



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Algoritmos incrementales de actualización para aproximaciones de bisimulación en XPath con datos

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Alejandro Grinberg

Director: Sergio Abriola

Codirector: Santiago Figueira

Buenos Aires, 2016

ALGORITMOS INCREMENTALES DE ACTUALIZACIÓN PARA APROXIMACIONES DE BISIMULACIÓN EN XPATH CON DATOS

XML (eXtensible Markup Language) es un lenguaje de marcas utilizado para almacenar datos en forma legible. XPath es un lenguaje que permite realizar consultas sobre documentos XML. En este trabajo nos enfocamos en XPath₌(↓), un fragmento de XPath que contiene el comportamiento de navegación y tests de (des)igualdad entre atributos de elementos. Un documento XML se puede modelar mediante un árbol de datos (*data-tree*).

La bisimulación es una relación entre dos *data-trees*. La importancia de esta noción es su correspondencia con la equivalencia lógica, lo que significa que todas las fórmulas del lenguaje que son válidas en un árbol y un nodo, son válidas también en el nodo correspondiente del otro árbol. Calcular la bisimulación puede ser muy costoso, por lo que se introducen las aproximaciones de bisimulación. Una instancia de una aproximación define un fragmento del lenguaje XPath₌(↓) en el cuál vale la correspondencia mencionada. Trabajamos con la ℓ -bisimulación y la *comp*-bisimulación, un nuevo tipo de aproximación que introducimos.

Planteamos un enfoque en el cuál comenzamos con una instancia de aproximación básica que vamos mejorando a través de actualizaciones incrementales. Estas actualizaciones se producen cuando un proceso de *mining* lo determina y se nutren de los aprendizajes realizados durante la evaluación de las *queries*. Desarrollamos e implementamos algoritmos para estos procesos para los dos tipos de aproximación mencionados, y definimos cómo debería ser una aproximación genérica. Creamos una herramienta *web* que pone en práctica los conceptos estudiados a lo largo de este trabajo.

Palabras claves: XML, XPath, XPath₌(↓), bisimulación, aproximación, ℓ -bisimulación, *comp*-bisimulación, actualización incremental, aprendizaje.

Índice general

1..	Introducción	1
1.1.	Antecedentes	1
1.2.	Objetivos	1
1.3.	Trabajo previo	2
1.4.	Contribuciones	7
1.5.	Preliminares	8
2..	Aproximaciones de bisimulación	10
2.1.	ℓ -Bisimulación	10
2.1.1.	Definición del fragmento	10
2.1.2.	Equivalencia lógica	10
2.1.3.	Bisimulación	10
2.1.4.	Relación entre equivalencia y bisimulación	11
2.2.	<i>comp</i> -Bisimulación	11
2.2.1.	Definición del fragmento	11
2.2.2.	Equivalencia lógica	11
2.2.3.	Bisimulación	11
2.2.4.	Relación entre equivalencia y bisimulación	12
2.2.5.	Consideraciones prácticas	15
2.2.6.	Ejemplos	16
3..	Actualización de aproximaciones	18
3.1.	Motivación	18
3.2.	Proceso de actualización	19
3.2.1.	ℓ -Bisimulación	19
3.2.2.	<i>comp</i> -Bisimulación	23
3.2.3.	Aproximación genérica de bisimulación	24
3.3.	Proceso de mining	26
3.3.1.	ℓ -Bisimulación	26
3.3.2.	<i>comp</i> -Bisimulación	32
4..	Implementación	34
4.1.	Aplicación Web	34
4.1.1.	Graficador árbol XML	35
4.1.2.	Generador árbol XML	36
4.1.3.	Test documento XML grande	37
4.1.4.	Generador aleatorio	38
4.2.	Mejoras de performance	39
4.2.1.	Reducción de tiempos	39
4.2.2.	Reducción del uso de memoria	41
4.3.	Consideraciones varias	42
4.4.	<i>comp</i> -Bisimulación	43
4.5.	Algoritmos	44

4.5.1.	Aproximaciones de bisimulación	44
4.5.2.	Cálculo inicial	46
4.5.3.	Evaluación de una expresión en un nodo	47
4.5.4.	Proceso de mining	48
4.5.5.	Proceso de actualización de la aproximación	50
5..	Experimentación	51
5.1.	Aprendizajes en ℓ -bisimulación	51
5.1.1.	Primeras pruebas	51
5.1.2.	Actualización según cantidad de aprendizajes	55
5.1.3.	Actualización según calidad de aprendizajes	56
5.2.	Actualización inmediata vs incremental en ℓ -bisimulación	59
5.2.1.	Resultados favorables para la actualización incremental	59
5.2.2.	Resultados favorables para la actualización inmediata	61
5.3.	<i>comp</i> -bisimulación	63
5.3.1.	Cálculo inicial y evaluación en Z y Z_\emptyset	64
5.3.2.	Actualización inmediata vs incremental	66
5.4.	Uso de índices	68
6..	Conclusiones	70
6.1.	Trabajo futuro	71

1. INTRODUCCIÓN

Con el creciente uso de XML en la Web, hay mucho interés en el procesamiento de queries sobre dichos documentos. Entre los lenguajes que permiten esto, se destaca XPath [1]. Core-XPath [2] es el fragmento de XPath 1.0 que contiene el comportamiento de navegación de XPath. La extensión de Core-XPath con tests de (des)igualdad entre atributos de elementos en un documento XML se llama Core-Data-XPath [3] o XPath₌.

Un concepto fundamental para XPath₌, desarrollado en [4], es el de *bisimulación* entre dos data trees, el cual implica que todas las formulas del lenguaje que son válidas en un árbol y un nodo son válidas en el nodo correspondiente del otro árbol (bajo ciertas condiciones). La autobisimulación es una relación de equivalencia que induce una partición entre los nodos de un árbol.

1.1. Antecedentes

La noción de bisimulación fue descubierta de manera independiente y relativamente simultánea por van Benthem, en el contexto de teoría de correspondencia modal; Milner y Park, en teoría de la concurrencia; y Forti y Honsell en teoría de conjuntos sin axioma de buena fundación. Estos últimos utilizan bisimulaciones para mostrar la equivalencia de objetos con estructura infinita no-inductiva y garantizar así extensionalidad de los modelos de su teoría [5]. Van Benthem obtiene la idea de bisimulación como una generalización del concepto de *p*-morfismo entre modelos; con ella caracteriza a la lógica modal básica como el fragmento de primer orden invariante bajo bisimulaciones (lo que se conoce como *Teorema de Caracterización de van Benthem*). Milner y Park fueron los que acuñaron el término *bisimulación*, técnica que utilizaron como herramienta para probar la equivalencia de procesos concurrentes [6, 7].

La idea de las bisimulaciones surge, entonces, como una herramienta teórica, principalmente para probar equivalencias. El costado algorítmico aparece en su aplicación a autómatas finitos: los algoritmos de minimización (algunos conocidos ya en la década de 1950) están simplemente calculando el cociente del autómata por su autobisimulación máxima. Estos algoritmos son muy importantes, por ejemplo, en el área de verificación: verificar es polinomial en el tamaño de un autómata que es exponencial en el tamaño de la fórmula; un algoritmo polinomial que devuelva un autómata equivalente pero de menor tamaño puede ser crucial en este contexto. Los mejores algoritmos para calcular bisimulaciones y minimizaciones módulo autobisimilaridad son variaciones del algoritmo de Paige y Tarjan [8] para calcular la partición más gruesa (*coarsest partition*) de un grafo, que es lineal en el tamaño del grafo.

1.2. Objetivos

Esta tesis tiene dos metas principales, centradas en la noción de bisimulación:

- Nuestro primer objetivo es el de analizar cómo podemos usar el conocimiento de queries realizadas en un árbol para pasar de una partición P_1 a otra partición P_2 : a partir de cierto mining sobre dicha información (revisando características como

profundidad, labels utilizados/comparados, etc.) y tomando en cuenta los tiempos involucrados (en responder una query, en actualizar la partición) decidimos como va a ser P_2 . Esta nueva partición se construye a través de un algoritmo de actualización incremental A , que toma como base la partición actual y el aprendizaje realizado. Analizamos cómo es A para el caso de ℓ -autobisimulaciones, un tipo de aproximación que consiste en una noción de bisimulación acotada en profundidad [4].

- En segundo lugar planteamos una nueva aproximación *comp*-bisimulación que restringe las formulas del tipo $\langle \alpha = \beta \rangle$ y $\langle \alpha \neq \beta \rangle$ para que sólo puedan comparar los datos de pares de nodos con ciertos labels. Experimentamos con esta aproximación para evaluar cómo repercute con respecto a ciertas métricas, como pueden ser memoria utilizada, tiempo de construcción, tiempo de respuesta, entre otras. Consideramos cómo es A para esta aproximación.

También estamos interesados en investigar si es posible utilizar alguno de los índices existentes (1-Index [9], F&B-Index [10, 11], A(k)-Index [12], APEX [13], D(k)-Index [14], M(k)-Index [15] entre otros) para responder las queries de XPath= (\downarrow) , prestando especial atención a las formulas del tipo $\langle \alpha = \beta \rangle$ y $\langle \alpha \neq \beta \rangle$.

1.3. Trabajo previo

En el marco de Core-XPath en grafos dirigidos y etiquetados (labels en los nodos) se puede y suele utilizar otro grafo de iguales características para resumir la estructura del primero. El objetivo de este segundo grafo es preservar todos los caminos del original, pero reduciendo la cantidad de nodos y aristas. Entre otras funciones, se puede utilizar como un índice para facilitar la evaluación de expresiones de camino, asociando cada nodo del índice con un conjunto de nodos del grafo original. Esta idea surge del mundo de las bases de datos relacionales, donde es usual y bien conocida la técnica de definir índices sobre las tablas de datos, para acelerar las queries SQL.

En general, cualquier partición de los nodos define un grafo índice, donde cada nodo del mismo se asocia a una clase de equivalencia. En particular, existen varias estructuras de índices que se basan en cierta variación de la noción de bisimulación.

El más simple y básico de dichos índices es el 1-Index [9], que se define directamente a partir de las particiones inducidas por la bisimulación (hacia atrás) modal. Esta partición se puede calcular en tiempo $O(m \log n)$ (donde n representa la cantidad de nodos y m la cantidad de aristas del grafo original) usando un algoritmo propuesto por Paige y Tarjan [8]. El 1-Index cubre queries de caminos entrantes (queries usando \uparrow).

El Forward and Backward-Index (al cual denominaremos F&B-Index), fue definido directamente sobre ideas propuestas en [10] y utiliza la noción de aristas inversas para capturar información tanto sobre los caminos entrantes como sobre los salientes. Podemos generar este índice agregando estas aristas inversas y luego calculando el 1-Index sobre el grafo modificado. Una propiedad interesante del F&B-Index es que es el índice más chico que cubre todas las expresiones de camino. La desventaja de esta estructura es que suele ser muy grande, lo que la hace poco performante.

El A(k)-Index [12] es el primero que utiliza el concepto de aproximación. La motivación de este trabajo es que no toda la estructura de un documento XML es interesante, y que caminos largos y complejos tienden a contribuir desproporcionadamente a la complejidad

del índice. Con el objetivo de reducir el tamaño del índice, y valiéndose de la suposición de que en la práctica son poco comunes las queries sobre dichos caminos, se explora el problema basándose en la bisimilitud local. Para esto utiliza la noción de k -bisimulación (una ℓ -bisimulación pero modal). Además de ser rápido para expresiones cortas, funciona bastante bien para expresiones de largo mayor a k , utilizando técnicas de validación.

Existen varios artículos [16, 17, 18, 19] que investigan y proponen algoritmos incrementales para mantener la bisimulación actualizada, pero sólo frente a cambios en el modelo de datos. Los cambios que observan son la inserción/borrado de aristas y la adición de nuevos subgrafos. En [16] se presentan algoritmos para ambos tipos de cambios para el 1-Index. Los mismos luego se extienden para el caso del F&B-Index y únicamente se discuten para el A(k)-Index. En [17] se mejoran los algoritmos de actualización sobre el 1-Index y se presentan nuevos para el A(k)-Index. Las publicaciones [18, 19] no trabajan directamente con índices, pero prestan especial atención a los casos en los que el grafo que representa al documento XML no tiene forma de árbol.

En [11] se trabaja con aproximaciones para generar índices F&B más pequeños, intentando revertir la desventaja comentada anteriormente. Para ello toman cuatro enfoques diferentes basándose en ciertas intuiciones que observan. Estos enfoques son:

- Existen labels dentro del modelo de datos que son de poco interés y/o no son utilizados en las queries, por lo que no es necesario indexarlos.
- Para adecuarse más a la estructura de ciertos documentos XMLs, se consideran las referencias entre entidades, que se representan en el grafo como aristas de tipo *idref*. Sin embargo, consideran que estas tienen menor prioridad frente a las aristas normales del grafo. Por lo tanto, permiten especificar el conjunto de aristas de tipo *idref* que se desean indexar.
- Como en el A(k)-Index, se supone que no toda la estructura es interesante y que las queries de caminos largos son poco frecuentes. Entonces se puede aprovechar el concepto de similitud local definiendo una longitud k_1 hacia adelante y otra longitud k_2 hacia atrás.
- Se define en el artículo la noción de profundidad en el árbol de un nodo en una query, que difiere de la noción de anidamiento. Una ventaja de esta nueva noción (frente a la otra), es que no depende de cómo se escriba sintácticamente la query. Los autores intuyen que es raro tener queries interesantes/significativas con un valor grande de profundidad en el árbol. Por lo tanto, definen una restricción dando un valor máximo.

Juntando todo estos enfoques, se enuncia que la definición de un índice consta de lo siguiente: un conjunto de labels a ser indexado, para cada una de las direcciones (hacia adelante y hacia atrás) un conjunto de aristas de tipo *idref* a ser indexado y el parámetro k indicando la longitud de la similitud local deseada, y el valor máximo de la profundidad en el árbol de un nodo en una query. Asignando los valores correspondientes, con esta definición se pueden generar el 1-Index, el F&B-Index, el A(k)-Index, entre otros. En general, una buena elección en la definición de un índice depende fuertemente tanto del modelo de datos como de las queries que se evalúan.

Con el mismo objetivo de generar índices pequeños, pero tomando en cuenta también las queries realizadas para soportar los caminos más frecuentes, se presenta APEX [13].

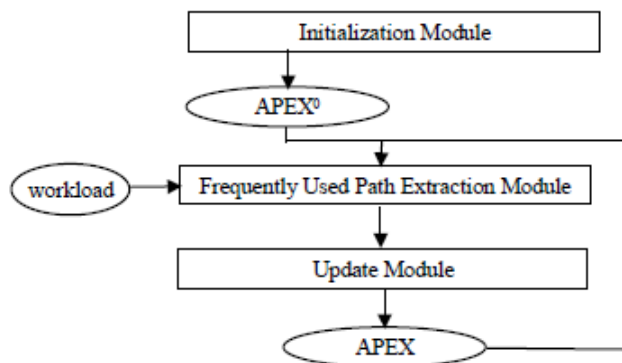


Fig. 1.1: Arquitectura de la herramienta de gestión de APEX (imagen tomada de [13])

APEX utiliza técnicas de data mining sobre el conjunto de queries ya ejecutadas para obtener patrones de secuencias que se repiten una cantidad de veces mayor o igual a un valor predeterminado por el usuario. Si se cumple dicha condición, entonces se considera al camino como uno frecuente. Se definen los caminos requeridos (es decir, los que el índice debe soportar con exactitud) como la unión entre los frecuentes con los de longitud uno (estos últimos equivalen a expresiones de nodo de tipo $\varphi = a$ con $a \in \mathbb{A}$). En la práctica, los autores mencionan que utilizan un algoritmo ingenuo que cuenta manualmente las subsecuencias que aparecen en las queries, ya que el tamaño del conjunto de queries a analizar no es lo suficientemente grande como para requerir algoritmos de data mining. Un módulo de inicialización genera primero $APEX^0$. Luego, a medida que se van ejecutando las queries sobre dicho índice, se van calculando los caminos más frecuentes. Estos se utilizan para actualizar incrementalmente el índice actual para llegar a otro más aproximado. Estos últimos pasos se repiten cuando se registran cambios en la carga de queries ejecutadas. La Figura 1.1 muestra una visión a grandes rasgos de lo descripto.

Un paso más allá presenta el $D(k)$ -Index [14] que cuenta con algoritmos incrementales frente a cambios en las queries realizadas (como APEX) y agrega otros para afrontar los cambios en el modelo de datos. Este índice también se basa en el concepto de similitud local, pero a diferencia del $A(k)$ -Index, permite especificar el valor de k para cada clase de equivalencia, dependiendo del análisis de las queries ya ejecutadas. Esta distinción permite mayor precisión y en consecuencia, una estructura más chica y performante. En los procesos de actualización del índice, también se puede aprovechar este concepto de similitud local para aislar las modificaciones a realizar.

En [15] no se presentan algoritmos de actualización frente a cambios en el modelo de datos, pero se mencionan cuatro limitaciones que tiene el $D(k)$ -Index y que los autores solucionan en sus índices $M(k)$ -Index y $M^*(k)$ -Index. Estas son:

- Sobre-refinamiento de nodos del índice irrelevantes: todos los nodos del índice con mismo label tienen el mismo valor k de similitud local, lo que es innecesario y restrictivo.

- Sobre-refinamiento para nodos de datos irrelevantes: al refinar un nodo del índice, se refinan todos los nodos de datos apuntados por aquel, aún los que no tienen que ver con el camino frecuente que se quiere tomar en cuenta.
- Sobre-refinamiento por padres sobre-calificados: para incrementar el valor de similitud local de un nodo del índice a k , el procedimiento de actualización del D(k)-Index usa la información acerca de los nodos padres en el índice. Si algún padre está sobre-calificado, es decir tiene un grado de similitud local mayor a $k - 1$, entonces el algoritmo va a sobre-refinar al nodo a un valor mayor a k .
- Resolución única por nodo: una vez que el D(k)-Index se refinó para soportar una expresión de camino larga, los índices de nodo correspondientes tendrán un valor de similitud local mayor. Este refinamiento puede potencialmente resultar en un particionamiento en varios nodos. Por lo tanto, una expresión de camino corta, que haga referencia a los mismos nodos de datos (entre otros), va a resultar más costosa de evaluar, ya que tiene que examinar potencialmente más nodos en el índice.

Para afrontar las dos primeras limitaciones, se propone el M(k)-Index, que ataca dichas restricciones permitiendo tener diferentes valores de similitud local k para diferentes nodos. Adicionalmente, y al igual que el D(k)-Index, se refina incrementalmente para soportar nuevas expresiones de camino frecuentes.

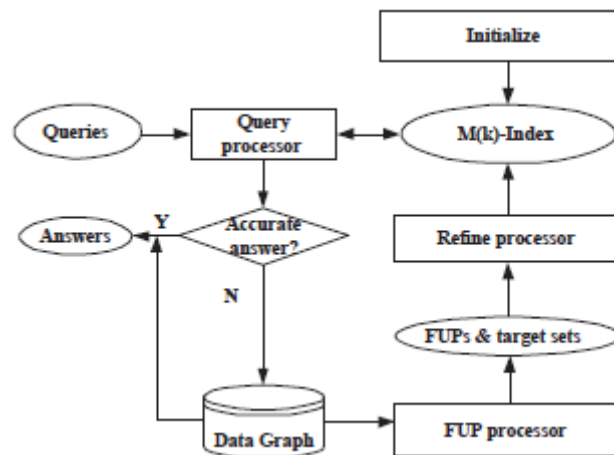


Fig. 1.2: Arquitectura de la herramienta de gestión del M(k)-Index (imagen tomada de [15])

De forma similar a APEX, se construye inicialmente una estructura con $k = 0$ (esencialmente un A(0)-Index). Luego, y en general, las queries se responden preferentemente utilizando el índice, pero si no hay garantías de precisión (pues la profundidad de la query es mayor al k actual) se validan estos resultados intermedios contra el grafo original. Con esta validación surge la respuesta a la consulta pero también información valiosa que se utilizará en el algoritmo de refinamiento si la query resulta ser una expresión de camino frecuente que se desea soportar. Cuando se obtiene una buena carga de información acerca de las queries ejecutadas, se pueden extraer de ellos las expresiones de camino más frecuentes y actualizar en consecuencia el índice para soportarlas. La Figura 1.2 muestra una visión a grandes rasgos de lo descrito.

Para afrontar todas las restricciones, se presenta el $M^*(k)$ -Index. Conceptualmente, esta estructura es una colección de índices I_0, I_1, \dots, I_K . Cada componente puede ser considerada como un $M(k)$ -Index que soporta los caminos frecuentes tanto como es posible, cumpliendo con la restricción de que el valor máximo de similitud local de I_i es i . Aunque aparenta necesitar mucho más espacio en memoria para representarse, su tamaño es similar al del $M(k)$ -Index. Esto se debe a que evita el sobre-refinamiento por padres sobrecalificados y a que aprovecha que muchos nodos y aristas se mantienen sin cambios entre un índice $M(i)$ y el subsiguiente $M(i+1)$. Esta estructura de colecciones permite mejorar la performance al responder expresiones de camino cortas, utilizando la componente indicada, pero también se puede aprovechar sus características para responder expresiones de camino largas. Esto se puede realizar evaluando la expresión en pasos: por ejemplo si se tiene la expresión $//l_1/l_2/l_3$, primero se puede evaluar l_1 en $M(0)$, de ahí saltar a los nodos correspondientes en $M(1)$ y evaluar l_2 , y así sucesivamente. La Figura 1.3 a continuación muestra un ejemplo de un $M^*(k)$ -Index. El grafo original se muestra en la parte (a). Adentro de cada nodo se muestra su label, y a la izquierda su Id. En la parte (b) se muestra el $M^*(2)$ -Index con sus 3 componentes I_0, I_1, I_2 . Adentro de cada nodo de cada componente se muestra su label, y a la izquierda el conjunto de nodos del grafo original que agrupan. Las flechas punteadas azules muestran la correspondencia entre un nodo en la componente I_i y sus nodos refinados en la componente I_{i+1} .

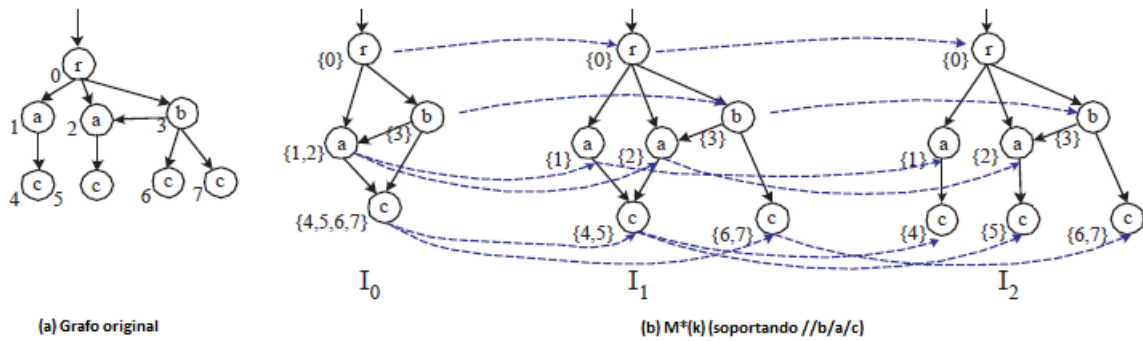


Fig. 1.3: Ejemplo de un $M^*(k)$ -Index (imagen tomada de [15])

1.4. Contribuciones

La principal diferencia entre el trabajo previo y el nuestro radica en el fragmento de XPath en el cual se trabaja. Mientras que la investigación citada se enmarca en CoreXPath, la nuestra lo hace en XPath₌. Nos proponemos analizar qué ideas y conceptos se pueden adaptar y/o utilizar para nuestro marco de trabajo.

Nuestra idea de algoritmo incremental se basa en los de APEX y M(k)-Index, pero con ciertas diferencias. En APEX, el índice a generar debe soportar ciertos caminos considerados como frecuentes. Para definir cuáles lo son, se necesita de un usuario externo que predetermine cierto valor. En nuestro algoritmo, por ejemplo de ℓ -bisimulación, la relación a generar debe soportar queries de cierta longitud. En nuestro caso, es el mismo sistema el que define dicha longitud, utilizando la información de las consultas ejecutadas. Otra diferencia a destacar es que en nuestro trabajo precisamos cuándo se realiza la actualización, es decir, qué condiciones se deben cumplir para realizar dicho proceso.

Definimos también un nuevo tipo de aproximación, basado en la comparación de datos de ciertos pares de labels, al cual denominamos *comp*-bisimulación. Trabajamos sobre esta aproximación tanto a nivel teórico como práctico. En este último aspecto, tanto para la *comp*-bisimulación como para la ℓ -bisimulación, implementamos un software que cuenta con todos los procesos abarcados (cálculo inicial, evaluación de expresiones, mining sobre la base de conocimientos, actualización de la aproximación). Esta herramienta es una aplicación web, que permite al usuario interactuar de diversas maneras. Las funcionalidades principales son la visualización y creación de arboles a partir de un documento XML, la evaluación de una o más queries, la generación de casos de prueba (XMLs y queries) aleatorios y la obtención de expresiones que distinguen y caracterizan clases de equivalencia.

Adicionalmente a las aproximaciones presentadas, caracterizamos una aproximación «genérica» y discutimos por cada proceso qué se puede generalizar y que no. Considerando estas distinciones, definimos en el código una clase (abstracta) *AproximacionGenerica* que cuenta con algunos métodos ya desarrollados y otros que son necesarios implementar.

Finalmente, realizamos varios experimentos sobre las decisiones tomadas, las implementaciones realizadas, los beneficios de las ideas presentadas en el trabajo, entre otras. Analizamos los resultados para sacar conclusiones y también como retroalimentación para reconsiderar algunas determinaciones tomadas.

1.5. Preliminares

Usamos el símbolo \mathbb{A} para denotar un alfabeto finito, y \mathbb{D} para denotar un dominio infinito (por ejemplo \mathbb{N}) de valores de datos. Sea $Trees(A)$ el conjunto de árboles ordenados y sin ranking sobre un conjunto arbitrario A . Decimos que \mathcal{T} es un árbol de datos/data tree si es un árbol de $Trees(\mathbb{A} \times \mathbb{D})$. La Figura 1.4 muestra un ejemplo de un data tree:

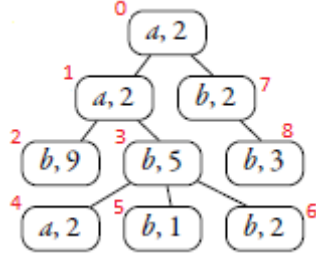


Fig. 1.4: Un data tree de $Trees(\mathbb{A} \times \mathbb{D})$ con $\mathbb{A} = \{a, b\}$ y $\mathbb{D} = \mathbb{N}$ (imagen tomada de [4])

Un data tree es de ramificación finita si cada nodo tiene una cantidad finita de hijos. Para cualquier data tree \mathcal{T} , denotamos por T a su conjunto de nodos. Usamos las letras x, y, z, v, w como variables para nodos. Dado un nodo $x \in T$ de \mathcal{T} , escribimos $label(x) \in \mathbb{A}$ para denotar la etiqueta del nodo, y $data(x) \in \mathbb{D}$ para denotar el valor del dato del nodo.

Dados dos nodos $x, y \in T$ escribimos $x \rightarrow y$ si y es un hijo de x , y $x \xrightarrow{n} y$ si y es un descendiente de x a distancia n . En particular, $\xrightarrow{1}$ es lo mismo que \rightarrow , y $\xrightarrow{0}$ es la relación de identidad. Escribimos $x \xrightarrow{*} y$ para denotar que (x, y) está en la clausura reflexiva transitiva de \rightarrow . $(x \xrightarrow{n})$ denota el conjunto de todos los descendientes de x a distancia n , y $(\xrightarrow{n} y)$ denota el único ancestro de y a distancia n (asumiendo que tiene uno).

Utilizamos el lenguaje de consultas XPath adaptado a data trees como abstracciones de documentos XML. Trabajamos con una simplificación de XPath, despojada de su azúcar sintáctico. Consideramos los fragmentos de XPath que corresponden al comportamiento de navegación de XPath 1.0 con (des)igualdad de datos. XPath₌ es un lenguaje que cuenta con expresiones de camino (que escribimos como α, β, γ) y expresiones de nodo (que escribimos como φ, ψ, η). El fragmento XPath₌(\downarrow) se define por recursión mutua como se muestra a continuación:

$$\begin{aligned} \alpha, \beta &::= \varepsilon \mid \downarrow \mid \alpha\beta \mid \alpha \cup \beta \mid [\varphi] \\ \varphi, \psi &::= a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle \alpha \rangle \mid \langle \alpha = \beta \rangle \mid \langle \alpha \neq \beta \rangle \quad a \in \mathbb{A} \end{aligned}$$

Definimos formalmente la semántica de XPath₌ como se muestra a continuación, siendo \mathcal{T} un data tree:

$$\begin{aligned}
\llbracket \varepsilon \rrbracket^{\mathcal{T}} &= \{(x, x) \mid x \in T\} \\
\llbracket \downarrow \rrbracket^{\mathcal{T}} &= \{(x, y) \mid x \rightarrow y\} \\
\llbracket \alpha\beta \rrbracket^{\mathcal{T}} &= \{(x, z) \mid (\exists y \in T) (x, y) \in \llbracket \alpha \rrbracket^{\mathcal{T}}, (y, z) \in \llbracket \beta \rrbracket^{\mathcal{T}}\} \\
\llbracket \alpha \cup \beta \rrbracket^{\mathcal{T}} &= \llbracket \alpha \rrbracket^{\mathcal{T}} \cup \llbracket \beta \rrbracket^{\mathcal{T}} \\
\llbracket [\varphi] \rrbracket^{\mathcal{T}} &= \{(x, x) \mid x \in \llbracket \varphi \rrbracket^{\mathcal{T}}\} \\
\llbracket [a] \rrbracket^{\mathcal{T}} &= \{x \in T \mid \text{label}(x) = a\} \\
\llbracket [\neg\varphi] \rrbracket^{\mathcal{T}} &= T \setminus \llbracket \varphi \rrbracket^{\mathcal{T}} \\
\llbracket [\varphi \wedge \psi] \rrbracket^{\mathcal{T}} &= \llbracket \varphi \rrbracket^{\mathcal{T}} \cap \llbracket \psi \rrbracket^{\mathcal{T}} \\
\llbracket [\varphi \vee \psi] \rrbracket^{\mathcal{T}} &= \llbracket \varphi \rrbracket^{\mathcal{T}} \cup \llbracket \psi \rrbracket^{\mathcal{T}} \\
\llbracket \langle \alpha \rangle \rrbracket^{\mathcal{T}} &= \{x \in T \mid (\exists y \in T) (x, y) \in \llbracket \alpha \rrbracket^{\mathcal{T}}\} \\
\llbracket \langle \alpha = \beta \rangle \rrbracket^{\mathcal{T}} &= \{x \in T \mid (\exists y, z \in T) (x, y) \in \llbracket \alpha \rrbracket^{\mathcal{T}}, (x, z) \in \llbracket \beta \rrbracket^{\mathcal{T}}, \text{data}(y) = \text{data}(z)\} \\
\llbracket \langle \alpha \neq \beta \rangle \rrbracket^{\mathcal{T}} &= \{x \in T \mid (\exists y, z \in T) (x, y) \in \llbracket \alpha \rrbracket^{\mathcal{T}}, (x, z) \in \llbracket \beta \rrbracket^{\mathcal{T}}, \text{data}(y) \neq \text{data}(z)\}
\end{aligned}$$

Para dar un ejemplo, tomemos como \mathcal{T} al árbol de la figura 1.4. Consideramos la fórmula $\varphi = \langle \downarrow[b \wedge \langle \downarrow[b] \neq \downarrow[b] \rangle] \rangle$. Esta expresión se satisface en los nodos que tienen un hijo con etiqueta b, que a su vez tiene dos hijos con etiqueta b con diferentes valores de datos. Por lo tanto, vale que $\llbracket \varphi \rrbracket^{\mathcal{T}} = \{1\}$.

Para un data tree \mathcal{T} y $u \in T$, escribimos $\mathcal{T}, u \models \varphi$ para denotar $u \in \llbracket \varphi \rrbracket^{\mathcal{T}}$, y decimos que \mathcal{T}, u satisface φ . Decimos que las expresiones de nodo φ, ψ de XPath₌ son equivalentes (notación: $\varphi \equiv \psi$) sii $\llbracket \varphi \rrbracket^{\mathcal{T}} = \llbracket \psi \rrbracket^{\mathcal{T}}$ para todos los data trees \mathcal{T} . De forma similar, las expresiones de camino α, β de XPath₌ son equivalentes (notación: $\alpha \equiv \beta$) sii $\llbracket \alpha \rrbracket^{\mathcal{T}} = \llbracket \beta \rrbracket^{\mathcal{T}}$ para todos los data trees \mathcal{T} .

En términos de poder expresivo, se puede ver que \cup es innecesario: cada expresión de nodo en XPath₌(\downarrow) tiene otra equivalente sin usar \cup en sus expresiones de camino [4]. Por lo tanto, de aquí en adelante vamos a asumir que las fórmulas no contienen unión de expresiones de camino.

Una fórmula (expresión de nodo) de XPath₌(\downarrow) está en forma normal si cualquier expresión de camino que contenga es ϵ o tiene la forma $[\psi_1]\downarrow[\psi_2] \dots \downarrow[\psi_n]$ con $n \geq 1$. Es fácil ver que cualquier expresión de nodo en XPath₌(\downarrow) tiene otra equivalente en forma normal. Por ejemplo, $\varphi = \langle \downarrow[\psi_1][\psi_2]\downarrow = \epsilon \rangle$ no está en forma normal, pero la expresión equivalente $\varphi' = \langle [\epsilon]\downarrow[\epsilon]\downarrow[\psi_1 \wedge \psi_2]\downarrow[\epsilon] = \epsilon \rangle$ sí lo está.

2. APROXIMACIONES DE BISIMULACIÓN

2.1. ℓ -Bisimulación

En esta sección resumimos las definiciones presentadas en [4] sobre esta aproximación.

2.1.1. Definición del fragmento

Escribimos $\text{dd}(\varphi)$ para denotar la **profundidad hacia abajo** de φ definida como sigue:

$$\begin{aligned} \text{dd}(a) &= 0 & \text{dd}(\lambda) &= 0 \\ \text{dd}(\varphi \oplus \psi) &= \text{máx}\{\text{dd}(\varphi), \text{dd}(\psi)\} & \text{dd}(\varepsilon\alpha) &= \text{dd}(\alpha) \\ \text{dd}(\neg\varphi) &= \text{dd}(\varphi) & \text{dd}([\varphi]\alpha) &= \text{máx}\{\text{dd}(\varphi), \text{dd}(\alpha)\} \\ \text{dd}(\langle\alpha\rangle) &= \text{dd}(\alpha) & \text{dd}(\downarrow\alpha) &= 1 + \text{dd}(\alpha) \\ \text{dd}(\langle\alpha \odot \beta\rangle) &= \text{máx}\{\text{dd}(\alpha), \text{dd}(\beta)\} \end{aligned}$$

donde $a \in \mathbb{A}$, $\oplus \in \{\wedge, \vee\}$, $\odot \in \{=, \neq\}$, y α es cualquier expresión de camino o el string vacío λ . Definimos $\ell\text{-XPath}_=(\downarrow)$ como el fragmento de $\text{XPath}_=(\downarrow)$ que consiste de todas las fórmulas φ con $\text{dd}(\varphi) \leq \ell$.

2.1.2. Equivalencia lógica

Sean \mathcal{T} y \mathcal{T}' árboles de datos, y sean $u \in T$, $u' \in T'$. Decimos que \mathcal{T}, u y \mathcal{T}', u' son **ℓ -equivalentes para $\text{XPath}_=(\downarrow)$** (notación: $\mathcal{T}, u \equiv_{\ell}^{\downarrow} \mathcal{T}', u'$) sii para toda expresión de nodo $\varphi \in \ell\text{-XPath}_=(\downarrow)$, vale que $\mathcal{T}, u \models \varphi$ sii $\mathcal{T}', u' \models \varphi$.

2.1.3. Bisimulación

Sean \mathcal{T} y \mathcal{T}' dos árboles de datos. Decimos que $u \in T$ y $u' \in T'$ son **ℓ -bisimilares para $\text{XPath}_=(\downarrow)$** (notación: $\mathcal{T}, u \leftrightarrow_{\ell}^{\downarrow} \mathcal{T}', u'$) si existe una familia de relaciones $(Z_j)_{j \leq \ell}$ en $T \times T'$ tal que $uZ_{\ell}u'$ y para todo $j \leq \ell$, $x \in T$ y $x' \in T'$ se cumple que

- **Harmony:** Si xZ_jx' entonces $\text{label}(x) = \text{label}(x')$.
- **Zig:** Si xZ_jx' , $x \xrightarrow{n} v$ y $x \xrightarrow{m} w$ con $n, m \leq j$ entonces existen $v', w' \in T'$ tales que $x' \xrightarrow{n} v'$, $x' \xrightarrow{m} w'$ y
 1. $\text{data}(v) = \text{data}(w) \Leftrightarrow \text{data}(v') = \text{data}(w')$,
 2. $(\xrightarrow{i} v) Z_{j-n+i} (\xrightarrow{i} v')$ para todo $0 \leq i < n$, y
 3. $(\xrightarrow{i} w) Z_{j-m+i} (\xrightarrow{i} w')$ para todo $0 \leq i < m$.
- **Zag:** Si xZ_jx' , $x' \xrightarrow{n} v'$ y $x' \xrightarrow{m} w'$ con $n, m \leq j$ entonces existen $v, w \in T$ tales que $x \xrightarrow{n} v$, $x \xrightarrow{m} w$ y los items 1, 2 y 3 de arriba se verifican.

2.1.4. Relación entre equivalencia y bisimulación

Teorema 1. $\mathcal{T}, u \stackrel{\downarrow}{\equiv}_{\ell} \mathcal{T}', u'$ sii $\mathcal{T}, u \equiv_{\ell}^{\downarrow} \mathcal{T}', u'$.

Demostración. Por las proposiciones 8 y 10 de [4]. □

2.2. comp-Bisimulación

En esta sección definimos nuestra aproximación y demostramos que existe una correspondencia entre la equivalencia lógica y la bisimulación. Para ello nos basamos en [4].

2.2.1. Definición del fragmento

Sea $comp \subseteq \mathbb{A} \times \mathbb{A}$ el conjunto de pares de labels cuyos datos me interesa comparar.

Definimos $comp\text{-XPath}_{=}(↓)$ como el fragmento de $\text{XPath}_{=}(↓)$ en el cual para toda fórmula del tipo $\langle \alpha \odot \beta \rangle$ vale que

- α y β están en forma normal y tienen la estructura $\gamma[a(\wedge\varphi)^*]^1$ con $a \in \mathbb{A}$.
- $(lastLabel(\alpha), lastLabel(\beta)) \in comp$, donde $lastLabel(\alpha) = lastLabel(\gamma[a(\wedge\varphi)^*]) = a$.

La asignación hecha por $lastLabel(\delta)$ tiene una ambigüedad en el caso que $\delta \equiv \gamma[a \wedge b \wedge \bigwedge_{i \in I} \varphi_i]$, con $a \neq b$. Sin embargo, en tal caso $\delta \equiv [\neg(\epsilon)]$, una expresión de camino siempre falsa que pertenece a $comp\text{-XPath}_{=}(↓)$ para cualquier $comp \subseteq \mathbb{A} \times \mathbb{A}$. Luego, se puede tomar cualquier asignación para $lastLabel(\delta)$ sin cambiar el poder expresivo de la lógica.

Observación 2. Son válidas las siguientes equivalencias:

- $\langle \gamma[a_1 \vee \dots \vee a_n] \odot \beta \rangle \equiv \langle (\gamma[a_1] \cup \dots \cup \gamma[a_n]) \odot \beta \rangle \equiv \langle \gamma[a_1] \odot \beta \rangle \vee \dots \vee \langle \gamma[a_n] \odot \beta \rangle$
- $\langle \gamma[\neg a_k] \odot \beta \rangle \equiv \langle \gamma[a_1 \vee \dots \vee a_{k-1} \vee a_{k+1} \vee \dots \vee a_n] \odot \beta \rangle$ si $\mathbb{A} = \{a_1, \dots, a_{k-1}, a_k, a_{k+1}, \dots, a_n\}$
- $\langle \gamma \odot \beta \rangle \equiv \langle \gamma[a_1 \vee \dots \vee a_n] \odot \beta \rangle$ si $\mathbb{A} = \{a_1, \dots, a_n\}$

Esto nos permite traducir fórmulas que por temas sintácticos no pertenecen al fragmento a otras equivalentes que sí lo están.

2.2.2. Equivalencia lógica

Sean \mathcal{T} y \mathcal{T}' árboles de datos, y sean $u \in T$, $u' \in T'$. Decimos que \mathcal{T}, u y \mathcal{T}', u' son **comp-equivalentes para $\text{XPath}_{=}(↓)$** (notación: $\mathcal{T}, u \equiv_{comp}^{\downarrow} \mathcal{T}', u'$) sii para toda expresión de nodo $\varphi \in comp\text{-XPath}_{=}(↓)$, vale que $\mathcal{T}, u \models \varphi$ sii $\mathcal{T}', u' \models \varphi$.

2.2.3. Bisimulación

Sean \mathcal{T} y \mathcal{T}' dos árboles de datos. Decimos que $u \in T$ y $u' \in T'$ son **comp-bisimilares para $\text{XPath}_{=}(↓)$** (notación: $\mathcal{T}, u \stackrel{\downarrow}{\equiv}_{comp} \mathcal{T}', u'$) si existe una relación $Z \subseteq T \times T'$ tal que uZu' y para todo $x \in T$ y $x' \in T'$ se cumple que

- **Harmony:** Si xZx' entonces $label(x) = label(x')$.

¹ Es decir, $a \wedge \bigwedge_{i \in I} \varphi_i$, donde I es un conjunto finito (y posiblemente vacío) de índices.

- **Zig:** Si xZx' , $x \xrightarrow{n} v$ y $x \xrightarrow{m} w$ entonces existen $v', w' \in T'$ tales que $x' \xrightarrow{n} v'$, $x' \xrightarrow{m} w'$ y
 1. $(label(v), label(w)) \in comp \Rightarrow (data(v) = data(w) \Leftrightarrow data(v') = data(w'))$,
 2. $(\xrightarrow{i} v) Z (\xrightarrow{i} v')$ para todo $0 \leq i < n$, y
 3. $(\xrightarrow{i} w) Z (\xrightarrow{i} w')$ para todo $0 \leq i < m$.
- **Zag:** Si xZx' , $x' \xrightarrow{n} v'$ y $x' \xrightarrow{m} w'$ entonces existen $v, w \in T$ tales que $x \xrightarrow{n} v$, $x \xrightarrow{m} w$ y los items 1, 2 y 3 de arriba se verifican.

2.2.4. Relación entre equivalencia y bisimulación

A continuación demostramos que $\Leftrightarrow_{comp}^\downarrow$ coincide con \equiv_{comp}^\downarrow en árboles de datos de ramificación finita.

Teorema 3. $\mathcal{T}, u \Leftrightarrow_{comp}^\downarrow \mathcal{T}', u'$ implica $\mathcal{T}, u \equiv_{comp}^\downarrow \mathcal{T}', u'$. La vuelta también vale cuando \mathcal{T} y \mathcal{T}' son de ramificación finita.

Proposición 4. $\mathcal{T}, u \Leftrightarrow_{comp}^\downarrow \mathcal{T}', u'$ implica $\mathcal{T}, u \equiv_{comp}^\downarrow \mathcal{T}', u'$.

Demostración. Probamos que para todo $\varphi, \alpha \in comp\text{-XPath}_=(\downarrow)$ vale lo siguiente:

1. Si xZx' entonces $\mathcal{T}, x \models \varphi \Leftrightarrow \mathcal{T}', x' \models \varphi$;
2. Si $x \xrightarrow{n} y$, $x' \xrightarrow{n} y'$ y $(\xrightarrow{i} y) Z (\xrightarrow{i} y')$ para todo $0 \leq i \leq n$, entonces $(x, y) \in \llbracket \alpha \rrbracket^{\mathcal{T}} \Leftrightarrow (x', y') \in \llbracket \alpha \rrbracket^{\mathcal{T}'}$.

Realizamos la prueba por inducción en $|\varphi| + |\alpha|$. Analizamos primero el ítem 1.

El caso base es $\varphi = a$ para cierto $a \in \mathbb{A}$. Por Harmony, $label(x) = label(x')$ y entonces $\mathcal{T}, x \models \varphi$ sii $\mathcal{T}', x' \models \varphi$. Los casos $\varphi = \{\neg\psi, \psi_1 \wedge \psi_2, \psi_1 \vee \psi_2\}$ son triviales.

Sea $\varphi = \langle \alpha = \beta \rangle = \langle \alpha_1[a(\wedge\psi)^*] = \beta_1[b(\wedge\psi')^*] \rangle$ con $(a, b) \in comp$. Mostramos sólo la implicación \Rightarrow (la otra implicación es análoga), por lo que asumimos que $\mathcal{T}, x \models \varphi$. Entonces existen $y, z \in T$ tal que $(x, y) \in \llbracket \alpha \rrbracket^{\mathcal{T}}$ (y en particular vale que $label(y) = a$), $(x, z) \in \llbracket \beta \rrbracket^{\mathcal{T}}$ (y en particular vale que $label(z) = b$) y $data(y) = data(z)$. En particular, para ciertos n, m vale que $x \xrightarrow{n} y$ y $x \xrightarrow{m} z$. Por zig sabemos que existen $y', z' \in T'$ tal que $x' \xrightarrow{n} y'$ y $x' \xrightarrow{m} z'$. Aplicando HI dos veces, llegamos a que $(x', y') \in \llbracket \alpha \rrbracket^{\mathcal{T}'}$ y $(x', z') \in \llbracket \beta \rrbracket^{\mathcal{T}'}$. Como también es cierto que $(label(y), label(z)) \in comp$ y teníamos que $data(y) = data(z)$, entonces $data(y') = data(z')$. Juntando todo, tenemos que $\mathcal{T}', x' \models \langle \alpha = \beta \rangle$.

Los casos $\varphi = \langle \alpha \neq \beta \rangle$ y $\varphi = \langle \alpha \rangle$ son similares al anterior.

Ahora veamos el ítem 2, mostrando nuevamente sólo la implicación \Rightarrow .

Los casos base son cuando $\alpha \in \{\varepsilon, \downarrow\}$. Si $\alpha = \varepsilon$ entonces $y = x$ y por ende $n = 0$. Como $y' = x'$, concluimos $(x', y') \in \llbracket \alpha \rrbracket^{\mathcal{T}'}$. Si $\alpha = \downarrow$ entonces $x \rightarrow y$ en \mathcal{T} , por ende $n = 1$. Como $x' \rightarrow y'$, concluimos $(x', y') \in \llbracket \alpha \rrbracket^{\mathcal{T}'}$.

Para el paso inductivo, sean $x_0, \dots, x_n \in T$ y $x'_0, \dots, x'_n \in T'$ tales que

$$\begin{aligned} x &= x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n = y && \text{en } \mathcal{T}, \\ x' &= x'_0 \rightarrow x'_1 \rightarrow x'_2 \rightarrow \dots \rightarrow x'_n = y' && \text{en } \mathcal{T}', \end{aligned}$$

y $x_i Z x'_i$ para todo $0 \leq i \leq n$. Asumamos, por contradicción, que $(x', y') \notin \llbracket \alpha \rrbracket^{\mathcal{T}'}$. Entonces, existe una subfórmula φ de α y $k \in \{0, \dots, n\}$ tal que $\mathcal{T}, x_k \models \varphi$ y $\mathcal{T}', x'_k \not\models \varphi$ como muestra el Lema 9 de [4], y además $\varphi \in \text{comp-XPath}_=(\downarrow)$ por ser subfórmula de $\alpha \in \text{comp-XPath}_=(\downarrow)$. Pero esto contradice la hipótesis inductiva 1. Entonces necesariamente debe valer $(x', y') \in \llbracket \alpha \rrbracket^{\mathcal{T}'}$. \square

Proposición 5. $\mathcal{T}, u \equiv_{\text{comp}}^{\downarrow} \mathcal{T}', u'$ implica $\mathcal{T}, u \leftrightarrow_{\text{comp}}^{\downarrow} \mathcal{T}', u'$ ($\mathcal{T}, \mathcal{T}'$ de ramificación finita).

Demostración. Fijamos $u \in T$ y $u' \in T'$ tal que $\mathcal{T}, u \equiv_{\text{comp}}^{\downarrow} \mathcal{T}', u'$. Definimos Z como

$$x Z x' \quad \text{sii} \quad \mathcal{T}, x \equiv_{\text{comp}}^{\downarrow} \mathcal{T}', x'.$$

Queremos mostrar que Z es una *comp*-bisimulación entre \mathcal{T}, u y \mathcal{T}', u' . Por hipótesis, $u Z u'$. Z cumple con Harmony por la fórmula $\varphi = a$ ($a \in \mathbb{A}$). Veamos ahora que Z satisface Zig (el caso para Zag es análogo). Supongamos $x Z x'$,

$$\begin{aligned} x &= v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_n = v && \text{en } \mathcal{T}, \\ x &= w_0 \rightarrow w_1 \rightarrow \dots \rightarrow w_m = w && \text{en } \mathcal{T}. \end{aligned}$$

y $\text{data}(v) = \text{data}(w)$ (el caso $\text{data}(v) \neq \text{data}(w)$ se demuestra de manera similar). Sea $P \subseteq T'^2$ definido como

$$P = \{(v', w') \mid x' \xrightarrow{n} v' \wedge x' \xrightarrow{m} w'\}.$$

Tenemos que P es finito porque \mathcal{T} es de ramificación finita y es no vacío porque $\mathcal{T}, x \equiv_{\text{comp}}^{\downarrow} \mathcal{T}', x'$, $\mathcal{T}, x \models \langle \downarrow^n \rangle$ y $\mathcal{T}, x \models \langle \downarrow^m \rangle$.

Sean $l_v = \text{label}(v)$ y $l_w = \text{label}(w)$ con $l_v, l_w \in \mathbb{A}$. Si $(l_v, l_w) \notin \text{comp}$ entonces no nos interesa si $\text{data}(v') = \text{data}(w')$. En caso contrario, si tenemos $\varphi = \langle \downarrow^n [l_v] = \downarrow^m [l_w] \rangle$, al valer que $\mathcal{T}, x \models \varphi$, también debe valer $\mathcal{T}', x' \models \varphi$ (pues $\varphi \in \text{comp-XPath}_=(\downarrow)$). Por lo tanto el conjunto $P_{\text{data}} \subseteq T'^2$ definido como $P_{\text{data}} = \{(v', w') \mid (v', w') \in P \wedge \text{data}(v') = \text{data}(w')\}$ también es finito y no vacío. Sea $P' = P_{\text{data}}$ en caso que $(l_v, l_w) \in \text{comp}$ y $P' = P$ en caso contrario.

Queremos ver que además existe un par $(v', w') \in P'$ tal que:

- I. $x' = v'_0 \rightarrow v'_1 \rightarrow \dots \rightarrow v'_n = v'$ en \mathcal{T}' ,
- II. $x' = w'_0 \rightarrow w'_1 \rightarrow \dots \rightarrow w'_m = w'$ en \mathcal{T}' ,
- III. $(\forall i \in \{0, \dots, n\}) \mathcal{T}, v_i \equiv_{\text{comp}}^{\downarrow} \mathcal{T}', v'_i$, y
- IV. $(\forall j \in \{0, \dots, m\}) \mathcal{T}, w_j \equiv_{\text{comp}}^{\downarrow} \mathcal{T}', w'_j$,

y por lo tanto Z satisface Zig. A modo de contradicción, asumamos que para todo $(v', w') \in P'$ satisfaciendo I y II vale alguna de las siguientes:

- (a) $(\exists i \in \{0, \dots, n\}) \mathcal{T}, v_i \not\equiv_{\text{comp}}^{\downarrow} \mathcal{T}', v'_i$, o
- (b) $(\exists j \in \{0, \dots, m\}) \mathcal{T}, w_j \not\equiv_{\text{comp}}^{\downarrow} \mathcal{T}', w'_j$.

Fijemos \top como cualquier tautología. Para cada $(v', w') \in P'$ definimos dos familias de expresiones de nodo,

$$\varphi_{v',w'}^0, \dots, \varphi_{v',w'}^n \quad \text{y} \quad \psi_{v',w'}^0, \dots, \psi_{v',w'}^m,$$

de la siguiente forma:

- Asumamos (a) y que i es el menor índice tal que $\mathcal{T}, v_i \not\equiv_{\text{comp}}^{\downarrow} \mathcal{T}', v'_i$. Sea $\varphi_{v',w'}^i \in \text{comp-XPath}_=(\downarrow)$ tal que $\mathcal{T}, v_i \models \varphi_{v',w'}^i$ pero $\mathcal{T}', v'_i \not\models \varphi_{v',w'}^i$. Para $k \in \{0, \dots, n\} \setminus \{i\}$, sea $\varphi_{v',w'}^k = \top$, y para $k \in \{0, \dots, m\}$, sea $\psi_{v',w'}^k = \top$.
- Asumamos que (a) no vale. Entonces (b) vale. Sea j el menor índice tal que $\mathcal{T}, w_j \not\equiv_{\text{comp}}^{\downarrow} \mathcal{T}', w'_j$. Sea $\psi_{v',w'}^j \in \text{comp-XPath}_=(\downarrow)$ tal que $\mathcal{T}, w_j \models \psi_{v',w'}^j$ pero $\mathcal{T}', w'_j \not\models \psi_{v',w'}^j$. Para $k \in \{0, \dots, m\} \setminus \{j\}$, sea $\psi_{v',w'}^k = \top$, y para $k \in \{0, \dots, n\}$, sea $\varphi_{v',w'}^k = \top$.

Para cada $i \in \{0, \dots, n\}$ y $j \in \{0, \dots, m\}$, sean

$$\Phi^i = \bigwedge_{(v',w') \in P'} \varphi_{v',w'}^i \quad \text{y} \quad \Psi^j = \bigwedge_{(v',w') \in P'} \psi_{v',w'}^j. \quad (2.1)$$

Como P' es finito, cada Φ^i y Ψ^j son fórmulas bien formadas de $\text{comp-XPath}_=(\downarrow)$. Finalmente, definamos $\alpha = [\Phi^0] \downarrow [\Phi^1] \downarrow \dots \downarrow [l_v \wedge \Phi^n]$ y $\beta = [\Psi^0] \downarrow [\Psi^1] \downarrow \dots \downarrow [l_w \wedge \Psi^m]$ donde $l_v = \text{label}(v)$ y $l_w = \text{label}(w)$ con $l_v, l_w \in \mathbb{A}$ ($\alpha, \beta \in \text{comp-XPath}_=(\downarrow)$). Es claro que por construcción $(x, v) \in \llbracket \alpha \rrbracket^{\mathcal{T}}$ y $(x, w) \in \llbracket \beta \rrbracket^{\mathcal{T}}$.

Analicemos primero el caso $(l_v, l_w) \notin \text{comp}$. Por lo dicho anteriormente, $\mathcal{T}, x \models \langle \alpha \rangle$ y $\mathcal{T}, x \models \langle \beta \rangle$. Veamos que $\mathcal{T}', x' \not\models \langle \alpha \rangle$ o $\mathcal{T}', x' \not\models \langle \beta \rangle$. Esto contradice $\mathcal{T}, x \equiv_{\text{comp}}^{\downarrow} \mathcal{T}', x'$, y así podríamos terminar. Supongamos que $\mathcal{T}', x' \models \langle \alpha \rangle$ y $\mathcal{T}', x' \models \langle \beta \rangle$. Entonces existe $(v', w') \in P'$ (ya que $P' = P$) tal que $(x', v') \in \llbracket \alpha \rrbracket^{\mathcal{T}'}$ y $(x', w') \in \llbracket \beta \rrbracket^{\mathcal{T}'}$.

Si se da el caso que $(l_v, l_w) \in \text{comp}$, entonces $\mathcal{T}, x \models \langle \alpha = \beta \rangle$. Veamos que $\mathcal{T}', x' \not\models \langle \alpha = \beta \rangle$. Esto contradice $\mathcal{T}, x \equiv_{\text{comp}}^{\downarrow} \mathcal{T}', x'$ (ya que $\langle \alpha = \beta \rangle \in \text{comp-XPath}_=(\downarrow)$), y así podríamos terminar. Supongamos que $\mathcal{T}', x' \models \langle \alpha = \beta \rangle$. Entonces existe $(v', w') \in P'$ (ya que $P' = P_{\text{data}}$) tal que $(x', v') \in \llbracket \alpha \rrbracket^{\mathcal{T}'}$ y $(x', w') \in \llbracket \beta \rrbracket^{\mathcal{T}'}$.

En particular y en cualquiera de los casos anteriores, I y II son verdaderos, y entonces vale (a) o (b). Si vale (a), tenemos por construcción que $(x', v') \notin \llbracket \alpha \rrbracket^{\mathcal{T}'}$, y si en cambio vale (b) es claro que $(x', w') \notin \llbracket \beta \rrbracket^{\mathcal{T}'}$. En cualquiera de los casos, llegamos a una contradicción. \square

2.2.5. Consideraciones prácticas

Hasta ahora presentamos ciertos formalismos sobre la *comp*-bisimulación, con las pruebas pertinentes. Más allá de la teoría, este nuevo tipo de aproximación puede ser implementado computacionalmente y pareciera tener beneficios prácticos. La motivación que llevó a idear esta aproximación fue aumentar los pares de nodos que son bisimilares en un árbol (es decir tener una relación con pocas particiones y muchos nodos en cada una de ellas). Esto se logra relajando las condiciones, en este caso, el ítem 1 de Zig y Zag. En consecuencia, se produce una disminución del tiempo de respuesta de las queries $\varphi \in \text{comp-XPath}_=(\downarrow)$: al tener menos particiones y más grandes, es esperable que se reduzca la cantidad de nodos en los que efectivamente hay que evaluar φ . Además, la complejidad del cálculo inicial puede disminuir porque la relajación de las condiciones hace que las verificaciones Zig-Zag sean más simples.

Existen en principio tres cuestiones de implementación a resolver. La primera consiste en cómo almacenar *comp*. La decisión no abarca solamente qué estructura de datos utilizar (teniendo en cuenta que deberá permitir operaciones de consulta, iterado, agregado, borrado), sino también qué hacer con respecto a cuestiones como simetría (donde existe un tradeoff entre espacio de memoria a utilizar y complejidad algorítmica). La segunda consiste justamente en cómo se modifica el algoritmo, específicamente la verificación de Zig y Zag, para chequear *comp* antes de realizar la comparación de datos.

La tercera y última respecta a qué hacer con las fórmulas φ tal que $\varphi \notin \text{comp-XPath}_=(\downarrow)$. Una (mala) opción es directamente omitir dichas queries y no contestarlas. Otra opción que funciona pero tampoco parece buena es evaluar la query entera (es decir, cada una de sus subqueries) en todos los nodos, sin tener en cuenta la bisimulación calculada. La mejor opción es aprovechar Z para las subqueries que sí pertenezcan a $\text{comp-XPath}_=(\downarrow)$ y hacer la verificación manual en aquellas que no pertenezcan. Entre estos, tenemos dos casos para analizar. El primero es cuando la subfórmula tiene la forma $\langle \alpha[a(\wedge \varphi_a)^*] \odot \beta[b(\wedge \varphi_b)^*] \rangle$ con $(a, b) \notin \text{comp}$. Aquí no podemos hacer más que lo dicho antes. El otro caso es la subfórmula $\langle \alpha \odot \beta \rangle$ con α y/o β sin tener la forma sintácticamente correcta $\gamma[c(\wedge \psi)^*]$, por ejemplo $\varphi = \langle \downarrow = \downarrow \rangle$. Aquí podemos realizar un procesamiento sobre la subfórmula para reescribirla cómo otra equivalente pero que pertenezca a $\text{comp-XPath}_=(\downarrow)$, utilizando las equivalencias dadas en la Observación 2. Este procesamiento puede ser costoso, pero quizás no tanto como la evaluación en todos los nodos.

Más allá de lo citado en la Observación 2 y a la tercera cuestión anterior, podemos utilizar otras equivalencias para optimizar el tiempo de respuesta frente a fórmulas φ tales que $\varphi \notin \text{comp-XPath}_=(\downarrow)$. La idea es retrasar la evaluación de las proposiciones que no pertenecen al fragmento, verificando primero las que sí pertenecen (ya que para estas se puede utilizar Z , lo que reduce los tiempos). La ejecución de estas subfórmulas permite evitar la evaluación de las otras que son lentas. Algunos ejemplos son:

- Si tenemos $\varphi = \varphi_1 \oplus \varphi_2$ con $\oplus \in \{\wedge, \vee\}$, $\varphi_1 \notin \text{comp-XPath}_=(\downarrow)$ y $\varphi_2 \in \text{comp-XPath}_=(\downarrow)$, es conveniente utilizar la fórmula equivalente $\varphi' = \varphi_2 \oplus \varphi_1$.
- Si tenemos $\varphi = \langle \alpha \odot \beta \rangle$ con $\varphi \notin \text{comp-XPath}_=(\downarrow)$ y $\alpha, \beta \in \text{comp-XPath}_=(\downarrow)$, puede ser conveniente utilizar la fórmula equivalente $\varphi' = \langle \alpha \rangle \wedge \langle \beta \rangle \wedge \langle \alpha \odot \beta \rangle$.

2.2.6. Ejemplos

La Figura 2.1 muestra un ejemplo de un data tree \mathcal{T} de $Trees(\mathbb{A} \times \mathbb{D})$ con $\mathbb{A} = \{A, B, C, D, E\}$ y $\mathbb{D} = \mathbb{N}$. Cada nodo está representado con su label y su dato, y en rojo se muestra su Id.

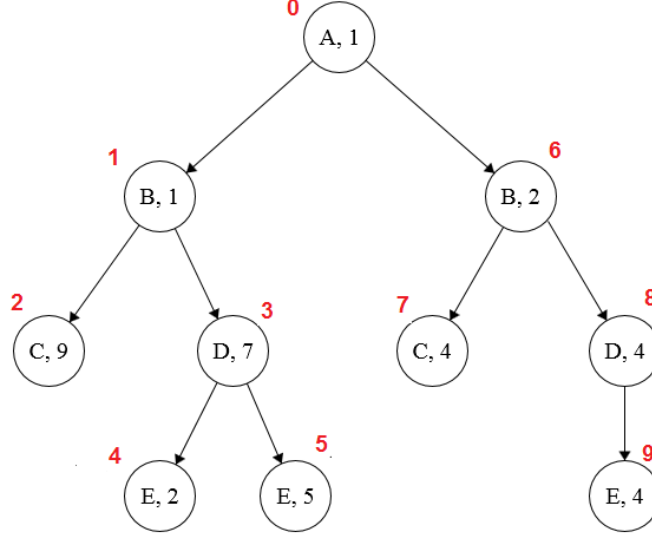


Fig. 2.1: Ejemplo de data tree para *comp*-autobisimulación

Podemos calcular la *comp*-autobisimulación para diferentes relaciones *comp*. Vamos a suponer que Z_{comp} es implícitamente reflexiva y simétrica, es decir $(x, x) \in Z_{comp}$ para todo $x \in T$ y si $(x, y) \in Z_{comp} \Rightarrow (y, x) \in Z_{comp}$ aunque no lo escribamos explícitamente.

- $comp = \emptyset$: Ninguna fórmula del tipo $\langle \alpha \odot \beta \rangle$ pertenece a $comp\text{-XPath}_=(\downarrow)$. De cierta forma, se descartan los datos de los nodos y así se reduce la *comp*-bisimulación a una noción de bisimulación modal. La relación Z_{comp^1} que describe esta *comp*-autobisimulación es $Z_{comp^1} = \{(1, 6), (2, 7), (3, 8), (4, 5), (4, 9), (5, 9)\}$.
- $comp = \mathbb{A} \times \mathbb{A}$: Todas las fórmulas del tipo $\langle \alpha \odot \beta \rangle$ con el formato correcto pertenecen a $comp\text{-XPath}_=(\downarrow)$. Teniendo en cuenta las equivalencias provistas en la Observación 2, todas las fórmulas son válidas y por lo tanto estamos en el caso de $\text{XPath}_=(\downarrow)$. La relación Z_{comp^2} que describe esta *comp*-autobisimulación es $Z_{comp^2} = \{(2, 7), (4, 5), (4, 9), (5, 9)\}$.
- $comp = \{(C, D)\}$: Sólo se permiten comparar datos de nodos con labels *C* y *D*. La relación Z_{comp^3} en este caso es $Z_{comp^3} = \{(2, 7), (3, 8), (4, 5), (4, 9), (5, 9)\}$.
- $comp = \{(E, E)\}$: Sólo se permiten comparar datos de nodos con label *E*. Vale $Z_{comp^4} = \{(2, 7), (4, 5), (4, 9), (5, 9)\}$. El par de nodos (3,8) no pertenece a Z_{comp^4} pues la fórmula $\langle \downarrow [E] \neq \downarrow [E] \rangle$ los distingue. El par (1,6) tampoco pertenece a Z_{comp^4} pues la fórmula $\langle \downarrow \downarrow [E] \neq \downarrow \downarrow [E] \rangle$ los distingue.

Con los tres primeros ejemplos se puede deducir la siguiente Observación:

Observación 6. *La comp-bisimulación efectivamente es una noción distinta a la bisimulación modal y a la bisimulación de XPath₌(↓).*

Demostración. Vale pues tenemos $Z_{comp^2} \subsetneq Z_{comp^3} \subsetneq Z_{comp^1}$. □

Los labels {A,B,C,D,E} podrían ser {Curso, Alumno, Parcial, Final, Ejercicio} respectivamente. En este caso, tomando el tercer ejemplo de los anteriores, tenemos $comp = \{(Parcial, Final)\}$. Esto podría ser resultado de que las comparaciones de datos sólo (o la mayoría) se hicieron entre nodos con dichos labels, por ejemplo para ver qué alumnos tienen la misma nota entre ambos tipos de exámenes. De esta forma, distinguimos a los alumnos 1 y 2 pues el primero tiene distintas notas (9 en el Parcial, 7 en el Final) y el segundo la misma (4 en ambos casos), y por lo tanto $(1, 6) \notin Z_{comp^3}$. Por otro lado, como no es interesante (no hubo una cantidad considerable de queries de este tipo) comparar notas de los Ejercicios del Final, vale que $(3, 8) \in Z_{comp^3}$.

Observación 7. *La \emptyset -bisimulación es equivalente a la bisimulación modal y por lo tanto, se pueden usar las herramientas desarrolladas para dicha noción, como por ejemplo los índices 1-Index, F&B-Index, etc.*

3. ACTUALIZACIÓN DE APROXIMACIONES

En este capítulo y en los posteriores, cuando hagamos referencia a bisimulaciones, vamos a estar trabajando en realidad con autobisimulaciones. Esto no es una restricción ni una pérdida de generalidad, ya que las bisimulaciones pueden pensarse como autobisimulaciones en el grafo que resulta de hacer la unión disjunta de los dos árboles de datos \mathcal{T} y \mathcal{T}' . Este grafo conserva las propiedades buenas de los árboles, en especial la propiedad que indica que la cantidad de caminos que se pueden recorrer en un punto haciendo Zig y Zag es polinomial respecto al tamaño del grafo. Por otro lado, siempre que nos referimos a una bisimulación o autobisimulación, nos referimos a la máxima.

En el capítulo anterior enunciamos y probamos que para ambos tipos de aproximación, dado un fragmento de XPath₌(\downarrow) existe una relación entre la equivalencia lógica (\equiv^{\downarrow}) y la bisimulación ($\leftrightarrow^{\downarrow}$). Esta última está dada por una Z (o una familia de ellas). Entonces, sólo sirve usar dicha Z para las fórmulas pertenecientes al fragmento tomado, pues son las únicas que se consideran para la equivalencia lógica. Por ejemplo, si tenemos una ℓ -bisimulación con $\ell = 1$, Z_1 sólo sirve para responder fórmulas con $\text{dd} \leq 1$.

Notación: vamos a escribir $P(\varphi)$ para denotar a las particiones $P_1 = \llbracket \varphi \rrbracket^{\mathcal{T}}$ y $P_2 = T \setminus \llbracket \varphi \rrbracket^{\mathcal{T}}$ resultantes de evaluar φ en \mathcal{T} . Vamos a escribir $Z + P(\varphi)$ para denotar al refinamiento de la relación Z_i por dichas particiones.

3.1. Motivación

El enfoque tradicional de calcular al inicio la bisimulación total es tanto costoso como innecesario. Dados los tipos de queries que se suelen ejecutar, tal exactitud no es requerida y genera un sobre-refinamiento contraproducente. Sin embargo, como usuarios desconocemos el nivel de aproximación que necesitamos. Si lo conociéramos, podríamos eliminar la parte de innecesaria, pero aún tendríamos un costo alto de actualización en el inicio.

Proponemos un nuevo enfoque combinando aproximaciones y actualización incremental. Esta metodología se basa en comenzar con una aproximación simple, que sea fácil de calcular y esté rápidamente operativa. Eliminamos el factor de sobre-refinamiento dejando que el sistema regule a qué aproximación actualizar, en base a las consultas previamente ejecutadas. Reducimos el factor costo de actualización utilizando los aprendizajes que resultan de evaluar expresiones fuera del fragmento del lenguaje. Para estos objetivos debemos sacrificar un poco los tiempos de evaluación, ya que ciertas evaluaciones las tendremos que hacer sobre el *data tree* sin poder aprovechar la relación de bisimulación más conveniente.

3.2. Proceso de actualización

3.2.1. ℓ -Bisimulación

Vamos a usar el ejemplo de la Figura 3.2 para ver ciertos detalles interesantes sobre la ℓ -bisimulación. En lo posterior, al dar cierto Z_k vamos a hacerlo definiendo por extensión un conjunto de particiones, de la forma $\{P_1, \dots, P_n\}$. Se entiende que para todo $x, y \in P_i$ vale $xZ_k y$, y para todo u, v tal que $u \in P_i, v \in P_j$ con $i \neq j$ NO vale $uZ_k v$.

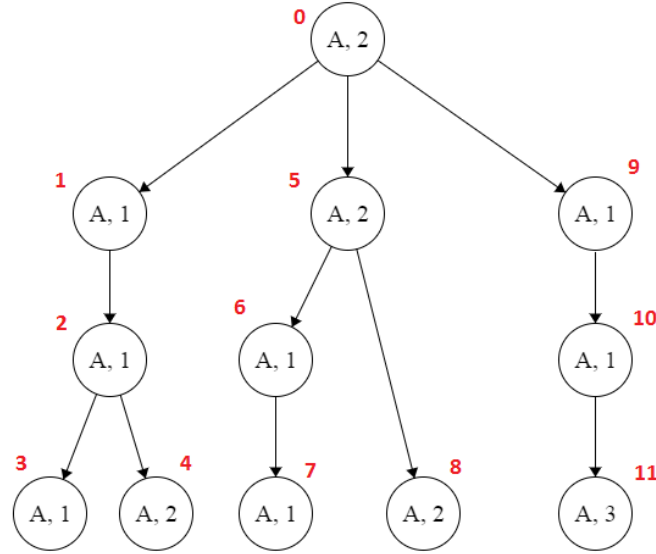


Fig. 3.1: Ejemplo de data tree para ℓ -bisimulación

En la 0-bisimulación sólo verificamos la condición de Harmony, por lo que particionamos a los nodos únicamente por su label. En este caso todos los nodos tienen el mismo label (A) por lo que tenemos $Z_0 = \{\{0,1,2,3,4,5,6,7,8,9,10,11\}\}$.

Para la 1-bisimulación ya vemos a los nodos hijos, y tenemos en cuenta las expresiones de nodo φ tal que $dd(\varphi) \leq 1$. Calculamos y obtenemos $Z_1 = \{P_1, P_2, P_3, P_4\}$ con $P_1 = \{0,5,2\}$, $P_2 = \{1,9,6\}$, $P_3 = \{8,3,4,7,11\}$ y $P_4 = \{10\}$. Veamos por qué se dividió la única partición de Z_0 en 4 particiones:

- En P_3 no vale $\langle \downarrow \rangle$.
- En P_2 vale $\langle \downarrow \rangle$ pero no vale $\langle \epsilon \neq \downarrow \rangle$.
- En P_4 vale $\langle \downarrow \rangle$ y $\langle \epsilon \neq \downarrow \rangle$ pero no vale $\langle \downarrow \neq \downarrow \rangle$.
- En P_1 vale $\langle \downarrow \rangle$, $\langle \epsilon \neq \downarrow \rangle$ y $\langle \downarrow \neq \downarrow \rangle$.

Para la 2-bisimulación ya vemos a los nodos hijos y a los hijos de estos, y tenemos en cuenta las expresiones de nodo φ tal que $dd(\varphi) \leq 2$. Calculamos y obtenemos $Z_2 = \{P_{11}, P_{12}, P_{13}, P_{21}, P_{22}, P_{23}, P_3, P_4\}$ con $P_{11} = \{0\}$, $P_{12} = \{5\}$, $P_{13} = \{2\}$, $P_{21} = \{1\}$, $P_{22} = \{9\}$, $P_{23} = \{6\}$, $P_3 = \{8,3,4,7,11\}$ y $P_4 = \{10\}$. Veamos por qué se dividieron las particiones P_1 y P_2 de Z_1 en 3 particiones cada una:

- En P_{13} no vale $\langle \downarrow \downarrow \rangle$. Lo mismo sucede para P_{23} .

- En P_{1_2} vale $\langle \downarrow \downarrow \rangle$ pero no vale $\langle \downarrow \downarrow \neq \downarrow \downarrow \rangle$. Lo mismo sucede para P_{2_2} .
- En P_{1_1} vale $\langle \downarrow \downarrow \rangle$ y $\langle \downarrow \downarrow \neq \downarrow \downarrow \rangle$. Lo mismo sucede para P_{2_1} .

Es importante notar que al pasar de una ℓ -bisimulación a una ℓ' -bisimulación con $\ell < \ell'$ estamos haciendo un refinamiento, es decir, tomamos cada partición de Z_ℓ y la mantenemos igual o la subdividimos. Lo que no puede suceder es que dos nodos que están en distinta partición de Z_ℓ luego se encuentren en la misma en $Z_{\ell'}$. Por ejemplo, P_{1_1} y P_{2_1} son particiones distintas, porque sigue ocurriendo que la primera valida $\langle \epsilon \neq \downarrow \rangle$ y la segunda no.

Proceso de actualización

Veamos ahora para este mismo ejemplo cómo funciona el proceso de actualización. Supongamos que empezamos en una 0-bisimulación. Como dijimos anteriormente, agrupamos sólo por label y así obtenemos $Z_0 = \{\{0,1,2,3,4,5,6,7,8,9,10,11\}\}$. Analicemos qué sucede a medida que llegan las siguientes queries:

- $\varphi_1 = \langle \downarrow \rangle$. Vale que $dd(\varphi_1) = 1 (> 0)$ y por lo tanto no puedo usar Z_0 . Ejecutamos la query y obtenemos como resultado $\llbracket \varphi_1 \rrbracket^T = \{0,1,5,9,2,6,10\}$. En Z_0 teníamos a todos los nodos en la misma partición. Este resultado nos dice que en Z_1 los nodos $\{0,1,5,9,2,6,10\}$ van a estar en una partición distinta al resto ($\{8,3,4,7,11\}$) ya que existe al menos una fórmula de profundidad 1 que los distingue. Guardamos este aprendizaje particionando T en $\llbracket \varphi_1 \rrbracket^T$ y $T \setminus \llbracket \varphi_1 \rrbracket^T$ obteniendo $\{P_1^1, P_2^1\}$ con $P_1^1 = \{0,1,5,9,2,6,10\}$ y $P_2^1 = \{8,3,4,7,11\}$. No podemos asegurar que $Z_1 = Z_0 + P(\varphi_1)$ pero sí es información que nos va a ser útil si en algún momento decidimos pasar a una 1-bisimulación.
- $\varphi_2 = \langle \epsilon \neq \downarrow \rangle$. Vale que $dd(\varphi_2) = 1 (> 0)$ y por lo tanto nuevamente no puedo usar Z_0 . Ejecutamos la query y obtenemos como resultado $\llbracket \varphi_2 \rrbracket^T = \{0,5,2,10\}$. En Z_0 teníamos a todos los nodos en la misma partición. Este resultado nos dice que en Z_1 los nodos $\{0,5,2,10\}$ van a estar en una partición distinta al resto ($\{1,9,6,8,3,4,7,11\}$) ya que existe al menos una fórmula de profundidad 1 que los distingue. Podemos guardar este nuevo aprendizaje, planteando ahora $P(\varphi_2) = \{P_1^2, P_2^2\}$ con $P_1^2 = \{0,5,2,10\}$ y $P_2^2 = \{1,9,6,8,3,4,7,11\}$.
- $\varphi_3 = \langle \downarrow \downarrow \downarrow \rangle$. Vale que $dd(\varphi_3) = 3 (> 0)$ y por lo tanto nuevamente no puedo usar Z_0 . Ejecutamos la query y obtenemos como resultado $\llbracket \varphi_3 \rrbracket^T = \{0\}$. En Z_0 teníamos a todos los nodos en la misma partición. Este resultado nos dice que en Z_3 el nodo $\{0\}$ va a estar en una partición distinta al resto ($\{1,2,3,4,5,6,7,8,9,10,11\}$) ya que existe al menos una fórmula de profundidad 3 que los distingue. Podemos guardar este nuevo aprendizaje, planteando ahora $P(\varphi_3) = \{P_1^3, P_2^3\}$ con $P_1^3 = \{0\}$ y $P_2^3 = \{1,2,3,4,5,6,7,8,9,10,11\}$.

Supongamos ahora que ejecutamos el proceso de mining (que veremos más adelante) y como resultado del mismo, decidimos actualizar a la 1-bisimulación. Queremos aprovechar la mayor cantidad posible de información aprendida para realizar esta actualización de manera más eficiente (más rápida en la práctica). Lo que vamos a hacer es tomar de nuestra «base de conocimiento» a todos los φ_j tal que $dd(\varphi_j) = 1$, en nuestro ejemplo son φ_1 y φ_2 . Luego, vamos a calcular \widetilde{Z}_1 (o podemos asumir que ya lo tenemos pues se recalcula

luego de cada query con profundidad 1) cómo $\widetilde{Z}_1 = Z_0 + P(\varphi_1) + P(\varphi_2) = \{\widetilde{P}_1, \widetilde{P}_2, \widetilde{P}_3\}$ con $\widetilde{P}_1 = \{0, 5, 2, 10\}$, $\widetilde{P}_2 = \{1, 9, 6\}$ y $\widetilde{P}_3 = \{8, 3, 4, 7, 11\}$. Finalmente, calculamos Z_1 a partir de \widetilde{Z}_1 con la siguiente idea (formalizada luego): tomamos cada partición \widetilde{P}_i de \widetilde{Z}_1 , verificamos para cada par de nodos en ella Zig y Zag, y si no cumplen refinamos la partición. En nuestro ejemplo, las particiones \widetilde{P}_2 y \widetilde{P}_3 se mantienen igual como P_2 y P_3 , mientras que la partición \widetilde{P}_1 la dividimos entre $P_1 = \{0, 5, 2\}$ y $P_4 = \{10\}$ llegando así al mismo Z_1 que mostramos en la sección anterior.

El cálculo de \widetilde{Z}_1 (ya sea que se realizó anteriormente o que se realiza al momento de querer actualizar) es un proceso iterativo, y es más preciso escribirlo como $((Z_0 + P(\varphi_1)) + P(\varphi_2))$. Se utilizan las particiones de $P(\varphi_2)$ para refinar las de $Z_0 + P(\varphi_1)$. De esta forma la partición P_1^1 de $Z_0 + P(\varphi_1)$ se refina en las dos particiones \widetilde{P}_1 y \widetilde{P}_2 de \widetilde{Z}_1 . Podemos plantear para el caso general la siguiente Observación:

Observación 8. *Tenemos hasta el momento $\widetilde{Z}_\ell = Z_{\ell-1} + P(\eta_1) + \dots + P(\eta_k)$. Refinamos \widetilde{Z}_ℓ en $\widetilde{Z}_\ell + P(\eta_{k+1})$ tomando el aprendizaje de la query η_{k+1} .*

Lo interesante de este enfoque es que cuanto más información se aprende, más preciso queda \widetilde{Z}_1 y probablemente también con particiones más chicas. Esto último provoca que el costo total de la verificación con Zig y Zag decremente, dado que hay menos pares de nodos para comprobar. Si la partición a verificar tiene n nodos, el peor caso en cuanto a cantidad es cuando cualquier par de nodos es no-bisimilar entre sí por Zag, lo que equivale a $\sum_{i=1}^{n-1} 2i = 2 \sum_{i=1}^{n-1} i = n(n-1)$ verificaciones. Vale entonces lo siguiente:

Observación 9. *Si \widetilde{Z}_ℓ tiene m particiones P_1, \dots, P_m , la cantidad máxima de verificaciones Zig-Zag a realizar es $\sum_{j=1}^m |P_j|(|P_j| - 1)$.*

En nuestro ejemplo, si me baso en Z_0 tengo en el peor caso 132 verificaciones, mientras que si me baso en \widetilde{Z}_1 tengo que hacer a lo sumo 12 verificaciones para \widetilde{P}_1 , otras 6 para \widetilde{P}_2 y 20 para \widetilde{P}_3 . Esto da un total de 38 verificaciones, que es menos de un tercio del otro caso.

En cuanto a la precisión, surge la pregunta de si es posible (y beneficioso) responder a queries usando \widetilde{Z}_1 (suponiendo que lo tenemos precalculado) en vez de simplemente Z_0 . Para queries de profundidad 0 es conveniente usar Z_0 , ya que en este caso el refinamiento no sólo no es útil sino que en la práctica es perjudicial (pues tengo mas particiones). Para queries de profundidad mayor o igual a 2 no me sirve ni Z_0 ni \widetilde{Z}_1 (ni aún Z_1 aunque lo tuviera). A las queries de profundidad 1 las divido entre las que nunca fueron ejecutadas y las que sí. Para el primer caso, el refinamiento provisorio \widetilde{Z}_1 no me da seguridad, y por lo tanto no puedo usarlo. Para el segundo caso, asumiendo que tengo almacenados todas las queries ejecutadas y que tengo un algoritmo que determine si la nueva query es igual a alguna de ellas (por ejemplo, comparando sintácticamente elemento a elemento), sí puedo responder con seguridad usando dicho refinamiento.

Es posible realizar actualizaciones de más de un nivel, es decir pasar de una ℓ -bisimulación a una $(\ell + k)$ -bisimulación con $k > 1$. Igualmente necesitamos calcular todas las aproximaciones intermedias $Z_{\ell+1}, \dots, Z_{\ell+k-1}$, para tener el conjunto entero $\{Z_0, \dots, Z_{\ell+k-1}\}$. Entonces, iterativamente calculamos las relaciones intermedias, tomando las queries con la profundidad correspondiente. Adicionalmente, resulta provechoso mantener almacenadas todas las Z_i más pequeñas que ya se calcularon para poder responder de forma más rápida (o a lo sumo igual) queries con profundidad menor a $\ell + k$.

Algo importante a destacar y demostrar es la siguiente Proposición:

Proposición 10. Z_ℓ refina a \widetilde{Z}_ℓ .

Demostración. Supongamos por el absurdo que Z_ℓ no refina \widetilde{Z}_ℓ . Existe entonces una partición $P \in Z_\ell$ de tamaño al menos dos, con nodos distintos $n_1, n_2 \in P$ tal que no existe ninguna partición $\widetilde{P} \in \widetilde{Z}_\ell$ que contenga a ambos n_1, n_2 . Sea \widetilde{P}_1 la partición de \widetilde{Z}_ℓ que contiene a n_1 y \widetilde{P}_2 la partición de \widetilde{Z}_ℓ que contiene a n_2 . Por el Lema 11 (a continuación), existen fórmulas en $\ell\text{-XPath}_=(\downarrow)$ que distinguen cada par de particiones distintas en \widetilde{Z}_ℓ . Sea $\varphi \in \ell\text{-XPath}_=(\downarrow)$ la fórmula que distingue \widetilde{P}_1 y \widetilde{P}_2 . Vale entonces que $n_1 \not\equiv_\ell^\downarrow n_2$ pues $n_1 \models \varphi$ y $n_2 \not\models \varphi$ (o al revés). Luego, por el Teorema 3 vale que $n_1 \not\equiv_\ell^\downarrow n_2$. Entonces no puede ser que n_1 y n_2 pertenezcan a la misma partición en Z_ℓ . Llegamos entonces a una contradicción que surgió de suponer que Z_ℓ no refina a \widetilde{Z}_ℓ . \square

Lema 11. Para cada par de particiones distintas P_1, P_2 en \widetilde{Z}_ℓ , existe una fórmula $\varphi \in \ell\text{-XPath}_=(\downarrow)$ que las distingue y que es satisfecha en todo nodo de P_1 .

Demostración. Observar que basta probar que hay una φ que distingue a las particiones: si φ las distingue, también lo hace $\neg\varphi$, y alguna de ellas debe valer en P_1 .

Ahora probamos el lema para cualquier ℓ por inducción en $n \geq 0$, donde $\widetilde{Z}_\ell = Z_{\ell-1} + P(\varphi_1) + P(\varphi_2) + \dots + P(\varphi_n)$.

- Caso base: tenemos $\widetilde{Z}_\ell = Z_{\ell-1}$. Sean $P_1, P_2 \in Z_{\ell-1}$ dos particiones distintas, y $n_1 \in P_1$ y $n_2 \in P_2$. Por estar en particiones distintas de $Z_{\ell-1}$, sabemos que $n_1 \not\equiv_{\ell-1}^\downarrow n_2$. Luego, por el Teorema 3 vale que $n_1 \not\equiv_{\ell-1}^\downarrow n_2$. Entonces existe $\varphi \in (\ell-1)\text{-XPath}_=(\downarrow)$ tal que $n_1 \models \varphi$ y $n_2 \not\models \varphi$ (o al revés). φ distingue las particiones P_1 y P_2 .
- Paso inductivo: suponemos que vale para $\widetilde{Z}_\ell^n = Z_{\ell-1} + P(\varphi_1) + P(\varphi_2) + \dots + P(\varphi_n)$ y tenemos que ver que también vale para $\widetilde{Z}_\ell^{n+1} = Z_{\ell-1} + P(\varphi_1) + P(\varphi_2) + \dots + P(\varphi_n) + P(\varphi_{n+1}) = \widetilde{Z}_\ell^n + P(\varphi_{n+1})$. Tomemos dos particiones cualesquiera distintas $\widetilde{P}_1^{n+1}, \widetilde{P}_2^{n+1} \in \widetilde{Z}_\ell^{n+1}$. Por construcción, tal cual fue explicado anteriormente en la Observación 8, \widetilde{Z}_ℓ^{n+1} es un refinamiento de \widetilde{Z}_ℓ^n . Entonces existen $\widetilde{P}_1^n, \widetilde{P}_2^n \in \widetilde{Z}_\ell^n$ tal que $\widetilde{P}_1^{n+1} \subseteq \widetilde{P}_1^n$ y $\widetilde{P}_2^{n+1} \subseteq \widetilde{P}_2^n$. Si $\widetilde{P}_1^n \neq \widetilde{P}_2^n$ (y luego son disjuntas), por hipótesis inductiva existe una fórmula que las distingue, y la misma también distingue a \widetilde{P}_1^{n+1} de \widetilde{P}_2^{n+1} . En caso contrario tenemos que $\widetilde{P}_1^n = \widetilde{P}_2^n$, y aquí se ve claramente que la fórmula que distingue \widetilde{P}_1^{n+1} de \widetilde{P}_2^{n+1} es φ_{n+1} . Además $\varphi_{n+1} \in \ell\text{-XPath}_=(\downarrow)$ por definición de \widetilde{Z}_ℓ (recordemos que tomábamos todas las fórmulas ψ tal que $\text{dd}(\psi) = \ell$). \square

A partir de este Lema surge el siguiente Corolario:

Corolario 12. Para cada clase de equivalencia P dentro de un conjunto de clases de equivalencia \mathbb{P} dado por \widetilde{Z}_ℓ , existe una fórmula $\psi \in \ell\text{-XPath}_=(\downarrow)$ tal que $\mathcal{T}, u \models \psi$ sii $u \in P$. Es decir, dicha fórmula caracteriza a la partición.

Demostración. Supongamos que tenemos el conjunto de clases de equivalencia $\mathbb{P}^n = P_1, \dots, P_n$ dado por \widetilde{Z}_ℓ . Dividamos en dos casos:

- Para el caso $n = 1$, tomamos el nodo u de P_1 y la fórmula que vale en todos los nodos de la clase es $\psi = \text{label}(u)$ (notar que $\psi \in \ell\text{-XPath}_=(\downarrow)$ para todo ℓ). Este caso se da por ejemplo cuando el árbol de datos tiene un único label y estamos viendo Z_0 .
- Para el caso $n > 1$, la fórmula que caracteriza la partición P_i de \widetilde{Z}_ℓ es $\psi_i = \bigwedge_{i \neq j} \varphi_{i,j}$ donde $\varphi_{i,j}$ es la expresión dada por el Lema 11 que distingue la partición P_i de P_j y que vale en P_i . Al ser una conjunción de fórmulas que valen en P_i , entonces $u \in P_i \Rightarrow \mathcal{T}, u \models \psi_i$. Por otro lado, si tenemos $u \in P_j$ con $i \neq j$, entonces $\mathcal{T}, u \not\models \varphi_{i,j}$ y por lo tanto $\mathcal{T}, u \not\models \psi_i$. Esto concluye en que $u \notin P_i \Rightarrow \mathcal{T}, u \not\models \psi_i$. De esta forma llegamos a que $\mathcal{T}, u \models \psi_i$ sii $u \in P_i$.

□

Observación 13. *La fórmula que caracteriza a cierta partición P de un conjunto de particiones dado por \widetilde{Z}_ℓ nunca es única. Por ejemplo, si la fórmula ψ caracteriza P , también lo hace la fórmula $\psi \wedge (a \vee \neg a)$.*

3.2.2. *comp*-Bisimulación

Al igual que para la ℓ -Bisimulación, la idea es comenzar con la aproximación más básica. En este caso tomamos $\text{comp} = \emptyset$ y calculamos la relación Z_{comp} correspondiente. Al ser esta una noción equivalente a la de bisimulación modal, podemos usar el algoritmo de Paige y Tarjan, o alguna de las ideas de los trabajos previos para calcularla. Esta aproximación irá luego mejorando, utilizando lo aprendido al responder ciertas consultas.

A medida que van llegando las queries para evaluar, se analiza qué tipo de expresión es. Si la subfórmula η que se está respondiendo es de la forma $\langle \alpha \odot \beta \rangle$, y como es posible asumir que se hicieron las transformaciones mencionadas anteriormente, podemos tomar $(\text{lastLabel}(\alpha), \text{lastLabel}(\beta))$. Si la tupla pertenece a comp , entonces se puede usar la relación para acelerar la respuesta. En caso contrario, debemos hacer la verificación manual en cada uno de los nodos, llegando al resultado deseado $\llbracket \eta \rrbracket^{\mathcal{T}}$. Conservamos este aprendizaje almacenando $P(\eta)$.

En determinado momento, se ejecuta el proceso de mining. En este caso dicho proceso retorna $L = \{(a_1, b_1), \dots, (a_k, b_k)\}$ el conjunto de pares de labels nuevos cuyos datos son interesantes de comparar. Vamos a suponer por el momento que sólo se agregan pares de labels y no se eliminan. Podemos calcular $\text{comp}' = \text{comp} \cup L$, queremos obtener $Z_{\text{comp}'}$. Para esto, vamos a obtener de nuestra base de conocimientos el conjunto de fórmulas $\Psi = \{\varphi \mid \varphi = \langle \alpha \odot \beta \rangle \text{ con } (\text{lastLabel}(\alpha), \text{lastLabel}(\beta)) \in L\}$. Si $\Psi = \{\varphi_1, \dots, \varphi_m\}$, entonces calculamos iterativamente $\widetilde{Z}_{\text{comp}'} = Z_{\text{comp}} + P(\varphi_1) + \dots + P(\varphi_m)$.

Finalmente, debemos hacer ciertas verificaciones manuales extras para llegar a $Z_{\text{comp}'}$. Tomando cada par de nodos en $\widetilde{Z}_{\text{comp}'}$, comprobamos que realmente sean comp' -bisimilares, chequeando Zig y Zag (teniendo en cuenta que pasamos de comp a comp') y refinando la relación cuando no cumplen. Terminadas estas validaciones, conseguimos $Z_{\text{comp}'}$.

Si admitimos que también se elimine un par de labels, entonces ante estos casos tenemos dos opciones. La primera es conservar la relación Z_{comp} actual, ya que lo que estamos haciendo es reduciendo el fragmento de la lógica de $\text{comp}\text{-XPath}_=(\downarrow)$ a $\text{comp}'\text{-XPath}_=(\downarrow)$ con $\text{comp}' \subsetneq \text{comp}$. La desventaja de esta alternativa es que tenemos un sobre-refinamiento, lo que en ciertos casos va a producir un deterioro de la performance. La otra opción es

ajustar Z_{comp} . En este caso no tenemos aprendizaje que nos facilite la actualización. Lo que se debe hacer es tomar (de forma ordenada) cada par de clases de equivalencia, verificar (adecuadamente) las condiciones de Zig y Zag entre un representante de cada clase y unificar las clases si la verificación es satisfactoria. Podríamos optimizar esta tarea si tuviéramos almacenadas las fórmulas que distinguen cada par de clases de equivalencia (que sabemos existen por el Lema 15). Si la fórmula que distingue entre dos clases pertenece a $comp'-XPath_{=}(↓)$ entonces las clases van a seguir siendo disjuntas. En caso contrario, cómo la fórmula no es única, tengo que necesariamente hacer la verificación con Zig y Zag.

Otra posible solución sería conservar todas las relaciones Z generadas, como sucede para la ℓ -bisimulación. El problema es que para esta aproximación el espacio de memoria requerido es $O(2^{\binom{n(n+1)}{2}})$ donde n simboliza el cardinal del conjunto \mathbb{A} de labels del documento. El número de subconjuntos distintos de un conjunto C es $2^{|C|}$, y en este caso $C \subseteq \mathbb{A} \times \mathbb{A}$ es el conjunto de pares de labels eliminando toda simetría (si $(a, b) \in C \Rightarrow (b, a) \notin C$). Luego, vale que $|C| = \frac{n(n+1)}{2}$. La necesidad de espacio exponencial (respecto a la cantidad de labels) hace impráctica esta solución, que sí funciona para la ℓ -bisimulación ya que en dicho caso, el espacio de memoria requerido es $O(h)$ con h siendo la altura del árbol.

En este proceso de actualización también sucede que cuanto más el sistema aprende, más preciso queda $\tilde{Z}_{comp'}$ y por ende menor cantidad de verificaciones de Zig y Zag hay que realizar. Al reducir la cantidad de dichas operaciones que resultan costosas, mejora la performance del proceso de actualización. Asimismo, valen las siguientes propiedades, análogas a las enunciadas para la ℓ -bisimulación:

Proposición 14. $Z_{comp'}$ refina a $\tilde{Z}_{comp'}$.

Lema 15. Para cada par de particiones distintas P_1, P_2 en $\tilde{Z}_{comp'}$, existe una fórmula $\varphi \in comp'-XPath_{=}(↓)$ que las distingue y que es satisfecha en todo nodo de P_1 .

Corolario 16. Para cada clase de equivalencia P dentro de un conjunto de clases de equivalencia \mathbb{P} dado por $\tilde{Z}_{comp'}$, existe una fórmula $\psi \in comp'-XPath_{=}(↓)$ tal que $\mathcal{T}, u \models \psi$ si $u \in P$. Es decir, dicha fórmula caracteriza a la partición.

Demostración. Similares a la de la Proposición 10, el Lema 11 y el Corolario 12 respectivamente. \square

3.2.3. Aproximación genérica de bisimulación

Vimos en la sub-sección anterior que varios conceptos se repiten entre el proceso de actualización para la ℓ -bisimulación y el proceso de actualización para la $comp$ -bisimulación. Las mismas se pueden generalizar para otras aproximaciones de bisimulación que se definan, mientras cumplan ciertas condiciones. La condición general es la definición de $(Z_i)_{i \in I}$ donde (I, \leq) es un conjunto parcialmente ordenado que es un semirretículo superior, esto es, cumple

$$\forall i, j \in I, \exists k \in I \mid i \leq k, j \leq k.$$

Deben valer:

- Si $i \leq j$, entonces Z_j refina a Z_i ,
- Para todo i , Z (dado por bisimulación) refina a Z_i

- Para toda fórmula o conjunto finito de fórmulas, si se utilizan para refinar Z_i en \tilde{Z}_i , existe $j \in I$ tal que Z_j refina a \tilde{Z}_i ,
- Debe existir una relación entre la equivalencia lógica (\equiv^\downarrow) y la bisimulación ($\leftrightarrow^\downarrow$), y entre cada Z_i con un fragmento correspondiente de la lógica, para que bajo ciertas condiciones (ejemplo: ramificación finita) valga que $\mathcal{T}, u \leftrightarrow_i^\downarrow \mathcal{T}', u'$ sii $\mathcal{T}, u \equiv_i^\downarrow \mathcal{T}', u'$.

En caso de cumplirse todas estas condiciones, son válidas y pueden ser utilizadas como resultados las propiedades análogas a la Proposición 10, el Lema 11 y el Corolario 12.

En el proceso genérico, primero se construye una aproximación simple, que quizás no es muy eficiente pero sí es fácil de calcular. A medida que van llegando las fórmulas para evaluar, se analiza si esta pertenece al fragmento del lenguaje en cuestión (que depende del tipo de aproximación tomada). En caso afirmativo, se intenta utilizar la relación Z de la aproximación para poder responder la consulta sin tener que evaluarla. En caso negativo, se evalúa la fórmula y se almacena el resultado como un aprendizaje.

Luego de responder cada query, se ejecuta un proceso de mining que determina si es conveniente actualizar la aproximación, y en caso afirmativo, que características debería tener, dependiendo esto del tipo de aproximación (el nuevo ℓ , el nuevo *comp*, etc.).

Durante la actualización, se obtienen de la base de conocimientos los aprendizajes que son de utilidad para este proceso (nuevamente esto depende del tipo de aproximación). Iterativamente se toman estos aprendizajes para refinar la Z actual en una \tilde{Z} . Este refinamiento es genérico y por lo tanto no depende del tipo de aproximación. Finalmente, refinando \tilde{Z} haciendo las verificaciones de Zig y Zag necesarias se llega a la Z' deseada. De este refinamiento, sólo la verificación de Zig y Zag depende del tipo de aproximación.

La virtud de este proceso consiste en no perder tiempo en calcular la bisimulación perfecta, sino estar rápidamente operativos con una aproximación básica. Esta se actualiza incrementalmente, valiéndose del aprendizaje realizado durante la evaluación de las queries realizadas. Con un proceso de mining eficiente, se realiza la actualización cuando hay una buena cantidad de aprendizaje, lo que resulta en un \tilde{Z} más aproximado y por lo tanto, menos operaciones costosas como lo son las verificaciones de Zig y Zag.

En el capítulo de Implementación, se muestran los algoritmos pertinentes con su correspondiente explicación. Con ello queda aún más claro qué es genérico y qué depende (y cómo) del tipo de aproximación.

3.3. Proceso de mining

Generar la bisimulación máxima puede ser muy costoso. Es por esto que las aproximaciones suelen ser herramientas útiles en la práctica, ya que tienen un costo mucho menor y pueden responder con rapidez y certeza a cierto conjunto de fórmulas.

En particular para la ℓ -bisimulación, cuanto más grande sea ℓ , más aproximada y certera va a ser la relación Z_ℓ (hasta llegar a la bisimulación máxima), pero también más costosa de calcular. Es por esto que suena conveniente comenzar con una aproximación con un ℓ pequeño (en un principio puede ser $\ell = 0$), que sea rápida de calcular y luego ir mejorando la precisión construyendo de manera incremental una ℓ' -bisimulación con $\ell' > \ell$. La misma idea vale para la *comp*-bisimulación.

Este método es aún más razonable si, como mencionamos en la sección anterior, podemos aprovechar la información aprendida para luego agilizar el proceso de actualización. Por la cantidad de información involucrada, no se utilizan algoritmos de *data mining*, pero el objetivo es similar: tomar decisiones según los datos recolectados.

Ahora bien, la pregunta que surge es cuándo realizar esta actualización; cuándo vale la pena afrontar el costo de este proceso en pos de obtener una bisimulación más aproximada. Una opción, pero que no parece ser muy útil, es que el usuario (o cierto usuario) lo defina, según sus criterios. Sin embargo, cargarle esta responsabilidad al dueño de los datos o a los usuarios que quieren consultarlos parece un despropósito. Aparece entonces el planteo de que el sistema se auto-regule, prescindiendo de actores externos. Como antes, hay una necesidad de tener criterios e información para poder tomar esta decisión.

A diferencia del proceso de actualización, el mining es más difícil de generalizar, ya que ciertas cuestiones a considerar son intrínsecas de cada aproximación.

3.3.1. ℓ -Bisimulación

Nuestra primera idea fue que la información que necesitamos sea la recolectada y aprendida hasta el momento. Nos alimentamos de nuestra «base de conocimientos» no sólo para poder realizar el proceso de actualización de una forma más eficiente, sino también para decidir cuando ejecutarlo. Los datos que precisamos y utilizamos son ciertas características de las queries ejecutadas (para este caso de la ℓ -bisimulación nos importa $dd(\varphi)$, la profundidad de la fórmula φ) y también mediciones de tiempos. Esto último es importante para hacer una buena valoración de la relación costo / beneficio. Es obvio que tener una mejor aproximación nos va a dar algún beneficio (podemos responder eficientemente las mismas queries que antes con las Z_i más chicas almacenadas y nuevas queries con esta mejor aproximación), pero quizás el costo que trae hace que no valga la pena. Por ejemplo, sabemos que si estamos en una ℓ -bisimulación y llega una query φ tal que $dd(\varphi) = \ell' > \ell$, entonces no podemos responder φ usando Z_ℓ . Debemos evaluar φ en cada uno de los nodos del árbol, y esto tarda cierto tiempo t . Si actualizáramos a $Z_{\ell'}$ probablemente podríamos reducir t . El punto es que quizás t no es tan grande (nuevamente, según algún criterio) como para que convenga dicha actualización, con el costo (tiempo, memoria, a nivel operación) que trae aparentada. Este también puede tratar de medirse, y aquí entra en juego lo visto en la sección anterior. Si el sistema «aprendió» mucho, entonces tendrá un $\widetilde{Z}_{\ell'}$ más cercano a $Z_{\ell'}$ y así el proceso de actualización resultará menos costoso.

La dificultad para definir estos criterios tiene que ver con la heterogeneidad de los documentos XML y con el desconocimiento de su estructura (más allá que se puedan co-

nocer ciertas características como altura, cantidad de labels distintos, cantidad promedio de hijos, etc.) y de cómo se realizarán las consultas sobre el mismo (cuántas, cuáles, por cuánto tiempo). Para afrontar esta situación, es necesario hacer ciertas suposiciones y también tomar decisiones que seguramente no sean las mejores para muchos casos.

Una de dichas suposiciones es que se ejecutan muchas queries contra el XML. En caso contrario, para muchos casos no convendría actualizar, ya que este proceso demora un tiempo considerable y es beneficioso sólo si después se lo aprovecha realizando varias consultas.

Mencionamos recientemente que el proceso de actualización resulta menos costoso si $\widetilde{Z}_{\ell'}$ se encuentra cercano a $Z_{\ell'}$. Si $\widetilde{Z}_{\ell'}$ alcanza a $Z_{\ell'}$ entonces conviene actualizar inmediatamente, ya que no podemos reducir más los tiempos del proceso de actualización y mientras tanto estamos teniendo tiempos de evaluación grandes para fórmulas de profundidad ℓ' . Esto puede ocurrir tanto por los refinamientos por aprendizajes o porque $\widetilde{Z}_{\ell'} = Z_{\ell} = Z_{\ell'}$. Si eso no sucede, entonces conviene esperar que se acerque lo «máximo posible» para reducir el tiempo del proceso de actualización. Todo esto funciona bien en la teoría, pero en la práctica no tenemos forma de conocer o medir esta «distancia».

No sabemos cuantas clases tendrá $Z_{\ell'}$, como para comparar contra las clases en $\widetilde{Z}_{\ell'}$. Tampoco hay otra característica de la relación que conozcamos y podamos constatar. Tuvimos en cuenta las siguientes opciones, aunque de alguna forma u otra fueron descartadas:

- Evaluar que se hayan ejecutado todas las fórmulas de profundidad ℓ' que son necesarias para llegar a $Z_{\ell'}$. Esto no es viable porque no se conoce cuales son dichas fórmulas, y pedir que se ejecuten todas las fórmulas posibles con dicha profundidad es un despropósito. Aún si se conociera, el conjunto probablemente no sería único e identificar que se hayan ejecutado efectivamente todas tampoco es algo simple.
- Tener una aproximación de cuánto tiempo demora la evaluación de una fórmula de profundidad ℓ' en $Z_{\ell'}$ y comparar dicho tiempo contra el que se demora en $\widetilde{Z}_{\ell'}$. Sin embargo, esta última no puede usarse para responder ese tipo de fórmulas con seguridad, así que no hay nada contra lo cual comparar dichos tiempos. Además, encontrar una fórmula «promedio» entre todas las existentes es una tarea delicada.
- Similar al punto anterior, tener una aproximación de cuánto tiempo demora el proceso de actualización para un data tree de ciertas características (cantidad de nodos, cantidad de labels distintos, entre otras posibles) y compararlo contra un cálculo en base al $\widetilde{Z}_{\ell'}$ conseguido. Este cálculo se haría contando la cantidad de verificaciones Zig-Zag necesarias tal cual se explica en la Observación 9 y multiplicando por un factor que corresponda a cuánto tarda en promedio una verificación Zig-Zag (para esto la idea era medirlo en varios puntos y tener un número aproximado). Más allá de la dificultad de esto último, los primeros experimentos ya mostraron que la cantidad calculada según lo indicado en dicha Observación es una cota muy grosera, y que el número de verificaciones resulta ser mucho menor, más aún cuando se hicieron varios refinamientos. Dichos resultados invalidan esta opción.

Desechadas todas esas opciones, decidimos cambiar el rumbo afrontando que no tenemos certezas sobre $Z_{\ell'}$ y por lo tanto no podemos medir la distancia que queríamos. La nueva idea que surgió fue calcular el porcentaje de refinamiento entre Z_{ℓ} y $\widetilde{Z}_{\ell'}$, definido como $\frac{|\widetilde{Z}_{\ell'}|}{|Z_{\ell}|} * 100$, y compararlo contra ciertos valores surgidos de la experimentación, que tomamos como «buenos».

Para los experimentos generamos documentos XML de forma «aleatoria» bajo ciertas condiciones: 5 labels distintos (A,B,C,D,E), 3 valores de datos distintos (1,2,3), entre 3 y 7 hijos cada nodo interno y una altura máxima de 22. Tomamos este tipo de documentos como una especie de peor caso, donde sabemos que vamos a tener muchas clases por la falta de patrones de padres e hijos entre los labels. Entre los documentos generados seleccionamos 6 de distinto tamaño, todos ellos medianos/grandes (22851, 50076, 75063, 100090, 150039 y 200082 nodos respectivamente). Para cada uno de estos 6 documentos, evaluamos 35 queries de profundidad 1 que contemplan cualquier distinción de nodos en una 1-bisimulación (para un documento con estos 5 labels). Luego procedimos a actualizar, verificando que la cantidad de clases en \tilde{Z}_1 (el refinamiento por los 35 aprendizajes) coincida con la cantidad de clases en Z_1 . Además de obtener esta cantidad de clases, contabilizamos la cantidad de verificaciones Zig-Zag que efectivamente tuvieron que hacerse y medimos el tiempo que demoró el proceso de actualización. Los resultados correspondientes a porcentajes de refinamiento entre Z_0 y \tilde{Z}_1 se muestran en la Tabla 3.1.

Teniendo estas métricas como las mejores alcanzables, experimentamos con cada documento reduciendo la cantidad de aprendizajes y volviendo a medir tiempos y porcentajes de refinamiento. En base a los tiempos obtenidos, y al mejor tiempo que ya teníamos medido, fuimos decidiendo qué porcentajes de refinamiento íbamos a exigir según la cantidad de nodos del documento. En general, tratamos de quedarnos con porcentajes que resultaran en tiempos menores o iguales al doble del mejor tiempo conseguido anteriormente.

Nodos	% Refinamiento
22851	27000
50076	36000
75063	41000
100090	64000
150039	114000
200082	160000

Tab. 3.1: Porcentaje de refinamiento entre Z_0 y \tilde{Z}_1 según cantidad de nodos

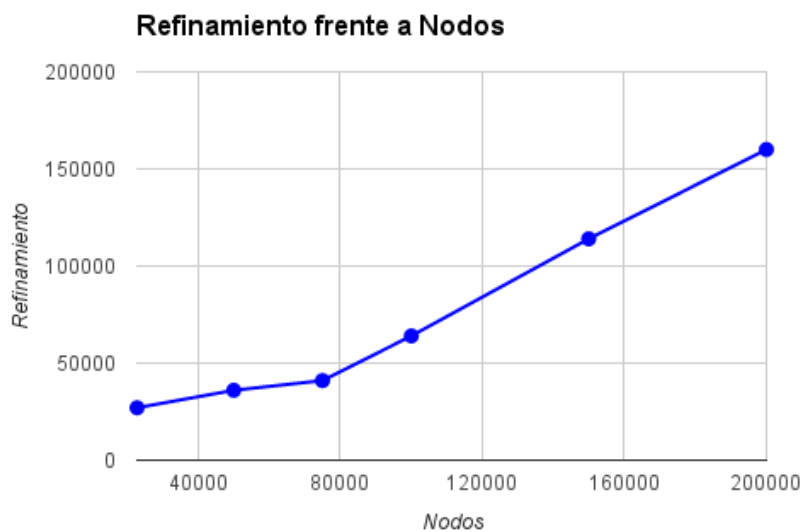


Fig. 3.2: Gráfico de interpolación de los valores de la tabla

Para una aproximación sobre cierto data tree, se alcanza el refinamiento deseado si se verifica que el porcentaje de refinamiento entre Z_ℓ y $\tilde{Z}_{\ell'}$ es mayor o igual al interpolado por los valores de la tabla según la cantidad de nodos del árbol.

Realizamos pruebas para actualizar a Z_2 y Z_3 , verificando en las mismas que los porcentajes esperables eran inalcanzables. Buscamos entonces por profundidad un coeficiente que reduzca el valor que se esperaba conseguir. Dichos coeficientes debían implicar un porcentaje alcanzable, pero al mismo tiempo que siga asegurando un tiempo aceptable de actualización. No pudimos encontrar para estos coeficientes valores relativos, por lo que tomamos valores absolutos correspondientes al promedio entre los resultados obtenidos. Sabiendo que los valores tomados podían no ser los mejores, realizamos algunas pruebas más para asegurarnos que igualmente existan casos en los cuales se alcancen los refinamientos deseados y que los costos de actualización en dichos casos no sean muy elevados.

La idea es actualizar cuando se alcanza el refinamiento deseado. Sin embargo, sabemos que muchas veces esto no va a ser posible. Tenemos que tener en cuenta también los casos en donde la cantidad de aprendizajes es chica, en donde la calidad de los mismos para refinar la relación es baja, y aquellos en los cuales las aproximaciones Z_ℓ permiten nulo o bajo refinamiento por ser iguales o encontrarse muy cerca de $Z_{\ell'}$. Para cualquiera de estos casos, bajo los criterios especificados hasta el momento, nunca se daría la actualización (cuando es algo que sí queremos lograr).

Para esto añadimos una nueva condición que permite actualizar, independiente de la otra. Dicha condición admite la actualización si se ejecutaron más de τ fórmulas de profundidad ℓ' (sin importar que haya repeticiones). Bajo la suposición tomada de que se ejecutan muchas queries, con esto nos aseguramos de eventualmente actualizar y alcanzar $Z_{\ell'}$. La elección de τ hay que hacerla cuidadosamente para no irse hacia ningún extremo: ni actualizar demasiado pronto (cortando posibles futuros aprendizajes que permitan llegar al porcentaje de refinamiento deseado) ni extenderlo demasiado (llegando hasta un punto de casi inanición o induciendo un tiempo de evaluación muy alto).

Tuvimos varias dudas y tardamos bastante en definir el valor de τ . En un primer momento, el enfoque estuvo dirigido a la cantidad de fórmulas posibles de profundidad ℓ' . A medida que ℓ' aumenta, también lo hace ℓ' -XPath $_{=}$ (\downarrow). Existen entonces más fórmulas que pueden distinguir dos nodos en la ℓ' -bisimulación, por lo que queríamos tener también más aprendizajes. El primer valor que definimos fue $2 * n^{2*\ell'}$ (n simbolizando la cantidad de labels distintos del data tree), correspondiente a un aproximado de la cantidad de expresiones de tipo $\langle \downarrow_1 X_1 \dots \downarrow_{\ell'} X_{\ell'} \odot \downarrow_1 Y_1 \dots \downarrow_{\ell'} Y_{\ell'} \rangle$ (donde $X_1, \dots, X_{\ell'}, Y_1, \dots, Y_{\ell'}$ son alguno de los n labels). Redujimos primero el $2 * n^{2*\ell'}$ a $n^{\ell'+1}$, y viendo que el aumento exponencial no era conveniente, volvimos a reducir a $(n^2 * \ell'^2)$. Experimentamos con este valor y también con otros similares (pero con un crecimiento menos pronunciado) cómo $(n^2 * \ell' * \sqrt{\ell'})$ y $(n^2 * \ell')$. Los resultados de estos experimentos nos mostraban que incluso para profundidades no muy grandes ($\ell' \geq 3$), el valor resultaba demasiado grande; esto ocasionaba que la actualización se retardara y por ende, aumentara notoriamente el tiempo de evaluación de las fórmulas que no pertenecían al fragmento del lenguaje. El tiempo total de evaluación se hacía tan grande que no ameritaba el uso de esta técnica de actualización incremental por aprendizajes.

Intentamos una solución de compromiso, utilizando una función constante (respecto a la profundidad) como es n^2 , pero llegando a resultados similares. Llegamos entonces a la conclusión que debíamos utilizar una función que decreciera a medida que aumentara ℓ'

(así como ocurre con el porcentaje de refinamiento deseable). A pesar de lo dicho anteriormente acerca del crecimiento de la cantidad de fórmulas que pueden distinguir dos nodos en la ℓ' -bisimulación, otras razones nos orientaban hacia este camino. Además del ya mencionado dato de los tiempos, para la mayoría de los casos ocurre que el refinamiento que se da a partir de cierta profundidad es chico, y en consecuencia, no hay tanto beneficio en el uso de los aprendizajes. Decidimos finalmente que valga $\tau = 3\kappa$ con $\kappa = \max\{\frac{n^2}{\ell'}, \frac{10}{\sqrt{\ell'}}\}$. El valor $\frac{10}{\sqrt{\ell'}}$ es para tener una cota inferior, para los casos donde n o $\frac{n^2}{\ell'}$ sean muy pequeños.

En la figura 3.3 graficamos para $\ell' = 1, 2, 3$ los tiempos de evaluación acumulativos (desde el inicio hasta llegar a $Z_{\ell'}$) según los distintos κ considerados. El experimento se realizó con un XML de 100 mil nodos y $n = 5$. Se puede observar que para ciertos κ hay un notorio incremento del tiempo de evaluación. En la sub-sección 5.2.1 presentamos otros experimentos con el objetivo de exponer y comparar el rendimiento del proceso incremental.

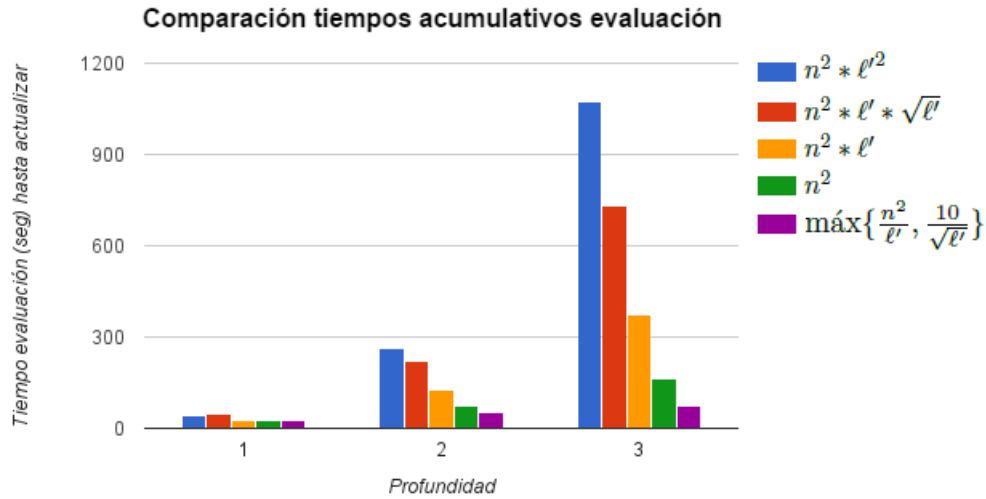


Fig. 3.3: Comparación de tiempos de evaluación según los distintos κ

Teniendo κ como base, experimentamos para ver si con esa cantidad de queries era posible conseguir los refinamientos deseados para actualizar a Z_1 . En estas primeras actualizaciones, los aprendizajes son muy útiles, por lo que actualizar antes de tiempo por tener un valor de τ chico puede ser malo. Viendo rápidamente que no era posible llegar a los refinamientos deseados, duplicamos a 2κ y volvimos a experimentar. Realizamos primero 50 ejecuciones donde en cada una generamos aleatoriamente un data tree (todos de 5 labels, en un rango entre 15 mil y 60 mil nodos) y $2 * (5^2/1) = 50$ queries de profundidad 1 también de forma aleatoria. De las 50 ejecuciones, sólo en 3 se llegó al refinamiento deseable y por ende se actualizó a Z_1 . Se volvió a ejecutar la prueba con otras 50 ejecuciones, pero modificando el algoritmo de generación de queries para forzar a que aparezcan más de la forma $\varphi = \langle \downarrow [l_1] \odot \downarrow [l_2] \rangle$ (pues dichas fórmulas son buenas para refinar). De estas 50 ejecuciones, en 38 se produjo la actualización. Repetimos estas dos tantas de pruebas, pero ya con $3\kappa - 1 = 74$ queries (el -1 es para evitar que se actualice por la otra condición). En este caso, 26 de la primera tanda y las 50 de la segunda lograron actualizar.

De haber tomado una cantidad aún más grande de queries, podríamos haber conseguido quizás más ejecuciones que lograran la actualización (para el caso de fórmulas

generados sin modificar el algoritmo), pero creímos que ya nos estábamos yendo hacia el otro extremo, y que se verían perjudicados los casos de documentos XML con poco refinamiento. Como mencionamos anteriormente, en documentos muy grandes y teniendo una relación Z_ℓ , el tiempo de evaluación de fórmulas de profundidad ℓ' es considerable, por lo que retrasar la actualización implica empeorar el rendimiento del sistema. Satisfechos con los resultados obtenidos, decidimos fijar $\tau = 3\kappa$.

Relacionado con lo anterior, realizamos otras pruebas con el fin de obtener cuántas queries de profundidad 1 son necesarias ejecutar para que el tiempo de evaluación total en Z_0 (es decir, sin actualizar) supere a la siguiente suma: tiempo de evaluación en Z_0 (antes de actualizar) + tiempo de actualización + tiempo de evaluación en Z_1 (después de actualizar, teniendo en cuenta que conservamos Z_0). Si permito actualizar luego de x queries, planteo la siguiente ecuación:

$$2x * tEval_{Z_0} = x * tEval_{Z_0} + tAct + x * tEval_{Z_1}$$

$$x = \frac{tAct}{tEval_{Z_0} - tEval_{Z_1}}$$

Queremos minimizar x para evitar los casos de inanición (pero verificando que no quede un valor muy chico). Dada la ecuación resultante, necesitamos tener un tiempo de evaluación $tAct$ bajo y una diferencia grande entre $tEval_{Z_0}$ y $tEval_{Z_1}$. Con esta motivación, generamos documentos XML con muchos nodos (para que $tEval_{Z_0}$ sea grande) y con pocas clases en Z_1 (para que $tEval_{Z_1}$ sea chico, al igual que el tiempo de actualización). Experimentando con varios documentos con estas características, encontramos que el valor mínimo aproximado era $x = 15$. Esto nos dice que en ningún caso es conveniente actualizar antes de evaluar 30 queries. Este resultado se ajustaba bien a nuestras decisiones, porque para actualizar a $\ell' = 1$ vale que $\kappa \geq 10$ y por lo tanto $\tau \geq 30$. Por otra parte, es cierto que en varios casos $2x < \tau$ y por lo tanto sería conveniente actualizar antes para evitar los tiempos altos de evaluación, pero tampoco es posible tomar un valor que sea el mejor para todos los casos. Optamos por las decisiones que nos parecían las más equilibradas y que no sean malas para ningún caso.

Agregamos una restricción al principio de todo el proceso que consiste en verificar que la cantidad de queries ejecutadas sea mayor a cierto número. Esto es útil para evitar tomar decisiones sobre una base de conocimientos pequeña, que puede no ser representativa. Ya que por las condiciones que definimos antes, no hay actualización posible con menos de κ fórmulas, entonces tomamos nuevamente este valor como cota. Esta restricción también evita que se realicen cálculos de más.

Consideramos agregar una condición extra que evite actualizar (directamente ni siquiera calcular $\tilde{Z}_{\ell'}$) si no hay una mínima cantidad de aprendizajes distintos de profundidad ℓ' . El objetivo de esta condición era evitar que sea actualizara demasiado «rápido», esperando para conseguir mayores aprendizajes y de esta forma acelerando aún más el proceso de actualización. Sin embargo, descartamos esta opción por dos grandes motivos. Primero, los porcentajes de refinamiento estaban ligados de cierta forma a una cantidad de aprendizajes. Por otro lado, el porcentaje pedido ya aseguraba un buen tiempo de actualización, y el tiempo que podía llegar a ahorrarse esperando un poco más no valía la pena frente al tiempo de evaluación en una Z menos aproximada.

El último punto que faltaba decidir era a qué ℓ' íbamos a querer actualizar. Sea dd_φ la profundidad de la fórmula φ . En un primer momento consideramos el conjunto de todas las expresiones evaluadas. Si ya ejecutamos las queries $\varphi_1, \dots, \varphi_n$ entonces tenemos $D = \{dd_{\varphi_1}, \dots, dd_{\varphi_n}\}$. Sin embargo, esto no resulta lo más conveniente: por ejemplo, si se ejecutaron un millón de fórmulas de profundidad 1, para actualizar a Z_2 se requerirán al menos otro millón de fórmulas (en este caso de profundidad 2) que mientras tanto deberán evaluarse en cada nodo del árbol, sin poder aprovechar la información de la bisimulación. Esto genera un incremento en los tiempos de evaluación grande e innecesario, ya que Z_1 se conserva aún luego de actualizar a Z_2 , y por lo tanto las queries de profundidad 1 se van a poder seguir respondiendo con la misma performance.

Si estamos en una ℓ -bisimulación, consideramos entonces el conjunto de expresiones $D' = \{\varphi_i \mid dd_{\varphi_i} > \ell\}$. Podemos elegir tomar un promedio sobre D' , pero no es una buena elección pues, como todo promedio, es muy sensible a valores extremos. Nos inclinamos por elegir otra medida estadística como es la mediana (aunque también consideramos tomar algún otro percentil).

En la sub-sección 4.5.4 se muestra el código de la función que decide si realizar o no la actualización. En el mismo se pueden apreciar las decisiones y restricciones explicadas.

3.3.2. *comp*-Bisimulación

Para la definición de este proceso, inicialmente tomamos como base lo trabajado para la ℓ -bisimulación, y luego lo fuimos modificando. La idea principal sin embargo es la misma: a partir de cierta cantidad de queries (para esperar a tener una base de conocimientos más representativa), se evalúa a qué tipo de aproximación (en este caso a qué conjunto $comp'$) se desea actualizar y se realiza este proceso si se validan ciertas condiciones.

En el caso de la ℓ -bisimulación utilizábamos el valor ℓ' (profundidad que se desea alcanzar) como parámetro de la función que define la cantidad mínima de queries, ya que el nivel de profundidad incide fuertemente en la cantidad de expresiones que pertenecen al fragmento del lenguaje. Esto no ocurre de forma similar en la *comp*-bisimulación, y además, de haberse agregado el tamaño de *comp* como parámetro, se habría favorecido a las primeras tuplas sobre las siguientes. Al ser a priori un comportamiento no deseado, se decidió simplemente usar la cantidad de labels distintas del árbol (n).

En un principio teníamos como cantidad mínima n^2 , al igual que para la ℓ -bisimulación. Sin embargo, decidimos cambiar el valor para tener en cuenta los experimentos con esa aproximación que concluyeron en que la función crecía demasiado rápido, lo que afectaba considerablemente la evaluación de fórmulas fuera del fragmento del lenguaje. En la *comp*-bisimulación usamos documentos con varios labels, para poder justamente mostrar como se ven afectados los procesos al considerar sólo la comparación de ciertas tuplas de labels. Por estos motivos, y luego de algunas pruebas, optamos por el nuevo valor $5n$.

Suponiendo que estamos en una *comp*-bisimulación, evaluamos qué conjunto Σ de tuplas de labels puede ser incorporado para actualizar a una *comp'*-bisimulación con $comp' = comp \cup \Sigma$. Las tuplas candidatas son aquellas que no pertenecen a *comp* y que se ejecutaron al menos κ veces. Decimos que la tupla (a, b) se ejecutó si se ejecutó la expresión $\langle \alpha \odot \beta \rangle$ con $a = lastLabel(\alpha)$ y $b = lastLabel(\beta)$.

En la anterior aproximación ocurría que al actualizar de una ℓ -bisimulación a una ℓ' -bisimulación, la relación Z_ℓ se conserva, por lo que las fórmulas de profundidad ℓ se resuelven en la ℓ' -bisimulación con igual performance que en la ℓ -bisimulación. Esto no sucede en la *comp*-bisimulación, ya que mantenemos una única relación Z (como ya dijimos, mantener en memoria todas las relaciones Z es muy costoso en espacio, necesitando un tamaño de $O(2^{\frac{n(n+1)}{2}})$). Al incorporar nuevas tuplas a *comp*, estamos refinando ciertas clases de Z_{comp} lo que impacta negativamente en la evaluación de fórmulas pertenecientes a *comp*-XPath $_{=}$ (\downarrow). Por este motivo, decidimos involucrar en la definición de κ la cantidad máxima de ejecuciones (C_{max}) entre las tuplas de *comp*, ya que esta es la que resulta más afectada. Resolvimos utilizar una fracción (la cuarta parte) de C_{max} , para evitar situaciones donde tener un C_{max} muy grande impida que otras tuplas se sumen a la aproximación. Luego, vale $\kappa = \max\{5n, \frac{C_{max}}{4}\}$. Aunque finalmente en la práctica el impacto negativo es mínimo, decidimos conservar esta funcionalidad.

La validación de las tuplas candidatas se realiza de manera individual e independiente. Una vez más, el criterio que se quiso tomar fue el de alcanzar cierto porcentaje de refinamiento, que a su vez asegure tiempos reducidos de actualización. En los experimentos realizados para esta aproximación, observamos que en la mayoría de los casos, el refinamiento generado por incluir una nueva tupla t en *comp* es muy acotado y que no dependía de la cantidad de nodos. En un momento intermedio, definimos que se alcanza el refinamiento deseable si los aprendizajes sobre t generan una relación intermedia $\tilde{Z}_{comp \cup t}$ tal que $|\tilde{Z}_{comp \cup t}| \geq |Z_{comp}| * 10$. Sin embargo, esto casi siempre era inalcanzable y además exigía calcular $\tilde{Z}_{comp \cup t}$ para cada t , lo que resultaba en un overhead de tiempos importante. Por ende, finalmente decidimos dejar como única condición para admitir la actualización la ejecución de más de τ fórmulas de la tupla t . Para intentar igualmente conseguir ciertos aprendizajes que produzcan un aceptable refinamiento, y para no retrasar demasiado la actualización, volvimos a definir τ como $\tau = 3\kappa$.

Cabe aclarar lo mencionado acerca de que a priori no se desea favorecer a las primeras tuplas sobre las siguientes. Supongamos un caso de un documento XML con 10 labels distintos, y una aproximación inicial con *comp* = \emptyset . Vale $\kappa = \max\{5n, \frac{C_{max}}{4}\} = \max\{5 * 10, 0\} = 50$ y por lo tanto $\tau = 3\kappa = 150$. Si ejecutamos 150 expresiones que contengan a la tupla t_1 (y a ninguna otra), se realizará la actualización a $Z_{\{t_1\}}$. Si luego se empiezan a ejecutar expresiones que contienen a la tupla t_2 (y a ninguna otra), para asegurar la actualización se necesitan otras 150 consultas, ya que $5n > \frac{C_{max}}{4} = \frac{150}{4}$. Es decir, la segunda tupla a incorporar necesita la misma cantidad de queries que la primera, sin haber favoritismos por el orden. Sin embargo, si luego de la actualización a $Z_{\{t_1\}}$ se realizaron otras 150 ejecuciones de expresiones que contienen a t_1 , va a valer $C_{max} = 300$ y en consecuencia $\kappa = 75$ y $\tau = 225$. Es cierto que aquí se necesitan más queries que contengan a t_2 para que esta se incorpore a *comp* (t_1 se incorporó luego de 150 queries y t_2 necesita 225), pero no es por un tema de orden de las ejecuciones, sino de cantidad.

Entre las tuplas candidatas nos quedamos entonces con aquellas que superan la cantidad τ de ejecuciones. Luego, calculamos la relación intermedia $\tilde{Z}_{comp'}$ incluyendo en esta a todas las tuplas a incorporarse. Al momento de efectuar el proceso de actualización, primero se agregan efectivamente las nuevas tuplas generando el nuevo *comp'* y luego se realiza el refinamiento por Zig-Zag para obtener $Z_{comp'}$.

4. IMPLEMENTACIÓN

4.1. Aplicación Web

Dentro de los objetivos de este trabajo, se encontraba la implementación de una aplicación que pudiera poner en práctica los conceptos teóricos y las ideas enunciadas. Con la misma pretendíamos experimentar, refinar las nociones que lo requerían y obtener resultados cuyo análisis concluyera en la confirmación de los beneficios expuestos.

En el transcurso del desarrollo y con el propósito de facilitar el testing, comenzamos a armar un módulo Web, que provee una perspectiva visual de la solución. Decidimos prestarle importancia a dicho módulo, no sólo por su motivación inicial, sino también como herramienta educativa o de difusión.

El módulo Web cuenta con cuatro páginas, tal cual se ve en la Figura 4.1. Adicionalmente, un botón permite ejecutar un test de regresión, útil para verificar que la aplicación siga funcionando correctamente luego de realizarle alguna modificación. Dicho test realiza comprobaciones básicas sobre la implementación de ambas aproximaciones, sobre los procesos de mining y actualización, y sobre las fórmulas que distinguen clases de equivalencia.

Las 4 páginas se detallan en las sub-secciones que siguen. Algunas funcionalidades (cómo la visualización del árbol correspondiente al documento) son útiles para XMLs chicos, mientras otras están pensadas para usarse con documentos de tamaño más grande. Para estas, se elimina el aspecto visual y se apunta más a estadísticas sobre los distintos procesos (cálculo inicial, evaluación, mining, actualización).

La aplicación fue implementada separando FrontEnd de BackEnd, y comunicando ambos vía AJAX y JSON. Para el FrontEnd utilizamos JQuery¹, la librería Uploadify² para la subida de archivos y la librería Cytoscape.js³ para dibujar el árbol que modela al documento XML. El BackEnd lo desarrollamos con ASP.NET⁴ y C#. Para transformar cadenas de símbolos en expresiones válidas usamos el parser ANTLR⁵.

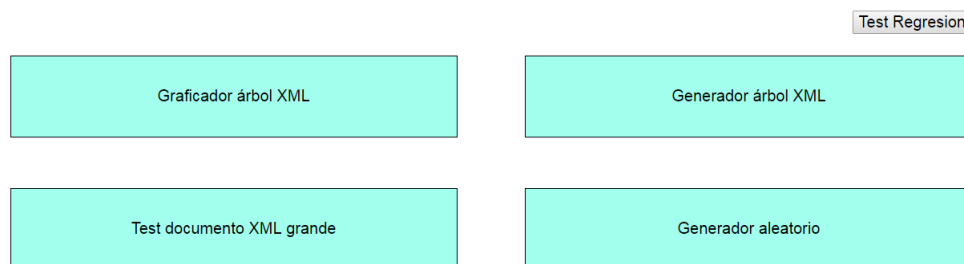


Fig. 4.1: Captura de pantalla de la página inicial del módulo Web

¹ <http://jquery.com/> - Versión 2.0.3

² <http://www.uploadify.com/> - Versión 3.2.1

³ <http://js.cytoscape.org/> - Versión 2.6.1

⁴ <https://www.microsoft.com/net/> - Versión 3.5

⁵ <http://www.antlr.org/> - Versión 4.3

4.1.1. Graficador árbol XML

El objetivo de la página *graficador.html* (Figura 4.2) es graficar un XML como árbol y luego operar con él. Se utiliza de la siguiente manera:

1. Se hace un upload de un documento XML, eligiendo si se quiere trabajar con una ℓ -bisimulación o una *comp*-bisimulación. El sistema comprueba que el documento sea válido, genera el data tree correspondiente e inicializa la aproximación elegida. Esta se guarda en la sesión del servidor para utilizarse luego. Finalmente, se devuelve una representación en forma de árbol en JSON.
2. A partir del JSON devuelto en el punto 1, se grafica el árbol. Cada nodo tiene asociado una tupla $\langle id, label, dato \rangle$ y además un color que corresponde a la clase de equivalencia. Dos nodos en la misma clase tienen el mismo color.
3. Es posible escribir consultas para evaluarlas contra el documento XML. Las consultas se escriben con una sintaxis similar a la utilizada en este trabajo (pero con algunos cambios por no permitir la gramática el uso de caracteres especiales como \downarrow o ϵ). Se muestra la gramática utilizada para permitirle al usuario la escritura de las queries. Al evaluar una de ellas, se listan los ids de los nodos que la satisfacen. Adicionalmente, en el árbol se produce un efecto visual (temporal) para remarcar dichos nodos. Luego de ciertas evaluaciones (según el documento y el proceso de mining) se actualiza la aproximación, modificándose los colores de los nodos en concordancia.
4. Durante la inicialización de la aproximación, en el refinamiento por aprendizajes y durante las verificaciones Zig-Zag, se calcula y almacena el dato de cuál expresión distingue a cada par de clases de equivalencia distintas (existe por Lema 11). Es posible ver la representación de dicha expresión seleccionando el par de nodos que se quiere distinguir (obteniendo primero la clase de cada nodo). Si los nodos pertenecen a la misma clase, se muestra el mensaje “Misma clase”. La expresión que se muestra se satisface en el nodo elegido en el selector de la izquierda y no satisface en el de la derecha. Es posible evaluar esta fórmula.
5. Utilizando el mismo diccionario mencionado en el punto 4, se puede calcular y mostrar la fórmula que caracteriza a una clase de equivalencia, es decir, la que vale en todos sus nodos y únicamente en ellos (ver Corolario 12). Es posible evaluar dicha fórmula, por ejemplo para verificar que realmente cumpla con dichas características.

En la figura 4.2 se muestra un ejemplo de un documento representado como grafo con las distintas operaciones que se pueden ejecutar sobre el mismo. El árbol es el de la Figura 3.2 y representa con colores a las clases de equivalencia de una 1-autobisimulación dada por Z_1 . La relación obtenida es la misma que la calculada oportunamente en la sub-sección 3.2.1. La fórmula que distingue a las clases de los nodos 1 y 5 no es idéntica a la mencionada, pero tal cual se explicó en la Observación, la misma no es única que los distinga. La clase de equivalencia 4 corresponde a los nodos hoja, por lo cual está correctamente caracterizada por la fórmula $\neg(\downarrow \epsilon)$.

L-Bisimulación Comp-Bisimulación
 1

2

Query: <||epsilon>

Resultado: 0,1,2,5,6,9,10 3

```

nexpr : snexpr
        | 'not' nexpr
        | nexpr 'and' nexpr
        | nexpr 'or' nexpr
        | '(' nexpr ')'
        #SimpleExpr
        #NotExpr
        #AndExpr
        #OrExpr
        #ParExpr

snexpr : [a-zA-Z0-9\-\_\_]+;
        | '<' pexpr '>'
        | '<' pexpr 'eq' pexpr '>'
        | '<' pexpr 'neq' pexpr '>'
        #label
        #EPathExpr
        #IPathExpr
        #DPathExpr

pexpr : 'epsilon'
        | '/' pexpr
        | '[' nexpr ']' pexpr
        #epsilon
        #DownPath
        #FilterPath

```

Nodo 1 4

$\neg \langle [A]e \neq [A]e \rangle$

Clase equivalencia 4 5

$\neg \langle \epsilon \rangle$

Fig. 4.2: Captura de pantalla de la página que permite graficar un XML y realizarle consultas

4.1.2. Generador árbol XML

La página *generarArbol.html* (Figura 4.3) da la posibilidad de armarse un árbol de prueba y luego exportar el XML correspondiente. Esto puede hacerse a partir de un XML existente o creando uno nuevo, teniendo que definir el nodo raíz. Luego, en cualquiera de los casos, es posible agregar nodos (indicando label, dato y padre), modificar un nodo (se especifica cuál y se permite cambiar el label y el dato) o borrarlo (se valida que el nodo no tenga descendientes ni sea el nodo raíz). Al presionar el botón “Generar XML” la aplicación guarda en la computadora del usuario el documento XML.

El objetivo de esta funcionalidad es poder crear o modificar casos de prueba para luego utilizarlos en la otra página, generando la aproximación deseada y pudiendo visualizar cómo se comporta.

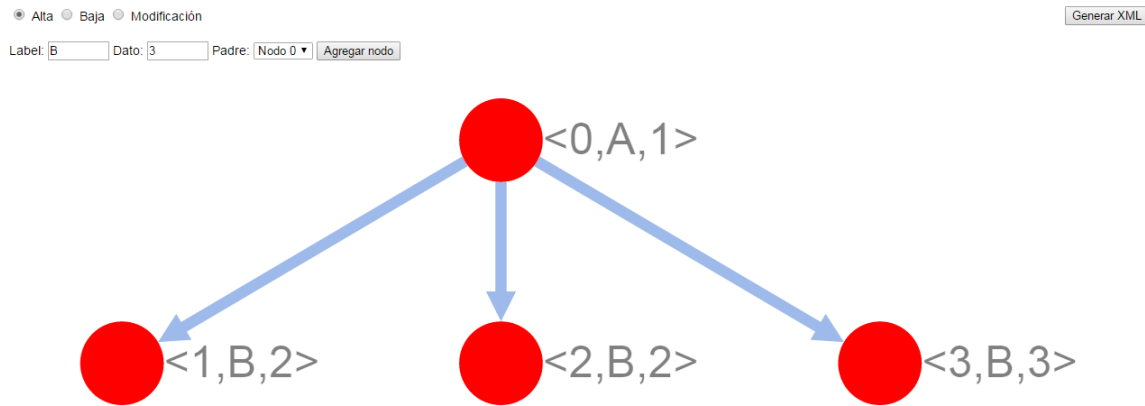


Fig. 4.3: Captura de pantalla de la página que permite crear/modificar un árbol y exportar el XML

4.1.3. Test documento XML grande

Para los casos en los cuales se trabaja con documentos XML grandes, se dificulta poder graficarlos, y por lo tanto no existen beneficios en el uso de las páginas mencionadas en las sub-secciones anteriores. La página *testDocGrande.html* (Figura 4.4) permite cargar un documento XML especificando si se quiere usar una ℓ -bisimulación o una *comp*-bisimulación, y luego ejecutar consultas contra dicho documento.

Por un lado, para ambas aproximaciones se permite ingresar una query (según las reglas de la gramática) y evaluarla. Se muestra como resultado la cantidad de nodos que satisfacen la expresión, el tiempo aproximado (en segundos) de evaluación, el tiempo aproximado (en segundos) de mining y el *hit rate*. Esta tasa de éxito se calcula como el cociente entre la cantidad de veces que se pudo utilizar la aproximación Z para responder una (sub)expresión de nodo sobre la cantidad total de (sub)expresiones de nodo evaluadas. La tasa de éxito es grande cuando se evalúa una fórmula que pertenece al fragmento del lenguaje (por ejemplo, se evalúa una fórmula de profundidad 1 estando en una 1-bisimulación) y chica o nula en caso contrario.

Por otro lado, para cada aproximación se presentan opciones de ejecuciones masivas y aleatorias. Para la ℓ -bisimulación se permite ejecutar cierta cantidad de fórmulas indicada por el usuario, eligiendo el grado de profundidad (entre 0 y la altura del árbol que representa al documento) y si se toma ese grado como exacto o máximo. La aplicación genera aleatoriamente las fórmulas según los parámetros y las evalúa contra la aproximación guardada en memoria. Se muestra como resultado la profundidad actual de la aproximación, la cantidad de clases de la relación Z actual, la cantidad de clases de la relación \tilde{Z} , el tiempo total de evaluación, el tiempo total de mining, la cantidad de actualizaciones que se produjeron y en caso de haber al menos una, el tiempo total de todas ellas. Adicionalmente se muestran todas las queries generadas y evaluadas.

Para la *comp*-bisimulación se permite ejecutar cierta cantidad de fórmulas indicada por el usuario, eligiendo el grado de profundidad máxima y opcionalmente, si se quiere usar fórmulas de tipo test (es decir, $\varphi = \langle \alpha \odot \beta \rangle$) y que pares de labels se quiere comparar (ofreciendo todos los pares posibles para que el usuario seleccione de forma múltiple). Se muestra como resultado lo mismo que en el caso anterior (incluyendo todas las queries generadas y evaluadas), pero en vez de la profundidad actual de la aproximación, se presentan las tuplas que conforman el conjunto *comp* actual.

L-Bisimulación Comp-Bisimulación

<pre></epsilon></pre>	<input type="button" value="Evaluar"/>	Cardinal resultado: 100 Tiempo evaluación: 0 Tiempo mining: 0 Hit Rate: 0.7 Clases Z monio: 43
<pre>nexpr : snexpr 'not' nexpr nexpr 'and' nexpr nexpr 'or' nexpr '(' nexpr ')'</pre>	<pre>#SimpleExpr #NotExpr #AndExpr #OrExpr #ParExpr</pre>	
<pre>snexpr : [a-zA-Z0-9\-__]+; '<' pexpr '>' '<' pexpr 'eq' pexpr '>' '<' pexpr 'neq' pexpr '>'</pre>	<pre>#Label #EPathExpr #IPathExpr #DPathExpr</pre>	
<pre>pexpr : 'epsilon' '/' pexpr '[' nexpr ']' pexpr</pre>	<pre>#epsilon #DownPath #FilterPath</pre>	

queries aleatorios de profundidad exacta

queries aleatorios de profundidad máxima

Profundidad actual: 1
 Clases Z actual: 59
 Cantidad actualizaciones: 0
 Tiempo evaluación: 0.01
 Tiempo mining: 0
 Clases Z monio: 43

- 1) $\neg B$
- 2) $\langle \epsilon \rangle \vee \langle \downarrow [\neg \langle \downarrow [\neg C] \epsilon \rangle] \epsilon \rangle$
- 3) $\neg \langle \downarrow [\langle \downarrow [\neg C] \epsilon \rangle] \epsilon \rangle$
- 4) $\neg \langle \downarrow [\neg B] \epsilon \rangle$
- 5) $B \vee B$
- 6) B
- 7) $\neg \langle \downarrow \epsilon \rangle$
- 8) $A \vee B$
- 9) $\neg \langle \downarrow [A \vee A] \epsilon \rangle$
- 10) $\langle \downarrow [\langle \downarrow [B] \epsilon \rangle] \epsilon \rangle \vee \langle \downarrow [\langle \downarrow [C \vee A] \epsilon \rangle \wedge \langle \downarrow \epsilon \rangle] \epsilon \rangle$

Fig. 4.4: Captura de pantalla de la página de testing sobre documentos grandes (la precisión decimal de los tiempos es de 2 dígitos, por eso algunos tiempos aparecen con valor 0)

4.1.4. Generador aleatorio

La página *generadorAleatorio.html* (Figura 4.5) permite generar aleatoriamente (especificando ciertos parámetros) documentos XML y queries. Estas funcionalidades fueron usadas durante la etapa de experimentación, y decidimos incluirlas como funcionalidad porque consideramos que pueden ser útiles para el usuario.

Para generar un XML es necesario indicar los labels y datos (lo que irá como valor del atributo *value*) del mismo, la cantidad mínima y máxima de hijos que tendrán los nodos internos (tomándose para cada uno de ellos un valor aleatorio en el rango) y la altura máxima del árbol. Cuanto más grandes sean estos valores, mayor tamaño tendrá el XML generado. La raíz del documento es el primero de los labels que se especifican. Como resultado se imprime un XML con las características apropiadas, y con el formato válido para utilizar en las otras páginas que conforman la aplicación.

Para generar las queries es necesario indicar los labels a tener en cuenta, la profundidad (exacta) de las mismas y la cantidad que se quiere generar. Opcionalmente, si se desea generar sólo fórmulas de tipo test (es decir, $\varphi = \langle \alpha \odot \beta \rangle$) se puede hacer, indicando los pares de labels se quiere comparar (esto es útil para la *comp*-bisimulación). En este caso lo que se imprime son las queries generadas, tanto en la notación de la gramática (para poder copiarlas y usarlas en alguna de las otras páginas) como en la notación usual.

Generar XML

*Ingrese los labels separados por coma

*Ingrese los datos separados por coma

*Ingrese el rango de hijos por nodo interno -

*Ingrese la altura máxima

```
<A value="1">
  <B value="1">
    <C value="1">
      <A value="1">
        <C value="2" />
        <A value="2">
          <A value="1" />
          <B value="1" />
        </A>
      </A>
    </C>
    <B value="1">
      <A value="2" />
    </B>
  </A>
</A>
```

Generar Queries

*Ingrese los labels separados por coma

*Ingrese la profundidad

*Ingrese la cantidad de queries que quiere generar

Ingrese las tuplas de comp en formato 'A|B,B|B'

```
not <//[B]epsilon>
-<↓[B]ε>

not <//[A]epsilon>
-<↓[A]ε>

<//[B or B]epsilon>
<↓[B ∨ B]ε>

C and <//[B or B]epsilon>
C ∧ <↓[B ∨ B]ε>

<[B]epsilon neq <//[C or B]epsilon>
<[B]ε ≠ ↓[C ∨ B]ε>
```

Fig. 4.5: Captura de pantalla de la página que permite generar XMLs y queries aleatorias

4.2. Mejoras de performance

4.2.1. Reducción de tiempos

Durante las primeras pruebas con documentos no tan chicos (entre 2 mil y 6 mil nodos), observamos que los tiempos asociados a los diferentes procesos (sobre todo al de actualización) eran muy grandes. Analizando el código en busca del origen de estos problemas, nos dimos cuenta que para este tipo de documentos, la estructura de diccionario que teníamos para almacenar las fórmulas que distinguen cada par de clases era inmensa. Por ejemplo, un documento XML con aproximadamente 6 mil nodos y 1250 clases en Z_1 producía un diccionario de más de 780 mil entradas. La operatoria (ya sea la iteración o la inserción de nuevos elementos) sobre una estructura de tal magnitud incrementaba los tiempos de manera considerable.

Como primera medida, modificamos la clave del diccionario para usar los ids de las clases de equivalencia, en vez de las entidades enteras. Como segunda y principal medida, ya que este diccionario era un adicional planteado para la página *graficador.html* y no era realmente necesario, decidimos dejar de usarlo en este modo.

Una vez realizadas dichas modificaciones, volvimos a ejecutar algunas pruebas, confirmando (a simple vista) que los tiempos habían mejorado y que la aplicación no fallaba. Con el objetivo de conseguir mayores certezas, diseñamos luego un pequeño experimento. Generamos 50 documentos al azar, con un tamaño mediano, y nos quedamos con los 25 más grandes. Por otro lado, tomamos 25 fórmulas distintas de profundidad 1 que nos aseguraran la actualización. Por cada uno de los documentos resultantes, evaluamos todas las queries y medimos los tiempos de mining y de actualización, primero usando el diccionario y luego sin usarlo. La Figura 4.6 muestra los resultados obtenidos.

Se puede observar en dicho gráfico cómo efectivamente crecen los tiempos de ambos procesos (actualización y mining) con el uso del diccionario, y cómo se acentúa la diferencia a medida que crece la cantidad de clases.

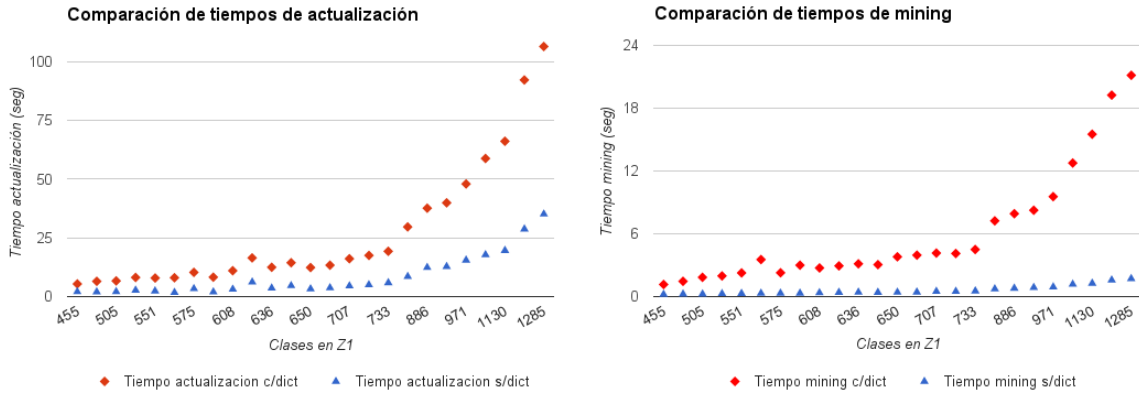


Fig. 4.6: Comparación de tiempos de actualización y mining con y sin el diccionario

Adicionalmente al cambio anterior, realizamos las siguientes modificaciones, que permitieron reducir de manera mucho más notoria los tiempos de los distintos procesos. En particular, el proceso de actualización para un documento XML de 22851 nodos pasó de demorar aproximadamente 20 minutos a tan sólo 1 segundo.

- La entidad *Aprendizaje* modela el aprendizaje sobre una expresión y tiene dos propiedades: *idNodosValidos* e *idNodosInvalidos*. En las mismas se almacenan los ids de los nodos que respectivamente satisfacen y no satisfacen la expresión. Dichas propiedades se utilizan para verificar que la expresión haya sido evaluada en todos los nodos del árbol y además en el refinamiento por aprendizaje.

Inicialmente, el tipo de ambas propiedades era *List<int>*. Sin embargo, dado que las operaciones que se realizaban sobre la estructura (*Add*, *Count*, *Contains*) no incluían la iteración ni importaba el orden de los ids en ella, la elección no era buena. Cambiamos el tipo por *HashSet<int>* y pudimos evidenciar una clara mejoría.

- Como se mencionó anteriormente, las verificaciones Zig-Zag juegan un rol fundamental en el proceso de actualización. Dada la cantidad de verificaciones realizadas, la performance de cada una de ellas impacta en gran medida en el tiempo total.

Parte de la verificación consiste en validar si los caminos candidatos cumplen con las condiciones pedidas. Por ejemplo, para la ℓ -bisimulación entre dos puntos x y x' , si se tiene un descendiente v a distancia n de x , se quiere encontrar un descendiente v' a distancia n de x' tal que $(\xrightarrow{i}v) Z_{\ell-n+i} (\xrightarrow{i}v')$ para todo $0 \leq i < n$ (ver punto 2 de Zig en la sub-sección 2.1.3). En este punto, es necesario verificar si los nodos $(\xrightarrow{i}v)$ y $(\xrightarrow{i}v')$ pertenecen a la misma clase de equivalencia en $Z_{\ell-n+i}$.

Para contestar esto, en un primer momento se iteraba sobre el listado de nodos de cada clase de equivalencia de la relación. La iteración se hacía primero para $(\xrightarrow{i}v)$ y luego para $(\xrightarrow{i}v')$. Viendo que para documentos grandes esto era muy imperformante, decidimos dejar de lado esa iteración monstruosa y en cambio, utilizar información que ya tuviera el nodo. Para esto, en la entidad *Nodo* se agregó la propiedad *bisimIds* de tipo *List<int>*, de forma tal que *bisimIds[i]* tuviera el id de la clase de equivalencia de Z_i a la que pertenece el nodo. Dicha lista se llena en el cálculo de la bisimulación inicial (Z_0) y en cada ejecución del proceso de actualización.

- Durante el proceso de mining, calculamos $\widetilde{Z}_{\ell'}$ para determinar el porcentaje de refinamiento y potencialmente también para usar como base para la actualización. Cada vez que se evalúa una fórmula (a partir de un número determinado), a continuación se ejecuta el proceso de mining y por lo tanto, el cálculo de $\widetilde{Z}_{\ell'}$. Esta repetición constante es al mismo tiempo poco performante e inútil, ya que entre una evaluación y otra se incorporan a lo sumo un par de aprendizajes.

Modificamos el código para guardar temporalmente y de forma separada estos pocos nuevos aprendizajes, y los usamos para refinar la relación $\widetilde{Z}_{\ell'}$ calculada hasta el momento. Este proceso incremental se condice con la Observación 8. Vale aclarar que al utilizar un único \widetilde{Z} , si el valor de ℓ' cambia, entonces debemos volver a calcular de cero el nuevo $\widetilde{Z}_{\ell'}$.

4.2.2. Reducción del uso de memoria

Una vez resueltos los mayores inconvenientes en lo que respecta a tiempos, pudimos empezar a trabajar con documentos aún mas grandes. Experimentamos con XMLs de 200 mil nodos, cuando antes no superábamos los 20 mil nodos. En estas nuevas pruebas, nos encontramos varias veces con fallas del sistema producto de la falta de memoria del proceso para continuar la ejecución (excepciones de tipo *OutOfMemoryException*). Para resolver estos problemas, tuvimos que realizar las siguientes mejoras:

- Ya mencionamos que la entidad Aprendizaje tiene las propiedades *idNodosValidos* e *idNodosInvalidos*, que se utilizan primero para verificar que la expresión haya sido evaluada en todos los nodos del árbol y luego en el refinamiento por aprendizaje. Más allá de estar almacenando sólo ids, para documentos grandes (por ejemplo 200 mil nodos), cada aprendizaje que se guarda involucra un gasto considerable de memoria. Revisando esto, nos dimos cuenta que en el refinamiento por aprendizaje no se utilizaban las dos propiedades, sino sólo una. Más aún, cualquiera de ellas podría usarse. Decidimos entonces usar *idNodosValidos* (suponiendo que para la mayoría de las fórmulas pocos nodos las satisfacen, por lo tanto este conjunto es más pequeño) y vaciar de la memoria el conjunto *idNodosInvalidos* una vez verificado que la expresión haya sido evaluada en todos los nodos del árbol.
- Los aprendizajes sobre expresiones de profundidad ℓ' se utilizan para refinar Z_{ℓ} en $\widetilde{Z}_{\ell'}$ y luego utilizar esta en el proceso de actualización para llegar a $Z_{\ell'}$ (con $\ell' > \ell$). Una vez realizado este proceso, dichos aprendizajes no tienen más utilidad, y conservarlos implica desperdiciar memoria. Por lo tanto, borramos dichos aprendizajes luego de la actualización.
- En la sub-sección 2.1.3 se define la ℓ -bisimilitud a partir de la existencia de una familia de relaciones $(Z_j)_{j \leq \ell}$ que cumplen ciertas condiciones. Para cumplir con esta definición, en principio almacenábamos en memoria a toda la familia de relaciones. En el cálculo de la bisimulación inicial (Z_0) y cada vez que se realizaba una actualización, se agregaba la correspondiente Z_i a la propiedad *familiaRelaciones* de tipo *Dictionary<int, Relacion>*. Cada instancia de *Relacion* tiene a su vez una propiedad de tipo *Dictionary<ClaseEquivalencia, List<Nodo>>*. Es decir, que si estamos en una ℓ -bisimulación, tenemos ℓ veces en memoria todos los nodos del documento.

Al agregar en la entidad *Nodo* la propiedad *bisimIds*, el uso práctico de almacenarse la familia de relaciones desapareció. Por lo tanto, decidimos borrar la propiedad *familiaRelaciones* para liberar una parte importante de espacio en memoria (en realidad, la propiedad se conservó pero solamente para el modo de documento chico, donde se utiliza para generar las expresiones que distinguen nodos).

4.3. Consideraciones varias

- El beneficio algorítmico que nos otorga la bisimulación es el de saber que todas las fórmulas que son válidas en un nodo, lo son también en todos los otros nodos bisimilares a ese. Utilizamos la relación Z para reducir los tiempos del proceso de evaluación. En nuestro código, cada expresión de nodo tiene una propiedad *cache* de tipo *Dictionary(int, bool)*. La entrada $\langle 1, True \rangle$ indica que esta expresión se satisface en todos los nodos de la clase de equivalencia con id 1.

Consideramos agregar y usar la propiedad *cache* también en las expresiones de camino. Sin embargo, en este caso la cuestión no era tan sencilla: el resultado de una expresión de camino en un nodo no es un valor booleano (como lo es en las expresiones de nodo), sino un conjunto de nodos. Mientras el valor booleano es el mismo para todos los nodos de una clase de equivalencia, el conjunto de nodos no lo es. Lo único que sí es igual es el estado de vaciedad del conjunto.

Evaluamos guardar si el conjunto era vacío o no, y sólo en caso de serlo, utilizar esta información para evitar la evaluación. Experimentando con esta nueva condición, verificamos que no había una mejoría en los tiempos y al mismo tiempo estábamos utilizando más memoria, por lo que decidimos descartar esta opción.

- Al evaluar la fórmula φ , si la misma no pertenece al lenguaje actual de la aproximación, se la almacena para utilizarla luego durante el proceso de refinamiento por aprendizajes. Sin embargo, si únicamente analizamos φ en su totalidad, podemos estar perdiendo información útil.

Por ejemplo si tenemos $\varphi = \langle \downarrow [A] \rangle \wedge \langle \downarrow [B] \rangle$ podemos refinar entre nodos que tienen un hijo con label A y otro hijo con label B , y nodos que no lo tienen. Sin embargo, usando el aprendizaje sobre las sub-fórmulas, podríamos refinar el conjunto de nodos que no satisfacen φ entre los que satisfacen $\langle \downarrow [A] \rangle$ y los que no.

Otro ejemplo es si tenemos $\varphi = \langle \downarrow [A] \rangle \wedge \langle \downarrow \downarrow [B] \rangle$. La profundidad de φ es 2, pero sin embargo también podemos incorporar un aprendizaje sobre la expresión $\langle \downarrow [A] \rangle$ que es de profundidad 1, y utilizar este para la actualización a Z_1 .

El cuidado que hay que tener, es que ciertas sub-fórmulas no se evalúan en todos los nodos. En los ejemplos anteriores, $\langle \downarrow [B] \rangle$ y $\langle \downarrow \downarrow [B] \rangle$ no se evalúan en los nodos que no satisfacen $\langle \downarrow [A] \rangle$. Por lo tanto, al momento de evaluar una fórmula, en primer momento guardamos el aprendizaje tanto de la fórmula entera como de todas sus sub-fórmulas (siempre hablando de las que no pertenecen al lenguaje actual de la aproximación). Luego de finalizar la evaluación, descartamos los aprendizajes parciales (los que no se ejecutaron en todos los nodos, aquí se utilizan las propiedades *idNodosValidos* y *idNodosInvalidos*) y guardamos únicamente los aprendizajes totales que sean nuevos (vacianado de la memoria el conjunto *idNodosInvalidos*).

- Realizamos un pre-procesamiento sobre los nodos del árbol para que cumplan con el formato requerido de tener un único atributo *value* y no tener texto dentro (sólo opcionalmente otros nodos). Por cada atributo distinto a *value*, lo agregamos a este último como *nombreAtributo:valorAtributo*; y luego lo eliminamos. Agregamos finalmente el texto, si es que tuviera alguno. Si inicialmente no existe el atributo *value*, se lo crea con el valor correspondiente.

Por ejemplo, transformamos el nodo $\langle A \text{ value}='v1' \text{ otroAtr}='v2' \rangle \text{texto} \langle /A \rangle$ en $\langle A \text{ value}='v1;\text{otroAtr}:v2;\text{texto}' / \rangle$.

4.4. *comp*-Bisimulación

En la sub-sección 2.2.5 se mencionaron las siguientes cuestiones de implementación que debían ser resueltas:

1. Cómo almacenar *comp*
2. Cómo modificar el algoritmo de verificación de Zig-Zag para chequear *comp*
3. Qué hacer con las fórmulas φ tal que $\varphi \notin \text{comp-XPath}=(\downarrow)$

Con respecto al primer punto, guardamos *comp* utilizando una lista de tuplas de string. Este tipo de estructura es útil porque permite las operaciones requeridas en tiempos razonables dada la cantidad de datos que almacena. Con el objetivo de ahorrar espacio en memoria decidimos que *comp* sea implícitamente simétrica. Por lo tanto, si queremos comparar entre labels *a* y *b*, agregamos una única tupla a la lista (por ejemplo, agregamos $\langle a, b \rangle$ y no $\langle b, a \rangle$) y asumimos que la otra también lo esta. Modificamos los algoritmos que realizan búsquedas en *comp* para tener en cuenta esta condición.

En lo tocante al segundo punto, en el algoritmo que verifica si los nodos x_1 y x_2 satisfacen las condiciones de Zig y Zag, primero se itera sobre los nodos v descendientes de x_1 mientras se valida que exista para cada uno de ellos, un nodo v' descendiente de x_2 tal que $(\xrightarrow{i}v)Z(\xrightarrow{i}v')$ para todo $0 \leq i < n = \text{distancia}(x_1, v)$. Las modificaciones realizadas se dan a continuación: dado v , luego de validarse que exista v' , se continúa verificando dicho descendiente sólo si $\text{label}(v)$ aparece en alguna tupla de *comp*. En caso afirmativo, se itera sobre los nodos w descendientes de x_1 , pero sólo los que cumplen que $\langle v, w \rangle \in \text{comp}$ (recordando lo mencionado de la simetría).

Finalmente, en lo referido al punto 3, hicimos lo que habíamos mencionado como la mejor opción: aprovechar Z para las subqueries que sí pertenezcan a $\text{comp-XPath}=(\downarrow)$ y hacer la verificación manual en aquellas que no pertenezcan. También habíamos comentado que era posible realizar un procesamiento que transforme fórmulas que no pertenezcan al fragmento del lenguaje (por su forma sintáctica) por otras equivalentes que sí lo hicieran. La implementación de este procesamiento con su correspondiente experimentación, junto a otras optimizaciones, queda como posible línea de investigación futura.

4.5. Algoritmos

4.5.1. Aproximaciones de bisimulación

```

1 public abstract class AproximacionBisimulacion{
2
3     public DataTree dataTree;
4     protected Relacion z;
5     protected Relacion zMonio;
6     protected BaseConocimiento bc;
7
8     ...
9
10    public void AgregarAprendizaje(Expression e, Nodo n, bool nodoValido) {
11        this.bc.AgregarAprendizaje(e, n.Id, nodoValido);
12    }
13
14    public ExpresionDeNodo VerificarZigZag(Nodo nodo1, Nodo nodo2) {
15        ExpresionDeNodo expr = this.VerificarUnoZigZag(nodo1, nodo2);
16        if (expr != null) return expr;
17
18        expr = this.VerificarUnoZigZag(nodo2, nodo1);
19        if (expr != null) return new ExpresionNot(expr);
20
21        return null;
22    }
23
24    protected Relacion ObtenerRefinamientoPorAprendizaje(Func<Expresion, bool>
25        func) {
26        var aprendizajes = this.bc.ObtenerAprendizajes(func);
27        var zMonioAprendizaje = (Relacion) this.z.Clone();
28        foreach (var aprendizaje in aprendizajes)
29        {
30            zMonioAprendizaje.RefinarPorAprendizaje(aprendizaje);
31        }
32        return zMonioAprendizaje;
33    }
34
35    protected abstract void CalcularBisimulacionInicial(bool modoDocChico);
36
37    protected abstract bool ActualizarBisimulacion();
38
39    protected abstract void RealizarActualizacion();
40
41    protected abstract ExpresionDeNodo VerificarUnoZigZag(Nodo n1, Nodo n2);
42
43    internal abstract bool PerteneceAlLenguaje(Expression f);
44
45    internal abstract bool ValidarCaminoCandidato(CaminoNodos caminoHaciaV,
46        CaminoNodos caminoHaciaVPrima, Dictionary<int, List<ExpresionDeNodo>>
47        listExprNodo);
48
49    internal abstract int ObtenerIdBisimulacion(ExpressionDeNodo e, Nodo n);
50
51    public abstract object ObtenerCaracteristica();
52 }

```

Algoritmo 4.1: Fragmento de la clase abstracta de aproximación

En el Algoritmo 4.1 mostramos un fragmento de la clase abstracta de aproximación, específicamente los métodos que son necesarios implementar por cada tipo de aproximación y otros genéricos que ya están implementados. Entre estos últimos se encuentra el método que agrega un aprendizaje (delegando dicha responsabilidad a la clase *BaseConocimiento*), el método que verifica Zig y Zag (pero que para cada una de ellas sí llama a un método específico del tipo de aproximación) y el método que obtiene \tilde{Z} (comenzando por Z , refina la relación iterando por los aprendizajes obtenidos).

Si se quisiera crear un nuevo tipo de aproximación, la misma debería extender de *AproximacionBisimulacion* e implementar los siguientes métodos:

- *CalcularBisimulacionInicial*: Calcula la relación básica e inicial.
- *ActualizarBisimulacion*: Proceso de mining, decide si realizar la actualización y en caso afirmativo, determina la característica de la nueva aproximación y genera \tilde{Z} .
- *RealizarActualizacion*: Proceso de actualización.
- *VerificarUnoZigZag*: Verifica la condición de Zig entre n_1 y n_2 .
- *PerteneceAlLenguaje*: Indica si la expresión pertenece al fragmento del lenguaje.
- *ValidarCaminoCandidato*: Valida la condición 2 de Zig entre v y v' .
- *ObtenerIdBisimulacion*: Obtiene el id de bisimulación de un nodo para evaluar una determinada expresión.
- *ObtenerCaracteristica*: Obtiene la característica que define a la instancia del tipo de aproximación (el nivel de profundidad para ℓ o las tuplas de labels para *comp*).

De esta forma, es simple agregar un nuevo tipo de aproximación a la herramienta desarrollada. Sin embargo, por malas decisiones de diseño, las clases de tipo *Expresion* quedaron ligadas a los tipos de aproximación estudiados. Esta clase tiene un atributo *profundidad* para usar en la ℓ -bisimulación y otro atributo *subexpresionesTest* para usar en la *comp*-bisimulación. Agregar un nuevo tipo de aproximación probablemente implique cambiar dicha clase y todas sus descendientes, para tener en cuenta algún otro tipo de característica de la fórmula. Forma parte del trabajo futuro modificar esta decisión de diseño y la implementación subsecuente.

4.5.2. Cálculo inicial

```

1 public class l-Aproximacion: AproximacionBisimulacion{
2
3     protected override void CalcularBisimulacionInicial(bool modoDocChico) {
4         this.profundidadActual = 0;
5         this.z = new Relacion(modoDocChico);
6         foreach (Nodo nodo in this.dataTree.Nodos)
7         {
8             this.z.AgregarNodoAClasePorLabel(nodo);
9         }
10
11         this.AgregarZeta();
12     }
13
14     ...
15 }

```

Algoritmo 4.2: Cálculo de aproximación inicial (Z_0) para ℓ -bisimulación

```

1 public class comp-Aproximacion: AproximacionBisimulacion{
2
3     protected override void CalcularBisimulacionInicial(bool modoDocChico) {
4         this.comp = new List<Tupla<string, string>>();
5         this.z = new Relacion(modoDocChico);
6         foreach (Nodo nodo in this.dataTree.Nodos)
7         {
8             this.z.AgregarNodoAClasePorLabelYProfundidad(nodo);
9         }
10
11         this.z.RefinarPorZigZag(this, (Relacion) this.z.Clone());
12     }
13
14     ...
15 }

```

Algoritmo 4.3: Cálculo de aproximación inicial (Z_0) para *comp*-bisimulación

Los Algoritmos 4.2 y 4.3 presentan el cálculo de la aproximación inicial para la ℓ -bisimulación y la *comp*-bisimulación respectivamente. En ambos primero se inicializa la característica en su valor inicial (nivel de profundidad $\ell = 0$, lista de tuplas de labels $comp = \emptyset$) y luego se construye una nueva relación Z .

Para la ℓ -bisimulación, la relación inicial Z_0 se construye simplemente creando una clase de equivalencia por cada label distinto en el documento XML y poniendo a cada nodo en la clase correspondiente a su label.

Para la *comp*-bisimulación, el proceso es más costoso, ya que es necesario realizar verificaciones Zig-Zag de profundidad acotada únicamente por la altura del árbol. Agregamos una optimización que consiste en particionar por label y profundidad, en vez de crear una única clase de equivalencia inicial con todos los nodos. Esto quizás no corresponde con la definición teórica de este tipo de aproximación, pero es válido (todos los nodos distinguidos por profundidad o label son distinguidos por cualquier *comp*-bisimulación) y en la práctica es útil. Refinamos dichas particiones haciendo las verificaciones Zig-Zag correspondientes (teniendo en cuenta que $comp = \emptyset$ y lo explicado en el punto 2 de la sección 4.4).

4.5.3. Evaluación de una expresión en un nodo

```

1 public abstract class ExpresionDeNodo{
2
3     private Dictionary<int , bool> cache = new Dictionary<int , bool>();
4
5     public bool EvaluarConCache(Nodo n, AproximacionBisimulacion ab) {
6         ab.ContabilizarEvaluacion();
7         if (ab.PerteneceAlLenguaje(this))
8             {
9             int idBisim = ab.ObtenerIdBisimulacion(this, n);
10            if (this.cache.ContainsKey(idBisim))
11                {
12                ab.IncrementarHits();
13                return this.cache[idBisim];
14            }
15
16            bool res = this.Evaluar(n, ab);
17            this.cache.Add(idBisim, res);
18            return res;
19        }
20        else
21            {
22            bool res = this.Evaluar(n, ab);
23            ab.AgregarAprendizaje(this, n, res);
24            return res;
25        }
26    }
27
28    public abstract bool Evaluar(Nodo n, AproximacionBisimulacion ab);
29 }

```

Algoritmo 4.4: Clase abstracta de expresiones de nodo con algoritmo de evaluación en un nodo

Decidimos que la lógica (y la responsabilidad) de evaluación esté en las clases expresiones y no en el nodo. Es por esto que tenemos varias clases de expresiones (una para cada tipo: expresión not, test igualdad, etc) y cada una de ellas tiene un método *Evaluar(Nodo n)*. En particular, todas las expresiones de nodo extienden de la clase abstracta *ExpresionDeNodo*. En esta clase, aparte de tener declarado el método *Evaluar* como abstracto (para que cada sub-clase lo implemente debidamente), está la implementación del método *EvaluarConCache*. Este método es el que saca provecho de la bisimulación reduciendo las evaluaciones efectivas (llamadas al método *Evaluar*). Debido a esto es que ambos métodos (*Evaluar* y *EvaluarConCache*) reciben, aparte del nodo, la aproximación de bisimulación que se está utilizando.

En el método *EvaluarConCache* se utiliza inicialmente la aproximación de bisimulación (llamemosla *ab*) para verificar si la expresión de nodo pertenece al fragmento del lenguaje. En caso negativo, se evalúa efectivamente la expresión y el resultado se guarda como aprendizaje antes de retornarlo. En caso afirmativo, se vuelve a utilizar *ab* para obtener el id de bisimulación del nodo (el id de la clase de equivalencia a la cual pertenece el nodo) y con dicho id consultar si ya se tiene almacenada la respuesta. Si se tiene se retorna, y sino, se realiza la evaluación, se guarda el resultado y luego se retorna. Podemos ver claramente aquí lo dicho anteriormente acerca de cómo es que usamos la relación de bisimulación como caché.

4.5.4. Proceso de mining

```

1 public class l-Aproximacion: AproximacionBisimulacion{
2
3     public override bool ActualizarBisimulacion() {
4         var exprTodas = this.bc.FormulasEvaluadas
5             .Where(e => e.Profundidad > this.profundidadActual)
6             .Select(e => e.Profundidad);
7
8         if (exprTodas.Count() == 0)
9         {
10            return false;
11        }
12
13        int mediana = CalcularMediana(exprTodas);
14        int cantLabels = this.dataTree.ObtenerLabels().Count;
15
16        double medidaMinima = Math.Max(Math.Pow(cantLabels, 2) / mediana,
17            10 / Math.Sqrt(mediana));
18        medidaMinima = Convert.ToInt32(medidaMinima);
19        if (exprTodas.Count() >= medidaMinima)
20        {
21            this.CalcularZMonio(mediana);
22            int cantEvaluadasProf = exprTodas.Count(e => e.Equals(mediana));
23
24            return (this.AlcanzaRefinamiento() ||
25                cantEvaluadasProf >= 3 * medidaMinima);
26        }
27
28        return false;
29    }
30
31    private void CalcularZMonio(int mediana) {
32        Func<Expresion, bool> func = exp => exp.Profundidad.Equals(mediana);
33        if (!mediana.Equals(this.profundidadNueva))
34        {
35            this.zMonio = ObtenerRefinamientoPorAprendizaje(func);
36            this.profundidadNueva = mediana;
37        }
38        else
39        {
40            ActualizarRefinamientoPorUltimosAprendizajes(func);
41        }
42    }
43
44    ...
45 }

```

Algoritmo 4.5: Algoritmo de mining para ℓ -bisimulación

En el mining para la ℓ -bisimulación, primero obtenemos todas las fórmulas evaluadas de profundidad mayor a la actual de la aproximación. De haber alguna, obtenemos la mediana de las profundidades de dichas fórmulas. Luego seteamos la medida mínima $\kappa = \max\{\frac{n^2}{\ell'}, \frac{10}{\sqrt{\ell'}}\}$ y verificamos que se hayan evaluado como mínimo esa cantidad de expresiones. En caso afirmativo, calculamos $\tilde{Z}_{\ell'}$ (intentando reutilizar los cálculos previos) y decidimos actualizar a $Z_{\ell'}$ si se alcanza el refinamiento deseado o si la cantidad de ejecuciones de expresiones de profundidad ℓ' alcanzan el valor $\tau = 3\kappa$.

```

1 public class comp-Aproximacion: AproximacionBisimulacion{
2
3     public override bool ActualizarBisimulacion() {
4         this.nuevasTuplas = new List<Tupla<string, string>>();
5         var cantLabels = this.dataTree.ObtenerLabels().Count;
6         var medidaMinima = 5 * cantLabels;
7         var listExprTest = ObtenerExpresionesTest();
8
9         if (listExprTest.Count() > medidaMinima)
10            {
11                var cantMaxEjecuciones = ObtenerCantMaxEjecuciones(listExprTest);
12                var fraccionMasEjecutada = cantMaxEjecuciones / 4;
13                medidaMinima = Math.Max(medidaMinima, fraccionMasEjecutada);
14
15                ObtenerNuevasTuplas(medidaMinima, listExprTest);
16
17                if (this.nuevasTuplas.Count > 0) {
18                    Func<Expresion, bool> funcTotal =
19                        expr => PerteneceAlLenguaje(expr, this.nuevasTuplas);
20
21                    this.zMonio = ObtenerRefinamientoPorAprendizaje(funcTotal);
22                }
23            }
24
25            return this.nuevasTuplas.Count > 0;
26        }
27
28        private void ObtenerNuevasTuplas(double medidaMinima,
29            IEnumerable<ExpresionTestCaminos> listExprTest) {
30
31            var tuplasCandidatas = (from se in listExprTest
32                where !TuplaLabelEnComp(se.ObtenerUltimosLabels())
33                group se by se.ObtenerUltimosLabels()
34                into tl
35                where tl.Count() > medidaMinima
36                select new {Tupla = tl.Key, Cantidad = tl.Count()});
37
38            this.nuevasTuplas = (from tc in tuplasCandidatas
39                where tc.Cantidad >= 3 * medidaMinima
40                select tc.Tupla).ToList();
41        }
42
43        ...
44    }

```

Algoritmo 4.6: Algoritmo de mining para *comp*-bisimulación

En el mining para la *comp*-bisimulación, primero obtenemos todas las expresiones de tipo test y verificamos que haya al menos $5n$. En caso afirmativo, obtenemos la cantidad máxima de ejecuciones (C_{max}) entre las tuplas del *comp* actual y con ella seteamos la medida mínima $\kappa = \max\{5n, \frac{C_{max}}{4}\}$. Luego, obtenemos las tuplas a agregar en *comp*. Para esto, entre todas las expresiones de tipo test, primero marcamos como candidatas aquellas tuplas que no estén en *comp* y cuya cantidad de ejecuciones supere κ . Luego, entre las tuplas candidatas, conservamos aquellas cuya cantidad de ejecuciones sea mayor o igual al valor $\tau = 3\kappa$. Si hay al menos una tupla que cumpla estas condiciones, decidimos actualizar a Z_{comp} calculando previamente \tilde{Z}_{comp} .

4.5.5. Proceso de actualización de la aproximación

```

1 public class l-Aproximacion: AproximacionBisimulacion{
2
3   protected override void RealizarActualizacion() {
4     var salteoNiveles = this.profundidadActual + 1 < this.profundidadNueva;
5     while (this.profundidadActual < this.profundidadNueva)
6     {
7       this.profundidadActual++;
8
9       Relacion relMonio = salteoNiveles ? ObtenerRefinamientoPorAprendizaje
10        (e => e.Profundidad.Equals(this.profundidadActual)) : this.zMonio;
11
12      this.z.RefinarPorZigZag(this, relMonio);
13
14      this.AgregarZeta();
15      this.bc.Limpiar(e => e.Profundidad.Equals(this.profundidadActual));
16    }
17  }
18  ...
19 }
20 }

```

Algoritmo 4.7: Actualización de la aproximación Z_ℓ

```

1 public class comp-Aproximacion: AproximacionBisimulacion{
2
3   protected override void RealizarActualizacion() {
4     this.comp.AddRange(this.nuevasTuplas);
5     this.z.RefinarPorZigZag(this, this.zMonio);
6   }
7
8   ...
9 }

```

Algoritmo 4.8: Actualización de la aproximación Z_{comp}

El Algoritmo 4.7 muestra el proceso de actualización para la ℓ -bisimulación. Como discutimos anteriormente, es posible (aunque no probable) que el proceso de mining indique que la nueva profundidad no es simplemente $\ell + 1$, sino un valor estrictamente mayor. También mencionamos que en tal caso, es necesario tener todas las relaciones Z_{ℓ_i} intermedias, ya que las mismas se utilizan para responder las fórmulas de dicha profundidad. El algoritmo contempla esto realizando actualizaciones que aumentan en una unidad la profundidad, hasta llegar al valor deseado. Si efectivamente sucede que $\ell' > \ell + 1$, entonces tenemos que calcular también las relaciones \tilde{Z}_{ℓ_i} intermedias, ya que el proceso de mining sólo calculó $\tilde{Z}_{\ell'}$ (aquí hay mejoras que se pueden hacer en el código, que conservamos como trabajo futuro). Calculamos entonces cada Z_{ℓ_i} utilizando el correspondiente \tilde{Z}_{ℓ_i} .

El Algoritmo 4.8 muestra el proceso de actualización para la $comp$ -bisimulación. En este caso, todo resulta más directo: simplemente se transforma $comp$ en $comp'$ agregando las nuevas tuplas determinadas por el proceso de mining, y se ejecuta el refinamiento por Zig-Zag a partir del $\tilde{Z}_{comp'}$ calculado previamente.

5. EXPERIMENTACIÓN

Realizamos durante el transcurso del trabajo varios experimentos, de diversos tipos y con distintas motivaciones. Algunos se hicieron sobre documentos de tamaño chico, para permitirnos confirmar que nuestros algoritmos (junto a sus respectivas implementaciones) fueran correctos. Otros se efectuaron con el fin de comprobar ciertas hipótesis que surgieron. Ciertas pruebas tuvieron como objetivo encontrar los valores necesarios para satisfacer los criterios que buscábamos. Una última serie, quizás la principal, tiene como propósito probar empíricamente que los conceptos desarrollados tienen sus beneficios, y que en determinados casos, son de gran utilidad.

Presentamos a continuación algunos de dichos experimentos. Es necesario aclarar que en lo referido a tiempos (evaluación, mining, actualización, etc.) lo que importa no es el número absoluto, sino el relativo a otras pruebas u otras medidas. El número absoluto depende de varios puntos (hardware de la computadora, software ejecutándose, etc.) que no son de interés para el alcance de este trabajo. Por otro lado, cuando mencionemos que generamos queries aleatoriamente, nos referimos a que fueron producto de nuestro generador aleatorio. Intentando ser equiprobable en cuanto a tipos de expresiones (pero imponiendo ciertas restricciones) y labels, se forman queries que pueden no tener sentido semántico, pero que sintácticamente son válidas.

5.1. Aprendizajes en ℓ -bisimulación

5.1.1. Primeras pruebas

Se ejecutaron pruebas sobre un XML generado aleatoriamente de 22851 nodos, con 5 labels distintos (A,B,C,D,E), 3 valores de datos distintos (1,2,3), entre 3 y 7 hijos cada nodo interno y con una altura de 10. Inicialmente la 0-bisimulación cuenta con 5 clases (una por cada label).

Se hicieron tres pruebas: la primera teniendo 10 fórmulas de aprendizajes, la segunda teniendo 5 fórmulas de aprendizaje y la tercera teniendo sólo 1 fórmula de aprendizaje. En cada prueba se realizaron 5 ejecuciones, donde primero se evaluaban 10 fórmulas, luego se actualizaba la bisimulación y finalmente se volvían a evaluar las mismas 10 fórmulas. Para cada ejecución se presentan y comparan el tiempo promedio de evaluación de una fórmula antes y después de la actualización. También se presenta el tiempo que se insumió para realizar el proceso de actualización.

Para este experimento, la actualización se realizaba si la cantidad de fórmulas evaluadas era mayor o igual a 10 y si la mediana de las profundidades de dichas fórmulas era mayor al ℓ de la aproximación. En las tres pruebas se ejecutan primero 10 fórmulas de profundidad 1, por lo que luego de evaluarse la última, se realiza el proceso de actualización. Finalmente, se vuelven a ejecutar otras 10 fórmulas de profundidad 1, pero en este caso no se realiza el proceso ya que la mediana es igual al ℓ alcanzado.

Prueba 1: 10 fórmulas de aprendizajes

Se ejecutaron 10 queries distintas, todas con profundidad 1:

- $\langle \downarrow \varepsilon \rangle$
- $\langle \downarrow \varepsilon = \varepsilon \rangle$
- $\langle \downarrow \varepsilon \neq \varepsilon \rangle$
- $\langle \downarrow \varepsilon \neq \downarrow \varepsilon \rangle$
- $\langle \varepsilon = \downarrow[A]\varepsilon \rangle$
- $\langle [B]\downarrow[C]\varepsilon = [B]\downarrow[D]\varepsilon \rangle$
- $\langle [D]\downarrow[E]\varepsilon \neq [D]\downarrow[E]\varepsilon \rangle$
- $\langle \downarrow[C]\varepsilon = \downarrow[D]\varepsilon \rangle$
- $\langle \downarrow[A]\varepsilon \neq \downarrow[A]\varepsilon \rangle$
- $\langle \downarrow[B]\varepsilon = \downarrow[D]\varepsilon \rangle$

Luego del refinamiento por aprendizajes, se obtuvo una relación \tilde{Z} de 173 clases.

Las mediciones de tiempos del promedio de evaluación de una fórmula en las 5 ejecuciones fueron las siguientes:

1. 75,67564 ms vs. 16,78251 ms
2. 75,96192 ms vs. 16,71141 ms
3. 79,02875 ms vs. 16,77047 ms
4. 77,88661 ms vs. 16,60237 ms
5. 76,00863 ms vs. 16,65493 ms

En todos los casos se evidencia una (clara) mejora en los tiempos de la segunda tanda de evaluaciones, aprovechando la información que provee la relación de bisimulación.

Las mediciones de tiempos (en segundos) del proceso de actualización fueron las siguientes:

1. 5,73 seg
2. 5,68 seg
3. 5,69 seg
4. 5,67 seg
5. 5,66 seg

Aproximadamente, considerando todos los casos, el proceso insumió 5,69 segundos.

Prueba 2: 5 fórmulas de aprendizajes

Se ejecutaron 5 queries distintas, 2 veces cada una, todas con profundidad 1:

- $\langle \downarrow \varepsilon \rangle$
- $\langle [B]\downarrow[C]\varepsilon = [B]\downarrow[D]\varepsilon \rangle$
- $\langle [D]\downarrow[E]\varepsilon \neq [D]\downarrow[E]\varepsilon \rangle$
- $\langle \downarrow[C]\varepsilon = \downarrow[D]\varepsilon \rangle$
- $\langle \downarrow[A]\varepsilon \neq \downarrow[A]\varepsilon \rangle$

Luego del refinamiento por aprendizajes, se obtuvo una relación \tilde{Z} de 29 clases.

Las mediciones de tiempos del promedio de evaluación de una fórmula en las 5 ejecuciones fueron las siguientes:

1. 84,28553 ms vs. 14,91081 ms
2. 82,72118 ms vs. 14,32161 ms
3. 80,15402 ms vs. 14,49167 ms
4. 76,60110 ms vs. 14,48953 ms
5. 78,79632 ms vs. 14,54602 ms

Aquí también se evidencia para todos los casos una (clara) mejora en los tiempos de la segunda tanda de evaluaciones, aprovechando la información que provee la relación Z_1 .

Las mediciones de tiempos (en segundos) del proceso de actualización fueron las siguientes:

1. 18,49 seg
2. 17,00 seg
3. 16,82 seg
4. 16,81 seg
5. 16,82 seg

Aproximadamente, considerando los 5 casos, el proceso insumió 17,19 segundos.

Prueba 3: 1 fórmula de aprendizaje

Se ejecutó 10 veces la query $\varphi = \langle \downarrow \varepsilon \rangle$.

Luego del refinamiento por aprendizajes, se obtuvo una relación \tilde{Z} de 10 clases (cada una de las 5 clases iniciales se refino en dos: las que cumplen φ y las que no).

Las mediciones de tiempos del promedio de evaluación de una fórmula en las 5 ejecuciones fueron las siguientes:

1. 45,70339 ms vs. 14,74150 ms
2. 43,61240 ms vs. 14,24700 ms
3. 43,61799 ms vs. 14,20790 ms
4. 47,31093 ms vs. 14,72269 ms
5. 48,90154 ms vs. 14,99925 ms

Nuevamente se evidencia para todos los casos una (clara) mejora en los tiempos de la segunda tanda de evaluaciones, por los mismas razones explicadas anteriormente.

Las mediciones de tiempos (en segundos) del proceso de actualización fueron las siguientes:

1. 47,65 seg
2. 43,60 seg
3. 43,94 seg
4. 44,04 seg
5. 43,26 seg

Aproximadamente, considerando los 5 casos, el proceso insumió 44,50 segundos.

Análisis y conclusiones de las pruebas

El objetivo principal de este experimento era mostrar la utilidad práctica (a nivel tiempos) del aprendizaje en el proceso de actualización. En la 3° prueba, donde se tiene únicamente una fórmula como aprendizaje (que es lo mínimo que se puede tener para actualizar), el tiempo aproximado de la actualización es de 44,5 segundos. En cambio, en la 1° prueba, donde se tienen 10 fórmulas distintas de aprendizaje, el tiempo se reduce en gran medida, siendo aproximadamente 5,7 segundos (casi 8 veces menos). El refinamiento producto de los aprendizajes en esta 1° prueba genera una relación intermedia de 173 clases contra 10 de la 3° prueba, lo que se refleja en los tiempos presentados.

La 2° prueba es un intermedio entre las otras, teniendo 5 fórmulas de aprendizaje, una relación intermedia de 29 clases y un tiempo aproximado de actualización de 17,2 segundos. Comparando contra la 3° prueba, teniendo menos del triple de clases se logra reducir el tiempo de actualización a más de la mitad. Comparando contra la 1° prueba, se tienen casi 6 veces menos clases y se tarda 3 veces más.

Un objetivo secundario tenía que ver con comprobar que efectivamente, el uso de la ℓ -bisimulación mejora los tiempos de evaluación de una fórmula que pertenece al fragmento ℓ -XPath $_{=}$ (\downarrow). Esto se ve claramente en cualquiera de las tres pruebas (pues resulta independiente del aprendizaje). En la segunda tanda, luego de la actualización, ya se tiene una 1-bisimulación, por lo que todas las fórmulas evaluadas (al ser de profundidad 1) pertenecen al lenguaje. Por ende, en la evaluación de las mismas, se puede usar la información de la relación Z_1 para saltarse nodos que ya se conoce si satisfacen o no a la expresión.

5.1.2. Actualización según cantidad de aprendizajes

Decidimos extender el alcance del experimento anterior, testeando con un rango más amplio de cantidad de aprendizajes y contabilizando no sólo el tiempo de actualización, sino también la cantidad real de verificaciones Zig-Zag realizadas durante dicho proceso. Las pruebas se hicieron utilizando el mismo documento XML de 22851 nodos. Se realizaron 20 ejecuciones E_1, \dots, E_{20} ; en la ejecución E_j se tomaban j fórmulas distintas de profundidad 1 de manera aleatoria. Para cada ejecución se midió el tiempo de actualización, la cantidad de clases en \tilde{Z}_1 luego del refinamiento por aprendizaje, la cantidad real de verificaciones Zig-Zag y la cota de verificaciones según la Observación 9. Presentamos a continuación los resultados:

Ejecución	Aprendizajes	Tiempo actualización (seg)	Clases \tilde{Z}_1	Verificaciones Zig-Zag	Verificaciones Zig-Zag teóricas
E_1	1	31,3173067	10	920649	91007166
E_2	2	35,9273064	15	932674	69159324
E_3	3	21,3616417	24	640096	70422494
E_4	4	12,5247263	45	362546	71092176
E_5	5	13,839512	50	375422	67730102
E_6	6	4,5173291	210	116268	67099200
E_7	7	5,3421415	201	152097	67287648
E_8	8	10,6725415	65	287189	67500570
E_9	9	6,2150295	146	170436	67213808
E_{10}	10	7,5561898	85	212068	67335220
E_{11}	11	4,0391133	264	110422	67081066
E_{12}	12	14,8831495	63	414496	67821502
E_{13}	13	8,0679539	161	207267	67307790
E_{14}	14	4,3585295	248	117494	67093456
E_{15}	15	3,1937015	472	92413	67027858
E_{16}	16	3,0708987	400	85339	67012740
E_{17}	17	1,7657053	830	57969	66950754
E_{18}	18	2,1519319	556	65903	66968754
E_{19}	19	1,8850292	756	60365	66954438
E_{20}	20	2,7042081	519	79554	67120956

Tab. 5.1: Varias mediciones según cantidad de aprendizajes

Observando los resultados de la tabla 5.1 podemos apreciar cómo disminuye la cantidad de verificaciones y el tiempo de actualización a medida que aumentamos la cantidad de fórmulas que usamos como aprendizaje. Aunque tengamos más pruebas y estadísticas, esta relación ya la conocíamos y no resulta novedosa. Lo que sí es nuevo y es interesante notar y mencionar es que no sólo importa la cantidad de fórmulas, sino también su «calidad» para refinar (esta calidad es relativa a cada data tree). Por ejemplo, en E_{11} la actualización demora unos 4 segundos, mientras que E_{12} tiene un tiempo de actualización de casi 15 segundos. El aumento en el tiempo se debe al incremento en la cantidad de verificaciones Zig-Zag. Esta situación se repite otras tantas veces y se puede apreciar en la Figura 5.1.

Ejecución	Clases \tilde{Z}_1	Tiempo actualización (seg)	Verificaciones Zig-Zag
E_1	303	4,7620146	130272
E_2	278	4,382163	121859
E_3	269	3,747362	101470
E_4	269	3,9509367	106106
E_5	245	4,7346778	114103
E_6	239	3,3860539	91674
E_7	231	3,7111432	98831
E_8	224	3,9936549	106963
E_9	222	4,2909274	127404
E_{10}	206	7,1859392	186580
E_{11}	203	6,996071	194102
E_{12}	197	4,3282027	113926
E_{13}	191	5,3529834	143580
E_{14}	180	7,0267294	187081
E_{15}	171	6,2007342	157225
E_{16}	168	6,0044215	165376
E_{17}	167	7,9747891	198472
E_{18}	162	5,1132008	138255
E_{19}	160	4,3036409	113605
E_{20}	157	4,5261556	121914
E_{21}	154	6,4075306	171789
E_{22}	153	5,9882813	158908
E_{23}	147	5,1955654	150786
E_{24}	141	15,655273	402985
E_{25}	140	5,9256782	168869
E_{26}	133	16,4809998	450082
E_{27}	131	7,6332086	188610
E_{28}	130	5,8653094	150126
E_{29}	125	6,6134636	184869
E_{30}	120	6,6315301	177004
E_{31}	116	8,2431199	220395
E_{32}	115	9,2447029	247681
E_{33}	113	6,8374485	173533
E_{34}	112	6,9496488	182508
E_{35}	108	8,2492865	231244
E_{36}	104	7,7014819	203379
E_{37}	100	9,5382363	235119
E_{38}	96	8,3962841	218878
E_{39}	93	7,3633554	191583
E_{40}	89	8,2989699	223536
E_{41}	86	10,0573365	281787
E_{42}	86	17,0918504	458992
E_{43}	81	9,1036508	238277
E_{44}	81	10,8178394	298087
E_{45}	76	11,0178214	312039
E_{46}	74	8,7524849	248835
E_{47}	73	9,6956434	256209
E_{48}	61	16,4452344	444630
E_{49}	55	21,512901	585516
E_{50}	33	19,9585096	547789

Tab. 5.2: Tiempo actualización y verificaciones Zig-Zag según cantidad de clases en \tilde{Z}_1

En la Figura 5.3 podemos apreciar nuevamente la linealidad entre verificaciones Zig-Zag y tiempo de actualización. El tiempo que demora la verificación del nodo x está vinculado a cuántos pares de descendientes v, w se visitan. Las pequeñas diferencias entre los tiempos de cada verificación son las que posiblemente producen las anomalías que se observan en el gráfico.

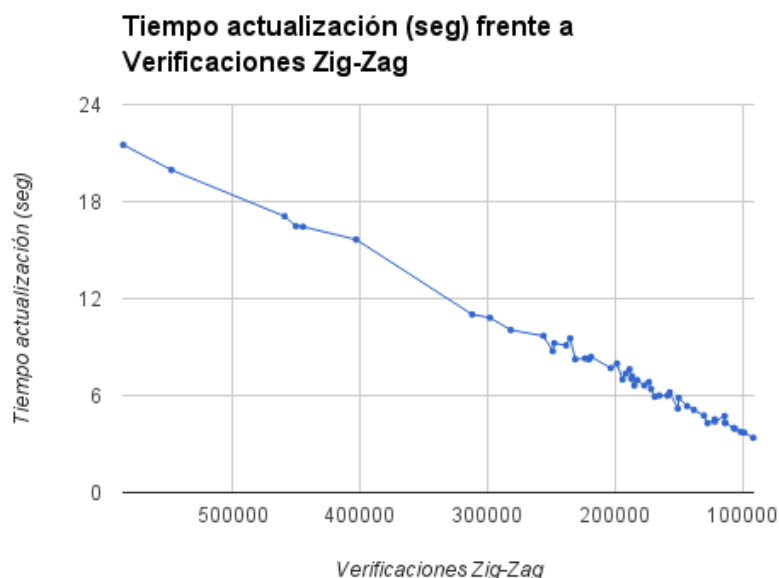


Fig. 5.3: Tiempo de actualización según verificaciones Zig-Zag

En las Figuras 5.4 y 5.5 graficamos tiempo de actualización y cantidad de verificaciones Zig-Zag en base a la cantidad de clases en \tilde{Z}_1 . Los dibujos de los gráficos son muy parecidos entre sí, justamente por la linealidad entre los dos factores que mencionamos antes.

Podemos inferir que tener más clases no significa tener menos cantidad de verificaciones Zig-Zag. Si tenemos muchas clases, pero no están bien distribuidas en cuanto a cantidad de nodos, entonces la cantidad de verificaciones Zig-Zag puede ser igualmente alta. Esto a su vez produce que el tiempo de actualización también sea considerable. Esta situación puede ocurrir por ejemplo cuando el refinamiento produjo muchas clases de un único nodo, pero el resto de las particiones continúan siendo muy grandes.

Idealmente para decidir si realizar la actualización, se debería contabilizar la cantidad de verificaciones Zig-Zag. Como ya discutimos previamente, esto no es posible y por ende debemos conformarnos con contar la cantidad de clases en \tilde{Z}_1 . Aunque estos experimentos nos demuestran que el número de clases no es garantía absoluta, sí es cierto que a partir de cierta cantidad de clases, la cantidad de verificaciones se mueve en un rango no tan amplio. Esto a su vez otorga un rango de tiempos de actualización que se puede tomar como aceptable. Por ejemplo, si tomamos como mínimo 222 clases en \tilde{Z}_1 , nos estamos asegurando (en este caso por lo menos) un tiempo de actualización menor a los 5 segundos (con tan sólo 10 fórmulas de aprendizaje). Podemos concluir que eligiendo bien dichos valores mínimos, el uso de la cantidad de clases en \tilde{Z}_1 es un buen criterio para el proceso de mining.

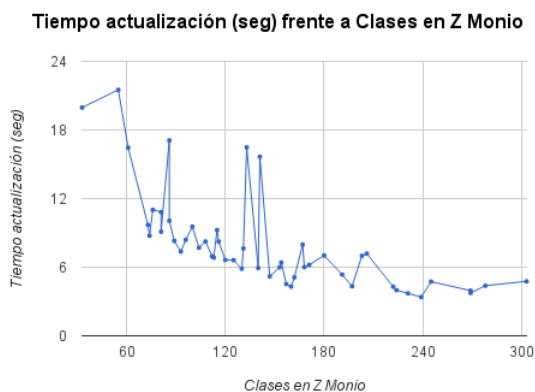


Fig. 5.4: Tiempo de actualización según cantidad de clases en \tilde{Z}_1

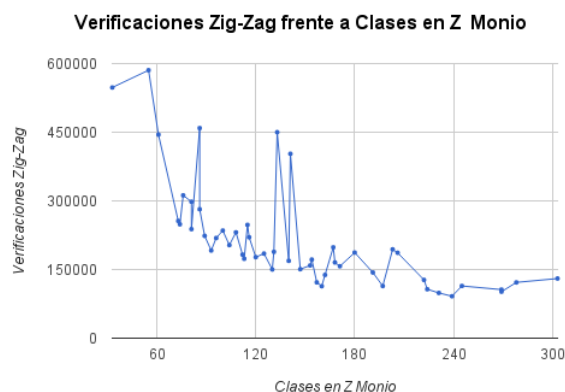


Fig. 5.5: Verificaciones Zig-Zag según cantidad de clases en \tilde{Z}_1

5.2. Actualización inmediata vs incremental en ℓ -bisimulación

5.2.1. Resultados favorables para la actualización incremental

En la sub-sección 3.3.1 discutimos acerca de cuáles eran los mejores valores para forzar la actualización, y cómo estos influyen en la utilidad de la actualización incremental. En el transcurso de la toma de dichas decisiones, y finalmente, como respaldo de las determinaciones finales, realizamos varios experimentos comparando los tiempos entre nuestra actualización incremental contra la actualización «inmediata». Llamamos inmediata al proceso de generar el Z_ℓ buscado en el cálculo inicial, antes de evaluar cualquier fórmula. Este enfoque trae aparejados tiempos de evaluación pequeños (pues nunca es necesario evaluar las expresiones en cada nodo, sino que se utiliza la relación Z_ℓ ya generada) y tiempos de actualización grandes (sobre todo para alcanzar los primeros niveles de aproximación).

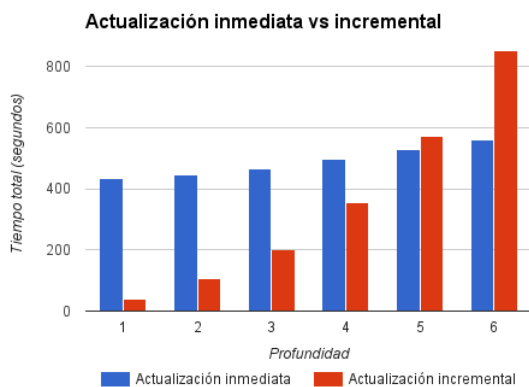


Fig. 5.6: Comparación de tiempos totales y acumulados entre actualización inmediata e incremental usando n^2

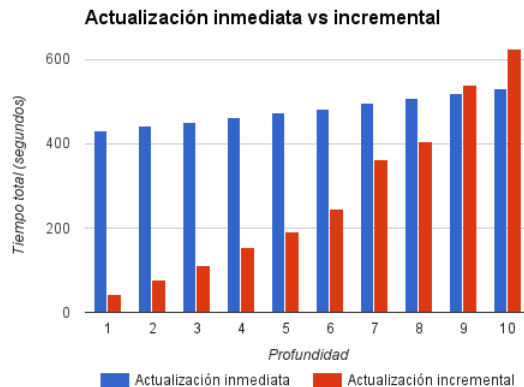


Fig. 5.7: Comparación de tiempos totales y acumulados entre actualización inmediata e incremental usando $\max\{\frac{n^2}{\ell'}, \frac{10}{\sqrt{\ell'}}\}$

Realizamos los experimentos ejecutando queries aleatorias sobre un XML de 100090 nodos y altura 17, con 5 labels distintos, 3 valores de datos distintos y entre 3 y 7 hijos cada nodo interno. Para los valores crecientes respecto a la profundidad (como $n^2 * \ell'^2$, $n^2 * \ell' * \sqrt{\ell'}$ y $n^2 * \ell'$) observamos que rápidamente se alcanzaban tiempos de evaluación (y por ende tiempos totales) demasiados altos, por lo que fueron descartados.

Las Figuras 5.6 y 5.7 grafican la comparación de tiempos totales (inicial + evaluación

+ mining + actualización) y acumulados (entre el inicio hasta la actualización al nivel de profundidad correspondiente) entre el algoritmo inmediato y el incremental para los valores n^2 y $\max\{\frac{n^2}{\ell'}, \frac{10}{\sqrt{\ell'}}\}$ respectivamente. En el primer caso, ya para la actualización a $\ell' = 5$ se evidencia que el máximo tiempo corresponde al caso incremental. En cambio, vemos en la Figura 5.7 que los tiempos del algoritmo incremental superan al algoritmo inmediato recién para $\ell' = 9$. Al encontrarnos satisfechos con estos resultados, terminamos eligiendo ese valor para nuestro algoritmo. Creemos que fórmulas de profundidad mayor o igual a 9 son poco probables de aparecer, aún así, si se quisieran tomar en cuenta, se podrían usar otros valores que tengan un decremento más pronunciado.

Las Tablas 5.3 y 5.4 (a continuación) recopilan la información de los experimentos realizados usando el valor elegido y funcionan como origen de datos para la Figura 5.7.

ℓ'	Queries ℓ'	Queries totales	Tiempo inicial	Tiempo evaluación	Tiempo mining	Tiempo actualización	Tiempo total
1	75	75	428,74	3,11	N/A	N/A	431,85
2	36	111	434,57	6,91	N/A	N/A	441,48
3	24	135	440,26	10,27	N/A	N/A	450,53
4	18	153	447,43	15,46	N/A	N/A	462,89
5	15	168	453,64	19,61	N/A	N/A	473,25
6	12	180	459,26	22,15	N/A	N/A	481,41
7	12	192	468,18	27,50	N/A	N/A	495,67
8	12	204	474,92	33,23	N/A	N/A	508,14
9	9	213	484,95	35,32	N/A	N/A	520,27
10	9	222	489,22	41,10	N/A	N/A	530,32

Tab. 5.3: Tiempos (en seg) algoritmo actualización inmediata (No Aplica mining ni actualización)

En la Tabla 5.3 se puede apreciar que del tiempo total, casi todo (más del 90%) corresponde al cálculo inicial, donde se realiza la actualización inmediata hasta llegar a la profundidad ℓ' correspondiente. También se puede observar que el costo grande se encuentra en el cálculo de Z_1 y que las posteriores actualizaciones no demoran mucho (sólo hay 1 minuto de diferencia entre el tiempo inicial para Z_1 y el de Z_{10}).

ℓ'	Queries ℓ'	Queries totales	Tiempo inicial	Tiempo evaluación	Tiempo mining	Tiempo actualización	Tiempo total
1	75	75	0,66	26,25	4,95	9,77	41,62
2	36	111	0,57	52,11	10,01	14,12	76,81
3	24	135	0,56	75,66	10,80	23,57	110,60
4	18	153	0,83	107,63	11,87	33,14	153,48
5	15	168	0,56	138,85	13,13	38,42	190,96
6	12	180	0,87	186,52	13,68	44,45	245,53
7	12	192	0,57	285,10	17,69	58,49	361,85
8	12	204	0,80	332,81	16,05	56,08	405,74
9	9	213	0,66	458,91	16,26	62,01	537,84
10	9	222	0,56	541,74	16,89	65,17	624,37

Tab. 5.4: Tiempos (en seg) algoritmo actualización incremental

En la Tabla 5.4 correspondiente al algoritmo incremental vemos que el tiempo inicial (en donde únicamente se calcula Z_0) es despreciable respecto al total y que por el contrario, el tiempo de evaluación es el que más repercute en el tiempo total. Esto se da sobre todo para las profundidades mayores, ya que su crecimiento es mucho más grande que el de los tiempos de mining y actualización. Reflejamos esta situación en la Figura 5.8:

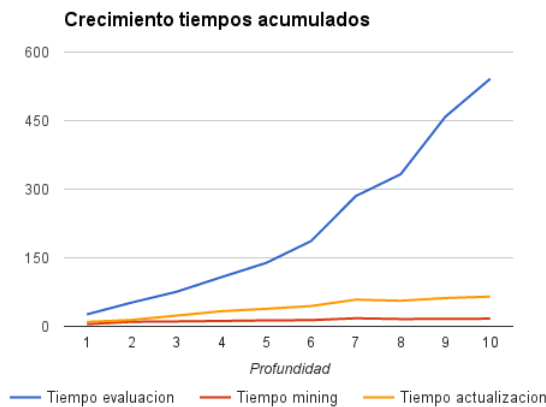


Fig. 5.8: Comparación del crecimiento de distintos tiempos acumulados

El gráfico de la función de crecimiento de los tiempos de actualización no es «suave» (sino que tiene etapas de crecimientos pronunciadas y otros no tanto) por la aleatoriedad de las queries generadas. En los niveles de profundidad altos, se ejecutan pocas queries y hay mucha dispersión entre sus tiempos de evaluación (por ejemplo para $\ell' = 10$ hay expresiones que demoran 84 segundos y otras tan sólo 4 segundos). Juntas, aleatoriedad y dispersión, se genera el comportamiento mencionado.

En este experimento estamos suponiendo que conocemos que nivel de profundidad queremos alcanzar. De no tomar esta suposición, el tiempo de cálculo inicial del algoritmo inmediato aumentaría mientras que esto no afecta de ningún modo al algoritmo incremental. Aún bajo esta suposición (poco realista), los tiempos del algoritmo incremental son mejores, porque aprovechan los aprendizajes. Sacrificando un poco los tiempos de evaluación, logramos reducir considerablemente los tiempos de actualización.

5.2.2. Resultados favorables para la actualización inmediata

Un benchmark XML tiene como objetivo evaluar la funcionalidad y/o la performance de una herramienta o sistema de procesamiento de XML. Uno de los benchmarks más comunes es XMark [20]. El mismo cuenta con un generador (*xmlgen*) que produce documentos XML modelando un sitio web de subastas.

Utilizamos *xmlgen* para generar un XML de tamaño similar al anterior. En este caso el documento cuenta con 100337 nodos y 74 labels distintos.

En el algoritmo incremental, para forzar la actualización a una 1-bisimulación, se necesitan 16428 queries ($\tau = 3\kappa = 3 * 74^2$). La magnitud de este valor nos hacía creer (ya antes de ejecutar la prueba) que los resultados iban a ser desfavorables a la actualización incremental. Ejecutamos primero varias tandas de 1000 queries, llegando a un tiempo de evaluación promedio de 375 segundos para cada tanda. Esto resulta en un tiempo pro-

porcional de 6160,5 segundos para las 16428 queries. Este tiempo equivale a más de 102 minutos, y no tiene en cuenta el tiempo inicial (menor a 1 segundo) ni los tiempos de mining y actualización.

Ejecutamos luego el algoritmo inmediato hasta alcanzar Z_1 . En este caso, el tiempo inicial fue aproximadamente 13 segundos, y el tiempo de evaluación de 1000 queries demoró aproximadamente 75 segundos. Esto resulta en un tiempo proporcional de 1232,1 segundos para las 16428 queries. Los 1245,1 segundos totales equivalen a poco menos de 21 minutos (en este caso no aplican los procesos de mining y evaluación).

Comparando únicamente los tiempos de evaluación (6160,5 segundos vs. 1232,1 segundos), el algoritmo incremental demora (exactamente) **5 veces más** que el inmediato. La relación se mantiene (aproximadamente) considerando los tiempos totales, ya que los 13 segundos de tiempo inicial en el caso inmediato se equilibran con los tiempos del proceso de mining y de actualización del algoritmo incremental.

Analizando los resultados, podemos llegar a dos conclusiones. En primera instancia, que el valor $\kappa = n^2$ resulta demasiado grande cuando n no es tan pequeño. En ejemplos anteriores, usamos documentos generados aleatoriamente, con una cantidad de labels chica (generalmente fue $n = 5$). En este caso, donde probamos con un XML «real», tenemos $n = 74$ lo que resulta en $\kappa = 5476$. Sería conveniente entonces encontrar una función de crecimiento respecto a la cantidad de labels que no fuera tan pronunciada. En segundo lugar, en este caso Z_1 tiene un total de 228 clases de equivalencia. Esto es sólo 3 veces más que las 74 clases iniciales en Z_0 . No sólo nunca se alcanzan los porcentajes de refinamiento deseables para un documento de este tamaño, sino que directamente se pierden todos los beneficios del enfoque incremental. La motivación del mismo es reducir los costosos tiempos de actualización que se dan en casos de mucho refinamiento, lo que claramente no ocurre para este XML. Nuevamente, el problema reside en que no conocemos de antemano el tamaño de Z_1 para optar inteligentemente entre los dos algoritmos. Sería interesante investigar y encontrar alguna característica del documento que pudiera orientar la elección.

En el caso de este documento en particular, la condición de poco refinamiento termina siendo más influyente que usar un valor alto como lo es $\kappa = n^2$. Si reducimos esto a $\kappa = 10$ y forzamos la actualización con $\tau = 3\kappa = 30$ obtenemos igualmente tiempos peores que el método inmediato. El tiempo total con dicho valor del algoritmo incremental es de 23,15 segundos, compuesto de 0,70 segundos de tiempo inicial, 11,25 segundos de evaluación, 1,90 segundos de mining y 9,30 segundos de actualización. El tiempo total en el algoritmo inmediato, ejecutando solamente 30 queries es de 15,25 segundos (13 segundos en el cálculo inicial y 2,25 segundos de evaluación).

Revisamos también la actualización a niveles de profundidad mayores, hasta $\ell = 5$: $|Z_2| = 1349$, $|Z_3| = 3128$, $|Z_4| = 5023$ y $|Z_5| = 5924$. A diferencia de lo que ocurría en las pruebas anteriores, en este caso el primer refinamiento no es tan grande y los siguientes sí generan más particiones. El mayor refinamiento aquí se da entre Z_1 y Z_2 . Esto se ve reflejado en el cálculo inicial: mientras que para alcanzar Z_1 el sistema demoraba 13 segundos, para alcanzar Z_2 (pasando antes por Z_1) demora 50 segundos. Este aumento también se refleja en el tiempo de actualización de Z_1 a Z_2 en el algoritmo incremental, y sumado a los tiempos de evaluación (mas allá de cuál κ se elija, si $\kappa = n^2$ o $\kappa = 10$), sigue siendo cierto que para este caso es conveniente el algoritmo inmediato.

5.3. *comp*-bisimulación

Los experimentos presentados en la Sección 5.1 fueron realizados para la ℓ -bisimulación. A pesar de esto (del tipo de pruebas y de los resultados numéricos), las conclusiones extraídas son válidas también para este caso. Esto sucede porque justamente dichos experimentos están centrados en la noción de aprendizaje, lo que es independiente de la aproximación utilizada. Algunas de dichas conclusiones son:

- Existe una utilidad práctica en el uso de aprendizajes que genera una reducción en los tiempos de actualización
- El uso de la bisimulación y de su correspondiente relación Z permite mejorar el rendimiento del proceso de evaluación (disminuyendo el tiempo que demora).
- Al aumentar la cantidad de aprendizajes, se reduce la cantidad de verificaciones Zig-Zag y el tiempo de actualización. Existe una relación lineal entre estos dos valores
- Además de la cantidad, influye la calidad de los aprendizajes para refinar. A su vez, no importa sólo tener más clases en \tilde{Z} , sino la distribución de los nodos en ellas.
- El ideal es contabilizar la cantidad de verificaciones Zig-Zag requeridas para actualizar, y con ello determinar cuando realizar dicho proceso. Sin embargo, como esto no es posible (y la única cota que se puede calcular es muy grosera), hay que conformarse con utilizar la cantidad de clases en \tilde{Z} . Esto puede ser igualmente un criterio aceptable tomando las decisiones correctas en lo referido a cuantas clases son requeridas para forzar la actualización.

A continuación exhibimos otros experimentos que sí están relacionados particularmente con la *comp*-bisimulación. Nos proponemos con los mismos mostrar en la práctica los beneficios del uso de esta aproximación, según lo discutido anteriormente.

En lo que respecta a memoria, no encontramos mejoras efectivas por utilizar esta aproximación. La cantidad de nodos es la misma, sólo que están distribuidos de diferente forma. Hay menos clases de equivalencia, pero el espacio en memoria que ocupa cada una de ellas es muy pequeño (sólo tiene atributos *id*, *label* y *profundidad*, todos de tipos nativos). El diccionario de listado de nodos por clase de equivalencia tiene menos entradas, pero cada listado es más grande por tener mayor cantidad de nodos.

Las ventajas referidas al uso de la *comp*-bisimulación están por lo tanto más ligadas a reducir los tiempos de determinados procesos, sobre todo el cálculo de la aproximación inicial. En cuanto a la evaluación de las expresiones pertenecientes al fragmento del lenguaje, los experimentos nos demostraron que los beneficios no eran tal cuál esperábamos.

Desarrollamos un generador de documentos, siguiendo cierta estructura para hacer unas primeras pruebas con la *comp*-bisimulación. Mostramos en la Figura 5.9 el esquema de dichos documentos. Se modela una universidad, con muchas cursadas (entre 20 y 40). Cada cursada tiene entre 20 y 40 alumnos, que a su vez tienen entre 10 y 20 finales cada uno. Por último, todos los finales tienen ejercicios 1 y 2, y el resto de los ejercicios (3, 4 y 5) son opcionales, por lo que aparecen con una probabilidad de 50% cada uno. En total son 9 labels distintos.

Si el *data* de cursadas y alumnos son *ids*, no tiene sentido que se refinen particiones por comparar dichos datos contra los *data* de los finales o ejercicios, que corresponden a puntajes. Podría suceder que tampoco se quisieran comparar todos los ejercicios entre sí, o que no se quisieran comparar los ejercicios contra los finales. Por estas razones, es conveniente utilizar para este caso la *comp*-bisimulación.

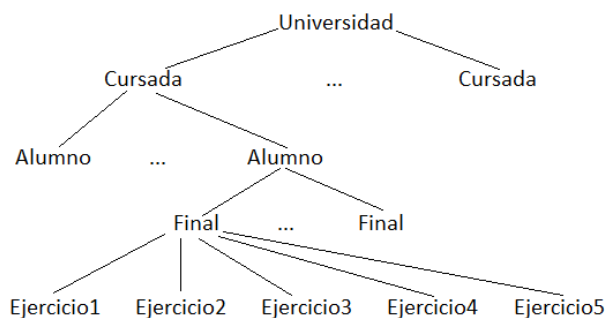


Fig. 5.9: Esquema del documento utilizado para las pruebas

Generamos varios documentos, hasta quedarnos con uno de nuestro gusto. El elegido tiene 80703 nodos: 1 universidad, 36 cursadas, 1141 alumnos, 17648 finales con sus respectivos ejercicios 1 y 2, 8816 ejercicios 3, 8930 ejercicios 4 y 8835 ejercicios 5. Inicialmente, Z_0 tiene 135 clases, mientras que el tamaño de la relación que cuenta con todas las tuplas de labels (Z , ya que corresponde a la bisimulación de $XPath_{=}(↓)$) es de 1521 clases.

5.3.1. Cálculo inicial y evaluación en Z y Z_0

Como primera prueba, comparamos los tiempos del proceso de cálculo inicial entre generar la relación total Z (con *comp* lleno) y generar la aproximación Z_0 . El tiempo promedio que demoró el caso total fue de 256 segundos contra 26 segundos del caso aproximado. La diferencia entre estos valores se debe a dos factores combinados. En primer lugar, lo explicado en el punto 2 de la sub-sección 4.4: durante la verificación por Zig-Zag entre x_1 y x_2 , en el caso de $comp = \emptyset$ sólo hacemos una iteración sobre los descendientes v de x_1 , mientras que en el otro caso hay una iteración sobre *pares* de descendientes v, w de x_1 . El segundo factor tiene que ver los costos de las verificaciones. El caso total puede llegar a tener mayor cantidad de verificaciones entre dos nodos y además tiene dos iteraciones anidadas (mientras que el caso aproximado tiene solo una iteración) pero al ser más restrictivo, en muchos casos puede encontrar rápidamente una condición que no se cumpla y terminar con dicha verificación. En la verificación entre x_1 y x_2 en el cálculo de Z_0 , podría suceder que haya que recorrer todos los descendientes de x_1 y para cada uno buscar todos los correspondientes descendientes de x_2 . El costo de esto puede ser tan alto que el cálculo inicial termina demorando más para la aproximación que para el caso total, lo que no pareciera ser muy conveniente¹.

En nuestro documento los dos factores se combinan bien: hay diferencias entre los nodos que no dependen de la comparación de datos, lo que impide los altos costos mencionados y produce cierto refinamiento. Las 9 clases iniciales (una por cada label, ya que no se repiten a distintas profundidades) se refinan en 135 clases en Z_0 , demorando sólo 26 segundos.

¹ En una prueba con otro documento (cuyo único refinamiento era producto de la comparación de datos) obtuvimos un tiempo inicial de 145 segundos para el método incremental y 35 segundos para el inmediato.

Luego del proceso de cálculo inicial, empezamos a experimentar con la evaluación de queries. Consideramos dos conjuntos de expresiones: las fórmulas de tipo test de datos o que contienen expresiones de ese tipo, y todo el resto de las fórmulas. El primer conjunto no pertenece al fragmento del lenguaje determinado por la \emptyset -bisimulación. Por lo tanto, esas fórmulas deben ser evaluadas en todos los nodos del árbol, consumiendo un tiempo considerable. En cambio, para las expresiones del segundo conjunto, se puede utilizar la información que provee la relación Z_\emptyset , resultando en tiempos menores. Esto se puede comparar contra la bisimulación total de $XPath_{=}(\downarrow)$: en este caso todas las fórmulas pertenecen al lenguaje y para todas se puede usar Z , pero al ser la relación más grande (está más refinada), la evaluación en ella es un poco más costosa. Realizamos unas pruebas simples generando queries aleatorias para validar empíricamente estos conocimientos:

1. 50 queries de profundidad entre 0 y 2. En promedio, el método aproximado demoró 9 segundos contra 1 segundo del total, y 16 de las 50 fórmulas contenían expresiones de tipo test de datos.
2. 50 queries de profundidad entre 0 y 3. En promedio, el método aproximado demoró 20.70 segundos contra 1.43 segundos del total, y 20 de las 50 fórmulas contenían expresiones de tipo test de datos.
3. 50 queries de profundidad entre 2 y 4. En promedio, el método aproximado demoró 52.20 segundos contra 2.38 segundos del total, y 35 de las 50 fórmulas contenían expresiones de tipo test de datos.
4. 50 queries de tipo test y profundidad entre 2 y 4. En promedio, el método aproximado demoró 53.47 segundos contra 1.09 segundos del total.
5. 50 queries que **no** fueran de tipo test de datos y de profundidad entre 2 y 4. En promedio, el método aproximado demoró 1.24 segundos contra 2.66 segundos del total.

Analizando los resultados, podemos ver en las 3 primeras pruebas que a medida que aumenta la profundidad, aumenta también la cantidad de expresiones de tipo test que aparecen (pues hay más expresiones en total y la distribución es «uniforme») y con ello, el tiempo que demora el método aproximado. La complejidad de las fórmulas hace que los tiempos del método total también crezcan, pero este crecimiento es pequeño y a un ritmo lento. En la primera prueba la mayoría de las fórmulas no contienen expresiones de tipo test, por lo que según la teoría, el método aproximado debería demorar menos que el total. En la práctica, para dichas fórmulas esto ocurre, pero las diferencias son muy pequeñas (en general, menos de 0.02 segundos) porque el sobre-refinamiento no es tanto y además, el método total se saltea ciertos procesamientos como verificar si una fórmula pertenece al lenguaje (porque todas pertenecen). Comparando las pruebas 3 y 4, vemos que los tiempos del método aproximado casi ni se modifican y que los del método total se reducen a más de la mitad. Lo primero se puede explicar con que pese al aumento de expresiones de tipo test, muchas no se lleguen a ejecutar (por estar luego de un \downarrow, \vee, \wedge) y lo segundo quizás se deba a los tipos de queries y su selectividad. La quinta prueba nos muestra que quitando todas las expresiones de tipo test, el método aproximado se comporta mejor, pero la diferencia es muy pequeña.

5.3.2. Actualización inmediata vs incremental

Realizamos luego otra prueba sobre el mismo documento XML. Este experimento consiste en 16 tandas de evaluación de 50 queries cada una, tanto sobre el caso inmediato que genera la bisimulación total de $\text{XPath}_{=}\downarrow$ como sobre el caso incremental que inicialmente genera la \emptyset -bisimulación. Las queries se generan de forma aleatoria con estas dos restricciones: la profundidad es entre 0 y 3 (inclusive) y las expresiones de tipo test sólo comparan nodos con labels *Ejercicio1* y *Ejercicio2*. El proceso de mining establece que la medida mínima es de $\kappa = 5n$ donde n es la cantidad de labels distintos, por lo que luego de $\tau = 3\kappa = 3 * 5 * 9 = 135$ ejecuciones de la tupla (*Ejercicio1*, *Ejercicio2*), se realizará la actualización que agrega ese par de labels a *comp*. En la Figura 5.10 mostramos los tiempos de evaluación (en segundos) totales por tanda, diferenciando entre algoritmo incremental e inmediato.

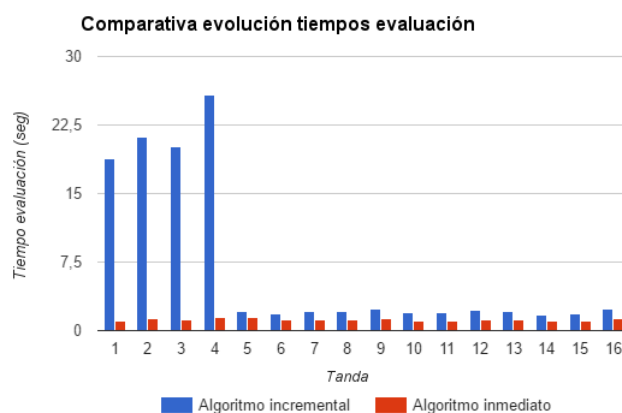


Fig. 5.10: Comparativa tiempos evaluación en algoritmo incremental e inmediato

Para el método incremental, la actualización se produjo luego de la query n°41 de la cuarta tanda. A partir de entonces, todas las consultas realizadas pertenecían al fragmento del lenguaje, por lo que los tiempos de dicho método disminuyeron notablemente. A pesar de la reducción, podemos observar que los tiempos siguieron siendo (apenas) mayores a los del método inmediato. Creemos que esto sucede por lo explicado antes: el sobre-refinamiento no es tanto y además, el método inmediato se saltea ciertos procesamientos que agregan un overhead que es pequeño pero resulta considerable a esta escala. Podemos visualizar la evolución de los tiempos acumulados de evaluación en la Figura 5.11.

Si tenemos en cuenta también el resto de los procesos, entonces la gráfica se modifica: en un principio hay una diferencia amplia por el cálculo inicial (es mayor el tiempo del algoritmo inmediato), que se reduce a partir de las evaluaciones. La brecha disminuye considerablemente en las primeras cuatro tandas de evaluación y lentamente durante el resto. Además, durante la cuarta tanda se produce en el método incremental la decisión de actualizar en el proceso de mining (que demora 3.08 segundos) y la actualización, que demora otros 24.80 segundos. Todo esto lo podemos ver en la Figura 5.12.

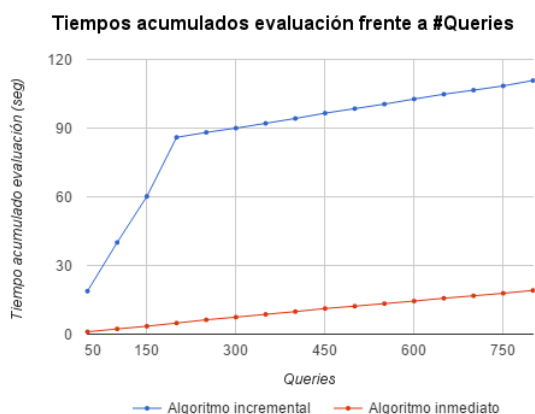


Fig. 5.11: Tiempos evaluación acumulados

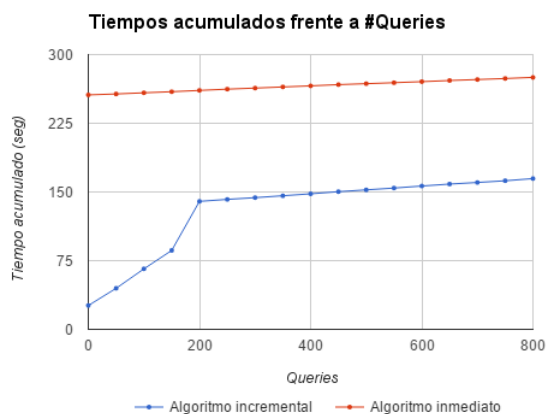


Fig. 5.12: Tiempos totalacumulados

En este caso la actualización demoró un tiempo considerable porque los aprendizajes no resultaron muy buenos. En la *comp*-bisimulación el refinamiento que se da es muy preciso y las queries generadas aleatoriamente no logran (al no tener en cuenta la estructura del documento) producir aprendizajes de «calidad». Para validar que los aprendizajes sean realmente útiles también para este tipo de aproximación, realizamos otra prueba en la que adicionamos a las queries aleatorias otras escritas por nosotros, que sabíamos eran buenas. De esta forma, logramos reducir el tiempo de actualización a 5.53 segundos. Estos aprendizajes generaron una relación intermedia $\tilde{Z}_{comp'}$ de 329 clases con una buena distribución de nodos, mientras que sólo usar las queries aleatorias consigue alrededor de 150 clases (sin una buena distribución). Respecto al mining, los tiempos que agrega son despreciables, salvo cuando decide realizar la actualización. Esto es porque no se realiza procesamiento complejo, salvo en dicho momento, cuando hay que generar la relación intermedia $\tilde{Z}_{comp'}$. Esta construcción sí es algo costosa, demorando entre 3 y 4 segundos en promedio.

Concluimos en que al menos para este caso los beneficios de la *comp*-bisimulación se ven reflejados más en el proceso de cálculo inicial que en el proceso de evaluación. No existe la suficiente diferencia de tamaño entre Z_{comp} (135 clases) y la relación total Z (1521 clases) para conseguir ventajas sustanciales al momento de evaluar fórmulas. Adicionalmente, es importante que la incorporación a *comp* de las tuplas pertinentes se produzca rápido, porque la evaluación en todos los nodos es muy costosa. Por lo tanto, es crucial (nuevamente) definir adecuadamente los criterios y valores que aparecen en el proceso de mining. La diferencia de tiempos «ganada» durante el cálculo inicial puede perderse si se evalúan muchas expresiones de test que no pertenecen al fragmento del lenguaje.

Intentamos generar otros documentos con distinto esquema para los cuales sí haya un beneficio en la evaluación usando el método incremental, pero no pudimos lograrlo. A pesar de alcanzar una brecha considerable entre los tamaños de Z_{comp} y la relación total Z , los tiempos no acompañaban dicha diferencia. Por ejemplo, generamos un XML de 25501 nodos con $|Z_{comp}| = 503$ y $|Z| = 6503$. Dejando de lado las fórmulas con expresiones de tipo test, esperábamos que para el resto la diferencia de 6000 clases entre las relaciones se haga valer. Sin embargo, los resultados no acompañaron del todo estos «deseos»: para 50 fórmulas aleatorias sin expresiones de test, la diferencia de tiempos fue de tan sólo 2 segundos aproximadamente. Para una fórmula «compleja» armada a mano y repetida 50 veces, la diferencia de tiempos fue de 6 segundos y esto fue lo mejor que pudimos conseguir.

5.4. Uso de índices

Mencionamos en la introducción de este trabajo la existencia de varios índices desarrollados y utilizados en el marco de Core-XPath que utilizan la noción de bisimulación modal. Mostramos también nuestro interés en investigar posibles adaptaciones de dichas herramientas a nuestro marco de trabajo: *data trees*, $\text{XPath}_=(\downarrow)$ y su correspondiente noción de bisimulación. Realizamos nuestro análisis, sobre una variante del 1-Index [9], por ser el más simple de todos. Para ilustrar ciertos detalles, presentamos un ejemplo:

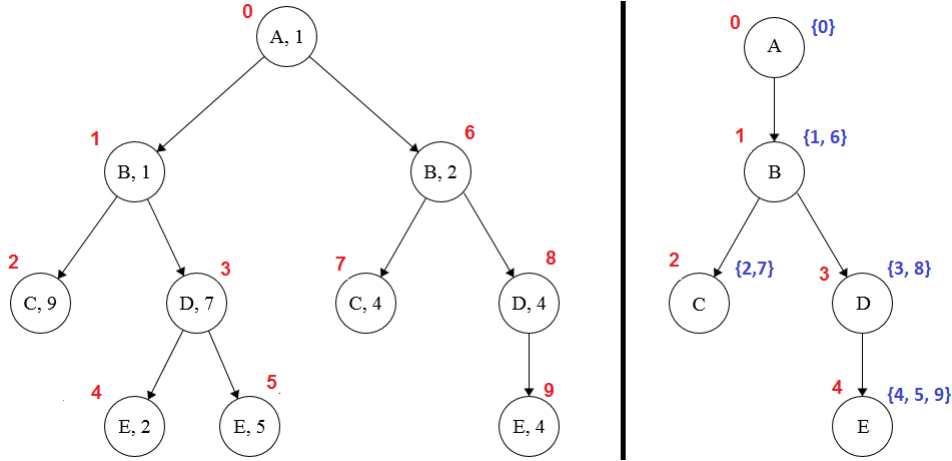


Fig. 5.13: Árbol original (izquierda) y su 1-Index «inverso» (derecha)

En la Figura 5.13, del lado izquierdo se encuentra el *data tree* DT utilizado como ejemplo de *comp*-bisimulación en la sub-sección 2.2.6. En el lado derecho se observa el 1-Index «inverso» (es decir, que mira los caminos salientes y no los entrantes) I . Dicho índice se construye a partir de la bisimulación modal de la siguiente forma: por cada clase de equivalencia se crea un nodo con el label correspondiente, y por cada arista $v \rightarrow v'$ en el árbol original se crea la arista $[v] \rightarrow [v']$ en el índice² (donde $[x]$ significa la clase de equivalencia del nodo x del árbol original). Si r era el nodo raíz en DT , entonces $[r]$ es la raíz de I . Si w es un nodo de I , denotamos $extent(w)$ al conjunto de nodos de DT que pertenecen a la clase de equivalencia w , es decir $extent(w) = \{v \in DT \mid [v] = w\}$. En nuestra Figura, mostramos a la derecha de cada nodo su correspondiente $extent$ en azul.

En la Observación 7 mencionamos que la *comp*-bisimulación con $comp = \emptyset$ se corresponde a la bisimulación modal. Podemos por lo tanto comprobar la validez de I revisando el Z_\emptyset presentado como primer ejemplo de la sub-sección 2.2.6.

El índice (tanto este tipo como el resto) se despoja del *data* de cada nodo, conservando sólo el comportamiento de navegación. Esto implica que exceptuando las queries de tipo test ($\langle \alpha \odot \beta \rangle$), las otras pueden ser evaluadas en el índice, sin perder validez. Este subconjunto de queries justamente se corresponde con el fragmento Core-XPath.

Surge entonces la inquietud de cómo contestar queries de tipo test utilizando la estructura construida. Para contestar una consulta $\langle \alpha \odot \beta \rangle$ en el *data tree* primero obtenemos $\llbracket \alpha \rrbracket^{DT}$ y $\llbracket \beta \rrbracket^{DT}$, y luego (si ninguna es vacía) recorreremos ambas colecciones buscando un par que cumpla con lo que indique el operador \odot , utilizando aquí el *data* del nodo almace-

² El índice así construido puede no ser un árbol, pero siempre es un grafo acíclico dirigido (DAG).

nado en el atributo *value*. Nuestra idea para el índice consistía en un procesamiento similar, evaluando α y β en el índice, pero en vez de recorrer $C_1 = \llbracket \alpha \rrbracket^I$ y $C_2 = \llbracket \beta \rrbracket^I$, haríamos la iteración sobre $extent(C_1)$ y $extent(C_2)$, pudiendo de esta forma acceder al dato del nodo. Sin embargo, dicha metodología resulta inválida, como veremos en el siguiente ejemplo.

Consideremos la fórmula $\varphi = D \wedge \langle \downarrow [E] \neq \downarrow [E] \rangle$. El único nodo en DT que satisface φ es el de id 3. Situémonos ahora en el índice, en el nodo de id 3 y label D. La primer subfórmula de la conjunción, puede contestarse directamente en I . Luego es necesario evaluar $\langle \downarrow [E] \neq \downarrow [E] \rangle$. Como dijimos en el parrafo anterior, obtenemos primero $C_1 = \llbracket \downarrow [E] \rrbracket^I = \{4\}$ y $C_2 = \llbracket \downarrow [E] \rrbracket^I = \{4\}$, y luego calculamos $extent(C_1) = \{4, 5, 9\}$ y $extent(C_2) = \{4, 5, 9\}$. Recorriendo estas dos últimas colecciones, el par $\langle 4, 5 \rangle$ satisface la condición de desigualdad de datos. Si se itera en otro orden (o si en DT el nodo 3 no tuviera al hijo 5) y se toma la tupla $\langle 4, 9 \rangle$, también se satisface la condición de desigualdad de datos, pero de forma errónea ya que dichos nodos pertenecen a sub-arboles distintos. Tenemos entonces que $3 \in \llbracket \varphi \rrbracket^I$, pero $extent(3) \not\subseteq \llbracket \varphi \rrbracket^{DT}$ pues $extent(3) = \{3, 8\}$ y $\llbracket \varphi \rrbracket^{DT} = \{3\}$. Lo que se debería hacer es, para cada nodo en $extent(\llbracket \varphi \rrbracket^I)$ (donde el $extent$ de un conjunto es la union de los $extent$ de cada elemento), validar en el árbol original si realmente se satisface φ . Esto habría que hacerse para cada fórmula que contenga a esta de tipo test, sino se podría caer en el mismo problema. Por ejemplo, si se tuviera la fórmula $\psi = \langle \downarrow [\varphi] \rangle$, $\llbracket \psi \rrbracket^I = \{1\}$ pues $\llbracket \varphi \rrbracket^I = \{3\}$, pero $\llbracket \psi \rrbracket^{DT} = \{1\} \neq \{1, 6\} = extent(1)$.

Llegamos a la conclusión que usar exclusivamente el índice en conjunto a la función $extent$ no es suficiente para contestar correctamente todas las queries de XPath $_{=}$ (\downarrow). El enfoque que también incluye la validación contra el árbol original sí funciona, y es análogo al que realizamos en nuestra aplicación para el caso de $comp = \emptyset$. Las fórmulas que son exclusivamente de Core-XPath (es decir, aquellas que no contienen como subfórmulas a expresiones de tipo test) son las únicas que pertenecen al fragmento del lenguaje \emptyset -XPath $_{=}$ (\downarrow) y por lo tanto las únicas que pueden utilizar la información de la bisimulación para agilizar la evaluación. En nuestro caso no tenemos un índice «materializado», pero tenemos Z_{\emptyset} , que equivale a la bisimulación modal, que a su vez es la que «refleja» el índice. Para todo el resto de las fórmulas (aquellas que sí contienen subfórmulas de tipo test), la evaluación se hace directamente en el árbol original.

Sin haber hecho ninguna análisis exhaustivo ni pruebas al respecto, creemos que con el resto de los índices (F&B-Index, A(k)-Index, etc.) ocurre el mismo inconveniente. A nivel general, este problema está relacionado a cómo cocientar por bisimulación con datos. Lo que se quisiera lograr es justamente un *data tree* más pequeño (como un índice) que sea bisimilar al original, y por lo tanto pueda ser aprovechado para mejorar el rendimiento en cuanto a tiempo y memoria. Si quisieramos convertir I en un *data tree* tendríamos el inconveniente de no saber qué valor de *data* ponerle a cada nodo. Más aún, no necesariamente hay una forma de elegirlo que nos asegure obtener un DAG bisimilar al árbol original. Existe toda una rama de investigación sobre este tema, que busca alguna propiedad que permita hacer el cociente sobre cualquier *data tree* o por lo menos, sobre cierta clase restringida (pero interesante) de *data trees*. Por el momento, lo único que podemos asegurar es que nuestras aproximaciones son refinamientos de ciertos índices (por ejemplo, Z_{comp} es un refinamiento del 1-Index invertido para cualquier $comp$ y Z_{ℓ} es un refinamiento del índice $A(\ell)$ [12]).

6. CONCLUSIONES

En este trabajo nos enfocamos en el estudio de las aproximaciones de bisimulación. Mencionamos anteriormente que el cálculo de la bisimulación total es costoso, y pudimos observar esto por ejemplo en los experimentos de la sub-sección 5.2.1. Además de ser costoso, muchas veces también resulta ineficiente: es poco probable que se realicen queries de profundidades grandes comparadas con la altura del *data tree* y que realmente se deseen comparar los datos de todos los pares de labels que contempla el documento XML. Por estos dos motivos, utilizar aproximaciones puede ser una buena decisión.

Para contemplar el nivel de profundidad, existe la ℓ -bisimulación. Por otro lado, a partir de la inquietud acerca de las comparaciones entre cualquier par de labels, introducimos un nuevo tipo de aproximación, al cuál denominamos *comp*-bisimulación. Definimos teóricamente este nuevo tipo y enunciarnos y demostramos ciertas propiedades, siendo la principal la que establece la correspondencia entre la equivalencia lógica y la bisimulación. A nivel práctico, la definición de *comp*-bisimulación permite reducir los tiempos del cálculo inicial y eliminar sobre-refinamiento innecesario. Mostramos con un ejemplo cómo funciona esta aproximación, concluyendo con el mismo que efectivamente esta noción es distinta a la bisimulación modal y a la bisimulación de XPath=(\downarrow). Luego, experimentamos con documentos grandes, pudiendo probar empíricamente los beneficios prácticos mencionados.

Acompañamos el uso de aproximaciones con un planteo de actualizaciones incrementales. Sabemos que probablemente no necesitemos llegar a la bisimulación total, pero como usuarios posiblemente desconozcamos qué nivel de aproximación querríamos alcanzar. Por lo tanto, proponemos que el mismo sistema se autorregule y defina cuándo y «a qué» actualizar, según las expresiones evaluadas. Bajo este enfoque, comenzamos con un proceso de cálculo inicial, en el cuál se genera la aproximación más básica y simple. Esto permite tener un sistema operativo rápidamente y a bajo costo. Las expresiones pertenecientes al fragmento del lenguaje definido por la aproximación pueden ser respondidas aprovechando la información que brinda la bisimulación (por su correspondencia con la equivalencia lógica). El resto de las expresiones requieren una verificación en cada nodo del árbol. Aprovechamos este procedimiento almacenando el resultado de la evaluación como aprendizaje.

Luego de cada evaluación, se realiza el proceso de mining. Por la cantidad de información involucrada, no se utilizan algoritmos de *data mining*, pero el objetivo es similar: tomar decisiones según los datos recolectados. En nuestro caso, tenemos las queries ejecutadas y los aprendizajes. A partir de la base de conocimientos formada, se decide si es conveniente realizar la actualización, y en caso afirmativo, cuál sera la nueva aproximación. La heterogeneidad de los XMLs y de las queries a realizar (tanto cantidad como tipo) dificulta en gran medida la definición de los criterios que determinan estas decisiones. Optamos por la actualización cuando hay un buen porcentaje de refinamiento (que es la única medida exacta que podemos conocer) o cuando hay una cantidad considerable de evaluaciones. Nuevamente, estos conceptos de «buen» y «considerable» resultan subjetivos, y obviamente no son los mejores para la totalidad de los casos.

El resultado del proceso anterior no es sólo la característica de la nueva aproximación, sino también la relación de bisimulación intermedia \tilde{Z} . En el cálculo de dicha relación es donde se saca el mayor provecho de los aprendizajes realizados y almacenados. La construcción de \tilde{Z} es sencilla, ya que únicamente consiste en el refinamiento iterativo de la relación por las dos particiones que determina cada aprendizaje. De esta forma, nos acercamos a la aproximación que buscamos alcanzar, y podemos reducir la cantidad de verificaciones Zig-Zag a realizar. Dado que esta cantidad prácticamente fija el tiempo total del proceso de actualización, la reducción implica una disminución de este tiempo.

En resumen, nuestro enfoque combinado se basa en comenzar con una aproximación simple, sacrificar un poco los tiempos de evaluación para generar aprendizaje y utilizar este conocimiento para realizar la actualización a un bajo costo. El proceso de mining se encarga de caracterizar la aproximación que debemos alcanzar a partir de las queries ejecutadas y tiene la difícil tarea de determinar cuando realizar la actualización, intentando que ni los tiempos de evaluación ni los de actualización sean muy elevados.

Desarrollamos una herramienta que implementa estos 4 procesos (cálculo inicial, evaluación de expresiones, mining y actualización). La misma es una aplicación web, que permite la interacción del usuario de diversos modos (más visuales o más estadísticos). Esta herramienta también puede ser utilizada con fines didácticos. Por nuestra parte, la empleamos a lo largo del trabajo para poder realizar pruebas que nos guiaran en las decisiones a tomar. Los experimentos presentados ponen de manifiesto que este enfoque combinado de aproximaciones y actualización incremental es tanto válido como útil. Tal cuál mencionamos, no siempre es el mejor camino, pero pudimos verificar que es eficiente para algunos casos. Lo mismo ocurre para la *comp*-bisimulación: es posible que su uso en determinados casos resulte poco provechoso, o quizás hasta desfavorable. Sin embargo, de la experimentación efectuada podemos concluir que la idea de este nuevo tipo de aproximación puede llegar a ser beneficiosa.

6.1. Trabajo futuro

A lo largo del trabajo fuimos mencionando ciertos temas cuyas problemáticas planteamos, pero cuyas resoluciones no pudimos abordar (o no del todo). Otros temas, por su magnitud, quedaron directamente fuera del alcance, pero su estudio puede resultar provechoso. Algunos de estos asuntos pueden ser tomados como futuras líneas de investigación:

- Investigación sobre la verificación de ciertas características del XML que puedan resultar en elegir si usar el algoritmo incremental (y con qué valores en los diferentes procesos) o el algoritmo de actualización inmediata.
- Definición de nuevos tipos de aproximación. Algunas posibles ideas relacionadas a los tipos estudiados son la ℓ -*comp*-bisimulación y la *comp*⁺-bisimulación.

La primera es la combinación entre ambos tipos y puede ser interesante si se logra mantener o mejorar los beneficios que provee cada uno de ellos.

El propósito de la *comp*⁺-bisimulación es restringir las comparaciones de datos a ciertas tuplas de cadenas de labels (en vez de tuplas de labels únicamente). Esto puede ser útil para documentos con labels repetidos para diferentes entidades: por

ejemplo, queremos permitir comparaciones del tipo $\langle [Docente] \downarrow [ColorOjos] = [Docente] \downarrow [ColorFavorito] \rangle$ pero no estas otras $\langle [Alumno] \downarrow [ColorOjos] = [Alumno] \downarrow [ColorFavorito] \rangle$. Esto no se puede hacer en la *comp*-bisimulación porque sólo miramos el último nivel $\langle ColorOjos, ColorFavorito \rangle$.

- Bases de datos dinámicas: poder afrontar la actualización frente a cambios en el modelo (en el documento XML). Es necesario analizar qué procesos se ven afectados y cómo pueden ser adaptados. Por ejemplo, ciertos aprendizajes dejan de tener valor y es necesario descartarlos.
- Implementación del modulo de optimización. Hay ciertas equivalencias sobre fórmulas que fueron mencionadas y que podrían usarse para reducir los tiempos de evaluación. Algunas equivalencias estaban relacionadas con la forma sintáctica que establece la *comp*-bisimulación y otras eran generales. El desarrollo de dicho módulo conlleva también la posterior experimentación para determinar su beneficio.
- Implementación de la eliminación de tuplas de *comp*. Mencionamos brevemente algunas opciones para afrontar esta modificación, pero no las estudiamos en profundidad.
- Mayor trabajo sobre Z_ℓ con $\ell \geq 4$, pues muchos experimentos los hicimos para niveles de profundidad bajos. Aunque creemos que los de profundidad baja son los más usuales, sería útil estudiar el comportamiento en niveles mayores y poder definir ciertos valores (sobre todo del proceso de mining) en concordancia.
- Adaptación del trabajo realizado para otros fragmentos de XPath₌ como XPath₌($\uparrow\downarrow$)
- Adaptación del trabajo realizado para DAGs o grafos en vez de arboles. Este cambio implica una dificultad de ordenes de magnitud mayor, por lo que habría que analizar qué de todo sigue siendo válido y conveniente.
- Mejoras en el diseño e implementación de la herramienta web. Por ejemplo, una de las ya mencionadas es desligar a las expresiones de las características que las aproximaciones necesitan. Otras tienen que ver con evitar recalcular \tilde{Z} , tanto para la ℓ -bisimulación como para la *comp*-bisimulación.
- Mejoras en el manejo de memoria. A pesar de las modificaciones realizadas y mencionadas, en ocasiones de documentos grandes y muchas queries juntas para evaluar, seguimos recibiendo excepciones que indican que el proceso se quedó sin memoria.
- Investigación sobre el uso de índices (1-Index, F&B-Index, A(k)-Index, etc) en algún fragmento de XPath₌. Nuestro análisis no fue muy profundo, y quizás encarando el problema desde otra perspectiva, se pueda llegar a una adaptación de dichos índices para usarlos de manera provechosa.
- Cociente de *data tree* por bisimulación con datos. Se busca alguna propiedad que permita generar un nuevo *data tree*, bisimilar al original pero más pequeño, sobre cualquier árbol de datos o sobre cierta familia interesante.
- Profundizar el estudio de la *comp*-bisimulación, en particular en lo referente al beneficio teórico de mejora en los tiempos de evaluación, que en la práctica no se pudo replicar con el éxito esperado.

Bibliografía

- [1] J. Clark and S. DeRose, “XML path language (XPath).” Website, 1999. W3C Recommendation. <http://www.w3.org/TR/xpath>.
- [2] G. Gottlob, C. Koch, and R. Pichler, “Efficient algorithms for processing XPath queries,” *ACM Trans. Database Systems*, vol. 30, no. 2, pp. 444–491, 2005.
- [3] M. Bojańczyk, A. Muscholl, T. Schwentick, and L. Segoufin, “Two-variable logic on data trees and XML reasoning,” *J. ACM*, vol. 56, no. 3, pp. 1–48, 2009.
- [4] D. Figueira, S. Figueira, and C. Areces, “Basic model theory of XPath on data trees,” in *ICDT*, pp. 50–60, 2014.
- [5] M. Forti and F. Honsell, “Set theory with free construction principles,” *Annali della Scuola Normale Superiore di Pisa - Classe di Scienze*, vol. 4.10, no. 3, pp. 493–522, 1983.
- [6] R. Milner., *A Calculus of Communicating Systems*, vol. 92 of *LNCS*. Springer, 1980.
- [7] D. Park, “Concurrency and automata on infinite sequences,” in *5th GI-Conference on Theoretical Computer Science*, vol. 104 of *LNCS*, pp. 167–183, 1981.
- [8] R. Paige and R. Tarjan, “Three partition refinement algorithms,” in *SIAM Journal on Computing*, 1987.
- [9] T. Milo and D. Suciu, “Index structures for path expressions,” in *ICDT: 7th International Conference on Database Theory*, 1999.
- [10] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, 1999.
- [11] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth, “Covering indexes for branching path queries,” in *Proceedings of SIGMOD*, 2002.
- [12] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, “Exploiting local similarity for efficient indexing of paths in graph structured data,” in *Proceedings of ICDE*, 2002.
- [13] J. Min, C. Chung, and K. Shim, “Apex: An adaptive path index for xml data,” in *Proceedings of SIGMOD*, 2002.
- [14] Q. Chen, A. Lim, and K. Ong, “D(k)-index: An adaptive structural summary for graph-structured data,” in *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, 2003.
- [15] H. He and J. Yang, “Multiresolution indexing of xml for frequent queries,” in *ICDE*, 2004.
- [16] R. Kaushik, P. Bohannon, J. Naughton, and P. Shenoy, “Updates for structure indexes,” in *VLDB*, 2002.

-
- [17] K. Yi, H. He, I. Stanoi, and J. Yang, “Incremental maintenance of xml structural indexes,” in *SIGMOD*, 2004.
 - [18] D. Saha, “An incremental bisimulation algorithm,” in *Arvind, V., Prasad, S. (eds.)*, 2007.
 - [19] J. Deng, B. Choi, J. Xu, and B. S.S., “Optimizing incremental maintenance of minimal bisimulation of cyclic graphs,” in *DASFAA*, 2011.
 - [20] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse, “Xmark: a benchmark for xml data management,” in *Proceedings of the Very Large Database Conference*, 2002.