Utilização de Chaves em Algoritmos de Detecção de Diferenças para XML

Rodrigo C. Santos [1], **Carmem Hara** [1]

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

`rodrigosantos@celepar.pr.gov.br, carmem@inf.ufpr.br`

Abstract. *Change detection algorithms for XML documents proposed in the literature have focused on the structural analysis of the document. When XML is used for data exchange, or when versions of a document are downloaded periodically, a matching process based on keys defined on the document can generate more meaningful results. In this paper, we use XML keys to determine which elements in different versions refer to the same entity in the real world, and therefore should be matched by the algorithm. We present an algorithm that extends an existing change detection algorithm with a preprocessing phase for pairing elements based on keys. An experimental study has been conducted to determine the impact of this approach on the execution time of the algorithm.*

Resumo. *Algoritmos de detecção de diferenças para documentos XML realizam uma análise estrutural do documento. Quando XML é utilizada para a troca de informações, uma comparação baseada em chaves definidas no documento pode gerar resultados mais significativos. Neste trabalho, propõe-se utilizar chaves XML para determinar quais elementos em diferentes versões representam a mesma entidade no mundo real e que portanto devem ser casados pelo algoritmo. A abordagem proposta neste artigo consiste em estender um algoritmo de detecção de diferenças existente com um pré-processamento para casar elementos baseado em chaves. Estudos experimentais foram conduzidos para determinar o impacto desta abordagem no tempo de execução do algoritmo.*

1. Introdução

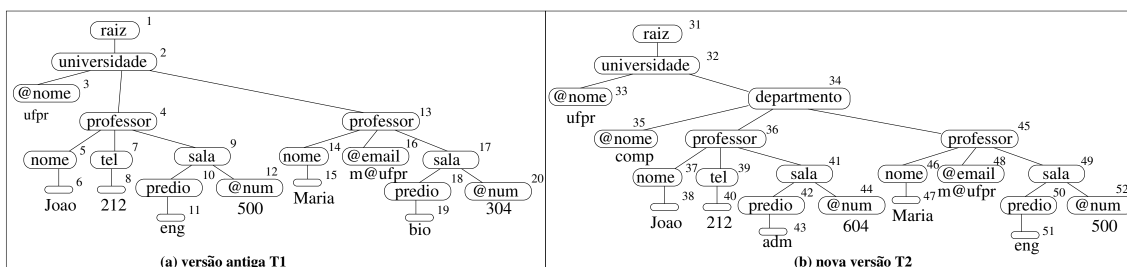
A *Linguagem de Marcação Extensível* (XML - *Extensible Markup Language*) [Bray et al. 1998] tornou-se um padrão para a troca de informações pela *Web*. Ela facilita o processo de publicação de dados, principalmente quando as informações disponibilizadas são alteradas de maneira constante. Por outro lado, os usuários podem estar interessados não somente nos valores atuais dos dados, mas também nas suas alterações. Por exemplo, eles podem querer saber quais os novos produtos adicionados a um catálogo, ou quais os produtos que tiveram seus preços alterados. Para auxiliar a tarefa de comparar duas versões de documentos XML, diversos algoritmos de detecção de diferenças, chamados de *algoritmos de diff*, foram propostos [Abiteboul et al. 2002, Al-Ekram et al. 2005, Wang et al. 2003, Xu et al. 2002].

A maioria dos algoritmos existentes são baseados em uma análise estrutural dos documentos. De forma similar aos algoritmos de diff para texto, a principal estratégia

consiste em procurar porções no texto que são idênticas em ambas as versões. Estes trechos são então “casados” e, com base nestes casamentos, uma seqüência de operações que transformam a versão antiga do documento para a nova versão é gerada como resultado do algoritmo. Esta seqüência é comumentemente chamada de *edit script* ou *delta*. Em diversas aplicações, os documentos XML não são árvores arbitrárias, mas possuem uma estrutura e semântica bem definidas. Uma estratégia baseada somente em similaridades de valores e de estrutura pode induzir o algoritmo a realizar casamentos de forma errônea.

Um estudo experimental foi conduzido para analisar os resultados de dois algoritmos de diff: XyDiff [Abiteboul et al. 2002] e X-Diff [Wang et al. 2003]. Nesse estudo, árvores XML foram modificadas através de operações de inserção, remoção e atualização, tanto de nodos internos como de folhas, para então analisar a qualidade dos resultados gerados. As conclusões desse estudo foram: (1) ambos os algoritmos são extremamente sensíveis a alterações na estrutura do documento, especialmente quando elas envolvem a inserção e remoção de nodos internos; (2) a existência de diversas subárvores idênticas ou similares induzem os algoritmos a realizar casamentos indevidamente, e estes casamentos são propagados tanto para os seus ascendentes, como para os seus descendentes. O exemplo abaixo ilustra estes problemas.

Exemplo 1 Considere os documentos XML, representados em forma de árvore, da Figura 1. Eles contêm informações sobre professores universitários. Cada `professor` possui um `nome`, uma `sala`, e opcionalmente um telefone (`tel`) ou `email`. Comparando a versão antiga com a nova, pode ser observado que tanto `Joao` como `Maria` mudaram de salas. Mais especificamente, a antiga sala de `Joao` é a nova sala de `Maria`. Se a estratégia do algoritmo de diff for procurar pelas maiores subárvores em comum nas duas versões, ele realiza o casamento dos elementos `sala` de `Joao` da versão antiga (nodo 9) com a nova sala de `Maria` (nodo 49). Este casamento é então propagado para os ascendentes, casando também os elementos `professor` (nodos 4 e 45). Ou seja, o elemento `professor` que corresponde a `Joao` é casado com aquele que corresponde a `Maria`. Consequentemente, o *edit script* contém a atualização no `nome` do `professor` que trabalha na `sala` casada. Porém, o resultado esperado, do ponto de vista semântico, seria uma atualização na sala do `professor`. Tendo conhecimento que `nome` identifica unicamente um `professor` no documento, ou seja, `nome` é chave de `professor`, casamentos de elementos `professor` baseados em seus nomes sempre geram resultados mais significativos do que casamentos baseados em similaridade de subárvores.



(a) versão antiga T1 (b) nova versão T2

Figura 1. Duas versões de árvores XML

Observe que nas versões apresentadas na Figura 1, não apenas os valores foram alterados, mas as suas estruturas também diferem. Na versão antiga, os professores estão organizados pelas universidades onde trabalham, enquanto na nova versão eles são su-

bordinados a departamentos. No estudo experimental realizado, o *edit script* gerado tanto pelo algoritmo XyDiff como pelo X-Diff contém operações que removem todas as subárvores com raiz em elementos `professor`, seguidas de inserções das mesmas subárvores como filhas dos novos elementos `departamento`. Isto se deve ao fato dos algoritmos somente casarem elementos que podem ser obtidos seguindo exatamente o mesmo caminho a partir da raiz do documento. Certamente, do ponto de vista semântico, este não é o resultado esperado, que consiste na inserção de um novo nível na árvore. De forma similar ao caso anterior, tendo conhecimento que `nome` identifica unicamente um `professor`, *independentemente do nível da árvore onde ele se encontra*, seria possível fazer os casamentos dos elementos `professor` corretamente, evitando suas remoções no resultado gerado. □

Este artigo propõe um algoritmo chamado XKeyDiff, que utiliza chaves XML [Buneman et al. 2002] para guiar a comparação entre documentos XML. Ou seja, primeiramente elementos nas versões são casados baseado em seus valores identificadores. Em seguida, é realizada uma análise estrutural para determinar suas diferenças e gerar o *edit script*. No Exemplo 1, seria dado como entrada para o algoritmo que `nome` é chave de `professor` em todo o documento. Uma estratégia baseada em chaves é natural na comparação entre bases de dados relacionais. Uma vez que XML tornou-se um padrão para troca de dados, é natural aplicar a mesma estratégia para este formato.

Contribuições. As principais contribuições deste artigo são:

- uma proposta com abordagem semântica para algoritmos de diff para XML;
- um algoritmo que realiza casamentos baseado em chaves;
- resultados experimentais que mostram o impacto da abordagem proposta.

Este trabalho detalha os resultados apresentados em [Santos e Hara 2007] com os algoritmos e o estudo experimental do impacto da proposta apresentado na Seção 4. Este é o primeiro trabalho que propõe a utilização de chaves no contexto de algoritmos de diff para XML e que gera resultados semanticamente corretos quando há alterações na estrutura do documento.

Trabalhos Relacionados. Existem diversos algoritmos de diff publicados, tanto para documentos textuais [FSF 2006] quanto para estruturas em forma de árvore [Tai 1979, Zhang et al. 1989]. XyDiff [Abiteboul et al. 2002] foi um dos primeiros algoritmos de diff propostos para XML. Ele foi projetado para ser utilizado em *datawarehouses* que armazenam um volume muito grande de dados e portanto teve como meta ser eficiente tanto no tempo como no espaço. O modelo utilizado pelo XyDiff é o de árvore ordenada. Quando o documento está acompanhado por uma DTD [Bray et al. 1998], os atributos definidos como identificadores (ID) são utilizados para casar elementos com base em seus valores. X-Diff [Wang et al. 2003] é um algoritmo que utiliza o modelo de árvore não ordenada. A proposta do algoritmo diffX [Al-Ekram et al. 2005] é realizar o casamento de fragmentos de árvore isolados, identificando similaridades entre as duas árvores nos fragmentos mais amplos possíveis. Uma estratégia diferente é utilizada pelo algoritmo KF-Diff [Xu et al. 2002], que procura, para cada nodo na árvore, caminhos distintos a partir da raiz. Quando tal caminho não existe, o que ocorre quando um nodo possui mais de um filho com o mesmo nome, este é trocado por um *campo-chave*, que é um valor contido na sua subárvore que o distingue dos demais. Embora exista, portanto, a

aplicação do conceito de chaves, este conceito é utilizado como uma técnica para derivar caminhos únicos, e não para expressar a semântica do documento, que é a proposta do presente trabalho.

Organização. O restante do trabalho está organizado da seguinte forma. A Seção 2 apresenta a definição de chaves XML. A Seção 3 descreve o algoritmo de diff semântico para XML, que é a proposta deste trabalho. Os resultados experimentais são apresentados na Seção 4. Finalmente, a Seção 5 conclui este artigo, propondo possíveis trabalhos futuros.

2. Chaves XML

Um documento XML pode ser representado em forma de árvore. Nodos da árvore podem ser de três tipos: elemento, atributo e texto, sendo que os nomes de atributos são precedidos de "@". Baseado nestes tipos, as funções $lab(n)$, e $val(n)$ são definidas da seguinte forma: se n é um nodo elemento então $lab(n)$ retorna o nome (*tag*) do elemento e $val(n)$ é indefinido; se n é um nodo atributo então $lab(n)$ representa o seu nome (*tag*) e $val(n)$, o seu valor (*string*) associado; se n é um nodo texto então $lab(n) =$ "S" e $val(n)$ é seu valor. A Figura 1 apresenta dois exemplos de árvores XML.

Para definir uma chave é necessário especificar: (1) o *contexto* no qual a chave deve ser satisfeita; (2) o conjunto *alvo*, que define o conjunto de elementos sobre os quais a chave é definida; e (3) os *valores identificadores* de elementos no conjunto alvo. Por exemplo, a chave do Exemplo 1 tem como contexto a `raiz` (todo o documento), o alvo é o conjunto de elementos `professor`, e possui um único valor identificador que é `nome`. A definição do contexto e do alvo envolve expressões de caminho. A linguagem de expressões de caminho adotada neste artigo é um subconjunto das expressões regulares [Hopcroft e Ullman 2000] e de XPath [Clark e DeRose 1999]:

$$Q \quad ::= \quad \epsilon \quad | \quad l \quad | \quad Q/Q \quad | \quad //$$

onde ϵ representa o caminho vazio, *l* é uma *tag*, "/" representa a operação de concatenação de duas expressões (*filho* em XPath), e "//" representa um caminho arbitrário de zero ou mais *tags*. Para simplificar, a concatenação de P, $//$ e Q é represetada por $P//Q$.

Seguindo a sintaxe proposta em [Buneman et al. 2002], uma chave XML é definida da seguinte forma:

$$K: \ (C, \, (T, \, \{P_1, \ldots, P_p\}))$$

onde K é o nome da chave, as expressões de caminho C e T definem o contexto (*context*) e o conjunto alvo (*target*), respectivamente, e $P_1, ..., P_p$ definem os valores identificadores (*key paths*). Neste artigo, são considerados apenas valores identificadores definidos por caminhos simples, ou seja, que não possuem "//". Uma chave é dita *absoluta* se o seu contexto é o caminho vazio ϵ, e *relativo*, caso contrário.

Exemplo 2 Usando esta sintaxe, algumas chaves sobre as árvores XML da Figura 1 podem ser definidas da seguinte forma:

- $k_1 : (\epsilon, (universidade, \{@nome\}))$: tendo como contexto a árvore toda (ϵ representa a raiz), uma `universidade` é unicamente identificada pelo seu atributo `@nome`.

- $k_2 : (universidade, (//professor, \{nome, tel\}))$: no contexto de cada subárvore com raiz em um elemento `universidade`, um `professor` é identificado pelos valores dos seus subelementos `nome` e `tel`. O elemento `professor` pode aparecer em qualquer nível da subárvore.

$\square$

Para definir a semântica de uma chave XML, a seguinte notação é utilizada: em um documento (árvore) XML, $n[\![P]\!]$ representa o conjunto de identificadores dos nodos que podem ser obtidos seguindo a expressão de caminho P a partir do nodo com identificador n. $[\![P]\!]$ é uma abreviação para $r[\![P]\!]$, onde r é a raiz da árvore. Por exemplo, na Figura 1(a), $[\![universidade]\!] = \{2\}$, $2[\![professor]\!] = \{4, 13\}$ e $[\![//nome]\!] = \{5, 14\}$. A semântica de uma chave XML é apresentada pela Definição 1.

Definição 1 *Uma árvore XML satisfaz uma chave XML* $(Q, (Q', \{P_1, \ldots, P_n\}))$ *se e somente se para todo n em $[\![Q]\!]$, e para todo n_1 e n_2 em $n[\![Q']\!]$, se existirem z_1 em $n_1[\![P_i]\!]$ e z_2 em $n_2[\![P_i]\!]$, para todo* i*, $1 \leq$ i $\leq n$, tal que $z_1 =_v z_2$, então $n_1 = n_2$.*

A definição acima envolve igualdade em árvores ($=_v$), portanto esta noção é formalizada pela Definição 2.

Definição 2 *Dada uma árvore XML T e dois nodos n_1 e n_2 em T, eles possuem o mesmo valor, representado por $n_1 =_v n_2$, se e somente se as seguintes condições forem satisfeitas: (1)* $\texttt{lab}(n_1) = \texttt{lab}(n_2)$*; (2) se n_1 e n_2 são nodos do tipo atributo ou texto então* $\texttt{val}(n_1) = \texttt{val}(n_2)$*; (3) se n_1 e n_2 são nodos do tipo elemento então: um atributo A é definido em n_1 se e somente se ele também é definido em n_2, e $val(n_1.A) = val(n_2.A)$; além disso, se $[d_1, \ldots, d_k]$ são subelementos de n_1 então n_2 possui subelementos $[d'_1, \ldots, d'_k]$, e para todo $i \in [1, k]$ existe $j \in [1, k]$ tal que $d_i =_v d'_j$.*

Por exemplo, a árvore XML da Figura 1(a) satisfaz a chave $(//professor, \{nome\})$ uma vez que $[\![//professor]\!] = \{4, 13\}$, e $4[\![nome]\!] \neq_v 13[\![nome]\!]$.

3. Um Algoritmo de Diff Semântico

Visando melhorar a qualidade semântica dos resultados gerados por algoritmos de diff para XML, nesta seção é apresentada uma nova abordagem, que utiliza a definição de chaves XML descrita na Seção 2 para realizar casamentos de elementos em duas versões de documentos XML. A arquitetura do algoritmo XKeyDiff, que implementa esta abordagem, é mostrado na Figura 2.

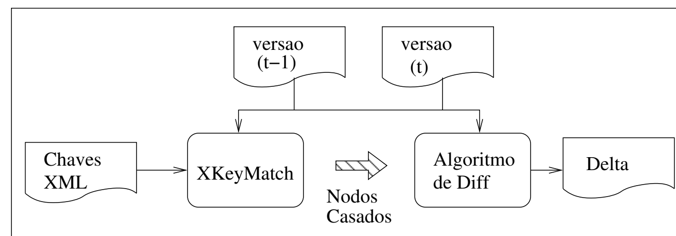


Figura 2. Algoritmo XKeyDiff

O algoritmo XKeyDiff recebe como entrada duas versões, v_{t-1} e v_t, de árvores XML e um conjunto de chaves XML Σ que são satisfeitas por ambas as versões. A saída

é um arquivo delta ou *edit script*. O algoritmo é composto por dois módulos: o XKeyMatch e um algoritmo de diff. O objetivo do módulo XKeyMatch é gerar um conjunto Γ de pares de elementos $[n_1, n_2]$, onde n_1 é um nodo em v_{t-1} que refere-se à mesma entidade que o nodo n_2 em v_t de acordo com Σ. O conjunto Γ é então dado como entrada para o módulo de diff que, baseado nos casamentos, compara as versões v_{t-1} e v_t e gera o *edit script*. Nesta abordagem, qualquer algoritmo de diff existente na literatura pode ser utilizado, bastando implementar uma comunicação desta abordagem com o algoritmo de diff escolhido. Assim, o restante desta seção é dedicado à apresentação do algoritmo de realização de casamentos baseado em chaves, chamado de XKeyMatch. O algoritmo XKeyMatch é baseado na construção de um autômato finito determinístico (AFD) a partir do conjunto Σ de chaves XML, chamado de KeyDFA(Σ). Usando este AFD é possível processar todas as chaves em Σ ao mesmo tempo, e todos os casamentos baseados em Σ podem ser gerados com um único percurso em pré-ordem em v_{t-1} e v_t. Mais especificamente, cada estado do AFD representa um conjunto de caminhos e armazena informações sobre todas as chaves que podem ser definidas sobre os nodos alcançados percorrendo tais caminhos. Assim, cada passo no percurso da árvore XML corresponde a uma mudança de estado do autômato; as informações sobre chaves armazenadas em cada estado são utilizadas para coletar os nodos que são candidatos aos casamentos. Depois de coletar todos os candidatos, o algoritmo XKeyMatch cria pares de nodos, um de cada versão, de acordo com seus valores identificadores. Estes passos do algoritmo são apresentados na Figura 3. Dadas duas árvores XML T_1 e T_2, e um conjunto de chaves XML Σ, o algoritmo primeiro constrói o autômato KeyDFA(Σ) (Linha 1). Depois disso, os candidatos são selecionados chamando a função `get_candidates` para ambas as árvores (Linhas 2 e 3). O conjunto de casamentos resultantes é gerado pela função `match`, que compara os valores dos candidatos previamente coletados (Linha 4). As próximas seções apresentam em detalhe cada um destes passos.

Função `XKeyMatch`
Entrada: árvores XML T_1, T_2, e um conjunto de chaves XML Σ
Saída: um conjunto de pares de nodos casados
1. *KeyDFA*:= *DFA*(Σ);
2. $candidates[T_1]$:= *get_candidates*(T_1, *KeyDFA*);
3. $candidates[T_2]$:= *get_candidates*(T_2, *KeyDFA*);
4. **return** (*match*(*candidates*[T_1], *candidates*[T_2], T_1, T_2);

Figura 3. Algoritmo XKeyMatch

3.1. Construção do Autômato

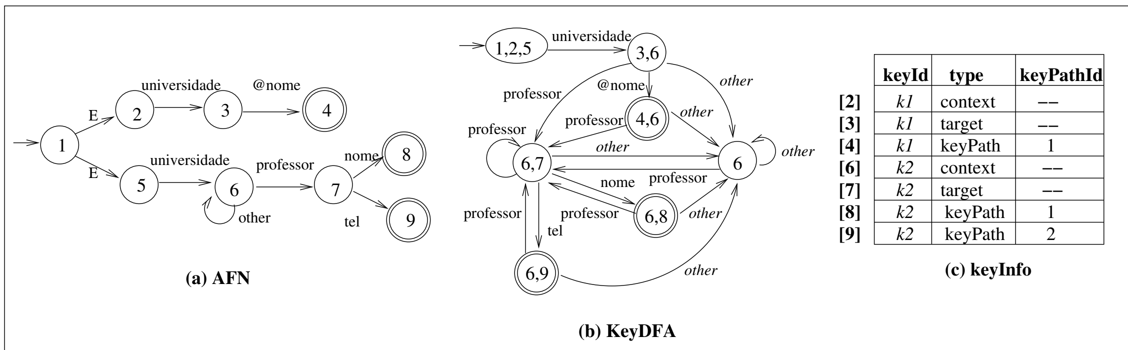
Dado um conjunto de chaves XML Σ, este passo do algoritmo XKeyMatch gera um autômato finito determinístico, chamado de KeyDFA(Σ). Os estados deste autômato armazenam informações para processar todas as chaves em Σ com um único percurso sobre a árvore XML T.

Seja $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, onde cada σ_i é da forma $(Q_i, (Q'_i, \{P_i^1, ..., P_i^{n_i}\}))$. A construção de KeyDFA(Σ) começa com a definição de um autômato finito não determinístico (AFN) para cada chave σ_i em Σ. Inicialmente, são construídos AFNs para cada caminho p em $\{Q_i, Q'_i, P_i^1, \ldots, P_i^{n_i}\}$, definidos como $M(p) = (N_p, L_p \cup \{other\}, \delta_p, S_p, F_p)$, onde N_p é um conjunto de estados, L_p é o alfabeto, δ_p é a função de transição,

S_p é o estado inicial e F_p é o conjunto de estados finais. Aqui, "*other*" é um caracter especial que casa com qualquer caracter. Estes autômatos têm "estrutura linear". Ou seja, se $p = l_1/\dots/l_m$ então $q_0 = S_p$, $q_m = F_p$ e $\delta_p(q_j, l_{j+1}) = q_{j+1}$ para todo j, $0 \leq j < m$. Se p contiver "//" então existe uma transição partindo de um estado para ele mesmo com "*other*". Ou seja, se $p = \dots//l_j\dots$ para algum j então $\delta_p(q_{j-1}, other) = q_{j-1}$. Os estados finais destes AFNs contêm informações sobre a chave considerada para a sua construção, armazenadas em uma estrutura chamada de *keyInfo*. Seja $F = \{F_{Q_i}, F_{Q'_i}, F_{P_i^1}, \dots, F_{P_i^{n_i}}\}$. Para cada $f \in F$, $keyInfo[f]$ contém as seguintes informações: (1) `keyId`: o identificador da chave σ_i; (2) `type`: o valor deste campo é *context* se $f = F_{Q_i}$, *target* se $f = F_{Q'_i}$ e *keyPath*, caso contrário; (3) `keyPathId`: o número de cada valor identificador, ou seja, se $f = F_{P_i^j}$ então $keyInfo[f].keyPathId = j$.

O AFN de cada chave σ_i é obtido fazendo com que o estado final de $M(Q_i)$ coincida com o estado inicial de $M(Q'_i)$, e o estado final de $M(Q'_i)$ coincida com os estados iniciais de $M(P_i^k)$, $1 \leq k \leq n_i$. Finalmente, o AFN para todas as chaves em Σ é construído criando-se um novo estado inicial com transições-ϵ para os estados iniciais de todos os $M(\sigma_i)$, $1 \leq i \leq n$. Dado o AFN $M(\Sigma)$, o autômato KeyDFA(Σ) é obtido utilizando o algoritmo padrão de transformação baseado na construção de conjuntos de estados [Hopcroft e Ullman 2000].

Exemplo 3 Considere o conjunto Σ de chaves XML definido no Exemplo 2. A Figura 4(a) mostra o AFN construído a partir destas chaves e a estrutura *keyInfo* correspondente é apresentada na Figura 4(c). O autômato resultante da transformação para um AFD é apresentado na Figura 4(b). É importante observar que, embora em $M(\Sigma)$ cada estado contenha informações de no máximo uma chave em Σ, após sua conversão para um AFD, cada estado de KeyDFA(Σ) contém informações sobre todos os estados correspondentes de $M(\Sigma)$. Por exemplo, na Figura 4(b), $keyInfo(\{3,6\}) = \{keyInfo[3], keyInfo[6]\}$. $\square$



	keyId	type	keyPathId
[2]	*k1*	context	––
[3]	*k1*	target	––
[4]	*k1*	keyPath	1
[6]	*k2*	context	––
[7]	*k2*	target	––
[8]	*k2*	keyPath	1
[9]	*k2*	keyPath	2

Figura 4. Construção do AFD

A construção do AFD descrita nesta seção é similar àquela definida em [Green et al. 2004], que trata do processamento expressões XPath em *streams* de dados XML, na qual o AFD é construído "preguiçosamente"(*lazily*). Esta abordagem foi escolhida porque o aumento do número de expressões avaliadas pode ocasionar um crescimento exponencial no número de estados do AFD. O mesmo problema existe no presente trabalho. Porém, geralmente o número de chaves definidas para cada documento é pequeno. Além disso, uma vez que alterações em chaves são raras, se versões de um mesmo documento XML são periodicamente avaliadas, o AFD pode ser armazenado localmente, ao invés de ser reconstruído a cada execução do algoritmo.

3.2. Seleção de Candidatos

Uma vez construído o autômato KeyDFA(Σ), ele é utilizado pelo algoritmo XKeyMatch para processar cada uma das árvores XML dadas como entrada a fim de coletar informações sobre os possíveis nodos candidatos a casamentos de acordo com Σ. O algoritmo de seleção de candidatos é mostrado na Figura 5.

Função *get_candidates*
Entrada: uma árvore T, KeyDFA$(\Sigma) = (Q, A, \delta, q_0, F)$
Saída: um conjunto de nodos candidatos a casamento
1. $keyValues := \{\}$;
2. **for each** $k \in \Sigma$ **do**
3. $\quad keyVal[k].contextNodes := \{\}$;
4. *traverse_dfa_tree*$(q_0, r(T), keyValues)$;
5. *candidates*:= $\{\}$;
6. **for each** $c \in nest_{keyPaths=\{keyPathId, keyPathNodeId\}}(keyValues)$ **do**
7. $\quad$ **if** $c.keyId = k_1$, $k_1 = (Q, (Q', \{P_1, \ldots, P_n\}))$, e para todo $i \in [1, n]$, $c.keyPaths$ contém um elemento com $keyPathId = i$ **then**
8. $\quad\quad$ *candidates*:= *candidates* $\cup \{c\}$
9. **return** *candidates*;

Procedimento *traverse_dfa_tree*
Entrada: estado atual $state$, nodo corrente $node$, um conjunto de valores identificadores $keyValues$
Saída: novos elementos adicionados a $keyValues$
10. **for each** $v \in keyInfo[state]$ **do**
11. $\quad k := v.keyId$;
12. $\quad$ **if** $v.type = context$ **then**
13. $\quad\quad$ **if** existe $c \in keyVal[k].contextNodes$ tal que $node$ é um descendente de c **then**
14. $\quad\quad\quad keyVal[k].contextNodes := keyVal[k].contextNodes \cup node$;
15. $\quad\quad$ **else** $keyVal[k].contextNodes := \{node\}$;
16. $\quad$ **else if** $v.type = target$ **then**
17. $\quad\quad keyVal[k].targetNode := node$;
18. $\quad$ **else**
19. $\quad\quad keyVal[k].keyPathNode := node$;
20. $\quad\quad keyVal[k].keyPathId := v.keyPathId$;
21. $\quad\quad$ **for each** $c \in keyVal[k].contextNodes$ **do**
22. $\quad\quad\quad keyValues := keyValues \cup \{[k, c, keyVal[k].targetNode, keyVal[k].keyPathNode, keyVal[k].keyPathId]\}$;
23. **for each** c que é filho de $node$ **do**
24. $\quad$ **if** $\delta(state, lab(c)) = new_state$ **then**
25. $\quad\quad$ *traverse_dfa_tree*$(new_state, c, keyValues)$;
26. $\quad$ **else**
27. $\quad\quad$ *traverse_dfa_tree*$(\delta(state, other), c, keyValues)$;

Figura 5. Algoritmo de seleção dos candidatos

Seja T uma destas árvores, e KeyDFA(Σ) = (Q, A, δ, q_0, F). Começando com a raiz de T e o estado inicial q_0, T é percorrida em pré-ordem e cada passo do percurso corresponde a um passo no processamento com o autômato. Durante o percurso, informações sobre as chaves são coletadas usando os dados armazenados em cada estado q do autômato. Por exemplo, suponha que o estado atual de KeyDFA seja q durante o processamento do nodo n em T. Se $keyInfo[q]$ contiver informações de um valor identificador de uma chave XML k, então o valor de n é chave para algum nodo n_t ancestral de n e, portanto, pode-se associar n_t a $val(n)$. Observe que k pode conter mais de um valor identificador. Neste caso, n_t é identificado como um candidato para casamento somente

se estiver associado a valores para todos os valores identificadores. Neste caso, dizemos que n_t é "chaveado" por k. Observe que quando a chave é relativa, também é necessário manter o contexto no qual esta chave é definida. Isto porque se um nodo n_t é chaveado relativamente a um nodo contexto n_c, esta chave não pode identificar unicamente n_t a menos que n_c também seja chaveado. Em outras palavras, precisamos de nodos que sejam unicamente identificados até a raiz da árvore.

Para manter as informações necessárias para determinar se um nodo é chaveado, associamos a cada chave $k = (Q, (Q', \{P_1, \ldots, P_n\}))$ em Σ uma estrutura, chamada de `keyVal[`k`]`, contendo os seguintes campos: (1) `contextNodes`: um conjunto de nodos em $[\![Q]\!]$ em um caminho único a partir da raiz; ou seja, se `contextNodes` contém n_1 e n_2, n_1 é descendente ou ascendente de n_2; (2) `targetNode`: o último nodo visitado em $n[\![Q']\!]$ para algum n em `contextNodes`; (3) `keyNode`: o último nodo visitado em `targetNode`$[\![P_i]\!]$; (4) `keyPathId`: número do valor identificador i, $1 \leq i \leq n$.

O algoritmo começa pela raiz da árvore $r(T)$ e pelo estado inicial q_0 de KeyDFA(Σ) (Linhas 1 a 4 do algoritmo). Os valores coletados por este procedimento são então agrupados de forma que cada grupo contenha todos os nodos correspondentes a valores identificadores ($[keyNode, keyPathId]$) associados a um mesmo nodo alvo de acordo com a mesma chave ($[keyId$, $contextNode$, $targetNode]$). Assim, para determinar se um nodo é chaveado por k basta verificar se em cada grupo há nodos para todos os valores identificadores de k (Linhas 5 a 8). O procedimento `traverse_dfa_tree` é responsável pela coleta de nodos que são valores identificadores à medida que a árvore XML é percorrida, colocando o resultado na variável `keyValues`. Observe que um mesmo nodo da árvore pode ser ao mesmo tempo contexto, alvo ou valor identificador de diferentes chaves em Σ, e esta informação encontra-se armazenada na estrutura $keyInfo[s]$, onde s é um estado em KeyDFA(Σ). Assim, se o estado atual do autômato KeyDFA(Σ) é s quando o nodo n da árvore está sendo processado, então cada elemento v em $keyInfo[s]$ é utilizado para preencher os campos de $keyVal[k]$, onde k é o identificador da chave em $keyInfo$ ($keyInfo[v].keyId$). Sempre que um nodo correspondente a um valor identificador de k é encontrado, todas as informações coletadas em $keyVal[k]$ são inseridas em `keyValues` (Linhas 10 to 22). Para dar continuidade ao percurso na árvore, o procedimento `traverse_dfa_tree` é chamado recursivamente para cada um dos filhos do nodo corrente (Linhas 23 a 27).

Exemplo 4 Considere as árvores XML T_1 e T_2 ilustradas na Figura 1(a) e 1(b), respectivamente, e o autômato KeyDFA(Σ) da Figura 4(b). O resultado da função `get_candidates` sobre T_1 e T_2 é mostrado na Figura 6(a) e (b). A primeira linha da tabela 6(a) foi coletada utilizando a chave k_1, enquanto o segundo candidato foi coletado de acordo com a chave k_2. Observe que o nodo 13 (um nodo `professor`) não foi incluído no conjunto, uma vez que ele não possui valor para o valor identificador `tel`. $\square$

keyId	context Nodes	target Node	[keyNode, keyPathId]
k_1	1	2	{[3,1]}
k_2	2	4	{[5,1], [7,2]}

(a) `get_candidates`(T_1, `keyDFA`)

keyId	context Nodes	target Node	[keyNode, keyPathId]
k_1	31	32	{[33,1]}
k_2	32	36	{[37,1], [39,2]}

(b) `get_candidates`(T_2, `keyDFA`)

Figura 6. Seleção de candidatos em T_1 e T_2

3.3. Casamentos

Uma vez coletados os nodos candidatos a casamentos das árvores XML T_1 e T_2, o procedimento `match` procura por nodos nestes conjuntos que possuem o mesmo valor de chave. Ou seja, são procurados nodos n_1 nos candidatos de T_1 e n_2 nos candidatos de T_2 que são identificados pela mesma chave k e que coincidam (ou seja, possuem igualdade de valor $=_v$) em todos os valores identificadores. Caso k seja uma chave relativa, para que aconteça o casamento de n_1 com n_2, é necessário que os contextos nos quais n_1 e n_2 se encontram também estejam casados. O procedimento `match` é mostrado na Figura 7.

Procedimento $match$
Entrada: listas de candidatos $candidates_1$ e $candidates_2$ e árvores XML T_1 e T_2
Saída: um conjunto de pares de nodos casados

1. $candidatePairs := \{\}$;
2. **for each** $c_1 \in candidates_1$ **do**
3.   **for each** $c_2 \in candidates_2$ **do**
4.   **if** $c1.keyId = c2.keyId$ **then**
5. $keyPairs := \{\}$;
6.   **for each** $p_1 \in c1.keyPaths$ **do**
7.   **for each** $p_2 \in c2.keyPaths$ **do**
8.   **if** $p_1.keyPathId = p_2.keyPathId$ e
                     $val(p_1.keyPathNode) = val(p_2.keyPathNode)$ **then**
9. $keyPairs := keyPairs \cup \{[keyPathId : p_1.keyPathId,$
   $keyPathNode_1 : p_1.keyPathNode,$
   $keyPathNode_2 : p_2.keyPathNode]\}$;
10.   **if** $keyPairs$ contém todos os valores identificadores de $c_1.keyId$ **then**
11. $candidatePairs := candidatePairs \cup \{[$`keyId` $:p_1.keyId,$
                         `v1` $:[c_1.contextNode, c_1.targetNode],$
                         `v2` $:[c_2.contextNode, c_2.targetNode],$ `keyPairs` $:keyPairs]\}$;
12. $matches := \{\}$;
13. $AuxMatches := \{[\{r(T_1)\}, \{r(T_2)\}]\}$;
14. **while** existem casamentos **do**
15.   **for each** $p \in candidatePairs$ **do**
16.   **if** existe $[s_1, s_2]$ em $AuxMatches$ tal que $p.contextNode_1 \in s_1$ e $p.contextNode_2 \in s_2$ **then**
17. $matches := matches \cup \{[p.targetNode_1, p.targetNode_2]\}$;
18.   **for each** $i \in \{1, 2\}$ **do**
19. $t_i := \{\}$;
20. $node := p.targetNode_i$;
21.   **while** $node \neq p.contextNode_i$ **do**
22. $t_i := t_i \cup \{node\}$;
23. $node$:= nodo pai de $node$ em T_i;
24. $AuxMatches := AuxMatches \cup [t_1, t_2]$;
25.   **for each** $v \in p.keyPairs$ **do**
26. $matches := matches \cup \{[v.keyPathNode1, v.keyPathNode2]\}$;
27.   **for each** n_1 e n_2 que são nodos correspondentes nas subárvores
                   com raiz em $v.keyPathNode1$ e $v.keyPathNode2$ **do**
28. $matches := matches \cup \{[n_1, n_2]\}$;
29.   **for each** n_1 e n_2 que são nodos correspondentes no caminho de $p.v1.targetNode$
                   para $v.keyPathNode1$, e de $p.v2.targetNode$ para $v.keyPathNode2$ **do**
30. $matches := matches \cup \{[n_1, n_2]\}$;
31. **return** $matches$;

Figura 7. Algoritmo de realização de casamentos

O procedimento recebe, como entrada, os conjuntos de candidatos $candidates_1$ e $candidates_2$, de T_1 e T_2, respectivamente, e obtém o conjunto de casamentos em dois passos. Primeiro, para cada chave k em Σ, os valores dos elementos c_1 em $candidates_1$ e c_2 em $candidates_2$ são comparados caso eles se refiram à mesma chave k (Linhas 1 a 9). Se os valores coincidirem para todos os valores identificadores em k, então, uma entrada na variável $candidatePairs$ é criada contendo as seguintes informações (Linhas 10 a 11): (1) `keyId`: identificador da chave k; (2) $\texttt{contextNode}_1$: o nodo contexto em T_1; (3) $\texttt{targetNode}_1$: o nodo alvo em T_1; (4) $\texttt{contextNode}_2$: o nodo contexto em T_2; (5) $\texttt{targetNode}_2$: o nodo alvo em T_2; (6) `keyPairs`: um conjunto de registros $[\texttt{keyPathId}, \texttt{keyPathNode}_1, \texttt{keyPathNode}_2]$, onde $\texttt{keyPathNode}_1$ e $\texttt{keyPathNode}_2$ são nodos identificadores em T_1 e T_2 que possuem o mesmo valor. Em seguida, o algoritmo verifica se os contextos de cada par de candidatos já foram casados. Para auxiliar nesta verificação, é utilizada uma variável chamada $AuxMatches$ (Linhas 12 a 24). Por fim, o procedimento inclui, para cada par de nodos casados que são valores identificadores, todos os pares de descendentes correspondentes nas duas versões e também todos os pares de ascendentes até os nodos dos seus nodos alvo (Linhas 25 a 30).

Exemplo 5 Considere novamente as árvores XML T_1 e T_2 da Figura 1 e a saída da função `get_candidates` sobre T_1 e T_2 da Figura 6(a) e (b). Depois de comparar os valores dos nodos chaveados, o conjunto de pares que são candidatos são aqueles mostrados na Figura 8. O primeiro candidato é incluído no conjunto porque os nodos 2 e 32 são nodos `universidade` em T_1 e T_2, respectivamente, com o mesmo valor de `@nome`. Similarmente, o segundo candidato é incluído no resultado porque os nodos 4 e 36 são nodos `professor` que coincidem tanto em `nome` como em `tel`. O conjunto de casamentos resultantes começa com o par $[1, 31]$, que são as raízes de T_1 e T_2. Para processar o primeiro par de candidatos, $[2, 32]$, primeiramente é verificado se os seus contextos já foram casados. Como a chave k_1 tem como contexto as raízes das árvores, que já foram casadas, o par $[2, 32]$ é inserido no resultado. Este casamento é propagado para os seus descendentes e o par $[3, 33]$ (nodos `@nome`) são também incluídos no resultado. Para o segundo par de candidatos, como os seus contextos, o par $[2, 32]$, já fazem parte dos casamentos realizados, $[4, 36]$ também é casado e inserido no resultado. A propagação deste casamento inclui no conjunto os seguintes pares: $[5, 37]$, $[6, 38]$ (nodos `nome`), $[7, 39]$ e $[8, 40]$ (nodos `tel`). □

keyId	context node$_1$	target node$_1$	context node$_2$	target node$_2$	[keyPathId, keyNode$_1$, keyNode$_2$]
1	1	2	31	32	{[1,3,33]}
2	2	4	32	36	{[1,5,37], [2,7,39]}

$match(candidates_1, candidates_2, T_1, T_2)$

Figura 8. Casamentos em T_1 e T_2

4. Estudo Experimental

Para determinar o custo adicional que a abordagem semântica proposta neste trabalho tem sobre o desempenho de um algoritmo de diff tradicional, diversos experimentos foram realizados. Para a composição do XKeyDiff, o algoritmo XyDiff foi utilizado como algoritmo de diff, bem como foi implementado o algoritmo XKeyMatch, em C++. Para a execução dos experimentos, foram elaborados quatro tipos de documentos XML, cada qual possuindo características comumente encontradas em documentos reais. O primeiro

tipo, chamado de *atores*, contém dados sobre atores de cinema, sendo tais dados coletados do site Imdb.com. O segundo tipo representa dados sobre campeonatos de futebol (*campeonatos*), contendo informações relativas ao local do evento, período de realização, times e jogadores participantes. O terceiro tipo, identificado como *planejamento*, possui dados sobre uma empresa, seus funcionários e as atividades designadas aos mesmos. Por último, o quarto tipo (*universidades*) possui dados sobre universidades, contendo informações sobre localização, setores, departamentos e professores. Tendo como base estes documentos, foram geradas 20 versões de cada tipo de documento, onde 5 destas versões contêm somente alterações do documento base, 5 versões contêm inserções, 5 versões contêm remoções e 5 versões contêm somente movimentações. Assim, foi gerada uma massa de dados de teste contendo 80 versões criadas a partir dos 4 tipos base. Cada versão possui, em média, o tamanho de 15Kb.

Os experimentos foram realizados em um computador com processador AMD Sempron 32 bits de 2 Ghz, com 1Gb de memória RAM. O sistema operacional utilizado foi o Linux e a versão do compilador de C++ utilizado foi o gcc 3.3. Para a validação dos documentos XML, bem como para disponibilizar as bibliotecas do DOM, foi utilizado o software Xerces-C++ Parser.

4.1. Experimento 1

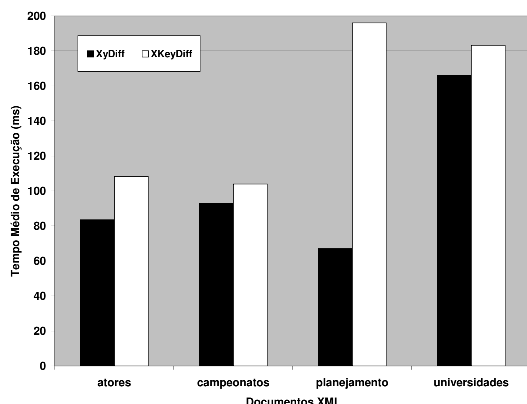
O primeiro experimento teve como objetivo avaliar o impacto do pré-processamento das versões dos documentos XML utilizando o algoritmo XKeyMatch, em comparação com a execução do algoritmo XyDiff isoladamente. O experimento consiste na execução tanto do algoritmo XyDiff como do XKeyDiff tendo, como entrada, um tipo base e uma de suas versões. Isto totalizou 80 execuções de cada algoritmo, sendo que para a execução do XKeyDiff apenas uma chave XML foi dada como entrada. A Figura 9(a) mostra os tempos de execução de cada um dos tipos de documento. No gráfico, o eixo horizontal representa os tipos e o eixo vertical refere-se ao tempo médio de execução (em milisegundos) de todas as suas versões. As barras pretas e brancas representam, respectivamente, o tempo relativo ao algoritmo XyDiff e o tempo relativo ao algoritmo XKeyDiff.

Como era esperado, o tempo médio de processamento do XKeyDiff foi maior que o XyDiff para todos os tipos. O acréscimo foi de 29% para os documentos do tipo *atores*, 11% para os documentos do tipo *campeonatos*, 192% para os documentos do tipo *planejamento* e 10% para os documentos do tipo *universidades*. Assim, em média, houve um aumento de 61% nos tempos de processamento com relação ao XyDiff. Este acréscimo decorre do custo de construção do AFD, bem como do processamento dos documentos utilizando este autômato. Uma observação relevante neste experimento foi o acréscimo demasiadamente grande no tempo de execução do XKeyDiff para os documentos do tipo *planejamento*. Este comportamento deve-se a estrutura deste tipo de documento e a chave XML utilizada neste teste. Como esta chave possibilitou a identificação de uma parte significativa de cada versão deste tipo de documento, o processamento do XKeyMatch sofreu um acréscimo de tempo maior do que nos outros tipos, que utilizaram chaves XML que identificaram somente uma pequena porção de cada documento. Conforme será visto no Experimento 2, esta mesma situação ocorre para documentos do tipo *campeonatos*, quando uma chave XML adicional é dada como entrada para o algoritmo.

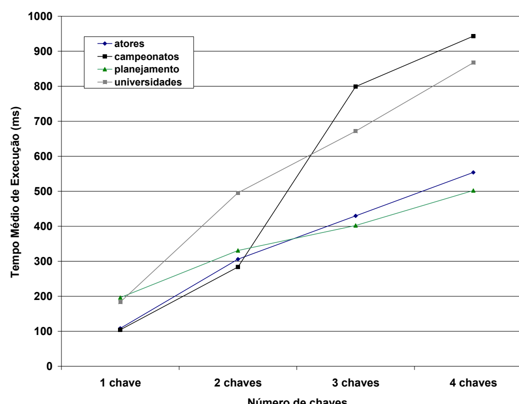
Embora a utilização do algoritmo XKeyDiff tenha apresentado um maior tempo de processamento para grande parte da massa de testes, constatou-se que, em 11% dos casos

avaliados, o tempo de execução do XKeyDiff foi melhor que o XyDiff, representando um decréscimo médio de 8% no tempo de execução. Através da análise dos dados, constatou-se que isto ocorre quando a chave XML é definida em um nível alto na árvore, ou seja, próximo à raiz. Neste caso, o XKeyMatch não precisa fazer um percurso completo na árvore XML, porque uma vez constatado que nenhum candidato pode ser encontrado em um ramo da árvore, ele não é percorrido pelo algoritmo. Além disso, como os casamentos são propagados para os descendentes dos nodos, quanto mais alto o nodo casado, maior a quantidade de casamentos adicionais gerados pela propagação. Isto desonera o tempo de processamento posterior do documento com o XyDiff, pois tais nodos não precisam ser mais considerados por esse algoritmo.

Este experimento mostrou que o tempo de percurso da árvore XML com o AFD construído para o processamento de chaves é significativamente mais alto que o seu percurso apenas com o algoritmo XyDiff, que é baseado no cálculo de uma função *hash* para cada nodo. Porém cabe ressaltar que o objetivo do XKeyDiff é na geração de resultados semanticamente mais significativos. Além disso, o algoritmo XyDiff foi escolhido tanto na implementação do XKeyDiff como na análise comparativa por ele ter sido desenvolvido tendo como meta ser eficiente tanto no tempo como no espaço.



(a) XyDiff versus XKeyDiff



(b) Impacto do número de chaves

Figura 9. Resultados dos experimentos

4.2. Experimento 2

Visto que quanto maior a quantidade de elementos casados baseado em chaves, maior a qualidade do resultado do algoritmo XKeyDiff, o segundo experimento teve como objetivo determinar o impacto que o número de chaves especificadas como entrada tem no desempenho do sistema. O experimento consiste na execução do algoritmo XKeyDiff tendo como entrada as oitenta versões criadas a partir dos quatro tipos base. Além disso, foram especificadas para cada versão, uma, duas, três e quatro chaves XML de entrada, totalizando 320 execuções do algoritmo. Os resultados gerados por estas execuções são mostrados na Figura 9(b), onde o eixo horizontal representa o número de chaves XML utilizadas, o eixo vertical representa o tempo em milisegundos (ms) e cada linha representa o tempo médio de execução das 20 versões existentes de cada tipo de documento.

Conforme pode ser verificado no gráfico, a utilização de um número maior de chaves XML causa um incremento no tempo de execução do XKeyDiff. A média destes

tempos de execução apontam um acréscimo de aproximadamente 148% na utilização de 2 chaves, sendo que a utilização gradativa de mais chaves (3 e 4) apresentem um acréscimo de, respectivamente, 69% e 25%. Este resultado demonstra que, embora o tempo de execução sofra uma acréscimo na utilização gradativa de chaves XML, este acréscimo tende a diminuir. Ou seja, quanto maior o número de chaves XML utilizadas, menor o acréscimo de tempo entre estes. Esta situação ocorre devido ao fato de que, quanto maior o número de chaves, a probabilidade de elas compartilharem expressões de caminho na sua definição é maior. Assim, elas também compartilham os estados do AFD gerado pelo algoritmo XKeyDiff. Observe que cada estado do AFD representa um conjunto de expressões de caminho e, durante o percurso da árvore XML, todos os nodos que podem ser atingidos por estas expressões são processadas por este estado. Assim, se o acréscimo de uma chave causa a criação de um pequeno número de estados, apenas uma pequena porção adicional da árvore será processada utilizando o AFD. Ou seja, seu impacto no tempo de processamento não é muito grande.

Um fator apresentado nos resultados da Figura 9(b) é o comportamento distinto da curva dos tempos de execução dos documentos do tipo *campeonatos*, no qual houve um acréscimo demasiadamente grande no tempo de execução utilizando 3 chaves. De forma similiar ao caso descrito no Experimento 1 para os documentos do tipo *planejamento*, a terceira chave definida causou a identificação de uma parte significativa dos documentos, que não era processada pelas duas primeiras chaves definidas, onerando, assim, o tempo de execução do algoritmo. Uma observação importante é que nos tipos de documentos XML utilizados no experimento, 4 chaves são suficientes para definir todas as identificações possíveis no documento através de chaves. Portanto, o tempo de execução médio para 4 chaves apresentado na Figura 9(b) reflete o tempo máximo necessário para processar estes documentos utilizando a proposta apresentada neste artigo.

Os experimentos apresentados nesta seção mostram que o impacto do algoritmo XKeyMatch depende muito da forma como a chave XML é definida, apresentando uma variação de 10% a 192% sobre o tempo de processamento do algoritmo XyDiff isoladamente. É importante ressaltar que em diversas aplicações a qualidade do algoritmo de diff é mais importante que o seu desempenho.

5. Conclusão

Este trabalho propõe uma nova abordagem no contexto de algoritmos de diff para XML. Ao invés de realizar somente uma análise estrutural, esta abordagem realiza uma análise semântica no documento XML, através do uso de chaves XML. A abordagem consiste em um pré-processamento das versões de um documento XML utilizando chaves, com o intuito de identificar e casar as entidades que possuam o mesmo valor para estes elementos. Estes casamentos são então passados para um algoritmo de diff, para que este possa realizar a detecção de diferenças com base nestes casamentos.

O estudo experimental realizado para determinar o impacto que a fase de pré-processamento tem sobre o processo de detecção de diferenças como um todo mostrou que ele é significativo. Embora o algoritmo XKeyMatch requeira que o usuário conheça o conteúdo dos documentos comparados, quando as chaves XML definidas como entrada expressam fielmente a sua semântica, o algoritmo sempre gera resultados mais significativos que outros baseados somente em similaridades de valor e estrutura.

Como trabalho futuro, pretende-se investigar como os dois passos do algoritmo XKeyDiff, que correspondem ao algoritmo XKeyMatch e o algoritmo de diff propriamente dito, podem ser combinados para que o algoritmo possa ser executado com um único percurso sobre as versões do documento. O pré-processamento com chaves apresentado requer apenas um percurso na árvore devido a construção de um AFD a partir do conjunto de chaves dado como entrada para o algoritmo. Como alterações na definição de chaves são bastante raras, é possível considerar o armazenamento do autômato construído para sua posterior utilização no processamento de novas versões do mesmo documento, ao invés de reconstruí-lo a cada execução.

Referências

Abiteboul, S., Cobéna, G., e Marian, A. (2002). Detecting changes in XML documents. *Proceedings of the 18th International Conference on Data Engineering (ICDE'2002), San Jose, CA, EUA*, páginas 41–52.

Al-Ekram, R., Adma, A., e Baysal, O. (2005). diffX: An Algorithm to Detect Changes in Multi-version XML Documents. *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Canada*, páginas 1–11.

Bray, T., Paoli, J., e Sperberg-McQueen, C. M. (1998). Extensible Markup Language (XML) 1.0. W3C Recommendation. Disponível em: http://www.w3.org/TR/.

Buneman, P., Davidson, S., Fan, W., Hara, C., e Tan, W. (2002). Keys for XML. *Computer Networks*, 39(5):473–487.

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

FSF (2006). Gnu diff. Disponível em: http://www.gnu.org/software/diffutils/diffutils.html.

Green, T. J., Gupta, A., Miklau, G., Onizuka, M., e Suciu, D. (2004). Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Trans. Database Syst.*, 29(4):752–788.

Hopcroft, J. E. e Ullman, J. D. (2000). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2a edição.

Santos, R. C. e Hara, C. (2007). A Semantical Change Detection Algorithm for XML. *Proceedings of Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2007), Boston, MA, EUA*, páginas 438–443.

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

Wang, Y., DeWitt, D. J., e Cai, J. (2003). X-Diff: An Effective Change Detection Algorithm for XML documents. *Proceedings of the 19th International Conference on Data Engineering (ICDE'2003), Bangalore, India*, páginas 519–530.

Xu, H., Wu, Q., Wang, H., Yang, G., e Jia, Y. (2002). KF-Diff+: Highly Efficient Change Detection Algorithm for XML Documents. *Confederated International Conferences DOA, CoopIS and ODBASE 2002, Irvine, CA, EUA*, páginas 1273–1286.

Zhang, K., Stgatman, R., e Shasha, D. (1989). Simple Fast Algorithm for the Editing Distance between Trees and Related Problems. *SIAM Journal on Computing*, 18:1245–1262.