“Oohay”: Indexação e busca em arquivos utilizando a estrutura de dados Árvore de Prefixos

Francisco Maradona de F. Morais[1], Daniel Marx Pinto Carvalho[2]
[1,2] Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (UFRN)
Natal – RN – Brasil

mrmorais@ufrn.edu.br,danielmarx@ufrn.edu.br

Abstract. *This paper shows a developed system to deal with indexing and searching problems, by using R-way prefixed tree. The approach used to solve the given problem is the minimization of memory and processing costs in a computational system. In addition to explore the applieds concepts to solve the problems, also is made a thinking about the proposed solution, looking over the pros, cons and alternatives that could be explored to apply in this kind of problems, that are similar to the adopted approach.*

Resumo. *Este trabalho apresenta os processos para desenvolvimento de um sistema de indexação e busca em árvore de prefixos. O problema apresentado é abordado a partir da aplicação de estruturas que visam a economia de recursos em memória e processamento de um sistema computacional. Além de visitar o conceito adotado para resolver o problema de indexação e busca, também é feita uma reflexão sobre a solução proposta apontando os prós, contras, e outras alternativas que poderiam ser avaliadas para uma solução similar à conhecida.*

1. Introdução

A indexação e busca são duas operações fundamentais em grandes sistemas de informação. Diariamente, por exemplo, utilizamos motores de busca como Google e Bing que conseguem retornar informações relevantes com muito pouco tempo de espera (milésimos de segundos, a maioria das vezes). Por trás destas ferramentas existem diversas estruturas organizadas para lidar com a grande massa de informações, e também para indexar páginas da internet a fim de torná-las pesquisáveis através de seus serviços. A técnica chamada *web crawling* utilizada pelo Google, por exemplo, utiliza softwares para visitar páginas na internet e indexar o conteúdo dessas páginas nos servidores da companhia [Google 2017]. A busca, por outro lado, é o processo de extrair dados dentro da base de conteúdo indexado.

O projeto “Oohay”, apresentado neste trabalho, tem como intenção desenvolver um sistema de indexação de arquivos baseado em estrutura de árvore e realização de busca otimizada de palavras na base de arquivos indexados. Ao fluxo de funcionamento da aplicação compete realizar tarefas de (a) inserção de um arquivo na base, indexando todas as palavras do arquivo e (b) realizar busca por termos e retornar os arquivos e linhas em que a palavra aparece.

A indexação de palavras é realizada pelo componente *Indexador*, que dado um arquivo (ou um caminho para um arquivo) gera uma lista de todas as palavras que aquele arquivo possui, posteriormente essas palavras são adicionadas à árvore de armazenamento.

O processo de busca é feito através do percorrimento na árvore de palavras que possui uma estrutura otimizada para obtenção do resultado minimizando o custo da operação.

O sistema possui também alguns requisitos paralelos que intensificam a experiência do usuário, tais como suporte a *blacklist*, uma lista de palavras que o usuário deseja que não sejam indexadas (estas palavras são repassadas para o sistema através de um arquivo de

texto); os dados da sessão do usuário é persistido, ou seja, ao encerrar e inicializar o software o usuário não precisará selecionar novamente os arquivos para indexação.

Outrossim, o software possui uma camada gráfica de interação com o usuário, através da qual serão inseridos e listados arquivos, resultados de busca, etc.

2. Funcionalidades

O sistema funciona com base em duas categorias bem definidas, que de modo geral são responsáveis por interpretar palavras de arquivos salvando as indicações do nome dos arquivos em que elas se encontram, as linhas em que aparecem em cada arquivo, e o número de aparições da palavra em cada uma dessas linhas, permitindo que posteriormente um usuário que venha a buscar alguma palavra contida na aplicação, tenha acesso a todas essas informações. Por essas características, cada palavras então é tratada como uma entidade, seguindo o modelo orientado a objetos, e cada instância da palavra encontrada terá essas informações com uma valor válido. Então, as duas categorias que contém as funcionalidades do sistema serão descritas como Indexação e Busca, estas fazem uso comum de uma estrutura de dados responsável pelo armazenamento das palavras.

A primeira categoria, Indexação, é responsável receber os arquivos por meio de uma interface com o usuário, que indica o arquivo a ser adicionado ao sistema por meio do caminho físico no computador. O arquivo deve ser de texto puro e seu conteúdo preenche a estrutura de dados, de modo que posteriormente palavras poderão ser consultadas. O usuário tem autonomia para adicionar arquivos, removê-los, o que consequentemente também remove a ocorrência de suas palavras do sistema, e atualizá-los. No caso de atualização, as palavras do arquivo que já estão armazenadas no sistema serão todas removidas e as atualizadas serão adicionadas, é reconhecido que embora esta abordagem não seja a mais eficiente, visto que muitas palavras podem não ter sido modificadas nesta atualização, outra abordagem mais eficiente exigiria uma técnica mais elaborada. Vale salientar que no sistema há uma arquivo de configuração identificado como blackList, que contém palavras que não serão consideradas nos arquivos adicionados pelo usuário, ou seja, palavras que estão na blackList nunca estarão na estrutura de dados e não serão encontradas em caso de busca.

Quanto à categoria de busca, esta é responsável por encontrar palavras com ocorrências nos arquivos, buscando-as na estrutura de dados do sistema. O usuário indica quais palavras deseja pesquisar e o tipo de busca, este pode ser AND ou OR. Numa busca AND, o sistema deverá retornar ao usuário as ocorrências de todas as palavras, se não houver a ocorrência de alguma delas, nenhuma é retornada. Já na busca OR, as palavras que forem encontradas têm suas ocorrências retornadas, indepente de alguma delas não ter sido encontrada. As ocorrências de uma palavra indicam os arquivos em que a palavra foi encontrada, as linhas em que se encontram neste arquivo e o número de aparições em cada linha.

O módulo de busca conta ainda com a funcionalidade de sugestões de palavras da base, semelhantes à que está sendo busca, caso a palavra buscada inicialmente não seja encontrada. Este método, bastante comum nos motores de busca, tem a intenção de atentar ao usuário para erros ortográficos. Essa função é feita com base na distância de Levenshtein, que identifica o número de operações necessárias para transformar uma palavra em outra. O algoritmo que tem como entradas duas Strings de, respectivamente, tamanhos m e n e

apresenta uma complexidade da ordem de m x n. A implementação do algoritmo de Levenshtein utilizada foi extraída do *wikibook* “Algorithm Implementation” [Wikimedia 2017]. Para criar um método de sugestão de palavras utilizando este algoritmo afere-se as distâncias de cada uma das palavras armazenadas à palavra alvo; ao fim do procedimento restará a palavra mais próxima possível do alvo, mas a distância entre a palavra alvo e a palavra mais próxima dela ainda pode ser grande suficiente para que não haja relação entre elas, é conveniente, portanto, definir uma distância de corte máxima aceitável para que a palavra seja utilizada como sugestão.

3. Descrição da solução

O projeto é composto por quatro classes principais, Oohay, *ArvoreTrie*, *Palavra* e *Indexador*, que trabalham em conjunto com outras classes auxiliares para realizar todas as funcionalidades da aplicação. A classe *Palavra* representa o tipo de objeto que será armazenado na estrutura de dados, uma objeto desse tipo contém um valor String, que é a palavra como o usuário conhece, e uma lista de objetos da classe *OcorrenciaArquivo*, que especifica as informações atreladas à palavra, cada ocorrência da palavra corresponde a uma ou várias aparições dela em uma linha específica de um arquivo. Nessa modelagem, cada palavra é tratada como única e o que varia são suas ocorrências, quando uma mesma String é encontrada em diferentes locais pelo indexador.

A classe *ArvoreTrie* representa a estrutura de dados escolhida para o projeto, essa é uma árvore digital, e seus métodos principais são os de inserção, busca e remoção de palavras em sua estrutura. Essa classe trabalha com objetos da classe *Node*, que contém um objeto da classe palavra e uma conjunto de outros *Node*, chamados nós filhos. Cada *Node* será uma das ramificações da árvore escolhida. Uma das principais decisões de projeto foi a forma como essa árvore foi desenvolvida, diferentemente da árvore Trie tradicional que segue um alfabeto específico, onde cada nó deve ter o número de filhos igual ao número de símbolos nesse alfabeto, na implementação da árvore deste projeto os nós filhos são armazenados dinamicamente à medida que são inseridos, não necessitando de uma gasto de memória extra para armazenar nós vazios. Em compensação, há certa perda de desempenho na busca por um filho específico, que é feita de forma linear.

Já a classe *Indexador* é a responsável por interpretar palavras de arquivos e transformá-las em objetos do tipo Palavra. Estas então são salvas numa lista e adicionadas à estrutura de dados do programa para que possam ser consultadas depois. Cada objeto de uma dessas listas é interpretado como único, referindo-se a somente uma String daquele arquivo. Ou seja, o indexador não identifica se uma palavra ocorre mais de uma vez numa mesma linha e a quantidade de vezes que ela se repete naquele arquivo, essa responsabilidade é passada para a estrutura de dados, que deverá receber cada uma dessas palavras e sua ocorrência e organizá-las da forma correta, seja adicionando um novo nó na estrutura ou atualizando as ocorrências de uma palavra que já tenha sido adicionada anteriormente.

Por fim, a classe *Oohay* é responsável por instanciar as demais e fazer uso de suas funcionalidades para prover uma interface com o usuário, que apenas precisará passar arquivos e buscar por palavras, sem saber como isso é feito.

Quanto à solução ao todo, esta está dividida em cinco módulos (pacotes) com tarefas bem definidas, que são: módulo *core*, que representa o núcleo do projeto, e é onde estão as classes principais e contém toda a implementação base da solução. Depois há o módulo de *user interface* contendo a interface gráfica, que fornece uma meio de fácil uso das funcionalidades do sistema ao usuário. Os pacotes *indexador* e *buscador*, são os que de fato implementam a interface gráfica das duas funcionalidades principais, no entanto as classes em *user interface* que as gerenciam e identificam interações do usuário com o sistema.

Para sua implementação, decidiu-se fazer uso do padrão de projeto Observer. Este é um conhecido padrão que define uma dependência *one-to-many* entre objetos de forma que quando o estado do objeto principal (*Observable*) é modificado ele notifica uma atualização aos seus dependentes (*Observers*) [Source Making 2017]. No “Oohay”, existe um objeto que funciona como comunicador entre a interface gráfica e os códigos do *Core* da aplicação. Este objeto, que possui o nome *CoreBinder* implementa duas interfaces *FileObservable* e *SearchObservable.*

O *FileObservable* notifica *FileObservers* a respeito de alterações na lista de arquivo (inserção, remoção e atualização); um dos *FileObservers* é o painel que implementa a tabela de arquivos. Este mesmo painel também realiza operações no estado da aplicação, por isso ele também pode realizar chamada de métodos no *CoreBinder*.

Por outro lado o *SearchObservable* notifica dois tipos de dados: lista de resultados para uma busca (que pode ser vazia caso não existam resultados) ou uma *Suggestion* que oferece ao usuário uma opção de busca de termo parecido com o que foi buscado, mas que retorna resultados.

Outro pacote que há no sistema é o de teste, onde foram implementados testes progressivos em novos componentes e funções, à medida que eram adicionados no projeto. A estrutura do *core* da aplicação desenvolvida está representada no Diagrama 1; o Anexo F contém o Diagrama da interface gráfica do sistema.

4. Estrutura de dados

Para armazenamento das informações do projeto, palavras dos arquivos e suas ocorrências, a estrutura de dados escolhida foi a Árvore Trie, também conhecida como árvore digital. Essencialmente esse tipo de árvore é utilizada para o armazenamento de textos de algum alfabeto especificado, cujos caracteres possuem uma posição fixa no array de filhos de qualquer nó da estrutura, de modo que ao ser inserida uma nova palavra na trie, cada uma das suas letras pertencerá a uma nó em níveis diferentes da árvore, formando muitas ramificações na estrutura à medida que inserções vão ocorrendo, podendo haver palavras que compartilham caminhos, caso haja prefixos em comum. A figura (1) apresenta a anatomia comum de uma árvore Trie, é importante notar que as palavras que de fato estão armazenadas são aquelas que possuem algum valor associado ao seu nó. O alfabeto adotado neste caso foi o latim, com 26 letras.

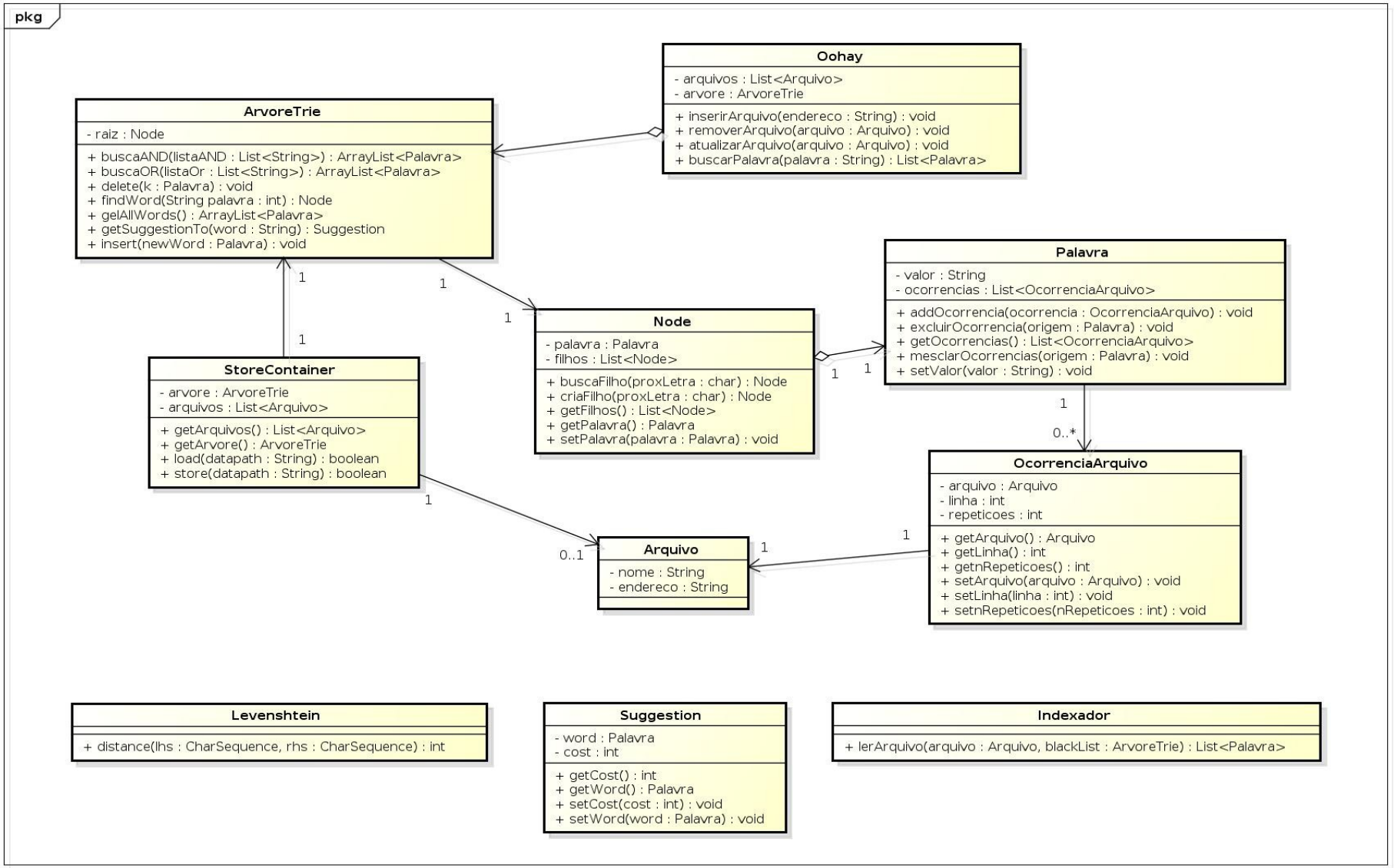

Diagrama 1: Diagrama de classes do sistema desenvolvido

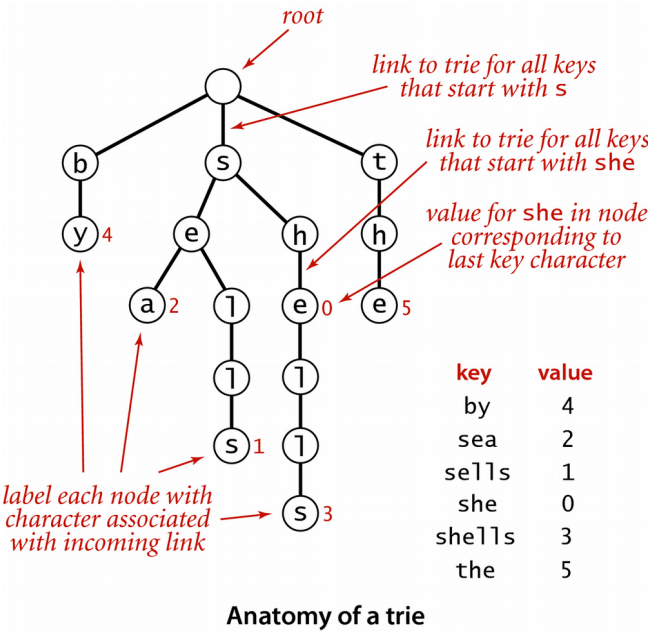

Figura1: Representação comum da árvore Trie
Fonte: https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html

Uma das principais desvantagens da Trie é justamente a sua necessidade de adoção de uma alfabeto específico, que proporciona a utilização de um número limitado de caracteres para as palavras na estrutura, além da necessidade de alocação extra memória, já que cada nó na estrutura possui uma array de R filhos, onde R é o tamanho do alfabeto, o que também a nomeia como R-Way, ficando comumente sobrecarregada com muitos ponteiros nulos.

Como forma de amenizar alguns desses problemas citados, bem como permitir a inserção de textos numa variedade maior de alfabetos, a estrutura da árvore Trie utilizada neste sistema sofreu algumas adaptações. A primeira destas foi nos valores dos objetos Palavra de cada nó, que segue o padrão apresentado na figura (2). O campo do tipo string da Palavra de um nó terá um caractere a menos que o da Palavra dos seus filhos (desconsiderando a raiz que não possui qualquer valor). O que identifica se uma Palavra está de fato na estrutura é o número de ocorrências que a ela está associada, assim, se uma nova palavra adicionada for o prefixo de uma já existente, por exemplo, basta apenas localizar esse prefixo na estrutura e incrementar suas ocorrências. Desse modo a esta árvore cabe mais a nomeação de “árvore de prefixos” que “R-Way”.

A segunda, e principal, adaptação é que não há a padronização de uma alfabeto a ser seguido. Os filhos de um nó são inseridos dinamicamente, e não numa posição específica derivada de uma espaço de memória reservado e, até então, nulo. Cada caractere de uma palavra é considerado, e mesmo palavras acentuadas são tratadas como diferentes de uma outra semelhante sem acentuação. Dessa forma, há uma ganho no consumo de memória da solução, que será minimizado, além de que permite uma maior flexibilização na escolha dos textos que um usuário pode inserir, não havendo a necessidade de checar se todo o seu conteúdo atende a uma especificação de alfabeto.

A principal desvantagem dessa adaptação é a perda de eficiência para localizar certo caractere na busca, pois para saber para qual ramificação avançar, os filhos de cada nó serão checados linearmente até que se encontre o caractere que corresponde ao próximo nó da sequência. Assim, essa busca pelo próximo nó da sequência que tinha complexidade O(1) na árvore R-Way, terá complexidade O(n) na árvore de prefixos desse projeto, uma preço aceitável a pagar, considerada as vantagens indicadas.

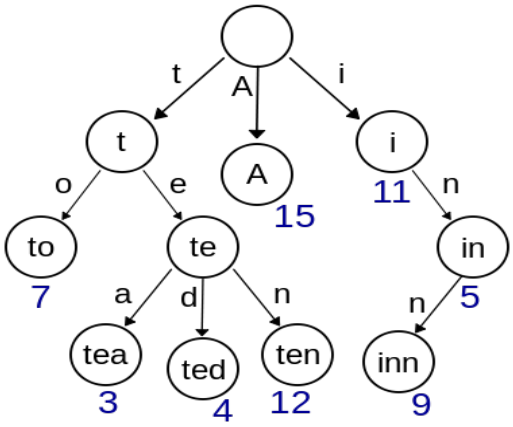

Figura 2: Representação da Trie adotada neste projeto
Fonte: https://pt.wikipedia.org/wiki/Trie

Consideradas então as adaptações na árvore, uma análise de complexidade nos seus principais métodos: buscas AND e OR, inserção e remoção, indicam que há uma perda de desempenho em relação à Trie R-Way. Enquanto nesta a complexidade dos seus métodos possui o mesmo limite superior que uma busca binária, O(log n), sendo n o número de filhos na árvore, os métodos desenvolvidos na aplicação deste projeto possuem limite superior igual a O(n * m), onde n é o número de filhos em cada nó e m é o número de caracteres da String da Palavra que se deseja adicionar, remover ou buscar. O tamanho de m, no entanto, é no máximo cerca de 40, considerando o número aproximado de caracteres das maiores palavras que existem. Esse resultado mostra então que há uma *tradeoff* entre esses dois tipos de árvores, pois para compensar o ganho de memória, a Trie deste projeto teve que perder desempenho.

Os pseudocódigos dos métodos da árvore Trie podem ser encontrados na seção anexo deste relatório.

5. Reflexão

O produto final do projeto alcançou o estado esperado e organizado na etapa de planejamento, este mérito é dado à arquitetura do sistema que foi muito bem definida antes de se iniciar o desenvolvimento de código fonte. Poucas decisões foram mudadas no decorrer do processo de codificação e estas decisões não causaram impactos e custos negativos, no sentido de adaptação do projeto, graças ao baixo acoplamento do código.

A equipe optou por utilizar a metodologia de *pair programming* em que todos os desenvolvedores, apesar de possuir suas responsabilidades no sistema, também revisam e colaboram com os códigos de outros desenvolvedores. Esta decisão de metodologia foi

importante para garantir qualidade de software. Outros fatores importantes, que também contribuem para a qualidade do código são a utilização de testes unitários, desenvolvidos com o padrão JUnit e documentação do código pelo padrão Javadoc.

Foram aplicadas ferramentas colaborativas no processo de desenvolvimento: serviço de comunicação por voz e de compartilhamento e edição de arquivos de texto relacionados ao projeto. Para o versionamento e compartilhamento de código fonte foram utilizados, respectivamente, a ferramenta git e o serviço de diretórios remotos GitHub[1].

Alguns conceitos que foram aprendidos no decorrer da disciplina “Linguagem de Programação II” foram fundamentais no projeto, como os conceitos de Programação Orientada à Objetos: classes, objetos e interfaces, que foram largamente aplicados no sistema; o padrão de projeto *Observer*, que foi útil na interface com usuário; a árvore Trie (que foi, também, muito bem abordada na disciplina “Estrutura de Dados Básicas II”) que foi o componente de armazenamento de dados e o algoritmo de Levenshtein, que despertou um interesse em buscar conhecimento extracurricular.

Algumas decisões de projeto, ou melhorias também poderiam ser adotadas, apesar de o sistema conseguir atender aos requisitos propostos. Uma decisão que poderia ser tomada é a aplicação de uma árvore mais otimizada, como a árvore P.A.T.R.I.C.I.A., que é uma versão comprimida da árvore de prefixos que foi aplicada. Outro tipo de árvore que poderia ser considerada, levando em conta adaptações e até mesmo testes para compreender a eficiência desta árvore para este tipo de problema, é a árvore A.V.L., uma árvore balanceada que possui uma complexidade de acesso mais eficiente que outras estruturas.

Uma melhoria, ou funcionalidade, que poderia ser desenvolvida é um sistema de *autocomplete* que sugere termos que combinam com o texto que o usuário digitou em tempo real, assim como o *autocorrect*, implementado, é um recurso bastante explorado por buscadores *online*.

Referências

Google (2017). Como a pesquisa organiza informações. Acesso em 4 de dezembro de 2017. Disponível em: https://www.google.com/search/howsearchworks/crawling-indexing/

Wikimedia (2017). “wikibook” Algorithm Implementation. Levenshtein distance (em inglês). Acesso em 4 de dezembro de 2017. Disponível em: https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#Java

Source Making (2017). Observer Design Pattern (em inglês). Acesso em 4 de dezembro de 2017. Disponível em: https://sourcemaking.com/design_patterns/observer

1. Todo código desenvolvido está disponibilizado em https://github.com/mrmorais/oohay/

Anexo A - Função de busca simples

```
funcao Node busca(String palavra) {
    retorne busca(raiz, palavra, 0);
}

funcao Node busca(Node node, String palavra, int level) {
        se (eVazia(palavra)) entao
                returne nulo;
        se (tamanho(palavra) == level) entao {
             se (node.getPalavra().getOcorrencias().size() > 0) entao
                retorne node;
             senao
                retorne nulo;
        }
        char currentChar = palavra.charAt(level);

        Node nodeFound = node.buscaFilho(currentChar);
        se (nodeFound == nulo)
              retorne nulo;
        retorne busca(nodeFound, palavra, level + 1);
}
```

Anexo B - Função de busca tipo And

```
funcao VetorDinamico<Palavra> buscaAnd( Lista<String> listaAnd)
{
  VetorDinamico<Palavra> listaRetorno
  listaRetorno instancia VetorDinamico<Palavra>
  para toda String p em listaAnd faca
  {
    Node findNode <= findWord(p)
    se findNode não nulo faca
    {
      listaRetorno.adiciona( findNode.getPalavra() )
    }
    senao
      retorne nulo
  }

  retorne listaRetorno;
}
```

Anexo C - Função de busca tipo OR

```
funcao VetorDinamico<Palavra> buscaOR( Lista<String> listaOr)
{
    VetorDinamico<Palavra> listaRetorno
    listaRetorno instancia VetorDinamico<Palavra>
    para toda String p em listaOr faca
    {
        Node findNode <= findWord(p)
        se findNode não nulo faca
        {
            listaRetorno.adiciona( findNode.getPalavra() )
        }
    }

    retorne listaRetorno;
}
```

Anexo D - Função delete

```
procedimento delete(Palavra k){
    raiz = delete(raiz, k, 0);
}

funcao Node delete(Node node, Palavra palavra, int level) {
    se (node == nulo) entao
       retorne nulo;

    se (level == tamanho(palavra.getValor()) ) entao
    {
       node.getPalavra().excluirOcorrencia(palavra);
    }
    senao
    {
       char currentChar = palavra.getValor().charAt(level);
       Node nodeFound = node.buscaFilho(currentChar);

       nodeFound = delete(nodeFound, palavra, level + 1);
    }

    se ( !vazio(node.getFilhos()) )
       retorne node;

    retorne nulo;
}
```

Anexo E - Função de inserção

```
Node inserir(Node no, Palavra palavra, int level) {
        char cChar = palavra[level];

        Node nodeFound = node.buscaFilho(cChar);

        se nodeFound == nulo então
                Node newFilho = node.criaFilho(cChar);


        se palavra.lenght - 1 == level então
                newFilho.palavra = palavra;
                retorna newFilho;
        fimse;
                retorne inserir(newFilho, palavra, level + 1);

        senão
                //Aqui adiciona nova ocorrência à palavra já existente na árvore
                se palavra.lenght - 1 == level então
                        nodeFound.palavra.mesclarOcorrencias(palavra);
                        retorna nodeFound;
                fimse;
        fimse;
        retorna inserir(newFilho, palavra, level + 1);
}
```

Anexo F – Diagrama de Classes da Interface Gráfica