RODRIGO CORDEIRO DOS SANTOS

XKEYMATCH: UM ALGORITMO SEMÂNTICO PARA DETECÇÃO DE DIFERENÇAS ENTRE DOCUMENTOS XML

CURITIBA

2006

RODRIGO CORDEIRO DOS SANTOS

XKEYMATCH: UM ALGORITMO SEMÂNTICO PARA DETECÇÃO DE DIFERENÇAS ENTRE DOCUMENTOS XML

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Carmem S. Hara

CURITIBA

2006

"O mais importante da vida não é a situação em que estamos, mas a direção para a qual nos movemos."

Oliver Wendell Holmes

Dedico este trabalho aos meus pais, Joselino Cordeiro dos Santos e Ivanisa Cavalli, pelos valores e ensinamentos dados, desde minha infância, nos primeiros aprendizados, até este momento especial de minha vida.

Agradeço primeiramente a *Deus* a oportunidade de realizar o Mestrado e, principalmente, a conquista deste título.

Agradeço a minha orientadora, *Carmem Hara*, a paciência e todo o aprendizado adquirido.

Agradeço a minha *família* o apoio que me foi dado do início ao fim de mais esta etapa da minha vida.

Agradeço a todos os meus caros *amigos* do Mestrado que, com alegria, em diversos momentos ajudaram a amenizar os momentos difíceis e complicados pelos quais passamos.

Agradeço a minha namorada, *Ledyvânia Franzotte*, o carinho e companheirismo demonstrados durante o Mestrado.

Por fim, agradeço, *in memorian*, aos *entes queridos* de minha família que infelizmente não puderam ver a conquista deste título, mas que, com certeza, estiveram presentes na minha memória e no meu coração em todos os momentos.

SUMÁRIO

LISTA DE FIGURAS V

LISTA DE TABELAS VI

RESUMO VII

ABSTRACT VIII

1 INTRODUÇÃO 1

1.1 Organização do trabalho 4

2 EXTENSIBLE MARKUP LANGUAGE (XML) 5

2.1 Introdução 5

2.2 Estrutura de documentos XML 6

2.3 Representação e manipulação de documentos XML 7

2.4 Expressões de caminho 10

2.5 Esquemas 11

2.5.1 Document Type Definition (DTD) 11

2.5.2 XML Schema 13

3 CHAVES PARA XML 16

3.1 Introdução 16

3.2 Expressões de caminho 17

3.3 Definição de chaves 18

3.4 Chaves absolutas e relativas 22

4 ALGORITMOS DE DETECÇÃO DE DIFERENÇAS 24

4.1 Introdução 24

4.2 Visão geral sobre algoritmos de *diff* 24

4.2.1 Problema de edição de string 25

4.2.2 Problema de edição de árvore 26

4.2.2.1 Selkow 26

4.2.2.2 Tai 27

4.2.2.3 Zhang e Shasha 28

4.2.2.4 Chawathe 28

4.2.3 Algoritmos de *diff* em XML . . . 29

4.2.3.1 XyDiff . . . 29

4.2.3.2 X-Diff . . . 31

4.2.3.3 KF-Diff . . . 32

4.2.3.4 DiffX . . . 33

4.2.4 Outros algoritmos de *diff* . . . 33

4.2.4.1 LaDiff e MH-Diff . . . 33

4.2.4.2 DeltaXML . . . 34

4.2.4.3 XMLtreediff . . . 34

4.3 Quadro comparativo . . . 34

5 ESTUDO DE CASO: ANÁLISE DE ALGORITMOS DE DIFF PARA XML 36

5.1 Introdução . . . 36

5.2 Operações de inserção/remoção . . . 37

5.3 Operações de alteração . . . 40

5.4 Operações de movimentação . . . 42

5.5 Conclusões . . . 45

6 SEMÂNTICA EM ALGORITMOS DE DIFF PARA XML 47

6.1 Introdução . . . 47

6.2 Visão geral do algoritmo XKeyMatch . . . 48

6.3 Construção das árvores XML . . . 50

6.4 Construção do AFD . . . 51

6.5 Seleção dos candidatos . . . 55

6.6 Casamentos . . . 60

7 EXPERIMENTOS 65

7.1 Introdução . . . 65

7.2 Experimento 1 . . . 66

7.3 Experimento 2 . . . 69

7.4 Experimento 3 . . . 71

8 CONCLUSÃO 75

8.1 Trabalhos futuros . . . 76

REFERÊNCIAS **79**

A SAÍDAS DOS ALGORITMOS XKEYMATCH E XYDIFF **80**

A.1 Introdução 80

A.2 Estrutura do edit script do XyDiff 80

A.3 Saídas do experimento 1 83

A.4 Saídas do experimento 2 86

A.5 Saídas do experimento 3 89

LISTA DE FIGURAS

1.1 Partes de versões de um documento XML sobre DVDs 2

2.1 Documento XML (carta) 6
2.2 Documento XML (compositores) e sua representação em árvore 8
2.3 Árvores percorridas por expressões de caminho 10
2.4 DTD de pessoas 12
2.5 XML Schema de pessoas 14
2.6 Definição de chave em XML Schema 15

3.1 Árvore XML (empresa) 18
3.2 Documento XML (empregados) 19
3.3 Partes de documentos XML com igualdade de valores 21
3.4 Documento XML (pessoas) 21

5.1 Inserção/Remoção de nodo atributo 37
5.2 Inserção/Remoção de subárvore 38
5.3 Inserção/Remoção de nodo elemento 39
5.4 Alteração de texto 40
5.5 Alteração de atributo 41
5.6 Movimentação de atributo 42
5.7 Movimentação de subárvore para outra posição 43
5.8 Movimentação de subárvore para outro nível 44

6.1 Arquitetura de um algoritmo de *diff* com o XKeyMatch 48
6.2 Algoritmo XKeyMatch 49
6.3 Algoritmo XKeyMatch - Principal 50
6.4 Versões de um documento XML sobre universidades 51
6.5 Caminhos opcionais em um AFN 52
6.6 Construção do AFD 54
6.7 Valores de chave e nodo a ser identificado 55
6.8 Diversos contextos possíveis para um nodo n_t 56

6.9 Algoritmo XKeyMatch - Seleção dos Candidatos 57

6.10 Algoritmo XKeyMatch - Casamentos 60

6.11 Situação de ambigüidade na propagação de casamentos 62

7.1 Versões de um documento XML sobre clubes de futebol 66

7.2 Casamentos via atributo ID da DTD 68

7.3 Versões de um documento XML sobre esportes 68

7.4 Versões de um documento XML sobre uma vizinhança 69

7.5 Casamentos de ascendentes 70

7.6 Versões de um documento XML sobre atribuição de tarefas 72

A.1 Estrutura do edit script do XyDiff 80

A.2 Edit Script sem XKeyMatch - clubes 83

A.3 Edit Script com XKeyMatch - clubes 84

A.4 Edit Script sem XKeyMatch - vizinhança 87

A.5 Edit Script com XKeyMatch - vizinhança 87

A.6 Edit Script sem XKeyMatch - tarefas 90

A.7 Edit Script com XKeyMatch - tarefas 90

LISTA DE TABELAS

3.1 Sintaxe utilizada para as expressões de caminho 17

4.1 Quadro comparativo dos algoritmos de *diff* apresentados 35

6.1 Resultados da etapa de seleção de candidatos: v_{t-1} 58

6.2 Resultados da etapa de seleção de candidatos: v_t 59

6.3 Resultados da etapa de casamentos: *candidatesPairs* 63

6.4 Resultados da etapa de casamentos: *matches* 64

A.1 Operações possíveis do edit script do XyDiff 81

A.2 Atributos das operações do XyDiff 82

RESUMO

Algoritmos de detecção de diferenças entre documentos XML existentes na literatura são focados em uma análise estrutural do documento. Quando XML é utilizado para troca de dados, ou quando diferentes versões de um documento são periodicamente verificadas, uma comparação baseada na semântica definida no documento pode representar resultados mais significativos. Neste trabalho, propõe-se o uso de chaves para XML no contexto destes algoritmos. Estas chaves determinam que elementos em diferentes versões representam a mesma entidade no mundo real. Um algoritmo de comparações, chamado XKeyMatch, foi elaborado, propondo um pré-processamento para encontrar elementos de acordo com uma classe de chaves para XML. Esta classe de chaves foi definida baseada em uma análise da qualidade de resultados de algoritmos de diff para XML encontrados na literatura. O objetivo deste pré-processamento é a realização de casamentos de entidades em diferentes versões de um documento XML, informando tais casamentos para um algoritmo de *diff*. Este algoritmo foi implementado e experimentos foram realizados, que possibilitaram verificar a efetividade da proposta deste trabalho.

ABSTRACT

XML diff algorithms proposed in the literature have focused on the structural analysis of the document. When XML is used for data exchange, or when different versions of a document are downloaded periodically, a matching process based on keys defined in the document can present more meaningful results. This work proposes the use of XML Keys in the context of diff algorithms. That is, XML keys determine which elements in different versions refer to the same entity in the real world, and therefore should be matched by the diff algorithm. A comparative analysis of two algorithms using this approach is conducted. Based on these results, an extension of these algorithms with a preprocessing phase for pairing elements according to a class of XML keys is proposed. This algorithm, called XKeyMatch, was implemented, and an experimental study has been conducted to show the effectiveness of the proposal.

Capítulo 1

Introdução

A *Linguagem de Marcação Extensível* (XML - *Extensible Markup Language*) [7] tornou-se um padrão para a troca de dados pela *Web*. O objetivo de seus projetistas era descrever uma linguagem de marcação flexível, escalonável e adaptável, indo diretamente de encontro às limitações que a *Linguagem de Marcação para Hipertexto* (HTML - *Hipertext Markup Language*) [27] possui frente à constante expansão da *Web*. A XML fez uma distinção entre os três principais elementos constituintes de um documento da *Web*: estrutura, conteúdo e apresentação, permitindo tratá-los separadamente. Logo, pode-se fazer, por exemplo, o tratamento dos dados do documento sem se preocupar com o formato em que estes devem ser apresentados.

A representação de dados em XML tornou-se muito popular nos últimos anos, principalmente como meio de publicação e transporte de dados via *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). Portanto, existem diversos trabalhos de algoritmos de *diff* para XML, ou seja, algoritmos que detectam as mudanças existentes entre versões de documentos XML [2, 34, 35, 4].

O seguinte exemplo ilustra esta necessidade. Suponha que uma pessoa queira adquirir, pela Internet, alguns DVDs de seu interesse. Em uma primeira visita ao *site*, a pessoa obtém uma lista de DVDs disponíveis, com informações relativas a estes. Em uma nova visita, após duas horas, a ferramenta de busca deste *site* deve coletar as informações alteradas neste período. Uma parte destas informações são mostradas na Figura 1.1.

```xml
<DVDs>
  <DVD>
    <Titulo>"Missao Impossivel"</Titulo>
    <Duracao>"110 minutos"</Duracao>
    <Preco>"R$ 30,00"</Preco>
    <Quantidade>"3"</Quantidade>
  </DVD>
  <DVD>
    <Titulo>"Tomb Raider"</Titulo>
    <Duracao>"100 minutos"</Duracao>
    <Preco>"R$ 35,00"</Preco>
    <Quantidade>"10"</Quantidade>
  </DVD>
  <DVD>
    <Titulo>"James West"</Titulo>
    <Duracao>"106 minutos"</Duracao>
    <Preco>"R$ 20,00"</Preco>
    <Quantidade>"20"</Quantidade>
  </DVD>
</DVDs>
```

(a) versão antiga

```xml
<DVDs>
  <DVD>
    <Titulo>"Tomb Raider"</Titulo>
    <Duracao>"100 minutos"</Duracao>
    <Preco>"R$ 30,00"</Preco>
    <Quantidade>"10"</Quantidade>
  </DVD>
  <DVD>
    <Titulo>"James West"</Titulo>
    <Duracao>"106 minutos"</Duracao>
    <Preco>"R$ 20,00"</Preco>
    <Quantidade>"10"</Quantidade>
  </DVD>
</DVDs>
```

(b) versão nova

Figura 1.1: Partes de versões de um documento XML sobre DVDs

Em um primeiro passo, um algoritmo de detecção de diferenças verifica se as duas versões do documento XML referente à lista de DVDs são idênticas. Se estas versões são distintas, o algoritmo faz a comparação entre as entidades presentes nas versões, tentando casar cada DVD da versão antiga com cada DVD da versão nova, para determinar quais DVDs ainda estão disponíveis, quais já foram vendidos e quais DVDs foram disponibilizados neste período. No exemplo, pode-se verificar que o DVD "Missão Impossível" não se encontra mais disponível, e que os DVDs "Tom Raider" e "James West" ainda estão disponíveis para venda. Em seguida, o algoritmo, para cada DVD ainda disponível, determina quais informações foram modificadas neste período. No exemplo, verifica-se que o preço do DVD "Tomb Raider" foi reduzido para R$ 30,00 e que a quantidade disponível do DVD "James West" diminuiu para 10 unidades.

Os algoritmos de *diff* para XML existentes identificam as alterações nos documentos baseados, principalmente, em uma análise sintática dos dados. Ou seja, estes algoritmos realizam comparações entre os documentos baseados na estrutura destes. A semântica, ou seja, o real significado dos dados, não é analisada. Desta maneira, estruturas tendenciosas do documento 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* [25], 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 à 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 à mesma entidade nas duas versões são identificados, e somente então as alterações nestes registros são detectadas.

Visando melhorar a qualidade das mudanças detectadas, propõe-se, neste trabalho, a utilização de uma abordagem semântica na detecção de mudanças entre documentos XML. O algoritmo proposto para tal fim, chamado *XKeyMatch*, realiza um pré-processamento das versões de um documento XML, identificando entidades idênticas entre estas, através do uso de chaves para XML. O objetivo deste pré-processamento é a realização de casamentos de entidades em diferentes versões de um documento XML, informando tais casamentos para um algoritmo de *diff*. A estratégia de iniciar a comparação através de chaves é natural quando duas bases de dados relacionais são comparadas. Já que o XML tornou-se um padrão para troca de dados, é também natural que utilizemos a mesma estratégia para este formato. 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.

Embora existam na literatura estudos comparativos entre algoritmos de *diff* para XML, estes são guiados principalmente por uma análise de desempenho. A qualidade dos resultados gerados não é avaliada. No estudo de caso apresentado neste trabalho, é comparada a qualidade do resultado de dois algoritmos de *diff*, XyDiff[2] e X-Diff[34]. Ou seja, foi analisado se as modificações necessárias para igualar os dois documentos comparados eram semanticamente corretas. Os resultados deste estudo permitiram a determinação de uma classe de chaves para XML, dentre aquelas propostas em [8], suficiente e necessária para solucionar os problemas identificados.

Para verificar o comportamento de algoritmos de *diff* utilizando esta abordagem, o algoritmo XKeyMatch foi implementado. Com o auxílio desta implementação, experi-

mentos foram conduzidos para avaliar a proposta introduzida.

Resultados iniciais utilizando esta abordagem e a implementação de uma ferramenta, chamada *XKeyDiff*, foram apresentados em [28]. Porém, o XKeyDiff considera uma classe de chaves para XML mais simples que aquela considerada neste trabalho, assim, não solucionando alguns dos principais problemas encontrados na detecção de diferenças somente estrutural.

1.1 Organização do trabalho

Este trabalho está organizado da seguinte forma. O Capítulo 2 descreve uma breve introdução da linguagem XML e das tecnologias relevantes ao trabalho. O Capítulo 3 descreve a definição de chaves para XML utilizada pelo XKeyMatch. Um estudo sobre algoritmos de *diff*, mostrando a evolução destes, se encontra no Capítulo 4.

Um estudo de caso, baseado em experimentos realizados em dois algoritmos de *diff* para XML, é apresentado no Capítulo 5. O Capítulo 6 descreve o algoritmo XKeyMatch, e resultados de experimentos realizados neste são descritos no Capítulo 7.

O Capítulo 8 contém a conclusão e trabalhos futuros para a continuação deste estudo.

O Apêndice A contém a execução dos experimentos realizados no Capítulo 7 pela ferramenta que implementa o algoritmo XKeyMatch. Também são apresentadas as saídas geradas pelo algoritmo de *diff* utilizado em conjunto a esta ferramenta.

Capítulo 2

Extensible Markup Language (XML)

2.1 Introdução

A *Linguagem de Marcação Extensível* (XML - *eXtensible Markup Language*) [7], um subconjunto da *Linguagem de Marcação Padrão Generalizada* (SGML - *Standard Generalized Markup Language*) [1], foi criada em 1996 e recomendada pelo *W3C* (*World Wide Web Consortium*) [32] em 1998. O objetivo de seus projetistas era descrever uma linguagem de marcação flexível, escalonável e adaptável, indo diretamente de encontro às limitações que a *Linguagem de Marcação para Hipertexto* (HTML - *Hipertext Markup Language*) [27] possui frente a constante expansão da *Web*.

A linguagem XML tornou-se um padrão para a troca de dados heterogêneos (semi-estruturados) pela *Web*, devido principalmente ao fato de fazer uma distinção entre os três principais elementos constituintes de um documento da *Web*: estrutura, conteúdo e apresentação. A *estrutura* diz respeito às regras que determinam se o documento está bem-formado. O *conteúdo* são os dados de negócio, ou seja, as informações que se quer enviar. A *apresentação* determina como o conteúdo deve ser disposto ao receptor do documento, a fim de que torne mais clara a sua legibilidade. A própria XML é responsável pelo conteúdo do documento. A parte de apresentação é tratada pela *Linguagem de Estilo Extensível* (XSL - *eXtensible Style Language*) [13]. A estrutura ou esquema do documento pode ser definida utilizando a linguagem de *Definição de Tipo de Documento* (DTD - *Document Type Definition*) ou *XML Schema* [19], dentre outros. A HTML, ao contrário, não faz tal distinção, concatenando o conteúdo com a apresentação, e não há regras que invalidem um documento deste tipo.

Existem ainda outras características que demonstram a potencialidade da linguagem:

- XML é extensível, ou seja, é uma metalinguagem que permite criar novas linguagens

de marcação inteiramente novas para descrever tipos de dados específicos como dados biológicos, matemáticos ou financeiros;

- Documentos no formato XML são altamente portáveis, podendo ser visualizados em qualquer editor ASCII/*Unicode*;
- A estrutura de documentos XML é semi-estruturada, ou seja, não há rigidez quanto a ordem e quantidade exata de elementos que o documento deve ter, a não ser que haja um esquema definido junto ao documento. Esta estrutura é excelente para bases de dados "mutantes", como as bases de dados biológicas;
- Ao contrário da HTML, XML é altamente rígida quanto à sintaxe, evitando interpretações errôneas e/ou dúbias dos dados do documento.
- As linguagens de programação podem manipular sem muitas dificuldades os dados de um documento XML. Basta utilizar um analisador sintático de XML. Este analisador verifica a sintaxe do documento e permite ao software processar e manipular os dados.

Neste capítulo serão apresentados conceitos sobre XML e tecnologias relativas a esta linguagem, relevantes ao presente trabalho. Estes conceitos serão utilizados e referenciados nos demais capítulos.

2.2 Estrutura de documentos XML

A Figura 2.1 mostra um exemplo simples de um documento XML. Todo documento XML deve possuir uma declaração ao estilo da primeira linha do exemplo dado. Esta linha indica que o documento é do formato XML, cuja versão é a 1.0 e que o documento está utilizando a codificação iso-8859-1.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<carta>
  <conteudo>Olá Mundo !</conteudo>
  <local cidade="Curitiba" estado="Parana" />
</carta>
```

Figura 2.1: Documento XML (carta)

Observe a estrutura de marcas utilizada no documento. Uma *marca* é todo item que começa com o caracter “<” e termina com o caracter “>”. Por exemplo, o item da segunda linha do exemplo (<`carta`>) é uma marca. Existem basicamente dois tipos de marcas: as *marcas de abertura*, como a marca acima citada, e as *marcas de fechamento*, como a marca da quinta linha do exemplo dado. As marcas de fechamento possuem, após o caracter de abertura (“<”), o caracter “/”. Uma marca pode ser ainda, ao mesmo tempo, de abertura e fechamento, conforme mostra a quarta linha do exemplo. Este tipo de marca possui o caracter de abertura “<” e o de fechamento “/ >”.

Observe agora os dados contidos no documento da Figura 2.1. O conteúdo de um documento XML é estruturado por *elementos*. Um elemento é um bloco delimitado por uma marca de abertura e uma marca de fechamento, inclusive elas. No exemplo, a terceira linha é um elemento, um bloco que define o conteúdo da carta. Um elemento pode possuir *atributos*, como os itens `cidade` e `estado` da quarta linha do exemplo. Os atributos contêm *valores*, como o valor “`Curitiba`” do atributo `cidade`. Estes valores sempre devem estar entre aspas. O *texto* de um elemento é o conteúdo entre as marcas de abertura e fechamento. No exemplo, “`Olá Mundo !`” é o texto do elemento `conteúdo`. Observe que `conteúdo` não é texto para `carta`, mas sim um subelemento deste. Todo documento XML deve possuir um e somente um elemento raiz. No exemplo dado, `carta` é o elemento raiz.

Perceba o aninhamento existente no documento. Na Figura 2.1, o elemento conteúdo está dentro do elemento `carta`. Isto quer dizer que o `conteúdo` é um elemento formador da carta, neste contexto. Note que as marcas envolvidas devem estar na seqüência correta, ou seja, um elemento que possui outro aninhado não pode ter uma marca de fechamento antes do elemento aninhado.

2.3 Representação e manipulação de documentos XML

Um documento XML, como foi citado na Seção 2.1, é um arquivo-texto. Porém, recuperar dados neste tipo de documento com técnicas tradicionais de acesso de arquivo seqüencial não seria algo fácil nem prático, principalmente no que diz respeito a adicionar e remover elementos dinamicamente [17].

O *Modelo de Objeto de Documento* (DOM [5] - *Document Object Model*) provê um

meio mais simples para acessar e manipular um documento XML. O DOM fornece um conjunto de objetos e métodos para representar documentos deste tipo, bem como para alterar, adicionar ou remover elementos de maneira simples. É uma interface neutra com relação a linguagens e plataformas [31]. O DOM serve como uma *interface de programação de aplicações* (API - *Application Programming Interface*) que permite que programadores suportem esta API, aumentando a interoperabilidade na *Web* [31].

Quando um analisador sintático de DOM analisa um documento XML, ele cria uma estrutura de árvore na memória que contém todos os dados relatados no documento [17]. Esta estrutura permite uma navegação simples pelos itens formadores do documento. Logo, os mesmos conceitos da teoria de árvores, como nodos, ascendentes e descendentes, profundidade da árvore, podem ser aplicados para o documento. A API contém o conjunto de métodos e objetos necessários para manipular e acessar esta estrutura.

Para o DOM, um documento XML é uma estrutura de nodos [8]. Estes nodos podem ser de vários tipos: elemento, atributo e texto. Cada nodo é um *objeto*, que possui *métodos* e *propriedades.* As propriedades descrevem características do nodo, como o nome, valores e nodos filhos. Os métodos permitem criar, excluir e acrescentar nodos, bem como acessar nodos ascendentes, descendentes e irmãos. Por exemplo, o método *nextSibling* acessa o próximo irmão de um nodo elemento. Já o método *firstChild* acessa o primeiro filho de um nodo elemento.

Observe o exemplo ilustrado na Figura 2.2. Este exemplo mostra uma representação simplificada de árvore que o DOM faria para o documento XML contido na mesma figura.

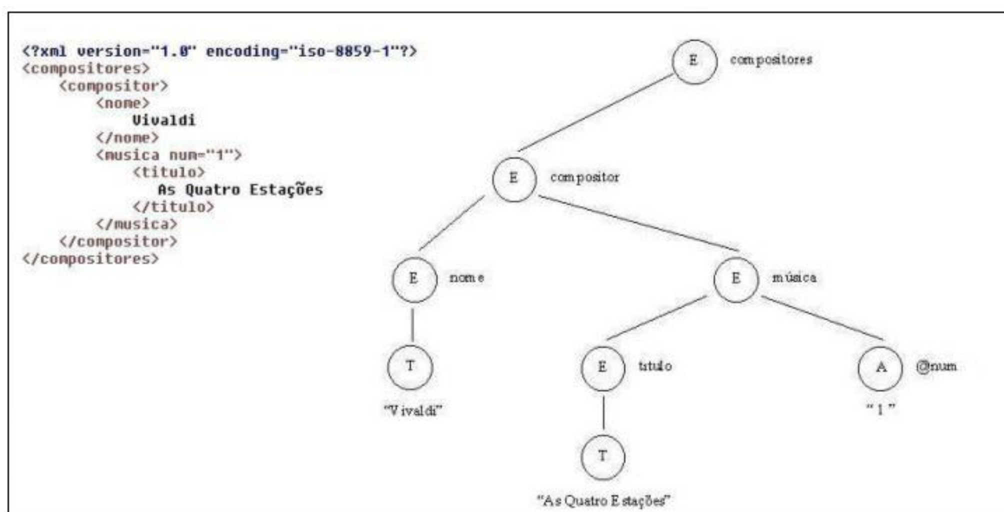


Figura 2.2: Documento XML (compositores) e sua representação em árvore

Os nodos E são nodos do tipo elemento, os nodos T são do tipo texto, e os nodos A são do tipo atributo. Nodos elemento carregam um *nome* (texto ao lado de cada nodo do tipo E). Nodos texto não possuem nome, mas apenas um *texto* (texto abaixo de cada nodo do tipo T). Já os nodos atributos possuem *nome* e *texto*. Somente nodos elementos podem conter filhos. No exemplo mostrado na Figura 2.2, como o elemento `compositor` está aninhado no elemento `compositores`, na representação em árvore, `compositor` é nodo filho de `compositores`. Os nodos atributo e texto sempre são terminais. O DOM especifica também como alcançar os filhos de um elemento. Elementos e textos são guardados em um *array*. O índice para este *array* é um número inteiro determinado pela ordem dos subelementos dentro deste elemento. Os atributos são guardados em um dicionário, pois o nome de um atributo de um determinado elemento deve ser único dentro deste.

Formalmente, esta representação de um documento XML pode ser definida como segue.

Definição 2.3.1 *Dado um conjunto infinito* `E` *de nomes de elemento, um conjunto infinito* `A` *de nomes de atributos e um símbolo* `S` *indicando textos, uma árvore XML é definida como* $T = (V, lab, ele, att, val, r)$ *onde:*

- `V` *é um conjunto de nodos em* `T`*;*
- `lab` *é uma função* `V` $\rightarrow$ `E` $\cup$ `A` $\cup$ $\{$`S`$\}$*;*
- `ele` *é uma função parcial de* `V` *para seqüências de vértices* `V` *tal que para qualquer* $v \in V$*, se* `ele(v)` *é definido então* `lab(v)` $\in$ `E`*;*
- `att` *é uma função parcial* `V` $\times$ `A` $\rightarrow$ `V` *tal que para qualquer* `v` $\in$ `V` *e* `l` $\in$ `A`*, se* `att(v,l)` $=$ `v'` *então* `lab(v)` $\in$ `E` *e* `lab(v')` $=$ `l`*;*
- `val` *é uma função parcial de* `V` *para valores de string tal que para qualquer nodo* `v` $\in$ `V`*,* `val(v)` *é uma string se e somente se ou* `lab(v)` $=$ `S` *ou* `lab(v)` $\in$ `A`*;*
- `r` *é um nodo distinto em* `V` *e é chamado de raiz de* `T`*.*

Esta definição estabelece o conceito de árvore aplicado ao documento XML. Conceitos definidos em estrutura de árvore podem ser utilizados na estrutura de árvore XML, tais como: `filhos(n)`, que representa os nodos encontrados no nível imediatamente abaixo do nodo `n`, na hierarquia da árvore XML, `descendentes(n)`, que representa todos os nodos

abaixo do nodo `n` na hierarquia de árvore XML, e `ascendentes(n)`, que representa todos os nodos acima de `n` na hierarquia da árvore XML.

2.4 Expressões de caminho

Visto que o DOM é baseado em uma estrutura de árvore, torna-se necessária uma ferramenta que possibilite percorrer esta estrutura, identificando e acessando partes ou subconjuntos de documentos XML. Uma *expressão de caminho* (*path expression*) pode ser utilizada para este propósito. Uma expressão de caminho é uma expressão envolvendo *nomes de nodos*, que descreve um conjunto de caminhos em uma árvore de documento [8]. Outra alternativa são as *expressões regulares*, utilizadas em diversas linguagens de programação, também úteis para dados semi-estruturados.

A *XML Path Language* (XPath [14] - Linguagem de Caminhos de XML), é a linguagem atualmente utilizada e recomendada pelo W3C. Ela usa uma sintaxe baseada em caminhos semelhante à sintaxe utilizada em sistemas de arquivos.

Observe as árvores ilustradas na Figura 2.3. Elas representam o mesmo documento XML.

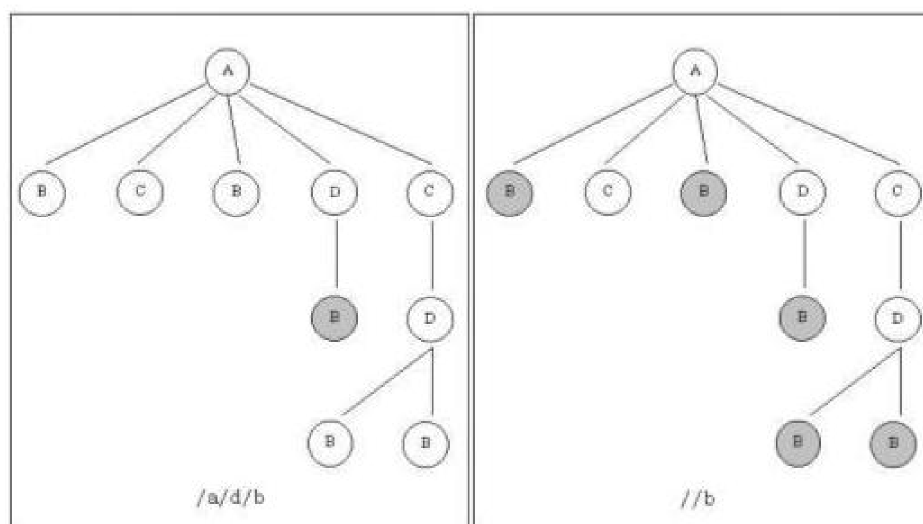


Figura 2.3: Árvores percorridas por expressões de caminho

A primeira expressão (/a/d/b) é uma expressão possível em XPath, que seleciona todos os nodos de nome *b*, cujo pai é um nodo de nome *d*, e este último deve ser um filho do nodo de nome *a* logo abaixo da raiz. XPath usa o caracter "/" tanto para especificar a raiz do documento quanto para definir uma concatenação de nodos. Neste mesmo exemplo, a

segunda expressão (//b) seleciona todos os nodos de nome *b*, não importando o nível que se encontra. O caracter "//" especifica uma procura em todos os descendentes do nodo atual.

XPath possui outras características além das expressões de caminho, como opções de filtro, definição do eixo do documento e outras características que estão fora do escopo deste trabalho.

2.5 Esquemas

Um documento XML é dito *válido* quando segue a estrutura lógica definida por uma linguagem de definição de esquema. Esta pode ser por exemplo, uma DTD (*Document Type Definition*) [7], ou um tipo de documento semelhante ao documento XML chamado de XML Schema [19], dentre outros.

A seguir, são apresentadas estas duas linguagens de definição de esquemas para XML, que estão entre as principais.

2.5.1 Document Type Definition (DTD)

A DTD (*Document Type Definition*, ou seja, Definição de Tipo do Documento) foi a primeira linguagem proposta para definir a estrutura de um documento XML. Os analisadores sintáticos de XML podem utilizá-la para validar ou não um documento.

Uma DTD é expressa utilizando uma notação de gramática na forma de EBNF (*Extended Backus-Naur Form*) e não uma sintaxe de XML [17].

Uma DTD pode ser declarada *interna* ou *externamente* [31]. Uma DTD interna é uma DTD definida no prólogo do documento XML, usada somente para o documento específico. Já uma DTD externa é um arquivo que pode ser usado apenas localmente ou pode ser público. Neste caso, o endereço URL (*Uniform Resource Locator*, ou seja, Localizador Universal de Recursos) deve ser informado no prólogo do documento XML, a fim de que possa ser compartilhado via *Web*.

Um exemplo de DTD é mostrado na Figura 2.4. Uma DTD descreve todos os atributos, elementos e outros itens formadores de um documento XML, bem como a ordem, a localização e quantidade que cada item deve ter.

Neste exemplo, estão sendo especificadas regras para elementos (marcas que começam

com !ELEMENT) e para atributos (marcas iniciadas por !ATTLIST). Após indicar o tipo do item, define-se o nome do elemento/atributo em questão. No caso de atributo, primeiro deve-se especificar o nome do elemento ao qual este pertence. A partir disto define-se então as regras que o elemento/atributo deve respeitar.

```
1  <!ELEMENT pessoas (pessoa)+>
2  <!ELEMENT pessoa(nome,email*,url*,link?)>
3  <!ATTLIST pessoa id ID #REQUIRED>
4  <!ELEMENT familia (#PCDATA)>
5  <!ELEMENT nome (#PCDATA|familia)*>
6  <!ELEMENT email (#PCDATA)>
7  <!ELEMENT url EMPTY>
8  <!ATTLIST url href CDATA #REQUIRED>
9  <!ELEMENT link EMPTY>
10 <!ATTLIST link
11    gerente IDREF #IMPLIED
12    subordinados IDREFS #IMPLIED>
```

Figura 2.4: DTD de pessoas

Na primeira linha, a DTD define o elemento `pessoas`, e que dentro deste deve haver 1 ou mais elementos `pessoa`. A indicação do número de ocorrências que um determinado item deve ter é indicado pelos caracteres "+" (um ou mais), "*" (zero ou mais) e "?"(um ou nenhum).

A segunda linha define que um elemento `pessoa` pode ter apenas os itens indicados dentro do parênteses. Os parênteses definem agrupamentos de itens possíveis neste elemento e a sua ordem, podendo também definir vários níveis de profundidade. O separador "|" implica uma condição de alternância (ou). A quinta linha do exemplo descreve que um elemento `nome` deve ter uma dentre as duas alternativas indicadas. Como existe o caracter "*" logo após os parênteses, isto indica que pode haver vários elementos `família` alternados com vários itens do tipo *pcdata* (tipo texto), por exemplo.

Existem outras opções que podem ser definidas para elementos e atributos. Pode-se definir, no caso de atributos, que um determinado item deva estar presente (#REQUIRED) ou que possa ser opcional (#IMPLIED) e, no caso de elementos, pode-se definir que o conteúdo do elemento seja vazio (EMPTY).

A característica mais relevante ao presente trabalho em relação às DTDs é a possibilidade de definir *atributos identificadores*. Neste exemplo, foram utilizados os tipos ID (linha 3), IDREF (linha 11) e IDREFS (linha 12). O tipo ID significa que o atributo tem um valor que identifica unicamente, no documento inteiro, o elemento que possui tal

atributo. É semelhante aos atributos de chave primária em uma base de dados relacional. Um atributo do tipo IDREF possui um valor de referência para uma outra instância do valor da identificação. Se os atributos ID são chaves primárias em um documento XML, o tipo IDREF é como uma chave estrangeira, inclusive preservando as questões da integridade referencial. O tipo IDREFS equivale a um conjunto de IDs ou chaves estrangeiras. A comparação com o modelo relacional é apenas ilustrativa, pois a estrutura de chaves em uma base de dados relacional é bem mais complexa que em uma DTD.

2.5.2 XML Schema

Outra alternativa, proposta pelo W3C, no que diz respeito a linguagens de definição de esquema, é formalmente chamada de *XML Schema*. Os conceitos básicos subjacentes a XML Schema são similares àqueles das DTDs, cuja função principal continua sendo a validação [31].

O XML Schema, ao contrário da DTD, segue as regras de sintaxe da XML e, portanto, pode ser avaliado por um analisador XML.

Um documento XML Schema define uma hierarquia e tipos para os dados, ao mesmo tempo que reforça a estrutura do documento e a integridade dos dados [31]. Os documentos XML Schema aperfeiçoam as DTDs por permitirem mais precisão na expressão de alguns conceitos no vocabulário e possuírem uma estrutura mais rica na descrição da informação [18]. Em um XML Schema, pode-se criar novos tipos de dados, os quais são divididos em duas categorias: tipos de dados simples e tipos de dados complexos. Um elemento é definido como sendo do tipo simples se possuir apenas texto, como por exemplo, o tipo inteiro, o tipo *string*, o tipo data, entre outros. Este tipo restringe o texto que pode aparecer no valor de um atributo ou no conteúdo de um elemento textual. Um elemento é do tipo complexo se permitir conter outros elementos ou atributos. Este tipo restringe o conteúdo de um elemento relativamente aos atributos e elementos filho que pode ter.

Um exemplo de documento definido com a linguagem XML Schema é mostrado na Figura 2.5. Este exemplo é a transformação da DTD definida na Figura 2.4 em XML Schema.

Neste exemplo, do mesmo modo que na DTD, são declarados os elementos e atributos que são permitidos em um documento XML. Nota-se, porém, que a hierarquia destes elementos e atributos não seguem mais a gramática EBNF, mas sim a própria hierarquia

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3      <xs:element name="email" type="xs:string"/>
4      <xs:element name="familia" type="xs:string"/>
5      <xs:element name="link">
6          <xs:complexType>
7              <xs:attribute name="subordinados" type="xs:IDREFS" use="optional"/>
8              <xs:attribute name="gerente" type="xs:IDREF" use="optional"/>
9          </xs:complexType>
10     </xs:element>
11     <xs:element name="nome">
12         <xs:complexType mixed="true">
13             <xs:choice>
14                 <xs:element ref="familia" minOccurs="0" maxOccurs="unbounded"/>
15             </xs:choice>
16         </xs:complexType>
17     </xs:element>
18     <xs:element name="url">
19         <xs:complexType>
20             <xs:attribute name="href" type="xs:string" use="required"/>
21         </xs:complexType>
22     </xs:element>
23     <xs:element name="pessoa">
24         <xs:complexType>
25             <xs:sequence>
26                 <xs:element ref="nome"/>
27                 <xs:element ref="email" minOccurs="0" maxOccurs="unbounded"/>
28                 <xs:element ref="url" minOccurs="0" maxOccurs="unbounded"/>
29                 <xs:element ref="link" minOccurs="0"/>
30             </xs:sequence>
31             <xs:attribute name="id" type="xs:ID" use="required"/>
32         </xs:complexType>
33     </xs:element>
34     <xs:element name="pessoas">
35         <xs:complexType>
36             <xs:sequence>
37                 <xs:element ref="pessoa" maxOccurs="unbounded"/>
38             </xs:sequence>
39         </xs:complexType>
40     </xs:element>
41 </xs:schema>
```

Figura 2.5: XML Schema de pessoas

definida pela estrutura que o formato XML oferece. Logo, elementos e atributos declarados dentro de um outro elemento devem, no documento XML, obedecer a esta hierarquia. Por exemplo, nas linhas 34 a 40, a declaração do elemento `pessoas` indica, através da declaração de tipo complexo (*complexType*), que o elemento `pessoa` deve estar logo abaixo, na hierarquia do documento, do elemento `pessoas`. Para a definição da hierarquia entre elementos, deve-se utilizar o tipo complexo, ou referenciando (através do uso do atributo `ref`) nomes do elementos declarados previamente ou declarando um elemento em outro.

Assim como no caso da DTD, que possui os símbolos “+” e “*” para definir o número de ocorrências de um elemento, pode-se, no XML Schema, para tal fim utilizar, na declaração do elemento, os atributos `minOccurs` e `maxOccurs`, indicando o mínimo e máximo

de ocorrência deste. O valor pode variar de 0 a sem limite (*unbounded*). Por exemplo, as linhas 14 e 29 possuem declarações de elementos com o número de ocorrência destes.

Atributos devem ser declarados dentro de um tipo complexo (como exemplo, na linha 20), indicando seu nome (atributo `name`), o tipo de dados (atributo `type`), e se o uso é opcional ou requerido (atributo `use`).

Nas linhas 11 a 17 é declarado o elemento `nome`, definido como tipo complexo. Nesta declaração, tem-se o atributo `mixed=true`, indicando que o elemento será misto, possuindo texto ou subelementos. Ainda nestas linhas, é utilizada a marca `choice`, que equivale ao operador OU (“|”) da DTD.

Para a definição de chaves em XML Schema, existem dois modos possíveis. O primeiro modo emula os tipos de atributos ID, IDREF e IDREFS provenientes da DTD. Como exemplo, na Figura 2.5, as linhas 31, 8 e 7 declaram atributos com os tipos ID, IDREF e IDREFS, respectivamente. O segundo modo, semelhante à definição de chaves que será explanada no Capítulo 3, utiliza expressões de caminho em XPath. Como exemplo, a Figura 2.6 mostra uma definição de chave neste modo. Esta definição indica que, dentro de um elemento `biblioteca`, um elemento `livro` é identificado unicamente pelo valor do elemento `isbn`. Os atributos `xpath` desta declaração permitem a definição de uma expressão de caminho em XPath, para selecionar os elementos a serem identificados (`selector`), e os elementos/atributos que os identificam (`field`). Também é possível a declaração de unicidade (*unique*) e referência (*keyref* e *keyrefs*) nesta definição de chaves.

```
.
.
.
<xs:element name="biblioteca">
        <xs:complexType>
                .../...
        </xs:complexType>
        <xs:key name="livro">
                <xs:selector xpath="livro" />
                <xs:field xpath="isbn" />
        </xs:key>
</xs:element>
.
.
.
```

Figura 2.6: Definição de chave em XML Schema

Capítulo 3

Chaves para XML

3.1 Introdução

O uso intenso da linguagem XML na troca de dados demonstra que a preocupação não está apenas na estrutura dos dados, mas também há uma preocupação sobre a semântica dos dados transmitidos [12].

Neste contexto, uma evolução sobre como acessar dados semi-estruturados está se tornando necessária, assim como ocorreu com os modelos de dados estruturados. Por exemplo, nos modelos relacionais as chaves são valoradas, ou seja, podem conter valores do "mundo real", valores textuais que possuem um significado associado à informação acessada. Logo, é natural que a XML incorpore um método baseado em valores para localizar dados em documentos do seu formato [9].

Chaves são uma parte essencial para o projeto de um banco de dados [8]. Com elas, pode-se identificar univocamente um subconjunto dos dados. Além disso, as chaves podem ser úteis para verificar se os dados estão corretos (ou seja, os dados respeitam as restrições de integridade definidas), permitem referenciar dados externos àquele conjunto de dados (chaves estrangeiras, ou *foreign keys*), como também podem garantir que uma alteração (*update*) afete somente os dados que foram indicados para alteração.

A importância de chaves para XML está sendo reconhecida e algumas especificações de chaves para XML estão sendo apresentadas [8]. A DTD e o XML Schema, conforme visto na Seção 2.5, já introduzem o conceito de chaves.

Na proposta de chaves da DTD, os atributos do tipo ID possuem uma série de inconvenientes. Primeiro, estes tipos de atributos são mais propriamente ponteiros do que chaves. Segundo, pode-se especificar no máximo um atributo deste tipo dentro de um elemento. Por último, estes atributos representam uma chave global, ou seja, a chave precisa ser válida para todo o documento e não para um subconjunto dele.

A proposta de chaves do XML Schema permite que restrições de integridade e referencial sejam definidas utilizando expressões de caminho em XPath. Apesar desta definição ser mais abrangente que a definição de chaves da DTD, observam-se, ainda, algumas limitações [8]. Primeiramente, a complexidade inerente da própria linguagem XPath dificulta a identificação de equivalências entre expressões, por sua vez dificultando sua implementação. Outro aspecto a ser ressaltado é a igualdade de valores, ou seja, o valor que identifica unicamente uma entidade. O XML Schema restringe esta igualdade somente para textos. Mas existem diversos casos de chaves que precisam de valores com estruturas mais complexas, como subárvores.

Para superar estas limitações, os autores de [8] propuseram uma nova estrutura de chaves para XML, definidas em termos de uma ou mais expressões de caminho (Seção 2.4). Logo, podem envolver mais de um atributo, subelementos ou estruturas mais genéricas. Estas chaves podem ser definidas para um subconjunto dos dados (*chaves relativas*) ou para todo o documento (*chaves absolutas*). Esta especificação de chave é independente de DTD ou qualquer outra definição de esquema. Esta nova proposta é discutida neste capítulo e será usada nos capítulos posteriores.

3.2 Expressões de caminho

Como descrito na Seção 2.4, uma expressão de caminho permite a navegação pela estrutura de um documento XML. A escolha de uma linguagem para este propósito deve levar em conta o poder de expressão desejado das chaves. Nesta proposta de chaves foi escolhido um subconjunto das expressões de caminho definidos pelo XPath e das expressões regulares [8]. A sintaxe é mostrada na Tabela 3.1.

Tabela 3.1: Sintaxe utilizada para as expressões de caminho

Caracteres	**Significado**
ϵ	caminho vazio
$nome$	nome de um elemento ou atributo
$//$	caminho em profundidade
$/$	concatenação de expressões de caminho

Para definir o significado de uma chave, foi usada a notação n[[P]] que representa o conjunto de nodos alcançáveis (selecionados) começando do nodo `n` e seguindo a expressão de caminho `P`. [[P]] é o mesmo que raiz[[P]]. Exemplos desta notação para a árvore XML da Figura 3.1:

- $[[empregado]] = \{13\}$
- $13[[contato]] = \{17\}$

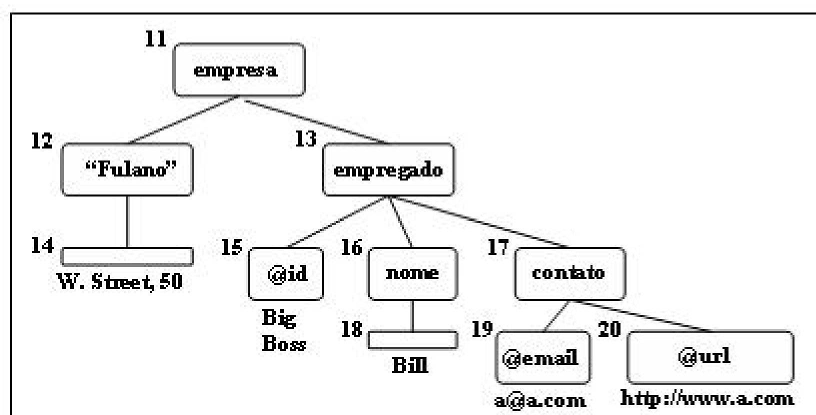


Figura 3.1: Árvore XML (empresa)

3.3 Definição de chaves

Para definir uma chave em XML, é necessário que se especifique um conjunto no qual a chave é definida e os itens que juntos identificam unicamente elementos neste conjunto [8]. 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 pode ser definida da seguinte maneira:

$$(\mathtt{Q}, \{P_1, \ldots, P_n\})$$

Onde `Q` é uma expressão de caminho e $\{P_1, \ \ldots\ldots, \ P_n\}$ é um conjunto de expressões de caminho. A expressão `Q` define um conjunto de nodos alvos, no qual a chave é definida. As expressões P_i definem os itens que juntos identificam unicamente elementos neste

conjunto. A expressão `Q` é chamada de caminho alvo (*target path*) e as expressões P_i são chamadas de caminhos chave (*key paths*).

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`.

Considere o documento mostrado na Figura 3.2. Uma chave possível seria:

`(/empregado/nome, {primeiro-nome, sobrenome})`

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!--   @version:     -->
<empregados>
  <empregado>
    <nome>
      <primeiro-nome>Fulano</primeiro-nome>
      <sobrenome>Tal</sobrenome>
    </nome>
  </empregado>
  <empregado>
    <nome>
      <primeiro-nome>Beltrano</primeiro-nome>
      <sobrenome>Tal</sobrenome>
    </nome>
  </empregado>
</empregados>
```

Figura 3.2: Documento XML (empregados)

Onde `empregado/nome` é o caminho alvo e os caminhos chave são `primeiro-nome` e `sobrenome`. O caminho alvo da chave identifica todos os nodos `nome` cujo pai é um nodo `empregado`, diretamente abaixo do nodo raiz do documento. O conjunto destes nodos é o conjunto alvo. O primeiro caminho chave identifica o conjunto de todos os nodos `primeiro-nome` cujo pai é um nodo `nome`. O segundo caminho chave identifica o conjunto de todos os nodos `sobrenome` cujo pai é um nodo `nome`. A chave é válida

para o documento se, para quaisquer nodos n_1 e n_2 do conjunto alvo, não existe nenhum nodo em n_1[[primeiro caminho chave]] com o mesmo valor de algum nodo de n_2[[primeiro caminho chave]] ou se não existe nenhum nodo em n_1[[segundo caminho chave]] com o mesmo valor de algum nodo de n_2[[segundo caminho chave]]. Note que, no documento da Figura 3.2, para o conjunto de nodos definido pelo segundo caminho chave há dois nodos com o mesmo valor (nodo `sobrenome` com valor "Tal") em dois nodos `empregado`. Mas como não há nodos com os mesmos valores no conjunto definido pelo primeiro caminho chave, a chave é válida para o documento.

Definição 3.3.1 *Uma árvore XML T satisfaz uma chave se e somente se para quaisquer nodos n_1 e n_2 do conjunto alvo alcançáveis via caminho alvo, toda vez que há uma intersecção não vazia de valores para cada caminho chave começando a partir de n_1 e de n_2, então n_1 e n_2 devem ser o mesmo nodo. Ou seja, um nodo n satisfaz uma chave específica* $(\mathtt{Q},\{P_1,\ldots\ldots,P_k\})$ *se e somente se para quaisquer n_1, n_2 em $[[Q]]$, se para todo i, $1 \leq i \leq k$, existe $z_1 \in n_1[[P_i]]$ e $z_2 \in n_2[[P_i]]$, tais que $z_1 =_v z_2$, então $n_1 = n_2$.*

Esta definição envolve dois tipos de igualdade. A primeira ($z_1 =_v z_2$) representa igualdade de valores dos nodos z_1 e z_2, enquanto a segunda ($n_1 = n_2$) representa identidade de nodos, ou seja, n_1 e n_2 são o mesmo nodo.

Definição 3.3.2 *Dois nodos, n_1 e n_2, de uma árvore XML T, possuem igualdade de valores ($n_1 =_v n_2$) se e somente se as seguintes condições são satisfeitas:*

- $\mathrm{lab}(n_1) = \mathrm{lab}(n_2)$;
- Se n_1 e n_2 são nodos do tipo atributo ou *string*, então $val(n_1) = val(n_2)$
- Para qualquer nodo (do tipo elemento ou atributo) $d1 \in filhos(n_1)$, existe $d_2 \in filhos(n_2)$ tal que $d_1 =_v d_2$, e vice-versa.

A última condição merece um pouco mais de atenção. Se os nodos a serem comparados são do tipo elemento, eles serão considerados iguais se os seus atributos forem iguais, não importando a ordem dos atributos dentro de cada nodo, e se estes nodos possuírem mais nodos logo abaixo destes, cada um destes nodos devem ser iguais também (considere todos estes "iguais" como igualdade de valores), não importando a ordem em que se encontram. Como exemplo, considere a Figura 3.3.

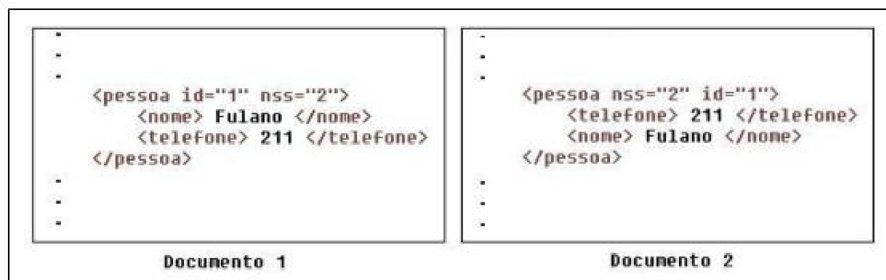


Figura 3.3: Partes de documentos XML com igualdade de valores

Mesmo que a ordem dos subelementos e atributos sejam diferentes, os elementos `pessoa` do documento 1 e do documento 2 possuem igualdade de valores. Esta última observação, referente a não importância da ordem dos filhos, é uma mudança da definição original. Na definição de [8], a ordem dos atributos não é importante, mas a ordem dos filhos elemento ou texto são relevantes.

Considere agora um outro exemplo de análise de chaves. Considere a seguinte chave: `(/pessoa,{nome, telefone})`. Este chave é satisfeita pelo documento mostrado na Figura 3.4, pois a primeira e a segunda pessoa possuem o mesmo nome, mas não possuem números de telefones em comum.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!--   @version:      -->
<pessoas>
  <pessoa>
    <nome>Fulano</nome>
    <telefone>211</telefone>
    <telefone>212</telefone>
  </pessoa>
  <pessoa>
    <nome>Fulano</nome>
    <telefone>213</telefone>
  </pessoa>
</pessoas>
```

Figura 3.4: Documento XML (pessoas)

Embora exista uma intersecção não vazia no primeiro caminho chave (`nome`), existe uma intersecção 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 intersecção não-vazia em ambos os caminhos chave em relação à primeira pessoa.

3.4 Chaves absolutas e relativas

As chaves descritas até a seção anterior são chamadas de *chaves absolutas*. Isto porque são chaves que têm como contexto todo o documento XML. Um contexto indica o "universo" no qual a chave deve ser válida. Este contexto não precisa ser necessariamente "global". Há também a possibilidade de especificar um escopo, ou seja, uma determinada área de interesse do documento, um subconjunto dos dados totais no qual a chave possui relevância. Chaves com esta características são chamadas de *chaves relativas*. Uma chave absoluta é uma chave relativa que possui como contexto o documento inteiro [8].

O formato de dados de cunho científico motivou o uso de chaves relativas. Considere a base de dados de seqüências de proteínas, chamada Swiss-prot [3]. Esta base possui um número de acesso para cada entrada (uma chave). Em cada uma destas entradas, existe ainda uma seqüência de citações, cada qual identificada por um número. Logo, para identificar uma citação, são necessários estes dois números [8]. Note que, para uma citação, o número de identificação deve ser único somente dentro de uma entrada. O contexto, neste caso, são as entradas (do primeiro nível). Perceba também que há uma hierarquia de chaves. As chaves relativas são para XML o que as entidades fracas são para as bases de dados relacionais. A chave de uma entidade fraca consiste da chave da entidade proprietária, mais alguma informação da entidade da qual ela depende [8]. A notação de uma chave relativa é a seguinte:

$$\texttt{(Q,(Q',}\{P_1,\ldots,P_n\}\texttt{))}$$

Onde `Q` é uma expressão de caminho que define o contexto no qual a chave deve ser válida (caminho contexto). `Q'` é o caminho alvo e P_1 a P_n são os caminhos chave.

Definição 3.4.1 *Um documento satisfaz uma chave relativa* `(Q, (Q', S))` *se e somente se para todos os nodos* n *pertencentes ao conjunto de nodos definidos pelo caminho contexto,* n *satisfaz a chave* `(Q', S)` *[8]. Ou seja,* `(Q, (Q', S))` *é uma chave relativa se* `(Q', S)` *é uma chave para todo o subdocumento com raiz em algum nodo pertencente ao conjunto de nodos definidos pelo caminho contexto [8].*

Alguns exemplos de chaves relativas para XML são dados a seguir.

- `(/universidade, ( //empregado, {@empregadoID}))`: em uma `universidade`, um `empregado` pode ser identificado pelo seu atributo `empregadoID`.

- `(/biblia, (livro, {nome}))`: dentro de uma `biblia`, um `livro` é identificado pelo `nome`.

Para descrever uma chave absoluta em termos de chave relativa, basta usar o nome do nodo raiz ou o símbolo "ϵ" como caminho contexto. Por exemplo, `(/pessoa, {nome, telefone})` é a forma reduzida de `(`ϵ`,(pessoa, {nome, telefone}))`

Observe a diferença entre chaves absolutas e chaves relativas. O primeiro exemplo, sobre universidade e empregado, não é o mesmo que `(/universidade//empregado, {@empregadoID})`. Observe que na chave relativa é permitido que empregados de diferentes universidades possam ter o mesmo valor para o atributo `empregadoID`. A chave absoluta não permite isto, por causa da sua definição. Esta última chave pode ser lida da seguinte maneira: "dentro do contexto de todo o documento, um empregado de uma universidade pode ser identificado pelo seu atributo empregadoID". A diferença é sutil. A primeira chave indica que é necessário que primeiro se identifique a universidade para depois identificar o empregado dentro desta universidade. Aqui há uma hierarquia de chaves. Na última chave, a idéia é que, em todo o documento, um nodo `empregado`, que é descendente de um nodo `universidade`, possui valor do atributo `empregadoID` diferente de todos os outros nodos `empregado` descendentes de um nodo `universidade`. Os nodos `universidade` em questão podem ser quaisquer, não necessitando que as universidades sejam as mesmas. A primeira chave somente teria o mesmo sentido que a última se o documento XML em questão tivesse exatamente 1 universidade.

No Capítulo 5, é apresentado um estudo de caso com dois algoritmos de *diff*. Após a análise dos resultados experimentais há a constatação de que, com a utilização do conceito de chaves para XML, é possível melhorar a qualidade dos resultados gerados por tais algoritmos. Esta extensão dos algoritmos de *diff* considerando chaves para XML é o assunto tratado no Capítulo 6.

Capítulo 4

Algoritmos de Detecção de Diferenças

4.1 Introdução

A representação de dados na forma semi-estruturada, dentre elas o XML, tornou-se muito popular nos últimos anos, principalmente como meio de publicação e transporte de dados via *Web* [12]. Os usuários não se interessam somente pelos valores atuais dos dados, mas também pelas mudanças. O usuário pode querer monitorar alterações (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).

Uma vez que a informação *on-line* altera-se constantemente, é necessário o uso de alguma ferramenta que detecte tais alterações. Logo, surgiu a necessidade de estudos sobre métodos eficientes de controle de mudanças de documentos no formato XML. Houve, então, um renovação no interesse por computar mudanças de dados.

Existem diversos trabalhos de *algoritmos de diff*, ou seja, algoritmos de detecção de diferenças entre documentos, para vários formatos de dados. Geralmente, estes algoritmos são utilizados para detectar diferenças entre versões de um mesmo documento.

Neste capítulo, serão apresentados conceitos e modelos de algoritmos de *diff*, visando mostrar toda a problemática envolvendo este tipo de algoritmo. Também serão abordadas a evolução dos algoritmos, bem como a forma que os conceitos básicos destes foram utilizados na construção de algoritmos de *diff* para XML.

4.2 Visão geral sobre algoritmos de *diff*

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 mu-

danças entre as duas versões, ou seja, dada a versão `t(i-1)` e o *edit script*, é possível chegar à versão `t(i)` [2]. Um *edit script*, também chamado de *delta*, 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. O número de operações de um *edit script*, ou seja, a quantidade de operações necessárias para uma versão `t(i-1)` ser transformada na versão `t(i)` é chamado de *edit distance* (distância) entre as versões, que representa também o *custo* desta transformação.

Para detectar diferenças entre versões de um determinado documento, é necessário, primeiramente, identificar quais entidades ou elementos são idênticos entre as versões, ou seja, aqueles que não sofreram modificação entre uma versão e outra. Este primeiro passo de um algoritmo de *diff* é denominado *casamento*. A partir do casamento destas entidades, outros passos do algoritmo podem verificar os elementos distintos e quais operações são necessárias para igualar uma versão à outra. O algoritmo XKeyMatch, proposto por este trabalho, realiza esta tarefa de casamentos de entidades idênticas.

Muitos trabalhos foram realizados sobre a problemática geral da detecção de diferenças entre documentos. Muitos destes trabalhos estão focados em computar diferenças entre arquivos do tipo texto, ou seja, realizando a comparação entre *strings*. Um exemplo de ferramenta que utiliza tal algoritmo é o GNU *diff* [21]. Chawathe et al. [11] afirmam que as técnicas empregadas por este tipo de algoritmo não podem ser utilizadas de maneira generalizada, para qualquer tipo de estrutura de dados, porque esta abordagem não leva em consideração a hierarquia e o contexto da informação. Um exemplo de tal estrutura são os dados em formato de árvore.

A seguir, alguns trabalhos são apresentados, no intuito de descrever os diversos tipos de abordagem existentes no contexto de algoritmos de *diff*.

4.2.1 Problema de edição de string

Também conhecido como *string-to-string correction problem*, este problema consiste em determinar a distância entre dois *strings*. Para isto, deve-se produzir uma lista de operações de edição (*edit script*) para transformar um *string* no outro, sendo desejável descobrir o custo mínimo para tal operação.

As operações são basicamente: alterar, inserir e remover um caracter.

Existem diversos trabalhos sobre este problema, sendo o algoritmo de Wagner e Fischer [33] um dos principais trabalhos sobre esta questão.

Dado um par de *strings*, `S` e `T`, com tamanho `m` e `n` respectivamente, a solução consiste em montar uma matriz `D` de tamanho `(n+1)` $\times$ `(m+1)`, sendo `D[i,j]` a distância entre o substring de `S` composto dos primeiros i-caracteres e o substring de `T` composto dos primeiros `j`-caracteres. Logo, a entrada `D[m,n]` é a distância minima entre os dois *strings*. As `m` entradas da linha 0 contém os custos de inserção de todos os caracteres em `T` e as `n` entradas da coluna 0 contém os custos de remover todos os caracteres de `S`.

O algoritmo para construir esta solução preenche primeiramente toda a linha 0 e a coluna 0. Após isso, cada célula da matriz é calculada verificando o custo mínimo para a operação entre os caracteres (custo mínimo entre inserir, remover e alterar). O custo de alterar um caracter pelo mesmo é 0.

Para coletar a lista de operações realizadas, outro algoritmo é utilizado, coletando as operações com menor custo. Se `D[m,n]` é igual a `D[m-1,j]`, mais o custo de remover o caracter, a operação realizada foi a remoção deste.

4.2.2 Problema de edição de árvore

Para determinar a distância entre duas árvores, solucionando o problema chamado *tree-to-tree correction problem*, os algoritmos de edição de árvore estenderam a solução de edição de *string* para os conceitos de árvore. Nesta seção, serão descritos alguns destes algoritmos.

4.2.2.1 Selkow

No algoritmo de Selkow [29], existem três operações básicas: inserção, remoção e alteração do texto de um nodo.

Estas operações possuem a seguinte restrição: só podem ser aplicadas nos nodos folhas da árvore. Ou seja, nodos internos não sofrem alterações. Ao inserir um nodo, este não pode conter filhos. Ao remover um nodo, deve-se remover toda a sua subárvore. O algoritmo recebe como entrada duas árvores, `A` e `B`, e calcula, recursivamente, o custo de alterar as raizes de `A` e `B`, através do cálculo dos custos de alterar as subárvores $A_1, A_2, ..., A_n$ de `A` para as subárvores $B_1, B_2, ..., B_n$ de `B`.

Em cada etapa do algoritmo, ou seja, em cada comparação dos filhos de um nodo `a`

da árvore `A` com os filhos de um nodo `b` da árvore `B`, o algoritmo constroi uma matriz de dimensão `n` $\times$ `m`, onde `n` é o número de filhos de `a` e `m` é o número de filhos de `b`. Cada célula `D[i,j]` desta matriz representa o custo mínimo de alterar as subárvores $A_1, A_2, ..., A_i$ para as subárvores $B_1, B_2, ..., B_j$.

Os custos são calculados da seguinte maneira:

- inserção: custo de editar $A_1, A_2, ..., A_i$ para $B_1, B_2, ..., B_{j-1}$, mais o custo de inserir B_j.
- remoção: custo de editar $A_1, A_2, ..., A_{i-1}$ para $B_1, B_2, ..., B_j$, mais o custo de remover A_i.
- alteração: custo de editar $A_1, A_2, ..., A_{i-1}$ para $B_1, B_2, ..., B_{j-1}$, mais o custo de recursivamente examinar o custo de editar A_i para B_j.

4.2.2.2 Tai

A abordagem de Tai [30] utiliza um algoritmo dinâmico, sem recursão, para a determinação do custo. As operações são as mesmas definidas por Selkow.

Inserir um nodo significa anexar alguns ou todos os filhos do pai deste nodo como filhos deste. Remover um nodo significa anexar todos os filhos do nodo removido como filhos do pai deste nodo.

Para adaptar o algoritmo de *strings* para árvores, este utiliza uma travessia em pré-ordem nas árvores, numerando cada nodo. O algoritmo utiliza a idéia de mapeamento entre nodos, ou seja, `t(x:y)` representa os nodos `x` e `y`, juntamente com as arestas do caminho entre eles.

O principal problema desta abordagem é que, ao contrário do caso do algoritmo de *string*, é necessário preservar a estrutura sobre a qual foram feitos os mapeamentos. No caso de *strings*, existe somente uma estrutura possível: da esquerda para direita. Já para o caso de árvores, um mapeamento só pode ser válido se os ancestrais de ambas as árvores foram mapeados durante o algoritmo.

Também é necessário armazenar as operações realizadas para obter o custo mínimo da transformação. Para preservar a estrutura, existem passos antecedentes ao algoritmo, para evitar mapeamentos ilegais.

4.2.2.3 Zhang e Shasha

O algoritmo de Zhang e Shasha [36] possui as mesmas características do algoritmo de Tai, porém realizando a numeração das árvores em uma travessia em pós-ordem. Esta abordagem precisa manter múltiplas soluções, no intuito de realizar um *backtrack* quando operações ilegais são realizadas.

Dois conceitos são importantes nesta abordagem: *raízes-chave* e *distância de floresta*. As raízes-chave são compostas pela raiz da árvore, mais todos os nodos que possuem irmãos à esquerda. Já a distância de floresta é a distância entre a subárvore de um nodo e a subárvore de outro nodo, mais a distância entre seus irmãos à esquerda e os irmãos à esquerda do outro nodo.

Para calcular o mapeamento de custo-mínimo de um nodo, o algoritmo precisa calcular o mapeamento de custo-mínimo de todos os filhos que são raízes-chaves, mais o custo do filho mais a esquerda. Isto na realidade representa a distância de floresta do seu filho mais a direita.

4.2.2.4 Chawathe

MMDiff e XMDiff são dois algoritmos apresentados por Chawathe [10], para detecção de diferenças entre árvores de documentos. O algoritmo XMDiff, baseado no algoritmo MMDiff (este em memória interna, o outro em memória externa), é baseado no algoritmo de Selkow, para árvores ordenadas, ou seja, a ordem dos filhos é relevante na detecção de alterações. Intuitivamente, o algoritmo constrói uma matriz na mesma concepção da matriz do problema de edição de *string*.

Uma técnica utilizada é a remoção de algumas "arestas" (representadas pela matriz) para forçar a remoção (ou inserção) não só do nodo, mas também de toda a sua subárvore. Mais precisamente, arestas diagonais existem se e somente se corresponderem a nodos que possuem a mesma profundidade na árvore, e arestas horizontais (respectivamente verticais) de `(x,y)` para `(x+1,y)` existem somente se a profundidade de `x+1` é menor que a profundidade do nodo `y+1`.

4.2.3 Algoritmos de *diff* em XML

Conforme mencionado na Seção 2.3, um documento XML pode ser representado com uma estrututa de árvore. A utilização de uma algoritmo de *string* para detecção de alterações é possível, porém, a estrutura de árvore que um documento XML fornece, com sua hierarquia e contexto, provê melhores soluções para detectar tais alterações, tanto em eficiência quanto semanticamente.

Nesta seção, serão descritos diversos trabalhos de algoritmos de *diff* para XML, todos utilizando a estrutura de árvore fornecida pelo documento XML, baseando-se em premissas utilizadas nos algoritmos de *diff* para árvores.

4.2.3.1 XyDiff

O algoritmo XyDiff [2] 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. Esta abordagem, para detectar alterações entre documentos XML, utiliza uma técnica chamada BULD (*Bottom-up lazy down*) que consiste numa varredura da árvore em *top-down*, realizando a análise dos casamentos em *bottom-up*, somente descendo na árvore se os nodos avaliados forem candidatos a casamento. Outro conceito importante utilizado pelo algoritmo é o de *árvores ordenadas*, ou seja, a ordem dos filhos é relevante na detecção de alterações. O algoritmo foi implementado na linguagem C++, utilizando o DOM para a manipulação de documentos XML.

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 ela existir. A saída produzida é um *edit script* no formato XML (também chamado de *delta*), que descreve as mudanças entre as versões. O algoritmo consiste nas seguintes etapas:

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

2. **Uso do atributo ID**: Em uma travessia em ambas as árvores, detectam-se os nodos que possuam atributos ID definidos pela DTD dos documentos. A existência de um atributo ID para um dado nodo provê uma condição única para casar tal nodo: este

casamento deve ter o mesmo valor de ID. Se tal par de nodos são encontrados em ambos os documentos, eles são casados.

3. **Computação de assinaturas e ordenação por peso**: Em uma travessia em ambas as árvores, computam-se as *assinaturas* e os *pesos* de cada nodo. A assinatura consiste no valor de uma função *hash* calculado utilizando o conteúdo do nodo e a assinatura dos filhos deste. Este valor representa o conteúdo da subárvore enraizada por um nodo. O peso representa o tamanho do conteúdo para nodos do tipo texto ou a soma dos pesos dos filhos de nodos do tipo elemento. Uma fila é então construída, para o documento novo, contendo os nodos priorizados pelos seus pesos. Esta fila é utilizada na etapa seguinte.

4. **Casamentos de nodos**: Utilizando a fila construída na etapa anterior, retira-se o nodo com maior peso e constrói-se uma lista de candidatos, lista esta contituída de todos os nodos do documento antigo que possuem a mesma assinatura do nodo em avaliação do documento novo. Dos nodos desta lista, escolhe-se o melhor candidato para casar-se com o nodo do documento novo. O melhor candidato é aquele cujo pai já casou com o pai do nodo a ser casado. Se não houver, verificam-se os níveis superiores. Uma vez casados, propaga-se o casamento dos nodos ascendentes e descendentes. Os níveis a serem verificados ou casados dependem do peso do nodo. Isto reduz o número de comparações que, de outra forma, seria excessivo.

5. **Construção do delta**: Realizados os casamentos, uma análise é feita com base nestes, montando o documento (*edit script*) que relata as mudanças entre as versões do documento XML. As mudanças são identificadas da seguinte maneira:

   - **Inserções / Exclusões / Alterações**: Encontram-se todos os nodos não casados na versão antiga do documento , marcando-os como removidos. Encontram-se todos os nodos não casados na nova versão do documento, marcando-os como inseridos. Se um nodo do tipo texto foi casado mas seu conteúdo foi modificado, marca-se tal nodo como alterado.

   - **Movimentos**: Encontram-se todos os nodos que foram casados, mas seus pais não. Isto corresponde a um movimento do tipo *strong move*. Ou seja, o nodo casado não possui o mesmo pai nas duas versões do documento. Outra situação

de movimentação refere-se ao caso em que um nodo possua o mesmo pai em ambas as versões do documento, porém a posição deste perante os seus irmãos está modificada. Isto corresponde a um movimento do tipo *weak move*.

As operações são reorganizadas e o documento *delta*, no formato XML, é construído.

4.2.3.2 X-Diff

X-Diff [34] é um algoritmo público para deteccção de diferenças entre documentos XML, elaborado na Universidade de Wisconsin. Este trabalho aplica o conceito de *árvores desordenadas*, que somente avalia o relacionamento pai-filho, não levando em conta a ordenação entre irmãos. A detecção das diferenças torna-se mais complexa, porém, segundo seus autores, é mais exata e adequada para aplicações de banco de dados em XML. Outra característica deste algoritmo é a ausência de operações de movimento. Implementações do algoritmo estão disponíveis em duas linguagens: C++ e Java. O algoritmo segue as seguintes etapas:

1. **Transformação das entradas**: O algoritmo transforma cada documento XML em uma árvore, fazendo o *parse* dos documentos para estruturas chamadas *Xtrees*. Esta estrutura é um subconjunto da interface para manipulação de documento existente no DOM.

2. **Computação de assinaturas**: Em ambas as árvores, durante o processo de *parsing*, o algoritmo X-Diff utiliza, para calcular uma assinatura, uma função especial de *hash*, *XHash*, que representa a subárvore enraizada no nodo. A particularidade é que a ordem entre irmãos é desconsiderada.

3. **Casamentos de nodos**: Esta etapa consiste no casamento dos nodos entre as versões do documento XML. No intuito de reduzir o espaço de busca, em um primeiro momento, o algoritmo avalia os valores *hash* dos elementos do segundo nível das árvores. Assim subárvores inteiras são casadas, evitando a comparação de seus nodos. O próximo passo realizado é a comparação, para cada nodo folha, das *distâncias* destes com os nodos da outra árvore XML. Os resultados desta comparação são armazenados em uma tabela para, posteriormente, avaliar os melhores casamentos. Após esta comparação, realiza-se o mesmo procedimento com os nodos

pais. Em cada ascensão de nível na árvore, realiza-se a comparação dos valores *hash* para a diminuição do espaço de busca. Após realizar estes procedimentos, a avaliação dos resultados armazenados na tabela começa com os nodos raízes, partindo para seus descendentes. Portanto o X-Diff realiza uma avaliação *top-down* nas estruturas de árvore, analisando os melhores candidatos coletados e casando-os com os respectivos nodos.

4. **Passo de otimização (opcional)**: Existe um passo opcional de otimização, que consiste em utilizar um limiar (*threshold*) para escolher o melhor candidato, evitando a comparação com todos os candidatos possíveis. Ou seja, aquele candidato que possuir um limiar menor é escolhido como candidato, desconsiderando a comparação com os demais candidatos.

5. **Contrução do delta**: Assim como o algoritmo XyDiff, após a etapa de casamentos, o *edit script* é construído. Este documento descreve as operações de inclusão, remoção e alteração. Como mencionado anteriormente, não são identificadas operações de movimento.

4.2.3.3 KF-Diff

KF-Diff [35] é um algoritmo de diff para XML que trata tanto árvores ordenadas quanto árvores desordenadas. O algoritmo transforma o *tree-to-tree correction* em comparações de árvores sem caminhos duplicados, ou seja, caminhos cuja sequência de nomes dos nodos não são duplicados em relação a outro caminho. Esta árvore sem duplicações de caminhos é chamada de *árvore de chave*. Para isto, o algoritmo utiliza-se de restrições de chave definidas em [20]. O conceito de chaves utilizado aqui na verdade é coletar nodos de um caminho que possuem o mesmo nome em relação a nodos de um outro caminho.

O algoritmo constrói uma lista que contém todos os nodos irmãos cujos nomes são idênticos. Para diferenciá-los, utiliza-se de nomes únicos, chamados de *key fields*, contidos nos nodos folhas descendentes destes nodos. Após esta análise, o algoritmo então não possui caminhos idênticos, facilitando a busca de diferenças entre as árvores dos documentos.

4.2.3.4 DiffX

A proposta do algoritmo DiffX [4] é realizar o mapeamento de fragmentos de árvore isolados, para identificar os casamentos mais amplos de fragmentos entre as versões. Este processo começa da raiz da árvore da versão antiga e é repetido até que todos os nodos da árvore tenham sido verificados para casamentos. O objetivo é assegurar um número máximo de casamentos na presença de alterações de estrutura em ambos os documentos.

De forma mais detalhada, dadas as árvores dos documentos `T1` e `T2`, o algoritmo começa identificando o mapeamento mais amplo entre fragmentos de `T1` e `T2`. Este mapeamentos de fragmentos são porções de dados (subároves) comuns entre os dois documentos. O algoritmo, começando pela raiz de `T1`, encontra todos os nodos em `T2` que podem casar com o nodo corrente de `T1`, realizando esta ação recursivamente para os filhos, até que não existam mais nodos a serem verificados ou casados. O nodo de `T2` que seja raiz do fragmento mais amplo casado com o nodo em `T1` é armazenado, juntamente com seus filhos. Se houver mais de um candidato, o escolhido é aquele com o id mais baixo (maior nível na árvore) e que casou ancestrais. O processo é repetido para todos os nodos não casados em `T1`.

4.2.4 Outros algoritmos de *diff*

Abaixo, estão descritos alguns algoritmos e ferramentas de *diff* para outros propósitos, ou que são ferramentas comerciais, sem publicação do algoritmo.

4.2.4.1 LaDiff e MH-Diff

Detecção de alterações também foram estudados para documentos Latex e com estruturas aninhadas, casos dos algoritmos LaDiff e MH-Diff [11].

O algoritmo LaDiff recebe como entrada duas versões de um documento em Latex e produz outro documento, também em Latex, com as alterações. O problema de detecção é dividido em dois problemas: o primeiro é encontrar o casamentos dos objetos entre as duas versões, já o segundo é computar o *edit script* mínimo.

O algoritmo MH-Diff detecta diferenças entre documentos aninhados transformando o problema de detecção de diferenças no problema de computar o custo mínimo de cobertura de arestas em um grafo bipartido. O processo consiste em construir um grafo induzido das

árvores dos documentos de entrada, podar o grafo induzido, encontrar o custo mínimo de cobertura das arestas do grafo induzido podado e finalmente usar a cobertura de arestas e conteúdos dos nós para verificar se são relacionados. Este algoritmo identifica alterações em documentos com estrutura de árvore desordenada. O algoritmo suporta operações de inserção, remoção, movimentação, alteração, cópia e cola de nodos.

4.2.4.2 DeltaXML

A ferramenta comercial DeltaXML [24] é capaz de comparar, unir e sincronizar documentos XML. Trata tanto árvores ordenadas como desordenadas. Ela pode receber arquivos XML de entrada com até 50 Megabytes de tamanho. Não exige obrigatoriamente documentos DTD (ou XML Schema), mas este pode ser usado para coletar informações de chaves (atributo ID).

O algoritmo utiliza uma técnica similar às apresentadas no problema de edição de *string*; mais precisamente utiliza o algoritmo D-Band [26], suportando tanto comparações de versões de documentos quantos comparações de versões com um documentos atual. Suporta operações de inserção, remoção e alteração.

Em experimentos relatados em [15], este algoritmo foi considerado o melhor custo-benefício, devido ao tempo de execução e os resultados serem próximos do custo-mínimo.

4.2.4.3 XMLtreediff

A IBM desenvolveu uma ferramenta de *diff* para XML, chamada XMLtreediff [16]. Esta ferramenta é baseada no algoritmo de Zhang e Shasha [36]. Utilizando o DOM e assinaturas hash, o algoritmo da ferramenta faz a poda de subárvores idênticas. Somente considera operações de inserção, remoção e alteração. Além dessas características, o XMLTreediff realiza comparações entre árvores ordenadas e desordenadas.

4.3 Quadro comparativo

A Tabela 4.1 mostra um quadro comparativo dos algoritmos apresentados neste capítulo. Neste, as complexidades dos algoritmos são mostradas, onde cada letra possui o seguinte significado. Para o algoritmo de *string*, tem-se: `i` e `j` - número de caracteres dos *strings*; para os algoritmos de árvores, tem-se: `n` - valor máximo entre o número de nodos de uma

árvore e o número de nodos da outra árvore, `d` - profundidade das árvores (valor máximo), `e` - distância entre as árvores (valor máximo). As operações ditas como básicas são as operações de inserção, remoção e alteração. Os valores contidos na coluna *Ordenação* indicam se o algoritmo utiliza o conceito de árvores ordenadas, desordenadas ou ambos. A coluna *Seção* possui o número da seção onde se encontra a descrição do algoritmo. Os campos da tabela com o símbolo "?" indicam que a informação não foi encontrada, devido à falta de documentação sobre estes dados. O símbolo "-", presente em determinados campos, indica que a informação da coluna não se aplica ao algoritmo.

Tabela 4.1: Quadro comparativo dos algoritmos de *diff* apresentados

Algoritmo	**Complexidade**	**Memória**	**Operações**	**Ordenação**	**Seção**
Wagner e Fischer	O(i+j)	?	básicas	-	4.2.1
Selkow	O(n^2d)	?	básicas	-	4.2.2.1
Tai	O(n^2d^4)	O(n^2d^2)	básicas	-	4.2.2.2
Zhang e Shasha	O(n^2d^2)	O(n^2)	básicas	-	4.2.2.3
XMDiff	O(n^2)	linear	básicas	ordenado	4.2.2.4
MMDiff	O(n^2)	quadrático	básicas	ordenado	4.2.2.4
XyDiff	O(n log n)	linear	básicas, movimentação	ordenado	4.2.3.1
X-Diff	O(n^2)	quadrático	básicas	desordenado	4.2.3.2
KF-Diff	O(n log n)	?	básicas	ambos	4.2.3.3
DiffX	O(n^2)	linear	básicas, movimentação	ordenado	4.2.3.4
LaDiff	O($ne + e^2$)	linear	básicas, movimentação	ordenado	4.2.4.1
MH-Diff	O(n^2 log n)	linear	básicas, movimentação	desordenado	4.2.4.1
DeltaXML	?	?	básicas	ambos	4.2.4.2
XMLtreediff	O(n^2)	quadrático	básicas	ordenado	4.2.4.3

Conforme pode ser visto na tabela acima, os algoritmos, em sua maioria, possuem a ordem de complexidade quadrática. Nos algoritmos de *diff* para árvores, o algoritmo de Selkow possui o melhor desempenho, principalmente pelo fato de apenas permitir operações em nodos folhas. Já no caso de algoritmos de *diff* para XML, o XyDiff mostra-se o melhor nos quesitos desempenho e uso de memória, devido à aplicação de heurísticas que determinam a realização ou não de comparações dos nodos de árvores XML.

Capítulo 5

Estudo de Caso: Análise de Algoritmos de Diff para XML

5.1 Introdução

No Capítulo 4 foram apresentados diversos algoritmos de *diff* para XML propostos na literatura. Muitos estudos que realizam a comparação destes algoritmos são baseados no desempenho ou no tamanho dos resultados (número de operações) gerados por estes. Um destes estudos é apresentado em [15]. Embora esta análise comparativa seja baseada nestes mesmos critérios (desempenho e tamanho), os autores daquele trabalho relatam que ainda são necessários mais esforços para avaliação da qualidade semântica dos resultados, uma vez que a semântica não é facilmente mensurável.

Neste capítulo, será proposta a utilização de chaves para XML como parâmetro para mensurar a qualidade dos resultados dos algoritmos de *diff* para XML. As chaves para XML determinam quais elementos em diferentes versões de um documento referem-se à mesma entidade no mundo real. Portanto, "bons" algoritmos de *diff* devem obter sucesso em casar tais elementos.

Utilizando este parâmetro, será analisado, em um estudo de caso, o comportamento dos dois algoritmos de diff para XML mais citados em trabalhos científicos: XyDiff (descrito na Seção 4.2.3.1) e X-Diff (descrito na Seção 4.2.3.2). O objetivo deste estudo é determinar os tipos de modificações no documento cuja análise estrutural falha em casar elementos semanticamente idênticos. Os resultados deste estudo permitiram guiar a definição de uma classe de chaves para XML necessária para resolver a maioria dos problemas de casamentos detectados neste estudo.

O estudo consiste na execução dos algoritmos de *diff* em documentos XML, analisando os resultados gerados na aplicação das seguintes operações básicas: inserção, remoção, alteração e movimentação. O *edit script* resultante é então analisado, verificando se estas

operações foram corretamente detectadas. Quando isto não ocorrer, são investigadas as causas destes casamentos incorretos, assim como são sugeridas que chaves para XML são úteis para corrigir esta detecção.

5.2 Operações de inserção/remoção

1. **Inserção/Remoção de um nodo atributo**

   A Figura 5.1 apresenta duas versões de um documento, onde a nova versão é obtida através da inserção de um novo atributo a um nodo elemento existente. Por sua vez, a versão antiga é obtida através da remoção de um nodo atributo.

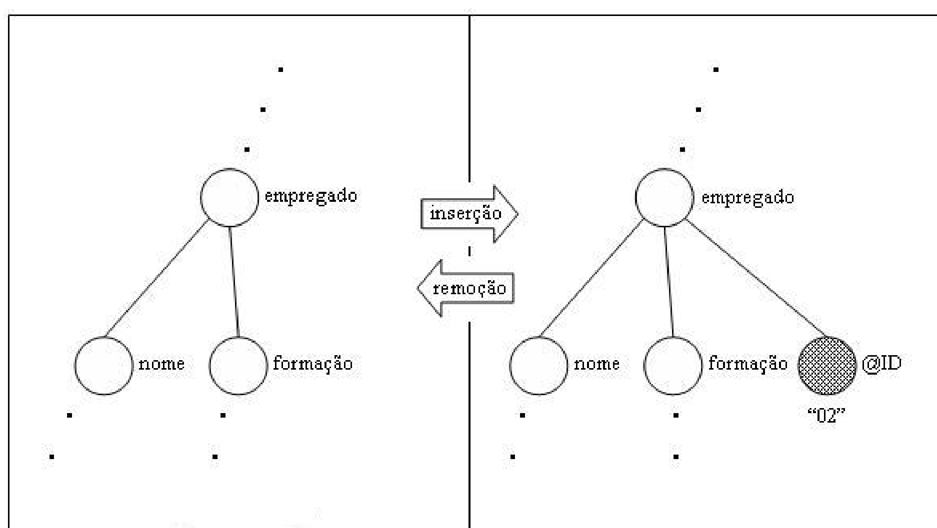


Figura 5.1: Inserção/Remoção de nodo atributo

Resultados dos algoritmos:

- **X-Diff**: Reconheceu corretamente as operações de inserção/remoção de nodos atributos.
- **XyDiff**: Existiram casos em que o atributo inserido foi casado incorretamente com um atributo de mesmo nome de elemento e valor já existente. Da mesma maneira, ocorreram casamentos indevidos na remoção de atributos que, ao invés de detectar a operação de remoção, casou o atributo removido com outro de mesmo nome de elemento e valor já existente.

Análise: O algoritmo XyDiff não avalia corretamente o contexto no qual cada nodo se encontra. Isto possibilita casamentos entre nodos atributos com mesmo valor, embora de contextos distintos.

Solução utilizando chaves para XML: Uma chave com atributo é ideal caso o documento utilize qualquer definição de chaves. O uso de uma DTD ou qualquer definição de esquema também permitiria a avaliação dos nodos atributos. Assim, pode-se identificar o nodo que possui o atributo, evitando computações custosas nas estruturas, bem como evitando casamentos semanticamente indevidos.

2. **Inserção/Remoção de uma subárvore não interna**

A Figura 5.2 apresenta duas versões de um documento, onde a nova versão é obtida através da inserção de uma subárvore em um nodo folha. A versão antiga é obtida através da remoção de uma subárvore.

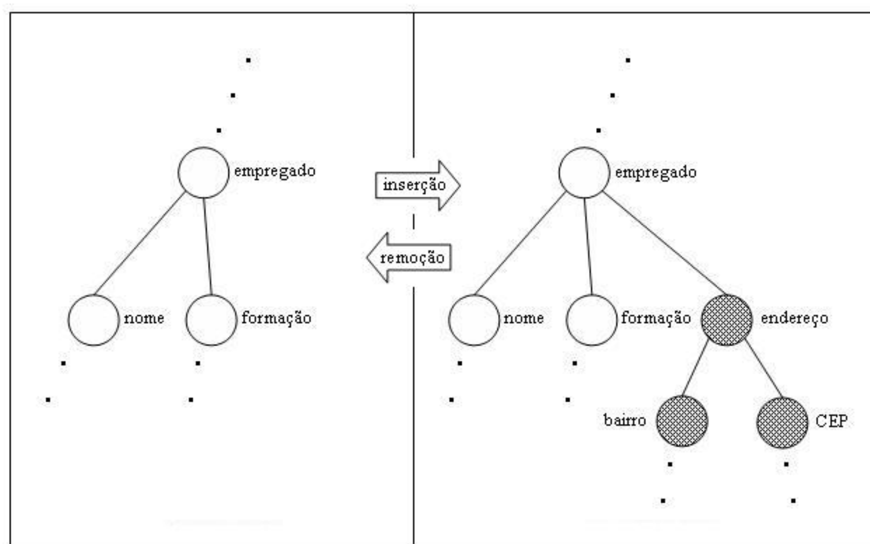


Figura 5.2: Inserção/Remoção de subárvore

Resultados dos algoritmos:

- **X-Diff**: Identificou, na maioria dos casos, a subárvore como inserida/removida, mesmo na presença de subárvores idênticas.
- **XyDiff**: Na presença de subárvores semelhantes à inserida, houve diversos casos de casamentos indevidos com tais subárvores.

Análise: A estratégia "gulosa" do algoritmo XyDiff para a realização de casamentos das subárvores maiores pode ocasionar o casamento indevido da subárvore inserida/ removida com outra subárvore idêntica já existente no documento, pois o contexto no qual as subárvores se encontram não é avaliado. Já o X-Diff possui uma estratégia de casamentos menos agressiva, embora possa realizar casamentos semanticamente incorretos devido à não utilização da informação sobre o contexto das subárvores.

Solução utilizando chaves para XML: Uma identificação, como o uso de uma chave relativa, determinando o contexto no qual se encontra a subárvore, pode resolver estas situações.

3. **Inserção/Remoção de um nodo elemento**

A Figura 5.3 apresenta duas versões de um documento, onde a nova versão é obtida através da inserção de um novo nodo elemento. Por sua vez, a versão antiga é obtida através da remoção de um nodo elemento.

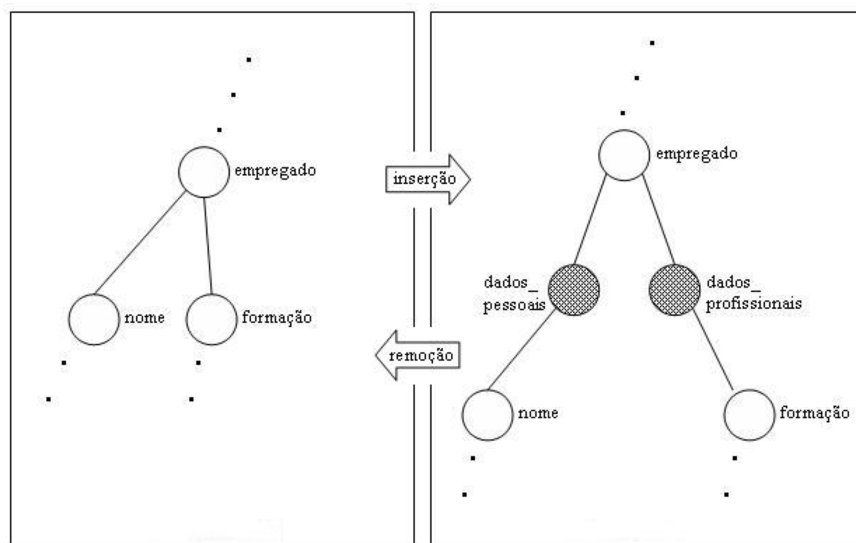


Figura 5.3: Inserção/Remoção de nodo elemento

Resultados dos algoritmos:

- **X-Diff**: O algoritmo considerou que todo o ramo com raiz no nodo adicionado foi inserido. Ou seja, detectou tal modificação como se toda a subárvore abaixo do novo nível hierárquico tivesse sido removida da versão antiga e uma nova subárvore fosse inserida na nova versão. Da mesma maneira, ao remover um

nodo elemento, o algoritmo não consegue identificar os descendentes deste, considerando estes como removidos da versão antiga e inseridos na nova versão.

- **XyDiff**: Idem ao X-Diff.

Análise: Percebe-se que a mudança de estrutura afeta a detecção de mudanças por parte destes algoritmos.

Solução utilizando chaves para XML: A utilização de uma chave com expressões de caminho com "//" poderia auxiliar em uma detecção independente de estrutura.

5.3 Operações de alteração

1. **Alterar o conteúdo de um nodo texto**

A Figura 5.4 apresenta duas versões de um documento, onde a nova versão é obtida através da alteração do conteúdo de um nodo texto.

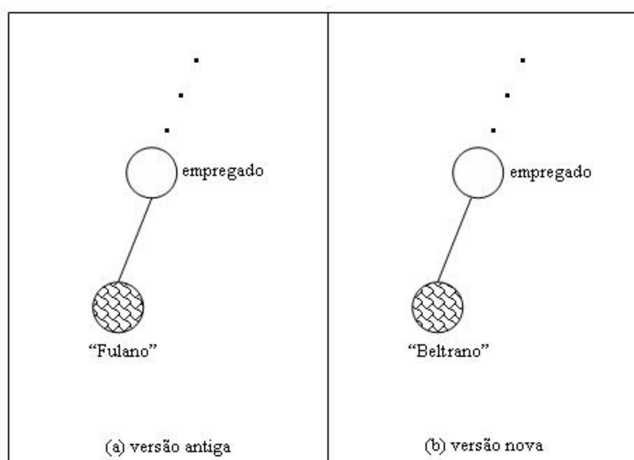


Figura 5.4: Alteração de texto

Resultados dos algoritmos:

- **X-Diff**: O algoritmo casou corretamente os nodos texto alterados.
- **XyDiff**: Similarmente à situação ocorrida nas operações de inserção e remoção, na presença de nodos semelhantes, a modificação do valor de um nodo folha, em diversos casos, resultou no casamento de nodo modificado com um nodo texto existente em outro ponto do documento com o mesmo valor.

Análise: Novamente, na presença de nodos e subárvores semelhantes, o algoritmo XyDiff, dada sua característica de casamento das subárvores maiores, pode realizar casamentos errôneos. O algoritmo X-Diff mostra um melhor comportamento neste tipo de operação, mas como não há análise da semântica do documento, este comportamento em alguns casos pode ser incorreto. Por exemplo, uma movimentação de um nodo texto geralmente é interpretado pelo X-Diff como alteração deste nodo. Este caso será demonstrado nos experimentos apresentados no Capítulo 7.

Solução utilizando chaves para XML: Da mesma maneira que nas operações de inserção e remoção, o uso de uma definição de chaves pode identificar o contexto aonde o nodo texto se encontra, evitando o casamento deste com outro nodo não pertencente a este contexto.

2. **Alterar o valor de um nodo atributo**

A Figura 5.5 apresenta duas versões de um documento, onde a nova versão é obtida através do conteúdo (valor) de um nodo atributo.

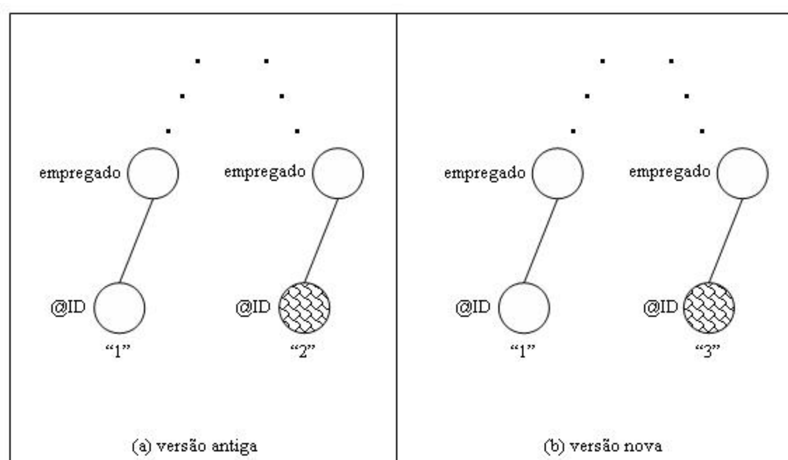


Figura 5.5: Alteração de atributo

Resultados dos algoritmos:

- **X-Diff**: O algoritmo casou os nodos atributos corretamente, mesmo com a mudança do valor. Isto semanticamente pode ser incorreto se o atributo for do tipo ID. Como o algoritmo não leva em consideração DTDs ou qualquer outro tipo de esquema, esta identificação não é possível.

- **XyDiff**: Da mesma maneira que o X-Diff, o algoritmo casou os nodos atributos corretamente. Como o algoritmo XyDiff permite a utilização de uma DTD para identificação de nodos atributos do tipo ID, é possível utilizar este recurso para o casamento correto destes nodos.

Análise: Na ausência de uma DTD ou qualquer definição de esquema, ou a ausência de uma definição de chaves com nodos atributos, o casamento correto destes depende do correto casamento dos elementos nos quais eles estão definidos. Ou seja, o casamento dos elementos-pais influenciam o casamentos dos seus atributos.

Solução utilizando chaves para XML: O comportamento descrito na análise mostra a importância do algoritmo de *diff* ter a informação sobre atributos identificadores ou utilizar uma definição de chaves com nodos atributos, pois os valores destes devem influenciar o casamento de seus elementos, e não o contrário.

5.4 Operações de movimentação

1. **Mover nodo atributo para outro nodo elemento**

   A Figura 5.6 apresenta duas versões de um documento, onde a nova versão é obtida através da movimentação de um nodo atributo.

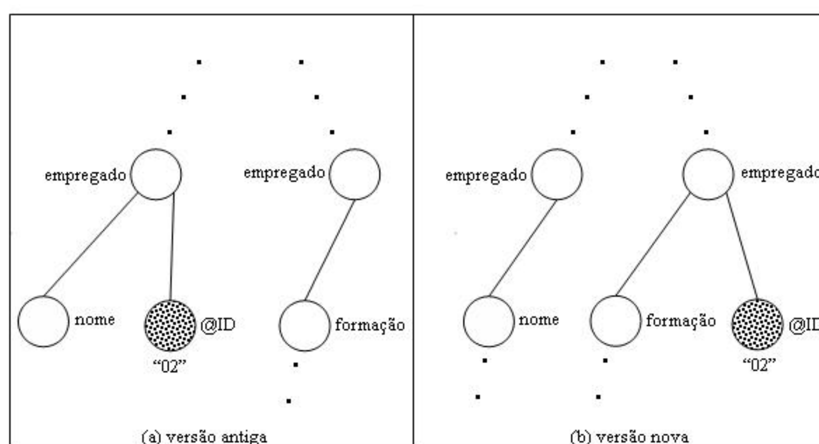


Figura 5.6: Movimentação de atributo

Resultados dos algoritmos:

- **X-Diff**: O algoritmo inverteu o casamento das subárvores devido a esta movimentação, ou seja, a movimentação do atributo pertencente ao elemento e_1

para o elemento e_2 causou o casamento do nodo e_2 da versão antiga com o nodo e_1 da versão nova. Se o atributo for do tipo ID, isto semanticamente é correto. Caso contrário, sem uma definição de chaves ou esquema, esta estratégia não é correta.

- **XyDiff**: O XyDiff considerou apenas que houve um remoção do atributo em um nodo e a inserção do atributo em outro nodo. Não detectou a modificação como uma operação de movimento.

Análise: O algoritmo X-Diff não possui um método para verificação dos tipos dos atributos. Esta avaliação é possível somente no XyDiff, com o uso da DTD.

Solução utilizando chaves para XML: A aplicação de uma definição de chaves, como o uso de uma chave com atributo, ou a definição de um esquema permitiria avaliar se o atributo é identificador, e assim aplicar a estratégia mais adequada.

2. **Mover uma subárvore para outra posição**

A Figura 5.7 apresenta duas versões de um documento, onde a nova versão é obtida através da movimentação de uma subárvore para outra posição da árvore do documento XML.

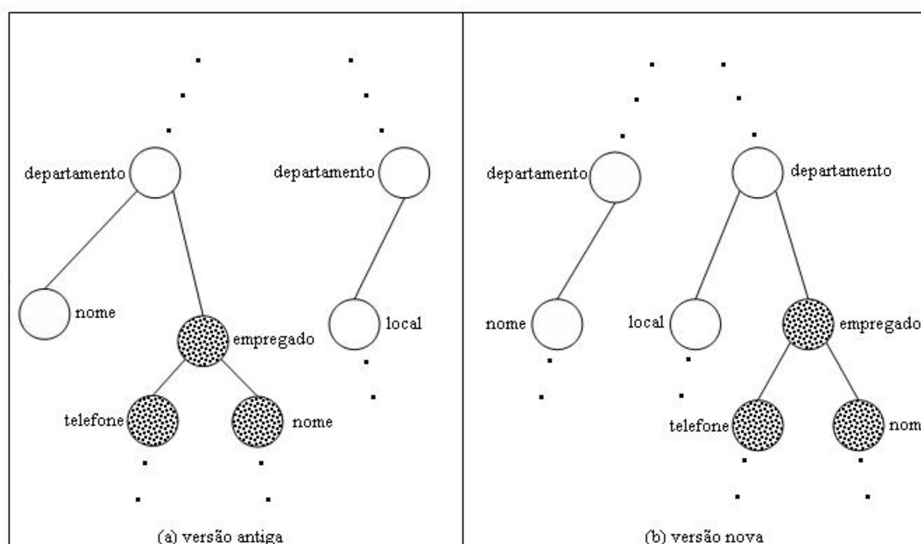


Figura 5.7: Movimentação de subárvore para outra posição

Resultados dos algoritmos:

- **X-Diff**: O X-Diff não possui operações de movimentação. Logo foi detectada uma remoção na versão antiga do documento, seguida por uma inserção da subárvore na versão nova do documento.
- **XyDiff**: O XyDiff, possuindo avaliação de operações de movimentação, detectou corretamente a alteração quando a subárvore movida possuía a mesma estrutura até a raiz; ou seja, se a subárvore foi movida do elemento e_1 para o elemento e_2 e o caminho da raiz da árvore do documento até e_1 era idêntico ao caminho da raiz até o nodo e_2.

Análise: O único problema que poderia ocorrer, para a operação de movimentação de uma subárvore, como a ilustrada na Figura 5.7, era o casamento com outra subárvore idêntica devido as escolhas de melhores candidatos realizadas pelos algoritmos.

3. **Mover uma subárvore para outro nível**

A Figura 5.8 apresenta duas versões de um documento, onde a nova versão é obtida através da movimentação de uma subárvore para outro nível na árvore do documento XML.

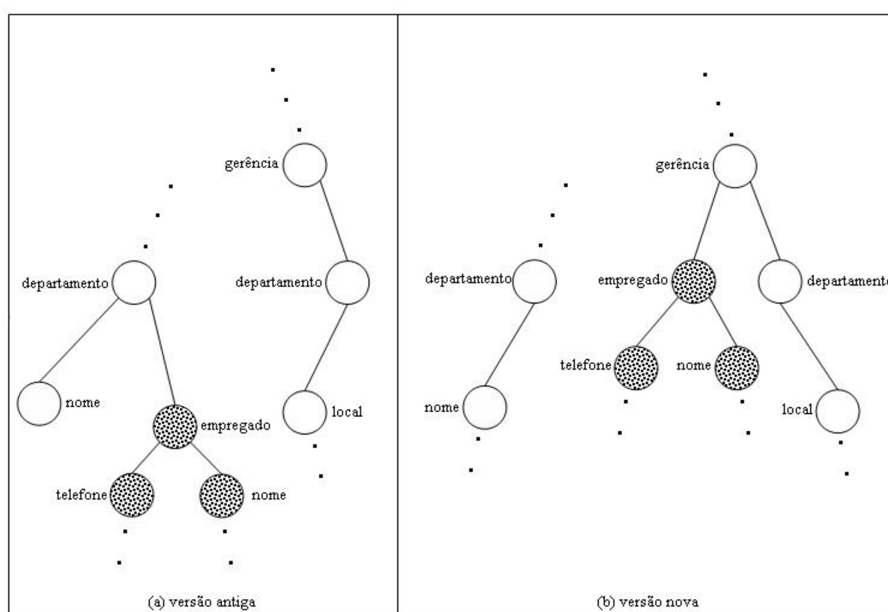


Figura 5.8: Movimentação de subárvore para outro nível

Resultados dos algoritmos:

- **X-Diff**: O X-Diff não possui operações de movimentação. Logo ele considerou que houve uma remoção na versão antiga do documento, seguida por uma inserção da subárvore na versão nova do documento.
- **XyDiff**: O XyDiff, possuindo avaliação de operações de movimentação, detecta incorretamente a alteração quando a subárvore movida não possui a mesma estrutura até a raiz. Ou seja, quando as estruturas ascendentes não eram idênticas, o algoritmo considerou a subárvore como removida da versão antiga e inserida na versão nova.

Análise: A não-detecção da operação de movimentação ocorre devido às características estruturais dos algoritmos, considerando os melhores candidatos aqueles que possuem a mesma estrutura desde o nodo raiz.

Solução utilizando chaves para XML: A utilização de uma chave independente de estrutura, ou a determinação do contexto onde esta subárvore se encontra, através do uso de chaves relativas, evitaria a não identificação deste tipo de operação.

5.5 Conclusões

Estes testes demonstram a fragilidade da análise somente estrutural na detecção de diferenças entre documentos e que a qualidade dos resultados obtidos pelas estratégias adotadas pelos dois algoritmos analisados variam de acordo com a situação. Os principais aspectos a serem destacados são os seguintes:

- **Alteração na estrutura**: Estes testes mostraram que ambos os algoritmos são extremamente sensíveis a mudanças estruturais no documento, especialmente quando tal mudança envolve a criação ou remoção de nodos internos. Em outras palavras, os algoritmos perdem a capacidade de identificar elementos quando eles são movidos para outros níveis. Neste caso, chaves para XML definidas com "//", que permitem a identificação de elementos independentemente do seu nível na árvore, são úteis para manter a identidade do nodo.

- **Comparações X Desempenho**: O custo de procurar casamentos entre nodos é mais alto se o algoritmo considerar o contexto destes nodos; ou seja, seus ascendentes, descendentes e irmãos. O estudo mostrou que não há garantia que uma simplificação de tal estratégia, como as que são empregadas pelos algoritmos analisados, produza bons resultados. Para garantir uma melhoria na qualidade seria necessário realizar uma análise estrutural mais complexa, com diversas comparações. Porém, com a estratégia baseada em chaves, a identificação seria mais precisa.

- **Semelhanças estruturais**: A presença de subárvores idênticas ou similares no documento pode induzir os algoritmos a fazer casamentos incorretos. Neste caso, para identificar unicamente nodos nestas subárvores, chaves absolutas não são suficientes, pois podem existir nodos que coincidem em seus valores. Assim, é necessário definir chaves relativas, que permitem identificar tais nodos no contexto de subárvores.

- **Propagação de erros**: A propagação de casamentos utilizando a estrutura do documento pode ocasionar problemas de casamentos errados. Por exemplo, a propagação de casamentos a ascendentes e descendentes, como é realizada pelo algoritmo XyDiff, pode propagar também os erros, uma vez que um casamento errado pode propagar outros casamentos também errados. Consequentemente, o *edit script* gerado pode ser demasiadamente grande e, portanto, difícil de ser analisado.

Deste estudo pode-se concluir que não existe uma única estratégia de análise estrutural do documento que sempre gere resultados semanticamente corretos, já que o comportamento esperado não depende da estrutura e sim da semântica dos dados. Assim, a proposta deste trabalho é que chaves para XML sejam definidas para os documentos que serão comparados, e que elas sejam utilizadas como entrada para algoritmos de *diff*. Este estudo auxiliou na definição da classe de chaves considerada nesta proposta. Dois aspectos importantes mostrados são que as chaves deveriam ser capazes de identificar nodos independentemente do seu nível na árvore e também dentro do contexto de subárvores. Assim, na definição do algoritmo XKeyMatch, será considerada uma classe de chaves para XML que permita o uso de chaves absolutas e relativas, bem como expressões de caminho em profundidade. Em [28], apenas chaves absolutas com caminhos simples são consideradas. No próximo capítulo, a proposta deste trabalho, que consiste na elaboração de um algoritmo semântico de *diff* utilizando chaves para XML, é apresentada.

Capítulo 6

Semântica em Algoritmos de Diff para XML

6.1 Introdução

Os resultados dos experimentos com os algoritmos X-Diff e XyDiff apresentados no capítulo anterior, bem como a análise destes resultados, mostraram que existem deficiências na detecção de diferenças entre documentos XML em algoritmos cuja análise é somente estrutural.

Os algoritmos apresentados no Capítulo 4, referentes a detecção de diferenças entre documentos XML, fazem pouca ou nenhuma utilização da semântica dos dados contidos nos documentos XML. O algoritmo XyDiff mostrou-se o único a utilizar tal característica, embora esta utilização seja extremamente limitada. O algoritmo utiliza, em uma etapa preliminar, a informação de atributos identificadores em DTDs. Conforme mostrado no Capítulo 3, este conceito de chaves é bastante simples, e possui uma série de limitações.

A proposta deste trabalho visa atenuar os problemas apontados pelos resultados dos testes realizados no capítulo anterior. Através de uma abordagem semântica, pretende-se verificar se este tipo de abordagem pode contribuir na detecção de diferenças entre documentos XML. Basicamente, a proposta consiste em um pré-processamento das informações contidas nas versões do documento XML a serem analisadas, visando detectar entidades semelhantes em tais documentos. Este pré-processamento utiliza as informações semânticas encontradas nas chaves para XML fornecidas pelo usuário. As chaves oferecem um meio de identificar univocamente porções de dados, oferecendo a segurança de que as informações casadas realmente trata-se de entidades equivalentes.

Deste modo, casamentos entre tais entidades poderiam ser realizados, e estas informações de casamentos podem então ser passadas a qualquer algoritmo de detecção de diferenças entre documentos XML, seja qual abordagem for. Este, por sua vez, pode realizar suas comparações, já possuindo informações de entidades equivalentes, diminuindo

a necessidade de processamentos e comparações por parte do algoritmo de *diff*.

Visando implementar esta solução, neste capítulo será proposto o algoritmo XKey-Match, um algoritmo para casamentos de entidades em diferentes versões de um documento XML. Este algoritmo utiliza a definição de chaves para XML descrita no Capítulo 3 para realizar um pré-processamento nas versões do documento, casando entidades e informando tais casamentos para um algoritmo de *diff*. O modelo de árvore para XML descrito no Capítulo 2 é utilizado para manipulação dos dados contidos nas versões do documento XML.

Uma visão geral deste algoritmo proposto e a descrição das etapas executadas por este são apresentadas nas seções a seguir.

6.2 Visão geral do algoritmo XKeyMatch

O algoritmo XKeyMatch recebe como entrada duas versões, v_{t-1} e v_t, e um conjunto de chaves para XML Σ que são válidas para ambas as versões. A saída é um conjunto Γ de pares de nodos (n_1, n_2), onde n_1 é um nodo em v_{t-1} que refere-se a mesma entidade n_2 (também um nodo) em v_t, de acordo com Σ. O conjunto Γ é então passado a um algoritmo de *diff* que, baseado nestes casamentos, faz a comparação das versões v_{t-1} e v_t. Como resultado, o algoritmo de *diff* gera o arquivo delta, contendo as informações sobre as mudanças ocorridas entre as versões. O desenho desta arquitetura, onde o algoritmo XKeyMatch se apresenta, é ilustrado na Figura 6.1.

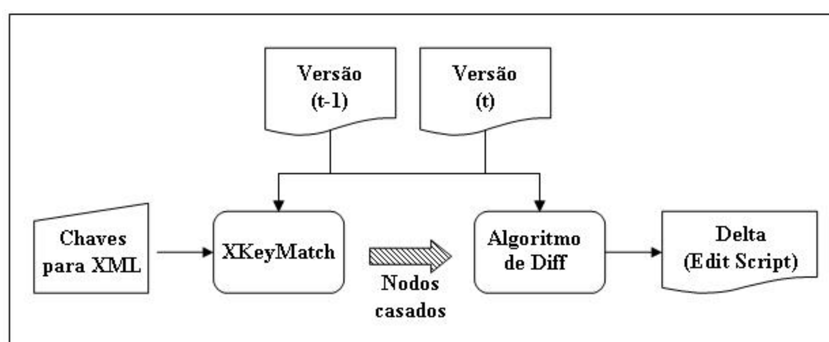


Figura 6.1: Arquitetura de um algoritmo de *diff* com o XKeyMatch

A primeira etapa do algoritmo XKeyMatch realiza a transformação das versões passadas como entrada em árvores XML, no intuito de possibilitar a manipulação destes dados. Logo, são geradas duas árvores XML, T_1 e T_2, baseadas nas versões v_{t-1} e v_t.

Na etapa seguinte, é realizada a construção de um autômato finito determinístico (AFD) a partir de um conjunto Σ de chaves para XML. Este AFD é chamado de KeyDFA(Σ). Com este AFD, todos os casamentos baseados em Σ podem ser executados com uma travessia nas árvores XML T_1 e T_2. Mais especificamente, o AFD representa um conjunto de caminhos definidos pelas chaves, bem como armazena informações a respeito destas. Cada passo da travessia na árvore XML corresponde a uma mudança de estado no autômato. As informações sobre chaves armazenadas nos estados do autômato são usadas para coletar nodos que são candidatos para casamentos.

A próxima etapa, relativa à seleção de candidatos, utiliza a estrutura KeyDFA(Σ) para realizar o processamento de cada árvore XML, obtendo, assim, informações de possíveis candidatos para casamentos, de acordo com o conjunto de chaves Σ.

As entidades identificadas pelas chaves, candidatas a casamentos, são utilizadas pela etapa de casamentos. Esta etapa compara as entidades coletadas da versão v_{t-1} com as entidades coletadas da versão v_t, identificando quais entidades ainda permanecem na nova versão, podendo, assim, serem casadas. Estas informações de casamentos são, então, passadas ao algoritmo de *diff*.

A Figura 6.2 mostra a constituição do algoritmo XKeyMatch, ilustrando as etapas executadas e os produtos gerados por estas.

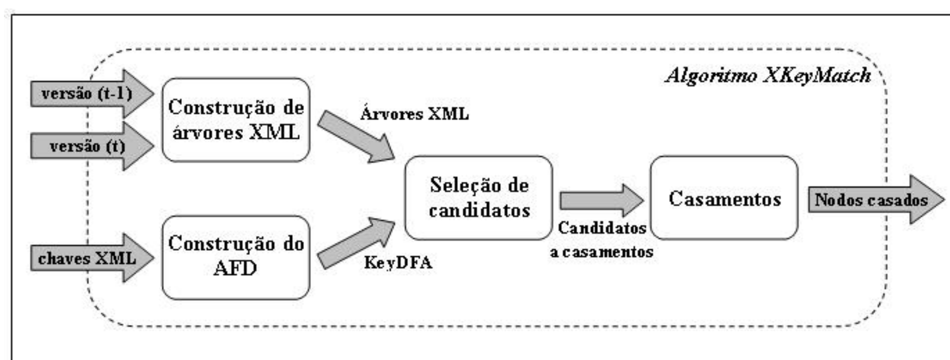


Figura 6.2: Algoritmo XKeyMatch

Estas etapas são executadas pelo algoritmo mostrado na Figura 6.3. Dadas duas versões de um documento XML, v_{t-1} e v_t, e um conjunto de chaves para XML Σ, primeiramente são construídas duas árvores XML, T_1 e T_2, baseadas nas versões v_{t-1} e v_t, respectivamente (Linhas 2 e 3). Após este passo, é construído o autômato KeyDFA(Σ) (Linha 4). Depois, os candidatos são selecionados pela função `get_candidates`, chamada

para ambas as árvores (Linhas 5 e 6). O conjunto de casamentos é computado pela pelo procedimento `match`, que compara os candidatos previamente coletados (Linha 7).

Algoritmo 1 Algoritmo XKeyMatch

```
1: function XKeyMatch(v_{t-1}, v_t, Σ)
2:     T1 ← XMLTreeConstruct(v_{t-1})
3:     T2 ← XMLTreeConstruct(v_t)
4:     KeyDFA ← DFA(Σ)
5:     candidates(T1) ← get_candidates(T1, KeyDFA)
6:     candidates(T2) ← get_candidates(T2, KeyDFA)
7:     return Match(candidates(T1), candidates(T2), T1, T2)
8: end function
```

Figura 6.3: Algoritmo XKeyMatch - Principal

As etapas do algoritmo estão descritas detalhadamente nas seções subseqüentes.

6.3 Construção das árvores XML

Como entrada, são passados ao algoritmo duas versões de um documento XML. Para que seja possível a leitura e manipulação de tais versões, estas são transformadas em árvores, representando a estrutura de cada versão. No algoritmo, o DOM (seção 2.3) é utilizado para este fim.

Portanto, a tarefa desta etapa é utilizar as versões v_{t-1} e v_t para a construção das árvores XML T_1 e T_2, de acordo com o modelo apresentado na Definição 2.3.1: $T = (V, lab, ele, att, val, r)$.

A construção é baseada na hierarquia do documento XML. Portanto, a hierarquia definida pelos elementos segue a hierarquia definida pelas marcas da versão do documento XML. Cada entidade de um documento XML representa um nodo na árvore. Este, por sua vez, pode ser de um dos seguintes tipos: elemento (E), atributo (A) ou texto (T).

Cada nodo possui um identificador. Este identificador é numérico, atribuído a cada nodo em um percurso em pós-ordem em cada árvore. Estes identificadores permitem que os nodos sejam acessados de maneira simples e única. Também permitem que seja passado ao algoritmo de *diff* subseqüente a informação de quais nodos foram casados pelo algoritmo XKeyMatch. Somente os nodos do tipo atributo não recebem identificadores, pois estes são únicos dentro de um nodo elemento. Logo, pode-se acessar um nodo atributo

através do identificador do nodo elemento que é pai deste, juntamente com o nome do atributo.

Um exemplo de duas versões de um documento XML em formato de árvore, identificando cada nodo através da numeração em um percurso em pós-ordem, é mostrado na Figura 6.4. Este exemplo será utilizado no decorrer deste capítulo, complementando exemplos subseqüentes.

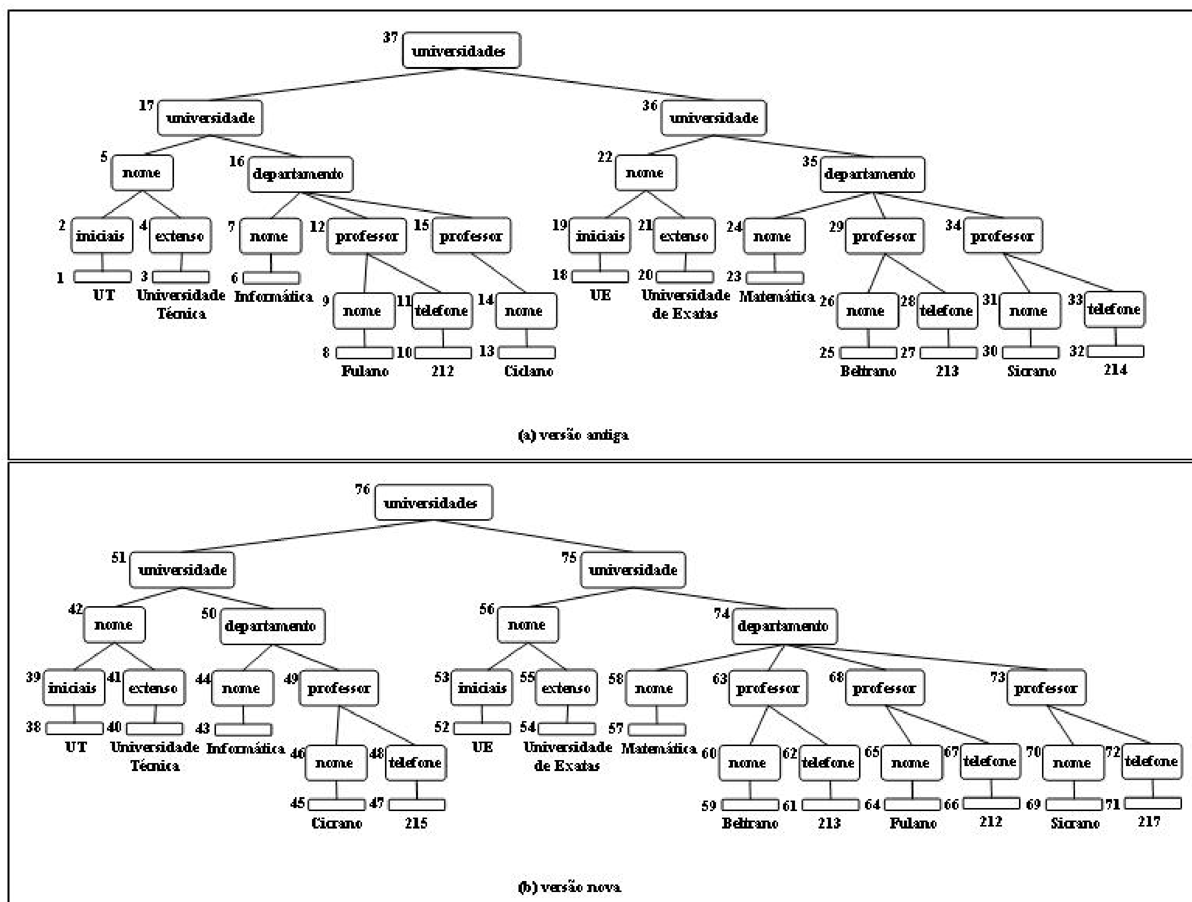


Figura 6.4: Versões de um documento XML sobre universidades

6.4 Construção do AFD

No contexto de documentos XML, os autômatos finitos são utilizados para realizar o processamento de expressões de caminho. Os principais trabalhos nesta área são relacionados ao processamento de expressões em XPath [22].

Tanto os autômatos finitos determinísticos quanto os não-determinísticos (AFN) são capazes de reconhecer precisamente conjuntos regulares. Entretanto, existe uma questão

de tempo/espaço a ser analisada: enquanto os autômatos finitos determinísticos podem levar a reconhecedores mais rápidos do que os não-determinísticos, um autômato finito determinístico possui um número maior de estados e transições que um autômato finito não determinístico equivalente [23].

O motivo pelo qual os autômatos finitos determinísticos determinam mais rapidamente cadeias de símbolos é o fato de possuir, no máximo, uma transição, a partir de cada estado, para qualquer símbolo de entrada, enquanto que os autômatos finitos não-determinísticos podem possuir múltiplas transições [23]. Isto significa, no caso do AFD, que existe somente um único percurso, rotulado por uma cadeia, a partir do estado inicial.

Esta questão é extremamente importante para justificar o uso de autômatos finitos determinísticos no processamento de expressões de caminho, uma vez que diminui drasticamente o espaço de busca neste processamento. Em um AFN podem existir vários percursos possíveis para uma mesma cadeia de entrada. Isto faz com que todos os caminhos possíveis tenham que ser considerados para encontrar um que leve à aceitação ou descobrir que nenhum deles o faz. Um exemplo é dado na Figura 6.5. Dada uma cadeia de símbolos a ser processada, onde o primeiro símbolo é `y` e o estado atual é `1`, existe a opção em seguir pela transição $\delta(1, y) = 2$ ou pela transição $\delta(1, \varepsilon) = 3$.

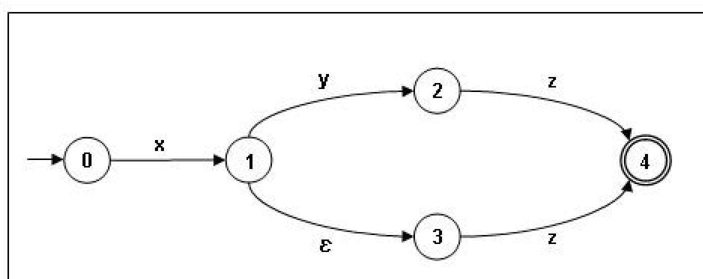


Figura 6.5: Caminhos opcionais em um AFN

Existe ainda uma outra questão relativa à construção de um autômato a partir de expressões de caminho. Esta construção é mais simples e direta transformando uma expressão de caminho em um autômato finito não-determinístico do que em um autômato finito determinístico.

Assim, a abordagem mais comum para a construção de um autômato para o processamento de expressões de caminho é a transformação da expressão em um autômato finito não-determinístico e posteriormente sua conversão em um determinístico.

Dado um conjunto Σ de chaves para XML, esta etapa do algoritmo XKeyMatch gera um autômato finito determinístico, chamado de KeyDFA(Σ). Cada estado deste autômato armazena informações para processar cada chave pertencente a Σ em uma única travessia sobre uma árvore XML T.

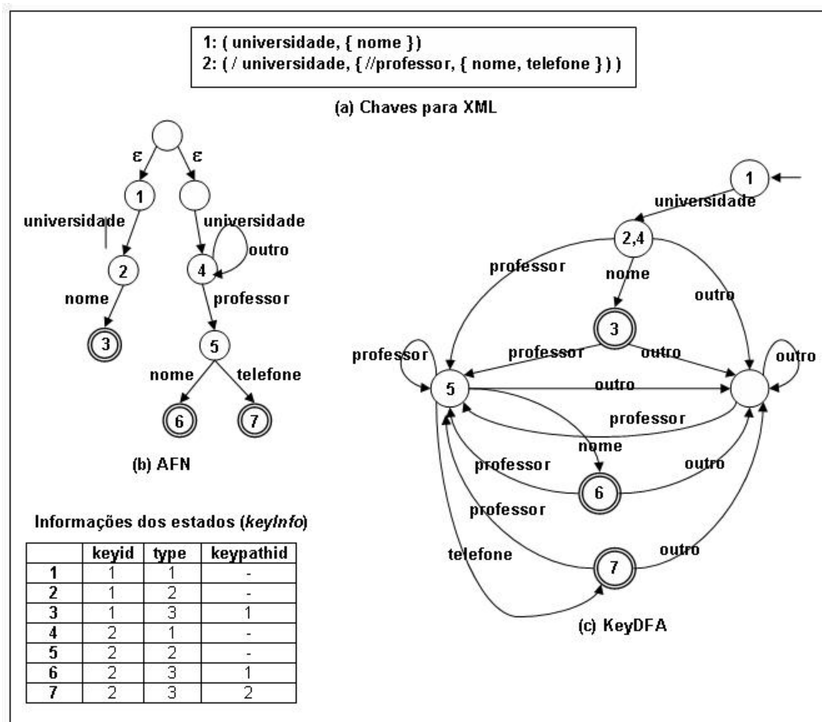
Seja $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, onde cada σ_i é escrito na forma $(Q_i, (Q'_i, \{P_i^1, ..., P_i^{n_i}\}))$. Primeiramente é necessário descrever um autômato finito não-determinístico $M(\Gamma_i)$, que será associado a cada chave σ_i em Σ. A construção de cada $M(\Gamma_i)$ inicia-se criando um AFN para cada caminho p em $\{Q_i, Q'_i, P_i^1, \ldots, P_i^{n_i}\}$, definido como $M(p) = (N_p, L_p \cup \{outro\}, \delta_p, S_p, F_p)$, 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, "outro" é uma palavra especial que significa qualquer palavra do alfabeto. Estes AFN's possuem uma "estrutura linear"; ou seja, se $p = l_1/\ldots/l_m$, então $\delta_p(S_p, l_1) = q_1$, para cada l_j, $1 \leq j < m$, $\delta_p(q_j, l_{j+1}) = q_{j+1}$, e $q_m = F_p$. Se p contém "//", então existe uma transição de um estado para ele mesmo com a palavra "outro". Ou seja, se $p = \ldots//l_j\ldots$ para algum j, então $\delta_p(q_{j-1}, outro) = q_{j-1}$, onde $q_0 = S_p$.

Além disto, os estados finais destes AFN's carregam informações sobre cada chave σ_i considerada para sua construção. A estrutura que armazena estas informações é chamada de `keyInfo`. Seja $F = \{F_{Q_i}, F_{Q'_i}, F_{P_i^1}, \ldots, F_{P_i^{n_i}}\}$. Para cada $f \in F$, *keyInfo*$[f]$ contém as seguintes informações:

- `keyId`: um identificador para σ_i;
- `type`: o valor deste campo é *context* se $f = F_{Q_i}$, *target* se $f = F_{Q'_i}$ e *keyPath* se $f = F_{P_i^j}$, $1 \leq j \leq n_i$;
- `keyPathId`: um identificador para cada caminho chave. Usado somente quando *keyInfo*$[f]$.*type* = *keyPath*; neste caso, se $f = F_{P_i^j}$, então *keyInfo*$[f]$.*keyPathId* $= j$.

O AFN $M(\Gamma_i)$ para a chave σ_i é obtido fazendo o estado final de $M(Q_i)$ coincidir com o estado inicial de $M(Q'_i)$, e fazendo o estado final de $M(Q'_i)$ coincidir com os estados iniciais de $M(P_i^j)$, $1 \leq j \leq n_i$. O AFN para todas as chaves em Σ, $M(\Sigma)$, é finalmente obtido criando um novo estado inicial, com transições-ϵ, dos estados inicias de todos os $M(\sigma_i)$, $1 \leq i \leq n$.

Um exemplo de um AFN construído utilizando a técnica descrita é mostrado na Figura 6.6(b).



	keyid	type	keypathid
1	1	1	-
2	1	2	-
3	1	3	1
4	2	1	-
5	2	2	-
6	2	3	1
7	2	3	2

Figura 6.6: Construção do AFD

Dado um AFN $M(\Sigma)$, KeyDFA(Σ) é obtido aplicando o algoritmo padrão de conversão AFN-AFD[23]. O autômato resultante para este exemplo é mostrado na Figura 6.6(c). Observe que, embora cada estado de $M(\Sigma)$ contenha informação de no máximo uma(1) chave em Σ, após a conversão, cada estado q' em KeyDFA(Σ) contém todas as informações de cada estado original (do AFN) representado por q'. Como exemplo, na Figura 6.6(c), $keyInfo(\{2,4\}) = \{keyInfo[2], keyInfo[4]\}$.

A construção do AFD descrita nesta seção é similar à definida em [22], que trata do processamento de *streams* de XML. Em [22], é avaliada a efetividade do processamento de um amplo número de expressões em XPath em *streams* quando o AFD é construído "preguiçosamente" (*lazily*). Esta abordagem de construção do AFD foi escolhida naquele trabalho devido ao crescimento exponencial do número de estados do AFD no crescimento do número de expressões. Aqui, embora tenha-se o mesmo problema, o número de chaves para cada documento é usualmente pequeno. Além disto, 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 em cada execução do algoritmo.

6.5 Seleção dos candidatos

Após construído o KeyDFA(Σ), o algoritmo XKeyMatch processa cada árvore XML passada como entrada, usando este autômato, obtendo, assim, informações de possíveis candidatos para casamentos, de acordo com o conjunto de chaves Σ. Seja T uma destas árvores, e KeyDFA(Σ) $= (Q, A, \delta, q_0, F)$. Começando com o nodo raiz de T e do estado inicial q_0 de KeyDFA, T é percorrido, e cada passo desta travessia corresponde a um passo no processamento do KeyDFA. Durante esta travessia, informações sobre valores de chave são coletadas usando os dados armazenados em cada estado q do autômato. Por exemplo, suponha que o estado atual de KeyDFA seja q quando a travessia em T se encontra no nodo n. Se *keyInfo*$[q]$ contiver informações de um caminho chave de $k \in \Sigma$, então o nodo chave n possui um valor de chave (*key value*, ou seja, valor que identifica unicamente uma entidade) para algum nodo n_t (nodo alvo, ou seja, o nodo que representa a entidade a ser identificada) e, portanto, pode-se associar n_t com $val(n)$. Observe que k pode conter mais de um(1) caminho chave P_i; ou seja $k = Q, (Q', \{P_1, \ldots, P_m\}))$, onde $m > 1$. Neste caso, n_t é identificado como um candidato para casamento somente se estiver associado com valores de todos os caminhos chave P_i, $1 \leq i \leq m$. Neste caso, dizemos que n_t é "chaveado" por k. Como exemplo, considere a Figura 6.7, que mostra uma árvore XML. Dada a chave mostrada na figura, o nodo `empregado` é o nodo n_t, chaveado por k. Os valores de chave que identificam o nodo empregado são os nodos `nome` e `matrícula`.

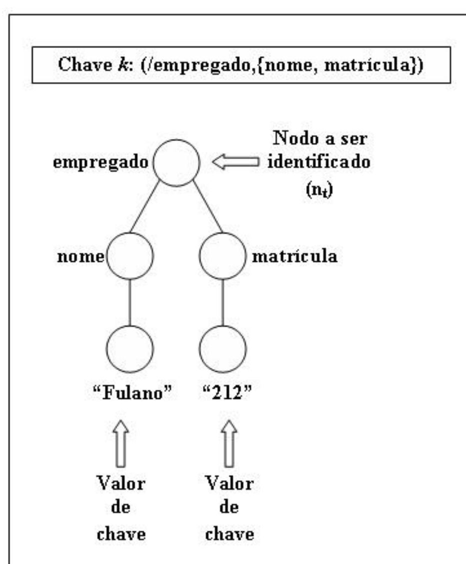


Figura 6.7: Valores de chave e nodo a ser identificado

Observe que quando procuram-se valores de chave para chaves relativas, 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 ao menos que n_c esteja chaveado. Em outras palavras, precisamos de nodos que sejam unicamente identificados até a raiz da árvore. Uma dificuldade ocorre quando o caminho contexto de uma chave contém "//". Como exemplo, considere a chave $(//A, (//B, \{C\}))$. Uma vez que o caminho da raiz para um nodo B (n_B) pode conter diversos nodos A, n_B pode ser chaveado relativamente com qualquer um destes. A Figura 6.8 ilustra um exemplo em que vários contextos são possíveis.

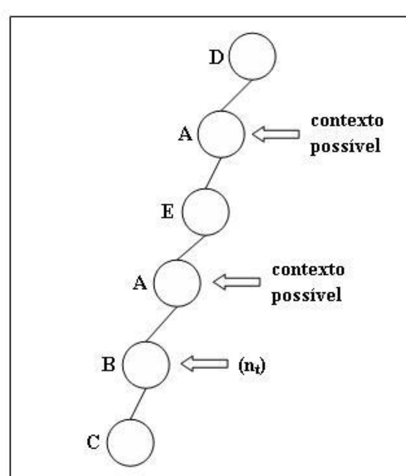


Figura 6.8: Diversos contextos possíveis para um nodo n_t

Para manter a informação necessária para determinar se um nodo é candidato a casamento, ao longo da travessia na árvore, associa-se a cada chave $k = (Q, (Q', \{P_1, \ldots, P_n\}))$ em Σ as seguintes informações, que serão armazenadas em uma estrutura chamada `keyVal`[k]:

- `contextNodes`: um conjunto de nodos em $[\![Q]\!]$ localizados ao longo de um caminho único a partir da raiz; ou seja, se `contextNodes` contém n_1 e n_2, ou n_1 é descendente ou ascendente do nodo n_2;

- `targetNode`: o último nodo visitado em $m[\![Q']\!]$ para algum m em `contextNodes`;

- `keyNode`: o último nodo visitado em `targetNode`$[\![P_i]\!]$.

- `keyPathId`: identificador do caminho chave i, $1 \leq i \leq n$;

O algoritmo de seleção dos candidatos é mostrado na Figura 6.9.

Algoritmo 2 Seleção dos candidatos

```
1:  function GET_CANDIDATES(T, KeyDFA(Σ) = (Q, A, δ, q0, F))
2:      keyValues ← {}
3:      for cada chave k em Σ do
4:          keyVal[k].contextNodes ← {}
5:      end for
6:      TRAVERSE_DFA_TREE(q0, r(T), keyValues)
7:      candidates ← {}
8:      for cada c ∈ nest_{keyPaths={keyPathId,keyPathNodeId}}(keyValues) do
9:          if c.keyId = k1, k1 = (Q, (Q', {P1, ..., Pn})), e para todo i em [1, n],
10:             c.keyPaths contém um elemento com keyPathId = i then
11:             candidates ← candidates ∪ {c}
12:         end if
13:     end for
14:     return candidates
15: end function

16: procedure TRAVERSE_DFA_TREE(state, node, keyValues)
17:     for cada v ∈ keyInfo[state] do
18:         if v.type = context then
19:             if existe c ∈ keyVal[v.keyId].contextNodes tal que node ∈ descendentes(c) then
20:                 keyVal[v.keyId].contextNodes ← keyVal[v.keyId].contextNodes ∪ node
21:             else
22:                 keyVal[v.keyId].contextNodes ← {node}
23:             end if
24:         else if v.type = target then
25:             keyVal[v.keyId].targetNode ← node
26:         else
27:             keyVal[v.keyId].keyPathNode ← node
28:             keyVal[v.keyId].keyPathId ← v.keyPathId
29:             for cada c em keyVal[v.keyId].contextNodes do
30:                 keyValues ← keyValues ∪ {[keyId, c, keyVal[v.keyId].targetNode, keyVal[v.keyId].keyPathNode,
31:                                            keyVal[v.keyId].keyPathId]}
32:             end for
33:         end if
34:     end for
35:     for cada nodo c em filhos(node) do
36:         if δ(state, lab(c)) = new_state then
37:             TRAVERSE_DFA_TREE(new_state, c, keyValues)
38:         else
39:             TRAVERSE_DFA_TREE(δ(state, outro), c, keyValues)
40:         end if
41:     end for
42: end procedure
```

Figura 6.9: Algoritmo XKeyMatch - Seleção dos Candidatos

Após a inicialização dos conjuntos `keyValues` e `keyVal[k]` para cada chave em Σ, o procedimento `traverse_dfa_tree` é chamado para obter todos os valores dos caminhos chave na árvore, começando da raiz $r(T)$ e do estado q_0 do KeyDFA(Σ) (Linhas 1 a 6). Aqui, `keyValues` é um conjunto de registros com os campos `keyId, contextNode, targetNode, keyPathNode, keyPathId`. Para determinar se um nodo contém valores para todos os caminhos chave de uma dada chave, é necessário o uso da função algébrica *nest*, que aninha os elementos em `keyValues` conforme segue: obtem-se o conjunto `keyPaths` de todos os valores de `[[keyPathNode, keyPathId]]` associados com o mesmo valor de `[keyId, contextNode, targetNode]`, e então é verificado se `keyPaths` contém valores para todos os caminhos chave de `keyID`. Se este for o caso, um registro

com estrutura aninhada é inserido no conjunto `candidates` (Linhas 7 a 13). Como exemplo da necessidade do uso desta função, considere a chave para XML e o documento da Figura 6.7. Se apenas o dado sobre o nome do empregado fosse coletado, não existindo dados sobre a matrícula, este candidato seria descartado. Isto porque a chave daquele exemplo exige dois valores de chave (`nome` e `matrícula`) para identificar unicamente o nodo `empregado`.

O procedimento *traverse_dfa_tree* é responsável por coletar informações, baseado nas chaves, ao longo da travessia na árvore com o KeyDFA(Σ). Observe que um nodo pode ser coletado por diferentes regras (como contexto, alvo, ou caminho chave) para chaves distintas em Σ, e esta informação é dada por `keyInfo[s]`, onde s é um estado de KeyDFA(Σ). Portanto, se o estado atual de KeyDFA(Σ), quando processa-se o nodo n, é s, para cada elemento v em `keyInfo[s]`, se v`.keyId` $= k$, então os valores em v são usados para preencher os campos de `keyVal[k]`. Toda vez que um nodo é identificado como um caminho chave de k, toda a informação coletada e armazenada em `keyVal[k]` é inserida em `keyValues` (Linhas 16 a 34). Uma vez que uma mesma chave pode ser definida em diferentes nodos contextos, como discutido anteriormente, um elemento é inserido em `keyValues` para cada nodo contexto c (Linhas 29 a 32). Para continuar a travessia, para cada filho c do nodo atual, é determinado o próximo estado do autômato de acordo com o nome de c. Assim, o procedimento *traverse_dfa_tree* é chamado recursivamente, tendo como parâmetros o próximo estado, o nodo c e a estrutura `keyValues` (Linhas 35 a 41).

Para exemplificar a execução desta etapa, considere as versões do documento da Figura 6.4 e a estrutura KeyDFA da Figura 6.6. A função `get_candidates`, para a versão v_{t-1}, gera os seguintes valores para o conjunto $candidates_1$, descritos na Tabela 6.1.

Tabela 6.1: Resultados da etapa de seleção de candidatos: v_{t-1}

	keyId	**contextNodes**	**targetNode**	**keyNodes/keyPathIds**
1	1	{[37]}	17	{[5/1]}
2	2	{[17]}	12	{[9,1],[11,2]}
3	1	{[37]}	36	{[22,1]}
4	2	{[36]}	29	{[26,1],[28,2]}
5	2	{[36]}	34	{[31,1],[33,2]}

Os candidatos 1 e 3 desta tabela foram coletados com dados da chave de keyId = 1 $((/universidade, \{nome\}))$. Já os candidatos 2, 4 e 5 foram coletados com dados da segunda chave, de keyId = 2 $((/universidade, (//professor, \{nome, telefone\})))$. Um aspecto a ser observado é que o nodo `professor`, de número 15, da versão v_{t-1}, não foi coletado, devido ao fato de que não possui valores para todos os caminhos chave da chave 2 (faltaram dados sobre telefones), conforme a análise feita nas linhas 9 a 11 do algoritmo desta etapa. Inicialmente os valores dos conjuntos keyNodes/keyPathIds, mostrados na Tabela 6.1, encontravam-se desunidos, sendo adicionados aos respectivos conjuntos pela função *nest* (Linha 8). Os candidatos desta tabela contém somente um (1) nodo para cada conjunto contextNodes. Isto porque não houve nenhum caso de diversos contextos possíveis durante a travessia realizada em um mesmo ramo de árvore.

Por sua vez, realizando a chamada da função `get_candidates`, para a versão v_t, tem-se os seguintes valores para o conjunto $candidates_2$, descritos na Tabela 6.2.

Tabela 6.2: Resultados da etapa de seleção de candidatos: v_t

	keyId	**contextNodes**	**targetNode**	**keyNodes/keyPathIds**
1	1	{[76]}	51	{[42,1]}
2	2	{[51]}	49	{[46,1],[48,2]}
3	1	{[76]}	75	{[56,1]}
4	2	{[75]}	63	{[60,1],[62,2]}
5	2	{[75]}	68	{[65,1],[67,2]}
6	2	{[75]}	73	{[70,1],[72,2]}

Nesta versão, os candidatos 1 e 3 foram coletados com dados da chave de keyId = 1 $((/universidade, \{nome\}))$ e os candidatos 2, 4, 5 e 6 foram coletados com dados da segunda chave, de keyId = 2 $((/universidade, (//professor, \{nome, telefone\})))$. Neste caso, não houve candidatos não coletados pela falta de valores de chave. Novamente, também não houve casos de diversos contextos para algum candidato coletado na versão v_t.

6.6 Casamentos

Dado um conjunto de candidatos para casamentos de cada árvore XML T_1 e T_2, o procedimento `match` procura por candidatos nestes conjuntos que possuem o mesmo valor de chave. Ou seja, procuram-se por nodos n_1 nos candidatos de T_1 e por nodos n_2 nos candidatos de T_2 que são chaveados pela mesma chave k e que coincidam nos valores de todos os caminhos chave de k. Caso k seja uma chave relativa, para que aconteça o casamento de n_1 com n_2, inclui-se a necessidade de que os contextos nos quais n_1 e n_2 se encontram também estejam casados. O procedimento `match` é mostrado na Figura 6.10.

Algoritmo 3 Casamentos

```
 1: function MATCH(candidates₁, candidates₂, T₁, T₂)
 2:     candidatePairs ← {}
 3:     for cada c₁ ∈ candidates₁ do
 4:         for cada c₂ ∈ candidates₂ do
 5:             if c1.keyId = c2.keyId then
 6:                 keyPairs ← {}
 7:                 for cada p₁ ∈ c1.keyPaths do
 8:                     for cada p₂ ∈ c2.keyPaths do
 9:                         if p₁.keyPathId = p₂.keyPathId e val(p₁.keyPathNode) =_v val(p₂.keyPathNode) then
10:                             keyPairs ← keyPairs ∪ {[keyPathId : p₁.keyPathId, keyPathNode₁ : p₁.keyPathNode,
11:                                                                   keyPathNode₂ : p₂.keyPathNode]}
12:                         end if
13:                     end for
14:                 end for
15:                 if keyPairs contém elementos para todos os caminhos chave da chave c₁.keyId then
16:                     candidatePairs ← candidatePairs ∪ {[keyId :p₁.keyId, v1 :[c₁.contextNode, c₁.targetNode],
17:                                                          v2 :[c₂.contextNode, c₂.targetNode], keyPairs :keyPairs]}
18:                 end if
19:             end if
20:         end for
21:     end for

22:     matches ← {}
23:     AuxMatches ← {[{r(T₁)}, {r(T₂)}]}
24:     while existir casamentos do
25:         for cada p ∈ candidatePairs do
26:             if existe [s₁, s₂] em AuxMatches tal que p.contextNode₁ ∈ s₁ e p.contextNode₂ ∈ s₂ then
27:                 matches ← matches ∪ {[p.targetNode₁, p.targetNode₂]}
28:                 for i ∈ {1, 2} do
29:                     tᵢ ← {}
30:                     node ← p.targetNodeᵢ
31:                     while node ≠ p.contextNodeᵢ do
32:                         tᵢ ← tᵢ ∪ {node}
33:                         node ← parent(node) in Tᵢ
34:                     end while
35:                 end for
36:                 AuxMatches ← AuxMatches ∪ [t₁, t₂]
37:                 for cada v em p.keyPairs do
38:                     matches ← matches ∪ {[v.keyPathNode1, v.keyPathNode2]}
39:                     for cada n₁ e n₂ que correspondem a nodos nas subárvores com raiz em v.keyPathNode1 e v.keyPathNode2 do
40:                         matches ← matches ∪ {[n₁, n₂]}
41:                     end for
42:                     for cada n₁ e n₂ que correspondem a nodos no caminho de p.v1.targetNode para v.keyPathNode1,
43:                             e de p.v2.targetNode para v.keyPathNode2 do
44:                         matches ← matches ∪ {[n₁, n₂]}
45:                     end for
46:                 end for
47:             end if
48:         end for
49:     end while

50:     return matches

51: end function
```

Figura 6.10: Algoritmo XKeyMatch - Casamentos

Este recebe, como entrada, os conjuntos de candidatos $candidates_1$ e $candidates_2$, respectivamente, de T_1 e T_2, e computa o conjunto de casamentos em dois passos. Pri-

meiro, para cada chave k em Σ, elementos c_1, c_2 em $candidates_1$ e $candidates_2$ têm seus valores comparados se eles se referem a mesma chave k (Linhas 1 a 14). Como exemplo, considerando novamente a árvore e a chave ilustradas na Figura 6.7, se um candidato c_1 em $candidates_1$ possuir esta árvore, e um candidato c_2 em $candidates_2$ possuir a mesma árvore, porém com os valores de matrícula diferentes, estes candidatos não poderiam ser considerados equivalentes, pois somente o valor dos nodos chave `nome` são iguais.

Para a comparação destes valores, foi utilizado o conceito de igualdade de valores, apresentado na Definição 3.3.2. Ou seja, nesta comparação, é avaliada a equivalência entre dois nodos e seus descendentes, caso estes os possuam. Se os valores coincidirem para cada caminho chave em k, então, uma entrada na variável `candidatePairs` é criada, contendo as seguintes informações (Linhas 15 a 18):

- `keyId`: identificador de chave k
- $\texttt{contextNode}_1$: o nodo contexto em T_1
- $\texttt{targetNode}_1$: o nodo alvo em T_1
- $\texttt{contextNode}_2$: o nodo contexto em T_2
- $\texttt{targetNode}_2$: o nodo alvo em T_2
- `keyPairs`: um conjunto de registros $[\texttt{keyPathId}, \texttt{keyPathNode}_1, \texttt{keyPathNode}_2]$, onde $\texttt{keyPathNode}_1$ e $\texttt{keyPathNode}_2$ são nodos chaves em T_1 e T_2, respectivamente, com o mesmo valor.

Em seguida, o algoritmo verifica os contextos de cada par de candidatos (Linhas 23 a 36). Ou seja, somente identificamos um par como casados se os nodos contextos já foram comparados e considerados equivalentes (Linha 26). Para auxiliar nesta verificação, usamos uma variável chamada $AuxMatches$, que armazena os nodos já casados entre as versões. Esta variável começa com os nodos raízes das árvores XML de cada versão.

Uma observação importante quanto aos nodos contextos é que o algoritmo não realiza a propagação de casamentos para contextos, e deixa para o algoritmo de *diff* a tarefa de determinar qual é o melhor casamento para os ascendentes. Para mostrar o motivo desta escolha, considere duas chaves, $k_x = (A/B, \{C\})$ e $k_y = (//A//B, \{C\})$, e as árvores XML da Figura 6.11.

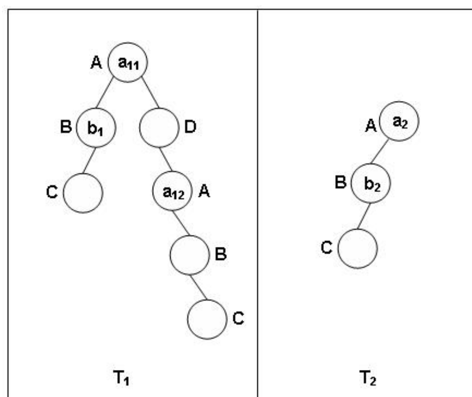


Figura 6.11: Situação de ambigüidade na propagação de casamentos

A chave k_x, utilizada para identificar um nodo b_1 em T_1, também indica que o valor do nodo C também identifica o nodo a_1^1, ascendente de b_1, uma vez que a estrutura de árvore garante que cada nodo B pode ser filho de um único nodo A. Isto é, a chave pode ser usada para propagar o casamento para ascendentes na árvore. Contudo, esta propagação não é direta se a chave considerada contiver "//". Agora, considerando a chave k_y, em T_1, existem nodos A a_1^1 e a_1^2 em um caminho da raiz para um nodo B, e em T_2 somente um (1) nodo A a_2 existe nas mesmas condições. Neste caso, não é claro para qual nodo A em T_1 o casamento deva ser propagado.

Contudo, informações sobre os nodos ascendentes podem ser necessárias para determinar casamentos utilizando chaves relativas. No exemplo dado anteriormente, considerando a chave k_y, não é relevante para qual nodo a_1^i o nodo a_2 é casado, mas este pode ser casado com qualquer um destes. Para manter esta informação, o algoritmo armazena em $AuxMatches$ um conjunto de pares $[s_1, s_2]$, onde s_1, s_2 são conjuntos de nodos que podem ser utilizados como contexto para um nodo alvo em T_1 e T_2, respectivamente, que foram casados previamente (Linhas 31 a 34). Neste exemplo, quando os nodos B são casados utilizando k_y, um par $[\{\{a_1^1, a_1^2, \ldots\}, \{a_2, \ldots\}]$ é incluído em $AuxMatches$. Portanto, quando são considerados os candidatos $[b_1, b_2]$ usando k_z, o algoritmo verifica se existe um par $[s_1, s_2]$ em $AuxMatches$ tal que s_1 contém o contexto de b_1 e s_2 contém o contexto de b_2. Se este for o caso, $[b_1, b_2]$ pode ser incluído no conjunto de casamentos.

Por fim, o procedimento inclui, para cada par de candidatos de nodos de caminho chave já incluso no conjunto de casamentos, todos os pares de descendentes de um nodo de caminho chave das duas versões e também todos os pares de ascendentes de nodos de caminho chave até os nodos de caminho alvo (Linhas 37 a 46).

Uma vez obtido o conjunto de pares de nodos casados, esta estrutura é passada como retorno (Linha 50) ao Algoritmo 1, concluindo, assim, a execução deste.

Considere novamente as versões do documento XML mostradas na Figura 6.4 para exemplificação desta etapa do algoritmo. Além destas, considere os valores coletados pela etapa de Seleção de Candidatos, demonstrados nas Tabelas 6.1 e 6.2. Executando a função `match`, tendo como parâmetros de entrada os candidatos das referidas tabelas, mais as árvores XML das versões dadas, tem-se, primeiramente, o preenchimento da variável candidatePairs (Linhas 1 a 21 do algoritmo desta etapa) com os valores mostrados na Tabela 6.3.

Tabela 6.3: Resultados da etapa de casamentos: *candidatesPairs*

	keyId	$contextNode_1$	$targetNode_1$	$contextNode_2$	$targetNode_2$	**keyPairs**
1	1	37	17	76	51	{[1,5,42]}
2	1	37	36	76	75	{[1,22,56]}
3	2	17	12	75	68	{[1,9,65],[2,11,67]}
4	2	36	29	75	63	{[1,26,60],[2,28,62]}

O par de candidatos 1 foi computado devido a união do candidato 1 de $candidates_1$ (Tabela 6.1) com o candidato 1 de $candidates_2$ (Tabela 6.2). A união foi possível devido a estes candidatos estarem de acordo com a definição da chave de keyId = 1 $((/universidade, \{nome\}))$, bem com o fato destes possuírem os mesmos valores de chaves para cada caminho chave (neste caso, o caminho chave `nome`). Por estes mesmos fatores, também ocorreu a união do candidato 3, de $candidates_1$, com o candidato 3 de $candidates_2$, resultando no par de candidatos 2.

Para a chave de keyId = 2 $((/universidade, (//professor, \{nome, telefone\})))$, foram computados os pares de candidatos 3 e 4, resultantes, respectivamente, da união do candidato 2, de $candidates_1$, com o candidato 5 de $candidates_2$, e da união do candidato 4, de $candidates_1$, com o candidato 4 de $candidates_2$. Isto ocorreu devido aos candidatos possuírem, em cada um dos pares, os mesmos valores de chaves para cada caminho chave (neste caso, os caminhos chave `nome` e `telefone`). O candidato 2 de $candidates_2$ por não possuir um candidato com caminhos chave equivalentes em $candidates_1$, foi descartado.

O mesmo caso não ocorreu para o candidato 5, de $candidates_1$, e 6, de $candidates_2$.

Estes, por sua vez, não puderam ser unidos em um par de candidatos, devido ao fato de que nem todos os caminhos chave de ambos possuem o mesmo valor de chave. Neste caso, o caminho chave não equivalente é o `telefone`, cujos nodos 33 e 72 não possuem igualdade de valor.

No intuito de efetivamente realizar os casamentos, preenchendo a estrutura que será retornada pela função `match`, são avaliados os contextos de cada par de candidatos (Linhas 22 a 50 do algoritmo desta etapa). Para os pares de candidatos 1 e 2, que são avaliados com a chave de keyId = 1, esta constatação é direta, pois esta é uma chave absoluta, sendo os contextos as raízes das árvores XML de cada versão. Logo, como o algoritmo sempre casa estes nodos raízes (Linha 23 do algoritmo desta etapa), os pares de candidatos 1 e 2 irão possuir seus pares de nodos casados.

Já para os pares de candidatos 3 e 4, avaliados com a chave de keyId = 2, esta verificação de contextos é necessária, pois tal chave é relativa, necessitando, portanto, do casamento dos contextos (nodos `universidade`) para tornar possível o casamento do nodos `professor`. Os nodos `universidade`, contextos desta chave, são casados devido a chave de keyId = 1. Isto torna possível o casamento dos pares de nodos do par de candidatos 4, pois o nodo 23 e o nodo 48, contextos, respectivamente, dos nodos 29 e 63, foram casados pela primeira chave. Porém, o par de candidatos 3 não pôde ter seus pares de nodos casados, pois seus nodos contextos não puderam ser casados (nodos 17 e 75).

A estrutura `matches`, representando os nodos casados pelo algoritmo XKeyMatch, para este exemplo, tem como valores os ilustrados na Tabela 6.4. Estes candidatos foram casados de acordo com as Linhas 37 a 46 do algoritmo desta etapa, cujos nodos ascendentes de cada nodo chave até o nodo alvo são casados, bem como seus nodos descendentes.

Tabela 6.4: Resultados da etapa de casamentos: *matches*

	Pares de candidatos	**Casamentos**
1	1	[17,51],[5,42],[2,39],[1,38],[4,41],[3,40]
2	2	[36,75],[22,56],[19,53],[18,52],[21,55],[20,54]
3	4	[29,63],[26,60],[25,59],[28,62],[27,61]

A efetividade do algoritmo XKeyMatch, proposto neste capítulo, em solucionar os problemas apresentados no estudo de caso do Capítulo 5, é o foco do próximo capítulo.

Capítulo 7

Experimentos

7.1 Introdução

Neste capítulo, são apresentados os resultados de experimentos conduzidos para avaliar a efetividade da proposta descrita no capítulo anterior, e para determinar se esta de fato soluciona os problemas detectados no estudo de caso reportado no Capítulo 5.

Para a execução destes experimentos, o algoritmo XKeyMatch foi implementado, em C++, utilizando DOM [5] para acessar e manipular os documentos XML. O XyDiff foi o algoritmo escolhido para receber, como entrada, o conjunto de casamentos encontrados pelo XKeyMatch e, assim, detectar as diferenças entre as versões de documentos XML. O XyDiff é um algoritmo que possui, como estratégia, o casamento das subárvores mais amplas primeiramente. Se todos os elementos equivalentes, de acordo com as chaves para XML, forem identificados pelo XKeyMatch, a estratégia do XyDiff torna-se adequada para a maioria dos casos. Além disto, a escolha do algoritmo XyDiff, em detrimento ao X-Diff, justifica-se pelo propósito do algoritmo XKeyMatch, cujo objetivo principal é a melhor representação possível das diferenças no que diz respeito ao aspecto semântico. O XyDiff detecta mais operações do que o X-Diff, com destaque à detecção de movimentações, ausente neste último. Para implementar a comunicação entre os algoritmos, algumas bibliotecas do XyDiff foram utilizadas.

O primeiro experimento realizado visa avaliar o comportamento do algoritmo quando existem mudanças na estrutura do documento XML. O segundo experimento revela como o algoritmo XKeyMatch melhora a qualidade dos resultados obtidos quando o documento contém várias subárvores com valores similares. Por último, o terceiro experimento mostra como alterações em nodos folhas do documento XML podem ser corretamente identificadas com uma verificação semântica. Em cada experimento, os resultados gerados pelo XyDiff com e sem o pré-processamento do algoritmo XKeyMatch são discutidos.

Logs gerados pelo algoritmo XKeyMatch são apresentados no Apêndice A. Neste apêndice também se encontram as saídas (*edit script*) do algoritmo XyDiff, tanto antes quanto depois da utilização do algoritmo XKeyMatch, mostrando a influência deste algoritmo na detecção de diferenças realizada pelo XyDiff.

7.2 Experimento 1

Para este experimento, foi utilizado um documento XML que contém dados sobre clubes de futebol. A Figura 7.1 ilustra duas versões do documento. Para cada clube, o documento contém seu nome e o ano de sua fundação. Na versão antiga do documento, estes clubes estão divididos por continentes, no caso Europa e América. Na versão nova, a estrutura do documento foi alterada. Agora, os clubes estão divididos por países, enquanto estes estão organizados por continentes. Portanto, a estrutura do documento inicialmente formada por continente/clube, foi redefinida como continente/país/clube.

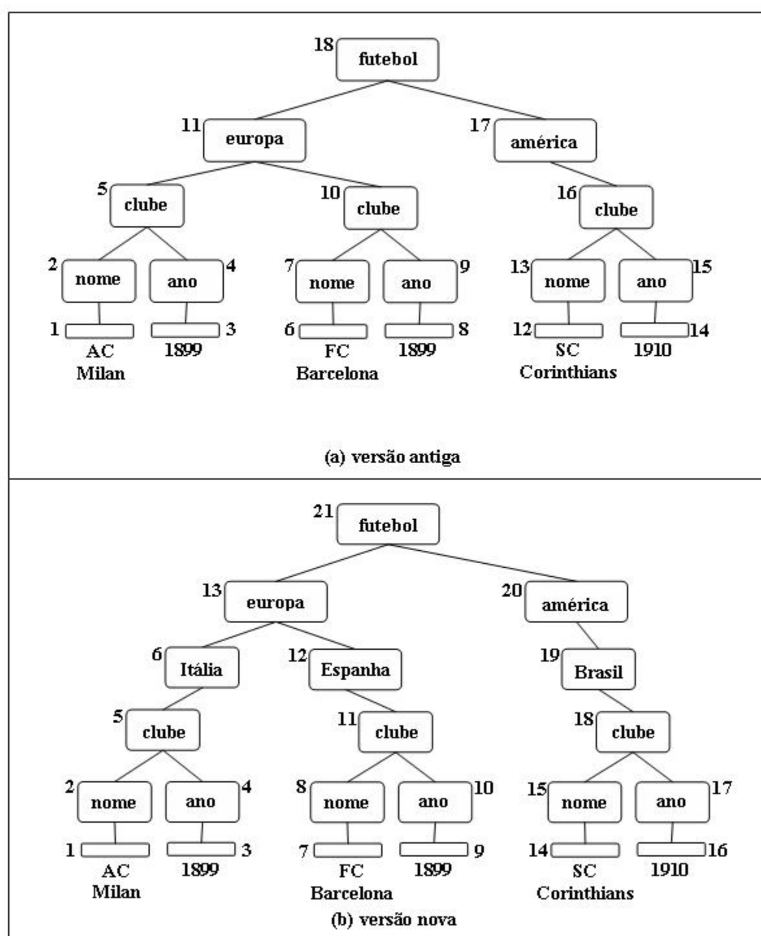


Figura 7.1: Versões de um documento XML sobre clubes de futebol

Algoritmos cuja análise do documento seja puramente *top-down*, como são os casos dos algoritmos XyDiff e X-Diff, não conseguem identificar que houve apenas uma mudança na estrutura do documento, com o acréscimo de mais um nível na sua hierarquia. No teste realizado no algoritmo XyDiff, este gerou, como resultado, a remoção das subárvores com raiz nos nodos `clube` e a inserção das subárvores com raiz nos elementos referentes aos países (`Itália`, `Espanha` e `Brasil`).

Do ponto de vista semântico, este resultado está incorreto, pois não houve remoção ou inserção de subárvores. Houve apenas o acréscimo de um nível na hierarquia do documento. O comportamento correto, neste sentido, seria o casamento das entidades, no caso, os clubes de futebol, em ambas as versões. Para alcançar este objetivo, a seguinte chave é definida e utilizada como entrada no algoritmo XKeyMatch:

(//clube,{nome})

Esta chave significa que, dentro de todo o contexto do documento, um clube é identificado pelo seu nome. Com o uso da expressão de caminho em profundidade (`//`), é possível identificar um clube independente do nível em que este se encontra no documento.

Utilizando esta chave para XML e as versões do documento XML como entradas para o algoritmo XKeyMatch, este obteve, como resultado, os casamentos entre as subárvores com raiz nos nodos `clube`. Ou seja, as entidades relacionadas aos clubes foram casadas.

Em posse desta informações, o algoritmo XyDiff foi executado. Desta vez, o algoritmo identificou que houve somente a mudança estrutural no documento, mostrando corretamente, como resultado, apenas a inserção dos nodos referentes aos países: `Itália`, `Espanha` e `Brasil`.

Uma possibilidade dada pelo algoritmo XyDiff é a utilização de uma DTD como entrada. Neste caso, atributos definidos como ID são usados para casar elementos anteriormente à etapa de análise estrutural. Esta etapa é ilustrada na Figura 7.2.

Neste experimento, o mesmo resultado gerado utilizando o XKeyMatch poderia ser obtido pela abordagem de casamento de ID's do XyDiff se no elemento `clube`, `nome` fosse um atributo do tipo ID. Isto mostra que casamentos baseados em ID's de DTD é uma abordagem interessante, mas sua aplicabilidade é limitada. Isto porque a definição de ID's é restrita a atributos e estes somente podem ser definidos no contexto de todo o documento.

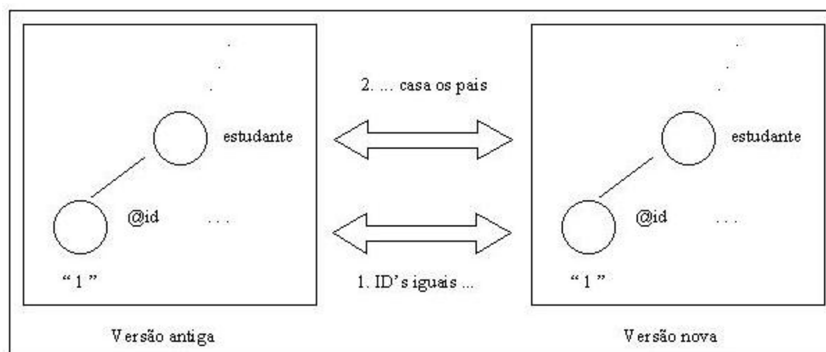


Figura 7.2: Casamentos via atributo ID da DTD

Para mostrar tal limitação, considere um documento XML que contenha dados sobre times de futebol e voleibol, contendo nome e ano de fundação destes. Este documento é ilustrado na Figura 7.3.

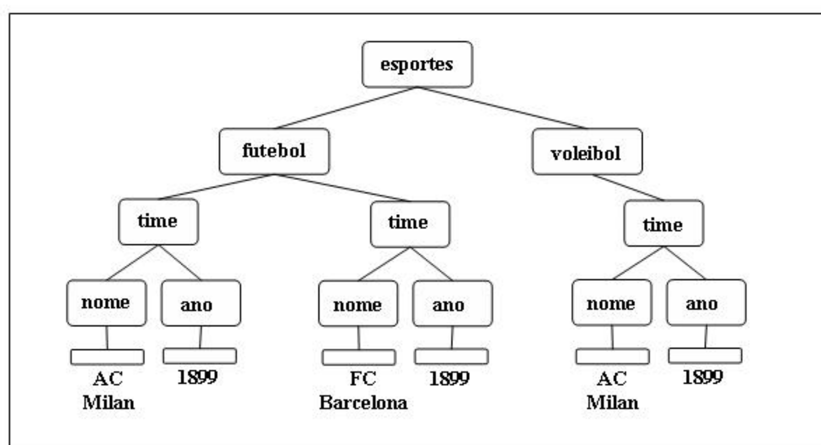


Figura 7.3: Versões de um documento XML sobre esportes

Considere que, neste documento, o elemento `nome` identifique um elemento `time` somente no contexto de cada esporte. Para este caso, as seguintes chaves poderiam ser definidas:

```
(/futebol,(//clube,{nome}))
(/voleibol,(//clube,{nome}))
```

Observe que, neste caso, o elemento `nome` não poderia ser definido como um atributo do tipo ID, uma vez que pode haver times de futebol e voleibol com o mesmo nome. Conforme mostra a Figura 7.3, isto ocorre no caso do clube `ACMilan`, que possui tanto um time de

futebol quanto um time de voleibol. Além disto, se a identificação do elemento `time` não fosse somente pelo elemento `nome`, mas sim uma subárvore com várias informações sobre tal time, esta identificação só poderia ser feita com o uso de chaves mais elaboradas, como as usadas pelo XKeyMatch.

7.3 Experimento 2

Neste segundo experimento, foram utilizadas as versões mostradas na Figura 7.4. Este documento XML possui informações relativas a uma vizinhança, divididas em famílias. Cada família possui um nome e informações relativas ao seu endereço atual.

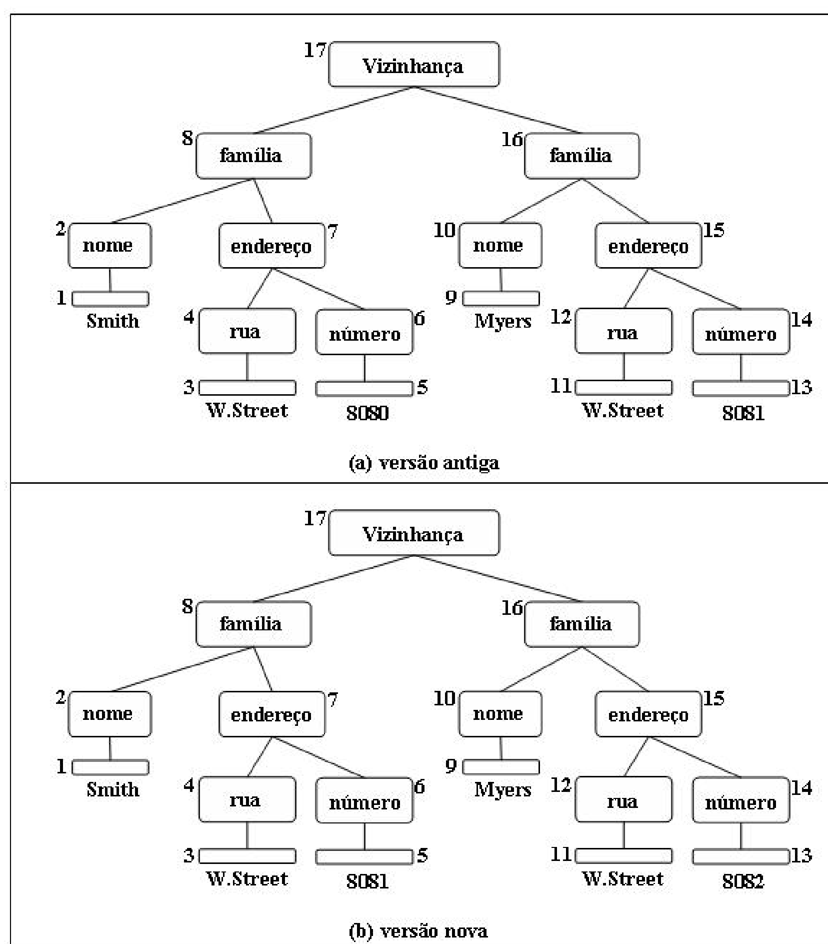


Figura 7.4: Versões de um documento XML sobre uma vizinhança

Considere, neste documento, a seguinte situação: a família Myers decide vender sua residência para a família Smith e morar em outra casa da vizinhança. Esta situação está ilustrada na nova versão do documento. A residência situada na W.Street, de número

8081, que na versão antiga pertencente à família Myers, agora está de posse da família Smith. A família Myers mudou-se para a casa de número 8082, na mesma rua.

Do ponto de vista semântico, o comportamento correto de um algoritmo de diff seria identificar que houve a mudança de endereço entre as famílias, casando as entidades relacionadas ao endereço de número 8081, indicando, então, esta movimentação.

Uma das principais estratégias do algoritmo XyDiff é o casamento de subárvores idênticas entre as versões e a propagação deste casamento aos ascendentes, começando pelas subárvores mais amplas, ou seja, subárvores com maior número de nodos descendentes. Esta estratégia é ilustrada na Figura 7.5.

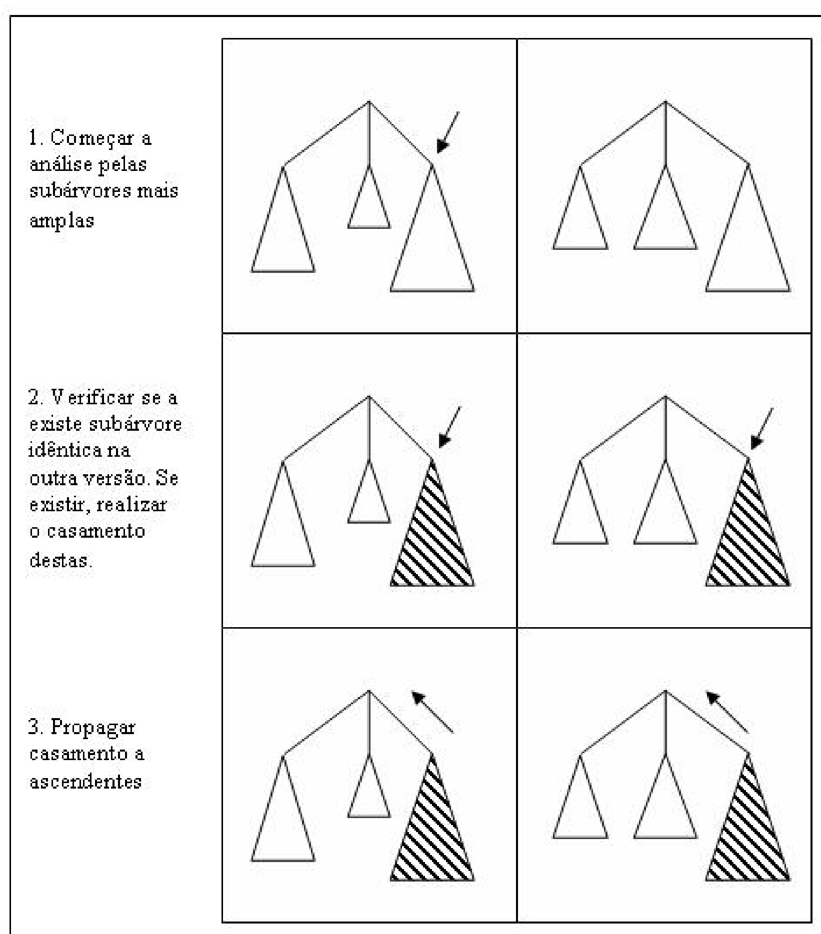


Figura 7.5: Casamentos de ascendentes

Como conseqüência, o algoritmo casa as subárvores do elemento `endereço` com o número 8081 de ambas as versões e, em seguida, propaga este casamento para os elementos `família`. Portanto, ao invés de identificar somente a mudança de endereços, o XyDiff realizou o casamento das famílias erradas, casando os nodos `family` de maneira incorreta.

Uma vez casando, de maneira incorreta, os elementos `familía`, o *edit script* resultante indica que o elemento `nome` da família foi alterado e não seu endereço. Isto provocou a identificação de movimentações de todas as subárvores entre os elementos `família`.

Nota-se, portanto, que a característica do algoritmo de casar os ascendentes, quando estes possuem, em ambas as versões, o mesmo nome (rótulo), nem sempre apresenta a melhor solução, uma vez que a semântica não é analisada para verificar se tais entidades ascendentes realmente devem ser casadas.

Para que o algoritmo possa identificar as famílias de maneira correta, uma informação sobre a semântica do documento, através de um chave para XML, precisa ser passada. A chave sugerida é a seguinte:

$$\texttt{(/família,\{nome\})}$$

Esta chave indica que, no contexto de todo o documento, uma família é identificada pelo nome.

Com esta chave para XML, mais as versões do documento XML, o XKeyMatch foi executado, resultando o casamento das famílias de maneira correta. Com estas informações, o XyDiff pode, então, ser executado. Como resultado, este, conforme esperado, identificou somente a mudança de endereço de número 8081 da família Myers para a família Smith, assim como identificou o novo endereço (8082) da família Myers.

Outra vez, a mesma observação sobre o uso de uma DTD por parte do XyDiff, como feita no experimento 1, pode ser feita aqui. Novamente, se `nome` não for um atributo do elemento `família` ou se a identificação de uma família fosse feita com uma estrutura mais complexa, não seria possível a identificação desta com o uso da DTD por parte do XyDiff.

7.4 Experimento 3

Para a realização deste último experimento, é utilizado um documento XML relativo a um planejamento e designação de tarefas a determinados empregados de um empresa. Duas versões deste documento XML são mostradas na Figura 7.6. Para cada empregado, são armazenados seu nome e uma relação de tarefas atribuídas a este. Dentro de uma relação de tarefas, cada tarefa possui um número associado. Conforme visto na figura, esta relação é representada pela subárvore com raiz no nodo `tarefas`.

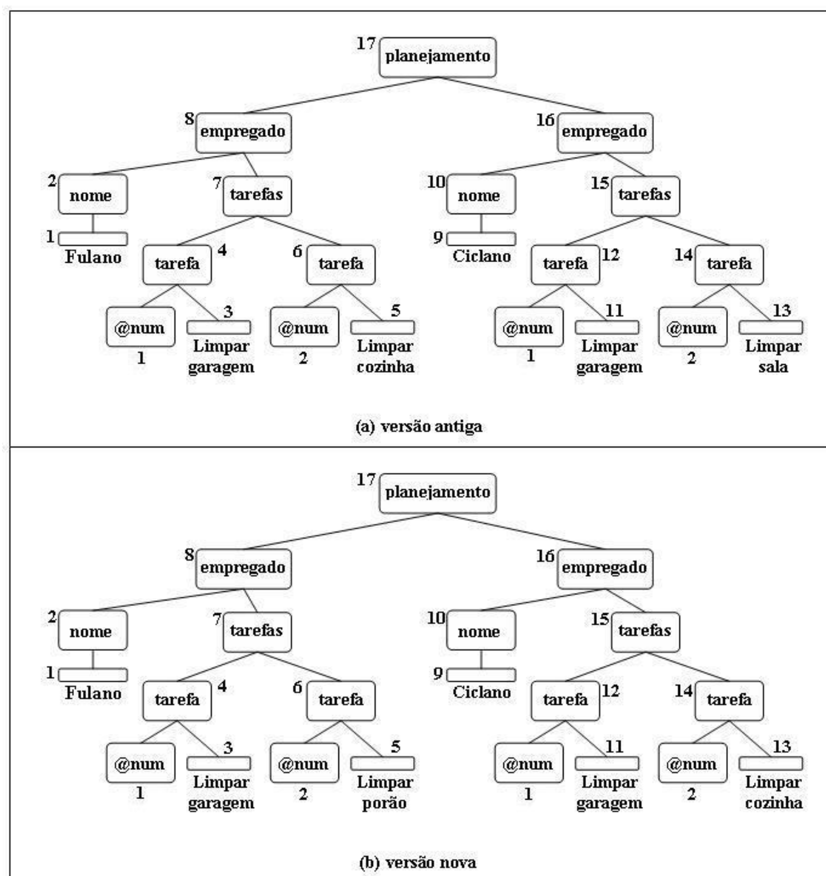


Figura 7.6: Versões de um documento XML sobre atribuição de tarefas

Suponha uma necessidade da empresa em designar novas tarefas a alguns de seus empregados, em substituição a outras tarefas anteriormente designadas a estes. Esta situação é representada na nova versão deste documento XML, apontando mudanças de atribuições de tarefas a dois empregados: o empregado de nome "Fulano" teve como nova atribuição a tarefa *Limpar porão* , em substituição à tarefa *Limpar cozinha*; já o empregado de nome "Ciclano" obteve como atribuição a tarefa *Limpar cozinha*, substituindo a tarefa *Limpar sala*.

Ao executar o algoritmo XyDiff com tais versões, sem a utilização do pré-processamento realizado pelo XKeyMatch, este trouxe, como resultado, o casamento da subárvore do elemento `tarefas` do empregado "Fulano" (da versão antiga) com a subárvore do elemento `tarefas` do empregado "Ciclano" (da versão nova). Isto provocou a propagação do casamento para os elementos `empregado` de cada uma das subárvores casadas, portanto, casando os empregados errados. Logo, esta propagação errônea gera, como conseqüência, um *edit script* com a informação de que os empregados foram trocados, provocando a

identificação de movimentações de todas as subárvores entre os elementos `empregado`.

Conforme pode-se observar, este problema é causado pelo mesmo motivo ocorrido no experimento 2: o casamento de subárvores propagam casamentos para seus ascendentes, enquanto estes tiverem o mesmo nome.

O comportamento correto, do ponto de vista semântico, seria a detecção, por parte do algoritmo XyDiff, da operação de alteração (*update*) do conteúdo dos nodos `tarefa` modificados. Ao contrário do experimento 2, em que o comportamento semântico adequado seria o casamento das entidades `endereço` e a detecção de operação de movimentação, neste experimento as operações de alteração dos nodos devem ser detectadas. O motivo pelo qual as entidades `endereço` devem ser casadas justifica-se pelo fato de que estas são entidades únicas dentro do documento. No caso das tarefas, a situação é outra, pois as tarefas não são entidades únicas, mas sim apenas uma informação pertencente à entidade `empregado`.

Estas situações distintas, mostradas nos experimentos 2 e 3, mostram que uma abordagem estrutural nem sempre é adequada para todos os casos, uma vez que a semântica, ou seja, as informações relativas ao documento, não são utilizadas para a avaliação e detecção de igualdade entre entidades de versões de um documento XML. Logo, a característica de identificação das subárvores mais amplas, realizada pelo XyDiff, para o caso do experimento 2, realizou com sucesso o casamento das entidades `endereco`, mas fracassou no caso do experimento 3, onde realizou o casamento das subárvores `tarefas` de maneira incorreta, no ponto de vista semântico.

Para contornar os problemas apresentados neste experimento, tanto do casamento incorreto das subárvores dos elementos `tarefas` quanto a propagação do casamento para os elementos `empregado`, é necessária a definição de chaves para XML que identifiquem corretamente as entidades `empregado` e que definam o contexto no qual cada relação de tarefas se apresenta. Estas chaves são as seguintes:

$$(\varepsilon,\texttt{(/empregado,\{nome\})})$$
$$\texttt{(/empregado,(//tarefa,\{@num\}))}$$

A primeira chave, de tipo absoluta, define que, dentro do contexto de todo o documento XML, um empregado é identificado por seu nome. Esta chave realizará adequadamente os casamentos das entidades `empregado`. Já a segunda chave, de tipo relativa, define que,

dentro do contexto de informações de um empregado, cada entidade `tarefa`, independente do nível em que se encontre abaixo deste, é identificado pelo atributo `num`. Com a identificação de cada empregado, realizada pela primeira chave, a segunda chave permitirá que, dentro da relação de tarefas de cada empregado, cada tarefa seja identificada corretamente pelo seu atributo identificador. Assim, a propagação de casamentos para ascendentes, realizada pelo XyDiff, desta vez irá efetuar esta ação corretamente, casando as entidades `tarefas` de maneira adequada.

Utilizando tais chaves e as versões do documento XML deste experimento, o algoritmo XKeyMatch foi executado, gerando os casamentos citados acima. Com tais informações, o algoritmo XyDiff foi executado e, desta vez, realizou corretamente a detecção de apenas as operações de alteração dos conteúdos dos nodos `tarefa` modificados.

A utilização de uma DTD como pré-processamento e casamento de nodos com atributos ID, implementada pelo algoritmo XyDiff, não possibilitaria a definição das chaves citadas como solução neste exemplo. Se cada atributo `num` de cada elemento `tarefa` fosse substituído por um atributo ID, o documento tornaria-se inválido, pois, neste exemplo, estes possuem valores iguais para atributos distintos (valor 1 em dois atributos, assim como o valor 2 associado aos atributos restantes). Também não seria possível definir uma chave para um determinado contexto, como a chave relativa deste exemplo.

Os experimentos apresentados neste capítulo demonstram que os algoritmos de *diff* para XML, utilizando somente uma abordagem estrutural na detecção de diferenças entre versões de documentos, podem realizar detecções falhas no aspecto semântico. Fica evidente, com tais experimentos, que o uso de uma abordagem semântica neste contexto, como a realizada pelo algoritmo XKeyMatch, pode melhorar, em diversos casos, a qualidade dos resultados obtidos por estes algoritmos de *diff*.

A efetividade da proposta depende muito da familiaridade do usuário com a estrutura e semântica dos documentos comparados. Ou seja, quanto maior a qualidade das chaves XML definidas, melhor é o resultado gerado pelo algoritmo XKeyMatch.

Capítulo 8

Conclusão

Este trabalho propôs uma nova abordagem no contexto de algoritmos de *diff* para XML. Ao contrário da maioria dos algoritmos de *diff*, cuja abordagem é limitada a uma análise estrutural sobre o documento, esta abordagem realiza uma análise semântica no documento XML, através do uso de chaves para XML. A abordagem consiste em um pré-processamento nas versões de um documento XML, utilizando chaves para XML, no intuito de identificar e casar entidades iguais. Estes casamentos são posteriormente passados a um algoritmo de *diff*, para que este possa realizar a detecção de diferenças com base nestes casamentos.

Um estudo de caso, analisando o comportamento de algoritmos de *diff*, foi realizado, no intuito de avaliar a qualidade semântica dos resultados apresentados pelos mesmos. Os resultados destes algoritmos, para a maioria das operações possíveis em documentos XML, apresentaram problemas semânticos, principalmente pelas más escolhas na etapa de realização de casamentos. Este estudo também auxiliou na definição da classe de chaves considerada nesta proposta, mostrando que para solucionar os problemas identificados durante os experimentos, o algoritmo XKeyMatch deveria considerar tanto chaves absolutas como relativas, bem como permitir a utilização de expressões de caminho em profundidade na sua definição.

A implementação deste algoritmo foi realizada, visando verificar a efetividade na solução de problemas apontados durante a análise realizada no estudo de caso. O algoritmo foi implementado como um pré-processamento de versões de documentos XML, realizando casamentos de entidades e passando tais informações para o algoritmo de *diff* XyDiff [3]. Este, por sua vez, pôde realizar suas tarefas em posse das informações passadas pelo XKeyMatch. Assim, foi possível analisar as saídas geradas pelo XyDiff, onde foram identificadas diversas melhorias na qualidade semântica dos resultados gerados.

As principais contribuições deste trabalho, para o tema abordado, foram as seguintes:

- A utilização de uma abordagem semântica no contexto de algoritmos de *diff* para XML, através do uso de chaves para XML com o objetivo de melhorar a qualidade dos algoritmos de *diff* para XML.
- A definição de um parâmetro de qualidade dos resultados gerados por algoritmos de *diff* para XML e a utilização deste, no estudo de caso realizado, para avaliação dos algoritmos X-Diff e XyDiff. Estudos comparativos existentes no contexto destes algoritmos, até então, somente avaliaram a performance e o tamanho dos resultados.
- Especificação e implementação de uma algoritmo de detecção de entidades para o contexto de algoritmos de *diff* para XML.

8.1 Trabalhos futuros

Como sugestões de trabalhos futuros neste tema, tem-se duas atividades relevantes: uma relacionada à realização de novos experimentos e outra relacionada à análise de complexidade do algoritmo XKeyMatch.

A primeira atividade consiste na realização de experimentos utilizando a abordagem proposta em massas de dados reais, analisando os resultados gerados e as chaves para XML úteis para este processo. Dados científicos, como os dados biológicos (e.g. Swiss-Prot [6]), possuem formato e comportamento adequados para tais experimentos.

Como segunda atividade, uma análise de complexidade deste algoritmo poderia ser realizada para analisar o impacto da fase de pré-processamento implementada por ele. Apesar da construção do autômato finito determinístico para o conjunto de chaves para XML poder causar um crescimento exponencial no número de estados, a quantidade de chaves definidas para um documento é geralmente pequena. Além disso, como alterações na definição de chaves são bastante raras, é possível considerar o armazenamento do AFD para sua posterior utilização no processamento de novas versões do documento. Como o custo da realização da fase de pré-processamento inclui um novo percurso sobre a árvore XML, além daquele feito pelo algoritmo de *diff* para análise estrutural, outro tópico para estudo futuro é investigar a possibilidade de combinar estes passos em um único percurso sobre a árvore.

REFERÊNCIAS

[1] ISO 8879:1986. Standard Generalized Markup Language (SGML). 2004. Disponível em http://www.iso.ch/cate/d16387.html.

[2] S. Abiteboul, G. Cobéna, e A. Marian. Detecting changes in XML documents. *ICDE'02*, 2002.

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

[4] R. Al-Ekram, A. Adma, e O. Baysal. diffx: an algorithm to detect changes in multi-version xml documents. *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, páginas 1–11. IBM Press, 2005.

[5] Vidur Apparao et al. Document Object Model (DOM) Level 1 Specification. outubro de 1998. Disponível em http://www.w3.org/TR/REC-DOM-Level-1/.

[6] A. Bairoch e R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. (28):15–18, 2000. Nuclear Acids Research.

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

[8] P. Buneman, S. Davidson, W. Fan, C. Hara, e W. Tan. Keys for XML. *WWW'10*, páginas 201–210, 2001.

[9] P. Buneman, S. Davidson, W. Fan, C. Hara, e W. Tan. Reasoning about keys for XML. *Database and programming languages*, 2001.

[10] S. Chawathe. Comparing hierarchical data in external memory. *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases*, páginas 90–101, Edinburgh, Scotland, U.K., 1999.

[11] S. Chawathe e H. Molina. Meaningful change detection in structured data. *SIGMOD*, páginas 26–37, 1997.

[12] Y. Chen, H. Davidson, e Y. Zheng. XKValidator: a constraint validator for XML. *CIKM'02*, 2002.

[13] J. Clark. XSL Transformations (XSLT). W3C Recommendation, novembro de 1999. Disponível em http://www.w3.org/TR/xslt.

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

[15] G. Cobena, T. Abdessalem, e Y. Hinnach. A comparative study for XML change detection. 2002. Disponível em http://citeseer.ist.psu.edu/696350.html.

[16] F. P. Curbera e D. A. Epstein. Fast Difference and Update of XML Documents. *XTech'99*, 1999.

[17] H.M. Deitel, P.J. Deitel, e T.R. Nieto. *Internet & world wide web: como programar*. Bookman, 2 edition, 2003.

[18] D. Martin et al. *Professional XML*. Ciência Moderna, Rio de Janeiro, BR, 2001.

[19] D. C. Fallside e P. Walmsley. XML schema part 0: Primer. 2000. Disponível em http://www.w3.org/TR/xmlschema-0/.

[20] W. Fan, P. Schwenzer, e K. Wu. Keys with upward wildcards for XML. *Lecture Notes in Computer Science*, 2113, 2001.

[21] FSF. Gnu diff. Disponível em http://www.gnu.org/software/diffutils/diffutils.html.

[22] T. Green, M. Onizuka, e D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. Relatório técnico, University of Washington, 2001.

[23] J. E. Hopcroft e J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Mass., 1979.

[24] Monsell EDM Ltd. Merging XML Changes with DeltaXML. Disponível em http://www.deltaxml.com/.

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

[26] E. W. Myers. An o(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[27] D. Raggett, A. L. Hors, e I. Jacobs. Hypertext Markup Language 4.01 Specification, 1999. Disponível em http://www.w3.org/TR/html4/.

[28] R. C. Santos e C. Hara. XKeyDiff - um algoritmo semântico para detecção de mudanças entre documentos XML. *REIC (Revista Eletronica de Iniciacao Cientifica)-ISSN 1519-8219*, (3), 2004.

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

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

[31] E. Titel. *Teoria e problemas de XML*. Bookman, 2003.

[32] W3C. World wide web consortium, 1994. Disponível em http://www.w3.org/.

[33] R. A. Wagner e M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, 1974.

[34] Y. Wang, D. J. DeWitt, e J. Cai. X-Diff: an effective change detection algorithm for XML documents. *ICDE*, páginas 519–530, 2003.

[35] H. Xu, Q. Wu, H. Wang, G.i Yang, e Y. Jia. Kf-diff+: Highly efficient change detection algorithm for xml documents. *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, páginas 1273–1286, London, UK, 2002. Springer-Verlag.

[36] K. Zhang, R. Stgatman, e D. Shasha. Simple fast algorithm for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18:1245–1262, 1989.

Apêndice A

Saídas dos Algoritmos XKeyMatch e XyDiff

A.1 Introdução

Neste apêndice, serão apresentadas as saídas (*logs*) geradas pelo algoritmo XKeyMatch, para cada um dos experimentos realizados no Capítulo 7, contendo comentários sobre cada saída descrita. Também serão mostradas as saídas geradas pelo algoritmo XyDiff para cada experimento realizado, consistindo em um *edit script*, ou seja, um documento XML que descreve as diferenças entre duas versões de um determinado documento. Será apresentada, para cada experimento, tanto a saída gerada sem a utilização do pré-processamento realizado pelo XKeyMatch, quanto a saída gerada utilizando o algoritmo de casamentos proposto neste trabalho.

Uma vez que o *edit script* gerado pelo XyDiff é um documento XML, existe a necessidade de uma explicação sobre o significado das nomes (rótulos) dos elementos definidos neste documento. Portanto, antes da apresentação dos resultados gerados pelo XKeyMatch para cada experimento, será apresentada uma explicação sobre a estrutura deste *edit script*.

A.2 Estrutura do edit script do XyDiff

O *edit script*, gerado pelo algoritmo XyDiff, é um documento XML contendo as diferenças detectadas pelo algoritmo para versões de um documento XML passadas como entrada para este. Este *edit script* possui a estrutura mostrada na Figura A.1.

```
<?xml version="1.0" encoding="ISO-8859-15" ?>
<unit_delta>
  <t to="versão v(t-1) do documento xml" from="versão v(t) do documento xml">operações</t>
</unit_delta>
```

Figura A.1: Estrutura do edit script do XyDiff

O documento XML apresentado nesta figura possui, na terceira linha, os nomes das versões que foram comparadas. Dentro da marca `t`, onde se encontra o texto `operações`, são listadas todas as operações necessárias para transformar a versão v_{t-1} na versão v_t.

As operações podem ser definidas tanto para elementos quanto para atributos. As operações para cada um destes tipos são listadas como segue:

- **Operações sobre atributos**:

<a(d ou i) a="nome atributo" v="valor do atributo" xid="ID"/>

<au a="nome atributo" nv="valor novo" ov="valor antigo" xid="ID"/>

<ai a="nome atributo" v="valor do atributo" new="true" xid="ID"/>

- **Operações sobre elementos**:

<(d ou i) xm="(IDs)" par="ID_pai" pos="num_posicao" (strong ou weak)move="yes"/>

<(d ou i) xm="(IDs)" par="ID_pai " pos="num_posicao"> subárvore </(d ou i)>

<(d ou i) xm="(IDs)" par="ID_pai " pos="num_posicao" update="yes">subárvore</(d ou i)>

Os itens presentes nas operações listadas acima são descritos nas Tabelas A.1 e A.2.

Tabela A.1: Operações possíveis do edit script do XyDiff

Operação	**Significado**
ai	*Attribute Insert* - indica a inserção de um atributo em um nodo
ad	*Attribute Delete* - indica a remoção de um atributo de um nodo
au	*Attribute Update* - indica a alteração do valor de um atributo de um nodo
d	*Delete* - indica a remoção de uma subárvore, ou indica uma operação de alteração ou mudança de posição
i	*Insert* - indica a inserção de uma subárvore, ou indica uma operação de alteração ou mudança de posição

Uma vez mostrada a estrutura do *edit script* gerado pelo XyDiff, as seções subseqüentes mostram, juntamente com a saída gerada por cada um dos experimentos realizados no Capítulo 7, as saídas geradas pelo XyDiff para cada um destes casos, com e sem o uso do algoritmo XKeyMatch.

Tabela A.2: Atributos das operações do XyDiff

Atributo	**Significado**
xm	ID do nodo ou da subárvore inserida, removida, alterada ou movida. Se existe mais de 1 nodo (ou seja, é uma subárvore), xm=(idinicial-idfinal)
par	ID do nodo pai
pos	Posição da subárvore indicada por xm dentro de seu pai, em relação aos seus irmãos
update=yes	Indica que o conteúdo da subárvore indicada por `xm` foi alterado. Se existe uma operação `d` com `update=yes`, necessariamente existe uma operação `i` com `update=yes` equivalente, sendo que o valor de `xm` muda na operação i
strongmove=yes	Indica que a subárvore indicada por `xm` possui pais diferentes entre as versões, indicados em `par`. Se existe uma operação `d` com `strongmove=yes`, necessariamente existe uma operação `i` com `strongmove=yes` equivalente, para outra pai `par`
weakmove=yes	Indica que a subárvore indicada por `xm` foi movida para outra posição indicada por `pos`. Se existe uma operação `d` com `weakmove=yes`, necessariamente existe uma operação `i` com `weakmove=yes` equivalente, para outra posição `pos`
new=true	Indica que o novo atributo é filho de um nodo também novo
a	Nome do atributo
v	Valor do atributo
nv e ov	Novo e antigo valor, respectivamente
xid	ID do nodo que possui o atributo

A.3 Saídas do experimento 1

Utilizando as versões do documento XML deste experimento (Figura 7.1), sobre mudanças estruturais realizadas no documento sobre clubes de futebol, foi gerado o seguinte *edit script* pelo XyDiff, ilustrado na Figura A.2.

```xml
<?xml version="1.0" encoding="ISO-8859-15" ?>
<unit_delta>
  <t IgnoreBlankTexts="yes" from="teams1.xml" to="teams2.xml">
    <d par="17" pos="1" xm="(12-16)">
      <clube>
        <nome>"SCCorinthiansPaulista"</nome>
        <ano>"1910"</ano>
      </clube>
    </d>
    <d par="11" pos="2" xm="(6-10)">
      <clube>
        <nome>"FCBarcelona"</nome>
        <ano>"1899"</ano>
      </clube>
    </d>
    <d par="11" pos="1" xm="(1-5)">
      <clube>
        <nome>"ACMilan"</nome>
        <ano>"1899"</ano>
      </clube>
    </d>
    <i par="11" pos="1" xm="(19-24)">
      <Italia>
        <clube>
          <nome>"ACMilan"</nome>
          <ano>"1899"</ano>
        </clube>
      </Italia>
    </i>
    <i par="11" pos="2" xm="(25-30)">
      <Espanha>
        <clube>
          <nome>"FCBarcelona"</nome>
          <ano>"1899"</ano>
        </clube>
      </Espanha>
    </i>
    <i par="17" pos="1" xm="(31-36)">
      <Brasil>
        <clube>
          <nome>"SCCorinthiansPaulista"</nome>
          <ano>"1910"</ano>
        </clube>
      </Brasil>
    </i>
  </t>
</unit_delta>
```

Figura A.2: Edit Script sem XKeyMatch - clubes

Com o uso do algoritmo XKeyMatch em conjunto com o algoritmo XyDiff, realizando o pré-processamento utilizando a chave definida no experimento, desta vez o seguinte *edit script* foi gerado, mostrado na Figura A.3.

```
<?xml version="1.0" encoding="ISO-8859-15" ?>
<unit_delta>
  <t IgnoreBlankTexts="yes" from="teams1.xml" to="teams2.xml">
    <i par="11" pos="1" xm="(19)">
      <italia />
    </i>
    <i par="11" pos="2" xm="(20)">
      <espanha />
    </i>
    <i par="17" pos="1" xm="(21)">
      <brasil />
    </i>
  </t>
</unit_delta>
```

Figura A.3: Edit Script com XKeyMatch - clubes

Para este experimento, o algoritmo XKeyMatch gerou o seguinte *log*, mostrando a execução das etapas do algoritmo.

```
++++++++++++++++++++++++++++++++++++++++
+ Etapa 1 - Construção das árvores XML +
++++++++++++++++++++++++++++++++++++++++
T1 - Ok !
T2 - Ok !

++++++++++++++++++++++++++++++++++
+ Etapa 2 - Construção do AFD +
++++++++++++++++++++++++++++++++++
Chaves de entrada **********************
numero da chave:  1
Token:  root
Token:  (
Token:  //
Token:  clube
Token:  {
Token:  nome
Token:  }

Estados do AFD *********************************
Est = 1 -> type = 1 keyId = 1 keypathId = 1
Est = 2 ->
Est = 3 -> type = 2 keyId = 1 keypathId = 1
Est = 4 -> type = 3 keyId = 1 keypathId = 1
```

```
Transicoes *************************************
( 1 , outro ) -> 2
( 1 , clube ) -> 3
( 2 , outro ) -> 2
( 2 , clube ) -> 3
( 3 , outro ) -> 2
( 3 , clube ) -> 3
( 3 , nome ) -> 4
( 4 , outro ) -> 2
( 4 , clube ) -> 3

++++++++++++++++++++++++++++++++++++
+ Etapa 3 - Seleção dos candidatos +
++++++++++++++++++++++++++++++++++++
Percorrendo as estruturas ...
Ok !

Dados da árvore T1
keyId = 1 contextNodes = 18 targetNode = 5 keyNodes/keyPathIds = {[2,1]}
keyId = 1 contextNodes = 18 targetNode = 10 keyNodes/keyPathIds = {[7,1]}
keyId = 1 contextNodes = 18 targetNode = 16 keyNodes/keyPathIds = {[13,1]}
Dados da árvore T2
keyId = 1 contextNodes = 21 targetNode = 5 keyNodes/keyPathIds = {[2,1]}
keyId = 1 contextNodes = 21 targetNode = 11 keyNodes/keyPathIds = {[8,1]}
keyId = 1 contextNodes = 21 targetNode = 18 keyNodes/keyPathIds = {[15,1]}

++++++++++++++++++++++++++++++++++++
+ Etapa 4 - Casamentos +
++++++++++++++++++++++++++++++++++++
Casa nodos ****************************
keypathnode1=2 keypathnode2=2
targetnode1=5 targetnode2=5

Casando próprios e filhos ...
Matching old(2 - nome) with new(2 - nome)
Matching old(1 - "ACMilan") with new(1 - "ACMilan")

Casando ancestrais ...
Matching old(5 - clube) with new(5 - clube)
```

```
Casa nodos ****************************
keypathnode1=7 keypathnode2=8
targetnode1=10 targetnode2=11

Casando próprios e filhos ...
Matching old(7 - nome) with new(8 - nome)
Matching old(6 - "FCBarcelona") with new(7 - "FCBarcelona")

Casando ancestrais ...
Matching old(10 - clube) with new(11 - clube)

Casa nodos ****************************
keypathnode1=13 keypathnode2=15
targetnode1=16 targetnode2=18

Casando próprios e filhos ...
Matching old(13 - nome) with new(15 - nome)
Matching old(12 - "SCCorinthiansPaulista") with new(14 - "SCCorinthiansPaulista")

Casando ancestrais ...
Matching old(16 - clube) with new(18 - clube)
```

Observe os valores presentes nos estados do AFD, referente ao tipo de informação armazena (*type*). Este pode ser: contexto (1), target (2) ou keypath (3).

Informações relativas ao casamento de contextos não apareceram no *log* deste experimento, pois trata-se de uma chave absoluta.

A.4 Saídas do experimento 2

Utilizando as versões do documento XML deste experimento (Figura 7.4), sobre alterações de endereço entre famílias de uma determinada vizinhança, foi gerado o seguinte *edit script* pelo XyDiff, ilustrado na Figura A.4.

Com o uso do algoritmo XKeyMatch em conjunto com o algoritmo XyDiff, realizando o pré-processamento utilizando a chave definida no experimento, desta vez o seguinte *edit script* foi gerado, mostrado na Figura A.5.

```
<?xml version="1.0" encoding="ISO-8859-15" ?>
<unit_delta>
  <t IgnoreBlankTexts="yes" from="neigh1.xml" to="neigh2.xml">
    <d par="16" pos="1" strongmove="yes" xm="(10)" />
    <d par="6" pos="1" update="yes" xm="(5)">"8080"</d>
    <d par="8" pos="1" strongmove="yes" xm="(2)" />
    <d par="17" pos="1" weakmove="yes" xm="(8)" />
    <i par="16" pos="1" strongmove="yes" xm="(2)" />
    <i par="17" pos="2" weakmove="yes" xm="(8)" />
    <i par="8" pos="1" strongmove="yes" xm="(10)" />
    <i par="6" pos="1" update="yes" xm="(18)">"8082"</i>
  </t>
</unit_delta>
```

Figura A.4: Edit Script sem XKeyMatch - vizinhança

```
<?xml version="1.0" encoding="ISO-8859-15" ?>
<unit_delta>
  <t IgnoreBlankTexts="yes" from="neigh1.xml" to="neigh2.xml">
    <d par="16" pos="2" strongmove="yes" xm="(15)" />
    <d par="6" pos="1" update="yes" xm="(5)">"8080"</d>
    <d par="8" pos="2" strongmove="yes" xm="(7)" />
    <i par="8" pos="2" strongmove="yes" xm="(15)" />
    <i par="16" pos="2" strongmove="yes" xm="(7)" />
    <i par="6" pos="1" update="yes" xm="(18)">"8082"</i>
  </t>
</unit_delta>
```

Figura A.5: Edit Script com XKeyMatch - vizinhança

Para este experimento, o algoritmo XKeyMatch gerou o seguinte *log*, mostrando a execução das etapas do algoritmo.

```
++++++++++++++++++++++++++++++++++++++++
+ Etapa 1 - Construção das árvores XML +
++++++++++++++++++++++++++++++++++++++++
T1 - Ok !
T2 - Ok !


++++++++++++++++++++++++++++++++++
+ Etapa 2 - Construção do AFD +
++++++++++++++++++++++++++++++++++
Chaves de entrada **********************
numero da chave:  1
Token:  root
Token:  (
Token:  familia
```

```
Token:  {

Token:  nome

Token:  }

Estados do AFD ********************************

Est = 1 -> type = 1 keyId = 1 keypathId = 1

Est = 2 -> type = 2 keyId = 1 keypathId = 1

Est = 3 -> type = 2 keyId = 1 keypathId = 1

Transicoes ************************************

( 1 , familia ) -> 2

( 2 , nome ) -> 3

++++++++++++++++++++++++++++++++++++

+ Etapa 3 - Seleção dos candidatos +

++++++++++++++++++++++++++++++++++++

Percorrendo as estruturas ...

Ok !

Dados da árvore T1

keyId = 1 contextNodes = 17 targetNode = 8 keyNodes/keyPathIds = {[2,1]}

keyId = 1 contextNodes = 17 targetNode = 16 keyNodes/keyPathIds = {[10,1]}

Dados da árvore T2

keyId = 1 contextNodes = 17 targetNode = 8 keyNodes/keyPathIds = {[2,1]}

keyId = 1 contextNodes = 17 targetNode = 16 keyNodes/keyPathIds = {[10,1]}

++++++++++++++++++++++++++++++++++++

+ Etapa 4 - Casamentos +

++++++++++++++++++++++++++++++++++++

Casa nodos ***************************

keypathnode1=2 keypathnode2=2

targetnode1=8 targetnode2=8
```

```
Casando próprios e filhos ...

Matching old(2 - nome) with new(2 - nome)

Matching old(1 - "Smith") with new(1 - "Smith")


Casando ancestrais ...

Matching old(8 - familia) with new(8 - familia)


Casa nodos ****************************

keypathnode1=10 keypathnode2=10

targetnode1=16 targetnode2=16


Casando próprios e filhos ...

Matching old(10 - nome) with new(10 - nome)

Matching old(9 - "Myers") with new(9 - "Myers")


Casando ancestrais ...

Matching old(16 - familia) with new(16 - familia)
```

A mesma observação apontada na saída do experimento 1, relativa ao casamento de contextos, é inerente também a este experimento.

Nodos inseridos ou que sofreram alteração (*update*) recebem novos identificadores.

A.5 Saídas do experimento 3

Utilizando as versões do documento XML deste experimento (Figura 7.6), sobre alterações de designações de tarefas atribuídas a empregados, foi gerado o seguinte *edit script* pelo XyDiff, ilustrado na Figura A.6.

Com o uso do algoritmo XKeyMatch em conjunto com o algoritmo XyDiff, realizando o pré-processamento utilizando a chave definida no experimento, desta vez o seguinte *edit script* foi gerado, mostrado na Figura A.7.

```
<?xml version="1.0" encoding="ISO-8859-15" ?>
<unit_delta>
- <t IgnoreBlankTexts="yes" from="work1.xml" to="work2.xml">
    <d par="14" pos="1" update="yes" xm="(13)">"Limparsala"</d>
    <d par="16" pos="1" strongmove="yes" xm="(10)" />
    <d par="17" pos="2" weakmove="yes" xm="(16)" />
    <d par="8" pos="1" strongmove="yes" xm="(2)" />
    <i par="17" pos="1" weakmove="yes" xm="(16)" />
    <i par="16" pos="1" strongmove="yes" xm="(2)" />
    <i par="14" pos="1" update="yes" xm="(18)">"Limparporao"</i>
    <i par="8" pos="1" strongmove="yes" xm="(10)" />
  </t>
</unit_delta>
```

Figura A.6: Edit Script sem XKeyMatch - tarefas

```
<?xml version="1.0" encoding="ISO-8859-15" ?>
<unit_delta>
- <t IgnoreBlankTexts="yes" from="work1.xml" to="work2.xml">
    <d par="6" pos="1" update="yes" xm="(5)">"Limparcozinha"</d>
    <d par="14" pos="1" update="yes" xm="(13)">"Limparsala"</d>
    <i par="14" pos="1" update="yes" xm="(19)">"Limparcozinha"</i>
    <i par="6" pos="1" update="yes" xm="(18)">"Limparporao"</i>
  </t>
</unit_delta>
```

Figura A.7: Edit Script com XKeyMatch - tarefas

Para este experimento, o algoritmo XKeyMatch gerou o seguinte *log*, mostrando a execução das etapas do algoritmo.

```
+++++++++++++++++++++++++++++++++++++++

+ Etapa 1 - Construção das árvores XML +

+++++++++++++++++++++++++++++++++++++++

T1 - Ok !

T2 - Ok !


+++++++++++++++++++++++++++++++++

+ Etapa 2 - Construção do AFD +

+++++++++++++++++++++++++++++++++

Chaves de entrada *********************

numero da chave:  1

Token:  root

Token:  (

Token:  empregado

Token:  {
```

```
Token:  nome

Token:  }

numero da chave:  2

Token:  empregado

Token:  (

Token:  //

Token:  tarefa

Token:  {

Token:  @num

Token:  }


Estados do AFD *********************************

Est = 1 -> type = 1 keyId = 1 keypathId = 1

Est = 2 -> type = 2 keyId = 1 keypathId = 1 type = 1 keyId = 2 keypathId = 1

Est = 3 ->

Est = 4 -> type = 3 keyId = 1 keypathId = 1

Est = 5 -> type = 2 keyId = 2 keypathId = 1

Est = 6 -> type = 3 keyId = 2 keypathId = 1


Transicoes *************************************

( 1 , empregado ) -> 2

( 2 , nome ) -> 4

( 2 , outro ) -> 3

( 2 , tarefa ) -> 5

( 3 , outro ) -> 3

( 3 , tarefa ) -> 5

( 4 , outro ) -> 3

( 4 , tarefa ) -> 5

( 5 , outro ) -> 3

( 5 , tarefa ) -> 5

( 5 , @num ) -> 6

( 6 , outro ) -> 3
```

```
( 6 , tarefa ) -> 5

++++++++++++++++++++++++++++++++++++
+ Etapa 3 - Seleção dos candidatos +
++++++++++++++++++++++++++++++++++++
Percorrendo as estruturas ...
Ok !

Dados da árvore T1
keyId = 1 contextNodes = 17 targetNode = 8 keyNodes/keyPathIds = {[2,1]}
keyId = 2 contextNodes = 8 targetNode = 4 keyNodes/keyPathIds = {[0,1]}
keyId = 2 contextNodes = 8 targetNode = 6 keyNodes/keyPathIds = {[-1,1]}
keyId = 1 contextNodes = 17 targetNode = 16 keyNodes/keyPathIds = {[10,1]}
keyId = 2 contextNodes = 16 targetNode = 12 keyNodes/keyPathIds = {[-2,1]}
keyId = 2 contextNodes = 16 targetNode = 14 keyNodes/keyPathIds = {[-3,1]}
Dados da árvore T2
keyId = 1 contextNodes = 17 targetNode = 8 keyNodes/keyPathIds = {[2,1]}
keyId = 2 contextNodes = 8 targetNode = 4 keyNodes/keyPathIds = {[-4,1]}
keyId = 2 contextNodes = 8 targetNode = 6 keyNodes/keyPathIds = {[-5,1]}
keyId = 1 contextNodes = 17 targetNode = 16 keyNodes/keyPathIds = {[10,1]}
keyId = 2 contextNodes = 16 targetNode = 12 keyNodes/keyPathIds = {[-6,1]}
keyId = 2 contextNodes = 16 targetNode = 14 keyNodes/keyPathIds = {[-7,1]}

++++++++++++++++++++++++++++++++++++
+ Etapa 4 - Casamentos +
++++++++++++++++++++++++++++++++++++
Casa nodos ****************************
keypathnode1=2 keypathnode2=2
targetnode1=8 targetnode2=8

Casando próprios e filhos ...
Matching old(2 - nome) with new(2 - nome)
```

```
Matching old(1 - "Fulano") with new(1 - "Fulano")

Casando ancestrais ...
Matching old(8 - empregado) with new(8 - empregado)

Casa nodos ****************************
keypathnode1=0 keypathnode2=-4
targetnode1=4 targetnode2=4

Casando ancestrais ...
Matching old(4 - tarefa) with new(4 - tarefa)

ListaDependencia ****************************
keypathnode1=0 keypathnode2=-6
targetnode1=4 targetnode2=12
contextnode1=8 contextnode2=16

Casa nodos ****************************
keypathnode1=-1 keypathnode2=-5
targetnode1=6 targetnode2=6

Casando ancestrais ...
Matching old(6 - tarefa) with new(6 - tarefa)

ListaDependencia ****************************
keypathnode1=-1 keypathnode2=-7
targetnode1=6 targetnode2=14
contextnode1=8 contextnode2=16

Casa nodos ****************************
keypathnode1=10 keypathnode2=10
targetnode1=16 targetnode2=16
```

```
Casando próprios e filhos ...

Matching old(10 - nome) with new(10 - nome)

Matching old(9 - "Ciclano") with new(9 - "Ciclano")


Casando ancestrais ...

Matching old(16 - empregado) with new(16 - empregado)


ListaDependencia *****************************

keypathnode1=-2 keypathnode2=-4

targetnode1=12 targetnode2=4

contextnode1=16 contextnode2=8


Casa nodos *****************************

keypathnode1=-2 keypathnode2=-6

targetnode1=12 targetnode2=12


Casando ancestrais ...

Matching old(12 - tarefa) with new(12 - tarefa)


ListaDependencia *****************************

keypathnode1=-3 keypathnode2=-5

targetnode1=14 targetnode2=6

contextnode1=16 contextnode2=8


Casa nodos *****************************

keypathnode1=-3 keypathnode2=-7

targetnode1=14 targetnode2=14


Casando ancestrais ...

Matching old(14 - tarefa) with new(14 - tarefa)
```

Desta vez, informações sobre os contextos são utilizadas para realizar os casamentos. Ao verificar pares de candidatos a casamentos, caso estes não possuam seus contextos casados, tal par irá para uma lista de dependências, aguardando que seus contextos sejam casados.

Uma observação sobre a saída gerada pelo XKeyMatch para este experimento: nodos atributos não possuem ID próprio, armazenando o valor de indexação a uma lista de atributos, acrescentado por um sinal negativo ($-$). Neste exemplo, os atributos estão numerados de -7 a 0.