Universidad del Bío Bío
Facultad de Ciencias Empresariales
Departamento de Sistemas de Información

Profesor Guía: Sr. Claudio Gutiérrez
Profesor Informante: Sr Patricio Gálvez

IMPLEMENTACIÓN Y RESULTADOS EMPÍRICOS PARA UN ALGORITMO DE DETECCIÓN DE CAMBIOS EN DOCUMENTOS XML.

HABILITACIÓN PROFESIONAL.

Concepción, 27 de Febrero de 2009

Alumnos:
Alejandro Barra S.
Andrés Landaeta S.

AGRADECIMIENTOS

Madre, GRACIAS por darme la posibilidad que de mi boca salga esa palabra, siempre serás mi inspiración para alcanzar mis metas, Gracias por enseñarme que todo se aprende y que todo esfuerzo es al final recompensa. Tu esfuerzo, se convirtió en tu triunfo y el mío.

Alejandro Barra S.

A mi familia por apoyarme.

Andrés Landaeta S.

RESUMEN

Los contenidos de la Web pueden ser expresados en un formato que pueda ser entendido, interpretado y usado por diferente(s) software, permitiéndoles buscar, compartir e integrar información de manera más sencilla.

Una vez que se concibe el diseño de XML también lo hace el concepto de DTD (Definición del Tipo de Documento) expresamente reflejado en la aplicación RDF. Mediante RDF podemos extraer la información de la Base de Datos para crear un formato más comprensible para las máquinas. Con esta información podemos realizar deducciones lógicas, combinar información, generar información nueva a partir de una ya existente, realizar consultas complejas en buscadores, entre otras.

La detección de cambios en XML se ha vuelto más importante en muchas áreas de aplicación. Los algoritmos de detección de cambios en documentos XML proporcionan a usuarios y aplicaciones una descripción más significativa de los cambios detectados. Cada algoritmo de detección de cambios en documentos XML se centra en su aspecto de detección de cambios tomando en cuenta sus propiedades y optimizaciones. Se proveerán descripciones de algoritmos de detección de cambio para documentos XML y descripción de ciertos tipos de propiedades que juegan un papel clave en estos algoritmos. Asimismo, se presentan algunos ejemplos de detección de cambios en XML aplicados en el área de fusión, versiones y sincronización de documentos XML. Se tratará de explicar los tipos de propiedades más importantes a considerar cuando la velocidad, exactitud o minimización es la exigencia principal.

La eficiencia del algoritmo propuesto reside en la detección de isomorfismo en los sub-árboles. La complejidad teórica del algoritmo es $O(n \log_m n)$, donde *n* es el número de vértices y *m* es el mayor número de sub-árboles que posee la raíz. El cual se compara con el desempeño del algoritmo X-Diff, de $O(n^2)$, mostrando algunos resultados preliminares sobre su tiempo de ejecución y la calidad del resultado final.

ÍNDICE

CAPÍTULO 1: *OBJETIVO GENERAL Y ESPECÍFICOS, RESUMEN DEL PROBLEMA.* 1
1.1 RESUMEN DEL PROBLEMA 2
1.2 OBJETIVOS DEL ESTUDIO 4
1.2.1 OBJETIVO GENERAL 4
1.2.2 OBJETIVOS ESPECÍFICOS 4
1.3 APORTE 5
1.4 LÍMITES 6
1.4.1 ABARCA 6
1.4.2 NO ABARCA 6
CAPÍTULO 2: *RESUMEN DE PROYECTO DE TÍTULO* 7
2.1 ORIGEN XML 8
2.2 ORIGEN RDF 8
2.2.1 PRESENTANDO RDF 9
2.3 ESTADO DEL ARTE XML-RDF 11
2.4 DETECCIÓN DE CAMBIOS 12
2.5 APLICACIONES 14
2.5.1 DOS-VÍAS DE FUSIÓN 14
2.5.2 TRES-VÍAS DE FUSIÓN 15
2.5.3 VERSIONES DEL DOCUMENTO XML 16
2.5.4 VERSIONES DEL CONTENIDO DEL SITIO WEB 16
2.5.6 SISTEMAS DE CONSULTAS 16
2.6 ALGORITMOS DE DETECCIÓN DE CAMBIOS 17
2.6.1 LADIFF 17
2.6.2 MH-DIFF 18
2.6.3 XML TREEDIFF 18
2.6.4 DIFFX 19
2.6.5 IBM'S XML DIFF & MERGE TOOL 19
2.6.6 3DM'S MATCHING ALGORITHM 20
2.6.7 XYDIFF 20
2.6.8 VM TOOLS 21
2.6.9 DIFFXML 22
2.6.10 KF-DIFF+ 22
2.6.11 XML DIFF AND PATCH 22
2.6.12 X-DIFF 23
2.6.13 DELTAXML 23
2.6.14 TREEPATCH 24
2.6.15 BIODIFF 24
2.7 PROPIEDADES DE LOS ALGORITMOS 26
2.7.1 ÁRBOL ORDENADO O DESORDENADO 26
2.7.2 RENDIMIENTO Y COMPLEJIDAD 26
2.7.3 CORRECCIÓN 27
2.7.4 USO DE MEMORIA 27
2.7.5 ENTRADA TAMAÑO 27
2.7.6 SEMÁNTICA 28

2.7.7 MÍNIMO DELTA SCRIPT 28
2.8 ALGORITMOS TENTATIVOS A IMPLEMENTAR 29
2.8.1 X-DIFF 29
2.8.2 LADIFF 32
2.8.3 DIFFX 36
2.9 CONCLUSIONES PROYECTO DE TÍTULO 38
CAPÍTULO 3: *ALGORITMO A DESARROLLAR.* 41
3.1 ALGORITMO A DESARROLLAR 42
3.1.1 ESQUEMA DEL ALGORITMO A DESARROLLAR 42
3.2 FUNCIONAMIENTO DEL ALGORITMO (REPRESENTACIÓN GRAFICA) 43
3.2.1 ÁRBOL ORIGINAL (T_1) 43
3.2.2 ÁRBOL MODIFICADO (T_2) 44
3.2.3 MATCHING 46
CAPÍTULO 4: *EVALUACIÓN DEL DESEMPEÑO DE LOS ALGORITMOS* 47
4.1 ESCENARIOS DE PRUEBA 48
4.2 RESULTADOS EMPÍRICOS 50
4.3 TENDENCIA DE LAS PRUEBAS EMPÍRICAS DE LOS ALGORITMOS 57
CONCLUSIÓN 59
CONCLUSIÓN GENERAL 60
OBJETIVO GENERAL HABILITACIÓN PROFESIONAL 60
OBJETIVOS ESPECÍFICOS HABILITACIÓN PROFESIONAL 61
CONCLUSIONES PERSONALES 62
BIBLIOGRAFÍA 63
ANEXO 1 67
ANEXO 2 70
ANEXO 3 75
ANEXO 4 85

INDICE DE FIGURAS

Figura 1.1: *Acceso a una Base de Datos usando Documentos XML*…… 3

Figura 2.1: *Descripción RDF*…….…….…….. 9

Figura 2.2: *Estructura RDF*..... 10

Figura 2.3: *Descripción de aplicaciones RDF*……...… 11

Figura 2.4: *Fundamento de dos-vías de fusión*……...…… 14

Figura 2.5: *Fundamento de tres-vías de fusión*…….…. 15

Figura 4.1: *Árboles Ordenados con 10% de cambios*……. 51

Figura 4.2: *Árboles Ordenados con 15% de cambios*..…..... 51

Figura 4.3: *Árboles Ordenados con 20% de cambios*...……...……... 52

Figura 4.4: *Árboles Ordenados con 25% de cambios*...……...……... 52

Figura 4.5: *Árboles Ordenados con 30% de cambios*...……...……... 53

Figura 4.6: *Árboles Desordenados con 10% de cambios*…… 54

Figura 4.7: *Árboles Desordenados con 15% de cambios*…… 55

Figura 4.8: *Árboles Desordenados con 20% de cambios*…… 55

Figura 4.9: *Árboles Desordenados con 25% de cambios*…… 56

Figura 4.10: *Árboles Desordenados con 30% de cambios*…… 56

Figura 4.11: *Tendencia Árboles Ordenados*.…...……...…… 57

Figura 4.12: *Tendencia Árboles Desordenados* ……...…… 58

CAPÍTULO 1: *OBJETIVO GENERAL Y ESPECÍFICOS, RESUMEN DEL PROBLEMA.*

En este Capítulo presenta los antecedentes generales que dieron la base al proyecto y que permitieron fundamentar la profundización del tema para desarrollarlo.

Este Capítulo está estructurado de la siguiente manera:

- Sección 1: Resumen del Problema.
- Sección 2: Objetivos del Estudio.
- Sección 3: Aportes.
- Sección 4: Límites.

1.1 RESUMEN DEL PROBLEMA

En la ciencia de la computación resulta de vital importancia el ahorro de recursos en servidores. Si consideramos que tenemos una base de datos de gran volumen la cual es actualizada de forma frecuente, generando un consumo alto de recursos y en el lado opuesto tenemos un considerable número de terminales quienes a su vez necesitan acceder a nuestra base de datos, pero las aplicaciones de estos terminales está construida en distintos lenguajes (Java, Visual Basic, Php, por mencionar algunos), lo lógico es que cada terminal haga una consulta a la base de datos haciendo una búsqueda entre los registros y sacando la información que necesite, pero eso nos da como resultado un consumo de recursos más alto. Además, recordemos que constantemente se ingresa, modifica y elimina información de la Base de Datos.

La utilización de XML (Anexo 1) en la interacción con las Bases de Datos, es una ventaja, debido a que este lenguaje es estándar, y con ello, la comunicación con otros lenguajes es compatible. Esta comunicación, se realiza cuando un terminal debe realizar una consulta a la Base de Datos. Se accede al archivo XML, que contiene la información a la que el terminal tiene acceso. Se compara el documento XML con la Base de Datos, en caso de modificación de esta última, se actualiza el XML y la terminal consulta todas las veces que sean necesarias el documento, siendo el proceso imperceptible al usuario, ya que dará la impresión que se está consultando a la Base de Datos. La comparación será realizada mediante la implementación de un algoritmo para la detección de cambios en documentos XML, el cual deberá actualizar el documento XML cuando así se requiera. De esta forma se ahorrará y se optimizará el tiempo de consultas a la Base de Datos. Tal como se ilustra en la Figura 1.1.

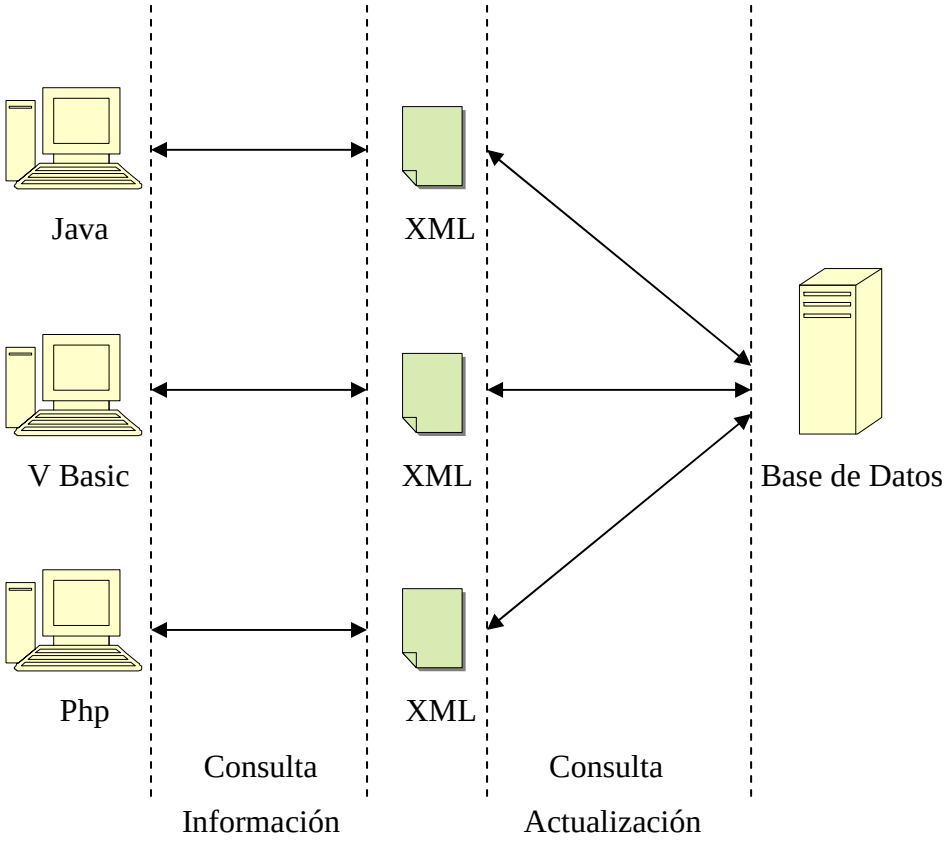


Figura 1.1: *Acceso a una Base de Datos usando Documentos XML*

1.2 OBJETIVOS DEL ESTUDIO

1.2.1 OBJETIVO GENERAL

Implementar y realizar pruebas empíricas para un eficiente algoritmo de detección de cambios en documentos XML.

1.2.2 OBJETIVOS ESPECÍFICOS

- Implementar algoritmo(s) tentativo(s).

- Realizar pruebas empíricas del algoritmo y contrastarlo con otra(s) propuesta(s) existente(s).

- Determinar el o los mejores algoritmos desarrollados para la detección de cambios en documentos XML.

1.3 APORTE

El aporte consiste en obtener uno o más algoritmos, con los cuales se puedan determinar modificaciones en documentos XML. Los que podrán ser ocupados como base para futuras implementaciones, así como para desarrollo de nuevos algoritmos o bien para el perfeccionamiento de éstos. Su uso será posible por ejemplo en:

- **Sindicación de noticias:** Trabaja a través de lectores de noticias RSS/RDF (Anexo 1), los cuales permiten agregar la dirección Web de un sitio y recibir las novedades y noticias de este. Esta tecnología trabaja como base con documentos XML. Comúnmente se ocupa en Blogs y páginas personales.

- **Actualización de software:** La actualización de programas, normalmente consiste en presionar un botón y obtener las actualizaciones del sitio que corresponde. Hay algunos sitios que trabajan con documentos XML, en los cuales está la información de los paquetes nuevos. En caso que no hubiese necesidad de actualizar el programa, la consulta sólo se realiza al documento XML, entregando el mensaje correspondiente, y no accesaría a la Base de Datos del sitio, donde se descargan las actualizaciones.

- **Consultas Bancarias:** En los Bancos las consultas, que hacen las personas a sus cuentas, son variadas por ejemplo: consultas de saldo, de depósito, de giro, entre otras. A su vez la cantidad de veces que realizan la misma consulta en un día puede ser reiterada. Es por ello, que se utilizan archivos XML en donde están los datos actualizados de las cuentas de los clientes, para en caso que un cliente desee saber el estado de su cuenta, la terminal accede el documento XML tantas veces como sea necesario, evitando con esto el colapso de la Base de Datos.

1.4 LÍMITES

1.4.1 ABARCA

- Este proyecto se enfocó en investigar y diseñar algoritmo(s) para detectar cambios en documentos XML.

- Se demostró a través de experimentos y pruebas empíricas las ventajas que nuestro algoritmo pretende lograr (Habilitación Profesional).

- Se hizo una comparación entre el algoritmos diseñado y los ya existentes (X-Diff [20], DiffX [25], por mencionar algunos).

1.4.2 NO ABARCA

- No se pretendió realizar un manual de aprendizaje sobre el lenguaje XML.

- No se diseñó un Software, debido a que la finalidad era desarrollar un algoritmo para la detección de cambios en documentos XML.

CAPÍTULO 2: *RESUMEN DE PROYECTO DE TÍTULO*

El objetivo general del Proyecto de Título consistió en: Analizar los algoritmos existentes y diseñar el algoritmo(s) tentativo a desarrollar para la detección de cambios en documentos XML. Para esto se establecieron una serie de objetivos específicos, los cuales fueron:

- Recopilar información de los diferentes algoritmos utilizados para la detección de cambios en documentos XML, que nos permita obtener todos los datos necesarios para tener una visión amplia del tema a tratar.

- Realizar un análisis a priori de los algoritmos utilizados para la detección de cambios en documentos XML.

- Determinar posibles algoritmos que nos permitan obtener soluciones tentativas para la detección de cambios en documentos XML.

Estos objetivos específicos permitieron lograr una exhaustiva revisión de los tópicos que resultarían fundamentales para sentar las bases teóricas que permitieron entender el trasfondo de este proyecto, los cuales son tratados a fondo en este Capítulo de Resumen del Proyecto de Título.

2.1 ORIGEN XML

XML es un sub-conjunto directo de “SGML” (Standard Generalized Markup Language, ISO 8879). El estándar “SGML” define la estructura y contenido de diferentes tipos de documentos electrónicos [1].

XML es un metalenguaje con el que es posible definir lenguajes para determinar la estructura y el contenido de documentos XML propios (creados con datos de los usuarios y no como los datos rígidos predefinidos de HTML). Así el usuario definirá sus propios elementos que conformarán el tipo de documentos deseado y cómo tienen que estar organizados para que sean correctos.

Actualmente existen aplicaciones que usan XML, una de ellas es Glade y se encuentra en plataforma Linux, es una aplicación gráfica que utiliza XML como documento que almacena datos cargando las estructuras gráficas en su ventana. Esto es un ejemplo que reafirma los beneficios que trae la utilización de XML como un medio de estandarización y una sintaxis fácil de analizar para la representación de datos entre otros [2].

2.2 ORIGEN RDF

El *Resource Description Framework* es una DTD (definición del tipo de documento) de XML o, una aplicación de metadatos (Anexo 1) (recursos que proveen información de sí mismos) que utiliza XML a fin de proporcionar un marco estándar para la interoperabilidad en la descripción de contenidos Web. RDF no es más que la infraestructura que permite esa restricción gracias a la codificación, reutilización e intercambio de metadatos estructurados. Con estas prerrogativas, interoperabilidad y estructuración, RDF es el modelo más promisorio para asociar información sobre el contenido de los recursos Web, y no es arriesgado decir que promete ser el modelo de descripción de la información para las bibliotecas digitales del siglo XXI, así como para optimizar de forma generalizada la búsqueda y recuperación en la Web [3].

2.2.1 PRESENTANDO RDF

RDF (Resource Description Framework) como su nombre lo indica es un framework para describir e intercambiar metadatos. Véase *Ejemplo 1*. RDF está construido en base a las siguientes reglas [3][4][5]:

- **Recurso:** Es cualquier objeto Web identificable unívocamente por un URI (Anexo 1), es decir, un identificador uniforme de recursos como un URL. Un recurso puede ser un documento HTML; una parte de una página Web como por ejemplo un elemento HTML o XML dentro de un documento fuente, una colección de páginas, un sitio Web completo; en síntesis cualquier recurso entendido como objeto de información.

- **Propiedad:** Son aspectos específicos, características, atributos o relaciones utilizadas para describir recursos, por ejemplo autor o título. Una propiedad necesita ser un recurso de forma tal que pueda tener sus propias propiedades.

- **Descripción:** Consiste en la combinación de un recurso, una propiedad y un valor. Estas partes son conocidas como el sujeto, predicado y el objeto, respectivamente. Una descripción es como se ilustra en la Figura 2.1.

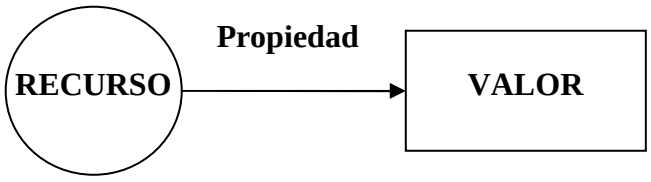


Figura 2.1: *Descripción RDF*.

Ejemplo 1:

En la figura 2.2 se presenta mediante un ejemplo la Estructura RDF [7]:

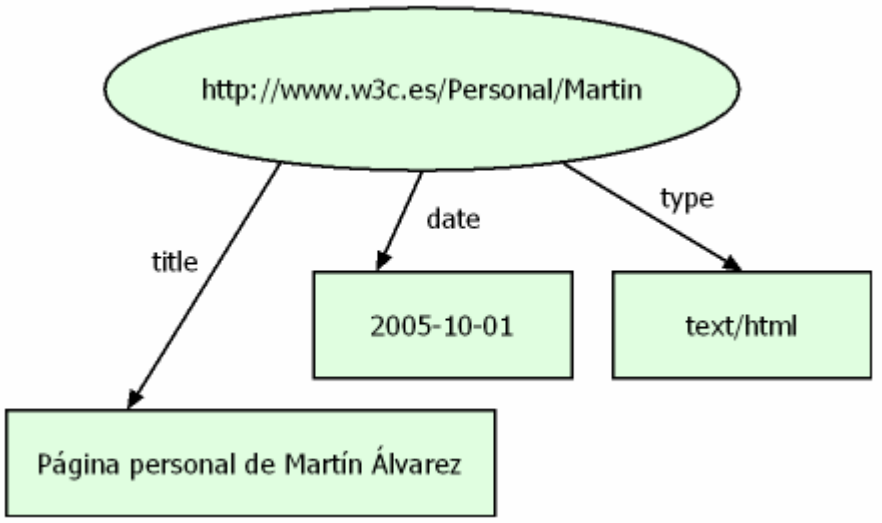


Figura 2.2: *Estructura RDF*.

A continuación, la Estructura RDF de la Figura 2.2 codificada completamente en RDF y XML [6][7]:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://description.org/schema/">
  <rdf:Description about="http://www.w3c.es/Personal/Martin">
    <s:type>text/html</s:type>
    <s:date>2005-10-01</s:date>
    <s:title>Página personal de Martín Álvarez</s:title>
  </rdf:Description>
</rdf:RDF>
```

Donde la etiqueta <?xml version="1.0"?> corresponde a la versión de XML, y la etiqueta <rdf:Description about="http://www.w3c.es/Personal/Martin"> corresponde a la estructura RDF, la cual contiene la identificación el tipo de documento, la fecha de creación y el título del ejemplo.

2.3 ESTADO DEL ARTE XML-RDF

Como XML es un metalenguaje con el que es posible definir lenguajes para determinar la estructura y el contenido de los documentos XML, el estado del arte lo podemos observar en los distintos tipos de implementaciones RDF que existen y se usan hoy en día.

Entre las aplicaciones RDF más populares podemos mencionar: RSS (Really Simple Syndication)[8][9][10], FOAF (Friend Of A Friend)[11][12], DOAC (Description Of A Career)[13], CC/PP (Composite Capabilities/Preferences Profile)[14]. Como se ilustra en la *Figura 2.3.*

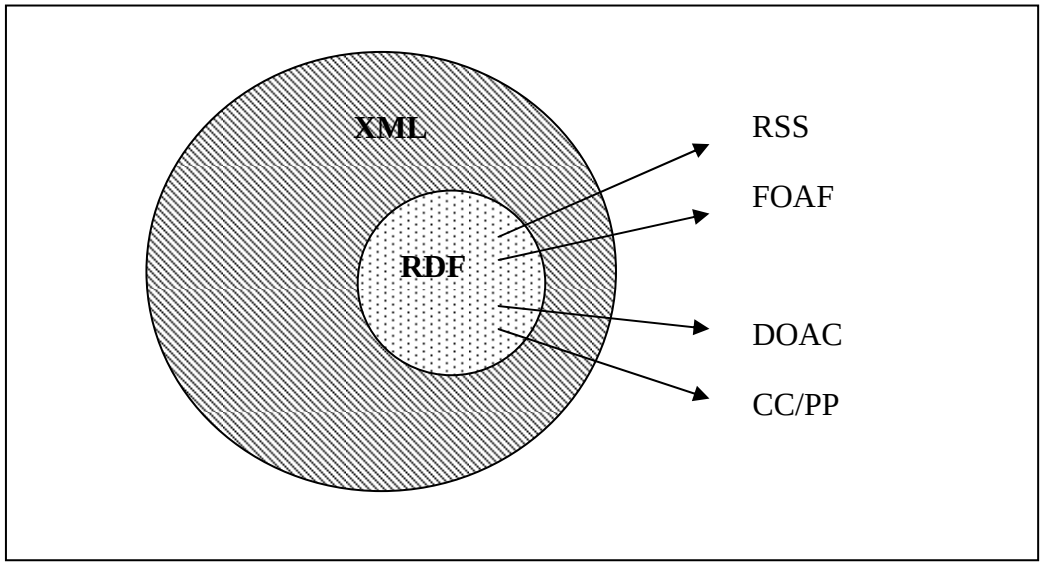


Figura 2.3: *Descripción de aplicaciones RDF*

2.4 DETECCIÓN DE CAMBIOS

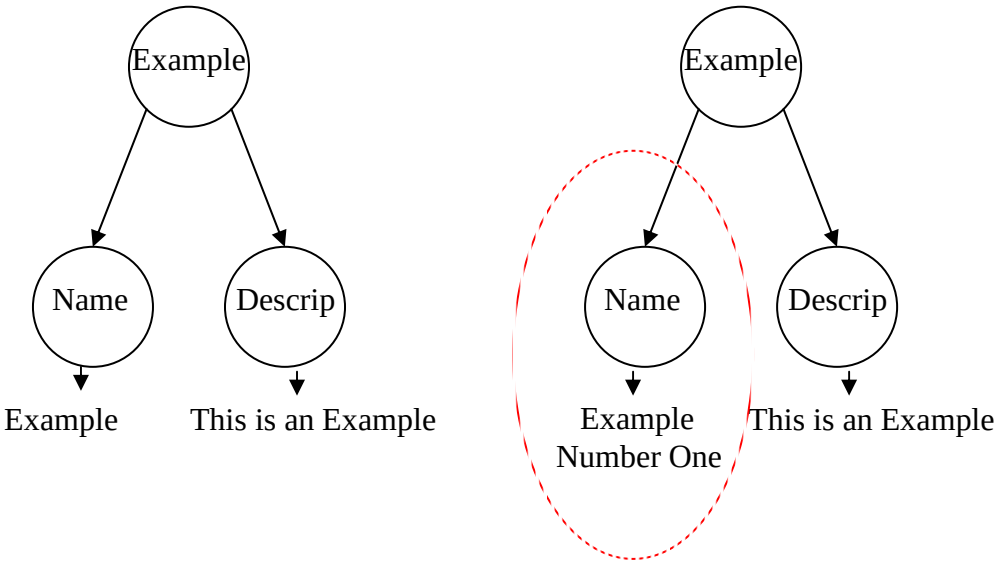
Para entender de mejor manera la detección de cambios en documentos XML se debe observar desde dos aristas:

- Al considerar los documentos XML como un árbol de datos estructurado, dos árboles se pueden comparar en igualdad de nodos y en igualdad de sub-árboles, ambos árboles XML pueden ser concordantes. Los nodos y sub-árboles que no son concordantes, implica que ambos árboles XML son diferentes. Véase *Ejemplo 2.*

- Al examinar los documentos XML, el algoritmo encuentra las secuencias de líneas comunes en ambos documentos, intercaladas con grupos de diferentes líneas. XML Diff (Anexo 1) considerará un árbol de datos estructurado jerárquicamente como un documento XML (basado en texto).

Ejemplo 2:

a) Dos versiones Diferentes de un árbol XML como grafo:



a) Visto como documento XML:

```
<example id=”1”>
<name>Example</name>
<descrip>This is an Example</descrip>
</example>

<example id=”1”>
<name>Example Number One</name>
<descrip>This is an Example</descrip>
</example>
```

El *Ejemplo 5* describe dos versiones de un árbol XML. La segunda versión contiene una palabra extra ‘number’. El algoritmo considera esto como un cambio en la línea, pero al analizar el documento como un árbol XML, el algoritmo sólo encuentra el nodo <name> cambiado. Cuando el documento contiene más nodos <example>, cada uno con un nodo <name>, el algoritmo puede utilizar una especie de ruta *(path)* de especificación para identificar la ubicación del nodo <name>.

Los cambios son a menudo reunidos en un delta script o delta documento. Como ilustra la *Figura 2.4* y la *Figura 2.5*. El delta script contiene una representación de esos cambios. El delta script puede ser usado como un documento de ruta *(path)*, en otras palabras, aplicar los cambios y recrear el documento cambiado de nuevo. Esto es muy útil cuando la versión original está presente en otro lugar y necesita ser actualizada sin la transferencia de todo el documento cambiado. La otra parte puede recrear el documento cambiado usando sólo el documento original y el delta script.

Los cambios también son llamados operaciones y los tipos de operaciones más comunes son ‘insertar’ y ‘eliminar’. La mayoría de los algoritmos consideran ‘actualizaciones’, que pueden especificarse como una combinación de ‘eliminar’ e ‘insertar’. Menos común es ‘mover’, que también puede especificarse como una combinación de ‘eliminar’ e ‘insertar’. La ventaja de ambos, ‘actualizar’ y ‘mover’, es que es menor el tamaño del delta script, mientras que el uso de ‘eliminar’ e ‘insertar’ podría conducir a un mayor tamaño del delta script. El uso de un tipo de operación ‘copia’ puede dar lugar a un delta script aún más pequeño, pero casi nunca es utilizado. Cuando una concordancia es encontrada en un determinado nodo, el algoritmo puede sacarlo de futuras comparaciones.

2.5 APLICACIONES

Cuando los documentos son guardados como documentos XML, se plantea la necesidad de un eficiente algoritmo de detección de cambios que proporcione mayor información acerca de los cambios y fusión de los documentos que mantienen la sintaxis XML intacta.

2.5.1 DOS-VÍAS DE FUSIÓN

La modificación en paralelo de documentos XML, plantea la necesidad de fusionar (combinar o mezclar) las copias en un solo documento XML. Con Dos-vías de fusión, no hay noción de la versión de los documentos, sólo las diferencias entre dos documentos que necesitan ser fusionados son detectadas, como se ilustra en la *Figura 2.4.*

Por lo general en algoritmos de Dos-vías de fusión, el usuario tiene que comprobar la mayoría de las diferencias antes de que se apliquen, por ejemplo, cuando se elimina una parte del fichero A, el algoritmo de fusión considerará la misma parte en el fichero B como un agregado, por lo que la eliminación no alcanzará al documento fusionado a menos que el usuario interfiera.

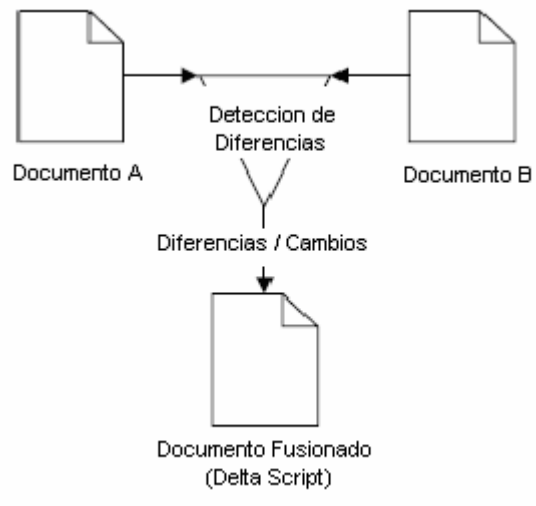


Figura 2.4: *Fundamento de dos-vías de fusión.*

2.5.2 TRES-VÍAS DE FUSIÓN

Tres-vías de fusión, es una variante de fusión cuando dos réplicas de un documento existente son editadas independientemente. *La Figura 2.5* ilustra, como el documento original y ambas replicas modificadas, son fusionadas en un documento (Delta Script).

Como anteriormente se ha descrito, una herramienta de Tres-vías de fusión siempre necesita tres documentos, el documentos original que es editado simultáneamente, y los resultantes dos documentos modificados independientemente. El documento original es primero comparado con uno de los documentos modificados. Las diferencias, o cambios, se reúnen en un delta documento que describe como el documento original debe ser modificado para crear el documento modificado. Después el documento original se compara con el segundo documento modificado, y el delta documento resultante se combina con el primer delta. El delta documento combinado se aplica luego al documento original resultando un documento fusionado que contiene todas las modificaciones de ambos documentos modificados [15][16].

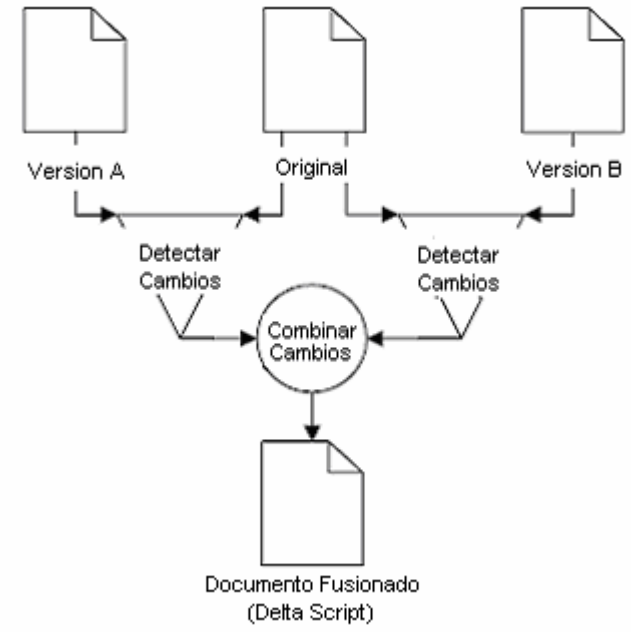


Figura 2.5: *Fundamentos de Tres-vías de fusión.*

2.5.3 VERSIONES DEL DOCUMENTO XML

El delta documento se utiliza en los Sistemas de Versiones Concurrentes (CVS), que mantienen un registro de revisiones y versiones del documento. Los Sistemas de Versiones de Gestión (MVS), al igual que CVS, ofrecen la posibilidad de "ir atrás en el tiempo" y recuperar una versión anterior de un documento determinado. Sin embargo, CVS y MVS no son eficientes para documentos estructurados como XML [17][18].

2.5.4 VERSIONES DEL CONTENIDO DEL SITIO WEB

WebDAV (Web-based Distributed Authoring y Versioning) permite a los usuarios realizar autoría de Web remota incluyendo las operaciones de crear, editar, copiar, mover y eliminar los recursos Web. WebDAV también permite a los usuarios el bloqueo exclusivo o bloqueo compartido de un recurso Web. En caso de bloqueo compartido, la edición simultánea de documentos necesita ser fusionada.

2.5.6 SISTEMAS DE CONSULTAS

Wang nos da dos ejemplos de sistema de consulta cuando una herramienta de detección de cambios puede ser muy útil [20]:

- Evaluación del Incremento de la Consulta. Con una consulta repetida el sistema de consulta sólo tiene que volver a evaluar la consulta en el delta datos, por lo tanto, el usuario no recibe resultado antiguos (desactualizados*).*

- Evaluación de Condición Trigger. Una herramienta de detección de cambios puede rápidamente informar las variaciones que corresponden a una determinada condición desencadenada (trigger*).*

2.6 ALGORITMOS DE DETECCIÓN DE CAMBIOS

Esta sección muestra una visión general de diferentes algoritmos de detección de cambios y sus propiedades. Algunos algoritmos tienen una clara conexión entre sí, se utilizan como ejemplo para otros algoritmos o se han mejorado o adaptado por otros algoritmos.

2.6.1 LADIFF

LaDiff es una herramienta de diferenciación para información estructurada jerárquicamente. Como entrada, toma dos versiones de un documento LaTeX produce un documento LaTeX con los cambios marcados. Utiliza un algoritmo llamado FastMatch, que utiliza una función *igualdad* para comparar nodos en un árbol ordenado.

FastMatch realiza primero la comparación *(matching)* de los nodos que aparecen en el mismo orden, a partir de las hojas del documento. Según Cobéna su costo es de O ($n * e + e^2$), donde n es el número de nodos hoja y ***e*** la suma del número de sub-árboles eliminados e insertados y el tamaño total de sub-árboles que se movieron para el menor delta script [21].

LaDiff maneja las operaciones de actualizar, insertar, eliminar e incluso mover. Pero LaDiff sólo soporta algunos elementos LaTeX (Anexo 1), por lo tanto, su FastMatch no puede utilizarse como un algoritmo de detección de cambios en documentos XML, pero sus algoritmos sub-yacentes han inspirado a otros en la creación de sus propios algoritmos [22].

2.6.2 MH-DIFF

MH-Diff puede considerarse como la contrapartida de LaDiff y es un algoritmo de detección de cambios *significativos* entre árboles de datos estructurados *jerárquicos*. Su objetivo es retratar los cambios entre dos árboles en un breve y descriptivo camino. Utiliza un delta script que le da la secuencia de operaciones necesarias para transformar el árbol original en el nuevo árbol.

MH-Diff maneja las operaciones de actualización, inserción y eliminación, además de las operaciones de mover y copiar. Las operaciones de mover y copiar resultan en un delta script de mayor calidad, sobre todo cuando el copiado o trasladado del sub-árbol es grande.

MH-Diff tiene, según sus autores una solución heurística en el peor de los casos O (n^3), donde n es el número de nodos. MH-Diff tiene un tiempo de complejidad promedio de O ($n^2 \log n$) [23].

2.6.3 XML TREEDIFF

XML TreeDiff es un conjunto *Beans* (Anexo 1) de Java. A diferencia de LaDiff y MH-Diff; XMLTreeDiff considera que la operación de actualizar se expresa como las operaciones de insertar y eliminar.

El costo de XMLTreeDiff es por lo menos de O (n^2), donde n es el número de nodos del árbol. XMLTreeDiff utiliza un óptimo algoritmo árbol-diferenciación junto con un rápido procedimiento de comparación *(matching)* de sub-árbol [24].

2.6.4 DIFFX

El algoritmo DiffX sirve para la detección de cambios entre dos versiones de un documento XML. Los cambios identificados son reportados como un script con las operaciones de edición. El script, cuando es aplicado a la primera versión del documento XML, producirá la segunda versión. El objetivo es optimizar el tiempo de ejecución de *mapping* (mapear) de los nodos entre las dos versiones y minimizar el tamaño del *delta script*. Para lograr este objetivo un fragmento es aislado del árbol usando la técnica de *mapping*, con el fin de identificar iterativamente el árbol de mayor concordancia entre los fragmentos de representaciones de árboles, de las dos versiones del documento. La técnica de *mapping* es lo bastante robusta como para manejar las diferencias, tanto en la estructura y el contenido de los dos árboles. El *delta script* generado a partir del *mapping* reconoce las diferencias de los elementos y atributos del modelo de datos XML. La operación de actualizar es una combinación de insertar y eliminar. El tiempo de ejecución del algoritmo es O (n^2).

2.6.5 IBM'S XML DIFF & MERGE TOOL

El XML Diff & Merge Tool fue creado por IBM, pero sólo implementa las funcionalidades básicas. Es un programa en Java, que compara un documento XML base, con otro documento XML y no admite tres vías de fusión. La herramienta señala las diferencias de utilización de símbolos y color.

Kyriakos Komvoteas señala que XML Diff Merge Tool es incapaz de producir siempre un resultado correcto, porque los desarrolladores han hecho la inexacta suposición de que el nodo está libre de conflictos, si todos los hijos del mismo nodo no tienen los mismos conflictos [26][27].

2.6.6 3DM'S MATCHING ALGORITHM

La tesis de maestría de Lindholm acerca de tres-vías de fusión [15], describe la herramienta de fusión y diferenciación (3DM), para documentos XML, que contiene un algoritmo para árboles XML concordantes que puede ser considerado como un algoritmo de detección de cambios. El algoritmo sólo maneja árboles ordenados.

Lindholm publicó su trabajo sobre tres-vías de fusión para documentos XML en el 2004 [15]. Él señala que "la construcción de concordancia de manera eficaz y precisa, es de gran importancia en la aplicación de fusión en documentos XML sin un único elemento de identificación".

Las operaciones soportadas por el algoritmo de concordancia 3DM son borrar, insertar, actualizar, mover y copiar. Lindholm ha diseñado la concordancia heurística de nodos uno a uno entre T_1 y T_0, así como entre T_0 y T_2, donde T_0 es el documento XML original y T_1 y T_2 son las versiones editadas simultáneamente del documento XML.

Lindholm también aplica su herramienta 3DM en un entorno móvil, donde se ofrece una manera flexible para implementar una aplicación con las capacidades de fusión de documentos XML [16][19].

2.6.7 XYDIFF

XyDiff es una herramienta para diferenciar árboles XML ordenados. Se centra en la mejora de tiempo y la gestión de la memoria. Para ello, utiliza un algoritmo muy eficiente que identifica los grandes sub-árboles que no sufren cambios entre dos versiones del documento XML. XyDiff guarda los cambios realizados a documentos XML en un delta documento.

El algoritmo XyDiff calcula los valores Hash y las ponderaciones para cada nodo en los árboles XML. Esto es para que cada nodo del documento XML original tenga un identificador único, XID. El XidMap es una técnica de identificación utilizada que recoge la lista de todos los identificadores persistentes en el documento XML en el orden prefijo de los nodos. XyDiff utiliza el formato XyDelta[36], el cual es un único documento XML que contiene todos los *siguientes deltas completados*, para guardar los cambios detectados.

XyDiff luego compara las firmas de dos nodos y si estas son iguales, los nodos son concordantes. Una de las prioridades es aceptar los nodos hijos no concordantes de todos los nodos, donde los nodos con el mayor tamaño obtienen la más alta prioridad. El algoritmo intenta hacer que el árbol coincida lo mejor posible para propagar la concordancia a los respectivos padres de ambos nodos en comparación. Cuando hay más de un candidato potencial, utiliza una simple norma heurística para decidir cuál de ellos comparar.

XyDiff soporta las operaciones de insertar, eliminar, actualizar y mover y logra un tiempo de complejidad de O (n log n). Sin embargo, debido a las normas utilizadas, no puede garantizar un resultado óptimo y en algunos casos incluso discordancia en los sub-árboles [21].

2.6.8 VM TOOLS

VM Tools es una colección de herramientas de Java orientada a XML, creada por VM Systems y está disponible como código abierto. Contiene una herramienta *Diff and Patch* para generar automáticamente diferencias entre la generación de documentos XML, y poder aplicar esas diferencias a uno de los documentos originales. La herramienta se centra en el tamaño mínimo del documento resultante. La documentación de VM Tools disponibles en la actualidad no nos proporciona una clara descripción de las propiedades como el tipo de las operaciones soportadas y el tiempo de complejidad [28].

2.6.9 DIFFXML

DiffXML es creado por Adrian Mouat y forma parte de un conjunto de utilidades de XML Diff and Patch, que opera en la estructura jerárquica de los documentos XML. El algoritmo es de código abierto y tiene un formato de salida totalmente independiente. El formato de salida, especificado en el lenguaje delta actualizado (DUL), maneja las operaciones insertar, actualizar, borrar y mover [27][29].

2.6.10 KF-DIFF+

KF-Diff + es un algoritmo de detección de cambios para documentos XML que maneja tanto árboles XML desordenados como ordenados.

El algoritmo transforma la tradicional corrección Tree-to-Tree en la comparación de árboles etiquetados sin duplicar los caminos. KF-Diff + logra alta eficiencia con el tiempo de complejidad de O (n), al igual que XyDiff, donde n es el número total de nodos. KF-Diff + se encarga de las operaciones de insertar, eliminar y actualizar [21][30].

2.6.11 XML DIFF AND PATCH

Microsoft XML Diff and Patch, es un conjunto de herramientas capaz de comparar, diferencias y rutas de dos documentos XML. La herramienta de comparación no tiene en cuenta el orden de atributos, ignora espacios en blanco, considera <q/> lo mismo que <q> </q> y no se preocupan por el documento de codificación. Ofrece la opción de ignorar los comentarios XML e instrucciones de procesamiento y algunas opciones interesantes acerca de nombres [31].

2.6.12 X-DIFF

X-Diff puede detectar de manera óptima la diferencia entre dos árboles XML desordenados en tiempo polinomial. Song y Bhowmick extendieron el algoritmo para su uso en la detección en cambios en datos genómicos y proteómicos (BioDiff). X-Diff integra características claves en la estructura de XML con una técnica estándar Tree-to-Tree de corrección de árboles.

X-Diff utiliza el concepto de nodo firma junto con una concordancia entre árboles XML para encontrar el mínimo costo en la concordancia y generar un mínimo costo del delta script. El algoritmo sólo especifica las tres operaciones básicas; insertar, eliminar y actualizar.

X-Diff parece ser similar a KF Diff+, pero X-Diff no podrá satisfacer los necesidades de los usuarios, debido a que el filtrado de X-Diff no es eficiente, ya que sólo puede reducir el tamaño del árbol removiendo el segundo nivel equivalente del árbol [20][21].

2.6.13 DELTAXML

Robin La Fontaine (EDM Monsell ltd.) creador de DeltaXML, la cual es una herramienta comercial, capaz de comparar, fusionar sincronización de documentos XML. DeltaXML maneja tanto árboles XML ordenados como desordenados, aunque el máximo tamaño del árbol es 50 MB. Debe ser explícita la declaración del tipo de árbol, ordenado o desordenado que se utiliza. Soporta ambos tipos de fusión de partes (dos-vías y tres-vías) DeltaXML utiliza un algoritmo árbol-concordancia, que se ejecuta en tiempo lineal. Soporta las operaciones de insertar, eliminar y actualizar.

Cobéna concluye en su trabajo que DeltaXML parece ser la mejor opción como algoritmo de detección, porque "puede correr muy rápido y sus resultados se acercan al mínimo delta Script"[32][33][34].

2.6.14 TREEPATCH

TreePatch es un código abierto XML Diff and Patch Tool, diseñado y creado por Kyriakos Komvoteas, como objeto de su tesis de maestría. Komvoteas usando como base el algoritmo DiffXML creado por Adrian Mouat.

Este algoritmo es fácil de utilizar, pero presenta dos grandes inconvenientes: en primer lugar, es relativamente lento y, en segundo lugar, no puede manejar documentos de gran tamaño.

Por otra parte, TreePatch utiliza un algoritmo de árbol-diferenciando en combinación con un rápido sub-árbol que busca concordancias para alcanzar los resultados, reduciendo considerablemente el tiempo complejidad y el espacio. Sin embargo, documento delta resultante sólo es aplicable al archivo original con el fin de crear la versión actualizada [27][29].

2.6.15 BIODIFF

Basado en X-Diff, BioDiff es un algoritmo especialmente diseñado para detectar cambios en genómica y datos proteómicos específicos. Reduce el tamaño de la serie de datos biológicos y se centra en especial en la semántica de los elementos XML [35].

CUADRO RESUMEN DE LOS ALGORITMOS DE DETECCIÓN DE CAMBIOS Y LAS PROPIEDADES

	Referencia	Tiempo /Complejidad		Memoria	Operaciones	Tipo Árbol	Notas
La-Diff	[22]	Lineal	$O(ne+e^2)$	Lineal	Básicas, Mover	Ordenado	Para LATEX
MH-Diff	[23]	Cuadrática	$O(n^2+\text{Log } n)$	?	Básicas, Mover, Copiar	Desordenado	
XML TreeDiff	[24]	Cuadrática	$O(n^2)$	Cuadrática	Básicas	Ordenado	
DiffX	[25]	Cuadrática	$O(n^2)$	Cuadrática	Básicas	Ambos	
IBM's XML Diff & Merge Tool	[26]	?	?	?	Básicas	?	Herramienta Comercial
3DM's matching algorithm	[15][16][19]	Lineal	O(n)	?	Básicas, Mover	Ordenado	
XyDiff	[21]	Lineal	O(n Log n)	Lineal	Básicas, Mover	Ordenado	
VM Tools	[28]	?	?	?	?	Desordenado	
DiffXML	[29]	Lineal	$O(ne+e^2)$	Lineal	Básicas, Mover	Ordenado	
KF-Diff+	[30]	Lineal	O(n)	?	Básicas	Ambos	
XML Diff and Patch	[31]	?	?	?	?	Ambos	Herramienta Comercial
X-Diff	[20]	Cuadrática	$O(n^2)$	Cuadrática	Básicas	Desordenado	
Delta XML	[32][33]	Lineal	?	Lineal	Básicas	Ambos	
Tree Patch	[27]	Lineal	$O(ne+e^2)$	Lineal	Básicas, Mover	?	
BioDIFF	[36]	Cuadrática	$O(n^2)$	Cuadrática	Básicas	Desordenado	Para datos Geonómicos y Proteómicos

Se describen las operaciones de actualizar, insertar y eliminar como básicas porque todos los algoritmos soportan estas operaciones.

2.7 PROPIEDADES DE LOS ALGORITMOS

Cada algoritmo tiene una serie de ventajas, los algoritmos son optimizados para determinadas propiedades. Estas propiedades se explican a continuación.

2.7.1 ÁRBOL ORDENADO O DESORDENADO

Los árboles XML pueden dividirse en dos categorías, árboles ordenados o desordenados. En ambos tipos de árboles, la relación padre e hijos es importante, pero en árboles ordenados el orden de izquierda a derecha de los hermanos también es significativo.

Por lo general a las aplicaciones no les importa si los nodos en un árbol XML están ordenados o no. Las aplicaciones normalmente se preocupan por el contenido de los árboles XML y no del orden, a menos que, por ejemplo se trate de un documento XML que representa una página Web, entonces el algoritmo de detección de cambios tiene que manejar el documento como un árbol XML ordenado.

La diferencia para el algoritmo de detección de los cambios es que el algoritmo tiene que detectar un movimiento (cuando un nodo cambia de posición).

2.7.2 RENDIMIENTO Y COMPLEJIDAD

Para aplicaciones de bases de datos y de servicios dinámico, el buen rendimiento y bajo uso de memoria es necesario, sobre todo con grandes volúmenes de datos.

2.7.3 CORRECCIÓN

Podemos asumir que la mayoría de los algoritmos son correctos, que ellos descubren un conjunto de operaciones que pueden transformar la antigua versión en una nueva versión del documento XML. Algunos algoritmos pueden cotejar nodos que no deben ser comparados. El resultado final después de aplicar el delta script a la versión antigua es todavía la nueva versión, pero el delta del script no es óptimo. Este tipo de exclusión ocurre cuando un algoritmo aplica ciertos criterios tales como "los nodos son 90% similares, por lo que deben ser iguales", y el algoritmo detiene la búsqueda en una mejor concordancia. Esto es útil cuando el tiempo es un problema, pero degrada el sentido de los *cambios* en el delta script, debido a que múltiples nodos pueden ser excluidos.

2.7.4 USO DE MEMORIA

Un algoritmo puede mejorar el uso de memoria mediante la eliminación de nodos concordantes de los árboles XML. Estos nodos no necesitan más comparación, porque ya están igualados. Especialmente en grandes cantidades de datos, esto puede ser útil.

2.7.5 ENTRADA TAMAÑO

Algunos algoritmos han sido especialmente diseñados para manejar grandes cantidades de datos. Otro, como XMLTreeDiff utilizan el Document Object Model, (DOM) el cual limita el tamaño máximo de entrada del documento. Microsoft XML Diff and Patch limita el tamaño del documento a sólo 100KB, mientras que DeltaXML limita el tamaño máximo del documento a 50 MB.

2.7.6 SEMÁNTICA

La semántica de los datos XML se puede utilizar en algunos algoritmos de manera rápida y con correcta concordancia de los nodos. Algunos pueden considerar claves, definidas como ID en la DTD, y los nodos coinciden con alguna clave y *tagname* (nombre de etiqueta).

2.7.7 MÍNIMO DELTA SCRIPT

Mínimo u óptimo delta script es un factor importante en los algoritmos de detección de cambios. Si el algoritmo y el delta script son utilizados en una versión del sistema de gestión, el delta script necesita ser primero exacto; segundo tener un tamaño mínimo. Pero cuando el tamaño de transferencia delta script es importante, el algoritmo necesita encontrar un delta script mínimo.

El uso de las operaciones mover y copiar pueden tener un gran impacto sobre el tamaño del delta del script. La operación mover reduce el tamaño del documento, pero tiene un impacto sobre la complejidad y la ejecución del algoritmo, debido a que la operación mover se puede representar como una combinación de las operaciones insertar y eliminar, el delta script identificará sólo operación mover y no dos operaciones insertar y eliminar, es así como el tamaño del delta script se minimiza. La operación copiar tiene mayor impacto y, por lo tanto usa un algoritmo para la detección de cambios. De este modo, para disminuir la complejidad y la ejecución de los algoritmos es que la mayoría de ellos sólo soportan las operaciones básicas de actualizar, insertar y eliminar.

2.8 ALGORITMOS TENTATIVOS A IMPLEMENTAR

A continuación se explicará de forma más detallada, aquellos algoritmos que tanto por su tiempo (orden), funcionamiento y disponibilidad de información, serian considerados para una futura implementación.

2.8.1 X-DIFF

Con el fin de diseñar un algoritmo eficiente para detectar los cambios en documentos XML, es necesario entender la estructura jerárquica en XML. Basado en el Document Object Model (DOM) sacado, en donde un documento XML puede representarse como un árbol.

Este documento analiza tres tipos de nodos en árboles DOM:

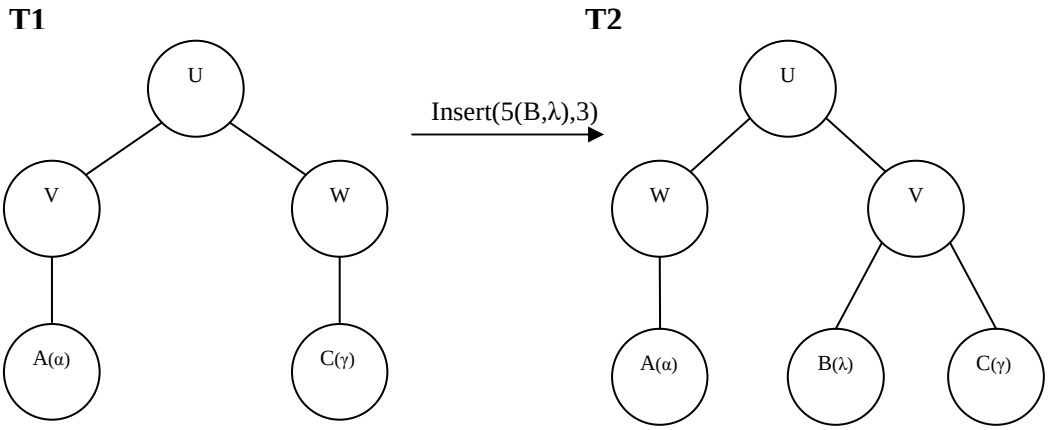
- Nodo Elemento - nodos no hoja con una etiqueta, *name*.
- Nodo Texto - nodos hoja con una etiqueta, *value*.
- Nodo Atributo - nodos hoja con dos etiquetas, *name* y *value*.

De acuerdo con la especificación DOM, los nodos elemento y nodos texto están ordenados, mientras que los nodos atributo están desordenados. En muchas aplicaciones, los documentos XML pueden ser tratados como árboles desordenados, mientras que el orden de izquierda a derecha entre hermanos no sea significativo.

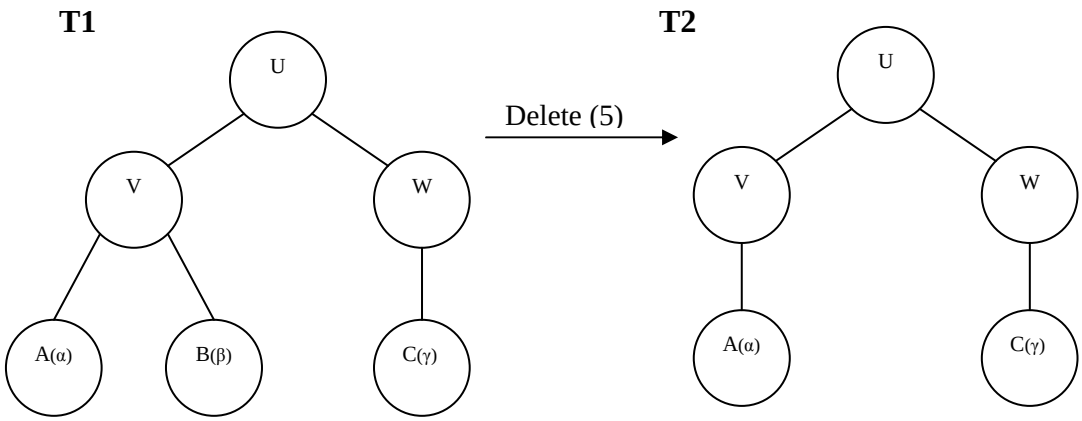
En X-Diff (Anexo 2), la detección de cambios se centra en los árboles desordenados. La mayoría de los criterios de corrección para un árbol ordenado no se puede aplicar a árboles desordenados porque su exactitud en general depende de la preservación del orden de izquierda a derecha para cuando se comparen los nodos. Dos árboles se denominan *isomórficos* si son idénticos, excepto por el orden de los hermanos. X-Diff considera que dos los árboles son equivalentes si son *isomórficos* [20].

OPERACIONES DE EDICION

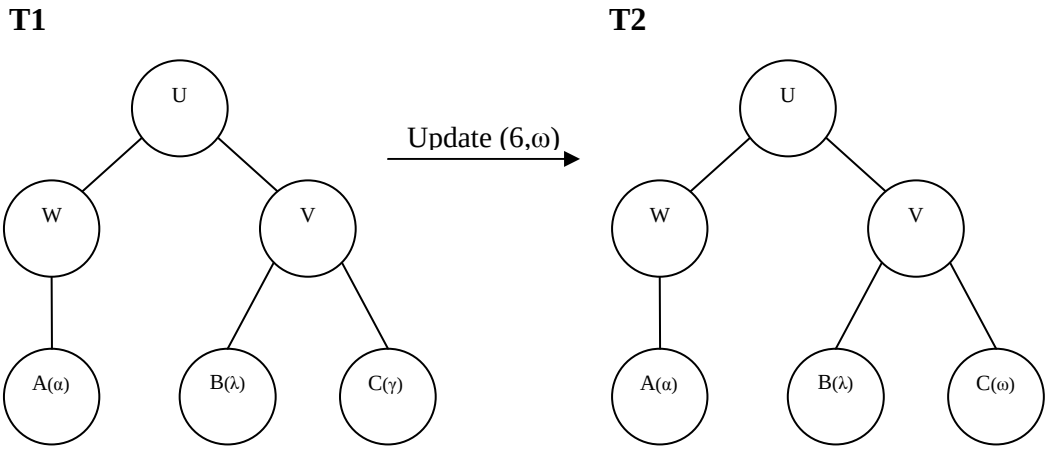
- **Insert (*x*(*name, value*), *y*):**Inserta un nodo *x*, con nodo nombre "name" y nodo valor "value", como un nodo hoja hijo del nodo *y*.



- **Delete (*x*):** Elimina un nodo hoja *x*.



- **Update (*x*, *new_value*):** Cambia el valor de un nodo hoja *x* a *new_value*. Cabe señalar, que *x* tiene que ser un texto o un atributo del nodo. Update sólo puede modificar un valor del nodo, pero no su nombre.



ESQUEMA DEL ALGORITMO X-DIFF

Dado dos documentos XML, DOC_1 y DOC_2, T_1 y T_2 son sus representaciones de árboles. X-Diff determina si DOC_2 es diferente de DOC_1 basado en el modelo desordenado. Si son diferentes, X-Diff encuentra el mínimo costo de concordancia de T_1 a T_2, y genera un óptimo delta script de ($T_1 \rightarrow T_2$) [20]:

- **Parsing y Hashing:** X-Diff analiza DOC_1 y DOC_2 dentro de XTrees en T_1 y T_2. Durante el proceso de análisis, X-Diff calculará un valor XHash para cada nodo, que se utiliza para representar todo el sub-árbol enraizado al nodo.

- **Matching:** En primer lugar, X-diff compara los valores de XHash de Root(T_1) y Root(T_2). T_1 y T_2 se consideran equivalente si dos valores XHash son iguales, de lo contrario, X-Diff encuentra $M_{min}(T_1, T_2)$, con un mínimo costo de concordancia entre dos árboles.

- **Generación del optimo delta script:** X-diff genera un optimo delta script E para ($T_1 \rightarrow T_2$), basado en el M_{min} (T_1, T_2) que se encuentran en el paso 2.

2.8.2 LADIFF

La principal tarea del algoritmo LaDiff (Anexo 2) es la búsqueda de cambios para determinar los nodos que en ambos árboles son correspondientes entre sí, intuitivamente estos son los nodos que permanecer sin cambios.

La noción de una correspondencia entre los nodos que tienen idénticos o similares valores, se formaliza como un *matching* entre nodos. Se considera que un *matching* es *parcial*, si sólo algunos nodos en ambos árboles son concordantes, mientras que un *matching total* es cuando todos los nodos de ambos árboles son concordantes.

Este algoritmo en su forma más simple se reduce a 5 fases las cuales son [22]:

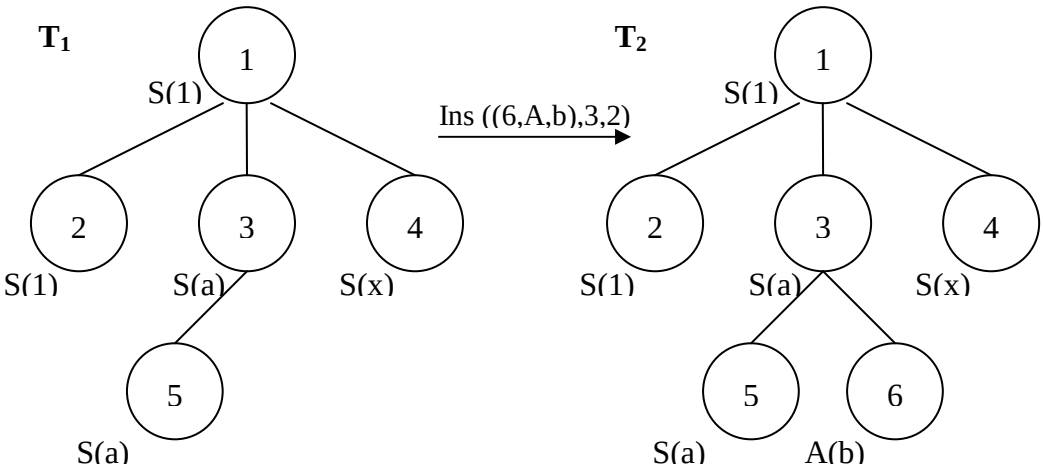
- Fase de inserción
- Fase de eliminación
- Fase de mover
- Fase de actualización
- Fase de alineamiento

Donde la fase de alineamiento (aling) no es mas de una secuencia operaciones de movimiento (move) que alinean los hijos de un nodo en particular siempre con la intención de realizar el mínimo de movimientos.

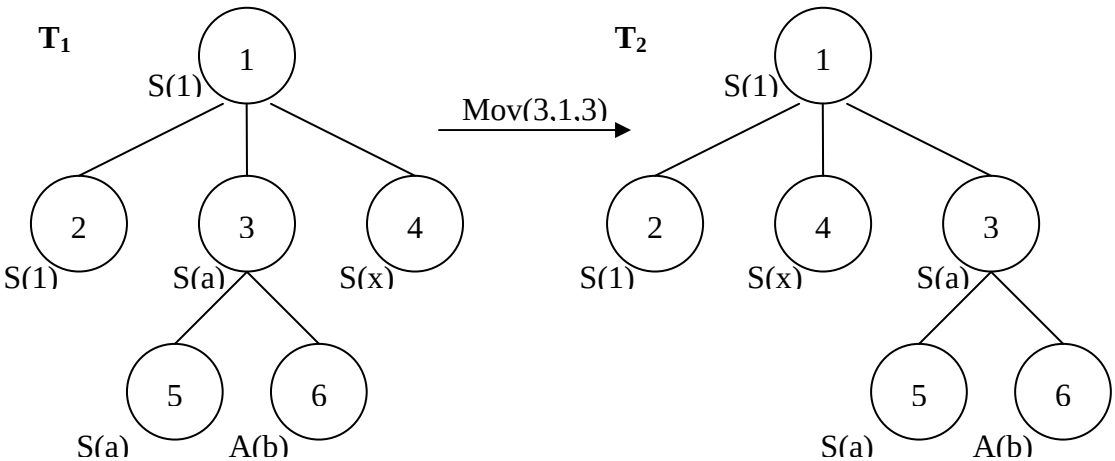
OPERACIONES DE EDICIÓN

- **Ins ((x; l; v), y; k):** La inserción de un nuevo nodo hoja *x* en T_1, indicado por *ins ((x; l; v), y; k).* Donde el nodo *x* con etiqueta *l* y valor *v* se inserta como el k-ésimo hijo de nodo y de T_1.

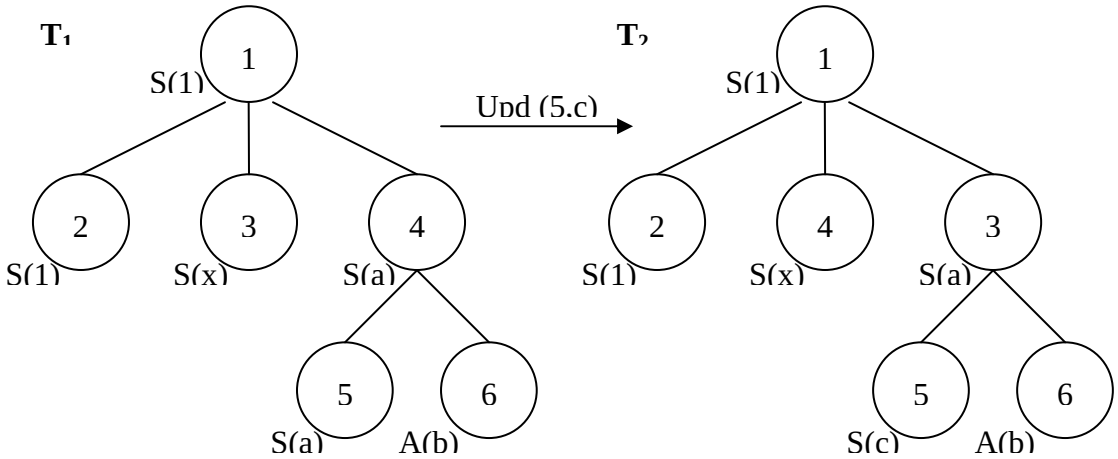
  Es decir si $u_1,\ldots, u_m$ son los hijos de *y* en T_1, entonces $1 \leq k \leq m + 1$ y $u_1,\ldots, u_{k-1}$; x; $u_k,\ldots, u_m$ son los hijos de *y* en T_2. El valor *v* es opcional, y se supone que es Null cuando es omitido.



- **Mov (x, y, k):** El desplazamiento de un sub-árbol de una posición a otra en T_1, es indicados por *mov (x, y; k).* T_2 es similar a T_1, excepto que *x* se convierte en el k-ésimo hijo de *y*. Todo el sub-árbol enraizado en *x* se mueve junto con esté.

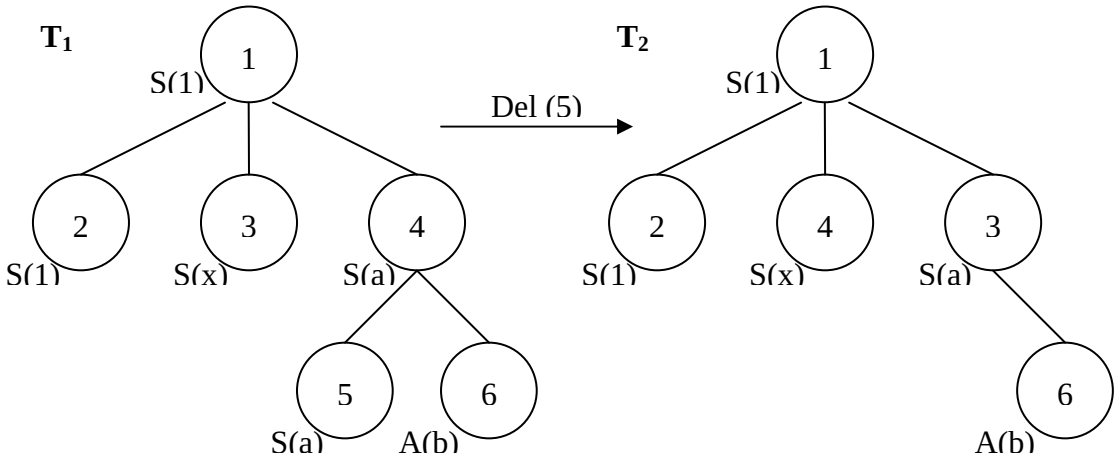


- **Upd (x, val):** La actualización del valor de un nodo *x* en T_1, es indicado por *upd (x, val)*. T_2 es similar a T_1, excepto que en T_2, el valor de *x* es *val*.



- **Del(x):** La eliminación de un nodo hoja *x* de T_1, es indicado por *Del(x)*. El resultado de T_2 es el mismo que T_1, excepto que no contiene nodo *x*.

Del(x) no cambia el orden de los hijos restantes. Esta operación sólo elimina un nodo hoja; para eliminar un nodo padre, primero, debemos pasar sus descendientes a nuevas posiciones o borrarlos.



ESQUEMA DEL ALGORITMO LADIFF

Ahora presentamos el algoritmo completo para obtener el optimo delta script *E*, conforme a un determinado matching *M*, entre los árboles T_1 y T_2. En el algoritmo, combinamos las cuatro operaciones (actualizar, insertar, alinear y mover) en un recorrido sobre T_2.

La fase de eliminación requiere un recorrido post-order de T_1 (que visita a cada nodo después de visitar a todos sus hijos). El orden en que los nodos son visitados y las operaciones de edición que son generadas resultan fundamental para el correcto desempeño del algoritmo (por ejemplo, la operación insertar puede necesitar antes una operación mover, si el nodo movido se convierte en el hijo del nodo insertado). El algoritmo realiza las operaciones de edición a T_1 y es añadido al delta script *E*. Cuando el algoritmo termina, T_1 es isomórfico a T_2.

El algoritmo también utiliza un matching *M'* que es inicialmente *M*, que añade coincidencias a este, a fin que M' contenga el matching resultante, una vez terminado el algoritmo. Sin olvidar asumir que las raíces de T_1 y T_2 son concordantes en *M*.

El algoritmo utiliza dos procedimientos, AlignChildren y FindPos, las llamadas hechas en el algoritmo DeltaScript están denotadas por (*) [22].

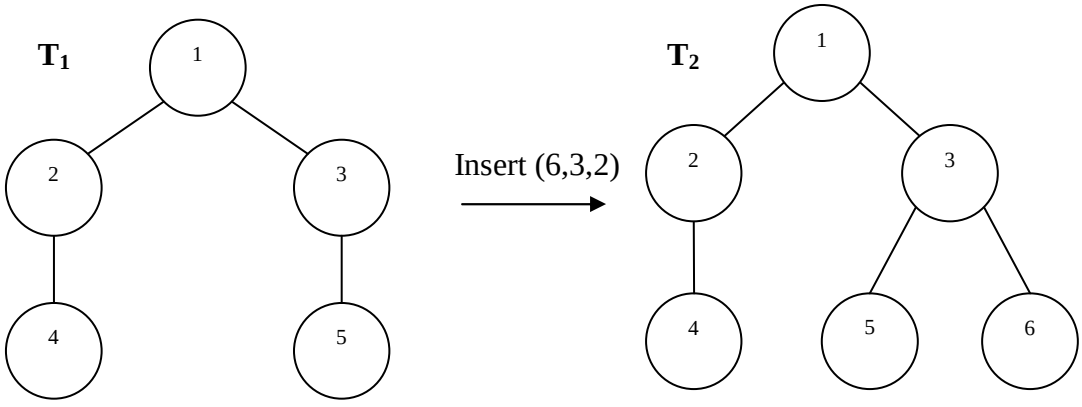
2.8.3 DIFFX

El modelo interno de datos para el algoritmo DiffX (Anexo 2) es el modelo de árbol para documentos XML. El nodo del árbol es etiquetado y tipiado. Los tres principales tipos de nodos son: elemento, atributo y texto. Dos nodos del mismo árbol o de dos árboles diferentes son considerados iguales, si ambos tienen tipos y valores iguales. Un nodo identificador únicamente identifica cada nodo a través de ambos árboles. Cada nodo conoce a su padre, a sus hijos por su posición o su propia posición entre los hermanos.

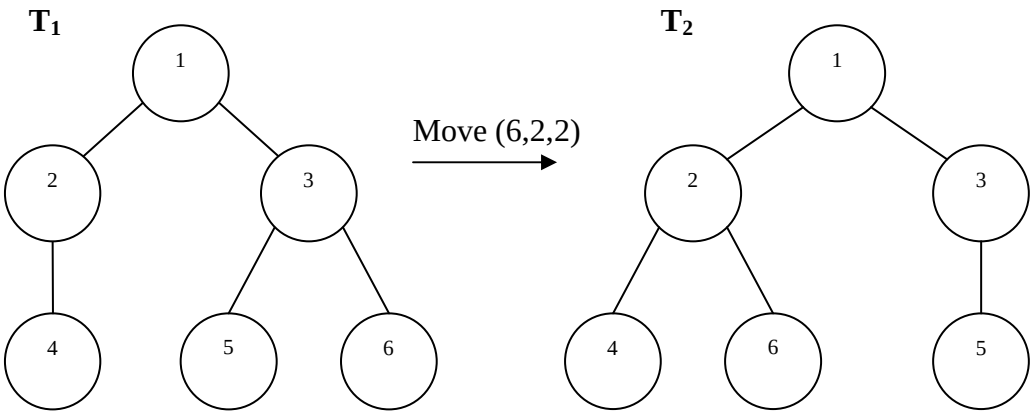
El siguiente paso es la generación de una secuencia de operaciones de edición desde el mapping (mapeo), que cuando se aplica a T_1 producirá T_2. La idea general es eliminar todos los nodos unmatched (no concordantes) desde T_1, insertando todos los nodos en T_2 y moviendo todos los nodos matching (concordantes) con padres unmatching o posición unmatching, a la posición en T_2. Las operaciones básicas para el delta script son: insertar, mover y eliminar [25].

OPERACIONES DE EDICIÓN

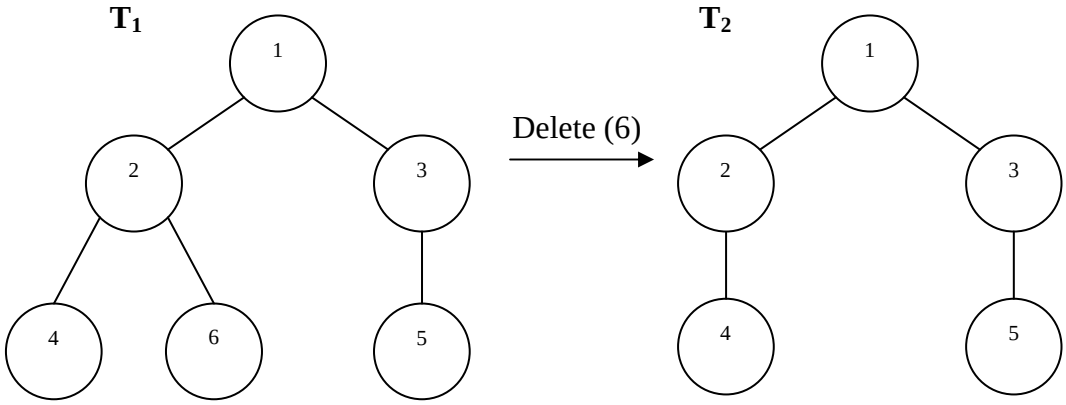
- **Insert (*x, y, p*):** Inserta el nodo *x* como el *p*-ésimo hijo del nodo *y*.



- **Move (*x*, *y*, *p*):** Mueve el sub-árbol enraizado del nodo *x* como el *p*-ésimo hijo del nodo *y*.



- **Delete (*x*):** Elimina el sub-árbol enraizado del nodo *x*.



ESQUEMA DEL ALGORITMO DIFFX

El algoritmo trabaja de la siguiente manera. En el top-down de nivel ordenado transversal de T_2, si el nodo actual es unmatched (no concordante) agrega una operación "*insert*" para este en el script. Si el nodo actual en T_2 encuentra un nodo en T1 con padres unmatched o con posición unmatched, agrega una operación "*move*" en el script. El nivel ordenado transversal, asegura que el padre de un nodo esta en el lugar correcto antes de "*insert*" o "move" un nodo hijo. Después de procesar todos los nodos matching (concordantes), los nodos unmatched en T1 serán localizados al fondo del árbol. Finalmente en un top-down de nivel ordenado transversal de T1, será agregada una operación "*delete*" para todos los nodos enraizados de un nodo unmatched, de esta forma se completará el delta script [25].

2.9 CONCLUSIONES PROYECTO DE TÍTULO

El objetivo general del Proyecto de Título era: Analizar los algoritmos existentes y diseñar el algoritmo(s) tentativo a desarrollar para la detección de cambios en documentos XML. Lo cual se logro con éxito en el Proyecto de Título. El algoritmo será explicado en detalle en el capítulo 3 de la Habilitación Profesional. Finalmente se puede obtener la siguiente conclusión:

En este proyecto, se estudió y analizó el funcionamiento de los algoritmos existentes para detección de cambios en documentos XML, para lo cual se entrega una descripción general de cada uno de ellos, indicando el tiempo de complejidad, los tipos de operaciones que realiza, así como el tipo de árbol con el que trabaja.

Posteriormente, se seleccionó aquellos algoritmos que resultaron más atractivos en su desempeño, describiendo las funciones que lo componen, y a su vez explicando y demostrando las operaciones que realizan.

A continuación del trabajo hecho anteriormente podemos responder a las tareas que se enmarcaron en los objetivos específicos, los cuales se detallan a continuación junto con los resultados y/o conclusiones para cada uno de ellos:

a) Recopilar información de los diferentes algoritmos utilizados para la detección de cambios en documentos XML, que nos permita obtener todos los datos necesarios para tener una visión amplia del tema a tratar.

Se han descrito algunos algoritmos diseñados específicamente para la detección de cambios en documentos XML, y otros algoritmos en los que la detección de cambios es sólo una parte del algoritmo. Donde existen claras relaciones entre algunos algoritmos, por ejemplo BioDiff es una extensión del algoritmo X-Diff y el algoritmo DiffXML es una mejora del algoritmo TreePatch, entre otros.

b) Realizar un análisis a priori de los algoritmos utilizados para la detección de cambios en documentos XML.

Se consideró para el análisis, el orden de complejidad de cada algoritmo presente en su respectivo paper (ver referencias), además se realizó un cuadro resumen con los aspectos más relevantes, como por ejemplo las operaciones que ocupaban en su desempeño. Véase “Cuadro Resumen de los Algoritmos de Detección de Cambios”.

c) Determinar posibles algoritmos que nos permitan obtener soluciones tentativas para la detección de cambios en documentos XML.

Se seleccionaron los algoritmos LaDiff, DiffX y X-Diff. LaDiff debido a que tiene un menor tiempo de complejidad de $O(ne+e^2)$ y además a sido base para el desarrollo de algunos de los algoritmos existentes para la detección de cambios en documentos XML. DiffX debido a la diversidad de operaciones que no se contemplaban en los otros algoritmos. X-Diff debido a que trabaja con la información de los nodos usando la función Hash.

Se ha considerado la inclusión de la función Hash, donde esta función de costos transforma los nodos de cada árbol en un valor numérico. Por otra parte, el valor numérico es asignado a cada al nodo, donde el valor Hash de cada nodo hijo, es pasado a su respectivo padre y así sucesivamente hasta llegar a la raíz del árbol, para así poder realizar un matching observando las raíces de los árboles y comparar el valor representativo de las estructuras T1 y T2. Para así poder determinar si son iguales, de lo contrario busca en cual sub-árbol esta presente el cambio, y así sucesivamente hasta dar con el nodo cambiado. Evitando con esto recorrer todos los nodos del árbol.

CAPÍTULO 3: *ALGORITMO A DESARROLLAR.*

En este Capítulo se presenta un nuevo algoritmo para detectar isomorfismo en árboles, el cual se diseñó y analizó su eficiencia en el Proyecto de Título, donde se muestra que el problema de árboles isomórficos puede ser resuelto de manera eficiente usando información de los nodos. La eficiencia del algoritmo reside en la detección de isomorfismo en los sub-árboles. La complejidad teórica del algoritmo es $O(n \log_m n)$, donde *n* es el número de vértices y *m* es el mayor número de sub-árboles que posee la raíz.

Este Capítulo está estructurado de la siguiente manera:

- Sección 1: Algoritmo a desarrollar
  - ✓ Esquema del Algoritmo a desarrollar
- Sección 2: Funcionamiento del algoritmo (representación grafica)
  - ✓ Algoritmo Original (T1)
  - ✓ Algoritmo Modificado (T2)
  - ✓ Matching

3.1 ALGORITMO A DESARROLLAR

En este trabajo, proponemos un nuevo algoritmo para detectar árboles isomórficos. Por otra parte, mostraremos que la determinación de árboles isomórficos es un tema que puede resolverse de manera eficiente utilizando la información de los nodos (Algoritmos eficientes que trabajan en la detección de sub-árboles isomórficos) [20][37].

El algoritmo esta implementado para trabajar con árboles desordenados, a través de la función *ordenarArbol*, la que realiza el ordenamiento de los nodos con la información que contienen; en el caso que sean árboles ordenados se omite esta función. Luego, a cada nodo hijo se le asigna un valor Hash, el cual es traspasado a su padre y así sucesivamente hasta la raíz, de tal manera que es posible saber si hubo algún cambio en los nodos inferiores de una rama determinada. A continuación, se hace un matching, el algoritmo verifica si los árboles son isomórficos; en caso que lo sean, entonces no es necesario comprobar las ramas de los árboles para buscar las modificaciones.

Nuestro algoritmo en el peor de los casos es $O(n \log_m n)$, donde *n* es el número de vértices y *m* es el mayor número de sub-árboles que posee la raíz.

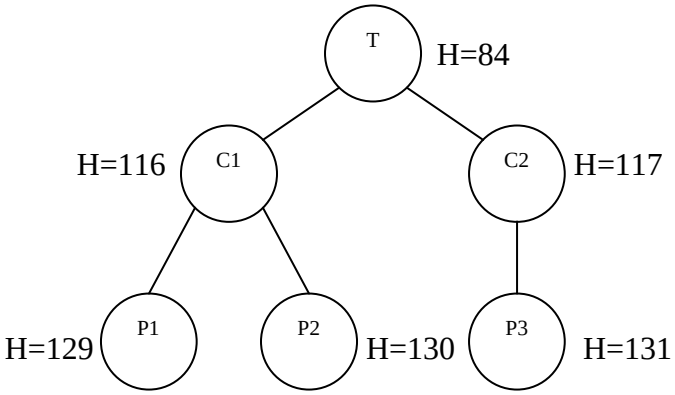
3.1.1 ESQUEMA DEL ALGORITMO A DESARROLLAR

Principalmente el algoritmo se compone de tres partes. En primer lugar, los árboles son ordenados por la función *ordenarArbol*. Como segundo paso, los valores Hash de los sub-árboles son obtenidos y con éstos el valor Hash del árbol. Y el tercer paso, es la comparación del Hash de los árboles mediante la función *matching*, en caso de ser iguales significa que los árboles son isomórficos.

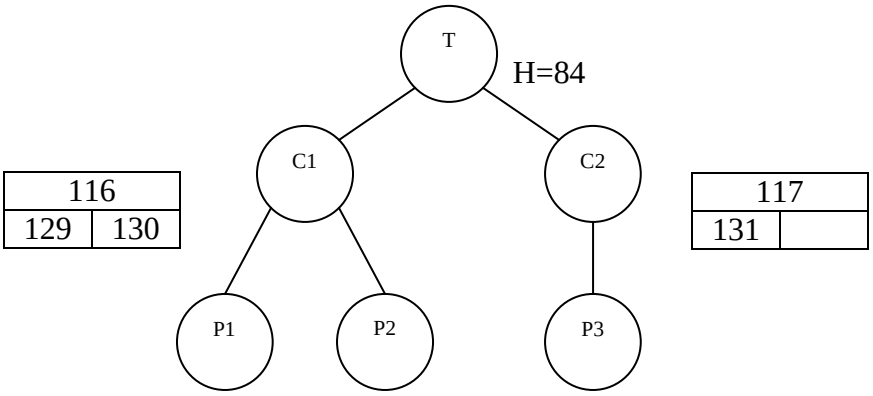
3.2 FUNCIONAMIENTO DEL ALGORITMO (REPRESENTACIÓN GRAFICA)

3.2.1 ÁRBOL ORIGINAL (T_1)

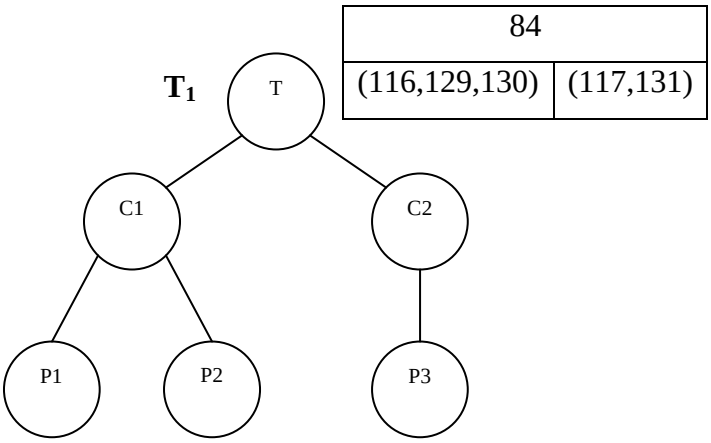
Se aplica la función *hashing* a cada nodo, el cual entrega un valor Hash para cada uno de ellos, obtenidos al transformar el contenido de los nodos a código ASCII (Anexo 1).



El valor representativo se trabaja en una estructura la cual contiene el valor Hash del nodo y de cada nodo hijo.

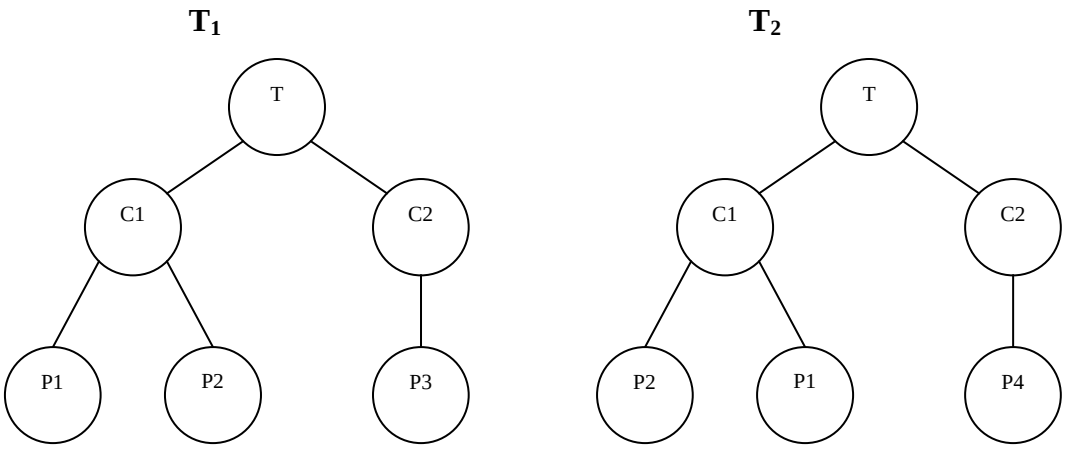


El valor Hash de cada nodo hijo, es pasado a su respectivo padre y así sucesivamente hasta la raíz del árbol.

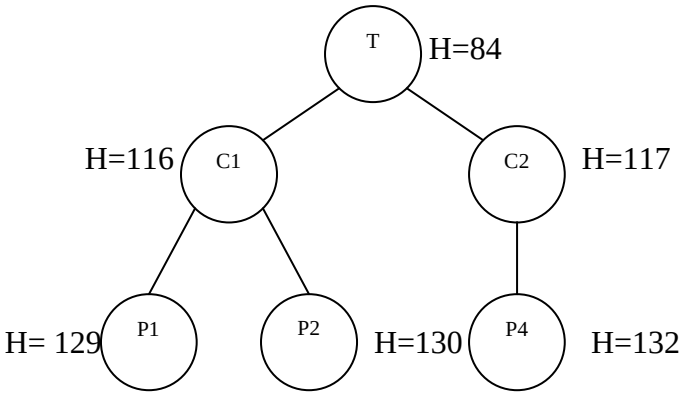


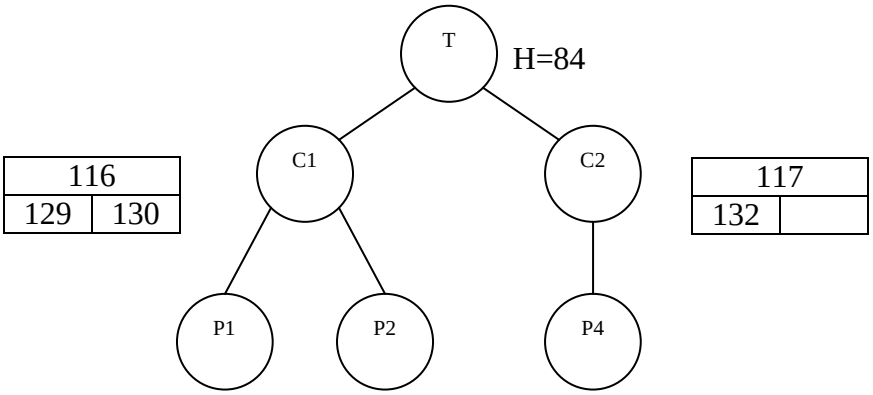
3.2.2 ÁRBOL MODIFICADO (T_2)

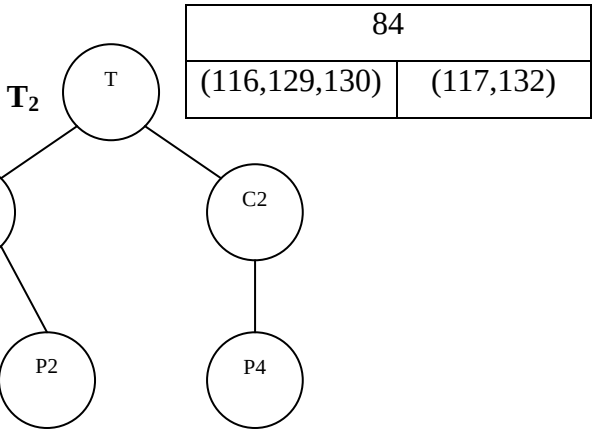
Función *ordenarArbol*, ordena el árbol modificado en función del original.



Se aplica la función Hash a cada nodo, el cual entrega una codificación para cada uno de estos.

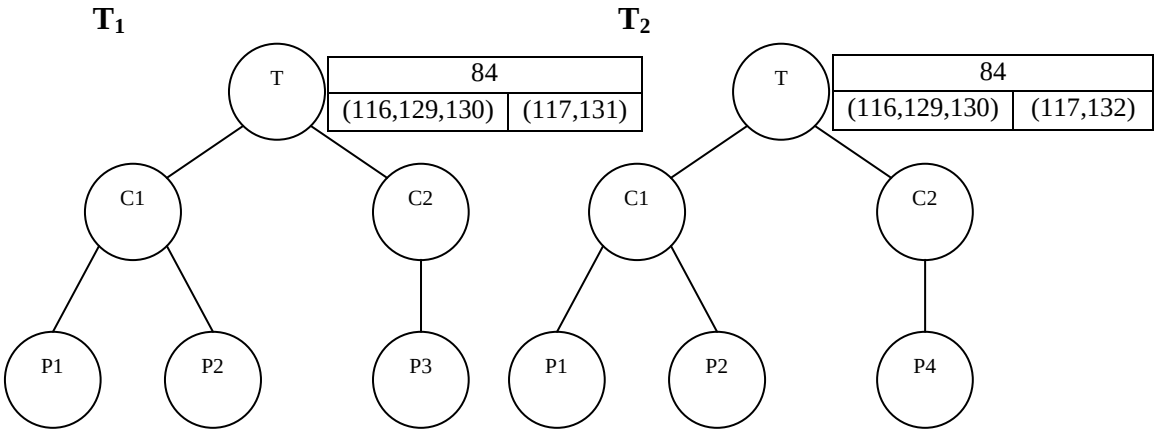






3.2.3 MATCHING

Se aplica la función *matching* a T_1 y T_2.



Se compara el valor representativo de la estructura de T_1 y T_2, determinando si hay matching (son iguales), de lo contrario se compara el valor de cada sub-árbol hasta encontrar donde fue la modificación. La estructura existente en cada nodo, contiene el valor Hash de si mismo y el valor de cada uno de sus nodos hijos, de esta forma es posible encontrar donde ocurrió el cambio. En la representación anterior de T1 y T2, el valor Hash total de T_1 es 707, y el de T_2 es 708. El cambio ocurrió en el nodo hijo de C_2, donde en T_1 era P_3 y se modificó por P_4, con esto el valor Hash del nodo se actualiza, al igual que el valor del padre y así hasta llegar a la raíz. Es así como la cantidad de visitas que en total se hacen al árbol son tres, para encontrar donde hubo modificación. El orden de ambos árboles es de: $6 \log_2 6$ en donde, *6* es el número de nodos y *2* es el mayor número de sub-árboles que posee la raíz.

CAPÍTULO 4: *EVALUACIÓN DEL DESEMPEÑO DE LOS ALGORITMOS*.

En este Capítulo se examina el desempeño del algoritmo propuesto, el cual se compara con el desempeño del algoritmo X-Diff, señalando algunos resultados preliminares sobre su tiempo de ejecución y resultados generales.

Este Capítulo está estructurado de la siguiente manera.

- Sección 1: Escenarios de prueba.
  - ✓ Parámetros Experimentales y Datos de Prueba.

- Sección 2: Resultados Empíricos.
  - ✓ Tiempo de Árboles Ordenados.
  - ✓ Tiempo de Árboles Desordenados.

- Sección 3: Resultados generales de las pruebas empíricas de los Algoritmos.

4.1 ESCENARIOS DE PRUEBA

En el siguiente punto se dan a conocer el o los algoritmos existentes que fueron utilizados para realizar las pruebas, señalando las características que nos llevaron a su elección.

4.1.1 PARÁMETROS EXPERIMENTALES Y DATOS DE PRUEBA

Considerando los algoritmos existentes, se ha optado por implementar el algoritmo X-Diff , debido a que es uno de los únicos que trabaja con la información que contienen los nodos, lo cual resulta conveniente a la hora de detectar si hubo o no cambios sin necesidad de recorrer todo el árbol, sólo lo recorre cuando hay cambios. No así DiffX, debido a que exista o no cambios en el árbol, el algoritmo lo recorre completamente. Por otra parte, el algoritmo LaDiff si bien presenta un tiempo menor, su función es detectar cambios en documentos Latex, y si se consideran los algoritmos basados en LaDiff, éstos presentan un tiempo de complejidad mayor a $O(n^2)$. Véase Capítulo 2. No se consideraron los algoritmos de O(n), debido a que no existe una descripción detallada de su funcionamiento y sólo existen las versiones ejecutables de ellos. Por ende, no es posible obtener experimentos que realmente reflejen su desempeño en el tiempo considerando. Más aún, queda como duda si realmente el algoritmo se comporta de manera lineal en el tiempo, sin olvidar que gran parte de estos algoritmos se restringen a un sólo tipo de árbol, ordenado o desordenado.

El algoritmo propuesto (Anexo 3) está implementado en C, al igual que X-Diff (Anexo 3). Ambos algoritmos leen dos versiones de un documento XML (el documento original y el modificado) y genera la diferencia. Todos los siguientes experimentos se realizaron sobre una Laptop, con procesador Pentium (R) Dual Core 1.73 GHz con 3 Gb de memoria Ram. El sistema operativo Microsoft Windows XP Professional.

El conjunto de datos corresponde a un catalogo de venta, cuya DTD (Definición del Tipo de Documento) se muestra en el Ejemplo 1. El tamaño de los documentos utilizados en nuestro experimento oscila entre los 4 kb a 40 kb los cuales poseen una determinada cantidad de nodos que varían en 50, 100, 150, 200, 250, 500 y 1000, 3000, 5000, 10000,15000 nodos, a los cuales se les aplicó un porcentaje de cambios de 10%, 15%, 20%, 25% y 30%, respectivamente. Los tres tipos de cambios: "insertar", "borrar" y "actualizar"; son realizados al azar en cada nivel.

Ejemplo 1. DTD Catalogo de Ventas:

```
<!ELEMENT Catalogo ( Ropa )
<!ELEMENT Ropa ( Hombre, Mujer )
<!ELEMENT Hombre ( Pr )
<!ELEMENT Mujer ( Pr )
<!ELEMENT Pr ( N, V )
<!ELEMENT N (#PCDATA) >
<!ELEMENT V (#PCDATA) >
```

En el caso de árboles ordenados el algoritmo propuesto utiliza el modelo *ordenado*, el cual proporciona un ahorro el tiempo al evitar reordenar un árbol que ya esta ordenado, con el fin de proporcionar una comparación ecuánime. Los cambios realizados en el documento XML no permutan el orden de los nodos, en ambos algoritmos.

4.2 RESULTADOS EMPÍRICOS

En este punto los resultados se han clasificado según el tipo de árbol, así como en cada figura se representa los tiempos de respuesta medidos en milisegundos (Ms) de cada algoritmo en relación a su porcentaje de cambios, tanto para árboles ordenados como desordenados.

4.2.1 TIEMPO DE ÁRBOLES ORDENADOS

En primer lugar, evaluamos el tiempo de ejecución de ambos algoritmos sobre documentos de diferentes tamaños. En las Figuras 4.1, 4.2, 4.3, 4.4 y 4.5 la cantidad de nodos se han modificado en 10%, 15%, 20%, 25% y 30% respectivamente, sin alterar el orden de los nodos del árbol.

A continuación se aprecia que X-Diff funciona relativamente bien en documentos de pequeño y mediano tamaño (hasta 250 nodos). Sin embargo, debido a la complejidad del algoritmo, su tiempo de ejecución es bastante mayor cuando los dos archivos de entrada son grandes. Por otra parte, el algoritmo propuesto es más eficiente observando que el total de los tiempos de ejecución arrojan una curva del tipo logarítmica, mostrando su real funcionamiento cuando la cantidad de nodos es superior a 250 nodos.

Gráficos Árboles Ordenados

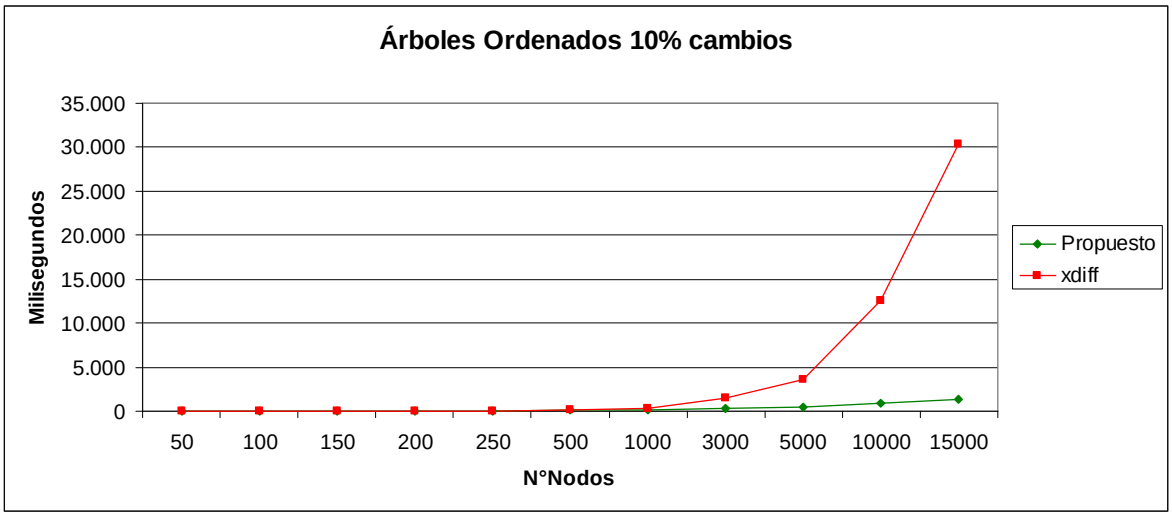


Figura 4.1. Árboles Ordenados con 10% de cambios.

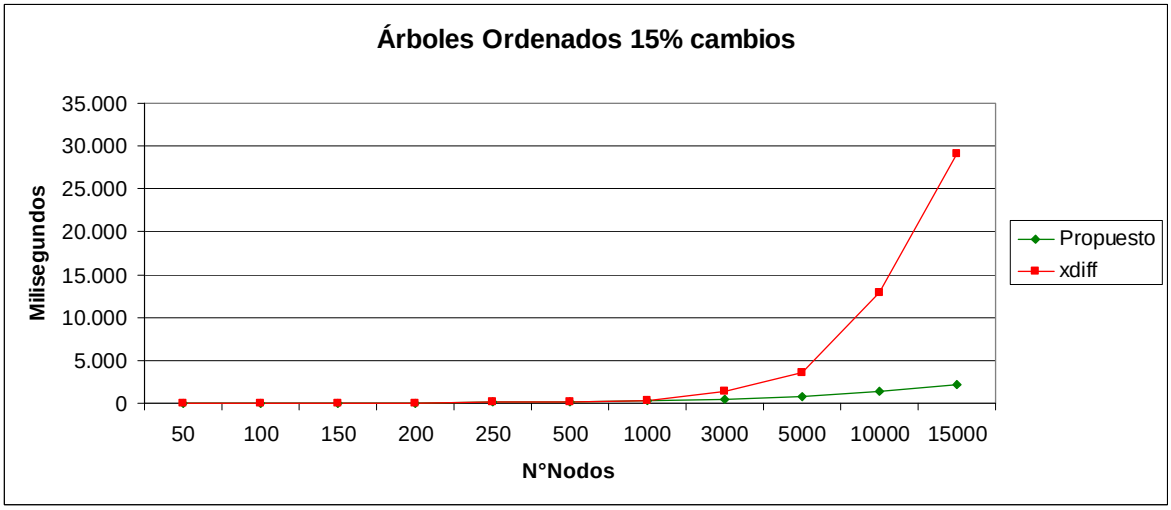


Figura 4.2. Árboles Ordenados con 15% de cambios.

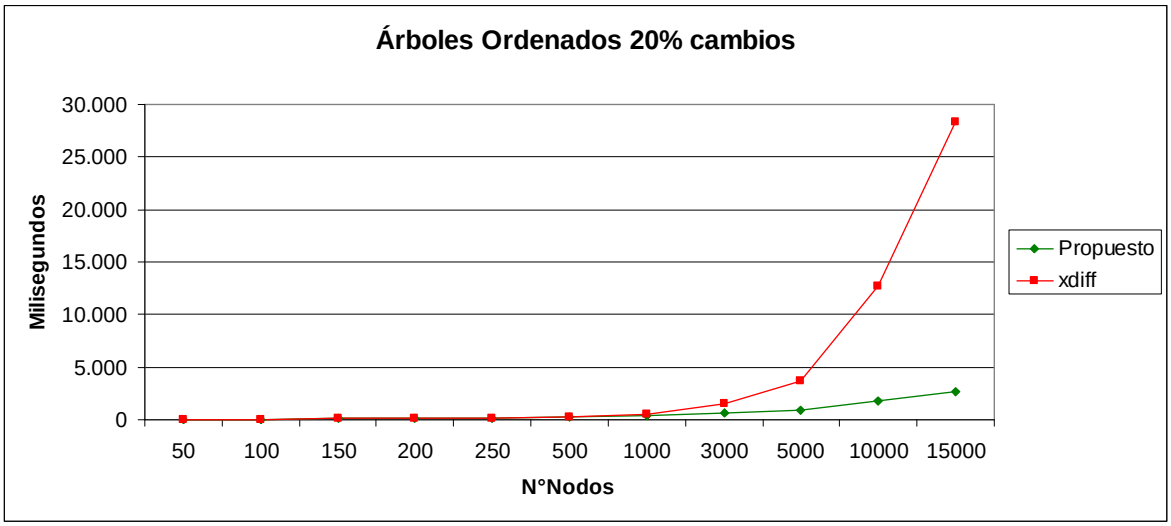


Figura 4.3. Árboles Ordenados con 20% de cambios.

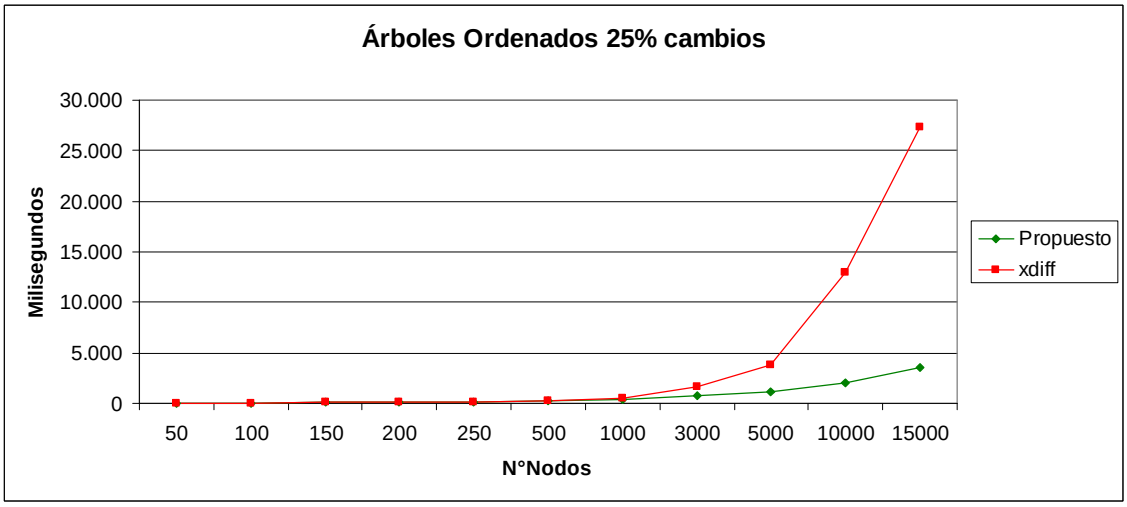


Figura 4.4. Árboles Ordenados con 25% de cambios.

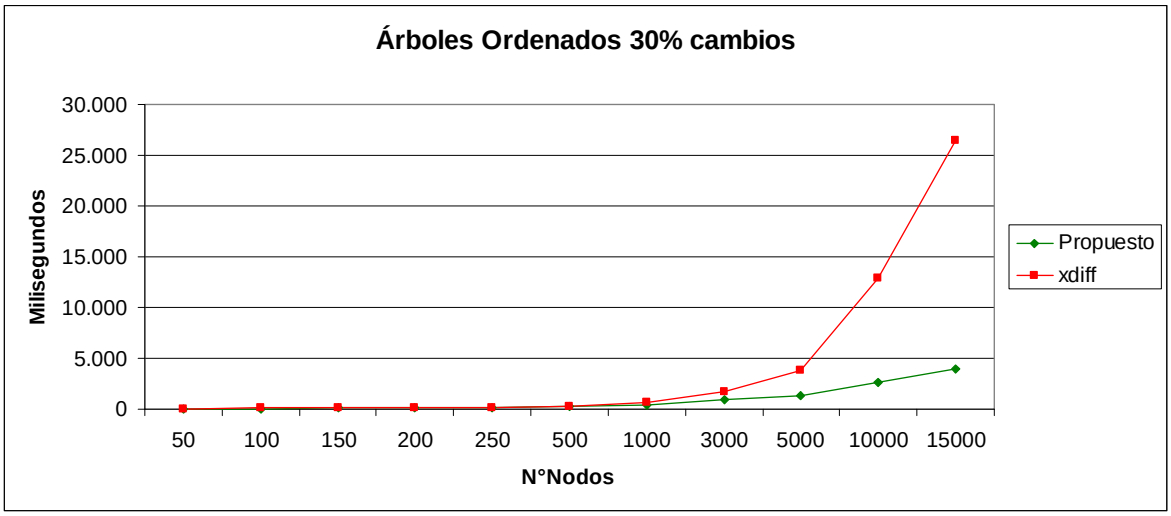


Figura 4.5. Árboles Ordenados con 30% de cambios..

4.2.2 TIEMPO DE ÁRBOLES DESORDENADOS

En primer lugar, evaluamos el tiempo de ejecución de ambos algoritmos sobre documentos de diferentes tamaños. En las Figuras 4.6, 4.7, 4.8, 4.9 y 4.10 la cantidad de nodos se han modificado en 10%, 15%, 20%, 25% y 30% respectivamente, alterando el orden de los nodos del árbol.

A continuación se aprecia (al igual que en caso de los árboles ordenados) que X-Diff funciona relativamente bien en documentos de tamaño pequeño (hasta 250 nodos). Sin embargo, debido a la complejidad del algoritmo, su tiempo de ejecución es bastante mayor cuando los dos archivos de entrada son grandes. Por otra parte, el algoritmo propuesto es más eficiente, observando que el total de los tiempos de ejecución arrojan una curva del tipo logarítmica, por lo que la tendencia sigue siendo similar al de los árboles ordenados pero el cambio notorio esta en los tiempos, ahora X-Diff aumenta sus tiempos de manera considerable. Mientras que el algoritmo propuesto si bien aumenta sus tiempos, lo hace en un porcentaje muy menor, casi despreciable.

GRÁFICOS ÁRBOLES DESORDENADOS

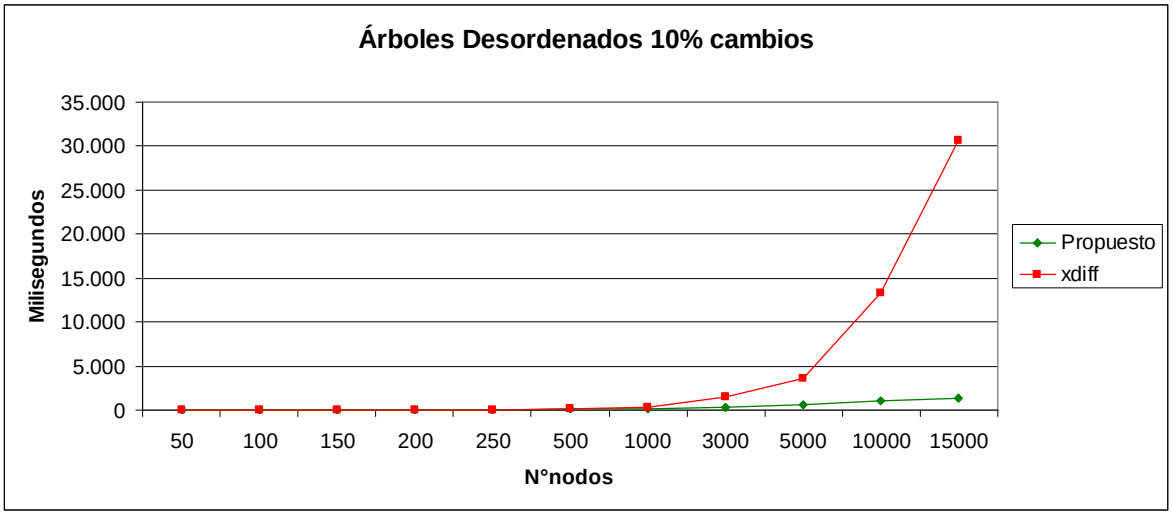


Figura 4.6. Árboles Desordenados con 10% cambios.

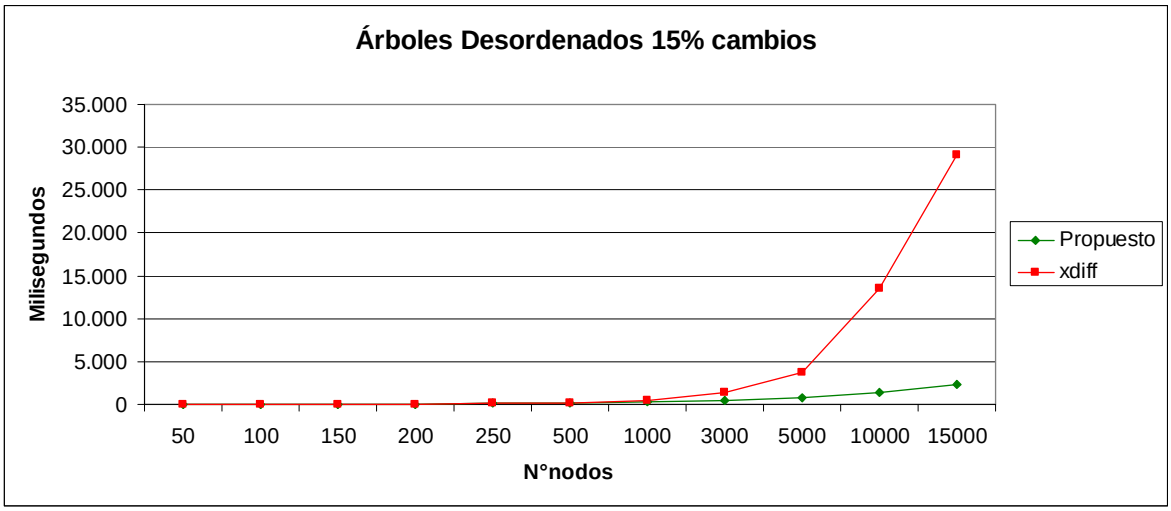


Figura 4.7. Árboles Desordenados con 15% cambios.

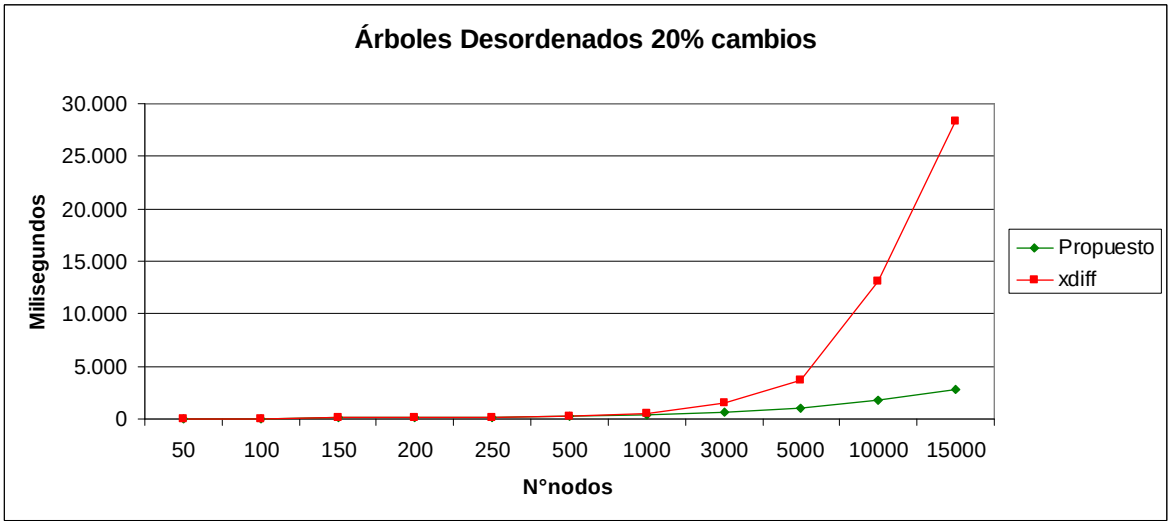


Figura 4.8. Árboles Desordenados con 20% cambios.

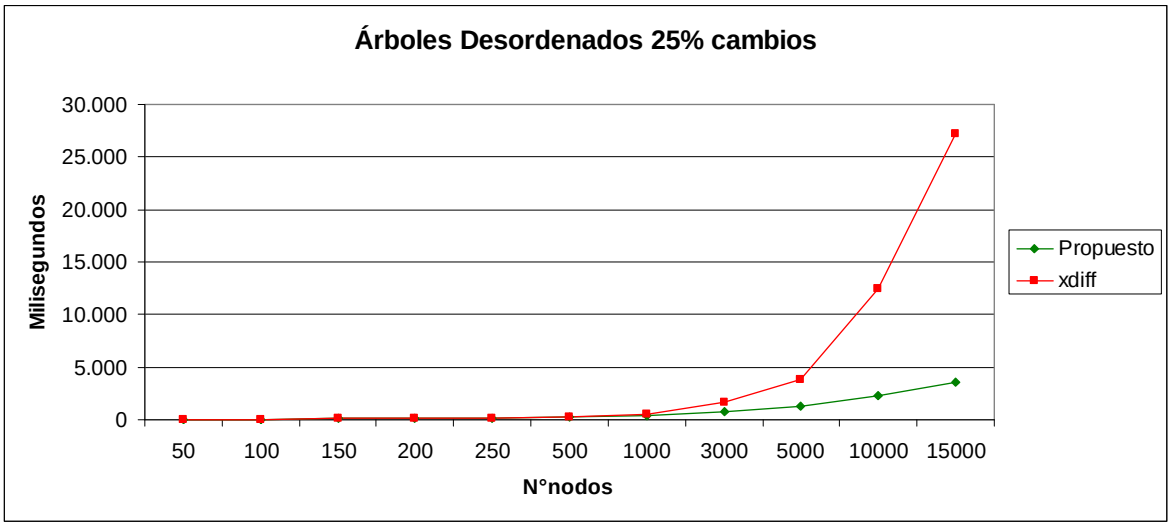


Figura 4.9. Árboles Desordenados con 25% cambios.

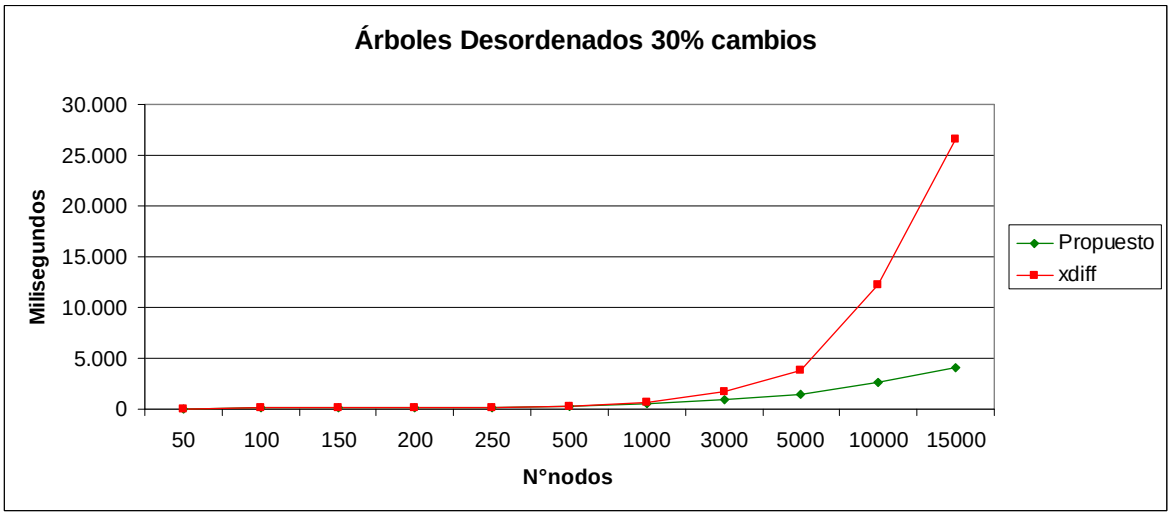


Figura 4.10. Árboles Desordenados con 30% cambios.

Esto es coherente con nuestro análisis de complejidad (para árboles ordenados y desordenados). Lo que demuestra que el tiempo de ejecución del algoritmo propuesto tiene mayor eficiencia en la medida que se incrementa el número de nodos en el documento XML.

4.3 TENDENCIA DE LAS PRUEBAS EMPÍRICAS DE LOS ALGORITMOS

En las siguientes figuras 4.11 y 4.12, se resume la tendencia de cada algoritmo en relación al tipo de árbol (ordenado y desordenado). Como se puede apreciar el algoritmo X-Diff muestra una curva cuadrática mientras que el algoritmo propuesto muestra una curva con tendencia logarítmica en especial cuando la cantidad de nodos es superior a 500.

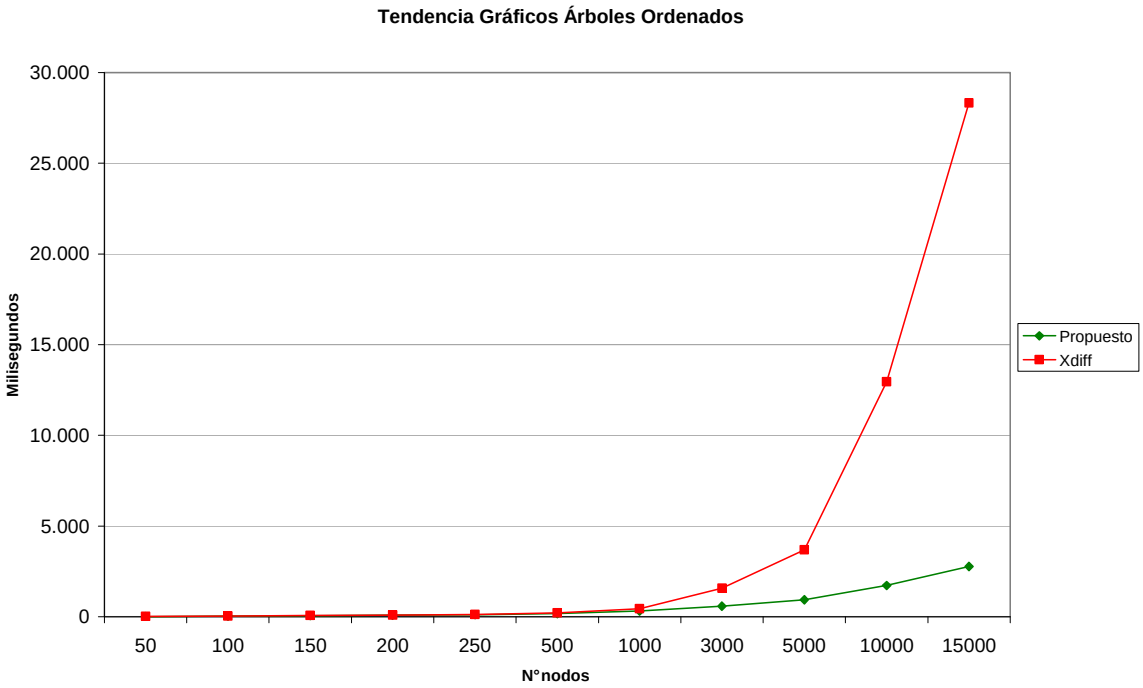


Figura 4.11. Tendencia Árboles Ordenados.

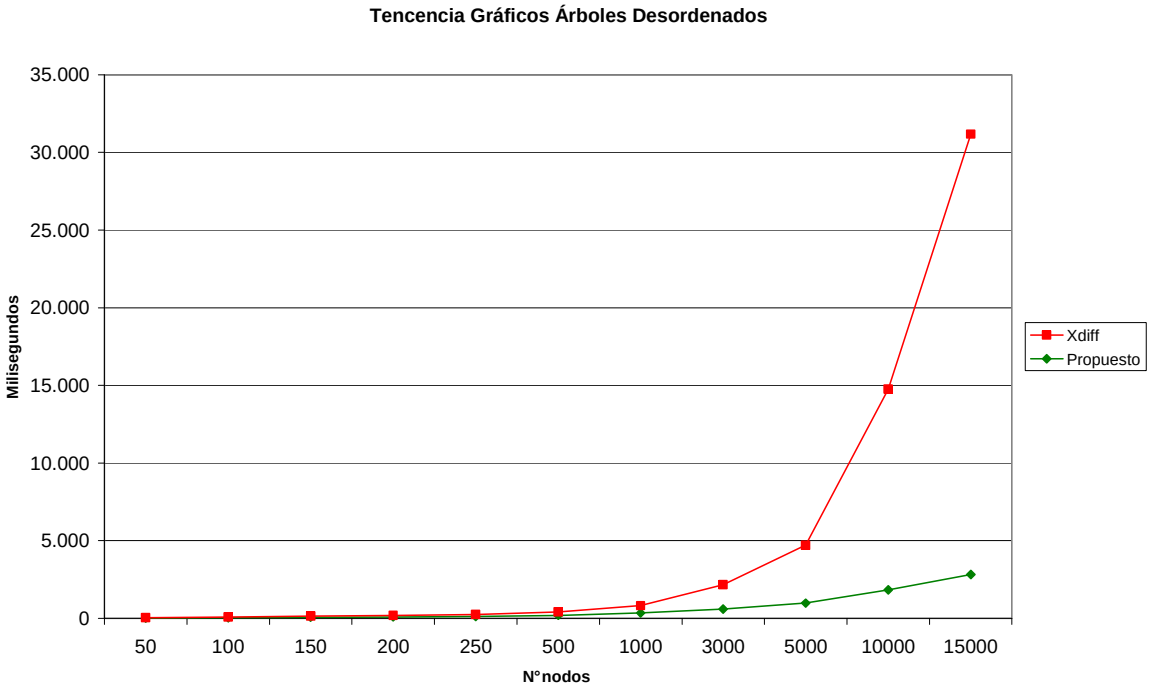


Figura 4.12. Tendencia Árboles Desordenados.

Las diferencias de las curvas de debe a que el algoritmo X-Diff hace una búsqueda secuencial, comparando cada nodo del árbol T1 con los del árbol T2, es decir, examina el primer elemento del árbol T1 en el árbol T2 hasta llegar al último, con un tiempo de complejidad de $O(n^2)$ donde *n* es el número de nodos. Por otro lado el algoritmo propuesto posee ciertas características del algoritmo MergeSort que es un algoritmo de ordenamiento, en donde si el árbol tiene más de dos nodos se divide en dos mitades invocando recursivamente el algoritmo y luego se hace un merge (fusión) de las dos mitades ordenadas, lo que hace que el tiempo de complejidad sea de $O(n \log_m n)$ donde *n* es el número de nodos y *m* es el mayor número de sub-árboles que posee la raíz, en el peor de los casos.

Con los resultados obtenidos vemos gráficamente que a medida que aumenta el número de nodos, el tiempo de ejecución del algoritmo propuesto versus el tiempo del algoritmo con el que lo comparamos (X-Diff), entrega una diferencia notoria entre las dos curvas, siendo una de ellas del tipo cuadrática en el caso del algoritmo X-Diff, mientras que en el caso del algoritmo propuesto es del tipo logarítmica, demostrando así que en todos estos casos el algoritmo propuesto es mejor.

CONCLUSIÓN

Durante el transcurso de este proyecto se abordó los tópicos y características de XML, con la finalidad de analizar, diseñar, implementar y probar un algoritmo para la detección de cambios para este tipo de documentos, logrando así desarrollar de manera satisfactoria los objetivos planteados originalmente.

Este Capítulo está estructurado de la siguiente manera:

- Sección 1: Conclusión General
  - ✓ Objetivo General
  - ✓ Objetivos Específicos
  - ✓ Conclusiones Personales

CONCLUSIÓN GENERAL

Todos los creadores de lenguajes pretenden defender su posición respecto a las ventajas de utilizar sus lenguajes y los promueven a toda costa. Si bien, es posible visualizar los beneficios de simplicidad y claridad que permiten una rápida comprensión de los documentos XML, es necesario trasladar este lenguaje hacia el procesamiento de datos que se maneja entre las máquinas, ya sea desde clientes a servidores, de servidores a servidores o entre aplicaciones.

Los documentos XML proveen de muchos beneficios entre los cuales se pueden mencionar la facilidad de manejo que aporta XML, porque contiene una estructura que no es difícil de aprender (a nivel de creación de documentos) y es posible manejarla a un nivel no tan bajo con un costo no muy alto, la transportabilidad de los datos a través de la Web (sin el formato de presentación) permiten menor congestión en la red y el manejo de los datos asociados a un significado, con el cual es posible un procesamiento mayor tanto en sistemas organizacionales como en Web, porque provee de un grado de inteligencia que se traduce en automatización y mejor servicio.

Finalizado el trabajo podemos concluir de manera exitosa como se abordaron los objetivos que se presentan en el siguiente punto.

OBJETIVO GENERAL HABILITACIÓN PROFESIONAL

Implementar y realizar pruebas empíricas para un eficiente algoritmo de detección de cambios en documentos XML.

Durante el periodo que abarca la Habilitación Profesional se implementó un eficiente algoritmo para la detección de cambios en documentos XML, basado en el algoritmo LPS (Longest Path Subgraph) para detección de cambios en grafos RDF [37]. Posteriormente se comparó con un algoritmo ya existente (X-Diff), logrando a través de pruebas empíricas concluir que el algoritmo propuesto es más eficiente en su funcionamiento y tiempo de ejecución.

OBJETIVOS ESPECÍFICOS HABILITACIÓN PROFESIONAL

a) Implementar algoritmo(s) tentativo(s).

La finalidad de este proyecto era realizar un algoritmo para la detección de cambios en documentos XML, así como la implementación de otros algoritmos ya existentes para su posterior comparación, de los cuales se eligió al algoritmo X-Diff, debido a que es uno de los únicos que trabaja con la información que contiene los nodos, lo cual resulta conveniente a la hora de detectar si hubo o no cambios, sin necesidad de recorrer todo el árbol. No así DiffX debido a que exista o no cambios en el árbol el algoritmo recorre completamente el árbol, a diferencia de X-Diff que sólo lo recorre cuando hay cambios. Por otra parte, el algoritmo LaDiff si bien presenta un tiempo menor, su función es detectar cambios en documentos Latex, y si se consideran los algoritmos basados en LaDiff, éstos presentan un tiempo de complejidad mayor a $O(n^2)$. No se consideraron los algoritmos de O(n), debido a que no existe una descripción detallada de su funcionamiento y sólo existe las versiones ejecutables de ellos. Por ende, no es posible obtener experimentos que realmente reflejen su desempeño en el tiempo considerando. Más aún, nos queda como duda si realmente el algoritmo se comporta de manera lineal en el tiempo, sin olvidar que gran parte de estos algoritmos se restringen a un sólo tipo de árbol (ordenados o desordenados).Véase Capítulo 2 y 4, Anexo 2 y 3.

b) Realizar pruebas empíricas del algoritmo y contrastarlo con otra(s) propuesta(s) existente(s).

Para esta etapa los algoritmos que fueron implementados son examinados tanto en la forma como en el tiempo en que encuentran las diferencias entre los árboles T1 y T2, para esto se evalúan los algoritmos con una diversidad de árboles, los cuales varían su en tamaño (50, 100, 150, 200, 250, 500, 1000, 3000, 5000, 10000, 15000 nodos), así como en su porcentaje de cambios (10%, 15%, 20%, 25% y 30%), con la finalidad de obtener la tendencia de su funcionamiento, los cuales son graficados en el Capítulo 6.

c) Determinar el o los mejores algoritmos desarrollados para la detección de cambios en documentos XML.

Con los resultados obtenidos en el Capítulo 6, vemos que a nivel de tiempo de ejecución del algoritmo propuesto, de $O(n \log_m n)$ donde *n* es el número de nodos y *m* es el mayor número de sub-árboles que posee la raíz, versus el algoritmo con el que lo comparamos, X-Diff de $O(n^2)$ donde *n* es el número de nodos, se procedió a registrar los resultados finales, los que de forma notoria entregaba dos curvas siendo una de ellas del tipo cuadrática en el caso del algoritmo X-Diff, mientras que en el caso del algoritmo propuesto se registra una curva del tipo logarítmica, las cuales a medida que los tamaños de los árboles T1 y T2 eran mayores, generaban una mayor diferencia entre ambas curvas favoreciendo en todos estos casos a la curva del algoritmo propuesto, ya que esta registra un tiempo de ejecución promedio menor a la del algoritmo X-Diff.

CONCLUSIONES PERSONALES

Queda como duda si aquellos algoritmo de O(n) se comportan de manera lineal en el tiempo, debido a que no existe una descripción detallada de su funcionamiento y sólo existen versiones ejecutables de ellos, no es posible obtener experimentos que realmente reflejen su desempeño en el tiempo. A raíz de esto se propone que en un trabajo futuro se investigue a fondo los algoritmos de detección de cambios para documentos XML de O(n) y se realicen la pruebas correspondientes para determinar si realmente cumplen con el orden que plantean.

BIBLIOGRAFÍA

1. Introducción a XML Claudio Burgos (2008) http://php.apsique.com/xml/introduccion/index.html.

2. El lenguaje extensible de marcas (XML) 1.0. Sidar (2008). http://www.sidar.org/recur/desdi/traduc/es/xml/xml1/index.html#sec-intro

3. RDF: Un modelo de metadatos flexible para las bibliotecas digitales del próximo milenio. Dpto. de Biblioteconomía y Documentación Universidad Carlos III de Madrid (2008). http://www.bib.uc3m.es/~mendez/publicaciones/7jc99/rdf.htm

4. Qué es RDF. Malditainternet. (2008). http://www.malditainternet.com/node/96

5. RDF. Wikipedia. (2008). http://www.w3.org/RDF

6. Introducción a la Web semántica. W3C. (2008). http://www.w3c.es/Presentaciones/2005/1018-WebSemanticaREBIUN-MA

7. RDF Especificación del Modelo y la Sintaxis. Sidar. (2008). http://www.sidar.org/recur/desdi/traduc/es/rdf/rdfesp.htm

8. Historia y Origen de RSS http://www.uatsap.com/rss/manual/2

9. RSS. Wikipedia. (2008) http: www.bcn.cl/**rss**

10. Guía fácil de RSS Yahoo. (2008) http://es.geocities.com/rss_guia_facil/como_crear_rss.html#8

11. What is FOAF?. Mfd Consult. (2008). http://xml.mfd-consult.dk/foaf/explorer

12. Sujeto-Predicado-Objeto. Wordpress. (2008). http://wses.wordpress.com/que-es/foaf/

13. DOAC. Wikipedia. (2008). http://ramonantonio.net/doac/0.1/

14. El Consorcio World Wide Web publica CC/PP 1.0 como una W3C Recommendation. W3C (2008). http://www.w3c.es/Prensa/2004/nota040115.html

15. Tancred Lindholm, A 3-way merging algorithm for synchronizing ordered trees – the 3DM merging and differencing tool for XML, Master's thesis, Helsinki University of Technology, Dept. Of Computer Science, Sept. 2001. http://www.cs.hut.fi/~ctl/3dm/thesis.pdf.

16. Tancred Lindholm, A three-way merge for XML documents, Proceedings of the 2004 ACM symposium on Document engineering, 1-10, Oct. 2004.

17. Shu Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo. XML Document Versioning, ACM SIGMOD Record, Volume 30, Issue 3, SPECIAL ISSUE: Special section on advanced XML data processing, 46-53, Sept. 2001.

18. Serge Abiteboul, Sophie Cluet, Guy Ferran, Marie-Christine Rousset, The Xyleme Project, Gemo Report number 248, Nov. 2001.

19. Tancred Lindholm, XML three-way merge as a reconciliation engine for mobile data, Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access, 93-97, Sept. 2003.

20. Yuan Wang, David J. DeWitt, Jin-yi Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. Proceedings of the 19th International Conference on Data Engineering, 519-530. Mar. 2003.

21. Gregory Cobéna, Serge Abiteboul, Amélie Marian: Detecting Changes in XML Documents. Proceedings of the 18th International Conference on Data Engineering, 41-52. Feb. 2002.

22. Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, Jennifer Widom: Change Detection in Hierarchically Structured Information. Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, p. 493-504. June, 1996.

23. Sudarshan S. Chawathe and Hector Garcia-Molina, Meaningful change detection in structured data, Proceedings of the 1997 ACM SIGMOD international conference on Management of data, p. 26-37. May. 1997.

24. XML TreeDiff by IBM alphaWorks, retired Nov. 1998. Idea of David Epstein, designed and implemented by Francisco Curbera. http://alphaworks.ibm.com/tech/xmltreediff/.

25. Raihan Al-Ekram, Archana Adma and Olga Baysal, DiffX: An Algorithm to Detect Changes in Multi-Version XML Documents, School of Computer Science, University of Waterloo. 2005.

26. Sudarshan S. Chawathe, Comparing Hierarchical Data in External Memory, Proceedings of the 25th International Conference on Very Large Data Bases, Sept. 1999.

27. XML Diff and Merge Tool by IBM alphaWorks, last update Mar. 2001, http://www.alphaworks.ibm.com/tech/xmldiffmerge.

28. Kyriakos Komvoteas. XML Diff and Patch Tool. Master's thesis, Master of Science in Distributed and Multimedia Information Systems, Sept 5. 2003 http://treepatch.sourceforge.net/.

29. VM Tools by VM Systems, last release Feb. 2002. http://www.vmsystems.net/vmtools/.

30. Adrian Mouat, DiffXML, June 2002, http://diffxml.sourceforge.net.

31. Haiyuan Xu, Quanyuan Wu, Huaimin Wang, Guogui Yang, Yan Jia. KF-Diff+: Highly Efficient Change Detection Algorithm for XML Documents. On the Move to Meaningful Internet Systems, DOA/CoopIS/ODBASE, p. 1273-1286. Nov. 2002.

32. XML Diff and Patch, Microsoft Corporation. http://apps.gotdotnet.com/xmltools/xmldiff/.
33. Monsell EDM Ltd. Merging XML Changes with DeltaXML. http://www.deltaxml.com/.

34. Robin La Fontaine, DeltaXML, Change Control for XML: Do It Right, (Director of Monsell EDM Ltd.), XML Europe 2003, May 2003.

35. Grégory Cobéna, Talel Abdessalem, Yassine Hinnach A comparative study for XML change detection, Gemo Report number 221, 2002. http://www-rocqbis.inria.fr/verso/Gemo/PUBLI/allbyyear.php ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/Gemo Report-221.pdf.

36. Song, Bhowmick. BioDiff An Effective Fast Change Detection Algorithm for Genomic and Proteomic Data. Proceedings of the Thirteenth ACM conference on Information and knowledge management. 146-147. Nov. 2004.

37. Claudio Gutiérrez-Soto, Pedro G. Campos and Julio Águila. Longest Path Subgraph: A novel and efficient Algorithm to Match RDF Graphs.

ANEXO 1

GLOSARIO

- APPLET: Es un componente de una aplicación que se ejecuta en el contexto de otro programa.

- ASCII: Es un código de caracteres basado en el alfabeto latino tal como se usa en inglés moderno y en otras lenguas occidentales.

- BEAN: Es un componente software que tiene la particularidad de ser reutilizable y así evitar la tediosa tarea de programar los distintos componentes uno a uno.

- DIFF: Comando de UNIX utilizados para la comparación de archivos que genera las diferencias entre dos archivos o los cambios realizados en un archivo determinado comparándolo con una versión anterior del mismo archivo.

- FEEDS: Programas o sitios que permiten leer fuentes Web.

- FRAMEWORK: Es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

- GENÓMICA: Conjunto de ciencias y técnicas dedicadas al estudio integral del funcionamiento, la evolución y el origen de los genomas. La genómica usa conocimientos derivados de distintas ciencias como son: biología molecular, bioquímica, informática, estadística, matemáticas, Física, entre otras.

- HASH: Función o método para generar claves o llaves que representen de manera casi unívoca a un documento, registro, archivo, etc.

- METADATOS: Datos estructurados y codificadas que describen características de instancias conteniendo informaciones para ayudar a identificar, descubrir, valorar y administrar las instancias descritas.

- LATEX: Es un lenguaje de marcado para documentos, y un sistema de preparación de documentos, formado por un gran conjunto de macros de TeX.

- PROTEÓMICOS: Es el estudio del proteoma, estudios que se han realizado tradicionalmente mediante la técnica de electrofóresis en gel de dos dimensiones. En la primera dimensión las proteínas se separan por isoelectroenfoque, que separa las proteínas con base en su carga eléctrica. En la segunda dimensión, las proteínas se separan por peso molecular utilizando SDS-PAGE. El gel se tiñe con Azul de Coomassie o Nitrato de Plata para visualizar las proteínas. Las manchas en el gel son las proteínas que han migrado a una localización específica y permite de esta forma identificarlas.

- RDF: Sigla en inglés Resource Description Framework, (Marco de Descripción de Recursos), es un framework para metadatos en la World Wide Web (WWW), desarrollado por el World Wide Web Consortium (W3C). Permite describir metadatos en sitios Web, proveyendo interoperabilidad entre las aplicaciones que intercambian información en lenguaje maquina por la Web.

- REPOSITORIO: Es un término utilizado en el dominio de las herramientas CASE, que puede definirse como la base de datos fundamental para el diseño.

- RSS: Sigla en ingles Really Simple Syndication. Es un sencillo formato de datos que es utilizado para re-difundir contenidos a suscriptores de un sitio Web. El formato permite distribuir contenido sin necesidad de un navegador, utilizando un software diseñado para leer estos contenidos.

- URI: Es una cadena corta de caracteres que identifica inequívocamente un recurso (servicio, página, documento, dirección de correo electrónico, enciclopedia, etc.). Normalmente estos recursos son accesibles en una red o sistema. Algunos URI pueden ser URL, URN o ambos.

- WEBLOGS: Sitios Web en el que se publican anotaciones (historias, artículos, posts entre otros) mediante un sistema de publicación sencillo, Una de las principales características es que las anotaciones son cronológicas y están ordenadas de más recientes a más antiguas («lo más nuevo arriba»). Normalmente se hace todo vía Web, sin que sea necesario software especial.

- WEBMASTER: Término comúnmente usado para referirse a las personas responsables de un sitio Web específico.

- WEB SEMÁNTICA: Es una Web extendida, dotada de mayor significado en la que cualquier usuario en Internet podrá encontrar respuestas a sus preguntas de forma más rápida y sencilla gracias a una información mejor definida. Al dotar a la Web de más significado y, por lo tanto, de más semántica, se pueden obtener soluciones a problemas habituales en la búsqueda de información gracias a la utilización de una infraestructura común, mediante la cual, es posible compartir, procesar y transferir información de forma sencilla. Esta Web extendida y basada en el significado, se apoya en lenguajes universales que resuelven los problemas ocasionados por una Web carente de semántica en la que, en ocasiones, el acceso a la información se convierte en una tarea difícil y frustrante.

- WORD WIDE CONSORTIUM (W3C): Es un consorcio internacional que produce estándares para la World Wide Web. Está dirigida por Tim Berners-Lee, el creador original de URL (Uniform Resource Locator, Localizador Uniforme de Recursos), HTTP (HyperText Transfer Protocol, Protocolo de Transferencia de HiperTexto) y HTML (Lenguaje de Marcado de HiperTexto) que son las principales tecnologías sobre las que se basa la Web.

- XML: Sigla en inglés de Extensible Markup Language (Lenguaje de Marcas Extensible), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium. Estándar que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

ANEXO 2

PSEUDOCÓDIGO ALGORITMO X-DIFF

Algoritmo Main X-Diff

```
Input: (DOC1, DOC2)
/* Parsing y Hashing */
Parse DOC1 to T1 and hash T1;
Parse DOC2 to T2 and hash T2;
/*Checking y Filtering */
If ( XHash(Root(T1)) = XHash(Root(T2)) )
    DOC1 and DOC2 are equivalent, stop.
Else
    Do Matching – Find a minimum-cost matching Mmin(T1,T2) from T1 to T2
/* Generando optimo delta script */
Do DeltaScript – Generate the minimum-cost edit script E from Mmin(T1,T2)
```

Algoritmo Matching (concordancia)

```
Input: Tree T1 and T2.
Ouput: a minimum-cost matching Mmin(T1,T2).
Initialize: set initial working sets.
N1 = {all leaf nodes in T1}, N2 = {all leaf nodes in T2}.
/* Tabla de Distancias */
Set the Distance Table DT = {}.
/* Paso 1: Reduce matching space */
Filter out next-level subtrees that have equal XHash values.
/* Paso 2: compute editing distance for (T1 • T2). */
Do {
    For every node x in N1
        For every node y in N2
                If  Signature(x) = Signature(y)
                    Compute Dist(x,y);
                    Save matching(x,y) with Dist(x,y) in DT.
    Set N1 = {parent nodes of previous nodes in N1}
          N2 = {parent nodes of previous nodes in N2}
} While (both N1 and N2 are not empty).
/* Paso 3: mark matchings on T1 and T2 */
Set Mmin(T1,T2) = {}
If Signature(Root(T1)) ≠ Signature(Root(T2))
    Return;               /* Mmin(T1,T2) = {} */
Else {
    Add(Root(T1),Root(T2)) to Mmin(T1,T2).
    For every non-leaf node mapping(x,y) ∈ Mmin(T1,T2) {
        Retrieve matchings between their child nodes that are stored in DT.
        Add child node matchings into Mmin(T1,T2).
    }
}
```

Algoritmo DeltaScript

```
Input: Tree T1 and T2, a minimum-cost matching Mmin(T1,T2), the distance table DT.
Output: an delta script E.
Inicialize: set E = Null;
x = Root(T1),  y = Root(T2).
If (x,y) ∉ Mmin(T1,T2)   /* Subtree deletion and insertion */
    Return "Delete(T1), Insert(T2)".
Else  If Dist(T1,T2) = 0
    Return "";
Elsex {
   For every node pair (xi, yj) ∈ Mmin(T1,T2), xi is a child node of x, yj is a child
   node of y.
        If  xi  and  yi are leaf nodes
            If Dist(xi, yj) = 0
                    Return "";
            Else           /* Update leaf node */
                    Add Update(xi, Value(yj)) to E;
        Else{                      /* Call subtree matching */
                Add DeltaScript(Txi, Tyj) to E;
                Return E;
        }
     For every node xi not in Mmin(T1,T2)
            Add "Delete(Txi)" to E;
     For every node  yj not in Mmin(T1,T2)
            Add "Insert(Tyj)" to E;
     Return E.
}
```

PSEUDOCÓDIGO ALGORITMO LADIFF

Algoritmo DeltaScript

E←., *M'* ←*M*

Visit the nodes of T_2 in breadth-first order

/ Combines the update, insert, align, and move phases */*

Let *x* be the current node in the breadth-first search of T_2 and let *y*= *p(x).*

Let *z* be the partner of *y* in *M'*. (*)

If *x* has no partner in *M'*

k←*FindPos(x)*

Append *Ins((w, a, v(x)), z, k)* to *E*, for a new identifier *w*.

Add (*w*, *x*) to *M'* and apply *Ins((w, a, v(x)), z, k)* to T_1.

Else If *x* is not the root /* *x* has a partner in *M'* */

Let *w* be the partner of *x* in *M'*, and let *v* = *p(w)* in T_1.

If *v(w)* • *v(x)*

Append *Upd(w, v(x))* to *E.*

Apply *Upd(w, v(x))* to T_1.

If *(y, x)* ∉ *M'*

Let *z* be the partner of *y* in *M'. (*)*

k←*FindPos (x)*

Append *Mov (w, z, k)* to *E.*

Apply *Mov (w, z, k)* to T_1.

AlingChildren (*w, x)*

Do a post-order traversal of T_1 /* the delete phase */

Let *w* be the current node in the post-order traversal of T_1.

If *w* has no partner in *M'* then append *Del(w)* to *E* and apply *Del(w)* to T_1.

E is a minimum cost edit script, *M'* is a total matching, and T_1 *is isomorphic to* T_2.

Función *AlignChildren (w, x)*

Mark all children of *w* and all children of *x* "out of order".

Let S_1 be the sequence of children of *w* whose partners are children of *x* and let S_2 be the sequence of children of *x* whose partners are children of *w*.

Define the function *equal(a, b)* to be trae if and only if *(a, b)* ∈ *M'*.

Let $S \leftarrow LCS(S_1, S_2, equal)$.

For each *(a, b)* ∈ *S*, mark nodes *a* and *b* "in order".

For each $a \in S_1$, $b \in S_2$ such that *(a, b)* ∈ *M* but *(a, b)* ∉ *S* {

k← *FindPos (b).*

Append *Mov(a, w, k)* to *E and apply Mov(a, w, k)* to T_1.

Mark a and b "in order".

}

Función *FindPos (x)*

Let y = p(x) in T_2 and let *w* be the partner of *x* (x ∈ T_1).

If *x* is the leftmost child of *y* that is marked "in order", return 1.

Find *v* ∈ T_2 where *v* is the rightmost sibling of *x* that is to the left of *x* and is marked "in order".

Let *u* be the partner of *v* in T_1.

Suppose *u* is the *i*th child of its parent (counting from let to right) that is marked "in order", return *i* + 1.

PSEUDOCÓDIGO ALGORITMO DIFFX

Isolated Tree Fragment Mapping

```
Procedure Mapping (T1, T2, M)
     Input T1, T2: tree
     Ouput M: map
Begin
     Index the nodes of T2
     Traverse T1 in a level-order sequence
          Let x be the current node
          If (x, _) ∈ M Then
               Skip current node
          End-If
          Let  y[] = all nodes from T2 equal to x
          M'' = ∅
          For i = 1 to size of y[]
               If (_, y[i]) ∉ M Then
                    M' = ∅
                    Match-Fragment ( x, y[i], M, M' )
                    If size of  M' > size of M'' then
                           Let M'' = M'
                    End-If
               End-If
          End-For
          Let M = M ∪ M''
     End-Traverse
End

Procedure Match-Fragment ( x, y, M, M' )
     Input x, y: node ; M: map
     Ouput: M': map
Begin
     If  (x, _) ∉ M   and  (_, y) ∉ M   and   node(x) = node(y)   then
          Let M' = M' ∪ {(x,y)}
          For i = 1 to minimum of number of children between x and y
               Match-Fragment ( i-th child of x, i-th child of  y, M, M' )
          End-For
     End-If
End
```

[Note: The _ (underscore) is a wildcard in the pair (x,_)]

Generación del Delta Script

```
Procedure Generate-Script ( T1, T2, M )
     Input T1, T2: tree ; M: map
     Output S: list
Begin
     Traverse T2 in a level-order sequence
          Let y be the current node
          If ( _, y ) ∉ M  then
               Add  to  S  "insert ( y, parent(y), position(y) )"
          Else
               Retrieve x such that (x,y) ∈ M
```

```
            If ( parent(x), parent(y) ) ∉ M   then
                 Add  to  S  "move ( x, parent(y), position(y) )"
            Else-If  position(x) ≠ position(y)  Then
                 Add  to  S  "move ( x, parent(y), position(y) )"
            End-If
        End-If
    End-Traverse
    Traverse T1 in a level-order sequence
        Let x be the current node
        If ( x, _ ) ∉ M Then
            Add  to  S  "delete ( x )"
        End-If
    End-Traverse
End
```

ANEXO 3

a) CODIFICACIÓN ALGORITMO PROPUESTO

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>


typedef struct nodo{
    int id;
    char etiqueta[50];
    int hash;
    int ordenado;
    struct nodo *padre;
    struct nodo *hermano;
    struct nodo *hijo;
}Tnodo;



//---------------------Funciones---------------------------


//inicializacion del arbol
Tnodo *raizArbol(char e[])
{
    Tnodo *n = (Tnodo *)malloc(sizeof(Tnodo));
    n->id=1;
    strcpy(n->etiqueta,e);
    n->hash=0;
    n->ordenado=0;
    n->hijo=NULL;
    n->hermano=NULL;
    n->padre=NULL;

    return n;
}


// identifica el ultimo hermano de la lista
// para agregar nodo nuevo al final
Tnodo **finalLista(Tnodo *a)
{
    if(a->hermano != NULL)
        a=*finalLista(a->hermano);

    return &a;
}


//individualizacion del nodo, padre de...
void **buscarNodo(Tnodo *a, Tnodo **pos, int p)
{
    if(a != NULL ){
        if(a->id == p){
                (*pos)=a;
        }else{
                buscarNodo(a->hijo,pos,p);
                buscarNodo(a->hermano,pos,p);
        }
    }
}

//agregar nuevo nodo
void agregarNodo(Tnodo **a, int i, int p, char e[], int h)
{
    Tnodo *pos=NULL;
    buscarNodo(*a,&pos,p);

    Tnodo *n = (Tnodo *)malloc(sizeof(Tnodo));
    if (n != NULL)
    {
        n->id=i;
        strcpy(n->etiqueta,e);
        n->hash=h;
        n->ordenado=0;
        n->hijo=NULL;
```

```
        n->hermano=NULL;
        n->padre=pos;

        if(pos->hijo != NULL)
                (*finalLista(pos->hijo))->hermano=n;
        else
                pos->hijo=n;
    }
}


//------------------------ Hash -------------------

//suma hash y asigna a cada padre su hash
void hashArbol(Tnodo *a, int *suma)
{
    if(a!=NULL){

        hashArbol(a->hermano,suma);
        hashArbol(a->hijo,suma);


        if(a->hijo!=NULL){  // al hash del padre se adiciona su id
           a->hash = a->hash + a->id;
        }

        if(a->padre!=NULL){  // al hash del padre se adiciona el hash de cada hijo
           a->padre->hash = a->padre->hash + a->hash;
        }
    }
}


//determina si los arboles son iguales
int matching(Tnodo *a1, Tnodo *a2)
{
    if(a1->hash == a2->hash){
        return 1;
    }
    return 0;
}


//---------------------- Ordenar ---------------------

// Localiza el nodo con posicion num, para la division
Tnodo **localizaNodo(Tnodo *a, int num)
{
    int i;
    for(i=0; i < num ; i++)
        a=a->hermano;

    return (&a);
}

// Copia la informacion que contine otro nodo.
void copiarNodo(Tnodo **b, Tnodo *a)
{
    Tnodo *n = (Tnodo *)malloc(sizeof(Tnodo));

    if (n != NULL)
    {
        n->id=a->id;
        strcpy(n->etiqueta,a->etiqueta);
        n->hash=a->hash;
        n->ordenado=1;
        n->hijo=a->hijo;
        n->hermano=NULL;
        n->padre=a->padre;

        if((*b) != NULL)
             (*finalLista(*b))->hermano=n;
        else
             (*b)=n;
    }
}
```

```
// Cambia la direccion de memoria a la que apuntan
// los nodos hijos del antiguo padre para que apunten al nuevo.
void copiarHijos(Tnodo *bio_h, Tnodo *ado)
{
    while(bio_h!=NULL)
    {
        bio_h->padre=ado;

         bio_h=bio_h->hermano;
    }
}

// Fusiona despues de la division, de forma ordenada
void fusiona(Tnodo *a, int ini, int med, int fin)
{
    Tnodo *aux=NULL;
    Tnodo *a_i=NULL;
    Tnodo *a_j=NULL;
    Tnodo *cpy_arbol=a;

    int i = ini; // Índice de la parte izquierda
    int j = med+1; // Índice de la parte derecha
    int k = 0; // Índice del vector aux


    /* Mientras ninguno de los indices llegue a su fin vamos realizando
       comparaciones. El elemento más pequeño se copia al vector aux */

    while (i <= med && j <= fin) {
        a_i=(*localizaNodo(a,i));   // nodo indice parte izq
        a_j=(*localizaNodo(a,j));   // nodo indice parte der


        if ((a_i->id) < (a_j->id)) {
            copiarNodo(&aux,a_i);
            ++i;
        }
        else {
            copiarNodo(&aux,a_j);
            ++j;
        }
        ++k;
    }

    /* Uno de los dos sub-vectores ya ha sido copiado del todo, simplemente
       debemos copiar todo el sub-vector que nos falte */

    for(a_i=(*localizaNodo(a,i)) ; (i <= med) ; i++, k++)
    {
        copiarNodo(&aux,a_i);
        a_i=a_i->hermano;
    }

    for(a_j=(*localizaNodo(a,j)) ; (j <= fin) ; j++, k++)
    {

        copiarNodo(&aux,a_j);
        a_j=a_j->hermano;
    }


    /* Copiamos los elementos ordenados de aux al vector original a,
        para no tener problemas con el indice, ocupamos la variable
        cpy_arbol, que es la copia de a */

    int n;
    for (n=0, cpy_arbol=(*localizaNodo(a,(ini+n))); n < k; n++)
    {
        cpy_arbol->id=aux->id;
        strcpy(cpy_arbol->etiqueta,aux->etiqueta);
        cpy_arbol->hash=aux->hash;
        cpy_arbol->ordenado=1;
        cpy_arbol->hijo=aux->hijo;
        cpy_arbol->padre=aux->padre;

        /* redireccionamos los hijos al nuevo padre */
        copiarHijos(aux->hijo,cpy_arbol);

        /* Avanzamos */
        cpy_arbol=cpy_arbol->hermano;
        aux=aux->hermano;
    }
}
```

```
// Determina la cantidad de hermanos, para dividir.
int cantidadHermanos(Tnodo *a)
{
    int i;
    for(i=0; a != NULL ; i++)
        a=a->hermano;

    return i;
}

//Funcion MergeSort, hace la divsion de los hermanos.
void merge_sort(Tnodo *a, int ini, int fin) {
    /* Si ini = fin el sub-vector es de un solo elemento y, por lo tanto
       ya está ordenado por definición */
    if (ini < fin) {
        merge_sort(a, ini, (ini + fin)/2);   // la primera mitad
        merge_sort(a, ((ini + fin)/2)+1, fin);   //la segunda mitad
        fusiona(a, ini, (ini + fin)/2, fin);
    }
}

//Funcion ordenar principal, baja por niveles
void ordenarArbol(Tnodo **a)
{
    if((*a)!=NULL){
        if((*a)->ordenado==0){
            int suma=0;
            suma=cantidadHermanos(*a);
            merge_sort((*a),0,suma-1);
        }
        ordenarArbol(&((*a)->hijo));
        ordenarArbol(&((*a)->hermano));
    }
}

//--------------- Modificaciones -----------------------------

// Cuenta e imprime todo los nodos hijo del nodo eliminado.
void diffSubArbolEliminado(Tnodo *a, int *suma)
{
    if(a!=NULL)
    {
        (*suma)=(*suma)+1;
        printf("-----ELIMINADO-----\n");
        printf("Nodo(%d) Etiqueta(%s)\n\n",a->id,a->etiqueta);
        diffSubArbolEliminado(a->hijo,suma);
        diffSubArbolEliminado(a->hermano,suma);
    }
}

// Cuenta e imprime todo los nodos hijo del nodo insertado.
void diffSubArbolInsertado(Tnodo *a, int *suma)
{
    if(a!=NULL)
    {
        (*suma)=(*suma)+1;
        printf("-----INSERTADO-----\n");
        printf("Nodo(%d) Etiqueta(%s)\n\n",a->id,a->etiqueta);
        diffSubArbolInsertado(a->hijo,suma);
        diffSubArbolInsertado(a->hermano,suma);
    }
}

// Cuenta e imprime todas las diferencias.
void diffArbol(Tnodo *a1, Tnodo *a2, int *suma)
{
    if(a1 != NULL && a2 != NULL )
    {
        if((a1->id == a2->id) && (a1->hash == a2->hash))
        {     /* Sino a hay cambios por esta rama, sigue a la sgte. */
            diffArbol(a1->hermano, a2->hermano, suma);
        }
        else{
            if((a1->id == a2->id) && (a1->hash != a2->hash))
            {     /* Baja hasta encontrar el nodo hoja modificado */
                diffArbol(a1->hijo, a2->hijo, suma);

                if(a1->hijo == NULL && a2->hijo == NULL){
                    (*suma)=(*suma)+1;
                    printf("-----ACTUALIZADO-----\n");
                    printf("Antes   =  Nodo(%d) Etiqueta(%s) Hash(%d)\n",a1->id,a1-
>etiqueta,a1->hash);
                    printf("Despues =  Nodo(%d) Etiqueta(%s) Hash(%d)\n\n",a2->id,a2-
>etiqueta,a2->hash);
                }
```

```
                  /* Sigue hacia el lado para encontrar otro nodo hoja modificado. */
                diffArbol(a1->hermano, a2->hermano, suma);
            }
            else
            {/*Si los ID son distintos es pq hay Eliminacion o Insercion, entre nodos*/
                if((a1->id != a2->id))
                {
                    if((a1->id < a2->id))
                    {  /* Determina si hay eliminacion, entre nodos. */
                        (*suma)=(*suma)+1;
                        printf("-----ELIMINADO-----\n");
                        printf("Nodo(%d) Etiqueta(%s)\n\n",a1->id,a1->etiqueta);
                        diffSubArbolEliminado(a1->hijo,suma);
                        diffArbol(a1->hermano, a2, suma);
                    }
                    else{
                        if((a1->id > a2->id))
                        {  /* Determina si hay alguna insercion, entre nodos. */
                            (*suma)=(*suma)+1;
                            printf("-----INSERTADO-----\n");
                            printf("Nodo(%d) Etiqueta(%s)\n\n",a2->id,a2->etiqueta);
                            diffSubArbolInsertado(a2->hijo,suma);
                            diffArbol(a1, a2->hermano, suma);
                        }
                    }
                }
            }
        }
    }
    else{  /* Determina si hay alguna Eliminacion o Insercion, al final. Alguna rama. */
        if(a1 != NULL && a2 == NULL)
        {
            (*suma)=(*suma)+1;
            printf("-----ELIMINADO-----\n");
            printf("Nodo(%d) Etiqueta(%s)\n\n",a1->id,a1->etiqueta);
            diffSubArbolEliminado(a1->hijo,suma);
            diffArbol(a1->hermano, a2, suma);
        }
        else{
            if(a1 == NULL && a2 != NULL)
            {
                (*suma)=(*suma)+1;
                printf("-----INSERTADO-----\n");
                printf("Nodo(%d) Etiqueta(%s)\n\n",a2->id,a2->etiqueta);
                diffSubArbolInsertado(a2->hijo,suma);
                diffArbol(a1, a2->hermano, suma);
            }
        }
    }
}

//--------------------Main-------------------

int main()
{
    Tnodo *a1 = NULL;
    Tnodo *a2 = NULL;
    int idnodo, padre, hash;
    char etiqueta[50], arbol[50];
    int suma=0;

    FILE *ptr;

    if(ptr = fopen("arboles.txt","r"))
    {
        a1=raizArbol("Catalogo");
        a2=raizArbol("Catalogo");

        while(fscanf(ptr,"%s %d %d %s %d",&arbol,&idnodo,&padre,etiqueta,&hash) != EOF)
        {
            if(strcmp(arbol,"a1") == 0)
                 agregarNodo(&a1,idnodo,padre,etiqueta,hash);
            else{
                 if(strcmp(arbol,"a2") == 0)
                      agregarNodo(&a2,idnodo,padre,etiqueta,hash);
            }
        }
        fclose(ptr);

        clock_t comienzo, fin;
        double mseg=0;

        comienzo=clock();

        /* Ordena el arbol a2 */
        ordenarArbol(&a2);
```

```
        /* Calcula hash */
        hashArbol(a1,&suma);
        hashArbol(a2,&suma);

        /* Determina si hay matching */
        if(matching(a1,a2))
            printf("\nARBOLES IGUALES\n\n");
        else{
            printf("\nARBOLES DISTINTOS\n\n\n");
            suma=0;
            diffArbol(a1,a2,&suma);   // Busca las diferencias
            printf("\nDIFERENCIAS: %d\n",suma);
        }

       fin=clock();
       mseg=((fin-comienzo)/(double)1);
       printf("MSEG: %g\n", mseg);


       if(ptr = fopen("salida.txt","a+"))
       {
           fprintf(ptr,"%g\n",mseg);
           fclose(ptr);
       }else{
        printf("Error: No se pudo abrir el archivo para guardar los tiempos.\n");
       }

    }else{
        printf("Error: No se pudo abrir el archivo\n");
    }

    getchar();
    return 0;
}
```

CODIFICACIÓN ALGORITMO X-DIFF

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>


typedef struct nodo{
    int id;
    char etiqueta[50];
    int hash;
    struct nodo *padre;
    struct nodo *hermano;
    struct nodo *hijo;
}Tnodo;


//---------------------Funciones--------------------------


Tnodo *raizArbol(char e[])
{
    Tnodo *n = (Tnodo *)malloc(sizeof(Tnodo));
    n->id=1;
    strcpy(n->etiqueta,e);
    n->hash=0;
    n->hijo=NULL;
    n->hermano=NULL;
    n->padre=NULL;

    return n;
}


Tnodo **finalLista(Tnodo *a)
{
    if(a->hermano != NULL)
        a=*finalLista(a->hermano);

    return &a;
}


void **buscarNodo(Tnodo *a, Tnodo **pos, int p)
{
    if(a != NULL ){
        if(a->id == p){
                (*pos)=a;
        }else{
                buscarNodo(a->hijo,pos,p);
                buscarNodo(a->hermano,pos,p);
        }
    }
}


void agregarNodo(Tnodo **a, int i, int p, char e[], int h)
{
    Tnodo *pos=NULL;
    buscarNodo(*a,&pos,p);

    Tnodo *n = (Tnodo *)malloc(sizeof(Tnodo));
    if (n != NULL)
    {
        n->id=i;
        strcpy(n->etiqueta,e);
        n->hash=h;
        n->hijo=NULL;
        n->hermano=NULL;
        n->padre=pos;

        if(pos->hijo != NULL)
                (*finalLista(pos->hijo))->hermano=n;
        else
                pos->hijo=n;
    }
}
```

```
//------------------------ Hash -------------------

void hashArbol(Tnodo *a, int *suma)
{
    if(a!=NULL)
    {
        hashArbol(a->hijo,suma);
        hashArbol(a->hermano,suma);

        if(a->hijo != NULL){
            (*suma)=(*suma)+ (a->id);
        }else{
            (*suma)=(*suma)+ (a->hash);
        }
    }
}

int matching(int suma1, int suma2)
{
    if(suma1 == suma2){
        return 1;
    }
    return 0;
}

//--------------- Modificaciones -----------------------------

void diffNodo(Tnodo *a, Tnodo **ubic, int idn)
{
    if(a != NULL ){
        if(a->id == idn){
                (*ubic)=a;
        }else{
                diffNodo(a->hijo,ubic,idn);
                diffNodo(a->hermano,ubic,idn);
        }
    }
}

void diffNodoMayor(Tnodo *a, Tnodo **ubic, int may)
{
    if(a != NULL ){
        if(a->id > may){
                (*ubic)=a;
        }else{
                diffNodoMayor(a->hijo,ubic,may);
                diffNodoMayor(a->hermano,ubic,may);
        }
    }
}

void busModificado(Tnodo *a1, Tnodo *a2, int *suma, int *limite)
{
    Tnodo *ubic=NULL;

    if(a1!=NULL)
    {
        diffNodo(a2,&ubic,a1->id);

        if(ubic == NULL)
        {
                (*suma)=(*suma)+1;
                printf("-----ELIMINADO-----\n");
                printf("Nodo(%d) Etiqueta(%s)\n\n",a1->id,a1->etiqueta);
        }else{
            if(a1->id == ubic->id)
            {
                if(a1->hijo==NULL && ubic->hijo==NULL && (a1->hash != ubic->hash))
                {
                    (*suma)=(*suma)+1;
                    printf("-----ACTUALIZADO-----\n");
                    printf("Antes   =  Nodo(%d) Etiqueta(%s) Hash(%d)\n",a1->id,a1-
>etiqueta,a1->hash);
                    printf("Despues =  Nodo(%d) Etiqueta(%s) Hash(%d)\n\n",ubic-
>id,ubic->etiqueta,ubic->hash);
                }
            }

        }
        (*limite)=a1->id;

        busModificado(a1->hijo,a2,suma,limite);
        busModificado(a1->hermano,a2,suma,limite);
    }
}
```

```
void busInsertado(Tnodo *a1, Tnodo *a2, int *suma, int *limite)
{
    Tnodo *ubic=NULL;

    if(a2!=NULL)
    {
        diffNodo(a1,&ubic,a2->id);
        if(ubic == NULL)
        {
                (*suma)=(*suma)+1;
                printf("-----INSERTADO-----\n");
                printf("Nodo(%d) Etiqueta(%s)\n\n",a2->id,a2->etiqueta);
        }

        busInsertado(a1,a2->hijo,suma,limite);
        busInsertado(a1,a2->hermano,suma,limite);
    }
}

void diffArbol(Tnodo *a1, Tnodo *a2, int *suma)
{
    Tnodo *cab_a1=a1;
    Tnodo *cab_a2=a2;
    int limite=0;

    busModificado(a1,a2,suma,&limite);
    busInsertado(a1,a2,suma,&limite);
}

//--------------------Main-------------------

int main()
{
    Tnodo *a1 = NULL;
    Tnodo *a2 = NULL;
    int idnodo, padre, hash;
    char etiqueta[50], arbol[50];
    int suma1=0, suma2=0;

    FILE *ptr;

    if(ptr = fopen("arboles.txt","r"))
    {
        a1=raizArbol("Catalogo");
        a2=raizArbol("Catalogo");

        while(fscanf(ptr,"%s %d %d %s %d",&arbol,&idnodo,&padre,etiqueta,&hash) != EOF)
        {
            if(strcmp(arbol,"a1") == 0)
                agregarNodo(&a1,idnodo,padre,etiqueta,hash);
            else{
                if(strcmp(arbol,"a2") == 0)
                    agregarNodo(&a2,idnodo,padre,etiqueta,hash);
            }
        }
        fclose(ptr);

        clock_t comienzo, fin;
        double mseg=0;

        comienzo=clock();

        hashArbol(a1,&suma1);
        hashArbol(a2,&suma2);

        if(matching(suma1,suma2))
            printf("\nARBOLES IGUALES\n\n");
        else{
            printf("\nARBOLES DISTINTOS\n\n\n");
            suma1=0;
            diffArbol(a1,a2,&suma1);
            printf("\nDIFERENCIAS: %d\n",suma1);
        }

       fin=clock();
       mseg=((fin-comienzo)/(double)1);
       printf("MSEG: %g\n", mseg);


       if(ptr = fopen("salida.txt","a+"))
       {
           fprintf(ptr,"%g\n",mseg);
           fclose(ptr);
       }else{
        printf("Error: No se pudo abrir el archivo para guardar los tiempos.\n");
       }
```

```
    }else{
        printf("Error: No se pudo abrir el archivo\n");
    }

    getchar();
    return 0;
}
```

ANEXO 4

a) ÁRBOL ORIGINAL (ORDENADO)

Contenido del documento XML de 50 nodos

arbol original	identificador del Nodo	identificador del Padre	contenido del nodo	valor hash del nodo
a1	2	1	Ropa	0
a1	3	2	Hombre	0
a1	4	3	Pr1	0
a1	5	4	N	1005
a1	6	4	V	2006
a1	7	3	Pr2	0
a1	8	7	N	3008
a1	9	7	V	4009
a1	10	3	Pr3	0
a1	11	10	N	5011
a1	12	10	V	6012
a1	13	3	Pr4	0
a1	14	13	N	7014
a1	15	13	V	8015
a1	16	3	Pr5	0
a1	17	16	N	9017
a1	18	16	V	10018
a1	19	3	Pr6	0
a1	20	19	N	11020
a1	21	19	V	12021
a1	22	3	Pr7	0
a1	23	22	N	13023
a1	24	22	V	14024
a1	25	3	Pr8	0
a1	26	25	N	15026
a1	27	25	V	16027
a1	28	2	Mujer	0
a1	29	28	Pr9	0
a1	30	29	N	17030
a1	31	29	V	18031
a1	32	28	Pr10	0
a1	33	32	N	19033
a1	34	32	V	20034
a1	35	28	Pr11	0
a1	36	35	N	21036
a1	37	35	V	22037
a1	38	28	Pr12	0
a1	39	38	N	23039
a1	40	38	V	24040
a1	41	28	Pr13	0

a1	42	41	N	25042
a1	43	41	V	26043
a1	44	28	Pr14	0
a1	45	44	N	27045
a1	46	44	V	28046
a1	47	28	Pr15	0
a1	48	47	N	29048
a1	49	47	V	30049
a1	50	28	Pr16	0
a1	51	50	N	31051
a1	52	50	V	32052

b) ÁRBOL MODIFICADO EN UN 10% (ORDENADO)

Contenido del documento XML de 50 nodos

arbol original	identificador del Nodo	identificador del Padre	contenido del nodo	valor hash del nodo
a2	2	1	Ropa	0
a2	3	2	Hombre	0
a2	4	3	Pr1	0
a2	5	4	N	1005
a2	6	4	V	36006
a2	7	3	Pr2	0
a2	8	7	N	3008
a2	9	7	V	4009
a2	10	3	Pr3	0
a2	11	10	N	5011
a2	12	10	V	6012
a2	13	3	Pr4	0
a2	14	13	N	7014
a2	15	13	V	36015
a2	16	3	Pr5	0
a2	17	16	N	9017
a2	18	16	V	10018
a2	19	3	Pr6	0
a2	20	19	N	11020
a2	21	19	V	12021
a2	22	3	Pr7	0
a2	23	22	N	13023
a2	24	22	V	14024
a2	25	3	Pr8	0
a2	26	25	N	15026
a2	27	25	V	35027
a2	28	2	Mujer	0
a2	29	28	Pr9	0

a2	30	29	N	17030
a2	31	29	V	18031
a2	32	28	Pr10	0
a2	33	32	N	19033
a2	34	32	V	20034
a2	35	28	Pr11	0
a2	36	35	N	21036
a2	37	35	V	22037
a2	38	28	Pr12	0
a2	39	38	N	23039
a2	40	38	V	24040
a2	41	28	Pr13	0
a2	42	41	N	25042
a2	43	41	V	34043
a2	44	28	Pr14	0
a2	45	44	N	27045
a2	46	44	V	28046
a2	47	28	Pr15	0
a2	48	47	N	29048
a2	49	47	V	30049
a2	50	28	Pr16	0
a2	51	50	N	31051
a2	52	50	V	33052