Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Aplanado eficiente de grandes sistemas de ecuaciones algebraico-diferenciales

Denise Marzorati

Director: Ernesto Kofman

Co-director: Joaquín Fernandez

15 de diciembre de 2020

Agradecimientos

A Dios primeramente; cuando parecía que no avanzaba, me ayudó de maneras que no esperaba. Después a mi familia: Hugo, Viviana y Martín, que me aguantaron muchas veces de mal humor cuando me frustraba, y siempre me alentaron a terminar. También quiero reconocer a los sobresalientes docentes que transmitieron no solo conocimiento, sino también el deseo de aprender con excelencia.

Resumen

En esta tesis se describe la formulación e implementación de nuevos algoritmos para convertir modelos orientados a objetos de gran escala en sistemas de ecuaciones. Estos algoritmos están principalmente basados en la teoría de grafos y tienen la propiedad de lograr un costo computacional constante con respecto al tamaño de los arreglos involucrados. Para esto, hacen uso de una nueva categoría de grafos denominados "Grafos Basados en Conjuntos", que permiten representar y manipular conjuntos de vértices y aristas de manera compacta.

La tesis muestra, en primer lugar, el desarrollo y la programación de una librería para representar y realizar operaciones sobre grafos basados en conjuntos. Luego, usando esta librería, se desarrolla e implementa un algoritmo para encontrar componentes conexas, que forma parte de la primera etapa de conversión de modelos en sistemas de ecuaciones.

Finalmente, se analizan distintos ejemplos que muestran la eficiencia de los nuevos algoritmos.

Índice general

Capítulo 1

Introducción

1.1. Motivación y planteo del problema

Diversas ramas de la ciencia y de la técnica utilizan modelos matemáticos de los sistemas que estudian. Sobre estos modelos matemáticos se realizan simulaciones para analizar el comportamiento de dichos sistemas. Dado que estas técnicas son aplicadas a sistemas cada vez más complejos y grandes, permanentemente surgen nuevos desafíos.

Los sistemas dinámicos continuos suelen ser representados mediante sistemas de ecuaciones diferenciales algebraicas (EDAs), y su simulación requiere de la resolución numérica de dichas ecuaciones, que puede hacerse a través de algoritmos específicos conocidos como solvers.

Los solvers requieren que los sistemas de ecuaciones estén escritos y ordenados de cierta forma, que generalmente no coincide con la manera en la que un/una especialista describe los modelos. Para convertir los modelos desde una representación orientada a objetos, tal como la usada por los lenguajes de modelado modernos, en una representación de ecuaciones ordenadas y estructuradas, como la requieren los solvers, se recurre a diversos algoritmos que conforman un "compilador de modelos". Estos algoritmos están basados, en su mayor parte, en la Teoría de Grafos.

La primera etapa de este proceso se denomina aplanado y requiere encontrar las componentes conexas de un grafo. Si bien existen algoritmos que resuelven este problema, los mismos se tornan computacionalmente muy costosos en sistemas de gran escala. Esto se debe a que *bloques de ecuaciones* son expandidos, lo que conlleva que el costo de los algoritmos implicados sea dependiente del tamaño de dichos bloques. Para salvar este problema, se apunta a explotar la presencia de estructuras repetitivas en dichos sistemas. Bajo ciertas condiciones, dichos bloques podrán trabajarse de manera compacta, permitiendo la aplicación de nuevos algoritmos cuyo costo computacional no es dependiente del tamaño de dichos bloques.

1.2. Estado del arte

Encontrar las componentes conexas de un grafo no dirigido es un problema clásico de la Teoría de Grafos, que es aplicado en diversos dominios. Se conocen varios algoritmos simples que resuelven este problema en tiempo lineal desde hace décadas [14]. Aún más, existen algoritmos paralelos que pueden resolver el problema en tiempo logarítmico, que datan del siglo pasado [13].

En particular, la etapa de aplanado de ecuaciones de modelos orientados a objetos [11] es un problema que puede ser resuelto encontrando componentes conexas en un grafo. Allí, los diferentes sub-modelos están relacionados por *conectores*, y las conexiones deben ser reemplazadas por ecuaciones. Aunque puede ser aceptable aplanar en tiempo lineal para ciertos casos, hay modelos que

son el resultado de acoplar miles de pequeños sub-modelos, y el costo se torna excesivo. Aún si el problema es resuelto en tiempo razonable, puede suceder que sistema de ecuaciones resultante sea tan grande que no pueda ser tratado en las siguientes etapas de compilación.

Afortunadamente, la mayoría de modelos de gran escala contienen muchas conexiones repetitivas, que son el resultado de usar bucles `for`, y esto puede ser aprovechado para reducir el costo computacional de las distintas etapas de compilación [3, 16, 9, 5, 18, 7, 15, 1, 17]. Sin embargo, la posibilidad de explotar la presencia dichas estructuras repetitivas en cada etapa requiere que las etapas anteriores hayan preservado una representación compacta. Aunque existen algunas implementaciones experimentales para casos de modelos particulares que preservan la representación compacta durante todo el proceso de compilación [5], no existe aún una solución general.

En particular para la etapa de aplanado, una solución general requiere encontrar conjuntos de conectores conexos, que pueden pertenecer a arreglos multi-dimensionales, resolviendo el problema sin expandir los arreglos en conectores individuales. Este problema es equivalente a encontrar componentes conexas en un grafo no dirigido, manteniendo ciertos vértices y ciertas aristas agrupadas, que constituye el objetivo de este trabajo.

La solución al problema de manipular grandes grafos agrupando vértices y aristas en conjuntos para producir sistemas de ecuaciones compactos fue recientemente propuesta con la introducción de *Grafos Basados en Conjuntos* [20]. Allí se pensó e implementó una solución compacta para los problemas de apareamiento y componentes fuertemente conexas en un grafo dirigido, para ordenar ecuaciones, como parte del compilador ModelicaCC [5, 19].

En este trabajo se usará la misma herramienta (Grafos Basados en Conjuntos), y se propondrá una solución general del algoritmo para encontrar componentes conexas en grafos no dirigidos. Se demostrará que, bajo ciertas condiciones, el costo computacional del algoritmo es independiente del tamaño de los vértices y las aristas. En consecuencia, el costo de la etapa de aplanado es independiente del tamaño de los arreglos de conectores.

Además de introducir y analizar el algoritmo, también se describe la implementación que se realizó en C++, incluida en ModelicaCC. Se concluye presentando algunos ejemplos para demostrar la eficiencia del nuevo enfoque.

1.3. Organización de la Tesina

En el capítulo 2 se presenta el problema de componentes conexas en un grafo no dirigido, y cómo este puede ser aprovechado para resolver el aplanado de modelos orientados a objetos. Por otro lado, se presenta el nuevo enfoque de Grafos Basados en Conjuntos, los cuales reemplazarán a los grafos no dirigidos en la etapa de aplanado, para disminuir el costo computacional.

En el capítulo 3 se describe un nuevo algoritmo que encuentra componentes conexas en un Grafo Basado en Conjuntos, para luego analizar el costo computacional del mismo. Para facilitar la comprensión, previamente se introducirá un algoritmo de componentes conexas para grafos no dirigidos, en el cual está basado el nuevo algoritmo.

En el capítulo 4 se detalla la implementación de una librería para Grafos Basados en Conjuntos, así como también la implementación del algoritmo de componentes conexas para los mismos.

En el capítulo 5 se desarrolla explícitamente cómo se usarán los Grafos Basados en Conjuntos en la etapa de aplanado de ecuaciones.

En el capítulo 6 se presentan ejemplos, y los resultados obtenidos de usar estas novedosas herramientas.

Capítulo 2

Conceptos previos

En este capítulo se presentarán los principales conceptos en los que se basan los temas desarrollados en la tesina. Aquí se terminará de comprender la motivación para hallar un nuevo enfoque.

2.1. Modelica

En un esfuerzo para unificar los distintos lenguajes de modelado, usados por las distintas herramientas de modelado y simulación, un consorcio de compañías de software, y grupos de investigación, propuso un lenguaje de modelado abierto, unificado, orientado a objetos, llamado *Modelica* [11, 10], que en las dos últimas décadas fue progresivamente adoptado por distintas herramientas de modelado y simulación.

Modelica permite la representación de sistemas de tiempo continuo, tiempo discreto, eventos discretos e híbridos. Modelos elementales de Modelica son descriptos mediante conjuntos de ecuaciones diferenciales y algebraicas, que pueden ser combinadas con algoritmos para describir evoluciones discretas. Estos modelos elementales pueden ser conectados a otros modelos, facilitando la construcción de modelos más complejos.

Los modelos Modelica pueden ser construidos y simulados usando distintas herramientas de software. OpenModelica [12] es la herramienta de código abierto más usada, mientras Dymola [8] y Wolfram System Modeler dominan en el ámbito comercial. Existen también algunas herramientas prototipo orientadas a distintos problemas, tal como JModelica [2] (para problemas de optimización), y ModelicaCC.

2.1.1. Compilación de Modelica

La simulación de modelos Modelica requiere que el código sea previamente compilado, de manera que el modelo orientado a objetos sea traducido a algún lenguaje de programación (usualmente C). El programa resultante debe contener un conjunto de ecuaciones diferenciales ordinarias (EDO), o ecuaciones diferenciales algebraicas (EDA), que pueden ser resueltas por los correspondientes solvers. El proceso de compilación se suele dividir en varias sub-etapas: aplanado, remoción de alias, reducción de índice, ordenamiento de ecuaciones, y generación de código.

Al iniciar el trabajo de esta tesina, el objetivo era enfocarse en una de las etapas finales de compilación: ordenamiento de ecuaciones. Sin embargo, como se ha mencionado anteriormente, es necesario que todas las etapas preserven la representación compacta. Además, es factible aplicar el enfoque de Grafos Basados en Conjuntos a las etapas posteriores, por lo que se ahorraría buena parte del trabajo en el futuro. Por estos motivos se decidió comenzar por la etapa de aplanado.

La semántica del lenguaje se ha especificado a través de reglas que traducen el código Modelica de cualquier modelo, a Modelica plano. Fue diseñado para facilitar la transformación simbólica de modelos, a través del mapeo de cada construcción del lenguaje a ecuaciones continuas, o instantáneas, que pertenecen a la estructura de Modelica plano. Como resultado, el usuario puede construir modelos fácilmente, disponiendo varias opciones, y luego el compilador lo traduce a Modelica plano. Este proceso de traducción se conoce como aplanado.

2.1.2. Clases

En esta sección se introducen de manera breve las principales construcciones de Modelica, junto con algunos ejemplos, que son necesarias explicar para las secciones posteriores. Si es necesario consultar la sintaxis formal completa, ver [4].

La unidad estructural fundamental en Modelica son las *clases*. Las clases proveen de estructura a los objetos, más bien conocidos como *instancias*. Las clases pueden contener ecuaciones, que proveen la base para el código ejecutable. Además de las ecuaciones, también se permite la existencia de código algorítmico "convencional", como poseen otros lenguajes. Todos los objetos en Modelica son instancias de una clase.

A continuación se presenta la estructura típica de una clase de Modelica:

```
class ClassName
  Declaration1
  Declaration2
  ...
equation
  equation1
  equation2
  ...
end ClassName;
```

En términos generales toda clase Modelica posee un nombre, una sección de declaración (de variables, constantes, etc.), y una sección de ecuaciones.

A su vez, las clases pueden especializarse; es decir, que además de contar con las propiedades generales de las clases, se agregan propiedades adicionales conocidas como "mejoras". La idea es describir de manera más precisa la función de la clase.

En particular, destacamos la clase *connector*, que será la clave para componer modelos para lograr modelos complejos. Al referirse a varias instancias de esta clase se hablará de *conectores*.

Si bien no trabajamos con el resto de las especializaciones, las listaremos, ya que las mismas pueden aparecer en los ejemplos de código que serán presentados: record, type, model, block, function, package, operator record, operator, operator function. Por otro lado, también mencionamos las clases predefinadas de Modelica que son: Real Type, Integer Type, Boolean Type, Enumeration Types, Clock Types, y algunas más que son pocos usadas.

Se presenta un ejemplo concreto para poder explicar la terminología que se usará a lo largo de este material. Este describe el comportamiento de un condensador ideal.

```
connector Pin
  Real v;
  flow Real i;
end Pin;
```

Listado 2.1: Conector de componentes eléctricos

En el mismo se muestra una clase llamada Pin (especialización: connector), que cuenta con dos *variables*, ambas de *tipo* Real, cuyos *identificadores* son "v" e "i" respectivamente. Dichas variables hacen referencia a la corriente y al voltaje.

Además, la variable "i" tiene un *modificador*, flow, que indica que "i" es una variable de flujo. A nivel de Modelica, este modificador indica que hay ciertas restricciones que deben cumplirse si una *instancia* de Pin interactúa con otra instancia de una clase connector.

```
1 model Capacitor
2   parameter Real C;
3   Pin p, n;
4   Real u;
5 equation
6   0 = p.i + n.i;
7   u = p.v - n.v;
8   C * der(u) = p.i;
9 end Capacitor;
```

Listado 2.2: Modelo de un capacitor ideal

Se ha presentado la clase Capacitor (especialización: model), que cuenta con un *parámetro*, "C" (que no varía en tiempo de ejecución), dos instancias del connector Pin, "p" y "n" (positivo y negativo), y una variable Real, "u" (diferencia de potencial). Luego de las declaraciones, hay tres ecuaciones, que describen el comportamiento del modelo. La primera es una expresión de las leyes de Kirchoff, la segunda es la misma definición de diferencia de potencial, y la tercera describe el comportamiento de un capacitor ideal.

En las tres ecuaciones se utiliza el operador ".", que permite el acceso a los miembros de una clase; de este modo "p.i" referencia a la variable "i", de la instancia de Pin, "p". Por otro lado, en la línea 9, se utiliza el operador "der", que modela la derivada del argumento con respecto al tiempo.

2.1.3. Conexiones Modelica

En los lenguajes de modelado orientados a objetos se hace una distinción entre distintos tipos de variables:

- Variables de esfuerzo: son aquellas que en una conexión se tornan iguales. Por ejemplo, en un mismo punto todas las temperaturas son iguales, todas las velocidades son iguales, etc.
- Variables de flujo: son aquellas que suman cero en una conexión. Por ejemplo, en un punto todas las fuerzas suman cero, todas las corrientes suman cero, etc.

Para poder modelar la interacción entre distintos conectores (o sub-componentes de los mismos), Modelica provee ecuaciones *connect*. Estas ecuaciones dictan el comportamiento de las variables de esfuerzo y de flujo.

La construcción connect no pertenece a Modelica plano, por lo cual es necesario transformar los modelos que la usen. A esta sub-etapa del proceso de aplanado se la denomina *generación de ecuaciones de conexión*.

Un *conjunto conexo* es un conjunto de variables conectadas a través de connects. Un conjunto conexo solo debe contener variables de flujo, o solo variables de esfuerzo. Dado un modelo Modelica, esta fase desea devolver el conjunto conexo de variables de esfuerzo, y del mismo modo, el conjunto conexo de variables de flujo.

Continuando con el ejemplo que desarrollamos en la sección anterior, se modela un circuito de dos capacitores en paralelo, con una fuente de tensión:

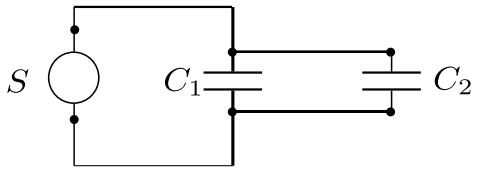


Imagen 2.1: *Capacitores en paralelo*

```
1  model SignalVoltage
2    Pin p, n;
3    parameter Real V;
4    Real i;
5  equation
6    V = p.v - n.v;
7    0 = p.i + n.i;
8    i = p.i;
9  end SignalVoltage
```

Listado 2.3: Modelo de fuente de tensión

```
1  model Circuit
2    Capacitor a, b;
3    SignalVoltage sv;
4  equation
5    connect(sv.n, a.n);
6    connect(sv.p, a.p);
7    connect(a.n, b.n);
8    connect(a.p, b.p);
9  end Circuit;
```

Listado 2.4: Modelo del circuito

No es trivial hallar los conjuntos conexos, pues se deben encontrar las conexiones transitivas. Por ejemplo, sv.n.i con b.n.i están conectadas de manera transitiva. En definitiva, para Circuit los conjuntos conexos resultantes son:

- Flujo: {{sv.n.i, a.n.i, b.n.i}, {sv.p.i, a.p.i, b.p.i}}
- Esfuerzo: {{sv.n.v, a.n.v, b.n.v}, {sv.p.v, a.p.v, b.p.v}}

Una vez que se han obtenido los conjuntos conexos, se pueden generar las ecuaciones que corresponden a cada connect. Para cada conjunto conexo de variables de flujo $c^{i}_{flow} = \{v^{k_1}_{flow}, ..., v^{k_m}_{flow}\}$, y para cada conjunto conexo de variables de esfuerzo $c^{m}_{eff} = \{v^{n_1}_{eff}, ..., v^{n_p}_{eff}\}$, generamos las siguientes ecuaciones:

- $v^{n_1}_{eff} = v^{n_2}_{eff} = ... = v^{n_{p-1}}_{eff} = v^{n_p}_{eff}$
- $v^{k_1}_{flow} + v^{k_2}_{flow} + ... + v^{k_{m-1}}_{flow} + v^{k_m}_{flow} = \Phi$

donde Φ es el cero, o un arreglo de ceros, dependiendo de la dimensión de las variables. Nuevamente, para el ejemplo presentado, los connect de Circuit serán reemplazados por:

```
1  model Circuit
2    Capacitor a, b;
3    SignalVoltage sv;
4  equation
5    // Esfuerzo
6    sv.n.i = a.n.i;
7    a.n.i = b.n.i;
8    sv.p.i = a.p.i;
9    a.p.i = b.p.i;
10
11   // Flujo
12   sv.n.v + a.n.v + b.n.v = 0;
13   sv.p.v + a.p.v + b.p.v = 0;
14 end Circuit;
```

Listado 2.5: Modelo del circuito sin connect

2.1.4. Grafos y conexiones

Definición 1 (Grafo).
Un grafo es un par G = (V, E), donde:

- V = {v_1, v_2, ..., v_i}, que se denomina conjunto de vértices o nodos.
- E = {e_1, e_2, ..., e_j}, donde e_k = {v_a, v_b} / 1 ≤ k ≤ j ∧ v_a, v_b ∈ V. Dicho conjunto es llamado conjunto de aristas.

Definición 2 (Subgrafo).
Se dice que S = (V_S, E_S) es un subgrafo de G = (V, E) si V_S ⊆ V ∧ E_S ⊆ E.

Definición 3 (Camino).
Dado un grafo G = (V, E), un camino es una secuencia ordenada de aristas de la forma c = ({v_1, v_2}, {v_2, v_3}, ..., {v_{n-2}, v_{n-1}}, {v_{n-1}, v_n}), donde $v_i \neq v_j$, 1 ≤ i, j ≤ n, i ≠ j.

Definición 4 (Componente conexa).
Una componente conexa C de un grafo G = (V, E), es un subgrafo C = (V_S, E_S) de G, para el cual se cumple que, dados v_a, v_b ∈ V_S cualesquiera, existe un camino en C que los une.

Se pretende poder estructurar el modelo con sus conexiones como un grafo, y en vez de utilizar el algoritmo de generación de ecuaciones, utilizar un algoritmo de componentes conexas. Este proceso es bastante intuitivo al observar las etapas descriptas:

1. Se representa cada variable como un nodo.
2. Por cada connect se agrega una arista entre las variables implicadas de los argumentos.
3. Se buscan las componentes conexas del grafo.

Si bien se podría buscar resolver este problema de manera directa, el enfoque de Grafos Basados en Conjuntos tiene el objetivo de proveer herramientas para las etapas posteriores de compilación; e incluso podría resultar útil para otros problemas.

2.1.5. Arreglos y bucles

Hasta el momento no se han presentado ejemplos que expliquen la necesidad de trabajar con un nuevo enfoque para los grafos. En esta sección se introducen los arreglos y bucles en Modelica, que son útiles para modelar grandes sistemas con estructuras repetitivas.

Un arreglo es una colección de valores, todos del mismo tipo. Cada arreglo tiene una cierta dimensionalidad. La cantidad de dimensiones de un arreglo es fija, y no puede ser cambiada en tiempo de ejecución.

En Modelica los arreglos son "rectangulares"; formalmente, dado un arreglo arr[s_1][s_2]...[s_n] de dimensión n, el arreglo cuenta con s_1 * s_2 * ... * s_n elementos, donde el índice de la i-ésima dimension puede tomar los valores j con $1 \leq j \leq s_i$.

Un arreglo se asigna en memoria declarando una variable, o bien llamando al constructor de arreglos. Los elementos del mismo pueden ser indexados por valores *Integer*, *Boolean* o *enumeration*.

En conjunto con esta construcción Modelica permite declarar ecuaciones For donde se puede tratar con arreglos de variables de manera intensiva. A continuación se presenta un ejemplo de red de resistores y capacitores (se omite la definición de algunos modelos intermedios ya que no son relevantes para lo que se busca explicar):

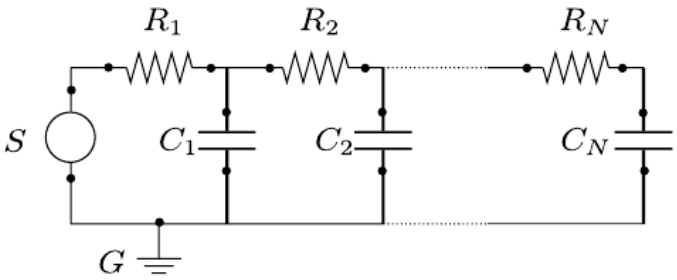


Imagen 2.2: *Red de resistores y capacitores*

```
1  model RCNetwork
2    Real N = 1000;
3    SignalVoltage S;
4    Ground G;
5    Resistor R[N];
6    Capacitor C[N];
7  equation
8    connect(S.p, R[1].p);
9    connect(S.n, G.p);
10
11   for i in 1:N-1 loop
12     connect(R[i].n, R[i + 1].p);
13   end for;
14
15   for i in 1:N loop
16     connect(C[i].p, R[i].n);
17     connect(C[i].n, G.p);
18   end for;
19 end RCNetwork;
```

Listado 2.6: Red de capacitores y resistencias

Los casos presentados anteriormente de conexiones en Modelica son bastante simples, pero al introducir estas construcciones es cuando se comprende la necesidad de trabajar bajo un nuevo enfoque. Hasta donde alcanza el conocimiento de este grupo, todos los compiladores reemplazan estos bucles por la descripción extensiva de todas las ecuaciones. Para el caso presentado, introduciría al menos 999 ecuaciones para el primer bucle, y al menos 2000 ecuaciones más para el segundo bucle.

Basta ver que en el ejemplo presentado, al incrementar N, el sistema de ecuaciones se vuelve inmanejable para las etapas posteriores (por ejemplo, N=100000, o mayor). En contraposición, el enfoque de Grafos Basados en Conjuntos busca tratar con cada bucle sin desenrollarlo.

2.1.6. ModelicaCC

ModelicaCC es un compilador de Modelica implementado en C++, creado para implementar y testear nuevos algoritmos de compilación de modelos Modelica. Las implementaciones que se presentarán en capítulos posteriores fueron hechas en esta herramienta.

En particular, la etapa de aplanado en este compilador se subdivide en dos etapas:

- **Instanciación de clases**. Este problema ya fue tratado en el compilador, sin perder la representación compacta [6], por lo que no forma parte de esta tesina.
- **Generación de ecuaciones planas**. Esta sub-etapa es que la que se encarga de eliminar los connect, tal como se ha detallado en la sección 2.1.3, siendo la motivación de este trabajo.

2.2. Algoritmos para componentes conexas

Para encontrar las componentes conexas de un grafo no dirigido existen múltiples algoritmos, ya que es un problema relativamente sencillo.

El modo más directo para hacerlo consiste en realizar una búsqueda a lo ancho o en profundidad en el grafo [14]. Se puede iterar a través de cada uno de sus vértices, haciendo una búsqueda en cada uno de ellos, agregando un nuevo componente cada vez que se trabaje con un nodo que no pertenece a ninguno de los componentes ya encontrados. Como puede observarse, al recorrer cada vértice, el costo dependerá de la cantidad de vértices del grafo.

Por otro lado, también se han propuesto algoritmos paralelos. En particular, el nuevo algoritmo que se presentará aquí tiene mucho en común con uno de ellos [13]. Este algoritmo busca agrupar vértices adyacentes en "super-vértices", y en sucesivas iteraciones, combinar los super-vértices, hasta que los mismos no presenten cambios. Luego, cada super-vértice contendrá todos los nodos de una única componente conexa. El costo del mismo es logarítmico.

La clave del algoritmo se encuentra en cómo representa los super-vértices: utiliza un arreglo de tamaño n (cantidad de vértices del grafo), donde D(i) es igual al vértice con menor numeración que pertenece al mismo super-vértice en el cual i está incluido. Así, cada super-vértice será representado por el menor vértice que lo compone.

A continuación se presentará una versión del mismo, ya que el nuevo algoritmo que se presentará en esta tesina tiene mucho en común, y puede facilitar la comprensión de este último. Para ver la versión original, con el análisis del costo consultar [13].

Algoritmo 1 Componentes conexas - Algoritmo paralelo [13]

1: **function** CONNECT(V, E) ▷ Todos los pasos se ejecutan en paralelo $\forall\ i \in V$
2: D(i)← i, $\forall$ i $\in$ V
3: C(i)← i, $\forall$ i $\in$ V

```
4:    for it_1 = 1:log_2(n) do
5:        minadj_i ← min_j(D(j) | {C(i), D(j)} ∈ E ∧ D(j) ≠ D(i)), ∀ i ∈ V
6:        if minadj_i = none then                          ▷ No hay super-vértices adyacentes
7:            C(i) ← D(i)
8:        else
9:            C(i) ← minadj_i                              ▷ Hay super-vértices adyacentes
10:       end if
11:       newrep_i ← min_j(C(j) | D(j) = i ∧ C(j) ≠ i), ∀ i ∈ V
12:       if newrep_i = none then          ▷ Ambos super-vértices han sido completamente combinados
13:           C(i) ← D(i)
14:       else
15:           C(i) ← newrep_i
16:       end if
17:       D(i) ← C(i), ∀ i ∈ V.
18:       for it_2 = 1:log_2(n) do
19:           C(i) ← C(C(i)), ∀ i ∈ V.
20:       end for
21:       D(i) ← min(C(i), D(C(i))), ∀ i ∈ V.
22:   end for
23:   return D
24: end function
```

Este algoritmo funciona del siguiente modo: como se ha mencionado, se irá modificando D, para encontrar todos los representantes del grafo. Inicialmente, en la línea 2, cada vértice se representará a sí mismo. En la línea 3 se define otro arreglo, C, que será un arreglo auxiliar, para ir calculando representantes "parciales", a partir de los representantes adyacentes (a partir de los cuales podrá definirse D). Básicamente C se utiliza para identificar los representantes de cada componente conexa ya identificada, y D para ir combinando los representantes de dichas componentes conexas.

En las líneas 5 a 16 se chequean si hay super-vértices adyacentes (es decir, existe una arista entre dos de ellos). Como se ha mencionado, C identifica componentes conexas, que deben ser combinadas en una única componente conexa. Es por eso que en la línea 3 se busca el menor representante de alguna componente adyacente. Luego, en la línea 11, para cada representante de una componente conexa, se actualiza su representante, al hallado entre las componentes conexas adyacentes (en caso de que estas existan).

Luego, en las líneas 19 a 21 se combinan los super-vértices que correspondan. En las líneas 18 a 21 se actualiza el representante de cada la componente conexa por completo (recordar que en la sección previa solo se actualizaron los representates), según corresponda.

2.3. Grafos Basados en Conjuntos

Los Grafos Basados en Conjuntos (GBC) fueron introducidos en [20], y tienen por objeto representar de manera compacta grafos de gran tamaño. A continuación se brinda la definición formal.

Definición 5 (Vértice-conjunto).
Un vértice-conjunto (*set-vertex*) es un conjunto de vértices $V = \{v_1, v_2, ..., v_n\}$.

Definición 6 (Arista-conjunto).
Dados dos vértices-conjunto V^a y V^b, con $V^a \cap V^b = \emptyset$, una arista-conjunto (*set-edge*) que une V^a con V^b es un conjunto de aristas no repetidas $E[\{V^a, V^b\}] = \{e_1, e_2, ..., e_n\}$ donde cada arista es un conjunto de dos vértices, $e_i = \{v_k^a \in V^a, v_l^b \in V^b\}$ donde $1 \leq i \leq n$.

Una manera simple de representar aristas-conjuntos es la que se propone a continuación (notar que la definición no impone esto, sino que es una representación conveniente para trabajar con estos elementos).

Se dirá que un mapa es una colección no ordenada de pares finita, donde el segundo valor de cada par representa el valor asignado al primer elemento del par. Dado el mapa map = $\{(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)\}$, se usará la notación $map(a_i) = b_i$ para indicar que b_i es el resultado de aplicar el mapa map a a_i.

Se define $map^{a,b}$ como un mapa, cuyo dominio serán los elementos de una arista-conjunto $E^a \in \mathcal{G}$, y cuya imagen estará contenida en $V^b \in \mathcal{V}$.

Sea $E^h = [\{V^i, V^j\}]$. Se puede caracterizar esta arista-conjunto usando dos mapas que relacionan cada arista "común" $e^h_k = \{v^i_r, v^j_s\} \in E^h$ con los vértices que une, $v^i_r = map^{h,i}(e^h_k)$ y $v^j_s = map^{h,j}(e^h_k)$. De este modo, la arista-conjunto queda definida por comprensión como:

$$E^h = \bigcup_{k=1}^{n} \{\{map^{h,i}(e^h_k), map^{h,j}(e^h_k)\}\}$$

Si bien la definición de E^h es compacta, lo que puede suceder es que la definición de $map^{h,i}$ y $map^{h,j}$ puede no ser compacta. Eso dependerá del grafo que se esté representando.

Definición 7 (Grafo Basado en Conjuntos).
Un GBC (*set-based graph*) es un par $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ donde:

- $\mathcal{V} = \{V^1, ..., V^n\}$ es un conjunto de vértices-conjunto disjuntos (es decir, $i \neq j \Rightarrow V^i \cap V^j = \emptyset$).
- $\mathcal{E} = \{E^1, ..., E^m\}$ es un conjunto de aristas-conjunto que une vértices-conjunto de $\mathcal{V}$, es decir, $E^i = E[\{V^a, V^b\}]$ con $V^a \in \mathcal{V}$ y $V^b \in \mathcal{V}$. Además, dadas dos aristas-conjunto $E^i, E^j \in \mathcal{E}$ con $i \neq j$, de modo que $E^i = E[\{V^a, V^b\}]$ y $E^j = E[\{V^c, V^d\}]$, se cumple $V^a \cup V^b \cup V^c \cup V^d \neq V^a \cup V^b$. Resumiendo, dos aristas-conjunto diferentes en $\mathcal{E}$ no pueden conectar los mismos vértices-conjunto.

Se presenta un ejemplo gráfico, $\mathcal{G} = (\mathcal{V}, \mathcal{E}) = (\{V^1, V^2, V^3, V^4\}, \{E^1, E^2, E^3\})$. En vez de agregar una figura para señalar las aristas-conjunto, se las puede distinguir por color:

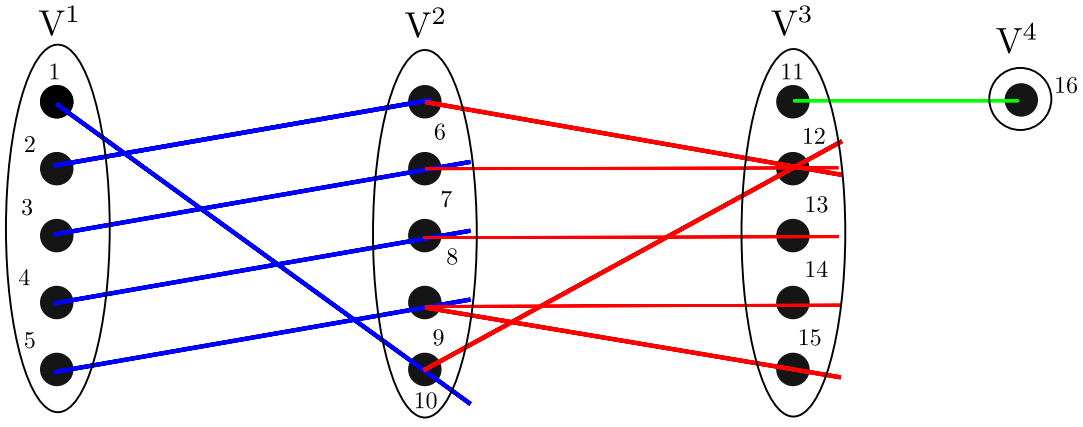


Imagen 2.3: *Ejemplo de GBC*

Primero, se asignarán nombres a las aristas:

- $e_1^1 = \{1, 10\}$
- $e_2^1 = \{2, 6\}$
- $e_3^1 = \{3, 7\}$
- $e_4^1 = \{4, 8\}$
- $e_5^1 = \{5, 9\}$
- $e_1^2 = \{6, 12\}$
- $e_2^2 = \{7, 12\}$
- $e_3^2 = \{8, 13\}$
- $e_4^2 = \{9, 14\}$
- $e_5^2 = \{9, 15\}$
- $e_6^2 = \{10, 12\}$
- $e_1^3 = \{11, 16\}$

Entonces, el GBC es:

- $V^1 = \{1, 2, 3, 4, 5\}$
- $V^2 = \{6, 7, 8, 9, 10\}$
- $V^3 = \{11, 12, 13, 14, 15\}$
- $V^4 = \{16\}$
- $E^1 = \{e_1^1, e_2^1, e_3^1, e_4^1, e_5^1\}$
- $E^2 = \{e_1^2, e_2^2, e_3^2, e_4^2, e_5^2, e_6^2\}$
- $E^3 = \{e_1^3\}$

Utilizando la representación de mapas:

- $map^{1,1} = \{(e_1^1, 1), (e_2^1, 2), (e_3^1, 3), (e_4^1, 4), (e_5^1, 5)\}$
- $map^{1,2} = \{(e_1^1, 10), (e_2^1, 6), (e_3^1, 7), (e_4^1, 8), (e_5^1, 9)\}$
- $map^{2,2} = \{(e_1^2, 6), (e_2^2, 7), (e_3^2, 8), (e_4^2, 9), (e_5^2, 9), (e_6^2, 10)\}$
- $map^{2,3} = \{(e_1^2, 12), (e_2^2, 12), (e_3^2, 13), (e_4^2, 14), (e_5^2, 15), (e_6^2, 12)\}$
- $map^{3,3} = \{(e_1^3, 11)\}$
- $map^{3,4} = \{(e_1^3, 16)\}$

Definición 8 (Grafo tradicional definido por un GBC).
Un grafo (puede ser dirigido) basado en conjuntos $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ con $\mathcal{V} = \{V^1, ..., V^n\}$, y $\mathcal{E} = \{E^1, ..., E^m\}$, define un grafo tradicional (dirigido), G = (V, E) donde:

$$V = \bigcup_{i=1}^{n} V^i; E = \bigcup_{i=1}^{m} E^i$$

Un GBC define un grafo tradicional equivalente, donde el conjunto de vértices y aristas del segundo es la unión de los vértices-conjunto y aristas-conjunto del primero. De esta manera, un grafo GBC contiene tanta información como un grafo tradicional.

Gráficamente, si se transforma el GBC presentado como ejemplo en la definición anterior:

Si se puede definir intensivamente las aristas-conjunto y los vértices-conjunto, se tendrá una representación compacta del grafo. Esto permitiría, bajo ciertas condiciones, que el costo computacional de los algoritmos para GBCs sea independiente del tamaño de los vértices-conjunto y aristas-conjunto, permitiendo trabajar con grandes estructuras.

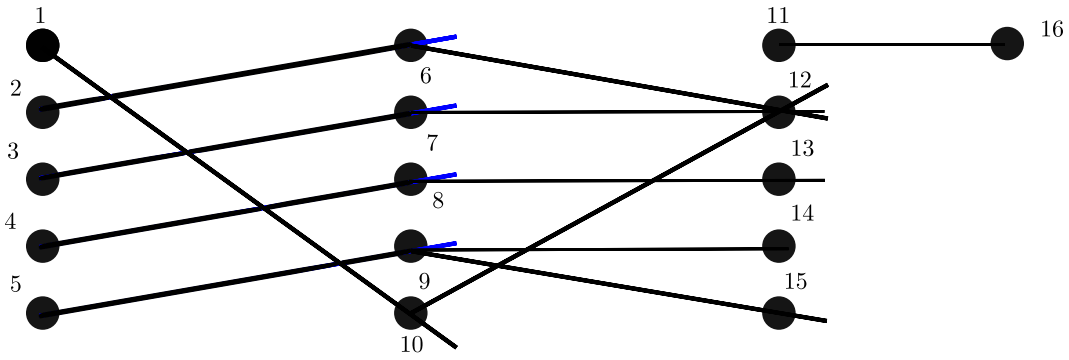


Imagen 2.4: *Ejemplo de grafo tradicional definido por un GBC*

Definición 9 (Grafo bipartito basado en conjuntos).
Un grafo bipartito basado en conjuntos es un GBC $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ donde se pueden hallar dos conjuntos disjuntos de vértices-conjunto $\mathcal{V}_1$, $\mathcal{V}_2$ tales que $\mathcal{V}_1 \cup \mathcal{V}_2 = \mathcal{V}$, de manera que para toda arista-conjunto $\mathrm{E}^k = \mathrm{E}[\{\mathrm{V}^a, \mathrm{V}^b\}] \in \mathcal{E}$, $\mathrm{V}^a \in \mathcal{V}_i \Rightarrow \mathrm{V}^b \notin \mathcal{V}_i$.

Definición 10 (Arista-conjunto dirigida).
Dados dos vértices-conjunto V^a y V^b tales que $\mathrm{V}^a \cap \mathrm{V}^b = \emptyset$, o $\mathrm{V}^a = \mathrm{V}^b$, una arista-conjunto dirigida desde V^a a V^b es un conjunto de aristas no repetidas $\mathrm{E}[(\mathrm{V}^a, \mathrm{V}^b)] = \{\mathrm{e}_1, \mathrm{e}_2, ..., \mathrm{e}_n\}$ donde cada arista es un par ordenado de vertices $\mathrm{e}_i = (\mathrm{v}^a_k \in \mathrm{V}^a, \mathrm{v}^b_l \in \mathrm{V}^b)$.

Definición 11 (Grafo dirigido basado en conjuntos).
Un grafo dirigido basado en conjuntos es un par $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ donde:

- $\mathcal{V} = \{\mathrm{V}^1, ..., \mathrm{V}^n\}$ es un conjunto de vértices-conjunto disjunto.
- $\mathcal{E} = \{\mathrm{E}^1, ..., \mathrm{E}^m\}$ es un conjunto de aristas-conjunto dirigidas que conectan vértices-conjunto de $\mathcal{V}$, es decir, $\mathrm{E}^i = \mathrm{E}[(\mathrm{V}^a, \mathrm{V}^b)]$, donde $\mathrm{V}^a \in \mathcal{V}$ y $\mathrm{V}^b \in \mathcal{V}$. Además, dadas dos aristas-conjuntos $\mathrm{E}^i, \mathrm{E}^j \in \mathcal{E}$ con i ≠ j, de manera que $\mathrm{E}^i = \mathrm{E}[(\mathrm{V}^a, \mathrm{V}^b)]$ y $\mathrm{E}^j = \mathrm{E}[(\mathrm{V}^c, \mathrm{V}^d)]$, entonces $\mathrm{V}^a \neq \mathrm{V}^c$ o $\mathrm{V}^b \neq \mathrm{V}^d$. Es decir, dos aristas-conjunto en $\mathcal{E}$ no puede conectar los mismos vértices-conjunto con la misma dirección.

Debe notarse que se han provisto ciertas definiciones que no se utilizarán en este trabajo, ya que se pretende aprovecharlas para etapas posteriores (por ejemplo, la de grafo basado en conjuntos dirigido).

Capítulo 3

Algoritmo de componentes conexas

Se presentará un algoritmo para hallar conjuntos de componentes conexas en un GBC; aunque previamente se mostrará una versión escalar, para facilitar la comprensión. Finalmente, se concluirá analizando el costo computacional del nuevo algoritmo.

3.1. Algoritmo de componentes conexas escalar

El algoritmo aquí presentado está basado en el trabajo de [13]. En particular:

- Se supone que existe un orden total entre los vértices (es decir, pueden ser representados por enteros o bien arreglos de enteros).
- Cada componente conexa es representado por uno de los vértices que lo componen, $v_k \in V$, que es el menor vértice del mismo.
- Existe un mapa $R_{map} : V \to V$, tal que $R_{map}(v_r) = v_k$ implica que $v_r \in V$ es parte del componente conexa representado por v_k.
- Como el representante de $R_{map}(v_r)$ es el mínimo vértice de la componente conexa, entonces $R_{map}(v_r) \leq v_r$, para todo $v_r \in V$.

Con esta representación, el algoritmo presentado a continuación devuelve un mapa de vértices a representates, R_{map}, que contiene toda la información acerca de los componentes conexas de un grafo $G = (V, E)$:

Algoritmo 2 Componentes conexas - Versión escalar

```
1:  function CONNECT(V, E)
2:      R_map ← Identity_map : V → V        ▷ Inicialmente todos los vértices se encuentran desconectados
3:      I_old ← ∅                                                          ▷ Imagen previa de D_map
4:      while I_old ≠ Image(R_map) do
5:          C_map ← R_map                                                  ▷ Nuevo mapa de componentes conexas
6:          for all v_r ∈ Image(R_map) do                                  ▷ Componente representado por v_r
7:              if ∃ {v_r, v_s} ∈ E then                                   ▷ v_r tiene vértices adyacentes
8:                  v_k ← min(R_map(v_b) : ({v_a, v_b} ∈ E ∧ R_map(v_a) = v_r))   ▷ Mínimo vértice conectado al
    componente representado por v_r
9:                  if v_k < v_r then
10:                     C_map(v_r) ← v_k          ▷ Conexión de componentes representados por v_r y v_k
```

11: $C_{map}(v_a) \leftarrow (C_{map} \circ C_{map})(v_a) = C_{map}(v_r) = v_k$ for all $v_a : C_{map}(v_a) = v_r$ ▷ Los vértices representados por v_r ahora serán representados por v_k
12: **end if**
13: **end if**
14: **end for**
15: $I_{old} \leftarrow \text{Image}(R_{map})$ ▷ Representantes de los componentes conexas anteriores
16: $R_{map} \leftarrow C_{map}$ ▷ Nuevas componentes conexas
17: **end while**
18: **return** R_{map}
19: **end function**

El algoritmo funciona del siguiente modo: comienza asumiendo que todos los vértices se encuentran desconectados, así que cada uno representa su propia componente conexa. Luego, itera hasta que la imagen de R_{map} se vuelve constante, lo que significa que no se pueden conectar más componentes.

En cada iteración se calcula un nuevo C_{map}, agregando conexiones entre componentes. Para cada componente representado por v_r, el algoritmo toma en cuenta todas las aristas que conectan vértices de este componente. De entre todas esas aristas, se queda con la que conecta a un cierto nodo v_b, con el menor representate $v_k = R_{map}(v_b)$ (podría pasar que $v_k = v_r$ si no hay conexión desde el componente representado por v_r a cualquiera de los componentes representados por un vértice menor). En ese caso, si el representate v_k es menor a v_r, el algoritmo conecta ambos componentes haciendo que $C_{map}(v_r) = v_k$. Además, reconecta todos los vértices conectados a v_r, de manera que ahora estén representados por v_k.

Aunque es sencillo probar que el procedimiento es correcto, es probablemente uno de los algoritmos menos eficientes para encontrar componentes conexas en un grafo. El costo computacional parece depender de manera cuadrática del número de vértices y aristas del grafo. Sin embargo, en el contexto de GBCs, este algoritmo puede ser implementado de manera que los costos sean independientes del tamaño de los distintos conjuntos implicados.

Un elemento clave del algoritmo presentado es que, en cada iteración, C_{map} se computa como una función del mapa R_{map}, y viceversa. De este modo, ambos mapas pueden ser computados a partir del otro en pocos pasos.

3.2. Algoritmo de componentes conexas para GBCs

A continuación presentaremos la versión basada en conjuntos del algoritmo 2. Consideraremos como entrada un GBC $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, y obtendremos un mapa R_{map}, que indica a qué componente conexa pertenece cada vértice del grafo.

Algoritmo 3 Algoritmo de componentes conexas para GBCs

1: **function** ConnectSBG($\mathcal{V}$, $\mathcal{E}$)
2: $V \leftarrow \bigcup V^i : V^i \in \mathcal{V}$
3: $(E^1_{map}, E^2_{map}) \leftarrow \text{edgeMaps}(\mathcal{E})$
4: $R_{map} \leftarrow \text{id} : V \to V$ ▷ Mapeo de vértices a representantes
5: $I_{old} \leftarrow \emptyset$
6: **while** $I_{old} \neq \text{Im}(R_{map})$ **do**
7: $ER^1_{map} \leftarrow R_{map} \circ E^1_{map}$ ▷ Mapeo de los vértices izquierdos a sus representantes
8: $ER^2_{map} \leftarrow R_{map} \circ E^2_{map}$ ▷ Mapeo de los vértices derechos a sus representates
9: $RR^1_{map} \leftarrow \text{MinAdjMap}(ER^1_{map}, ER^2_{map})$ ▷ Mapeo de los vértices izquierdos al mínimo representante adyacente del lado derecho

10: $\text{RR}^2_{map} \leftarrow$ MinAdjMap(ER^2_{map}, ER^1_{map}) ▷ Mapeo de los vértices derechos al mínimo representante adyacente del lado izquierdo
11: $\tilde{R}_{map} \leftarrow$ MinMap(R_{map}, MinMap(RR^1_{map}, RR^2_{map}))
12: $\text{I}_{old} \leftarrow$ Im(R_{map})
13: $\text{R}_{map} \leftarrow (\tilde{R}_{map})^{\infty}$
14: **end while**
15: **return** R_{map}
16: **end function**

En este nuevo algoritmo se ha hecho uso de las siguientes funciones y notación:

- La función edgeMaps($\mathcal{E}$) devuelve dos mapas: un mapa de conexiones izquierdas, $\text{E}^1_{map} : \text{E} \to \text{V}$, y un mapa de conexiones derechas, $\text{E}^2_{map} : \text{E} \to \text{V}$, definidos de la siguiente manera. Para cada arista conjunto $\text{E}^h \in \mathcal{E}$, que conecta los vértices-conjunto V^i y V^j, se satisface:

$$E^1_{map}(e^h_k) = map^{h,i}(e^h_k), \forall e^h_k \in E^h$$

$$E^2_{map}(e^h_k) = map^{h,j}(e^h_k), \forall e^h_k \in E^h$$

  Se debe notar que para arista-conjunto hay dos posibles definiciones de E^1_{map} y E^2_{map}, dependiendo de cuál se asocie a i, y cuál a j (ya que las aristas-conjunto son no dirigidas).

- La función Im calcula la imagen del mapa que recibe como argumento.

- La función MinMap, calcula el mapa mínimo entre ambos mapas que recibe como argumento. Naturalmente la operación se encuentra definida para la intersección de ambos dominios.

- La función MinAdjMap(map_1, map_2) calcula un mapa map_3 tal que:

$$map_3(v) = min(map_2(e) : map_1(e) = v)$$

  En el contexto del algoritmo, al usarlo en las líneas 9 y 10, v es un vértice representante de alguna componente conexa (ya que la imagen ER^1_{map} y ER^2_{map} está conformada por representantes), y e es una arista. Entonces, para todas las aristas tales que $\text{map}_1(\text{e}) = \text{v}$, la función elige la arista para la cual $\text{map}_2(\text{e})$ es mínimo, y define $\text{map}_3(\text{v}) = \text{map}_2(\text{e})$. De este modo, $\text{map}_3(\text{v})$ es el mínimo representante conectado a otro vértice representado por v, a través de map_2.

- $(\tilde{R}_{map})^{\infty}$ es notación que se utiliza para referise al resultado de aplicar $\tilde{R}_{map}$ a sí mismo hasta alcanzar un punto fijo.

El algoritmo de esta sección comparte ciertas ideas comunes con aquel de la sección anterior. La mayor diferencia quizás radica en la iteración de $\tilde{R}_{map}$ en sí mismo, que es realizada al final del ciclo. La convergencia se encuentra asegurada ya que $\tilde{R}_{map}$ siempre será menor o igual a la identidad (todo representante es menor o igual que cualquiera de los vértices a los cuales representa), y su dominio es finito.

3.3. Análisis del costo computacional

Se dirá que dos componentes conexas están conectadas si, y solo si, su representante es el mismo. Por otro lado, se hablará de componente conexa más extensa a aquella que posea el camino más extenso, es decir, el camino con mayor cantidad de aristas.

Teorema: *la cantidad de iteraciones necesarias para hallar todas las componentes conexas es O(2 · log₂(N)), donde N es la cantidad de aristas en la componente conexa más extensa.*

Demostración

Supóngase que después de q iteraciones, una componente conexa R representada por v_r, tiene aristas a otras componentes conexas S_1, S_2, ..., S_k representadas por v_1^s, v_2^s, ..., v_k^s. El primer objetivo es demostrar que R estará conectada con alguna S_i en no más de dos iteraciones.

Si $v_i^s < v_r$, para algún i, R pasará a ser representada por $\min(v_i^s)$ / $v_i^s < v_r$. Luego, R y S_i se conectarán tras una iteración, demostrando el punto anterior.

Caso contrario, se está en la situación en que $v_r < v_i^s$, i = 1, 2, ..., k. Si en la iteración q + 1 R y alguna S_i se conectan, quedará también demostrado el punto en cuestión.

En caso de que en la iteración q + 1 no se conecten, se puede asegurar que cada S_i es adyacente a otra componente conexa T_i, representada por v_i^t, que representará a S_i en la iteración q + 1. Se puede asegurar la existencia de T_i por el absurdo: si T_i no existiera, S_i sería representada por v_r en la iteración q + 1, pues $v_r < v_i^s$. Este razonamiento es el que permite concluir que $v_i^t < v_r$, pues v_i^t será el representante de S_i, siendo menor a v_r. De este modo, en la iteración q + 2, v_i^t representará a R. Así, en dos iteraciones, se ha conectado a R con S_i.

En definitiva, se necesitan a lo sumo dos iteraciones para conectar dos componentes conexas adyacentes. Esto es equivalente a decir que, después de dos iteraciones, el número de componentes conexas a ser conectadas se reduce a la mitad. En conclusión, todas ellas serán conectadas en 2 · $\log_2(N)$ iteraciones. □

A primera vista, este lema indica que el número de iteraciones necesarias para llegar a un resultado puede depender de la cantidad de elementos con la que se está trabajando. Sin embargo, si se cumple alguna de las siguientes condiciones, las cantidad necesaria será independiente del tamaño de los arreglos:

- La estructura del grafo es tal que cada componente conexa tiene a lo sumo k vértices, siendo k independiente del tamaño de los dominios. En este caso, el lema asegura la independencia.
- La independencia también puede asegurarse si cada una de las componentes conexas C_i puede ser particionada en dos componentes conexas C_i^1 y C_i^2 tales que: C_i^1 verifica la propiedad anterior, y los vértices de C_i^2 conforman un conjunto estable, pero están conectados a algún vértice de la primer componente.

  En este caso, en a lo sumo dos iteraciones se conectarán C_i^1 y C_i^2 (tal como se ha visto en la prueba del Teorema), conformando la componente conexa C_i. Como C_i^1 cumple con la primera condición, la misma puede ser hallada con complejidad $O(2 \cdot \log_2(k))$. Por ende, C_i puede ser hallada con costo $O(2 \cdot \log_2(k) + 2) = O(2 \cdot \log_2(k))$. Como k no depende del tamaño de los vértices-conjunto involucrados, se cumple la observación.
- Finalmente, se verificará la independencia si se cumple la condición anterior, permitiendo caminos ordenados entre los vértices de la segunda componente conexa C_i^2. Un camino v_1 - v_2 - ... v $_p$ es ordenado si $v_1 < v_2 < ... < v_p$.

Si este es el caso, entonces todos los v_i serán conectados entre sí en una única iteración, o serán conectados a la primer componente conexa C_i^1 en a lo sumo dos pasos. Por ende, C_i puede ser hallada con costo $O(2 \cdot \log_2(k) + 2) = O(2 \cdot \log_2(k) + 1) = O(2 * \cdot \log_2(k))$.

Por la representación que se ha escogido para GBCs, alguna de estas condiciones se cumplirá, pues se ha dotado de orden a las estructuras que componen a los grafos. En general serán necesarias pocas iteraciones del algoritmo, lo que es deseable, pues las iteraciones son costosas. En los ejemplos que presentamos en el capítulo 6 veremos que solo es necesario iterar una única vez para lograr el resultado deseado.

Capítulo 4

Implementaciones

En este capítulo se desarrollaran las implementaciones que se realizaron de los distintos elementos en C++, algunas de las cuales forman parte del compilador de Modelica, ModelicaCC.

No se tiene conocimiento de ninguna implementación para GBC. Sí se tenía conocimiento de la existencia de *grafos vectorizados*, que son similares a los GBCs, pero limitados a una única dimensión. De todos modos, no se consultó la implementación para desarrollar esta librería.

4.1. Librería de GBCs

Primeramente, se implementaron GCBs como una librería independiente del compilador, para que pueda ser utilizada en otros proyectos.

Las definiciones, y los algoritmos aquí presentados tienen una traducción casi directa para su implementación. De hecho, los algoritmos pueden ser considerados como porciones de pseudo código. De este modo, este documento sirve como guía para analizar la librería implementada para GBC.

Se debe destacar que a partir de este momento se utilizará simbología matemática para las entidades introducidas, dado que facilitan la comprensión al lector, que rápidamente tendrá una idea intuitiva de lo que hara la operación. Su uso deberá ser inferido por el contexto, donde se dejará en claro de qué tipo de estructura se está hablando.

Para implementar GBCs de manera compacta es necesario contar con una implementación compacta de vértices-conjunto, y aristas-conjunto. Se buscó definir conjuntos y mapas de manera compacta (ver 2.3 en caso de ser necesario), que son los elementos necesarios para definir vértices-conjunto y aristas-conjunto.

Para ello, se decidió trabajar del siguiente modo: primero se dará una definición matemática compacta de dichos elementos, que sea cercana a la implementación. Esto permitirá razonar acerca de la implementación, así como también demostrar propiedades matemáticamente. Luego, la implementación será un quasi copia de la definición matemática.

En esta tesis se han utilizado unos pocos lemas, pero la mayoría de los algoritmos no presentan una prueba de que los resultados que obtienen sean correctos. Sería deseable hacerlo, así como demostrar más propiedades acerca de las diferentes estructuras definidas.

4.1.1. Diseño

Al desarrollar esta librería se tuvo en cuenta que las estructuras implementadas eran nuevas, y que luego podrían definirse de otro modo, por lo que se pensó en presentar clases abstractas para los clientes de la misma, y contar con diversas clases concretas que implementen las diversas definiciones.

Para ello, se dio al menos una implementación de cada elemento a ser representado (clase concreta). Luego, haciendo uso de la construcción "typedef" de C++, se definen las clases abstractas. De este modo, si se quieren hacer cambios en la implementación, se define una nueva clase concreta para cada elemento afectado, y en los typedefs se reemplazan las clases concretas anteriores por las nuevas.

Las definiciones se hicieron de maneral incremental, donde se parte de una unidad básica, y luego las nuevas estructuras hacen uso de las unidades básicas, u otras estructuras ya definidas. Esto ayuda a tener un aprendizaje incremental al momento de introducirse a la librería. También es cierto que implica una mayor dependencia entre las estructuras definidas, pero es un inconveniente de definición, y no de implementación.

Esto es lo que asegura que, dando la implementación de las unidades básicas, y las posibles formas en que pueden combinarse, se presenta implícitamente la implementación de cada elemento que formará parte de la librería. Es por eso que en la mayor parte de este capítulo se trabajará con definiciones matemáticas, y no con la implementación directamente.

Esta librería fue desarrollada para ser usada en el compilador ModelicaCC, por lo que, aunque se ha intentado ser lo más general posible, puede que ciertas decisiones de implementación se vean influenciadas por este hecho. En cualquier caso, es un buen comienzo para aquel que desee extenderla y utilizarla en otro entorno.

Además de la librería, se escribió una test-suite que puede resultar útil para comprender cómo debe ser usada por otros programas.

4.1.2. Contenedores

Algunas clases tendrán operaciones cuyo resultado será una colección de elementos de la clase. Dado que se cuentan con diversas estructuras que pueden describir a una colección (listas, vectores, conjuntos, etc.), cada clase que tenga una operación de este tipo contará con template de tipo que parametriza dicha colección. Nuevamente, esto se hizo para que se puedan hacer cambios en la implementación de GBCs, sin que esto signifique grandes cambios para los clientes.

La implementación actual usa std::list para colecciones ordenadas, y boost::unordered_set para colecciones no ordenadas (estos serían los modos de combinar las unidades básicas). Para el segundo caso se optó por la opción de la librería boost, ya que permiten la definición de tipos incompletos, que es lo que habilita la definición de tipos recursivos (por ejemplo, al definir intervalos queremos que el resultado de la diferencia sea un conjunto de intervalos, lo cual es recursivo).

Las operaciones estarán basadas en operaciones de sus contrapartes matemáticas. Como todavía se encuentra en estudio y desarrollo, puede que esto no siempre suceda.

4.1.3. Aclaraciones previas

Algunas operaciones estarán sobrecargadas, y se podrá inferir su tipo por el tipo de los operandos.

Se usará [] para hacer referencia a arreglos. El operador ++ denotará la concatenación de los mismos, y # su longitud. Además, se utilizará el operador $[a_1, a_2, ..., a_k](i)$ para referenciar la i-ésima componente del arreglo. Dicho operador se utilizará para conjuntos, que en este caso, selecciona un elemento al azar del conjunto.

Si se itera sobre arreglos, se recorren los elementos en orden. Si se itera sobre conjuntos, se irá tomando un elemento al azar, sin repetirlos.

En muchos casos se ejemplificará para con caso unidimensional, ya que facilita la comprensión de las operaciones presentadas; y se puede razonar análogamente para el caso multidimensional.

Para denotar valores multidimensionales se utilizarán los paréntesis, y las comas como separadores.

Los algoritmos se han intentado describir de la manera más sencilla y clara posible, por lo que no se ha tenido en cuenta el costo computacional para explicar las operaciones. La implementación sí toma en cuenta este factor.

4.1.4. Intervalos

Modela un conjunto unidimensional de valores naturales consecutivos. Dichos valores son consecutivos con un cierto salto específico (se verá a continuación).

Los intervalos están compuestos por tres naturales: inicio, salto y fin; salto y fin pueden ser iguales a cero. Dado el intervalo i = [k:l:m] = $\{c \mid \exists\, d \in \mathbb{N}_0 \mid c = k + l * d \wedge c \leq m\}$, donde k es el inicio, l el salto, m el fin.

Desde el punto de vista téorico los naturales que componen a un intervalo pueden tomar cualquier valor (incluido el infinito). Para la implementación se define como infinito al máximo entero representable en C++, al cual llamaremos Inf.

Con este primer tipo de datos se puede apreciar la observación respecto de definición/implementación: los intervalos serán fácilmente implementados usando una estructura con tres `int` como miembros, y al momento de su creación se harán los chequeos correspondientes. Asimismo, la definición de intervalos implícitamente se usará en las diversas operaciones implementadas (por ejemplo, para implpementar la pertenencia a intervalos es necesario hacer uso de la definición). En el apéndice A se presenta la implementación de intervalos, para que se pueda comparar implementación con definición (las operaciones se encuentran definidas con otros nombres, se han comentado los nombres utilizados en este documento).

Creación de intervalos

Al crear un intervalo, puede suceder que la cota superior en verdad no sea parte de los elementos descriptos por el intervalo. Por ejemplo, 20 no pertenece al intervalo [0:3:20] = {0, 3, 6, 9, 12, 15, 18}, debido al valor de la cota inicial y del salto.

Es por este motivo que al crear un intervalo (en la implementación) [lo:st:hi] se ajusta la cota superior de manera que hi = k * step + lo, donde k es un natural. Para el caso anterior, el intervalo resultante sería [0:3:18].

Algoritmo 4 Creación de intervalos

```
1:  function CreateInterval(lo, st, hi)
2:      if lo ≥ 0 ∧ st > 0 ∧ hi ≥ 0 then
3:          if lo ≤ hi ∧ hi < ∞ then
4:              hi ← (hi - lo) % st
5:              return [lo:st:hi]
6:          else if lo ≤ hi ∧ hi = ∞ then
7:              return [lo:st:∞]
8:          else
9:              return [1:1:0]        ▷ lo > hi, se devuelve un intervalo vacío, pues la cota inferior es
    mayor que la cota superior
10:         end if
11:     else if lo ≥ 0 ∧ st = 0 ∧ lo = hi then                ▷ Intervalo de un único elemento
12:         return [lo:1:lo]
```

```
13:     else
14:         return [1:1:0]      ▷ Intervalo vacío, pues la cota inferior es mayor que la cota superior
15:     end if
16: end function
```

Al operar con intervalos en la librería, no se modifican las instancias recibidas como argumento, sino que siempre se crea un nuevo intervalo, con los valores necesarios, para realizar este ajuste.

Pertenencia a intervalos

Dado un natural, se desea comprobar si el mismo pertenece al intervalo [lo:st:hi]:

Algoritmo 5 Pertenencia a intervalos

```
1: function ∈(x, [lo:st:hi])
2:     if x < lo ∨ x > hi then
3:         return false
4:     end if
5:     if (x - lo) % st = 0 then                    ▷ % es el operador resto
6:         return true
7:     end if
8:     return false
9: end function
```

Intersección de intervalos

La intersección de los intervalos i_1 e i_2 tiene como resultado un nuevo intervalo. La cota inferior y superior son relativamente fáciles de calcular, pero el salto no es tan trivial.

El nuevo salto debe ser un múltiplo de ambos saltos, caso contrario, se podrían incluir elementos que pertenecen exclusivamente a solo uno de los intervalos (si es que los saltos son distintos). Además de ser común, el múltiplo debe ser mínimo, para no dejar fuera elementos que sí pertenecen a la intersección de intervalos.

Dados $i_1 = [l_1, s_1, h_1]$, $i_2 = [l_2, s_2, h_2]$:

Algoritmo 6 Intersección de intervalos

```
1: function ∩(i1, i2)
2:     m ← mcm(s1, s2)
3:     l3 ← min(k) : k ∈ i1 ∧ k ∈ i2
4:     h3 ← max(k) : k ∈ i1 ∧ k ∈ i2
5:     return [l3:m:h3]
6: end function
```

donde mcm(a, b) es mínimo común múltiplo de a y b.

Ejemplos:

- [10:2:20] ∩ [0:3:25] = [12:6:18]
- [14:2:16] ∩ [12:3:15] = []
- [1:1:10] ∩ [1:1:10] = [1:1:10]

Diferencia de intervalos

Para trabajar la diferencia, se tuvo presente una propiedad que vale para conjuntos: $c_1 \setminus c_2 = c_1 \setminus (c_1 \cap c_2)$. Lo que sucederá es que el resultado puede no ser un intervalo. Sin embargo, podemos obtener una colección no ordenada de intervalos equivalente al resultado deseado.

Por ejemplo, si tenemos [0:2:20] y [0:3:30], el resultado es: {2, 4, 8, 10, 14, 16, 20} (no hay salto constante). Esto puede describirse como la unión de [2:6:20] y [4:6:16].

Dados dos intervalos $i_1 = [lo_1{:}st_1{:}hi_1]$ e $i_2 = [lo_2{:}st_2{:}hi_2]$, si la intersección $i_1 \cap i_2$ tiene salto m, luego los elementos exclusivos de i_1 pueden ser descriptos mediante m/st_1 - 1 intervalos. Solo se diferencian entre sí en su cota inferior (que a su vez condiciona a la cota superior), distanciadas en st_1 unidades una de la otra.

Cabe destacar que este es un modo de definir la operación de diferencia de intervalos, ya que el costo de la misma no es dependiente de la cantidad de elementos que componen al intervalo (aunque sí de los saltos de ambos intervalos). Además, contando en la implementación con alguna estructura que represente colecciones no ordenadas (ver 4.1.2), ya estarán definidas todas las operaciones necesarias para implementar esta función. Naturalmente, se podría haber definido de otra manera.

Dados $i_1 = [l_1, s_1, h_1]$ e $i_2 = [l_2, s_2, h_2]$:

Algoritmo 7 Diferencia de intervalos

1: **function** $\setminus(i_1, i_2)$
2: $[l_3{:}s_3{:}h_3] \leftarrow i_1 \cap i_2$
3: **if** $[l_3{:}s_3{:}h_3] = \emptyset$ **then**
4: **return** $\{i_1\}$
5: **else**
6: pre $\leftarrow [l_1{:}s_1{:}l_3 - 1]$
7: post $\leftarrow [h_3 + s_1{:}s_1{:}h_1]$
8: **for** $k \leftarrow 1$; $k < s_3 / s_1$; $k \leftarrow k + 1$ **do**
9: $inter_k \leftarrow [l_3 + k * s_1{:}s_3{:}h_3]$
10: **end for**
11: inter $\leftarrow \bigcup_{k=1}^{s_3/s_1-1} \{inter_k\}$
12: **return** {pre} $\cup$ inter $\cup$ {post}
13: **end if**
14: **end function**

Ejemplos:

- [0:2:30] \ [] = [0:2:30]
- [0:2:30] \ [10:3:40] = {[0:2:8], [12:6:24], [14:6:26], [30:2:30]}

4.1.5. Multi-intervalos

Modelan un conjunto multidimensional de valores consecutivos. Se define como el producto cartesiano de una colección ordenada de intervalos (hiperrectángulos). Dados k intervalos i_1, i_2, ..., i_k podemos definir un multi-intervalo, de dimensión k, mi = i_1 x i_2 x ... x i_k.

Una restricción sobre los intervalos que componen un multi-intervalo es que ninguno de ellos debe ser vacío. Por ejemplo, si se permitiera el multi-intervalo [1:1:10] x [], se presupone que se trabajará con dos dimensiones, sin embargo, ese multi-intervalo provee información de una única dimensión. Este chequeo se realiza en la creación de multi-intervalos.

Pertenencia de multi-intervalos

Dado un elemento e = (e_1, e_2, ..., e_n), y el multi-intervalo mi = i_1 x i_2 x ... x i_k se define la pertenencia como:

Algoritmo 8 Pertenencia a multi-intervalos

```
1:  function ∈(e, mi)
2:      res ← true
3:      if n = k then
4:          for j ← 1; j ≤ k; j ← j + 1 do
5:              if ¬(e_j ∈ i_j) then
6:                  res ← false
7:              end if
8:          end for
9:      else
10:         res ← false
11:     end if
         return res
12: end function
```

Intersección de multi-intervalos

El resultado de la intersección de dos multi-intervalos es la colección que se obtiene al intersecar el i-ésimo elemento del primer multi-intervalo con el i-ésimo elemento del segundo multi-intervalo.

Dados dos multi-intervalos $mi_1 = i_1$ x i_2 x ... x i_k, $mi_2 = j_1$ x j_2 x ... x j_k:

Algoritmo 9 Intersección de multi-intervalos

```
1: function ∩(mi_1, mi_2)
2:     return (i_1 ∩ j_1) x (i_2 ∩ j_2) x ... x (i_k ∩ j_k)
3: end function
```

Ejemplos ($[]_k$ se utilizará para representar un multi-intervalo vacío de dimensión k):

- [0:2:20] x [30:2:40] x [25:1:30] ∩ $[]_3$ = $[]_3$
- [1:1:10] x [1:1:10] x [1:1:10] ∩ [30:1:40] x [30:1:40] x [30:1:40] = $[]_3$
- [1:1:10] x [1:1:10] ∩ [5:2:15] x [10:20:100] = [5:2:10] x [10:20:10]

Diferencia de multi-intervalos

Nuevamente, en base a la teoría de conjuntos, la idea es utilizar la propiedad de que $c_1 \setminus c_2 = c_1 \setminus (c_1 \cap c_2)$. El resultado será una colección de multi-intervalos disjuntos. Si bien no es necesario que los multi-intervalos sean disjuntos, se ha impuesto esta condición por cuestiones de eficiencia (al haber duplicados se debe operar con más elementos).

Dados $mi_1 = i_1$ x i_2 x ... x i_k, $mi_2 = j_1$ x j_2 x ... x j_k, $mi_3 = mi_1 \cap mi_2 = l_1$ x l_2 x ... x l_k. Por definición de producto cartesiano, e intersección de multi-intervalos se tiene que $mi_1 = \{(a_1, ..., a_k) / a_h \in i_h, h = 1, ..., k\}$, $mi_2 = \{(b_1, ..., b_k) / b_h \in j_h, h = 1, ..., k\}$, $mi_3 = \{(c_1, ..., c_k) / c_h \in i_h \cap j_h, h = 1, ..., k\}$. La idea es devolver $mi_4 = \{(d_1, ..., d_k) / \exists\ h / d_h \in i_h \wedge d_h \notin j_h \wedge (\forall q \neq h : d_q$

$\in i_q \vee d_q \in j_q)\}$. Para eso, se puede ir dimensión por dimensión, planteando la diferencia en cada una de ellas, y luego unir los resultados obtenidos.

A continuación se presenta una primera versión del algoritmo:

Algoritmo 10 Diferencia de multi-intervalos, primera versión

```
1:  function \(mi1, mi2)
2:      mi3 ← mi1 ∩ mi2
3:      for q ← 0; q < k; q ← q + 1 do
4:          for r ← 0; r < k; r ← r + 1 do
5:              if r ≠ q then
6:                  elem_{r+1} ← i_{r+1}
7:              end if
8:          end for
9:          diff ← i_{q+1} \ l_{q+1}
10:         res_{q+1} ← ∅
11:         for all s ∈ diff do
12:             elem_{q+1} ← s
13:             res_{q+1} ← res_{q+1} ∪ {elem_1 x elem_2 x ... x elem_k}
14:         end for
15:     end for
16:     return ⋃_{t=1}^{k} {res_t}
17: end function
```

La particularidad de este algoritmo es que no devuelve una colección disjunta de multi-intervalos. Por ejemplo:

- $res_w = i_1$ x i_2 x ... x i_{w-1} x s_1 x i_{w+1} x i_{w+2} x ... x i_k,
- $res_{w+1} = i_1$ x i_2 x ... x i_w x s_2 x i_{w+2} x i_{w+3} x ... x i_k

Se puede observar que ambos multi-intervalos se diferencian en la w-ésima y w+1-ésima componente. En la w-ésima componente se tiene s_1 e i_w, donde puede suceder que $s_1 \cap i_w \neq \emptyset$, ya que $s_1 \in i_w \setminus l_w$, y del mismo modo, para la w+1-ésima componente se cuenta con i_{w+1} y s_2, donde también puede darse $s_2 \cap i_{w+1} \neq \emptyset$ pues $s_2 \in i_{w+1} \setminus l_{w+1}$. Es decir, en cada dimensión puede haber elementos que pertenecen a ambos intervalos, y por ende los multi-intervalos no serían disjuntos.

Se presenta la versión que sí devuelve una colección disjunta:

Algoritmo 11 Diferencia de multi-intervalos, versión final

```
1:  function \(mi1, mi2)
2:      mi3 ← mi1 ∩ mi2
3:      for q ← 0; q < k; q ← q + 1 do
4:          for r ← 0; r < k; r ← r + 1 do
5:              if r < q then
6:                  elem_{r+1} ← l_{r+1}
7:              else if r > q then
8:                  elem_{r+1} ← i_{r+1}
9:              end if
10:         end for
11:         diff ← i_{q+1} \ l_{q+1}
```

12: $\text{res}_{q+1} \leftarrow \emptyset$
13: **for all** s ∈ diff **do**
14: $\text{elem}_{q+1} \leftarrow$ s
15: $\text{res}_{q+1} \leftarrow \text{res}_{q+1} \cup \{\text{elem}_1 \text{ x elem}_2 \text{ x ... x elem}_k\}$
16: **end for**
17: **end for**
18: **return** $\bigcup_{t=1}^{k} \{\text{res}_t\}$
19: **end function**

Es deseable que los multi-intervalos sean disjuntos, pues si no se está información redudante. Desde el punto de vista matemático no es un problema. Sin embargo, la primera versión de implementación se desarrolló usando arreglos para representar conjuntos. Es por eso que se obtenían multi-intervalos repetidos, y no es lo que se deseaba. En versiones posteriores se cambió la estructura que representaba conjuntos, de modo que el inconveniente desapareció, pero de todos modos se aprovechó este algoritmo para implementar la operación.

Ejemplos:

- [0:2:20] x [30:2:40] x [25:1:30] \ [5:3:15] x [30:2:30] x [27:1:35] = {[0:2:6] x [30:2:40] x [25:1:30], [10:6:10] x [30:2:40] x [25:1:30], [12:6:12] x [30:2:40] x [25:1:30], [16:2:20] x [30:2:40] x [25:1:30], [8:6:14] x [32:2:40] x [25:1:30], [8:6:14] x [30:2:30] x [25:1:26]}
- [0:2:20] x [30:2:40] x [25:1:30] \ [5:3:15] x [30:2:30] x [25:1:35] = {[0:2:6] x [30:2:40] x [25:1:30], [10:6:10] x [30:2:40] x [25:1:30], [12:6:12] x [30:2:40] x [25:1:30], [16:2:20] x [30:2:40] x [25:1:30], [8:6:14] x [32:2:40] x [25:1:30]}

Producto cruz de multi-intervalos

Dados $\text{mi}_1 = \text{i}_1$ x i_2 x ... x i_k, $\text{mi2}_2 = \text{j}_1$ x j_2 x ... x j_l, se tiene que mi_1 x_{mi} $\text{mi}_2 = \text{i}_1$ x i_2 x ... x i_k x j_1 x j_2 x ... x j_l

Ejemplo:

- [1:1:10] x [2:1:20] x_{mi} [50:5:100] = [1:1:10] x [2:1:20] x [50:5:100]

Elemento mínimo de multi-intervalos

Dado $\text{mi}_1 = \text{i}_1$ x i_2 x ... x i_k, el elemento mínimo de dicho multi-intervalo será el conformado por cada uno de los elementos mínimos de cada intervalo. Es decir, $\min(\text{mi}_1) = (\min(\text{i}_1), \min(\text{i}_2),$ ..., $\min(\text{i}_k))$.

Ejemplo:

- min([3:3:33] x [8:1:15] x [100:20:1000]) = (3, 8, 100)

Comparación de multi-intervalos

La idea es definir la operación < para multi-intervalos. En definitiva se devolverá como resultado el valor true si al ir comparando dimensión por dimensión, el elemento mínimo del intervalo del primer multi-intervalo es menor al del izquierdo, caso contrario se devolverá false (recordar que los intervalos conforman una colección ordenada para obtener un multi-intervalo).

Dados $\text{mi}_1 = \text{i}_1$ x i_2 x ... x i_k, $\text{mi}_2 = \text{j}_1$ x j_2 x ... x j_k, la relación $\text{mi}_1 < \text{mi}_2$ se define de la siguiente manera:

Algoritmo 12 Comparación de multi-intervalos

```
1: function <(mi_1, mi_2)
2:     for m ← 0; m < k; m ← m + 1 do
3:         if min(i_{m+1}) < min(j_{m+1}) then
4:             return true
5:         end if
6:     end for
7:     return false
8: end function
```

Ejemplos:

- [3:3:40] x [5:2:20] < [3:10:50] x [10:1:20] = true
- [3:3:40] x [5:2:20] < [3:10:50] x [5:1:20] = false

4.1.6. Conjuntos atómicos

Modelan un conjunto de multi-intervalos con un único elemento. Serán el componente básico de los conjuntos de multi-intervalos genéricos. No aportan nueva funcionalidad, sino que se encuentran presentes para explicitar que dicho conjunto está compuesto por un único multi-intervalo. Por este motivo, no se detallan sus operaciones.

En la implementación, cada conjunto atómico está compuesto por un multi-intervalo. Al operar entre conjuntos atómicos se hacen llamadas a las operaciones de los multi-intervalos que los componen. Todo multi-intervalo que forme parte del resultado, se usa para crear un conjunto atómico, que será parte del resultado.

Las operaciones implementadas para conjuntos atómicos son las siguientes:

- Intersección.
- Diferencia.
- Producto cruz.
- Elemento mínimo.
- Comparación (menor).

4.1.7. Conjuntos

Para diferenciar este concepto de su uso tradicional, cada vez que se haga referencia a esta nueva construcción se utilizará *italica*.

Modelan un conjunto multidimensional de valores seccionalmente consecutivos. Esto permite representar una mayor cantidad de elementos, de manera intensiva.

Cabe destacar que todos los conjuntos atómicos que componen a un *conjunto* deben tener la misma dimensión (restricción similar a la presentada para multi-intervalos). Esta restricción se chequea al crear el conjunto.

La dimensión de un *conjunto* es la dimensión de cualquiera de sus conjuntos atómicos (y no la cantidad de ellos). Además, los conjuntos atómicos deben ser disjuntos (nuevamente, por razones de eficiencia).

Pertenencia

Un elemento pertenece al *conjunto* si pertenece a alguno de sus conjuntos atómicos. Dado el *conjunto* s = {as^1, as^2, ..., as^n}, x ∈ s ⇔ x ∈ as^i, 1 ≤ i ≤ n.

Ejemplos:

- (5, 5) ∈ {{[1:1:10] x [1:1:10]}, {[25:1:30] x [25:1:30]}}
- (5, 15) ∉ {{[1:1:10] x [1:1:10]}, {[25:1:30] x [25:1:30]}}

Intersección de *conjuntos*

Para intersecar dos *conjuntos* se debe pensar que cada *conjunto* está compuesto por conjuntos atómicos. Por ende se deben considerar todas las posibles intersecciones entre conjuntos atómicos del primer *conjunto*, y aquellos pertenecientes al segundo *conjunto*.

Dados dos *conjuntos* s_1, s_2, tales que sus dimensiones son iguales:

Algoritmo 13 Intersección de *conjuntos*

```
1: function ∩(s1, s2)
2:     res ← ∅
3:     for all a_i ∈ s1 do
4:         for all b_j ∈ s2 do
5:             res ← res ∪ {a_i ∩ b_j}
6:         end for
7:     end for
8:     return res
9: end function
```

Ejemplo:

- {{[0:2:20] x [30:2:40] x [25:1:30]}, {[15:3:15] x [35:3:40] x [27:1:35]}} ∩ {{[0:1:100] x [20:3:50] x [28:1:28]}} = {{[0:2:20] x [32:6:38] x [28:1:28]}, {[15:3:15] x [35:3:40] x [28:1:28]}}

Diferencia de *conjuntos*

Dados dos conjuntos de la misma dimensión $s_1 = a_1 \cup a_2 \cup ... \cup a_j$, $s_2 = b_1 \cup b_2 \cup ... \cup b_k$, y su intersección $s_3 = c_1 \cup c_2 \cup ... \cup c_l$. Nuevamente, se desea sacar provecho de la propiedad de conjuntos que establece que $d_1 \setminus d_2 = d_1 \setminus (d_1 \cap d_2)$, y de la definición de resta de conjuntos atómicos.

La idea es ir elemento por elemento de s_1, restándole todos los elementos de la intersección (notar que serán j * k restas). Como se desea obtener un conjunto de conjuntos atómicos disjuntos (por definición de *conjunto*), para cada elemento de s_1 se tendrá un resultado parcial, que será al cual se le irán restando los elementos de la intersección.

Algoritmo 14 Diferencia de *conjuntos*

```
1: function \(s1, s2)
2:     res ← ∅
3:     s3 ← s1 ∩ s2
4:     for all a ∈ s1 do
5:         aux ← {a}
```

```
6:          for all c ∈ s_3 do
7:              for all aux_i ∈ aux do
8:                  aux ← aux ∪ (aux_i \ c)
9:              end for
10:         end for
11:         res ← res ∪ aux
12:     end for
13:     return res
14: end function
```

Ejemplos:

- {{[0:1:10] x [0:3:9]}} \ {{[0:1:10] x [0:3:9]}} = {}
- {{[3:3:30] x [3:3:30]}, {[75:8:99] x [100:1:200]}} \ {{[1:1:100] x [1:1:100]}, {[1:1:100] x [150:5:250]}} = {{[75:8:99] x [153:5:198]}, {[75:8:99] x [152:5:197]}, {[75:8:99] x [154:5:199]}, {[75:8:99] x [101:1:149]}, {[75:8:99] x [151:5:196]}}

Unión de *conjuntos*

Si bien la definición es bastante sencilla, se debe tener presente que el resultado debe contener conjuntos disjuntos. Es por ello que no se opera directamente uniendo ambos conjuntos de conjuntos atómicos, sino que primero se obtiene la diferencia de conjuntos, y luego se realiza la unión.

Algoritmo 15 Unión de *conjuntos*

```
1: function ∪(s_1, s_2)
2:     diff ← s_2 \ s_1
3:     return s_1 ∪ diff                          ▷ Unión de conjuntos
4: end function
```

Producto cruz de *conjuntos*

Dado que cada *conjunto* está compuesto por conjuntos atómicos, los cuales cuentan con un único multi-intervalo cada uno, el objetivo es realizar el producto cruz entre todos los pares posibles (cada elemento del par pertenece a un *conjunto* distinto).

Algoritmo 16 Producto cruz de *conjuntos*

```
1: function x_s(s_1, s_2)
2:     res ← ∅
3:     for all a ∈ s_1 do
4:         for all b ∈ s_2 do
5:             res ← res ∪ {a x_as b}          ▷ Producto cruz de conjuntos atómicos
6:         end for
7:     end for
8:     return res
9: end function
```

Ejemplo:

- {{[1:1:10] x [1:1:10]}, {[25:1:30] x [25:1:30]}} x_s {{[5:5:25]}, {[50:1:75]}} = {{[1:1:10] x [1:1:10] x [5:5:25]}, {[1:1:10] x [1:1:10] x [50:1:75]}, {[25:1:30] x [25:1:30] x [5:5:25]}, {[25:1:30] x [25:1:30] x [50:1:75]}}

Elemento mínimo de un *conjunto*

El elemento mínimo de un *conjunto* se puede calcular comparando los conjuntos atómicos que componen a cada conjunto atómico (recordar que la operación $<$ se encuentra definida para conjuntos atómicos). De este modo, se obtiene el conjunto atómico con el menor elemento mínimo, y se devuelve su mínimo.

Algoritmo 17 Elemento mínimo de un *conjunto*

```
 1: function MIN(s)
 2:     if s ≠ ∅ then
 3:         min_s ← x : x ∈ s
 4:         for all as_1 ∈ s do
 5:             if as_1 < min_s then
 6:                 min_s ← as_1
 7:             end if
 8:         end for
 9:         return MIN(min_s)                    ▷ MIN de conjuntos atómicos
10:     else
11:         return ()                            ▷ Colección ordenada vacía
12:     end if
13: end function
```

Ejemplo:

- MIN({{[5:1:10] x [5:1:10]}, {[5:1:7] x [15:5:30]}, {[3:1:4] x [100:1:1000]}}) = (3, 100)

4.1.8. Mapas lineales

Modelan expresiones lineales afines multidimensionales (IMPORTANTE: solo la expresión, no toman en cuenta ningún dominio). No son completamente análogos a una función. Servirán para describir las conexiones de Modelica. Más adelante se usará el término "expresión" para hacer referencia a los mapas lineales.

Se desacopla la expresión del dominio ya que es más sencillo modificar cada clase al tenerlas desacopladas (que es una de las premisas del desarrollo de la librería).

Un mapa lineal es una dupla de arreglos de reales (notar que ambos arreglos deben tener el mismo tamaño). A los elementos del primer arreglo se los denominará "ganancia".

Se permite el infinito para dichos reales. Estos valores se utilizan para distinguir casos especiales al operar con los mapas (ver inverso de un mapa lineal, dos secciones más adelante). Al permitir el infinito, se permite construir mapas lineales con ganancias infinitas. Pensandolo gráficamente en una dimensión, se representaría mediante una recta paralela al eje de las ordenadas, lo cual no es una función, de aquí que se haga la observación de que no son completamente análogos.

Dado que este valor se usa para distinguir ciertos resultados, en la implementación es importante que los clientes de la librería no construyan mapas lineales que lo utilicen. De hecho, cuando se usen estas estructuras para resolver el problema de aplanado, no tendría sentido definir un mapa de este tipo.

Los mapas lineales son la segunda unidad básica de la librería. Se implementarán como una estructura con dos miembros: dos arreglos de `float`, a los cuales se le harán los chequeos correspondientes (mismo tamaño).

Se llamará *dimensión* al tamaño de alguno de los arreglos.

Si bien los mapas lineales son una dupla de arreglos, para facilitar la comprensión al lector, al dar una instancia de alguno de ellos, se escribirá como un arreglo de expresiones lineales afines. Es decir, lsm = ([m^1, m^2, ..., m^k], [h^1, h^2, ..., h^k]) se escribirá como lsm = [m^1 * x + h^1, m^2 * x + h^2, ..., m^k * x + h^k].

Composición de mapas lineales

La composición de mapas es la misma que se hace al componer leyes de dos funciones, componente a componente. Dados los mapas lm_1 = ([m_1^1, m_1^2, ..., m_1^k], [h_1^1, h_1^2, ..., h_1^k]), lm_2 = ([m_2^1, m_2^2, ..., m_2^k], [h_2^1, h_2^2, ..., h_2^k]):

Algoritmo 18 Composición de mapas lineales

```
1: function ∘(lm1, lm2)
2:     res_m ← [m1^1 * m2^1, m1^2 * m2^2, ..., m1^k * m2^k]
3:     res_h ← [m1^1 * h2^1 + h1^1, m1^2 * h2^2 + h1^2, ..., m1^k * h2^k + h1^k]
4:     return (res_m, res_h)
5: end function
```

Ejemplo:

- [3 * x + 2, 2 * x - 4] ∘ [x - 5, 0.5 * x + 0.5] = [3 * x - 13, x - 3]

Inverso de un mapa lineal

Nuevamente, el inverso de un mapa lineal se calcula tal cual como se hace para el inverso de una función lineal, componente a componente. Se distingue el caso para el cual

Dado lm = ([m^1, m^2, ..., m^k], [h^1, h^2, ..., h^k]):

Algoritmo 19 Inverso de un mapa lineal

```
1: function INV_lm(lm)
2:     res_m ← []
3:     res_h ← []
4:     for j ← 1; j ≤ k; j ← j + 1 do
5:         if m^j ≠ 0 then
6:             res_m ← res_m ++ [1/m^j]
7:             res_h ← res_m ++ [-h^j/m^j]
8:         else
9:             res_m ← res_m ++ [∞]
10:            res_h ← res_m ++ [-∞]
11:        end if
12:    end for
13:    return (res_m, res_h)
14: end function
```

Se puede observar que el valor infinito fue utilizado para definir el mapa inverso de aquellos mapas que cuenten con ganancias iguales a 0. Ciertas operaciones que se definirán más adelante, y que hacen uso del inverso, al encontrarse con estos valores, sabrán que el mapa original contaba con alguna ganacia igual a 0 (ver mapa adyacente mínimo compacto, sección 4.2).

Ejemplo:

- INV_{lm}([3 * x - 2, 0.5 * x + 4]) = [$\frac{1}{3}$ * x + $\frac{2}{3}$, 2 * x - 8]

4.1.9. Mapas seccionalmente lineales atómicos

Modelan una función lineal afín, incluyendo dominio. Se definen para simplificar la posterior definición de los mapas seccionalmente lineales. Están compuestos por un conjunto atómico, y un mapa lineal. Se hará referencia al conjunto atómico como "dominio", y al mapa lineal como "expresión".

Nuevamente, como en el caso de los mapas lineales, la instancia de estos mapas se escribirá de manera más sencilla. Dado lsm = (as, ([m^1, m^2, ..., m^k], [h^1, h^2, ..., h^k])), se escribirá como (as, [m^1 * x + h^1, m^2 * x + h^2, ..., m^k * x + h^k].

Creación de mapas seccionalmente lineales atómicos

Al crear un mapa seccionalmente lineal atómico se debe verificar que la dimensión del conjunto atómico es igual a la del mapa lineal. Además, se debe recordar que los conjuntos están finalmente compuestos por naturales (incluido el 0); por esto también es necesario chequear que la aplicación del mapa lineal al conjunto atómico devuelve un conjunto atómico.

Dados un mapa lineal lm = ([m^1, m^2, ..., m^k], [h^1, h^2, ..., h^k]), y un conjunto atómico as = {i^1 x i^2 x ... x i^j} se define la función COMPATIBLE, que se utiliza en la creación de mapas seccionalmente lineales atómicos, para hacer los chequeos correspondientes:

Algoritmo 20 Compatibilidad

```
1:  function COMPATIBLE(as, lm)
2:      if k = j then
3:          for q ← 1; q ≤ k; q ← q + 1 do
4:              if m^q < ∞ then
5:                  i^q = [lo:st:hi]
6:                  if lo * m^q + h^q ∈ ℕ0 ∧ hi * m^q + h^q ∈ ℕ0 ∧ m^q * st ∈ ℕ then
7:                      return true
8:                  else
9:                      return false
10:                 end if
11:             else
12:                 return true
13:             end if
14:         end for
15:     else
16:         return false
17:     end if
18: end function
```

Ejemplos:

- COMPATIBLE({[3:3:9] x [10:4:18]}, [$\frac{1}{3}$ * x + 0, $\frac{1}{2}$ * x + 1]) = true
- COMPATIBLE({[3:1:9] x [10:5:15]}, [$\frac{1}{3}$ * x + 0, $\frac{1}{2}$ * x + 1]) = false
- COMPATIBLE({[3:2:9] x [10:5:15]}, [$\frac{1}{2}$ * x + $\frac{1}{2}$, 1 * x + 1]) = true

Imagen de mapas seccionalmente lineales atómicos

Esta operación toma como argumento un mapa seccionalmente lineal atómico, y un conjunto atómico. Luego se calcula la imagen del mapa lineal aplicado a la intersección del dominio con el argumento, dando como resultado un nuevo conjunto atómico. Dado lsm = (as_{dom}, ([m^1, m^2, ..., m^k], [h^1, h^2, ..., h^k])):

Algoritmo 21 Imagen de mapas seccionalmente lineales atómicos

```
1:  function IM(lsm, as)
2:      inter ← as ∩ as_dom
3:      inter = {int_1 x int_2 x ... x int_k}
4:      for j ← 1; j ≤ k; j ← j + 1 do
5:          int_j = [lo:st:hi]
6:          if m_j < ∞ then
7:              res_j ← [lo * m_j + h_j:st * m_j:hi * m_j + h_j]
8:          else
9:              res_j ← [0:1:∞]
10:         end if
11:     end for
12:     return {res_1 x res_2 x ... x res_k}
13: end function
```

Puede resultar extraño el resultado del else en la línea 9. La manera más sencilla de razonarlo es gráficamente: se trata de una recta paralela al eje de las ordenadas, es decir, devuelve todos los posibles valores de los naturales.

Ejemplo:

- IM(({[1:1:10] x [1:1:10] x [1:1:10]}, [2 * x, 3 * x, x - 1]), {[1:5:30] x [5:1:10] x [5:1:10]}) = {[2:10:12] x [15:3:30] x [4:1:9]}

Preimagen de mapas seccionalmente lineales atómicos

De igual manera que la imagen, toma como argumento un conjunto atómico. Se desea calcular la preimagen de la intersección del argumento con la imagen del mapa seccionalmente lineal atómico. Dado lsm = (as_{dom}, ([m^1, m^2, ..., m^k], [h^1, h^2, ..., h^k])):

Algoritmo 22 Preimagen de mapas seccionalmente lineales atómicos

```
1: function PREIM(lsm, as)
2:     fullIm ← IM(lsm, as_dom)
3:     interIm ← fullIm ∩ as
4:     inv ← INV_lm (([m^1, m^2, ..., m^k], [h^1, h^2, ..., h^k]))
5:     newLSM ← (interIm, inv)
6:     lsm = (as_dom, lm)
7:     return as_dom ∩ IM(newLSM, interIm)
8: end function
```

Ejemplo:

- PREIM(({[1:1:10] x [1:1:10] x [1:1:10]}, [2 * x, 3 * x, x - 1]), {[1:1:10] x [1:1:10] x [1:1:10]}) = {[1:1:5] x [1:1:3] x [2:1:10]}

4.1.10. Mapas seccionalmente lineales

Modelan una colección de funciones afines con sus respectivos dominios. Están compuestos por una colección ordenada de *conjuntos*, junto con una colección ordenada de mapas lineales. El orden es necesario para establecer qué dominio se corresponde con cada expresión (notar que se llamará dominio a cada componente de la colección de mapas lineales, y expresión a cada componente de la colección de conjuntos).

Nuevamente se impone la restricción de que los conjuntos sean disjuntos, por motivos de eficiencia. Como sucedió anteriormente, los mapas y *conjuntos* deben ser compatibles componente a componente, y todos los mapas lineales y *conjuntos* deben tener la misma dimensión. La *dimensión* de un mapa seccionalmente lineal es la de cualquiera de sus componentes. Dichos chequeos se realizan en la creación de los mapas seccionalmente lineales.

Un mapa seccionalmente lineal será de la forma lsm = ([s^1, s^2, ..., s^k], [lm^1, lm^2, ..., lm^k]), donde s^i es un *conjunto* y lm^i es un mapa lineal.

Imagen y preimagen de mapas seccionalmente lineales

Ambas operaciones se pueden definir de manera sencilla, ya que se puede recorrer componente por componente de ambos arreglos, trabajando con pares de *conjuntos* y mapas lineales. A su vez, recorriendo cada conjunto, se obtienen pares de mapas lineales y conjuntos atómicos. De este modo se utilizan las operaciones imagen y preimagen de mapas seccionalmente lineales atómicos. Luego, solo resta unir los resultados obtenidos.

Dado lsm = ([s^1, s^2, ..., s^k], [lm^1, lm^2, ..., lm^k]), y el *conjunto* c:

Algoritmo 23 Imagen de mapas seccionalmente lineales

```
1:  function Im(lsm, c)
2:      res ← ∅
3:      lsm = (s, lm)
4:      for all s^i ∈ s do
5:          inter ← s^i ∩ c
6:          for all as^h ∈ inter do                ▷ as^h es un conjunto atómico
7:              im ← Im((as^h, lm^i), as^h)        ▷ Imagen de mapas seccionalmente lineales atómicos
8:              res ← res ∪ {im}
9:          end for
10:     end for
11:     return res
12: end function
```

El algoritmo para calcular la preimagen utiliza una idea similar, con algunos cambios. Dado lsm = ([s^1, s^2, ..., s^k], [lm^1, lm^2, ..., lm^k]), y el *conjunto* c:

Algoritmo 24 Preimagen de mapas seccionalmente lineales

```
1:  function PreIm(lsm, c)
2:      res ← ∅
3:      lsm = (s, lm)
4:      for all s^i ∈ s do
5:          for all as^h ∈ s^i do                  ▷ as^h es un conjunto atómico
6:              for all c^q ∈ c do                 ▷ c^q es un conjunto atómico
```

```
7:                 preim ← PreIm((as^h, lm^i), c^i)   ▷ Pre imagen de mapas seccionalmente lineales atómicos
8:                 res ← res ∪ {preim}
9:             end for
10:        end for
11:    end for
12:    return res
13: end function
```

Ejemplos:

- Im(([{{[1:1:5] x [1:1:5]}, {[10:1:15] x [10:1:15]}}, {{[20:3:30] x [20:3:30]}, {[45:5:50] x [45:5:50]}}], [[x, x], [2 * x, 2 * x]]), {{[1:1:5] x [1:1:5]}, {[10:1:15] x [10:1:15]}, {[20:3:30] x [20:3:30]}, {[45:5:50] x [45:5:50]}}) = {{[1:1:5] x [1:1:5]}, {[10:1:15] x [10:1:15]}, {[40:6:60] x [40:6:60]}, {[90:10:100] x [90:10:100]}}
- PreIm(([{{[1:1:10] x [1:1:10]}, {[20:5:30] x [20:5:30]}}, {{[11:1:14] x [11:1:14]}, {[1:1:10] x [50:5:70]}}], [[0 * x + 3, 0 * x + 4], [2 * x, 2 * x + 1]]), {{[0:1:25] x [0:1:25]}}) = {{[1:1:10] x [1:1:10]}, {[20:5:30] x [20:5:30]}, {[11:1:12] x [11:1:12]}}

Composición de mapas seccionalmente lineales

Para obtener el resultado de la composición, se deben componer las expresiones de ambos mapas, y además, se debe calcular el dominio del nuevo mapa. Se deben considerar todas las posibles composiciones, por lo que cada par del primer mapa se compondrá con cada par del segundo mapa.

Dados $\text{lsm}_1 = ([\text{s}_1^1, \text{s}_1^2, ..., \text{s}_1^k], [\text{lm}_1^1, \text{lm}_1^2, ..., \text{lm}_1^k])$, $\text{lsm}_2 = ([\text{s}_2^1, \text{s}_2^2, ..., \text{s}_2^j], [\text{lm}_2^1, \text{lm}_2^2, ..., \text{lm}_2^j])$:

Algoritmo 25 Composición de mapas seccionalmente lineales

```
1: function ∘(lsm1, lsm2)
2:     res_exp ← []
3:     res_lm ← []
4:     lsm1 = (c1, lmap1)
5:     lsm2 = (c2, lmap2)
6:     for all s1^h ∈ c1 do
7:         for all s2^m ∈ c2 do
8:             im ← Im(lsm2, s2^m)                ▷ Cálculo del nuevo dominio
9:             inter ← im ∩ s1^h
10:            newDom ← PreIm(lsm2, inter)
11:            if newDom ≠ ∅ then
12:                newExp ← lm1^h ∘ lm2^m
13:                res_dom ← res_dom ++ [newDom]
14:                res_lm ← res_lm ++ [newExp]
15:            end if
16:        end for
17:    end for
18:    return (res_dom, res_lm)
19: end function
```

Ejemplo:

- ([{{[1:1:10] x [1:1:5]}, {[20:2:30] x [20:2:30]}}, {{[15:3:18] x [12:3:18]}}], [[2 * x + 1, 3 * x], [0, 0]]) ∘ ([{{[1:1:30] x [1:1:30]}}], [[x + 1, x + 2]]) = ([{{[1:1:9] x [1:1:3]}, {[19:2:29], [18:2:28]}}, {{[14:3:17] x [10:3:16]}}], [[2 * x + 3, 3 * x + 6], [0, 0]])

Combinación de mapas seccionalmente lineales

La idea de combinar dos mapas es devolver un nuevo mapa que se comporte como el primer argumento en el dominio del mismo, y luego como el otro mapa en la diferencia entre ambos dominios. Dados $\text{lsm}_1 = ([\text{s}_1^1, \text{s}_1^2, ..., \text{s}_1^k], [\text{lm}_1^1, \text{lm}_1^2, ..., \text{lm}_1^k])$, $\text{lsm}_2 = ([\text{s}_2^1, \text{s}_2^2, ..., \text{s}_2^j], [\text{lm}_2^1, \text{lm}_2^2, ..., \text{lm}_2^j])$:

Algoritmo 26 Combinación de mapas seccionalmente lineales

```
 1: function COMBINE(lsm1, lsm2)
 2:     res_dom ← [s_1^1, s_1^2, ..., s_1^k]
 3:     res_lm ← [lm_1^1, lm_1^2, ..., lm_1^k]
 4:     if k = 0 then                                   ▷ Mapa vacío
 5:         return lsm2
 6:     else if j = 0 then                              ▷ Mapa vacío
 7:         return lsm1
 8:     else
 9:         wholeDom ← ∅                                ▷ wholeDom es un conjunto
10:         for all s_1^m do
11:             wholeDom ← wholeDom ∪ s_1^m
12:         end for
13:         for all s_2^m do
14:             newDom ← s_2^m \ wholeDom
15:             if newDom ≠ ∅ then
16:                 res_dom ← res_dom ++ [newDom]
17:                 res_lm ← res_lm ++ [lm_2^m]
18:             end if
19:         end for
20:     end if
21:     return (res_dom, res_lm)
22: end function
```

Ejemplo:

- COMBINE(([{{[1:1:10] x [1:1:10] x [1:1:10]}, {[1:1:10] x [20:3:30] x [20:3:30]}}, {{[1:1:10] x [20:3:30] x [35:5:50]}, {[35:5:50] x [35:5:50] x [20:3:30]}}], [[x, x, x], [3, 3, x + 1]]), ([{{[1:1:20] x [1:1:20] x [1:1:20]}}], [[x + 1, x, x]])) = (([{{[1:1:10] x [1:1:10] x [1:1:10]}, {[1:1:10] x [20:3:30] x [20:3:30]}}, {{[1:1:10] x [20:3:30] x [35:5:50]}, {[35:5:50] x [35:5:50] x [20:3:30]}}], [[x, x, x], [3, 3, x + 1]]), ({{[11:1:20] x [1:1:20] x [1:1:20]}, {[1:1:10] x [11:1:19] x [1:1:20]}, {[1:1:10] x [20:3:20] x [1:1:19]}, {[1:1:10] x [1:1:10] x [11:1:20]}}, [[x + 1, x, x]]))

Atomización de mapas seccionalmente lineales

El objetivo de esta operación es devolver un mapa seccionalmente lineal, cuya colección de *conjuntos* esté compuesta por *conjuntos* de un único conjunto atómico. Dado $\text{lsm} = ([\text{s}^1, \text{s}^2, ..., \text{s}^k], [\text{lm}^1, \text{lm}^2, ..., \text{lm}^k])$:

Algoritmo 27 Atomización de mapas seccionalmente lineales

```
1: function ATOMIZE(lsm)
2:     res_dom ← []
3:     res_lm ← []
4:     lsm = ([s^1, s^2, ..., s^k], lm)
5:     for all s^i do
6:         for all as^j ∈ s^i do                    ▷ as^j es un conjunto atómico
7:             res_dom ← res_dom ++ [{as^j}]
8:             res_lm ← res_lm ++ [lm^i]
9:         end for
10:    end for
11:    return (res_dom, res_lm)
12: end function
```

Ejemplo:

- ATOMIZE(([{{[1:1:10] x [1:1:10]}, {[20:3:50] x [20:3:50]}}, {{[21:3:50] x [20:3:50]}}]), [[x, x], [2 * x, 1]]) = ([{{[1:1:10] x [1:1:10]}}, {{[20:3:50] x [20:3:50]}}, {{[21:3:48] x [20:3:48]}}]), [[x, x], [x, x], [2 * x, 1]])

4.1.11. Grafos Basados en Conjuntos

Una vez definidos los *conjuntos*, y los mapas seccionalmente lineales, se puede representar los vértices-conjunto como *conjuntos*, y cada arista-conjunto mediante un par de mapas seccionalmente lineales. Esta representación puede resultar restrictiva, pero para la aplicación a aplanado resulta razonable.

Vértices-conjunto, aristas-conjunto

En la implementación cada vértice-conjunto está compuesto por un *conjunto*, y cierta información adicional para depuración y otras funciones. Del mismo modo las aristas-conjunto estarán compuestas por dos mapas seccionalmente lineales, cuyo dominio será igual (recordar la representación dada en 2.3). Se hará referencia al primer mapa como "lado izquierdo", y al segundo como "lado derecho".

Los GBCs fueron finalmente implementados como un grafo de la librería boost, cuyos nodos son vértices-conjunto, y cuyas aristas son aristas-conjunto. Básicamente, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, donde $\mathcal{V}$ es un conjunto de vértices-conjunto, y $\mathcal{E}$ un conjunto de aristas-conjunto.

Dado que la estructura de boost internamente guarda la información de qué nodo se ha conectado con cual nodo, no es necesario guardar algún tipo de colección de conexiones entre vértices-conjunto.

Para explicar las operaciones que se presentarán a continuación, solo cierta información es relevante. Por ese motivo, al referirse al vértice-conjunto V, en realidad se estará tratando con el *conjunto* que lo compone. Del mismo modo, al hablar de una arista-conjunto E se dirá que la misma es una tupla de mapas seccionalmente lineales, es decir, $E = (\mathrm{lsm}_{left}, \mathrm{lsm}_{right})$.

4.2. Algoritmo de componentes conexas para Grafos Basados en Conjuntos

Dado $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, se devolverá un mapa seccionalmente lineal, que representaría al arreglo de representantes R_{map} del capítulo anterior (ver 3.1):

Algoritmo 28 Componentes conexas de Grafos Basados en Conjuntos

function ConnectedComponents($\mathcal{G}$)
  if $\mathcal{V} \neq \emptyset$ **then**
    $\mathrm{aux}_s \leftarrow \emptyset$ ▷ Conjunto de componentes conexas iniciales
    for all v $\in \mathcal{V}$ **do**
      $\mathrm{aux}_s \leftarrow \mathrm{aux}_s \cup$ v
    end for
    lm $\leftarrow$ ([1], [0]) ▷ Cada vértice se representa a sí mismo, inicialmente
    $\mathrm{R}_{map} \leftarrow (\mathrm{aux}_s, \mathrm{lm})$
    if $\mathcal{E} = \emptyset$ **then**
      return R_{map}
    else
      $\mathrm{E}^1 \leftarrow \mathcal{E}$(i)
      $\mathrm{E}^1 = (\mathrm{lsm}_{left}, \mathrm{lsm}_{right})$
      for all $\mathrm{E}_i \in \mathcal{E}$ **do** ▷ Se obtienen mapas derechos e izquierdos
        $\mathrm{E}^i = (\mathrm{lsm}^i_{left}, \mathrm{lsm}^i_{right})$
        $\mathrm{lsm}_{left} \leftarrow$ Combine($\mathrm{lsm}^i_{left}, \mathrm{lsm}_{left}$)
        $\mathrm{lsm}_{right} \leftarrow$ Combine($\mathrm{lsm}^i_{right}, \mathrm{lsm}_{right}$)
      end for
      $\mathrm{old}_{im} \leftarrow \emptyset$
      $\mathrm{new}_{im} \leftarrow \mathrm{aux}_s$
      $\mathrm{diff}_{im} \leftarrow \mathrm{aux}_s$
      while $\mathrm{diff}_{im} \neq \emptyset$ **do**
        $\mathrm{ER}^1_{map} \leftarrow \mathrm{R}_{map} \circ \mathrm{lsm}_{left}$ ▷ Representantes del lado izquierdo
        $\mathrm{ER}^2_{map} \leftarrow \mathrm{R}_{map} \circ \mathrm{lsm}_{right}$ ▷ Representantes del lado derecho
        $\mathrm{R}^1_{map} \leftarrow$ MinAdj($\mathrm{ER}^1_{map}, \mathrm{ER}^2_{map}$) ▷ Menor vértice del lado derecho, adyacente a un representante izquierdo
        $\mathrm{R}^2_{map} \leftarrow$ MinAdj($\mathrm{ER}^2_{map}, \mathrm{ER}^1_{map}$) ▷ Menor vértice del lado izquierdo, adycente a un representante derecho
        $\mathrm{R}^1_{map} \leftarrow$ Combine($\mathrm{R}^1_{map}, \mathrm{D}_{map}$) ▷ Todavía falta conectar representantes, hay que combinar el resultado anterior con el nuevo resultado
        $\mathrm{R}^2_{map} \leftarrow$ Combine($\mathrm{R}^2_{map}, \mathrm{D}_{map}$)
        $\mathrm{aux}_R \leftarrow$ MinMap($\mathrm{R}^1_{map}, \mathrm{R}^2_{map}$) ▷ Selección de los menores representantes
        $\mathrm{old}_{im} \leftarrow \mathrm{new}_{im}$
        $\mathrm{new}_{im} \leftarrow$ Im($\mathrm{aux}_R, \mathrm{aux}_s$)
        $\mathrm{diff}_{im} \leftarrow \mathrm{old}_{im} \setminus \mathrm{new}_{im}$
        if $\mathrm{diff}_{im} \neq \emptyset$ **then**
          $\mathrm{R}_{map} \leftarrow \mathrm{aux}_R$
          $\mathrm{R}_{map} \leftarrow$ MapInf(R_{map}) ▷ Actualización de representante de una componente conexa
          $\mathrm{new}_{im} \leftarrow$ Im($\mathrm{R}_{map}, \mathrm{aux}_s$)

```
            end if
          end while
        end if
    end if
    return R_map
end function
```

Se puede observar que aquello que fue presentado en la sección 3.2 tiene traducción casi directa a este algoritmo. Resta presentar las operaciones utilizadas para esta operación, lo que se hará a continuación.

Mínimo seccionalmente lineal atómico

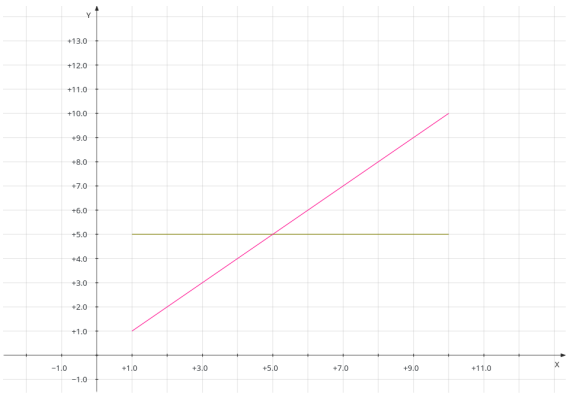
Dados dos mapas lineales, y un conjunto atómico (dominio), se desea obtener el mínimo mapa seccionalmente lineal para el mencionado dominio. Nuevamente, se debe pensar los mapas lineales como funciones afines: dado un cierto dominio, pueden ser comparadas, y obtener la mínima de ellas. Puede que sea necesario seccionar el dominio, de acuerdo a las expresiones de los mapas. La clave para razonar esta operación es operar dimensión por dimensión, hasta encontrar diferencia en una de ellas (tal como se hace para el mínimo de multi-intervalos).

Dados el conjunto atómico as = $\{i_1 \text{ x } i_2 \text{ x } ... \text{ x } i_k\}$, y los mapas lineales $lm_1 = ([m_1^1, m_1^2, ..., m_1^k], [h_1^1, h_1^2, ..., h_1^k])$, $lm_2 = ([m_2^1, m_2^2, ..., m_2^k], [h_2^1, h_2^2, ..., h_2^k])$:

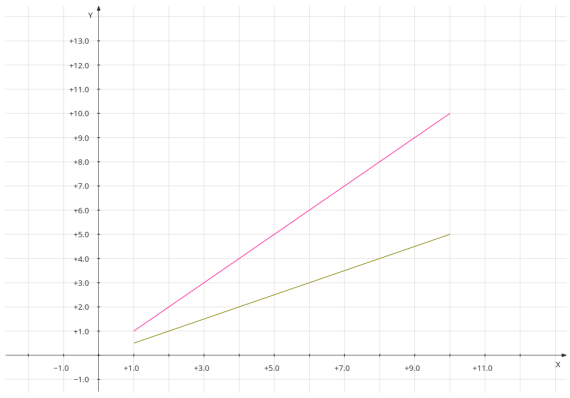
Algoritmo 29 Mínimo seccionalmente lineal atómico

```
1: function MinAtom_ls(as, lm_1, lm_2)
2:     res_dom = [{as}]
3:     res_lm = [lm_1]
4:     for q ← 1; q ≤ k; q ← q + 1 do
5:         if m_1^q ≠ m_2^q then
6:             inter ← (h_2^q - h_1^q) / (m_1^q - m_2^q)        ▷ Intersección de las expresiones
7:             i_q = [lo:st:hi]
8:             if inter ≤ lo then                              ▷ La intersección se da antes del dominio
9:                 if m_1^q < m_2^q then
10:                    res_lm ← [lm_2]
11:                end if
12:            else if inter > hi then                         ▷ La intersección se da después del dominio
13:                if m_2^q > m_1^q then
14:                    res_lm ← [lm_2]
15:                end if
16:            else                                            ▷ La intersección está en el dominio
17:                bef ← [lo:st:⌊inter⌋]
18:                aft ← [⌈inter⌉:st:hi]
19:                part_bef ← {{i_1 x i_2 x ... x i_{q-1} x bef x i_{q+1} x ... x i_n}}
20:                part_aft ← {{i_1 x i_2 x ... x i_{q-1} x aft x i_{q+1} x ... x i_n}}
21:                res_dom = [part_bef, part_aft]
22:                if m_1^q > m_2^q then
23:                    res_lm ← [lm_1, lm_2]
24:                else
25:                    res_lm ← [lm_2, lm_1]
```

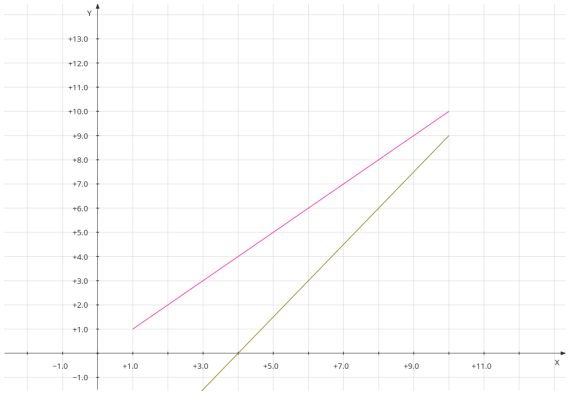
26: **end if**
27: **end if**
28: **break**
29: **else if** $\mathrm{h}_1^q \neq \mathrm{h}_2^q$ **then**
30: **if** $\mathrm{h}_1^q > \mathrm{h}_2^q$ **then**
31: $\mathrm{res}_{lm} \leftarrow [\mathrm{lm}_2]$
32: **end if**
33: **break**
34: **end if**
35: **end for**
36: **return** $(\mathrm{res}_{dom}, \mathrm{res}_{lm})$
37: **end function**



(a) *Intersección en el dominio*



(b) *Intersección "antes" del dominio*



(c) *Intersección "después" del dominio*

Para ejemplificar, y hablando de manera coloquial, se ha presentado el caso para una dimensión, con las tres posibilidades factibles (se presentan segmentos en vez de rectas, para explicitar el dominio). Si la intersección no se encuentra en el dominio, se devolverá un mapa seccionalmente lineal con arreglos de tamaño 1. Caso contrario, tendrán tamaño 2: el primer para para "antes" de la intersección, y el segundo para "después" de la intersección.

Ejemplo:

- MinAtom$_{ls}$({[2:2:20] x [1:1:10] x [3:3:50]}, [x + 60, 2 * x + 2, 35], [x + 60, 2 * x + 2, x + 10]) = ([{{[2:2:20] x [1:1:10] x [3:3:24]}}, {{[2:2:20] x [1:1:10] x [27:3:50]}}], [[x + 60, 2 * x + 2, x + 10], [x + 60, 2 * x + 2, 35]])

Mínimo seccionalmente lineal

Es análogo al caso del mínimo seccionalmente lineal atómico, solo que en este caso el dominio será un *conjunto*, y no un conjunto atómico. Aprovechando la operación anterior, se recorrerá conjunto atómico por conjunto atómico, calculando el mínimo linealmente seccional para cada uno de ellos, y se combinarán los resultados obtenidos.

Para comprender este algoritmo es clave tener presente que siempre se busca el mínimo entre los mismos mapas, solo varía el dominio. Dado el *conjunto* $s = \{as_1, as_2, ..., as_n\}$, y los mapas lineales lm_1, lm_2:

Algoritmo 30 Mínimo seccionalmente lineal

```
1: function MIN_ls(s, lm1, lm2)
2:     res ← MINATOM_ls(as1, lm1, lm2)          ▷ Inicialización del resultado parcial
3:     res = (res_dom, res_lm)
4:     res_dom = [s1_1, s1_2, ..., s1_r]                              ▷ max(r) = 2
5:     res_lm = [lm1_1, lm1_2, ..., lm1_r]                            ▷ max(r) = 2
6:     for q ← 1; q ≤ n; q ← q + 1 do
7:         auxRes ← MINATOM_ls(as_q, lm1, lm2)
8:         auxRes = (auxRes_dom, auxRes_lm)
9:         for t ← 1; t ≤ #(auxRes_dom); t ← t + 1 do   ▷ auxRes_dom puede tener dos conjuntos
10:            if auxRes_lm(t) = res_lm(1) then                        ▷ Antes de la intersección
11:                res_dom(1) ← res_dom(t) ∪ auxRes_dom(t)
12:            else
13:                if #res_dom = 1 then                                ▷ Intersección
14:                    res_dom(2) ← auxRes_dom(t)
15:                    res_lm(2) ← auxRes_lm(t)
16:                else                                    ▷ Después de la intersección
17:                    res_dom(2) ← res_dom(2) ∪ auxRes_dom(t)
18:                end if
19:            end if
20:        end for
21:    end for
22:    return (res_res, res_lm)
23: end function
```

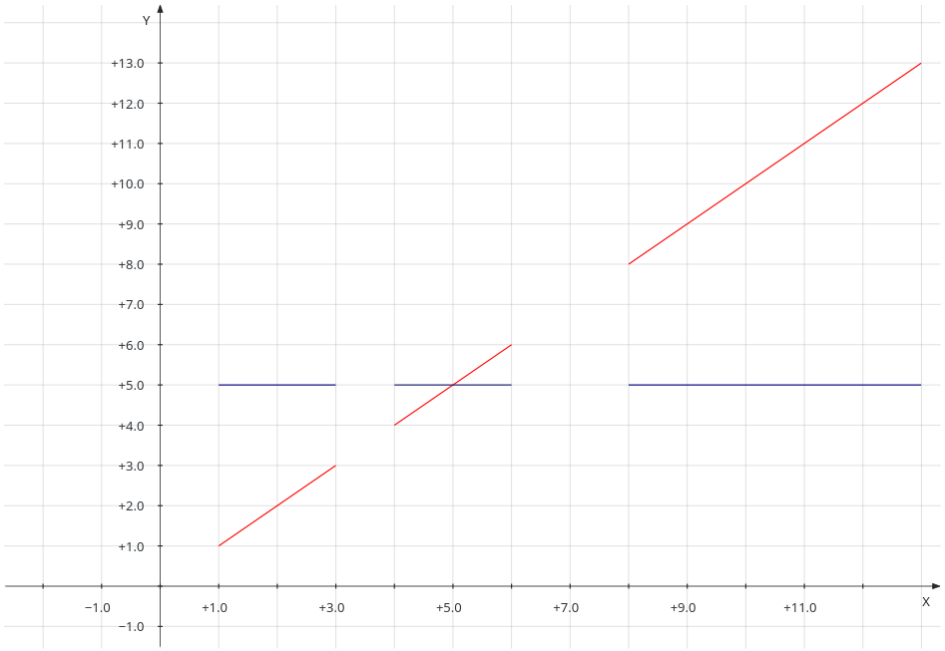


Imagen 4.2: *Mínimo seccionalmente lineal, ejemplo*

Observando el ejemplo y el algoritmo propuesto se puede observar que el primer tramo se tratará en las líneas 10-11, el segundo en las líneas 12-16, y el tercero en las 16-17.

Ejemplo:

- MIN_{ls}({{[1:1:5]}, {[10:1:15]}, {[20:2:30]}}, [12], [2 * x - 12]) = ([{{[1:1:5]}, {[10:1:12]}}, {{[20:2:30]}, {[13:1:15]}}], [[2 * x - 12], [12]])

Mapa mínimo

Se desea definir el mapa mínimo entre dos mapas, para los elementos que sean comunes a ambos. Solo se deben intersecar los conjuntos del primer mapa con los del segundo, obtener los mínimos seccionalmente lineales, y combinar los resultados.

Dados $\text{lsm}_1 = ([\text{s}_1^1, \text{s}_1^2, ..., \text{s}_1^k], [\text{lm}_1^1, \text{lm}_1^2, ..., \text{lm}_1^k])$, $\text{lsm}_2 = ([\text{s}_2^1, \text{s}_2^2, ..., \text{s}_2^j], [\text{lm}_2^1, \text{lm}_2^2, ..., \text{lm}_2^j])$:

Algoritmo 31 Mapa mínimo

```
function MIN_map(lsm_1, lsm_2)
    res_dom ← []
    res_lm ← []
    for all s_1^p do
        for all s_2^q do
            dom ← s_1^p ∩ s_2^q
            if dom ≠ ∅ then
                partialMin ← MIN_ls(dom, lm_1^p, lm_2^q)
                if res_dom = [] then                          ▷ Inicialización del resultado
                    (res_dom, res_lm) ← partialMin
                else
                    (res_dom, res_lm) ← COMBINE(partialMin, (res_dom, res_lm))
                end if
```

```
        end if
      end for
    end for
    return (res_dom, res_lm)
  end function
```

Ejemplo:

- Min_{map}([[{{[1:1:10]}, {[15:3:30]}}, {{[12:3:12]}, {[50:5:100]}}], [[x], [x + 1]]), ([{{[1:2:20]}, {[30:5:60]}}, {{[75:5:90]}, {[95:1:100]}}], [[100], [x]])) = ([{{[75:5:90]}, {[95:5:100]}}, {{[50:5:60]}}, {{[1:2:9]}, {[15:6:18]}, {[30:15:30]}}], [[x], [x + 1], [x]])

Mapa adyacente mínimo compacto

Dada una conexión, para cada elemento del lado izquierdo, se desea obtener el menor elemento adyacente del lado derecho. Es la clave para recorrer el grafo buscando representantes para cada vértice.

Dados dos mapas seccionalmente lineales $\text{lsm}_1 = (\text{dom}_1, \text{lm}_1)$, $\text{lsm}_2 = (\text{dom}_2, \text{lm}_2)$, tal que lsm_1 es compacto (es decir, sus respectivos arreglos tienen tamaño uno), y ambos mapas tienen el mismo dominio (la diferencia es que lsm_1 aplicará la misma expresión a su dominio, mientras que lsm_2 lo puede hacer por tramos). Se desea calcular un mapa seccionalmente lineal $\text{lsm}_3 = (\text{lm}_3, \text{dom}_3)$; con lm_3, tal que $\text{lm}_3(\text{v}) = \min(\text{lm}_2(\text{e}) : \text{lm}_1(\text{e}) = \text{v})$, $\text{dom}_3 = \text{Im}(\text{dom}_1 \cap \text{dom}_2, \text{lm}_1)$). Para calcular lsm_3 se aprovechan las siguientes observaciones:

- Si lm_1 es biyectivo (es decir, los multiplicadores no son ceros),
  $\text{lm}_3(\text{v}) = \min(\text{lm}_2(\text{e}) : \text{e} = \text{Inv}_{lm}(\text{v})) \Rightarrow$
  $\text{lm}_3(\text{v}) = \min(\text{lm}_2(\text{Inv}_{lm}(\text{v}))) =$
  $\text{lm}_2(\text{Inv}_{lm}(\text{v})) \Rightarrow \text{lm}_3 = \text{lm}_2 \circ \text{Inv}_{lm}(\text{lm}_1)$
- Si lm_1 es constante (es decir, los multiplicadores son iguales a cero), sea
  $\text{k} : \text{lm}_1(\text{e}) = \text{k}, \forall\ \text{e}$
  $\text{lm}_3(\text{k}) = \min(\text{lm}_2(\text{e}) : \text{lm}_1(\text{e}) = \text{k}) \Rightarrow$
  $\text{lm}_3(\text{k}) = \min(\text{lm}_2(\text{e}))$, pues la condición es una tautología.

Algoritmo 32 Mapa adyacente mínimo compacto

```
1:  function MinAdj_comp(lsm_1, lsm_2)
2:      domInv ← Im(lsm_1, dom_1)
3:      mapInv ← Inv_lm(lm_1(1))
4:      lsm_inv ← (mapInv, domInv)
5:      minIm2 ← Min(Im(lsm_2, dom_1))
6:      minDom1 ← Min(dom_1)
7:      mapInv = (m_inv, h_inv)
8:      for all m_j ∈ m_inv do
9:          if m_j = ∞ then                          ▷ El mapa original es constante
10:                                                  ▷ Se modifica el mapa inverso para poder componer
11:             m_inv(j) ← 0
12:             h_inv(j) ← minDom1(j)
13:         end if
14:     end for
```

15: res ← $\text{lsm}_2 \circ \text{lsm}_{inv}$ ▷ Primer ítem de la observación
16: res = (res_{dom}, res_{lm})
17: res_{lm} = (m_{res}, h_{res})
18: **for all** $\text{m}_j \in \text{m}_{inv}$ **do**
19: **if** $\text{m}_j = \infty$ **then** ▷ Se usa el segundo ítem de la observación
20: m_{res}(j) ← 0
21: h_{res}(j) ← minIm2(j)
22: **end if**
23: **end for**
24: **return** (res_{dom}, (m_{res}, h_{res}))
25: **end function**

Ejemplo:

- MinAdj$_{comp}$(([{{[85:1:100] x [85:1:100] x [85:1:100]}, {[150:5:200] x [150:5:200] x [85:1:100]}}], [[x, 1, x]]), ([{{[75:5:150] x [80:5:150] x [85:5:150]}}, {{[200:1:200] x [200:1:200] x [200:1:200]}}], [[3, 3, 3], [x - 10, x - 10, -10]])) = ([{{[85:5:100] x [1:1:1] x [85:5:100]}, {[150:5:150] x [1:1:1] x [85:5:100]}}], [[3, 3, 3]])

Mapa adyacente mínimo

Esta operación es una generalización de la anterior operación, para cualquier tipo de mapas seccionalmente lineales.

Dados $\text{lsm}_1 = ([\text{s}_1^1, \text{s}_1^2, ..., \text{s}_1^k], [\text{lm}_1^1, \text{lm}_1^2, ..., \text{lm}_1^k])$, $\text{lsm}_2 = ([\text{s}_2^1, \text{s}_2^2, ..., \text{s}_2^j], [\text{lm}_2^1, \text{lm}_2^2, ..., \text{lm}_2^j])$:

Algoritmo 33 Mapa adyacente mínimo

1: **function** MinAdj(lsm_1, lsm_2)
2: **if** k > 0 **then** ▷ Mapa no vacío
3: partialLSM ← ($[\text{s}_1^1]$, $[\text{lm}_1^1]$)
4: res ← MinAdj$_{comp}$(partialLSM, lsm_2)
5: **for all** c_1^q : q > 1 **do**
6: partialLSM ← ($[\text{s}_1^q]$, $[\text{lm}_1^q]$)
7: partialRes ← MinAdj$_{comp}$(partialLSM, lsm_2)
8: minMap ← Min$_{map}$(res, partialRes)
9: minMap = (minMap_{dom}, minMap_{lm})
10: res ← Combine(partialRes, res) ▷ Mínimo adyacente para partialLSM
11: **if** minMap **then**
12: res ← Combine(minMap, res) ▷ El anterior partialLSM puede ser menor en la intersección
13: **end if**
14: **end for**
15: **else**
16: res ← ([], [])
17: **end if**
18: **return** res
19: **end function**

Puede resultar extraño la necesidad de combinar los distintos resultados más de una vez (líneas 10 y 12) ¿Por qué no basta con agregar el nuevo resultado con el Combine de la línea 10? Lo que

sucede es que res puede ser menor a partialRes, en el dominio común de ambos; de allí las líneas 11 a 13.

Ejemplo:

- MinAdj(([{{[10:1:100] x [10:1:100]}, {[101:2:200] x [101:2:200]}}, {{[10:1:100] x [101:2:200]}}], [[x, x], [2 * x, 2 * x]]), ([{{[5:5:50] x [5:5:50]}}, {{[51:3:80] x [51:3:80]}, {[90:1:150] x [95:1:150]}}], [[1, x], [2 * x + 3, 2 * x + 1]])) = ([{{[180:2:200] x [202:4:298]}}, {{[101:2:149] x [101:2:149]}, {[90:1:100] x [95:1:100]}, {[51:3:80] x [51:3:80]}}, {{[10:5:50] x [10:5:50]}}], [[x + 3, x + 1], [2 * x + 3, 2 * x +1], [1, x]])

Reducción de mapas

En la línea 13 del algoritmo de componentes conexas para GBC se introduce una operación, que representa la composición del mapa argumento consigo mismo, hasta converger a un punto fijo del mismo. La reducción de mapas consiste en hallar el resultado de dicha operación, pero sin realizar la composición.

Actualmente esta operación solo permite como argumento mapas cuyas expresiones tengan ganancia igual a 1 o 0. Esto es debido a que es fácil hallar a qué mapa converge la reducción, sin necesidad de componer. Se debe recordar que cada mapa seccionalmente lineal está compuesto por un arreglo de dominios y expresiones, por lo que, si para un cierto dominio con su respectiva expresión se converge a valores que no pertenecen al dominio, esto es válido ya que dichos valores pueden pertenecer a otro dominio del mapa.

Se analiza el caso unidimensional, pues las expresiones son independientes una de las otras en cada dimensión.

- Si m = 1, y h = 0, se trata de la identidad, por lo cual la expresión converge a sí misma.
- Si m = 0, se converge a la expresión resultante con una única iteración.
- Para el caso en que m = 1, y h $\neq$ 0, llamando a la expresión f(x) = x + h, se utilizan tres lemas (las demostraciones se encuentran en el anexo B):
  - Dado el intervalo [lo:st:hi], y h $\in \mathbb{Z}$, si st divide a h, entonces [lo:st:hi] = [lo:|h|:hi] $\cup$ [lo + st:|h|:hi] $\cup$ [lo + 2 * st:|h|:hi] $\cup$ ... $\cup$ [lo + ($\frac{|h|}{st}$ - 1) * st:|h|:hi]
  - Dado el intervalo [lo:h:hi] = dom(f), donde h > 0, se tiene que dom(f^i) = [lo:h:hi - h * (i - 1)] $\wedge$ im(f^i) = [lo + h * i:h:hi + h]
  - Dado el intervalo [lo:|h|:hi] = dom(f), donde h < 0, se tiene que dom(f^i) = [lo + |h| * (i - 1):|h|:hi] $\wedge$ im(f^i) = [lo + h:|h|:hi + h * i]

  Los dos últimos lemas nos indican que después de $\frac{hi-lo}{|h|} + 1$ iteraciones el dominio será vacío, y explicitan la imagen a la que convergen las iteraciones (por ejemplo, en el segundo lema se converge a [hi + h:h:hi + h], lo que puede ser descripto mediante una expresión constante). Por este motivo, dado el intervalo original, podemos subdividirlo en subdominios con el primer lema, y para cada subdominio se devuelve la expresión a la que converge con el segundo o tercer lema.

La operación de reducción trabajará sobre un mapa, en una única dimensión (esto para simplificar su implementación). Es por eso que recibe como argumentos un mapa seccionalmente lineal lsm = ([s^1, s^2, ..., s^k], [lm^1, lm^2, ..., lm^k]), y un natural d que indique la dimensión sobre la cual trabajar:

Algoritmo 34 Reducción de intervalos

```
1: function ReduceMapN(lsm, d)
2:     res_dom ← [s^1, s^2, ..., s^k]
3:     res_lm ← [lm^1, lm^2, ..., lm^k]
4:     for all lm^q do                                  ▷ Se recorren todas las expresiones y dominios
5:         lm^q = ([m_1^q, m_2^q, ..., m_j^q], [h_1^q, h_2^q, ..., h_j^q])
6:         if m_d^q = 1 ∧ h_d^q ≠ 0 then                ▷ Se modifica la d-ésima dimensión
7:             if h | st then
8:                 off ← |h_d^q| / st
9:                 for all as ∈ s^q do                  ▷ as es un conjunto atómico
10:                    as = {i^1 x i^2 x ... x i^j}
11:                    i^d = [lo:st:hi]
12:                    if hi - lo > off * off then
13:                        aux_dom ← []                 ▷ aux_dom es un arreglo de conjuntos
14:                        aux_lm ← []                  ▷ aux_lm es un arreglo de expresiones
15:                        for t ← 0; t < off; t ← t + 1 do
16:                            aux_m ← [m_1^q, m_2^q, ..., m_{d-1}^q, 0, m_{d+1}^q, ..., m_j^q]
17:                            new_h ← lo + t * st + h_d^q
18:                            if h > 0 then
19:                                new_h ← hi - (hi - lo) % st + h_d^q   ▷ % es el resto
20:                            end if
21:                            aux_h ← [h_1^q, h_2^q, ..., h_{d-1}^q, new_h, h_{d+1}^q, ..., h_j^q]
22:                            aux_i ← [lo + t * st:|h|:hi]
23:                            aux_s ← {{i^1 x i^2 x .. x i^{d-1} x aux_i x i^{d+1} x ... x i^j}}
24:                            aux_dom ← aux_dom ++ [aux_s]
25:                            aux_lm ← aux_lm ++ [(aux_m, aux_h)]
26:                        end for
27:                        if #s^q = 1 then
28:                            res_dom ← [s^1, s^2, ..., s^{q-1}, s^{q+1}, s^{q+2}, ..., s^k]
29:                            res_lm ← [lm^1, lm^2, ..., lm^{q-1}, lm^{q+1}, lm^{q+2}, ..., s^k]
30:                        else
31:                            aux_s ← s^q \ {as}
32:                            res_dom ← [s^1, s^2, ..., s^{q-1}, aux_s, s^{q+1}, s^{q+2}, ..., s^k]
33:                        end if
34:                        res_dom ← res_dom ++ aux_dom
35:                        res_lm ← res_lm ++ aux_lm
36:                    end if
37:                end for
38:            end if
39:        end if
40:    end for
41:    return (res_dom, res_lm)
42: end function
```

El if de la línea 10 está presente, pues si el off definido en la línea 6 es muy grande, se particionará el dominio en muchos subdominios, y será más costoso que realizar unas pocas composiciones (como el off es grande, los valores se irán fuera de dominio rápidamente).

Cabe destacar que para la aplicación particular de búsqueda de componentes conexas, las expresiones serán del tipo x - h (pues el mapeo es a representantes, que son los menores elementos de cada componente conexa). Se ha trabajado de manera general, pensando en que la librería pueda aplicarse en otros casos.

Ejemplo (notar que en la dimensión 2 se puede reducir con esta operación, pero en las dimensiones 1 y 3 no sería posible):

- ReduceMapN(([{{[4:1:15] x [4:1:15] x [4:1:15]}, {[4:1:15] x [20:2:25] x [4:1:15]}}, {{[4:1:15] x [15:5:50] x [20:2:25]}, {[20:2:25] x [40:5:45] x [40:5:45]}}], [[x, x - 3, 3 * x], [2 * x, x, 4 * x + 4]]), 2) = ([{{[4:1:15] x [4:3:15] x [4:1:15]}}, {{[4:1:15] x [5:3:15] x [4:1:15]}}, {{[4:1:15] x [6:3:15] x [4:1:15]}}, {{[4:1:15] x [20:2:25] x [4:1:15]}}, {{[4:1:15] x [15:5:50] x [20:2:25]}, {[20:2:25] x [40:5:45] x [40:5:45]}}], [[x, 1, 3 * x], [x, 2, 3 * x], [x, 3, 3 * x], [x, x - 3, 3 * x], [2 * x, x, 4 * x + 4]])

Mapa infinito

La idea es obtener la sucesiva aplicación de un mapa a sí mismo, hasta encontrar un punto fijo, pero sin realizar en verdad dichas aplicaciones. Para ello, la operación aplicará la reducción a todas las dimensiones. Luego, se calculan las iteraciones requeridas para converger (recordar que si el h es muy grande, ReduceMapN no hará cambios, por lo que es necesario iterar).

Dado un mapa seccionalmente lineal lsm = ([s^1, s^2, ..., s^k], [lm^1, lm^2, ..., lm^k]):

Algoritmo 35 Mapa infinito

```
1:  function MapInf(lsm)
2:      (res_dom, res_lm) ← ReduceMapN(lsm, 1)
3:      for i ← 2; i ≤ k; i ← i + 1 do
4:          (res_dom, res_lm) ← ReduceMapN((res_dom, res_lm), i)
5:      end for
6:      maxit ← 0
7:      for g ← 1; g ≤ k; g ← g + 1 do
8:          lm^g = ([m_1^g, m_2^g, ..., m_j^g], [h_1^g, h_2^g, ..., h_j^g])
9:          a ← max(x) / x = m_f^g * |h_f^g|, 1 ≤ f ≤ j
10:         if a > 0 then                          ▷ Algún dominio no fue reducido
11:             its ← 0
12:             for dim ← 0; dim < j; dim ← dim + 1 do
13:                 if m_d^i im = 1 ∧ h_d^i im ≠ 0 then
14:                     for all as ∈ s^i do
15:                         as = {i^1 x i^2 x ... x i^j}
16:                         i^dim = [lo:st:hi]
17:                         its ← max(its, ⌈(hi−lo)/|h_dim^g|⌉)
18:                     end for
19:                 end if
20:             end for
21:             maxit ← maxit + its
22:         else
23:             b ← min(m_f^g), 1 ≤ f ≤ j
24:             if b = 0 then
25:                 maxit ← maxit + 1
```

```
            end if
        end if
    end for
    maxit ← ⌊log2maxit + 1⌋
    for i ← 0; i < maxit; i ← i + 1 do
        (res_dom, res_lm) ← (res_dom, res_lm) ∘ (res_dom, res_lm)
    end for
    return (res_dom, res_lm)
end function
```

Se debe recordar que para aplicar ReduceMapN se han impuesto restricciones sobre las expresiones de los mapas que recibe como argumento. Estas restricciones también aseguran que la aplicación del mapa infinito no diverge, pues se ha listado a qué convergen los tres tipos de expresiones permitidas.

Capítulo 5

Aplanado de modelos Modelica

Una vez que se cuenta con el nuevo enfoque, es necesario explicar cómo se ha aplicado a la etapa de aplanado de modelos Modelica. De este modo, de aquí en más, se estará haciendo referencia a la implementación de GBCs (y demás estructuras definidas).

Esto se ha aplicado en ModelicaCC, pero, en teoría, podría ser utilizado en otras herramientas.

Se debe recordar que la primera parte del aplanado se encontraba resuelta anteriormente, por lo que en esta sección se detalla cómo es que se eliminan los connect de un modelo Modelica en ModelicaCC.

Para generar las ecuaciones que corresponden, se sub-divide el problema en dos pasos: primero se construirá el GBC, se buscan las componentes conexas, y en base al resultado se generan las ecuaciones.

El modelo que llega como argumento a esta etapa ya se encuentra parseado, y las jerarquías se han eliminado. La estructura del objeto argumento es recursiva, por lo que se puede ir recorriendo componente por componente, para tratar los connect. Puede pensarse al argumento como un AST modificado para eliminar ciertas jerarquías.

5.1. Construcción del grafo

Restricciones

El enfoque desarrollado de GBCs impone ciertas restricciones en los modelos Modelica que podrán ser representados a través del mismo:

- Si hay loops anidados, cada contador es independiente de los demás. Cada arista-conjunto está compuesta por dos mapas seccionalmente lineales; es decir, las expresiones implicadas son lineales. En caso de permitir contadores dependientes, sería necesario permitir expresiones no lineales.
- El otro factor que condiciona las expresiones de los mapas seccionalmente lineales de las aristas-conjunto es la expresión de los índices. Dichas expresiones deben ser lineales, para poder ser representadas.

Estas restricciones no son taxativas. Es decir, un modelo que permita estas construcciones podría ser representado por el nuevo enfoque. Sin embargo, sería necesario particionar los vértices-conjunto, y las aristas-conjunto para trabajar de forma extensiva. Esto no ha sido implementado, por lo que los modelos que no cumplan las propiedades no serán aceptados.

Por ejemplo, el siguiente modelo no será aceptado:

```
1  model RCNetwork
2    Real N = 1000;
3    SignalVoltage S;
4    Ground G;
5    Resistor R[N];
6    Capacitor C[N];
7  equation
8    connect(S.p, R[1].p);
9    connect(S.n, G.p);
10
11   for i in 1:N-1 loop
12     for j in 1:i loop
13       connect(R[i].n, R[j].p);
14     end for;
15   end for;
16
17   for i in 1:N loop
18     connect(C[i].p, R[i].n);
19     connect(C[i * i].n, G.p);
20   end for;
21 end RCNetwork;
```

Listado 5.1: Modelo no representable

Se puede observar que en la línea 12 se declara un loop con un contador dependiente del contador del loop de la línea 12. Por otro lado, en el connect de la línea 19, una de las expresiones de los índices es i * i, lo cual no es lineal.

Si se quisiera representar el connect de la línea 13, se podrían crear dos mapas seccionalmente lineales para las aristas-conjunto, cada uno con 999 dominios y sus respectivas expresiones. Esto sería equivalente a trabajar de manera escalar.

Además, la operación de reducción introdujo restricciones adicionales. En particular, si hay un connect(v[m_1^1 * i_1 + h_1^1, ..., m_k^1 * i_k + h_k^1], w[m_1^2 * j_1 + h_1^2, ..., m_k^2 * j_k + h_k^2]), debe cumplirse que $(m_q^1 \neq m_q^2 \Rightarrow m_q^1 = 0 \vee m_q^2 = 0) \wedge i_q = j_q$, $1 \leq q \leq k$. Actualmente se está trabajando en generalizar un poco más esta operación.

Construcción del grafo

No se entrará en detalles de implementación respecto de este proceso, pues sería necesario detallar cuestiones de la etapa de *parsing*. A grandes rasgos, los pasos son:

- Se recorre la sección de ecuaciones recursivamente, buscando los connect. Si elemento recorrido es un bucle, se guarda en un estado global la información acerca del contador del mismo.
- Al llegar a un connect se chequea que los conectores que lo componen cumplan con las restricciones necesarias. En caso de que sí las cumplan, se crea un vértice-conjunto para cada variable de los arreglos de conectores que componen al connect (si no había sido creado). En este paso se tienen en cuenta variables de flujo y esfuerzo (ya que se debe recordar que no deben conectarse entre sí). Las demás construcciones no son modificadas.
- Luego, se chequea si existe una arista-conjunto que una ambos vértices-conjunto. En caso de que no, la misma se crea, detallando las aristas que la componen, y si ya existía, se agregan las nuevas aristas.
- Se prosigue repitiendo los pasos anteriores, hasta recorrer todo el modelo.

Lo que recibe la etapa de aplando de ModelicaCC es un AST, que primeramente se modifica, luego del *parsing* para eliminar jerarquías. Ahora se deben tratar los connect. Para ello, se recorre la sección de ecuaciones, que en el AST es una lista de ecuaciones. Además del AST, se cuenta con un entorno donde se encuentra guardada información respecto del estado de variables, información de las variables, etc.

Dada la lista de ecuaciones eqs = [eq_1, eq_2, ..., eq_n], del modelo $\mathcal{M}$, y un estado s, el proceso (de manera muy simplificada) para construir el grafo es el siguiente:

Algoritmo 36 Creación del GBC a partir del modelo Modelica

```
1: function MakeSBG(eqs, s)
2:     V ← {}
3:     E ← {}
4:     for i ← 1; i ≤ n; i ← i + 1 do
5:         if IsConnect(eq^i) then
6:             eq^i = connect(left, right)
7:             left = c1[e1, e2, ..., ej]                ▷ c1 es instancia de algún conector
8:             right = c2[f1, f2, ..., fk]               ▷ c2 es instancia de algún conector
9:             validconnect ← true
10:            allvars ← Vars(s)
11:            if j = k then
12:                for q ← 1; q ≤ j; q ← q + 1 do
13:                    for all v1 ∈ allvars do      ▷ Las expresiones de los índices deben ser lineales
14:                        for all v2 ∈ allvars do
15:                            if IsIn(v1, e_q) ∧ IsIn(v2, e_q) ∧ v1 ≠ v2 then
16:                                validconnect ← false
17:                            end if
18:                            if IsIn(v1, f_q) ∧ IsIn(v2, f_q) ∧ v1 ≠ v2 then
19:                                validconnect ← false
20:                            end if
21:                        end for
22:                    end for
23:                end for
24:                    ▷ Por los chequeos realizados, las expresiones de los índices de c1 y c2 son expresiones lineales
25:                for r ← 1; r ≤ j; r ← r + 1 do
26:                    e_r = m1 * counter_r^1 + h1
27:                    f_r = m2 * counter_r^2 + h2
28:                    if m1 ≠ m2 ∧ m1 ≠ 0 ∧ m2 ≠ 0 then ▷ Condición impuesta por la reducción
29:                        validconnect ← false
30:                    end if
31:                end for
32:                if validconnect then
33:                    V ← AddVertex(left, V)
34:                    V ← AddVertex(right, V)
35:                    E ← AddEdge(left, right, E)
36:                else
37:                    V ← {}
38:                    E ← {}
```

```
39:                 break
40:             end if
41:         else
42:             V ← {}
43:             E ← {}
44:             break
45:         end if
46:     else if IsFor(eq^i) then
47:         counters ← VarsEq(eq^i)
48:         s' ← SaveIndexes(counters, s)
49:         if s' ≠ ∅ then
50:             foreqlist ← GetEquations(eq^i)
51:             MakeSBG(foreqlist, s')
52:         else
53:             return ({}, {})
54:         end if
55:     end if
56: end for
57: return (V, E)
58: end function
```

A continuación se presenta un esbozo de SaveIndex, ya que en esta función es donde se chequea que los contadores de los loops anidados sean independendientes unos de los otros. Dada una ecuación for eq, y un estado s:

Algoritmo 37 Información de los contadores

```
1: function SaveIndexes(counters, s)
2:     if counters = [] then
3:         return ∅
4:     else
5:         counters = [c_1, c_2, ..., c_n]
6:         c_1 = (v, e)
7:         evale ← EvalLinearExp(e, s)
8:         if evale ≠ none then
9:             s' ← SaveIndex(v, evale, s)
10:            SaveIndexes([c_2, c_3, ..., c_n], s')
11:        else
12:            return ∅
13:        end if
14:    end if
15: end function
```

Coloquialmente, se describe el comportamiento de las funciones utilizadas:

- IsConnect: devuelve un booleano para indicar si la ecuación argumento es un connect o no.
- Vars: devuelve un conjunto que contiene a todas las variables que se encuentran en el entorno argumento.

- IsIn: chequea si la expresión pasada como primer argumento hace uso de la variable pasada como segundo argumento.
- IsFor: devuelve un booleano para indicar si la ecuación argumento es una ecuación for o no.
- SaveIndexes: dada una ecuación for, guarda agrega la información referida a los contadores del bucle (nombre, valor inicial, salto, y cota superior) al estado pasado como segundo argumento, y devuelve el nuevo estado resultante.
- GetEquations: dada una ecuación for, obtiene la lista de ecuaciones que componen a la misma.
- VarsEq: dada una ecuación for como argumento, devuelve una lista de duplas (v, e), donde cada v es el nombre de cada contador, y e la expresión que determina el valor de v en cada iteración.
- VarsExpr: dada una expresión como argumento, devuelve un conjunto de variables usadas en la expresión.
- EvalLinearExp: evalúa una expresión lineal e, en el estado s. En caso de que e no sea lineal, se devuelve none.
- SaveIndex: guarda la variable v, con el valor evale en el entorno s.
- AddVertex: agrega un vértice-conjunto al conjunto de vértices-conjunto pasado como segundo argumento. Es necesario procesar el primer argumento para crear el vértice-conjunto a insertar, y por este motivo se utiliza esta función, en vez de directamente usar la unión de conjuntos. Esta función chequea si el vértice-conjunto ya pertenecía al conjunto o no. En caso afirmativo, no agrega nada.
- AddEdge: agrega una arista-conjunto al conjunto de aristas-conjuntos pasado como tercer argumento. Del mismo modo que en AddVertex, es necesario procesar los dos primeros argumentos para agregar la arista-conjunto. Además, esta función chequea si la arista-conjunto ya pertenecía al conjunto o no. Si ya se encontraba en el conjunto, agrega las nuevas conexiones que aporte el connect, caso contrario, la crea con los datos de left y right.

Ejemplo

Se modelará el siguiente circuito eléctrico de dos dimensiones, conformado por N x M celdas, cada una de las cuales cuenta con 4 conectores (izquierda, derecha, arriba, abajo), descarga a tierra, y una fuente:

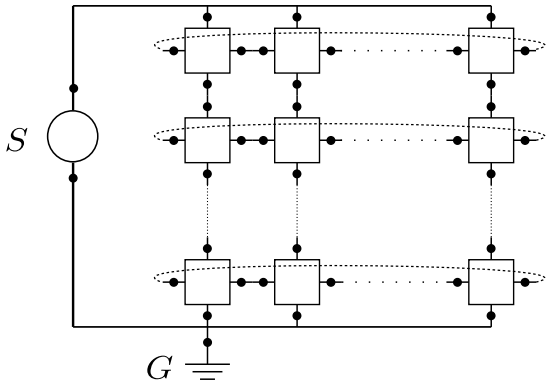


Imagen 5.1: *Red 2D*

En Modelica puede ser modelado del siguiente modo (fijando N = 1000, M = 1000):

```
model Test
  connector Electrical
    Real eff;
    flow Real fl;
  end Electrical;

  model Obj
    Electrical n, p;
  end Obj;

  model Cell
    Electrical u, d, l, r;
  end Cell;

  Real N = 1000;
  Real M = 1000;
  Obj S[1, 1];
  Obj G[1, 1];
  Cell C[N, M];
equation
  connect(S.n, G.p);

  for i in 1:N, j in 1:M-1 loop
    connect(C[i, j + 1].l, C[i, j].r);
  end for;

  for j in 1:M loop
    connect(S.p, C[1, j].u);
    connect(C[N, j].d, G.p);
  end for;

  for i in 1:N-1, j in 1:M loop
    connect(C[i + 1, j].u, C[i, j].d);
  end for;

  for i in 1:N loop
    connect(C[i, 1].l, C[i, M].r);
  end for;
```

```
39 end Test;
```

Listado 5.2: Red de capacitores y resistencias 2D

Luego de la primer etapa de aplanado (eliminación de jerarquías y demás), el modelo resultante es el siguiente:

```
1  model Test
2    Real N=1000;
3    Real M=1000;
4    Real S_n_eff[1, 1];
5    flow Real S_n_fl[1, 1];
6    Electrical S_n[1, 1];
7    Real S_p_eff[1, 1];
8    flow Real S_p_fl[1, 1];
9    Electrical S_p[1, 1];
10   Real G_n_eff[1, 1];
11   flow Real G_n_fl[1, 1];
12   Electrical G_n[1, 1];
13   Real G_p_eff[1, 1];
14   flow Real G_p_fl[1, 1];
15   Electrical G_p[1, 1];
16   Real C_u_eff[N, M];
17   flow Real C_u_fl[N, M];
18   Electrical C_u[N, M];
19   Real C_d_eff[N, M];
20   flow Real C_d_fl[N, M];
21   Electrical C_d[N, M];
22   Real C_l_eff[N, M];
23   flow Real C_l_fl[N, M];
24   Electrical C_l[N, M];
25   Real C_r_eff[N, M];
26   flow Real C_r_fl[N, M];
27   Electrical C_r[N, M];
28 equation
29     connect(S_n , G_p);
30   for i in 1:N,j in 1:M-1 loop
31         connect(C_l[i,j+1] , C_r[i,j]);
32   end for;
33   for j in 1:M loop
34         connect(S_p[1,1] , C_u[1,j]);
35         connect(C_d[N,j] , G_p[1,1]);
36   end for;
37   for i in 1:N-1,j in 1:M loop
38         connect(C_u[i+1,j] , C_d[i,j]);
39   end for;
40   for i in 1:N loop
41         connect(C_l[i,1] , C_r[i,M]);
42   end for;
43 end Test;
```

Listado 5.3: Red de capacitores y resistencias 2D, parcialmente aplanado

El GBC resultante es $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, donde (notar que los nombres de los vértices fueron elegidos para que sean análogos a los del modelo):

- $\mathcal{V} = \{S_n, G_p, C_l, C_r, S_p, C_u, C_d\}$
  - $S_n = \{\{[1{:}1{:}1] \text{ x } [1{:}1{:}1]\}\}$

- G_p = {{[2:1:2] x [2:1:2]}}
- C_l = {{[3:1:1002] x [3:1:1002]}}
- C_r = {{[1003:1:2002] x [1003:1:2002]}}
- S_p = {{[2003:1:2003] x [2003:1:2003]}}
- C_u = {{[2004:1:3003] x [2004:1:3003]}}
- C_d = {{[3004:1:4003] x [3004:1:4003]}}

▪ $\mathcal{E}$ = {E_0, E_1, E_2, E_3, E_4, E_5}

E_0 representa la conexión de S_n con G_p.

E_1 representa las conexiones interiores de los conectores izquierdos con los conectores derechos de las celdas.

E_2 representa las conexiones de los conectores superiores con la fuente.

E_3 representa las conexiones de los conectores de abajo con la descarga a tierra.

E_4 representa las conexiones entre los conectores de abajo y los de arriba.

E_5 representa las conexiones de los conectores izquierdos de la primera columna con los conectores derechos de la última columna.

- E_0^{left} = ([{{[1:1:1] x [1:1:1]}}], [0 * x + 1, 0 * x + 1])
- E_0^{right} = ([{{[1:1:1] x [1:1:1]}}], [0 * x + 2, 0 * x + 2])
- E_1^{left} = ([{{[2:1:1001] x [2:1:1000]}}], [1 * x + 1, 1 * x + 2])
- E_1^{right} = ([{{[2:1:1001] x [2:1:1000]}}], [1 * x + 1001, 1 * x + 1001])
- E_2^{left} = ([{{[1002:1:1002] x [1001:1:2000]}}], [0 * x + 2003, 0 * x + 2003])
- E_2^{right} = ([{{[1002:1:1002] x [1001:1:2000]}}], [0 * x + 2004, 1 * x + 1003])
- E_3^{left} = ([{{[1003:1:1003] x [2001:1:3000]}}], [0 * x + 4003, 1 * x + 1003])
- E_3^{right} = ([{{[1003:1:1003] x [2001:1:3000]}}], [0 * x + 2, 0 * x + 2])
- E_4^{left} = ([{{[1004:1:2002] x [3001:1:4000]}}], [1 * x + 1001, 1 * x - 997])
- E_4^{right} = ([{{[1004:1:2002] x [3001:1:4000]}}], [1 * x + 2000, 1 * x + 3])
- E_5^{left} = ([{{[2003:1:3002] x [4001:1:4001]}}], [1 * x - 2000, 0 * x + 3])
- E_5^{right} = ([{{[2003:1:3002] x [4001:1:4001]}}], [1 * x - 1000, 0 * x + 2002])

Es importante aclarar que durante toda la etapa las funciones utilizadas trabajan con vértices-conjunto y aristas-conjunto por separado, de manera tal que no surgen problemas entre los dominios de los mapas seccionalmente lineales, y los *conjuntos* que conforman los vértices-conjunto (podría pensarse que se trata de distintos dominios).

5.2. Generación de ecuaciones

Habiendo tenido el GBC para el modelo correspondiente, y habiendo hallado sus componentes conexas, cada componente conexa es un conjunto conexo tal como se ha descripto en 2.1.3. Por ende, para cada componente conexa:

- Se distingue si es de variables de esfuerzo o flujo.
- Se generan las ecuaciones que correspondan.

Para el ejemplo de la sección anterior, el resultado del algoritmo de componentes conexas es el siguiente:

[({{[1003:1:2002] x [2002:1:2002]}}, [1 * x + -1000, 0 * x + 3]) ,
({{[3:1:1002] x [3:1:3]}}, [1 * x + 0, 1 * x + 0]),
({{[3004:1:4002] x [3004:1:4003]}}, [1 * x + -999, 1 * x + -1000]),
({{[2005:1:3003] x [2004:1:3003]}}, [1 * x + 0, 1 * x + 0]),
({{[2004:1:2004] x [2004:1:3003]}}, [0 * x+ 2003, 0 * x + 2003]),
({{[2003:1:2003] x [2003:1:2003]}}, [1 * x + 0, 1 * x + 0]),
({{[1003:1:2002] x [1003:1:2001]}}, [1 * x + -1000, 1 * x + -999]),
({{[3:1:1002] x [4:1:1002]}}, [1 * x + 0, 1 * x + 0]),
({{[4003:1:4003] x [3004:1:4003]}}, [0 * x + 1, 0 * x + 1]),
({{[2:1:2] x [2:1:2]}}, [0 * x + 1, 0 * x + 1]),
({{[1:1:1] x [1:1:1]}}, [1 * x + 0, 1 * x + 0])]

Por ejemplo, ({{[1003:1:2002] x [2002:1:2002]}}, [1 * x + -1000, 0 * x + 3]), es equivalente a decir que C_r[i][1000] será representado por C_l[i][1], con $1 \leq i \leq 1000$.

Y finalmente, el modelo aplanado:

```
1  model Test
2    Real N=1000;
3    Real M=1000;
4    Real S_n_eff[1, 1];
5    flow Real S_n_fl[1, 1];
6    Real S_p_eff[1, 1];
7    flow Real S_p_fl[1, 1];
8    Real G_n_eff[1, 1];
9    flow Real G_n_fl[1, 1];
10   Real G_p_eff[1, 1];
11   flow Real G_p_fl[1, 1];
12   Real C_u_eff[N, M];
13   flow Real C_u_fl[N, M];
14   Real C_d_eff[N, M];
15   flow Real C_d_fl[N, M];
16   Real C_l_eff[N, M];
17   flow Real C_l_fl[N, M];
18   Real C_r_eff[N, M];
19   flow Real C_r_fl[N, M];
20 equation
21   for i in 1:1:1,j in 1:1:1 loop
22     G_p_eff = S_n_eff;
23   end for;
24   for i in 1000:1:1000,j in 1:1:1000 loop
25     C_d_eff[i,j] = S_n_eff;
26   end for;
27   for i in 1:1:1,j in 1:1:1 loop
28     S_n_fl+sum(C_d_fl[1*i+999, 1:1:1000])+G_p_fl = 0;
29   end for;
30   for i in 1:1:1000,j in 1:1:999 loop
31     C_r_eff[i,j] = C_l_eff[1*i+0,1*j+1];
32   end for;
33   for i in 1:1:1000,j in 2:1:1000 loop
```

```
34      C_r_fl[1*i+0,1*j+-1]+C_l_fl[1*i+0,1*j+0] = 0;
35    end for;
36    for i in 1:1:1,j in 1:1:1000 loop
37      C_u_eff[i,j] = S_p_eff;
38    end for;
39    for i in 1:1:1,j in 1:1:1 loop
40      sum(C_u_fl[1*i+0, 1:1:1000])+S_p_fl = 0;
41    end for;
42    for i in 1:1:999,j in 1:1:1000 loop
43      C_d_eff[i,j] = C_u_eff[1*i+1,1*j+0];
44    end for;
45    for i in 2:1:1000,j in 1:1:1000 loop
46      C_d_fl[1*i+-1,1*j+0]+C_u_fl[1*i+0,1*j+0] = 0;
47    end for;
48    for i in 1:1:1000,j in 1000:1:1000 loop
49      C_r_eff[i,j] = C_l_eff[1*i+0,1*j+-999];
50    end for;
51    for i in 1:1:1000,j in 1:1:1 loop
52      C_r_fl[1*i+0,1*j+999]+C_l_fl[1*i+0,1*j+0] = 0;
53    end for;
54  end Test;
```

Continuando con el ejemplo, en las líneas 48 a 53 se puede observar las ecuaciones generadas por la componente conexa ({{[1003:1:2002] x [2002:1:2002]}}, [1 * x + -1000, 0 * x + 3]).

En particular, para las variables de flujo puede suceder que todos los elementos de un vértice-conjunto sean representados por el mismo representante (conexión constante). Esto implica expresar una sumatoria de todos los nodos del vértice-conjunto. Obviamente, se desea evitar el desenrrollado del vértice-conjunto, por lo que se introdujo una nueva construcción, que no formaba parte de las expresiones que volvía como resultado el parse: AddAll.

AddAll guarda la información de los índices de la sumatoria, y la idea es que etapas posteriores del proceso de compilación sepan distinguirla, y puedan escribir las ecuaciones correspondientes sin expandir.

Capítulo 6

Experimentos computacionales

Se presentarán tres ejemplos, para los cuales se detallará el modelo Modelica que se obtiene como resultado de la etapa de aplanado, y el tiempo de ejecución para la segunda parte de dicha etapa, variando el tamaño N (y M, para el caso de dos dimensiones) de los arreglos de conectores implicados. Las pruebas se realizaron en un equipo con las siguientes características:

- Procesador: Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz
- Memoria: 8GB
- Sistema operativo: Ubuntu 16.04.6

Para un cierto N se realizaron diez ejecuciones, calculando el promedio de las mismas. Como límite de tiempo de ejecución se fijará una hora, para completar la etapa de aplanado.

Se podrá observar que, aunque se incrementa N, el tiempo necesario para completar la ejecución prácticamente no varía.

Para medir el tiempo de ejecución se usó la función clock() en C++, de la cabecera time.h, que mide la cantidad de ticks del CPU desde su inicio.

6.1. Instancia 1

```
1  model Test
2    connector Electrical
3      Real eff;
4      flow Real fl;
5    end Electrical;
6
7    model Obj
8      Electrical n, p;
9    end Obj;
10
11   Real N = 1000;
12   Obj S;
13   Obj R[N];
14   Obj G;
15   Obj C[N];
16 equation
17   connect(S.p, R[1].p);
```

```
  connect(S.n, G.p);

  for i in 1:N-1 loop
    connect(R[i].n, R[i + 1].p);
  end for;

  for i in 1:N loop
    connect(C[i].p, R[i].n);
    connect(C[i].n, G.p);
  end for;
end Test;
```

Listado 6.1: Ejemplo 1: red de resistores y capacitores

```
model Test
  Real N=1000;
  Real S_n_eff;
  flow Real S_n_fl;
  Real S_p_eff;
  flow Real S_p_fl;
  Real R_n_eff[N];
  flow Real R_n_fl[N];
  Real R_p_eff[N];
  flow Real R_p_fl[N];
  Real G_n_eff;
  flow Real G_n_fl;
  Real G_p_eff;
  flow Real G_p_fl;
  Real C_n_eff[N];
  flow Real C_n_fl[N];
  Real C_p_eff[N];
  flow Real C_p_fl[N];
equation
  for i in 1:1:1 loop
    G_p_eff = S_n_eff;
  end for;
  for i in 1:1:1000 loop
    C_n_eff[i] = S_n_eff;
  end for;
  for i in 1:1:1 loop
    S_n_fl+sum(C_n_fl[1:1:1000])+G_p_fl = 0;
  end for;
  for i in 1000:1:1000 loop
    C_p_eff[i] = R_n_eff[1*i+0];
  end for;
  for i in 1000:1:1000 loop
    C_p_fl[1*i+0]+R_n_fl[1*i+0] = 0;
  end for;
  for i in 1:1:999 loop
    C_p_eff[i] = R_p_eff[1*i+1];
  end for;
  for i in 1:1:999 loop
    R_n_eff[i] = R_p_eff[1*i+1];
  end for;
  for i in 2:1:1000 loop
    R_n_fl[1*i+-1]+R_p_fl[1*i+0]+C_p_fl[1*i+-1] = 0;
  end for;
  for i in 1:1:1 loop
```

```
    R_p_eff[i] = S_p_eff;
  end for;
  for i in 1:1:1 loop
    S_p_fl+R_p_fl[1*i+0] = 0;
  end for;
end Test;
```

Listado 6.2: Ejemplo 1, aplanado

A continuación se presenta una tabla de comparación entre ModelicaCC, y OpenModelica (en segundos):

	N = 1000	N = 10000	N = 100000	N = 1000000
ModelicaCC	0.0308303	0.0284611	0.0259772	0.029076699999999994
OpenModelica	3.227	308.044	-	-

Tabla 6.1: *Resultados, instancia 1*

Se puede observar que para OpenModelica el costo aumenta, a medida que aumenta el tamaño del modelo. Para N = 100000 y N = 1000000, OpenModelica no completa la ejecución dentro del tiempo límite.

6.2. Instancia 2

```
model Test
  connector Electrical
    Real eff;
    flow Real fl;
  end Electrical;

  model Obj
    Electrical n, p;
  end Obj;

  Real N = 1000;
  Obj S;
  Obj R[N];
  Obj G;
  Obj C[N];
equation
  connect(S.p, R[1].p);
  connect(S.n, G.p);

  for i in 1:N-1 loop
    connect(R[i].n, R[i + 1].p);
  end for;

  for i in 1:N loop
    connect(C[i].p, R[i].n);
    connect(C[i].n, G.p);
  end for;

  for i in 1:N-1 loop
    connect(C[i].n, C[i + 1].n);
  end for;
```

```
32 end Test;
```

Listado 6.3: Ejemplo 2: red de resistores y capacitores

```
1  model Test
2    Real N=1000;
3    Real S_n_eff;
4    flow Real S_n_fl;
5    Real S_p_eff;
6    flow Real S_p_fl;
7    Real R_n_eff[N];
8    flow Real R_n_fl[N];
9    Real R_p_eff[N];
10   flow Real R_p_fl[N];
11   Real G_n_eff;
12   flow Real G_n_fl;
13   Real G_p_eff;
14   flow Real G_p_fl;
15   Real C_n_eff[N];
16   flow Real C_n_fl[N];
17   Real C_p_eff[N];
18   flow Real C_p_fl[N];
19 equation
20   for i in 1:1:1 loop
21     G_p_eff = S_n_eff;
22   end for;
23   for i in 1:1:1 loop
24     C_n_eff[i] = S_n_eff;
25   end for;
26   for i in 1000:1:1000 loop
27     C_n_eff[i] = S_n_eff;
28   end for;
29   for i in 2:1:999 loop
30     C_n_eff[i] = S_n_eff;
31   end for;
32   for i in 1:1:1 loop
33     S_n_fl+sum(C_n_fl[2:1:999])+C_n_fl[1*i+999]+C_n_fl[1*i+0]+G_p_fl = 0;
34   end for;
35   for i in 1000:1:1000 loop
36     C_p_eff[i] = R_n_eff[1*i+0];
37   end for;
38   for i in 1000:1:1000 loop
39     C_p_fl[1*i+0]+R_n_fl[1*i+0] = 0;
40   end for;
41   for i in 1:1:999 loop
42     C_p_eff[i] = R_p_eff[1*i+1];
43   end for;
44   for i in 1:1:999 loop
45     R_n_eff[i] = R_p_eff[1*i+1];
46   end for;
47   for i in 2:1:1000 loop
48     R_n_fl[1*i+-1]+R_p_fl[1*i+0]+C_p_fl[1*i+-1] = 0;
49   end for;
50   for i in 1:1:1 loop
51     R_p_eff[i] = S_p_eff;
52   end for;
53   for i in 1:1:1 loop
54     S_p_fl+R_p_fl[1*i+0] = 0;
```

```
  end for;
end Test;
```

Listado 6.4: Ejemplo 2, aplanado

	N = 1000	N = 10000	N = 100000	N = 1000000
ModelicaCC	0.0377328	0.0400227	0.426934	0.389899
ModelicaCC	3.394	423.326	-	-

Tabla 6.2: *Resultados, instancia 2*

Nuevamente se observa que para N = 100000 y N = 1000000, OpenModelica no completa la ejecución dentro del tiempo límite.

6.3. Instancia 3

```
model Test
  connector Electrical
    Real eff;
    flow Real fl;
  end Electrical;

  model Obj
    Electrical n, p;
  end Obj;

  model Cell
    Electrical u, d, l, r;
  end Cell;

  Real N = 1000;
  Real M = 100;
  Obj S[1, 1];
  Obj G[1, 1];
  Cell C[N, M];
equation
  connect(S.n, G.p);

  for i in 1:N, j in 1:M-1 loop
    connect(C[i, j + 1].l, C[i, j].r);
  end for;

  for j in 1:M loop
    connect(S.p, C[1, j].u);
    connect(C[N, j].d, G.p);
  end for;

  for i in 1:N-1, j in 1:M loop
    connect(C[i + 1, j].u, C[i, j].d);
  end for;

  for i in 1:N loop
    connect(C[i, 1].l, C[i, M].r);
  end for;
```

```
end Test;
```

Listado 6.5: Ejemplo 3: red de resistores y capacitores, 2 dimensiones

```
model Test
  Real N=1000;
  Real M=1000;
  Real S_n_eff[1, 1];
  flow Real S_n_fl[1, 1];
  Real S_p_eff[1, 1];
  flow Real S_p_fl[1, 1];
  Real G_n_eff[1, 1];
  flow Real G_n_fl[1, 1];
  Real G_p_eff[1, 1];
  flow Real G_p_fl[1, 1];
  Real C_u_eff[N, M];
  flow Real C_u_fl[N, M];
  Real C_d_eff[N, M];
  flow Real C_d_fl[N, M];
  Real C_l_eff[N, M];
  flow Real C_l_fl[N, M];
  Real C_r_eff[N, M];
  flow Real C_r_fl[N, M];
equation
  for i in 1:1:1,j in 1:1:1 loop
    G_p_eff = S_n_eff;
  end for;
  for i in 1000:1:1000,j in 1:1:1000 loop
    C_d_eff[i,j] = S_n_eff;
  end for;
  for i in 1:1:1,j in 1:1:1 loop
    S_n_fl+sum(C_d_fl[1*i+999, 1:1:1000])+G_p_fl = 0;
  end for;
  for i in 1:1:1000,j in 1:1:999 loop
    C_r_eff[i,j] = C_l_eff[1*i+0,1*j+1];
  end for;
  for i in 1:1:1000,j in 2:1:1000 loop
    C_r_fl[1*i+0,1*j+-1]+C_l_fl[1*i+0,1*j+0] = 0;
  end for;
  for i in 1:1:1,j in 1:1:1000 loop
    C_u_eff[i,j] = S_p_eff;
  end for;
  for i in 1:1:1,j in 1:1:1 loop
    sum(C_u_fl[1*i+0, 1:1:1000])+S_p_fl = 0;
  end for;
  for i in 1:1:999,j in 1:1:1000 loop
    C_d_eff[i,j] = C_u_eff[1*i+1,1*j+0];
  end for;
  for i in 2:1:1000,j in 1:1:1000 loop
    C_d_fl[1*i+-1,1*j+0]+C_u_fl[1*i+0,1*j+0] = 0;
  end for;
  for i in 1:1:1000,j in 1000:1:1000 loop
    C_r_eff[i,j] = C_l_eff[1*i+0,1*j+-999];
  end for;
  for i in 1:1:1000,j in 1:1:1 loop
    C_r_fl[1*i+0,1*j+999]+C_l_fl[1*i+0,1*j+0] = 0;
  end for;
```

```
54 end Test;
```

Listado 6.6: Ejemplo 3, aplanado

	N, M = 1000	N, M = 10000	N, M = 100000	N, M = 1000000
ModelicaCC	0.04883539	0.0505217	0.0469597	0.0506757

Tabla 6.3: *Resultados, instancia 3*

En ninguno de los casos OpenModelica completó la tarea en el tiempo establecido, por lo que no se presentan los resultados aquí.

Por otro lado, se pudo observar que el proceso ocupó buena parte de la memoria disponible, cosa que no sucedió en ModelicaCC.

Capítulo 7

Conclusiones

Concluyendo, se ha presentado la teoría de Grafos Basados en Conjuntos, junto con un algoritmo de componentes conexas para este tipo de estructuras. Dicho conocimiento se aplicó en la etapa de aplanado de un modelo Modelica. Esto permite trabajar a lo largo de toda la etapa de manera intensiva con arreglos de conectores, si el modelo presenta estructuras repetitivas.

Gracias a esto, ciertos modelos que anteriormente debían ser descartados en las primeras etapas de compilación, por el costo computacional que conllevan, ahora podrán ser aplanados. Además, las etapas posteriores recibirán código compacto, que será mucho más fácil de tratar, y terminar de compilar.

Si bien las restricciones impuestas a los modelos que pueden ser aplanados con el algoritmo propuesto pueden parecer restrictivas, este enfoque permite trabajar con modelos que de otro modo no podrían ser tratados por las herramientas existentes.

Finalmente, se demostró la gran diferencia de utilizar estas herramientas en contraposición con las que usan grafos tradicionales, con algunos ejemplos prácticos, donde se hace clara la ventaja.

7.1. Trabajo futuro

- Formalización de los resultados obtenidos: muchas de las operaciones se han definido de manera intuitiva, pero no se ha provisto de una demostración matemática de que los resultados sean correctos.
- Generalización de operaciones: la representación que se usó para la implementación de GBCs es la que introduce algunas restricciones en los modelos. Además, la operación de reducción de mapas seccionalmente lineales agrega algunas más. Si se logra generalizar esta operación, podrían tratarse aún más modelos.
- Aplicación a etapas posteriores: el objetivo de este trabajo era comenzar desde las etapas más tempranas de compilación para evitar expandir los arreglos de variables, y trabajar de manera intensiva a lo largo de todo el proceso. La etapa de aplanado es un excelente punto de partida, ya que es una etapa relativamente sencilla, y este trabajo es útil para introducirse a los GBCs. Lo que se ha desarrollado será útil para los nuevos algoritmos (todas las operaciones de intervalos, multi-intervalos, etc.).

Bibliografía

[1] Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. Towards a high-performance modelica compiler. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, number 157. Linköping University Electronic Press, 2019.

[2] Johan Åkesson, Magnus Gäfvert, and Hubertus Tummescheit. Jmodelica—an open source platform for optimization of modelica models. In *6th Vienna International Conference on Mathematical Modelling*, 2009.

[3] Matthias Arzt, Volker Waurich, and Jörg Wensch. Towards utilizing repeating structures for constant time compilation of large modelica models. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 35–38, 2014.

[4] Modelica Association. *Modelica® - A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.4*. 2017.

[5] EC Federico Bergero, Mariano Botta, and Ernesto Kofman. Efficient compilation of large scale modelica models. In *11th International Modelica Conference*, 2015.

[6] Mariano Botta. *Aplanado eficiente de grandes modelos Modelica*. Tesina de Licenciatura en Cs. de la Computación. Universidad Nacional de Rosario, 2015.

[7] Willi Braun, Francesco Casella, Bernhard Bachmann, et al. Solving large-scale modelica models: new approaches and experimental results using openmodelica. In *12 International Modelica Conference*, pages 557–563. Linkoping University Electronic Press, 2017.

[8] Dag Brück, Hilding Elmqvist, Sven Erik Mattsson, and Hans Olsson. Dymola for multi-engineering modeling and simulation. In *Proceedings of Modelica 2002*, 2002.

[9] Francesco Casella. Simulation of large-scale models in modelica: State of the art and future perspectives. In *11th International Modelica Conference*, pages 459–468, 2015.

[10] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: a Cyber-Physical Approach"*. Wiley-IEEE Press, 2015.

[11] 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.

[12] Peter Fritzson, Peter Aronsson, Adrian Pop, Hakan Lundvall, Kaj Nystrom, Levon Saldamli, David Broman, and Anders Sandholm. Openmodelica-a free open-source environment for

system modeling, simulation, and teaching. In *2006 IEEE Conference on Computer Aided Control System Design*, pages 1588–1595. IEEE, 2006.

[13] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[14] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[15] Xiaolin Qin, Juan Tang, Yong Feng, Bernhard Bachmann, and Peter Fritzson. Efficient index reduction algorithm for large scale systems of differential algebraic equations. *Applied Mathematics and Computation*, 277:10–22, 2016.

[16] Joseph Schuchart, Volker Waurich, Martin Flehmig, Marcus Walther, Wolfgang E Nagel, and Ines Gubsch. Exploiting repeated structures and vectorization in modelica. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, number 118, pages 265–272. Linköping University Electronic Press, 2015.

[17] Gerald Schweiger, Henrik Nilsson, Josef Schoeggl, Wolfgang Birk, and Alfred Posch. Modeling and simulation of large-scale systems: A systematic comparison of modeling paradigms. *Applied Mathematics and Computation*, 365:124713, 2020.

[18] Kristian Stavåker. *Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures*. PhD thesis, Linköping University Electronic Press, 2015.

[19] Pablo Zimmermann. Resolución de lazos algebraicos en grandes modelos modelica. Tesina de Licenciatura en Cs. de la Computación. Universidad Nacional de Rosario, 2018.

[20] 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.

Anexos

Apéndice A

Generación de conjuntos conexos

Se presenta formalmente cómo es que se construyen los conjuntos conexos a partir de un modelo Modelica $\mathcal{M}$. El resultado deseado se obtiene llamando a GenerateSets($\mathcal{M}$):

Algoritmo 38 Inicialización de conjuntos conexos

1: **function** Initialize(M)
2: $\quad \mathrm{cc}_{flow} \leftarrow \emptyset$
3: $\quad \mathrm{cc}_{eff} \leftarrow \emptyset$
4: $\quad$ **for all** connect(x, y) $\in$ M **do**
5: $\quad\quad \mathrm{cc}^{xy}_{flow} \leftarrow \emptyset$, $\mathrm{cc}^{xy}_{eff} \leftarrow \emptyset$
6: $\quad\quad$ **for all** var $\mathrm{v}_x : \mathrm{v}_x \in$ x **do**
7: $\quad\quad\quad$ **if** is_{flow} v_x **then**
8: $\quad\quad\quad\quad \mathrm{cc}^{xy}_{flow} \leftarrow \mathrm{cc}^{xy}_{flow} \cup \{\mathrm{v}_x\}$
9: $\quad\quad\quad$ **else**
10: $\quad\quad\quad\quad \mathrm{cc}^{xy}_{eff} \leftarrow \mathrm{cc}^{xy}_{eff} \cup \{\mathrm{v}_x\}$
11: $\quad\quad\quad$ **end if**
12: $\quad\quad$ **end for**
13: $\quad\quad$ **for all** var $\mathrm{v}_y : \mathrm{v}_y \in$ y **do**
14: $\quad\quad\quad$ **if** is_{flow} v_y **then**
15: $\quad\quad\quad\quad \mathrm{cc}^{xy}_{flow} \leftarrow \mathrm{cc}^{xy}_{flow} \cup \{\mathrm{v}_y\}$
16: $\quad\quad\quad$ **else**
17: $\quad\quad\quad\quad \mathrm{cc}^{xy}_{eff} \leftarrow \mathrm{cc}^{xy}_{eff} \cup \{\mathrm{v}_y\}$
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end for**
20: $\quad\quad \mathrm{cc}_{flow} \leftarrow \mathrm{cc}_{flow} \cup \{\mathrm{cc}^{xy}_{flow}\}$
21: $\quad\quad \mathrm{cc}_{eff} \leftarrow \mathrm{cc}_{eff} \cup \{\mathrm{cc}^{xy}_{eff}\}$
22: $\quad$ **end for**
23: $\quad$ **return** (cc_{flow}, cc_{eff})
24: **end function**

INITIALIZE genera dos conjuntos conexos por cada connect: uno para variables de esfuerzo, y otro para variables de flujo. A cada uno de ellos se deben agregar las variables del tipo correspondiente que estén implicadas en el connect. Luego, solo resta combinar los conjuntos conexos obtenidos, ya que ellos pueden tener elementos compartidos, lo que es indicativo de una conexión transitiva.

Algoritmo 39 Generación de conjuntos conexos

```
1: function GenerateSets(M)
2:     (cc_flow, cc_eff) ← Initialize(M)
3:     res_flow ← cc_flow
4:     res_eff ← cc_eff
5:     oldres_flow ← res_flow
6:     oldres_eff ← res_eff
7:     repeat
8:         oldres_flow ← res_flow
9:         for all cc^i_flow ∈ oldres_flow do
10:            for all cc^j_flow ∈ oldres_flow \ {cc^i_flow} do
11:                if cc^i_flow ∩ cc^j_flow ≠ ∅ then
12:                    res_flow ← res_flow ∪ {(cc^i_flow ∪ cc^j_flow)} \ {cc^i_flow} \ {cc^j_flow}
13:                else
14:                    res_flow ← res_flow ∪ {cc^i_flow} ∪ {cc^j_flow}
15:                end if
16:            end for
17:        end for
18:    until oldres_flow = res_flow
19:    repeat
20:        oldres_eff ← res_eff
21:        for all cc^i_eff ∈ oldres_eff do
22:            for all cc^j_eff ∈ oldres_eff \ {cc^i_eff} do
23:                if cc^i_eff ∩ cc^j_eff ≠ ∅ then
24:                    res_eff ← res_eff ∪ {(cc^i_eff ∪ cc^j_eff)} \ {cc^i_eff} \ {cc^j_eff}
25:                else
26:                    res_eff ← res_eff ∪ {cc^i_eff} ∪ {cc^j_eff}
27:                end if
28:            end for
29:        end for
30:    until oldres_eff = res_eff
31:    return (res_flow, res_eff)
32: end function
```

A continuación se presentan algunos modelos para ilustrar el funcionamiento de GenerateSets:

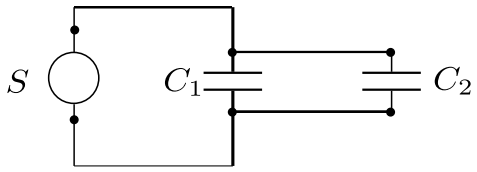


Imagen A.1: *Capacitores en paralelo*

```
connector Pin
  Real v;
  flow Real i;
end Pin;
```

Listado A.1: Conector de componentes eléctricos

```
model Capacitor
  parameter Real C;
  Pin p, n;
  Real u;
equation
  0 = p.i + n.i;
  u = p.v - n.v;
  C * der(u) = p.i;
end Capacitor;
```

Listado A.2: Modelo de un capacitor ideal

```
model SignalVoltage
  Pin p, n;
  parameter Real V;
  Real i;
equation
  V = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end SignalVoltage
```

Listado A.3: Modelo de fuente de tensión

```
model Circuit
  Capacitor a, b;
  SignalVoltage sv;
equation
  connect(sv.n, a.n);
  connect(sv.p, a.p);
  connect(a.n, b.n);
  connect(a.p, b.p);
end Circuit;
```

Listado A.4: Modelo del circuito

Los resultados se muestran a continuación:

1: INITIALIZE(Circuit) = ({{sv.n.i, a.n.i}, {sv.p.i, a.p.i}, {a.n.i, b.n.i}, {a.p.i, b.p.i}}, {{sv.n.v, a.n.v}, {sv.p.v, a.p.v}, {a.n.v, b.n.v}, {a.p.v, b.p.v}})
2: GENERATESETS(Circuit) = ({{sv.n.i, a.n.i, b.n.i}, {sv.p.i, a.p.i, b.p.i}}, {{sv.n.v, a.n.v, b.n.v}, {sv.p.v, a.p.v, b.p.v}})

Puede observarse que el algoritmo detecta las conexiones transitivas, y forma los conjuntos conexos de manera correcta.

Apéndice B

Implementación de intervalos

En este apéndice se presenta la implementación que se realizó en C++ de intervalos. Debe considerarse que se han definido algunas operaciones extra ya que el lenguaje así lo requiere (por ejemplo, la función de hasheo para poder definir colecciones no ordenadas de intervalos).

```
1  template<template<typename Value, typename Hash = boost::hash<Value>,
2                    typename Pred = std::equal_to<Value>,
3                    typename Alloc = std::allocator<Value>> class CT>
4  struct IntervalImp1{
5    int lo;
6    int step;
7    int hi;
8    bool empty;
9
10   int gcd(int a, int b){
11     int c;
12
13     do{
14       c = a % b;
15       if(c > 0){
16         a = b;
17         b = c;
18       }
19     } while (c != 0);
20
21     return b;
22   }
23
24   int lcm(int a, int b){
25     if(a < 0 || b < 0)
26       return -1;
27
28     return (a * b) / gcd(a, b);
29   }
30
31   // Creacion de intervalos
32   IntervalImp1(){};
33   IntervalImp1(bool isEmpty){
34     lo = -1;
35     step = -1;
36     hi = -1;
37     empty = isEmpty;
```

```
38  };
39  IntervalImp1(int vlo, int vstep, int vhi){
40    if(vlo >= 0 && vstep > 0 && vhi >= 0){
41      empty = false;
42      lo = vlo;
43      step = vstep;
44
45      if(vlo <= vhi && vhi < Inf){
46        int rem = std::fmod(vhi - vlo, vstep);
47        hi = vhi - rem;
48      }
49
50      else if(vlo <= vhi && vhi == Inf){
51        hi = Inf;
52      }
53
54      else{
55        //WARNING("Wrong values for subscript (check low <= hi)");
56        empty = true;
57      }
58    }
59
60    else if(vlo >= 0 && vstep == 0 && vhi == vlo){
61      empty = false;
62      lo = vlo;
63      hi = vhi;
64      step = 1;
65    }
66
67    else{
68      //WARNING("Subscripts should be positive");
69      lo = -1;
70      step = -1;
71      hi = -1;
72      empty = true;
73    }
74  }
75
76  int lo_(){
77    return lo;
78  }
79
80  int step_(){
81    return step;
82  }
83
84  int hi_(){
85    return hi;
86  }
87
88  bool empty_() const{
89    return empty;
90  }
91
92  // Pertenencia a intervalos
93  bool isIn(int x){
94    if(x < lo || x > hi || empty)
95      return false;
96
```

```
97      float aux = fmod(x - lo, step);
98      if(aux == 0)
99        return true;
100
101     return false;
102   }
103
104   // Interseccion de intervalos
105   IntervalImp1 cap(IntervalImp1 &inter2){
106     int maxLo = max(lo, inter2.lo), newLo = -1;
107     int newStep = lcm(step, inter2.step);
108     int newEnd = min(hi, inter2.hi);
109
110     if(!empty && !inter2.empty)
111       for(int i = 0; i < newStep; i++){
112         int res1 = maxLo + i;
113
114         if(isIn(res1) && inter2.isIn(res1)){
115           newLo = res1;
116           break;
117         }
118       }
119
120     else
121       return IntervalImp1(true);
122
123
124     if(newLo < 0)
125       return IntervalImp1(true);
126
127     return IntervalImp1(newLo, newStep, newEnd);
128   }
129
130   // Diferencia de intervalos
131   CT<IntervalImp1> diff(IntervalImp1 &i2){
132     CT<IntervalImp1> res;
133     IntervalImp1 capres = cap(i2);
134
135     if(capres.empty){
136       res.insert(*this);
137       return res;
138     }
139
140     if(capres == *this)
141       return res;
142
143     // "Before" intersection
144     if(lo < capres.lo){
145       IntervalImp1 aux = IntervalImp1(lo, 1, capres.lo - 1);
146       IntervalImp1 left = cap(aux);
147       res.insert(left);
148     }
149
150     // "During" intersection
151     if(capres.step <= (capres.hi - capres.lo)){
152       int nInters = capres.step / step;
153       for(int i = 1; i < nInters; i++){
154         IntervalImp1 aux = IntervalImp1(capres.lo + i * step, capres.step, capres.hi);
155         res.insert(aux);
```

```
156       }
157     }
158
159     // "After" intersection
160     if(hi > capres.hi){
161       IntervalImp1 aux = IntervalImp1(capres.hi + step, 1, hi);
162       IntervalImp1 right = cap(aux);
163       res.insert(right);
164     }
165
166     return res;
167   }
168
169   int minElem(){
170     return lo;
171   }
172
173   // Cardinalidad del intervalo
174   int size(){
175     int res = (hi - lo) / step + 1;
176     return res;
177   }
178
179   bool operator==(const IntervalImp1 &other) const{
180     return (lo == other.lo) && (step == other.step) && (hi == other.hi) &&
181            (empty == other.empty);
182   }
183
184   bool operator!=(const IntervalImp1 &other) const{
185     return (lo != other.lo) || (step != other.step) || (hi != other.hi) ||
186            (empty != other.empty);
187   }
188
189   size_t hash(){
190     return lo;
191   }
192 };
193
194 template<template<typename Value, typename Hash = boost::hash<Value>,
195                   typename Pred = std::equal_to<Value>,
196                   typename Alloc = std::allocator<Value>> class CT>
197 size_t hash_value(IntervalImp1<CT> inter){
198   return inter.hash();
199 }
```

Apéndice C

Resultados complementarios

Aquí se presentan las demostraciones de los teoremas presentados en el capítulo 4, y también se adicionan algunos teoremas auxiliares.

Lema 1 .
Dado el intervalo [lo:st:hi], y h $\in \mathbb{Z}$, h $\neq 0$, si st divide a h (h | st), entonces [lo:st:hi] = [lo:|h|:hi] $\cup$ [lo + st:|h|:hi] $\cup$ [lo + 2 * st:|h|:hi] $\cup$ ... $\cup$ [lo + ($\frac{|h|}{st}$ - 1) * st:|h|:hi]

Demostración. Se demuestra la igualdad:
$\subseteq$) Sea x $\in$ [lo:st:hi] $\Rightarrow \exists$ k $\in \mathbb{N}_0$ / x = lo + k * st
k $\in \mathbb{N}_0 \wedge$ st $\in \mathbb{N} \Rightarrow$ k * st $\in \mathbb{N}_0$.
Por otro lado, h $\in \mathbb{Z} \wedge$ h$\neq$ 0 $\Rightarrow$ |h| $\in \mathbb{N}$
Por el algoritmo de la división, $\exists!$(q, r) $\in \mathbb{N}_0^2$ / k * st = |h| * q + r con r < |h| $\Rightarrow$
x = lo + |h| * q + r
Llamando h$' = \frac{|h|}{st}$, se tiene que h$'$ $\in \mathbb{N}$ por hipótesis; luego:
x = lo + (k * st - h$'$ * st * q) + |h| * q = lo + st * (k - h$'$ * q) + |h| * q
En particular,
r = st * (k - h$'$ * q) < |h| $\Rightarrow$ k - h$'$ * q < $\frac{|h|}{st}$ $\Rightarrow$ k - h$'$ * q $\leq \frac{|h|}{st}$ - 1 (recordar que todos estos números pertenecen a $\mathbb{N}$ o $\mathbb{N}_0$)
Llamando r' = k - h$'$ * q,
r = st * r' $\geq 0 \wedge$ st $> 0 \Rightarrow$ r' ≥ 0
k $\in \mathbb{N}_0$, h$'$ $\in \mathbb{N}$, q $\in \mathbb{N}_0 \Rightarrow$ r' $\in \mathbb{Z}$
Por las dos implicancias anteriores, r' $\in \mathbb{N}$. Luego:
x = lo + r' * st + |h| * q, donde q $\in \mathbb{N}_0 \wedge$ r $\leq \frac{|h|}{st}$ - 1
Además, x $\leq$ hi pues x $\in$ [lo:st:hi]
Por ende, x $\in$ [lo + r' * st:|h|:hi] con r' $\in \mathbb{N}_0 \wedge$ r' $\leq \frac{|h|}{st}$ - 1
$\supseteq$) Sea x [lo + r' * st:|h|:hi], r' $\in \mathbb{N}_0 \wedge$ r' $\leq \frac{|h|}{st}$ - 1 $\Rightarrow$
$\exists$ k $\in \mathbb{N}_0$ / x = lo + r' * st + |h| * k
Por hipótesis, h | st $\Rightarrow$ |h| | st $\Rightarrow \exists$ k' $\in \mathbb{N}$ / |h| = k' * st $\Rightarrow$
x = lo + r' * st + k' * st * k = lo + st * (r' + k' * k)
r' $\in \mathbb{N}_0 \wedge$ k' $\in \mathbb{N} \wedge$ k $\in \mathbb{N}_0 \Rightarrow$ r' + k' * k $\in \mathbb{N}_0$
Además, x $\leq$ hi pues x $\in$ [lo + r' * st:|h|:hi]
Por ende, x $\in$ [lo:st:hi] □

Lema 2 .
Dada la expresión f(x) = m * x + h donde m, h $\in \mathbb{N}$, y su respectivo dominio [lo:st:hi], entonces im(f) = [m * lo + h:m * st:m * hi + h]

Demostración. im(f) = {y / ∃ x ∈ dom(f) / f(x) = y}
⊆) Sea y ∈ im(f) ⇒
∃ x ∈ [lo:st:hi] / m * x + h = y ⇒
∃ k ∈ $\mathbb{N}_0$ / x = lo + k * st ∧ x ≤ hi ∧ m * x + h = y ⇒
∃ k ∈ $\mathbb{N}_0$ / y = m * x + h = m * (lo + k * st) + h = m * lo + h + k * st ∧ y = m * x + h ≤ m * hi + h ⇒
y ∈ [m * lo + h:m * st:m * hi + h]
⊇) Sea y ∈ [m * lo + h:m * st:m * hi + h] ⇒
∃ k ∈ $\mathbb{N}_0$ / y = m * lo + h + k * m * st ∧ y < m * hi + h ⇒
∃ k ∈ $\mathbb{N}_0$ / y = m * (lo + k * st) + h ∧ y < m * hi + h
Veamos que x = lo + k * st ∈ dom(f) (solo queda demostrar que x ≤ hi):
y = m * (lo + k * st) + h ∧ y < m * hi + h ⇒
$\frac{y-h}{m}$ = x ∧ y < m * hi + h ⇒
x = $\frac{y-h}{m}$ < hi
Luego, x ∈ dom(f). Resta ver si f(x) = y:
f(x) = f(lo + k * st) = m * (lo + k * st) + h = y □

Lema 3 .
Dada la expresión f(x) = x + h, h ≠ 0, se tiene que f^i(x) = x + i * h

Demostración. Por inducción sobre i:

- Caso base:

  f(x) = x + 1 * h = x + h

  Se cumple el teorema para el caso base.

- Caso inductivo:

  Se supone que f^i(x) = x + i * h

  f^{i+1}(x) = f(f^{i+1}(x)) = f(x + i * h) = x + i * h + h = x + (i + 1) * h

  Se cumple el teorema para el caso inductivo.

□

Lema 4 .
Dada la expresión f(x) = x + h, h > 0, y el intervalo [lo:h:hi] = dom(f), se tiene que dom(f^i) = [lo:h:hi - h * (i - 1)] ∧ im(f^i) = [lo + h * i:h:hi + h]

Demostración. Por inducción sobre i:

- Caso base:

  dom(f) = [lo:h:hi - h * (1 - 1)] = [lo:h:hi]

  im(f) = [lo + h * 1:h:hi + h] = [lo + h:h:hi + h]

  Se cumple el teorema para el caso base.

- Caso inductivo:

  Se supone que dom(f^i) = [lo:h:hi - h * (i - 1)] ∧ im(f^i) = [lo + h * i:h:hi + h]

  dom(f^{i+1}) = dom(f^i ∘ f) = {x ∈ dom(f) / f(x) ∈ dom(f^i)}

$dom(f^{i+1}) = dom(f \circ f^i) = \{x \in dom(f^i) \ / \ f^i(x) \in dom(f)\}$

¿Se cumplirá que $dom(f^{i+1}) = [lo:h:hi - h * i]$?

$\subseteq$) Sea $x \in dom(f^{i+1})$. Luego, $x \in dom(f^i) \wedge f^i(x) \in dom(f) \Rightarrow$

$\exists\, k \in \mathbb{N}_0 \ / \ x = lo + k * h \wedge f^i(x) \leq hi \Rightarrow$

$\exists\, k \in \mathbb{N}_0 \ / \ x = lo + k * h \wedge x < hi - h * i \Rightarrow$

$x \in [lo:h:hi - h * i]$

$\supseteq$) Sea $x \in [lo:h:hi - h * i] \Rightarrow$

$\exists\, k \in \mathbb{N}_0 \ / \ x = lo + k * h \wedge x \leq hi - h * i \Rightarrow$

$\exists\, k \in \mathbb{N}_0 \ / \ x = lo + k * h \wedge x \leq hi \Rightarrow$

$x \in dom(f)$

Además,

$f(x) = lo + k * h + h = lo + h\ (k + 1) \Rightarrow$

$\exists\, k' \in \mathbb{N}_0 \ / \ f(x) = lo + k' * h$

Por otro lado

$x \leq hi - h * i \Rightarrow$

$f(x) = x + h \leq hi - h * i + h = hi - h * (i - 1)$

Luego, $f(x) \in dom(f^i)$

Así, se concluye que $dom(f^{i+1}) = [lo:h:hi - h * i]$

De este modo, se tiene que $im(f^{i+1}) = [lo + h * i:h * 1:hi - h * i + h * i] = [lo + h * i:h:hi]$

□

Lema 5 .
Dada la expresión $f(x) = x + h$, $h < 0$, y el intervalo $[lo:|h|:hi] = dom(f)$, se tiene que $dom(f^i) = [lo + |h| * (i - 1):|h|:hi] \wedge im(f^i) = [lo + h:|h|:hi + h * i]$

Demostración. Por inducción sobre i:

- Caso base:

  $dom(f) = [lo + |h| * (1 - 1):|h|:hi] = [lo:|h|:hi]$

  $im(f) = [lo + h:|h|:hi + h * 1] = [lo + h:|h|:hi + h]$

  Se cumple el teorema para el caso base.

- Caso inductivo:

  Se supone que $dom(f^i) = [lo + |h| * (i - 1):|h|:hi] \wedge im(f^i) = [lo + h:|h|:hi + h * i]$

  $dom(f^{i+1}) = dom(f^i \circ f) = \{x \in dom(f) \ / \ f(x) \in dom(f^i)\}$

  $dom(f^{i+1}) = dom(f \circ f^i) = \{x \in dom(f^i) \ / \ f^i(x) \in dom(f)\}$

  ¿Se cumplirá que $dom(f^{i+1}) = [lo + |h| * i:|h|:hi]$?

  $\subseteq$) Sea $x \in dom(f^{i+1})$. Luego, $x \in dom(f^i) \wedge f^i(x) \in dom(f) \Rightarrow$

  $\exists\, k \in \mathbb{N}_0 \ / \ f^i(x) = x + h * i = lo + k * |h| \wedge h < 0 \Rightarrow$

$\exists$ k $\in \mathbb{N}_0$ / x = lo - h * i + k * |h| $\wedge$ h $<$ 0 $\Rightarrow$

$\exists$ k $\in \mathbb{N}_0$ / x = lo + |h| * i + k * |h| $\Rightarrow$

Por otro lado,

h $<$ 0 $\wedge$ i $>$ 0 $\Rightarrow$ h * i $<$ 0 $\Rightarrow$ f^i(x) = x + h * i $<$ x

x $\in$ dom(f^i) $\Rightarrow$ x $\leq$ hi $\Rightarrow$ f^i(x) $\leq$ hi

Por ende, x $\in$ [lo + |h| * i:|h|:hi]

$\supseteq$) Sea x $\in$ [lo + |h| * i:h:hi] $\Rightarrow$

$\exists$ k $\in \mathbb{N}_0$ / x = lo + |h| * i + k * |h| $\Rightarrow$

$\exists$ k $\in \mathbb{N}_0$ / x = lo + |h| * i - |h| + k * |h| + |h|$\Rightarrow$

$\exists$ k' = k + 1 $\in \mathbb{N}_0$ / x = lo + |h| * (i - 1) + k' * |h|

Por otro lado, x $\leq$ hi.

Por ende, x $\in$ dom(f^{i+1})

Además,

f^i(x) = lo + |h| * i + k * |h| + h * i $\wedge$ h $<$ 0 $\Rightarrow$

f^i(x) = lo - h * i + k * |h| + h * i = lo + k * |h| $\Rightarrow$

$\exists$ k $\in \mathbb{N}_0$ / f^i(x) = lo + k * |h|

Por otro lado, x$\leq$ hi

Por ende, f^{i+1}(x) $\in$ dom(f)

Así, se concluye que dom(f^{i+1}) = [lo + |h| * i:|h|:hi]

De este modo, se tiene que im(f^{i+1}) = [lo + |h| * i + h * (i + 1):|h| * 1:hi + h * i] = [lo - h *i + h * (i + 1):|h|:hi + h * i] = [lo + h:|h|:hi + h * i]

□