UNIVERSIDADE FEEVALE

JONATHAN EGÍDIO SZABLEVSKI DE MOURA

EXTRAÇÃO AUTOMÁTICA DE DADOS ESTRUTURADOS DE VAGAS DE EMPREGO EM PÁGINAS WEB

Novo Hamburgo

2018

JONATHAN EGÍDIO SZABLEVSKI DE MOURA

EXTRAÇÃO AUTOMÁTICA DE DADOS ESTRUTURADOS DE VAGAS DE EMPREGO EM PÁGINAS WEB

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Feevale

Orientador: Rodrigo Rafael Villarreal Goulart

Novo Hamburgo
2018

JONATHAN EGÍDIO SZABLEVSKI DE MOURA

EXTRAÇÃO AUTOMÁTICA DE DADOS ESTRUTURADOS DE VAGAS DE EMPREGO EM PÁGINAS WEB

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Feevale

APROVADO EM: ___ / ___ / ______

____
RODRIGO RAFAEL VILLARREAL GOULART
Orientador – Feevale

____
DANIEL DALALANA BERTOGLIO
Examinador interno – Feevale

____
GUILLERMO NUDELMAN HESS
Examinador interno – Feevale

Novo Hamburgo
2018

RESUMO

Este trabalho trata da extração automática de dados estruturados a partir de páginas HTML. Programas que extraem dados estruturados a partir de dados semiestruturados em páginas na *web* são chamados de *wrappers*. Técnicas de extração automática permitem que aplicações extraiam dados sem a necessidade de intervenção humana no processo de criação dos *wrappers*. A pesquisa explora um dos problemas encontrados dentro do contexto de atuação da *startup* Jober. O Jober utiliza uma abordagem manual para o desenvolvimento de *wrappers* para coleta de dados relacionados a vagas de emprego em diferentes *websites*, o que limita a sua capacidade de escalar sua operação. A pesquisa tem como objetivo a proposta do protótipo de um *software* capaz de extrair dados estruturados a partir de páginas *web* contendo vagas de emprego, de modo automatizado e não supervisionado. São apresentadas os conceitos e técnicas de extração automática mais relevantes para o cenário analisado. A proposta de solução é discutida, apresentando-se as etapas de geração de *wrappers*, extração de dados e identificação da lista de vagas. O protótipo é avaliado através de testes de extração em páginas de listagem de diferentes *websites*. Os resultados demonstram que o protótipo é capaz de identificar a lista de vagas na maioria das situações, porém muitos dados irrelevantes são extraídos junto das informações de cada vaga.

Palavras-chave: Processamento de linguagem natural. Mineração de textos. *Wrapper*. Extração automática.

ABSTRACT

This work treats about the automatic extraction of structured data from HTML pages. Programs that extract strutured data from semi-structured data in web pages are called wrappers. Automatic extraction techniques allow applications to extract data without the need of human intervention in the process of wrapper development. The research explores one of the problems found in the context of operation of the startup Jober. The Jober uses a manual approach for the development of wrappers for the gathering of data related to job opportunities in different websites, what limits its capacity of scaling the operation. The research has as objective the proposal of the prototype of a software capable of extracting structured data from web pages containing job postings, in an automated and non-supervised way. The most relevant automatic extraction concepts and techniques for the studied scenario are presented. The proposed solution is discussed, presenting the stages of wrapper generation, data extraction and job list identification. The prototype is evaluated through extraction tests on listing pages of different websites. The results demonstrate that the prototype is able to identify the job list in most situations, but many irrelevant data is extracted next to the information of each job.

Keywords: Natural-language processing. Text mining. Wrapper. Automatic extraction.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de listas aninhadas 15
Figura 2 – O algoritmo *G-STM* 18
Figura 3 – O alinhamento de uma matriz de pares com três árvores 23
Figura 4 – Segmento de página contendo registros de dados 25
Figura 5 – Exemplo de árvore DOM contendo registros de dados 27
Figura 6 – Algoritmo para cálculo de correspondência entre árvores 34
Figura 7 – Pseudocódigo da função *Gstm* 35
Figura 8 – Exemplo da geração de *strings* a partir de árvores similares 37
Figura 9 – Pseudocódigo da função *DetectLists* 38
Figura 10 – Examplo de geração de gramática para uma *string* de símbolos 40
Figura 11 – Pseudocódigo da função *GrammarGeneration* 41
Figura 12 – Pseudocódigo da função *UpdateW* 42
Figura 13 – Pseudocódigo das funções *RegularExpression* e *ProcessStatePattern* . . 44
Figura 14 – Exemplo de gramática com *jumps* para o estado q_0 45
Figura 15 – Pseudocódigo da função *CollapseJumps* 46
Figura 16 – Pseudocódigo da função *ExtendConjunction* 46
Figura 17 – Pseudocódigo das funções *ExtractData* e *ExtractExpression* 47
Figura 18 – Pseudocódigo da função *ExtractConjunction* 48
Figura 19 – Pseudocódigo da função *ExtractDisjunction* 48
Figura 20 – Pseudocódigo da função *ExtractOptional* 49
Figura 21 – Pseudocódigo da função *ExtractRepeat* 49
Figura 22 – Pseudocódigo da função *ExtractSymbol* 50
Figura 23 – Exemplo da correspondência entre conjuntos *DataItemSet* 53

LISTA DE TABELAS

Tabela 1 – Resultados da comparação entre G-STM e RoadRunner 20
Tabela 2 – Resultados da comparação entre G-STM e EXALG 20
Tabela 3 – Resultados da comparação entre DEPTA e FiVaTech 24
Tabela 4 – Resultados experimentais em páginas One-DR no *dataset* TBDW versão 1.02 29
Tabela 5 – Resultados experimentais em páginas One-DR no *dataset* dos autores . 30
Tabela 6 – Resultados experimentais em páginas One-ASO no *dataset* TBDW versão 1.02 30
Tabela 7 – Resultados experimentais em páginas One-ASO no *dataset* dos autores 30
Tabela 8 – Resultados da extração de vagas do protótipo 53
Tabela 9 – Resultados da extração de itens de dados do protótipo 54

LISTA DE ABREVIATURAS E SIGLAS

AutoRM	*Automatic Data Record Mining*
DOM	*Document Object Model*
G-STM	*Generalized Tree Matching*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
OWL	*Ontology Web Language*
PTA	*Partial Tree Alignment*
RDF	*Resource Description Framework*
STM	*Simple Tree Matching*
URL	*Uniform Resource Locator*

SUMÁRIO

1 Introdução **10**

2 Embasamento Teórico **13**

2.1 *A Generalized Tree Matching Algorithm Considering Nested Lists* 14

2.1.1 Introdução 14

2.1.2 Formulação do cenário e problema 16

2.1.3 Proposta de solução 17

2.1.4 Avaliação da proposta 19

2.1.5 Conclusões 20

2.2 *FiVaTech: Page-level web data extraction from template pages* 21

2.2.1 Introdução 21

2.2.2 Formulação do cenário e problema 21

2.2.3 Proposta de solução 22

2.2.4 Avaliação da proposta 23

2.2.5 Conclusões 24

2.3 *AutoRM: An effective approach for automatic web data record mining* . . . 24

2.3.1 Introdução 24

2.3.2 Formulação do cenário e problema 25

2.3.3 Proposta de solução 26

2.3.4 Avaliação da proposta 29

2.3.5 Conclusões 30

3 Proposta de Solução **32**

3.1 Geração Automática de *Wrapper* 33

3.1.1 Função *Gstm* 34

3.1.2 Função *DetectLists* 35

3.1.3 Função *GrammarGeneration* 38

3.1.4 Função *UpdateW* 40

3.2 Extração de Dados 42

3.2.1 Expressões Regulares 43

3.2.2 Extração de itens de dados 46

3.3 Identificação da Lista Principal 49

4 Avaliação do Protótipo **51**

4.1 Método de avaliação 51

4.2 Resultados 53

5 Conclusão **55**

Referências **57**

1 INTRODUÇÃO

Atualmente, a *web* é a grande fonte de conhecimento da população em geral. Considerada como uma novidade promissora décadas atrás, sua relevância no dia a dia das pessoas cresceu de modo significativo, e hoje funciona como plataforma para uma grande esfera de *websites* e aplicações. Dentro deste contexto, é possível perceber que o acesso à imensa quantidade de dados presentes em páginas HTML (*HyperText Markup Language*) possibilita a criação de aplicações ainda mais complexas e relevantes, ou, como explica Liu (2011, pág. 363, tradução nossa): "Com mais e mais empresas e organizações disseminando informações na *web*, a habilidade de extrair esses dados das páginas *web* está se tornando cada vez mais importante".

Um dos grandes desafios para a utilização dos dados presentes na *web* reside no fato de que o formato HTML foi concebido originalmente apenas como um meio de transmissão de dados, fazendo com que a sua natureza desestruturada dificulte consultas mais elaboradas (CRESCENZI et al., 2001). Embora existam propostas já estabelecidas com relação à utilização de dados estruturados na *web*, como as tecnologias de *web* semântica, RDF (*Resource Description Framework*) e OWL (*Ontology Web Language*) (BIZER et al., 2005), até o presente momento não há uma estimativa de curto prazo para que essas práticas sejam adotadas de maneira massiva pela comunidade de internautas.

Existem muitos *websites* que possuem páginas "semiestruturadas": coleções de páginas dinâmicas, que são geradas a partir de um mesmo *template* com base em dados relacionais (normalmente fornecidos por um sistema de banco de dados). Com o objetivo de explorar essa característica, diversos pesquisadores estudaram técnicas para o desenvolvimento de *wrappers*: programas que realizam a extração de dados estruturados a partir de estruturas similares no HTML exibido por *websites* dinâmicos.

De acordo com Liu (2011), existem três abordagens principais para a construção de *wrappers*: uma abordagem manual, a indução de *wrappers* e a extração automática. Na abordagem manual, é necessário que o programador analise o código-fonte da página em busca dos padrões, construindo um programa que extraia os dados pertinentes. A indução de *wrappers* consiste na extração semi-automática de dados, onde o programa obtém regras de extração a partir de múltiplas páginas "rotuladas" manualmente, usando essas regras para extrair dados de outras páginas similares. Finalmente, a extração automática busca realizar a extração não supervisionada dos dados a partir de padrões encontrados em uma ou mais páginas. As primeiras pesquisas sobre a extração automática a receberem notoriedade foram os algoritmos RoadRunner (CRESCENZI et al., 2001) e EXALG (ARASU; GARCIA-MOLINA, 2003). Desde então, pesquisadores vêm desenvolvendo novas técnicas para melhorar a qualidade de sistemas de extração de dados.

O tema desta pesquisa tem como cenário o estudo de caso de um dos problemas encontrados dentro do ambiente de trabalho da *startup* Jober. Jober é uma empresa que tem como missão agregar vagas de emprego de inúmeros sites na internet e exibi-las aos usuários de acordo com cada perfil. Para garantir a escalabilidade do projeto, é necessário que operação da *startup* seja consideravelmente automatizada. Ou seja, o *software* da empresa deve ser capaz de indexar as vagas a partir de *websites* da internet, armazená-las e priorizá-las de acordo com o perfil de cada usuário.

Atualmente, a empresa utiliza na extração de dados de empregos uma abordagem manual, onde são desenvolvidos *wrappers* específicos para cada *website*. Embora funcional, esta abordagem não atende plenamente as necessidades de escalabilidade do projeto. Há a necessidade de escrita de um novo *wrapper* para cada novo *website* que é adicionado à plataforma, assim como é preciso reescrever esse *wrapper* no momento em que o *template* do *website* passar por mudanças. Mesmo as técnicas de indução de *wrappers* não se encaixam perfeitamente às necessidades do projeto, devido à dependência da intervenção humana no momento de rotular as páginas.

Em vista das dificuldades citadas acima, o tema do trabalho está voltado à abordagem da extração automática. Ou seja, a pesquisa busca o desenvolvimento de um protótipo que permita a extração não supervisionada de dados estruturados a partir de páginas *web* contendo vagas de emprego. É importante citar que a pesquisa se foca especificamente na etapa de extração de dados textuais a partir de páginas de listagem de vagas de emprego. Etapas complementares, como a obtenção das páginas HTML através de *crawlers* e a rotulação dos dados extraídos, possuem certas particularidades que podem ser estudadas em trabalhos futuros.

A pesquisa aborda três eixos principais: embasamento teórico, proposta de solução e avaliação do protótipo e resultados. O embasamento se dá através da pesquisa bibliográfica, que busca identificar técnicas relevantes para o problema abordado. A proposta de solução busca utilizar as técnicas estudadas, através de procedimento experimental, para a construção de um protótipo para a geração de *wrappers* e extração de dados a partir de páginas HTML. A avaliação, por sua vez, verifica a efetividade do protótipo em relação ao objetivo da pesquisa, descrevendo as qualidades e dificuldades encontradas durante os testes práticos.

Os resultados obtidos durante os testes com o protótipo demonstram sua capacidade para identificação da lista de vagas em boa parte das páginas avaliadas. Porém os resultados evidenciam também a necessidade do aperfeiçoamento do método de filtragem dos dados extraídos, pois muitos dados irrelevantes acabam sendo obtidos junto das informações das vagas. De qualquer modo, o resultado permite compreender as dificuldades relacionadas ao processo de extração automática e sugerir possíveis soluções a serem testadas em novas pesquisas.

Este trabalho contribui em dois aspectos. O primeiro deles é devido ao fato de que o estudo de técnicas de extração automática ainda é consideravelmente relevante. Existe um grande potencial para o uso de dados estruturados presentes implicitamente em páginas *web*, e a pesquisa realizada ajuda a entender na prática certas abordagens e dificuldades que são encontradas dentro dessa área de estudo.

A segunda contribuição refere-se à missão da *startup* Jober de facilitar a busca de empregos através da indexação de vagas de emprego de múltiplos *websites*. Um possível aperfeiçoamento do método de extração de vagas irá, invariavelmente, auxiliar a *startup* a se aproximar do seu objetivo e assim aproximar candidatos de possíveis oportunidades no mercado de trabalho.

Este documento descreve a pesquisa realizada e o desenvolvimento do protótipo mencionado anteriormente, e está segmentada da seguinte maneira: o Capítulo 2 relata conceitos e técnicas relevantes ao objetivo do trabalho. O Capítulo 3 descreve a proposta de solução formulada a partir do embasamento teórico. A avaliação do protótipo e seus resultados são apresentados e discutidos no Capítulo 4. Ao final, o Capítulo 5 destaca as principais conclusões da pesquisa.

2 EMBASAMENTO TEÓRICO

Durante a pesquisa, não foram encontrados trabalhos sobre extração de dados estruturados que operem exclusivamente dentro do contexto de páginas contendo vagas de emprego. Os principais estudos sobre extração automática sempre tiveram como foco o uso de técnicas generalistas, ou seja, que independem do domínio dos dados a serem extraídos. Ainda assim, técnicas presentes em trabalhos como Jindal e Liu (2010) e Kayed e Chang (2010) demonstram índices de acerto promissores nos seus resultados de extração de *websites* de diferentes domínios.

Técnicas de extração automática buscam identificar e extrair automaticamente registros de dados inseridos dentro de *templates* em páginas *web*. Essas técnicas são viáveis pois, conforme a explicação de Liu (2011, p. 382, tradução nossa):

> Extração automática é possível porque os registros de dados (instâncias de tuplas) em um *website* são geralmente codificados através de um número muito pequeno de *templates* fixos. É possível encontrar esses *templates* através da mineração de padrões repetidos em múltiplos registros de dados.

O termo ***template*** é utilizado frequentemente por trabalhos situados no contexto de extração de dados a partir de páginas *web*. Na prática, ele se refere à estrutura HTML na qual os dados estruturados são inseridos para compor a página *web*. Em *websites* dinâmicos, esses *templates* são compartilhados por múltiplas páginas que precisam representar diferentes registros de dados. Esses registros de dados, embora possuam valores distintos, serão pertencentes à um mesmo *esquema*.

Em termos simples, o **esquema** de uma página se refere ao tipos de dados presentes nela, assim como a relação entre esses tipos. Por exemplo, um tipo de dado *empresa* pode conter dados de tipos simples, como a *string* representando o nome da empresa, assim como por tipos mais complexos, como um tipo aninhado *localização*, que seria por sua vez composto por outros tipos. Os sistemas apresentados a seguir, em geral, buscam definir modelos de dados que representam uma suposição de como o esquema de uma página pode estar representado.

Boa parte das técnicas de extração de dados também trabalham com o conceito de **árvore DOM** (*Document Object Model*). Devido à estrutura aninhada da linguagem HTML, onde *tags* podem conter outras *tags* dentro de si, é possível representar uma página *web* através de uma árvore lógica. Conhecidas como árvores DOM, essas estruturas utilizam a *tag* raiz da página como o nodo raiz da árvore, e as *tags* filhas são representadas como nodos filhos do novo raiz. Essa representação continua de maneira recursiva até os nodos-folha da árvore, que serão as *tags* que não possuírem outras *tags* aninhadas. Esse

tipo de representação é utilizado por diversos tipos de *software*. Um exemplo disso são *scrapers*: programas que acessam páginas específicas e navegam pela árvore DOM em busca de dados relevantes.

As pesquisas da área estudada também compartilham métodos de avaliação em comum. Dentre eles, é possível citar as medidas de precisão e revocação. Conforme explica Manning e Schütze (1999), muitos sistemas de Recuperação da Informação são avaliados com base na ideia de que devem extrair itens relevantes a partir de um grupo de itens não classificados. De acordo com a implementação do sistema, cada item extraído pertencerá à um dos seguintes grupos: Verdadeiro Positivo (VP), Verdadeiro Negativo (VN), Falso Positivo (FP) e Falso Negativo (FN).

Deste modo, a **precisão** representa o percentual de itens identificados corretamente em relação ao total de itens identificados como sendo corretos pelo sistema, ou seja:

$$P = \frac{VP}{VP + FP} \tag{2.1}$$

Por sua vez, a **revocação** mede o percentual entre a quantidade de itens identificados corretamente em relação à quantidade de itens que deveriam ter sido identificados corretamente:

$$R = \frac{VP}{VP + FN} \tag{2.2}$$

Outra medida bastante utilizado é o ***F-score***. Essa métrica se baseia no valor de precisão (P) e revocação (R), além de um valor α utilizado para a ponderação entre P e R (geralmente, $\alpha = 0{,}5$):

$$F = \frac{1}{\alpha\frac{1}{P} + (1 - \alpha)\frac{1}{R}} \tag{2.3}$$

As próximas seções apresentam pesquisas e técnicas relevantes avaliadas durante a fundamentação teórica do trabalho. Esses trabalhos servirão como base para o desenvolvimento do protótipo da pesquisa.

2.1 *A GENERALIZED TREE MATCHING ALGORITHM CONSIDERING NESTED LISTS*

2.1.1 Introdução

O trabalho de Jindal e Liu (2010) aborda um desafio importante dentro do estudo de técnicas de extração automática: a identificação e extração de listas aninhadas a partir

de páginas HTML. Um exemplo de listas aninhadas é exibido na Figura 1, onde cada filme da lista possui uma lista de atores, além de seus próprios atributos. Na época, nenhuma das abordagens presentes era capaz de lidar corretamente com este tipo de cenário. Através do uso de técnicas de *tree matching* baseadas na pesquisa de Yang (1991), os autores obtiveram bons resultados ao extrair dados estruturados, tanto para os casos onde apenas uma página é utilizada como entrada para o algoritmo, quanto para os que utilizam múltiplas páginas. Algoritmos de *tree matching* buscam verificar a correspondência entre duas estruturas em árvores distintas. A utilização dessas técnicas é natural nesse contexto, pois arquivos HTML possuem *tags* aninhadas que formam uma estrutura em árvore.

Figura 1: Exemplo de listas aninhadas

Fonte: Lista de filmes do *website* www.imdb.com

Listas são extremamente frequentes em páginas *web*. Exemplos comuns são listagens de itens como produtos, postagens, entre outros. Devido à isso, a correta identificação dessas estruturas é essencial para a extração eficaz de dados estruturados. Porém, esse desafio não é tão simples, pois itens de listas não precisam ser exatamente iguais entre si. A presença de valores opcionais pode fazer com que dois itens de um mesmo tipo sejam representados com estruturas HTML bastante diferentes, dificultando a identificação da lista. Um exemplo disso seria uma listagem de produtos, onde alguns possuem valores promocionais e outros não. Além disso, esses itens podem conter outras listas aninhadas (conforme Figura 1), o que aumenta ainda mais a complexidade da tarefa.

A pesquisa de Jindal e Liu (2010) busca realizar duas contribuições:

1. Generalizar um algoritmo de tree matching para que ele possa trabalhar correta-

mente com listas. O algoritmo simple tree matching é estendido com a habilidade de identificar listas, gerando o algoritmo generalized tree matching;

2. Até o momento, era comum a utilização de diferentes algoritmos dependendo da quantidade de páginas utilizadas como entrada para o algoritmo. Certos algoritmos extraem dados a partir de uma única página através da identificação de padrões repetidos dentro delas. Outras técnicas focam na extração a partir de duas ou mais páginas, utilizando a diferença entre elas para criar um modelo que possa extrair dados daquele *template*. O método proposto inova ao permitir sua utilização em ambos cenários.

2.1.2 Formulação do cenário e problema

Os autores definem dados estruturados como “relações aninhadas”, ou seja, objetos tipados que podem conter conjuntos e tuplas aninhadas. Os seguintes tipos são definidos:

- Existe uma lista de *tipos básicos*, $B = \{B_1, B_2, ..., B_k\}$. Cada B_i é um tipo atômico e seu domínio $dom(B_i)$ é um conjunto de constantes;
- Se $T_1, T_2, ..., T_n$ são tipos básicos ou conjuntos, então $[T_1, T_2, ..., T_n]$ é um *tipo tupla* com o domínio $dom([T_1, T_2, ..., T_n]) = \{[v_1, v_2, ..., v_n] \mid v_i \in dom(T_i)\}$;
- Se T é um tipo tupla, então $\{T\}$ é um *tipo conjunto* com o domínio $dom(\{T\})$ sendo o domínio $dom(T)$;

Jindal e Liu (2010, pág. 932, tradução nossa) descreve: “Um tipo básico B_i é análogo ao tipo de um atributo em um banco de dados relacional, por exemplo, *string* ou *int*. No contexto da *web*, B_i é uma *string* de caracteres, um arquivo de imagem, etc.”. Uma instância de um tipo tupla é chamada de *registro de dados*, enquanto uma instância de um tipo conjunto é chamado de lista.

Dada essas definições, a pesquisa define duas formulações para o problema da extração de dados em páginas HTML:

Extração baseada em múltiplas páginas. Recebe como entrada uma coleção W de k *strings* HTML, que codificam k instâncias do mesmo tipo. A saída deve ser um tipo σ, e uma coleção C de instâncias do tipo σ, fazendo com que exista uma função *enc* capaz de gerar W a partir de C.

Extração baseada em uma única página de listagem. A entrada é uma única *string* S, que contêm k *substrings* de caracteres não sobrepostas $s_1, s_2, \ldots, s_k$ com cada s_i codificando uma instância de um tipo conjunto W_i. A saída são k tipos tupla $\sigma_1, \sigma_2, ..., \sigma_k$ e k coleções $C_1, C_2, ..., C_k$ de instâncias dos tipos tupla, sendo que para cada coleção C_i existe uma codificação HTML enc_i que habilite enc_i: $C_i \mapsto W_i$.

2.1.3 Proposta de solução

Para solucionar os problemas citados anteriormente, Jindal e Liu (2010) propõe uma nova versão do algoritmo STM (*Simple Tree Matching*), generalizada para identificar listas corretamente.

O algoritmo STM busca realizar o mapeamento entre duas árvores distintas. Considerando-se X uma árvore, *x[i]* se configura como o nodo de índice i pertencente à X, após a navegação através de todos os níveis da árvore. Um *mapeamento M* entre as árvores A e B é um conjunto de pares ordenados *(i, j)*, onde cada um dos nodos pertence à uma árvore. Todos os pares $(i_1, j_1), (i_2, j_2) \in M$ devem atender às seguintes condições:

1. $i_1 = i_2$ se e apenas se $j_1 = j_2$;
2. $A[i_1]$ está na esquerda de $A[i_2]$ se e apenas se $B[j_1]$ estiver na esquerda de $B[j_2]$;
3. $A[i_1]$ é um ancestral de $A[i_2]$ se e apenas se $B[j_1]$ for um ancestral de $B[j_2]$;

De maneira simples, é possível definir o algoritmo STM como um mapeamento entre nodos equivalentes dentro de duas árvores. O objetivo do algoritmo é encontrar o mapeamento que gere o maior número de pares entre as duas árvores. Uma descrição mais detalhada do algoritmo pode ser encontrada em Yang (1991). Dentro do contexto da *web*, as árvores a serem mapeadas são geradas através do DOM das páginas HTML, onde cada nodo representa uma instância de uma *tag* HTML.

O algoritmo proposto pelos autores se trata do G-STM (*Generalized Tree Matching*). Ele detecta listas através da geração de tuplas no formato *(score, nodesA, nodesB)*, onde *score* representa a pontuação normalizada da correspondência entre as árvores A e B, e os valores *nodesA* e *nodesB* representam a quantidade de nodos nas árvores A e B, respectivamente. Esses valores são necessários para a detecção de listas.

A função principal do algoritmo G-STM recebe como parâmetro as árvores A e B que devem ser comparadas ou, mais especificamente, os dois nodos raiz de ambas as árvores. Caso esses nodos sejam distintos (ou seja, suas *tags* HTML sejam diferentes), a função retorna a pontuação 0. Caso contrário, é criada uma matriz W com a pontuação de correspondência de cada uma das subárvores de A com cada uma das subárvores de B. Essa pontuação é calculada através da chamada recursiva da própria função com os nodos filhos de A e B. Após essa etapa, a função *Detect-Lists* é utilizada para detectar listas através da geração de uma gramática regular para as árvores, conforme é explicado a seguir. Ao final, o algoritmo utiliza programação dinâmica para obter a pontuação final da correspondência de A e B. O pseudocódigo do algoritmo é apresentado na Figura 2.

A função *Detect-Lists* tem a responsabilidade de identificar listas a partir da matriz de pontuação W gerada anteriormente. Logicamente, espera-se que uma lista possua certo

Figura 2: O algoritmo *G-STM*

```
Algoritmo: G-STM(A, B)
se as raízes das árvores A e B contêm símbolos diferentes então
        retorne (0, A.nodos, B.nodos)
senão
        k ← o número de subárvores no primeiro nível de A
        n ← o número de subárvores no primeiro nível de B
        Inicialização: m[i, 0] ← 0 para i = 0, …, k;
                       m[0, j] ← 0 para j = 0, …, n;
para i = 1 para k faça
        para j = 1 para n faça
                W[i, j] ← G-STM(A_i, B_j)
(W, nodos_A, nodos_B) ← Detect-Lists(W, A, B);
para i = 1 para k faça
        para j = 1 para n faça
                m[i, j] ← MAX(m[i, j - 1], m[i - 1, j],
                              m[i - 1, j - 1] + W[i, j].pontuação)
retorne (m[k, n] + 1, nodos_A, nodos_B)
```

Fonte: Jindal e Liu (2010, pág. 933, tradução nossa)

grau de semelhança entre seus itens. Os autores exploram essa ideia, utilizando os valores de correspondência da matriz W como base para identificar quais itens pertencem ou não à listas. Os nodos que corresponderem recebem símbolos iguais, e os que não corresponderem à nenhum outro recebem símbolos distintos. A decisão sobre a correspondência ou não de dois nodos é realizada através de *thresholds* (limiares) definidos empiricamente pelos autores. Ao final dessa etapa, são geradas duas strings, uma para a árvore A e outra para a B. Ambas as strings são passadas para a função *Grammar-Generation*. Se a gramática resultante for igual, isso significa que uma lista foi encontrada. Neste caso, a função *UpdateW* é chamada para atualizar os valores da matriz de pontuação.

Através da *string* obtida durante a função *Detect-Lists*, é possível gerar uma expressão regular que descreve essa mesma *string*. Dentro do algoritmo G-STM, essa etapa é realizada pela função *Grammar-Generation*. A expressão regular é obtida através da construção de um autômato finito não determinístico, que é expandido conforme os símbolos da *string* são lidos. Os símbolos que representam uma lista são equivalentes à uma expressão regular no formato *(D)+*, que representa uma ou mais ocorrências de *D*.

Ao final da função *Detect-Lists* é realizada a chamada para *UpdateW*. Essa função atualiza a matriz de pontuação de acordo com as listas identificadas. Tanto para a árvore A quanto para B, o primeiro nodo de cada lista recebe a nova pontuação baseada na média de sua correspondência com os nodos repetidos na outra árvore. O resto dos nodos das listas recebe a pontuação 0. Essa etapa garante que a pontuação do cálculo dos nodos superiores não seja afetada pela pontuação das listas identificadas neste nível da árvore.

De acordo com os autores, o algoritmo G-STM permite apenas calcular a correspondência entre duas árvores. Nos casos em que for necessário utilizar múltiplas páginas para a construção do *wrapper*, é necessária a aplicação do método de alinhamento parcial de árvores PTA (*Partial Tree Alignment*). Conforme explica Liu (2011), o PTA funciona através da expansão contínua de uma árvore semente, conforme ela é comparada com as árvores que se deseja alinhar. Em conjunto do algoritmo G-STM, o PTA permitirá a criação de uma única árvore *template* que poderá extrair dados a partir de páginas utilizando a mesma estrutura. Para realizar a extração de dados a partir de múltiplas páginas, é necessário construir uma árvore DOM para cada uma das páginas, utilizando-as como entrada para o algoritmo G-STM. Nos casos onde a extração é realizada apenas com uma página de entrada, é possível utilizar o mesmo método, porém a árvore da página deverá ser duplicada para que possa servir de entrada para o algoritmo.

2.1.4 Avaliação da proposta

Os autores avaliam a técnica em duas etapas: uma para os casos em que múltiplas páginas são usadas para a extração, e outra para quando apenas uma página é utilizada. Como ambos os problemas eram tratados de maneira distinta na época da publicação da pesquisa, o desempenho do algoritmo foi comparado com diferentes técnicas, cada uma focada em apenas um dos problemas. Foram utilizados três *datasets* para a avaliação, sendo que dois deles são públicos: Yamada et al. (2004) e Zhao et al. (2005). O último foi coletado pelos próprios autores. Todas as páginas utilizadas são páginas de listagem e, com exceção de algumas do último *dataset* citado, todas foram obtidas através de formulários de pesquisa em *websites*.

Com base na extração a partir de múltiplas páginas, o algoritmo G-STM foi comparado com os algoritmos RoadRunner (CRESCENZI et al., 2001) e EXALG (ARASU; GARCIA-MOLINA, 2003).

Nos primeiros testes da comparação com o RoadRunner, o algoritmo G-STM recebia 2 páginas aleatórias de cada *website* para gerar o *template*, enquanto o resto das páginas eram utilizadas para extração de dados. O algoritmo RoadRunner, por sua vez, recebia todas as páginas para extração. Os resultados demonstram uma clara vantagem do algoritmo G-STM, conforme é exibido na Tabela 1. É importante citar que o RoadRunner não funcionou em todos os *websites*, e por isso alguns de seus resultados não foram incluídos no cálculo. Isso ocorreu porque as páginas utilizadas tinham o mesmo número de itens, o que impede o RoadRunner de identificar listas.

Os testes com o algoritmo EXALG não puderam ser realizados em todas as páginas dos *datasets*, pois ele não se encontra disponível publicamente. Para a comparação, foram utilizados apenas as páginas para as quais os resultados do EXALG foram publicados. Os resultados da comparação neste *dataset* específico são apresentadas na Tabela 2.

Tabela 1: Resultados da comparação entre G-STM e RoadRunner

Sistema	**G-STM**	**RoadRunner**
Precisão	93.9%	91.5%
Revocação	95.4%	70%
F-score	0.94	0.79

Fonte: Jindal e Liu (2010)

Tabela 2: Resultados da comparação entre G-STM e EXALG

Sistema	**G-STM**	**EXALG**
Precisão	100%	100%
Revocação	100%	90%
F-score	1.0	0.94

Fonte: Jindal e Liu (2010)

Para os testes de extração baseados em uma única página, o G-STM foi comparado com o sistema DEPTA (ZHAI; LIU, 2005). Em todos os *datasets*, o G-STM demonstrou-se superior, principalmente no dataset gerado pelos autores. Jindal e Liu (2010) citam três principais fontes de erros do algoritmo G-STM nos primeiros testes: itens sendo incluídos em listas erroneamente, problemas no alinhamento de nodos devido à *tags* irregulares, e alguns nodos de listas não sendo identificados devido à grande diferença em relação aos outros itens da lista. Para este último caso, os autores acreditam que a utilização de informação visual (a disposição dos nodos após da renderização do navegador) possa melhorar a taxa de acerto.

Como última avaliação, o G-STM foi comparado com o sistema MSE (ZHAO; MENG; YU, 2006) em relação à sua capacidade de segmentar registros de dados. Esse teste busca avaliar o quão bem os algoritmos são capazes de identificar os limites da localização dos registros nas páginas. Nos *datasets* públicos, o G-STM obteve a precisão e revocação de 100% e 96.5%, enquanto o MSE obteve 100% e 98%. Porém, no *dataset* gerado pelos autores, o MSE não foi capaz de identificar os registros de dados em 9 dos 22 *websites* testados. Nos 13 *websites* restantes, o MSE alcançou precisão e revocação de 97.3% e 65.7%, enquanto o algoritmo G-STM obteve 100% e 95%.

2.1.5 Conclusões

Jindal e Liu (2010) concluem sua pesquisa retomando as duas contribuições do algoritmo desenvolvido. A primeira contribuição é a capacidade que o G-STM possui de identificar listas aninhadas, algo inédito na época. A segunda é o fato que o algoritmo permite a extração de dados nos dois problemas formulados: quando utiliza-se uma única página como entrada e quando utiliza-se múltiplas páginas. Em ambas as situações, o algoritmo demonstrou resultados superiores aos sistemas já utilizados no estado da arte

na época da publicação. Os autores ainda citam que o sistema desenvolvido foi testado em um contexto comercial e, na época, estava no processo de licenciamento para a iniciativa privada.

2.2 *FIVATECH: PAGE-LEVEL WEB DATA EXTRACTION FROM TEMPLATE PAGES*

2.2.1 Introdução

O trabalho de Kayed e Chang (2010) apresenta uma abordagem para extração de dados automática chamada *FiVaTech*. Muitos *websites* possuem páginas geradas dinamicamente: os valores dos registros de dados são primeiramente obtidos a partir de um esquema, para depois serem inseridos dentro de um *template* específico. Deste modo, todas as páginas geradas através do mesmo *template* possuem similaridades entre si. Os autores exploram essa característica em sua proposta, que tem como objetivo a extração automática do esquema e *template* de um *website* a partir de páginas HTML.

2.2.2 Formulação do cenário e problema

Kayed e Chang (2010) definem as páginas *web* como o resultado da inserção de instâncias de dados dentro de um *template*. Essas instâncias, provenientes de um banco de dados, sempre possuirão um esquema em comum entre elas. Deste modo, os autores definem os diferentes tipos presente em um esquema:

1. Um tipo básico (β) representa uma *string* de *tokens*, onde cada *token* é constituído por algumas unidades básicas de texto, como, por exemplo, uma frase simples;
2. Se $\Phi_1, \Phi_2, \ldots, \Phi_n$ são tipos, então a sua lista ordenada $\Phi = <\Phi_1, \Phi_2, \ldots, \Phi_n>$ também é um tipo. É possível dizer que o tipo Φ é construído a partir dos tipos $\Phi_1, \Phi_2, \ldots, \Phi_n$ usando um tipo construtor de ordem *n*. Uma instância do tipo-*n* encontra-se na forma $<x_1, x_2, \ldots, x_n>$ onde $x_1, x_2, \ldots, x_n$ são instâncias dos tipos $\Phi_1, \Phi_2, \ldots, \Phi_n$, respectivamente. Um tipo Φ é chamado de:
    a) tupla, indicado por $<\Phi>$, se sua cardinalidade é igual a 1 para cada instanciação.
    b) opcional, indicado por $(\Phi)?$, se sua cardinalidade é 0 ou 1 para cada instanciação.
    c) conjunto, indicado por Φ, se sua cardinalidade é maior que 1 para alguma instanciação.

Nesta pesquisa, os autores optaram por trabalhar a página como uma estrutura em árvore, pois tanto o esquema quanto a página HTML em si são estruturas em árvore.

Isso contrasta com o EXALG (ARASU; GARCIA-MOLINA, 2003), que interpreta a página como uma *string* de símbolos. Considerando a definição de *template* utilizada pelos autores, é possível perceber que os valores dos tipos básicos que compõe as instâncias de dados sempre irão se encontrar nos nodos folha das árvores geradas. Assim, o foco da abordagem é a identificação de nodos folha distintos, que por sua vez permitirão a identificação dos construtores de tuplas (ou seja, o esquema por trás dessas tuplas).

Dada essas definições, o problema que a técnica proposta busca resolver é proposto do seguinte modo por Kayed e Chang (2010, pág. 16, tradução nossa):

> Dado um conjunto de n árvores DOM, $DOM_i = (T, x_i)$ $(1 \leq i \leq n)$, geradas de um *template* desconhecido T e valores $x_1, \ldots, x_n$, deduza o *template* e valores a partir do conjunto de árvores apenas. Chamamos este problema de extração de informações no nível da página. Se apenas uma página ($n = 1$) que contêm construtores de tupla é dada como entrada, o problema é deduzir o *template* para o esquema contido dentro dos construtores de tupla. Chamamos este problema de tarefa de extração de informações no nível de registro.

2.2.3 Proposta de solução

A técnica proposta pelos autores consiste, resumidamente, na geração de uma árvore padronizada a partir da combinação das árvores DOM das páginas de treinamento. Essa árvore se chama *fixed/variant pattern tree* (árvore de padrão fixo/variável), e permite que dados em comum entre as páginas sejam identificados. Deste modo, busca-se identificar o *template* e esquema utilizado na geração das páginas.

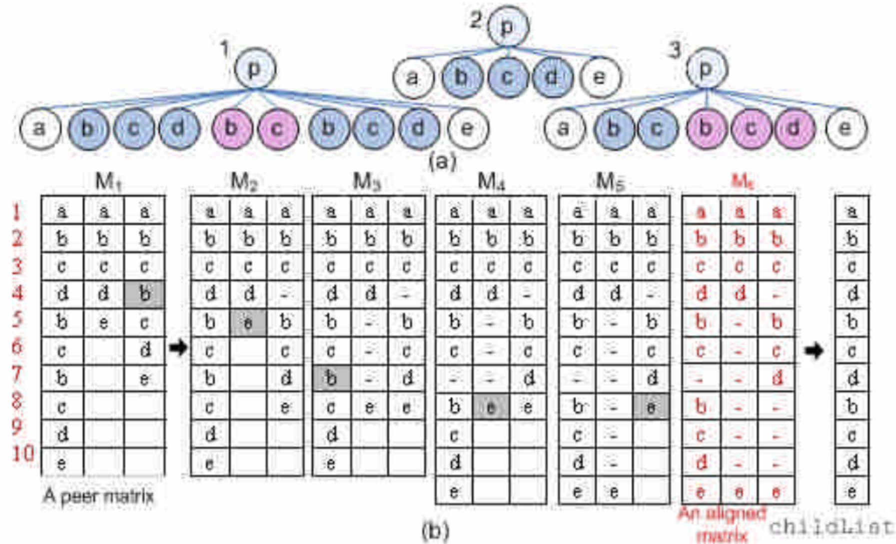
O processo de geração da *pattern tree* se inicia com os nodos raiz das árvores de treinamento (no caso, os nodos da *tag <html>*) e realiza uma busca em profundidade a partir dos nodos filhos, até chegar nas folhas. Para cada nodo interno, o algoritmo utiliza os nodos filhos do primeiro nível para montar uma matriz de pares. A partir dessa matriz, o algoritmo realiza as seguintes operações: **reconhecimento de nodos pares**, **alinhamento da matriz de pares**, **mineração de padrões** e **combinação de nodos opcionais**.

Durante a etapa de **reconhecimento de nodos pares**, o algoritmo *simple tree matching* proposto por Yang (1991) é aplicado aos nodos para que seja identificada as suas similaridades. Nodos cuja correspondência for maior que uma limiar δ específico são combinados em um mesmo símbolo na matriz de pares.

Após essas alterações, o **alinhamento da matriz de pares** busca alinhar essa matriz para transformá-la em uma *matriz de pares alinhada*. Resumidamente, essa etapa é similar à um alinhamento de *strings*: o algoritmo itera linha por linha da matriz, até o momento em que todas as linhas não vazias contenham ou o mesmo símbolo, ou nodos filhos que representam tipos básicos. Ao final, será gerada uma lista de símbolos com todas

as linhas alinhadas. A Figura 3 apresenta a progressão dessa etapa enquanto o algoritmo realiza o alinhamento de uma matriz com três árvores.

Figura 3: O alinhamento de uma matriz de pares com três árvores



Fonte: Kayed e Chang (2010, pág. 18)

A etapa de **mineração de padrões** recebe como entrada a lista de símbolos gerada na etapa anterior e busca identificar possíveis repetições na ocorrência de símbolos. Quando uma repetição é identificada, apenas uma ocorrência é mantida na lista, sendo marcada como um tipo conjunto. Esta etapa resulta em uma lista de símbolos sem padrões repetidos.

Por último, a etapa de **combinação de nodos opcionais** busca identificar nodos que não aparecem obrigatoriamente em todas as páginas. Para realizar essa verificação, o algoritmo verifica a linha da matriz de pares correspondente à cada nodo para extrair o seu vetor de ocorrência. Caso ele não conste em todas as páginas (ou em todas as repetições de um tipo tupla), o nodo será marcado como opcional. Esta etapa também agrupa nodos adjacentes que possuam vetores de ocorrência iguais.

A partir da *pattern tree*, é possível obter o esquema das páginas com a simples remoção de nodos com filho único e preservando todos os nodos filhos de tipo básico. Para identificar o *template* dos tipos identificados, os autores utilizam um algoritmo que cruza a árvore gerada, aplicando métodos específicos para filtrar os nodos que não pertencem ao *template*.

2.2.4 Avaliação da proposta

Kayed e Chang (2010) realizaram dois experimentos com a proposta desenvolvida: o primeiro compara o FiVaTech com o EXALG (ARASU; GARCIA-MOLINA, 2003), enquanto o segundo avalia a capacidade de extração de registros de resultado de busca em

relação aos sistemas DEPTA, ViPER (SIMON; LAUSEN, 2005) e MSE (ZHAO; MENG; YU, 2006).

No primeiro experimento, foi avaliada a capacidade a capacidade de extração do esquema dos sistemas EXALG e FiVaTech. Essa avaliação é realizada através da comparação entre os esquemas extraídos de diferentes *websites* e os esquemas manualmente construídos pelos pesquisadores. Enquanto o EXALG obteve valores de precisão e revocação de respectivos 90,6% e 75,8%, o FiVaTech atingiu os percentuais de 90,6% e 95,1%.

O segundo experimento busca avaliar a capacidade do FiVaTech de extrair registros de resultados de busca. Normalmente, essa atividade consiste em duas etapas: a identificação dos limites dos registros de dados (ou seja, onde eles se localizam na página) e o alinhamento desses registros de dados. Os valores das métricas analisadas na comparação com o sistema DEPTA são exibidos na Tabela 3.

Tabela 3: Resultados da comparação entre DEPTA e FiVaTech

Etapas	**Identificação de registros**		**Alinhamento de registros**	
Sistemas	**DEPTA**	**FiVaTech**	**DEPTA**	**FiVaTech**
Precisão	91,1%	98,0%	48,4%	90,1%
Revocação	53,9%	95,7%	48,9%	89,1%

Fonte: Kayed e Chang (2010)

Como os sistemas ViPER e MSE focam apenas na identificação de limites dos registros, a comparação com esses sistemas não considerou o alinhamento dos registros. Os resultados obtidos nesta comparação foram consideravelmente similares.

2.2.5 Conclusões

Em sua conclusão, os autores reafirmam o objetivo da pesquisa: a proposta de uma nova abordagem para a extração de dados na *web*. Através da utilização de um novo algoritmo para alinhamento de *strings* de símbolos, o FiVaTech permite a identificação do esquema e *template* de páginas HTML. Foram obtidos bons resultados utilizando-se 2 a 3 páginas como entrada para o algoritmo, e, de acordo com os autores, a avaliação se mais páginas podem melhorar esse resultado necessita de um estudo mais aprofundado.

2.3 *AUTORM: AN EFFECTIVE APPROACH FOR AUTOMATIC WEB DATA RECORD MINING*

2.3.1 Introdução

A pesquisa de Shi et al. (2015) apresenta uma nova abordagem para a extração automática de dados, chamada **AutoRM** (*Automatic Data Record Mining*). Assim como as abordagens citadas anteriormente, o AutoRM se baseia na ideia de que páginas *web*

consistem de registros de dados codificados em objetos semiestruturados, compostos por *tags* HTML. Registros de dados se encontram dentro de **regiões de dados**: segmentos das páginas que apresentam listas de registros de dados similares. Os registros de dados em si são compostos por nodos irmãos, contendo as informações no formato de texto, imagens, entre outros. De acordo com pesquisa realizada pelos autores, essa suposição é verdadeira em 99,8% dos casos.

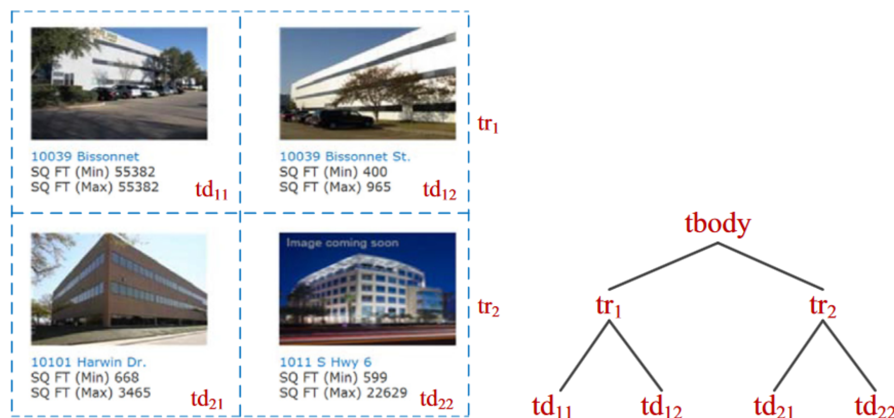
O AutoRM busca minerar registros de dados de todas as regiões de dados de uma página, de maneira automática. Para isso, utiliza suposições específicas de como os registros de dados são codificados na árvore DOM da página, além de utilizar informações visuais relacionadas ao tamanho dos elementos depois que são renderizados pelo navegador *web*. A abordagem realiza a extração de dados a partir de três etapas:

1. Construir a árvore da página utilizando os recursos disponibilizados pelo navegador *web*;

2. Identificar todos os conjuntos de *C-Records* (*Candidate data records*, ou seja, possíveis registros de dados) similares e adjacentes, a partir da árvore da página. Esses *C-Records* consistem de registros de dados ou objetos que contêm registros de dados. A Figura 4 demonstra um segmento de página *web* com quatro registros de dados (td_{11}, td_{12}, td_{21} e td_{22}), sendo que esses registros estão separados por dois nodos-pai (tr_1 e tr_2);

3. Extrair registros de dados reais a partir dos conjuntos de *C-Records* identificados.

Figura 4: Segmento de página contendo registros de dados



Fonte: Shi et al. (2015)

2.3.2 Formulação do cenário e problema

A segunda etapa de operação do AutoRM, conforme descrito anteriormente, é a identificação de *C-Records*. Shi et al. (2015) propõe uma subdivisão desta etapa em duas

partes: a identificação de uma *CG-Region* (*Coarse-Grained Data Region*) e a segmentação da *CG-Region* em *C-Records* similares.

De acordo com Shi et al. (2015, pág. 2, tradução nossa), uma *CG-Region* consiste de "[...] uma seção da página *web* contendo um ou mais conjuntos de *C-Records* similares adjacentes e alguma informação irrelevante.". De acordo com os autores, a identificação de *CG-Regions* permite que muito das informações irrelevantes da página (como nodos HTML sem conteúdo) sejam removidas da etapa de segmentação, que possui uma carga de processamento maior.

A segunda parte trata da segmentação da *CG-Region* em *C-Records*. Para isso, a solução proposta precisa tratar dos seguintes problemas:

Problema 1. Como determinar o nodo inicial do primeiro *C-Record* dentro de uma *CG-Region*?

Problema 2. Como identificar os limites entre os *C-Records*, assim como o último nodo pertencente ao último *C-Record* dentro de uma *CG-Region*?

Dentro do Problema 2, os autores definem duas formas de organização de *C-Records*. Na **Forma 1**, os *C-Records* são segmentados por separadores semelhantes que não carregam nenhum tipo de conteúdo (ou seja, não possuem textos, imagens, ...). Na **Forma 2**, que é exemplificada pelos nodos tr_1 e tr_2 na Figura 4, os *C-Records* não possuem nenhuma divisão entre si. Esta definição de formas é pertinente ao trabalho, pois o AutoRM define diferentes algoritmos para extrair *C-Records* de cada uma delas.

2.3.3 Proposta de solução

A abordagem utilizada pelo AutoRM se inicia pela extração de *C-Records* presentes na árvore DOM da página *web*, que serve de entrada para o algoritmo.

A partir do nodo raiz da árvore, é realizado o agrupamento dos nodos filhos de primeiro nível (ou seja, os nodos que se encontram diretamente abaixo do nodo raiz) com base na técnica de agrupamento aglomerativo (KAUFMAN; ROUSSEEUW, 1990). Após isso, o AutoRM busca identificar *CG-Regions* presentes dentro desses agrupamentos e, para cada *CG-Region*, extrair seus *C-Records*.

Após essa etapa, podem existir nodos filhos que não compõe nenhum *C-Record* identificado. Se este for o caso, pode ser que alguns dos filhos desses nodos, por sua vez, possuam *C-Records*. Devido à isso, o algoritmo é chamado de maneira recursiva para cada um dos agrupamentos que restaram.

O algoritmo de agrupamento aglomerativo de Kaufman e Rousseeuw (1990) é utilizado pelos autores para realizar o agrupamento entre diferentes árvores, o que é um processo fundamental dentro da área de extração automática. Porém, ao contrário do G-

STM (JINDAL; LIU, 2010), que utiliza um cálculo de similaridade simplificado baseado em correspondência entre árvores, Shi et al. (2015) utilizam o método de Lu et al. (2013), que considera diversos fatores para o cálculo de similaridade entre nodos, como atributos visuais, tipos de dados, caminho no DOM da página, entre outros.

A identificação de *CG-Regions* proposta pelos autores estabelece que uma *CG-Region* deve obedecer as seguintes propriedades:

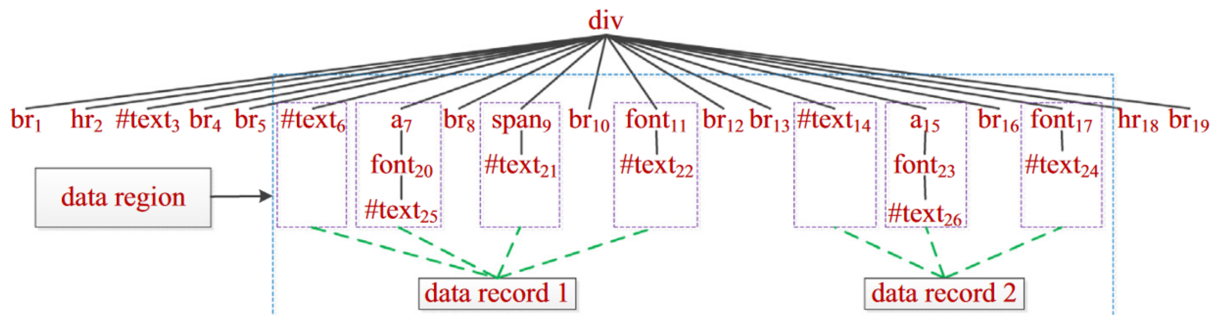
Propriedade 1. Uma região de dados deve conter um ou mais conjuntos de nodos similares.

Propriedade 2. Se uma região de dados contém mais de um conjunto de nodos similares, então o intervalo dos índices de um conjunto deve interceptar o intervalo de outro conjunto.

Os nodos pertencentes à árvore são classificados em dois tipos: nodos separadores e não separadores. Nodos separadores são aqueles cuja sub-árvore não possui textos, imagens ou outro tipo de dado relevante. Caso contrário, o nodo é classificado como não separador.

A partir da utilização dos nodos não separadores e da propriedade 2, o algoritmo para identificação de *CG-Regions* busca mesclar os agrupamentos de nodos até que nenhum dos grupos restantes possuam intervalos conflitantes. Um exemplo da identificação de uma *CG-Region* pode ser realizado com os nodos apresentados na Figura 5. Considerando os nodos filhos apresentados, se obtém os agrupamentos não separadores $\{\#texto_3, \#texto_6, \#texto_{14}\}$, $\{a_7, a_{15}\}$, $\{span_9\}$ e $\{font_{11}, font_{17}\}$, com os respectivos intervalos de [3, 14], [7, 15], [9, 9], [11,17]. Ao final da execução, a lista de *CG-Regions* do exemplo contém apenas uma região identificada, com o conjunto de nodos $\{\#texto_3, br_4, br_5, \#texto_6, a_7, br_8, span_9, br_{10}, font_{11}, br_{12}, br_{13}, \#texto_{14}, a_{15}, br_{16}, font_{17}\}$.

Figura 5: Exemplo de árvore DOM contendo registros de dados



Fonte: Shi et al. (2015)

Após a identificação das *CG-Regions*, é necessário realizar a sua segmentação em *C-Records*. Para cada nodo não separador C_i presente na *CG-Region* identificada, o algoritmo

gera um conjunto de possíveis *C-Records* que poderiam ser extraídos da *CG-Region*, caso fosse considerado que o nodo C_i é o início do primeiro *C-Record* da região de dados. Essa estratégia busca resolver o Problema 1, citado anteriormente.

Após a iteração por todos os nodos, esta etapa tem como produto uma lista de conjuntos de *C-Records*. Como apenas um dos conjuntos de *C-Records* deve ser o correto, os autores definem duas propriedades que servem como indicativo de qual conjunto será selecionado:

Propriedade 3. Um *C-Record* de alta qualidade cobre uma grande área da página.

Propriedade 4. Um *C-Record* de alta qualidade possui alta similaridade com os outros *C-Records*, enquanto os nodos dentro de cada *C-Record* tendem a possuir uma baixa similaridade entre si.

A partir dessas duas propriedades, o AutoRM seleciona o melhor conjunto de *C-Records* minerados a partir de métricas como a quantidade de *pixels* ocupada pelos nodos e a similaridade entre os *C-Records*.

Para a geração de cada conjunto de *C-Records* a partir de um nodo inicial C_i, o AutoRM recebe a região CR_i compreendida pelos nodos C_i, $C_i + 1$, ..., C_n, onde n é o número total de nodos da *CG-Region* identificada. Neste momento, são utilizados os algoritmos *separator-based* e *head-order-based* para identificar *C-Records* que estejam, respectivamente, na **Forma 1** ou na **Forma 2**. Caso a região CR_i não possua nodos separadores, o algoritmo *head-order-based* é aplicado diretamente. Caso contrário, ambos os algoritmos são aplicados e o melhor resultado é selecionado com base nas propriedades **3** e **4** citadas acima.

Esta última etapa busca atender o **Problema 2**. Ambos os algoritmos exploram certas características identificadas pelo autores para cada uma das formas definidas: o algoritmo *separator-based* identifica *C-Records* através da segmentação da região pelos separadores, enquanto o algoritmo *head-order-based* trata da segmentação de *C-Records* adjacentes (que não possuem separadores entre si).

Após todas as etapas anteriores, a última tarefa do AutoRM é a extração de registros de dados reais a partir dos *C-Records* identificados. Para cada conjunto de *C-Records* extraídos, o algoritmo busca extrair novos *C-Records* aplicando as etapas citadas acima para cada *C-Record*. Isso é necessário pois, de acordo com os autores, os *C-Records* extraídos a partir do nodo raiz da página podem não ser registros de dados, mas sim conter registros de dados dentro de si. A busca por *C-Records* continua de maneira recursiva até que a qualidade dos *C-Records* (de acordo com a **Propriedade 3**) seja inferior à um limiar definido pelos autores através de testes. Ao final, os *C-Records* extraídos serão retornados como registros de dados.

2.3.4 Avaliação da proposta

Shi et al. (2015) avaliaram sua abordagem AutoRM através da comparação com os sistemas CTVS (SU et al., 2012), G-STM (JINDAL; LIU, 2010), ViDE (LIU; MENG; MENG, 2010) e DEPTA (ZHAO; MENG; YU, 2006). Os autores utilizaram dois *datasets* durante os experimentos, o *dataset* público TBDW (YAMADA et al., 2004) e outro gerado pelos próprios autores. Os autores obtiveram acesso à uma versão executável do ViDE, e implementaram sua própria versão do DEPTA, de acordo com o trabalho original. Para os sistemas CTVS e G-STM, a comparação foi realizada apenas com relação ao *dataset* TBDW, para o qual esses sistemas divulgaram seus resultados.

Os autores dividiram as páginas em dois tipos distintos:

1. **One-DR.** Páginas em que existe apenas um conjunto de registro de dados similares, contido em uma única região de dados. De acordo com os autores, todas as abordagens comparadas funcionam neste tipo de página. Um exemplo deste tipo de página é presente na Figura 5.

2. **One-ASO.** Páginas em que existe apenas um conjunto de registro de dados similares, incorporados dentro de objetos que contém regiões de dados menores. Os autores citam que apenas os sistemas AutoRM e o DEPTA são preparados para esse tipo de página. Um exemplo deste tipo de página é presente na Figura 4.

Para cada um dos tipos citados acima, os autores realizam uma subdivisão em dois tipos, um para a **Forma 1** e outro para a **Forma 2**. Deste modo, os sistemas são comparados em 4 tipos distintos (**One-DR-F1**, **One-DR-F2**, **One-ASO-F1** e **One-ASO-F2**). Um resumo das comparações em relação ao tipo **One-DR** para ambos *datasets* é apresentado na Tabela 4 e na Tabela 5.

Tabela 4: Resultados experimentais em páginas One-DR no *dataset* TBDW versão 1.02

Etapas	**One-DR-F1**			**One-DR-F2**		
Sistemas	**ViDE**	**AutoRM**	**DEPTA**	**ViDE**	**AutoRM**	**DEPTA**
Precisão	100%	100%	100%	99,2%	99,8%	98,9%
Revocação	100%	100%	100%	98,3%	99,6%	99,6%

Fonte: Shi et al. (2015)

As comparações em relação ao tipo **One-ASO**, por sua vez, são apresentados na Tabela 6 e na Tabela 7. Como os autores não tiveram acesso aos sistemas CTVS e G-STM, eles não puderam calcular o desempenho desses sistemas em relação às páginas do tipo **One-DR**.

Tabela 5: Resultados experimentais em páginas One-DR no *dataset* dos autores

Etapas	**One-DR-F1**			**One-DR-F2**		
Sistemas	**ViDE**	**AutoRM**	**DEPTA**	**ViDE**	**AutoRM**	**DEPTA**
Precisão	88,6%	95,4%	82,8%	97,8%	99,6%	96,9%
Revocação	85,4%	96,9%	73,3%	91,5%	98,5%	89,9%

Fonte: Shi et al. (2015)

Tabela 6: Resultados experimentais em páginas One-ASO no *dataset* TBDW versão 1.02

Etapas	**One-ASO-F1**		**One-ASO-F2**	
Sistemas	**DEPTA**	**AutoRM**	**DEPTA**	**AutoRM**
Precisão	100%	100%	100%	100%
Revocação	100%	100%	60%	100%

Fonte: Shi et al. (2015)

Tabela 7: Resultados experimentais em páginas One-ASO no *dataset* dos autores

Etapas	**One-ASO-F1**		**One-ASO-F2**	
Sistemas	**DEPTA**	**AutoRM**	**DEPTA**	**AutoRM**
Precisão	97,6%	98,2%	97,7%	100%
Revocação	71,4%	96%	82%	99,6%

Fonte: Shi et al. (2015)

2.3.5 Conclusões

Shi et al. (2015) propõe uma nova abordagem para a extração automática de dados estruturados de páginas *web*, chamado AutoRM. Em sua pesquisa, os autores demonstraram novos algoritmos que realizam suposições mais robustas de como os registros de dados são codificados na página. A partir dos resultados experimentais demonstrados, é possível concluir que essa abordagem é efetiva, obtendo desempenho superior em relação às outras abordagens comparadas.

Como continuidade para a sua pesquisa, os autores apontam certas limitações que podem ser estudadas no futuro:

1. O AutoRM funciona apenas em páginas cuja regiões de dados contenham mais de um registro de dados. Essa limitação é presente em todas as abordagens que se baseiam na extração a partir de uma única página;

2. A abordagem não extrai e alinha itens de dados entre registros similares. Essa funcionalidade seria útil para a correta exibição dos registros de dados encontrados na página;

3. O AutoRM não gera um *wrapper* para a extração de outras páginas a partir de uma

página de exemplo. A geração de *wrappers* é uma abordagem utilizada por outros sistemas que buscam eficiência para a mineração de registros de dados.

3 PROPOSTA DE SOLUÇÃO

A partir da pesquisa apresentada, é possível perceber certas abordagens em comum entre os trabalhos analisados. As técnicas G-STM (JINDAL; LIU, 2010), FiVaTech (KAYED; CHANG, 2010) e AutoRM (SHI et al., 2015) utilizam a árvore DOM das páginas como entrada para os algoritmos. A extração de registros de dados é realizada através da travessia recursiva da árvore, onde a correspondência e alinhamento de nodos é realizada em cada nível.

As pesquisas de Jindal e Liu (2010) e Kayed e Chang (2010) estabeleceram modelos de dados para auxiliá-los na definição de problema. Estes modelos de dados definem como os dados são estruturados em diferentes tipos, guiando o desenvolvimento dos seus algoritmos. Baseado nestes trabalhos, a proposta de solução utiliza o seguinte modelo de dados:

1. É estabelecido o conjunto de *tipos básicos* $B = \{B_1, B_2, ..., B_k\}$. Cada tipo B_i é atômico e uma instância de B_i representa uma *string* ou outro componente atômico (como uma imagem ou vídeo);
2. Se $T_1, T_2, ..., T_n$ são *tipos básicos* ou *tipos conjunto*, então $[T_1, T_2, ..., T_n]$ representa um *tipo tupla*, onde cada instância é representada por $[v_1, v_2, ..., v_n]$, onde os valores $v_1, v_2, ..., v_n$ pertencem, respectivamente, aos tipos $T_1, T_2, ..., T_n$;
3. Se T é um tipo tupla, então $\{T\}$ é um *tipo conjunto* cujos valores são instâncias do tipo T.

Conforme explicado anteriormente, a extração de dados de vagas de empregos da startup Jober é realizada, atualmente, através de uma abordagem manual. *Wrappers* são desenvolvidos para a extração dos dados específicos que são pertinentes para a aplicação. Esses *wrappers* estão integrados junto de um *scrapper*, que é responsável por gerenciar as requisições das páginas, que são realizadas através do protocolo HTTP (*Hypertext Transfer Protocol*).

O protótipo desenvolvido funciona dentro deste contexto. Porém, de modo a facilitar a implementação da aplicação e manter o foco na pesquisa realizada, certos aspectos do contexto de execução foram abstraídos durante o desenvolvimento. Mais especificamente, foram abstraídas as etapas de requisição dos conteúdos das páginas, assim como as etapas de armazenamento dos registros no banco de dados da aplicação. Com isso, é possível delimitar o escopo do protótipo como um *software* que recebe como entrada páginas HTML já coletadas e retorna como saída uma lista de dados estruturados das vagas da página.

Além disso, outros aspectos essenciais para a extração de dados de vagas de emprego - como a rotulação dos dados extraídos e a extração a partir de páginas de detalhe - não foram abordados nesta pesquisa, devido à limitações de tempo. Estes aspectos não abordados são mencionados com maior detalhe no Capítulo 5. Assim, o protótipo proposto foi desenvolvido para receber como entrada páginas HTML contendo listas de vagas de emprego.

Este capítulo descreve os componentes presentes no protótipo desenvolvido. A seção 3.1 explica as etapas necessárias para a geração de *wrappers*. A seção 3.2 demonstra os detalhes de como os *wrappers* são utilizados para a extração de dados a partir de páginas HTML. Ao final, a seção 3.3 detalha como a lista principal da página é identificada a partir dos dados estruturados obtidos.

3.1 GERAÇÃO AUTOMÁTICA DE *WRAPPER*

A abordagem adotada para geração de *wrappers* foi baseada no trabalho de Jindal e Liu (2010). Recebendo como entrada a página web no formato HTML, o protótipo gera um *wrapper* capaz de extrair itens de dados de uma página que contenha o mesmo *template*. Para isso, é necessário obter a representação da árvore DOM da página. O protótipo foi desenvolvido com a linguagem de programação *Go* (THE. . . , 2018), e utiliza o pacote de funções *html* (PACKAGE. . . , 2018) para interpretar o arquivo HTML da página e obter os nodos que a constituem.

Porém, para facilitar a navegação através da árvore DOM e evitar operações repetidas, essa árvore é convertida para um formato simplificado, utilizado por todo o protótipo. A unidade básica utilizada neste formato é o *NodeTree*, que representa um nodo específico da árvore da página. Um objeto *NodeTree* é composto por três propriedades:

- *Node:* referência para o objeto **html.Node* original, que é gerado originalmente pelo pacote *html*;
- *Children:* lista com referências para os nodos *NodeTree* filhos;
- *Size:* quantidade de itens presentes na propriedade *Children*.

Durante o processo de conversão árvore para o formato *NodeTree*, são filtrados nodos de elementos que não contenham conteúdo pertinente (como as *tags* *<style>*, *<script>*, ...). A remoção desses nodos melhora o desempenho da geração de *wrappers*, pois evita comparações desnecessárias.

Após a conversão da árvore, a função *Gstm* (discutida na próxima subseção) é invocada. Como a função *Gstm* exige duas árvores para o seu funcionamento, a árvore é “clonada” e ambas as cópias são utilizadas como entrada para a execução do algoritmo.

3.1.1 Função *Gstm*

A função *Gstm* implementada é responsável por calcular a pontuação da correspondência máxima entre duas árvores. Para isso, ela utiliza o conceito de **mapeamento máximo** entre duas árvores. Essa técnica utiliza recursividade para calcular a quantidade de pares de nodos similares entre os níveis de duas árvores diferentes. Como define Jindal e Liu (2010, p. 387, tradução nossa):

> Considerando que A e B são duas árvores, e $i \in A$ e $j \in B$ são dois nodos em A e B, respectivamente. Uma **correspondência** entre duas árvores é definida como um mapeamento M em que, para cada par $(i,j) \in M$ onde i e j são nodos não-raiz, (pai(i), pai(j)) $\in M$. Uma **correspondência máxima** é uma correspondência com o maior número de pares possíveis.

Tendo as árvores $A = R_{\text{A}} : \langle A_1, ..., A_k\rangle$ e $B = R_{\text{B}} : \langle B_1, ..., B_k\rangle$, onde R_A e R_B são as raízes de A e B, e A_i e B_j são as árvores de índices i e j de A e B, respectivamente. O número máximo de pares entre as árvores A e B é definida por *W(A, B)*. Se R_A e R_B representarem símbolos distintos, *W(A, B)* $= 0$. Caso contrário, *W(A, B)* $= m(\langle A_1, ..., A_k\rangle, \langle B_1, ..., B_k\rangle) + 1$, onde $m(\langle A_1, ..., A_k\rangle, \langle B_1, ..., B_k\rangle)$ representa o número de pares na correspondência máxima entre as subárvores de A e B. A descrição da técnica é exibida na Figura 6.

Figura 6: Algoritmo para cálculo de correspondência entre árvores

$$W(A,B) = \begin{cases} 0 & \text{if } R_A \neq R_B \\ m(\langle A_1,\ldots,A_k\rangle,\langle B_1,\ldots,B_n\rangle)+1 & \text{otherwise} \end{cases};$$

$$m(\langle\rangle, \langle\rangle) = 0 \qquad \text{// } \langle\rangle \text{ represents an empty sub-tree list.}$$

$$m(s, \langle\rangle) = m(\langle\rangle, s) = 0 \qquad \text{// } s \text{ matches any non-empty sub-tree list}$$

$$\begin{aligned} m(\langle A_1, \ldots, A_k\rangle, \langle B_1, \ldots, B_n\rangle) = \max(&m(\langle A_1, \ldots, A_{k\text{-}1}\rangle, \langle B_1, \ldots, B_{n\text{-}1}\rangle) + W(A_k, B_n), \\ &m(\langle A_1, \ldots, A_k\rangle, \langle B_1, \ldots, B_{n\text{-}1}\rangle), \\ &m(\langle A_1, \ldots, A_{k\text{-}1}\rangle, \langle B_1, \ldots, B_n\rangle)). \end{aligned}$$

Fonte: Liu (2011, pág. 387)

O pseudocódigo da função *Gstm* é apresentado na Figura 7. A execução se encerra na linha 3 caso os nodos-raiz de A e B sejam diferentes. Caso contrário, as linhas 5-10 geram a matriz W através de chamadas recursivas para *Gstm*. A função *DetectLists* é chamada na linha 12 para atualizar a matriz W. Após isso, as linhas 14-23 realizam o cálculo da correspondência máxima entre A e B.

No geral, o código é bastante similar ao pseudocódigo apresentado anteriormente na Figura 2. A principal diferença é a adaptação da função para retornar a gramática gerada na função *DetectLists*. Essa alteração foi realizada pois o trabalho original de Jindal e Liu (2010) não demonstrou de maneira clara como as gramáticas geradas são armazenadas para a sua posterior utilização durante a extração de dados.

Figura 7: Pseudocódigo da função *Gstm*

```
1  Função: Gstm(A, B)
2       se os nodos−raiz de A e B (A.Node e B.Node, respectivamente)
            diferem então
3           retorne (0, A.Size, b.Size, nulo)
4       senão
5           W ← matriz de largura A.Size e altura B.Size
6           para i = 0 até A.Size−1 faça
7               para j = 0 até B.Size−1 faça
8                   W[i, j] ← Gstm(A.Children[i], B.Children[j])
9               fimpara
10          fimpara
11
12          (W, NodesA, NodesB, Grammar) ← DetectLists(W, A, B)
13
14          M ← matriz de largura A.Size e altura B.Size
15          para i = 0 até A.Size−1 faça
16              para j = 0 até B.Size−1 faça
17                  M[i, j] ← MAX(
18                        ValidValue(M, i, j−1),
19                        ValidValue(M, i−1, j),
20                        ValidValue(M, i−1, j−1) + W[i, j].Score
21                  )
22              fimpara
23          fimpara
24
25          Score ← 1
26          se A.Size > 0 e B.Size > 0 então
27              Score ← Score + M[A.Size−1, B.Size−1]
28          fimse
29
30          retorne (Score, NodesA, NodesB, Grammar)
31      fimse
32
33 Função: ValidValue(M, i, j)
34      se i < 0 ou j < 0 então
35          retorne 0
36      senão
37          retorne M[i, j]
38      fimse
```

Fonte: próprio autor

3.1.2 Função *DetectLists*

A função *DetectLists* é responsável pela identificação de listas e geração de gramáticas para a extração de dados. Essa função recebe como argumentos a matriz W, além das árvores A e B sendo comparadas. A matriz W, como foi explicado na seção anterior, armazena a pontuação da comparação entre todos os nodos-filho da árvore A com os nodos-filho da árvore B.

Uma lista nada mais é que um conjunto de itens similares. Logo, a primeira etapa dessa função é verificar similaridade entre os nodos-filhos de ambas as árvores. Para isso,

é gerada uma nova matriz *NormW*, que armazena os valores de W normalizados. Essa processo de normalização impede que a quantidade de nodos em uma árvore influencie na comparação das pontuações que serão realizadas posteriormente. A normalização de cada valor $W[i, j]$ é obtida através da divisão da pontuação *Score* pelo maior valor entre as quantidades *NodesA* e *NodesB*. De modo formal, é descrita pela seguinte fórmula:

$$NormW[i,j] = \frac{W[i,j].Score}{max(W[i,j].NodesA, W[i,j].NodesB)} \tag{3.1}$$

É então realizada a comparação entre os nodos-filhos com base nos valores da matriz *NormW*. Cada par de subárvores (i, j) das árvore A e B será considerado similar caso atendam dois requisitos, apontados por Jindal e Liu (2010):

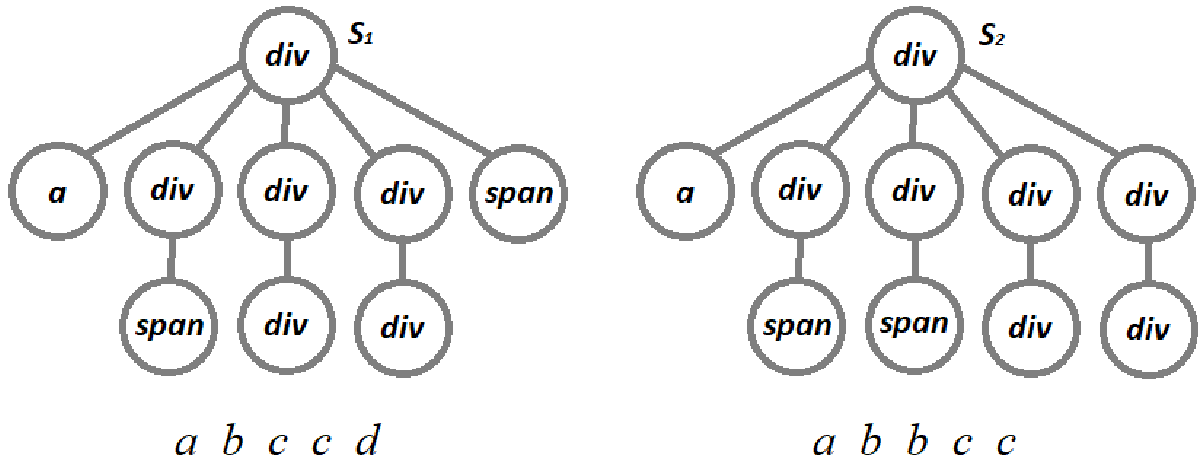
- A pontuação normalizada $NormW[i, j]$ deve ser maior que um *threshold* τ_1 específico. Seguindo a pesquisa dos autores, τ_1 foi definido como 50%;
- De acordo com as observações dos autores, uma subárvore que compõe uma lista junto de outras subárvores não deve ter uma pontuação $NormW[i, j]$ muito distante da pontuação máxima que esta subárvore possui com qualquer outra. Por isso, a proporção de $NormW[i, j]$ pela pontuação máxima não pode ser inferior ao *threshold* τ_2, estabelecido como 70%. Essa comparação é realizada tanto para a subárvore A_i quanto para B_j.

A comparação de similaridade entre subárvores é utilizada então para a geração de duas *strings* de símbolos, uma para cada árvore. Os símbolos, neste caso, representam as subárvores que foram comparadas entre as duas árvores. As subárvores similares que atenderem os critérios acima recebem o mesmo símbolo, enquanto subárvores distintas recebem símbolos diferentes. A Figura 8 exibe um exemplo de duas *strings* geradas a partir de árvores com subárvores similares. A partir das subárvores de S_1 e S_2 é possível obter as *strings* *abccd* e *abbcc*, respectivamente.

Após a geração das duas *strings* *treeStringA* e *treeStringB*, a função *GrammarGeneration* é executada para cada uma delas. Essa função retorna uma gramática que representa a sequência de símbolos presente em cada *string*. Essas gramáticas permitem a identificação de padrões nas *strings* encontradas, sendo utilizadas na extração de dados que ocorre depois da geração do *wrapper*. A função *GrammarGeneration* é explicada em maiores detalhes na próxima subseção.

A parte final da função *Detect-Lists* trata da atualização da matriz W de acordo com as possíveis listas identificadas a partir das gramáticas. Caso as gramáticas sejam parcialmente correspondentes, a função *UpdateW* é utilizada para atualizar a matriz W. A correspondência parcial entre duas gramáticas é avaliada através da comparação das

Figura 8: Exemplo da geração de *strings* a partir de árvores similares



Fonte: próprio autor

expressões regulares de cada uma, ignorando partes das expressões que não forem listas. A correspondência parcial será melhor explicada na subseção 3.2.1. Essa etapa de atualização da matriz W é necessária para que a quantidade de itens em cada lista seja reduzida à apenas uma instância. Isso impede que a quantidade de itens em uma lista influencie possíveis comparações nos níveis superiores das ávores A e B. Mais detalhes da função *UpdateW* são avaliados na subseção 3.1.4.

A Figura 9 exibe o pseudocódigo da função *DetectLists*. As linhas 2-7 geram a matriz *NormW*, utilizando a fórmula 3.1. As linhas 12-24 preenchem as *strings* de acordo com os símbolos obtidos pelas correspondências encontradas entre A e B. As linhas 29-37 obtém as gramáticas a partir das *strings*, chamando *UpdateW* se necessário.

A versão da função *DetectLists* utilizada nesta pesquisa possui algumas adaptações em relação à original, proposta por Jindal e Liu (2010). Uma delas é a utilização da *treeStringA* e *treeStringB* para a geração uma última gramática final que aceita ambas as árvores A e B. O trabalho original não demonstrava de modo claro como as gramáticas geradas nos diversos níveis das árvores eram armazenadas para a extração. Ao retornar essa gramática ao final da função, ela pode ser armazenada na matriz W dos níveis superiores, durante a chamada de *Gstm*.

Para completar a ligação entre as gramáticas dos diversos níveis da árvore, a criação dos símbolos que preenchem as *strings treeStringA* e *treeStringB* foi alterada para registar as diferentes gramáticas geradas pelas subárvores que se encaixam nos critérios de similaridade citados anteriormente. Ou seja, toda vez que é detectada a similaridade entre as subárvores A_i e B_j, a gramática $NormW[i, j].Grammar$ é atrelada ao símbolo gerado. Isso permite saber quais gramáticas devem ser executadas durante a extração de dados, conforme será explicado melhor na próxima seção.

Figura 9: Pseudocódigo da função *DetectLists*

```
1  Função: DetectLists(W, A, B)
2      NormW ← matriz de largura A.Size e altura B.Size
3      para i = 0 até A.Size−1 faça
4          para j = 0 até B.Size−1 faça
5              NormW[i, j] ←W[i, j].Score / MAX(W[i, j].NodesA, W[i,
                   j].NodesB)
6          fimpara
7      fimpara
8
9      treeStringA ← string de tamanho A.Size
10     treeStringB ← string de tamanho B.Size
11
12     para i = 0 até A.Size−1 faça
13         para j = 0 até B.Size−1 faça
14             maxA ← valor máximo que A.Children[i] possui com
                   qualquer outra subárvore de B
15             maxB ← valor máximo que B.Children[j] possui com
                   qualquer outra subárvore de A
16
17             se NormW[i, j] > τ1 e maxA > τ2 e maxB > τ2 então
18                 s ← símbolo gerado a partir de A.Children[i] e B.
                       Children[j], contendo W[i, j].Grammar
19
20                 treeStringA[i] ← s
21                 treeStringB[j] ← s
22             fimse
23         fimpara
24     fimpara
25
26     se A.Node é um nodo textual ou (A.Size = 0 e B.Size = 0) então
27         retorne (W, 1, 1, nulo)
28     senão
29         NodesA ← A.Size, NodesB ← B.Size
30         grammarA ← GrammarGeneration(treeStringA)
31         grammarB ← GrammarGeneration(treeStringB)
32
33         se grammarA e grammarB são parcialmente correspondentes
               então
34             (W, NodesA, NodesB) ← UpdateW(W, treeStringA, grammarA
                   , treeStringB, grammarB)
35         fimse
36
37         final ← GrammarGeneration(treeStringA, treeStringB)
38         retorne (W, NodesA, NodesB, final)
39     fimse
```

Fonte: próprio autor

3.1.3 Função *GrammarGeneration*

GrammarGeneration é a função responsável por induzir uma gramática a partir de *strings* contendo uma sequência de símbolos. O termo gramática, neste contexto, representa um autômato finito não-determinístico (AFND) contendo diferentes estados e

transições entre eles. A partir das gramáticas geradas, é possível obter expressões regulares que auxiliam o processo de extrações de dados.

É importante citar que o método apresentado se trata de uma heurística para tentar inferir a gramática por trás de uma sequência de símbolos, e pode gerar resultados incorretos em casos onde a sequência de símbolos apresenta um padrão mais complexo. Como os próprios autores Jindal e Liu (2010) apontam durante a explicação da proposta original, a inferência completa de gramáticas regulares a partir de exemplos positivos apenas não é viável.

A função *GrammarGeneration* tem uma premissa simples: a partir da leitura de cada símbolo presente nas *strings*, a gramática é expandida de modo a criar estados e transições que permitam a identificação de outras *strings* no mesmo formato. Cada gramática, representada aqui pelo tipo *TreeGrammarNfa*, contém algumas propriedades básicas:

- *States:* os estados (tipo *NfaState*) pertencentes à gramática;
- *Transitions:* as transições existentes entre os estados. Por exemplo: o estado *q1*, lendo o símbolo *a*, vai para *q2*;
- *startState:* o estado inicial da gramática;
- *acceptState:* o estado final da gramática.

A proposta original de Jindal e Liu (2010) utiliza apenas uma *string* para a geração da gramáticas. A adaptação para o uso de múltiplas *strings* foi adotada neste trabalho para permitir que sejam geradas gramáticas mais flexíveis, com base nas *strings* geradas na função *DetectLists*. Nesta primeira etapa, as *strings* são ordenadas de maneira decrescente, para que depois cada uma seja interpretada sequencialmente durante a expansão da gramática.

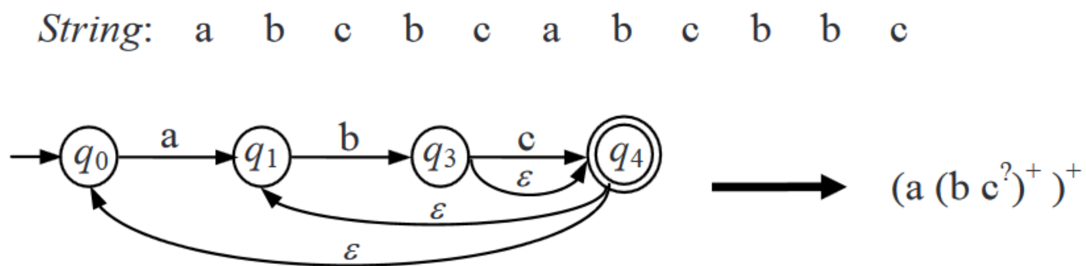
A leitura dos símbolos de cada *string* é então realizada. A partir do estado inicial, que é considerado como o estado atual q_c, cada símbolo s é lido e avaliado com base nos seguintes critérios:

1. Se já existir uma transição do estado atual q_c para outro estado q_n utilizando o símbolo s ($\delta(q_c, s) = q_n$), nenhum adição é realizada e estado atual é definido como q_n (ou seja, $q_c \leftarrow q_n$);
2. Senão, caso já exista algum transição $\delta(q_i, s) = q_j$ que utilize o símbolo s, são criadas transições com o símbolo ϵ que conectam o estado q_c com q_i. Além disso, q_c recebe q_j ($q_c \leftarrow q_j$);

3. Se nenhum dos critérios anteriores for atendido, a transição $\delta(q_c, s) = q_{c+1}$ é adicionada à gramática. Caso o estado q_{c+1} não exista, ele é criado e também adicionado à gramática.

A Figura 10 exibe o exemplo de um AFND gerado a partir de uma *string* de símbolos. Ao lado dele, é possível visualizar a expressão regular que representa a gramática obtida.

Figura 10: Examplo de geração de gramática para uma *string* de símbolos



Fonte: Liu (2011, pág. 411)

A Figura 11 exibe o pseudocódigo da função *GrammarGeneration*. Além da adaptação para a utilização de múltiplas *strings*, outras adições que não constavam de maneira explícita na função original foram realizadas. Boa parte dessas adições são de estruturas de dados que auxiliam a manipulação dos estados e transições das gramáticas geradas.

3.1.4 Função *UpdateW*

A função *UpdateW* é utilizada para atualizar a matriz W de acordo com as listas identificadas através das gramáticas geradas na etapa anterior. Conforme foi explicado anteriormente, essa atualização é necessária para impedir que a quantidade de itens identificados em cada lista influencie nas pontuações das comparações entre subárvores acima na hierarquia DOM da página, garantindo assim melhores resultados nas comparações. A função recebe como parâmetro a matriz W, junto das *strings* e gramáticas geradas na função *DetecLists*.

A primeira etapa da função é obter os padrões de repetição que constam nas expressões regulares das gramáticas. Esses padrões são identificados pelo formato $(D)^+$. Por exemplo, na expressão regular $a^+b(cd)^+e^?$ é possível obter as repetições a^+ e $(cd)^+$. Para cada padrão identificado, são obtidos os símbolos pertencentes à esses padrões (considerando os exemplos citados, seriam resgatados os símbolos a, c e d). A subseção 3.2.1 explica como essas expressões são obtidas a partir de gramáticas.

Para cada um dos símbolos S identificados, são recuperadas na matriz W as pontuações $W[i, j]$ que foram marcadas como S. A partir dessas pontuações, é obtida a pon-

Figura 11: Pseudocódigo da função *GrammarGeneration*

```
1  Função: GrammarGeneration(...strings)
2      grammar ← nova gramática
3      stringList ← a lista de strings, ordenada de modo decrescente
4      para cada string em stringList faça
5          Induce(grammar, string)
6      fimpara
7
8      retorne grammar
9
10 Função: Induce(grammar, string)
11     qc ← grammar.startState
12     para cada símbolo s em string faça
13         se existe δ(qc, s) = qn então
14             qc ← qn
15         senão se existe δ(qi, s) = qj então
16             utiliza transições ϵ para conectar qc e qi
17             qc ← qj
18         senão
19             qm ← estado qc + 1 (caso não exista, é gerado)
20             δ(qc, s) = qm
21             qc ← qm
22         fimse
23     fimpara
```

Fonte: próprio autor

tuação média *avgScore* entre todas elas, assim como a quantidade média de subárvores que os nodos marcados com S possuem, definida como *avgNodes*.

A ideia básica dessa função é colapsar as diversas subárvores marcadas com S para apenas uma subárvore, que represente a pontuação média de todas elas. Na prática isso é realizado de modo simples: a primeira ocorrência $W[i, j]$ do símbolo S em cada árvore recebe a pontuação *avgScore*, enquanto a pontuação do restante das ocorrências é zerada.

Ao final, os valores de *avgNodes* de cada símbolo são somados entre si para compor os valores de *nodesA* e *nodesB*, que são retornados junto de W. Além disso, ambas as variáveis recebem também a quantidade de subárvores do restante dos nodos que não foram marcados por algum dos símbolos colapsados.

O pseudocódigo da função *UpdateW* é exibido na Figura 12. As expressões de repetição são recuperadas na linha 4. Para cada símbolo presente em cada expressão, são colapsados as pontuações para apenas uma instância: nas linhas 11-23, a pontuação média é calculada e inserida na primeira ocorrência do símbolo em cada *string*. As demais ocorrências recebem o valor 0. Nas linhas 31-37, a quantidade de subárvores dos nodos que não foram colapsados são somadas aos valores *nodesA* e *nodesB*.

Figura 12: Pseudocódigo da função *UpdateW*

```
1  Função: UpdateW(W, treeStringA , grammarA, treeStringB , grammarB)
2      nodesA ← 1
3      nodesB ← 1
4      repeatList ← expressões de repetição presentes nas gramáticas
          grammarA e grammarB
5
6      para cada repeat em repeatList faça
7          symbols ← símbolos presentes na expressão repeat
8          para cada S em symbols faça
9              pairs ← índices das subárvores marcadas com o símbolo
                  S em treeStringA e treeStringB
10
11             avgScoreSum ← 0
12             avgNodesSum ← 0
13             count ← 0
14             para cada {i, j} em pairs faça
15                 avgScoreSum ← avgScoreSum + W[i, j].Score
16                 avgNodesSum ← avgNodesSum + W[i, j].NodesA + W[i,
                      j].NodesB
17                 count ← count + 1
18             fimpara
19
20             avgScore ← avgScoreSum / count
21             W[pair.i, pair.j].Score ← avgScore, onde pair ← pairs[
                  x] e x = 0
22             W[pair.i, *].Score ← 0, onde pair ← pairs[x] e x > 0
23             W[*, pair.j].Score ← 0, onde pair ← pairs[x] e x > 0
24
25             avgNodes ← avgNodesSum / (count*2)
26             nodesA ← nodesA + avgNodes
27             nodesB ← nodesB + avgNodes
28         fimpara
29     fimpara
30
31     para cada i de treeStringA que não está em repeatList faça
32         nodesA ← nodesA + w[i, 0].NodesA
33     fimpara
34
35     para cada j em treeStringB que não está em repeatList faça
36         nodesB ← nodesB + w[0, j].NodesB
37     fimpara
38
39     retorne (W, nodesA, nodesB)
```

Fonte: próprio autor

3.2 EXTRAÇÃO DE DADOS

A partir dos *wrappers* gerados, é possível então realizar a extração de dados de páginas HTML. Um *wrapper* gerado a partir de um *template* T só consegue extrair dados de páginas que também sigam o mesmo *template* T. A extração é realizada através das gramáticas obtidas durante a geração do *wrapper*. A partir da gramática do nodo raiz até

as gramáticas de níveis inferiores, a árvore DOM da página é percorrida recursivamente até que todos os itens de dados (ou seja, nodos de texto) sejam extraídos.

Os itens de dados são extraídos dos nodos HTML conforme a expressão regular de cada gramática que for utilizada. Na prática, cada expressão descreve qual é o padrão de nodos que devem ser extraídos a partir de uma determinada árvore. A pesquisa de Jindal e Liu (2010) não detalha na prática como as expressões regulares são utilizadas durante a extração de dados. Deste modo, a proposta explicada a seguir não reflete totalmente a abordagem do trabalho original.

Esta seção se divide em duas partes. A subseção 3.2.1 demonstra como expressões regulares são geradas a partir de gramáticas. A utilização dessas expressões para a subsequente extração é explicada na subseção 3.2.2.

3.2.1 Expressões Regulares

As expressões regulares, no contexto desta pesquisa, são utilizadas para a extração de dados estruturados, sendo geradas a partir das gramáticas obtidas anteriormente. Cada expressão é composta por outras subexpressões, que por sua vez também podem ser compostas de outras subexpressões indefinidamente. Uma expressão regular permite validar se uma sequência de nodos é aceita ou não pela gramática correspondente.

A unidade básica de uma expressão são símbolos, representados aqui pelo tipo *GrammarSymbol*. Esses símbolos, utilizados anteriormente para formar *strings* na função *DetectLists*, determinam o tipo de nodo que deve ser identificado para que a sequência de nodos seja válida. Além disso, cada símbolo referencia as gramáticas que podem ser utilizadas para a extração das subárvores dos nodos identificados.

A geração de expressões regulares do protótipo trabalha com cinco tipos de expressões. Com exceção do tipo *GrammarSymbol*, todos podem conter outras expressões de mesmo tipo ou não. Além de *GrammarSymbol*, o restante dos tipos definidos são explicados abaixo:

- *Conjunction:* representa uma sequência de símbolos ou expressões. Um exemplo seria a expressão ABC;
- *Disjunction:* representa uma disjunção de expressões, como $A|B|C$;
- *Repeat:* representa uma repetição, permitindo a extração de listas. Apresenta o formato A^+;
- *Optional:* representa uma expressão opcional, e é representado por $A^?$.

A geração da expressão regular se inicia pela função *RegularExpression*, que recebe como parâmetro a gramática para a qual se deseja obter a expressão. Após a criação de

uma expressão do tipo *Conjunction* vazia, é realizada a chamada para a função *ProcessStatePattern*, passando o estado inicial da gramática.

A função *ProcessStatePattern* recebe como parâmetros a gramática em si, um estado q_c (representado pelo tipo *NfaState*) e a conjunção a ser ampliada. Ainda há um último parâmetro opcional *stopState*, que permite interromper a execução da função caso seu valor seja igual ao estado q_c. No geral, o funcionamento desta função é simples: se q_c possui transições ϵ para estados q_i onde $i > c$, essas transições (apelidadas de *jumps*) são colapsadas sequencialmente para expressões opcionais(*Optional*) ou de repetição(*Repeat*) através da função *CollapseJumps*. Caso contrário, a função *ExtendConjunctionUntil* é utilizada para continuar a construção da expressão.

A Figura 13 exibe o pseudocódigo da funções *RegularExpression* e *ProcessStatePattern*. As linhas 2-4 chamam a função *ProcessStatePattern*, passando o estado inicial da gramática e uma expressão *Conjunction* vazia. Na linha 11 é obtida a pilha de *jumps* do estado *current*. Se esta pilha não estiver vazia, a função *CollapseJumps* é invocada e retorna os novos valores de *current* e *conjunction*, que serão reutilizados na chamada recursiva da própria função novamente. Se *stack* estiver vazia, a função *ExtendConjunction* é utilizada.

Figura 13: Pseudocódigo das funções *RegularExpression* e *ProcessStatePattern*

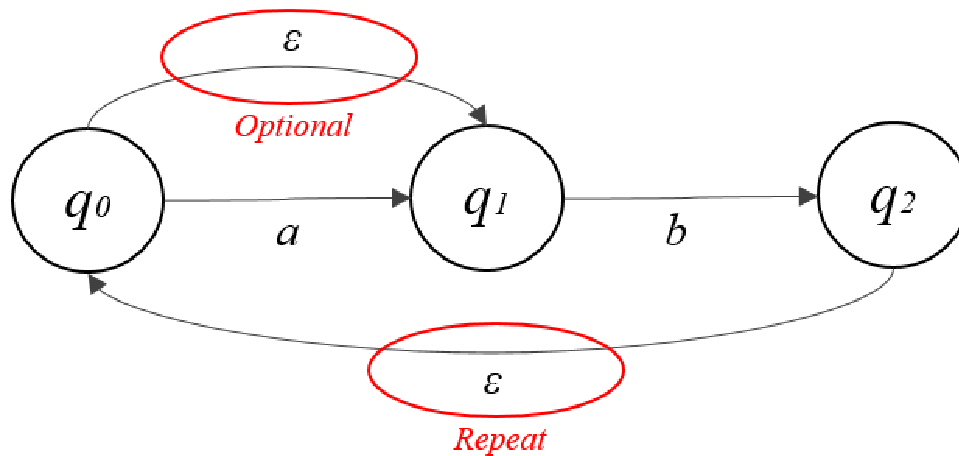
```
1  Função: RegularExpression(grammar)
2      current ← grammar.startState
3      conjunction ← nova expressão do tipo Conjunction
4      retorne ProcessStatePattern(grammar, current, conjunction,
          nulo)
5
6  Função: ProcessStatePattern(grammar, current, conjunction,
       stopState)
7      se current = stopState então
8          retorne conjunction
9      fimse
10
11     stack ← ExtractJumpStack(current)
12     se stack não estiver vazia então
13         (current, conjunction) ← CollapseJumps(grammar, current,
               conjunction, stack)
14         retorne ProcessStatePattern(grammar, current, conjunction,
               stopState)
15     senão
16         retorne ExtendConjunction(grammar, current, conjunction,
               stopState)
17     fimse
```

Fonte: próprio autor

A função *ExtractJumpStack* retorna uma pilha de *jumps* ordenados para o estado q_c. Esses *jumps* são compostos por estados q_i ligados à transições vazias de entrada ou

saída que envolvam q_c. Transições de saída são aquelas que saem do estado q_c para os estados q_i, a partir símbolo ϵ - ou seja, $\delta(q_c, \epsilon) = q_s$. Transições de entrada são aquelas que seguem o caminho inverso: saem dos estados q_i e chegam em q_c, ou $\delta(q_i, \epsilon) = q_c$. Na prática, transições ϵ de saída representam expressões opcionais, enquanto transições ϵ de entrada representam repetições. Utilizando como exemplo a gramática apresentada na Figura 14, é possível obter a seguinte lista de *jumps* para o estado q_0: *Jump*$\{q_1$, *Optional*$\}$, *Jump*$\{q_2$, *Repeat*$\}$.

Figura 14: Exemplo de gramática com *jumps* para o estado q_0



Fonte: próprio autor

A lista ordenada de *jumps* é utilizada pela função *CollapseJumps* para gerar expressões dos tipos *Optional* e *Repeat*. Para cada *jump* da lista, a função gera a expressão até o estado determinado, aninhando as expressões internas dentro das expressões dos *jumps* maiores. O pseudocódigo da função é exibido na Figura 15.

Por último, a função *ExtendConjunction* retorna uma nova conjunção a partir das transições não-vazias do estado q_c. Caso q_c tenha apenas uma transição não-vazia, o símbolo responsável pela transição será adicionado à nova conjunção. Caso existam múltiplas transições, será gerada uma expressão do tipo *Disjunction*, contendo os diferentes símbolos de cada transição. A Figura 16 demonstra o pseudocódigo da função *ExtendConjunction*.

É importante citar como as expressões geradas também são utilizadas para verificar a correspondência entre gramáticas. Essa verificação é realizada a partir da comparação recursiva entre as subexpressões de cada expressão regular. A comparação entre símbolos também pode ser realizada de maneira simples através dos símbolos *GrammarSymbol*, que registram o tipo e valor dos nodos em que foram baseados. O tipo informa se o nodo era um elemento HTML(como *<body>*, *<head>*, *<a>*, ...) ou se diz respeito à um nodo de texto.

Conforme citado anteriormente na subseção 3.1.4, a função *DetectLists* verifica a correspondência parcial entre gramáticas. Essa correspondência parcial é avaliada com

Figura 15: Pseudocódigo da função *CollapseJumps*

```
1  Função: CollapseJumps(grammar, current, conjunction, stack)
2      conj ← nova expressão do tipo Conjunction
3      para cada jump em jumps faça
4          state ← jump.state
5          se é a primeira iteração (ou seja, jump = jumps[0]) então
6              c ← ExtendConjunction(grammar, current, conj, state)
7          senão
8              c ← ProcessStatePattern(grammar, current, conj, state)
9          fimse
10
11         se houve alterações em conj então
12             aninhar conj em expressões Repeat e/ou Optional, de
                   acordo com jump
13         fimse
14
15         current ← state
16     fimpara
17
18     conjunction ← conjunction + conj
19     retorne (current, conjunction)
```

Fonte: próprio autor

Figura 16: Pseudocódigo da função *ExtendConjunction*

```
1  Função: ExtendConjunction(grammar, current, conjunction, stopState
       )
2      transitions ← todas as transições não-vazias de current
3
4      se existe apenas uma transição δ(current, S) = qf então
5          conjunction ← conjunction + S
6          retorne ProcessStatePattern(grammar, qf, conjunction,
               stopState)
7      senão
8          d ← nova expressão do tipo Disjunction
9          para cada δ(current, S) = qf faça
10             c ← nova expressão do tipo Conjunction
11             c ← c + S
12             d ← d + ProcessStatePattern(grammar, qf, c, stopState)
13         fimpara
14
15         retorne conjunction + d
16     fimse
```

Fonte: próprio autor

base apenas nas expressões do tipo *Repeat*, caso existam em ambas.

3.2.2 Extração de itens de dados

A partir das expressões regulares, é possível realizar a extração de dados estruturados das árvores DOM. Recursivamente, o algoritmo navega através tanto da árvore

quanto das expressões, realizando a comparação entre nodos e símbolos. A correspondência entre um nodo n e um símbolo S só é completa quando as gramáticas presentes em S conseguem extrair corretamente os itens de dados das subárvores de n.

A Figura 17 exibe o pseudocódigo da função *ExtractData*. A linha 3 chama *ExtractData*, passando como parâmetros a lista de subárvores de *nodeTree* e a expressão regular obtida pela gramática *grammar*. A função *ExtractData*, por sua vez, invoca a função correta de acordo com o tipo de *expression*.

Figura 17: Pseudocódigo das funções *ExtractData* e *ExtractExpression*

```
1  Função: ExtractData(grammar, nodeTree)
2      expression ← RegularExpression(grammar)
3      retorne ExtractExpression(nodeTree.Children, expression)
4
5  Função: ExtractExpression(children, expression)
6      t ← tipo(expression)
7      se t = Conjunction então
8          retorne ExtractConjunction(children, expression)
9      senão se t = Disjunction então
10         retorne ExtractDisjunction(children, expression)
11     senão se t = Optional então
12         retorne ExtractOptional(children, expression)
13     senão se t = Repeat então
14         retorne ExtractRepeat(children, expression)
15     senão se t = GrammarSymbol então
16         retorne ExtractSymbol(children, expression)
17     fimse
```

Fonte: próprio autor

A Figura 18 exibe o pseudocódigo da função *ExtractConjunction*. A linha 2 cria uma unidade do tipo *DataItem*, que representa uma tupla com diversos valores distintos. As linhas 4-11 preenchem essa unidade com os valores retornados por *ExtractExpression*. Se apenas um dado foi extraído, ele é retornado diretamente na linha 14. Senão, é retornada toda a tupla *di* na linha 16.

A Figura 19 exibe o pseudocódigo da função *ExtractDisjunction*. As linhas 2-7 testam todas as subexpressões de *disjunction*. Caso algumas delas funcione, a unidade *dt* é retornada imediatamente. Se nenhuma tentativa for bem sucedida, as linhas 9-10 retornam erro.

A Figura 20 exibe o pseudocódigo da função *ExtractOptional*. A partir da subexpressão de *optional* obtida na linha 2, é realizada a tentativa de extração. Caso não ocorra erro, o dado é retornado. Caso contrário, um dado do tipo *EmptyData* é retornado nas linhas 8-9. O tipo *EmptyData* indica que o dado opcional não está presente na lista de nodos *children*.

A Figura 21 exibe o pseudocódigo da função *ExtractRepeat*. A linha 2 inicia uma

Figura 18: Pseudocódigo da função *ExtractConjunction*

```
1  Função: ExtractConjunction(children, conjunction)
2      di ← nova unidade do tipo DataItem
3      nc ← children
4      para cada expression em conjunction faça
5          (nc, dt, error) ← ExtractExpression(nc, expression)
6          se error != nulo então
7              retorne (children, nulo, error)
8          fimse
9
10         di ← di + dt
11     fimpara
12
13     se di conter apenas uma unidade dt extraído então
14         retorne (nc, dt)
15     senão
16         retorne (nc, di)
17     fimse
```

Fonte: próprio autor

Figura 19: Pseudocódigo da função *ExtractDisjunction*

```
1  Função: ExtractDisjunction(children, disjunction)
2      para cada expression em disjunction faça
3          (nc, dt, error) ← ExtractExpression(children, expression)
4          se error = nulo então
5              retorne (nc, dt)
6          fimse
7      fimpara
8
9      error ← nenhuma das expressões em disjunction extraiu children
10     retorne (children, nulo, error)
```

Fonte: próprio autor

unidade do tipo *DataList*, que representa as listas a serem extraídas a partir de *children*. Nas linhas 5-8, busca-se extrair o primeiro item da lista. Em caso de erro, a função retorna imediatamente. Se essa primeira extração é corretamente realizada, as linhas 10-13 iteram novamente sobre *nc* até que todos os itens sejam extraídos.

A Figura 22 exibe o pseudocódigo da função *ExtractSymbol*. Nas linhas 2-5 e 10-13 são retornados erros caso não existam nodos para extração ou se o primeiro nodo da lista não for compatível com *S*. As linhas 15-18 verificam se o *first.Node* for um nodo textual (ou seja, é um trecho de texto da página). Nesse caso, é retornada uma unidade *DataText* com seu conteúdo. Por último, as linhas 20-30 tentam continuar a extração para as subárvores de *first* através da chamada para *ExtractData*.

Figura 20: Pseudocódigo da função *ExtractOptional*

```
1  Função: ExtractOptional(children, optional)
2      expression ← optional.expression
3      (nc, dt, error) ← ExtractExpression(children, expression)
4      se error = nulo então
5          retorne (nc, dt)
6      fimse
7
8      ed ← nova unidade do tipo EmptyData
9      retorne (children, ed)
```

Fonte: próprio autor

Figura 21: Pseudocódigo da função *ExtractRepeat*

```
1  Função: ExtractRepeat(children, repeat)
2      expression ← repeat.expression
3      dl ← nova unidade do tipo DataList
4
5      (nc, dt, error) ← ExtractExpression(children, expression)
6      se error != nulo então
7          retorne (children, nulo, error)
8      fimse
9
10     enquanto for possível extrair de nc faça
11         (nc, dt) ← ExtractExpression(nc, expression)
12         dl ← dl + dt
13     fimenquanto
14
15     retorne (nc, dl)
```

Fonte: próprio autor

3.3 IDENTIFICAÇÃO DA LISTA PRINCIPAL

Após as etapas de geração de *wrappers* e extração, são obtidos os dados estruturados de toda a página HTML que serviu de entrada para o algoritmo. Porém, nem todos esses dados são relevantes para o objetivo do protótipo, que é a extração de dados de vagas de emprego a partir de páginas de listagem de vagas. Desse modo, se torna necessária a identificação da lista principal da página.

Nesta pesquisa, a identificação da lista principal é realizada através de algumas fases de filtragem e ordenação dos dados extraídos. Essas fases funcionam como heurísticas, que buscam filtrar listas que provavelmente não contenham as vagas de emprego da página. A primeira fase é a obtenção de todas listas extraídas. A partir desse conjunto de listas, serão removidas aquelas que não atenderem aos seguintes critérios:

- A lista deve ser composta por dados do tipo *DataItem*;

Figura 22: Pseudocódigo da função *ExtractSymbol*

```
1  Função: ExtractSymbol(children, S)
2      se children estiver vazia então
3          error ← não é possível extrair a partir de uma lista vazia
4          retorne (children, nulo, error)
5      fimse
6
7      first ← children[0]
8      rest ← children[1, ..., n]
9
10     se first.Node não é correspondente à S então
11         error ← nodo first é incompatível com símbolo S
12         retorne (children, nulo, error)
13     fimse
14
15     se first.Node é um nodo textual então
16         dt ← nova unidade do tipo DataText
17         retorne (rest, dt{first.Node})
18     fimse
19
20     se S possui gramáticas então
21         para cada grammar em S.Grammars faça
22             (dt, error) ← ExtractData(grammar, first)
23             se error = nulo então
24                 retorne (rest, dt)
25             fimse
26         fimpara
27     senão
28         ed ← nova unidade do tipo EmptyData
29         retorne (rest, ed)
30     fimse
```

Fonte: próprio autor

- A lista deve conter pelo menos dois itens;
- A quantidade média de itens de dados em cada índice da lista deve ser maior ou igual a três;
- O desvio padrão da quantidade de itens de dados em cada índice deve ser de no máximo 3,5.

A partir das listas restantes, a lista principal da página será selecionada como aquela que possuir o maior tamanho. De acordo com os experimentos realizados, esses critérios conseguem extrair a listagem de vagas da página corretamente em boa parte dos casos.

4 AVALIAÇÃO DO PROTÓTIPO

Considerando que o objetivo da pesquisa é o desenvolvimento de um protótipo para a extração de vagas de emprego de páginas web, a avaliação foi planejada para verificar a eficácia do protótipo ao extrair dados de páginas HTML. Mais precisamente, busca-se comparar os dados extraídos pelo protótipo aos dados extraídos pelo processo de *wrappers* manuais utilizado atualmente pela *startup*.

A avaliação do protótipo é realizada de maneira automatizada. Inicialmente, *datasets* de comparação são obtidos a partir do *crawler* já existente, que recupera as páginas de listagem e utiliza os *wrappers* manuais para a extração. A partir dos dados extraídos por esse processo, é possível realizar a comparação com os dados resultantes do processo de extração do protótipo desenvolvido. Conforme mencionado anteriormente, o protótipo não realiza a rotulação dos itens de dados extraídos. Devido à isso, a comparação considera apenas os valores dos itens de dados extraídos pelas duas abordagens, e não os rótulos que estes itens possam vir a ter. Essa limitação da pesquisa é abordada novamente no Capítulo 5.

Este capítulo apresenta as seguintes seções: a seção 4.1 explica em detalhes o método de avaliação utilizado na pesquisa. Ao final, a seção 4.2 demonstra os resultados obtido durante a avaliação do protótipo.

4.1 MÉTODO DE AVALIAÇÃO

O método de avaliação aplicado na pesquisa busca comparar os dados extraídos pelo protótipo com os dados extraídos pelo método atualmente utilizado na *startup* Jober. Este método busca levantar as diferenças entre os dados coletados pelas duas abordagens, permitindo assim a identificação de melhorias para o protótipo.

Para que a avaliação seja corretamente realizada, é necessário que as mesmas páginas sejam extraídas por ambas as abordagens. Desse modo, o primeiro passo da avaliação é a coleta de páginas de listagem de vagas de emprego. O *crawler* utilizado atualmente pela Jober foi adaptado para coletar e armazenar N páginas de distintos *websites* de emprego. Junto da coleta, já é realizada também a extração através dos *wrappers* manuais. Os dados extraídos de cada página são então armazenados no formato JSON (BRAY, 2017), junto da página da qual ele foi extraído.

Os dados dos múltiplos *websites* em conjunto formam então um *dataset*. Nesta pesquisa, são coletadas páginas de quatro *websites* diferentes: *Indeed* (INDEED, 2018), *InfoJobs* (INFOJOBS, 2018), *VAGAS.com.br* (VAGAS. . . , 2018) e *Empregos.com.br* (EMPREGOS. . . , 2018). Esses *websites* já são atualmente indexados pela plataforma da *star-*

tup, por isso não houve muito esforço para a geração dos *wrappers* necessários para a extração. Para cada um deles, foram coletadas cinco páginas de listagem, contendo os empregos mais recentes registrados em cada plataforma.

Após a obtenção dos *datasets*, é possível realizar a comparação. O protótipo possui um módulo específico para esta função, que itera sobre os *datasets* e realiza a extração e comparação automaticamente, para cada página. Posteriormente, os resultados são exibidos para o usuário.

Para realizar a comparação entre os dois tipos de dados, é necessário convertê-los para um formato comum. Este formato é representado pelo tipo *DataItemSet*, que funciona basicamente como um conjunto de itens de dados. Como a avaliação utilizada nesta pesquisa não busca avaliar rótulos para os dados, esses conjuntos possuem apenas os itens de dados puros, não-estruturados. A conversão dos dados de cada *dataset* é simples: após o *parsing* do arquivo *JSON*, os itens de dados de cada vaga são inseridos em um objeto *DataItemSet*.

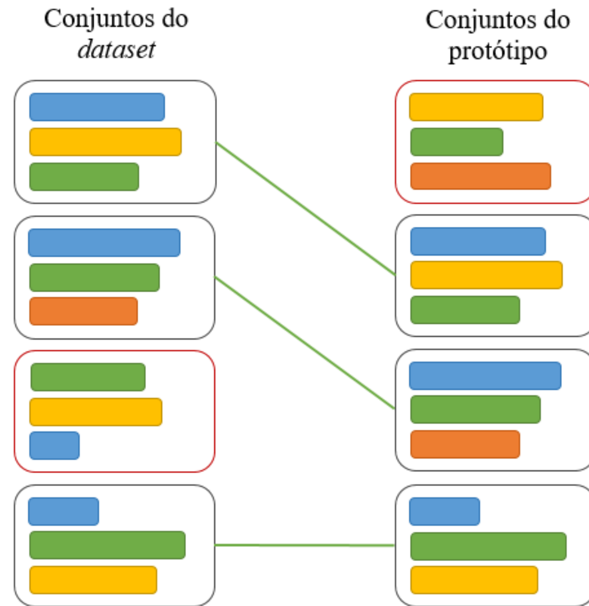
O dados provenientes da extração realizada pelo protótipo são convertidos de maneira parecida. Cada item da lista principal é convertido em um conjunto *DataItemSet*, onde os itens de dados são registrados. Itens de dados duplicados (ou seja, que contenham o mesmo conteúdo) ou vazios (como, por exemplo, itens do tipo *EmptyData*) são removidos do conjunto final. Além disso, também são filtrados os itens que estiverem presentes em mais de 80% dos conjuntos gerados. Essa filtragem impede que itens de dados relativos ao *template* sejam considerados como itens das vagas. O limiar de 80% foi obtido empiricamente, através das observações realizadas durante os testes.

A partir das etapas anteriores, são obtidas duas listas de conjuntos: uma representando as vagas da página do *dataset*, e outra com as vagas extraídas do protótipo. Essas listas são percorridas na tentativa de realizar a correspondência entre duas vagas. Duas vagas são consideradas correspondentes quando o *F-score* da comparação de seus itens de dados é superior à um limiar L específico. A Figura 23 ilustra uma situação onde algumas vagas são corretamente alinhadas ente duas listas.

Nos testes do protótipo, o valor L utilizado foi de 35%, definido empiricamente ao se verificar que vagas distintas não atingiam esse limiar. É importante citar que os trabalhos referenciados durante o embasamento teórico infelizmente não entram em detalhes de como foram realizadas as avaliações de registros de dados durante os seus experimentos. A conclusão desta pesquisa dá sugestões de como a avaliação poderia ser aprimorada em trabalhos futuros.

Ao final, o módulo apresenta as métricas obtidas durante a comparação. As quantidades de verdadeiros positivos (VP), verdadeiros negativos (VN) e falsos positivos (FP) entre os conjuntos permitem obter os valores de precisão, revocação e *F-score*.

Figura 23: Exemplo da correspondência entre conjuntos *DataItemSet*



Fonte: próprio autor

4.2 RESULTADOS

Esta seção apresenta os resultados obtidos durante a avaliação. O *dataset* utilizado foi coletado no dia 31 de outubro de 2018, a partir de quatro *websites*: *Indeed*, *InfoJobs*, *VAGAS.com.br* e *Empregos.com.br*. Para cada um, foram obtidas as primeiras cinco páginas da listagem de empregos. Conforme será explicado mais adiante, o protótipo não foi capaz de extrair vagas de emprego do *website Empregos.com.br*. Com isso, são exibidos aqui apenas os resultados obtidos com os outros três *websites*.

Os resultados obtidos durante o teste são exibidos em duas diferentes perspectivas: em relação ao número de vagas identificadas e em relação ao número de itens de dados identificados. O primeiro caso corresponde à quantidade de vagas que foram corretamente relacionadas entre o *dataset* e os dados extraídos através do protótipo, e é exibido na Tabela 8. As métricas de *Precisão*, *Revocação* e *F-score* seguem as definições explicadas no Capítulo 1.

Tabela 8: Resultados da extração de vagas do protótipo

Websites	**VP**	**FP**	**FN**	**Precisão**	**Revocação**	***F-score***
Indeed	14	13	65	52,9%	17,7%	0.26
InfoJobs	51	29	29	63.8%	63.8%	0.63
VAGAS.com.br	190	0	10	100%	95%	0.97

Fonte: próprio autor

A Tabela 9 exibe os resultados obtidos considerando a extração de itens de dados

das páginas. Para que um item de dado extraído pelo protótipo seja considerado como Verdadeiro Positivo, ele precisa pertencer à um conjunto que tenha sido correspondido na comparação à nível de vaga. Ou seja, itens de dados que pertençam à um conjunto Falso Positivo ou Falso Negativo serão automaticamente considerados Falso Positivo e Falso Negativo, respectivamente.

Tabela 9: Resultados da extração de itens de dados do protótipo

Websites	**VP**	**FP**	**FN**	**Precisão**	**Revocação**	***F-score***
Indeed	52	446	314	10,4%	14,2%	0.12
InfoJobs	143	459	309	23.8%	31.6%	0.27
VAGAS.com.br	725	470	470	64%	60,7%	0.62

Fonte: próprio autor

Os resultados obtidos para o *website Indeed* demonstra a dificuldade do protótipo em delimitar corretamente certos itens dentro da página. Devido à estrutura DOM das páginas deste *website*, certas vagas se encontravam em destaque dentro de outros elementos, diferentemente das vagas comuns. As gramáticas obtidas nestes casos eram bastante complexas, gerando muitas vezes *wrappers* incompletos.

Os resultados obtidos para os *websites InfoJobs* e *VAGAS.com.br* já se revelaram promissores. A identificação de vagas demonstrou ser capaz de extrair as listas de empregos de modo satisfatório, porém a presença de itens de dados irrelevantes dentro de cada vaga fez com que a avaliação automática não identificasse a correspondência entre algumas vagas.

Verificou-se que as páginas do *website Empregos.com.br* não puderam ser extraídas devido à problemas na geração de gramáticas específicas durante a travessia da árvore DOM das páginas. O protótipo não foi otimizado para o suporte à falhas, fazendo com que uma única gramática inválida interrompa a geração de todo o *wrapper*. O próximo capítulo dá sugestões de como esse problema pode ser abordado.

5 CONCLUSÃO

Este trabalho tem como objetivo o desenvolvimento de um protótipo para extração automática de dados de vagas de emprego a partir de páginas web. Com esse objetivo, foram estudadas técnicas de extração automática relevantes para o contexto da pesquisa, sendo aplicadas posteriormente durante o desenvolvimento da proposta de solução e avaliação do protótipo.

Ao longo do desenvolvimento do protótipo, houveram certas dificuldades durante a implementação das técnicas de extração. Essas dificuldades se originaram, em grande parte, do fato de que os trabalhos utilizados como referência teórica não descrevem de maneira clara algumas das técnicas aplicadas. Embora as dificuldades tenham sido contornadas de modo satisfatório durante o desenvolvimento, é recomendável uma pesquisa mais detalhada dessas técnicas para possíveis continuações da pesquisa.

Os resultados obtidos demonstram que a abordagem da extração automática de vagas é promissora, embora ainda esteja longe do ideal em seu estado atual. O protótipo desenvolvido demonstra-se capaz de identificar listas de vagas de empregos em diferentes tipos de páginas, de maneira não-supervisionada. Porém, muitos dados irrelevantes acabam sendo extraídos em conjunto aos dados realmente importantes. Os resultados obtidos durante a avaliação destacam a necessidade de uma melhor filtragem dos dados extraídos. Além disso, é possível apontar outros aspectos a serem aperfeiçoados:

- *Geração de gramáticas:* certas páginas apresentam sequências de nodos que geram gramáticas inválidas a partir das técnicas utilizadas. Melhorias nessa etapa podem auxiliar a obtenção de *wrappers* mais eficazes. Uma técnica que pode ser melhor explorada, por exemplo, seria a integração com o algoritmo PTA, conforme detalha Liu (2011);

- *Reutilização de wrappers:* uma das qualidades da utilização de *wrappers* é a possibilidade da sua reutilização para extração de múltiplas páginas de mesmo *template*. Trabalhos futuros podem explorar esse benefício para a melhoria do desempenho na extração de grandes quantidades de páginas;

- *Suporte à falhas:* o protótipo desenvolvido não possui suporte à possíveis falhas que venham a ocorrer durante a geração de *wrappers* ou durante a extração de dados. Quando uma exceção ocorre, o processo é interrompido no mesmo instante, perdendo-se todo o progresso. A identificação e contenção dessas exceções seria de grande utilidade nesses casos;

- *Extração de atributos:* muitos dados relevantes sobre as vagas encontram-se apenas em atributos dos nodos HTML das páginas. Durante a pesquisa realizada, não foram identificados trabalhos que realizem a extração de atributos. Pesquisas futuras poderiam abordar essa funcionalidade adicional.

Outro ponto a ser aperfeiçoado em novas propostas de pesquisa é o método de avaliação aplicado. Considerando que a finalidade do protótipo desenvolvido se encontra dentro do contexto de atuação da *startup* Jober, o método se demonstrou válido para a comparação com os dados já extraídos pela abordagem de *wrappers* manuais. Porém, abordagens mais rigorosas devem ser exploradas futuramente. A utilização de *datasets* públicos, por exemplo, permitiria a comparação da proposta de solução com técnicas já existentes.

O objetivo desta pesquisa se restringe apenas à extração de dados estruturados a partir de páginas web. Porém, no ambiente da *startup*, esse tema funciona dentro de um contexto maior, envolvendo outras atividades. Continuações dessa pesquisa podem explorar outros pontos necessários para a obtenção das informações de vagas de emprego:

- *Integração com crawler:* a abordagem utilizada atualmente na *startup* já opera em conjunto com um *crawler*, responsável por obter as páginas que serão processadas pelos *wrappers*. A proposta apresentada nesta pesquisa poderia ser integrada de maneira parecida;

- *Extração a partir de páginas de detalhe:* páginas de listagem comumente possuem *links* para as páginas próprias de cada vaga, onde informações extras podem ser consultadas. Essas páginas específicas de cada vaga são conhecidas como páginas de detalhe. Novas pesquisas podem estender a proposta atual para incluir a extração de dados a partir dessas páginas;

- *Rotulação dos dados:* a rotulação dos itens de dados é imprescindível para a completa obtenção das informações apresentadas pelas páginas. Abordagens para a anotação de dados, como a apresentada por Lu et al. (2013), poderiam ser exploradas e integradas ao protótipo;

- *Integração entre dados de diferentes websites:* é comum que empresas publiquem suas vagas em mais de um *website*, com a finalidade de atingir o maior número possível de pessoas. Técnicas para a integração de dados permitiriam que empresas e vagas similares pudessem ser relacionadas a partir de dados de diferentes fontes.

REFERÊNCIAS

ARASU, A.; GARCIA-MOLINA, H. Extracting structured data from web pages. In: ACM. *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. [S.l.], 2003. p. 337–348. Citado 4 vezes nas páginas 10, 19, 22 e 23.

BIZER, C. et al. The impact of semantic web technologies on job recruitment processes. In: *Wirtschaftsinformatik 2005*. [S.l.]: Springer, 2005. p. 1367–1381. Citado na página 10.

BRAY, T. *The javascript object notation (json) data interchange format*. [S.l.], 2017. Citado na página 51.

CRESCENZI, V. et al. Roadrunner: Towards automatic data extraction from large web sites. In: *VLDB*. [S.l.: s.n.], 2001. v. 1, p. 109–118. Citado 2 vezes nas páginas 10 e 19.

EMPREGOS.COM.BR. 2018. Disponível em: <https://www.empregos.com.br/>. Acesso em: 31 out. 2018. Citado na página 51.

INDEED. 2018. Disponível em: <https://www.indeed.com.br/>. Acesso em: 31 out. 2018. Citado na página 51.

INFOJOBS. 2018. Disponível em: <https://www.infojobs.com.br/>. Acesso em: 31 out. 2018. Citado na página 51.

JINDAL, N.; LIU, B. A generalized tree matching algorithm considering nested lists for web data extraction. In: SIAM. *Proceedings of the 2010 SIAM International Conference on Data Mining*. [S.l.], 2010. p. 930–941. Citado 16 vezes nas páginas 13, 14, 15, 16, 17, 18, 20, 27, 29, 32, 33, 34, 36, 37, 39 e 43.

KAUFMAN, L.; ROUSSEEUW, P. J. Finding groups in data. an introduction to cluster analysis. *Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics, New York: Wiley, 1990*, 1990. Citado na página 26.

KAYED, M.; CHANG, C.-H. Fivatech: Page-level web data extraction from template pages. *IEEE Transactions on Knowledge and Data Engineering*, IEEE, v. 22, n. 2, p. 249–263, 2010. Citado 6 vezes nas páginas 13, 21, 22, 23, 24 e 32.

LIU, B. Web data mining: Exploring hyperlinks, contents, and usage data. Springer, 2011. Citado 6 vezes nas páginas 10, 13, 19, 34, 40 e 55.

LIU, W.; MENG, X.; MENG, W. Vide: A vision-based approach for deep web data extraction. *IEEE Transactions on Knowledge and Data Engineering*, IEEE, v. 22, n. 3, p. 447–460, 2010. Citado na página 29.

LU, Y. et al. Annotating search results from web databases. *IEEE transactions on knowledge and data engineering*, IEEE, v. 25, n. 3, p. 514–527, 2013. Citado 2 vezes nas páginas 27 e 56.

MANNING, C. D.; SCHÜTZE, H. *Foundations of statistical natural language processing*. [S.l.]: MIT press, 1999. Citado na página 14.

PACKAGE html. 2018. Disponível em: <https://godoc.org/golang.org/x/net/html>. Acesso em: 03 nov. 2018. Citado na página 33.

SHI, S. et al. Autorm: An effective approach for automatic web data record mining. *Knowledge-Based Systems*, Elsevier, v. 89, p. 314–331, 2015. Citado 7 vezes nas páginas 24, 25, 26, 27, 29, 30 e 32.

SIMON, K.; LAUSEN, G. Viper: augmenting automatic information extraction with visual perceptions. In: ACM. *Proceedings of the 14th ACM international conference on Information and knowledge management*. [S.l.], 2005. p. 381–388. Citado na página 24.

SU, W. et al. Combining tag and value similarity for data extraction and alignment. *IEEE Transactions on knowledge and Data Engineering*, IEEE, v. 24, n. 7, p. 1186–1200, 2012. Citado na página 29.

THE Go Programming Language. 2018. Disponível em: <https://golang.org/>. Acesso em: 03 nov. 2018. Citado na página 33.

VAGAS.COM.BR. 2018. Disponível em: <https://www.vagas.com.br/>. Acesso em: 31 out. 2018. Citado na página 51.

YAMADA, Y. et al. Testbed for information extraction from deep web. In: ACM. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. [S.l.], 2004. p. 346–347. Citado 2 vezes nas páginas 19 e 29.

YANG, W. Identifying syntactic differences between two programs. *Software: Practice and Experience*, Wiley Online Library, v. 21, n. 7, p. 739–755, 1991. Citado 3 vezes nas páginas 15, 17 e 22.

ZHAI, Y.; LIU, B. Web data extraction based on partial tree alignment. In: ACM. *Proceedings of the 14th international conference on World Wide Web*. [S.l.], 2005. p. 76–85. Citado na página 20.

ZHAO, H. et al. Fully automatic wrapper generation for search engines. In: ACM. *Proceedings of the 14th international conference on World Wide Web*. [S.l.], 2005. p. 66–75. Citado na página 19.

ZHAO, H.; MENG, W.; YU, C. Automatic extraction of dynamic record sections from search engine result pages. In: VLDB ENDOWMENT. *Proceedings of the 32nd international conference on Very large data bases*. [S.l.], 2006. p. 989–1000. Citado 3 vezes nas páginas 20, 24 e 29.