Comparação entre Diferentes Implementações de BK-trees para o Problema de Busca por Intervalo

Andre Luciano Rakowski[1], Natan Luiz Paetzhold Berwaldt[12], Mauricio Vielmo Schmaedeck[1], Sergio Luis Sardi Mergen[1]

[1]Universidade Federal de Santa Maria
Santa Maria – RS – Brasil

[2]Programa de Educação Tutorial - Ciência da Computação

{alrakowski, nlberwaldt, mvschmaedeck, mergen}@inf.ufsm.br

Resumo. *No contexto de recuperação de informação, o problema de busca por similaridade para consultas por intervalo é caracterizado pela busca de palavras cuja distância de edição até a consulta seja menor do que um limite estipulado. BK-tree é um método de indexação que traz bons resultados para esse tipo de consulta. Experimentos mostram que, apesar da sua simplicidade, na busca textual sua eficiência é superior à outras abordagens. No entanto, nenhuma informação foi dada a respeito da estrutura da árvore. Neste artigo são analisadas formas diferentes de implementação desta estrutura de indexação. Os experimentos elaborados comparam as variações investigadas em termos de custo em espaço, tempo de indexação e tempo de busca.*

Abstract. *In the context of information retrieval, similarity search for range queries is the problem of finding words whose edit distance to a query are smaller than a determined distance. BK-trees is a pivot based indexing mechanism that achieves good results for this kind of query. Experiments show that, despite its simplicity, the efficiency is superior to other approaches when it comes to text searching. However, no information is provided concerning the underlying structure of the tree. In this paper, we analyze different ways of implementing this indexing structure. Our experiments compare the investigated variations in terms of space cost, indexing time and search time.*

1. Introdução

A busca por similaridade de palavras tem por objetivo encontrar, dentro de um dicionário, palavras que sejam parecidas a alguma palavra chave usada como consulta [Chen et al. 2017]. A área da recuperação de informação pode usar esse tipo de busca em diversos cenários, como por exemplo, sugerindo melhores termos a serem usados em consultas por palavra chave. O problema pode ser reformulado a partir do conceito de distância entre as palavras. Nesse sentido, existe a busca por intervalo, cujo objetivo é procurar por todas as palavras que estejam a uma distância máxima r da palavra de consulta.

Ao longo dos anos, diversas técnicas foram propostas com a finalidade de averiguar a distância entre duas palavras. Uma que ganhou notoriedade é chamada de distância de edição de Levensthein [Navarro 2001]. A distância mede o número de operações de

edição necessárias para transformar uma palavra na outra. A importância dessa técnica é o fato de ela respeitar a inigualdade triangular, o que permite que sejam inseridas em uma estrutura de índice em espaço métrico. Valer-se de uma estrutura de índice agiliza a busca por intervalo, evitando que todo o dicionário precise ser consultado [Zezula et al. 2006].

Uma das técnicas de indexação em espaço métrico é uma árvore chamada BK-tree [Burkhard and Keller 1973]. Sua construção é baseada no conceito de pivôs, e seu algoritmo é razoavelmente simples. Apesar da simplicidade, tem boa capacidade de filtragem, ou seja, consegue podar um número considerável de ramos da árvore que não levam ao resultado desejado.

Recentemente, o trabalho de [Chen et al. 2017] mediu o desempenho dessa estrutura em termos de espaço e tempo de resposta. No entanto, o trabalho não detalha a forma como que a árvore foi implementada. É importante que esse aspecto seja analisado, ainda mais levando em consideração que existem possibilidades de implementação mais dispendiosas em termos de memória.

Dessa forma, este artigo visa criar diferentes implementações da estrutura de dados BK-tree, e usá-las em testes de desempenho para o problema de busca por intervalo. O objetivo principal é verificar quais delas se sobressaem em termos de espaço e tempo de busca e tempo de indexação. Além disso, o artigo analisa se o uso adicional de memória é compensado por um ganho significativo no tempo de resposta.

O artigo está estruturado da seguinte forma: A seção 2 apresenta as principais técnicas de indexação para busca por similaridade baseadas em pivôs. A seção 3 as diferentes formas de implementação consideradas para BK-trees. A seção 4 é reservada aos experimentos. As conclusões são apresentadas na seção 5.

2. Busca em Espaço Métrico baseada em Pivôs

Os algoritmos de busca em espaço métrico são aqueles que calculam a distância d entre objetos a e b (a partir de uma função $d(a, b)$) de modo que quatro propriedades sejam satisfeitas: 1)simetria: $d(a, b)$ = $d(b, a)$; 2) não negatividade: $d(a, b) >= 0$, 3) identidade: $d(a, b) = 0$, se $a = b$; 4) inigualdade triangular: $d(a, b) <= d(a, c) + d(b, c)$ [Zezula et al. 2006].

Diversas funções satisfazem essas propriedades. Para o caso específico em que se mede a distância entre palavras, uma das funções mais conhecidas e usadas é a distância de edição de Levenshtein [Navarro 2001]. Essa distância mede o número de operações de remoção, inserção e substituição de caracteres que transformam uma palavra em outra. Por exemplo, a palavra 'erbd' está a uma distância dois de 'errc', pois uma pode ser transformada na outra com duas operações de substituição.

As distâncias podem ser usadas em mecanismos de filtragem baseada em pivôs [Chen et al. 2015]. Nesse contexto, os pivôs são objetos para os quais são guardadas distâncias até outros objetos indexados. Sabendo-se a distância de um pivô p até um objeto o, e a distância de um objeto de consulta q até p, pode-se determinar, através da inigualdade triangular, se o objeto o está ou não a uma distância máxima de q. Esse conceito é empregado em estruturas de índice em forma de árvore. As principais estruturas são BK-trees [Burkhard and Keller 1973], FQ-tree [Baeza-Yates et al. 1994] e VP-Trees [Yianilos 1993].

Cada nó de uma BK-tree possui uma palavra indexada. Uma aresta entre um nó pai e um nó filho leva à palavras que estejam a uma mesma distância do nó pai. Dado um objeto de busca q e uma distância máxima permitida r, e sabendo-se a distância d do nó atual n até um ramo específico, o algoritmo de busca visita o ramo somente se $d(q,n) - r <= d <= d(q,n) + r$. Ao ser visitado, a palavra armazenada no nó é adicionada ao resultado somente se ela estiver à uma distância máxima r do objeto q .

Por exemplo, a Figura 1 mostra a visão abstraída de uma árvore que contém as palavras 'sbbd', 'sbes', 'sbie', 'erad', 'erbd', 'errc' e 'wei'. Como pode-se ver, o nó raiz leva a ramos que contenham palavras que estejam à distâncias 2, 3 e 4 da palavra 'sbbd'. Supondo que o objeto de consulta seja a palavra 'sbia', e que $r = 1$, os nós marcados na figura são aqueles que satisfizeram essa condição (os demais sequer foram visitados). Dentre os nós que foram efetivamente visitados, apenas 'sbie' foi adicionado à resposta.

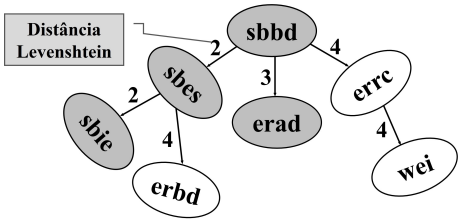


Figura 1. Exemplo de BK-tree utilizada para verificação ortográfica.

A estrutura de dados FQ-tree (Fixed Queries trees) utiliza a mesma abordagem das BK-trees. A diferença reside no fato de o mesmo pivô ser utilizado para todos os nós que estejam no mesmo nível da árvore. Como pode-se ver pelo segundo nível da árvore da Figura 1, as BK-trees não possuem esse requisito, já que três pivôs diferentes são utilizados.

Por fim, a estrutura de dados VP-tree (Vantage Point tree) caracteriza-se por procurar escolher pivôs que dividam o espaço métrico de forma balanceada, de modo que metade dos objetos estejam a uma distância até um pivô menor do que um raio definido, enquanto a outra metade esteja a uma distância maior. Durante a construção da árvore, essa divisão ocorre recursivamente, até que não seja mais possível realizar divisões (ou seja, quando restar apenas um objeto no ramo criado. Uma extensão, chamada MVP-Tree (Multiple Vantage Point tree), permite que cada nó possua múltiplos pivôs, em vez de um só [Bozkaya and Ozsoyoglu 1997].

De todas as abordagens mencionadas, a BK-tree é a que possui a menor complexidade de implementação. Isso se deve ao fato de ela ser a única onde os ramos da árvore são gerados sem preocupação com balanceamento (os ramos são criados conforme a ordem com que as palavras são indexadas). Essa característica pode levar à árvores desbalanceadas, aumentando o número de nós acessados a cada busca. No entanto, essa afirmação se aplica a objetos com poucas dimensões (como distâncias geográficas). Quando os objetos são palavras, muitas dimensões estão envolvidas (os caracteres de cada palavra). Isso faz com que os objetos se encontrem todos igualmente afastados entre si, o que dificulta à obtenção de um bom fator de balanceamento [Chávez and Navarro 2003].

De fato, experimentos recentes mostram que, quando usada para busca de pala-

vras, as BK-trees apresentam um tempo de processamento inferior às demais estruturas [Chen et al. 2017]. Apesar de ser um resultado relevante, nenhuma informação foi fornecida referente à como essa estrutura foi implementada.

3. Proposta

A seção 2 apresentou um exemplo abstrato de uma BK-tree. Já esta seção discute diferentes estratégias para a representação concreta dos nós dessa árvore. Duas estratégias básicas foram trabalhadas: baseada em vetores e baseada em listas encadeadas. Para cada uma delas, duas implementações foram desenvolvidas. A escolha da implementação pode levar a diferenças no desempenho, considerando custo de memória, de indexação, e principalmente, custo de busca. As próximas seções apresentam as vantagens e desvantagens de cada variação.

3.1. Processamento Geral de uma Busca

Antes de apresentar as variações implementadas, é importar mencionar o que está envolvido durante o processo de busca por intervalo em uma BK-tree. Esse processo está descrito em pseudo-código no Algoritmo 3.1. A função recebe quatro parâmetros: o nó pai sendo processado no momento, a palavra usada como consulta, a distância máxima permitida e a lista de palavras a serem retornadas.

O algoritmo é dividido em duas partes. Na primeira parte ocorre a aplicação da função de distância, responsável por calcular a distância de edição entre a palavra do nó e a palavra da consulta. Caso a distância encontrada d satisfaça o critério de busca (for menor ou igual a $dist$), a palavra é adicionada nos resultados. A segunda parte se encarrega de visitar recursivamente todos os filhos do nó pai que estejam à uma distância mínima de $d - dist$ e uma distância máxima de $d + dist$. Devido à propriedade da inigualdade triangular, esses são os únicos ramos que podem conter respostas para a consulta. Uma checagem é necessária para certificar-se de que a distância mínima não seja negativa.

```
BUSCA_INTERVALO(no_pai, consulta, dist, palavras_retornadas)
 1: if no_pai.palavra == ∅ then
 2:     return
 3: end if
 4: d ← distancia(no_pai.palavra, consulta)
 5: if d <= dist then
 6:     adicionar no_pai.palavra em palavras_retornadas
 7: end if
 8: min_dist ← d - dist
 9: if min_dist < 0 then
10:     min_dist = 0
11: end if
12: max_dist ← distancia + dist
13: busca_filhos(no_pai, consulta, min_dist, max_dist, palavras_retornadas)
```

Algorithm 3.1: ESTRUTURA GERAL DO ALGORITMO DE BUSCA POR INTERVALO.

A forma com que os filhos são acessados depende da forma como a árvore foi implementada. As próximas seções apresentam diferentes possibilidades de estruturação dos nós, salientando a forma como o acesso é realizado.

3.2. Estratégia Baseada em Vetores

Na estratégia baseada em vetores, cada nó possui um vetor que armazena os seus filhos. A distância de edição entre um pai e um filho é representada pela posição correspondente

nesse vetor. Por exemplo, um nó na posição dois indica que esse nó está a uma distância de edição igual a dois em relação ao seu pai.

As variações dessa abordagem devem-se ao uso de alocação estática ou dinâmica. Em ambos os casos, a principal vantagem é o acesso direto. Ou seja, todos os nós que estejam dentro intervalo de busca podem ser acessados pelos seus índices. Já a desvantagem se refere ao excessivo consumo de memória. As próximas seções discorrem a respeito dessas questões.

3.2.1. Uso de Alocação Estática

Com vetores estáticos, o tamanho do vetor em cada nó é predefinido. Para garantir que a capacidade do vetor seja suficiente, o tamanho deve suportar a maior distância de edição possível entre uma palavra de busca q e uma palavra indexada p. Considerando $D = \{p_i, ..., p_n\}$ como sendo o dicionário de palavras indexadas, e $tam(p)$ como a função que retorna o número de carateres de p, a maior distância possível pode ser definida como $H \leftarrow \arg\max_{p \in A} tam(p)$. Ou seja, ela corresponde ao tamanho da maior palavra indexada.

A Figura 2 (a) ilustra o uso de vetores estáticos para uma configuração qualquer de nós. Observe que todos os vetores possuem o mesmo tamanho (cinco). Nesse exemplo hipotético, a maior palavra que pode ser indexada possui quatro caracteres. Ou seja, o vetor tem espaço para distâncias que variem de zero ao máximo possível(quatro).

O Algoritmo 3.2 descreve como a busca nos filhos ocorre quando os vetores são alocados estaticamente. O processo é baseado em um laço, onde em cada iteração é acessado um dos filhos do intervalo permitido.

```
BUSCA_FILHOS(no_pai, consulta, min_dist, max_dist, palavras_retornadas)
1: while min_dist <= max_dist do
2:     busca_Intervalo(no_pai.vetor_filhos[min_dist], consulta, palavras_retornadas)
3:     min_dist += 1
4: end while
```

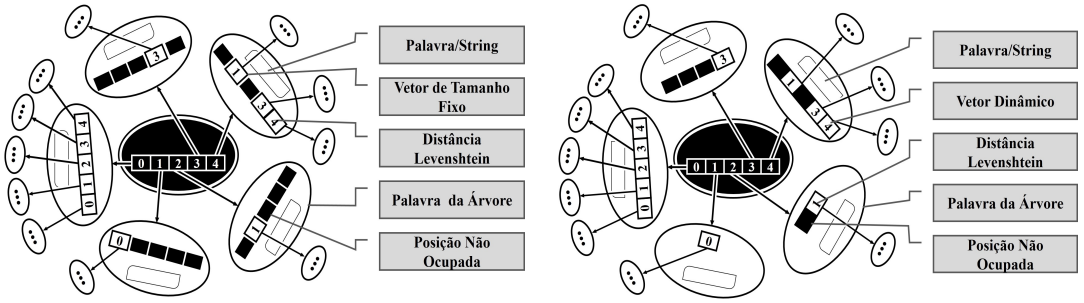
Algorithm 3.2: BUSCAS DOS FILHOS DE UMA BK-TREE USANDO VETORES ESTÁTICOS.

Um aspecto negativo dessa estratégia refere-se ao problema da sub-utilização do espaço. Mesmo que a distância máxima usada por um nó pai seja inferior a H, todas as posições são alocadas. O desperdício de espaço é relativo ao grau médio de um nó. Quanto menor o grau, maior o desperdício. Além disso, é necessário descobrir o maior tamanho de palavra antes da construção do índice. Atualizações são permitidas, desde que as novas palavras não ultrapassem o tamanho definido.

3.2.2. Uso de Alocação Dinâmica

O uso de alocação dinâmica implica que o tamanho de cada vetor seja definido em tempo de execução, podendo ser realocados quando surgir a necessidade. Ao contrário da versão anterior, os vetores possuem tamanhos distintos, que coincidem com a maior distância necessária.

A Figura 2 (b) ilustra o uso de vetores dinâmicos para a mesma configuração de nós usada na Figura 2 (a). Observe que vetor dinâmicos acarretam em um menor consumo de memória, em comparação com os vetores estáticos. A diferença entre as duas abordagens reside na parte final dos vetores. Com alocação dinâmica, os espaços que sucedem a última posição preenchida não são sequer criados.



(a) Vetores estáticos. (b) Vetores dinâmicos.

Figura 2. BK-tree implementada com vetores.

A busca nos filhos pode ser realizada pelo mesmo algoritmo empregado com vetores dinâmicos, desde que o valor máximo não ultrapasse o tamanho do vetor. Esse limite superior pode ser garantido pela inclusão de um comando condicional no código que altera max_dist caso haja necessidade.

Um aspecto negativo dos vetores dinâmicos é o maior custo para criação do índice, pois surge a sobrecarga relativa às realocações. Por outro lado, o custo é amortizado pela redução de espaço necessário, o que simplifica o gerenciamento de memória por parte do sistema operacional. Apesar da redução de espaço, ainda existe um desperdício associado às posições intermediárias do vetor não preenchidas. Não é possível abrir mão desses espaços intermediários não usados sem perder o acesso direto, razão principal para o uso de vetores.

3.3. Estratégia Baseada em Listas

Na estratégia baseada em listas, cada nó tem acesso ao primeiro filho. Além disso, nós são encadeados para que os irmãos sejam acessíveis. Um nó pai consegue acessar todos os filhos passando pelo primeiro filho e seguindo o encadeamento dos irmãos. Cada nó conta também com um valor numérico que simboliza a sua distância para o respectivo pai.

As variações dessa abordagem devem-se ao uso de encadeamento ordenado ou não. Em ambos os casos, a principal vantagem no uso de listas encadeadas é o menor consumo de memória. Já a desvantagem se refere a necessidade de caminhamento na lista para localizar um nó específico. As próximas seções discorrem a respeito dessas questões.

3.3.1. Uso de Listas Desordenadas

Com listas desordenadas, um nó filho é inserido sempre no começo da lista, independente do seu valor de distância. A Figura 3 (a) ilustra o uso de listas desordenadas. Nesse caso, o nó raiz possui filhos distantes dele por uma, três e quatro posições. Por sua vez, o filho cuja distância é um também possui uma descendência composta por quatro filhos. Como não se pode determinar a ordem com que os nós são inseridos na árvore, é natural que os valores de distância dos nós irmãos para o nó pai estejam dispostos aleatoriamente.

O Algoritmo 3.3 descreve como a busca ocorre usando listas desordenadas. A lista é percorrida do início ao fim para que os filhos sejam acessados. A função de busca por intervalo é chamada para todo nó acessado cuja distância satisfaça o critério. Como não existe ordenação, a pesquisa necessariamente visita todos nós da lista.

```
BUSCA_FILHOS(no_pai, consulta, min_dist, max_dist, palavras_retornadas)
1: no_filho ← no_pai.prim_filho
2: while no_filho ≠ ∅ do
3:     if no_filho.d ∈ [min_dist, max_dist] then
4:         busca_Intervalo(no_filho, consulta, palavras_retornadas)
5:     end if
6:     no_filho ← no_filho.irmao
7: end while
```

Algorithm 3.3: BUSCAS DOS FILHOS DE UMA BK-TREE USANDO LISTAS DESORDENADAS.

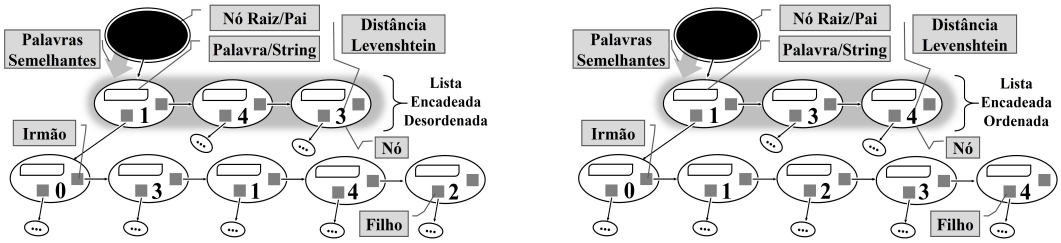
Além do baixo consumo de memória, destaca-se como ponto positivo a eficiência na atualização. Como o nó a ser inserido substituirá o primeiro nó filho, a atualização da lista ocorre em tempo constante, sendo necessário apenas ajuste dos nós envolvidos. Outro ponto que merece destaque é o baixo consumo de memória. O aspecto negativo refere-se ao custo do percorrimento da lista. Como não existe ordenação, a pesquisa necessariamente visita todos nós de forma exaustiva. Quanto maior for o grau médio do nó, mais custosa se torna a busca.

3.3.2. Uso de Listas Ordenadas

Com listas ordenadas, um nó filho é inserido na lista na posição que mantenha os nós ordenados pela distância. A Figura 3 (b) ilustra o uso de listas ordenadas, usando os mesmos valores apresentados na Figura 3 (a). Independente da ordem com que os nós sejam inseridos na árvore, existe a garantia de que eles permanecem ordenados.

O algoritmo 3.4 descreve como a busca ocorre usando listas ordenadas. Assim como na versão anterior, a lista é percorrida para que os filhos sejam acessados, e a função de busca por intervalo é chamada para todo nó acessado cuja distância satisfaça o critério. No entanto, como a lista está ordenada, a busca pode ser encerrada quando se visitar algum nó que tenha ultrapassado a distância máxima permitida.

Uma vantagem desta versão é a redução no custo do percorrimento da lista. Enquanto na versão desordenada todos os n filhos precisam ser acessados, a ordenação reduz o número de acessos para $\frac{n}{2}$, na média. Em contrapartida, a atualização é mais cara. Em média $\frac{n}{2}$ nós precisam ser acessados até que o lugar correto seja encontrado, enquanto que na lista desordenada nenhum acesso extra é necessário.



(a) Listas encadeadas não ordenadas.

(b) Listas encadeadas ordenadas.

Figura 3. BK-tree implementada com listas encadeadas.

```
BUSCA_FILHOS(no_pai, consulta, min_dist, max_dist, palavras_retornadas)
1: no_filho ← no_pai.prim_filho
2: while no_filho <> ∅ do
3:     if no_filho.d > [max_dist] then
4:         return
5:     end if
6:     if no_filho.d >= [min_dist] then
7:         busca_Intervalo(no_filho, consulta, palavras_retornadas)
8:     end if
9:     no_filho ← no_filho.irmao
10: end while
```

Algorithm 3.4: BUSCAS DOS FILHOS DE UMA BK-TREE USANDO LISTAS ENCADEADAS ORDENADAS.

4. Experimentos

O objetivo desta seção é avaliar as quatro variações de BK-trees apresentadas no decorrer do artigo. Os critérios de avaliação incluem o tempo de indexação, espaço ocupado e tempo de busca. Os tempos relatados correspondem a média de 30 execuções, descartando os 10% piores e 10% melhores resultados.

Por permitir um gerenciamento de memória mais ajustado, os algoritmos foram implementados em C. As execuções foram realizadas em uma máquina Core i5, com 6 GigaBytes de memória utilizando o Sistema Operacional aberto Lubuntu de 64 bits. O dicionário usado contém 994.703 palavras da língua portuguesa[1]. A maior palavra tem 29 caracteres, e na média cada palavra tem 12 caracteres.

4.1. Tempo de Indexação

Não houve diferença significativa no tempo de indexação para as quatro variações. Em qualquer um dos casos, o tempo para construir as árvores ficou em torno de 28 segundos. O resultado demonstra que o cálculo da distância de edição, necessário para fazer o roteamento da palavra a ser inserida, tem um custo predominante.

O tempo necessário para localização dos nós (nas listas ordenadas) tem pouca importância devido ao baixo grau de saída de cada nó (média de 1 filho por nó). Como as listas tem poucos elementos, o custo para localização acaba sendo baixo.

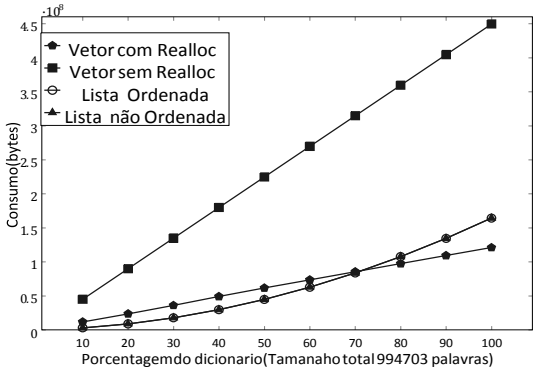
O custo da realocação (nos vetores dinâmicos) também se mostrou insignificante. No pior cenário, o sistema operacional precisaria encontrar espaço contíguo para apenas

[1]Disponível em http://natura.di.uminho.pt/download/sources/Dictionaries/wordlists/wordlist-ao-20170814.txt.xz

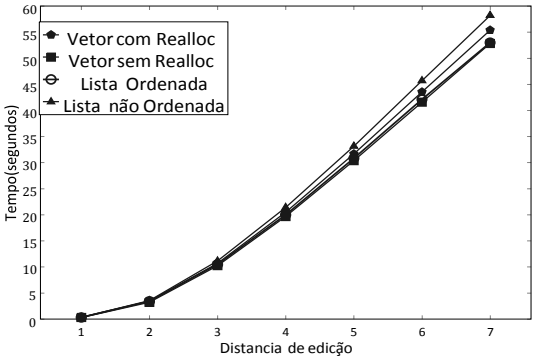
29 ponteiros (que é a maior distância de edição possível), o que poderia ser feito sem grande esforço.

4.2. Espaço Ocupado

A Figura 4 (a) apresenta o espaço ocupado na memória em bytes que cada estrutura de árvore precisou para a indexação das 994.703 palavras. O gráfico mostra como o consumo é afetado quando partes maiores do dicionário são consideradas.



(a) Custo de Memória Variando o Tamanho do Dicionário.



(b) Tempo de Busca Variando a Distância Máxima Permitida.

Figura 4. Resultados.

Como era esperado, o uso de vetores estáticos leva ao pior resultado. A diferença que existe para as demais abordagens é um efeito direto da diferença entre o tamanho da maior palavra e o tamanho médio. Na verdade, as outras três estratégias alocam menos espaços do que esse tamanho médio de palavra. Por exemplo, os vetores dinâmicos alocam espaço suficiente para uma distância média igual a 4. Como a indexação aproxima palavras parecidas, a distância de um filho para o pai acaba sendo na maioria dos casos menor do que o tamanho médio de palavra.

Também é interessante notar que os vetores dinâmicos passam a ocupar menos memória que os concorrentes conforme o dicionário cresce. O motivo está relacionado aos buracos dos vetores dinâmicos, criados para os espaços intermediários não preenchidos. Para dicionários suficientemente grandes, menos espaços vazios são deixados, o que leva a um melhor aproveitamento da memória.

4.3. Tempo de Busca

A Figura 4 (b) apresenta o tempo em segundos necessário para responder 25 consultas, com distâncias máximas de edição variando de um a sete. As 25 consultas são palavras escolhidas aleatoriamente do dicionário.

Os resultados mostram que há pouco diferença entre os algortimos, especialmente quando se usa distâncias pequenas. O motivo mais uma vez está atrelado ao peso que o cálcullo da distância de edição tem frente às demais operações necessárias. Apesar da diferença irrisória, os vetores estáticos e as listas ordenadas apresentaram os melhores resultados em todos os casos. O ganho se torna mais evidente conforme a distância

aumenta. De qualquer forma, a busca por intervalo geralmente encontra resultados satisfatórios usando distâncias menores do que 3. Por exemplo, com distância de edição igual a um já foi possível trazer uma média de 4 resultados para as 25 consultas usadas no experimento.

5. Considerações Finais

Este artigo mostrou as diferenças estruturais entre quatro variações do algoritmo de busca por similaridade BK-tree. Através de exemplos, mostrou-se as vantagens e desvantagens de cada abordagem, levando em consideração tempo de indexação, espaço ocupado em memória e tempo de busca.

Os experimentos realizados mostraram que o desempenho na carga e na busca(para distâncias pequenas) é muito parecido. Sendo assim, a escolha por uma abordagem pode levar em consideração o consumo de memória. Nesse quesito, vetores dinâmicos mostraram-se superiores. Os resultados também serviram para constatar que o principal custo tanto na busca quanto na indexação é o cálculo da distância de edição. O algoritmo baseado em programação dinâmica que encontra a distância para as palavras m e n tem complexidade em tempo proporcional a $O(m.n)$. Uma tema interessante para trabalhos futuros é a definição de outro tipo de distância de edição, tão relevante quanto à distância de Levensthein, mas cuja complexidade em tempo seja menor.

Referências

Baeza-Yates, R., Cunto, W., Manber, U., and Wu, S. (1994). Proximity matching using fixed-queries trees. In *Annual Symposium on Combinatorial Pattern Matching*, pages 198–212. Springer.

Bozkaya, T. and Ozsoyoglu, M. (1997). Distance-based indexing for high-dimensional metric spaces. In *ACM SIGMOD Record*, volume 26, pages 357–368. ACM.

Burkhard, W. A. and Keller, R. M. (1973). Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236.

Chávez, E. and Navarro, G. (2003). Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters*, 85(1):39–46.

Chen, L., Gao, Y., Li, X., Jensen, C. S., and Chen, G. (2015). Efficient metric indexing for similarity search. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 591–602. IEEE.

Chen, L., Gao, Y., Zheng, B., Jensen, C. S., Yang, H., and Yang, K. (2017). Pivot-based metric indexing. *Proceedings of the VLDB Endowment*, 10(10):1058–1069.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88.

Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–321.

Zezula, P., Amato, G., Dohnal, V., and Batko, M. (2006). *Similarity search: the metric space approach*, volume 32. Springer Science & Business Media.