UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Filipe Simões de Mendonça

ALGORITMO DE BUSCA AVANÇADA NA APLICAÇÃO NFSEARCH

Santa Maria, RS
2019

Filipe Simões de Mendonça

ALGORITMO DE BUSCA AVANÇADA NA APLICAÇÃO NFSEARCH

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

ORIENTADOR: Prof. Giovani Rubert Librelotto

455
Santa Maria, RS
2019

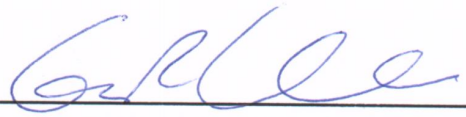
©2019
Todos os direitos autorais reservados a Filipe Simões de Mendonça. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.
End. Eletr.: fsmendonca@inf.ufsm.br

Filipe Simões de Mendonça

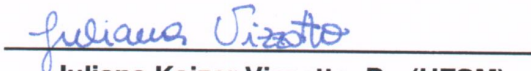
ALGORITMO DE BUSCA AVANÇADA NA APLICAÇÃO NFSEARCH

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação, Área de Concentração em , da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

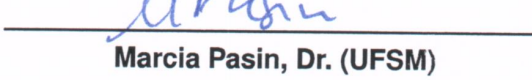
Aprovado em 8 de julho de 2019:



Giovani Rubert Librelotto, Dr. (UFSM)
(Presidente/Orientador)



Juliana Kaizer Vizzotto, Dr. (UFSM)



Marcia Pasin, Dr. (UFSM)

Santa Maria, RS
2019

RESUMO

ALGORITMO DE BUSCA AVANÇADA NA APLICAÇÃO NFSEARCH

AUTOR: Filipe Simões de Mendonça
ORIENTADOR: Giovani Rubert Librelotto

Tendo em vista o alto número de pessoas que frequentam supermercados, farmácias e demais estabelecimentos que emitam nota fiscal, uma aplicação que compare preços e ofereça ao consumidor uma opção de menor preço poderia auxiliar na rotina da população. Com essa motivação foi desenvolvida a aplicação NFSearch, um comparador de preços de notas fiscais cadastradas pelos usuários. Com esta aplicação, também se torna necessário que haja uma busca de qualidade, a partir da criação de um algoritmo de busca que utilize métodos de métrica de *string*, para realizar correção de erros e reconhecimento de abreviações, tanto nas notas fiscais como na busca dos usuários. Para tornar possível a criação do algoritmo desejado, o algoritmo de Levenshtein é um pilar para o desenvolvimento, considerando o mesmo como o melhor método de métrica de *string* desenvolvido até hoje. Com o algoritmo implementado e adicionado a aplicação NFSearch, é possível a criação de uma nova maneira de realizar compras no dia-a-dia do cidadão comum, com o objetivo de que o usuário consiga sempre a melhor oferta para o que ele desejar.

Palavras-chave: Nota Fiscal Eletrônica. Algoritmo de Levenshtein. Métricas de String.

ABSTRACT

ADVANCED SEARCH ALGORITHM IN NFSEARCH APPLICATION

AUTHOR: Filipe Simões de Mendonça
ADVISOR: Giovani Rubert Librelotto

Given the high number of people who go daily in supermarkets, pharmacies and other establishments that issue receipts, an application that compares prices and gives the consumer a lower price option could help the routine of the population. With this motivation was planned to application NFSearch, a price comparator of receipts registered by users. With the application in thesis, it is also necessary that there is a quest for quality, for this arises the idea of creating a search algorithm that uses string metric methods, to perform error correction and recognition of abbreviations, both in the receipts as in the search of users. To make the creation of the desired algorithm possible, Levenshtein's algorithm becomes a mainstay for development, considering it as the best string metric method developed to date. It is hoped that with the algorithm being implemented and added to NFSearch, it will be possible to create a new way of making everyday purchases of the ordinary citizen, in order that the user always get the best offer for what he wants.

Keywords: Eletronic Receipt. Levenshtein's Algorithm. String Metrics.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

$CNPJ$	Cadastro Nacional de Pessoas Jurídicas
$HTML$	Hypertext Markup Language
$ICMS$	Imposto sobre Circulação de Mercadoria e Serviços
IPI	Imposto sobre Produtos Industrializados
KG	Kilograma
$LTDA$	Limitada
$NF\text{-}e$	Nota Fiscal Eletrônica
$Qtde$	Quantidade
$Sefaz\text{-}RS$	Secretaria da Fazenda do Rio Grande do Sul
UF	Unidade Federativa
UN	Unidade

SUMÁRIO

1 INTRODUÇÃO

Cada vez mais a sociedade continua consumindo em maior quantidade, tanto em produtos permanentes como em produtos com validade, geralmente alimentos e produtos cosméticos (AQUINO; MARTINS, 2007). O comércio como um todo, dos de âmbito familiar até as grandes empresas, está fazendo o uso de notas fiscais vinculadas a Secretaria da Fazenda, para ter mais tranquilidade financeira evitando problemas de impostos com a Receita Federal.

Devido a essa mudança de cenário e que com o uso de notas fiscais, os produtos comprados possuem informações de fácil acesso, seria útil o uso de uma ferramenta que organize e use essas informações com a população comum, para dar ao usuário opções de compras diferentes do que ele está acostumado, podendo resultar no encontro de preços mais baixos e produtos que o mesmo não encontra perto da sua residência, além de opções diferentes, possivelmente de menor preço e próximas de sua residência para as compras do dia-a-dia do consumidor.

Quando um consumidor vai às compras em estabelecimentos comerciais em sua cidade, ele espera sempre encontrar o melhor preço possível para um produto, preferencialmente perto do local aonde ele reside. Para resolver esse problema, este trabalho propõe uma solução com base em Notas Fiscais Eletrônicas (NF-e) com um algoritmo de busca que possa corrigir erros de digitação e reconhecer tipos de produtos, mesmo sem os mesmos estarem especificados na descrição da Nota Fiscal, por exemplo busca por produtos de limpeza, que não estão descritos assim nem em Notas Fiscais Eletrônicas nem em Notas Fiscais comuns. Além do básico proposto, outras funcionalidades como comparação de preço entre estabelecimentos e onde um usuário pode encontrar produtos de uma lista de compras dada pelo mesmo e gastando menos do que gastaria e em um local próximo.

O objetivo deste trabalho consiste em desenvolver um sistema que sirva para o usuário comum usar no seu dia-a-dia, de maneira satisfatória, tanto no levantamento de produtos buscados quanto na variedade de informação retornada. O sistema deve ser de fácil uso, para ter fácil aceitação e de fácil adaptação à população comum.

Com o desenvolvimento do trabalho, propõe-se desenvolver um algoritmo de busca com o uso de métrica de *strings*, para conseguir fazer reconhecimento de erros em palavras buscadas, além de conseguir prever abreviações e tipos de produtos corretamente, mesmo sem informações do mesmo estando contidas na nota fiscal.

Para realizar o trabalho, primeiro há a necessidade de existir uma aplicação base de comparação de preço de produtos, já que as comparações são feitas para prever discrepâncias em NF-e. Tendo isso, é importante verificar os tipos de erros que podem ocorrer na entrada e na saída de informações, tanto erros simples de escrita, descrições diferentes em NF-e e também que tipos de busca o usuário pode fazer.

Esta monografia está estruturada da seguinte forma. No capítulo 2, estão os conceitos e tecnologias que serão usadas no trabalho e uma explicação sobre elas, para que torne simples o entendimento deste trabalho. No capítulo 3, a explicação e a demonstração de como será possível a realização do objetivo deste trabalho, com o uso de algoritmos e casos gerais para validar o motivo de cada algoritmo. O capítulo 4, são mostrados os seus experimentos e resultados finais. Com o capítulo 5 é mostrada a conclusão do trabalho, além de eventuais melhorias que podem ocorrer.

2 REFERENCIAL TEÓRICO

Neste capítulo são vistos conceitos que serão pertinentes ao entendimento do desenvolvimento da tese, seja em questão de algoritmo ou em questão de conhecimento para o assunto tratado.

2.1 NOTA FISCAL ELETRÔNICA

As Notas Fiscais Eletrônicas foram criadas em 2005 no II ENAT (Encontro Nacional de Administradores Tributários), em decorrência de uma ementa constitucional do ano anterior. Como dito no site da Secretaria da Fazenda (RECEITA FEDERAL, 2019), as NF-e têm como objetivo mudar significativamente o processo de emissão e gestão de informações fiscais, para beneficiar contribuintes e administrações tributárias. Para administrações é útil o aumento da confiabilidade da Nota Fiscal, melhorar o processo de controle fiscal, diminuição da sonegação de imposto e aumento da arrecadação do mesmo. No lado do contribuinte comprador, se garante uma melhor precisão na informação contida na nota, evitando erros de escrita em notas fiscais feitas à mão. Para o contribuinte vendedor, as NF-e reduzem o custo de impressão, de papel e armazenagem, além do incentivo ao uso de eletrônicos no comércio.

No Brasil, a União, os estados, o Distrito Federal e cada município possuem autonomia política, financeira e administrativa, apesar da Constituição Federal prever atribuições e limitações para os mesmos. Em um país com descentralização fiscal forte, a integração entre as administrações tributárias é desejado devido ao alto custo público e privado que é gerado pela autonomia fiscal e tributária, além de passar para a população um Estado ineficiente, sem definição e custoso.

A integração e o compartilhamento de informações fiscais têm como objetivos aumentar o controle e fiscalização de informações tributárias entre as diferentes administrações do país, além de racionalizar e modernizar a administração fiscal e tributária nacional, visando redução de custos e a burocracia causada pelas diferenças formas de administração. O projeto das NF-e preveu o investimento em tecnologia e em sistemas de informação no comércio, melhorando o atendimento em unidades administrativas e modernizando parques tecnológicos.

O objetivo das NF-e foi implantar um modelo nacional para documento fiscal, que substituísse o modelo de emissão de documento em papel, possuindo assinatura digital do estabelecimento remetente para validar em vias jurídicas. Com a implantação do modelo de NF-e, ficou mais fácil acompanhar em tempo real as operações comerciais e significou uma grande melhora para a fiscalização de operações e prestações de tributos com base

no ICMS e IPI.

2.1.1 Estrutura de uma NF-e

A estrutura de NF-e no estado do Rio Grande do Sul é definida pela Sefaz-RS (SECRETARIA DA FAZENDA DO RIO GRANDE DO SUL, 2019), conforme apresentado na Figura 2.1. Portanto, não é padrão de um modelo nacional, pois cada UF tem seu próprio modelo de NF-e, podendo eles serem correlacionados ou não.

Figura 2.1 – Exemplo de Nota Fiscal Eletrônica.

PGL DISTRIBUICAO DE ALIMENTOS LTDA

CNPJ: 87.397.865/0019-40 Inscrição Estadual: 1090354913

Av. Presidente Vargas, 1976, SANTA MARIA, Santa Maria, RS

DANFE NFC-e - Documento Auxiliar da Nota Fiscal Eletrônica para Consumidor Final - Via Consumidor

NFC-e não permite aproveitamento de crédito de ICMS

Emissão normal

NFC-e nº: 4268 Série: 208 Data de Emissão: 26/03/2018 15:12:37

Consulte pela Chave de Acesso em https://www.sefaz.rs.gov.br/NFCE

CHAVE DE ACESSO

4318 0387 3978 6500 1940 6520 8000 0042 6812 6151 2136

Protocolo de Autorização: 143180353391586

CONSUMIDOR

Consumidor não identificado

Código	Descrição	Qtde	Un	Vl Unit	Vl Total
407410	FEIJAO PRETO FRITZ FRIDA 1KG	1	UN	5,49	5,49
33680	HORTI - TOMATE LONGA VIDA KG	0,5	KG	4,99	2,49
392380010	PAO FITPAN MULTIGRAOS 350G	1	UN	5,79	5,79
2040	FRUTI - BANANA PRATA KG	0,865	KG	4,29	3,71
447940	PADARIA - GAGETAO URUGUAIO PERUZZO KG	0,273	KG	12,49	3,40
372030	OVOS BRANCOS DZ	1	UN	4,29	4,29

Valor total R$	25,17
Valor descontos R$	0,00
FORMA PAGAMENTO	VALOR PAGO R$
Cartão de Débito	25,17

Versão XSLT: 1.10

Fonte: Retirado do site da SECRETARIA DA FAZENDA DO RIO GRANDE DO SUL (2019)

Visualmente, a NF-e é dividida em 6 blocos. O primeiro bloco, mostrado em 1, contém informações sobre o estabelecimento emissor da Nota Fiscal, com nome, CNPJ,

Inscrição Estadual e localização. No bloco 2 existem informações auxiliares, principalmente para Notas Fiscais que permitem aproveitamento de crédito de ICMS. O terceiro bloco possui informações únicas da NF-e, como número, série, data de emissão e a chave de acesso, que é usada para a verificação da NF-e no site da Secretaria da Fazenda do Rio Grande do Sul. No bloco abaixo está a identificação do consumidor, com apenas o seu CPF, caso o mesmo tenha aceitado se identificar na emissão da nota no estabelecimento.

O quinto bloco contém as informações dos produtos comprados, com seu código (código próprio dado por cada supermercado, sem relação com código de barra), descrição, quantidade, unidade (para separar produtos unitários e produtos com preço por Kg), valor unitário (valor total para produtos por Kg e para produtos em unidade) e o valor total por produto. O sexto bloco contém informações do pagamento da conta, o valor total, valor de desconto dado e a forma de pagamento.

Vale ressaltar, que apesar do modelo NF-e ter sido adotado nacionalmente, cada estado tem seu próprio modelo de NF-e e cada estabelecimento comercial controla elas de sua forma. Por isso, não existe padrão em produtos descritos em notas fiscais, podendo assim um mesmo produto ser catalogado de diferentes maneiras.

Figura 2.2 – Exemplo de descrições diferentes para um mesmo produto.

Codigo	Produto
311320	REFRI FANTA LARANJA ZERO PET 2L
00500049408	&FANTA LAR ZR PET
57172	REFRI FANTA LARANJA 2L

Fonte: Autor do Trabalho, retirado do site (NFSearch, 2018)

Como pode ser visto na Figura 2.2, não existe padrão na catalogação de um produto, podendo o mesmo produto ter código e descrição diferentes. Nesse quesito entra o objetivo do trabalho, um algoritmo que retorne um resultado satisfatório para o usuário, independente da maneira que os estabelecimentos cataloguem seus produtos.

2.2 ALGORITMO DE BUSCA

Um algoritmo de busca é qualquer algoritmo que resolva um problema de busca, ou seja, recuperar informações armazenadas dentro de alguma estrutura de dados, dada uma entrada, e entregar uma saída contendo sua solução, após um certo número de operações.

Algoritmos de busca são apresentados com um argumento "K", e o problema é encontrar qual registro armazenado na estrutura de dados buscada tem K como chave. Após a busca estar completa, existem duas possibilidades: Ou a busca é bem sucedida, tendo localizado o registro, seja ele único ou não, contendo K. Ou não teve sucesso, tendo determinado que K não é encontrado em lugar nenhum. É assim que Donald Knuth (KNUTH, 1998) descreve o conceito básico de algoritmos de busca. O algoritmo de busca apropriado depende da estrutura de dados que está sendo usada e também pode usar o conhecimento prévio sobre os dados que serão buscados. Algumas estruturas são construídas com o propósito de facilitar a busca, deixando os algoritmos de busca mais rápidos ou mais eficientes.

Existem diversos tipos de algoritmos de busca, e cada um deles possui uma estrutura diferente, por exemplo, uma Hash Table, que é uma estrutura de dados que associa entradas como chaves de pesquisa a valores. Seu objetivo é que a partir de uma chave simples dada como entrada, fazer uma busca rápida e obter o retorno desejado.

Neste trabalho não veremos um algoritmo de busca com foco em alguma estrutura de dados. Na aplicação criada para o trabalho, é usado um banco de dados em PostgreSQL com tabelas para cada campo das notas fiscais. O algoritmo pretendido tem como diferencial o uso de cálculo de distância de palavra, para prevenir erros simples de digitação e descrições diferentes, como foi visto na seção anterior.

2.3 DISTÂNCIA DE PALAVRA

Um algoritmo de distância de palavra é um algoritmo que compara duas *strings* e retorna a distância de edição entre elas. A distância de edição é dada pela comparação entre as duas *strings*. Cada modificação que se faz necessária na primeira *string* para se assemelhar à segunda *string*.

Em (YUJIAN; BO, 2007), quantificar a similaridade entre *strings* é um problema científico importante que tem atraído muito interesse porque as informações-chave podem ser expressas por sequências símbolicas em muitas aplicações, tais como recuperação de texto, processamento de sinais e biologia computacional. Apesar de serem pouco vistos em livros e artigos mais atuais, existem alguns métodos de métricas de *strings* muito conhecidos, entre eles o algoritmo da distância de Hamming e o algoritmo de Levenshtein.

2.3.1 Distância de Hamming

A distância de Hamming(HAMMING, 1950) é um algoritmo de cálculo de distância de palavra unicamente para *strings* que possuem tamanho igual, não sendo possível com-

parar *strings* de tamanhos diferentes. Existem diversas formas de usar um algoritmo desse estilo. Uma implementação simples consiste em primeiramente comparar o tamanho das duas *strings* desejadas, para verificar se o cálculo é possível, e em seguida comparar caractere por caractere, se são diferentes ou não, como mostrado no código da Figura 2.3.

Figura 2.3 – Código em Python do Algoritmo de Hamming

```
def hamming_distance(s1,s2):
    sum=0
    if len(s1) != len(s2):
        raise ValueError("Impossível calcular
                          com tamanhos diferentes")
    for char1,char2 in zip(s1,s2):
        if char1 != char2:
            sum+=1
    return sum
```

Fonte: Autor do trabalho

Devido a sua simplicidade, o algoritmo de Hamming compara apenas *strings* de tamanhos iguais, o que o torna inconveniente para o desejo do trabalho. Por exemplo, se formos comparar as *strings* “Hamming” e “Levenshtein” será retornada uma mensagem de erro, já que o tamanho da palavra “Hamming” é 7 e o tamanho da palavra “Levenshtein” é 11.

2.3.2 Distância de Levenshtein

Com a impossibilidade de usar a distância de Hamming, a distância de Levenshtein(LEVENSHTEIN, 1966) é a melhor opção para alcançar o objetivo do trabalho. A distância de Levenshtein, assim como a de Hamming, também é um algoritmo de cálculo de distância de palavra, mas diferentemente do outro, pode realizar esse cálculo com quaisquer *strings* dadas como entradas, independente do seu tamanho.

Na distância de Levenshtein, a distância é dada não pelas diferenças entre caracteres, mas sim pelo número de operações para transformar uma *string* ’s1’ em uma *string* ’s2’. Para explicação, “operações” seriam inserção, remoção ou substituição de um caractere na primeira *string*. Como exemplo, podemos considerar as palavras *kitten* e *sitting*. A distância de Levenshtein entre elas é de 3, ou seja, são necessárias 3 operações para

transformar *kitten* em *sitting*. As operações ditas são as seguintes:

1. Substituição do ’k’ de *kitten* por ’s’, transformando em *sitten*
2. Substituição do ’e’ de *sitten* por ’i’, transformando em *sittin*
3. Inserção de ’g’ em *sittin*, transformando em *sitting* e chegando na palavra desejada.

Existem vários métodos que o algoritmo de Levenshtein pode ser implementado. Um método que o algoritmo pode ser feito é o mostrado no código a seguir, usando um método iterativo, otimizado para memória, como apresentado no código da Figura 2.4.

Figura 2.4 – Código em Python do Algoritmo de Levenshtein

```
def minimumEditDistance(s1,s2):
    if len(s1)>len(s2):
        s1,s2 = s2,s1
    distances = range(len(s1)+1)
    for index2,char2 in enumerate(s2):
        newDistances = [index2+1]
        for index1,char1 in enumerate(s1):
            if char1==char2:
                newDistances.append(distances[index1])
            else:
                newDistances.append(1 + min((distances[index1],
                                             distances[index1+1],
                                             newDistances[-1])))
        distances = newDistances
    return distances[-1]
```

Fonte: Retirado do site Rosetta Code(Rosetta Code, 2019)

No cálculo da distância de Levenshtein, é sempre favorável que a *string* ’s1’ seja menor que a *string* ’s2’, por isso é feita a troca entre as duas no começo do código. Com isso é criado uma variável ’distance’ que recebe um intervalo de 0 ao tamanho de ’s1’ somado de 1, que é a distância máxima que a palavra pode ter. Em seguida, é feito um laço de repetição onde são verificadas as distâncias entre caracteres, sejam eles iguais ou não, para calcular quais operações são necessárias. Ao fim das operações, o último valor do intervalo contido em ’distance’, que é alterado ao fim de cada repetição do laço, é a distância de edição de ’s1’ para ’s2’.

Existe também uma versão do algoritmo de Levenshtein que retorna a *string* já corrigida para a busca, mas pode ser custoso devido ao tamanho que a *string* pode conter,

além de não ser possível usar a lógica de que 's1' precisa ser menor que 's2', já que a *string* que necessitaria de correção pode ser a maior.

2.4 TRABALHOS RELACIONADOS

Existem muitos serviços semelhantes ao proposto no trabalho em outros ramos de produtos e de negócio, como por exemplo o Buscapé[1], mostrado na Figura 2.5, uma captura de tela do site, que faz comparação entre preços de produtos e permite a compra pelo mesmo, além de ter muitas lojas oficiais que vendem produtos pelo próprio site. Porém o Buscapé oferece comparação entre produtos a serem comprados, não entre os preços de produtos já comprados, o que difere do objetivo da aplicação base para o algoritmo proposto.

Figura 2.5 – *Screenshot* do site do Buscapé.

Fonte: Retirado do site do Buscapé em 25.Jun.2019

Devido ao citado anteriormente sobre o Buscapé, ele não pode ser considerado um comparativo útil na questão de detecção de erros. Um trabalho que fez semelhante, usando também a distância de Levenshtein, foi o trabalho publicado em (FRANCISCO; AMBRóSIO, 2016). Este trabalho aborda o problema de plágio no ensino de programação introdutória no contexto de um sistema de administração e correção automática de listas de exercícios. A proposta apresentada combina uma estratégia de normalização com o algoritmo da distância de edição de Levenshtein.

O algoritmo de Levenshtein foi usado como um verificador de similaridades em códigos de alunos feitos em exercícios práticos. O algoritmo proposto recebe como entrada duas *strings* normalizadas (retirando comentários, declarações e espaçamentos de códigos) geradas a partir de dois arquivos de texto e retorna o número de operações neces-

[1] https://www.buscape.com.br/

sárias para transformar uma *string* na outra, ou seja, a distância. Como a detecção de plágio tenta verificar a similaridade entre os códigos, foi considerado que quanto menor for a distância entre o par de *strings* mais eles são similares.

Por sua vez, falando novamente do âmbito comercial, no estado do Rio Grande do Sul foi criado pela Receita Federal o aplicativo Menor Preço[2], mostrado na Figura 2.6, que armazena o conteúdo de NF-e recém emitidas pela receita e exibe os produtos com seu preço de compra e sua localidade, logo após do produto ter sido comprado. O Menor Preço se assemelha mais ao aplicativo usado, porém ele exibe informações mais simples e não oferece outras funcionalidades que existem ou que estão previstas para a aplicação existente.

Figura 2.6 – *Screenshot* do Aplicativo Menor Preço.

Fonte: Retirado da página do aplicativo na Play Store[3]

Tendo em vista os dois softwares relacionados, o algoritmo proposto no trabalho visa corrigir alguns erros de escrita e de reconhecimento de produtos. O Buscapé já possui um certo índice de acerto nas buscas e o mesmo usa cache para criar um gabarito de consultas para retorno das buscas. Mas como a aplicação NFSearch, que é a base do algoritmo, não tem tanta semelhança com o Buscapé, como citado anteriormente, então é possível focar apenas no Menor Preço. E como o Menor Preço é uma busca exata sobre um campo digitado, o algoritmo que será feito aqui visa retornar mais resultados que o aplicativo Menor Preço.

[2]https://receita.fazenda.rs.gov.br/conteudo/9896

2.5 ANÁLISE DO CAPÍTULO

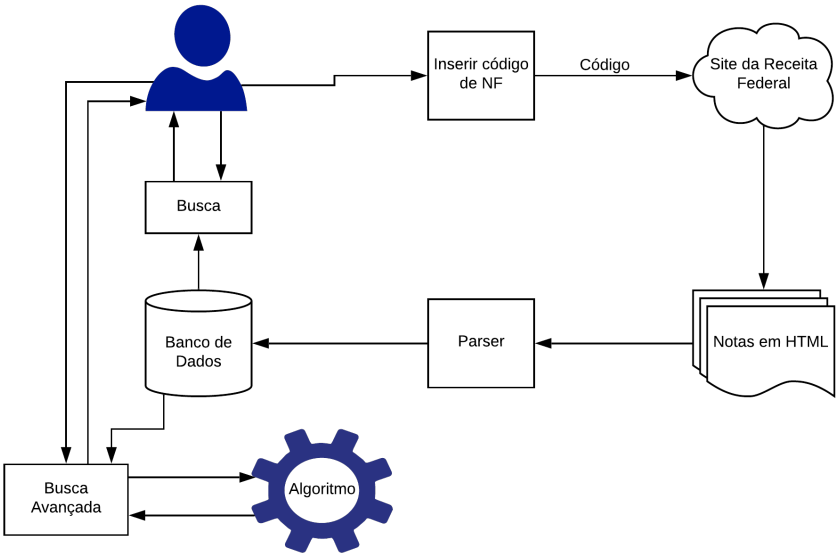
Com o que foi visto nesse capítulo, é correto afirmar que o algoritmo de Levenshtein é o melhor em comparação, devido ao fato do mesmo trabalhar com *strings* de diferentes tamanhos, para cumprir o objetivo do trabalho que é criar um algoritmo de busca que seja eficaz na busca para uma aplicação que use notas fiscais para popular o banco, uma vez que cada estabelecimento descreve os produtos de uma maneira diferente. Então, usando o conceito básico de um algoritmo de busca, apenas que realize busca em um banco de dados, e o algoritmo de Levenshtein, já existe uma base para atingir o resultado desejado.

3 METODOLOGIA E DESENVOLVIMENTO

Para cumprir os objetivos propostos no trabalho, primeiro é desejada uma aplicação base, que contenha informações existentes em NF-e salvas em um banco de dados, independente do seu tipo. A aplicação necessita de um *parser* que recolha as informações existentes no HTML de uma NF-e e o armazene no banco de dados.

A aplicação precisa que o usuário entre com o código da NF-e da sua compra. Então é feita uma requisição ao site da Sefaz-RS que retorna uma página HTML com o conteúdo da NF-e. O HTML é trabalhado por um parser que pega as informações dos produtos e do estabelecimento. Essas informações são salvas em uma base de dados que fica pronta para quando o usuário quiser buscar seus produtos.

Figura 3.1 – Arquitetura do NFSearch.



Fonte: Autor do Trabalho

Conforme apresentado na Figura 3.1, o algoritmo proposto está relacionado com as etapas de busca feita pelo usuário, na parte do algoritmo que seleciona as respostas a serem retornadas. Um mesmo usuário pode inserir códigos de NF no sistema, que armazenará no banco de dados o conteúdo delas, e também pode realizar buscas neste mesmo banco de dados, em busca do produto desejado. Esta parte representada no Fluxograma é o NFSearch, que será tratado na próxima seção. Este trabalho foca em aperfeiçoar o NFSearch e dar ao usuário uma busca avançada, cujo retorno de produtos será mais completo do que a busca comum do NFSearch.

3.1 NFSEARCH

Visando o uso de NF-e emitidas pelo comércio, com o objetivo de criar uma aplicação de consulta de preço, foi criada a aplicação NFSearch. A NFSearch [1] consiste em uma aplicação web com um banco de dados contendo informações sobre produtos comprados por usuários em estabelecimentos que emitem NF-e. Como ideia básica da aplicação, o banco deve ser sempre preenchido pelos próprios usuários, com leitura do QRCode existente na versão física de uma nota fiscal, tendo assim uma leva de informações entre a população consumidora. A aplicação foi desenvolvida em Python, com o framework Django e hospedado na plataforma em nuvem Heroku.

Para extrair as informações das NF-e, foi feito um parser HTML, com o auxílio da biblioteca Beautiful Soup(Bs4 Documentation, 2015), uma biblioteca para Python que salva o HTML passado para ela localmente e permite a manipulação e a busca de informações de uma maneira simples e intuitiva. O parser foi feito com base nas *tags* HTML que existem dentro de uma NF-e e principalmente os atributos de cada uma dessas *tags*.

Figura 3.2 – Trecho de código HTML referente aos produtos na NF-e

```
<table class="NFCCabecalho" border="0">
    <tr>
        <td class="borda-pontilhada-3D NFCDetalhe_Item">Código</td>
        <td class="borda-pontilhada-3D NFCDetalhe_Item">Descrição</td>
        <td class="borda-pontilhada-3D NFCDetalhe_Item">Qtde</td>
        <td class="borda-pontilhada-3D NFCDetalhe_Item">Un</td>
        <td class="borda-pontilhada-3D NFCDetalhe_Item"">Vl Unit</td>
        <td class="borda-pontilhada-top-botton">Vl Total</td>
```

Fonte: Sefaz-RS(RECEITA FEDERAL, 2019)

Vale ressaltar, como pode ser visto no código da Figura 3.2, que uma mesma *tag* pode representar mais de uma parte da NF-e, por esse motivo que os atributos de cada *tag* são importantes.

O parser foi feito com base na árvore do HTML, verificando se a *tag* onde ficam as informações dos produtos adquiridos é filha da *tag* com atributo *class* que valor "NFC-Detalhe_Item". Ao encontrar as informações desejadas, o passo seguinte é extrair as informações contidas no conteúdo no HTML, com exceção do preço que levava em conta a quantidade de produtos comprados, que não é necessária para a aplicação.

Outro tipo importante de informações são as informações sobre a localização do estabelecimento, mostradas na Figura 3.3, com estado, cidade e o endereço dentro da

[1] https://nfsearch.herokuapp.com

cidade. As informações estão armazenadas dentro de uma *tag* “td”, com o atributo *class* com valor de "NFCCabecalho_Subtitulo1". A *tag* “td” é filha da *tag* “tr” e da *tag* “table” com o atributo *class* com valor “NFCCabecalho”. Caso o parser encontre essa sequência de *tags*, o banco de dados recebe o endereço do supermercado e o associa ao produto.

Figura 3.3 – Trecho de código HTML referente ao estabelecimento na NF-e

```
<table class="NFCCabecalho" border="0">
    <tr>
        <td class="NFCCabecalho_SubTitulo1" align="left">
        Av. Presidente Vargas,
        1976,
        SANTA MARIA,
        Santa Maria,
        RS</td>
    </tr>
</table>
```

Fonte: Sefaz-RS(RECEITA FEDERAL, 2019)

Com a aplicação criada, foram pensados problemas na hora do usuário buscar o produto desejado, como por exemplo, erros de digitação, descrições muito diferentes para um mesmo produto e o tipo de cada produto não estando descrito na NF-e. Para minimizar os erros que foi planejado o algoritmo que é o objetivo no texto: um algoritmo de busca que faça uso de métricas de distância de palavra para aproximar o resultado da busca por produtos mais precisa.

3.2 NECESSIDADE DE APRIMORAMENTO NA BUSCA DO NFSEARCH

Uma vez as notas fiscais obtidas a partir do site da Secretaria da Fazenda estiverem armazenadas no banco de dados da aplicação, conforme descrito na seção anterior, é possível a realização de buscas diretamente sobre esta base.

Entretanto, algumas particularidades nas possíveis buscas devem ser levadas em consideração, tais como: erros de escrita pelo usuário no campo de busca, resultados de comparações inadequados (como o mesmo produto cadastrado em estabelecimentos distintos com nomes diferentes), uso de abreviações na nomenclatura dos produtos e problemas com descrições incompletas de produtos em notas fiscais. Isso se deve, em parte, por não haver um padrão na forma de descrever cada produto em uma nota fiscal, pois cada estabelecimento faz uma descrição à sua maneira.

Figura 3.4 – Duas descrições diferentes para um mesmo produto.

Codigo	Produto
74680	REFRI PEPSI TWIST PET 2L
00500253573	&PEPSI TWIST

Fonte: (NFSearch, 2018)

Como pode ser visto na Figura 3.4, um mesmo produto, que por ser vendido em dois estabelecimentos distintos, está com duas descrições diferentes. Agora, considerando que fosse feita uma busca simples por “Refri” no NFSearch, o segundo item “&PEPSI TWIST” não seria retornado, o que não é desejável no contexto da aplicação, já que se espera que o consumidor tenha acesso a todas as variações possíveis de um produto buscado. Outra busca que causaria problemas no contexto mostrado seria uma busca pela palavra "Refrigerante", que não mostraria nenhum dos dois produtos usados como exemplo na Figura 3.4.

Além desses erros de caso, existe também a necessidade de corrigir alguns erros de escrita do usuário. Tendo esse tipo de problema em mente e usando os conhecimentos de comparação de *strings* mostrados anteriormente, é possível esboçar o algoritmo que é o objetivo do trabalho, o qual está descrito na seção seguinte.

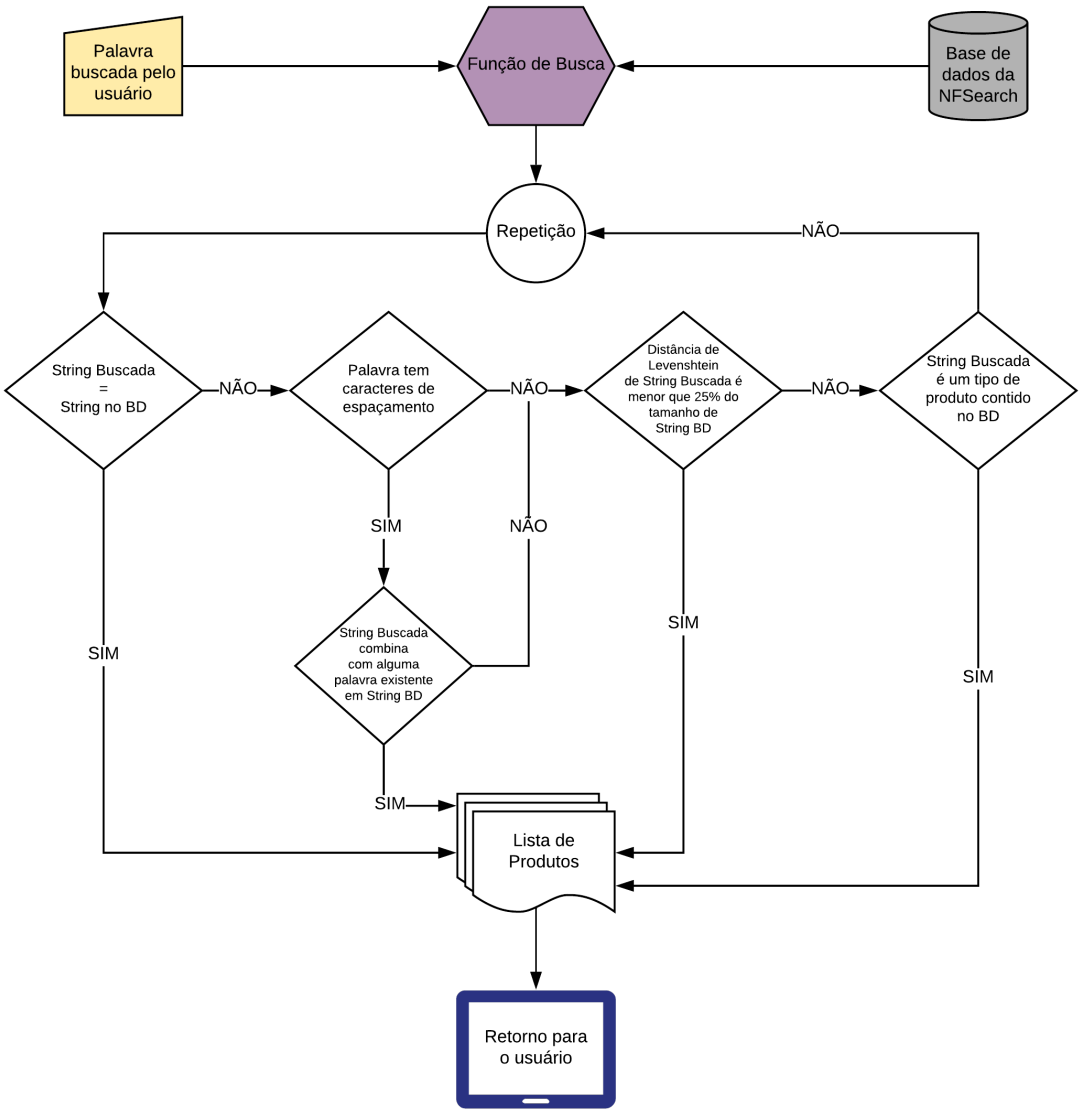
3.3 CRIAÇÃO DO ALGORITMO

Na Figura 3.5 vemos como é o funcionamento do algoritmo proposto no trabalho por meio de um fluxograma. O algoritmo depende da entrada de dados do usuário e das palavras armazenadas no banco de dados da aplicação previamente, e em seguida entra em um laço de repetição, que só acaba quando já foram verificadas todas as palavras do banco de dados.

Dentro do laço de repetição existem 4 estruturas de seleção, a primeira verifica se a palavra pesquisada é igual a palavra contida no banco de dados (Seção 3.3.1). Caso negativo, a segunda verificação conta os espaços da palavra no banco para verificar se é uma palavra composta e logo em seguida faz comparação palavra por palavra procurando combinações (Seção 3.3.2). Novamente, caso negativo, a próxima estrutura verifica se o cálculo de Levenshtein entre a palavra buscada e a palavra contida no banco é menor ou igual ao tamanho da palavra no banco (Seção 3.3.3). Em caso de nova negativa, a ultima

selecao verifica se o tipo do produto buscado combina com o que esta no banco de dados (Seção 3.3.4).

Figura 3.5 – Fluxograma do Algoritmo.



Fonte: Autor do Trabalho

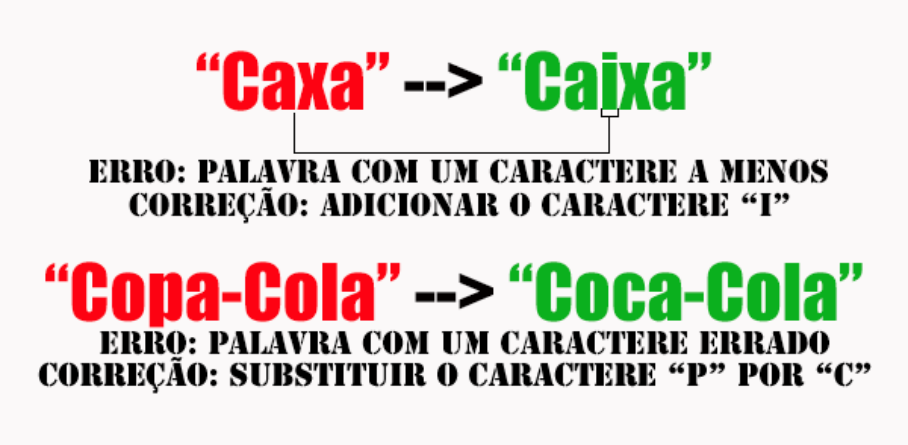
Os processos apresentados na Figura 3.5 estão descritos nas subseções seguintes.

3.3.1 Erros de escrita pelo usuário no campo de busca

Com os problemas possíveis já listados e expostos, o algoritmo já possui uma série de itens e fatores que precisam ser corrigidos para uma busca satisfatória. Como ponto de partida será tomada a correção de erros gramaticais simples.

Para esta parte do algoritmo, foi pensado que tipos de erros de escrita deveriam ser priorizados. Erros muito complexos causariam uma penalidade não desejável ao algoritmo, por isso, na parte de erros de escrita seriam considerados apenas casos simples, como por exemplo a falta de caracteres no momento da digitação ou troca de caracteres no momento da busca.

Ilustração 3.1 – Representação visual dos erros de escrita previstos pelo trabalho.



Fonte: Autor do Trabalho

Como foi exemplificado na Figura 3.1, alguns erros simples de escrita precisam de apenas uma operação para serem corrigidos, ou seja, tem distância de Levenshtein de 1. Por isso, considerando algumas *strings* que foram vistas e estudadas, foi montado um método de reconhecimento de erro e a correção.

Para ser feito, será considerada uma distância de Levenshtein máxima de 25%, para corrigir erros simples. Com essa limitação, poderão ser corrigidos os erros e evita chances de pegar palavras muito distintas do desejado. Com as informações necessárias e a limitação definida, já existe a possibilidade de escrever um algoritmo para buscar uma palavra e corrigi-la ou ver se ela tem semelhança com alguma palavra no banco. Vale ressaltar que as *strings* que servem como comparação a *string* buscada devem ser as *strings* salvas no banco de dados da aplicação.

Algoritmo 1: VERIFICAÇÃO DE ERROS SIMPLES

```
1 início
2     se levenshtein(StringBuscada) <= (tamanho(StringBanco)/4) então
3         retorna StringBanco
4     fim
5 fim
```

Com esse algoritmo podemos comparar as duas palavras na função “levenshtein()”, que retorna a distância de edição de Levenshtein entre as duas *strings*, e verificar se uma *string* é menor que 25% do tamanho da outra *string*. Caso seja, a *string* que está no banco, ou seja, a *string* de informações do produto que foi buscado, será retornado como resultado.

Com o erro mais simples e direto sendo tratado, podemos agora seguir para alguns erros mais específicos, que podem ser tratados de maneira semelhante e de fácil entendimento.

3.3.2 Reconhecimento de Abreviações

O passo seguinte no reconhecimento é reconhecer abreviações de palavras, sejam elas abreviações ou “apelidos” de uma palavra. Usando como exemplo o retorno esperado de "Refrigerante ABC 2L", caso o usuário procure apenas por "refri", não encontrará nem pelo cálculo simples da distância de Levenshtein nem pelo uso da verificação de *string* composta, que entre “Refri” e “Refrigerante” é de 7, um resultado que é configurado como um erro grande e não é considerado uma combinação.

Algoritmo 2: VERIFICAÇÃO DE ABREVIAÇÃO

início
  se *Último caractere de* $levenshtein(StringBuscada) =$ ”.” **então**
    Remove o caractere ”.”
  fim
  se $tamanho(StringBuscada) > tamanho(StringBanco)$ **então**
    $StringBuscada \leftarrow StringBanco$
    $StringBanco \leftarrow StringBuscada$
  fim
  se $StringBuscada \in StringBanco$ **então**
    retorna *True*
  senão
    retorna *False*
  fim
fim

A função proposta verifica primeiramente, na linha 3, se a abreviação possui o caractere ".". Caso haja remove “.” e faz uma nova verificação, na linha 5, de se a palavra buscada é maior que a *string* contida no banco de dados. Sendo maior, é feita uma troca de valores entre as duas para fazer a comparação correta. Com isso, pode ser enfim verificado se existe uma abreviação na NF-e ou na busca. A comparação, considerando que é uma abreviação e não uma sigla com diminuição, como por exemplo "RFRGNT"para tratar de "Refrigerante", é feita de maneira simples de se “StringBuscada” está contida em “StringBanco” e dá retorno caso seja verdadeiro ou não.

3.3.3 Manipulação de Palavras Compostas

Com os erros simples de escrita e abreviações com retorno correto, o próximo passo é solucionar uma espécie de problema semelhante. Caso o usuário esteja buscando de uma maneira simples alguma descrição que seja complexa de uma NFe, como por exemplo, caso o usuário busque por "ABC"e no banco de dados haja a informação "Refrigerante ABC 2L", com a comparação simples feita na seção 3.3 a busca não retornará sucesso, mas o produto é o desejado pelo usuário.

Visando solucionar tal problema, seria benéfico dividir a *string* contida no banco de dados por cada palavra contida nela e comparar com a *string* buscada. Para isso, é feita uma verificação se a palavra no banco possui espaços ou não.

Algoritmo 3: CONTAGEM DE ESPAÇOS

```
1 início
2     Contador = 0
3     para cada Char ∈ String faça
4         se Char = então
5             Contador ← Contador + 1
6         fim
7     fim
8 fim
9 retorna Contador
```

Com o algoritmo para a contagem de espaços, podemos usá-la para verificar se uma palavra é ou não composta no algoritmo para a solução do problema.

Algoritmo 4: COMBINAÇÕES EM PALAVRA COMPOSTA

```
1  início
2      Combinacoes = 0
3      Lista = []
4      se ContagemEspacos(StringBanco) = True então
5          Lista = palavras contidas em StringBanco
6      fim
7      para cada Palavra ∈ Lista faça
8          se levenshtein(StringBuscada, Palavra) <= (tamanho(Palavra)/4) então
9              Combinacoes ← Contador + 1
10         senão
11             se VerificacaoAbreviacao(StringBuscada, Palavra) = True então
12                 Combinacoes ← Contador + 1
13             fim
14         fim
15         se Combinacoes > 0 então
16             retorna True
17         senão
18             retorna False
19         fim
20     fim
21 fim
```

No algoritmo representado a função, apresentada anteriormente no Algoritmo 3, “ContagemEspacos(StringBanco)”, na linha 4, verifica se a *string* contém caracteres de espaçamento. Caso haja salva as palavras de “StringBanco” em uma lista, nenhuma das palavras contendo espaçamento. Com isso, enquanto a lista não estiver vazia, são feitas verificações vistas anteriormente, como a de erros comuns de levenshtein (na linha 8), citada na sub-seção 3.3.1 e a verificação de abreviações (na linha 11), vista em 3.3.2. Caso alguma das verificações aconteça, é incrementado o valor do contador de combinações. Caso o contador possua valor superior a zero, retorna *True*, ou seja, a palavra composta possui combinações com o que foi buscado. Caso o contador esteja zerado, o retorno é falso.

3.3.4 Reconhecimento de Tipos de Produtos

A última parte do algoritmo de busca desejado é o reconhecimento de tipos dos produtos buscados pelo usuário. O tipo de um produto, para título de conhecimento, é a categoria do produto, como por exemplo, uma esponja de aço catalogada na nota fiscal como "Marca de Esponja Unidade". Caso o usuário faça a busca apenas por "esponja de aço", é desejado que no retorno da busca, a "Marca de Esponja unidade"apareça na busca. Para contornar esse problema, foi pensada numa solução baseada em uma tabela de banco de dados, considerando que o contexto do trabalho é uma aplicação que usa banco de dados.

Algoritmo 5: RECONHECIMENTO DE TIPO DE PRODUTO

```
1  início
2      se StringBuscada ∈ tabela de tipo de BancoDados então
3          retorna True
4      senão
5          se VerificacaoAbreviacao(StringBuscada, Palavra em
             BancoDados) = True então
6              retorna True
7          fim
8      fim
9      retorna False
10 fim
```

A função proposta no Algoritmo 5, de comparação de tipos de produtos, verifica o banco de dados e compara "StringBuscada"com a tabela de tipo do banco de dados (linha 5). É feita uma verificação simples de semelhança, já que se espera que o tipo seja uma palavra pequena, então não há necessidade de verificar erros e uma verificação de abreviações, para evitar quaisquer perda de informação no retorno. Caso a palavra passe em qualquer um dos dois testes, confirma que a palavra que foi buscada é do tipo do produto verificado, contido no banco de dados.

Vale frisar que para esse método funcionar, no momento que o banco de dados for criado, é necessário que os desenvolvedores preencham a tabela de tipo manualmente, da maneira mais simples e objetiva possível para descrever o produto. Já que numa aplicação dessa espécie, de comparação de preços, é necessária uma carga inicial antes de liberar ao público comum, é plausível que os tipos sejam adicionados manualmente. Além disso, os resultados gerados dependem inteiramente de como o desenvolvedor catalogar o produto.

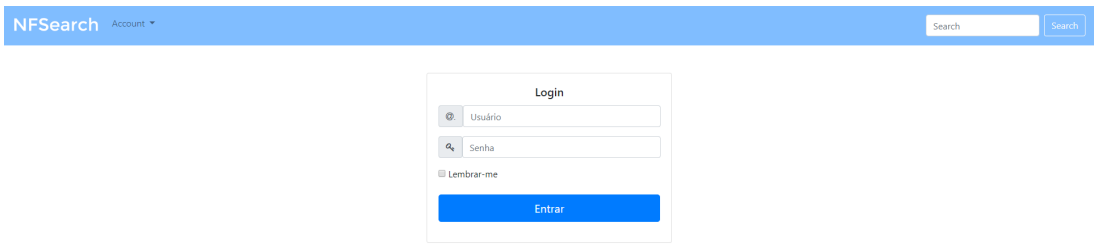
4 ESTUDOS DE CASO

Com o algoritmo pronto, para concluir a tese é desejável uma série de experimentos para que seja possível validar o algoritmo que foi proposto, além de ver se os resultados desses experimentos obtiveram um retorno melhor do que uma busca comum. Para esses experimentos, são consideradas descrições de NF-e reais, com palavras que é sabido que podem dar erro ou retornos incompletos em uma busca comum. As palavras buscadas também estão relacionadas, para que os experimentos sejam direto ao ponto e mais explicativos que for possível para provar o que foi proposto.

Para realizar os experimentos, foi planejada uma listagem de produtos contidos em notas fiscais reais, de diferentes redes de supermercados e de produtos variados, para tornar os experimentos o mais próximo possível da realidade. Os produtos usados para ser feito o experimento estão todos contidos em (NFSearch, 2018), somando um total de 196 produtos.

Os experimentos foram feitos no software NFSearch, citado na Seção 3.1, executado localmente as mudanças propostas no trabalho.

Figura 4.1 – Tela inicial da versão web do NFSearch.



Fonte: (NFSearch, 2018)

Com as palavras para serem buscadas, o passo seguinte é escolher palavras que melhor se adéquam aos produtos existentes na base de dados. Considerando palavras que atinjam o melhor resultado possível, que provem a veracidade de cada teste e que englobem um número significativo de retornos, chegamos na seguinte lista:

- Pepsi
- Chocolate
- Escova de Dentes

Com esses três itens sendo buscados já é possível ter uma lista de retorno que prove o algoritmo que foi proposto. Em cada experimento será especificado qual das soluções propostas que gerou o resultado retornado e uma explicação sobre o motivo de terem

sido selecionadas. É importante saber que nos testes é ignorado se as palavras usadas estão em caixa alta ou em caixa baixa.

4.1 EXPERIMENTO 1 - *STRING* PEPSI

Neste experimento será testada a palavra “Pepsi” na amostra dada, com resultados na Tabela 4.1 abaixo.

Produto	Retorno Comum	Retorno com o Algoritmo
REFRI PEPSI TWIST PET 2L	✓	✓
&PEPSI TWIST		✓

Tabela 4.1 – Resultados dos Experimentos Realizados com a Palavra "Pepsi"

Na busca pela palavra "Pepsi", notou-se que o primeiro retorno seria dado por uma busca comum, verificando se a palavra buscada está na palavra verificada. Já o segundo retorno não seria garantido em uma busca comum, já que o caractere & na frente de “Pepsi” poderia causar erros de reconhecimento caso fosse feita apenas uma busca por semelhança. Nota-se também que o que causou o retorno de “&Pepsi Twist” foi o cálculo de distância de Levenshtein, mostrado na Seção 3.3.1, que verificou apenas a diferente no "&", ou seja, distância de edição com valor igual a 1, que é menor que 25% do tamanho da palavra "Pepsi", que é 1,2.

O retorno ocorrido na aplicação NFSearch pode ser visto na Figura 4.2 mostrada a seguir

Figura 4.2 – Retorno da Palavra "Pepsi".

Fonte: (NFSearch, 2018)

4.2 EXPERIMENTO 2 - *STRING* CHOCOLATE

No experimento desta seção será testada a palavra "Chocolate", na Tabela 4.2 abaixo.

Produto	Retorno Comum	Retorno com o Algoritmo
ALFAJOR EL AGUILA CHOCO BR		✓
CHOCOLATE SNICKERS 100 CAL	✓	✓
TABLETE CHOC SHOT AMENDOIM 135G LACTA		✓
MOUSSE KIBON CHOCOLA		✓
CHOC MINI TAB NESTLE		✓
CHOC TAB DIVINE COOK		✓
CHOC MINI DIVINE LEIT		✓

Tabela 4.2 – Resultados dos Experimentos Realizados com a Palavra "Chocolate"

Neste experimento foi usada a palavra “Chocolate”, que gerou sete retornos, exibidos na Tabela 4.2. O primeiro retorno foi dado pela verificação de abreviações, descrito na Seção 3.3.2, já que ao dividir a *string* “ALFAJOR EL AGUILA CHOCO” ocorre a comparação entre “Chocolate” e "CHOCO". “CHOC” está contido em “Chocolate”, portanto, “CHOCO” é considerada uma abreviação da palavra “Chocolate”.

O segundo retorno ocorre somente pela verificação de palavra composta, presente na Seção 3.3.3, comparando “Chocolate” com “Chocolate” após dividir a *string* existente do banco de dados.

Os demais retornos também ocorre pelo mesmo motivo do primeiro. A verificação de abreviação ocorre após a divisão da *string* contida no banco de dados e a comparação entre “Chocolate” e "CHOC", ou “CHOCOLA”, no experimento com “MOUSSE KIBON CHOCOLA”.

O retorno ocorrido na aplicação NFSearch pode ser visto na Figura 4.3 mostrada a seguir

Figura 4.3 – Retorno da Palavra "Chocolate".

Fonte: (NFSearch, 2018)

4.3 EXPERIMENTO 3 - *STRING* ESCOVA DE DENTES

Na Tabela 4.3 a seguir estão testes com a palavra "Escova de Dentes"

Produto	Retorno Comum	Retorno com o Algoritmo
ESC DENTAL COLGAT		✓

Tabela 4.3 – Resultados dos Experimentos Realizados com a Palavra "Escova de Dentes"

Neste caso exibido há a verificação por tipo acontecendo, entrando na condição que foi colocada no capítulo 3, sub seção 3.3.4, de que o retorno nesses casos dependeria de como o produto é catálogado. No experimento em foco, o produto “ESC DENTAL COLGAT” é mais corretamente catalogado como "Escova de Dentes".

Considerando isso, foi feita a comparação entre a palavra buscada e o tipo de produto e portanto o retorno foi dado. Levando em conta o caso específico dessa amostra de 30 produtos, o retorno do mesmo produto também poderia ser dado na parte da avaliação de abreviatura, já que a comparação entre “ESC” e “Escova de Dentes” reconheceria “ESC” como abreviação, o que na amostra dada não seria problemático, mas em uma gama maior de produtos, pode causar retornos errados.

O retorno ocorrido na aplicação NFSearch pode ser visto na Figura 4.4 mostrada a seguir

Figura 4.4 – Retorno da Palavra "Escova de Dentes".

Fonte: (NFSearch, 2018)

4.4 RESULTADOS FINAIS

Com os experimentos realizados, comprovou-se que o algoritmo proposto no trabalho consegue ofertar um resultado mais preciso do que uma busca comum, como exibido a seguir.

Busca	Retorno Comum	Retorno com o Algoritmo
Pepsi	1	2
Chocolate	1	7
Escova de Dentes	0	1

Tabela 4.4 – Resultados dos Experimentos Realizados

Apesar de os resultados para a amostra usada terem sido satisfatórios, algumas possíveis atualizações e aperfeiçoamentos foram notados durante os experimentos e durante o desenvolvimento.

5 CONCLUSÃO

Neste trabalho foi explicado um algoritmo que se propõe a solucionar alguns problemas de retorno que podem ocorrer em uma aplicação feita com base em NF-e. Para alcançar esse objetivo, foram vistas métricas de *strings* para medir a distância de edição entre duas *strings* dadas para a função. Com essas informações foi possível planejar como solucionar os problemas de retorno.

O algoritmo proposto para realizar o objetivo se baseou em quatro problemas encontrados durante o estudo de caso: erro de digitação do usuário no campo de busca, abreviações usadas para referenciar produtos, erros de combinação em *strings* compostas e reconhecimento de tipos de produtos.

Com a realização desse algoritmo, foi possível testá-lo e comprovar que ele fornece uma contribuição para o problema descrito, apesar de não ser a melhor possível. Por este motivo foram pensadas eventuais maneiras de melhorar o algoritmo e torná-lo mais automático possível. Com esse objetivo, a maior alteração planejada seria na parte de verificação de tipo. A ideia é de remover a adição manual nas tabelas de tipo do banco de dados e usar técnicas de reconhecimento de padrões(MARQUES, 1999) com uso de redes neurais para que o NFSearch faça o reconhecimento automático do tipo de produtos recém cadastrados, tornando assim o algoritmo mais preciso e mais eficaz possível, evitando inclusive erros dos desenvolvedores nos tipos dos produtos.

Além do tipo de produtos, um método menos suscetível a erros no momento do reconhecimento de abreviações seria benéfico, já que o atual, apesar de eficaz como foi visto, pode dar retornos errados ao usuário. Dando de exemplo alguma palavra no banco de dados que possua a *sub string* “de” pode ser facilmente interpretada como abreviação de alguma palavra que apenas possua “de” em seu corpo. Um exemplo desse ocorrido seria na pesquisa por “Detergente”, que teria retorno errado caso no banco haja uma *string* semelhante a "Pasta de Dente", será dado um retorno errado ao usuário, o que não é de forma alguma desejado.

As demais alterações previstas são na aplicação NFSearch em si e não no algoritmo de busca, já que para o produto em que foi feito, ele é considerado satisfatório.

REFERÊNCIAS BIBLIOGRÁFICAS

AQUINO, C. A. B.; MARTINS, J. C. de O. Ócio, lazer e tempo livre na sociedade do consumo e do trabalho. **Revista SUBJETIVIDADES**, v. 7, n. 2, 2007.

Bs4 Documentation. **Beautiful Soup Documentation**. 2015. Acessado em 13 jul. 2019. Disponível em: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.

FRANCISCO, R. E.; AMBRóSIO, A. P. Uso do algoritmo distância de edição com técnicas de pré-processamento para apoiar a identificação de plágio em códigos-fonte de problemas de programação introdutória. **Revista Brasileira de Sistemas de Informação**, v. 9, n. 2, 2016.

HAMMING, R. Error detecting and error correcting codes. **The Bell System Technical Journal**, v. 29, n. 2, 1950.

KNUTH, D. **The Art of Computer Programming**: Sorting and searching. 2. ed. [S.l.]: Addison-Wesley, 1998. v. 3, 780 p.

LEVENSHTEIN, V. Binary codes capable of correcting deletions, insertions, and reversals. **Doklady Akademii Nauk SSSR**, v. 10, n. 8, 1966.

MARQUES, J. S. **Reconhecimento de Padrões**: Métodos estatísticos e neuronais. 2. ed. [S.l.: s.n.], 1999.

NFSearch. **NFSearch**. 2018. Acessado em 13 jul. 2019. Disponível em: <https://nfsearch.herokuapp.com/>.

RECEITA FEDERAL. **Portal da Nota Fiscal Eletrônica**: Sobre a nf-e. Secretaria da Fazenda, 2019. Acesso em 29 abr. 2019. Disponível em: <http://www.nfe.fazenda.gov.br/portal/sobreNFe.aspx?tipoConteudo=HaV+iXy7HdM=>.

Rosetta Code. **Levenshtein Code**. 2019. Acessado em 11 mai. 2019. Disponível em: <http://rosettacode.org/wiki/Levenshtein_distance#Python>.

SECRETARIA DA FAZENDA DO RIO GRANDE DO SUL. **Sefaz RS - Nota Fiscal Eletrônica - Consultas**: Nota fiscal do consumidor (nfc-e). Receita Estadual, 2019. Acesso em 29 abr. 2019. Disponível em: <https://www.sefaz.rs.gov.br/NFCE/NFCE-COM.aspx>.

YUJIAN, L.; BO, L. Normalized levenshtein distance metric. **IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENC**, v. 29, n. 6, p. 1091–1095, 2007.

APÊNDICE A – CÓDIGO FONTE DO ALGORITMO AVANÇADO DE BUSCA

```
#Função contadora de espaços, para verificar
#se uma string é composta ou não
def count_spaces(s1):
    count=0
    for c in s1:
        if c == " ":
            count+=1
    return count
#Função para o cálculo de Levenshtein
def levenshtein(s1, s2):
    if len(s1) < len(s2):
        return levenshtein(s2, s1)
    # len(s1) >= len(s2)
    if len(s2) == 0:
        return len(s1)
    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1
            deletions = current_row[j] + 1
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row
    return previous_row[-1]
#Verifica se uma string é composta ou não
def string_compose(s1, s2):
    count=0
    s2 = s2.split()
    for words in s2:
        if levenshtein(s1, words) <= (len(words)/2):
            count+=1
        elif verifi_abrev(s1, words):
            count+=1
    if count > 0:
        return True
    else:
        return False
#Verifica se há ou não abreviações em strings
```

```
def verifi_abrev(s1, s2):
    if s1[-1:] = ".":
        s1 = s1[:-1]
    if len(s1) > len(s2):
        s1,s2 = s2,s1
    if s1 in s2:
        return True
    else:
        return False
#Realiza comparação de tipos de produtos
def type_comp(s1, db):
    if s1 in db['tipo']:
        return True
    elif verifi_abrev(s1, db['produto']):
        return True
    else:
        return False
def recognizer(s1, db):
    l = []
    for line in db:
        if s1 == line['produto']:
            l.append(line['produto'])
        #Seção 3.3, sub seção 3.3.3
        elif count_spaces(line['produto']) > 0:
            if string_compose(s1, line['produto']):
                l.append(line['produto'])
        #Seção 3.3, sub seção 3.3.1
        elif levenshtein(s1, line['produto']) <= (len(line['produto'])/2):
            l.append(line['produto'])
        #Seção 3.3, sub seção 3.3.4
        elif type_comp(s1, line):
            l.append(line['produto'])
    return l
```