, su dominio corresponde a la intersección de los dominios de los mapas participantes. Dicha intersección entre dos dominios está contenida en ambos, lo que implica que su mínimo perimetral se encuentra dentro de los multi-intervalos definidos por el

mínimo y el máximo perimetral de los operandos.

Al fijar A_i, todas sus sumas posibles con mapas de B tendrán su mínimo perimetral dentro del multi-intervalo definido por el mínimo y el máximo perimetral de A_i. Como consecuencia, cualquier suma con dominio no vacío generada tendrá un mínimo perimetral mayor o igual que el mínimo perimetral de A_i, bajo el operador $<$ de los naturales multi-dimensionales.

Esto garantiza que tales sumas deben insertarse en el *piecewise map* resultado después de aquellas cuyo dominio tenga un mínimo perimetral que sea estrictamente menor,bajo el operador $<$ de los naturales multi-dimensionales, al de A_i, es decir, las generadas por los mapas anteriores de A. Una vez procesado A_i, se avanza hacia A_{i+1} y se continúa la construcción del *piecewise map* resultado de la misma manera.

La Figura 5.4 ilustra gráficamente esta situación. Allí se observa todas las intersecciones de los dominios de las sumas producidas a partir del mapa A_i con los mapas de B. En particular los dominios tienen el mismo nombre que los mapas a los que pertenecen. Y como se ve, estas sumas que realizan la intersección de sus dominios tienen un mínimo perimetral mayor o igual que el del dominio de A_i, y se insertan a continuación de los mapas previamente procesadas cuyo mínimo perimetral sea menor al de A_i. Esto mismo ocurre con las de A_{i+1} cuya suma con B_k se debe insertar después de los mapas con $A1$ y $A2$ como dominio.

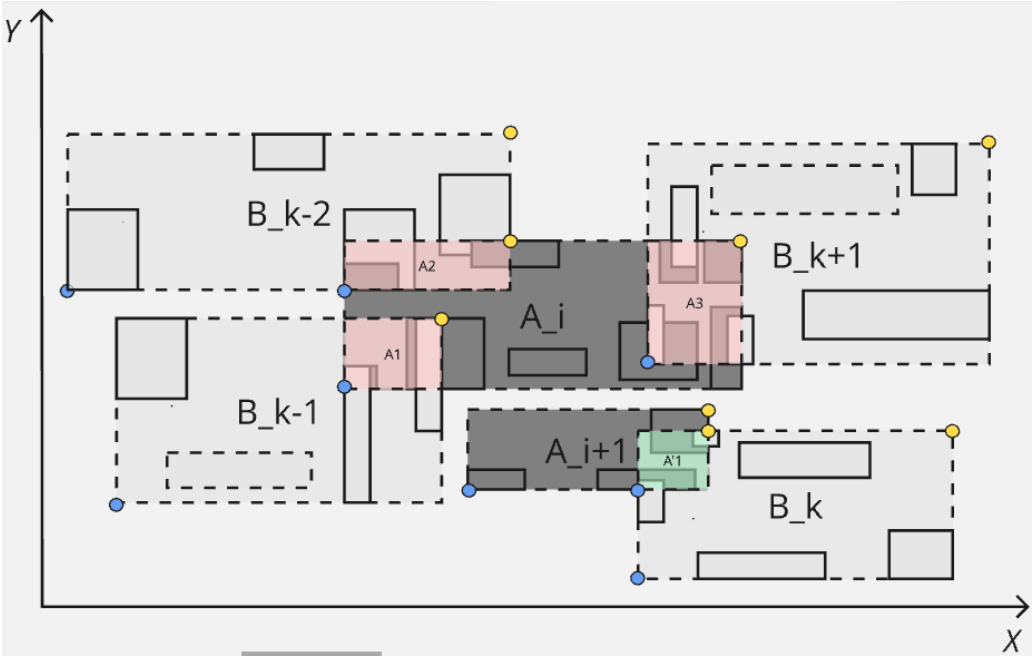


Figura 5.4: Criterio de ordenamiento de la suma para *piecewise map* ordenados de las sumas en función de su dominio y, en particular, de sus mínimos perimetrales.

Adicionalmente, no es necesario comenzar a verificar la posición de inserción de las sumas en el *piecewise map* resultado desde el principio en cada iteración de los elementos de A. Dado el orden intrínseco de los *piecewise map* involucrados, se cumple que

$$minPer_i \leq minPer_{i+1}$$

donde $minPer_i$ y $minPer_{i+1}$ son los minimos perimetrales de A_i y A_{i+1} respectivamente.

Esto implica que las sumas generadas por A_{i+1} necesariamente deben insertarse a partir de una posición igual o posterior a aquella donde comenzaron a colocarse las sumas de A_i con los elementos de B.

Esto desencadena la siguiente modificación del criterio de ordenamiento:

Criterio de ordenamiento

Supóngase que se realiza la suma entre dos *piecewise maps* ordenados, A y B, y que se están evaluando las posibles sumas del i-ésimo mapa de A, denotado por A_i, con todos los mapas de B. Ademas se cuenta con un *piecewise map* resultado C.

Todas las sumas no vacías generadas con A_i deben insertarse en C **después** de aquellas sumas generadas por los mapas $A_0, A_1, \ldots, A_{i-1}$, cuyos mínimos perimetrales sean estrictamente menores al mínimo perimetral A_i, bajo el operador $<$ de naturales multi-dimensionales.

Adicionalmente **si** la posición a partir de la cual se colocan las sumas de A_i en el *piecewise map* resultante es j, con $0 \leq j < \kappa(C)$, **entonces** aquellas generadas por A_{i+1} se insertaran a partir de j' tal que $j \leq j' < \kappa(C)$.

Igualdad de imágenes - `equalImage`

El caso de esta operación es muy similar al de la operación de igualdad: admite todos los criterios de optimización y tampoco requiere un criterio de ordenamiento. Esta última característica, a diferencia del caso de la igualdad, no se debe a que la operación devuelva un valor de verdad, sino a que su resultado es un conjunto en sí mismo, el cual tiene su implementación propia y no se debe alterar.

Mínimo adyacente - `minAdjMap`

Ahora es el turno de la operación `minAdjMap`, la cual puede utilizar todos los criterios de optimización, salvo el de selección, ya que nuevamente el orden de los argumentos tiene relevancia.

En cuanto al criterio de ordenamiento, aparece un problema que se repite tanto aquí como en varias operaciones sobre *piecewise maps* ordenados: la imagen. La imagen de un mapa, aunque es un conjunto, posee una cohesión que depende tanto del conjunto dominio como de la colección de expresiones lineales del mapa. Por ejemplo, dos mapas pueden tener un mismo dominio, pero basta con que alguna de sus expresiones varíe para que sus imágenes sean distintas. Algo similar ocurre si varía el dominio.

Entonces, ordenar mapas cuyo dominio corresponde a la imagen resulta una tarea particularmente compleja. En estos casos, el orden del *piecewise map* original aporta escasa o nula información útil para ordenar los mapas derivados, cuyos dominios provienen de una imagen.

Una alternativa es realizar un ordenamiento estrictamente lineal, lo cual implica un costo computacional elevado, especialmente cuando el *piecewise map* resultante contiene un gran número de mapas.

En conclusión, para poder brindar un orden a un *piecewise map* cuyo contenido son mapas con dominios derivados de imágenes, sería necesario tener en cuenta no solo el dominio, sino también las expresiones y la forma de dichas expresiones. Por esta razón, se concluyó que lo más óptimo es utilizar un algoritmo de ordenamiento preexistente, lo cual simplifica el entendimiento de esta y de las demás operaciones de *piecewise maps* ordenados. Así se evita la necesidad de escribir y mantener código de criterios de ordenamiento que, probablemente, tendría un rendimiento inferior.

Resta - `-`

Por ultimo se encuentra la operación `-`, la cual admite todos los criterios de optimización, con excepción del criterio de selección, ya que se trata de una operación donde el orden de los argumentos es relevante y, por lo tanto, no puede alterarse. En cuanto al orden de la salida, ocurre algo particular.

En el núcleo de la operación desordenada, cuando se calcula la resta entre dos mapas, el proceso genera un resultado parcial únicamente para los dos mapas involucrados. El tamaño de este resultado parcial, o la cantidad de mapas que contiene, depende exclusivamente de la cantidad de dimensiones con las que se esté trabajando. Luego, este resultado parcial se incorpora al *piecewise map* resultado de la operación.

Ahora bien, para garantizar que la salida se encuentre correctamente ordenada, se optó por emplear un algoritmo de ordenamiento preexistente. De este modo, se evita mantener la invariante de orden durante la operación en el *piecewise map* resultante y en el resultado parcial, y en su lugar, antes de devolver el *piecewise map* resultante, se procede a ordenarlo utilizando dicho algoritmo.

De esta forma, se evita recurrir a estrategias ineficientes, como ordenar cada resultado parcial para luego aplicar la operación `concatenation`.

5.3.2. Restricción de dominio - `restrict`

El caso de la operación `restrict` es diferente al de las operaciones vistas anteriormente. Sin embargo, esto no significa que no puedan adaptarse los criterios definidos previamente para esta operación.

En particular, la operación `restrict` aplicada a un *piecewise map* desordenado, como ya se vio, consiste en restringir el dominio de cada uno de sus mapas individuales en base a un conjunto, siendo ambos argumentos de la operación, y quedando como parte del resultado solo aquellos mapas del *piecewise map* desordenado cuyos dominios restringidos son no vacíos.

Ahora bien, al dotar de orden al *piecewise map* argumento y requerir que el resultado también esté ordenado, la operación puede beneficiarse de los siguientes criterios de optimización y ordenamiento:

5.3.2.1. Criterios de optimización

Criterio de parada

Sean A un *piecewise map* ordenado y S un conjunto. Supóngase que se evalúa la restricción de dominio entre ambos, y en particular se considera la posible restricción de domino del dominio de un mapa A_i de A, con $0 \leq i < \kappa(A)$, con respecto a S. Entonces vale lo siguiente:

Si el máximo perimetral de S es estrictamente menor que el mínimo perimetral del conjunto dominio de A_i en la primera dimensión, **entonces**:

- la intersección entre el dominio de A_i y S resulta vacía **y** no se calcula la restricción de dominio para $A_{i'}$ con respecto a S, $\forall i' \mid i \leq i' < \kappa(A)$;
- **y, entonces,** puede terminarse directamente con la operación `restrict`.

Criterio de solapamiento

Sean A un *piecewise map*, S un conjunto y A_i un mapa de A tal que:

$$0 \leq i < \kappa(A).$$

Se establece entonces que, en el caso de realizar la restricción de dominio de A sobre S y querer calcular la restricción de A_i con respecto a S:

- **Si** existe solapamiento entre el conjunto dominio de A_i y S, **entonces**:
  - **Si** ambos conjuntos son *densos*, la intersección entre ellos es necesariamente no vacía y se puede proceder con el calculo de la restricción del dominio de A_i en base a S.
  - **Si** al menos uno de ellos no es denso, la intersección *puede* ser no vacía, pero no se garantiza de modo que se debe proceder de igual manera.
- **Si no** existe solapamiento entre el conjunto dominio de A_i y S, **entonces** la intersección entre ellos es necesariamente vacía, independientemente de si son densos o no, y por ende puede obviarse el calculo de la restricción del dominio de A_i en base a S.

5.3.2.2. Criterio de ordenamiento

En este caso, lo que debe mantenerse ordenado en el *piecewise map* resultante son todas las restricciones de dominio no vacías de los mapas del *piecewise map* ordenado argumento. La restricción de dominio de un mapa consiste únicamente en reemplazar su dominio original por la intersección entre dicho dominio y el conjunto que también llega como argumento.

Ahora bien, como se discutió en el criterio de ordenamiento para la suma, el mínimo perimetral del dominio

restringido está contenido dentro de alguno de los multi-intervalos definidos por los máximos y mínimos perimetrales del dominio del mapa a restringir y del conjunto que restringe. Ademas aquí también vale que no es necesario buscar la posición de inserción desde el comienzo del *piecewise map* resultado. Esto permite redefinir el criterio de ordenamiento de la siguiente manera:

Criterio de ordenamiento

Supóngase que se realiza la restricon de dominio entre un *piecewise map* ordenado A y un conjunto S, y que se están evaluando la restricción de dominio del i-ésimo mapa de A, denotado por A_i, con respecto a S. Ademas se cuenta con un *piecewise map* resultado C.

La restricción no vacía generada con A_i deben insertarse en C **después** de aquellas restricciones no vacías generadas por los mapas $A_0, A_1, \ldots, A_{i-1}$, cuyos mínimos perimetrales sean estrictamente menores al mínimo perimetral de A_i, bajo el operador $<$ de naturales multi-dimensionales.

Adicionalmente **si** la posición a partir de la cual se colocan las sumas de A_i en el *piecewise map* resultante es j, con $0 \leq j < \kappa(C)$, **entonces** aquellas generadas por A_{i+1} se insertaran a partir de j' tal que $j \leq j' < \kappa(C)$.

5.3.3. Combinación - `combine`

El siguiente caso que se analizará es el de la operación `combine`, una de las ya introducidas previamente en el capítulo de conceptos preliminares.

Esta operación, como ya se vio, aplicada sobre *piecewise maps* desordenados, toma dos de ellos y realiza una suerte de unión, devolviendo un único *piecewise map* desordenado. Este contiene todos los mapas del primero, junto con aquellos mapas del segundo cuyos dominios son una restricción, conteniendo ahora solo aquellos valores no presentes en el dominio total del primer *piecewise map*.

Debido a la naturaleza de su funcionamiento y requerir preservar el orden en el resultado, esta operación emplea únicamente un criterio de optimización y un criterio de ordenamiento, ambos reformulados de los vistos para las *operaciones estructuralmente similares*.

5.3.3.1. Criterio de optimización

Trabajando con dos *piecewise maps* A y B, los únicos mapas de B que pueden formar parte del resultado son aquellos donde la diferencia de su domino con el dominio total de A sea no vacía. El dominio modificado resultante para cada uno de los mapas de B será precisamente dicha diferencia. Por lo tanto, lo único que se puede hacer es omitir la diferencia cuando se sabe que no es necesaria, y de esta observación se deriva el siguiente criterio de solapamiento:

Criterio de solapamiento

Sean A y B dos *piecewise maps*, B_j un mapa de B, tal que:

$$0 \leq j < \kappa(B).$$

y sea S el dominio de A. Se establece entonces que, en el caso de realizar la combinación entre A y B y al estar considerando B_j :

- **Si** existe solapamiento entre el conjunto dominio de B_j y S, **entonces**:
  - **Si** ambos conjuntos son *densos*, la intersección entre ellos es necesariamente no vacía y se debe proceder con el calculo de la diferencia para obtener el nuevo dominio de B_j.
  - **Si** al menos uno de ellos no es denso, la intersección *puede* ser no vacía, pero no se garantiza de modo que se debe proceder de igual manera.
- **Si no** existe solapamiento entre el conjunto dominio de B_j y S, **entonces** la intersección entre ellos es necesariamente vacía, independientemente de si son densos o no, y por ende puede obviarse el calculo de la diferencia,la cual devolvería el mismo dominio de B_j, y el dominio de B_k queda intacto.

5.3.3.2. Criterio de ordenamiento

En cuanto al orden, al realizar la diferencia entre dos conjuntos, por ejemplo, entre dos conjuntos C y D, el nuevo mínimo perimetral del conjunto resultante $C - D$ debe encontrarse contenido dentro del multi-intervalo definido por el mínimo y el máximo perimetral del conjunto original C.

Esto se debe a que la operación de diferencia remueve elementos de C, sin añadir otros nuevos. Esta observación es casi análoga a lo que ocurría en el caso de la operación de suma, discutido en en este capitulo con respecto a la suma de *piecewise map* ordenados. Análogamente a lo visto también en la suma, gracias a lo anterior y al orden de los *piecewise maps*, ocurre que si B_j con su dominio modificado se debería insertar a partir de una posición i, con $0 \leq i < \kappa(C)$, en el *piecewise map* resultado, entonces B_j con su dominio modificado debería ubicarse en i' tal que $i \leq i' < \kappa(C)$

En consecuencia, se postula el siguiente criterio de ordenamiento derivado del visto para el operador `+`:

Criterio de ordenamiento

Supóngase que se realiza la combinación entre dos *piecewise maps* ordenados, A y B, y que se deba calcular el nuevo dominio para el mapa j-ésimo de B, denotado por B_j, con respecto al dominio de A, S. Ademas se cuenta con un *piecewise map* resultado C.

Si el nuevo dominio de B_j no es vacío, es decir, la resta entre el conjunto dominio de B_j y S es no vacía; **entonces** el mapa B_j con dominio restringido por S estará ubicado en el *piecewise map* resultante después de todos aquellos mapas cuyo mínimo perimetral sea menor que el minino perimetral del dominio original de B_j, bajo el operador $<$ de naturales multi-dimensionales.

Adicionalmente **si** la posición a partir de la cual se coloca de B_j con dominio modificado en el *piecewise map* resultante es i, con $0 \leq i < \kappa(C)$, **entonces** el mapa modificado de B_{j+1} se insertaría, si su dominio no es vacío, a partir de i' tal que $i \leq i' < \kappa(C)$.

5.3.4. Concatenación - `concatenation`

El caso de la operación concatenación, es básicamente homólogo al del la unión disjunta de conjuntos, y por ende haciendo las respectivas modificaciones, se puede reutilizar lo visto para la unión disjunta de conjuntos ordenados aquí.

5.3.4.1. Criterio de ordenamiento

Criterio de ordenamiento

Sean $A = \{A_0, A_1, \ldots, A_{n-1}\}$ y $B = \{B_0, B_1, \ldots, B_{m-1}\}$ dos *piecewise maps* no vacíos, disjuntos y ordenados. Sea C un *piecewise map* inicialmente vacío que contendrá el resultado de la fusión ordenada de A y B. Y sean A_i un mapa de A y B_j un mapa de B, con índices tales que:

$$0 \leq i < \kappa(A), \quad 0 \leq j < \kappa(B).$$

Entonces, al realizar la unión disjunta entre A y B vale que:

- **Si** el mínimo perimetral de A_i es menor, bajo el operador $<$ definido para naturales multi-dimensionales, que el mínimo perimetral de B_j, entonces A_i debe aparecer **antes** que B_j en el *piecewise map* C.
- **Si** el mínimo perimetral de A_i no es menor, bajo el operador $<$ definido para naturales multi-dimensionales, que el mínimo perimetral de B_j, entonces B_j debe aparecer **antes** que A_i en el *piecewise map* C.

5.3.5. Desplazamiento de dominio - `offsetDom`

Al buscar optimizar y ordenar la salida en la operación `offsetDom`, cuando esta recibe como argumentos dos *piecewise maps* ordenados, es posible adaptar y reutilizar los criterios previamente establecidos para la operación `restrict`.

Como se observa en su versión desordenada, para que un mapa de A sea incluido en el resultado, debe existir un desplazamiento no vacío generado por la imagen de O en base al dominio del mapa. Ahora bien, al trabajar con *piecewise maps* ordenados, es posible anticipar si dicho desplazamiento será vacío o no. Para ello, pueden aplicarse criterios de optimización derivados, y adaptados, de aquellos utilizados en la restricción de dominio.

Cabe destacar que, al trabajar con la imagen de un *piecewise map* para generar la salida, nuevamente se hará uso de un algoritmo de ordenamiento en lugar de diseñar un criterio de ordenamiento específico, como se hizo para *minAdjMap*.

5.3.5.1. Criterios de optimización

Criterio de parada

Sean A y O *piecewise maps* ordenados y S el conjunto dominio de O. Supóngase que se evalúa el desplazamiento de dominio entre A y O, y en particular se considera el posible desplazamiento de dominio del dominio de un mapa A_i de A, con $0 \leq i < \kappa(A)$, con respecto a O. Entonces vale lo siguiente:

Si el máximo perimetral de S es estrictamente menor que el mínimo perimetral del conjunto dominio de A_i en la primera dimensión, **entonces**:

- la intersección entre el dominio de A_i y S resulta vacía **y** no se calcula el desplazamiento de dominio para $A_{i'}$ con respecto a S, $\forall i' \mid i \leq i' < \kappa(A)$;
- **y, entonces,** puede terminarse directamente con la operación `offsetDom`.

Criterio de solapamiento

Sean A y O *piecewise maps* ordenados, S el conjunto dominio de O y A_i un mapa de A tal que:

$$0 \leq i < \kappa(A).$$

Se establece entonces que, en el caso de realizar el desplazamiento de dominio de A en base a O, y querer desplazar el dominio de A_i en base a O:

- **Si** existe solapamiento entre el conjunto dominio de A_i y S, **entonces**:
  - **Si** ambos conjuntos son *densos*, la intersección entre ellos es necesariamente no vacía y se puede proceder con el calculo del desplazamiento de dominio de A_i en base a O.
  - **Si** al menos uno de ellos no es denso, la intersección *puede* ser no vacía, pero no se garantiza de modo que se debe proceder de igual manera.
- **Si no** existe solapamiento entre el conjunto dominio de A_i y S, **entonces** la intersección entre ellos es necesariamente vacía, independientemente de si son densos o no, y por ende puede obviarse el calculo del desplazamiento de dominio de A_i en base a O.

5.3.6. Composición - `composition`

Al intentar optimizar y ordenar el *piecewise map* resultante en la operación `composition`, cuando esta recibe como argumentos dos *piecewise maps* ordenados, es posible adaptar y reutilizar los criterios previamente establecidos para la operación suma y para las *operaciones estructuralmente similares*.

Como se observa en su versión desordenada optimizada 22, para que el resultado de componer un mapa de A, a, con uno de B, b, sea incluido en el resultado final, debe tener un dominio no vacío. Es decir, la intersección del domino a con la imagen de b debe ser no vacía. Teniendo esto último presente se pueden redefinir los criterios de optimización para las *operaciones estructuralmente similares* y el criterio de ordenamiento que se le dio a la suma.

5.3.6.1. Criterios de optimización

Criterio de parada

Sean A y B dos *piecewise maps* ordenados. Supóngase que se está evaluando la composición entre ambos, y en particular se consideran las posibles composiciones entre un mapa B_j de B, con $0 \leq j < \kappa(B)$, y los mapas de A. Dado un índice i tal que $0 \leq i < \kappa(A)$, vale lo siguiente:

Si el máximo perimetral del dominio de A_i es estrictamente menor que el mínimo perimetral del conjunto imagen de B_j en la primera dimensión, **entonces**:

- la intersección entre el dominio de $A_{i'}$ y la imagen de B_j resulta vacía **y** no se calcula la composición entre $A_{i'}$ y B_j, $\forall i' \mid i \leq i' < \kappa(A)$;
- **y, entonces,** puede continuarse directamente con verificando las posibles composiciones de B_{j+1}(si existe) con los mapas de A.

Criterio de solapamiento

Sean A y B dos *piecewise maps*, A_i un mapa de A y B_j uno de B, con índices tales que:

$$0 \leq i < \kappa(A), \quad 0 \leq j < \kappa(b).$$

Se establece entonces que, en el caso de realizar la composición entre A y B, y querer calcular la composición de A_i y B_j:

- **Si** existe solapamiento entre los conjuntos dominio de A_i e imagen de B_j, **entonces**:
  - **Si** ambos conjuntos son *densos*, la intersección entre ellos es necesariamente no vacía y se puede proceder con el calculo de la composición entre A_i y B_j.
  - **Si** al menos uno de ellos no es denso, la intersección *puede* ser no vacía, pero no se garantiza de modo que se debe proceder de igual manera.
- **Si no** existe solapamiento entre los conjuntos dominio de A_i e imagen de B_j, **entonces** la intersección entre ellos es necesariamente vacía, independientemente de si son densos o no, y por ende puede obviarse el calculo de la composición entre A_i y B_j.

5.3.6.2. Criterio de ordenamiento

En este caso tenemos que al realizar la composición entre dos mapas, su dominio es un subconjunto del dominio original del primer mapa a componer. Por ende, al fijar el primer mapa, todos sus composiciones no vacías con otros mapas, tendrán un domino con un mínimo perimetral igual o menor que el del dominio del mapa fijado, bajo el operador $<$ de naturales multi-dimensionales. Esto es muy similar a lo que ocurría en la operación *restrict*. Ademas aquí también vale que no es necesario buscar la posición de inserción desde el comienzo del *piecewise map* resultado como se vio para las restricción de dominio.

Criterio de ordenamiento

Supóngase que se realiza la composición entre dos *piecewise maps* ordenados, A y B, y que se están evaluando las posibles composiciones del j-ésimo mapa de B, denotado por B_j, con todos los mapas de A. Ademas se cuenta con un *piecewise map* resultado C.

Todas las composiciones no vacías generadas con B_j deben insertarse en C **después** de aquellas composiciones generadas por los mapas $B_0, B_1, \ldots, B_{k-1}$, cuyos mínimos perimetrales sean estrictamente menores al mínimo perimetral de B_j, bajo el operador $<$ de naturales multi-dimensionales.

Adicionalmente **si** la posición a partir de la cual se colocan las composiciones de B_j en el *piecewise map* resultante es i, con $0 \leq i < |C|$, **entonces** aquellas generadas por B_{j+1} se insertaran a partir de i' tal que $i \leq i' < \kappa(C)$.

5.3.7. Pseudoinversa - `firstInv`

El caso de la operación `firstInv` resulta particularmente interesante, ya que permite aplicar criterios de optimización tanto entre los elementos de la entrada, como en elementos del propio algoritmo interno de la misma operación.

Tal como se observa en su versión desordenada y optimizada, para que un mapa A sea considerado para el cálculo interno de la operación, la diferencia entre la imagen del mapa restringida al subdominio S y el conjunto de elementos ya visitados, denotado como *visited*, debe ser no vacía. Esto puede suceder si, y solo si, se cumplen simultáneamente las siguientes condiciones:

- La imagen de A restringida al subdominio S es no vacía.
- La diferencia entre dicha imagen y el conjunto *visited* es también no vacía.

Esto permite aplicar entonces dos conjuntos de criterios de optimización: uno entre los mapas del *piecewise map* y el conjunto que llegan como argumento y otro entre los mapas del *piecewise map* y el conjunto de valores visitados *visited*.

Dado que aquí también se trabaja sobre la imagen en la generación de la salida, el ordenamiento de la misma queda delegado a un algoritmo de ordenamiento preexistente nuevamente. No obstante, los criterios de optimización aplicables serán presentados a continuación.

5.3.7.1. Criterios de optimización

Criterio de parada

Sean A un *piecewise map* ordenado y S un conjunto. Supóngase que se evalúa la pseudoinversa entre ambos, y en particular se considera la posible pseudoinversión de un mapa A_i de A, con $0 \leq i < \kappa(A)$, con respecto a S. Entonces vale lo siguiente:

Si el máximo perimetral de S es estrictamente menor que el mínimo perimetral del conjunto dominio de A_i en la primera dimensión, **entonces**:

- la intersección entre el dominio de A_i y S resulta vacía **y** no se calcula la pseudoinversión de $A_{i'}$ con respecto a S, $\forall i' \mid i \leq i' < \kappa(A)$;
- **y, entonces,** puede terminarse directamente con la operación `firstInv`.

Criterio de solapamiento

Sean A un *piecewise map*, S un conjunto y A_i un mapa de A tal que:

$$0 \leq i < \kappa(A).$$

Se establece entonces que, en el caso de realizar la pseudoinversa de A en base a S y querer calcular la pseudoinversión de A_i con respecto a S:

- **Si** existe solapamiento entre el conjunto dominio de A_i y S, **entonces**:
  - **Si** ambos conjuntos son *densos*, la intersección entre ellos es necesariamente no vacía y se puede proceder con el calculo de la imagen de A_i en base a S para el calculo de la pseudoinversión de A_i en base a S.
  - **Si** al menos uno de ellos no es denso, la intersección *puede* ser no vacía, pero no se garantiza de modo que se debe proceder de igual manera.
- **Si no** existe solapamiento entre el conjunto dominio de A_i y S, **entonces** la intersección entre ellos es necesariamente vacía, independientemente de si son densos o no, y por ende puede obviar el calculo de la imagen de A_i en base a S y el calculo de la pseudoinversión de A_i en base a S.

Criterio de solapamiento para *visited*

Sean A un *piecewise map*, S un conjunto y A_i un mapa de A tal que:

$$0 \leq i < \kappa(A),$$

sea I la imagen no vacia de A_i en base a S y *visited* el conjunto de valores ya visitados por las pseudoinversas de los mapas anteriores a A_i. Se establece entonces que, en el caso de realizar la pseudoinversa de A en base a S y querer calcular la pseudoinversión de A_i con respecto a S:

- **Si** existe solapamiento entre el conjunto I y *visited*, **entonces**:
  - **Si** ambos conjuntos son *densos*, la intersección entre ellos es necesariamente no vacía y se puede proceder con el calculo de la pseudoinversión de A_i en base a S y utilizando $I - visited$ de no ser vacía.
  - **Si** al menos uno de ellos no es denso, la intersección *puede* ser no vacía, pero no se garantiza de modo que se debe proceder de igual manera.
- **Si no** existe solapamiento entre el conjunto dominio de A_i y S, **entonces** la intersección entre ellos es necesariamente vacía, independientemente de si son densos o no, y por ende se puede proceder con el calculo de la pseudoinversión de A_i en base a S y utilizando I.

5.4. Implementaciones adicionales

Al igual que en el caso de los conjuntos ordenados, aquí también se definieron ciertas operaciones adicionales que resultan útiles para la implementación de los *piecewise maps* ordenados, pero que no pueden formar parte de la interfaz de los *piecewise maps*.

Pseudocódigo y notación: A partir de este capitulo en adelante se utilizara la siguiente notación para simplificar lo mas posible el pseudocódigo:

- Se utilizara la notación $\llcorner\urcorner$ para hacer referencia a una entrada de mapa (`MapEntry`).
- Se utilizara la notación $\lfloor\rceil$ para hacer referencia a un perímetro (`SetPerimeter`).
- Ahora A_i representa la i-esima entrada de mapa o `MapEntry` de A siendo A un *piecewise map* ordenado y con $0 \leq i < \kappa(A)$. Es el equivalente a `A.pieces_[i]` en C++.
- Dada a una entrada de mapa (`MapEntry`) de un *piecewise map* ordenado A:
  - maxPer(a): en C++ es equivalente a `a.second.second`, lo que permite acceder al máximo perimetral asociado al conjunto dominio del mapa contenido en la entrada de mapa.
  - minPer(a): en C++ es equivalente a `a.second.first`, lo que permite acceder al mínimo perimetral asociado al conjunto dominio del mapa contenido en la entrada de mapa a.
  - map(a): es equivalente a, en C++, `a.first` que permite acceder al mapa que está contenido en la entrada de mapa a.
  - setPer(a): es equivalente a, en C++, `a.second` que permite acceder al perímetro del mapa que está contenido en la entrada de mapa a.
- Dado p un perímetro (`SetPerimeter`) de un entrada de mapa:
  - maxPer(p): es equivalente a, en C++, `p.second` que permite acceder al máximo perimetral que tiene asociado el perímetro p.
  - minPer(p): es equivalente a, en C++, `a.first` que permite acceder al mínimo perimetral que tiene asociado el perímetro p.

5.4.1. Calculo de perímetro - `calculatePerimeter`

Esta operación adicional permite calcular el perímetro de un conjunto, es decir, determinar su mínimo y máximo perimetral.

La función `calculatePerimeter` resulta fundamental, ya que proporciona estos dos valores clave que son utilizados tanto para establecer el orden de los mapas dentro de un *piecewise map* ordenado, así como también para habilitar todas las optimizaciones descritas en la Sección 5.3.

Algorithm 37 Cálculo del perímetro de un conjunto

Require: S es un conjunto de multi-intervalos.
Ensure: Devuelve un perímetro.

```
1:  function CALCULATEPERIMETER(S)
2:      d := ARITY(S)
3:      max_per := (0)^d                          ▷ MD_NAT de ceros de longitud d
4:      min_per := (Inf)^d                        ▷ MD_NAT de valores Inf de longitud d
5:      for all s ∈ S do
6:          candidate_max := MAXELEM(s)
7:          candidate_min := MINELEM(s)
8:          for i = 0 to d − 1 do
9:              max_per[i] := máx(max_per[i], candidate_max[i])
10:             min_per[i] := mín(min_per[i], candidate_min[i])
11:         end for
12:     end for
13:     return ⌞min_per, max_per⌝                 ▷ Perímetro resultante
14: end function
```

5.4.2. Cálculo de solapamiento - `doInt`

Esta operación funciona como análoga a su homónima definida para conjuntos ordenados, que manejaba solapamiento entre multi-intervalos. Su propósito es verificar si existe solapamiento entre dos conjuntos, utilizando únicamente sus valores mínimos y máximos perimetrales de los conjuntos.

Para ello, se analizan los perímetros de los conjuntos y se evalúa si existe solapamiento entre ellos.

El pseudocódigo correspondiente puede verse a continuación en el Algoritmo 38.

Algorithm 38 Calculo de solapamiento de conjuntos

Require: P_1 y P_2 son dos perímetros.
Ensure: Devuelve `true` si los conjuntos de los perímetros se solapan, `false` en caso contrario.

```
1:  function DOINT(P_1, P_2)
2:      max_1 := maxPer(P_1)
3:      min_1 := minPer(P_1)
4:      max_2 := maxPer(P_2)
5:      min_2 := minPer(P_2)
6:      d := ARITY(max_1)
7:      for j = 0 to d − 1 do
8:          if max_1[j] < min_2[j] or max_2[j] < min_1[j] then
9:              return false                      ▷ No hay solapamiento
10:         end if
11:     end for
12:     return true                               ▷ Solapamiento detectado
13: end function
```

5.4.3. Creación de entradas de mapas - `createMapEntry`

Esta operación se encarga simplemente de gestionar la creación de una entrada de mapa, *MapEntry*, a partir de un mapa dado como argumento. Durante este proceso, se calcula su perímetro con el fin de completar la

información faltante y construir correctamente la entrada de mapa.

El pseudocódigo correspondiente puede verse en el Algoritmo 39.

Algorithm 39 Función de creación de entradas para mapas

Require: M es un mapa.
Ensure: Devuelve una entrada de mapa.
1: **function** CREATEMAPENTRY(M)
2: $sp :=$ calculatePerimeter(DOM(M))
3: **return** $\lfloor M,\ sp \rceil$ ▷ Entrada de mapa resultante
4: **end function**

5.4.4. Comparación de entradas de mapas - `mapEntryComp`

Esta operación actúa únicamente como un predicado, tomando dos entradas de mapa (`MapEntry`) y comparando cuál de ellas tiene un mapa menor, basándose exclusivamente en los mínimos perimetrales que ambas entradas contienen dentro sus correspondientes perímetros.

En particular, esta función tiene un único uso dentro de toda la implementación de *piecewise map* ordenados: se utiliza como operador de comparación en el algoritmo de ordenamiento preexistente que se menciono en las secciones anteriores.

El pseudocódigo correspondiente a esta operación es sumamente simple y puede verse en el Algoritmo 40.

Algorithm 40 Comparación de mínimos perimetrales de dos entradas de mapas

Require: E_1 y E_2 son dos entradas de mapas
Ensure: Devuelve `true` si el mínimo perimetral de E_1 es menor que el de E_2
1: **function** MAPENTRYCOMP(E_1, E_2)
2: $min\_1 :=$ minPer(E_1)
3: $min\_2 :=$ minPer(E_2)
4: **return** $min\_1 < min\_2$
5: **end function**

5.4.5. Agregar entrada al final - `pushBack`

Esta operación permite insertar una entrada de mapa al final de un *piecewise map* ordenado, asumiendo que su dominio no esté vacío. Su propósito principal es facilitar la incorporación de nuevas entradas sin necesidad de realizar un ordenamiento lineal completo, en los casos en que no se desea o no sea necesario verificar nuevamente si el dominio del mapa a insertar es vacío.

El pseudocódigo correspondiente se muestra en el Algoritmo 41.

Algorithm 41 Agregar entrada al final para *piecewise maps* ordenados

Require: A es un *piecewise map* ordenado y E es una entrada para mapa cuyo mapa tiene un dominio no vacío.
Ensure: Agrega E al final de A
1: **function** PUSHBACK(A, E)
2: $A := A \frown \ll E \gg$
3: **end function**

5.4.6. Agregar mapa al final - `pushBack`

Esta operación permite insertar un mapa al final de un *piecewise map* ordenado, asumiendo que su dominio no esté vacío. Es básicamente análoga a la operación adicional anterior.

El pseudocódigo correspondiente se muestra en la figura 42.

Algorithm 42 Agregar mapa al final para *piecewise maps* ordenados

Require: A es un *piecewise map* ordenado y M es un mapa con dominio no vació.
Ensure: Agrega E al final de A
1: **function** PUSHBACK(A, M)
2: $A := A \frown\!\!\ll \text{CREATEMAPENTRY}(M) \gg$
3: **end function**

5.4.7. Inserción con pista - `emplaceHint`

Esta operación inserta un nuevo mapa en un *piecewise map* ordenado utilizando una posición sugerida (*hint*), que es un número positivo. A partir de dicha pista, se avanza hasta encontrar la ubicación adecuada según el valor mínimo perimetral del mapa a insertar en comparación con los mínimos perimetrales de las entradas existentes en el *piecewise map*. Esto permite mantener el orden sin necesidad de recorrer toda la estructura.

El pseudocódigo correspondiente se muestra en el Algoritmo 43.

Algorithm 43 Inserción ordenada de mapas con pista para *piecewise maps* ordenados

Require: A es un *piecewise map* ordenado, M es un mapa y $Hint$ es un natural y una posición sugerida
Ensure: Inserta la entrada correspondiente a M en posición ordenada a partir del $Hint$ en A
1: **function** EMPLACEHINT(A, M, $hint$)
2: $end := \text{SIZE}(A)$
3: $it := Hint$
4: $mpe := \text{createMapEntry}(M)$
5: **while** $it \neq end$ **do**
6: **if** $\text{minPer}(A_{it}) < \text{minPer}(mpe)$ **then**
7: $it{+}{+}$
8: **else**
9: **break**
10: **end if**
11: **end while**
12: $A := A_{0:it-1} \frown\!\!\ll mdi \gg\!\frown A_{it:end-1}$
13: **end function**

5.4.8. Avance de pista - `advanceHint`

Esta función ajusta el valor de una pista (*hint*), es un número positivo, avanzando posiciones después de esta, mientras los mapas de un *piecewise map* ordenado presenten un mínimo perimetral menor que un valor de referencia dado. La lógica es similar a la utilizada en la operación anterior, aunque con algunas diferencias clave que pueden observarse en el pseudocódigo del Algoritmo 44.

Algorithm 44 Avance de pista para *piecewise maps* ordenados

Require: A es un *piecewise map* ordenado, $Crit$ es un mínimo perimetral y $Hint$ es un natural y una posición sugerida
Ensure: Avanza el $Hint$ mientras los mínimos perimetrales sean menores que $Crit$
1: **function** ADVANCEHINT(A, $Crit$, $Hint$)
2: $end := \text{SIZE}(A)$
3: **while** $Hint \neq end$ **do**
4: **if** $\text{minPer}(A_{Hint}) < Crit$ **then**
5: $Hint{+}{+}$
6: **else**
7: **break**
8: **end if**
9: **end while**
10: **end function**

5.5. Implementaciones

Llegado este punto, es momento de abordar las implementaciones concretas de las distintas operaciones sobre *piecewise maps* ordenados.

Este capítulo se organiza en cuatro secciones principales: una dedicada a aquellas operaciones cuya implementación no requirió modificaciones, una que aborda las operaciones adaptadas para poder funcionar correctamente en el contexto de *piecewise maps* ordenados, y finalmente otra correspondiente a las operaciones que fueron optimizadas.

Cabe destacar que, debido al cambio estructural de la variable `pieces_` en los *piecewise maps* ordenados con respecto a su contraparte desordenada, todas las operaciones debieron ser adaptadas para poder funcionar correctamente con esta nueva estructura interna.

Sin embargo, esta adaptación estructural no será considerada como una modificación en sí misma dentro del análisis, ya que responde únicamente a un requisito de compatibilidad con la representación de datos, y no a un cambio en la lógica o el comportamiento de las operaciones.

5.5.1. Operaciones sin cambios

Al implementar los *piecewise maps* ordenados sobre la base de la versión desordenada, se observó que ciertas operaciones no podían beneficiarse del orden para su optimización, pero tampoco alteraban dicho orden en la salida. Esto se debe, principalmente, a dos razones: o bien son operaciones que no devuelven un *piecewise map*, o bien el *piecewise map* resultante ya se encuentra ordenado.

Dentro del primer conjunto de operaciones se incluyen, por ejemplo: `dom`, que devuelve el dominio total de un *piecewise map*; `arity`, que informa la aridad del mismo; `image`, que retorna su imagen; entre otras.

Dado que estas operaciones son numerosas y su comportamiento no resulta central para los objetivos de esta tesina, además de que sus implementaciones se mantuvieron sin modificaciones, no se detallarán aquí. Todas ellas pueden consultarse en el archivo *pw_map.cpp*, disponible en el repositorio de GitHub.

En el segundo conjunto se incluyen operaciones como `compact`, cuyo algoritmo no elimina el orden del *piecewise map* resultante, o `mapInf`, cuya lógica interna se basa exclusivamente en operaciones que preservan dicho orden.

Al igual que en el caso anterior, dado que este conjunto tampoco es reducido y que las implementaciones no presentan particularidades relevantes para los objetivos de este escrito, se omite un desarrollo mas detallado aquí. Si se desea consultarlas en profundidad puede dirigirse al archivo correspondiente disponible en el repositorio.

5.5.2. Operaciones adaptadas al orden

Este conjunto de operaciones puede dividirse, a su vez, en dos subgrupos: por un lado, aquellas que debieron ser forzosamente sometidas a un algoritmo de ordenamiento, y por otro, aquellas que fueron adaptadas específicamente para preservar el orden a medida que se calcula el resultado.

Dentro del primer grupo se encuentra, por ejemplo, la operación `inverse`, la cual trabaja sobre el inverso de los mapas. Esta particularidad dificulta considerablemente la preservación del orden durante la construcción del resultado, ya que ordenar mapas invertidos resulta complejo si no se realiza de forma estrictamente lineal, tal como se discutió en el capítulo dedicado a las optimizaciones para los *piecewise maps* ordenados. Por este motivo, la operación `inverse` mantiene los mapas sin ordenar durante su ejecución y, únicamente antes de devolver el resultado, aplica un algoritmo de ordenamiento empleando la operación `mapEntryComp` como criterio de comparación.

Otra operación en este grupo es la operación `reduce`, en su versión sin argumentos. Como se analizó en el capítulo de conceptos preliminares, esta operación tiene un comportamiento particular. Internamente, invoca a una versión intermedia que recibe un mapa como argumento, la cual a su vez llama a una tercera implementación que recibe un intervalo y una expresión lineal. La cuestión radica en que esta última sí devuelve un *piecewise map* ordenado, pero la versión intermedia no, lo que provoca que al llegar a la versión sin argumentos, los

resultados intermedios no estén ordenados.

En lugar de diseñar un criterio de ordenamiento específico para la versión de `reduce` que recibe mapas, o de realizar un ordenamiento estrictamente lineal, y además de incorporar la operación `concatenation`, se optó por una alternativa más simple y eficiente: aplicar directamente un algoritmo de ordenamiento sobre el resultado final, utilizando nuevamente `mapEntryComp` como criterio de comparación.

En cuanto al algoritmo de ordenamiento utilizado se aprovechó la operación `sort` disponible en la librería estándar.

Dentro del segundo grupo se encuentra la operación `emplaceBack`, que en esta nueva versión requiere realizar verificaciones adicionales. En particular, si el nuevo mapa no debe insertarse estrictamente al final del *piecewise map*, la operación recurre a un procedimiento de búsqueda lineal para encontrar su posición óptima mediante el uso de `emplaceHint`.

Para profundizar en estos dos grupos de operaciones, puede consultarse el archivo *pw_map.cpp*, disponible en el repositorio. Allí se presentan las adaptaciones de las distintas operaciones en función de sus homólogas para *piecewise maps* desordenados.

5.5.3. Operaciones optimizadas

En esta sub-sección se presentarán todas aquellas operaciones que sí pudieron ser optimizadas mediante el aprovechamiento del orden en los *piecewise maps*.

5.5.3.1. Operaciones estructuralmente similares

Como se analizó en la Sub-sección 4.3.1, varias de estas operaciones comparten una estructura común, dividida en dos fases principales: una fase de comparación y un núcleo de operación. Para ejemplificar esta organización estructural se presentó, en su momento, un pseudocódigo bastante abstracto que capturaba dicho patrón estructural.

Basándose en ese esquema general, se desarrolló una operación concreta que implementa esta estructura de forma reutilizable. Dicha implementación puede observarse en el Algoritmo 45 y 46, donde se incorporan todos los criterios de optimización previamente definidos y válidos para las diferentes operaciones vistas y analizadas, de manera análoga a como se hizo en la operación de intersección entre conjuntos ordenados. De esta manera cada operación puede utilizar esta operación de esqueleto, haciendo los cambios correspondientes a través de sus argumentos.

Cabe destacar además tres aspectos complementarios. En primer lugar, se incorporó un argumento adicional, *order_mts*, a la operación general, el cual permite evitar la inversión del orden de los *piecewise maps* argumentos debido al criterio de selección. Esta funcionalidad es necesaria para aquellas operaciones, como `-` y `minAdjMap`, que no admiten el criterio de selección y requieren mantener el orden original de los *piecewise maps*.

En segundo lugar, se introdujo la variable `r_pos`, como en la intersección de conjuntos ordenados, la cual es utilizada específicamente en la suma *piecewise maps* ordenados.

Por último, los argumentos *Set_in*, *Set_out* y *Ord_map* están presentes debido a que las distintas operaciones utilizan, tanto internamente como en sus resultados, diferentes tipos de datos. Por ejemplo, la operación de suma (`+`) utiliza únicamente un *piecewise map* ordenado como resultado, mientras que `equalImage` devuelve un conjunto. La función de cada uno de estos argumentos es la siguiente:

- *Set_in*: conjunto utilizado internamente durante la operación, pero que no se devuelve como resultado.
- *Set_out*: conjunto que se genera como resultado de la operación.
- *Ord_map*: *piecewise map* ordenado que se genera como resultado de la operación.

En consecuencia, cada operación hará uso de los argumentos en función de lo que requieran, tanto en el núcleo de la operación, como en cuanto la salida que necesiten.

Algorithm 45 Procesado de *piecewise maps* ordenados — Parte 1: Preparación

Require: C y D son *piecewise maps* ordenados

```
1:  function processOrdMaps(C, D Set_in, Set_out, Ord_map, Process, Order_mts)
2:      B := C
3:      A := D
4:      if ¬Order_mts and size(D) < size(C) then
5:          B := D
6:          A := C
7:      end if

8:      indices := []                          ▷ Es una lista simplemente enlazada
9:      for i = 0; i < size(B); i := i + 1 do
10:         indices := indices ++ [i]
11:     end for

12:     r_pos := 0
13: end function
```

Algorithm 46 Procesado de *piecewise maps* ordenados — Parte 2: verificación

```
1:  function processOrdMaps(continuación)
2:      for all a ∈ A do
3:          map_a := map(a)
4:          p_a := setPer(a)
5:          i := 0

6:          while i ≠ length(indices) do

7:              b := B_indices[i]
8:              map_b := map(b)
9:              p_b := setPer(b)
10:             if maxPer(p_b)[0] < minPer(p_a)[0] then
11:                 indices := indices ◁ i
12:                 continue
13:             end if
14:             if maxPer(p_a)[0] < minPer(p_b)[0] then
15:                 break
16:             end if

17:             if doInt(p_b,p_a) then
18:                 Process(C, map_b, map_a, Set_in, Set_out, Ord_map, r_pos)
19:             end if

20:             i ++
21:         end while

22:         if indices == [] then
23:             break
24:         end if
25:     end for
26: end function
```

Ahora bien, tanto en el caso de la suma como en el de las demás operaciones que seguían la estructura mencionada anteriormente, se realizaron las siguientes modificaciones: por un lado, se adaptó cada operación para que utilice a `processMapsOrd` como esqueleto estructural, encargándose de preparar todos los argumentos necesarios para su ejecución. Por otro lado, se separó el núcleo de cada una de las operaciones en una nueva función, la cual puede ser pasada como argumento al parámetro *Process* de `processMapsOrd`. Para esto último, se definió un tipo específico para representar este tipo de funciones núcleo, denominado **ProcessFunc**, el cual puede encontrarse en el archivo *pw_map.hpp* de [2].

A continuación, se presentará cada una de ellas en detalle.

Suma - `+`

En el caso de la operación de suma, como se mencionó anteriormente, se adaptó para el uso de la operación esqueleto `processMapsOrd`, y se separo su núcleo de la operación en una función denominada `processAdd`. Ambas tienen sus pseudocódigos en los Algoritmos 47 y 48, respectivamente.

En particular, el **criterio de ordenamiento** definido para la suma se aplica íntegramente en la función que contiene el núcleo de la operación (ver Algoritmo 48), específicamente entre las líneas 9 y 10. Cabe destacar que este criterio no se ejecuta de la misma forma que en la operación de intersección entre conjuntos ordenados. Dado que todas las operaciones comparten el mismo esqueleto mediante la función `processMapsOrd`, incluir la operación `advanceHint` dentro de `processMapsOrd` añadiría un costo adicional innecesario a las demás operaciones, ya que su uso forma parte únicamente del criterio de ordenamiento de la suma. Por lo tanto, la operación `advanceHint` se utiliza directamente en la función que implementa el núcleo de la suma.

Algorithm 47 Suma de *piecewise maps* ordenados — Parte 1: Preparación

Require: A y B son *piecewise maps* ordenados
Ensure: Devuelve un nuevo *piecewise map* ordenado resultado de todas las sumas no vacías entre los mapas de A como los de B

```
1: function +(A, B)
2:     set_in := {}
3:     set_out := {}
4:     Ord_pwmap :=≪≫
5:     processOrdMaps(A, B, set_in, set_out, Ord_pwmap, processAdd, false)
6:     return Ord_pwmap
7: end function
```

Algorithm 48 Suma de *piecewise maps* ordenados — Parte 2: Procesamiento del núcleo de la suma

Require: M_1 y M_2 son dos mapas, Set_in y Set_out son conjuntos, Ord_pwmap es un *piecewise map* ordenado, O_pos es un valor positivo y C es un *piecewise map* ordenado que llama la operación

```
1: function processAdd(C, M_1, M_2, Set_in, Set_out, Ord_pwmap, O_pos)
2:     res_add := M_1 + M_2
3:     dom_add := dom(res_add)
4:     empty := isEmpty(dom_add)
5:     if ¬ isEmpty(empty) then
6:         dom_m_2 := dom(M_2)
7:         per := calculatePerimeter(dom_m_2)
8:         hintCrit := minPer(per)
9:         advanceHint(Ord_pwmap, hintCrit, O_pos)
10:        emplaceHint(Ord_pwmap, res_add, O_pos)
11:    end if
12:    return
13: end function
```

Igualdad de imágenes - `equalImage`

Para el caso de la igualdad de imágenes, o `equalImage`, esta fue modificada y, adicionalmente, se definió la función `processEqualImage`, que actúa como el núcleo de la operación. Ambas pueden verse en los Algoritmos 49 y 50, respectivamente.

Dado que esta operación no requiere un criterio de ordenamiento, su núcleo fue trasladado prácticamente

sin modificaciones a `processEqualImage`.

Algorithm 49 Igualdad de imagines de *piecewise maps* ordenados — Parte 1: Preparación

Require: A y B son dos *piecewise maps* ordenados
Ensure: Devuelve el conjunto `Set` donde A y B son iguales en sus imágenes
1: **function** EQUALIMAGE(A, B)
2: $set\_in := \{\}$
3: $Set\_out := \{\}$
4: $ord\_pwmap :=\ll\gg$
5: PROCESSORDMAPS($A, B, set\_in, Set\_out, ord\_pwmap$, processEqualImage, `false`)
6: **return** Set_out
7: **end function**

Algorithm 50 Igualdad de imagines de *piecewise maps* ordenados — Parte 2: Procesamiento del núcleo de la igualdad de imagenes

Require: M_1 y M_2 son dos mapas, Set_in y Set_out son conjuntos, Ord_pwmap es un *piecewise map* ordenado, O_pos es un valor positivo y C es un *piecewise map* ordenado que llama la operación
1: **function** PROCESSEQUALIMAGE(C, M_1, M_2, Set_in, Set_out, Ord_pwmap, O_pos)
2: $dom\_m\_1 :=$ DOM(M_1)
3: $dom\_m\_2 :=$ DOM(M_2)
4: $cap\_dom :=$ INTERSECTION(dom_m_1, dom_m_2)
5: $empty :=$ ISEMPTY(cap_dom)
6: **if** $\neg$ ISEMPTY($empty$) **then**
7: $m\_1\_cap := cap\_dom \mapsto$ EXP(M_1)
8: $m\_2\_cap := cap\_dom \mapsto$ EXP(M_2)
9: **if** $m\_1\_cap == m\_2\_cap$ **then**
10: $Set\_out :=$ DISJOINTCUP(Set_out, cap_dom)
11: **end if**
12: **end if**
13: **return**
14: **end function**

Resta - `-`

La resta para *piecewise maps* ordenados se modificó para funcionar utilizando la operación `processOrdMaps`, tal como se había planteado, cuyo pseudocódigo se presenta en el Algoritmo 51. El núcleo de la operación se implementó en la función `processMinus`, cuyo pseudocódigo se muestra en los Algoritmos 52, 53 y 54, debido a su gran extensión.

En el pseudocódigo del Algoritmo 51 se observa el uso de un algoritmo de ordenamiento preexistente, empleado para garantizar que el *piecewise map* resultante quede correctamente ordenado.

Mínimo adyacente - `minAdjMap`

Al igual que en los casos anteriores, la operación se descompone en dos partes: la operación *minAdjMap*, que emplea la operación esqueleto `processOrdMaps`, cuyo pseudocódigo se presenta en el Algoritmo 55; y la operación `processMinAdjMap`, que implementa el núcleo de la operación propiamente dicha, cuyo pseudocódigo puede observarse en el Algoritmo56.

Adicionalmente, en ambos operaciones se emplea un algoritmo de ordenamiento basado en la operación `compMapEntry`. En el caso de la operación presentada en el Algoritmo 55, dicho ordenamiento se aplica en la línea 8 para ordenar el resultado final. En el segundo caso, el ordenamiento se realiza antes de utilizar el resultado en las operaciones de *piecewise maps* ordenados, en la línea 29.

Igualdad - `==`

El caso de la operación de igualdad para *piecewise maps* ordenados resulta particularmente peculiar. Esta operación comparte una estructura muy similar con la intersección de conjuntos desordenados. Sin embargo, se diferencia en que el núcleo de la operación, ya que debe retornar el valor de verdad `false` en caso de que los operandos no sean iguales. Este requerimiento introduce una complicación adicional, ya que implicaría incluir chequeos en la operación esqueleto, `processMapsOrd` si se desea utilizar, lo cual no es deseable porque afectaría

Algorithm 51 Resta de *piecewise maps* ordenados — Parte 1: Preparación

Require: A y B son *piecewise maps* ordenados
Ensure: Devuelve un nuevo *piecewise maps* ordenados que contiene todas las restas no vacías de los mapas de A menos los de B

```
1:  function −(A, B)
2:      Ord_map := ≪≫
3:      emptyA := isEmpty(A)
4:      emptyB := isEmpty(B)
5:      if emptyA or emptyB then
6:          return Ord_map
7:      end if
8:      all := [0 : 1 : Inf]                                  ▷ Intervalo completo
9:      d := arity(A)
10:     set_in := {all^d}
11:     set_out := {}
12:     processOrdMaps(A, B, set_in, set_out, Ord_map, processMinus, true)
13:     sort(Ord_map, mapEntryComp)
14:     return Ord_map
15: end function
```

Algorithm 52 Resta de *piecewise maps* ordenados — Parte 2-1: Procesamiento del núcleo de la resta

Require: M_1 y M_2 son dos mapas, Set_in y Set_out son conjuntos, Ord_pwmap es un *piecewise map* ordenado, O_pos es un valor positivo y C es un *piecewise map* ordenado que llama la operación

```
1:  function processMinus(C, M_1, M_2, Set_in, Set_out, Ord_pwmap, O_pos)
2:      dom_1 := dom(M_1)
3:      dom_2 := dom(M_2)
4:      dom := intersection(dom_1, dom_2)
5:      if isEmpty(dom) then
6:          return
7:      end if
8:      minus_exp := exp(M_1) − exp(M_2)
9:      d := arity(Set_in)
10:     ith := ≪ Set_in ↦ [1 ∗ x + 0]^d ≫
11:     Parte 2-2...
12:     restriction := restrict(ith, dom)
13:     Ord_pwmap := Ord_pwmap ⌢ restriction
14:     return
15: end function
```

Algorithm 53 Resta de *piecewise maps* ordenados — Parte 2-2: Procesamiento del núcleo de la resta

Require: M_1 y M_2 son dos mapas, Set_in y Set_out son conjuntos, Ord_pwmap es un *piecewise map* ordenado, O_pos es un valor positivo

```
1: function processMinus
2:     for j := 0 to arity(M_1) − 1 do
3:         m := slope(minus_exp_j)
4:         h := offset(minus_exp_j)
5:         (begin_neg, end_neg, begin_pos, end_pos) := (0, Inf, 0, Inf)
6:         if m = 0 then
7:             if h < 0 then
8:                 (begin_pos, end_pos) := (1, 0)
9:             else
10:                (begin_neg, end_neg) := (1, 0)
11:            end if
12:        else if m > 0 then
13:            cross := −h/m
14:            if cross ≥ 0 then
15:                bp := toNat(cross)                ▷ toNat trunca el valor racional
16:                if bp > 0 then
17:                    end_neg := bp − 1
18:                else
19:                    (begin_neg, begin_neg) := (1, 0)
20:                end if
21:            end if
22:        else
23:            cross := −h/m
24:            if cross ≥ 0 then
25:                ep := toNat(cross)                ▷ toNat trunca el valor racional
26:                if ep > 0 then
27:                    begin_neg := ep + 1
28:                end if
29:            else
30:                (begin_pos, end_pos) := (1, 0)
31:            end if
32:        end if
33:        Parte 2-3...
34:    end for
35: end function
```

Algorithm 54 Resta de *piecewise maps* ordenados — Parte 2-3: Procesamiento del núcleo de la resta y guardado de resultados parciales

Require: M_1 y M_2 son dos mapas, Set_in y Set_out son conjuntos, Ord_pwmap es un *piecewise map* ordenado, O_pos es un valor positivo

```
1: function processMinus
2:     jth :=≪≫
3:     neg := [begin_neg : 1 : end_neg]
4:     pos := [begin_pos : 1 : end_pos]
5:     for all mpe ∈ ith do
6:         m := map(mpe)
7:         mdi := dom(m)[0]
8:         e := exp(m)
9:         if ¬isEmpty(neg) then
10:            mdi_j := neg
11:            e_j := 0
12:            new_map := mdi ↦ e
13:            pushBack(jth, new_map)
14:        end if
15:        if ¬isEmpty(pos) then
16:            mdi_j := pos
17:            e_j := minus_exp_j
18:            new_map := mdi ↦ e
19:            pushBack(jth, new_map)
20:        end if
21:    end for
22:    ith := jth
23: end function
```

Algorithm 55 Mínimo adyacente — Parte 1: Preparación

Require: A y B son *piecewise maps* ordenados
Ensure: Devuelve un nuevo *piecewise maps* ordenado

```
1: function minAdjMap(A, B)
2:     R :=≪≫
3:     set_in := {}
4:     set_out := {}
5:     processOrdMaps(A, B, set_in, set_out, R, processMinAdjMap, true)
6:     sort(R, mapEntryComp)
7:     return R
8: end function
```

Algorithm 56 Mapa mínimo adyacente — Parte 2: Procesamiento del núcleo del mínimo adyacente

Require: M_1 y M_2 son dos mapas, Set_in y Set_out son conjuntos, Ord_pwmap es un *piecewise map* ordenado, O_pos es un valor positivo y C es un *piecewise map* ordenado que llama la operación

```
1: function processMinAdjMap(C, M_1, M_2, Set_in, Set_out, Ord_pwmap, O_pos)
2:     dom_1 := dom(M_1)
3:     dom_2 := dom(M_2)
4:     dom_int := intersection(dom_1, dom_2)
5:     if isEmpty(dom_int) then
6:         return ≪≫
7:     end if
8:     dom_res := image(M_1, dom_int)
9:     exp_1 := exp(M_1)
10:    im_2 := image(M_2, dom_int)
11:    if ¬isConstant(exp_1) then
12:        exp_2 := exp(m_2)
13:        exp_1_i := inverse(exp_1)
14:        e_res := composition(exp_2, exp_1_i)
15:    else
16:        min_elem := minElem(im_2)
17:        d := arity(im_2)
18:        e_res := [0 * x + min_elem[0], ..., 0 * x + min_elem[d]]
19:    end if
20:    empty_res := isEmpty(dom_res)
21:    if ¬isEmpty(dom_res) then
22:        ith := dom_res ↦ e_res
23:        ith_pw :=≪ ith ≫
24:        again := intersection(dom_res, set_in)
25:        empty_again := isEmpty(again)
26:        if ¬isEmpty(again) then
27:            ord_pwmap_aux := Ord_pwmap
28:            sort(ord_pwmap_aux, mapEntryComp)
29:            aux_res := restrict(ord_pwmap_aux, dom_res)
30:            min_map := minMap(aux_res, ith_pw)
31:            comb := combine(min_map, ith_pw)
32:            new_res := combine(comb, ord_pwmap_aux)
33:            new_res := staticCast(new_resPtr)
34:            Ord_pwmap := ord_pwmap_aux
35:            dom_i := dom(ith_pw)
36:            set_in := cup(set_in, dom_i)
37:        else
38:            pushBack(ord_pwmap, ith)
39:            set_in := disjointCup(set_in, dom_res)
40:        end if
41:    end if
42: end function
```

a todas las demás operaciones que reutilizan dicha estructura.

Por esta razón, y con el objetivo de preservar una reducción tangible en los tiempos de ejecución, se optó por realizar una copia casi directa de la operación `processMapsOrd` dentro de la operación de igualdad, sin separar su núcleo de la operación como se hizo en los casos anteriores. Y como se puede ver en el pseudocódigo expuesto en los Algoritmos 57 y 58, la operación quedo prácticamente igual a la intersección de conjuntos ordenados, salvo por el núcleo de la operación y el hecho de no tener criterio de ordenamiento.

Algorithm 57 Igualdad de *piecewise maps* ordenados — Parte 1: Preparación

Require: C y D son *piecewise maps* ordenados
Ensure: Devuelve `true` sin los *piecewise maps* son iguales. En caso contrario, devuelve `false`

```
1:  function ==(C, D)
2:      R := ≪≫_⟨f⟩
3:      if DOM(C) ≠ DOM(D) then
4:          return false
5:      end if
6:      if C == D then
7:          return true
8:      end if

9:      B := C
10:     A := D
11:     if SIZE(D) < SIZE(C) then
12:         B := D
13:         A := C
14:     end if

15:     indices := []                          ▷ Es una lista simplemente enlazada
16:     for i = 0; i < SIZE(B); i := i + 1 do
17:         indices := indices ++ [i]
18:     end for

19:     Parte 2...
20: end function
```

5.5.3.2. Restricción de dominio - `restrict`

La operación `restrict`, como se analizó en la Sub-sección 5.3 para esta operación, cuenta con un **criterio de parada**, un **criterio de solapamiento** y un **criterio de ordenamiento**.

En el pseudocódigo del Algoritmo 59 puede verse cómo quedó la operación tras aplicar todos los criterios mencionados:

- El **criterio de parada** se encuentra implementado entre las líneas 22 y 23.
- El **criterio de solapamiento** aparece en la línea 14, gracias al uso de la operación adicional `doInt`.
- Finalmente, el **criterio de ordenamiento** se implementa mediante las operaciones auxiliares `emplaceHint` y `advanceHint`, ubicadas en las líneas 19 y 20 respectivamente, junto con el uso de la variable *r_pos* declarada en la línea 7.

5.5.3.3. Combinación - `combine`

Como se analizó previamente, la operación `combine` cuenta únicamente con un criterio de optimización, además de su correspondiente criterio de ordenamiento. En el pseudocódigo Algoritmo 60 puede observarse el código final de la operación, donde:

Algorithm 58 Igualdad de *piecewise maps* ordenados — Parte 2: verificación

```
1: function ==(CONTINUACIÓN)
2:     for all a ∈ A do
3:         map_a := map(a)
4:         p_a := setPer(a)
5:         i := 0

6:         while i ≠ LENGTH(indices) do

7:             b := B_{indices[i]}
8:             map_b := map(b)
9:             p_b := setPer(b)
10:            if maxPer(p_b)[0] < minPer(p_a)[0] then
11:                indices := indices ◁ i
12:                continue
13:            end if
14:            if maxPer(p_a)[0] < minPer(p_b)[0] then
15:                break
16:            end if

17:            if DOINT(p_b,p_a) then
18:                dom_a := DOM(map_a)
19:                dom_b := DOM(map_b)
20:                cap_dom := DOM(dom_a, dom_b)
21:                if ¬ISEMPTY(cap_dom) then
22:                    if CARDINAL(cap_dom) == 1 then
23:                        m_1 := cap_dom ↦ EXP(map_a)
24:                        m_2 := cap_dom ↦ EXP(map_b)
25:                        if IMAGE(m_1) ≠ IMAGE(m_2) then
26:                            return false
27:                        end if
28:                    else
29:                        if EXP(map_a) ≠ EXP(map_b) then
30:                            return false
31:                        end if
32:                    end if
33:                end if
34:            end if

35:            i ++
36:        end while

37:        if indices == [] then
38:            break
39:        end if
40:    end for

41:    return true
42: end function
```

Algorithm 59 Restricción de dominio para *piecewise maps* ordenados

Require: A es un *piecewise map* ordenado y S es un conjunto
Ensure: Devuelve un nuevo *piecewise map* cuyos mapas están restringidos en su domino por S

```
 1: function RESTRICT(A, S)
 2:     R := ≪≫
 3:     if ISEMPTY(S) then
 4:         return R
 5:     end if
 6:     r_pos := 0
 7:     s_sp := CALCULATEPERIMETER(S)
 8:     s_max_per := maxPer(s_sp)
 9:     for all a ∈ A do
10:         a_map := map(a)
11:         a_sp := setPer(a)
12:         a_min_per := minPer(a)
13:         if DOINT(a_sp, s_sp) then
14:             rest_map := RESTRICT(a_map, S)
15:             if ¬ ISEMPTY(rest_map) then
16:                 ADVANCEHINT(R, a_min_per, r_pos)
17:                 EMPLACEHINT(R, rest_map, r_pos)
18:             end if
19:             continue
20:         end if
21:         if s_max_per[0] < a_min_per[0] then
22:             break
23:         end if
24:     end for
25:     return R
26: end function
```

- El **criterio de solapamiento** se aplica, como en otros casos, mediante la operación adicional `doInt`, ubicada en la línea 19.
- El **criterio de ordenamiento** se implementa mediante el uso de las funciones auxiliares `emplaceHint` y `advanceHint`, en las líneas 27 y 28 respectivamente, junto con la utilización de la variable *r_pos*.

5.5.3.4. Concatenación - `concatenation`

Esta operación, como ya se analizó, es básicamente análoga a la operación de unión disjunta de conjuntos ordenados. Por este motivo, su implementación resulta sumamente similar, como puede apreciarse en el pseudocódigo del Algoritmo 61.

5.5.3.5. Composición - `composition`

La operación de composición de *piecewise maps* ordenados, luego de aplicarle todos los criterios de optimización y de ordenamiento que se plantearon para ella, quedó como se puede ver en el pseudocódigo del Algoritmo 62. En este se observa que los criterios quedaron dispuestos de la siguiente manera:

- El **criterio de parada** se encuentra implementado entre las líneas 22 y 23.
- El **criterio de solapamiento** aparece en la línea 15, gracias al uso de la operación adicional `doInt`.
- Y, por ultimo, el **criterio de ordenamiento** se implementa mediante las operaciones auxiliares `emplaceHint` y `advanceHint`, ubicadas en las líneas 18 y 10 respectivamente, junto con el uso de la variable *r_pos* declarada en la línea 4.

Algorithm 60 Combinación para *piecewise maps* ordenados

Require: A y O son *piecewise maps* ordenados
Ensure: Devuelve un nuevo *piecewise map* ordenado que resulta de añadir a A los mapas de O restringidos en su dominio a valores no presentes en el dominio de A

```
1: function COMBINE(A, O)
2:     R := A
3:     if ISEMPTY(A) then
4:         return O
5:     end if
6:     if ISEMPTY(O) then
7:         return A
8:     end if
9:     r_pos := 0
10:    dom_a := DOM(A)
11:    a_sp := CALCULATEPERIMETER(dom_a)
12:    for all o ∈ O do
13:        o_map := map(o)
14:        o_sp := setPer(o)
15:        dom_o_map := DOM(o_map)
16:        exp_o_map := EXP(o_map)
17:        res_comb := dom_o_map ↦ exp_o_map
18:        if DOINT(o_sp, a_sp) then
19:            new_dom := DIFFERENCE(dom_o_map, dom_a)
20:            if ISEMPTY(new_dom) then
21:                continue
22:            end if
23:            res_comb := new_dom ↦ exp_o_map
24:        end if
25:        hint_crit := minPer(o_sp)
26:        ADVANCEHINT(R, hint_crit, r_pos)
27:        EMPLACEHINT(R, res_comb, r_pos)
28:    end for
29:    return R
30: end function
```

Algorithm 61 Concatenación para *piecewise maps* ordenados

Require: A y B son *piecewise maps* ordenados disjuntos
Ensure: Devuelve un nuevo *piecewise map* ordenado con los mapas de A y de B

```
1:  function CONCATENATION(A, B)
2:      R := ≪≫
3:      if ISEMPTY(A) then
4:          return B
5:      end if
6:      if ISEMPTY(B) then
7:          return A
8:      end if

9:      end_a := SIZE(A)
10:     last_a := minPer(A_{end_a−1})
11:     first_b := minPer(B_0)
12:     if last_a < first_b then
13:         return A ⌢ B
14:     end if
15:     end_b := SIZE(B)
16:     last_b := minPer(B_{end_b−1})
17:     first_a := minPer(A_0)
18:     if last_b < first_a then
19:         return B ⌢ A
20:     end if

21:     i_a := 0
22:     i_b := 0
23:     for ; i_a ≠ end_a and i_b ≠ end_b; do

24:         min_a := minPer(A_{i_a})
25:         min_b := minPer(B_{i_b})

26:         if min_a < min_b then
27:             pushBack(R, A_{i_a})
28:             i_a ++
29:         else
30:             pushBack(R, B_{i_b})
31:             i_b ++
32:         end if
33:     end for

34:     R := R ⌢ A_{i_a:end_a−1}
35:     R := R ⌢ B_{i_b:end_b−1}

36:     return R
37: end function
```

Algorithm 62 Composición para *piecewise maps* ordenados

Require: A y B son dos *piecewise maps* ordenados
Ensure: Devuelve un nuevo *piecewise map* cuyos mapas son las composiciones no vacías de los mapas de A con B

```
1: function COMPOSITION(A, B)
2:     R :=≪≫
3:     r_pos := 0
4:     for all b ∈ b do
5:         b_map := map(b)
6:         img := image(b_map)
7:         i_sp := CALCULATEPERIMETER(img)
8:         i_max_per := maxPer(i_sp)
9:         ADVANCEHINT(R, minPer(b), r_ros)
10:        for all a ∈ A do
11:            a_map := map(a)
12:            a_sp := setPer(a)
13:            a_min_per := minPer(a_p)
14:            if DOINT(a_sp, i_sp) then
15:                comp_map := COMPOSITION(a_map, b_map)
16:                if ¬ ISEMPTY(comp_map) then
17:                    EMPLACEHINT(R, comp_map, r_pos)
18:                end if
19:                continue
20:            end if
21:            if i_max_per[0] < a_min_per[0] then
22:                break
23:            end if
24:        end for
25:    end for
26:    return R
27: end function
```

5.5.3.6. Pseudoinversa - `firstInv`

Ahora es el turno de la operación `firstInv`, la cual presenta la particularidad de disponer de dos **criterios de solapamiento** debido a la forma en que actúa internamente. Estos, junto con el **criterio de parada**, pueden apreciarse con claridad en el pseudocódigo del Algoritmo 63. En particular, los criterios se distribuyen de la siguiente manera:

- El **criterio de parada** se encuentra implementado entre las líneas 40 y 42.
- El **criterio de solapamiento** aparece en la línea 14, gracias al uso de la operación adicional `doInt` aplicada a el perímetro de a y el de S.
- Por otro lado, el**criterio de solapamiento para** *visisted* aparece en la línea 20, gracias al uso de la operación adicional `doInt` aplicada a el perímetro de *img* y el de *visited*.

Nuevamente en este caso, se hace uso de un algoritmo de ordenamiento para poder acomodar correctamente la salida de la operación.

5.5.3.7. Desplazamiento de dominio - `offsetDom`

Finalmente, se concluye con la operación `offsetDom`, la cual, como se mencionó anteriormente, dispone de un **criterio de parada** y un **criterio de solapamiento**. Al igual que en el caso anterior, se emplea un algoritmo de ordenamiento para garantizar que la salida de la operación quede correctamente ordenada.

El pseudocódigo de esta operación se presenta en el Algoritmo 64, donde puede observarse que:

- El **criterio de parada** se aplica entre las líneas 19 y 21.
- El **criterio de solapamiento** aparece en la línea 10, a través de la operación auxiliar `doInt`, aplicada sobre el perímetro de a y el de O.

‘‘‘

Algorithm 63 Pseudoinversa para *piecewise maps* ordenandos

Require: A es un *piecewise map* ordenando y S es un conjunto
Ensure: Devuelve las pseudoinversas de los mapas de A restringidas en su dominio para que sean disjuntas dos a dos

```
1: function FIRSTINV(A, S)
2:     R := ≪≫
3:     if ISEMPTY(A) or ISEMPTY(S) then
4:         return R
5:     end if
6:     s_sp := calculatePerimeter(S)
7:     s_max := maxPer(s_sp)
8:     visited := {}
9:     for all a ∈ A do
10:        a_sp := setPer(a)
11:        a_m := MAP(a)
12:        m_min := minPer(a_sp)
13:        if doInt(a_sp, s_sp) then
14:            img := IMAGE(a_m, S)
15:            if ¬ISEMPTY(img) then
16:                if ¬ISEMPTY(visited) then
17:                    i_sp := calculatePerimeter(img)
18:                    v_sp := calculatePerimeter(visited)
19:                    if doInt(i_sp, v_sp) then
20:                        diff_dom := DIFFERENCE(img, visited)
21:                        if ISEMPTY(diff_dom) then
22:                            continue
23:                        else
24:                            img := diff_dom
25:                        end if
26:                    else
27:                    end if
28:                else
29:                end if
30:                pre := PREIMAGE(a_m, img)
31:                exp_val := EXP(a_m)
32:                new_map := pre ↦ exp_val
33:                inv := MININV(new_map)
34:                pushBack(R, inv)
35:                visited := DISJOINTCUP(visited, img)
36:            end if
37:            continue
38:        end if
39:        if s_max[0] < m_min[0] then
40:            break
41:        end if
42:    end for
43:    sort(R, mapEntryComp)
44:    return R
45: end function
```

Algorithm 64 Desplazamiento de dominio para *piecewise maps* ordenados

Require: A es un *piecewise map* ordenado, O es un *piecewise map* ordenado que actuara de offset
Ensure: Devuelve un *piecewise map* ordenado cada mapa de A desplazado según la imagen de O

```
1: function OFFSETDOM(A, O)
2:     R := ≪≫
3:     o_sp := calculatePerimeter(DOM(O))
4:     o_dom_max := maxPer(o_sp)
5:     for all a ∈ A do
6:         a_m := map(a)
7:         a_sp := setPera
8:         m_min := minPera_sp
9:         if doInt(a_sp, o_sp) then
10:            a_m_dom := DOM(a_m)
11:            new_dom := IMAGE(O, a_m_dom)
12:            exp_val := EXP(a_m)
13:            if ¬ ISEMPTY(new_dom) then
14:                new_map := new_dom ↦ exp_val
15:                pushBack(R, new_map)
16:            end if
17:        end if
18:        if o_dom_max[0] < m_min[0] then
19:            break
20:        end if
21:    end for
22:    sort(R, mapEntryComp)
23:    return R
24: end function
```

Capítulo 6

Resultados

Para la realización de las pruebas de rendimiento correspondientes a las implementaciones desarrolladas, tanto para conjuntos ordenados como para *piecewise maps*, se utilizó un equipo con las siguientes especificaciones técnicas:

- **Procesador:** AMD Ryzen™ 7 4700U.
- **Memoria RAM:** 8 GB DDR4 a 3200 MHz.
- **Sistema operativo:** Windows 10.

Todas las pruebas fueron ejecutadas en una máquina virtual configurada sobre el software *Oracle VM VirtualBox*. Esta máquina virtual se utilizó de manera constante con el equipo conectado a la corriente eléctrica, garantizando así un entorno estable para la medición del rendimiento. La configuración de la máquina virtual fue la siguiente:

- **Procesador virtual:** 4 núcleos del AMD Ryzen™ 7 4700U.
- **Memoria RAM asignada:** 2.5 GB DDR4 a 3200 MHz.
- **Sistema operativo:** Ubuntu 24.04.1 LTS.

6.1. Casos de prueba sintéticos para conjuntos ordenados

Con el propósito de evaluar el rendimiento y validar los avances obtenidos con respecto a las demás implementaciones de conjuntos desarrolladas, se diseñó y ejecutó una amplia batería de casos de prueba sintéticos para las operaciones principales de conjuntos. Estos fueron específicamente construidos para medir el tiempo de ejecución de las operaciones fundamentales de conjuntos, expresado en milisegundos. El código correspondiente se encuentra disponible en el archivo `set_perf.cpp`, ubicado en la carpeta *test/performance*, donde se pueden consultar más detalles sobre su constitución y estructura [2].

A continuación, se presentan los resultados obtenidos, junto con una comparación detallada entre las distintas implementaciones: conjuntos ordenados, ordenados densos y desordenados; con el propósito de analizar el rendimiento de la versión ordenada frente a las alternativas disponibles. Los valores que se presentan en las gráficas corresponden al **mínimo**, **máximo** y **promedio** de los tiempos de ejecución obtenidos tras ejecutar múltiples veces cada caso de prueba sintético. Es importante destacar que todos los porcentajes y comparaciones mostrados se encuentran calculados sobre los valores **promedio**, *salvo en las pruebas de escalabilidad*, donde por cada cantidad de multi-intervalos se ejecutó una única vez el caso de prueba.

- ***Casos de prueba sintéticos unidimensionales y densos:***

  Se compararan las tres implementaciones disponibles utilizando conjuntos conteniendo 10.000 multi-intervalos unidimensionales densos.

- `intersection`: En la Figura 6.1 (a), se observa que los conjuntos ordenados presentan un rendimiento comparable al de los conjuntos ordenados densos, con un incremento del tiempo de ejecución relativo de aproximadamente **28.97 %**.

  Por su parte, la Figura 6.1 (b) revela una diferencia significativa entre los conjuntos ordenados y desordenados, destacándose una mejora de aproximadamente **34.18 veces** en rendimiento, lo que equivale a una reducción relativa del **97.08 %** en el tiempo de ejecución al utilizar conjuntos ordenados con respecto a los desordenados.

  La operación `intersection` en conjuntos ordenados presenta un crecimiento del costo proporcional a la cantidad de intersecciones que deben resolverse entre multi-intervalos. En particular, si el número de intersecciones crece de manera lineal, el tiempo de ejecución también lo hace. Este comportamiento, ilustrado en la Figura 6.2 y visible a través del Cuadro 6.1, cumple con uno de los objetivos fundamentales de esta tesina. En contraste, la implementación basada en conjuntos desordenados evidencia un crecimiento cuadrático en el tiempo de ejecución.

	Tiempo de ejecución en milisegundos	
Cantidad de multi-intervalos	**Conjuntos ordenados**	**Conjuntos desordenados**
2000	8	41
4000	18	134
8000	28	476
16000	36	1853
32000	62	7553
64000	137	32203

Cuadro 6.1: Cuadro de escalado del tiempo de ejecución de la operación `intersection`

- `complement`: Esta operación cuenta con una implementación específica para conjuntos ordenados. Tal como se muestra en la Figura 6.3, su rendimiento se sitúa entre el de las otras dos variantes, con una reducción relativa del tiempo de ejecución del **28.75 %** respecto a la versión desordenada.
- `disjointCup`: En el caso de la operación `disjointCup` se preveía un rendimiento inferior en la implementación para conjuntos ordenados debido al costo del ordenamiento. Pero, gracias al **criterio de ordenamiento** aplicado, se logró un rendimiento similar para conjuntos ordenados que para conjuntos ordenados densos. Teniendo un aumento relativo del tiempo de ejecución del **40.82 %** respecto a la versión desordenada (ver Figura 6.4). Eso se debe que la implementación desordenado no realiza procedimiento adicional mas allá de la inserción de los elementos.
- `cup`: Al estar implementada a partir de otras operaciones, la eficiencia de `cup` depende directamente de estas. La implementación sobre conjuntos ordenados muestra una mejora del **26.09 %** en el tiempo de ejecución respecto a la versión con conjuntos desordenados (ver Figura 6.5).
- `difference`: Similar a `cup`, esta operación se basa en funciones auxiliares. La versión con conjuntos ordenados reduce el tiempo de ejecución en un **19.05 %** en comparación con la implementación desordenada (ver Figura 6.6).

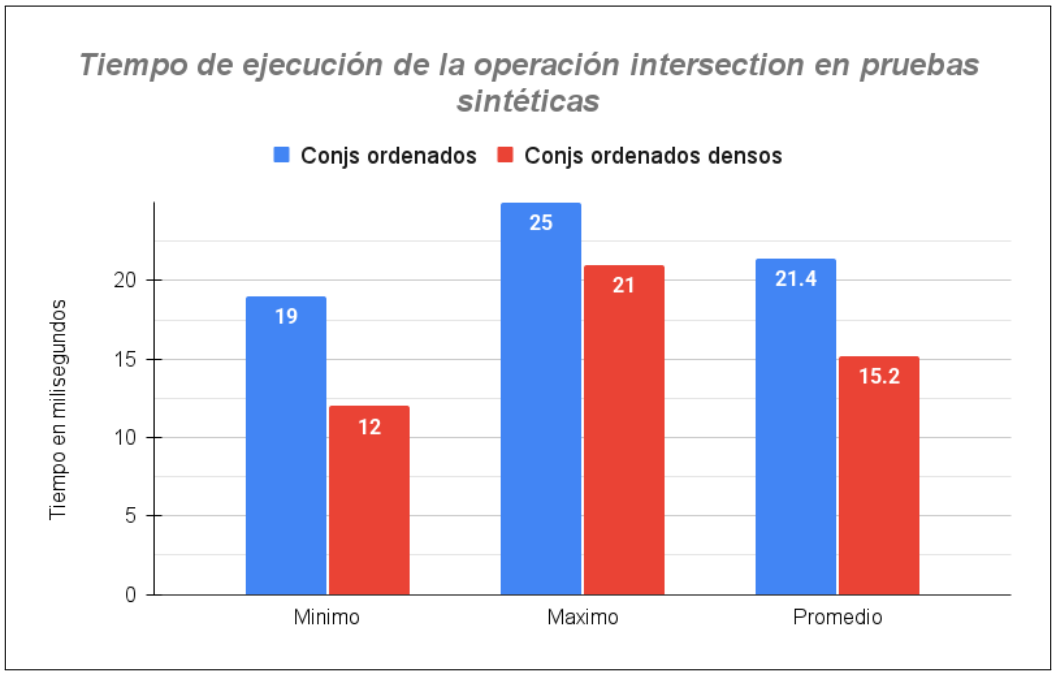
En promedio, las implementaciones para conjuntos ordenados alcanza una mejora palpable frente a la versión desordenada al trabajar con conjuntos unidimensionales densos, con excepción de la operación `intersection`, donde la mejora fue especialmente destacada y `disjointCup` donde la pérdida era esperable.

A continuación, se analiza el rendimiento sobre conjuntos con multi-intervalos tridimensionales y con paso variable, para evaluar el comportamiento en un contexto más general.

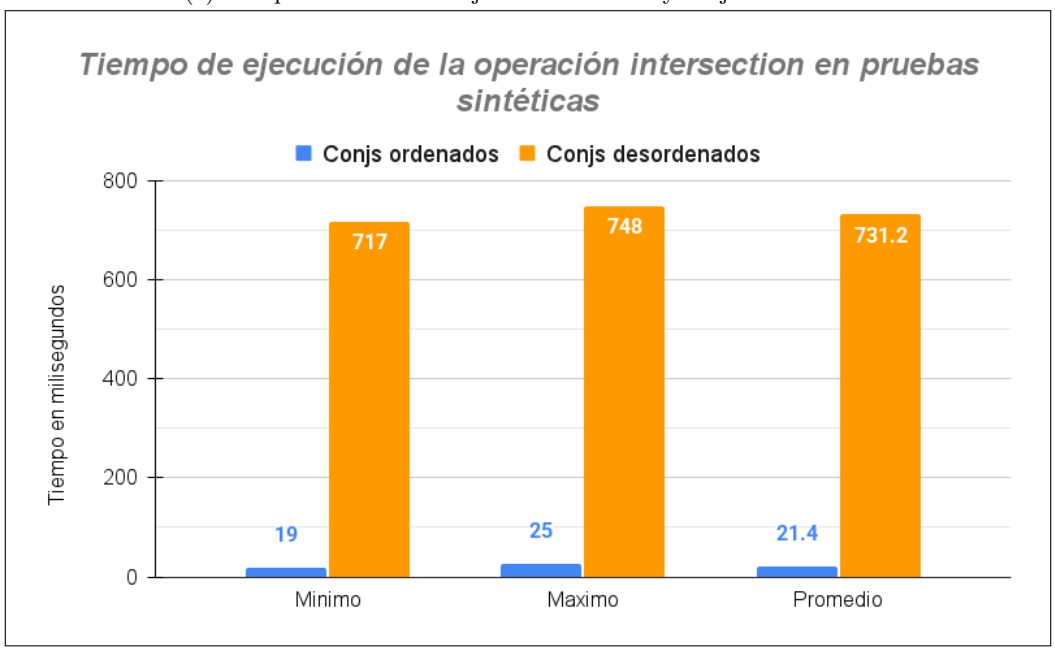
- ***Casos de prueba sintéticos tridimensionales y no densos:***

  En este escenario, se utilizaron conjuntos de 1.000 multi-intervalos con tres dimensiones y con paso dos en la primera dimensión, comparando en esta ocasión la implementación ordenada frente a la desordenada.

  - `intersection`: Aunque la diferencia de rendimiento no es tan marcada como en el caso unidimensional, debido a la cantidad de elementos de los conjuntos, la versión ordenada logra una reducción relativa del **58.82 %** en el tiempo de ejecución (Figura 6.7).



(a) Comparación entre conjuntos ordenados y conjuntos densos.



(b) Comparación entre conjuntos ordenados y conjuntos desordenados.

Figura 6.1: Comparación del tiempo de ejecución de la operación `intersection` entre las distintas implementaciones de conjuntos.

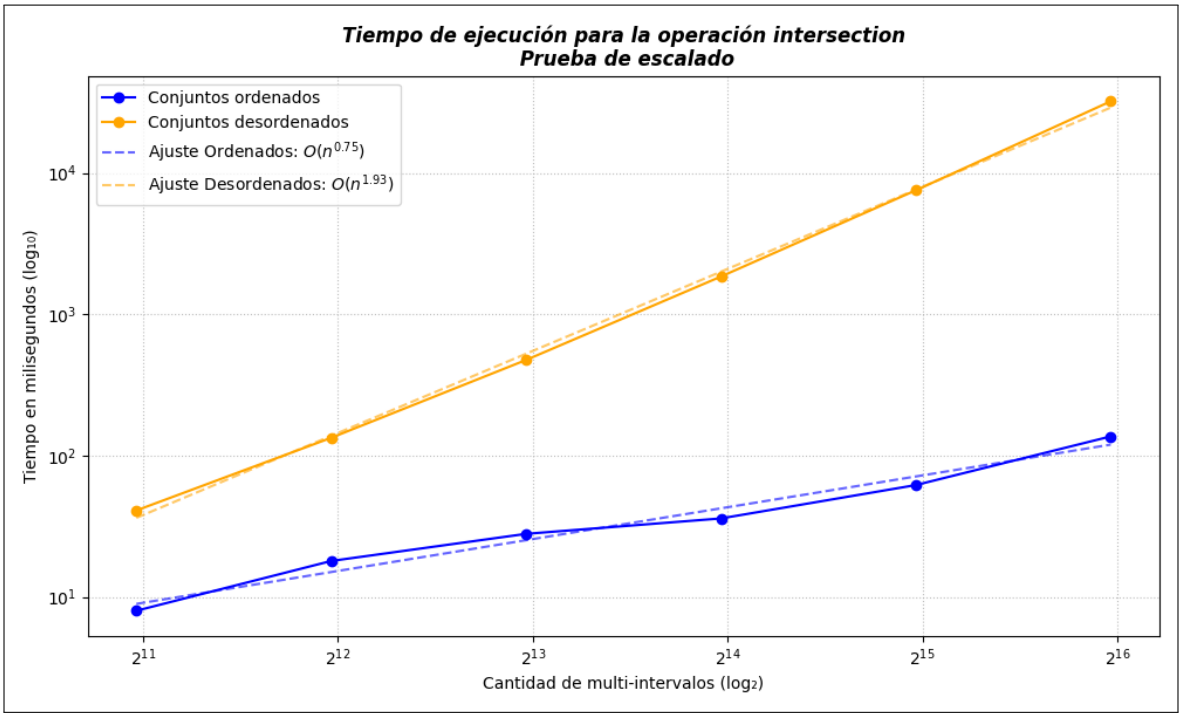


Figura 6.2: Gráfica del escalado en el tiempo de ejecución de la operación `intersection`.

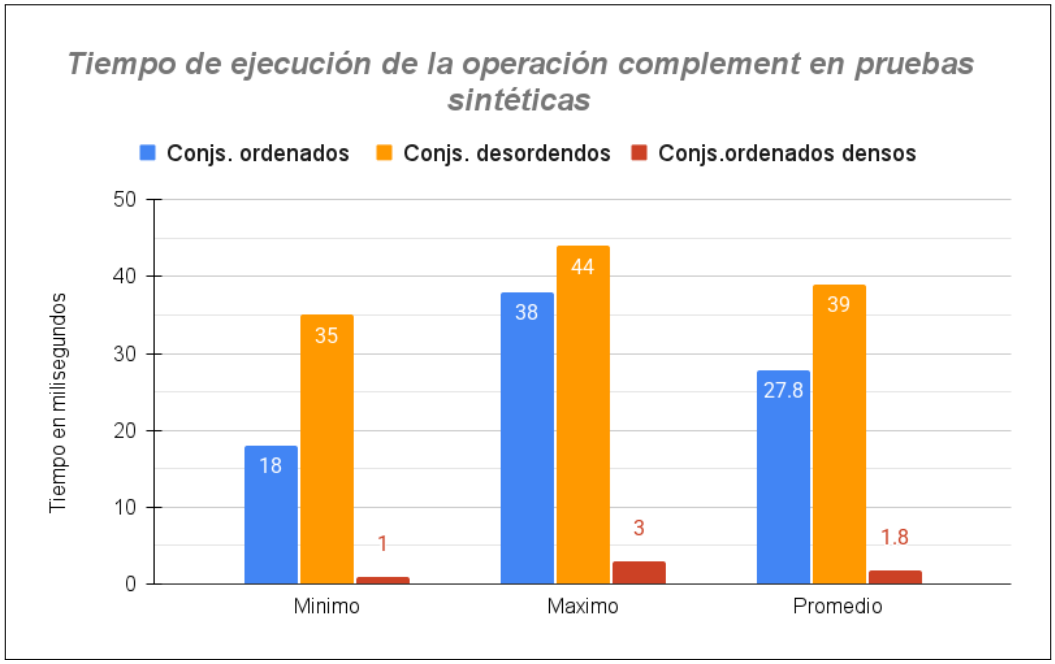


Figura 6.3: Comparación del tiempo de ejecución de la operación `complement`.

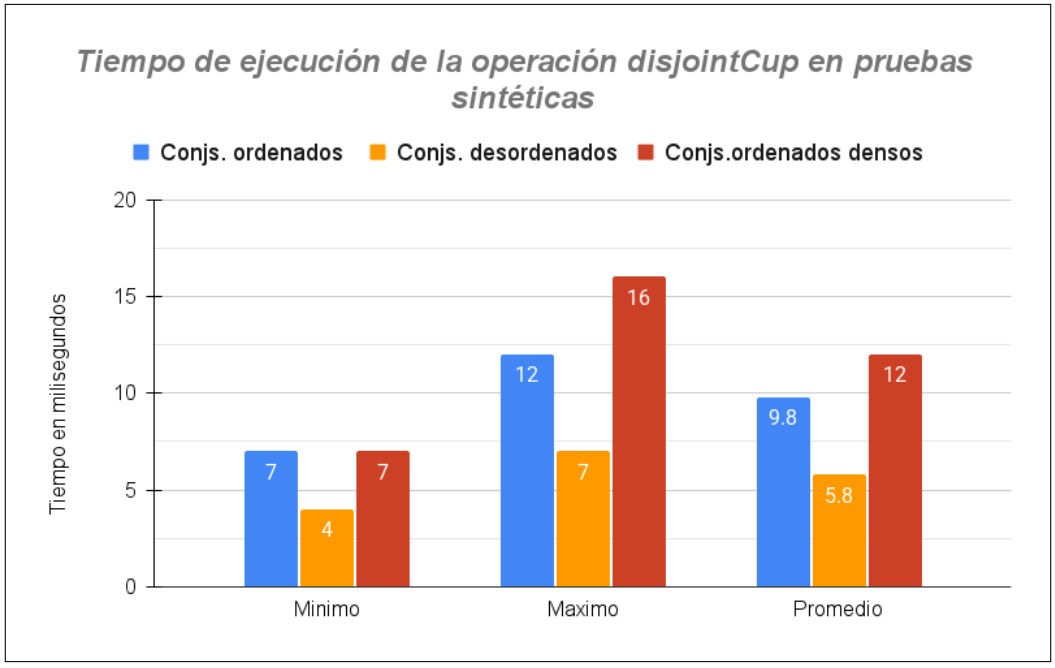


Figura 6.4: Comparación del tiempo de ejecución de la operación `disjointCup`.

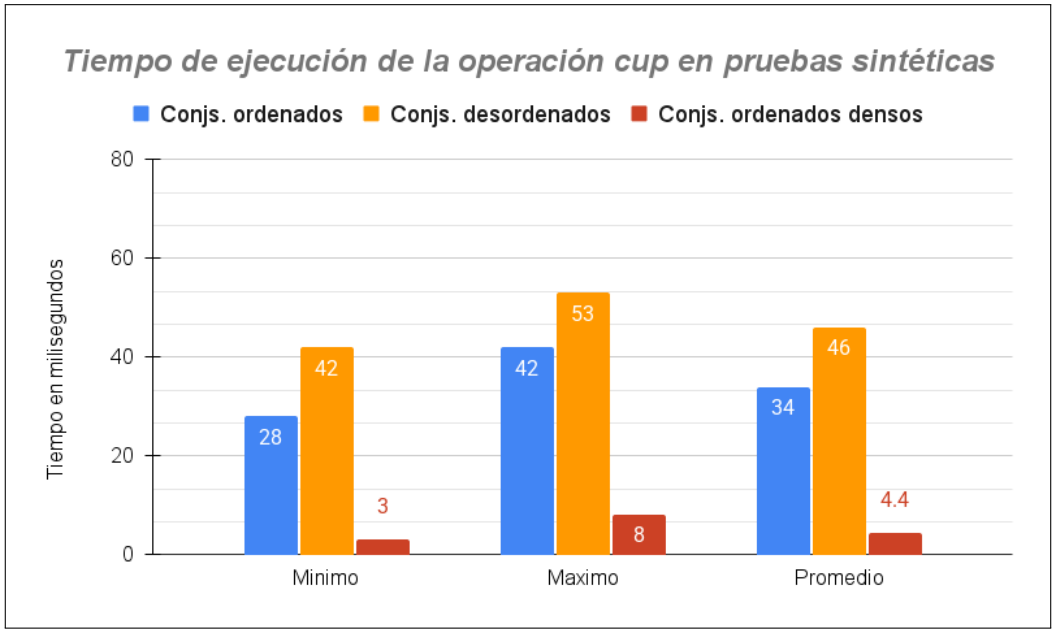


Figura 6.5: Comparación del tiempo de ejecución de la operación `cup`.

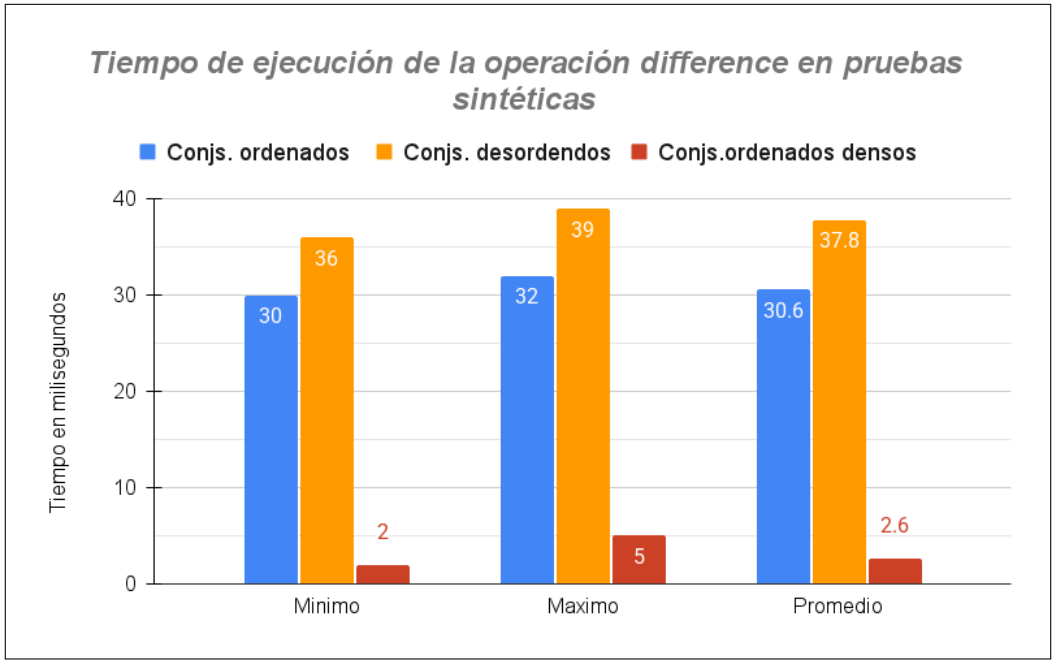


Figura 6.6: Comparación del tiempo de ejecución de la operación `difference`.

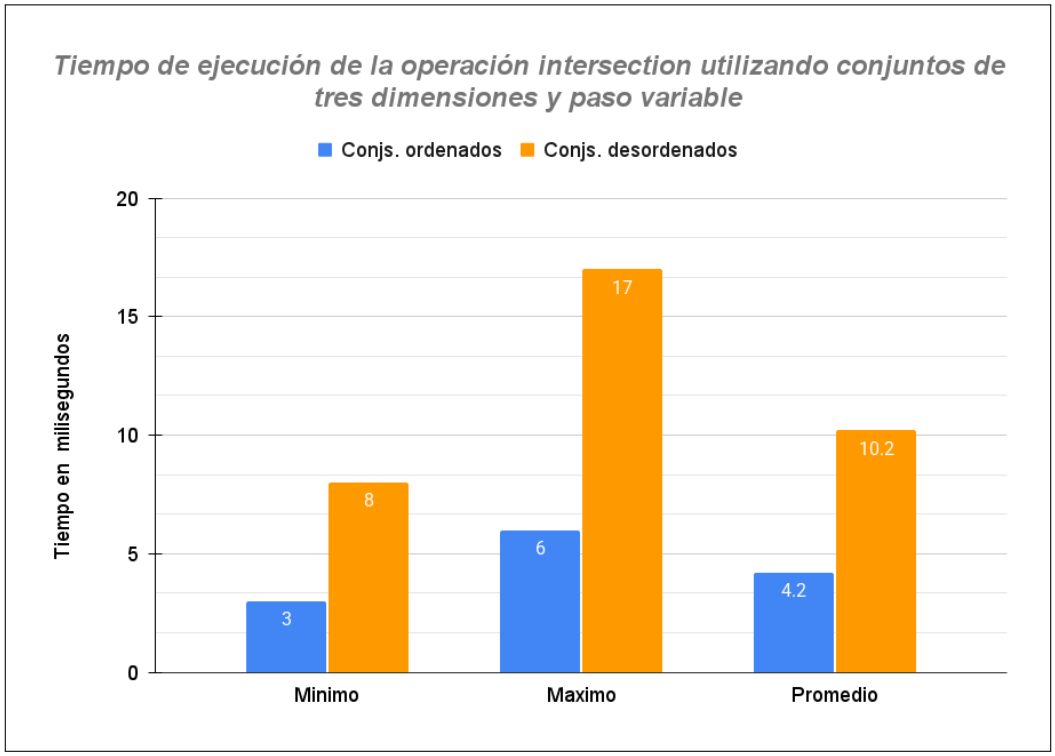


Figura 6.7: Tiempo de ejecución de `intersection` con conjuntos tridimensionales con paso distinto de 1, comparando conjuntos ordenados y desordenados.

- `complement`: En este caso, la optimizaciones permiten aprovechar las múltiples dimensiones disponibles para alcanzar una reducción relativa del **98.87 %**, lo que representa una mejora de aproximadamente **88.10 veces** (Figura 6.8).

  En cuanto al análisis de escalabilidad de esta operación, se observa que la implementación que utiliza conjuntos ordenados presenta un escalado aproximadamente lineal o sublineal, al aumentar la cantidad de multi-intervalos. En contraste, este comportamiento no se mantiene al utilizar la versión basada en conjuntos desordenados, como se muestra en la Figura 6.9, cuyos datos previenen del Cuadro 6.2.

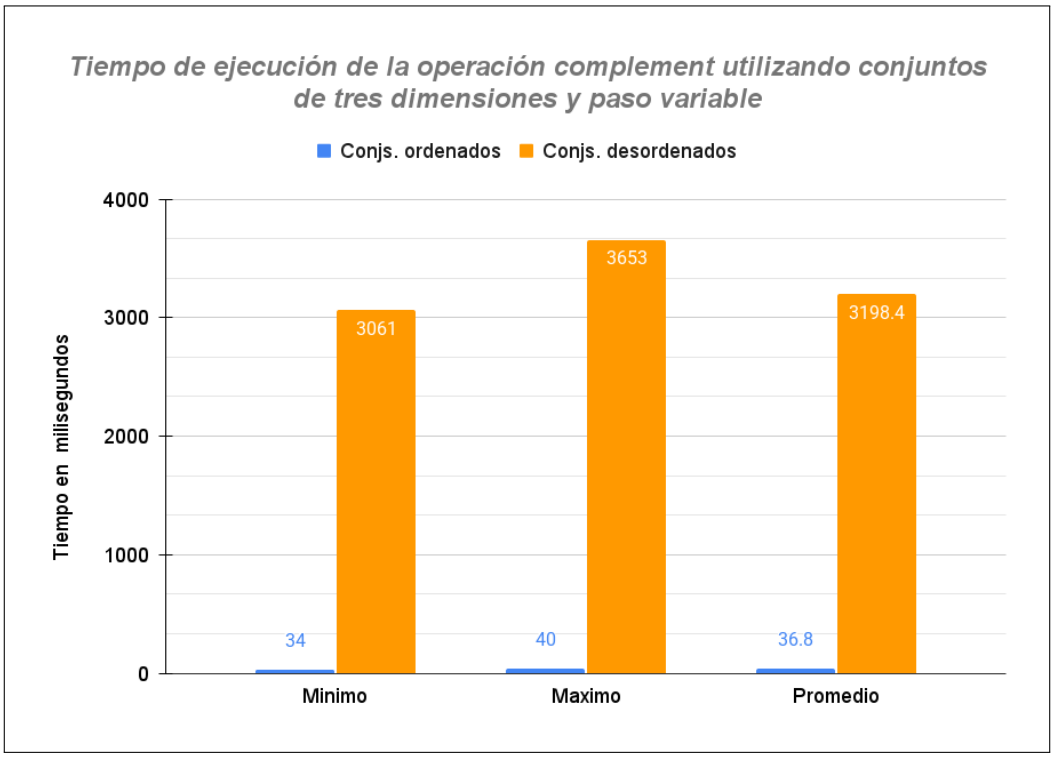


Figura 6.8: Tiempo de ejecución de `complement` con conjuntos tridimensionales con paso distinto de 1, comparando conjuntos ordenados y desordenados.

	Tiempo de ejecución en milisegundos	
Cantidad de multi-intervalos	**Conjuntos ordenados**	**Conjuntos desordenados**
250	13	249
500	23	813
1000	26	3165
2000	58	12392
4000	107	50184
8000	192	200823

Cuadro 6.2: Cuadro de escalado del tiempo de ejecución de la operación `complement`

- `cup`: Dependiente de `complement`, esta operación se beneficia de sus optimizaciones, logrando una mejora del **97.93 %** o **48.41 veces** más rápido que la versión desordenada (Figura 6.10).
- `difference`: También construida sobre otras operaciones, `difference` logra una mejora del **98 %**, equivalente a una ejecución **50.15 veces** más rápida que la versión desordenada (Figura 6.11).
- `disjointCup`: Nuevamente se puede ver como se tiene un empeoramiento en el rendimiento de la operación `disjointCup` para conjuntos ordenandos con respecto a la misma para conjuntos desordenados. Se obtiene entonces un aumento relativo del tiempo de ejecución del **75.61 %** respecto a la versión desordenada (Ver Figura 6.12). Esta prueba es la única dentro de este apartado en la cual se usaron 5000 multi-intervalos por conjunto.

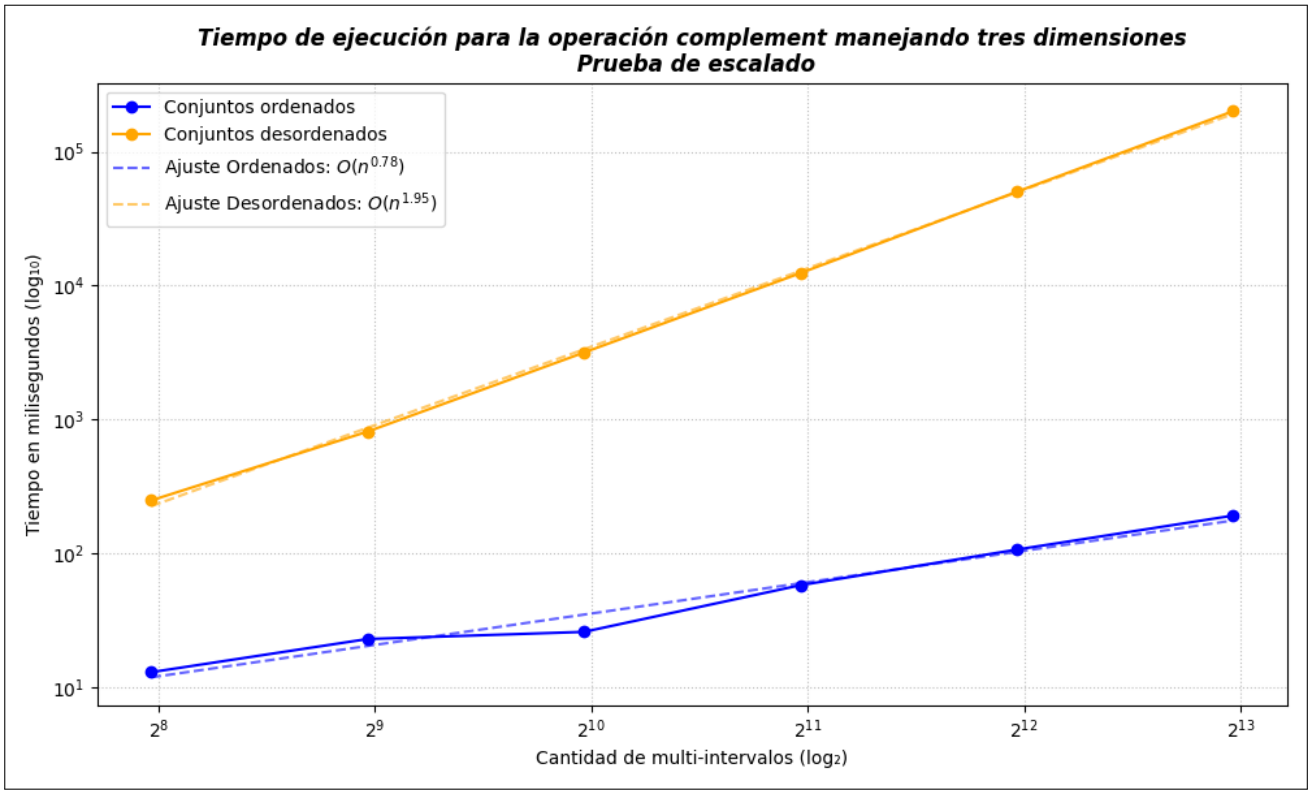


Figura 6.9: Análisis del escalado en el tiempo de ejecución de la operación `complement`.

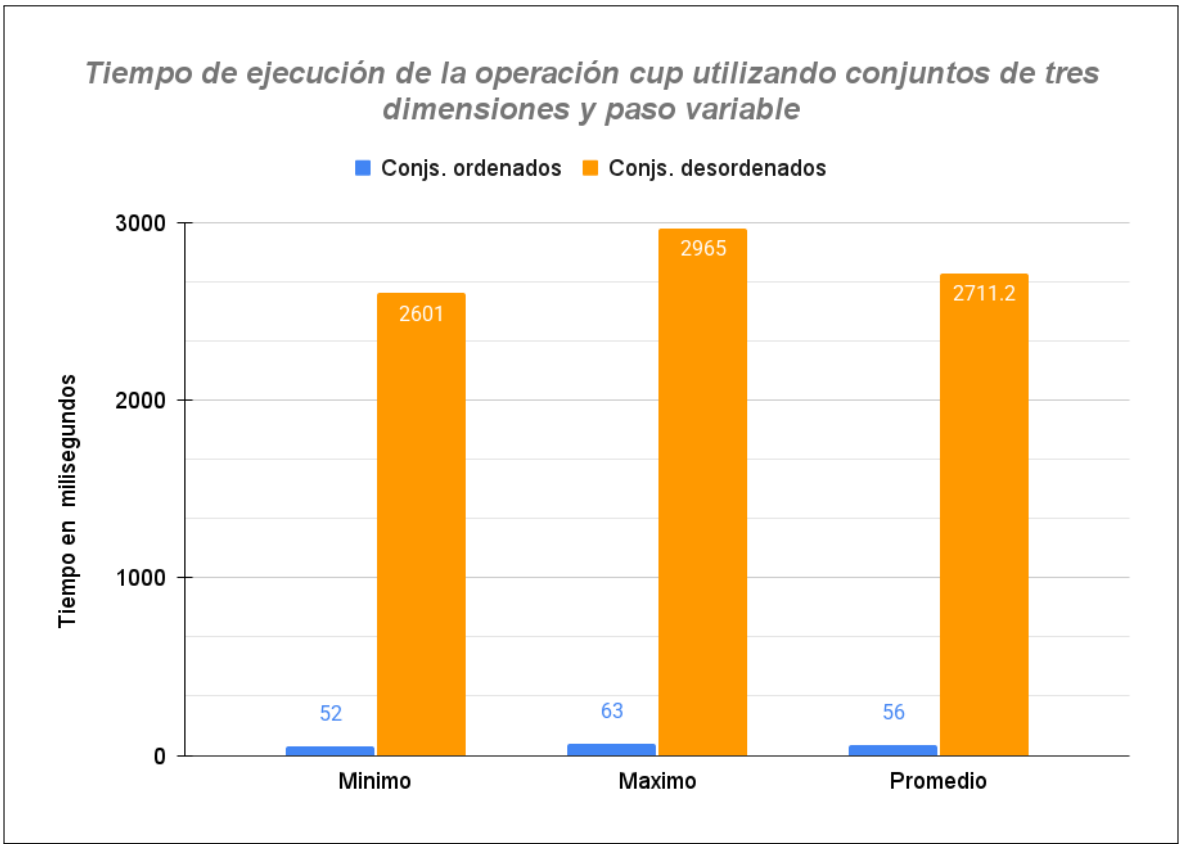


Figura 6.10: Tiempo de ejecución de `cup` con conjuntos tridimensionales con paso distinto de 1, comparando conjuntos ordenados y desordenados.

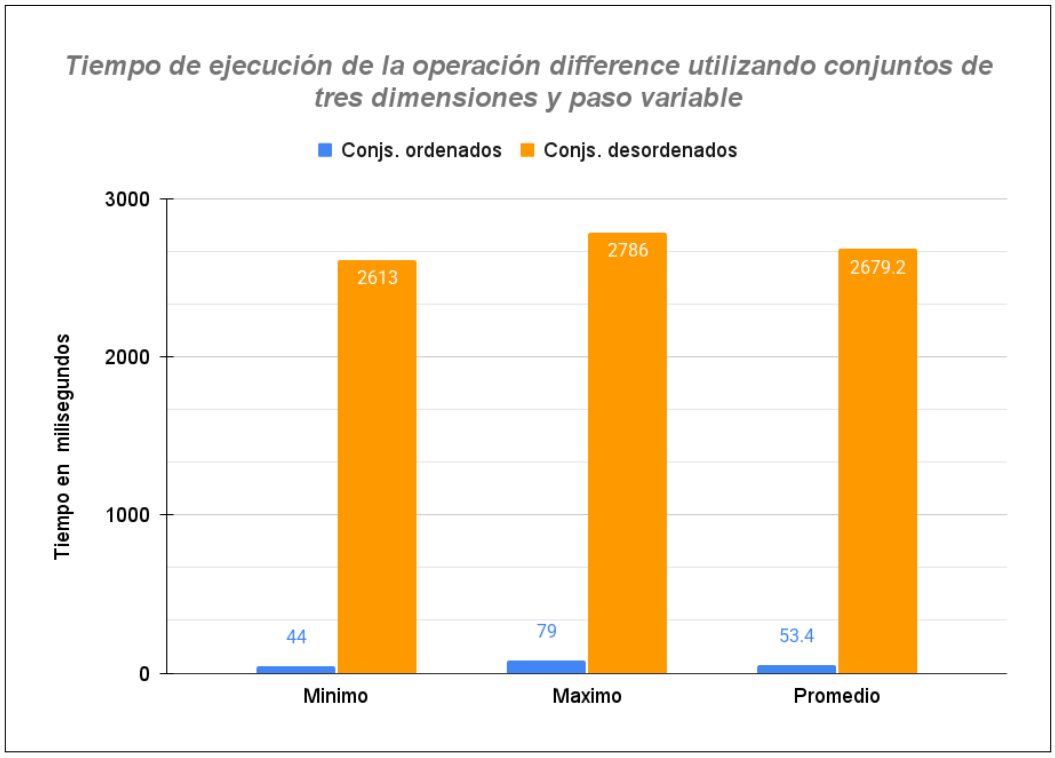


Figura 6.11: Tiempo de ejecución de `difference` con conjuntos tridimensionales con paso distinto de 1, comparando conjuntos ordenados y desordenados.

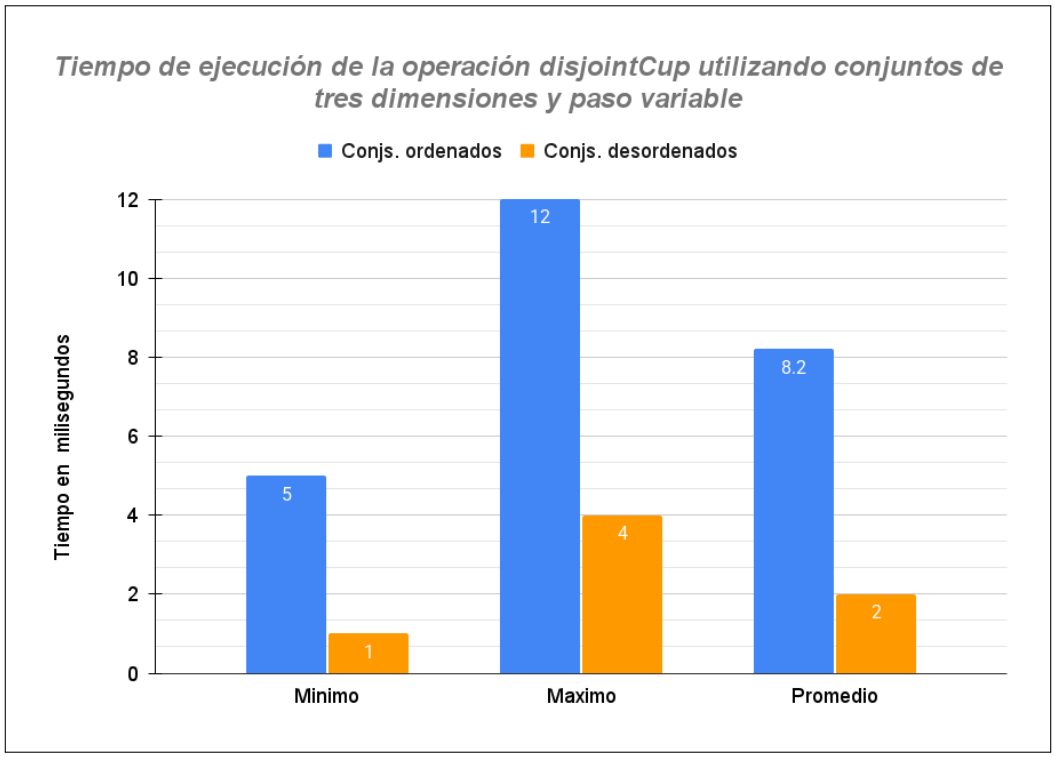


Figura 6.12: Tiempo de ejecución de `disjointCup` con conjuntos tridimensionales con paso distinto de 1, comparando conjuntos ordenados y desordenados.

6.2. Casos de prueba sintéticos para *piecewise maps* ordenados

Al igual que en la sección anterior, se diseñó un conjunto extenso de casos de prueba sintéticos para las operaciones de los *piecewise maps*, con el propósito de evaluar el rendimiento de las dos implementaciones disponibles.

De manera análoga, se presentan a continuación los resultados obtenidos, junto con una comparación detallada entre las variantes de *piecewise maps* ordenados y desordenados. Los valores representados en las gráficas corresponden al **mínimo**, **máximo** y **promedio** de los tiempos de ejecución registrados tras múltiples repeticiones de cada caso de prueba sintético, tal como se procedió en la sección anterior. Además, al igual que entonces, todos los porcentajes y comparaciones mostrados se calcularon sobre los valores **promedio**, *excepto en las pruebas de escalabilidad*, donde solo se dio una ejecución de los casos por las cantidad de mapas.

Cabe señalar que, en esta ocasión, no se considerarán casos de prueba en una única dimensión, dado que, tal como se observó en el análisis de los conjuntos, las optimizaciones no aportan mejoras significativas en dicho contexto. Por lo tanto, se trabajará exclusivamente con tres dimensiones. Adicionalmente, a diferencia que en el caso de conjuntos, cada caso de prueba sintéticos maneja argumentos muy distintos entre si. Por lo que si se desea ver estos en profundidad se debe chequear el archivo *pwmap_perf.cpp* en la carpeta *tests/performance* dentro del repositorio.

- `combine`: La primera operación que se analizará es `combine`, la cual, por su propia naturaleza, se presenta como una candidata a no resultar más eficiente en los *piecewise maps* ordenados. En la Figura 6.13 se observa una reducción del rendimiento de la versión ordenada en comparación con la desordenada. En este caso, el tiempo de ejecución muestra un incremento relativo del **42,89 %** con respecto a la implementación desordenada.

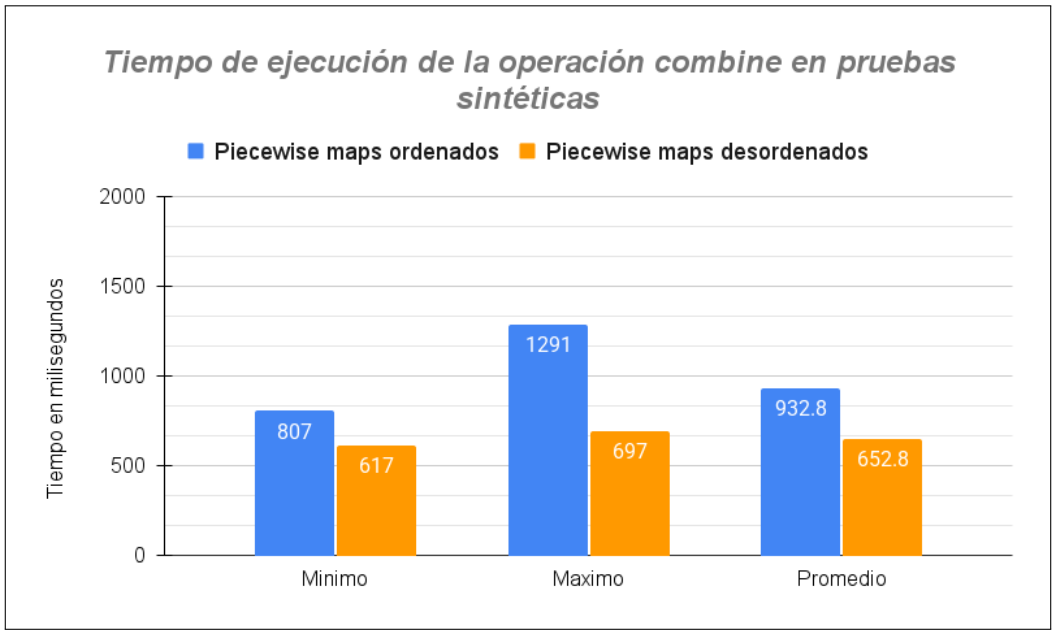


Figura 6.13: Tiempo de ejecución de `combine` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

- `concatenation`: Esta es otra de las operaciones en las que, debido a su propia naturaleza, el orden resulta desfavorable. En la Figura 6.17 se observa que la versión para los *piecewise maps* ordenados presenta un incremento relativo en el tiempo de ejecución del **16,22 %** en comparación con la versión desordenada.

- `restrict`: A continuación se analiza la operación `restrict`, la cual en este caso sí presenta una mejora. En la Figura 6.15 se observa que, aunque la ganancia no es muy significativa, la versión ordenada logra una disminución relativa del tiempo de ejecución del **13,82 %** en comparación con la versión desordenada.

  En el Cuadro 6.3 y en la Figura 6.16 se evidencia que, incluso con las optimizaciones introducidas en la versión ordenada, ambas implementaciones presentan un escalado similar. No obstante, puede apreciarse

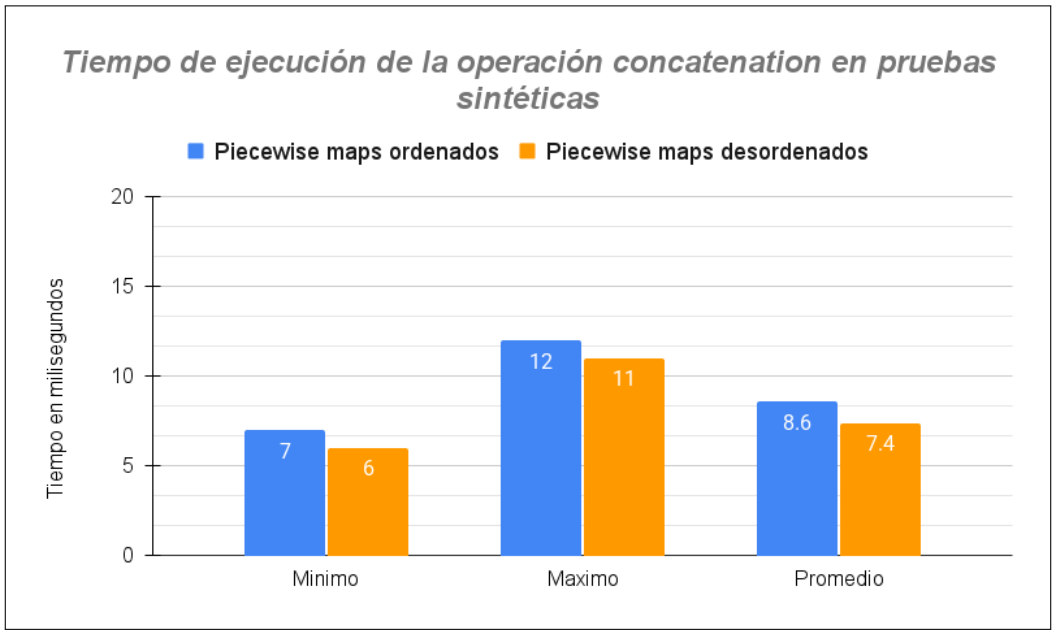


Figura 6.14: Tiempo de ejecución de `concatenation` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

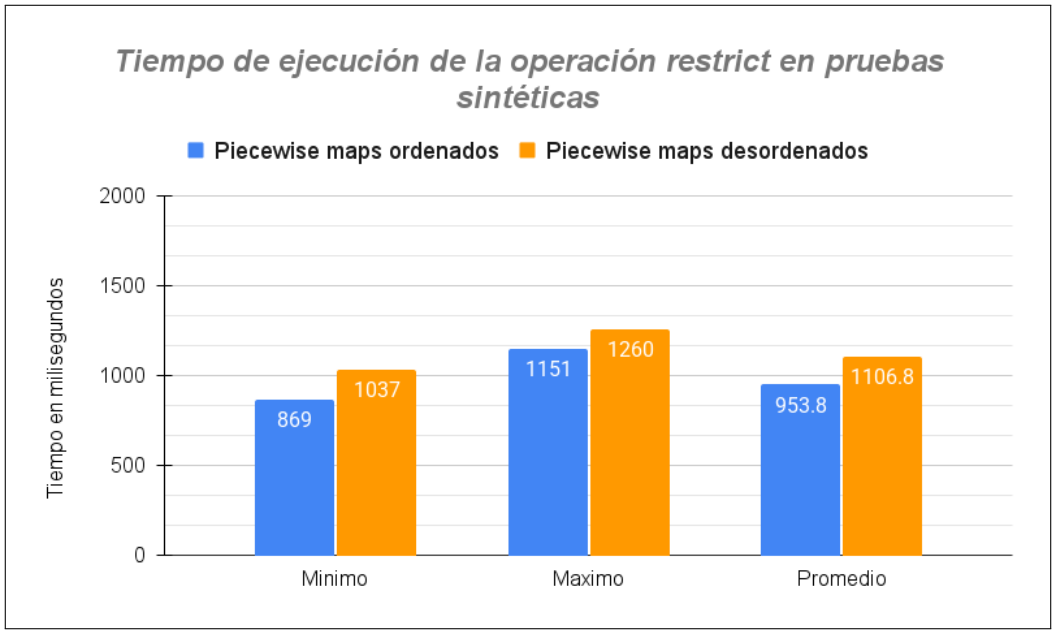


Figura 6.15: Tiempo de ejecución de `restrict` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

que la versión ordenada logra un desempeño levemente superior en términos de escalabilidad.

	Tiempo de ejecución en milisegundos	
Cantidad de mapas	**Piecewise maps ordenados**	**Piecewise maps desordenados**
16	3	3
32	7	12
64	25	27
128	94	106
256	370	424
512	1469	1665
1024	5846	6725

Cuadro 6.3: Cuadro de escalado del tiempo de ejecución de la operación `restrict`

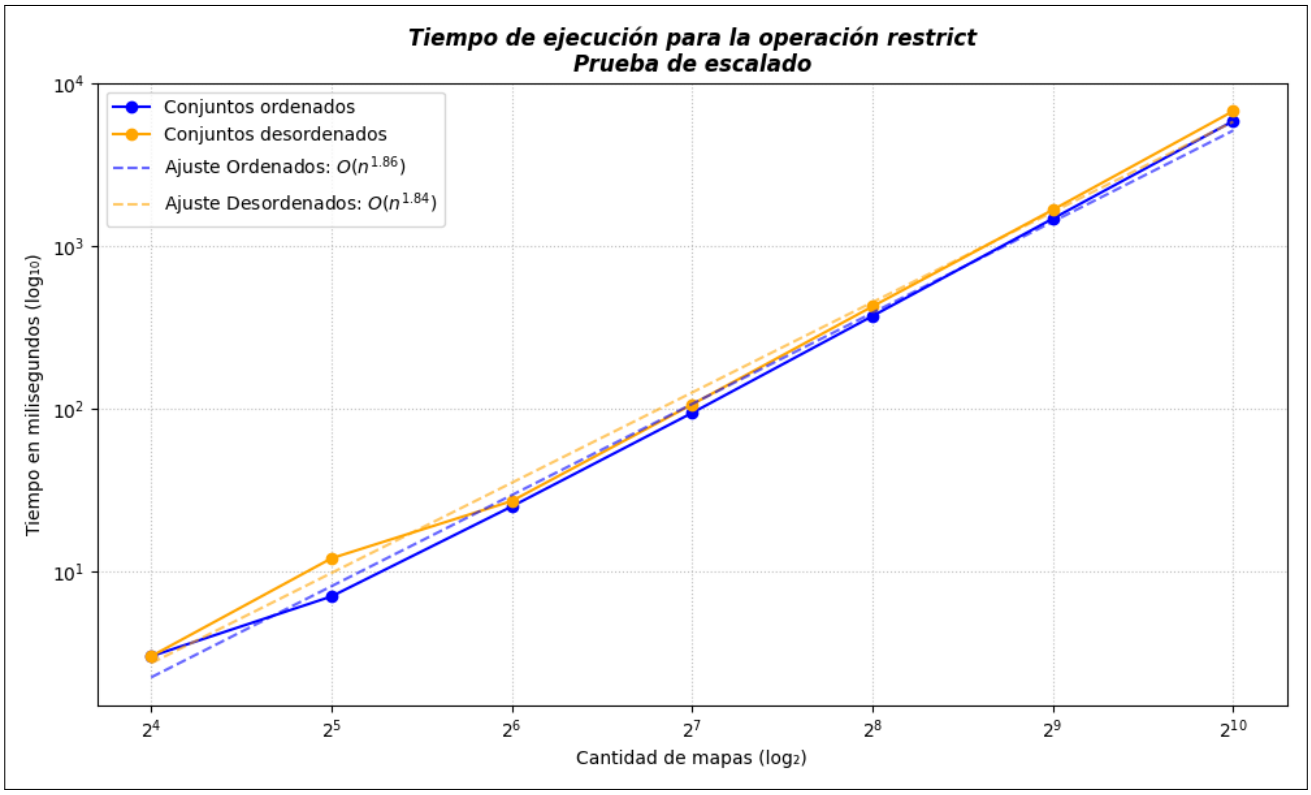


Figura 6.16: Análisis del escalado en el tiempo de ejecución de la operación `restrict`.

- `composition`: El caso de la operación `composition` resulta particularmente interesante, ya que su implementación, en términos generales, incluye un doble ciclo *for*. En la Figura 6.14 se aprecia la marcada diferencia en el rendimiento entre ambas versiones. En este caso, la versión ordenada logra una reducción relativa del tiempo de ejecución del **92,63 %** en comparación con la versión desordenada.

  En el Cuadro 6.4 y en la Figura 6.18 se observa que la operación implementada para los *piecewise maps* ordenados presenta un mejor escalado en comparación con la implementación desordenada. Sin embargo, aún está un poco lejos de alcanzar un comportamiento verdaderamente lineal.

- `offsetDom`: En este caso nuevamente se evidencia una mejora significativa al emplear el orden. En la Figura 6.19 se observa que la versión ordenada logra una reducción relativa del tiempo de ejecución del **46,31 %** en comparación con la versión desordenada.

- `firstInv`: El caso de la operación `firstInv` resulta particularmente peculiar. En la Figura 6.20 se observa que ambas versiones de la operación presentan un rendimiento prácticamente equivalente, con una reducción relativa del tiempo de ejecución de apenas **2,73 %** en la versión ordenada respecto de la desordenada.

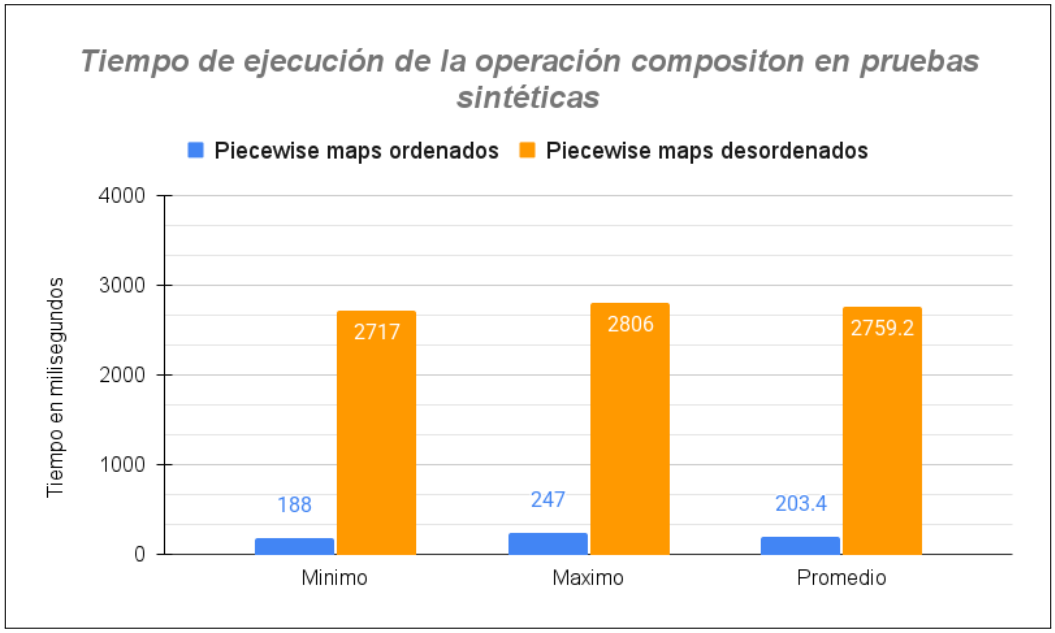


Figura 6.17: Tiempo de ejecución de `composition` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

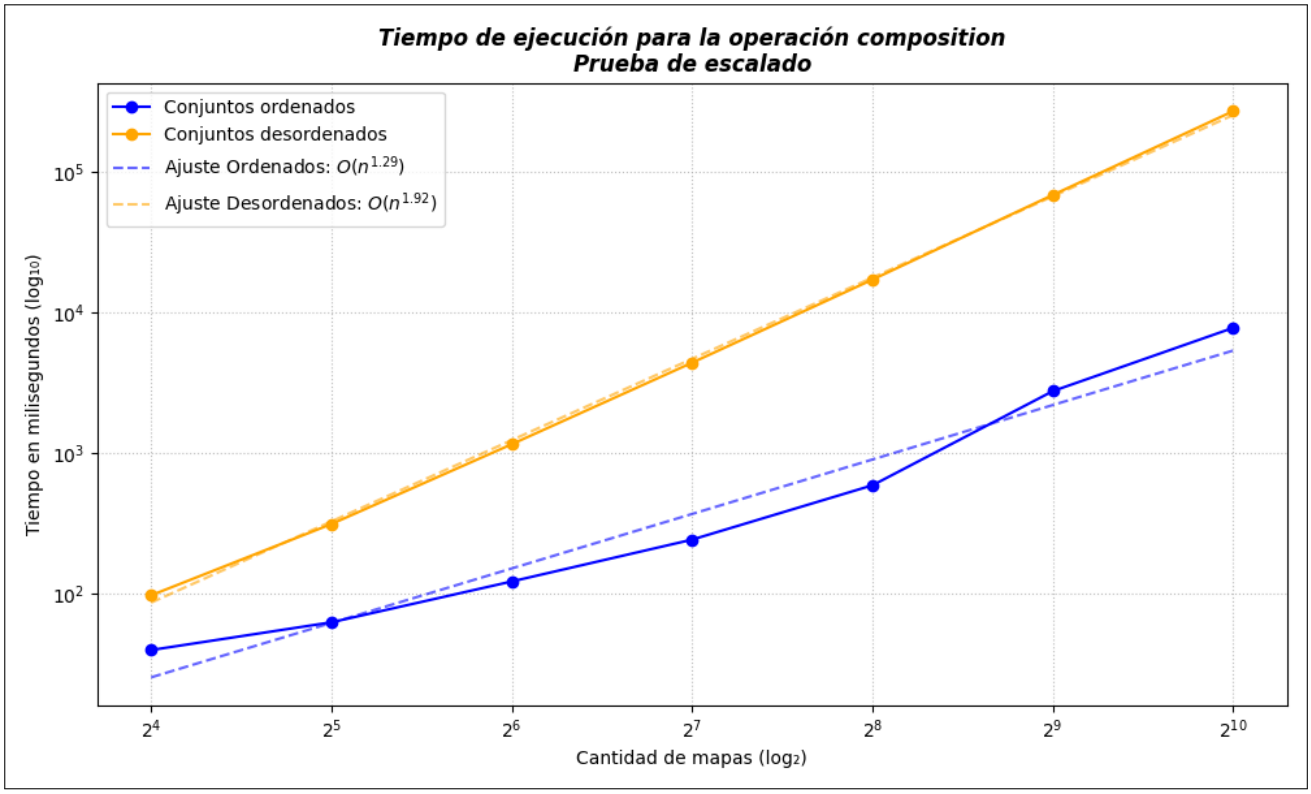


Figura 6.18: Análisis del escalado en el tiempo de ejecución de la operación `composition`.

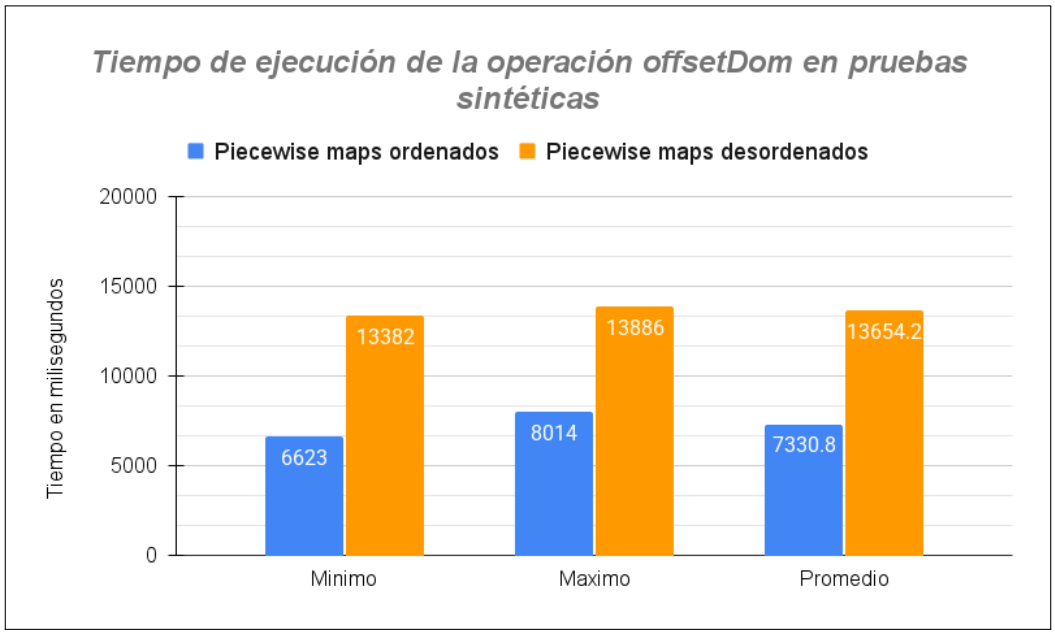


Figura 6.19: Tiempo de ejecución de `offsetDom` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

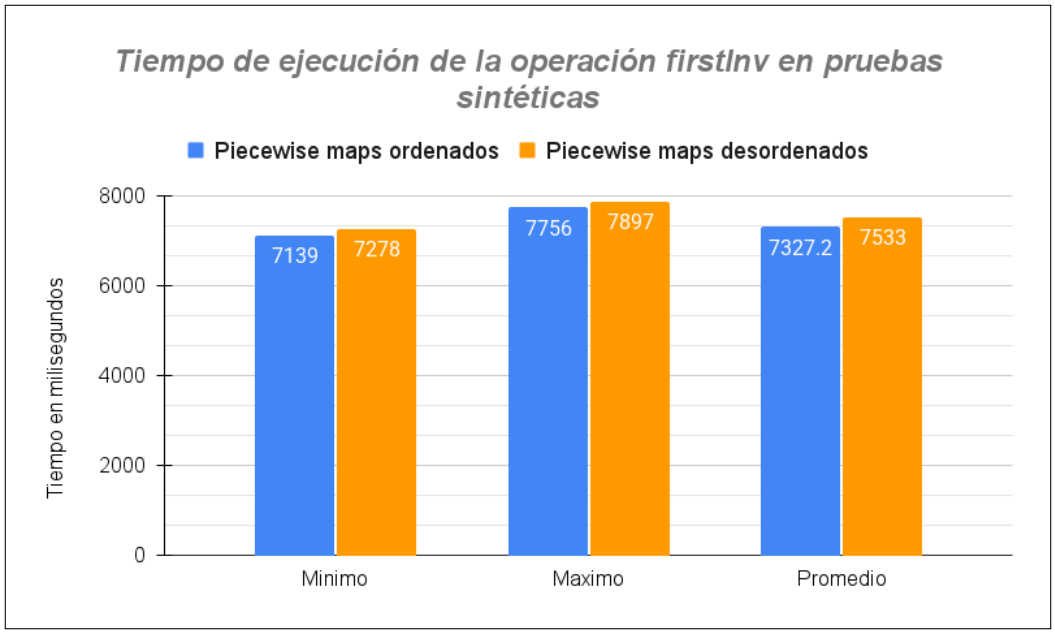


Figura 6.20: Tiempo de ejecución de `firstInv` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

	Tiempo de ejecución en milisegundos	
Cantidad de mapas	**Piecewise maps ordenados**	**Piecewise maps desordenados**
16	40	98
32	63	315
64	123	1164
128	244	4423
256	594	17240
512	2776	68963
1024	7822	272032

Cuadro 6.4: Cuadro de escalado del tiempo de ejecución de la operación `composition`

- `+`: Con la operación de suma se da inicio a aquellas que se benefician del uso de la función `processMapsOrd`. En particular, en la Figura 6.21 se observa una reducción relativa del tiempo de ejecución del **95,89 %** en comparación con la implementación de los *piecewise maps* desordenados, siendo la versión ordenada aproximadamente **24,35** veces más rápida.

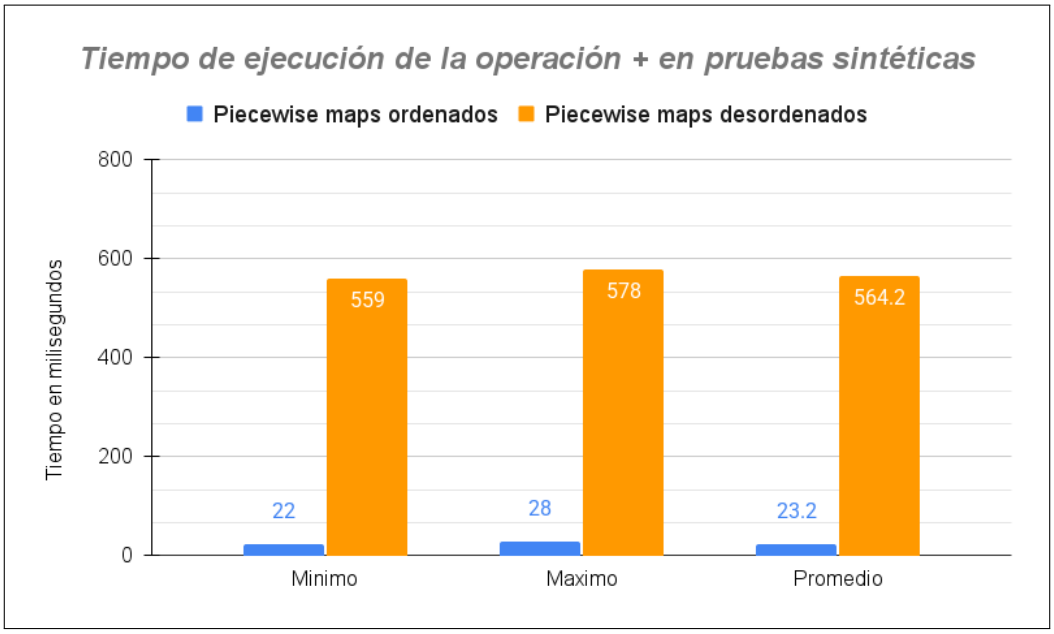


Figura 6.21: Tiempo de ejecución de `+` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

En el Cuadro 6.5 y en la Figura 6.22 se observa cómo la operación de suma varía significativamente en su escalado entre las distintas versiones. En este caso, la versión implementada para los *piecewise maps* ordenados presenta un escalado mucho más eficiente que la versión desordenada, mostrando un escaldo lineal.

- `-`: El caso de la operación de `resta` no resulta tan desproporcionado como el de la `suma`. En la Figura 6.23 se observa que la versión ordenada de los *piecewise maps* logra una reducción relativa del tiempo de ejecución del **80,15 %** con respecto a la implementación desordenada. Siendo **5** veces mas rápida.

- `equalImage`: Esta operación, al igual que la suma, se ve completamente optimizada gracias al uso de la operación `processMapsOrd`. En la Figura 6.24 se aprecia una reducción relativa del tiempo de ejecución del **98,70 %** con respecto a la versión desordenada, siendo la versión ordenada unas **76.82** veces más rápida.

- `minAdjMap`: Se analiza finalmente la última operación, la cual nuevamente demuestra cómo el orden contribuye a mejorar el rendimiento. En la Figura 6.25 se evidencia una reducción relativa del tiempo de ejecución del **74,63 %** en comparación con la versión desordenada.

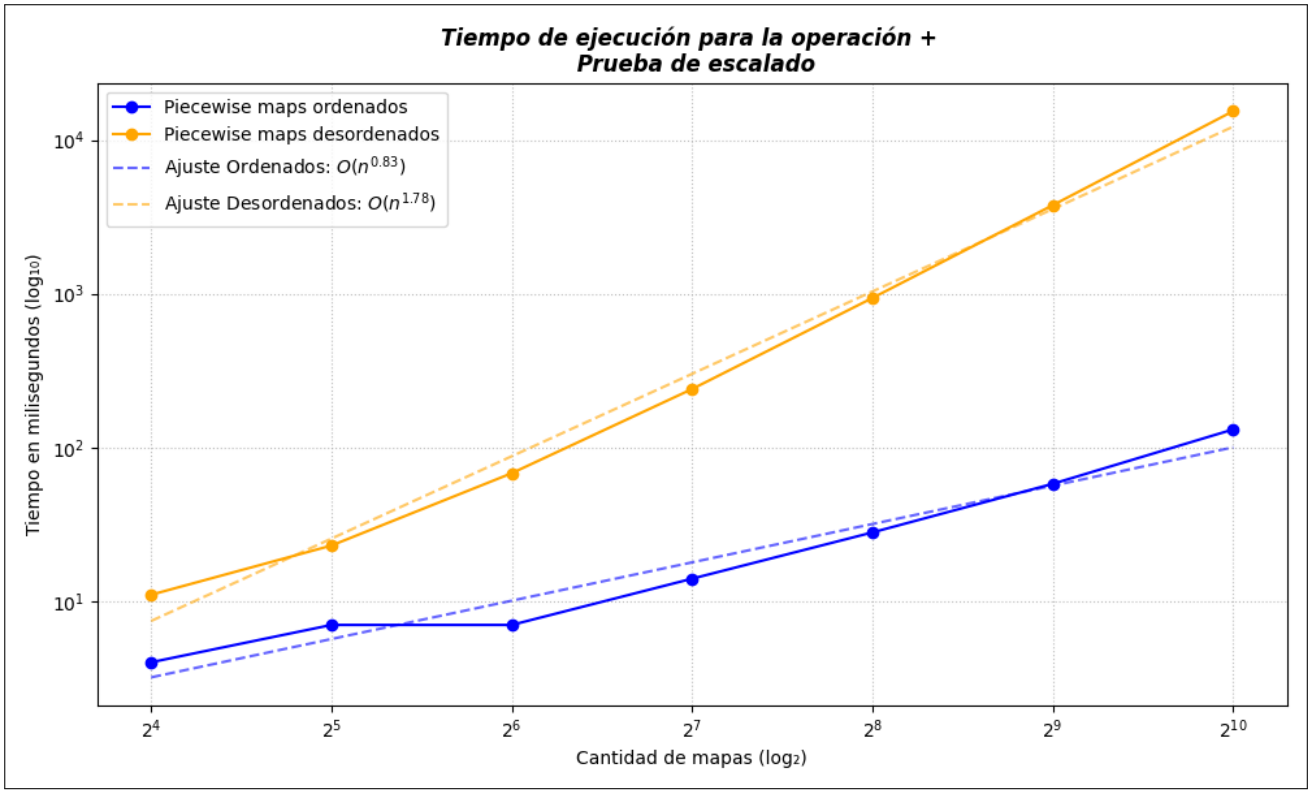


Figura 6.22: Análisis del escalado en el tiempo de ejecución de la operación +.

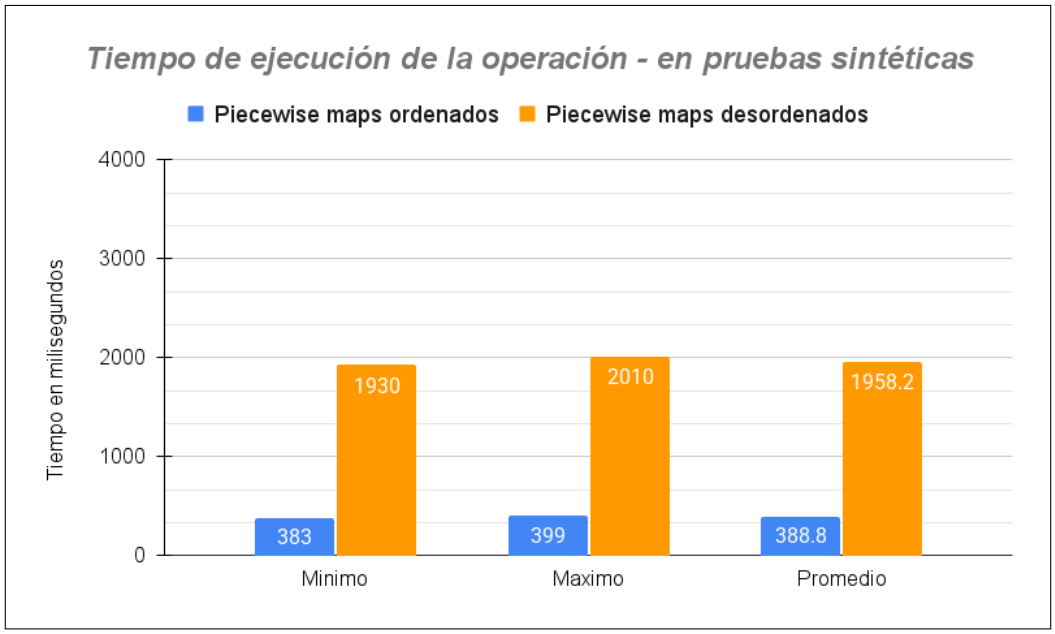


Figura 6.23: Tiempo de ejecución de – utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

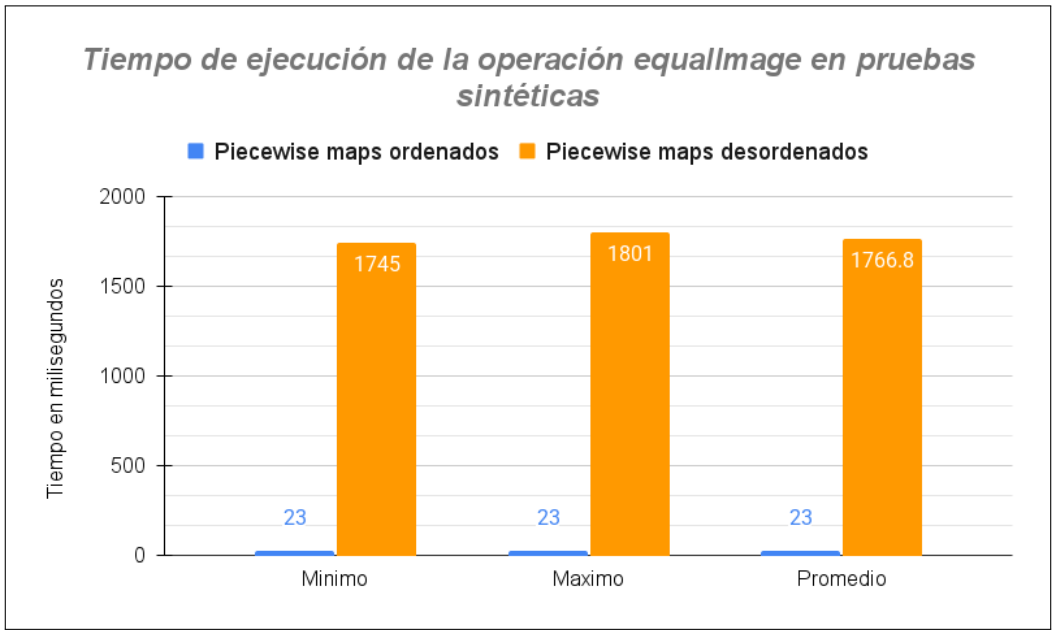


Figura 6.24: Tiempo de ejecución de `equalImage` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

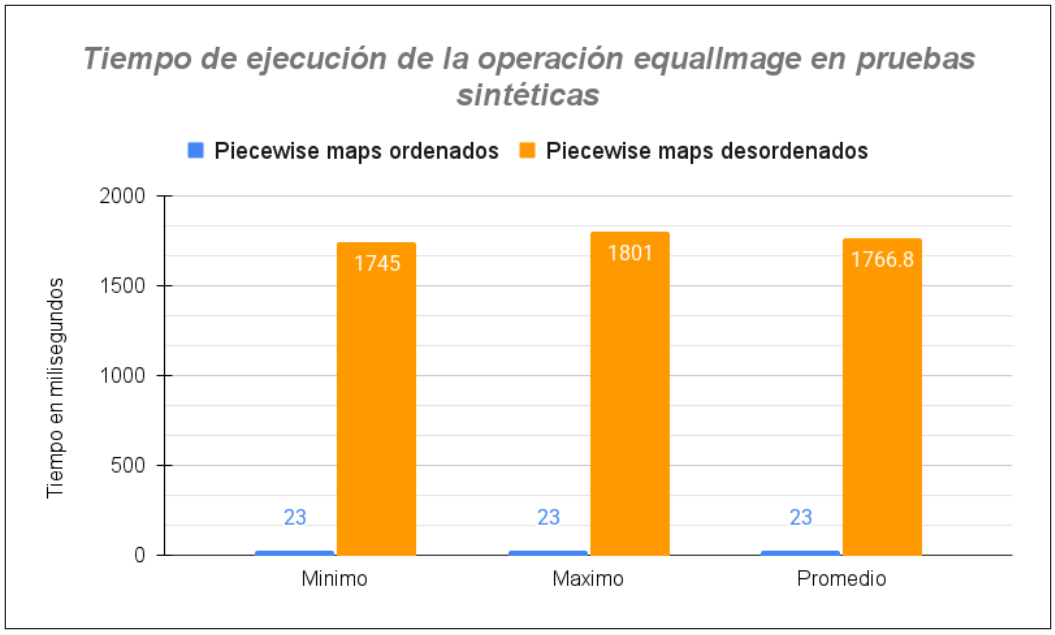


Figura 6.25: Tiempo de ejecución de `minAdjMap` utilizando tres dimensiones, comparando *piecewise maps* ordenados y desordenados.

	Tiempo de ejecución en milisegundos	
Cantidad de mapas	**Piecewise maps ordenados**	**Piecewise maps desordenados**
16	4	11
32	7	23
64	7	68
128	14	240
256	28	939
512	58	3771
1024	131	15424

Cuadro 6.5: Cuadro de escalado del tiempo de ejecución de la operación `+`

En el Cuadro 6.6 y en la Figura 6.26 se observa que ambas versiones presentan un escalado similar, aunque la versión que utiliza el orden muestra un comportamiento un poco más eficiente.

	Tiempo de ejecución en milisegundos	
Cantidad de mapas	**Piecewise maps ordenados**	**Piecewise maps desordenados**
16	1	2
32	7	15
64	15	33
128	37	98
256	104	347
512	375	1436
1024	1473	5544

Cuadro 6.6: Cuadro de escalado del tiempo de ejecución de la operación `minAdjMap`

6.3. Casos de prueba

En la sección anterior se trabajó con casos de prueba sintéticos para conjuntos, diseñados específicamente para verificar el rendimiento de las distintas operaciones sobre conjuntos en las diferentes implementaciones. Sin embargo, también resulta fundamental analizar cómo se desempeñan las implementaciones bajo condiciones más realistas. Dado que los conjuntos y los *piecewise maps* se crearon para representar SBGs de manera compacta se evaluara el rendimiento de dichas estructura con diversas corridas de algoritmos disponibles para SBG.

Por ello, en esta sección se emplearán distintos escenarios o casos de prueba, con el objetivo de observar cómo varía el rendimiento de las distintas implementaciones tanto de conjuntos como de *piecewise maps*.

En particular, para estos tests se considerarán cuatro mediciones de tiempo:

- **Total match exec time:** tiempo total dedicado a la ejecución del algoritmo de matching.
- **Total SCC exec time:** tiempo total empleado en el algoritmo de componentes fuertemente conexas (SCC).
- **Total topological sort exec time:** tiempo total utilizado para realizar el ordenamiento topológico.
- **Total time:** suma de los tres tiempos anteriores, representando el tiempo total de ejecución.

En particular, se considerarán cuatro versiones o combinaciones de implementaciones:

- **Versión 0 (conjuntos y *piecewise maps* desordenados):** esta fue la primera versión, o más precisamente, combinación de implementaciones, incluida en la biblioteca SBG, capaz de operar sobre una o más dimensiones.

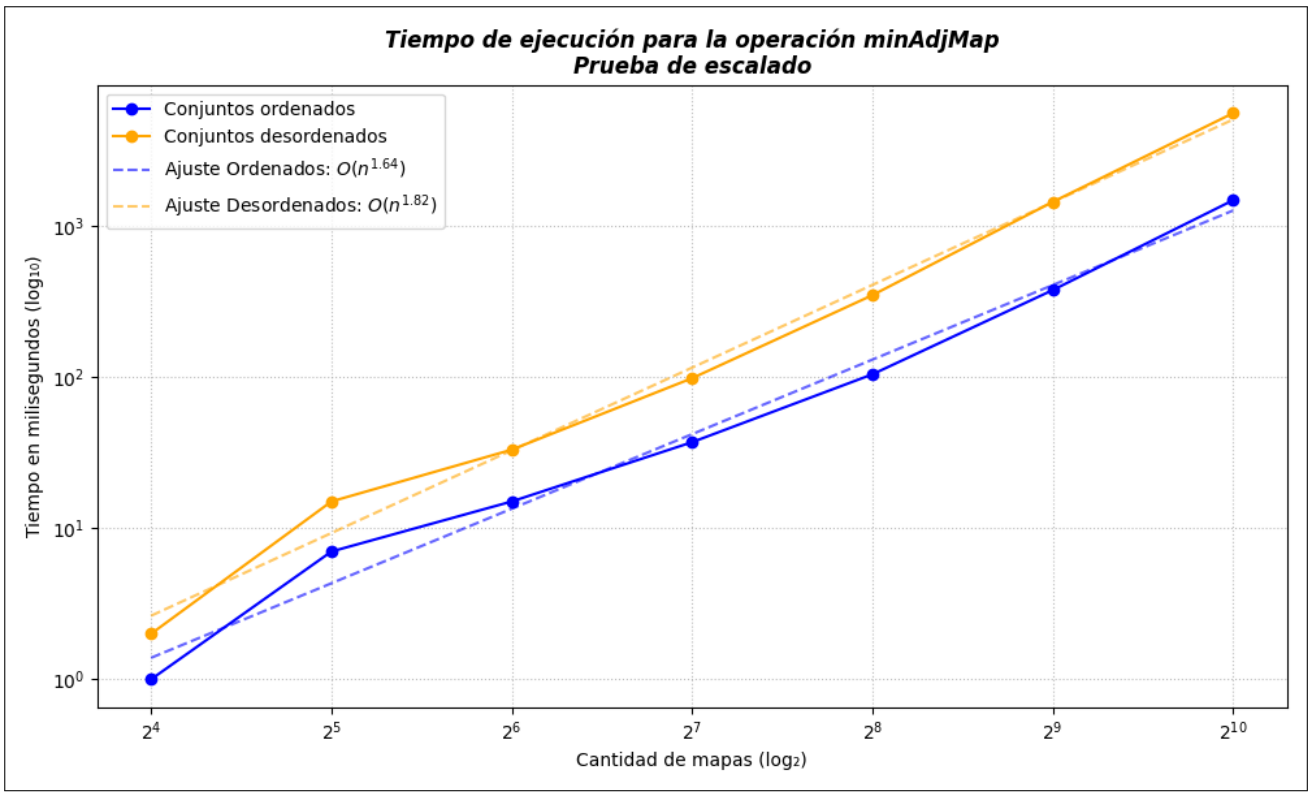


Figura 6.26: Análisis del escalado en el tiempo de ejecución de la operación `minAdjMap`.

- **Versión 1 (conjuntos ordenados densos y *piecewise maps* desordenados):** corresponde a la segunda combinación introducida en la biblioteca, en la que se optimizan los conjuntos mediante una representación densa, aunque limitada al uso de una única dimensión.
- **Versión 2 (conjuntos ordenados y *piecewise maps* desordenados):** en esta segunda versión se incorpora la implementación general de conjuntos ordenados, optimizando en base al orden de los conjuntos. Esta versión permite trabajar con múltiples dimensiones nuevamente.
- **Versión 3 (conjuntos y *piecewise maps* ordenados):** esta tercera versión utiliza la implementación ordenada de los *piecewise maps* ademas de la de conjuntos, también pudiendo trabajar con múltiples dimensiones.

Estas versiones permiten analizar la evolución en los tiempos de ejecución según las distintas alternativas o combinaciones presentes en la libreria SBG

6.3.1. Caso de prueba en una dimensión

Inicialmente, se comenzará con un caso de prueba en el que se trabaja sobre una única dimensión. El código correspondiente se encuentra disponible en el archivo *pw_map_test_1dim.test*, ubicado en la carpeta *test* dentro del repositorio.

En la Figura 6.27 se presentan los distintos tiempos de ejecución obtenidos para el caso de prueba mencionado, utilizando las diferentes versiones introducidas previamente. En este caso, el caso de prueba fue ejecutado múltiples veces con el objetivo de obtener un promedio consistente de todos los tiempos de ejecución para todas las implementaciones, a partir del cual se realizan los análisis. A partir de estos resultados, pueden extraerse las siguientes conclusiones:

- Las versiones que incorporan orden son más eficientes que la Versión 0, especialmente en lo que respecta al *Total match exec time*. En particular, se observa una reducción relativa de **57.59 %**, **45.25 %** y **36.36 %** para las Versiones 1, 2 y 3, respectivamente, en comparación con la Versión 0.

- De forma similar, también se registra una reducción en el *Total SCC exec time* para las versiones que utilizan orden, en relación con la Versión 0.
- En lo que respecta al *Total topological sort exec time*, se observa que todas las versiones presentan un rendimiento prácticamente equivalente.
- En base a los puntos anteriores, puede concluirse que la reducción observada en el *Total time* de las versiones con orden se debe principalmente a las mejoras en el *Total match exec time*. En efecto, se obtienen reducciones relativas del **44.50 %**, **35.36 %** y **35.83 %** para las Versiones 1, 2 y 3, respectivamente, en comparación con la Versión 0.

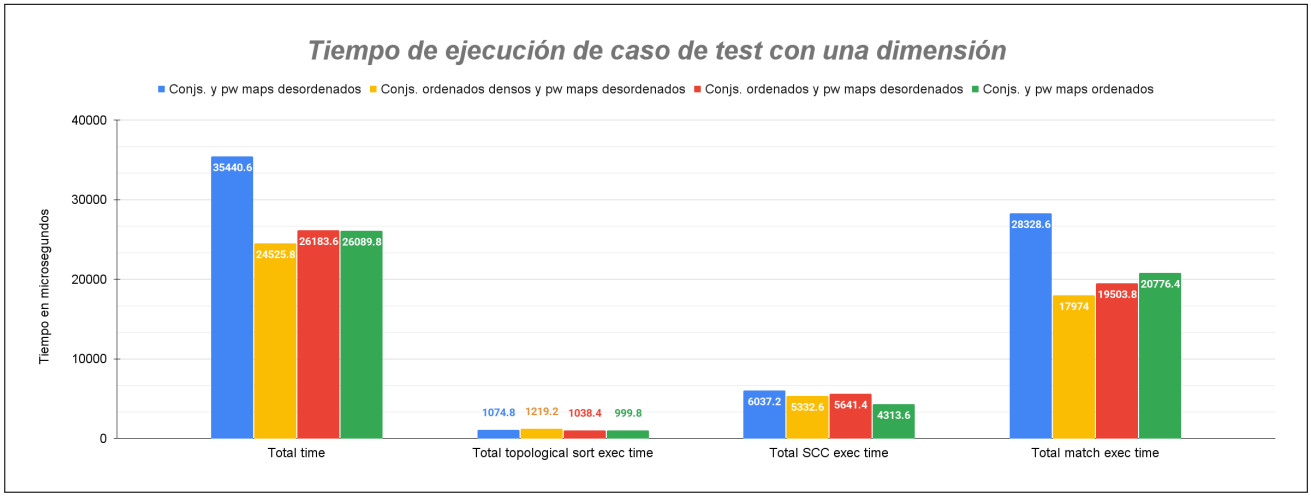


Figura 6.27: Comparación de los diferentes tiempos de ejecución de las diferentes versiones planteadas bajo un caso de prueba en una dimensión.

Ahora bien, este caso de prueba está basado en un grafo de tamaño relativamente pequeño, por lo que resulta relevante analizar cómo se comportan las distintas versiones a medida que se incrementa el tamaño del grafo, es decir, cómo escalan. Para evaluar esto, los casos de prueba considerados utilizan una constante de repetición que toma el grafo original y lo replica la cantidad de veces indicada, generando así un único grafo más grande. De esta manera, es posible observar cómo escalan las diferentes combinaciones al aumentar el tamaño del grafo del caso de prueba.

6.3.2. Escalado - Caso de prueba en una dimensión

Con lo anteriormente mencionado, se procede a realizar un análisis de escalabilidad sobre los cuatro tiempos de ejecución presentados para cada una de las versiones propuestas.

6.3.2.1. Escalado del *Total Match exec time*

En el Cuadro 6.7 y la Figura 6.28 se observa cómo escala el *Total Match exec time* a medida que se duplica el tamaño del grafo. Se puede notar que los tiempos de ejecución crecen de forma aproximadamente cuadrática en todas las versiones. Sin embargo, aquellas versiones que incorporan ordenamiento escalan de manera significativamente más lenta que la Versión 0.

A pesar de que las versiones con orden escalan de forma similar entre sí, es importante destacar que su *reducción relativa* respecto a la Versión 0 mejora progresivamente a medida que aumenta el tamaño del grafo. En particular incluso, la Versión 3, la más optimizada, escala aún más lentamente que la Versión 1, lo cual se debe al uso más eficiente de los *piecewise maps* ordenados.

Con un único grafo, las reducciones relativas en el *Total Match exec time* con respecto a la Versión 0 son de un **22.67 %** para la Versión 2 y de un **53.33 %** para la Versión 3. Al llegar a 64 copias del grafo original, estas mejoras aumentan a **58.34 %** y **65.42 %**, respectivamente.

En cuanto a la comparación entre la Versión 3 y la Versión 1, esta última siendo la más optimizada para conjuntos en una dimensión, se observa que en el caso de un solo grafo la Versión 3 rinde peor, con un incremento relativo del **27.27 %**. Sin embargo, al llegar a 64 copias del grafo, la tendencia se revierte y la Versión 3 logra una reducción del **17,68 %** respecto a la Versión 1. Esto sugiere que, a medida que el tamaño del grafo continúe creciendo, esta ventaja relativa debería incrementarse aún más todo gracias a la implementación *piecewise maps* ordenados.

	***Total Match exec time* en milisegundos**			
Cantidad de copias	**Versión 0**	**Versión 1**	**Versión 2**	**Versión 3**
1	30	11	22	14
2	52	46	43	37
4	141	90	106	90
8	459	298	323	233
16	2811	1362	1490	782
32	8142	4107	5532	3056
64	40422	16974	16838	13975

Cuadro 6.7: Cuadro comparativo del *Total Match exec time* de las diferentes versiones planteadas bajo un caso de prueba en una dimensión, variando el tamaño del caso de prueba

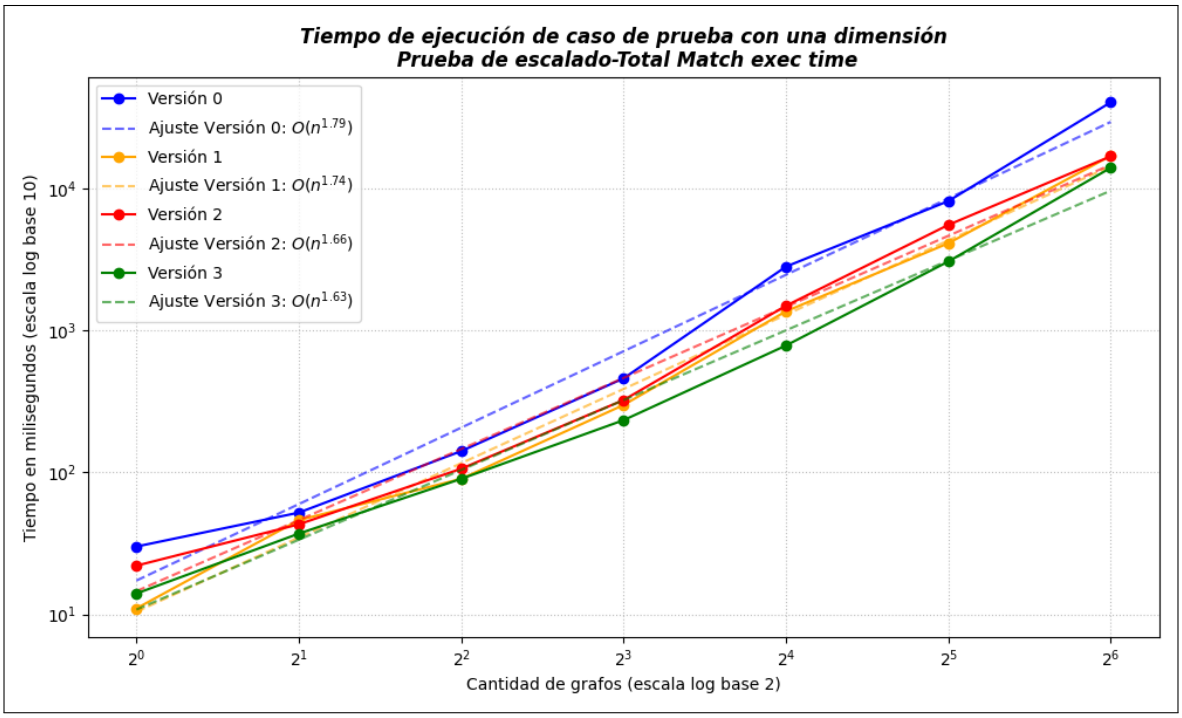


Figura 6.28: Gráfico comparativo del *Total Match exec time* de las diferentes versiones planteadas bajo un caso de prueba en una dimensión, variando el tamaño del caso de prueba.

6.3.2.2. Escalado del *Total SCC exec time*

En cuanto a este tiempo de ejecución, pueden observarse algunos comportamientos interesantes en el Cuadro 6.8 y la Figura 6.29. En ella se aprecia nuevamente cómo el tiempo tiende a crecer de forma aproximadamente cuadrática para todas las versiones. Sin embargo, las versiones que incorporan orden muestran un comportamiento particular: aunque resultan más eficientes que la Versión 0, su escalabilidad parece verse afectada por la complejidad del ordenamiento.

Específicamente, a medida que se incrementa el tamaño del grafo, las versiones con mayor manejo de orden escalan de forma menos favorable. En la figura se observa cómo la Versión 2 escala peor que la Versión 1, y a su vez, la Versión 3 escala peor que la Versión 2. Esto puede deberse a como es que utiliza las operaciones disponibles de conjuntos y *piecewise maps* el algoritmo SCC.

Al comienzo, la Versión 2 presenta una reducción relativa del **57.14 %** en comparación con la Versión 1, mientras que la Versión 3 tiene un rendimiento equivalente al de la Versión 2. Sin embargo, al llegar a 64 copias del grafo original, la situación se revierte: la Versión 2 muestra un aumento relativo del **17.45 %** respecto a la Versión 1, y la Versión 3 se comporta aún peor, con un aumento del **38.44 %** en relación con la Versión 2.

	***Total SCC exec time* en milisegundos**			
Cantidad de copias	**Versión 0**	**Versión 1**	**Versión 2**	**Versión 3**
1	4	7	3	3
2	9	5	7	8
4	19	12	16	19
8	59	34	46	53
16	426	233	306	316
32	975	415	908	781
64	5185	1690	2046	2833

Cuadro 6.8: Cuadro comparativo del *Total SCC exec time* de las diferentes versiones planteadas bajo un caso de prueba en una dimensión, variando el tamaño del caso de prueba

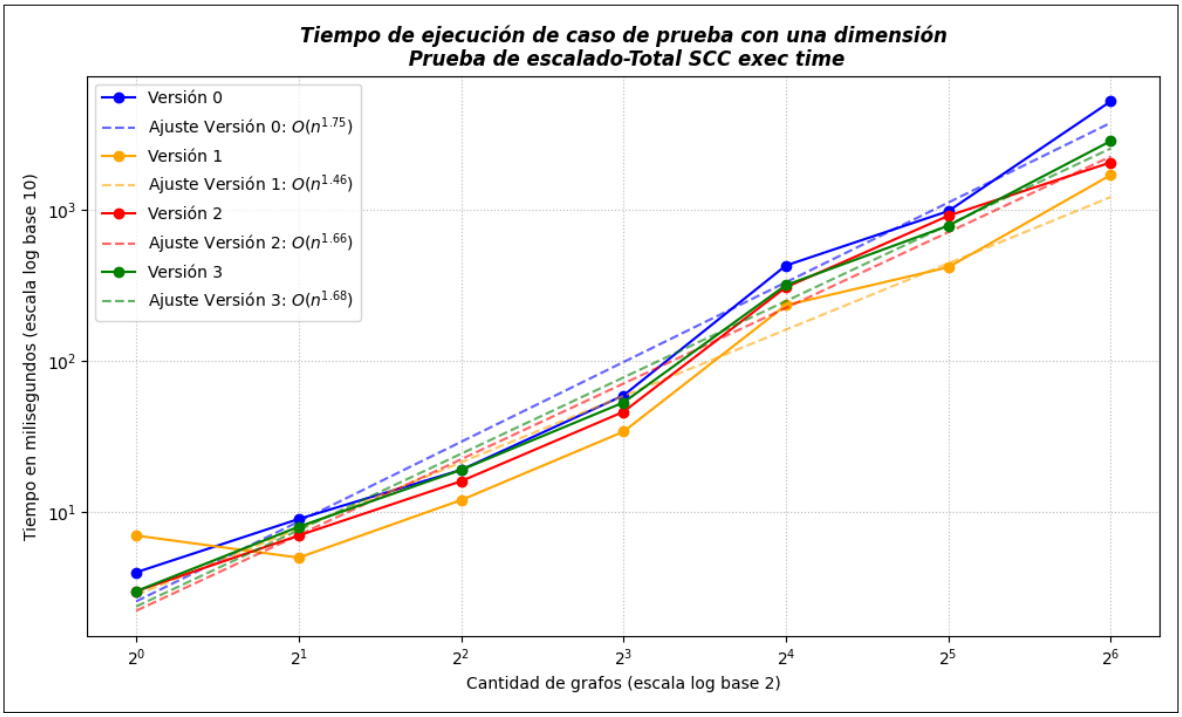


Figura 6.29: Gráfico comparativo del *Total SCC exec time* de las diferentes versiones planteadas bajo un caso de prueba en una dimensión, variando el tamaño del caso de prueba.

6.3.2.3. Escalado del *Total topological sort exec time*

En este caso, nuevamente se observan fenómenos relevantes tanto en el Cuadro 6.9 como en la Figura 6.30. Los resultados muestran que todas las versiones escalan con una complejidad cercana a la cúbica, lo cual resulta preocupante desde el punto de vista del rendimiento. Este comportamiento indica que, a medida que crece la cantidad de copias del grafo original involucrados, el tiempo de ejecución se incrementa de forma abrupta, lo cual limita la viabilidad de estas implementaciones en contextos de gran escala.

Lo que resulta particularmente llamativo es el hecho de que la Versión 2 presenta un rendimiento aún peor que la Versión 0, lo cual contradice las expectativas, dado que esta última se consideraba como la implementación base o menos optimizada. Solo las Versiones 1 y 3 logran escalar de manera más eficiente que la Versión 0, aunque aun así, sus tasas de crecimiento siguen siendo altas. Esta situación sugiere que, si bien se han logrado algunas mejoras, aún existe un margen considerable para optimizar los algoritmos subyacentes.

	Total topological sort exec time en milisegundos			
Cantidad de copias	**Versión 0**	**Versión 1**	**Versión 2**	**Versión 3**
1	0	1	0	0
2	3	2	3	4
4	15	11	15	15
8	89	85	122	104
16	682	558	609	974
32	3441	3859	5107	2998
64	25100	34945	27622	22304

Cuadro 6.9: Cuadro comparativo del *Total topological sort exec time* de las diferentes versiones planteadas bajo un caso de prueba en una dimensión, variando el tamaño del caso de prueba

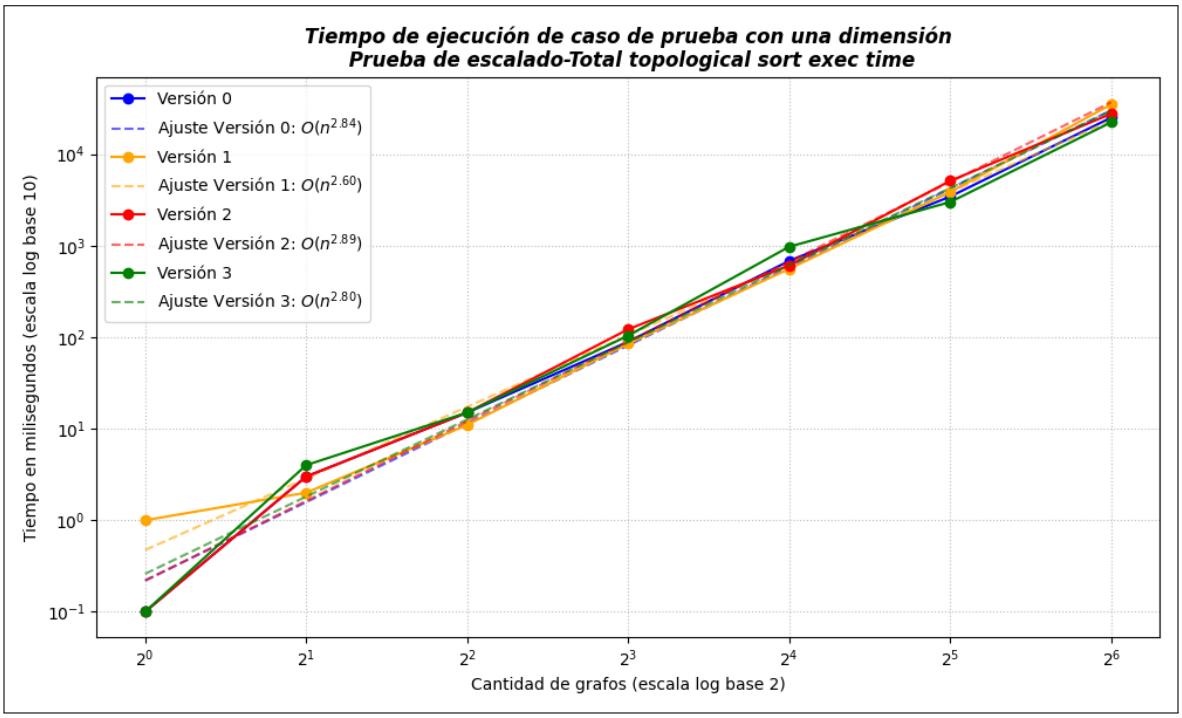


Figura 6.30: Gráfico comparativo del *Total topological sort exec time* de las diferentes versiones planteadas bajo un caso de prueba en una dimensión, variando el tamaño del caso de prueba.

6.3.2.4. Escalado del *Total time*

Por último, se presenta el tiempo total de ejecución, o *Total time*, el cual se obtiene al sumar todos los tiempos anteriores. De esta manera, es posible analizar cómo escala la ejecución completa del caso de prueba. En el Cuadro 6.10 y la Figura 6.31 se observan los escalados correspondientes a las distintas versiones propuestas.

En este caso, se aprecia un comportamiento muy particular, con un escalado aproximadamente cuadrático para todas las Versiones, donde las versiones ordenadas superan por muy poco o incluso igualan el escalado de la Versión 0. Adicionalmente, puede notarse que la Versión 3 es la que mejor se desempeña al alcanzar el máximo número de copias del grafo original, obteniendo una reducción relativa del **44.69 %**, **27.03 %** y **15.88 %** en comparación con las Versiones 0, 1 y 2, respectivamente. Claramente cuanto mejor escalen las Versiones mas se acrecentara la reducción relativa, o aumento relativo, entre las mismas a medida que crezca la cantidad de copias, pero como aquí no escalan tan distinto, las reducciones o aumentos no cambian demasiado.

6.3.3. Caso de prueba en dos dimensiones

Ahora bien, al trabajar únicamente con una sola dimensión, no se está aprovechando completamente las optimizaciones implementadas tanto en los conjuntos como en los *piecewise maps* ordenados. Es por ello que a

	Total time en milisegundos			
Cantidad de copias	**Versión 0**	**Versión 1**	**Versión 2**	**Versión 3**
1	34	19	25	17
2	64	53	53	49
4	175	113	137	124
8	607	417	491	390
16	3919	2153	2405	2072
32	12558	8381	11547	6835
64	70707	53609	46506	39112

Cuadro 6.10: Cuadro comparativo del *Total time* de las diferentes versiones planteadas bajo un caso de prueba en una dimensión, variando el tamaño del caso de prueba

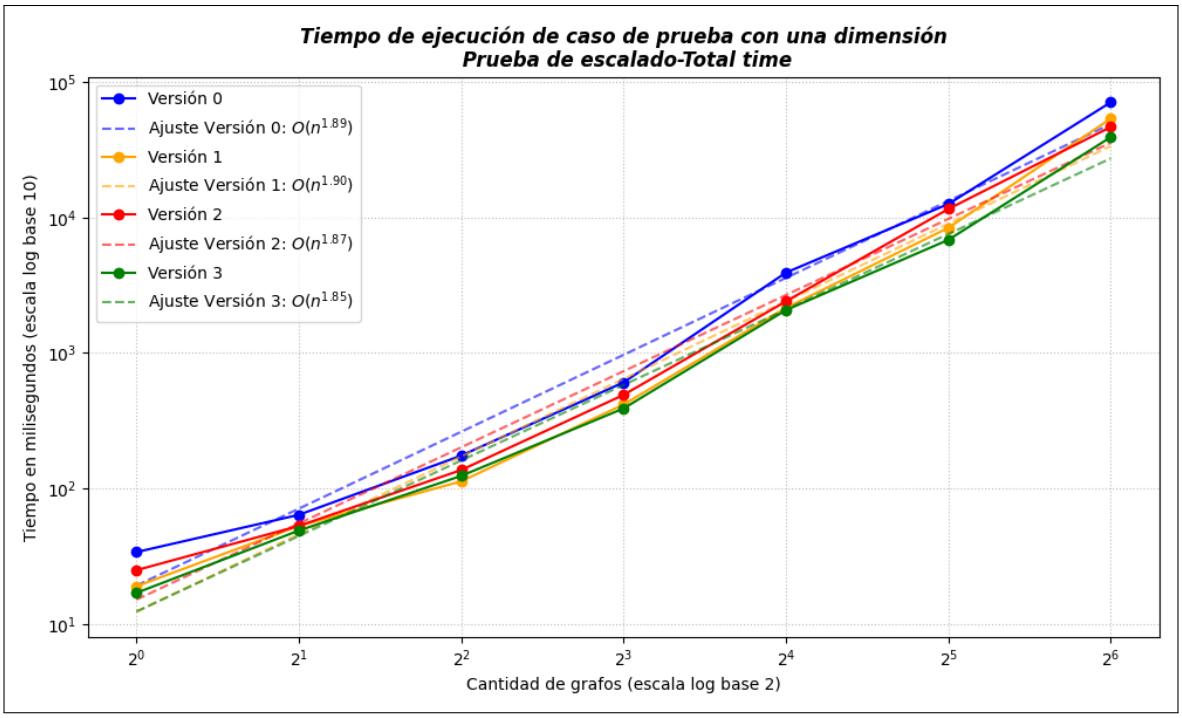


Figura 6.31: Gráfico comparativo del *Total time* de las diferentes versiones planteadas bajo un caso de prueba en una dimensión, variando el tamaño del caso de prueba.

continuación se presentará un nuevo caso de prueba, esta vez en dos dimensiones. El código correspondiente se encuentra disponible en el archivo *pw_map_test_2dim.test*, ubicado en la carpeta *test* dentro del repositorio.

Cabe destacar que, en esta ocasión, no se dispone del valor correspondiente a *Total topological sort exec time*, por lo que dicho tiempo de ejecución será omitido tanto en el análisis como en el cálculo del *Total time*.

En la Figura 6.32 se presentan los diferentes tiempos de ejecución analizados en este caso para todas las versiones, con excepción de la Versión 1, la cual solo es válida en el contexto unidimensional. Nuevamente los valores resultan el promedio de múltiples ejecuciones para cada una de las versiones. A partir de este gráfico, pueden destacarse las siguientes observaciones:

- Las versiones que incorporan orden en su implementación resultan más eficientes que la Versión 0, especialmente en lo que respecta al *Total match exec time*. En particular, se observa una reducción relativa del **30.70 %** y **43.74 %** para las Versiones 2 y 3, respectivamente.
- De forma similar, se evidencia una disminución en el *Total SCC exec time*, con reducciones relativas del **29.96 %** y **38.87 %** para las Versiones 2 y 3 en comparación con la Versión 0.
- En cuanto al *Total time*, se registra una mejora general con reducciones relativas del **30.61 %** y **42.87 %** también para las Versiones 2 y 3 respecto a la Versión 0.

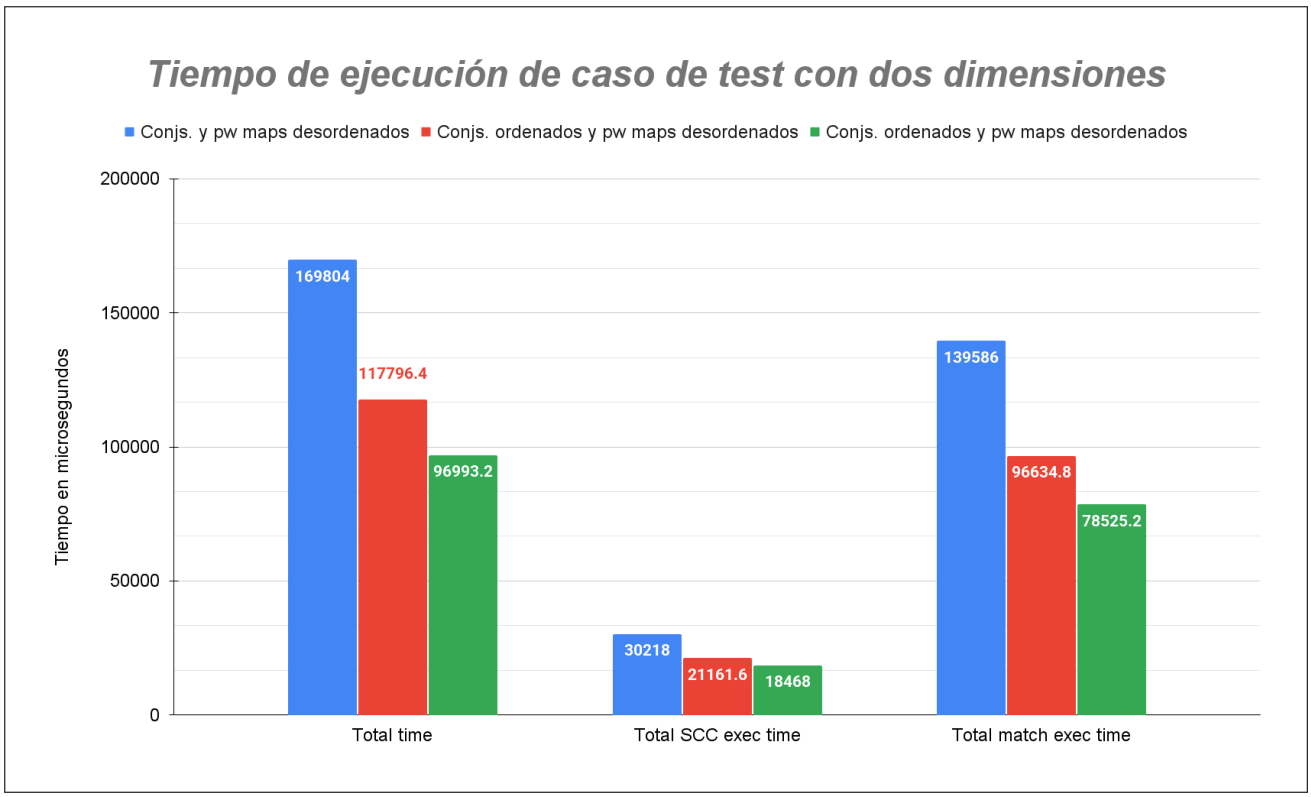


Figura 6.32: Comparación de los tiempos de ejecución de las diferentes versiones planteadas bajo un caso de prueba en dos dimensiones.

6.3.4. Escalado - Caso de prueba en dos dimensiones

Nuevamente se chequeará cómo es que escalan los tiempos de ejecución a medida de que el grafo crece en tamaño. A continuación, se analizará cada una de las medidas por separado.

6.3.4.1. Escalado del *Total Match exec time*

Como se observa en el Cuadro 6.11 y en la Figura 6.33, en esta ocasión el tiempo de ejecución presenta un comportamiento que escala de forma cuadrática o incluso peor en el caso de la Versión 0. En cambio, las Versiones 2 y 3 muestran un crecimiento subcuadrático, destacándose la Versión 3 como la más eficiente.

En particular, para el caso más simple con una sola copia del grafo original, se observa una reducción relativa del **31.86 %** y **34.51 %** en las Versiones 2 y 3, respectivamente, en comparación con la Versión 0. Esta mejora se vuelve aún más significativa a medida que crece el grafo, ya que con 32 copias, la reducción alcanza el **76.54 %** para la Versión 2 y el **81.86 %** para la Versión 3, siendo **4.26** veces y **5.51** veces mas rápidas respectivamente que la Versión 0.

	***Total Match exec time* en milisegundos**		
Cantidad de copias	**Versión 0**	**Versión 2**	**Versión 3**
1	113	77	74
2	389	241	200
4	1549	802	663
8	7842	2951	2127
16	31561	11212	9066
32	184108	43177	33401

Cuadro 6.11: Cuadro comparativo del *Total Match exec time* de las diferentes versiones planteadas bajo un caso de prueba en dos dimensiones, variando el tamaño del caso de prueba

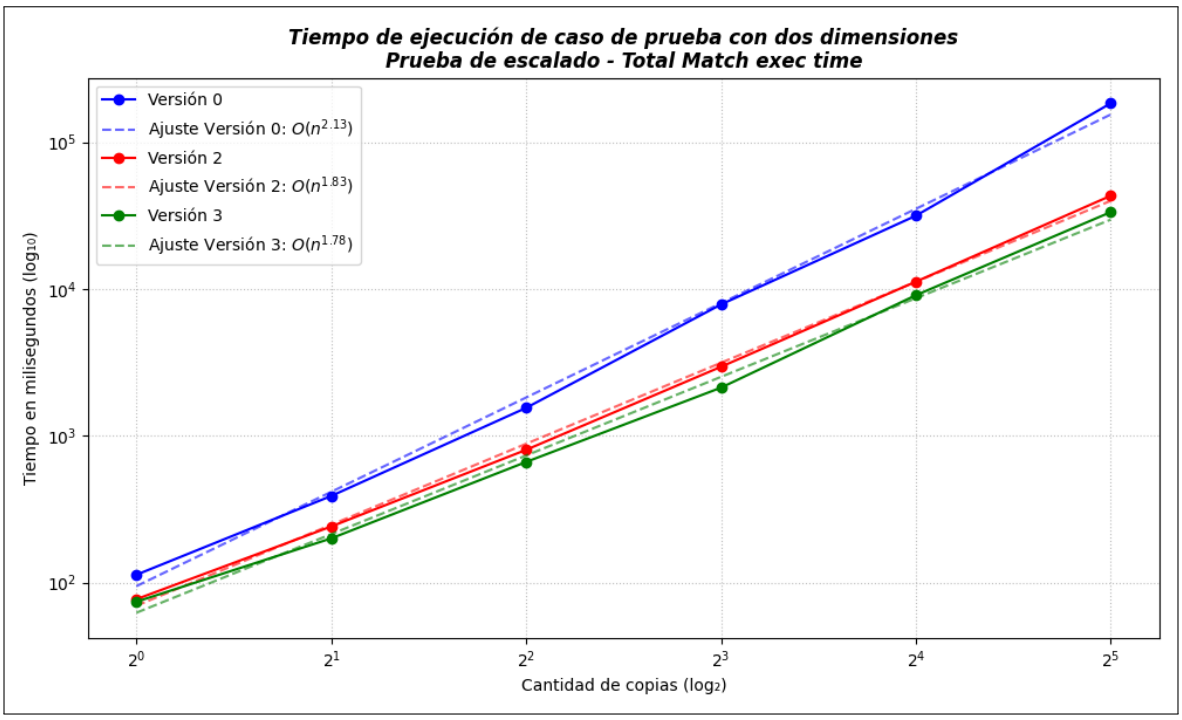


Figura 6.33: Gráfico comparativo del *Total Match exec time* de las diferentes versiones planteadas bajo un caso de prueba en dos dimensiones, variando el tamaño del caso de prueba.

6.3.4.2. Escalado del *Total SCC exec time*

En esta ocasión, en la Figura 6.34 y en el Cuadro 6.12, se observa una situación particular. Aunque nuevamente el escalado general parece similar al visto en el *Total Match exec time*, en este caso no es la Versión 3 la que presenta el mejor escalabilidad, sino la Versión 2.

Esta diferencia podría explicarse por el tipo de operaciones utilizadas en la implementación con *piecewise maps* ordenados, las cuales podrían estar introduciendo una sobrecarga adicional que afecta el tiempo de ejecución, a pesar de las mejoras estructurales de dicha versión.

	***Total SCC exec time* en milisegundos**		
Cantidad de copias	**Versión 0**	**Versión 2**	**Versión 3**
1	25	17	19
2	64	45	51
4	477	144	302
8	1413	429	477
16	3597	896	1462
32	20215	3435	3939

Cuadro 6.12: Cuadro comparativo del *Total SCC exec time* de las diferentes versiones planteadas bajo un caso de prueba en dos dimensiones, variando el tamaño del caso de prueba

6.3.4.3. Escalado del *Total time*

Como era de esperarse en base a los resultados anteriores, en el Cuadro 6.13 y en la Figura 6.35 se puede evidenciar que se repite el mismo patrón observado en el análisis de escalado del *Total Match exec time*. En esta oportunidad, la diferencia relativa respecto a la Versión 0 alcanza un **77.19 %** en el caso de la Versión 2 y un **81.73 %** para la Versión 3 con el máximo de cantidad de copias del grafo original, confirmando nuevamente la eficacia de las optimizaciones implementadas. En este caso, dado que la diferencia de escalado con respecto a la Versión 0 es considerablemente mayor, al aumentar la cantidad de copias estas diferencias tenderían a incrementarse aún más. Lo mismo ocurre al comparar la Versión 2 con la Versión 3, donde también se espera

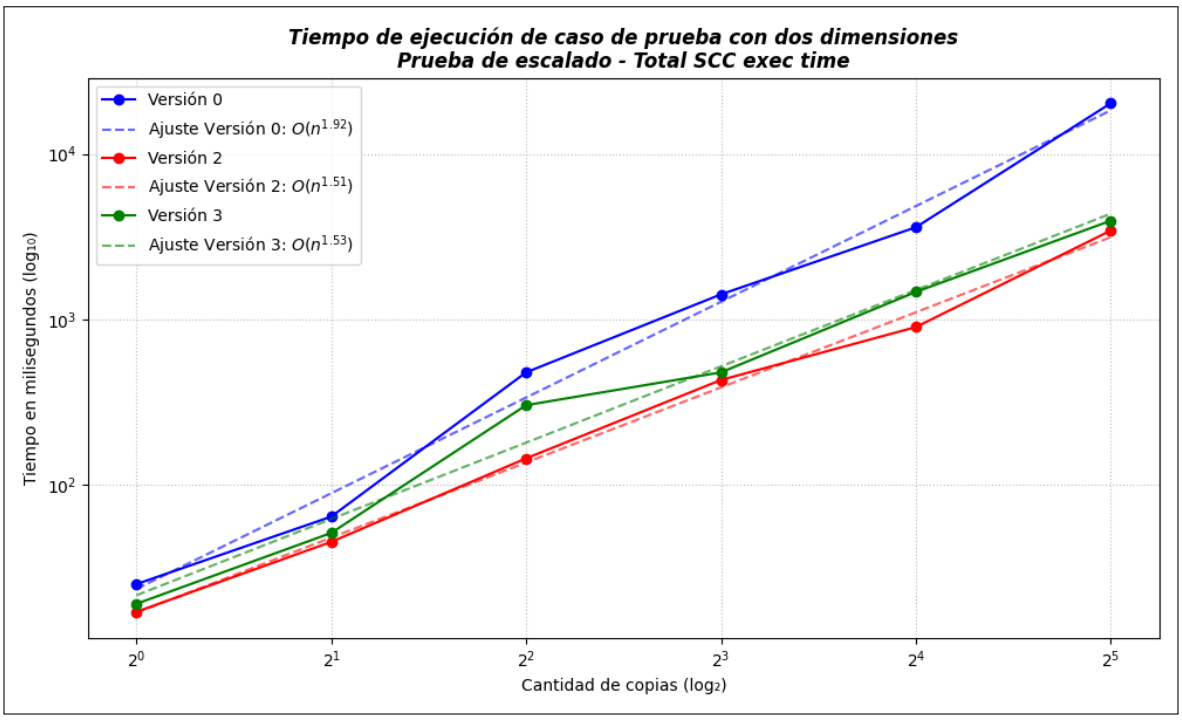


Figura 6.34: Gráfico comparativo del *Total SCC exec time* de las diferentes versiones planteadas bajo un caso de prueba en dos dimensiones, variando el tamaño del caso de prueba.

que la reducción relativa se amplíe.

	***Total time* en milisegundos**		
Cantidad de copias	**Versión 0**	**Versión 2**	**Versión 3**
1	138	94	93
2	453	286	251
4	2026	946	965
8	9255	3380	2604
16	35158	12108	10528
32	204323	46612	37340

Cuadro 6.13: Cuadro comparativo del *Total time* de las diferentes versiones planteadas bajo un caso de prueba en dos dimensiones, variando el tamaño del caso de prueba

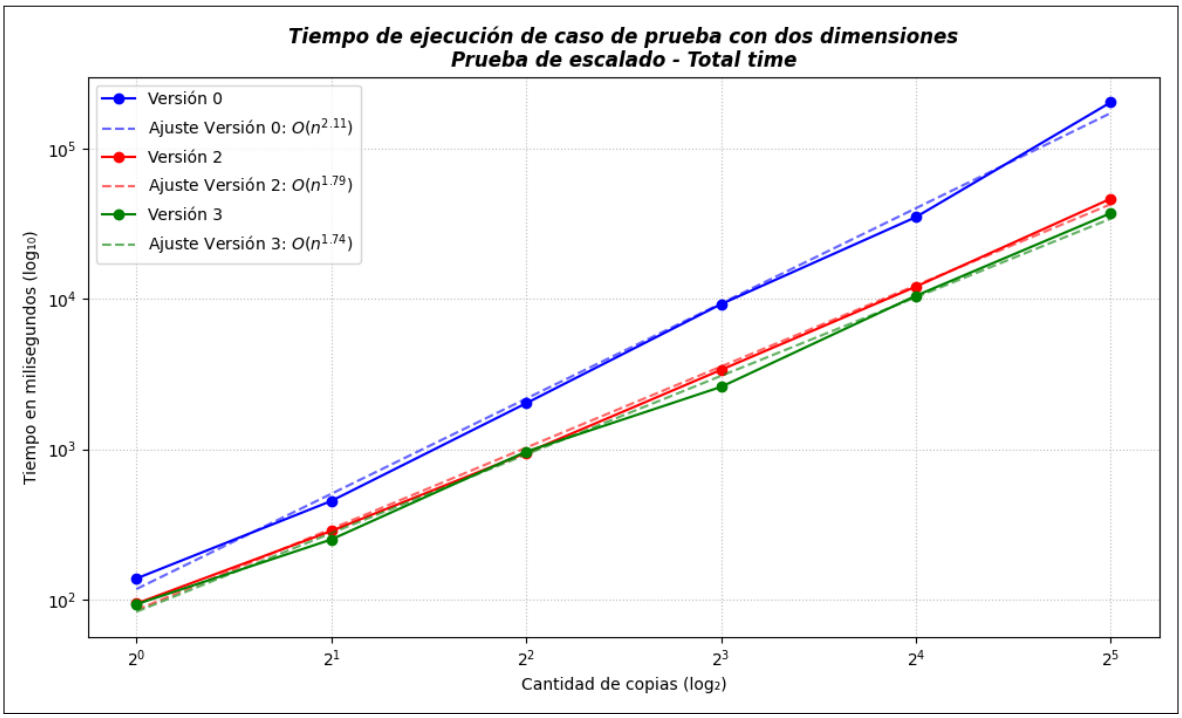


Figura 6.35: Gráfico comparativo del *Total time* de las diferentes versiones planteadas bajo un caso de prueba en dos dimensiones, variando el tamaño del caso de prueba.

Capítulo 7

Conclusiones finales y trabajos futuros

Conclusiones finales

Durante el desarrollo de esta tesina se obtuvieron los siguientes resultados:

- Se logró con éxito construir una implementación de conjuntos ordenados capaz de manejar múltiples dimensiones y paso igual o distinto a uno. Además, esta presenta un rendimiento superior al de la implementación de conjuntos desordenados, especialmente cuando se utilizan más de una dimensión como se vio en los casos de prueba sintéticos y en los casos de prueba.

  Sin embargo, en comparación con la implementación de conjuntos ordenados densos, el rendimiento de la presente implementación en los casos de prueba sintéticos con una única dimensión y paso igual a uno resulta notablemente inferior en ciertos casos, como en las operaciones de complemento o unión, mientras que, por ejemplo, en la intersección el rendimiento es muy similar.

- Se consiguió desarrollar una implementación de *piecewise maps* ordenados, constituyendo así la primera implementación de *piecewise maps* que aprovecha el orden para mejorar su desempeño.

  Además, los resultados de los casos de prueba muestran que esta implementación presenta un rendimiento superior al de los *piecewise maps* desordenados, aunque la mejora no es tan marcada como la observada en la implementación de conjuntos ordenados frente a los conjuntos desordenados. Cabe destacar que las pruebas se realizaron utilizando los *piecewise maps* desordenados optimizados, y no la versión original con la que se comenzó el desarrollo de esta tesina.

Por ende, en conclusión, se lograron cumplir los objetivos planteados para esta tesina, incluso alcanzando las mejoras de rendimiento esperadas. No obstante, hacia el final del desarrollo ciertas cuestiones, la cuales, resultan lo suficientemente relevantes como para ser enunciadas, pueden utilizarse para el desarrollo de nuevos trabajos en el futuro. Estas serán ampliadas en la siguiente sección dedicada a los trabajos futuros.

Trabajos futuros

En este trabajo se presentaron, en un capítulo propio, un conjunto de optimizaciones aplicadas a ciertas operaciones de los *piecewise maps* desordenados. Estas surgieron durante el desarrollo de la implementación ordenada, ya que se partió de la versión desordenada para construir esta última. Sin embargo, estas no son las únicas optimizaciones que pueden realizarse sobre los *piecewise maps* desordenados, ni significa que, por no haberse presentado ninguna para los conjuntos desordenados, no existan optimizaciones posibles para estos últimos.

En particular, los diferentes criterios de solapamiento enunciados a lo largo de este trabajo no requieren del orden para funcionar correctamente. Por ende, los criterios aplicados en las operaciones de conjuntos ordenados también podrían emplearse en conjuntos desordenados; de igual forma, en el caso de los *piecewise maps*, los criterios definidos para los ordenados podrían aplicarse a los desordenados.

Cabe destacar que, además de utilizar estos criterios en las implementaciones desordenadas, también sería posible trasladarlos a estructuras de jerarquía inferior. Es decir, el criterio de solapamiento utilizado en conjuntos podría desplazarse a la implementación de multi-intervalos; y, de manera análoga, los utilizados a nivel de *piecewise maps* podrían trasladarse a conjuntos.

No obstante, esta migración debería realizarse con cautela, con el objetivo de no disminuir el rendimiento de ninguna de las implementaciones que harían uso de estos criterios.

Apéndice A: Notación

Notación		
Hipótesis	**Simbología**	**Significado**
a, b y c son naturales	$[a : b : c]$	Intervalo con inicio a, paso b y fin c.
	`NAT`	Sinónimo de `unsigned long long` en C++, representa a los naturales. Siempre que se hable de naturales en el código se hace referencia a este tipo.
	`Interval`	Tipo que representa a los intervalos en C++. Siempre que se hable de intervalos en el código se hace referencia a este tipo.
	`Inf`	Valor máximo disponible para los naturales que es representado por `NAT`.
	$[1 : 1 : 0]$	Notación para el intervalo vació.
$i_0 \times i_2 \times \dots i_{k-1}$ son intervalos	$i_0 \times i_2 \times \dots i_{k-1}$	Multi-intervalo de k dimensiones, es decir, con k intervalos $i_0, i_2, \dots, i_{k-1}$.
	`MultiDimInter/SetPiece`	Tipos que representan a los multi-intervalos en C++. Siempre que se hable de multi-intervalos en el código se hace referencia al primero en cuanto a las operaciones de los multi-intervalos y al segundo en todas las demás ocasiones.
	`MD_NAT`	Sinónimo de `vector<NAT>` en C++, representa a los naturales multi-dimensionales. Siempre que se hable de naturales multi-dimensionales en el código se hace referencia a este tipo.
	$\{\}$	Notación para el conjunto vació.
	`SetDelegate`	Tipo abstracto en C++, representa una interfaz común para todas las implementaciones concretas de conjuntos para aplicar el patrón de diseño *delegate*.
	`Set`	Tipo que permite manipular conjuntos en `C++`. Contiene alguna de las implementaciones concretas de conjuntos, lo que posibilita trabajar de manera independiente de la implementación específica utilizada, aplicando así el patrón de diseño *Delegate*.
mdi es un multi-intervalo e i esta entre 0 y k con k siendo la cantidad de dimensiones de mdi	$mdi[i]$	Es el intervalo de la dimensión i.
	$\emptyset_{mdi}$	Representa el multi-intervalo vacío, es decir, sin ningún intervalo.
A es un conjunto o *piecewise map*	$kappa(A)$	Representa el numero de multi-intervalos del conjunto A o de mapas del *piecewise map* A respectivamente.
a, b, y c objetos en C++ y $metod$ es un método de a	$metod(a, b, c)$	Representa `a.metod(b,c)` en C++, la instancia a que invoca al método es el primer argumento.
a, y b objetos en C++ y # es un método/operador de a	$a\#b$	Representa `a.operador(b)` en C++.
A es un conjunto e i esta entre 0 y $\kappa(A) - 1$	A_i	Es el i-ésimo multi-intervalo de A, corresponde a `pieces_[i]` en C++.
i y j son dos números naturales tal que $i < j$ y A es un conjunto	$A_{i:j}$	Es el subconjunto de $A = \{A_i, A_{i+1}, \dots, A_j\}$.
i y j son dos números naturales tal que $i > j$ y A es un conjunto	$A_{i:j}$	Es el subconjunto de $A = \{\}$.
A es un conjunto	$A_{0:-1}$	Es el subconjunto de $A = \{\}$.
A es un *piecewise map* desordenado e i esta entre 0 y $\kappa(A) - 1$	A_i	Es el i-ésimo mapa de A, corresponde a `pieces_[i]` en C++.
i es un intervalo y k es un natural	i^k	Es un multi-intervalo de dimensión k.
A, B conjuntos o *piecewise maps*	$A \frown B$	Devuelve un nuevo objeto que contiene primero los elementos de A seguidos por los de B, es decir, una concatenación ordenada: $A_0, A_1, \dots, B_0, B_1, \dots$.
	$\{\}$	Es un conjunto vacío, es decir, sin multi-intervalos.
	$\ll\gg$	Es un *piecewise map* vació, es decir, sin mapas.
	$[]$	Es una lista simplemente enlazada.
l y l' son listas simplemente enlazadas	$l +\!\!+ l'$	Se concatenan las listas simplemente enlazadas, primero se colocan los elementos de l y luego de l'.
$list$ es una lista simplemente enlazada y i es un elemento dentro de la lista	$list \triangleleft i$	Elimina el elemento i-ésimo de la lista en cuestión.
a es un número natural	$(a)^d$	Es un natural multi-dimensional de longitud d o de d dimensiones donde en cada una vale a.
a y b valores naturales multi-dimensionales	$\llcorner a, b \urcorner$	Es un perímetro que es en sí un par de elementos. En este caso el perímetro contiene a a y b.
m es un mapa y p un perímetro	$\lfloor m, p \rceil$	Es una entrada de mapa que es en sí un par de elementos. En este caso la entrada de mapa contiene a m y p.

Notación		
Hipótesis	**Simbología**	**Significado**
A es un *piecewise map* ordenado e i está entre 0 y $\kappa(A) - 1$	A_i	Siendo el capitulo ***piecewise maps* ordenados**, dentro de la sección: ▪ **Criterios de optimización y ordenamientos**: Es el i-ésimo mapa de A, corresponde a `pieces_[i].first` en C++. ▪ **Implementaciones adicionales/Implementaciones**: Es la i-ésima entrada de mapa de A, corresponde a `pieces_[i]` en C++.
i y j son dos números naturales tal que $i < j$ y A es un *piecewise map* ordenado	$A_{i:j}$	Es el *piecewise map* ordenado $\ll A_i, A_{i+1}, \ldots, A_j \gg$.
i y j son dos números naturales tal que $i > j$ y A es un *piecewise map* ordenado	$A_{i:j}$	Es el *piecewise map* ordenado vacío $\ll\gg$.
A es un *piecewise map* ordenado	$A_{0:-1}$	Es el *piecewise map* ordenado vacío $\ll\gg$.
e es una expresión lineal y d un número natural	e^d	Es una expresión lineal d dimensional o de d dimensiones tal que en todas sus dimensiones tiene la expresión e, tal que $[e_0, e_1, \ldots, e_{d-1}]$.

Bibliografía

[1] Federico Bergero, Mariano Botta, and Ernesto Kofman. Efficient compilation of large scale modelica models. In *11th International Modelica Conference*, 2015.

[2] CIFASIS. CIFASIS/sb-graph. `https://github.com/CIFASIS/sb-graph/tree/sb-graph-dev`, 2025. Repositorio en GitHub.

[3] Mike Dempsey. Dymola for multi-engineering modelling and simulation. In *2006 IEEE Vehicle Power and Propulsion Conference*, pages 1–6. IEEE, 2006.

[4] Peter Fritzson and Peter Bunus. Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings 35th Annual Simulation Symposium. SS 2002*, pages 365–380. IEEE, 2002.

[5] Peter Fritzson and Vadim Engelson. Modelica—a unified object-oriented language for system modeling and simulation. In *European Conference on Object-Oriented Programming*, pages 67–90. Springer, 1998.

[6] Peter Fritzson, Adrian Pop, Karim Abdelhak, Adeel Asghar, Bernhard Bachmann, Willi Braun, Daniel Bouskela, Robert Braun, Lena Buffoni, Francesco Casella, et al. The openmodelica integrated environment for modeling, simulation, and model-based development. *Modeling, Identification and Control*, 41(4):241–295, 2020.

[7] Denise Marzorati, Joaquin Fernández, and Ernesto Kofman. Efficient connection processing in equation–based object–oriented models. *Applied Mathematics and Computation*, 418:126842, 2022.

[8] Denise Marzorati, Joaquín Fernández, and Ernesto Kofman. Efficient matching in large dae models. *ACM Transactions on Mathematical Software*, 2024.

[9] Denise Marzorati, Joaqun Fernandez, and Ernesto Kofman. Aplanado eficiente de grandes sistemas de ecuaciones algebraico-diferenciales. In *FCEIA-ECEN-DCC- Trabajos Finales de Grado (trabajos finales, proyectos y tesinas)*, 2020.

[10] Kirill Rozhdestvensky, Vladimir Ryzhov, Tatiana Fedorova, Kirill Safronov, Nikita Tryaskin, Shaharin Anwar Sulaiman, Mark Ovinis, and Suhaimi Hassan. Description of the wolfram systemmodeler. In *Computer Modeling and Simulation of Dynamic Systems Using Wolfram SystemModeler*, pages 23–87. Springer, 2020.

[11] Pablo Zimmermann, Joaquín Fernández, and Ernesto Kofman. Set-based graph methods for fast equation sorting in large dae systems. In *Proceedings of the 9th International Workshop on Equation-based Object-oriented Modeling Languages and Tools*, pages 45–54, 2019.