XKeyDiff - Um algoritmo semântico para detecção de mudanças entre documentos XML

Rodrigo C. Santos[1], Carmem S. Hara (orientadora)[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – 81.531-990 – Curitiba – PR – Brasil

rcs00br@yahoo.com.br, carmem@inf.ufpr.br

Abstract. *There are several diff algorithms for XML proposed in the literature. Some of them favor the quality of the changes detected while others aim at performance. In both approaches only the structure of the document is considered. In this paper, we present a diff algorithm in which XML keys are used to match elements in the documents prior to their structural analisys. The effectiveness of this approach in improving the results of an existing diff algorithm is shown by an example.*

Resumo. *Existem diversos trabalhos de algoritmos de diff para documentos XML, alguns concernentes à qualidade da detecção das mudanças, outros preocupados com a eficiência do método utilizado. Em ambas as abordagens, a análise do documento é apenas estrutural. Este trabalho propõe um algoritmo que faz uso de chaves para XML e um outro algoritmo de diff, a fim de trabalhar com a semântica dos dados. Um exemplo é descrito, demonstrando a significativa melhora dos resultados obtidos que a abordagem proposta proporciona.*

1. Introdução

A representação de dados em XML tornou-se muito popular nos últimos anos, principalmente como meio de publicação e transporte de dados na *Web*. Uma vez que as informações *on-line* são alteradas de maneira constante, é necessário o uso de alguma ferramenta que detecte tais alterações. Além disto, os usuários não se interessam somente pelos valores correntes dos dados, mas também pelas alterações. O usuário pode querer monitorar mudanças (por exemplo, novos produtos que foram adicionados ao catálogo), ou querer buscar informações de versões antigas de um documento (por exemplo, buscar o preço de um produto há um mês). Por isso, existem diversos trabalhos de algoritmos de *diff*, ou seja, algoritmos que detectam as mudanças existentes entre documentos [Abiteboul et al., 2002a].

Os algoritmos de *diff* existentes possuem, basicamente, duas preocupações: a qualidade da detecção das mudanças e a eficiência do método aplicado para esta detecção. A qualidade diz respeito à exatidão das diferenças detectadas, bem como a inexistência de informações redundantes sobre estas diferenças. A eficiência diz respeito ao desempenho do método utilizado para análise das diferenças entre os documentos. Um método ineficiente pode ser inviável para aplicações que trabalham com uma grande quantidade de dados como, por exemplo, a detecção das mudanças de documentos a serem armazenados em um *datawarehouse*.

Estes algoritmos, mesmo aqueles cuja preocupação principal é a qualidade, identificam as alterações nos documentos baseados em uma análise sintática dos dados. A semântica, ou seja, o real significado dos dados, não é analisada. Desta maneira, estruturas tendenciosas de documentos podem gerar detecções errôneas de mudanças.

Além disto, existe uma preocupação na área de banco de dados sobre a qualidade dos dados. Uma das áreas que tem recebido bastante atenção atualmente é a de *Data Cleansing* [Maletic and Marcus, 2000], ou seja, a detecção automática de possíveis erros nos dados de entrada. Na área de *datawarehouse*, o *data cleansing* envolve a tarefa de identificar registros duplicados ou errôneos que referem-se a mesma entidade e armazenados em bases de dados distintas. Por exemplo, imagine que uma empresa mantenha um *datawarehouse*, onde armazena tanto dados importados de outras bases, bem como dados gerados na própria empresa. De tempos em tempos, as bases de dados externas disponibilizam novas versões de seus dados e, portanto, o *datawarehouse* deve ser atualizado. O processo de atualização seria enormemente facilitado se fosse baseado em um algoritmo que detectasse as diferenças entre as duas versões dos dados baseado na identidade de suas entidades. Ou seja, primeiramente os registros que referem-se a mesma entidade nas duas versões são identificados, e somente então as alterações nestes registros são detectadas.

Baseado nesta idéia, neste artigo propomos um algoritmo de *diff* para XML, chamado *XKeyDiff*, que combina duas técnicas: chaves para XML e um algoritmo de *diff* para XML. O objetivo do trabalho é determinar se a combinação das técnicas efetivamente resulta na melhoria da qualidade das mudanças detectadas entre documentos.

O conceito de chaves para XML é análogo ao conceito de chaves nas bases de dados relacionais; ou seja, elas permitem identificar unicamente um elemento no documento. Neste trabalho, adotamos a definição de chaves para XML proposta por [Buneman et al., 2001], por ser independente da definição do tipo do documento e por ser mais expressiva que a definição de chaves existentes em *DTDs* (*Document Type Definition*) [Bray et al., 1998]. O algoritmo de *diff* escolhido foi o *XyDiff* [Abiteboul et al., 2002a], devido a disponibilidade do seu código fonte, bem como pela facilidade de estendê-lo para considerar chaves para XML.

O restante do trabalho está organizado da seguinte forma. A seção 2 descreve a definição de chaves para XML e a seção 3 apresenta o algoritmo XyDiff. A junção destas técnicas pelo algoritmo XKeyDiff é descrita na seção 4. Conclusões e trabalhos futuros são apresentados na seção 5.

2. Chaves para XML

O conceito de chaves é essencial para o projeto de um banco de dados [Buneman et al., 2001]. Com elas, pode-se identificar unicamente um elemento dentro de um conjunto dos dados. Além disto, as chaves podem ser úteis para verificar se os dados estão corretos, ou seja, se os dados respeitam as restrições de integridade definidas, bem como para referenciar dados externos na forma de chaves estrangeiras.

Nos últimos anos, a importância de chaves para XML tem sido reconhecida e especificações de chaves existem tanto nos padrões para definição de tipo de documento (DTD) como em *XML Schema* [Fallside, 2000]. Porém, a definição de chaves em DTDs apresenta uma série de inconvenientes. Primeiro, a chave desta definição é mais propriamente um ponteiro do que chave. Segundo, somente uma chave pode ser definida para

cada elemento e ela é composta por um único atributo. Por último, a chave precisa ser válida para todo o documento e não apenas para uma parte dele.

Para superar estas limitações, uma definição mais expressiva de chaves para XML foi proposta em [Buneman et al., 2001]. Embora algumas idéias apresentadas por esta proposta tenham sido incorporadas na versão atual do *XML Schema* [Thompson, 2002], a proposta original é independente de qualquer tipo de definição de esquema. Ela utiliza a representação em árvore do documento XML e envolve expressões de caminho, que serão detalhadas na próxima seção.

2.1. Modelo de árvore e expressões de caminho

Um documento XML pode ser modelado como uma árvore [Titel, 2003]. Logo, os conceitos de estrutura de árvore, como nodos, filhos, ascendentes e descendentes, podem ser aplicados para o documento. Um exemplo de árvore XML está ilustrado na Figura 1. As letras no interior dos nodos da árvore identificam seus tipos, que podem ser: elemento (**E**), atributo (**A**) e texto (**T**). Nodos elemento possuem apenas um nome; nodos texto não possuem nome, mas apenas um texto; já os nodos atributo possuem tanto um nome (prefixado pelo caractere ”@”) como um texto. Além disto, apenas nodos elemento podem conter filhos.

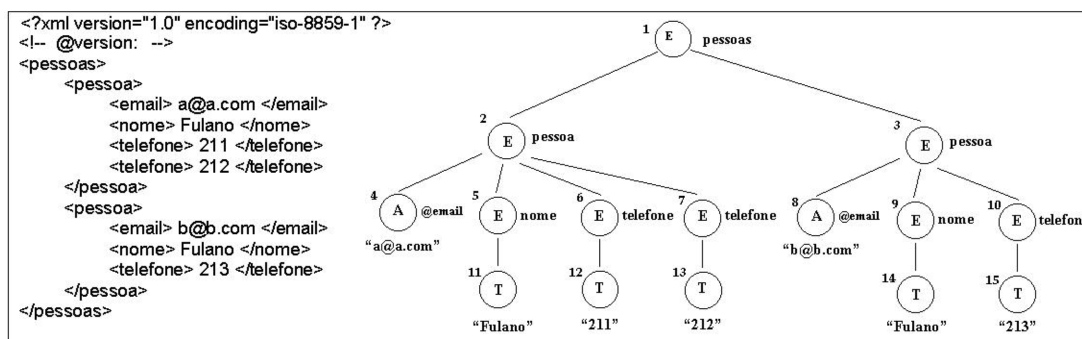

Figura 1: Um documento XML e sua representação em forma de árvore

Para percorrer a estrutura em árvore de um documento XML, torna-se necessária uma linguagem que identifique partes desta árvore. Uma *expressão de caminho* pode ser utilizada para este propósito. Uma expressão de caminho envolve nomes de nodos e descreve um conjunto de caminhos em uma árvore. A escolha da linguagem deve levar em conta o poder de expressão desejado das chaves [Buneman et al., 2001]. Nesta proposta, foi escolhido um subconjunto das expressões de caminho definidas pelo *XPath* (*XML Path Language*) [Clark and DeRose, 1999] e das expressões regulares. A sintaxe é mostrada na Tabela 1.

Tabela 1: Sintaxe utilizada para as expressões de caminho

Caracteres	Significado
ϵ	caminho vazio
e	nome de um elemento ou atributo
//	um caminho arbitrário
/	concatenação de expressões de caminho

Alguns exemplos de expressão de caminho definidos na árvore XML da Figura 1 são: $/pessoa/nome$, tendo como resultado os nodos 5 e 9, e $//telefone$, resultando nos nodos 6, 7 e 10 . Para simplificar, usamos $P//Q$ para representar a concatenação das expressões P, // e Q. Por exemplo, $//nome$ é a concatenação de ϵ, // e $nome$.

2.2. Definição de chaves para XML

Para definir uma chave em XML, é necessário que se especifique um conjunto de elementos no qual a chave irá ser definida e os itens que identificam unicamente elementos neste conjunto [Buneman et al., 2001]. Fazendo uma analogia com o modelo relacional, este conjunto seria o conjunto de tuplas identificado pelo nome de uma relação e os itens seriam as colunas formadoras da chave. Uma chave para XML pode ser definida da seguinte maneira:

$$(Q, \{P_1, ..., P_n\})$$

onde Q é uma expressão de caminho e $\{P_1, ..., P_n\}$ é um conjunto de expressões de caminho. A expressão Q define um conjunto de nodos, conjunto este chamado de *conjunto alvo*. Este conjunto é o conjunto no qual a chave irá ser definida. As expressões P_i definem os itens que juntos identificam unicamente elementos neste conjunto, e são chamadas de *caminhos chave*.

Alguns exemplos de chaves para XML são dados a seguir.

- $(//pessoa, \{id\})$: Qualquer elemento $pessoa$, independente do nível em que se encontra na árvore, se tiver um subelemento id, é identificado unicamente pelo valor do id.
- $(/empresa/empregado, \{nome, telefone\})$: Qualquer $empregado$ de qualquer $empresa$ é identificado unicamente pelo valor do subelemento $nome$ juntamente com o valor do subelemento $telefone$.

O conceito de quando uma árvore XML satisfaz uma chave para XML é dado por:

Definição: Uma árvore XML T satisfaz uma chave se e somente se para quaisquer nodos t_1, t_2 do conjunto alvo, toda vez que houver uma interseção não vazia de valores para cada caminho chave $P_1, ..., P_n$ a partir dos nodos t_1, t_2, então t_1 e t_2 são os mesmos nodos.

Considere, como exemplo, a seguinte chave: $(/pessoa, \{nome, telefone\})$. Esta chave é satisfeita pelo documento mostrado na Figura 1, pois a primeira e a segunda pessoa possuem o mesmo nome, mas não possuem números de telefone em comum. Ou seja, embora exista uma interseção não vazia no primeiro caminho chave ($nome$), existe uma interseção vazia no segundo caminho chave ($telefone$). A chave ainda seria válida para o documento mesmo se fosse retirado o telefone da segunda pessoa, uma vez que $\{211, 212\}$ (da primeira pessoa) $\bigcap$ $\{\}$(da segunda pessoa) = $\{\}$. Porém, a chave seria inválida para o documento se fosse adicionado um telefone de número 211 ou de número 212 para a segunda pessoa, uma vez que existiria uma interseção não-vazia em ambos os caminhos chave em relação à primeira pessoa.

O objetivo deste trabalho é aplicar a definição de chaves apresentada nesta seção para melhorar a qualidade da detecção de mudanças entre dois documentos XML. A próxima seção descreve o algoritmo de detecção utilizado.

3. O Algoritmo XyDiff

O algoritmo XyDiff [Abiteboul et al., 2002a] foi concebido para verificar alterações entre versões de documentos XML para um projeto que investiga *datawarehouses* dinâmicos para armazenamento de volumes maciços de dados. Por causa deste contexto, o algoritmo é eficiente em termos de velocidade e espaço de memória, em detrimento da qualidade da detecção de mudanças. O algoritmo foi implementado na linguagem C++, utilizando o *DOM* (*Document Object Model*) [Apparao et al, 1998] para a manipulação de documentos XML. O código-fonte encontra-se para *download* em [Abiteboul et al., 2002b].

3.1. Edit Script

A função essencial de um algoritmo de *diff* é tentar encontrar um *edit script* entre duas versões de um documento de tempos $t(i{-}1)$ e $t(i)$. Este *edit script* representa as mudanças entre as duas versões, ou seja, dada a versão $t(i-1)$ e o *edit script*, é possível chegar à versão $t(i)$ [Abiteboul et al., 2002a]. Um *edit script* é formado por uma seqüência de operações de edição para converter um documento em outro. As operações de edição são as operações básicas que podem ocorrer em um documento. Como o XyDiff trabalha com a representação em forma de árvore do documento XML, as operações de edição utilizadas por ele descrevem operações sobre nodos, como ilustrado na Tabela 2.

Tabela 2: Operações de edição

Operações	**Significado**
`Inserção`$(x(nome_x, valor_x),\ y)$	inserção de um nodo x, cujo nome é $nome_x$ e valor é $valor_x$, como filho folha do nodo y.
`Inserção`$(T_x,\ y)$	inserção de uma subárvore T_x com raiz em x, no nodo y.
`Remoção`(x), `Remoção`(T_x)	remoção de um nodo folha x, ou de uma subárvore T_x cujo nodo raiz é x.
`Alteração`$(x,\ novo_valor)$	alteração do valor do nodo folha x para $novo_valor$. O nodo x deve ser ou do tipo texto ou do tipo atributo.
`Movimentação`(x, pos) `Movimentação`$(T_x,\ pos)$	indica que o nodo folha x (ou a subárvore T_x com raiz em x) foi movido para a posição pos relativa aos seus irmãos.
`Movimentação`(x, y, pos) `Movimentação`$(T_x,\ y,\ pos)$	indica que o nodo folha x (ou a subárvore T_x com raiz em x) foi movido para o pai y, na posição pos dentro deste novo pai.

3.2. Estrutura do algoritmo

O algoritmo recebe como entrada duas versões de um documento XML. Uma outra entrada, opcional, é a DTD que define o esquema das versões, se houver. A saída produzida é um documento, também no formato XML (chamado *delta*), que representa o *edit script*, descrevendo as mudanças entre as versões. Esta estrutura é mostrada na Figura 2.

As etapas do algoritmo são, basicamente:

1. **Transformação das entradas**: No intuito de aproveitar a estrutura hierárquica do documento XML, o algoritmo transforma cada entrada em uma árvore.

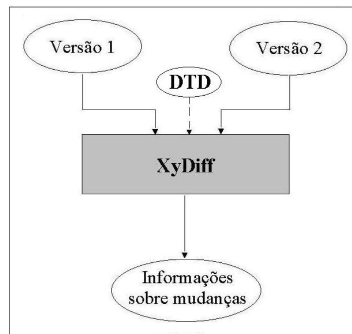

Figura 2: Estrutura do Algoritmo XyDiff

2. **Realização de casamentos**: Para detectar as mudanças entre as versões, torna-se necessária a comparação das partes em comum. Primeiramente, o algoritmo procura casar nodos que possuam atributos do tipo ID da DTD. Entre os nodos não casados, o algoritmo tenta detectar as subárvores mais amplas que foram inalteradas entre a versão nova e a antiga. Cada nodo casado pode então propagar o casamento para ascendentes e descendentes. Em sua última etapa, o algoritmo procura casar nodos que eventualmente não foram casados nas etapas anteriores.
3. **Construção do delta**: realizado os casamentos, uma análise é feita com base nestes, montando o documento (delta) que relata as mudanças entre as versões do documento XML.

Para exemplificar o funcionamento do algoritmo, considere as duas versões de um mesmo documento XML sobre atores e seus filmes, ilustradas na Figura 3.

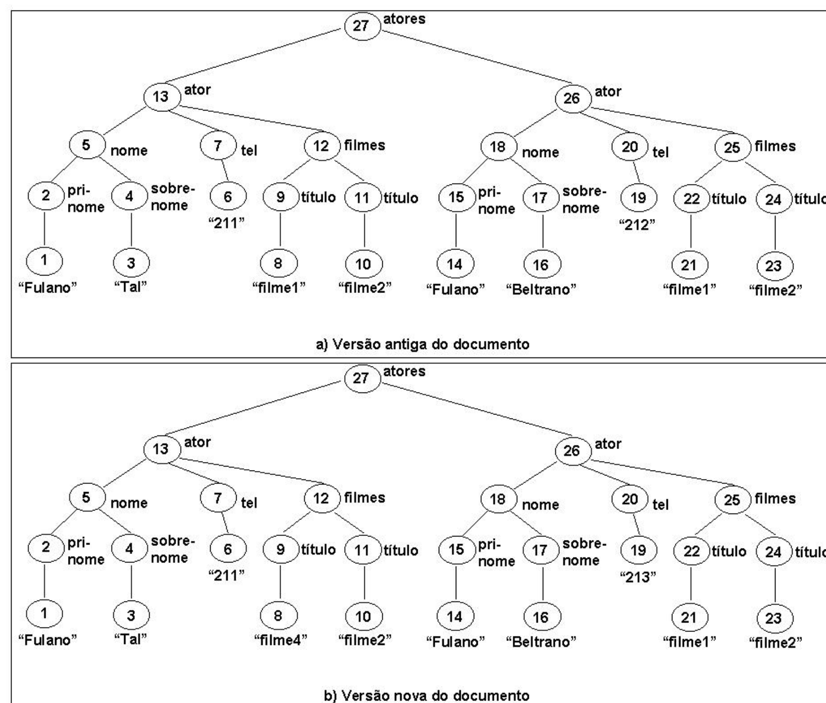

Figura 3: Versões de um documento XML

Note que as únicas diferenças entre as versões estão nos nodos 8 e 19. O valor antigo do nodo 8 era *filme1*, que passou a ser *filme4*. Já o nodo 19 mudou de *212* para

213. Logo, é natural que o resultado final de um algoritmo de *diff* gere somente estas diferenças.

Porém, isto não acontece utilizando o algoritmo XyDiff. Como ele possui uma regra de casamentos "gulosa", ou seja, que casa as subárvores mais amplas tão logo sejam encontradas subárvores semelhantes, o primeiro casamento que o algoritmo faz é do nodo *filmes* do ator Fulano Tal na árvore original com o nodo *filmes* do ator Fulano Beltrano na segunda versão, já que estas são as maiores subárvores inalteradas entre as duas versões. Isto está ilustrado na Figura 4, onde nodos casados pelo algoritmo possuem o mesmo número nas duas versões. Este casamento errôneo é propagado para o nodo pai, causando o casamento do nodo *ator* referente a Fulano Tal da versão antiga com o nodo *ator* referente a Fulano Beltrano na nova versão. Assim, o *edit script* entre as versões inclui as seguintes operações:

- movimentações: o primeiro filho *ator* de *atores*, passa a ser o seu segundo filho; os nodos *nome* e *tel* do primeiro nodo *ator* passam a ter como pai o segundo nodo *ator* no documento original; similarmente, o nodo *nome* do segundo *ator* passa a ter como pai o primeiro nodo *ator* no documento original.
- alteração: como o nodo *filmes* do ator Fulano Beltrano da versão original é casado com o nodo *filmes* do ator Fulano Tal na segunda versão, nesta o *título* do primeiro filme é alterado para *filme4*.
- remoção/inserção: como ocorreu o casamento indevido dos nodos *ator*, não foi detectada uma alteração do *tel* do ator Fulano Beltrano, mas uma remoção do nodos 19 e 20 no documento original seguida da inserção dos nodos 29 e 30.

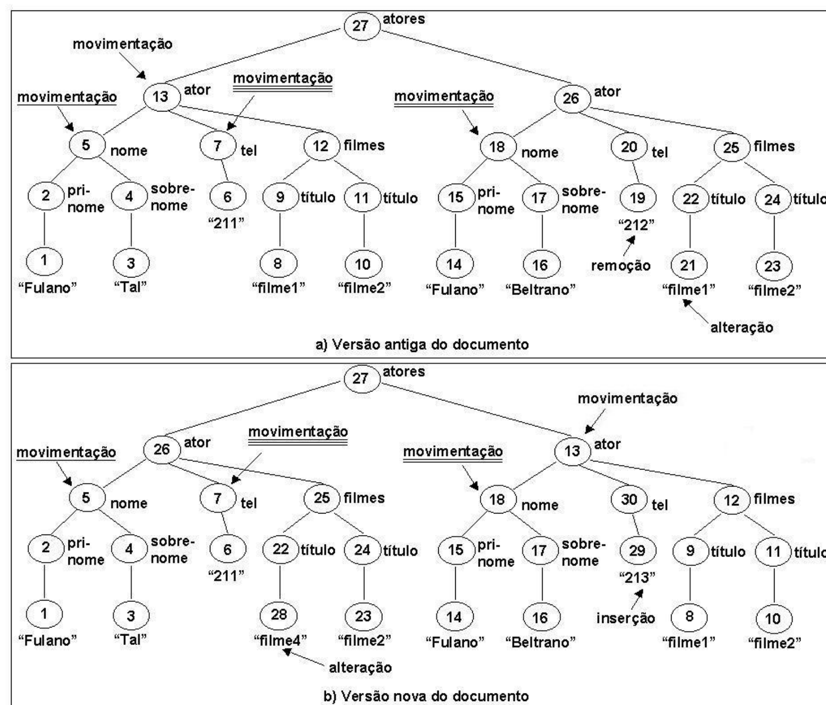

Figura 4: Diferenças entre as versões do documento XML detectadas pelo XyDiff

O tipo de alteração ilustrado neste exemplo não é incomum. Se existirem duas ou mais subárvores idênticas, qualquer alteração em uma das árvores pode resultar em casamentos errôneos, já que o contexto em que elas se encontram não é considerado.

Além disto, uma das mudanças mais comuns em aplicações com evolução de esquema freqüente é a inserção de novos atributos ou elementos. Este tipo de alteração também pode resultar em casamentos indevidos, uma vez que altera a subárvore original.

Para minimizar este problema, na próxima seção descreveremos o algoritmo XKeyDiff, que utiliza as chaves para XML, descritas na seção 2, para fazer o casamento inicial de nodos nas duas versões.

4. O Algoritmo XKeyDiff

Nesta seção apresentamos o algoritmo XKeyDiff, um algoritmo de *diff* para XML desenvolvido para trabalhar não apenas com a estrutura sintática dos dados, mas considera também a sua semântica. Ele é resultado da combinação das técnicas de chaves para XML e do algoritmo de *diff* XyDiff, descritos nas seções anteriores.

4.1. Estrutura do algoritmo

O algoritmo XKeyDiff foi implementado na linguagem C++, como um módulo do algoritmo XyDiff, utilizando e estendendo as estruturas de dados contidas nele. Este módulo é chamado entre as etapas 1 e 2 (descritas na seção 3) do algoritmo XyDiff. Ou seja, a leitura e transformação dos documentos de entrada em árvores XML continuam a cargo do XyDiff. Estas árvores, juntamente com as chaves para XML definidas pelo usuário, são passadas para o XKeyDiff, que realiza o casamento de nodos baseado nestas chaves. Terminadas as tarefas do XKeyDiff, os resultados são passados ao XyDiff, que prossegue com suas ações. A Figura 5 ilustra esta organização.

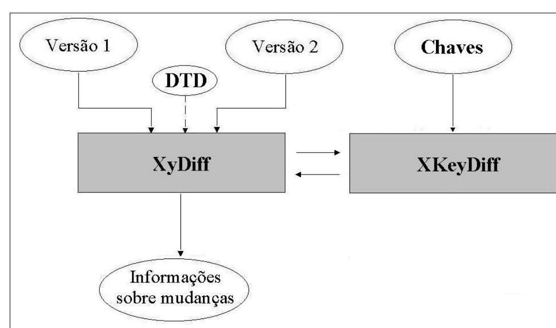

Figura 5: Estrutura do Algoritmo XKeyDiff

De forma mais detalhada, o algoritmo XKeyDiff possui as seguintes etapas:

1. **XyDiff - etapa 1:** A leitura dos documentos e a conseqüente transformação em árvores XML é executada pelo algoritmo XyDiff.
2. **Leitura das chaves:** O arquivo-texto que contém as chaves passadas pelo usuário é lido e uma estrutura de árvore com as chaves passadas é montada.
3. **Análise das chaves:** É realizada a comparação entre a árvore de chaves e as árvores XML, visando encontrar porções unicamente identificadas pelas chaves. Se for detectada tal porção, em ambas as árvores XML, os nodos envolvidos são casados.
4. **XyDiff - demais etapas:** Nesta etapa, o algoritmo XyDiff continua sua execução, tentando fazer casamentos dos nodos não casados pelo XKeyDiff.

Para ilustrar o funcionamento do algoritmo, considere novamente as duas versões mostradas na Figura 3. Suponha que a seguinte chave é definida para este documento:

(/ator, $\{$*nome, filmes/titulo*$\}$*)*

Esta chave define que o *nome* e o *título* de um filme identificam unicamente um *ator*. Baseado nesta chave, o algoritmo faz o casamento correto dos nodos *ator* nas duas versões. Este casamento é então propagado aos seus descendentes pelo algoritmo XyDiff. Desta forma, o delta resultante do algoritmo conterá apenas as alterações do *título* do primeiro filme do primeiro ator e o *telefone* do segundo ator.

É importante notar que a chave utilizada neste exemplo não pode ser expressa em termos de IDs em DTDs, já que envolve mais de um valor, tampouco no *XML Schema*, já que neste padrão as chaves devem ser definidas em termos de atributos (ou seja, *strings*) e não em termos de subárvores, como é o caso do elemento *nome* de ator, que possui subelementos *pri-nome* e *sobre-nome*.

5. Conclusão

Existem diversos estudos sobre algoritmos de *diff* para estrutura de árvore. Kuo-Chung Tai [Tai, 1979] foi a primeira proposta não exponencial. O grande problema do algoritmo é quando os nodos estão envolvidos em trocas de posições. A complexidade quadrática também é um problema. No algoritmo de Lu [Lu, 1979], quando um nodo de uma versão casa com o da outra versão, usa-se o algoritmo de *string edit* para casar seus respectivos filhos. O algoritmo de *string edit* procura um *edit script* para transformar uma string x em y. Selkow [Selkow, 1977] propõe a restrição de inserções e remoções somente para as folhas da árvore. Usando este fato no algoritmo de Lu, a complexidade é diminuída. Um algoritmo mais recente, chamado de DiffMK [Sun Microsystems, 1994], utiliza uma lista representando o documento XML, e utiliza o *diff* do Unix, não usando a estrutura de árvore. O MH-Diff [Chawathe and Molina, 1997] possui a idéia mais próxima do algoritmo XyDiff. Seu critério para casamento é baseado na subseqüência comum mais ampla para todo nodo começando das folhas do documento.

O algoritmo XKeyDiff, proposto neste trabalho, faz o uso de chaves para XML para aumentar a qualidade das mudanças detectadas por um algoritmo de *diff* para documentos neste formato. Mostramos, através de um exemplo, que esta abordagem efetivamente melhora a qualidade do resultado gerado, o que é extremamente importante para determinadas aplicações como *datawarehouse* e *data cleansing*.

O presente trabalho trouxe, como contribuição, a aplicação do conceito de chaves para XML no contexto de algoritmos de *diff* que, até então, eram baseados unicamente na estrutura sintática do documento. Não conhecemos nenhum outro trabalho na literatura que utiliza uma abordagem semântica neste contexto.

A implementação atual possui algumas limitações: considera somente chaves que devem ser satisfeitas no documento como um todo e não somente em partes dele e utiliza expressões de caminho simples (sem "//"). Estas limitações reduzem um pouco a capacidade de análise de chaves nos documentos, embora estas simplificações permitam um aumento no desempenho do algoritmo.

Trabalhos futuros, como a inclusão de chaves definidas em partes do documento e expressões de caminho mais complexas irão permitir a eliminação de etapas do algoritmo de *diff* utilizado, bem como irão prover um aumento na capacidade de análise semântica. Porém, tais implementações devem ser cautelosas quanto à eficiência do método, procurando aumentar a capacidade de análise sem levar o desempenho do algoritmo em níveis de aplicabilidade inaceitáveis. Experimentos com dados reais deverão ser realizados a fim de verificar a qualidade e a viabilidade quanto à eficiência do algoritmo. Dados científicos, como os dados biológicos, possuem formato e comportamento adequados para tais experimentos.

Referências

Abiteboul, S., Cobéna, G., and Marian, A. (2002a). Detecting Changes in XML Documents. *ICDE'02*.

Abiteboul, S., Cobéna, G., and Marian, A. (2002b). XyDiff, tools for detecting changes in XML documents. Disponível em http://www-rocq.inria.fr/~cobena/cdrom/www/xydiff/eng.htm.

Apparao et al, V. (1998). Document Object Model (DOM) Level 1 Specification. Disponível em http://www.w3.org/TR/REC-DOM-Level-1/.

Bray, T., Paoli, J., and Sperberg-McQueen, C. M. (1998). Extensible Markup Language (XML) 1.0. Disponível em http://www.w3.org/TR/REC-xml/.

Buneman, P., Davidson, S., Fan, W., Hara, C., and Tan, W. (2001). Keys for XML. *WWW'10*, pages 201–210.

Chawathe, S. and Molina, H. (1997). Meaningful change detection in structured data. *SIGMOD*, pages 26–37.

Clark, J. and DeRose, S. (1999). XML Path Language (XPath). W3C Working Draft. Disponível em http://www.w3.org/TR/xpath/.

Fallside, D. C. (2000). XML schema part 0: Primer. Disponível em http://www.w3.org/TR/xmlschema-0/.

Lu, S. (1979). A tree-to-tree distance and its application to cluster analisys. *IEEE Transaction on Pattern Analisys and Machine Intelligence*.

Maletic, J. and Marcus, A. (2000). Data Cleansing: Beyond Integrity Analysis. *Proceedings of The Conference on Information Quality (IQ2000), Massachusetts Institute of Technology, Boston, MA, USA*, pages 200–209.

Selkow, S. M. (1977). The tree-to-tree editing problem. *Information Processing Letters*, 6:184–186.

Sun Microsystems (1994). DIFFMK. Disponível em http://www.sun.com/xml/developers/diffmk.

Tai, K. (1979). The tree-to-tree correction problem. *Journal of the ACM*, 3(26):422–433.

Thompson, H. (2002). Personal communication.

Titel, E. (2003). *Teoria e problemas de XML*. Bookman.