Rúben Jozafá Silva Belém

Combinando Busca Binária Exata e Busca Aproximada em Sistemas de Complementação Automática de Consultas

Manaus, AM - Brasil

2022

Rúben Jozafá Silva Belém

Combinando Busca Binária Exata e Busca Aproximada em Sistemas de Complementação Automática de Consultas

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Amazonas como parte dos requisitos necessários para a obtenção do grau de Mestre em Informática.

Universidade Federal do Amazonas – UFAM

Instituto de Computação – ICOMP

Programa de Pós-Graduação em Informática – PPGI

Orientador: Prof. Dr. Edleno Silva de Moura

Manaus, AM - Brasil

2022

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Belém, Rúben Jozafá Silva
B454c Combinando busca binária exata e busca aproximada em sistemas de complementação automática de consultas / Rúben Jozafá Silva Belém . 2022
94 f.: il. color; 31 cm.

Orientador: Edleno Silva de Moura
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Processamento de consultas. 2. Preenchimento automático. 3. Complementação automática de consultas. 4. Tolerante a erros. 5. Trie. I. Moura, Edleno Silva de. II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

"Combinando busca binária exata e busca aproximada em sistemas de complementação automática de consultas"

RÚBEN JOZAFÁ SILVA BELÉM

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Edleno Silva de Moura - PRESIDENTE

Prof. João Marcos Bastos Cavalcanti - MEMBRO EXTERNO

Prof. Altigran Soares da Silva - MEMBRO INTERNO

Manaus, 25 de Março de 2022

Dedico este trabalho aos meus pais, que muitas vezes acreditaram mais em mim do que eu mesmo acreditei. À minha esposa, que esteve comigo nos dias de luta e nos dias de glória. À minha família, amigos e colegas que possam de alguma forma terem contribuído para que este trabalho pudesse ser elaborado.

Agradecimentos

A Deus, em primeiro lugar, que me capacitou a completar este trabalho, preservou a minha saúde, e me forneceu todos os recursos materiais e intelectuais necessários para isso.

Ao meu pai, M.Sc. Ruan Josemberg Silva Belém, que ouviu minhas dificuldades e me aconselhou, alegrou-se comigo a cada avanço, e não se resguardou de ter conversas difíceis porém necessárias para que eu pudesse terminar este trabalho.

À minha mãe, Vanusa Silva Belém, por levar muito peso sobre suas costas para que eu pudesse me concentrar no desenvolvimento deste trabalho.

À minha esposa, Keren Rosa Belém, por todo o apoio, por orar muito por mim, ouvir meus desabafos, confortar-me nos momentos de ansiedade e preocupação, por cuidar de mim e dividir responsabilidades para que eu pudesse focar neste trabalho.

Ao meu orientador, professor Dr. Edleno Silva de Moura, pela oportunidade de realizar este trabalho, e por muitas vezes me desafiar, fazendo-me ter contato com capacidades que eu não imaginava possuir. Antes mesmo de entrar para a graduação já conhecia alguns de seus feitos e já o admirava. Sinto-me honrado por essa orientação.

Ao Programa de Pós-Graduação de Informática e a todos os professores que participaram da minha jornada. Meu profundo agradecimento.

Ao ICOMP por prover tudo que precisei para desenvolver minhas pesquisas.

Aos parceiros Daniel Xavier e Berg Ferreira do grupo de pesquisa BDRI, por todo o conhecimento compartilhado. O apoio de vocês foi muito importante para mim.

Ao meu amigo Rodrigo Acioli que por muitas vezes ouviu meus desabafos e nunca deixou de me encorajar a seguir em frente, e também aos meus amigos da faculdade Timóteo Fonseca, Ivo Machado, Gabriel Pereira, Ruan Barros, e Juliana Castro, pois eles tiveram grande contribuição para que eu fizesse uma ótima graduação, a qual me proporcionou uma excelente base para realizar este trabalho.

Ao meu saudoso amigo Caio Pinheiro, que tinha planos de entrar novamente para o programa de Mestrado, mas antes que o conseguisse veio a falecer durante o período de desenvolvimento deste trabalho. Muito obrigado pelo incentivo, pelos conselhos, pela amizade! Dedico este trabalho também a você.

Lembre da minha ordem: "Seja forte e corajoso!
Não fique desanimado, nem tenha medo, porque eu, o Senhor, seu Deus,
estarei com você em qualquer lugar para onde você for!"
(Bíblia Sagrada, Josué 1:9)

Resumo

A complementação automática de consultas é uma funcionalidade importante para sistemas de busca modernos, e consiste em sugerir consultas a cada tecla digitada pelo usuário. As soluções mais eficientes dos últimos anos têm utilizado índices de árvore *Trie* para indexar as sugestões de consultas (Ji et al., 2009; Li et al., 2011; Xiao et al., 2013; Deng et al., 2016; Zhou et al., 2016). No entanto, essa estrutura pode acabar consumindo uma grande quantidade de memória, que é um recurso caro e limitado. Este trabalho apresenta uma abordagem em dois níveis que utiliza o algoritmo ICPAN (Li et al., 2011) de busca tolerante a erros em um índice de árvore *Trie* no primeiro nível e combina busca sequencial com binária no segundo, tendo o objetivo de averiguar se essa combinação possibilita tornar o segundo nível mais eficiente sem que a acurácia dos resultados seja prejudicada. A hipótese inicial era a de que é possível realizar busca binária em vez de sequencial quando todos os erros de digitação tolerados já estiverem sido "esgotados" no primeiro nível. No entanto, concluímos no decorrer desta pesquisa que utilizar busca binária no segundo nível impede o método de recuperar algumas sugestões de consulta que deveria, e também o faz trazer outras que ultrapassam a quantidade τ de erros de digitação tolerados. Diante desse problema, também propusemos uma heurística de ativação seletiva da busca binária. Experimentamos três versões de métodos em dois níveis, cada uma com determinada modificação no segundo nível e comparamos seus níveis de acurácia, seus desempenhos, e utilização de memória. Ambas as versões que utilizam busca binária com e sem heurística demonstram-se imprecisas para $\tau \leq 2$, porém com uma boa precisão para $\tau = 3$. Nesse caso em específico o modelo que utiliza busca binária sem heurística obteve o melhor desempenho em comparação às outras versões, e também desempenhou melhor que outros métodos encontrados na literatura em alguns casos. Além disso, todos os três modelos propostos demonstraram redução significativa da quantidade de memória necessária para realizar a complementação automática de consultas.

Palavras-chave: Processamento de consultas, preenchimento automático, complementação automática de consultas, tolerante a erros, dois níveis, Trie, busca binária.

Abstract

Query autocompletion is an important feature for moderen search engines that stands for suggesting queries at each user keystroke. The most efficient solutions in the past years have been using Tries to index the query suggestions (Ji et al., 2009; Li et al., 2011; Xiao et al., 2013; Deng et al., 2016; Zhou et al., 2016). However, this structure may lead to more memory usage, which is a costly and limited resource. In this work, we introduce a two-level structure that uses the ICPAN (Li et al., 2011) error-tolerant search algorithm at the first level and combines sequential and binary search at the second level, with the aim of verify whether this combination can increase the efficiency of the second level without affecting the results accuracy. Our initial hypothesis was that it was possible to perform a binary search instead of a sequential when all the typing errors had already "ran out" at the first level. Nonetheless, we conclude in our research that using binary search at the second level prevents the method of retrieving some query suggestions, and also makes it retrieve some suggestions that exceed the limit τ of typing errors tolerated. Facing this problem, we also proposed a heuristic to selectively activate the binary search. We have experimented with three versions of two-level structure, with each one having a modification at the second level. We compared their accuracy levels, their performances, and their memory usage. Both versions that use the binary search with and without the heuristic showed some lack of accuracy for $\tau \leq 2$. However, they had good accuracy for $\tau = 3$. In this specific case the model that uses binary search without the heuristic had the best performance when compared with other versions, and also had performed better than other methods found in the literature in some scenarios. Besides that, all the three proposed models have shown a significant reduction of memory needed to run error-tolerant query autocompletion.

Key-words: Query processing, autocompletion, query autocompletion, error-tolerant, two level, Trie, binary search.

Lista de ilustrações

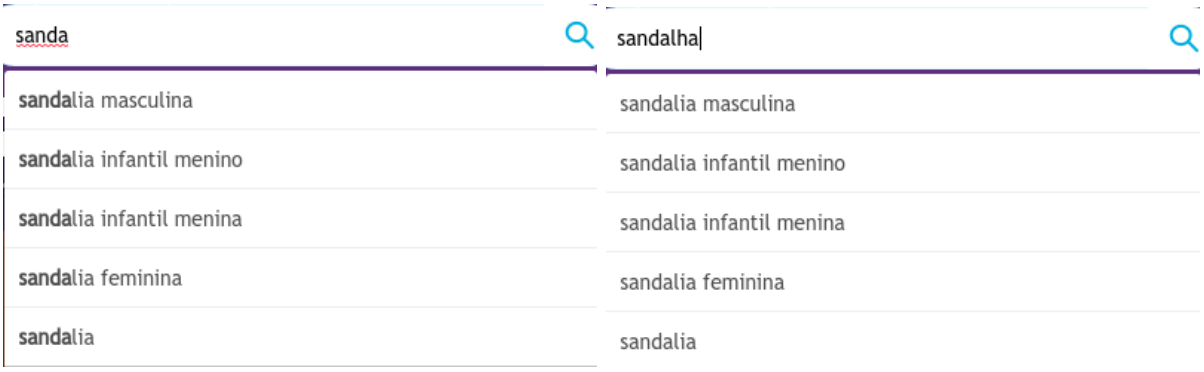
Lista de tabelas

Lista de Algoritmos

Sumário

1 Introdução

A forma tradicional de utilizar sistemas de busca é realizar consultas por meio de palavras-chave a partir das quais o sistema responde com resultados potencialmente relevantes para o usuário. Se o usuário não sabe exatamente como expressar a sua consulta ele pode refiná-la e aprimorá-la por meio de "tentativa e erro" até que encontre a informação que se está buscando, o que pode tornar o processo demorado e sujeito a erros. A complementação automática de consultas é um mecanismo que pode auxiliar usuários na tarefa de busca e que tem sido amplamente adotado atualmente. Essa funcionalidade consiste em, a partir de um prefixo já digitado pelo usuário, tentar "advinhar" a consulta final que será digitada (Di Santo et al., 2015). Com isso, não é necessário escrever toda a consulta, o que além de poupar tempo, pode ajudar a formular uma consulta mais precisa.



(a) Exemplo de complementação automática de consultas comum.

(b) Exemplo de complementação automática de consultas tolerante a erros.

Figura 1: Complementação automática de consultas em uma plataforma de *e-commerce*.

A complementação automática de consultas pode ocorrer de duas formas. A primeira está representada na Figura 1a, na qual o usuário insere o prefixo de consulta *"sanda"* e o sistema automaticamente sugere cinco possíveis consultas relacionadas cujos prefixos correspondem exatamente ao que foi consultado (destacados em negrito em cada consulta). No entanto, é comum que haja erros de ortografia no conteúdo digitado. Cerca de 10% a 20% das consultas são digitadas erroneamente (Broder et al., 2009). Para tratar tais situações é necessário que esses erros sejam considerados na busca pelas sugestões que serão retornadas. Esse problema é denominado complementação automática tolerante a erros (CATE). Quando o usuário não sabe escrever corretamente uma palavra é provável que ele consulte por um prefixo que contenha erros de digitação. A segunda forma da complementação automática, presente na Figura 1b, representa uma resolução para esse problema. Apesar da digitação errada do prefixo *"sandalha"* o sistema ainda assim foi capaz de exibir sugestões cujos prefixos correspondem aproximadamente ao que

foi consultado, pois sua complementação automática de consultas é tolerante a erros. Tal tolerância pode economizar os esforços do usuário de 40% até 60% (Ji et al., 2009).

Esse mecanismo pode ser visto como uma versão especializada do problema de casamento de padrão. Nesse problema há uma janela de busca com o mesmo tamanho do padrão p buscado que é movida da esquerda para a direita em um texto t. A cada movimento, o padrão p é buscado dentro dessa janela. Esse casamento de padrão pode ser realizado de forma exata, ou aproximada. O problema de casamento exato de padrão é definido como encontrar todas as ocorrências $t_{j',j} = t_{j'}..t_j$ de um padrão $p = p_1, .., p_m$ em um texto $t = t_1, .., t_n$, sendo $m \leq n$ (Faro and Lecroq, 2013).

Já o problema de casamento aproximado de padrão é definido como encontrar todas as sub-cadeias de caracteres $t_{j',j} = t_{j'}..t_j$ de um texto $t = t_1, .., t_n$ cuja distância de edição para um padrão $P = p_1, .., p_m$ esteja dentro de um limiar τ, ou seja, $ed(t_{j',j}, P) \leq \tau$, sendo $ed(s_1, s_2)$ uma função que calcula a distância de edição entre s e t. A distância de edição entre duas cadeias de caracteres s_1 e s_2 é definida como *o menor número de operações necessárias para transformar* s_1 *em* s_2. Comumente as operações permitidas nesse contexto são a *inserção*, *deleção* ou *substituição* de um caractere.

Essa versão aproximada do problema de casamento de padrão pode ser aplicada ao problema de CATE. Seja p um prefixo de consulta digitado pelo usuário, $S = \{Q_1, Q_2, ..., Q_n\}$ um conjunto de n cadeias de caracteres que representa todas as possíveis sugestões de consulta, e τ um limite máximo de distância de edição tolerado. O problema de CATE consiste em encontrar um subconjunto de sugestões S' cujos elementos correspondem à p com no máximo τ erros. Ou seja, é preciso que cada consulta presente em S' possua pelo menos um prefixo que possa possa ser transformado em p com no máximo τ operações de edição.

Para exemplificar, seja $\tau = 2$, $p =$ "*sapatho*" e S = *{"sapatilha preta", "salaminho italiano", "sapinho verde"}*. De acordo com a definição apresentada acima, o subconjunto de S correspondente às consultas que serão sugeridas será *S' = {"sapatilha preta", "sapinho verde"}*. No caso da consulta "salaminho italiano", seus possíveis prefixos são *{"s", "sa", "sal", "sala", ... , "salaminho italiano"}*, e nenhum deles pode ser transformado em $p =$ "*sapatho*" com no máximo $\tau = 2$ edições, portanto, ela não pode ser retornada como resposta. Por outro lado, a consulta "sapatilha preta" possui como um de seus prefixos "sapat", que pode ser transformado em p com a inserção dos caracteres "h" e "o" no final (duas operações de inserção). Há também a consulta "sapinho verde" com o prefixo "sapinho", que pode ser transformado em p ao substituir a quarta e quinta letra por "a" e "t", respectivamente.

A tendência comum entre os melhores métodos propostos recentemente no estado da arte para solucionar esse problema (Chaudhuri and Kaushik, 2009; Ji et al., 2009; Li et al., 2011; Xiao et al., 2013; Deng et al., 2016; Zhou et al., 2016) é a de utilizar uma

estrutura chamada *Trie* (Fredkin, 1960) para indexar os textos das sugestões de consulta. A *Trie* é uma árvore *M-ária* cujos nós são vetores de tamanho M contendo em suas células caracteres que pertencem a um alfabeto Σ. O nó raiz ϵ representa uma cadeia de caracteres vazia. Cada nó em um nível l representa o conjunto de todos os elementos que começam com uma sequência p (que pode ser obtida ao caminhar partindo do nó ϵ até o nó em questão) de l caracteres. Sendo assim, todos os prefixos em comum dentre os itens indexados compartilham dos mesmos nós. Quando o usuário faz uma consulta por um prefixo, esses métodos utilizam a *Trie* para calcular todos os prefixos similares, e a partir deles sugerir as consultas.

Porém, uma vez que cada nó da árvore pode ter até $|\Sigma|$ filhos, essa estrutura tende a consumir alta quantidade de memória quando a base de consultas indexada é muito grande e não há muitos prefixos em comum entre os itens. Essa situação pode dificultar o uso desses métodos para solucionar o problema de CATE em alguns casos. O método ICAN (Ji et al., 2009), por exemplo, ao indexar uma base de aproximadamente 10 milhões de itens consome $\sim 9GB$ de memória. Além disso, o complemento automático de consultas em sistemas de busca deve ocorrer muito rapidamente, de forma quase imperceptível para o usuário. Para isso, cada complementação deve ocorrer em menos de $100ms$ (Ji et al., 2009).

Uma possível abordagem para diminuir o uso de memória utilizada para processar as consultas é a estratégia de busca em dois níveis. No primeiro nível é possível realizar a busca em um índice *Trie* que indexa somente uma parte inicial dos textos da base de sugestões de consulta. As folhas encontradas por essa busca podem estar associadas com um conjunto de itens, em vez de um único texto (restante da sugestão de consulta). Então, no segundo nível, cada elemento desse conjunto é analisado (um por um, de forma sequencial) para determinar quais sugestões desse conjunto de fato vão ser sugeridas.

Costa Xavier (da Costa Xavier, 2019) realizou um trabalho preliminar que confirmou a hipótese de que é possível utilizar uma abordagem de busca em dois níveis para o problema de complementação automática de consultas tolerante a erros. Em sequência, Gama Ferreira (da Gama Ferreira, 2020) aprimorou significativamente a busca em dois níveis ao utilizar utilizar o mesmo algoritmo (BEVA) de busca nos dois níveis da *Trie* e ao apresentar uma nova estratégia de inserção de chaves em seu índice que acelera o processamento das consultas por utilizar o sistema de cache das máquinas de maneira mais eficiente.

A abordagem em dois níveis pode proporcionar uma ótima redução consumo de memória como dito anteriormente. No entanto, um dos maiores problemas com essa abordagem é o aumento do tempo de processamento necessário para sugerir consultas para um prefixo, pois a classe de complexidade computacional da busca sequencial realizada no segundo nível normalmente está acima da classe de complexidade dos algoritmos de

busca que podem ser utilizados nas árvores *Trie* do primeiro nível.

Neste trabalho estudamos alternativas para otimizar a busca em uma abordagem de dois níveis com o objetivo de computar as complementações de consulta tolerando erros, mantendo um bom desempenho aliado à redução da memória utilizada. Investigamos os impactos de uma heurística que visa aproveitar informações de erros de distância de edição provenientes do processamento da consulta no primeiro nível para acelerar o processamento no segundo nível, estudando seus efeitos no tempo de processamento e na acurácia de resultados (medição de quantas sugestões o método deixou de encontrar e quantas sugestões encontradas na verdade não deveriam ser consideradas).

Para implementar as ideias propostas utilizamos no primeiro nível o algoritmo ICPAN (Li et al., 2011), o qual é uma otimização do algoritmo ICAN (Ji et al., 2009), para realizar a busca no índice *Trie*. Ambos utilizam uma estratégia incremental que consiste em processar um prefixo de consulta $p = c_1c_2...c_x$ caractere a caractere, e "ativar" nós da árvore *Trie* a cada etapa do processamento. Para o i-ésimo caractere processado, cada nó ativo n representa um prefixo cuja distância de edição $\xi_n^{p'}$ entre $p' = p[c_1..c_i]$ é menor ou igual ao limite de tolerância τ estabelecido. O ICPAN é mais eficiente que o ICAN pois utiliza um conceito de "nós ativos pivotais" para reduzir o conjunto de nós ativados durante toda a computação. Essa redução implica que menos itens precisarão ser verificados no segundo nível, e portanto é vantajosa para a abordagem em dois níveis.

Quanto ao segundo nível, criamos a hipótese de que é possível em alguns casos realizar uma busca binária em vez de uma busca sequencial. Então, estudamos o quão efetiva seria uma abordagem que combina a estratégia de realizar tanto buscas sequenciais quanto binárias no segundo nível. Descobrimos no decorrer da pesquisa que apesar dessa estratégia atingir tempos de processamento até 2 vezes mais rápidos do que a abordagem padrão de dois níveis, ela faz o método recuperar itens que não devia, e também o faz deixar de recuperar itens relevantes. No entanto, ao tolerar $\tau = 3$ erros de digitação na busca, os métodos que utilizam busca binária podem apresentar bons níveis de acurácia.

1.1 Problema de pesquisa

A principal hipótese estudada nesta dissertação é a de que é possível implementar um sistema de complementação automática em dois níveis combinando busca sequencial e busca binária no segundo nível, de forma que a eficiência do processamento aumente e que a acurácia dos resultados não seja prejudicada. As propostas apresentadas por da Costa Xavier (2019) e da Gama Ferreira (2020) indicam que a busca em dois níveis é uma abordagem promissora para sistemas de complementação automática. Ao iniciar esta pesquisa, tivemos a ideia de combinar busca binária com sequencial no segundo nível. Apesar de parecer uma ideia simples a princípio, sua implementação trouxe consigo

desafios que são apresentados ao longo do trabalho. Nossa pesquisa teve como objetivo responder às seguintes questões de pesquisa: i) É possível adaptar sistemas de busca em dois níveis para tirarem proveito da busca binária no segundo nível? ii) Quais são as adaptações necessárias e limitações de uso da ideia? iii) Há vantagens práticas na combinação de busca binária e busca sequencial no segundo nível?

O restante desta dissertação está organizado da seguinte forma: No Capítulo 2 nós mostramos os trabalhos relacionados e os algoritmos existentes na literatura que abordam o problema de complementação automática de consultas tolerante a erros. No Capítulo 3 introduzimos uma definição formal do problema e apresentamos alguns conceitos necessários para uma compreensão mais completa do trabalho. No Capítulo 4 detalhamos a implementação dos métodos em dois níveis propostos. No Capítulo 5 apresentamos e analisamos os resultados dos experimentos realizados, e por fim, no Capítulo 6, expomos nossas conclusões do trabalho e também possíveis direções para trabalhos futuros.

2 Trabalhos relacionados

O problema de CATE foi primeiramente estudado por Chaudhuri and Kaushik (2009) e Ji et al. (2009). Ambos propuseram soluções baseadas em computar, de forma incremental, um conjunto de nós ativos de uma *Trie*, a qual funciona como índice para a base de consultas que poderão ser sugeridas. A ideia geral comum entre os dois algoritmos é a de que os algoritmos calculam um conjunto de "nós ativos" que correspondem a todos os prefixos encontrados na *Trie* similares a cada letra nova de um prefixo de consulta p digitado pelo usuário, dentro de um limite de distância de edição τ. A partir desses prefixos calculados, é possível recuperar na própria árvore o restante do texto das consultas que serão sugeridas.

Chaudhuri et al. Chaudhuri and Kaushik (2009) apresentam um método de complementação automática de consultas que leva em consideração erros de digitação no prefixo. Antes disso, para cenários em que havia uma tabela de frases para pesquisar, a forma mais comum de preenchimento automático era a comparação exata de caracteres. Dois algoritmos de complementação automática tolerantes à falha de digitação foram propostos (Chaudhuri and Kaushik, 2009). O primeiro é baseado nos algoritmos de distância de edição em q-gramas(Arasu et al., 2006; Chaudhuri et al., 2006; Xiao et al., 2008). O segundo é um algoritmo baseado em *Trie*, que mantém um conjunto de *nós ativos* de forma incremental para processar as complementações. O método utiliza distância de edição clássica como medida de similaridade entre cadeias de caracteres. Esse algoritmo é muito similar ao proposto por Ji et al. (2009), porém, realiza uma etapa a mais, que consiste em pré-computar o conjunto de nós ativos para todas as possíveis consultas com até certo tamanho. Uma vez que o número de nós ativos pode ser muito grande ao realizar complementação de consultas tolerante a erros, essa estratégia ajuda a reduzir o tempo de processamento das consultas.

Para abordar o problema de CATE Ji et. al (Ji et al., 2009) propõem um algoritmo chamado *ICAN* que computa nós ativos de forma incremental, ou seja, processa o prefixo consultado caractere a caractere, como mencionado anteriormente. Nosso método proposto no capítulo 4 utiliza uma otimização desse algortimo, o ICPAN (Li et al., 2011). Portanto, ambos serão descritos com mais detalhes a seguir, e também serão mais aprofundados no capítulo 3.

Seja p um prefixo consultado e τ um limite de distância de edição. Um nó n da *Trie* é chamado *nó ativo de p em relação à τ* quando a distância de edição entre a cadeia de caracteres de n e a de p estiver dentro do limite τ, ou em outras palavras, $ed(n, p) \leq \tau$. Os nós-folha da subárvore com raiz em n são denominados "palavras similares à p" (Ji

et al., 2009).

Considerando uma consulta de prefixo p com apenas uma palavra-chave, para processar os nós ativos de forma eficiente é necessário computar e armazenar um conjunto de tuplas $\Phi_p = \{\langle n, \xi_n \rangle\}$ tal que (1) cada n é um nó ativo em relação à p com $\xi_n = ed(n, p) \leq \tau$) e (2) quaisquer nós ativos de p estão presentes em Φ_p, que é o *conjunto de nós ativos de p* (Ji et al., 2009). Quando o usuário digita mais um caractere $p + 1$ após ter digitado p, o conjunto Φ_p de nós ativos de p pode ser usado para computar o conjunto Φ_{p+1} de nós ativos da nova consulta.

Uma desvantagem do algoritmo ICAN é que pode ser bastante custoso manter o conjunto de nós ativos para grandes quantidades de dados. Para atenuar esse problema, Li et al. (Li et al., 2011) desenvolveram um método otimizado denominado *ICPAN*, que realiza a poda de nós ativos desnecessários durante o processamento de consultas enquanto continua conseguindo computar todos os itens similares ao prefixo consultado. Isso permite a redução do espaço necessário para armazenar os nós ativos computados e também melhora o desempenho da busca, uma vez que não é necessário verificar todos os nós ativos para a computação incremental.

O ICPAN define um subconjunto dos nós ativos que pode ser usado para computar todas as palavras similares à consulta de forma eficiente e incremental. Para isso, Li et al. (Li et al., 2011) estabelecem a seguinte observação: dado um nó ativo n de p, se para qualquer transformação de n para p com $ed(n, p)$ operações de edição, não houver uma operação de correspondência (caracteres iguais) no último caractere de n, e for possível deletar o último caractere de n, o nó n não é mantido, e sim o seu nó pai. Assim, surge o conceito de *nó pivô ativo*.

Considerando um prefixo de consulta p, um nó da *Trie* n é um nó pivô ativo de p em relação à τ se, e somente se (1) n é um nó ativo de p e (2) se existe uma transformação de p para n com $ed(n, p)$ operações, e a operação no último caractere de n é uma correspondência. Em outras palavras, a operação nesse caractere de n não é nem uma deleção $ed(n, p) \neq ed(n', p) + 1$ e nem uma substituição $ed(n, p) \neq ed(n', p') + 1$, onde n' e p' são respectivamente os prefixos de n e p que não contêm o último caractere.

Dada uma consulta de prefixo p_x, de forma análoga ao ICAN, é preciso computar e armazenar um conjunto de tuplas $\Psi_{p_x} = \{\langle n, \xi_n^{p_x}, p_i, \xi_n^{p_i} \rangle\}$ (Li et al., 2011). Em cada tupla, n é um nó pivô ativo de p_x. O elemento p_i é um prefixo de p_x tal que os últimos caracteres de n e p_i são iguais. Se não existir tal prefixo, então $p_i = \epsilon$ (cadeia de caracteres vazia). Se existirem múltiplos prefixos com essa condição, somente aquele com o menor comprimento é selecionado. $\xi_n^{p_i} \leq \tau$ é uma distância de transformação entre o nó n para p_i com uma operação de correspondência entre seus últimos caracteres. $\xi_n^{p_x} \leq \tau$ é a distância de transformação entre o nó n para p_x obtida ao primeiro transformar o nó n para p_i e então inserir os caracteres após p_i. O ICPAN é um algoritmo que computa Ψp_x e garante

que cada nó pivô ativo de p_x aparece como o n em uma tupla de Ψp_x.

Em contrapartida, Xiao et al. (Xiao et al., 2013) argumentam que a eficiência dos métodos supramencionados criticamente depende da quantidade de nós ativos, e que tal número é tipicamente muito grande e também linear no tamanho da base de sugestões indexada, ou até mesmo exponencial no tamanho do alfabeto no pior caso. Então, os autores resolvem seguir em outra direção, investigando se é possível melhorar drasticamente o desempenho do processamento da consulta ao processar previamente a base e construir um índice de grandes proporções. Para isso, propuseram uma solução denominada IncNGTrie que consiste em indexar as "variações marcadas de deleção" (VMD) das sugestões em uma Trie e então manter um pequeno conjunto de nós ativos durante o processamento.

O conteúdo indexado não é o texto original das sugestões de consulta e sim suas variantes marcadas de τ-deleções, as quais são geradas ao deletar no máximo τ caracteres dos textos. Então, quando o usuário digita uma consulta o método computa as variantes da mesma e as pesquisa na árvore Trie. Essa técnica pode ser realizada de forma incremental e eficiente, mantendo um pequeno conjunto de nós ativos (reduzindo o tamanho em até 3 ordens de grandeza comparado aos métodos ICAN e ICPAN, por exemplo). Apesar dessa abordagem ter reduzido o tempo necessário para processar as consultas em comparação às outras soluções presentes na literatura, o tamanho dos índices também se demonstrou muito maior que dos outros métodos, representando uma severa restrição à utilização do IncNGTrie quanto à memória. Qin et al. (Qin et al., 2020) apresentam o IncNGTrie+, uma melhoria do IncNGTrie que reduziu tanto o tempo de processamento quanto o tamanho do índice construindo, diminuindo a quantidade de memória requerida, apesar de ainda precisarem de mais memória do que os métodos BEVA e META, por exemplo.

Zhou et al. (Zhou et al., 2016) propõem uma estrutura geral que engloba métodos da complementação automática existentes, o "BEVA", caracterizando diferentes classes de algoritmos e a quantidade mínima de informações que eles precisam manter para diversas restrições. É proposta também uma nova estratégia de armazenamento de "nós ativos de fronteira" (menor conjunto de nós ativos possível, sem redundâncias), eliminando inteiramente os relacionamentos entre nós pais e filhos. Essa abordagem realiza computação de distância de edição através de uma nova estrutura de dados chamada "*edit vector automaton*" (EVA), conseguindo calcular novos nós ativos e seus estados associados de forma eficiente através de pesquisas de tabela. O modelo é capaz de suportar limiares de grande distância através de um autômato. Os estudos do trabalho indicam que o método supera as abordagens até então existentes, tanto em tempo quanto na eficiência do espaço.

Deng et al. (Deng et al., 2016) desenvolveram um método denominado "META", uma estrutura baseada em correspondências que calcula as respostas com base em caracteres correspondentes entre os dados das consultas e os dados indexados. A estrutura desenvolvida é capaz de reduzir cálculos redundantes, organizando-os através de índices

de árvore compacta. Para processar as respostas às consultas foi proposto um método incremental que responde eficientemente consultas *top-k*.

Todos os trabalhos supracitados utilizam *Trie* como seu principal índice para realizar a computação de prefixos similares ao prefixo de consulta digitado pelo usuário. Isso fornece uma forte base para considerar que essa estrutura de dados realmente é vantajosa para o problema de CATE em se tratando do tempo de processamento. No entanto, as árvores *Trie* tendem a consumir uma alta quantidade de memória quando a base de consultas indexada é muito grande e não há muitos prefixos em comum entre os itens. Uma das formas de contornar esse problema é utilizar uma estratégia de indexação e busca em dois níveis.

Manber et al. (Manber et al., 1994) apresentam o "GLIMPSE" (*GLobal IMPlicit Search*) ou "Busca Implícita Global", uma ferramenta que possibilita a realização de consultas para sistemas de arquivos com baixa indexação, onde apenas 2% e 4% do tamanho do texto é indexado. O GLIMPSE apresentou uma abordagem até então nova para a indexação e realização de consultas de arquivos, denominada de "busca em dois níveis".

A ideia da busca em dois níveis aplicada no GLIMPSE consiste em combinar índices invertidos completos com busca sequencial sem indexação. Essa abordagem se baseia na observação de que a busca sequencial é rápida o suficiente para textos de vários *megabytes* de tamanho, e que por isso não é necessário indexar todas as palavras do documento com suas localizações exatas. Na busca em dois níveis, o índice não fornece os locais exatos das palavras, mas apenas uma indicação da região onde a resposta possa ser encontrada, funcionando como uma espécia de "filtro". Nessa abordagem no entanto, apesar de economizar bastante espaço de indexação, a velocidade de busca se mostrou inferior aos modelos que utilizam apenas indexadores baseados em lista invertida.

Navarro et al. (Navarro et al., 2000) estudaram uma possível combinação entre índice invertido completo e pesquisa sequencial sem indexação. A pesquisa completa em coleções de texto é feita com o uso de um índice que aponta para blocos em vez de apontar para cada posição no texto. Primeiramente, a pesquisa é realizada no índice para detectar blocos que podem corresponder à consulta e, em seguida, no segundo nível é feita uma pesquisa sequencial para encontrar a lista real de ocorrências no texto.

Quanto ao índice utilizado no primeiro nível, é possível utilizar índices baseados em árvores *Trie*. Heinz et al. (Heinz et al., 2002) propõem a estrutura *Burst Trie*, a qual pode ser considerada como um outro exemplo de índice de dois níveis, porém é aplicada em outro contexto. As *Burst Tries* são coleções de pequenas estruturas de dados denominadas "recipientes". Esses "recipientes" são análogos ao segundo nível nos moldes explicados anteriormente, e são acessados através de uma *Trie* comum, a qual pode ser considerada como um primeiro nível.

Um problema comum a todos os métodos de complementação automática de consultas tolerante a erros supracitados é o alto consumo de memória, uma vez que todos utilizam a estrutura *Trie* para indexar todo o texto de cada consulta da base de dados. Tendo como influência a combinação de índice invertido e pesquisa sequencial sem indexação ("busca em dois níveis"), uma forma de reduzir a quantidade de memória necessária seria indexar somente alguns caracteres iniciais de cada consulta na árvore *Trie*, e utilizar de pesquisa sequencial caso o processamento precise ir além das folhas da árvore.

Costa Xavier (da Costa Xavier, 2019) realizou um trabalho preliminar que confirmou ser possível utilizar uma abordagem de busca em dois níveis para o problema de CATE. Nesse método, define-se uma quantidade fixa L de caracteres (profundidade máxima da *Trie*), e então somente os L primeiros caracteres de cada consulta são indexados na árvore. Dado um prefixo de consulta P tal que $|P| > L$, a computação de sugestões é dividida em duas etapas, ou níveis. O primeiro nível utiliza o algoritmo ICAN (Ji et al., 2009) para processar os L primeiros caracteres de P. Após o término, resta um conjunto de nós ativos da *Trie* referentes à sub-cadeia $P[1..L]$. Então, no segundo nível, é realizada uma busca sequencial em cada consulta referenciada por cada nó ativo.

Após esse primeiro trabalho, Gama Ferreira (da Gama Ferreira, 2020) aprimorou significativamente a busca em dois níveis ao utilizar utilizar o mesmo algoritmo (BEVA) de busca nos dois níveis da *Trie*. Além disso, o trabalho propõe uma estratégia de inserção de chaves em seu índice *Trie* que acelera o processamento das consultas por utilizar o sistema de cache das máquinas de maneira mais eficiente. Essa mudança na forma de inserir chaves tem potencial para causar melhorias também em outros algoritmos baseados em *Trie* presentes na literatura. A busca sequencial no segundo nível deste método é mais eficiente que a de da Costa Xavier (2019), porque os estados de computação de distância de edição já existentes no primeiro nível podem ser usados para poupar redundâncias computacionais no segundo nível. Tal método atingiu uma ótima economia de memória em comparação ao algoritmo original BEVA, com o aditivo de conseguir manter um desempenho semelhante.

Os trabalhos de da Costa Xavier (2019) e da Gama Ferreira (2020) demonstraram que é possível combinar a estratégia de busca em índices *Trie* com busca sequencial para resolver o problema de CATE. Tal combinação reduz significativamente o uso de memória, enquanto mantém o tempo de processamento dentro do limite ideal de $100ms$ para uma boa parte dos casos experimentados. Nesta dissertação, exploramos a possibilidade de realizar otimizações em alguns casos específicos de processamento de consultas no segundo nível ao combinar busca sequencial com busca binária, visando obter ganhos de desempenho.

3 Referencial Teórico

O problema de casamento de padrão consiste em buscar as ocorrências de uma dada cadeia de caracteres p em um texto t. Nele há uma janela, na qual o padrão é buscado, que possui o mesmo tamanho do padrão e é movida da esquerda para a direita pelo texto. Há dois tipos de problemas de casamento de padrão:

- Casamento exato de padrão
- Casamento aproximado de padrão

O problema de casamento exato de padrão é definido como encontrar todas as ocorrências $t_{j',j} = t_{j'}..t_j$ de um padrão $p = p_1, .., p_m$ em um texto $t = t_1, .., t_n$, sendo $m \leq n$ (Faro and Lecroq, 2013).

Já o problema de casamento aproximado de padrão é definido como encontrar todas as sub-cadeias de caracteres $t_{j',j} = t_{j'}..t_j$ de um texto $t = t_1, .., t_n$ cuja distância de edição para um padrão $P = p_1, .., p_m$ esteja dentro de um limiar τ, ou seja, $ed(t_{j',j}, P) \leq \tau$, sendo $ed(s_1, s_2)$ uma função que calcula a distância de edição entre s e t. Nesse caso, o tamanho de cada ocorrência pode variar de $m - \tau$ até $m + \tau$. A distância de edição entre duas cadeias de caracteres s_1 e s_2 é definida como *o menor número de operações para transformar* s_1 *em* s_2. As operações mais permitidas comumente nesse contexto seguem as definições da *distância de edição de Levenshtein* (Levenshtein, 1966), que são:

1. **Inserção** de um único caractere. Se $s =$ “ac”, inserir o símbolo b na posição do meio, por exemplo, produz $s =$“abc”. Também pode ser representada por $\epsilon \rightarrow b$.
2. **Deleção** de um único caractere torna $s =$ “abc” em $s =$ “bc”, por exemplo. Também pode ser representada por $a \rightarrow \epsilon$.
3. **Substituição** de um caractere b por um símbolo $q \neq b$ modifica $s =$ “abc” para $s =$ “aqc”. Também pode ser representada por $b \rightarrow q$.

Esse problema é encontrado em várias aplicações práticas, por exemplo em buscas por sequências de DNA, ou para prover sugestões de correções de erros de digitação cometidos pelos usuários ao realizarem consultas em sistemas de busca.

3.1 Problema da complementação automática de consultas tolerante a erros

O problema de complementação automática de consultas tolerante a erros pode ser visto como uma versão especializada do problema de casamento de padrão. Seja p uma cadeia de caracteres que representa uma consulta de prefixo já digitada pelo usuário, S um conjunto de cadeias de caracteres que representa a base de consultas e τ um limite máximo de distância de edição tolerado. O problema consiste em encontrar todos os elementos de S que correspondem à p com no máximo τ erros.

Um elemento s qualquer do conjunto S possui vários prefixos. Para $s =$ "*sapato*", seus possíveis prefixos são *{"s", "sa", "sap", "sapa", "sapat", "sapato"}*. Se qualquer um deles puder ser transformado em p com um número de operações menor ou igual à τ, então s corresponde à p com no máximo τ erros, e fará parte da resposta do problema. A solução para o problema deve atender a uma restrição de computar os resultados em um tempo menor que a velocidade média de digitação de um usuário comum, a qual é aproximadamente $100ms$ (Ji et al., 2009).

3.2 Opções de solução para o problema de CATE

As soluções para o problema de casamento de padrão podem ser divididas em dois grupos. O primeiro engloba abordagens baseadas em busca sequencial por toda a base de consultas. O segundo grupo consiste em abordagens que criam índices a partir da base de consultas em um momento anterior ao processamento de requisições.

3.2.1 Busca sequencial

Os algoritmos de busca sequencial que não utilizam índices consistem em percorrer todos os itens da base de consultas, e comparar o prefixo de consulta com cada item para determinar quais consultas devem fazer parte da resposta. No contexto de CATE, essa comparação consiste em verificar se o prefixo consultado pode ser transformado em algum dos possíveis prefixos do item da base com no máximo τ operações. Descreveremos a seguir uma solução de busca sequencial baseada em um algoritmo de programação dinâmica proposto por Levenshtein (1966).

Um método conhecido para calcular a distância de edição entre duas cadeias de caracteres A e B, com tamanhos n e m respectivamente, é o algoritmo de programação dinâmica que preenche uma matriz M de tamanho $(n+1) \cdot (m+1)$. Cada célula $M[i, j]$ armazena a distância de edição entre os prefixos de tamanho i e j das cadeias A e B, respectivamente. Esses valores podem ser computados "linha à linha" ou "coluna à coluna" com base na seguinte equação de recorrência:

$$M[i,j] = \begin{cases} max(i,j), \text{se } min(i,j) = 0 \\ \min \begin{cases} M[i-1,j]+1 \\ M[i,j-1]+1 \\ M[i-1,j-1]+\Delta(A[i],B[j]) \end{cases} \text{, senão} \end{cases}$$

Nessa equação, $\Delta(x,y) = 0$ se $x = y$ (sendo x e y caracteres) e $\Delta(x,y) = 1$ caso contrário. Nos exemplos a seguir deve-se considerar que a contagem das células da matriz começa a partir de 0, ou seja, a célula do canto superior esquerdo é referenciada por $M[0,0]$. As condições para as bordas da matriz são $M[0,j] = j$ e $M[i,0] = i$.

		0	1	2	3	4
		ϵ	**c**	**a**	**p**	**a**
0	ϵ	0	1	2	3	4
1	**s**	1	1	2	3	4
2	**a**	2	2	1	2	3
3	**p**	3	3	2	1	2
4	**o**	4	4	3	2	2

Tabela 1: Cálculo de distância de edição entre o prefixo de consulta “capa”, e a sugestão de consulta “sapo” com a matriz de programação dinâmica.

A Tabela 1 demonstra a matriz M resultante para o cálculo de distância entre o prefixo de consulta $p =$ “*capa*” de tamanho $n = 4$, e a sugestão $s =$ “*sapo*” de tamanho $m = 4$. Nela, a primeira coluna e a primeira linha são referentes a uma cadeia de caracteres vazia, representada por ϵ. Para obter a distância de edição entre p e s basta recuperar o valor de $M[n,m] = 2$, ou seja, o valor da célula do canto inferior direito da matriz. A complexidade de tempo para calcular a matriz completa é $O(n \cdot m)$.

Ainda considerando o exemplo presente na Tabela 1, é preciso atentar para a última coluna $M[i,n]$ da matriz. O elemento $M[0,n]$ é referente à distância de edição entre “capa” e uma palavra vazia $\epsilon =$ “”. Já os elementos $M[1,n]$ e $M[2,n]$ são respectivamente referentes à distância entre “capa“ e “s”, e à distância entre “capa” e “sa”, e assim sucessivamente. Para o problema de CATE, basta que qualquer valor de $M[i,n] \leq \tau$ para que uma consulta s seja considerada como resposta para o problema. Devido a isso, uma vez que essa matriz é calculada de cima para baixo e da esquerda para direita, é possível interromper o processamento caso um valor calculado para a última coluna seja menor ou igual a τ, e já considerar o item como resposta. Se $\tau = 3$ por exemplo, quando o elemento $M[2,4] = 3$ é calculado na Tabela 1, a sugestão “sapo” já pode ser considerada como resposta para o prefixo de consulta “capa”, pois $M[2,4] \leq \tau$, e então não é mais necessário calcular as linhas 3 e 4 da matriz. Além disso, *também não é necessário considerar todos os caracteres da sugestão s no cálculo da matriz.* Basta calcular a matriz de distância

de edição entre p, e os $p + \tau$ primeiros caracteres de s, e o problema de CATE pode ser resolvido.

Seja $\mathcal{S}$ um conjunto de sugestões de consulta, p um prefixo de consulta digitado pelo usuário, τ um limiar de erros de digitação tolerados, e *Respostas* um conjunto de sugestões inicialmente vazio. Para cada sugestão s de $\mathcal{S}$ aplica-se o procedimento definido acima. Se s contiver um prefixo similar à p com no máximo τ operações de distância de edição, então é considerada uma sugestão de consulta válida, e adicionada ao conjunto de resposta *Respostas*. Ao final desse processamento, o conjunto *Respostas* representará um subconjunto de $\mathcal{S}$ que contém todos os prefixos similares à p considerando o limiar τ, e portanto, também será considerado a solução do problema de CATE.

Apesar de existirem algoritmos de busca sequencial mais eficientes em comparação ao que foi apresentado acima, esse é importante pois fornece uma base de entendimento para os outros algoritmos mais complexos que estudaremos aqui.

3.2.2 Busca com utilização de índices

Os métodos baseados em índices utilizam estruturas de dados características para diminuir o tempo de processamento das consultas. Há diversas alternativas de estruturas de índices especializadas para o problema de CATE, como por exemplo índices invertidos (Baeza-Yates and Ribeiro-Neto, 1999), árvores de sufixo, ou árvores *Trie*. Dentre essas estruturas, a mais adotada dentre os algoritmos mais recentes da literatura é a árvore *Trie* (Chaudhuri and Kaushik, 2009; Ji et al., 2009; Li et al., 2011; Xiao et al., 2013; Deng et al., 2016; Zhou et al., 2016).

Mesmo que a solução de busca sequencial não seja viável, Costa Xavier e Gama Ferreira (da Costa Xavier, 2019; da Gama Ferreira, 2020) estudaram a possibilidade de combiná-la com métodos baseados em índices, visando obter soluções que economizem memória enquanto mantêm uma boa velocidade de processamento das consultas. Essa abordagem é chamada “busca em dois níveis”, e utiliza índices para reduzir o espaço de busca para então realizar uma busca sequencial nesse espaço reduzido.

3.3 Trie

A estrutura de dados *Trie* (ou “árvore digital”) é uma árvore que armazena um conjunto de palavras e são capazes de recuperar qualquer termo indexado em um tempo proporcional ao seu comprimento, independentemente do tamanho e quantidade total das palavras armazenadas. Os caracteres das palavras armazenadas nessa estrutura pertencem a um alfabeto predefinido Σ, e cada nó da árvore contém um caractere armazenado como chave. A árvore começa com o nó raiz que representa uma palavra vazia ϵ (palavra com 0 caracteres). Cada nó possui um conjunto de nós filhos com tamanho menor ou igual

a $|\Sigma|$, sendo $|\Sigma|$ o tamanho do alfabeto. Para cada nó da árvore é possível caminhar do nó raiz até o mesmo e computar um prefixo, concatenando os caracteres dos nós presentes nesse caminho. Visando a simplicidade, iremos nos referir aos nós pelo seu prefixo correspondente de forma intercambiável.

A inserção de uma nova palavra K começa com uma operação de busca para encontrar o caminho na árvore com o maior número de caracteres em comum com K. Essa busca começa definindo o nó raiz como o nó "atual", denotado por N_{atual}, e a posição atual *pos* na palavra inserida como 1, que representa o primeiro caractere da cadeia. Então, repete-se o procedimento de procurar por algum nó filho de N_{atual} cujo caractere chave é igual a $K[pos]$. Caso esse nó seja encontrado, N_{atual} passará a apontar para ele, e a variável *pos* é incrementada em 1. Caso não seja encontrado, cria-se um novo nó filho em N_{atual} com o valor de $K[pos]$ como sua chave, e N_{atual} passa a apontar para esse novo nó, e *pos* é incrementada em 1. Esse processo é repetido até que o último caractere de K seja alcançado.

A busca por uma palavra K na *Trie* consiste em um procedimento parecido com a inserção. No entanto, a busca resulta em falha no momento em que não se encontra um nó filho de N_{atual} com chave igual a $K[pos]$. Quando o algoritmo segue com sucesso até um nó folha da *Trie*, se todos os caracteres de K foram encontrados, então a busca indica que a palavra foi encontrada. Considerando uma busca linear nos conjuntos de nós filhos de cada nó da *Trie*, o custo de busca e inserção de nova palavra é $O(|\Sigma| \cdot m)$, onde m é o tamanho da palavra. Uma característica importante é que esse custo não depende da quantidade de palavras já inseridas na *Trie*, o que torna essa estrutura uma boa opção para a indexação de cadeias de caractere. Além disso, também possibilita busca eficiente por padrões em uma grande base de texto, como ocorre no problema de CATE.

Além da busca exata por palavras na *Trie*, também é possível realizar busca aproximada, ou seja, tolerante a erros de digitação. A Figura 2 mostra um exemplo de busca exata e busca aproximada pelo prefixo de consulta "capa" em uma *Trie*. A busca exata consiste em visitar apenas os nós da *Trie* cujos prefixos estão contidos exatamente no prefixo de consulta. Já na busca aproximada, é necessário visitar nós "vizinhos" para encontrar prefixos similares que se encontram dentro de um limiar τ. Essa necessidade aumenta bastante a complexidade de processamento das consultas.

Os prefixos da *Trie* que se encontram dentro do limiar τ de tolerância de erros são chamados de "nós ativos", e a busca aproximada é realizada de forma incremental a partir dos nós ativos do prefixo de consulta anterior. A Figura 2 contém um exemplo de apenas alguns nós que podem ser ativos durante o processamento incremental do prefixo de consulta "capa", mas não todos. A forma completa de processamento dos nós ativos será mais detalhada a seguir.

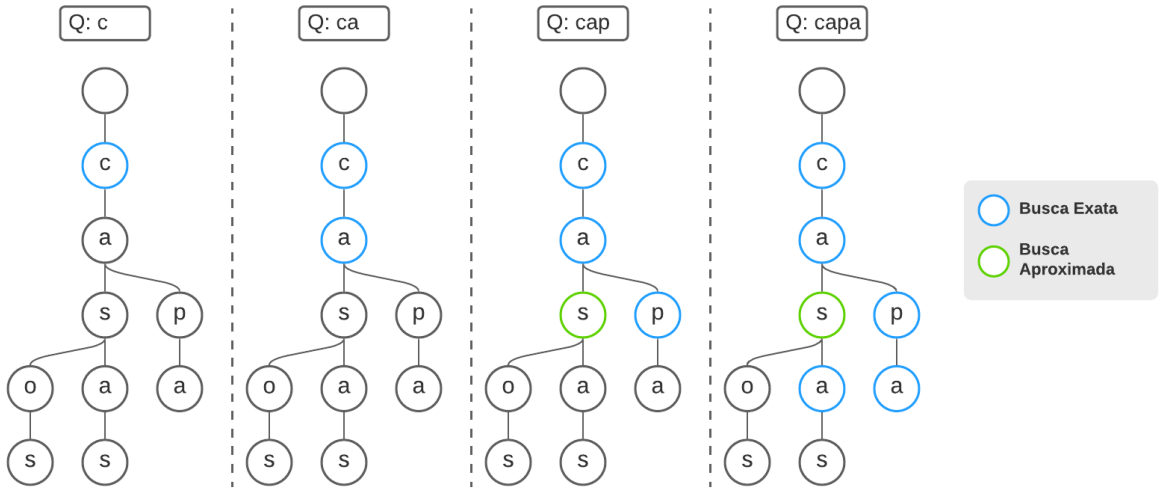


Figura 2: Árvore *Trie* com exemplo de busca exata e aproximada pelo prefixo de consulta "capa". Os nós com o contorno azul representam o caminho realizado pela busca exata, e os nós com contorno verde o caminho realizado pela busca aproximada. O caminho da busca aproximada pode conter nós do caminho da busca exata. Esse exemplo considera $\tau = 1$.

3.3.1 Nós ativos

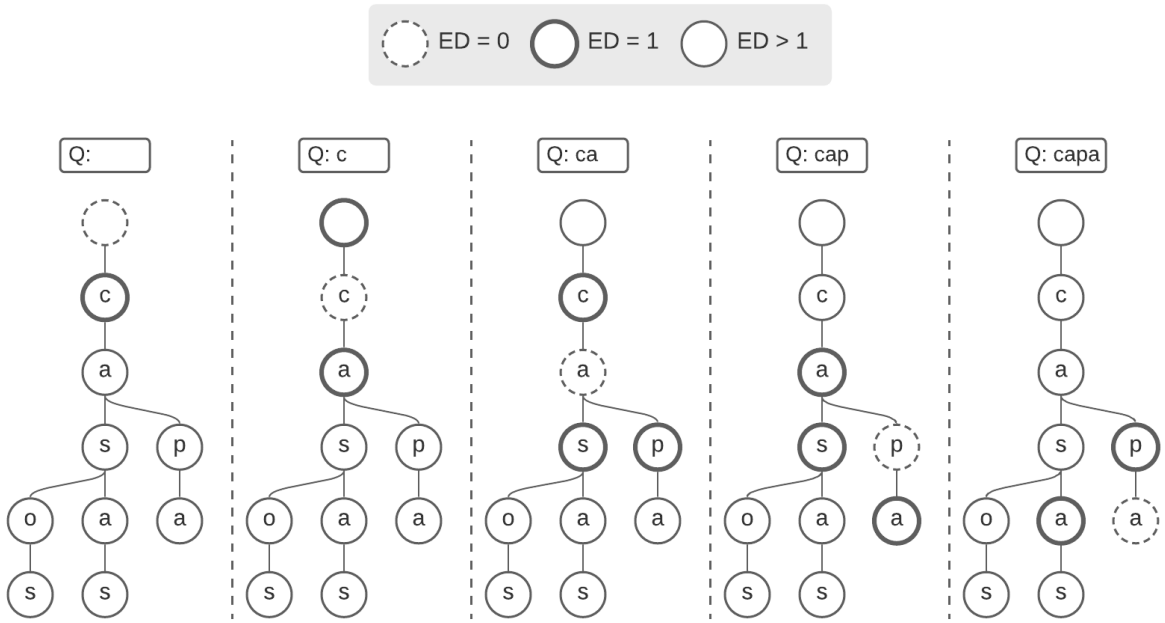


Figura 3: Exemplo de ativação de nós da *Trie* durante o processamento do prefixo de consulta $p =$ "*capa*", considerando $\tau = 1$.

Um nó é considerado ativo quando a distância de edição entre seu prefixo e o prefixo consultado é menor ou igual ao limite τ considerado. O conjunto de nós ativos para um prefixo de consulta p é definido como $V_p = \{p_d | p_d \in P(D) \wedge ed(p_d, p) \leq \tau\}$, onde

$P(D)$ é o conjunto de prefixos da base de dados D, p_d é um prefixo pertencente a $P(D)$ e p é o prefixo de consulta.

A Figura 3 demonstra um exemplo de ativação de nós do índice *Trie* à medida que o processamento do prefixo de consulta "capa" vai ocorrendo. O limiar de erros tolerados é $\tau = 1$. No início do processamento considera-se o termo consultado como sendo uma cadeia de caracteres vazia, cujo valor é representado por ϵ. A partir disso, é necessário ativar todos os nós (prefixos) da *Trie* cujos prefixos possam ser transformados no prefixo de consulta, o qual é ϵ inicialmente. Portanto, os prefixos ϵ (representado pelo nó raiz da *Trie*) e "c" são considerados nós ativos, pois $ed(\epsilon, \epsilon) = 0$ (círculo tracejado na Figura 3) e $ed(\text{“}l\text{”}, \epsilon) = 1$ (círculo de borda grossa). Após isso, o prefixo consultado passa a ser "c", a primeira letra de "capa". Então, é necessário analisar os nós já ativos (e seus respectivos filhos) do prefixo consultado anteriormente para computar os novos nós ativos para "c". A análise de cada nó ativo termina quando se encontra um prefixo fora do limiar τ. Os novos nós ativos computados são ϵ, "c" e "ca", porque $ed(\epsilon, \text{“}c) = 1$, $ed(\text{“}c\text{”}, \text{“}c) = 0$ e $ed(\text{“}ca\text{”}, \text{“}c) = 1$. Todos os outros prefixos além desse possuem distância de edição maior que o limite $\tau = 1$, portanto não são considerados nós ativos. Quando o prefixo de consulta passa a ser "ca", o nó raiz já não é mais um nó ativo, pois $ed(\epsilon, \text{“}ca\text{”}) = 2$, o nó "c" continua como nó ativo porém agora com uma distância igual a 1 em relação a "ca", e dois novos nós são ativados, sendo eles "cas" e "cap", ambos também com distância igual a 1. Esse algoritmo é repetido até chegar no final do prefixo de consulta "capa", cujos nós ativos são "casa" e "capa". O objetivo de computar nós ativos é o de utilizá-los para identificar itens similares ao prefixo consultado.

3.4 ICAN

O algoritmo proposto por Ji et al. (2009) denomina-se "Computando Nós Ativos Incrementalmente" (*Incrementally Computing Active Nodes* – ICAN). Considerando uma consulta de prefixo p com apenas uma palavra-chave, para processar os nós ativos de forma eficiente é necessário computar e armazenar um conjunto de tuplas $\Phi_p = \{\langle n, \xi_n \rangle\}$ tal que (1) cada n é um nó ativo em relação à p com $\xi_n = ed(n, p) \leq \tau$ e (2) quaisquer nós ativos de p estão presentes em Φ_p, que é o *conjunto de nós ativos de p* (Ji et al., 2009). Quando o usuário digita mais um caractere $p + 1$ após ter digitado p, o conjunto Φ_p de nós ativos de p pode ser usado para computar o conjunto Φ_{p+1} de nós ativos da nova consulta.

3.4.1 Descrição do algoritmo

Antes do usuário digitar qualquer caractere, o prefixo de consulta é considerado uma cadeia vazia ϵ, o qual possui um conjunto de nós ativos Φ_ϵ correspondente, que é inicializado como $\Phi_\epsilon = \{\langle n, \xi_n \rangle \mid \xi_n = |n| \leq \tau\}$. Esse conjunto inclui todos os nós n

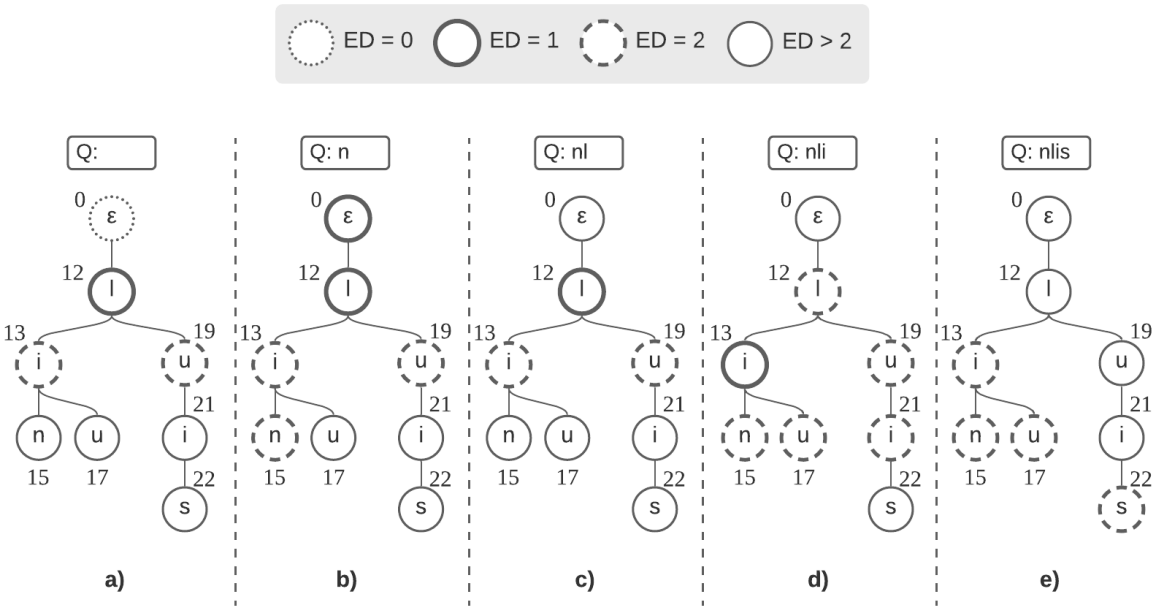


Figura 4: Busca aproximada pelo prefixo de consulta p =“nlis” em uma *Trie*, considerando o limiar de distância de edição $\tau = 2$. **(a)** Inicialização; **(b)** consulta por “n”; **(c)** consulta por “nl”; **(d)** consulta por “nli”; **(e)** consulta por “nlis”

da *Trie* cujos prefixos correspondentes possuem um tamanho $|n|$ que é menor ou igual ao limiar τ. No exemplo da Figura 4, o primeiro passo é inicializar o conjunto $\Phi_\epsilon = \{\langle n_0, 0\rangle, \langle n_{12}, 1\rangle, \langle n_{13}, 2\rangle, \langle n_{19}, 2\rangle\}$

Suponhamos que após o usuário digitar o prefixo de consulta $p_x = c_1c_2...c_x$, o conjunto de nós ativos Φ_{p_x} para p_x é computado. Quando o usuário digitar um novo caractere c_{x+1} e formar um novo prefixo de consulta p_{x+1}, o algoritmo computa o conjunto $\Phi_{p_{x+1}}$ para p_{x+1} a partir de Φp_x da seguinte maneira: inicialmente, inicializa-se Φp_{x+1} como um conjunto vazio; então, para cada tupla $\langle n, \xi_n\rangle$ em Φ_{p_x}, apenas os descendentes de n são examinados como candidatos de nós ativos para p_{x+1}, como é ilustrado na Figura 5. Nesse processo, é preciso atentar para os seguintes casos:

Considerando o nó n: Seja n um nó ativo de p_x. É possível transformar n para p_{x+1} com $\xi_n + 1$ operações de edição ao realizar primeiro uma transformação de n para p_x (com ξ_n operações) e então deletar o último caractere c_{x+1}. Se $\xi_n + 1 \leq \tau$, então adicionamos a tupla $\langle n, \xi_n + 1\rangle$ em Φ_{x+1}. Considerando a tupla $\langle n_0, 0\rangle \in \Phi_\epsilon$ no exemplo da Figura 4, quando o usuário digita o primeiro caractere “n” adiciona-se $\langle n_0, 1\rangle$ em Φ_n, pois é possível realizar uma operação de deleção na letra “n” com distância de edição igual a 1. É importante ressaltar que devido à igualdade $\Phi_{p_x} = \Phi_{c_1c_2..c_x}$, o símbolo Φ_n nesse contexto refere-se ao *conjunto de nós ativos para o prefixo de consulta “n”*, e não possui relação com a variável n utilizada para representar nós da *Trie* anteriormente.

Considerando os filhos do nó n: Para cada filho n_c de um nó n, há dois possíveis

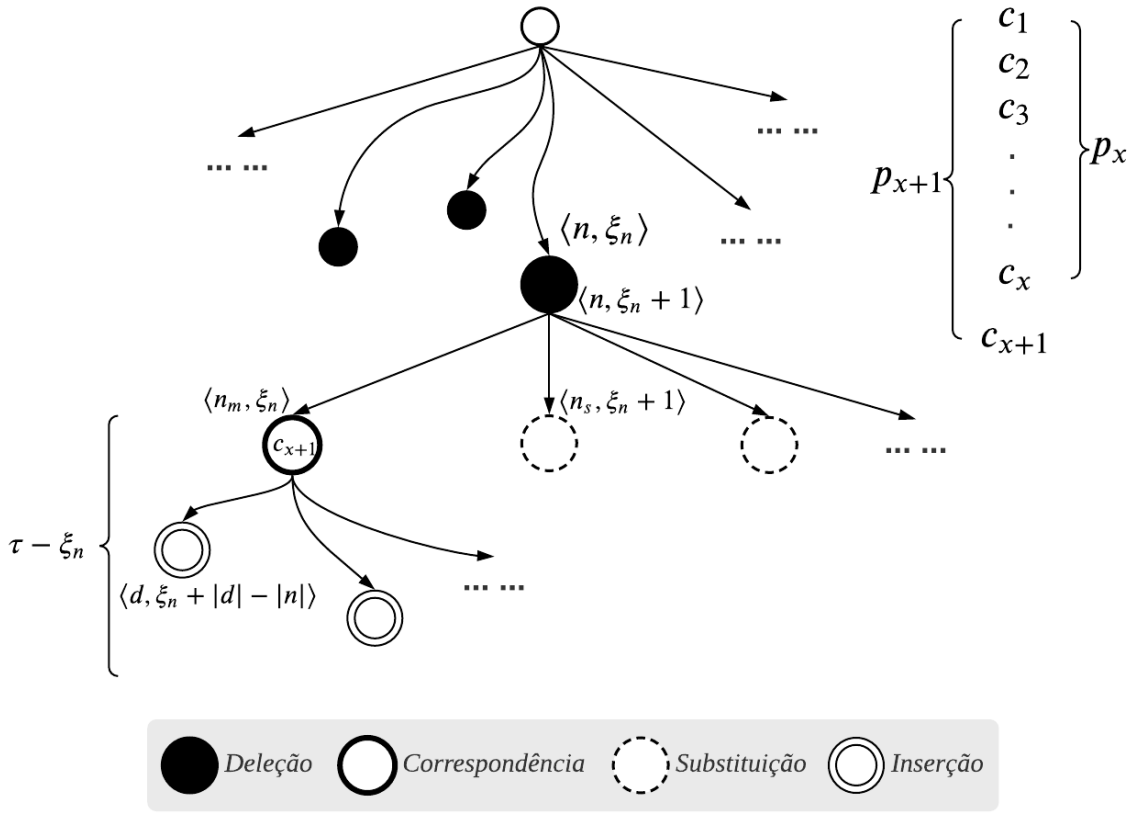


Figura 5: Computação incremental do conjunto de nós ativos $\Phi_{p_{x+1}}$ a partir do conjunto Φp_x. O nó ativo de Φp_{x+1} considerado é $\langle n, \xi_n \rangle$.

casos:

Caso 1: O nó filho n_c tem um caractere diferente de c_{x+1}. A Figura 5 mostra um nó n_s, onde "s" refere-se à operação de *substituição*. É possível transformar n_s em p_{x+1} com $\xi_n + 1$ operações ao transformar primeiro n em p_x (com ξ_n operações) e então substituir o caractere de n_s por c_{x+1}. Se $\xi_n + 1 \leq \tau$ então adiciona-se $\langle n_s, \xi_n + 1 \rangle$ em $\Phi_{p_{x+1}}$. Esse caso corresponde a substituir o caractere de n_s por c_{x+1}. No exemplo da Figura 4, consideremos o caso em que o usuário digita o primeiro caractere "n". Para $\langle n_0, 0 \rangle \in \Phi_\epsilon$, o nó 12 é filho do nó 0 e possui a letra "l". Uma vez que é possível aplicar uma operação de substituição de "l" por "n" com 1 operação edição, adiciona-se $\langle n_{12}, 1 \rangle$ em Φ_n.

Caso 2: O nó filho n_c possui um caractere que corresponde ao caractere c_{x+1} (o caractere de n_c é igual a c_{x+1}). A Figura 5 mostra o nó n_m, onde "m" refere-se à operação de *correspondência*. É possível transformar n_m para p_{x+1} com ξ_n operações ao transformar primeiro n em p_x (com ξ_n operações) e então igualando o caractere de n_m com c_{x+1}. Adiciona-se $\langle n_m, \xi_n \rangle$ em $\Phi_{p_{x+1}}$. Adicionalmente, se $\xi_n < \tau$ então as seguintes operações também são necessárias: para cada descendente d de n_m que dista no máximo $\tau - \xi_n$ letras de n_m, é preciso adicionar $\langle d, \xi_d \rangle$ em $\Phi_{p_{x+1}}$, sendo $\xi_d = \xi_n + |d| - |n_m|$. No exemplo da Figura 4, suponhamos que o usuário digite o primeiro caractere "l". Para

$\langle n_0, 0\rangle \in \Phi_\epsilon$, o nó 12 é filho do nó 0 e possui a letra "l". Uma vez que o caractere do nó 12 corresponde ao caractere "l", $\langle n_{12}, 0\rangle$ é adicionado em Φ_l. Além disso, o nó 13 é filho do nó 13. Uma vez que é possível adicionar o caractere de n_{13} ("i") após o nó 12 com 1 operação de edição, adiciona-se $\langle n_{13}, 1\rangle$ em Φ_l.

Para manter apenas as distâncias de edição mínimas no conjunto de nós ativos, durante a computação de $\Phi_{p_{x+1}}$, toda vez que se deseja adicionar uma tupla $\langle v, \xi_1\rangle$ ao conjunto é possível que o mesmo já contenha a tupla $\langle v, \xi_2\rangle$ para o mesmo nó v no conjunto. Então, se $\xi_1 \geq \xi_2$ a nova tupla não é adicionada. Se $\xi_1 < \xi_2$, então a nova tupla substitui a original no conjunto. Ou seja, para um mesmo nó v da *Trie* no novo conjunto mantém-se apenas a sua menor distância de transformação para o prefixo de consulta p_{x+1}.

3.5 ICPAN

De acordo com o que foi dito na seção 3.3.1, o principal propósito de computar os nós ativos é usá-los para identificar itens similares ao prefixo consultado. Considerando o algoritmo ICAN, o conjunto de nós ativos final para o prefixo de consulta pode acabar sendo muito grande, o que aumenta o tempo de processamento. Diante desse problema Li et al. (2011) propuseram uma melhoria no algoritmo que consiste em podar nós ativos desnecessários, mas ainda mantendo a possibilidade de computar as respostas à consulta. As principais vantagens desse método são a redução do espaço de memória necessário para armazenar o conjunto de nós ativos, e também a redução do tempo de processamento da consulta, pois não é necessário examinar todos os nós ativos. Segue um exemplo de poda que reflete a intuição por trás desse método. Ainda considerando o exemplo da Figura 5, seja "nl" prefixo consultado com um limiar $\tau = 2$, onde $\Phi_{nl} = \{\langle n_{12}, 1\rangle, \langle n_0, 2\rangle, \langle n_{13}, 2\rangle, \langle n_{19}, 2\rangle\}$. No entanto, apesar de n_{13} ("li") e n_{19} ("lu") serem nós ativos, não é necessário mantê-los, pois é possível utilizar o nó ativo n_{12} ("l") para computar as palavras "li" e "lu". Ou seja, é necessário manter apenas o nó ativo "l" para computar o mesmo conjunto de itens similares ao prefixo "nl" consultado.

3.5.1 Nós pivô ativos

Seja n um nó ativo de p. Se para qualquer transformação de n para p com $ed(n, p)$ operações a operação no último caractere de n não for uma correspondência (caracteres iguais), e for possível deletar o último caractere de n, o nó n não é mantido, e sim o seu nó pai. Assim, surge o conceito de *nó pivô ativo* (Li et al., 2011).

Considerando novamente o prefixo "nl" no exemplo da Figura 5, sabe-se que "l" é um nó ativo, e que "li" e "lu" são nós ativos cuja distância de edição entre "nl" é igual a 2, e seus últimos caracteres não correspondem aos caracteres do prefixo "nl". É

possível derivar esses dois nós ativos a partir do nó "l" ao inserir os caracteres "i" e "u", respectivamente. Além disso, também é possível computar as palavras similares "li" e "lu" ao visitar os nós folhas que são descendentes de "l". Portanto, não é necessário manter esses dois nós ativos. Nesse exemplo, o nó "l" é chamado *nó pivô ativo*, pois serve como "pivô" para computar outras palavras similares à consulta sem a necessidade de ativar mais nós. Para um prefixo de consulta p, um nó n é um nó pivô ativo de p em relação ao limiar de distância de edição τ, se e somente se n for um nó ativo de p e se existir uma transformação de p em n com $ed(n, p)$ operações de edição, e a operação no último caractere de n for uma correspondência. Ou seja, a operação no último caractere de n não é nem uma deleção $ed(n, p) \neq ed(n', p)+1$ e nem uma substituição $ed(n, p) \neq ed(n', p')+1$, onde n' e p' são respectivamente os prefixos de n e p que não contêm o último caractere.

Para computar de forma eficiente os nós pivô ativos, Li et al. (2011) desenvolveram o algoritmo "Computando Nós Pivô Ativos Incrementalmente" (*Incrementally Computing Pivotal Active Nodes* – ICPAN). Sendo p_x um prefixo de consulta, o objetivo é computar e armazenar um conjunto de 4-uplas $\Psi_{p_x} = \{\langle n, \xi_n^{p_x}, p_i, \xi_n^{p_i}\rangle\}$. Em cada 4-upla do conjunto, n é um nó pivô ativo de p_x, e p_i *é um prefixo de* p_x *cujos últimos caracteres são iguais aos últimos caracteres de* n. Se esse prefixo não existir, então $p_i = \epsilon$. Se houver mais de uma possibilidade para p_i, então seleciona-se aquele com o menor número de caracteres. O quarto elemento da tupla, $\xi_n^{p_i} \leq \tau$ é a distância de edição entre o nó n e o prefixo p_i com operações de correspondência entre os últimos caracteres de n e p_i. Já o segundo elemento da tupla, $\xi_n^{p_x} \leq \tau$ é a distância de edição entre o nó n e o prefixo de consulta p_x, obtida transformando primeiro n em p_i, e então inserindo os caracteres restantes da diferença entre p_x e p_i. A partir disso, deriva-se a igualdade $\xi_n^{p_x} = \xi_n^{p_i} + |p_x| - |p_i|$.

3.5.2 Descrição do algoritmo

Antes do usuário digitar qualquer caractere, o prefixo de consulta é considerado uma cadeia vazia ϵ, e então seu conjunto de nós pivô ativos correspondente é inicializado como $\Psi_\epsilon = \{\langle r, 0, \epsilon, 0\rangle\}$, onde r é o nó raiz, já que a raiz é o único nó pivô ativo para ϵ de acordo com as definições estabelecidas anteriormente. Considerando ainda o mesmo exemplo da Figura 5, no qual o usuário digita o prefixo de consulta "nlis" caractere a caractere, e o limiar de distância de edição é $\tau = 2$, o primeiro passo é inicializar o conjunto como $\Psi_\epsilon = \{\langle n_0, 0, \epsilon, 0\rangle\}$, sendo n_0 o nó raiz.

Suponhamos que após o usuário digitar o prefixo de consulta $p_x = c_1c_2...c_x$ o conjunto de nós ativos Ψ_{p_x} para p_x é computado. Quando o usuário digitar um novo caractere c_{x+1} e formar um novo prefixo de consulta p_{x+1}, o algoritmo computa o conjunto $\Psi_{p_{x+1}}$ para p_{x+1} a partir de Ψp_x da seguinte maneira: inicialmente, inicializa-se Ψp_{x+1} como um conjunto vazio; então, para cada 4-upla $\langle n, \xi_n^{p_x}, p_i, \xi_n^{p_i}\rangle$, apenas os descendentes de n são examinados como candidatos de nós pivô ativos para p_{x+1}, como é ilustrado na

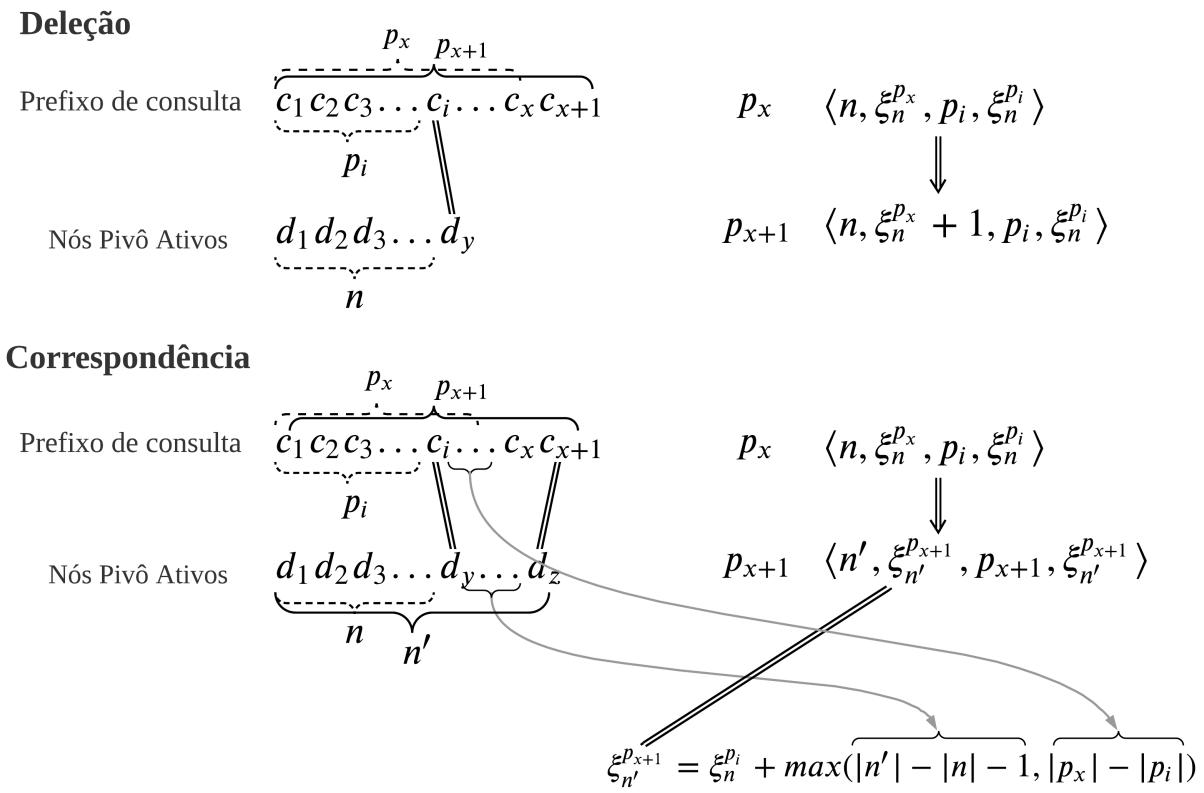


Figura 6: Computação incremental de nós pivô ativos, sendo $c_i = d_y$ e $c_{x+1} = d_z$.

Figura 6. Nesse processo, é preciso atentar para os seguintes casos (todos os exemplos a seguir também consideram a árvore *Trie* da Figura 4):

Considerando o nó n: Seja n um nó pivô ativo de p_x. É possível transformar n em p_{x+1} com $\xi_n^{p_x}+1$ operações de edição ao primeiro transformar n em p_x (com $\xi_n^{p_x}$ operações) e então deletar o último caractere c_{x+1}. Se $\xi_n^{p_x}+1 \leq \tau$, então adiciona-se $\langle n, \xi_n^{p_x}+1, p_i, \xi_n^{p_i}\rangle$ ao conjunto $\Psi_{p_{x+1}}$. Se o usuário digitar "n" por exemplo, uma vez que é possível realizar uma operação de deleção do caractere "n" com distância de edição igual a 1, a partir da 4-upla $\langle n_0, 0, \epsilon, 0\rangle \in \Psi_\epsilon$ é possível adicionar $\langle n_0, 1, \epsilon, 0\rangle$ em Ψ_n.

Considerando os descendentes do nó n: Consideremos os descendentes de n *que possuem o caractere* c_{x+1} e que não distem de n mais do que $\tau - \xi_n^{p_i} + 1$ passos. Para um descendente n' nessa condição, é possível transformar n' em p_{x+1} com os seguintes passos: (1) transformar n em p_i; (2) transformar os caracteres após n e antes de n' nos caracteres $c_{i+1}...c_x$; e (3) verificando a operação de correspondência do caractere n' com c_{x+1}. Isso implica que é possível transformar n' em p_{x+1} com $\xi_{n'}^{p_{x+1}} = \xi_n^{p_i} + max(|n'|-|n|-1, |p_x|-|p_i|)$ operações. Se $\xi_{n'}^{p_{x+1}} \leq \tau$, então adiciona-se $\langle n', \xi_{n'}^{p_{x+1}}, p_{x+1}, \xi_{n'}^{p_{x+1}}\rangle$ em $\Psi_{p_{x+1}}$. Por exemplo, se o usuário digitar o primeiro caractere "n" e considerando $\langle n_0, 0, \epsilon, 0\rangle \in \Psi_\epsilon$, uma vez que o nó n_{15}("lin") corresponde a "n", então adiciona-se $\langle n_{15}, 2, \text{“}lin\text{”}, 2\rangle$ em Ψ_n.

Semelhantemente ao ICAN, também é necessário manter a distância de edição mínima no conjunto de 4-uplas. Durante o processamento do novo conjunto $\Psi_{p_{x+1}}$, para

um mesmo nó n, mantém-se somente a distância de edição entre o nó n e o prefixo de consulta p_{x+1}. Toda vez que se deseja adicionar uma 4-upla $\langle n, \xi_n^{p_{x+1}}, p_i, \xi_n^{p_i} \rangle$ ao conjunto é possível que o mesmo já contenha a 4-upla $\langle n, \xi'^{p_{x+1}}_n, p_j, \xi'^{p_j}_n \rangle$ para o mesmo nó n no conjunto. Se $\xi'^{p_{x+1}}_n > \xi_n^{p_{x+1}}$, então a nova 4-upla *não é adicionada*. Se $\xi'^{p_{x+1}}_n = \xi_n^{p_{x+1}}$ e $|p_j| < |p_i|$, então a nova 4-upla substitui a 4-upla original (isso garante que apenas o prefixo com o menor número de caracteres seja mantido, como descrito em 3.5.1). Por fim, se $\xi'^{p_{x+1}}_n < \xi_n^{p_{x+1}}$, então a nova 4-upla *substitui a original que já estava no conjunto.*

O ICPAN possui uma situação a mais para tratar em comparação ao ICAN: a remoção de nós ativos que não são pivôs. Para cada 4-upla $\langle n, \xi_n^{p_{x+1}}, p_i, \xi_n^{p_i} \rangle$, **se** $\mathbf{p_i \neq p_{x+1}}$, então é feita uma verificação para cada nó "ancestral" $n_a \neq n$ de n. Se $\langle n_a, \xi_{n_a}^{p_{x+1}}, p_a, \xi_{n_a}^{p_a} \rangle \in \Psi_{p_{x+1}}$ *e* $\xi_{n_a}^{p_a} + max(|p_{x+1}| - |p_a|, |n| - |n_a|) < \xi_n^{p_{x+1}}$, então é preciso remover $\langle n, \xi_n^{p_{x+1}}, p_i, \xi_n^{p_i} \rangle$ do conjunto. É necessário que isso aconteça pois nessas condições há uma transformação do nó n para p_{x+1} com uma distância de edição menor do que $\xi_n^{p_{x+1}}$ sem que haja correspondência no último caractere do nó n, ou seja, n não é um nó pivô ativo do prefixo de consulta p_{x+1}.

Tanto o ICAN quanto o ICPAN produzem no final do processamento um conjunto de nós ativos. Os dois algoritmos consideram que cada nó folha n_{folha} da árvore *Trie* possui uma lista invertida L de "*IDs*" dos itens que possuem o prefixo referente ao nó n_{folha}. Então, para cada nó ativo do conjunto final obtido, obtém-se todos os nós folhas dentre seus descendentes. *A lista de IDs resultante das interseções entre as listas de todos esses nós folhas obtidos é a lista de itens similares ao prefixo consultado, ou seja, a resposta para o problema de CATE.*

3.6 Busca em dois níveis

Uma possível estratégia de busca em dois níveis para o problema de CATE consiste indexar somente parte dos itens da base de dados em uma árvore *Trie*, e combinar um algoritmo de busca aproximada para essa estrutura (primeiro nível) com o algoritmo de busca sequencial apresentado na seção 3.2.1 (segundo nível). Para isso, o prefixo de consulta também precisa ser divido em duas partes que são processadas por cada nível. Nessa abordagem, uma vez que os itens são indexados apenas parcialmente, o conjunto de nós ativos obtidos no final da execução do algoritmo do primeiro nível recebe uma nova função. Normalmente, tal conjunto é utilizado para computar os itens similares ao prefixo consultado, mas nessa abordagem de dois níveis, passa a ser usado como filtro para a busca sequencial do segundo nível, que é realizada somente nos itens referenciados por esses nós ativos.

3.7 Busca binária com elementos repetidos na coleção

Considerando a abordagem em dois níveis, neste trabalho analisamos a hipótese de que é possível utilizar não somente busca sequencial no segundo nível, mas também busca binária em alguns casos especiais, que são nós ativos com uma determinada característica explicada com mais detalhes no capítulo 4. Nesses casos, é necessário realizar buscas binárias nas listas invertidas dos nós folha descendentes dos nós ativos.

O melhor algoritmo para busca em vetores ordenados é o de busca binária. Esse método requer que a coleção em que se fará a busca esteja ordenada. A busca binária padrão compara o valor buscado com o valor do elemento presente exatamente no meio da coleção. Se eles não forem iguais, a metade da coleção na qual o valor buscado não pode ser encontrado é desconsiderada, e a busca continua na metade restante, considerando novamente o valor do elemento do meio. Esse processo se repete até que o valor seja encontrado. Se a busca terminar e a metade restante estiver vazia, a busca não encontrou o valor. A complexidade da busca binária é $O(\log n)$, enquanto a da busca sequencial é $O(n)$.

Porém, em algumas situações em que coleção possui elementos duplicados, pode ser necessário encontrar não a posição de um valor duplicado qualquer, mas a posição da primeira ocorrência do valor (limite inferior) juntamente com a posição da sua última ocorrência na coleção (limite superior). Para isso é necessário realizar primeiro uma busca binária especializada para encontrar o limite inferior, e depois realizar novamente outra busca binária para encontrar o limite superior. Após essas duas buscas, será obtido um intervalo $R = [limInferior, limSuperior]$ referente às posições da coleção nas quais se pode encontrar o valor duplicado que foi buscado. Nas buscas binárias por limite inferior e superior utilizadas no algoritmo 7 da seção 4.6, quando o valor buscado não é encontrado consideramos os limites inferior e superior como sendo negativos, ou seja, $R = [-1, -1]$ por exemplo.

4 Método proposto

Neste capítulo estudaremos uma abordagem em dois níveis para a complementação automática de consulta tolerante a erros que usa o algoritmo *ICPAN* no primeiro nível, e usa tanto pesquisa sequencial quanto binária no segundo nível. O primeiro nível serve como um filtro que seleciona candidatos à resposta para serem processados no segundo nível, responsável por determinar quais candidatos são qualificados como resposta final.

O intuito dessa abordagem é reduzir o uso de memória e ao mesmo tempo manter o desempenho quanto ao tempo de processamento da consulta. Os sistemas de busca podem se beneficiar dessa redução pois ela possibilita diminuir os custos com memória mantendo o mesmo conjunto de dados, ou então pode permitir o aumento do conjunto de dados sem que haja um grande impacto na quantidade de memória usada pelo método.

Seja $\mathcal{D}$ a base de dados considerada nos parágrafos a seguir, composta pelos seguintes itens: *{insects, integer, integral, integrity, intellect, intelligent, invest, invested, investigate, telepathic, telepathy, telephone, telephoto, teleport, teleprompter}*. Atribuímos um valor de *id* a cada item, partindo do número 0, como é possível observar na tabela de itens da Figura 7. A etapa de indexação considera que todos os itens estão ordenados em ordem lexicográfica crescente, e para que os exemplos fiquem mais simples cada item contém apenas uma palavra.

4.1 Indexação

Seja w a cadeia de caracteres de um item, $|w|$ o tamanho da cadeia, e $w[i]$ uma indicação do i-ésimo caractere de w, com $i = 1$ referindo-se ao primeiro caractere. A expressão $w[i..j]$ representa uma subcadeia de caracteres de w começando na i-ésima posição e terminando na j-ésima. Consideramos também que se $j > |w|$, então $w[i..j] = w[i..|w|]$, e também que $w[i..]$ representa a subcadeia que começa no i-ésimo caractere, e termina no último caractere de w. Sendo assim, para $w =$ "*sapato*", temos que $w[3..27] =$ "*pato*" e $w[4..] =$ "*ato*", por exemplo.

Como parte da implementação da estrutura proposta, indexamos as descrições textuais dos itens em uma árvore *Trie* $\mathcal{T}$ mantida em memória. Seja também λ a altura máxima permitida para $\mathcal{T}$, considerando que o nível do nó ϵ (nó $N0$ na Figura 7) é igual a 0. Dessa forma, o nó $N7$ tem altura igual a 3, por exemplo. Para cada cadeia w de um item inserimos em $\mathcal{T}$ apenas o prefixo $Prefixo_\lambda(w)$, onde $Prefixo_\lambda(s) = s[1..\lambda]$ é uma função que retorna os λ primeiros caracteres da cadeia s passada como parâmetro.

Cada nó n em $\mathcal{T}$ contém um caractere $n.caractere$, um número de identificação

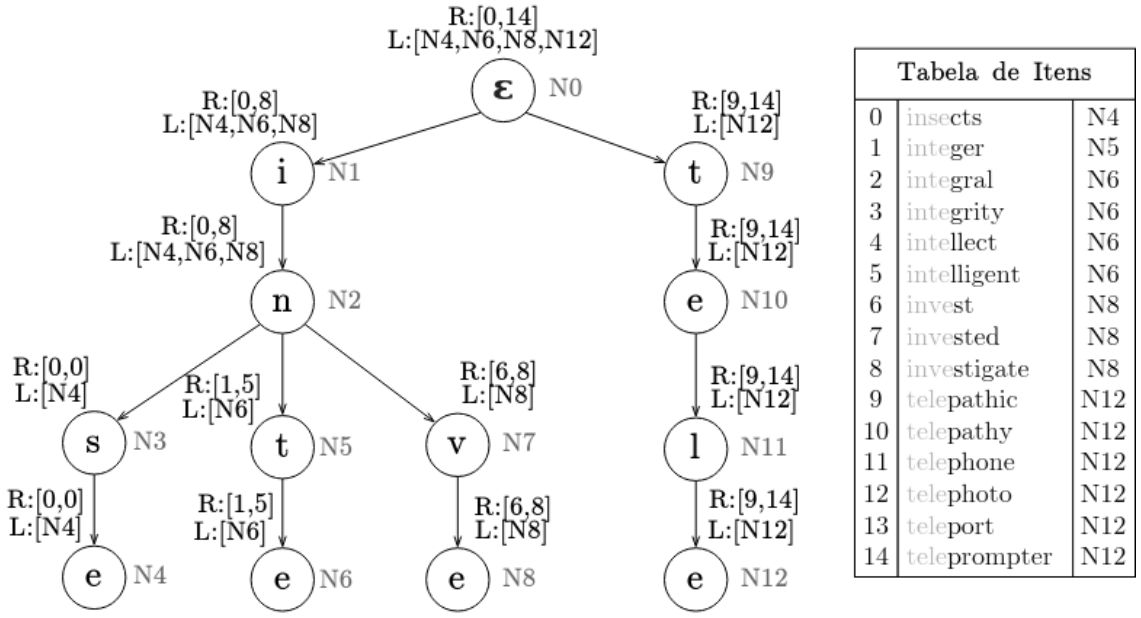


Figura 7: Árvore *Trie* com os prefixos de tamanho $\lambda = 4$ indexados, na qual cada nó possui intervalos R de *ids* e listas L de nós folha dentre os seus descendentes.

$n.id$, e a sua altura $n.altura$ correspondente na árvore (sendo a altura do nó ϵ igual a 0). Também possui um intervalo $n.R = [menor, maior]$ no qual $menor$ e $maior$ representam respectivamente o menor e o maior id dos itens em $\mathcal{T}$ que compartilham o mesmo prefixo p_n obtido ao caminhar partindo do nó ϵ até n. Esse intervalo é análogo à lista invertida mencionada no fim da seção 3.5.2. Para uma subárvore com raiz no nó n, todos os itens nessa subárvore irão compartilhar o mesmo prefixo p_n. Objetivando a simplicidade, iremos nos referir a n pelo seu prefixo p_n correspondente de forma intercambiável, assim como foi feito na seção 3.3.

Por fim, cada nó n possui uma lista $n.L$ que contém todos os nós folha de sua subárvore. Quando o próprio n é uma folha, não possui subárvore, então a lista conterá apenas o próprio n, ou seja, $n.L = [n.id]$.

A Figura 7 mostra uma *Trie* $\mathcal{T}$ após a indexação dos $\lambda = 4$ primeiros caracteres de cada item em $\mathcal{D}$. Também mantém-se uma estrutura de dicionário $\mathcal{H}$, representada pela "Tabela de Itens", que mapeia o valor do id de um item ao texto do restante da sua descrição, ou seja, $w[5..]$ (texto após o 4º caractere).

Essa estrutura é utilizada para reconstruir a descrição completa de um item a partir de qualquer nó de prefixo p. Apesar da Figura 7 mostrar o texto completo de cada item na "Tabela de Itens", os 4 primeiros caracteres em cinza claro são na verdade obtidos ao caminhar por $\mathcal{T}$. Para possibilitar essa reconstrução, também guardamos em $\mathcal{H}$ o id do nó correspondente ao prefixo $Prefixo_\lambda(w)$ do item. Supondo por exemplo a reconstrução de "*investigate*", com $id = 7$ em $\mathcal{H}$, basta processar o prefixo p do nó $N8$, que é $p =$ *"inve"*, como mostra a Figura 7, e então concatenar p com "*stigate*", que é o conteúdo

presente em $\mathcal{H}$ para a chave $id = 7$.

Algoritmo 1 Construção de um índice *Trie* $\mathcal{T}$ a partir de sugestões de consultas de uma base $\mathcal{D}$

```
1: function ConstruirIndiceTrie(D, H, T)
2:     for Q ∈ D do
3:         item.noPrefixo ← inserirPrefixo(T, Prefixo_λ(Q.texto), Q.id)
4:         if |Q.texto| < λ then
5:             item.restante ← ϵ
6:         else
7:             item.restante ← Q.texto[(λ + 1)..]
8:         H[Q.id] ← item
9:     construirListasDeFolhas(T)
```

O Algoritmo 1 descreve a construção de um índice *Trie* a partir de uma base de sugestões de consulta. Esse algoritmo tem como parâmetro: $\mathcal{D}, \mathcal{H}, \mathcal{T}$, os quais representam, respectivamente, a base de dados com as sugestões de consulta que serão indexadas, a estrutura de dicionário mencionada anteriormente, e a árvore *Trie* ainda vazia. Esse algoritmo assume que os itens da base $\mathcal{D}$ estão em ordem lexicográfica crescente. O laço de repetição da linha 2 define o procedimento feito para cada sugestão de consulta Q da base $\mathcal{D}$. Na linha 3, chama-se a função $inserirPrefixo$ (que será detalhada a seguir), a qual insere o prefixo $Prefixo_\lambda(Q.texto)$ (primeiros λ caracteres do texto da sugestão de consulta) na *Trie*, e retorna o nó da *Trie* referente ao prefixo inserido. Então, esse nó é atribuído ao atributo $noPrefixo$ do $item$. Na linha 4, há uma condição para verificar se o tamanho do texto de Q é menor que λ. Se for, então o atributo $restante$ do $item$ recebe o valor de texto vazio ϵ, na linha 5. Senão, recebe o valor do restante do texto de Q após o λ-ésimo caractere, na linha 7. Com o item já montado, associa-se a chave $Q.id$ com o $item$ no dicionário $\mathcal{H}$ na linha 8, e o laço de repetição segue para a próxima sugestão de consulta de $\mathcal{D}$, se houver. Após o termino do laço, chama-se a função $construirListasDeFolhas$ na linha 9. Essa função itera sobre cada nó n da *Trie* $\mathcal{T}$, preenchendo as listas $n.L$ de cada nó, as quais contêm todos os nós folha com raiz em n.

Algoritmo 2 Inserção de um prefixo de sugestão de consulta em $\mathcal{T}$

```
1: function inserirPrefixo(T, prefixo, id)
2:     raiz ← obterRaiz(T)d
3:     n ← raiz
4:     raiz.R.maior = id
5:     for caractere ∈ prefixo do
6:         n ← n.inserirFilho(caractere, id)
7:     n.marcadoFolha ← Verdadeiro
8:     n.L.inserir(id)
9:     return n
```

O Algoritmo 2 descreve a inserção de um prefixo de sugestão de consulta (o qual pode na verdade ser a consulta inteira). Esse algoritmo tem como parâmetro: $\mathcal{T}, prefixo, id$, os quais representam a árvore *Trie*, o prefixo de sugestão de consulta, e a identificação numérica da consulta, respectivamente. Na linha 4, atualiza-se o valor *maior* do intervalo $raiz.R = [menor, maior]$ para ser igual ao id da sugestão de consulta que está sendo inserida. Então, cada caractere do $prefixo$ (linha 5) é inserido na *Trie* um a um na (linha 6). A cada inserção, o nó n é atualizado com último nó que foi inserido ou apenas percorrido (caso todo o caminho já exista na árvore). Após essas inserções, na linha 8 o nó n é marcado como folha. Por fim, o id da sugestão de consulta é inserido na lista L do nó n na linha 9 e a função retorna n na linha 10.

4.2 Algoritmo geral da abordagem em dois níveis

Como descrito na seção 4.1, somente os λ primeiros caracteres de cada sugestão de consulta da base são indexados na *Trie*. Então, devido a esse fato, temos a premissa de que somente os λ primeiros caracteres de um prefixo de consulta p devem ser considerados na computação do conjunto de nós ativos. Com tal conjunto computado, cada nó é visitado, e recupera-se sua lista de nós folha. Para cada uma das listas invertidas desses nós folha, é possível realizar a busca sequencial apresentada na seção 3.2.1.

Algoritmo 3 Algoritmo geral do processamento em dois níveis

```
1: function PROCESSARCONSULTADOISNIVEIS(T, p, τ)
2:     if |p| + τ ≤ λ then return MetodoCATE(T, p, τ)
3:     sugestoes ← ∅
4:     nosAtivos ← computarNosAtivos(T, Prefixo_λ(p), τ)        ▷ Primeiro Nível
5:     for n ∈ nosAtivos do                                     ▷ Segundo Nível (até a linha 6)
6:         for n_folha ∈ n.L do
7:             sugestoes ← sugestoes ∪ buscaSequencial(p, n_folha.R, τ)
8:     return sugestoes
```

O Algoritmo 3 é uma representação geral da abordagem em dois níveis para o problema de CATE. Ele tem como parâmetros $\mathcal{T}, p$ e τ, os quais representam um índice *Trie*, um prefixo de consulta, e o limiar de erros de digitação, respectivamente. O algoritmo tem início na linha 2 com uma verificação da condição $|p| + \tau \leq \lambda$, ou seja, se o tamanho do prefixo de consulta mais o número de erros de digitação tolerados é menor do que a altura máxima do índice *Trie*. Quando essa condição é verdade é possível utilizar somente o algoritmo de CATE (representado pela função *MetodoCATE*) para computar todos os resultados para p, sem a necessidade de ativar o segundo nível. O prefixo de consulta pode sofrer no máximo τ inserções como erros de digitação, e se esse tamanho máximo ($|p| + \tau$) estiver dentro do limite λ, os nós indexados na *Trie* são o suficiente para responder à

consulta. Essa lógica estará presente também nos Algoritmos 6 e 7, com o método ICPAN sendo executado.

Caso a condição seja falsa, o algoritmo segue com o processamento em dois níveis. O conjunto de resposta *sugestoes* é iniciado como vazio na linha 3, o qual conterá no fim da execução as sugestões de consultas com prefixos similares a p considerando uma diferença de distância de edição de no máximo τ. Na linha 4, é executada a função $computarNosAtivos$, que computa o conjunto de nós ativos da *Trie* $\mathcal{T}$ para o prefixo $Prefixo_\lambda(p) = p[1..\lambda]$ considerando o limiar τ. O algoritmo utilizado para tal computação pode ser, em tese, qualquer um baseado em processamento de nós ativos de árvores *Trie*, como por exemplo o ICAN, ICPAN, ou BEVA. O conjunto final de nós ativos é armazenado na variável *nosAtivos*. Essa etapa caracteriza o primeiro nível.

A linha 5 caracteriza o início do processamento do segundo nível. Nela há um laço de repetição que itera sobre cada nó ativo do conjunto *nosAtivos*, e na linha 6, há outro laço que itera sobre cada nó folha com raiz em n. Então, na linha 7, a função *buscaSequencial* determina quais dentre as sugestões de consultas referenciadas pelo intervalo R de n_{folha} são respostas válidas, e então essas sugestões são unidas com o conjunto final de respostas. O algoritmo dessa função é o descrito na seção 3.2.1. Por fim, na linha 8 o conjunto de respostas é retornado.

4.3 Primeiro nível

O algoritmo de busca utilizado no primeiro nível será o ICPAN, proposto por Li et al (Li et al., 2011). Como descrito no capítulo 3 o ICPAN utiliza um algoritmo incremental que calcula os prefixos semelhantes a um prefixo consultado, tolerando uma quantidade de erros de digitação previamente definida como o limite de distancia de edição τ. Uma característica desse algoritmo é a "ativação" de nós da *Trie* que correspondem a um prefixo que pode ser obtido considerando erros no prefixo consultado.

Ao considerar a utilização do ICPAN no primeiro nível, é preciso atentar para dois pontos: (1) uma vez que o primeiro nível serve apenas como filtro para o processamento do segundo nível, o algoritmo computará nós (pivô) ativos referentes a prefixos semelhantes a uma parte $p[1..\lambda]$ de um prefixo de consulta p, e não semelhantes a p por completo. Esse conjunto será referido como $\Psi_{p[1..\lambda]}$ daqui em diante; (2) uma premissa da concepção original desse algoritmo é que todo o texto da sugestão de consulta estará indexado na *Trie*, o que não acontece na abordagem em dois níveis.

Dessa maneira, o propósito original de utilização dos nós ativos foi modificado. Alguns nós importantes para o processamento no segundo nível acabam sendo desativados durante o processamento incremental do ICPAN, e deixam de estar presentes em $\Psi_{p[1..\lambda]}$. A partir dessa situação observamos em nossa pesquisa que o conteúdo do conjunto $\Psi p[1..\lambda]$

resultante da computação original do ICPAN *não é suficiente para resolver completamente o problema de CATE na abordagem em dois níveis.*

4.3.1 Conjunto de nós folha virtuais ativos

Devido ao problema de nós desativados mencionado acima, faz-se necessário um mecanismo que preserve esses nós ativos importantes para que eles também estejam no conjunto final de nós ativos de $p[1..\lambda]$. O mecanismo proposto neste trabalho para resolver esse problema consiste em computar, simultaneamente ao conjunto de nós ativos Ψ do ICPAN, um conjunto ψ de "nós folha virtuais ativos" auxiliar que manterá esses nós importantes que acabariam sendo desativados durante o processamento incremental dos nós ativos.

Como foi descrito na seção 3.5.2, a computação do conjunto de nós ativos para um prefixo de consulta $p = c_1c_2c_3...c_n$ ocorre de forma incremental: inicializa-se um conjunto com os nós ativos para a cadeia vazia ϵ, representado por Ψ_ϵ; então, quando o primeiro caractere c_1 de p é processado, os nós de Ψ_ϵ e alguns de seus descendentes são examinados e, dependendo de algumas condições, são inseridos no conjunto de nós ativos para a cadeia de caracteres c_1, representado por Ψ_{c_1}; quando o caractere c_2 é processado, analogamente ao passo anterior os nós de Ψ_{c_1} e alguns de seus descendentes são examinados para determinar se serão inseridos no conjunto de nós ativos para a cadeia de caracteres c_1c_2, representado por $\Psi_{c_1c_2}$. Esse processo continua até que se obtenha o conjunto $\Psi_p = \Psi_{c_1c_2c_3...c_n}$, o qual será chamado de "conjunto final". Os conjuntos gerados antes do final serão chamados "conjuntos intermediários".

Na abordagem de dois níveis proposta neste trabalho, em que apenas os λ primeiros caracteres das sugestões de consulta são indexados, a computação de nós ativos ocorre somente até o conjunto intermediário $\Psi_{p[1..\lambda]}$, ou seja, o conjunto de nós ativos para os λ primeiros caracteres de p. No entanto, existem alguns nós em especial que só podem ser encontrados no conjunto final, quando o processo continua após o λ-ésimo caractere. Quando se processa somente $p[1..\lambda]$, esses nós em especial são inseridos em alguns dos conjuntos intermediários, como em $\Psi_{c_1c_2}$, mas já não permanecem mais a partir de $\Psi_{c_1c_2c_3}$ em diante, por exemplo, e também não estarão inclusos no conjunto $\Psi_{p[1..\lambda]}$, que é utilizado para o processamento do segundo nível. A consequência desse problema é que algumas sugestões de consulta que só poderiam ser obtidas através desses nós não são analisadas no segundo nível, e então o algoritmo em dois níveis proposto deixa de trazer todos os resultados que deveria. Uma vez que ainda há texto do prefixo de consulta após o λ-ésimo caractere para considerar no cálculo de distância de edição que irá determinar se uma sugestão deve ser sugerida, esses nós precisam ser mantidos de alguma forma para também serem analisados no segundo nível.

Denominamos esses nós que acabam "se perdendo" durante a computação do con-

junto de nós ativos como "nós folha virtuais ativos". Seja Ψ_{c_x} o conjunto de nós ativos para um prefixo de consulta $p_x = c_1c_2...c_x$ digitado anteriormente, c_{x+1} um novo caractere digitado pelo usuário que resulta no novo prefixo de consulta $p_{x+1} = c_1c_2...c_xc_{x+1}$. Diante disso, é necessário examinar cada nó de Ψ_{c_x} e alguns de seus descendentes para computar o conjunto $\Psi_{c_{x+1}}$. Um nó é considerado como folha virtual ativo se, logo após examiná-lo em Ψ_{c_x}, não ocorre nenhuma inserção ou atualização em $\Psi_{c_{x+1}}$, pois não foi possível realizar nenhuma substituição ou inserção de acordo com as regras de manutenção de conjunto de nós ativos do ICPAN definidas na seção 3.5.2. O nome "nó folha virtual ativo" se dá pelo fato de que esse nó ativo, mesmo não sendo um nó folha de fato na *Trie*, não gera nenhuma atualização do conjunto $\Psi_{c_{x+1}}$ e também não vai gerar atualizações nos próximos conjuntos (quando houver).

Para manter os nós folha virtuais ativos encontrados durante a computação dos conjuntos de nós ativos, nós projetamos uma modificação no algoritmo de manutenção dos nós ativos do ICPAN para manter também um conjunto ψ de nós folha virtuais ativos, representada no Algoritmo 4.

O algoritmo tem como entrada os parâmetros Ψ_{p_x}, ψ e τ, c_{x+1}, os quais representam, respectivamente, o conjunto de nós ativos já computado para o prefixo digitado anteriormente, o conjunto de nós folha virtuais ativos, o limite de tolerância de erros de digitação, e o novo caractere do prefixo de consulta a ser processado. Na linha 2 o conjunto de nós ativos para o prefixo atualmente processado é criado, inicializado como um conjunto vazio. Na linha 3, a variável $ativaFolhasVirtuais$ recebe o resultado de uma expressão booleana que verifica se o tamanho do prefixo atual é maior que a diferença entre o limite λ de altura do primeiro nível e o limite de tolerância de erros τ.

O resultado dessa expressão controlará se o conjunto ψ será atualizado ou não. Sem esse controle, o conjunto ψ acabará armazenando uma quantidade muito grande de nós, o que prejudicará o desempenho de processamento do segundo nível. Se $\lambda = 5$ e $\tau = 3$ por exemplo, ao considerar qualquer prefixo da *Trie* com tamanho igual a 5, ao aplicar a operação de $\tau = 3$ deleções, a subcadeia resultante terá obrigatoriamente tamanho $\lambda - \tau = 2$. Então, a heurística de ativação do conjunto ψ se baseia nesse limite de τ deleções.

Na linha 4 define-se o laço de repetição de iteração sobre todas as tuplas de Ψ_{p_x}. Na linha 5 é chamada a função $processarNoComICPAN$, que analisa o nó ativo presente na *tupla* e alguns de seus descendentes de acordo com o algoritmo ICPAN apresentadas na seção 3.5.2, além de atualizar (ou não) o conjunto de nós ativos atual $\Psi_{p_{x+1}}$. Para possibilitar a atualização do conjunto ψ, foi necessário modificar essa função originalmente utilizada no método ICPAN, para que ela passasse a retornar um valor booleano indicando se houve alguma alteração no conjunto $\Psi_{p_{x+1}}$. Esse valor é atribuído à variável *gerouAtualizacao*. Na linha 6 verificamos se o nó da *tupla* não gerou atualização no con-

Algoritmo 4 Computação incremental de nós ativos e nós folha virtuais

```
1: function computarNosAtivosIncrementalmente(Ψ_{p_x}, ψ, τ, c_{x+1})
2:     Ψ_{p_{x+1}} ← ∅
3:     ativaFolhasVirtuais ← |p_{x+1}| > (λ − τ)
4:     for tupla ∈ Ψ_{p_x} do
5:         gerouAtualizacao ← processarNoComICPAN(tupla, c_{x+1}, Ψ_{p_{x+1}})
6:         if ¬gerouAtualizacao ∧ ativaFolhasVirtuais then
7:             ψ ← ψ ∪ {tupla}
8:     return Ψ_{p_{x+1}}, ψ
```

junto $\Psi_{p_{x+1}}$ e se o conjunto ψ deve ser atualizado. Se a expressão for verdadeira, então o conjunto ψ é atualizado na linha 7. Por fim,. na linha 8 retorna-se uma tupla contendo os conjuntos $\Psi_{p_{x+1}}, \psi$.

4.3.2 Computação de nós ativos no primeiro nível

Com a implementação do conjunto de nós folha virtuais ativos estabelecida é possível seguir para a computação do conjunto de nós ativos do primeiro nível, os quais serão utilizados como filtro para o segundo nível, como descrito na seção 4.2. Ao fim desse processo os conjuntos $\Psi_{p[1..\lambda]}$ e ψ terão sido computados (para os λ primeiros caracteres de um prefixo de consulta p), e a união $\Psi_{p_[1..\lambda]} \cup \psi$ determina o "conjunto final" $\mathcal{F}$ que será de fato utilizado no segundo nível.

No entanto, uma consequência dessa união é que o conjunto $\mathcal{F}$ possuirá nós folha virtuais ativos que foram ativados com diferentes subcadeias de $p[1..\lambda]$. Isto é, sendo $\lambda = 6$, $\mathcal{F}$ conterá alguns nós folha virtuais que foram ativados quando $p[1..4]$ foi processado e outros que foram ativados ao processar $p[1..5]$, por exemplo. Diante dessa situação, torna-se necessário armazenar mais uma informação nas tuplas que compõem os conjuntos de nós ativos para realizar a busca binária do método IP2LB proposto neste trabalho (a seção 4.6 possui mais detalhes).

As tuplas passarão a conter um elemento a mais, denominado δ_n, cujo valor representa o índice do caractere de $p[1..\lambda]$ que estava sendo processado no momento em que o nó n foi adicionado/atualizado nos conjuntos Ψ ou ψ, ou seja, tanto os nós ativos comuns quanto os nós folha virtuais ativos passarão a possuir o δ em suas respectivas tuplas. Se por exemplo $\lambda = 5$ e $p[1..\lambda] =$ "*sapat*", e um nó ativo foi adicionado ao conjunto Ψ quando a subcadeia $p[1..3] =$ "*sap*" foi processada, então $\delta_n = 3$ é inserido na tupla referente ao nó n.

Algoritmo 5 Computação de nós ativos para o primeiro nível

```
1: function computarNosAtivos(𝒯, p, τ)
2:     raiz ← obterRaiz(𝒯)
3:     Ψ = {⟨raiz, 0, ϵ, 0⟩}
4:     ψ = ∅
5:     for caractere ∈ Prefixo_λ(p) do
6:         Ψ, ψ ← computarNosAtivosIncrementalmente(Ψ, ψ, τ, caractere)
7:     ℱ ← Ψ ∪ ψ
8:     return ℱ
```

O Algoritmo 5 descreve esse procedimento, e tem como parâmetros $\mathcal{T}, p$ e τ que representam, respectivamente, o índice *Trie*, o prefixo de consulta, e o limiar de erros de edição tolerado. Na linha 2, o nó raiz da *Trie* é obtido. Na linha 3, o conjunto de nós ativos é inicializado com o nó raiz. Nesse momento, esse conjunto representa o conjunto Ψ_λ de nós ativos para a cadeia vazia de caracteres. Esse conjunto será sobrescrito algumas vezes no decorrer do algoritmo, e no fim será representará o conjunto $\Psi_{p[1..\lambda]}$ de nós ativos para $p[1..\lambda]$. Na linha 4 o conjunto ψ de nós folha virtuais ativos é inicializado como vazio. Então, na linha 5 define-se um laço de repetição para iterar sobre os λ primeiros caracteres de p. Na linha 6 é chamada a função $computarNosAtivosIncrementalmente$, a qual foi descrita anteriormente na seção 4. Os conjuntos retornados pela função sobrescrevem as variáveis Ψ e ψ. Já na linha 7 atribui-se a união entre o conjunto Ψ de nós ativos (que nesse momento representa o conjunto $\Psi_{p[1..\lambda]}$) e o conjunto ψ à $\mathcal{F}$, que é retornada pela função na linha 8 a seguir.

4.4 Segundo nível

Quando o conjunto final de nós ativos $\mathcal{F} = \Psi_{p[1..\lambda]} \cup \psi$ é computado para $p[1..\lambda]$ no primeiro nível, é preciso iniciar a etapa do segundo nível. Como dito anteriormente, cada nó n pertencente a esse conjunto possui uma lista L de nós folha, da qual cada nó n_{folha} tem um intervalo R que contém o menor e maior id das sugestões de consulta que começam com o prefixo formado ao percorrer o caminho da raiz até n_{folha}. A etapa do segundo nível consiste em, para todas as sugestões de consulta referenciadas por todos esses nós folha, determinar quais delas possuem um prefixo similar à p com no máximo τ erros de edição.

4.5 Utilizando somente busca sequencial

Para atingir esse objetivo, uma forma é realizar uma busca sequencial dentre essas sugestões de acordo com o algoritmo descrito na seção 3.2.1. Seja n um dos nós presentes no conjunto $\mathcal{F}$, n_{folha} um dos nós folha presentes em $n.L$. O intervalo $n_{folha}.R =$

$[minimo, maximo]$ representa uma lista de *ids* do dicionário $\mathcal{H}$ de sugestões indexadas. Ao aplicar nessa lista o algoritmo de busca sequencial descrito na seção 3.2.1, determinaremos quais das sugestões que começam com o prefixo n_{folha} são similares ao prefixo de consulta p considerando τ erros de digitação. Essa busca sequencial é realizada em todos os outros nós folhas de n. Além disso, esse procedimento que é realizado para n também é executado para todos os outros nós em $\mathcal{F}$. Desse modo, obtém-se todos as sugestões de consulta similares à p, a resposta do problema de CATE.

Na configuração de dois níveis proposta neste trabalho é possível realizar uma otimização nos cálculos de distância de edição com matrizes de *Levenshtein* para a acelerar a execução da busca sequencial. Sendo n um nó ativo qualquer de $\mathcal{F}$, obviamente todas as sugestões que podem ser obtidas a partir desse nó começarão com o prefixo n. A consequência disso é que todas as matrizes de *Levenhstein* calculada para determinar a similaridade entre p e uma sugestão qualquer com o prefixo n conterão um conjunto de linhas em comum (especificamente, as $|n| + 1$ primeiras linhas). É redundante computar essas $|n| + 1$ linhas em cada verificação entre p e uma sugestão com prefixo n realizada na busca sequencial para esse nó.

Dessa forma, é possível eliminar essa redundância ao computar a linha de número $|n| + 1$ da matriz gerada no cálculo de $ed(n, p)$ (os caracteres de p ficam nas colunas da matriz) antes de iniciar a busca sequencial. Se algum dos valores da última coluna da matriz de $ed(n, p)$ for menor ou igual a τ, todas as sugestões com prefixo n podem automaticamente fazer parte do conjunto de resposta para o problema. Por fim, todos os cálculos de matrizes de *Levenhstein* para as sugestões com o prefixo n podem iniciar a partir dessa linha da matriz previamente computada. Denominamos esse método em dois níveis como "IP2L" (**ICPAN 2-*Level***), que está descrito no Algoritmo 6.

Algoritmo 6 Complementação automática de consultas tolerante a erros com o IP2L

```
1:  function processarConsultaIP2L(T, H, p, τ)
2:      if |p| + τ ≤ λ then return ICPAN(T, p, τ)
3:      sugestoes ← ∅
4:      F ← computarNosAtivos(T, Prefixo_λ(p), τ)              ▷ Primeiro Nível
5:      for ⟨n, ξ_n^{p[1..λ]}, p_i, ξ_n^{p_i}⟩ ∈ F do          ▷ Segundo Nível (até a linha 13)
6:          ultimaLinhaComum ← ultimaLinhaLevenhstein(n, p)
7:          for n_folha ∈ n.L do
8:              for id ∈ n_folha.R do
9:                  sugestao ← recuperarTexto(H[id])
10:                 if ultimaLinhaComum[|p|] ≤ τ then
11:                     sugestoes ← sugestoes ∪ {sugestao}
12:                     continue
13:                 if similar(sugestao, p, ultimaLinhaComum, τ) then
14:                     sugestoes ← sugestoes ∪ {sugestao}
15:     return sugestoes
```

Esse algoritmo define a função $processarConsultaIP2L$, que tem como parâmetros $\mathcal{T}, \mathcal{H}, p$ e τ, os quais representam, respectivamente, o índice *Trie*, o dicionário "Tabela de Itens" mencionado na seção 4.1, o prefixo de consulta, e o limiar de erros de digitação. Na linha 2 há a verificação da condição para executar apenas o método ICPAN de acordo com o que foi mencionado na seção 4.2. Na linha 3 inicializa-se o conjunto *sugestoes* como vazio. Ao final da execução da função esse conjunto conterá todas as sugestões com prefixo similares p considerando no máximo τ erros. Na linha 4, o conjunto $\mathcal{F}$ recebe o conjunto retornado pela execução da função $computarNosAtivos$ descrita no Algoritmo 5.

Então, na linha 5 define-se um laço de repetição que itera sobre cada 4-upla (nó pivô ativo) do conjunto $\mathcal{F}$. O primeiro passo do laço é calcular a última linha da matriz de *Levenhstein* para $ed(n, p)$, com a função $ultimaLinhaLevenhstein$ na linha 6. O resultado será atribuído à variável $ultimaLinhaComum$, que será utilizada mais à frente para otimizar o processamento no segundo nível.

Nesse contexto é importante ressaltar que n é um nó da *Trie* e não uma cadeia de caracteres. A função $ultimaLinhaLevenhstein$ internamente recupera o prefixo formado pelo caminho da raiz até o nó n para utilizá-lo no cálculo da matriz de *Levenhstein*.

A linha 7 define um laço de repetição que itera sobre cada nó folha n_{folha} de $n.L$, e a linha 8 a seguir também define outro laço no qual itera-se sobre cada valor de id representado pelo intervalo $n_{folha}.R = [minimo, maximo]$. Na linha 9 recupera-se o texto completo da sugestão com a execução da função $recuperarTexto$, passando como parâmetro o item referenciado pela chave id no dicionário $\mathcal{H}$. Então, a linha 10 define a condição para verificar se o valor da última coluna da $ultimaLinhaComum$ é menor ou igual a τ. Se for, então a sugestão de consulta é adicionada ao conjunto *sugestoes* na linha 11 e o laço de repetição segue para a próxima iteração. Essa função concatena o prefixo formado pelo caminho da raiz até o nó $\mathcal{H}[id].noPrefixo$ e o concatena com a cadeia $\mathcal{H}[id].restante$. A linha 13 define a condição que verifica se a *sugestao* recuperada é similar à p considerando o limiar τ através da execução da função *similar*, que inicia o cálculo da matriz de *Levenhstein* de $ed(sugestao, p)$ a partir da $ultimaLinhaComum$.

A função *similar* retorna $Verdadeiro$ caso $ed(sugestao, p) \leq \tau$, e $Falso$ caso contrário. Além disso, se a qualquer momento do cálculo da matriz houver um valor na última coluna que seja menor ou igual a τ, a função *similar* interrompe sua execução e retorna $Verdadeiro$. A explicação para essa interrupção encontra-se na seção 3.2.1. Se a sugestão for similar, então é adicionada ao conjunto *sugestoes* na linha 14. Por fim, a função retorna o conjunto *sugestoes* computado na linha 15.

4.6 Combinando busca sequencial com binária

Uma desvantagem do método IP2L supramencionado é que o segundo nível pode acabar tendo uma execução lenta devido à grande quantidade de buscas sequenciais que precisam ser realizadas. Na tentativa de efetuar uma melhoria do tempo de execução, formulamos a hipótese de que é possível realizar buscas binárias em determinados nós ativos em vez da busca sequencial, combinando os dois tipos de busca no segundo nível. O pressuposto para realizar a busca binária é o de que ela pode ser aplicada em casos onde todos os erros de digitação possíveis já foram "esgotados" na computação do primeiro nível para a subcadeia $p[1..\lambda]$. Quando isso acontece, espera-se que seja possível realizar comparações simples entre cadeias de caracteres no segundo nível sem a necessidade de calcular matrizes de *Levenhstein*.

Considerando o método de dois níveis proposto neste trabalho, é possível categorizar dois tipos de nós ativos que se encontram no conjunto $\mathcal{F}$:

- *Nó ativo comum* – são os nós do subconjunto $\{n_c \in \mathcal{F} \mid \xi_{n_c}^{p[1..\lambda]} < \tau\}$. Quando a distância de edição entre o nó ativo e $p[1..\lambda]$ é menor do que τ, ainda há erros de digitação restantes para serem verificados no segundo nível. Portanto, para as folhas com raiz nesse nó é necessário realizar a busca sequencial tolerante a erros.
- *Nó ativo de borda* – são os nós do subconjunto $\{n_b \in \mathcal{F} \mid \xi_{n_b}^{p[1..\lambda]} = \tau\}$. Quando a distância de edição entre o nó ativo e $p[1..\lambda]$ é igual a τ, consideramos que a quantidade máxima de erros de digitação tolerada foi atingida. Portanto, seria possível realizar comparações exatas no segundo nível para as folhas com raiz nesse nó.

Essa categorização dos nós ativos permite diminuir a complexidade da busca realizada no segundo nível nos casos de nós ativos de borda, pois a comparação simples de caracteres é bem mais rápida de se realizar do que a computação de distância de edição, e desse modo a busca sequencial nesses nós seria mais eficiente. No entanto, uma vez que é possível realizar buscas exatas em nós ativos de borda, é muito mais eficiente realizar a busca binária em vez de busca sequencial. Além disso, em nossos experimentos constatamos que a quantidade de nós ativos de borda representam cerca de 80% a 90% do total de nós ativos do conjunto $\mathcal{F}$, para $\tau \geq 2$. Portanto, a modificação do método IP2L para combinar a busca binária com sequencial no segundo nível representa um potencial de tornar o método mais eficiente.

Diferentemente das buscas sequenciais realizadas nos nós ativos comuns, nas quais o valor procurado no segundo nível é p por completo, nossa hipótese é que nas buscas binárias o valor procurado deverá ser a subcadeia $p[(\delta_n + 1)..]$ (isto é, o restante dos caracteres de p a partir do δ_n-ésimo caractere). De acordo com o que foi apresentado na seção 4.3.2, alguns nós ativos de $\mathcal{F}$ foram ativados para uma subcadeia de p menor

que $p[1..\lambda]$. Levando em conta um nó ativo desses, alguns caracteres de $p[1..\lambda]$ não foram considerados em sua ativação, especificamente os caracteres da subcadeia $p[(\delta_n + 1)..\lambda]$. Portanto, deve-se pesquisar por $p[(\delta_n + 1)..]$ nas buscas binárias do segundo nível para que esses caracteres também sejam considerados no processamento da consulta.

Seja s uma sugestão de consulta qualquer referenciada por um nó ativo de borda n_b, e $s' = s[(n_b.altura + 1)..]$ uma subcadeia de s cujos primeiros $n_b.altura$ (tamanho do prefixo n_b) caracteres foram removidos. Para que s seja inclusa no conjunto de respostas para o prefixo de consulta p, basta que os primeiros caracteres de s' sejam exatamente iguais a $p[(\delta_n+1)..]$. Essa é a condição de igualdade considerada na busca binária realizada nos nós ativos de borda.

Ademais, uma condição necessária para realizar a busca binária é que os dados estejam ordenados. Por essa razão a coleção $\mathcal{D}$ de sugestões de consulta é ordenada em ordem lexicográfica crescente antes de ser indexada na árvore *Trie*, como explicado na seção 4.1. A consequência decorrente desse procedimento é que os itens referenciados pelos intervalos R dos nós estão também em ordem lexicográfica crescente, pois eles referenciam uma porção da da base D ordenada. *Assim, realizamos buscas binárias em todos os nós folha que possam ser alcançados a partir de um nó ativo de borda qualquer.*

No entanto, a subcadeia pesquisada $p[(\delta_n + 1)..]$ pode ocorrer em várias sugestões referenciadas em um intervalo e devido à ordenação, essas ocorrências sempre estarão próximas umas das outras no intervalo. Consequentemente, é necessário realizar duas buscas binárias, sendo uma pelo limite inferior, e outra pelo limite superior. Caso o valor pesquisado não for encontrado, então atribui-se o valor -1 a ambos limites. Essa busca binária com elementos repetidos na coleção é a mesma detalhada na seção 3.7. Os limites obtidos representam um subintervalo de R (que pode ser igual a R) cujas sugestões de consulta referenciadas podem ser inclusas na resposta para o prefixo de consulta p. Denominamos esse método como "IP2LB" (**I**C**P**AN **2**-***L****evel with*** ***B****inary Search*), que está descrito no Algoritmo 7.

Algoritmo 7 Complementação automática de consultas tolerante a erros com o IP2LB

```
1:  function processarConsultaIP2LB(𝒯, ℋ, p, τ)
2:      if |p| + τ ≤ λ then return ICPAN(𝒯, p, τ)
3:      sugestoes ← ∅
4:      ℱ ← computarNosAtivos(𝒯, Prefixo_λ(p), τ)                ▷ Primeiro Nível
5:      for ⟨n, ξ_n^{p[1..λ]}, p_i, ξ_n^{p_i}, δ_n⟩ ∈ ℱ do      ▷ Segundo Nível (até a linha 24)
6:          if ξ_n^{p[1..λ]} = τ then                            ▷ Caso em que n é um nó ativo de borda
7:              for n_folha ∈ n.L do
8:                  p' ← p[(δ_n + 1)..]
9:                  limInferior ← buscaBinariaLimInf(p', n_folha.R, n.altura)
10:                 limSuperior ← buscaBinariaLimSup(p', n_folha.R, n.altura)
11:                 if limInferior ≥ 0 and limSuperior ≥ 0 then
12:                     i ← limInferior
13:                     while i ≤ limSuperior do
14:                         sugestao ← recuperarTexto(ℋ[i])
15:                         sugestoes ← sugestoes ∪ {sugestao}
16:                         i ← i + 1
17:         else                                                 ▷ Caso em que n é um nó ativo comum
18:             ultimaLinhaComum ← ultimaLinhaLevenhstein(n, p)
19:             for n_folha ∈ n.L do
20:                 for id ∈ n_folha.R do
21:                     sugestao ← recuperarTexto(ℋ[id])
22:                     if ultimaLinhaComum[|p|] ≤ τ then
23:                         sugestoes ← sugestoes ∪ {sugestao}
24:                         continue
25:                     if similar(sugestao, p, ultimaLinhaComum, τ) then
26:                         sugestoes ← sugestoes ∪ {sugestao}
27:     return sugestoes
```

Esse algoritmo define a função $processarConsultaIP2LB$, que tem como parâmetros $\mathcal{T}, \mathcal{H}, p$ e τ, os quais representam, respectivamente, o índice *Trie*, o dicionário "Tabela de Itens" mencionado na seção 4.1, o prefixo de consulta, e o limiar de erros de digitação. Assim como os parâmetros as próximas linhas seguem iguais às do Algoritmo 6 até a linha 6, onde o tratamento do nó ativo de borda é introduzido. Essa linha define a condição que verifica se o nó ativo n iterado é um nó ativo de borda.

Caso seja, todos os nós folha com raiz em n são iterados no laço de repetição definidos na linha 7. O laço inicia com o cálculo do valor que será pesquisado nas buscas binárias na linha 8. Então, nas linhas 9 e 10 são executadas as buscas binárias por p' no intervalo $n_{folha}.R$ para obter os limites inferior e superior, respectivamente. As funções de busca binária precisam também do tamanho do prefixo n (representado por $n.altura$) para calcular a subcadeia s' de cada sugestão de consulta, como descrito anteriormente. Se os limites obtidos tiverem valor positivo, como verifica a linha 11, então inicia-se o processo de adicionar os itens referenciados pelos limites. A linha 12 inicializa a variável

i com o valor do limite inferior $limInferior$. A linha 13 estabelece um laço de repetição que irá iterar sobre todos os itens referenciados. Na linha 14, uma sugestão de consulta é recuperada pela função $recuperarTexto$, e então é adicionada ao conjunto de $sugestoes$. Por fim, o bloco formado pelas linhas 17 a 27 definem o procedimento realizado para os nós ativos comuns, que é o mesmo já apresentado no Algoritmo 6.

4.7 Aprimorando a acurácia do método IP2LB

Descobrimos em nossos experimentos que o tratamento de nós de borda do método IP2LB suprarreferido deu origem ao problema de não obter todas as sugestões que deveria para um prefixo de consulta p, além de também trazer algumas sugestões que possuem um pouco mais do que τ erros de digitação. Essa situação caracteriza o método IP2LB como impreciso já que ele não consegue responder o problema de CATE integralmente.

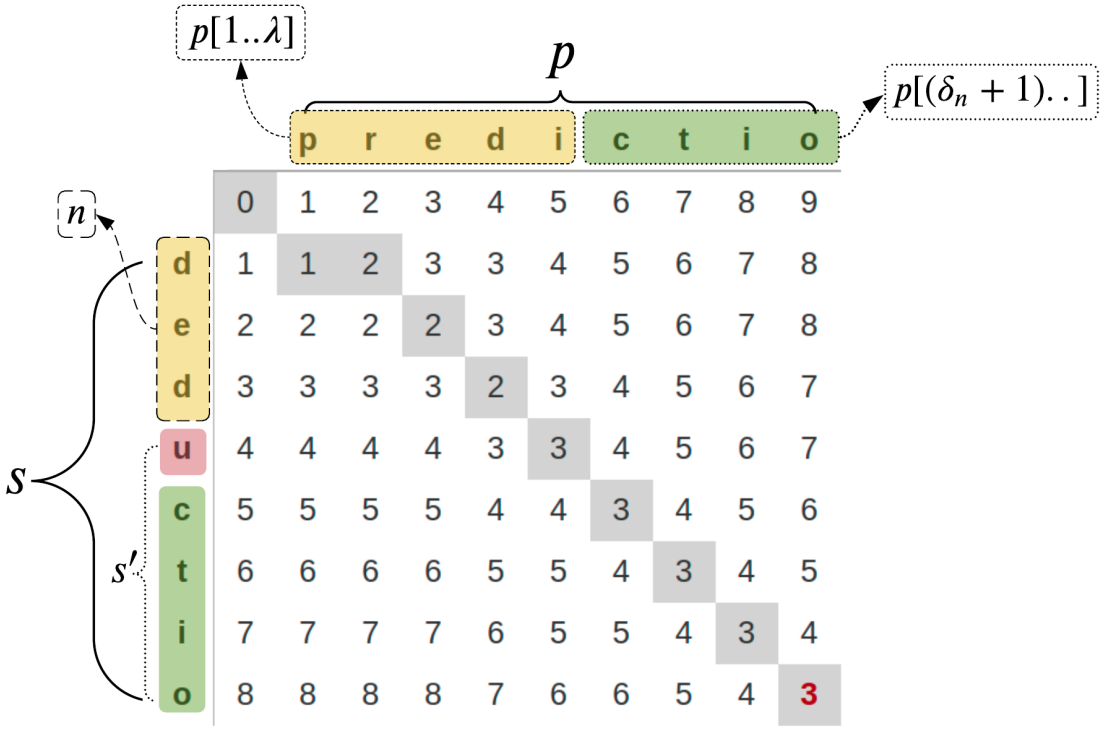


Figura 8: Um contra-exemplo possível que prova a imprecisão do método IP2LB, composto de uma Matriz de *Levenhstein* para o cálculo de distância de edição entre $p =$ "*predictio*" (colunas da matriz) e a sugestão de consulta $s =$ "*deductio*" (linhas da matriz)

.

A Figura 8 demonstra um contra-exemplo que prova a incapacidade que o IP2LB possui de recuperar todas as sugestões de consultas similares à p com até τ erros. Na figura, $\lambda = 5$, $\delta_n = 5$, $\tau = 3$. Os textos destacados em verde são, respectivamente, $p[(\delta_n + 1)..] =$ "*ctio*" (o texto que é procurado pelas buscas binárias no segundo nível) e uma subcadeia $s'[2..]$ de $s' = s[(\delta+1)..] =$ "*uctio*", sendo $s'[2..] = p[(\delta_n+1)..]$ nesse exemplo

em específico. Há também os textos "*ded*", que é o prefixo obtido a partir do caminho entre raiz da *Trie* e o nó ativo de borda n, e $p[1..\lambda]$ = "*pred*", que é o prefixo utilizado na computação de nós ativos no primeiro nível. Ambos estão coloridos em amarelo para destacar a relação entre n e $p[1..\lambda]$, sendo $ed(n, p[1..\lambda] = \tau$. Além disso, há também a Matriz de *Levenhstein* do cálculo de distância de edição entre p = "*predictio*" e a sugestão de consulta s = "*deductio*", que aliás possui o valor 3 como podemos observar no valor da célula do canto inferior direito. Uma vez que $ed(s, p) \leq \tau$, então s deve ser uma das consultas sugeridas como resposta da CATE para o prefixo de consulta p.

No entanto, quando o algoritmo IP2LB realiza as buscas binárias em n, o nó ativo de borda desse exemplo, s não será incluída no conjunto de respostas para a consulta p pois $p[(\delta_n + 1)..]$, que mesmo sendo uma subcadeia de s', não é encontrado nos primeiros caracteres de s' como estabelecido na seção 4.6. Se não houvesse o caractere "u" (destacado em vermelho) em s, ou seja, se s fosse "*dedctio*", o algoritmo IP2LB consideraria s como uma das sugestões válidas como resposta para a consulta. Essa falha também ocorre com outras sugestões no segundo nível do IP2LB, que por consequência não retorna alguns resultados que deveria.

Sendo assim, percebemos que não é possível aplicar a busca binária em todos os nós ativos cuja distância de edição é igual ao limiar τ sem que haja perda de resultados. Diante disso, na tentativa de resolver esse problema formulamos a hipótese de que é possível melhorar a acurácia do IP2LB restringindo a realização da busca binária para apenas alguns nós ativos de borda. Sendo $\langle n_b, \xi_{n_b}^{p[1..\lambda]}, p_i, \xi_{n_b}^{p_i}, \delta_{n_b} \rangle$ a 5-upla de um nó ativo de borda n_b, realizamos a busca binária nesse nó somente se $\delta_{n_b} = \lambda$, ou em outras palavras, se o caractere de $p[1..\lambda]$ que estava sendo processado quando n_b foi ativado foi o λ-ésimo caractere. Essa é uma proposta de adaptação necessária para a utilização da ideia de combinar busca binária e sequencial no segundo nível, sendo uma possível resposta para a questão de pesquisa ii), apresentada na seção 1.1. Denominamos esse método como "IP2LRB" (**ICP**AN **2**-***L**evel with **R**estricted **B**inary Search*). O algoritmo para o IP2LRB é idêntico ao Algoritmo 7, com apenas uma modificação da condição verificada na linha 6 para esta: "$\xi_n^{p[1..\lambda]} = \tau \wedge \delta_n = \lambda$".

Essa restrição faz com que a busca binária passe a ser realizada em somente 15% $\sim$ 40% do total de nós ativos de $\Psi_{p[1..\lambda]}$, em contraste aos 75% $\sim$ 95% do método IP2LB. O efeito disso é que o tempo de processamento de consultas do IP2LRB aproxima-se do tempo do método IP2L, porém trazendo um pouco mais de resultados que antes estavam faltando com o IP2LB. No entanto, mesmo com essa restrição o IP2LRB não consegue ter acurácia absoluta, pois ainda não traz 100% dos resultados que deveria. Com isso, pensando na questão de pesquisa i) concluímos que não é possível adaptar sistemas de busca em dois níveis para tirarem proveito da busca binária no segundo nível sem que haja algum impacto na acurácia do método.

5 Resultados

Nesse capítulo nós apresentamos os resultados do estudo de parâmetros dos métodos propostos, suas vantagens e desvantagens e também os cenários onde podem ser melhor aplicados. Além disso, também comparamos os métodos propostos com alguns *baselines* (métodos usados para comparação de resultados experimentais). A partir dos experimentos espera-se responder as seguintes perguntas: i) É possível adaptar sistemas de busca em dois níveis para tirarem proveito da busca binária no segundo nível? ii) Quais são as adaptações necessárias e limitações de uso da ideia? iii) Há vantagens práticas na combinação de busca binária e busca sequencial no segundo nível?

5.1 Configuração dos experimentos

5.1.1 Métodos experimentados

Nos experimentos são analisados os seguintes algoritmos:

- **ICAN** – Um método baseado na computação incremental de nós ativos. (Ji et al., 2009)
- **ICPAN** – Uma otimização do método **ICAN** que se baseia na computação incremental de nós pivô ativos. (Li et al., 2011)
- **META** – Um método que utiliza indexação em árvores compactas e que consegue reduzir computações redundantes no processamento de consultas. (Deng et al., 2016)
- **BEVA** – Um método baseado em manter um conjunto de "nós ativos de fronteira" e o "autômato de vetores de edição" para processar as consultas. (Zhou et al., 2016)
- **IP2L** – Método que segue a abordagem de processamento em dois níveis, utilizando o ICPAN no primeiro nível, e busca sequencial simples no segundo nível. (Seção 4.5)
- **IP2LB** – Variação do método IP2L, com utilização de busca binária e sequencial no segundo nível. (Seção 4.6)
- **IP2LRB** – Variação do método IP2LB, com uma restrição na condição para ativar a busca binária no segundo nível. (Seção 4.7)

As implementações dos métodos ICAN, ICPAN e META foram retiradas de um repositório[1] do *GitHub*. Esta pesquisa produziu contribuições quanto a esse repositório: (1)

[1] https://github.com/TsinghuaDatabaseGroup/Autocompletion/tree/master/threshold

correção de um *memory leak* no código dos arquivos "*main*" dos métodos ICAN e ICPAN; (2) A inicialização do conjunto de nós ativos do método ICAN estava incompleta, e foi corrigida levando em consideração a descrição de inicialização presente no artigo original; (3) Correção de funcionamento do método ICPAN. Uma parte do fluxo do código de expansão do conjunto de nós pivô ativos estava com um cálculo errôneo para o limite de profundidade que seria analisada a partir de um nó pivô ativo. No código do repositório a expressão para definir esse limite é $\tau - \xi_n^{p_{x+1}} + 1$, quando na verdade deveria ser $\tau - \xi_n^{p_i} + 1$ como explicado anteriormente na seção 3.5.2.

5.1.2 Ambiente de experimentação e bases de dados

Os métodos propostos no capítulo 4 foram implementados na linguagem *C++* visando obter um melhor desempenho, e também possibilitar uma comparação mais fidedigna com os *baselines*, os quais também foram todos implementados em *C++*. Todos os códigos-fonte dos *baselines* são os originais utilizados pelos autores de seus respectivos artigos, com exceção do método BEVA, cuja implementação é própria e foi realizada no trabalho de da Gama Ferreira (2020).

Todos os experimentos foram realizados em uma máquina de servidor com processador *Intel ® Xeon E5 4617 2.90 GHz*, com *64GB* de memória *RAM*, tendo o *Ubuntu 18.04.1 LTS* como sistema operacional. Os algoritmos foram compilados com o *gcc 7.4.0*.

Utilizamos as seguintes bases de dados:

- **AOL**[2] – Um histórico de *logs* de consultas da plataforma *AOL*, que possui cerca de 140 mil consultas únicas e datadas entre março e maio de 2006. Para os experimentos foi gerado um arquivo de texto no qual cada linha representa um item.
- **USADDR**[3] – Um conjunto com mais de 7 milhões de endereços e localizações dos Estados Unidos da América, extraídos da coleção *SimpleGEO CC0*. Dessa base, os itens foram extraídos e inseridos em um único arquivo de texto no qual cada linha representa um item.

Base de Dados	Itens	Palavras	Tamanho médio das palavras	Tamanho médio dos itens
AOL	143.591	690.685	6,2338	30,0728
USADDR	7.699.149	25.432.812	5,7738	20,5595

Tabela 2: Estatísticas das bases de dados

Todos os experimentos para medir o tempo de processamento nessas bases foram realizados com 500 itens extraídos aleatoriamente da base de dados para servirem como

[2] http://www.ccc.ipt.pt/~ricardo/experiments/AOL_DS.html
[3] http://archive.org/download/2011-08-SimpleGeo-CC0-Public-Spaces/

prefixos de consulta, com tamanhos variando de 3 a 13 (aumentando de 2 em 2), e τ variando entre 1, 2 e 3. Todos esses prefixos de consulta foram alterados com um número de erros de digitação que está dentro do intervalo $[0, \tau]$ em uma posição qualquer da cadeia de caracteres. O número de erros e a posição dos erros é escolhida aleatoriamente. Para cada execução de cada algoritmo, também foi medido seu pico de memória utilizado.

Ademais, também utilizamos uma base de dados extraída de um sistema real de complementação automática de consultas da empresa Jusbrasil[4], um empreendimento que une Direito e Tecnologia e fornece um serviço de busca vertical para seus usuários. Essa base de dados contém 23.375.740 itens. Além disso, há também um *log* com 648.264 prefixos de consulta submetidos para esse sistema, do qual extraímos 1000 prefixos. Ele contém os prefixos máximos digitados pelos usuários antes de clicarem no botão que executa a busca, ou antes de clicarem em uma opção sugerida pelo sistema. Essa base de dados será referida como JUSBRASIL, a qual está detalhada na Tabela 3. Nos experimentos realizados com essa base utilizamos os textos dos prefixos de consulta na íntegra, sem variar o tamanho, e os valores de τ experimentados foram variados entre 1, 2 e 3. Além disso, também não alteramos o texto de nenhum prefixo de consulta para inserir quaisquer erros de digitação, preservando o padrão de digitação orgânico dos usuários.

Arquivo	**Itens**	**Palavras**	**Tamanho médio das palavras**	**Tamanho médio dos itens**
sugestões de consulta	23.374.740	91.340.225	6,0582	26,0839
prefixos de consulta	1.000	2.845	6,1364	18,6230

Tabela 3: Estatísticas das sugestões de consulta e dos prefixos de consultas provindos da base de dados da JUSBRASIL.

5.2 Impactos da busca binária na acurácia dos métodos em dois níveis

Como demonstrado na seção 4.7, os métodos IP2LB e IP2LRB ora deixam de recuperar sugestões que deveriam fazer parte da resposta, e ora recuperam sugestões que não deveriam ser sugeridas. Portanto, faz-se necessário analisar o quão impactante é esse comportamento em comparação com um método que traz todas as respostas corretas, como o BEVA, por exemplo, pois seria irrelevante sacrificar a capacidade de obter um conjunto de sugestões minimamente aceitável em troca de uma menor velocidade de processamento (devido à busca binária).

Para realizar tal análise, fizemos uma adaptação das métricas *precisão* e *revocação*, que são métricas utilizadas no contexto da Recuperação de Informação. A abordagem

[4] http://www.jusbrasil.com.br

padrão gira em torno da noção de documentos *relevantes* e *irrelevantes* (Christopher D. Manning, Prabhakar Raghavan, 2008) em relação à informação que o usuário necessita. Considerando o problema de CATE, nessa adaptação as sugestões de consulta retornadas pelos métodos propostos no capítulo 4 para o prefixo de consulta p são consideradas "documentos". Estabelecemos que uma sugestão é relevante se e somente se ela pode ser encontrada no conjunto de respostas computado pelo método BEVA para o mesmo prefixo de consulta p. Ou seja, sendo S é o conjunto de sugestões computadas por um método considerando τ erros de digitação, e $s \in S$ é uma sugestão qualquer de S, s é relevante se e somente se $s \in Beva(p, \tau)$, e irrelevante caso contrário.

	Sugerida pelo BEVA (Relevante)	**Não sugerida pelo BEVA (Irrelevante)**
Sugerida pelo método	verdadeiro positivo (vp)	falso positivo (fp)
Não sugerida pelo método	falso negativo (fn)	verdadeiro negativo (vn)

Tabela 4: Tabela do teste de hipóteses da relevância de uma sugestão sugerida por um método de CATE em comparação com as sugestões sugeridas pelo método BEVA.

A Tabela 4 contém o teste de hipóteses considerando esse comparativo com o método BEVA. Considera-se "verdadeiro positivo (vp)" quando a sugestão foi sugerida tanto pelo método analisado quanto pelo BEVA. Se foi sugerida pelo método mas não pelo BEVA, então é um caso de "falso positivo (fp)". Uma sugestão que foi sugerida pelo BEVA mas não pelo método representa um "falso negativo (fn)". As sugestões não sugeridas nem pelo método e nem pelo BEVA representam o "verdadeiro negativo (vn)". Essas definições são utilizadas no cálculo da métrica *F1-Score*, descrita a seguir.

5.2.1 A métrica *F1-Score*

As duas métricas mais frequentes e básicas para a eficácia de um sistema de recuperação de informação são a *precisão* e a *revocação* (Christopher D. Manning, Prabhakar Raghavan, 2008). Seus valores variam entre 0 e 1, mas são comumente representadas em forma de porcentagem.

No contexto apresentado na seção 5.2, *Precisão (P)* é a fração de sugestões computadas (ou recuperadas) que são relevantes:

$$Precisao = \frac{\#(itens\ relevantes\ recuperados)}{\#(itens\ recuperados)}$$

Já a *Revocação (R)* é a fração de sugestões relevantes que foram recuperadas:

$$Revocacao = \frac{\#(itens\ relevantes\ recuperados)}{\#(itens\ relevantes)}$$

Uma outra forma é escrever essas equações é utilizando os termos da tabela de contingência apresentada anteriormente (Tabela 4. Nesse caso, a precisão e revocação seriam, respectivamente:

$$P = \frac{vp}{vp + fp}$$
$$R = \frac{vp}{vp + fn}$$

Há uma medida única que combina a precisão e a revocação, chamada *F1-Score*, a qual é definida pela média harmônica entre a precisão e revocação:

$$F_1 = \frac{2 \cdot P \cdot R}{P + R}$$

Há uma variação dessa métrica na qual a média harmônica é ponderada, sendo possível dar mais ênfase no resultado para a precisão ou para a revocação. Neste trabalho utilizamos pesos iguais na métrica para ter uma avaliação mais geral de como a busca binária afeta os métodos IP2LB e IP2LRB.

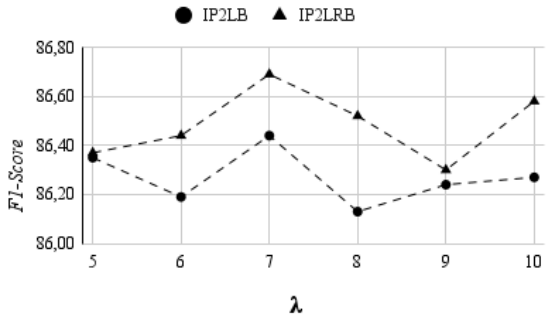
A média harmônica é utilizada no lugar da média aritmética pois sempre é possível obter 100% de revocação ao simplesmente fazer o método sugerir todas as sugestões da base, e portanto é possível obter uma média aritmética de 50% a partir desse mesmo processo, enquanto o mesmo não acontece com a média harmônica. Isso sugere que a média aritmética não é adequada para esse caso (Christopher D. Manning, Prabhakar Raghavan, 2008).

5.2.2 Experimentos com a base de dados da Jusbrasil

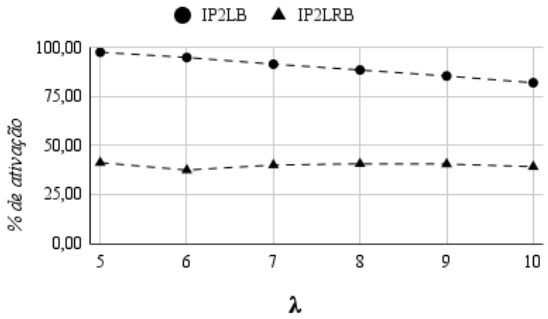
Para averiguar os impactos da utilização da busca binária na acurácia dos métodos propostos realizamos experimentos na base JUSBRASIL, a qual foi extraída de um sistema real de complementação automática de consultas. Esse experimento foi realizado visando responder as questões de pesquisa i) e iii) levantadas na seção 1.1. Em um sistema de dois níveis para CATE normalmente espera-se que todas as sugestões aproximadas dentro do limite τ sejam retornadas. No entanto, se o ao utilizar busca binária no segundo nível o modelo ainda trouxer grande maioria dos resultados corretos, então utilizar essa técnica pode ser vantajoso.

Tendo τ variando de 1 a 3, e λ de 5 a 10, medimos o tempo de processamento médio e o *F1-Score* médio para cada um dos três métodos propostos, e também para o método BEVA. A métrica *F1-Score* é um valor real que varia de 0 a 1, mas que será

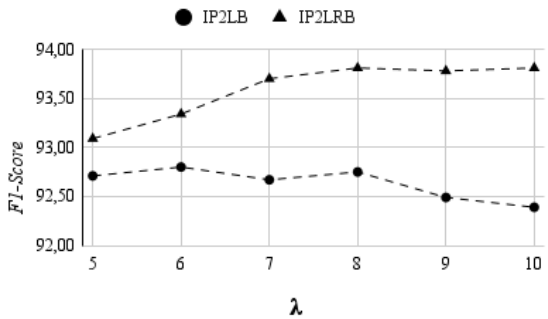
multiplicado por 100 em todas as tabelas para representação de forma percentual. De acordo com o que foi apresentado anteriormente, as respostas computadas pelo método BEVA são utilizadas como base para o cálculo de F1-Score dos outros métodos, portanto apresenta sempre o valor "100". Além disso, uma vez que o método IP2L computa o mesmo conjunto de respostas que o método BEVA, também possui sempre o valor "100" para o *F1-Score*. Diferentemente dos experimentos das próximas seções, não variamos o tamanho dos prefixos de consulta. Mantivemos exatamente o texto da base de consultas com seus tamanhos originais.



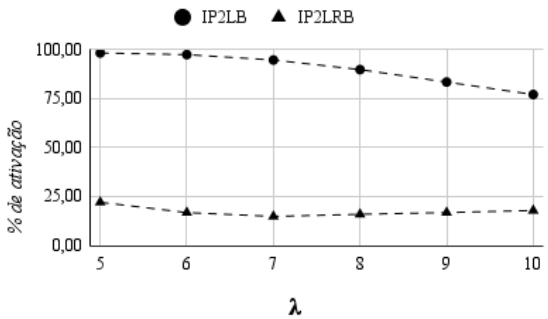
(a) Valores de *F1-Score* dos métodos IP2LB e IP2LRB para $\tau = 1$.



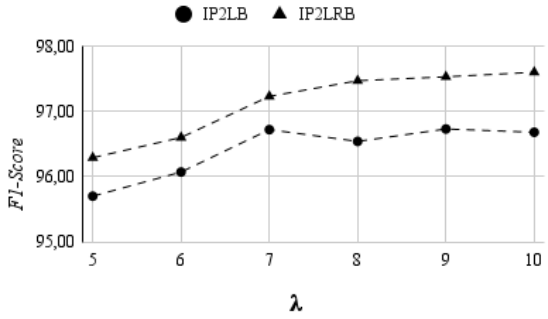
(b) Percentual de ativação de busca binária dos métodos IP2LB e IP2LRB para $\tau = 1$.



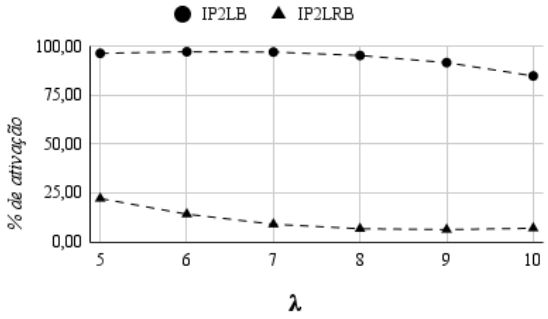
(c) Valores de *F1-Score* dos métodos IP2LB e IP2LRB para $\tau = 2$.



(d) Percentual de ativação de busca binária dos métodos IP2LB e IP2LRB para $\tau = 2$.



(e) Valores de *F1-Score* dos métodos IP2LB e IP2LRB para $\tau = 3$.



(f) Percentual de ativação de busca binária dos métodos IP2LB e IP2LRB para $\tau = 3$.

Figura 9: Valores de *F1-Score* e percentuais de ativação de busca binária para os métodos IP2LB e IP2LRB, com τ variando de 1 a 3 e λ de 5 a 10, para a base JUSBRASIL.

	$\tau = 1$		$\tau = 2$		$\tau = 3$	
Método	**Tempo de proc.**	**F1-Score**	**Tempo de proc.**	**F1-Score**	**Tempo de proc.**	**F1-Score**
IP2L-5	187,94	100,00	1906,07	100,00	15319,65	100,00
IP2LB-5	48,00	86,35	189,08	92,71	1642,66	95,70
IP2LRB-5	90,79	86,37	1170,75	93,09	8907,34	96,29
IP2L-6	97,93	100,00	792,27	100,00	8870,53	100,00
IP2LB-6	31,33	86,19	136,07	92,80	1145,92	96,07
IP2LRB-6	40,34	86,44	604,66	93,34	7339,66	96,60
IP2L-7	62,62	100,00	301,75	100,00	5607,94	100,00
IP2LB-7	17,86	86,44	76,48	92,67	720,42	96,72
IP2LRB-7	23,79	86,69	214,88	93,70	5112,13	97,23
IP2L-8	38,76	100,00	144,07	100,00	1905,70	100,00
IP2LB-8	12,78	86,13	49,68	92,75	328,13	96,54
IP2LRB-8	16,87	86,52	92,02	93,81	1778,79	97,47
IP2L-9	24,72	100,00	91,75	100,00	637,83	100,00
IP2LB-9	7,72	86,24	35,60	92,49	192,10	96,73
IP2LRB-9	10,75	86,30	64,30	93,78	567,71	97,53
IP2L-10	15,76	100,00	67,52	100,00	326,71	100,00
IP2LB-10	5,47	86,27	27,61	92,39	155,62	96,68
IP2LRB-10	7,70	86,58	49,50	93,81	279,60	97,60
BEVA	1,03	100,00	10,52	100,00	63,35	100,00

Tabela 5: Tempo de processamento (ms) e F1-Score para os métodos IP2LB e IP2LRB e o método BEVA, variando o parâmetro λ de 5 a 10 e τ de 1 a 3 na base de dados JUSBRASIL.

	$\tau = 1$			$\tau = 2$			$\tau = 3$		
Método	**Nós Ativos**	**Ativação busca bin.**	**F1-Score**	**Nós Ativos**	**Ativação busca bin.**	**F1-Score**	**Nós Ativos**	**Ativação busca bin.**	**F1-Score**
IP2LB-5	84,98	97,50%	86,35	2858,21	98,15%	92,71	26935,19	96,32%	95,70
IP2LRB-5	84,98	41,23%	86,37	2858,21	22,04%	93,09	26935,19	22,15%	96,29
IP2LB-6	59,18	94,82%	86,19	2064,35	97,30%	92,80	27976,40	97,14%	96,07
IP2LRB-6	59,18	37,42%	86,44	2064,35	16,79%	93,34	27976,40	14,14%	96,60
IP2LB-7	40,98	91,42%	86,44	1228,55	94,55%	92,67	23171,90	96,98%	96,72
IP2LRB-7	40,98	40,01%	86,69	1228,55	14,85%	93,70	23171,90	8,99%	97,23
IP2LB-8	32,78	88,44%	86,13	678,04	89,66%	92,75	15020,53	95,24%	96,54
IP2LRB-8	32,78	40,62%	86,52	678,04	15,92%	93,81	15020,53	6,69%	97,47
IP2LB-9	28,30	85,39%	86,24	428,66	83,32%	92,49	8289,18	91,59%	96,73
IP2LRB-9	28,30	40,51%	86,30	428,66	16,81%	93,78	8289,18	6,25%	97,53
IP2LB-10	24,84	81,94%	86,27	317,95	76,96%	92,39	4625,44	84,77%	96,68
IP2LRB-10	24,84	39,17%	86,58	317,95	17,86%	93,81	4625,44	6,95%	97,60

Tabela 6: Média de nós ativos, percentual médio que indica em quanto dos nós ativos foi realizada a busca binária e F1-Score para os métodos IP2LB e IP2LRB, variando λ de 5 a 10 e o τ de 1 a 3, na base JUSBRASIL.

Para $\tau = 1$ os modelos IP2LB e IP2LRB obtiveram valores de *F1-Score* muito próximos, apesar de baixos (menores do que 87%. O IP2LRB foi superficialmente mais preciso, como mostra a Figura 9a. O IP2LB seguiu com os menores valores de tempo de processamento. Essa diferença de tempo de processamento ocorreu pois a busca binária foi utilizada em um grande percentual dos nós ativos no modelo IP2LB. Com $\lambda = 5$, por exemplo, o IP2LB obteve $97,5\%$ de ativação da busca binária contra $41,23\%$ do IP2LRB, e

com $\lambda = 10$ o IP2LB obteve $81,94\%$ contra $39,17\%$ do IP2LRB, como mostra a Figura 9b. As linhas de tendência para *F1-Score* não são muito claras para os dois métodos.

Nas Figuras 9b, 9d e 9f (as quais representam os dados das Tabelas 5 e 6) nota-se que a curva de percentual de ativação segue um padrão decrescente para o modelo IP2LB, ou seja, à medida em que a quantidade de caracteres indexados na *Trie* (valor de λ) aumenta, a busca binária ocorre em menos nós. Esse comportamento é esperado pois quanto maior o valor de λ, maior é a possibilidade de o primeiro nível do método ser o suficiente para conseguir responder à consulta utilizando nós ativos comuns, e não os nós de borda.

Já o método IP2LRB apresenta um padrão de curva diferente. À medida em que λ aumenta, o percentual de ativação começa a diminuir um pouco mas em seguida quase não apresenta mais diminuição. Isso provavelmente acontece devido à condição de restrição para ativação de busca binária que há no método. Quanto maior o valor de λ, mais raro é que surjam nós ativos de borda que foram ativados quando o λ-ésimo caractere foi processado pelo método. É provável que a maioria dos erros de digitação em um sistema de busca real como o da Jusbrasil ocorra nos primeiros caracteres da consulta, fazendo com que as sugestões sejam ativadas na *Trie* antes de serem propagadas para o segundo nível.

Os dois modelos não se demonstraram muito vantajosos para $\tau = 1$. Apesar de terem o tempo de processamento 2 ou até 3 vezes menores do que o IP2L para alguns valores de λ, os valores de *F1-Score* parecem ser muito baixos quando se trata de um sistema real de CATE. Uma possível explicação para tais valores é que para $\tau = 1$, a quantidade de nós ativos de borda é bem maior do que a quantidade de nós ativos comuns pois nesse caso os nós ativos comuns representam apenas correspondência sem erros de digitação (exata), então a busca binária será majoritariamente utilizada no segundo nível. Uma vez que ela é suscetível a recuperar algumas sugestões a mais, e também a ignorar outras sugestões que deveriam ser recuperadas, ativá-la com mais frequência pode causar diminuição significativa da acurácia.

Algo interessante a se notar é que enquanto os valores de *F1-Score* foram próximos, há uma diferença considerável entre os percentuais de ativação de busca binária para os dois métodos. Um fato que pode ser extraído da definição dos dois métodos descritos no capítulo 4 é que para um mesmo prefixo de consulta p todo o conjunto de nós em que a busca binária foi ativada no método IP2LRB está completamente incluso no conjunto de nós de borda ativados no método IP2LB.

A partir disso, uma possível explicação que surge é a de que os valores de *F1-Score* do método IP2LRB definem um limite superior para os valores de *F1-Score* do método IP2LB. É improvável que tal método ultrapasse os valores do IP2LRB pois realiza busca binária em um percentual de nós bem maior. Ao que tudo indica, o número maior de

buscas binárias do IP2LB não impactou tanto a acurácia para $\tau = 1$, e ajudou a reduzir o tempo de processamento em relação ao IP2LRB.

Para $\tau = 2$, podemos ver com mais clareza o impacto do aumento da variável λ nos valores de *F1-Score*, além de os valores em si também estarem mais altos do que para $\tau = 1$. Isso quer dizer que os dois métodos se apresentaram mais precisos quando passaram a tolerar mais um erro de digitação. Por outro lado o método IP2LB apresenta um padrão de diminuição gradual da métrica, e o IP2LRB apresenta uma linha de tendência logarítmica, que começa a estabilizar a partir de $\lambda = 8$.

Há também uma maior disparidade entre os valores de *F1-Score para os dois métodos*, chegando por exemplo a uma diferença de 1,42 para $\lambda = 10$. Uma vez que a curva de ativação do método IP2LRB não apresenta muita variação à medida em que λ cresce, e o valor de *F1-Score* aumenta um pouco e depois estabiliza, é provável que haja um crescimento da quantidade de prefixos de consulta que são processados apenas no primeiro nível, o que explicaria o aumento da métrica. Como demonstrado na seção 4.7, a busca binária introduz erro na recuperação das sugestões. Portanto, quanto menos ela é ativada, maior a acurácia, porém maior também é o tempo de processamento.

Com $\tau = 3$, a linha de tendência para o método IP2LRB ainda permanece com características logarítmicas, mas agora o IP2LB também assume um formato próximo de linha tendência logarítmica (com exceção do ponto para $\lambda = 8$). Além disso, há o ápice de *F1-Score* para os dois métodos, chegando a $97,60\%$ para o IP2LRB com $\lambda = 10$ e $96,73\%$ para o IP2LB com $\lambda = 9$. Mais uma vez, verificamos que aumentar o valor de τ também aumentou a acurácia dos métodos. A hipótese para explicar esse fenômeno é que, devido à natureza exponencial do índice *Trie* para o problema do CATE, e ao algoritmo ICPAN para computação de nós ativos, a quantidade de nós ativos de borda aumenta consideravelmente junto com o aumento de τ; além disso, quanto mais erros são tolerados na busca no índice e mais profunda é a busca adiante nos níveis dos nós, menores são as chances de que eles contenham sugestões que respondam à consulta. Portanto, nesse caso a busca binária acaba auxiliando a não despender muito tempo processando esses nós.

Tal cenário de $\tau = 3$ pode apresentar uma vantagem prática da combinação de busca binária e sequencial no segundo nível, estando relacionado à questão de pesquisa iii). O método IP2LB obteve um tempo de processamento 3 vezes menor do que o IP2L e quase 3 vezes menor também do que o IP2LRB para $\lambda = 9$. Para $\lambda = 10$ houve uma redução dos tempos de processamento dos métodos IP2L e IP2LRB, porém o IP2LB ainda permaneceu à frente, e apresentando uma boa acurácia. Na Tabela 5 também há os tempos de processamento do método BEVA para τ variando de 1 a 3. O método desempenha muito bem para a base JUSBRASIL, obtendo média de tempo de processamento menor do que *100ms* em cada valor de τ. No entanto, como será demonstrado na seção 5.3.2, o BEVA consome quase o triplo de memória do que qualquer um dos três métodos descritos no

capítulo 4.

Considerando o limite de 100ms para tempo de resposta (Ji et al., 2009), os métodos IP2L e IP2LRB demonstram-se lentos para $\tau = 3$. O método IP2LB também ultrapassou essa marca em 55ms aproximadamente, apesar de ser o mais rápido dos métodos de nois níveis cenário. Os métodos IP2LB e IP2LRB nada mais são do que o método IP2L com a ativação da busca binária nos nós de borda, com diferentes critérios para ativá-la.

Há espaço para a criação de um modelo que utiliza a estratégia do IP2L para $\tau \leq 2$, onde a acurácia é mais importante, e a do IP2LB para $\tau > 2$ onde mais importa a velocidade de resposta, já que o espaço amostral de sugestões de consultas que devem ser retornadas aumenta bastante. A diferença de *F1-Score* entre o IP2LB e IP2LRB para $\tau = 3$ foi inexpressiva, portanto faz sentido utilizar o IP2LB nesse possível novo modelo por conta de seu menor tempo de processamento. Nesse caso, seria possível atingir a marca de 100ms para $\tau \leq 3$ em bases um pouco menores do que a JUSBRASIL. Por fim, a questão de pesquisa i) pode ser empiricamente respondida pelos experimentos realizados, os quais demonstram que não é possível adaptar sistemas de busca em dois níveis para tirarem proveito da busca binária no segundo nível sem que haja algum impacto na acurácia do método.

5.3 Avaliando os parâmetros dos métodos propostos

Nesta seção iremos realizar a seleção do parâmetro λ para os métodos IP2L, IP2LB e IP2LRB propostos no capítulo 4, com valores variando de 5 a 10. Não analisamos valores abaixo de 5 pois o processamento da consulta ficaria custoso demais devido à frequente ativação do segundo nível. Semelhantemente, também não analisamos valores acima de 10 pois grande parte do processamento da consulta ocorreria no primeiro nível. O objetivo dessa seção é estudar os efeitos de diferentes valores de λ nos três métodos e escolher o melhor valor para os métodos considerando a acurácia, o tempo de processamento das consultas, e também consumo de memória.

5.3.1 Tempo de processamento de prefixos de consulta

O tempo de processamento das consultas é uma métrica muito importante para avaliar sistemas de CATE. Essa métrica permite distinguir os algoritmos lentos dos rápidos para então escolher quais irão proporcionar a melhor experiência para o usuário, evitando atrasos notáveis e exibindo respostas a cada caractere digitado. Nós medimos o tempo de processamento em cada um dos dois níveis, portanto, o valor da coluna "total" nas tabelas dos experimentos é igual a soma dos valores das colunas "1º" e "2º" e representa o valor total do tempo de processamento. As Tabelas 7 e 8 contêm, considerando uma tolerância de $\tau = 1$ erros de digitação, as medições dos tempos de processamento para

os variados tamanhos de prefixos de consulta. Semelhantemente, as Tabelas 9 e 10 para $\tau = 2$ e as Tabelas 11 e 12 para $\tau = 3$.

Para facilitar a visualização dos dados contidos nas tabelas, também serão apresentadas figuras com um gráfico de barras empilhadas e agrupadas pelo valor de λ variando de 5 a 10, para cada tamanho do prefixo de consulta. Em cada gráfico, para cada valor de λ há um grupo de 3 barras, uma para cada método proposto no capítulo 4. Cada barra é empilhada, contendo os tempos de processamento do primeiro nível (cor mais intensa) e segundo nível (cor mais clara). A altura final das barras empilhadas representa o tempo total de processamento do método.

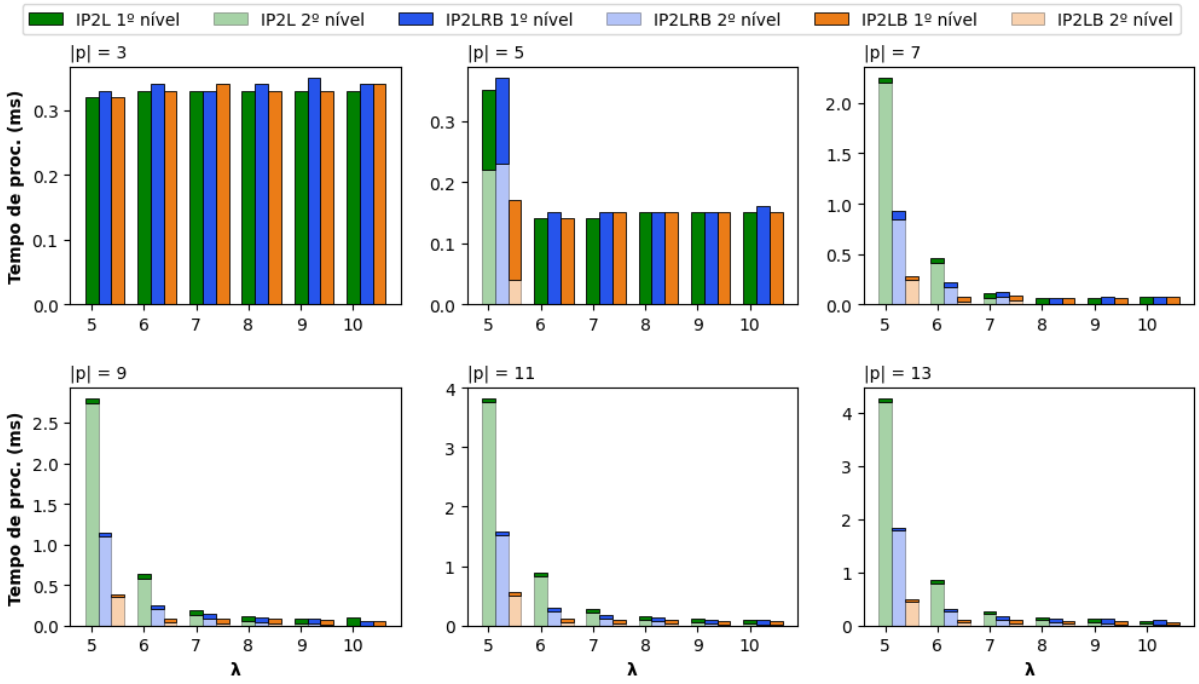


Figura 10: Gráficos de barras empilhadas agrupadas por método (IP2L, IP2LB e IP2LRB) com o tempo de processamento (ms) em cada valor de λ variando de 5 a 10, para $\tau = 1$ e a base AOL.

Para exemplificar, no primeiro gráfico para $|p| = 3$ e base AOL da Figura 10 pode-se observar que há apenas barras com cores intensas, sem o empilhamento. Isso se deve ao fato de que o segundo nível não foi ativado em nenhum dos três métodos para $\tau = 1$ e $\lambda = 5$ pois a condição $|p| + \tau \leq \lambda$ $(3 + 1 \leq 5)$ explicada na seção 4.2 foi satisfeita. Também é possível verificar na Tabela 7 a ausência de valores para a coluna "2º" no grupo de $|p| = 3$ junto às linhas dos métodos IP2L-5, IP2LB-5 e IP2LRB-5. À medida em que o tamanho de p vai aumentando, também o segundo nível vai sendo ativado para maiores valores de λ.

Um padrão evidente e interessante que ocorre nos gráficos de $|p| = 3$ e $|p| = 5$ na Figura 10 é a semelhança de altura das barras quando somente o primeiro nível é ativado. Isso acontece porque o algoritmo do primeiro nível, o ICPAN, é compartilhado

entre os três métodos, então essas barras representam simplesmente a média de tempo de processamento desse algoritmo para os devidos tamanhos de prefixo de consulta. É importante ressaltar que nas condições em que os métodos IP2L, IP2LB e IP2LRB foram apresentados no capítulo 4, qualquer um desses três métodos será equivalente ao ICPAN se $\lambda = \infty$. Nos 4 gráficos de barras seguintes da Figura 10 fica difícil notar esse mesmo padrão por conta do achatamento das barras devido aos altos tempos de processamento para $\lambda = 5$, no entanto podemos observar na Tabela 7 que quando o segundo nível não é ativado em um valor de λ qualquer, os valores de tempo total de processamento para os três métodos são estritamente próximos. Além disso, também é possível notar esse mesmo padrão para os valores de $\tau = 2$ e $\tau = 3$ na base AOL, e também para os valores de τ de 1 a 3 na base USADDR.

Para $|p| = 5$ podemos observar o segundo nível sendo ativado quando $\lambda = 5$. O tempo do primeiro nível permanece em torno de $0,14ms$ para os três métodos, mas a variação do tempo de processamento no segundo nível é quem determina boa parte do tempo de processamento final. No entanto, quanto maior o λ, menor é a influência do segundo nível no tempo de processamento final, chegando até mesmo a dividir valores semelhantes ao tempo do primeiro nível em alguns casos. Curiosamente o IP2LRB obteve um tempo ligeiramente maior no segundo nível em relação ao IP2L. Essa situação não acontece para valores maiores de $|p|$, porém se repete para as duas bases também com outros valores de τ, como podemos observar nas Figuras 11, 12, 13, e 15. Uma possível explicação é a de que o percentual de ativação de busca binária não é tão alto para esses casos no método IP2LRB, então uma parte significativa do processamento do segundo nível acaba sendo a busca sequencial, a qual é mais dispendiosa, enquanto as buscas binárias são executadas com frequência na situação de pior caso. O IP2LB foi duas vezes mais rápido do que o IP2L, mas vale relembrar que para $\tau = 1$ e $\tau = 2$ a acurácia dos métodos IP2LB e IP2LRB pode deixar a desejar.

Na Figura 11 é possível observar a diferença de ordem de grandeza do eixo vertical de Tempo de Processamento em comparação com a Figura 10, o que é esperado devido à magnitude de sugestões indexáveis da base USADDR. Todos os modelos em todos os valores de λ conseguem manter a média de tempo de processamento bem distante de $100ms$ para todos os tamanhos de prefixo de consulta testados e $\tau = 1$, mesmo para uma base maior como a USADDR. Também nota-se que os padrões de cada gráfico de barras da Figura 11 são muito similares aos padrões da Figura 10. Isso demonstra que uma diferença expressiva no tamanho da base como há entre USADDR e AOL não desestabiliza os métodos nos diversos valores de λ. Se o tempo de processamento estiver alto demais devido ao tamanho de uma base, desde que haja memória suficiente sempre é possível experimentar aumentar o valor de λ para encontrar um ponto de equilíbrio entre o desempenho e memória. Para $|p| = 3$ os três métodos em todos os valores de λ apresentaram tempo de processamento um pouco mais do que $6ms$, diminuindo bastante

	$\|p\| = 3$			$\|p\| = 5$			$\|p\| = 7$			$\|p\| = 9$			$\|p\| = 11$			$\|p\| = 13$		
Método	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**
IP2L-5	0,32		0,32	0,13	0,22	0,35	0,05	2,2	2,24	0,05	2,74	2,79	0,05	3,76	3,81	0,05	4,21	4,26
IP2LB-5	0,32		0,32	0,13	0,04	0,17	0,04	0,24	0,28	0,04	0,35	0,39	0,05	0,51	0,56	0,05	0,45	0,50
IP2LRB-5	0,33		0,33	0,14	0,23	0,36	0,08	0,85	0,94	0,05	1,1	1,15	0,06	1,52	1,58	0,06	1,78	1,84
IP2L-6	0,33		0,33	0,14		0,14	0,05	0,41	0,46	0,05	0,58	0,63	0,05	0,83	0,89	0,05	0,8	0,85
IP2LB-6	0,33		0,33	0,14		0,14	0,05	0,03	0,08	0,05	0,04	0,09	0,05	0,06	0,11	0,05	0,06	0,11
IP2LRB-6	0,34		0,34	0,15		0,15	0,05	0,17	0,22	0,05	0,2	0,26	0,05	0,24	0,29	0,05	0,26	0,31
IP2L-7	0,33		0,33	0,14		0,14	0,05	0,06	0,11	0,06	0,13	0,19	0,06	0,21	0,27	0,06	0,21	0,27
IP2LB-7	0,34		0,34	0,15		0,15	0,05	0,04	0,09	0,05	0,03	0,08	0,06	0,04	0,10	0,06	0,04	0,09
IP2LRB-7	0,33		0,33	0,15		0,15	0,05	0,07	0,12	0,06	0,08	0,14	0,06	0,12	0,18	0,06	0,11	0,17
IP2L-8	0,33		0,33	0,15		0,15	0,06		0,06	0,06	0,05	0,11	0,06	0,1	0,16	0,06	0,1	0,16
IP2LB-8	0,33		0,33	0,15		0,15	0,06		0,06	0,06	0,02	0,08	0,06	0,03	0,08	0,06	0,03	0,08
IP2LRB-8	0,34		0,34	0,15		0,15	0,06		0,06	0,06	0,04	0,10	0,06	0,07	0,13	0,06	0,07	0,13
IP2L-9	0,33		0,33	0,15		0,15	0,06		0,06	0,06	0,02	0,08	0,06	0,06	0,12	0,06	0,06	0,12
IP2LB-9	0,33		0,33	0,15		0,15	0,06		0,06	0,06	0,01	0,07	0,06	0,02	0,08	0,06	0,02	0,08
IP2LRB-9	0,35		0,35	0,15		0,15	0,08		0,08	0,06	0,02	0,08	0,06	0,04	0,10	0,08	0,04	0,12
IP2L-10	0,33		0,33	0,15		0,15	0,07		0,07	0,10		0,10	0,06	0,03	0,09	0,06	0,03	0,10
IP2LB-10	0,34		0,34	0,15		0,15	0,07		0,07	0,06		0,06	0,06	0,01	0,07	0,06	0,01	0,07
IP2LRB-10	0,34		0,34	0,16		0,16	0,07		0,07	0,06		0,06	0,07	0,02	0,09	0,08	0,02	0,10

Tabela 7: Tempo de processamento (ms) dos métodos IP2L, IP2LB e IP2LRB para prefixos de consulta com tamanho $3, 5, 6, 9, 11$ e 13, valores de λ variando de 5 a 10, e para $\tau = 1$ na base de dados AOL.

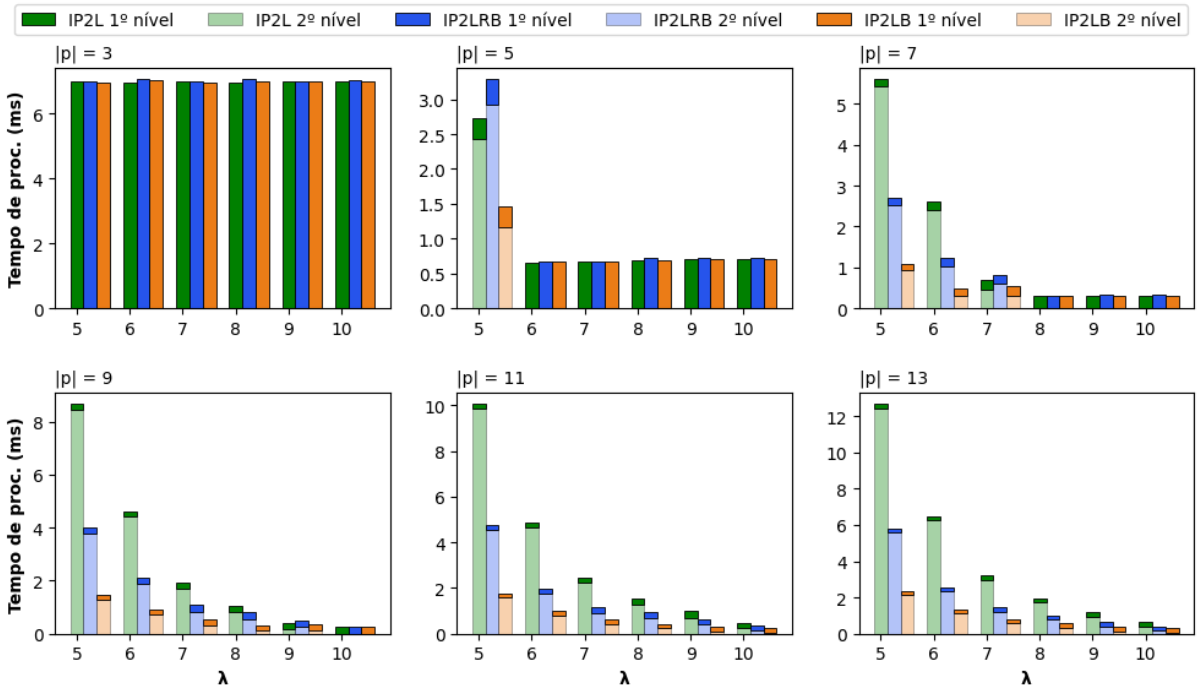


Figura 11: Gráficos de barras empilhadas agrupadas por método (IP2L, IP2LB e IP2LRB) com o tempo de processamento (ms) em cada valor de λ variando de 5 a 10, para $\tau = 1$ e a base USADDR.

para valores maiores de $|p|$. Isso ocorre porque buscas por prefixos pequenos são muito custosas em sistemas de CATE, pois o conjunto de nós ativos fica muito populoso (Xiao et al., 2013; da Gama Ferreira, 2020).

A eficiência do segundo nível do IP2LB devido à busca binária pode ser notada nos gráficos. A barra laranja de cor clara é menor do que as barras claras verde e azul na grande maioria dos casos, principalmente para valores menores de λ como 5 e 6. Para os

gráficos de $|p| = 11$ e $|p| = 13$ por exemplo, onde o segundo nível é ativado para todos os valores de λ, podemos observar que nos dois gráficos os tempos de processamento do método IP2LB começam bem mais abaixo do que os tempos dos outros dois métodos nos primeiros valores de λ. Tal padrão é comum em todos os gráficos de barras para $|p| = 11$ e $|p| = 13$ desta seção, ou seja, para as bases USADDR e AOL, e $\tau = 1$ até $\tau = 3$.

	$\|p\| = 3$			$\|p\| = 5$			$\|p\| = 7$			$\|p\| = 9$			$\|p\| = 11$			$\|p\| = 13$		
Método	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**
IP2L-5	6,99		6,99	0,30	2,43	2,73	0,19	5,41	5,60	0,21	8,44	8,64	0,20	9,85	10,06	0,21	12,44	12,66
IP2LB-5	6,96		6,96	0,30	1,16	1,45	0,17	0,93	1,10	0,18	1,3	1,47	0,18	1,59	1,77	0,20	2,14	2,34
IP2LRB-5	7,00		7,00	0,36	2,93	3,29	0,18	2,52	2,70	0,25	3,76	4,01	0,21	4,55	4,75	0,22	5,59	5,81
IP2L-6	6,96		6,96	0,66		0,66	0,20	2,4	2,60	0,21	4,42	4,63	0,21	4,66	4,87	0,22	6,25	6,47
IP2LB-6	7,02		7,02	0,67		0,67	0,19	0,31	0,50	0,20	0,71	0,91	0,19	0,8	0,99	0,21	1,12	1,32
IP2LRB-6	7,07		7,07	0,68		0,68	0,20	1,03	1,23	0,21	1,88	2,09	0,22	1,77	1,99	0,20	2,35	2,55
IP2L-7	6,98		6,98	0,68		0,68	0,23	0,47	0,69	0,21	1,7	1,91	0,21	2,22	2,43	0,22	2,98	3,20
IP2LB-7	6,96		6,96	0,68		0,68	0,23	0,31	0,54	0,20	0,32	0,52	0,20	0,44	0,64	0,23	0,58	0,80
IP2LRB-7	7,01		7,01	0,68		0,68	0,22	0,61	0,84	0,26	0,84	1,10	0,23	0,92	1,15	0,21	1,23	1,45
IP2L-8	6,96		6,96	0,69		0,69	0,31		0,31	0,22	0,82	1,04	0,23	1,3	1,53	0,23	1,74	1,97
IP2LB-8	7,00		7,00	0,69		0,69	0,32		0,32	0,21	0,12	0,34	0,21	0,23	0,44	0,23	0,33	0,56
IP2LRB-8	7,06		7,06	0,72		0,72	0,31		0,31	0,29	0,53	0,81	0,27	0,69	0,96	0,22	0,78	1,00
IP2L-9	6,98		6,98	0,70		0,70	0,32		0,32	0,23	0,19	0,43	0,34	0,68	1,03	0,24	0,93	1,17
IP2LB-9	7,00		7,00	0,70		0,70	0,31		0,31	0,24	0,12	0,36	0,23	0,1	0,33	0,25	0,15	0,40
IP2LRB-9	7,00		7,00	0,73		0,73	0,34		0,34	0,26	0,25	0,51	0,24	0,39	0,63	0,23	0,42	0,65
IP2L-10	7,00		7,00	0,70		0,70	0,31		0,31	0,27		0,27	0,24	0,23	0,47	0,24	0,4	0,64
IP2LB-10	6,98		6,98	0,70		0,70	0,32		0,32	0,27		0,27	0,23	0,03	0,26	0,25	0,05	0,31
IP2LRB-10	7,04		7,04	0,72		0,72	0,35		0,35	0,26		0,26	0,24	0,13	0,38	0,23	0,18	0,41

Tabela 8: Tempo de processamento (ms) dos métodos IP2L, IP2LB e IP2LRB para prefixos de consulta com tamanho $3, 5, 6, 9, 11$ e 13, valores de λ variando de 5 a 10, e para $\tau = 1$ na base de dados USADDR.

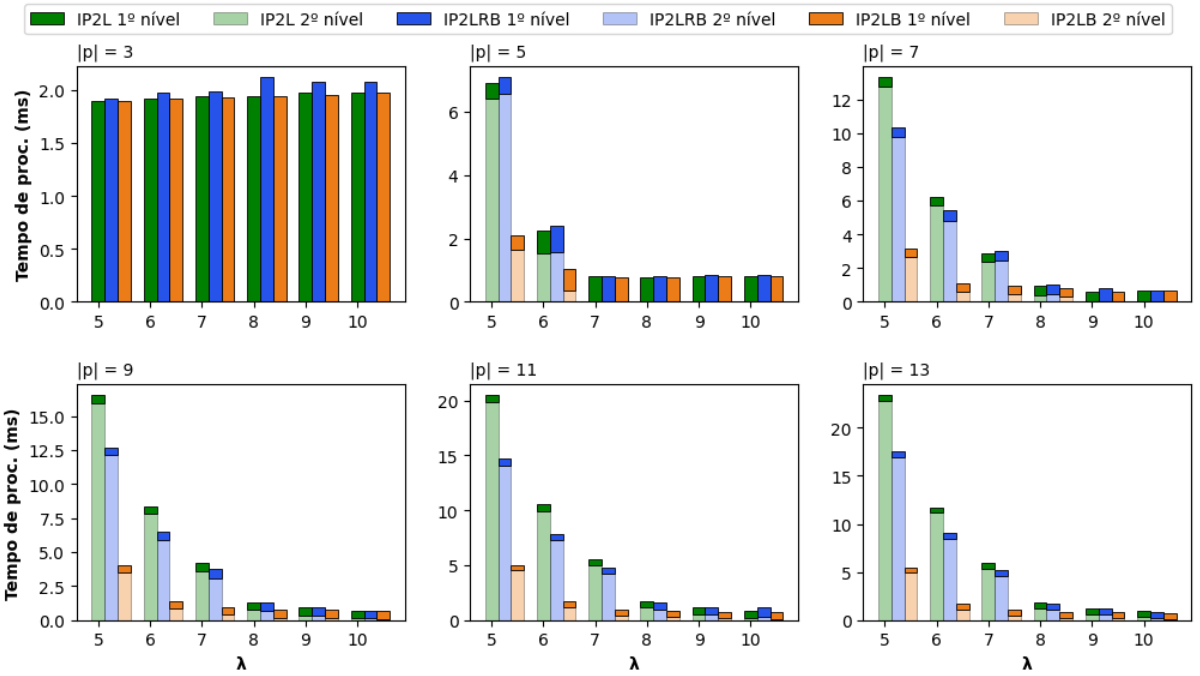


Figura 12: Gráficos de barras empilhadas agrupadas por método (IP2L, IP2LB e IP2LRB) com o tempo de processamento (ms) em cada valor de λ variando de 5 a 10, para $\tau = 2$ e a base AOL.

Para $\tau = 2$ houve um aumento em geral na ordem de grandeza da média de tempo

de processamento para valores mais baixos de λ como 5 e 6 para o IP2L, tanto para AOL quanto para USADDR, na qual tal aumento é ainda mais evidente. Na Tabela 8 temos por exemplo o IP2L-10 com $0,47ms$ para $|p| = 11$ contra $7,05ms$ para o mesmo método e tamanho de p na Tabela 10. O aumento no valor de τ faz com que o conjunto de nós ativos do primeiro nível cresça bastante, e como consequência a quantidade de itens para serem buscados sequencialmente também aumenta no segundo nível. No entanto, os métodos seguem tendo uma boa média de tempo para $\lambda = 10$. Podemos verificar na Tabela 9 para $|p| = 13$ por exemplo que o IP2L-10 foi cerca de 25 vezes mais rápido do que o IP2L-5, mas apesar disso o método IP2LB-10 foi somente 7 vezes mais rápido que o IP2LB-5. Comparando somente o tempo no segundo nível ainda para $|p| = 13$, por exemplo, temos que o segundo nível do IP2L-10 foi cerca de 81 vezes mais rápido que o do IP2L-5. Considerando que há pouca variação entre os tempos do primeiro nível à medida em que $|p|$ vai aumentando, um valor maior de λ de fato beneficia o segundo nível, pois o filtro do primeiro nível fica muito mais preciso.

Com a mudança do valor de τ o critério de ativação do segundo nível também mudou, por isso os gráficos da Figura 12 e 13 possuem padrões diferentes nas sequências de grupos de barras. Para $|p| = 5$ por exemplo o segundo nível agora é ativado para $\lambda = 6$, pois $|p| + \tau > \lambda$ $(5 + 2 > 6)$. Além disso, os tempos do IP2LRB ficaram mais relativamente próximos dos tempos do IP2L quando $\tau = 2$ tanto para a base AOL quanto para USADDR. Isso talvez tenha acontecido porque o aumento de τ causou uma multiplicação significativa de nós ativos no segundo nível, mas o percentual de ativação da busca binária foi bem baixo (o que é esperado para o IP2LRB em alguns casos, devido à sua condição de restrição de ativação).

	$\|p\| = 3$			$\|p\| = 5$			$\|p\| = 7$			$\|p\| = 9$			$\|p\| = 11$			$\|p\| = 13$		
Método	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**
IP2L-5	1,89		1,89	0,49	6,4	6,89	0,53	12,78	13,32	0,54	15,96	16,50	0,56	19,89	20,44	0,56	22,8	23,37
IP2LB-5	1,89		1,89	0,46	1,64	2,10	0,50	2,64	3,14	0,51	3,53	4,03	0,52	4,51	5,03	0,54	4,9	5,44
IP2LRB-5	1,92		1,92	0,53	6,56	7,09	0,56	9,75	10,31	0,58	12,11	12,68	0,59	14,11	14,71	0,67	16,91	17,58
IP2L-6	1,92		1,92	0,70	1,54	2,24	0,52	5,68	6,21	0,53	7,81	8,34	0,57	9,95	10,52	0,56	11,15	11,70
IP2LB-6	1,92		1,92	0,70	0,34	1,04	0,51	0,56	1,07	0,52	0,86	1,39	0,53	1,15	1,68	0,53	1,15	1,67
IP2LRB-6	1,97		1,97	0,80	1,58	2,37	0,66	4,78	5,44	0,57	5,92	6,49	0,59	7,25	7,84	0,70	8,41	9,11
IP2L-7	1,94		1,94	0,79		0,79	0,55	2,35	2,90	0,56	3,62	4,18	0,59	4,95	5,54	0,58	5,33	5,91
IP2LB-7	1,93		1,93	0,76		0,76	0,54	0,44	0,98	0,53	0,38	0,91	0,56	0,44	1,01	0,56	0,47	1,03
IP2LRB-7	1,98		1,98	0,81		0,81	0,56	2,46	3,02	0,64	3,09	3,73	0,63	4,18	4,81	0,65	4,58	5,23
IP2L-8	1,94		1,94	0,77		0,77	0,56	0,41	0,97	0,56	0,76	1,33	0,61	1,13	1,74	0,60	1,26	1,86
IP2LB-8	1,94		1,94	0,78		0,78	0,55	0,28	0,83	0,57	0,19	0,76	0,58	0,25	0,84	0,59	0,26	0,85
IP2LRB-8	2,12		2,12	0,82		0,82	0,58	0,43	1,00	0,61	0,71	1,33	0,65	0,97	1,62	0,65	1,09	1,74
IP2L-9	1,97		1,97	0,80		0,80	0,61		0,61	0,58	0,34	0,92	0,63	0,53	1,16	0,62	0,6	1,21
IP2LB-9	1,95		1,95	0,79		0,79	0,61		0,61	0,60	0,14	0,74	0,61	0,16	0,78	0,61	0,17	0,78
IP2LRB-9	2,07		2,07	0,85		0,85	0,82		0,82	0,61	0,36	0,98	0,68	0,5	1,17	0,62	0,54	1,15
IP2L-10	1,97		1,97	0,80		0,80	0,65		0,65	0,60	0,12	0,71	0,63	0,24	0,87	0,64	0,28	0,92
IP2LB-10	1,97		1,97	0,81		0,81	0,64		0,64	0,59	0,07	0,65	0,63	0,08	0,71	0,63	0,09	0,72
IP2LRB-10	2,08		2,08	0,86		0,86	0,67		0,67	0,60	0,12	0,71	0,92	0,27	1,19	0,65	0,25	0,90

Tabela 9: Tempo de processamento (ms) dos métodos IP2L, IP2LB e IP2LRB para prefixos de consulta com tamanho $3, 5, 6, 9, 11$ e 13, valores de λ variando de 5 a 10, e para $\tau = 2$ na base de dados AOL.

Podemos observar através da Tabela 10 que o modelo IP2L-5 dificilmente poderia

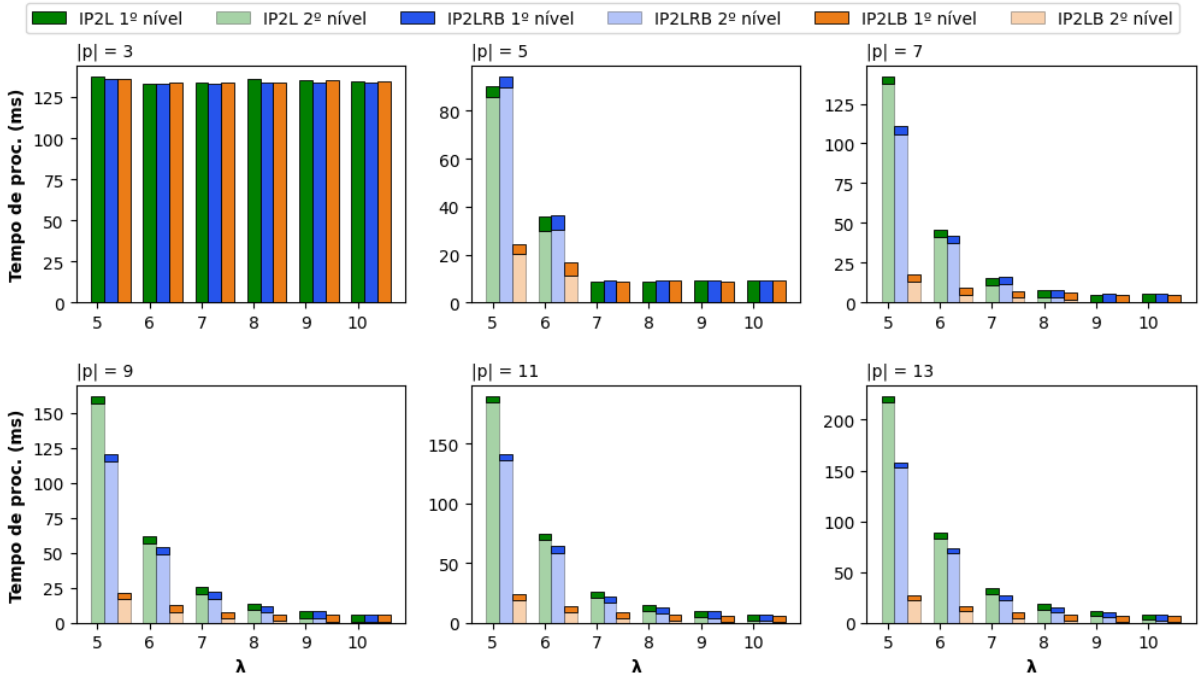


Figura 13: Gráficos de barras empilhadas agrupadas por método (IP2L, IP2LB e IP2LRB) com o tempo de processamento (ms) em cada valor de λ variando de 5 a 10, para $\tau = 2$ e a base USADDR.

ser utilizado em um sistema real com base tão grande quanto a USADDR. O único caso em que há média de tempo abaixo de $100ms$ é para $|p| = 5$ e isso quando τ ainda é igual a 2. Já o modelo IP2L-10 possui média igual a $8,12ms$ para $|p| = 13$, por exemplo. Essas evidências reforçam a ideia de que deve ser considerado um fator de proporcionalidade entre o valor de λ e o tamanho da base de sugestões que será indexada, visando um equilíbrio entre a quantidade de memória utilizada e o tempo de processamento.

É importante ressaltar que os três métodos atingiram tempo maior que $100ms$ para $|p| = 3$ na base USADDR, como indica claramente o primeiro gráfico da Figura 13. Essa situação pode ser problemática em um sistema real de CATE com base de tamanho similar à USADDR. Considerando $|p| = 3$, observamos nas Tabelas 8 e 10 que o aumento de $\tau = 1$ para $\tau = 2$ levou uma média de tempo de processamento dos métodos de $7ms$ para $134ms$. Nesse cenário não se pode esperar bons resultados para $\tau = 3$. Essa expectativa negativa é confirmada no primeiro gráfico da Figura 11. No entanto, para amenizar esse problema uma opção é utilizar uma técnica de *cache* para manter conjuntos de nós pré-ativados para consultas pequenas de tamanho menor ou igual a 3, por exemplo. Esse tipo de técnica de *cache* está presente na implementação do BEVA utilizada nesta pesquisa, e comprovadamente provoca grande diminuição no tempo de processamento para prefixos de consulta pequenos como $|p| \leq 3$, como veremos adiante na seção 5.4.

Para $\tau = 3$ ocorre uma "anomalia" no gráfico de $|p| = 3$ tanto para a base AOL quanto para USADDR. Uma vez que $|p| + \tau > \lambda$ $(3 + 3 > 5)$ o segundo nível é ativado,

	$\|p\| = 3$			$\|p\| = 5$			$\|p\| = 7$			$\|p\| = 9$			$\|p\| = 11$			$\|p\| = 13$		
Método	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**
IP2L-5	137,40		137,40	4,15	85,93	90,08	4,39	137,62	142,02	4,78	156,54	161,32	5,00	184,45	189,45	5,02	217,55	222,57
IP2LB-5	135,80		135,80	4,11	20,16	24,27	4,04	13,49	17,52	4,34	17,11	21,45	4,38	19,24	23,62	4,73	21,69	26,42
IP2LRB-5	136,31		136,31	4,36	89,91	94,27	4,89	105,91	110,81	4,90	115,56	120,46	5,14	136,14	141,28	5,09	152,69	157,78
IP2L-6	133,11		133,11	5,93	29,82	35,76	4,63	41,2	45,83	5,02	56,85	61,87	5,16	69,2	74,36	5,42	83,32	88,74
IP2LB-6	133,65		133,65	5,82	11,11	16,93	4,28	5,16	9,44	4,62	7,64	12,26	4,74	9,2	13,93	5,03	11,48	16,50
IP2LRB-6	133,36		133,36	5,96	30,36	36,32	4,74	37,07	41,82	5,27	48,9	54,16	5,31	58,76	64,07	5,32	68,33	73,65
IP2L-7	133,90		133,90	8,66		8,66	4,45	10,93	15,38	4,93	20,44	25,37	4,96	21,4	26,36	5,20	28,4	33,60
IP2LB-7	133,71		133,71	8,91		8,91	4,39	3,04	7,43	4,65	3,09	7,74	4,84	3,89	8,74	5,12	4,84	9,96
IP2LRB-7	133,32		133,32	9,11		9,11	4,40	11,47	15,88	5,11	16,85	21,97	5,15	17,2	22,35	5,21	22,22	27,43
IP2L-8	136,30		136,30	8,82		8,82	4,53	3,09	7,62	4,83	8,88	13,71	4,99	9,91	14,90	5,12	13,24	18,36
IP2LB-8	133,62		133,62	9,04		9,04	4,49	1,56	6,05	4,67	1,25	5,92	4,92	1,89	6,80	5,30	2,54	7,84
IP2LRB-8	133,62		133,62	9,28		9,28	4,63	3,15	7,77	4,83	7,24	12,08	5,26	7,74	13,00	5,19	10,12	15,31
IP2L-9	135,12		135,12	9,02		9,02	4,95		4,95	4,91	3,05	7,96	5,01	4,54	9,55	5,21	6,45	11,66
IP2LB-9	135,07		135,07	8,95		8,95	4,94		4,94	4,85	0,96	5,80	5,03	0,93	5,97	5,44	1,39	6,83
IP2LRB-9	133,89		133,89	9,34		9,34	5,28		5,28	5,08	3,28	8,36	5,66	3,88	9,54	5,26	5,09	10,35
IP2L-10	134,59		134,59	9,24		9,24	5,22		5,22	5,04	0,79	5,83	5,14	1,91	7,05	5,31	2,81	8,12
IP2LB-10	134,60		134,60	9,31		9,31	5,08		5,08	4,99	0,54	5,54	5,08	0,45	5,53	5,59	0,69	6,28
IP2LRB-10	133,78		133,78	9,26		9,26	5,44		5,44	4,96	0,81	5,76	5,42	1,65	7,07	5,44	2,19	7,64

Tabela 10: Tempo de processamento (ms) dos métodos IP2L, IP2LB e IP2LRB para prefixos de consulta com tamanho $3, 5, 6, 9, 11$ e 13, valores de λ variando de 5 a 10, e para $\tau = 2$ na base de dados USADDR.

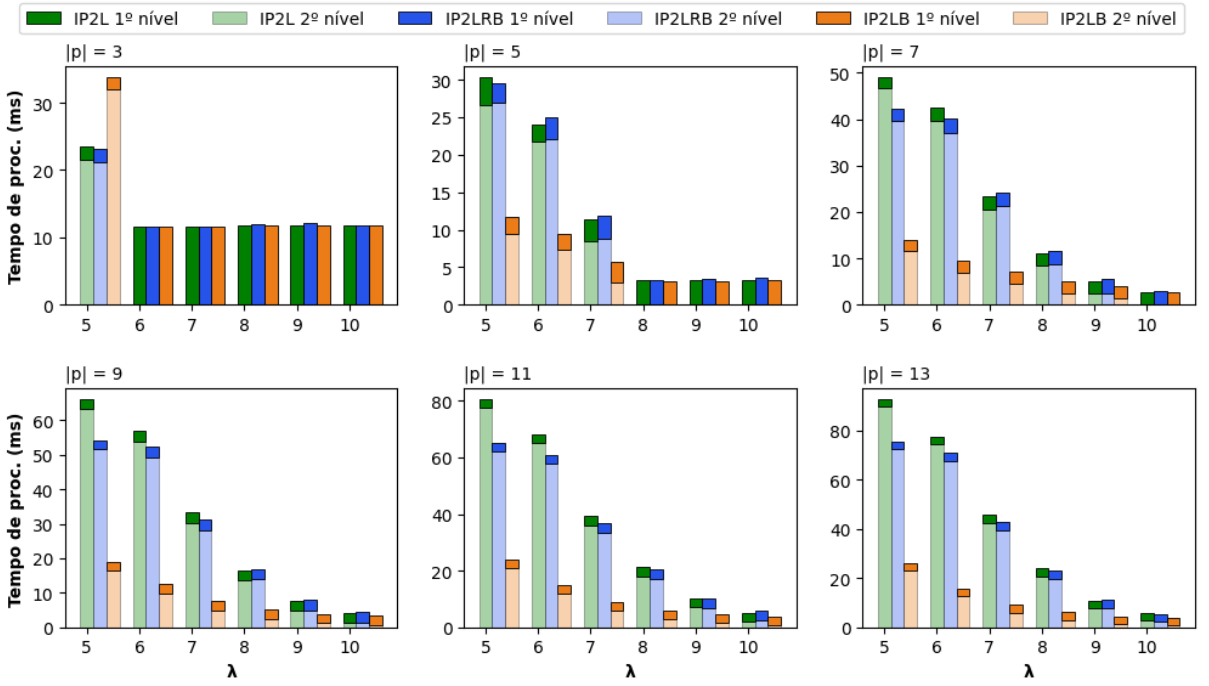


Figura 14: Gráficos de barras empilhadas agrupadas por método (IP2L, IP2LB e IP2LRB) com o tempo de processamento (ms) em cada valor de λ variando de 5 a 10, para $\tau = 3$ e a base AOL.

e nesse caso a média de tempo de processamento IP2LB se mostrou maior que a dos outros dois métodos. No método IP2LB a busca binária é ativada em grande parte dos nós ativos, tendo muita participação no tempo total de execução do segundo nível, então é plausível considerar que essa diferença acontece possivelmente devido à busca binária estar se aproximando com frequência do pior caso, e também estar sendo realizada em listas muito grandes. Além disso, na base USADDR os três métodos não tiveram um bom desempenho para $|p| = 3$ em nenhum valor de λ, com tempos de processamento maiores

do que $500ms$. Os métodos originais ICAN e ICPAN também sofrem com esse problema, como veremos mais adiante na seção 5.4. Como mencionado anteriormente essa situação pode ser mitigada ao utilizar estratégias de *cache* de nós ativos.

Na base AOL os tempos de processamento para $\lambda = 5$ com tamanhos de prefixos $|p| = 11$ e $|p| = 13$ já começam a se aproximar de $100ms$ nos modelos IP2L e IP2LRB. No entanto, vale ressaltar que o IP2L-5 por exemplo utiliza duas vezes menos memória do que o IP2L nessa base. Se um sistema real de CATE com base de sugestões de tamanho similar à AOL receber poucas consultas com tamanho maior do que 13 caracteres (o que é bem provável pois consultas longas são menos frequentes nesses sistemas), vale a pena considerar utilizar o IP2L-5 ou IP2L-6, pois apresentariam um equilíbrio interessante entre memória e desempenho pois demonstraram valores de tempo de processamento abaixo de $100ms$ em todos os tamanhos de prefixo de consulta testados para $\tau = 1$ e $\tau = 2$, como se pode conferir nas Tabelas 7 e 9, além de possuírem acurácia garantida.

Até então, para os casos em que o segundo nível foi ativado em todos os valores de λ para as duas bases e os valores de τ, todos os métodos demonstraram uma tendência exponencial decrescente em função de λ para os valores de tempo de processamento, configurando uma curva que se aproxima do padrão $f(\lambda) = a \cdot e^{-b \cdot \lambda}$, no qual e é a constante de *Euler*, e a e b são constante reais tal que $0 < b < 1$. Por exemplo, as funções que podemos obter a partir da interpolação dos valores de tempo total de processamento na base USADDR com $\tau = 1$ e $|p| = 9$ para os métodos IP2L, IP2LB e IP2LRB são respectivamente $T_{IP2L}(\lambda) = 209 \cdot e^{-0,61\lambda}$, $T_{IP2LB}(\lambda) = 13 \cdot e^{-0,41\lambda}$, e $T_{IP2LRB}(\lambda) = 59,7 \cdot e^{-0,52\lambda}$. Vale relacionar esse padrão exponencial decrescente com o fato de que a quantidade de nós no índice *Trie* segue uma tendência exponencial crescente em função do λ. À medida em que λ vai aumentando, a quantidade de nós indexados na *Trie* aumenta e o tempo de processamento diminui até que se estabilize em uma "constante" (cada vez mais próximo do método ICPAN original).

Podemos observar nas Figuras 14 e 15 que o desempenho do segundo nível do IP2L é muito afetado com um τ maior como $\tau = 3$ e valores de $|p|$ maiores do que λ. As listas de candidatos pro segundo nível nesses casos são imensas, tornando muito custoso realizar a computação da distância de edição em grande quantidade. O modelo IP2LRB consegue se beneficiar da ativação da busca binária mesmo que em um baixo percentual de nós, tendo sua média de tempo de processamento um pouco reduzida. No entanto, fica bem nítida a diferença de utilizar a busca binária em todos os nós de borda nesses casos como o IP2LB faz, pois ele se demonstrou ser de 2 a 6 vezes mais rápido para $|p| \geq 5$. Nota-se que há semelhança nos padrões das barras entre as duas bases para um mesmo valor de τ. A mudança que ocorre é basicamente na escala do eixo vertical. Essa característica é interessante porque provavelmente indica que os métodos possuem os mesmos padrões de desempenho em bases de tamanhos muito diferentes, não tendo casos de anomalia

causados por uma base muito grande, por exemplo.

Por fim, considerando o limite de $100ms$, na base USADDR o único valor de λ para o qual os métodos desempenham bem é $\lambda = 10$. O IP2L-10 e IP2LRB-10 obtiveram respectivamente uma média de $106,94ms$ e $105,68ms$ para $|p| = 11$, além de $117,05ms$ e $114,52ms$ para $|p| = 13$. Já o método IP2LB obteve $90,67ms$ para $|p| = 11$ e $97,25ms$ para $|p| = 13$, permanecendo ainda como o mais rápido.

	$\|p\| = 3$			$\|p\| = 5$			$\|p\| = 7$			$\|p\| = 9$			$\|p\| = 11$			$\|p\| = 13$		
Método	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**
IP2L-5	1,89	21,59	23,48	3,66	26,66	30,33	2,44	46,62	49,06	2,55	63,39	65,95	2,74	77,6	80,34	2,75	89,67	92,42
IP2LB-5	1,83	31,96	33,79	2,19	9,46	11,65	2,33	11,6	13,93	2,44	16,6	19,04	2,88	21,07	23,95	2,62	23,27	25,89
IP2LRB-5	1,85	21,23	23,07	2,52	27,01	29,54	2,65	39,68	42,33	2,74	51,56	54,30	2,83	62,21	65,05	2,98	72,22	75,20
IP2L-6	11,53		11,53	2,27	21,78	24,05	2,85	39,62	42,46	2,98	53,9	56,88	3,20	65,07	68,27	3,18	74,41	77,59
IP2LB-6	11,54		11,54	2,18	7,34	9,53	2,67	6,91	9,58	2,83	9,86	12,69	2,97	11,75	14,72	2,95	12,78	15,73
IP2LRB-6	11,61		11,61	2,89	22,08	24,97	3,02	37,14	40,16	3,14	49,25	52,39	3,38	57,6	60,98	3,53	67,63	71,15
IP2L-7	11,66		11,66	2,86	8,53	11,40	2,94	20,42	23,35	3,13	30,29	33,42	3,28	35,98	39,26	3,32	42,51	45,83
IP2LB-7	11,61		11,61	2,83	2,9	5,72	2,78	4,44	7,22	2,90	4,86	7,76	3,12	5,74	8,85	3,08	5,98	9,05
IP2LRB-7	11,66		11,66	3,15	8,77	11,92	3,06	21,18	24,24	3,17	28,11	31,28	3,41	33,16	36,58	3,52	39,14	42,66
IP2L-8	11,70		11,70	3,21		3,21	2,70	8,41	11,11	2,92	13,69	16,61	3,12	18,01	21,12	3,16	20,68	23,83
IP2LB-8	11,83		11,83	3,14		3,14	2,59	2,57	5,17	2,81	2,36	5,17	3,07	2,99	6,06	3,00	3,07	6,08
IP2LRB-8	11,97		11,97	3,28		3,28	2,96	8,67	11,63	2,94	13,87	16,81	3,24	17,04	20,28	3,31	19,82	23,13
IP2L-9	11,72		11,72	3,21		3,21	2,57	2,46	5,04	2,80	4,87	7,67	3,04	7,02	10,06	2,96	7,95	10,91
IP2LB-9	11,72		11,72	3,20		3,20	2,55	1,38	3,93	2,72	1,2	3,91	2,95	1,47	4,42	2,91	1,52	4,42
IP2LRB-9	12,17		12,17	3,48		3,48	3,04	2,57	5,61	3,07	4,93	8,00	3,37	6,88	10,24	3,30	7,72	11,02
IP2L-10	11,81		11,81	3,23		3,23	2,77		2,77	2,71	1,39	4,10	2,95	2,19	5,15	2,96	2,59	5,55
IP2LB-10	11,86		11,86	3,22		3,22	2,77		2,77	2,71	0,72	3,43	2,92	0,81	3,73	2,92	0,83	3,74
IP2LRB-10	11,85		11,85	3,56		3,56	2,94		2,94	3,11	1,43	4,55	3,40	2,29	5,70	2,96	2,49	5,45

Tabela 11: Tempo de processamento (ms) dos métodos IP2L, IP2LB e IP2LRB para prefixos de consulta com tamanho $3, 5, 6, 9, 11$ e 13, valores de λ variando de 5 a 10, e para $\tau = 3$ na base de dados AOL.

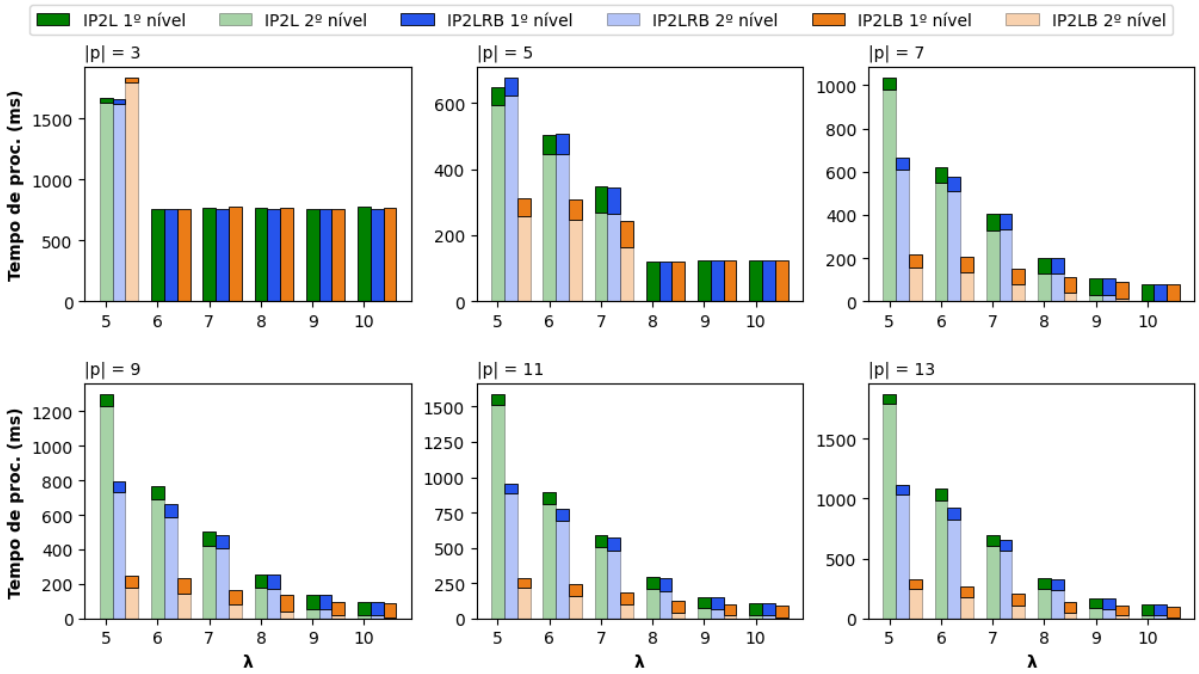


Figura 15: Gráficos de barras empilhadas agrupadas por método (IP2L, IP2LB e IP2LRB) com o tempo de processamento (ms) em cada valor de λ variando de 5 a 10, para $\tau = 3$ e a base USADDR.

	$\|p\| = 3$			$\|p\| = 5$			$\|p\| = 7$			$\|p\| = 9$			$\|p\| = 11$			$\|p\| = 13$		
Método	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**	**1º**	**2º**	**total**
IP2L-5	41,39	1625,46	1.666,86	53,27	593,77	647,05	56,90	977,63	1.034,54	63,45	1233,59	1.297,04	69,20	1514,61	1.583,81	74,64	1790,53	1.865,17
IP2LB-5	36,97	1795,61	1.832,59	53,68	256,66	310,34	61,36	157,97	219,33	65,11	179,83	244,94	69,46	215,19	284,65	75,59	246,6	322,20
IP2LRB-5	40,79	1614,34	1.655,14	53,94	622,48	676,42	57,86	609,44	667,30	64,44	729,97	794,41	71,53	885,04	956,57	75,67	1036,04	1.111,71
IP2L-6	757,67		757,67	58,56	445,29	503,85	71,06	550,25	621,31	77,33	688,58	765,90	84,22	812,75	896,98	97,82	983,63	1.081,45
IP2LB-6	753,98		753,98	60,19	246,68	306,87	72,04	134,18	206,22	91,43	144,78	236,21	83,84	160,45	244,29	90,18	180,22	270,40
IP2LRB-6	757,52		757,52	59,61	446,8	506,41	71,78	507,34	579,12	77,71	584,64	662,35	85,72	687,85	773,56	91,75	827,45	919,19
IP2L-7	762,05		762,05	77,60	268,94	346,54	78,44	327,75	406,19	80,88	420,2	501,08	87,99	502,22	590,21	92,55	602,49	695,04
IP2LB-7	776,22		776,22	79,32	163,18	242,50	72,95	76,95	149,90	81,90	82,62	164,52	90,41	95,85	186,26	95,57	108,26	203,82
IP2LRB-7	758,10		758,10	77,28	266,47	343,75	73,56	331,23	404,79	81,73	403,39	485,13	87,21	481,91	569,12	93,28	560,85	654,14
IP2L-8	763,83		763,83	119,97		119,97	71,12	129,78	200,90	80,09	177,22	257,31	84,63	206,1	290,73	88,22	245,58	333,80
IP2LB-8	767,97		767,97	120,75		120,75	71,36	39,17	110,53	95,76	38,54	134,30	86,43	41,6	128,03	90,55	49,52	140,06
IP2LRB-8	756,98		756,98	119,51		119,51	73,81	126,84	200,64	83,00	173,06	256,07	89,44	195,64	285,08	89,38	240,48	329,86
IP2L-9	761,45		761,45	124,54		124,54	73,61	31,91	105,52	80,44	55,04	135,48	83,15	70,06	153,21	86,41	83,45	169,85
IP2LB-9	761,63		761,63	124,65		124,65	74,05	14,17	88,21	79,34	17,14	96,48	83,38	18,81	102,20	88,58	22,19	110,76
IP2LRB-9	758,85		758,85	122,46		122,46	74,57	31,53	106,10	79,59	56,64	136,23	84,99	67,52	152,51	87,77	79,25	167,02
IP2L-10	777,05		777,05	124,02		124,02	77,19		77,19	79,20	15,92	95,12	84,51	22,43	106,94	87,83	29,22	117,05
IP2LB-10	762,56		762,56	123,70		123,70	77,57		77,57	78,73	7,7	86,43	82,84	7,84	90,67	87,37	9,88	97,25
IP2LRB-10	761,06		761,06	124,05		124,05	78,20		78,20	81,38	16,18	97,56	84,00	21,68	105,68	87,05	27,47	114,52

Tabela 12: Tempo de processamento (ms) dos métodos IP2L, IP2LB e IP2LRB para prefixos de consulta com tamanho $3, 5, 6, 9, 11$ e 13, valores de λ variando de 5 a 10, e para $\tau = 3$ na base de dados USADDR.

5.3.2 Consumo de memória

Os sistemas de CATE mantêm seus índices e os textos das sugestões armazenados diretamente na memória pois isso possibilita uma maior velocidade de acesso às informações e também responder às consultas por prefixos com mais velocidade. No entanto, é necessário equilibrar bem essa relação entre tempo de processamento e memória, pois ela é um recurso limitado e custoso. Nesse cenário, quanto maior o índice maior é a quantidade de memória utilizada. Por exemplo método IncNG *Trie* (Xiao et al., 2013), apesar de ser o mais rápido presente na literatura, consome uma imensa quantidade de memória, o que dificulta utilizá-lo em alguns cenários práticos.

A Tabela 13 apresenta a quantidade média de memória (para todas as medições de memória para τ variando de 1 a 3 e $|p|$ variando de 3 a 13) em *MegaBytes* de cada um dos três métodos propostos, com λ variando de 5 a 10, para cada uma das bases AOL, USADDR, JUSBRASIL. A Figura 16 contém um gráfico de barras que representa os dados da Tabela 13 para o consumo de memória dos métodos para a base USADDR.

Podemos observar tanto na Tabela 13 que a média de memória utilizada pelos três métodos é muito similar para cada valor de λ experimentado, restando apenas a diferença de tempo de processamento para se comparar entre os métodos para um mesmo valor de λ. No entanto, as quantidades de memória diferem de forma relevante entre $\lambda = 5$ e $\lambda = 8$, ou $\lambda = 6$ e $\lambda = 10$, por exemplo. Na base USADDR o método IP2L-5 utilizou uma média de 1283,41 *MegaBytes* de memória, enquanto o IP2L-8 utilizou 2244,95, quase 75% a mais. Para a mesma base, o método IP2LB-6 utilizou 1446,06 enquanto o IP2LB-10 consumiu cerca de 250% a mais com 3675,87. Nota-se que à medida em que λ aumenta, cresce cada vez mais a diferença entre a quantidade de memória utilizada em relação ao valor anterior de λ.

A Figura 16 mostra o padrão de crescimento da média de memória utilizada para a base USADDR, por exemplo. A tendência de crescimento parece ser uma função expo-

Método	AOL Memória (MB)	USADDR Memória (MB)	JUSBRASIL Memória (MB)
IP2L-5	37,99	1283,41	4757,45
IP2LRB-5	38,01	1282,76	4757,45
IP2LB-5	37,57	1253,34	4604,35
IP2L-6	45,22	1456,02	4733,27
IP2LRB-6	45,20	1455,46	4733,30
IP2LB-6	45,03	1446,06	4526,59
IP2L-7	55,92	1771,89	4680,71
IP2LRB-7	55,96	1771,86	4680,71
IP2LB-7	55,88	1770,87	4680,68
IP2L-8	69,22	2244,95	5018,19
IP2LRB-8	69,23	2244,90	5018,26
IP2LB-8	69,14	2243,90	5018,23
IP2L-9	84,22	2881,84	5604,38
IP2LRB-9	84,21	2881,87	5604,40
IP2LB-9	84,17	2881,72	5604,42
IP2L-10	100,67	3675,90	6496,82
IP2LRB-10	100,70	3675,88	6496,84
IP2LB-10	100,70	3675,87	6496,88

Tabela 13: Quantidades de memória em *MegaBytes* utilizadas pelos métodos IP2L, IP2LB, e IP2LRB, variando o parâmetro λ de 5 até 10, para as bases AOL, USADDR, e JUSBRASIL.

nencial crescente, representada na Figura 16 pela linha azul que acompanha os grupos de barras. Quando não há muitos prefixos em comum nos itens indexados em uma *Trie* ela pode acabar crescendo em um passo exponencial no tamanho do alfabeto Σ. No entanto, vale ressaltar que a quantidade de memória utilizada pode estagnar em uma constante para um valores de λ muito grandes. Se a maior sugestão de consulta de uma base possuir um tamanho de 20 caracteres, as quantidade de memória utilizadas para cada valor de λ para $\lambda \geq 20$ serão iguais. Feita a ressalva, podemos analisar o gráfico "localmente" para $\lambda \leq 10$.

A linha de tendência apresentada na Figura 16 é uma representação da função $f(\lambda) = 1190 \cdot e^{0,22 \cdot \lambda}$. Ora, esse é um formato de função (exponencial crescente) similar às funções de tendência de tempo de processamento (exponencial decrescente) apresentadas na seção 5.3.1. Quando há poucos nós na *Trie* utilizada no primeiro nível o tempo de processamento é bem alto, mas à medida em que λ aumenta, também há um crescimento com tendência exponencial no número total de nós dessa *Trie* que por consequência aumenta a quantidade de informação disponível para a busca tolerante a erros, reduzindo o tempo de processamento. Essa é uma forte evidência da relação entre a memória e desempenho de um método de CATE, principalmente nos métodos que seguem a abordagem em dois níveis.

Para a base JUSBRASIL ocorreu uma peculiaridade nas quantidades de memória:

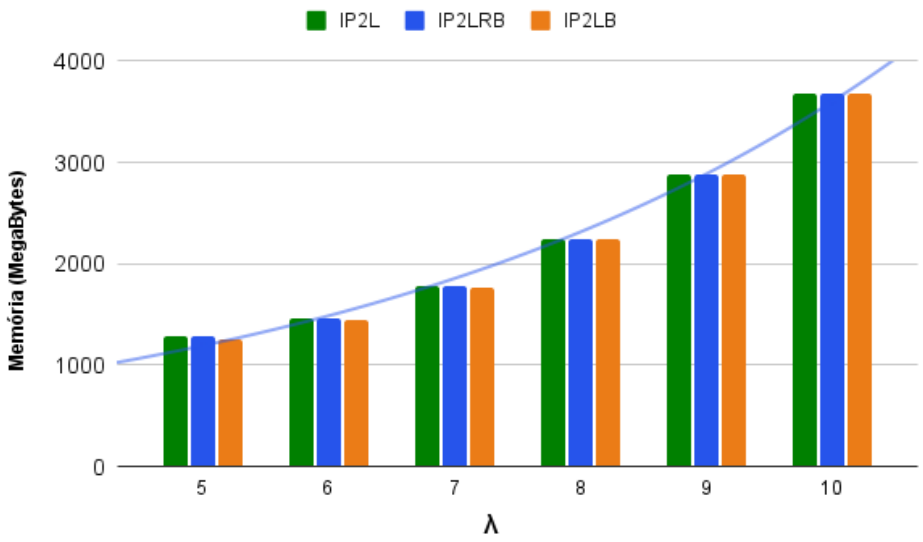


Figura 16: Gráfico de barras agrupadas por método (IP2L, IP2LB e IP2LRB) com a média de memória (*MegaBytes*) utilizada para cada valor de λ variando de 5 a 10, na base USADDR.

entre $\lambda = 5$ e $\lambda = 7$ a memória utilizada vai diminuindo em um passo lento, e só a a partir de $\lambda = 8$ começa a aumentar, com um passo mais rápido Isso acontece provavelmente devido à natureza da base, a qual foi extraída de *logs* de um sistema real de CATE. É mais frequente a quantidade de prefixos em comum entre os primeiros caracteres das sugestões de consulta. A partir de $\lambda = 8$ os prefixos já começam a ficar mais difusos, e a Trie começa a crescer bastante em largura, aumentando quantidade de nós necessários para indexá-los.

O contexto e cenário de um sistema de CATE podem influenciar a escolha do valor ideal para λ. Deve-se levar em conta o tamanho médio dos prefixos de consulta que se espera receber no sistema, o número de sugestões na base para se indexar, o tamanho médio dessas sugestões, e a quantidade de memória disponível.

Por fim, selecionamos os modelos IP2L-10, IP2LRB-10 e IP2LB-10 para serem comparados com os *baselines* na seção 5.4 a seguir. Dentre os valores de λ e τ experimentados o valor $\lambda = 10$ foi o que mais aproximou o tempo de processamento dos métodos do limite de $100ms$ para bases grandes como USADDR e JUSBRASIL. Além disso, mesmo para $\lambda = 10$ a economia de memória em relação ao método ICPAN original chega a ser mais de 50% como será demonstrado na próxima seção.

5.4 Comparação com os métodos anteriores

Nesta seção iremos comparar os métodos IP2L-10, IP2LB-10 e IP2LRB-10 com os métodos ICAN, ICPAN, e META (cujos códigos foram providos pelos autores) e também

o método BEVA (implementação própria) nas bases AOL e USADDR. Analisaremos as médias de tempo de processamento em dois cenários: (1) variando o limiar τ de distância de edição para um prefixo de consulta mais curto de tamanho 5, e também mais longo, de tamanho 13; (2) média de tempos de processamento para $\tau = 3$, variando o tamanho do prefixo de 3 a 13, de 2 em 2. Por fim, analisaremos também a utilização de memoria.

5.4.1 Variando o valor do limiar de distância de edição

As Tabelas 14 e 15 contêm os tempos de processamento para todos os algoritmos variando τ de 1 a 3, e com os tamanhos de prefixo de consulta fixados em 5 e 13, respectivamente, com o objetivo de analisar o resultados para tamanhos de prefixos de consulta mais pequenos e mais longos. Nesse último cenário o segundo nível pode vir a ter um mal desempenho quando precisa buscar por cadeias de caracteres muito longas, principalmente o IP2L que realiza apenas busca sequencial. Quanto maior é o tamanho da cadeia de caracteres buscada no segundo nível, maior é o tempo de execução do cálculo da matriz de *Levenhstein*. Já que λ está fixado em 10, para o tamanho $|p| = 13$ o segundo nível é sempre ativado para os valores de τ experimentados.

	AOL			**USADDR**		
Método	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 1$	$\tau = 2$	$\tau = 3$
IP2L-10	0,15	0,80	3,23	0,70	9,24	124,02
IP2LB-10	0,15	0,81	3,22	0,70	9,31	123,70
IP2LRB-10	0,16	0,86	3,56	0,72	9,26	124,05
ICAN	0,09	1,59	10,97	0,30	8,14	165,98
ICPAN	0,17	1,05	4,22	0,73	10,79	154,05
BEVA	0,25	1,42	3,92	0,72	9,31	51,85
META	0,07	1,46	25,88	0,33	52,31	4572,78

Tabela 14: Tempos de processamento dos algoritmos em dois níveis propostos com $\lambda = 10$ e os outros *baselines*, para as bases de sugestões de consulta AOL e USADDR e $|p| = 5$, com τ variando de 1 a 3.

Na base AOL, podemos observar na Tabela 14 para $\tau = 1$ que os tempos entre os métodos de dois níveis foram muito similares. O método ICAN desempenhou melhor do que todos com exceção do META, que foi o mais rápido. É importante ressaltar que, para $\lambda = 10$ e $|p| = 5$, apenas o primeiro nível é ativado para todos os valores de τ experimentados.

Um claro efeito desse fato é que para $\tau = 2$, por exemplo, podemos notar que os algoritmos de dois níveis propostos obtiveram tempo de processamento menores do que o ICPAN. Isso ocorre porque como a altura máxima da árvore é limitada no valor de λ, há muito menos nós para verificar e ativar do que no método ICPAN, que indexa as sugestões por completo no índice *Trie*. Outro efeito é que os tempos entre os três métodos são bastante similares. Podemos notar esses dois padrões ocorrendo para todos os valores

de τ nas bases AOL e USADDR. Para $\tau = 2$ o método ICAN deixou de ser um dos mais rápidos, para se tornar o mais lento. Os métodos de dois níveis agora atingiram os menores tempos, o que é plausível pois como mencionado anteriormente, o primeiro nível desses métodos possuem um conjunto muito menor de nós para processar quando comparado aos métodos ICAN e ICPAN.

Para $\tau = 3$ os métodos de dois níveis seguem novamente com os menores tempos. Também é possível observar que os métodos ICAN e META sofreram grande aumento de tempo de processamento ao tolerar mais erros na busca. Em uma base pequena como a AOL verificamos que os métodos em dois níveis foram mais rápidos do que o BEVA, atualmente o método estado-da-arte.

Na base USADDR vemos novamente para $\tau = 1$ que os métodos ICAN e META obtiveram os menores tempos. Tais métodos se demonstraram eficientes ao tolerar apenas um erro de digitação. O ICPAN e BEVA, e todos os métodos de dois níveis obtiveram tempos bem próximos. Para $\tau = 2$ o método BEVA obteve um tempo próximo ao dos métodos de dois níveis. Diferentemente da base AOL, em uma base maior como a USADDR identificamos que o método BEVA obteve um desempenho bem próximo aos algoritmos de dois níveis, no entanto, demonstrou-se muito mais eficiente do que todos os outros métodos para $\tau = 3$ com um tempo de 51,85ms, sendo o único a atingir um tempo de processamento menor do que 100ms.

	AOL			**USADDR**		
Método	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 1$	$\tau = 2$	$\tau = 3$
IP2L-10	0,10	0,92	5,55	0,64	8,12	117,05
IP2LB-10	0,07	0,72	3,74	0,31	6,28	97,25
IP2LRB-10	0,10	0,90	5,45	0,41	7,64	114,52
ICAN	0,11	1,90	13,89	0,34	10,20	215,06
ICPAN	0,10	1,02	4,75	0,35	8,07	130,38
BEVA	0,33	1,80	5,49	0,92	12,13	78,85
META	0,20	2,95	38,81	0,80	74,64	6006,38

Tabela 15: Tempos de processamento dos algoritmos em dois níveis propostos com $\lambda = 10$ e os outros *baselines*, para as bases de sugestões de consulta AOL e USADDR e $|p| = 13$, com τ variando de 1 a 3.

Na Tabela 15 para a base AOL podemos observar que quando $\tau = 1$ há uma diferença sutil entre o tempo do IP2LB-10 e os tempos dos outros dois métodos de dois níveis. Com $|p| = 13$ o segundo nível está sendo ativado, então as diferenças de eficiência do segundo nível começam a aparecer. O IP2LB-10 foi o que obteve o menor tempo de processamento. Vale ressaltar no entanto que a acurácia é baixa para $\tau = 1$ e relativamente baixa para $\tau = 2$, como foi demonstrado na seção 5.2. O tempo do ICPAN foi bem similar aos do IP2L-10 e IP2LRB-10. Para $\tau = 2$ pode-se notar que os tempos dos métodos IP2L-10 e IP2LRB-10 foram um pouco menores do que o ICPAN. A diferença entre o tempo

do IP2LB-10 e os outros dois métodos de dois níveis já aparenta ser um pouco mais expressiva. Os algoritmos de dois níveis obtiveram os menores tempos de processamento. Para $\tau = 3$ o bom desempenho do IP2LB-10 pode ser notado com mais clareza pois atingiu um tempo de 3,74ms (menor tempo dentre os algoritmos), sendo aproximadamente 1,5 vezes mais rápido do que o IP2L-10, que obteve um tempo de 5,55ms. O IP2L-10 foi o único método em dois níveis que teve um tempo de processamento maior do que o BEVA para $\tau = 3$. Além disso, os tempos do IP2L-10 e IP2LRB-10 agora foram maiores do que o ICPAN.

Na base USADDR e $\tau = 1$, o método IP2L-10 obteve um tempo quase 2 vezes mais lento do que o ICPAN, no entanto, ainda com um tempo eficiente de menos de 1ms. Além disso, todos os *baselines* também obtiveram tempo menor que 1ms. O IP2LB-10 obteve o menor tempo para $\tau = 1$ e $\tau = 2$, no entanto, possui os problemas de acurácia nesses casos mencionados na seção 5.2. Para $\tau = 2$, o IP2LRB-10 ficou logo atrás do IP2L-10, obtendo o segundo menor tempo. Também podemos verificar que o desempenho de 8,12ms do IP2L-10 se aproximou mais do ICPAN, com 8,07ms, ou seja, uma diferença de apenas 0,05ms. Para $\tau = 3$, em uma base grande como a USADDR a dificuldade do desafio de CATE se torna mais nítida, bem como a eficiência do método BEVA, que foi o único a conseguir um tempo de processamento menor do que 100ms sem ter problemas de acurácia, e ainda com uma boa margem em relação aos outros métodos. No entanto, vale destacar que o método IP2LB-10 também obteve tempo menor do que 100ms, e como $\tau = 3$, sua acurácia é mais passível de se utilizar em um sistema real de CATE.

Outra situação ocorrida é que o IP2L-10 obteve 117,05ms um tempo de processamento menor do que o ICPAN, com 130,38ms. E preciso considerar que o IP2L-10 realiza uma busca tolerante a erros no primeiro nível em uma árvore de altura profundidade máxima igual a 10, e então buscar sequencialmente no segundo nível por p. Além disso, para valores maiores de λ, frequentemente o tamanho das listas em que se realiza a busca sequencial e menor quando o segundo nível é ativado. Uma vez que $|p| = 13$ e $\tau = 3$, o ICPAN precisa realizar a busca tolerante a erros em uma profundidade máxima de $|p| + \tau = 16$ caracteres para responder ao prefixo de consulta p. Diferentemente do que ocorreu na base AOL para $\tau = 3$, o experimento indica que possivelmente em uma base grande como o USADDR e com uma tolerância maior a erros como $\tau = 3$, o processamento de nós com uma maior profundidade na árvore do ICPAN é mais lento do que combinar o processamento desses nós em uma profundidade menor com busca sequencial.

5.4.2 Variando o tamanho do prefixo de consulta

As Tabelas 16 e 17 contêm os tempos de processamento médios para todos os algoritmos com o tamanho do prefixo de consulta variando de $|p| = 3$ até $|p| = 13$ de 2 em 2 com $\tau = 3$ para as bases AOL e USADDR, respectivamente. Consideramos apenas

$\tau = 3$ seguindo a literatura, pois isso nos permite analisar o processamento das consultas dos métodos com um número relevante de erros de digitação. Valores acima de 3 não são comuns em cenários práticos de CATE.

	AOL					
Método	$\|p\| = 3$	$\|p\| = 5$	$\|p\| = 7$	$\|p\| = 9$	$\|p\| = 11$	$\|p\| = 13$
IP2L-10	11,81	3,23	2,77	4,10	5,15	5,55
IP2LB-10	11,86	3,22	2,77	3,43	3,73	3,74
IP2LRB-10	11,85	3,56	2,94	4,55	5,70	5,45
ICAN	14,31	10,97	12,40	12,99	13,93	13,89
ICPAN	11,21	4,22	4,17	4,41	4,70	4,75
BEVA	0,004	3,92	4,90	5,22	5,45	5,49
META	11,27	25,88	31,77	34,76	37,57	38,81

Tabela 16: Tempos de processamento médios dos métodos propostos com $\lambda = 10$ e dos *baselines* para $\tau = 3$, na base de sugestões de consultas AOL, variando o tamanho do prefixo de consulta de 3 a 13, de 2 em 2.

Na Tabela 16 podemos observar que o método BEVA obteve um tempo de processamento muito baixo de 0,004ms para um tamanho pequeno de prefixo de consulta como $|p| = 3$. Esse fato também se repetiu para a base USADDR, como podemos ver na Tabela 17. Isso ocorre porque a implementação do BEVA utilizada neste trabalho implementa uma política de *cache* de nós ativos para reduzir o tempo de processamento para prefixos de consulta muito curtos. Uma característica que os métodos em dois níveis, o ICAN, ICPAN, BEVA e META possuem em comum é que para valores maiores de τ e prefixos muito curtos há uma quantidade massiva de nós da *Trie* para ativar/visitar. Os autores do IncNG*Trie* (Xiao et al., 2013) e IncNG*Trie*+ (Qin et al., 2020) denominam esse problema como "explosão da fase inicial". O restante dos métodos obtiveram tempos de processamento próximos, com exceção do ICAN, que foi o mais lento com 14,31ms. Com $|p| = 5$ a política de *cache* do BEVA já não é mais ativada, e seus tempos entram em uma faixa mais próxima dos outros métodos; os métodos de dois níveis atingiram os menores tempos de processamento, e assim permanece até $|p| = 9$. No entanto, seguem obtendo tempos competitivos com o ICPAN e BEVA para $|p| = 11$ e $|p| = 13$. Também vale destacar que o método IP2LB-10 seguiu com os menores tempos de processamento a partir de $|p| = 5$ até $|p| = 13$.

Na Tabela 16 para $|p| = 3$ vemos novamente o impacto da política de *cache* do BEVA, que respondeu as consultas com uma média de 0,01ms apenas. Os métodos de dois níveis, o META e o ICPAN obtiveram tempo maior do que 700ms para responder às consultas, e com certeza também se beneficiariam de uma política de *cache* como a do BEVA. O META seguiu com os maiores tempos em todos os tamanhos de p. Além disso, nem o ICAN ou o ICPAN conseguiram obter uma média de tempo menor do que 100ms em qualquer valor de $|p|$.

	USADDR					
Método	$\|p\| = 3$	$\|p\| = 5$	$\|p\| = 7$	$\|p\| = 9$	$\|p\| = 11$	$\|p\| = 13$
IP2L-10	777,05	124,02	77,19	95,12	106,94	117,05
IP2LB-10	762,56	123,70	77,57	86,43	90,67	97,25
IP2LRB-10	761,06	124,05	78,20	97,56	105,68	114,52
ICAN	381,29	165,98	171,89	217,98	208,36	215,06
ICPAN	759,18	154,05	110,71	115,35	124,94	130,38
BEVA	0,01	51,85	68,33	75,49	77,23	78,85
META	2400,16	4572,78	5140,76	5656,81	5864,68	6006,38

Tabela 17: Tempos de processamento médios dos métodos propostos com $\lambda = 10$ e dos *baselines* para $\tau = 3$, na base de sugestões de consultas USADDR, variando o tamanho do prefixo de consulta de 3 a 13, de 2 em 2.

Diferentemente da base AOL, a eficiência do método BEVA fica mais clara em uma base maior como a USADDR. O método BEVA obteve os menores tempos para todos os tamanhos de $|p|$ como podemos observar na Tabela 17. Para $|p| = 7$ e $|p| = 9$ os métodos de dois níveis foram os únicos além do BEVA a atingir tempos menores do que $100ms$. O método IP2L-10 por exemplo obteve $77{,}18ms$ para $|p| = 7$, sendo cerca de 1,4 vezes mais rápido do que o ICPAN com $110{,}71ms$. No entanto, a partir de $|p| = 11$ os métodos IP2L-10 e IP2LRB-10 seguem com tempos maiores do que $100ms$. É necessário destacar o desempenho do IP2LB-10, que obteve tempos abaixo de $100ms$ para todos os tamanhos de prefixo de consulta a partir de $|p| = 7$. Além disso, o método IP2L-10 (que não tem problemas com acurácia) demonstrou médias de tempo menores do que o ICPAN para todos os tamanhos de prefixo de consulta a partir de $|p| = 5$, algo que no cenário experimentado pode indicar que a solução em dois níveis se demonstrou mais vantajosa do que o método original utilizado no primeiro nível no quesito tempo de processamento.

5.4.3 Consumo de memória

Quando comparamos o consumo de memória, notamos que os métodos propostos apresentam uma vantagem em comparação aos *baselines*. A Tabela 18 apresenta a quantidade média de memória utilizada pelos métodos propostos e os de base para os 3 valores de τ experimentados (cada célula contém a média aritmética entre as quantidades de memória medidas para $\tau = 1$, $\tau = 2$ e $\tau = 3$), nas bases AOL, USADDR e JUSBRASIL. Para todas as bases podemos observar que os métodos propostos obtiveram o menor consumo de memória.

Na base AOL, os algoritmos de dois níveis economizaram cerca de 82% de memória em relação ao ICAN, 70% em relação ao ICPAN, e 50% em relação ao BEVA. Além disso, vale comentar que o ICAN foi o método com os maiores consumos de memória. Isso ocorre provavelmente porque, devido ao funcionamento do algoritmo ICAN quanto à manutenção dos conjuntos de nós ativos para o processamento da consulta, o tamanho

	AOL	USADDR	JUSBRASIL
Método	**Memória (MB)**	**Memória (MB)**	**Memória (MB)**
IP2L-10	107,11	4070,68	6747,79
IP2LB-10	107,25	4070,73	6747,86
IP2LRB-10	107,19	4070,71	6747,76
ICAN	603,76	10833,27	-
ICPAN	362,59	9807,05	-
BEVA	215,91	5962,14	19232,93
META	262,31	7481,40	-

Tabela 18: Quantidades médias de memória em *MegaBytes* utilizadas pelos *baselines* e métodos IP2L, IP2LB, e IP2LRB durante o processamento das consultas para os 3 valores de τ experimentados, com o parâmetro $\lambda = 10$ para as bases AOL, USADDR, e JUSBRASIL.

desses conjuntos pode ser demasiadamente grande. Isso causa uma adição à quantidade de memória já utilizada para o índice. Para ilustrar essa adição basta comparar o método ICAN com o ICPAN. Os dois possuem índices com o mesmo tamanho, mas diferem bastante quanto ao tamanho dos conjuntos de nós ativos (diferença entre as cardinalidades $|\Phi_p|$ e $|\Psi_p|$, respectivamente). Na base USADDR o método ICPAN utilizou cerca de 9,57GB de memória e o BEVA utilizou 5,82GB, enquanto os métodos de dois níveis utilizaram cerca de 4GB, apresentando uma economia de 58% e 31%, respectivamente.

Na base JUSBRASIL, podemos observar uma economia ainda mais significativa em comparação com o BEVA. Os algoritmos de dois níveis economizaram cerca de quase 65%, pois utilizaram 6,58GB indexando os $\lambda = 10$ primeiros caracteres das sugestões de consulta, enquanto o BEVA utilizou 18,78GB indexando o texto completo de cada item da base. Ainda há espaço para experimentar aumentar o valor de λ de forma que o IP2LB-λ provavelmente atinja tempos menores que $100ms$ para $\tau = 3$ e se torne utilizável para a base JUSBRASIL, enquanto garante uma boa economia de memoria em relação ao método BEVA.

Diante dos resultados dos experimentos concluímos que a abordagem em dois níveis no contexto do problema de CATE provou-se efetiva. Além disso, também verificamos que a utilização da busca binária no segundo nível tanto nos métodos IP2LB quanto IP2LRB pode deixar a desejar quanto à acurácia para $\tau = 1$ e $\tau = 2$, mas pode atingir uma acurácia melhor para $\tau = 3$, o qual é o limiar de tolerância que ocasionou os maiores tempos de processamento nos métodos de CATE nos experimentos. O método IP2LB apresenta uma redução significativa do tempo total de processamento em comparação com o IP2L para $\tau = 3$, ultrapassando também o ICAN e ICPAN em todos os casos nas duas bases experimentadas, com exceção de $|p| = 3$. O IP2L se demonstrou efetivo pois também teve um melhor desempenho do que o ICPAN em quase todos os casos, e assim como o IP2LB e IP2LRB também utilizou menores quantidades de memória.

6 Conclusão

A complementação automática de consultas tolerante a erros (ou "CATE") tem se demonstrado uma funcionalidade importante das aplicações de busca pois auxilia os usuários a encontrar o que procuram. Os métodos mais eficientes para solução do CATE encontrados na literatura utilizam índices de árvore *Trie* em suas soluções, o que pode ocasionar um grande consumo de memória, um recurso limitado e de alto custo. Uma forma de tentar mitigar esse problema é a abordagem de busca em dois níveis, que consiste em dividir o processamento do prefixo de consulta em duas partes. Na primeira parte, realiza-se uma busca tolerante a erros em um índice *Trie* que indexa apenas uma quantidade limitada de λ caracteres das sugestões de consulta. Essa busca funciona como um filtro para que restem apenas candidatos de resposta, que serão buscados sequencialmente no segundo nível para serem ou não sugeridos. No entanto, essa busca sequencial realizada no segundo nível pode ser muito custosa e tornar o processamento mais lento.

Com o objetivo de averiguar se é possível tornar o segundo nível mais eficiente ao combinar busca sequencial com busca binária sem que a acurácia do algoritmo seja prejudicada, propusemos nessa dissertação três métodos de busca em dois níveis que executam o algoritmo ICPAN no primeiro nível. O primeiro é o IP2L, que utiliza apenas busca sequencial no segundo nível e não apresenta imprecisão nos resultados. Esse modelo serve como uma base de comparação com os outros dois quanto ao tempo de processamento e também acurácia; O segundo é o IP2LB, que utiliza uma combinação de busca sequencial com a binária no segundo nível. A ideia inicial era utilizar busca binária no segundo nível irrestritamente em casos que permitissem comparação exata de caracteres, ou seja, somente quando todos os erros já tiverem sido processados no primeiro nível. No entanto, descobrimos em nossos experimentos iniciais que a busca binária utilizada dessa forma no IP2LB deixa de recuperar alguns resultados que deveria, e também traz outros resultados que não devia. Na tentativa de mitigar essa imprecisão propusemos o terceiro método, chamado IP2LRB, que é em suma uma versão do IP2LB porém com um critério mais restrito para utilização da busca binária que visa tornar mais precisa recuperação das sugestões de consulta.

Descobrimos em nossos experimentos que mesmo possuindo valores rápidos de tempo de processamento (principalmente o IP2LB), os dois métodos que utilizam busca binária não são tão precisos em uma base de dados de um sistema real quando toleram $\tau = 1$ e $\tau = 2$ erros de digitação, obtendo valores de *F1-Score* entre 86% e 93%, tendo como base de comparativo os resultados do método BEVA, atual estado-da-arte. No entanto, ao tolerar $\tau = 3$ erros, apresentaram valores de *F1-Score* em torno de 96% com indexação de $\lambda = 10$ caracteres no primeiro nível. Nesse cenário o método IP2LB se mostrou 2 vezes

mais rápido do que o IP2L.

Em outro experimento com outra grande base de sugestões em um cenário similar o método IP2LB apresentou uma redução de 20% do tempo de processamento em relação ao ICPAN, método utilizado no primeiro nível. Esses resultados indicam que, em um cenário onde se tolere um pouco de ruído nas respostas de uma consulta em prol de maior eficiência, utilizar a busca binária no segundo nível para $\tau = 3$ pode ser bastante vantajoso. Além disso, em grande maioria dos casos os resultados de tempo de processamento dos métodos propostos demonstram-se competitivos com o ICPAN, principalmente quando o tamanho do prefixo consultado é menor do que λ. Quanto ao consumo de memória, conseguimos atingir um bom resultado obtendo em todos os três métodos uma economia de 30% a 65% em relação aos outros métodos da literatura. O método IP2LRB obteve tempos maiores do que o IP2LB, e ligeiramente menores do que o IP2L, porém sem uma diferença significativa de acurácia em relação ao IP2LB. Esse é um indício de que, tratando-se de utilizar um método de dois níveis com busca binária, é mais vantajoso utilizar a abordagem do IP2LB em um sistema real. Os experimentos indicam que o contexto e cenário de um sistema de CATE podem influenciar a escolha do valor ideal para λ. Deve-se levar em conta o tamanho médio dos prefixos de consulta que se espera receber no sistema, o número de sugestões na base para se indexar, o tamanho médio dessas sugestões, e a quantidade de memória disponível.

Para trabalhos futuros há várias direções possíveis. Uma delas é utilizar a estratégia de combinação de busca sequencial e binária do IP2LB, porém com método BEVA no primeiro nível, pois é o atual estado-da-arte; Também é possível estudar um método híbrido que utiliza o IP2L para $\tau \leq 2$ e o IP2LB para $\tau = 3$, já que nesse caso o tempo de processamento tende a aumentar bastante no IP2L, e o IP2LB apresenta um menor tempo e uma boa acurácia nesse caso; Outra opção que pode ser ainda mais eficiente do que a busca binária é utilizar um índice reverso formado por um vetor de mapas *hash*, cuja primeira posição é referente ao primeiro caractere de cada item da base, a segunda é referente ao segundo caractere, e assim por diante. Cada elemento do vetor mapeia um caractere a um conjunto compactado de *bits* que armazena os *ids* dos itens da base que contêm o caractere referente à posição em que o mapa se encontra no vetor. Então, de forma progressiva e respeitando as relações de posição dos caracteres, esse índice pode ser consultado a cada caractere processado do texto que se quer buscar no segundo nível para no fim obter o conjunto de itens que contenham o padrão buscado; Há ainda a possibilidade de estudar o impacto de outros algoritmos de cálculo de distância entre cadeias de caractere na busca sequencial do segundo nível, como o de Ukkonen (Ukkonen, 1985) e um "Autômato de Levenhstein" (Schulz and Mihov, 2002), por exemplo.

Referências

Arasu, A., V. Ganti, and R. Kaushik

2006. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, Pp. 918–929. VLDB Endowment. Citado na página 22.

Baeza-Yates, R. A. and B. A. Ribeiro-Neto

1999. *Modern Information Retrieval*. ACM Press / Addison-Wesley. Citado na página 30.

Broder, A., P. Ciccolo, E. Gabrilovich, V. Josifovski, D. Metzler, L. Riedel, and J. Yuan

2009. Online expansion of rare queries for sponsored search. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, P. 511–520, New York, NY, USA. Association for Computing Machinery. Citado na página 17.

Chaudhuri, S., V. Ganti, and R. Kaushik

2006. A primitive operator for similarity joins in data cleaning. In *Data Engineering, 2006. ICDE'06. proceedings of the 22nd International Conference on*, Pp. 5–5. IEEE. Citado na página 22.

Chaudhuri, S. and R. Kaushik

2009. Extending autocompletion to tolerate errors. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, Pp. 707–718. ACM. Citado 3 vezes nas páginas 18, 22 e 30.

Christopher D. Manning, Prabhakar Raghavan, H. S.

2008. *Introduction to Information Retrieval*. ISBN: 0521865719. Citado 2 vezes nas páginas 60 e 61.

da Costa Xavier, D.

2019. Um método em dois níveis para completação automática de sentenças. '. Citado 4 vezes nas páginas 19, 20, 26 e 30.

da Gama Ferreira, V. D. B.

2020. Optimizing beva with two-level indexes. '. Citado 6 vezes nas páginas 19, 20, 26, 30, 58 e 69.

Deng, D., G. Li, H. Wen, H. Jagadish, and J. Feng

2016. Meta: an efficient matching-based method for error-tolerant autocompletion. *Proceedings of the VLDB Endowment*, 9(10):828–839. Citado 6 vezes nas páginas 8, 9, 18, 24, 30 e 57.

Di Santo, G., R. McCreadie, C. Macdonald, and I. Ounis
2015. Comparing approaches for query autocompletion. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, P. 775–778, New York, NY, USA. Association for Computing Machinery. Citado na página 17.

Faro, S. and T. Lecroq
2013. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2). Citado 2 vezes nas páginas 18 e 27.

Fredkin, E.
1960. Trie memory. *Commun. ACM*, 3(9):490–499. Citado na página 19.

Heinz, S., J. Zobel, and H. E. Williams
2002. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)*, 20(2):192–223. Citado na página 25.

Ji, S., G. Li, C. Li, and J. Feng
2009. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th international conference on World wide web*, Pp. 371–380. Citado 13 vezes nas páginas 8, 9, 18, 19, 20, 22, 23, 26, 28, 30, 33, 57 e 66.

Levenshtein, V. I.
1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, Pp. 707–710. Citado 2 vezes nas páginas 27 e 28.

Li, G., S. Ji, C. Li, and J. Feng
2011. Efficient fuzzy full-text type-ahead search. *The VLDB Journal—The International Journal on Very Large Data Bases*, 20(4):617–640. Citado 11 vezes nas páginas 8, 9, 18, 20, 22, 23, 30, 36, 37, 45 e 57.

Manber, U., S. Wu, et al.
1994. Glimpse: A tool to search through entire file systems. In *Usenix Winter*, Pp. 23–32. Citado na página 25.

Navarro, G., E. S. De Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates
2000. Adding compression to block addressing inverted indexes. *Information retrieval*, 3(1):49–77. Citado na página 25.

Qin, J., C. Xiao, S. Hu, J. Zhang, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane
2020. Efficient query autocompletion with edit distance-based error tolerance. *The VLDB Journal*, 29(4):919–943. Citado 2 vezes nas páginas 24 e 82.

Schulz, K. U. and S. Mihov

2002. Fast string correction with levenshtein automata. *International Journal on Document Analysis and Recognition*, 5(1):67–85. Citado na página 86.

Ukkonen, E.

1985. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118. Citado na página 86.

Xiao, C., J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane

2013. Efficient error-tolerant query autocompletion. *Proceedings of the VLDB Endowment*, 6(6):373–384. Citado 8 vezes nas páginas 8, 9, 18, 24, 30, 69, 76 e 82.

Xiao, C., W. Wang, and X. Lin

2008. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the VLDB Endowment*, 1(1):933–944. Citado na página 22.

Zhou, X., J. Qin, C. Xiao, W. Wang, X. Lin, and Y. Ishikawa

2016. Beva: An efficient query processing algorithm for error-tolerant autocompletion. *ACM Transactions on Database Systems (TODS)*, 41(1):5. Citado 6 vezes nas páginas 8, 9, 18, 24, 30 e 57.