Mateus Coutinho Marim

Relatório
Indexação e busca de documentos

Trabalho parte da disciplina de Algoritmos e Estruturas de Dados da turma do mestrado em Ciência da Computação de 2020 do PPGCC.

Universidade Federal de Juiz de Fora – UFJF

Departamento de Ciência da Computação – DCC

Programa de Pós-Graduação em Ciência da Computação – PPGCC

Juiz de Fora - MG

9 de setembro de 2020

Sumário

1 Organização do projeto

O projeto para a solução do problema de indexação e busca de documentos foi feito utilizando a linguagem C++, por haver muitas abstrações como a orientação a objetos sem muito peso para o tempo de execução dos programas, e o CMake que permite melhor organização do projeto em pastas e também gerenciamento das dependências para a compilação do código.

1.1 Requerimentos

- Compilador para o C++, recomendada a utilização das versões mais recentes do GCC/g++;
- Gnuplot[1] para plotar os dados dos experimentos em tempo de execução;
- Bibliotecas Boost[2], utilizadas pelo envólucro para o gnuplot.

1.2 Organização das pastas

Os arquivos do projeto foram organizadas em várias pastas para prover uma melhor navegação para quem for utilizá-lo. Abaixo está uma descrição objetiva da estrutura de pastas.

- *CLI*: pasta contendo o arquivo principal com o código da aplicação para linha de comando;
- *data*: contém os *datasets* para serem processados;
- *Dictionary*: códigos com a implementação das estratégias de solução do problema de indexação e busca de documentos;
- *Experiments*: arquivos de código necessários para a execução dos experimentos;
- *Plots*: gráficos dos experimentos executados;
- *Utils*: códigos de funções utilitárias utilizadas em várias partes do projeto.

[1] Instalação no Ubuntu: *sudo apt install gnuplot*
MacOS: *brew install gnuplot*
[2] Instalação no Ubuntu: *sudo apt install libboost1.71-all-dev*
MacOS: Instalar o MacPorts e executar o comando *sudo port install boost*

1.3 Utilização

O principal motivo da escolha do CMake para a organização do projeto é que ele permite a compilação e execução do programa em diversas plataformas, a forma mais simples de compilar o projeto é a execução dos seguintes comandos a partir da pasta principal:

```
mkdir build
cd build
cmake ..
make
```

Após a fase de compilação do projeto, ainda dentro da pasta *build*, basta executar os executáveis dentro das pastas *CLI* e *Experiments* para utilizar a aplicação de linha de comando ou rodar os experimentos respectivamente.

Abaixo estão os *scripts* que automatizam esse processo em sistemas baseados no linux:

- `build.sh`: compila todo o projeto e copia os executáveis para a pasta principal;
- `buildRunExperiments.sh`: compila o projeto e executa os experimentos, salvando os gráficos gerados na pasta *Plot*.

Para executar a aplicação de linha de comando (CLI), é recomendado executar os seguintes comandos na pasta principal do projeto, onde `<strategy>` tem que ser substituído pelas strings `hash` ou `trie`, dependendo da estratégia que se queira utilizar:

```
./build.sh
./news_index <strategy>
```

De forma análoga, também é possível executar os experimentos separadamente:

```
./build.sh
./exp <strategy>
```

Para executar os experimentos e salvar os gráficos na pasta *Plot*, basta rodar o comando `./buildRunExperiments.sh`.

2 Visão geral

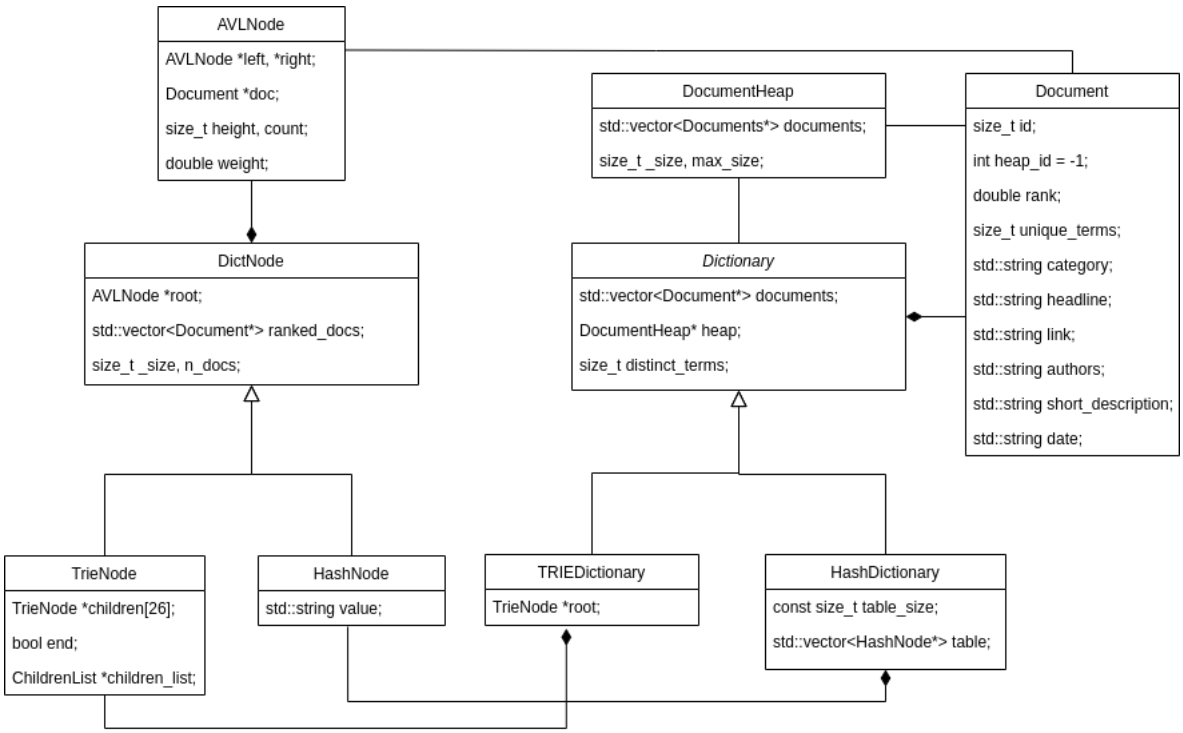

Figura 1 – Diagrama de classes do sistema, as funções membro das classes foram omitidas por simplicidade.

A Figura 1 mostra o diagrama de classes da solução do problema, foram omitidas as funções membros das mesmas de forma a deixar o diagrama mais limpo, mas elas serão citadas nas próximas seções quando necessárias.

2.1 Classe base `Dictionary`

A classe abstrata `Dictionary` contém os métodos e propriedades comuns às estratégias implementadas, dessa forma as classes derivadas que implementam as estratégias podem reaproveitar o código da mesma e qualquer alteração feita nesse código aproveitado vai se refletir nas classes derivadas, ou seja, qualquer otimização que for feita também vai ser aproveitada. Outra função da classe *Dictionary* é de forçar que as estratégias sigam um padrão. A complexidade de cada uma das funções depende da estratégia implementada pelas classes derivadas.

Funções virtuais sem implementação (estratégias devem implementar):

- `computeTermsParameters()`: computa os pesos dos termos em cada documento;

- `find(word)`: retorna o nó referente ao termo pesquisado;
- `insert(word, document)`: insere o termo e o documento dados na estrutura de dados.

Funções virtuais com implementação (não precisam ser sobrescritas):

- `findByTerms(words)`: retorna os documentos mais relevantes dados os termos de pesquisa;
- `insert(path)`: insere todos documentos em um *dataset*;
- `insert(document)`: insere um documento na estrutura.

2.2 Classe base `DictNode`

A classe `DictNode` representa os termos incluídos no dicionário e o seu índice invertido correspondente. Além das informações do termo, também é implementada uma árvore AVL para armazenar os documentos em que o termo está incluído, assim é possível inserir um novo documento ou incrementar o seu contador em $O(log(\lambda))$, onde λ é o número de documentos em que o termo se encontra. Outra vantagem de se utilizar uma AVL é que por ela ser uma ABB é possível retornar os documentos ordenados pelo seu índice.

2.3 Computação do *inverse document frequency* (IDF)

Após a inserção de todos documentos no dicionário é feita a pré-computação dos pesos de cada termo nos documentos, para isso a ABB implementada no *DictNode* é percorrida chamando a função `computeWeights()` para computar o peso de cada um dos termos em cada um dos documentos, apesar de a pré-computação aumentar o custo da inserção dos documentos na estrutura de dados ela diminui o custo de pesquisa dos documentos mais relevantes dados os termos para a pesquisa.

2.4 Estratégia utilizando tabela hash

Para a primeira escolha de estratégia foi escolhido fazer a implementação de uma tabela hash de endereçamento aberto por permitir que sejam feitas otimizações de memória de acordo com o número de termos únicos sendo inseridos. Foram consideradas como chaves da tabela os termos sendo inseridos na mesma, dessa forma, quando há a tentativa de inserção de um termo já existente na tabela o documento em que o termo aparece é inserido ou tem o seu contador de frequência do termo atualizado pela árvore AVL

implementada pela classe `DictNode`. Nos casos das colisões com elementos que tem as mesmas chaves geradas pela função de hash foi implementado o tratamento de colisões por tentativa linear. No Capítulo 3 serão mostrados o porque da escolha da tentativa linear e também outras escolhas feitas com base no *dataset* escolhido para testes.

2.4.1 Função hash escolhida

Para a função hash foi escolhido o algoritmo `djb2` desenvolvido por Dan Berstein. O mesmo não só provê uma boa distribuição, mas também uma velocidade alta em diferentes conjuntos de chaves e tamanhos de tabelas[1]. O porque dessa função funcionar tão bem nunca foi explicado de forma adequada[2].

2.4.2 Complexidade computacional

Como sabemos, para encontrar a posição em que o elemento na tabela, tanto o algoritmo de pesquisa quanto o de inserção tem pior caso de $O(\alpha)$, ou seja, tem tempo constante. Como na inserção também temos que os documentos são inseridos no índice invertido que é implementado como uma árvore AVL, para N termos distintos inseridos na tabela e λ documentos o pior caso da inserção será de $O(N * log(\lambda))$.

2.5 Estratégia com Trie R-*way*

A árvode Trie foi escolhida por permitir, assim como na tabela hash, tempo constante na inserção dos termos de pesquisa, sendo que o pior caso tanto da inserção e da pesquisa é o tamanho h do maior termo inserido, ou seja, $O(1)$.

2.5.1 Otimizações

Como os termos inseridos são compostos de caracteres alfa-numéricos é necessário que hajam duas listas de filhos em cada nó, os caracteres numéricos geralmente não são muito frequentes nos termos inseridos, dessa forma, para otimizar o consumo de memória, foi utilizado um vetor contíguo para armazenar os caracteres do alfabeto e uma lista encadeada para alocação de memória para os nós numéricos, assim, por os caracteres numéricos não serem muito frequentes, o consumo de memória vai ser menor do que a utilização de um vetor contíguo e o tempo de execução da pesquisa não vai ser afetado de forma significativa.

[1] S. Shah and A. Shaikh, "Hash based optimization for faster access to inverted index,"2016 International Conference on Inventive Computation Technologies (ICICT), Coimbatore, 2016, pp. 1-5, doi: 10.1109/INVENTIVE.2016.7823270.

[2] Ram, Napa, et al. "Application of Data Structure in the field of Cryptography."Proceedings of the International Conference, Computational Systems for Health& Sustainability, RV College of Engineering. 2015.

Outra otimização no consumo de memória foi na alocação dos vetores contíguos dos filhos alfabéticos dos nós da Trie. Algo comum na implementação de Tries é a alocação do vetor de filhos toda vez que um nó é criado, levando a um consumo excessivo de memória ao se levar em consideração que os nós folhas não apontam para nenhum filho, assim, a implementação feita só aloca o vetor de filhos se o nó sendo inserido for interno, como as folhas formam uma grande quantidade de nós, o ganho no consumo de memória é bem considerável.

2.5.2 Complexidade computacional

A implementação da Trie não difere muito da tradicional, apesar da lista de caracteres numéricos, por serem pouco frequentes podemos simplificar a análise e considerar apenas o vetor de caracteres alfabéticos. Assim, temos que a complexidade vai ser o custo h para encontrar a posição para inserir o novo nó mais o custo de se inserir o documento ou atualizar o contador do termo no índice invertido, considerando que o termo apareça em λ documentos e que tenham N termos únicos, temos que a complexidade de inserção é de $O(N * h * log(\lambda))$, como h é constante por ser o tamanho do termo, a complexidade no pior caso se torna $O(N * log(\lambda))$.

2.6 Pesquisa no `Dictionary`

A pesquisa no dicionário de documentos é dependente da pesquisa da estratégia implementada, para cada termo de pesquisa o algoritmos procura o mesmo na estrutura de dados da estratégia e atualiza o rank do documento com base nos pesos calculados na construção da estrutura. Como forma de otimização da pesquisa, os documentos que vão aparecer no resultado são armazenados em uma *heap* para que não seja necessário posteriormente ordenar os documentos e também só são adicionados na mesma os documentos com o rank diferente de zero. Como foi mostrado nas Sessões 2.5 e 2.4, para N termos únicos no dicionário e λ documentos, a complexidade de pesquisa nas estruturas é de $O(N * log(\lambda))$, logo, para ρ termos temos que a complexidade da pesquisa é na ordem de $O(\rho * N * log(\lambda))$.

3 Experimentos

Com o objetivo de testar a eficiência das soluções propostas e confirmar o resultado da complexidade computacional foram executados experimentos para medir o tempo de execução, o consumo de memória das soluções e o número médio de comparações feitos para encontrar o nó correspondente aos termos pesquisados. O *dataset* de notícias coletado da *HuffPost* chamado de *News Category Dataset*[1] com mais de 200 mil *headlines* e descrições de notícias. Por ser possível tirar informações a priori do *dataset*, o tamanho da tabela hash foi definido como o próximo número primo a partir de $1.3 * N$ sendo N igual ao número de termos únicos em todos documentos, dado pelo valor de 90561 termos únicos. Os experimentos foram executados em um computador com o processador Intel Core I5-7200U, 8GB DDR4 de memória RAM e sistema operacional Ubuntu 20.04 64-bit.

Também podemos estimar para a tabela hash o número de comparações médio esperado baseando-se no fator de carga α que dá a fórmula $0.5 + 0.5/(1 - \alpha)$. Dado que ao inserir todos termos na tabela hash o fator de carga $\alpha = 0.79$, podemos esperar que o número médio de comparações seja de aproximadamente 2.88. Foi a partir desse resultado que foi tomada a decisão de implementar a tentativa linear para tratamento das colisões, já que é o suficiente para obter um bom desempenho.

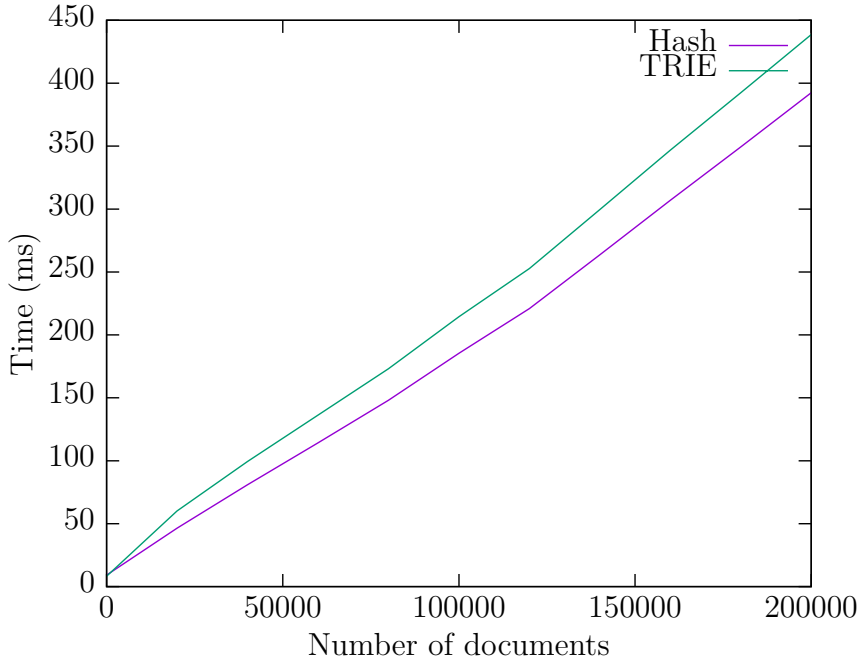

Figura 2 – Consumo de memória médio a cada 20000 documentos inseridos

[1] https://www.kaggle.com/rmisra/news-category-dataset/data

3.1 Consumo de memória

Para o experimento de consumo de memória, foi medido o consumo de memória RAM a cada 20000 documentos inseridos na estrutura de dados, onde cada documento representa uma notícia. Na Figura 2 podemos ver que a tabela hash teve um consumo de memória aproximadamente 12.5% menor que a solução utilizando a Trie. No total, após a construção da tabela hash e pré-computação do peso de todos termos, o programa consumiu 394MB e com a Trie 440MB de RAM, nesse caso, o consumo de memória da Trie foi apenas 11.67% maior.

3.2 Tempo de inserção

Nesta sessão iremos analisar os resultados dos experimentos de tempo de execução para pesquisa e inserção nas estruturas de dados usadas nas estratégias implementadas. Para gerar os resultados dos experimentos na inserção, foram medidos as médias dos tempos a cada 20 mil documentos inseridos. Podemos ver na Figura 3 que a média do tempo de inserção na tabela hash foi menor em quase todos os casos, e que é confirmada a análise da complexidade complexidade computacional pois as curvas geradas indicam um comportamento na ordem de $O(N * log(\lambda))$. A média do tempo de inserção de todos documentos na tabela hash e na trie foram iguais a $0.19ms$ e $0.21ms$ respectivamente.

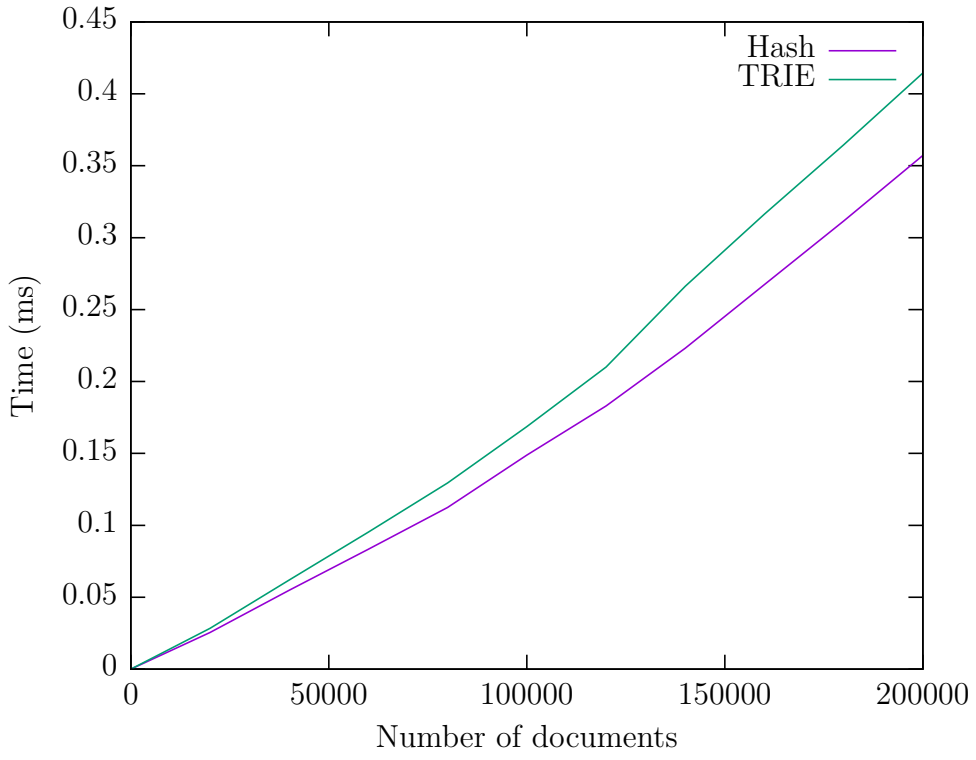

Figura 3 – Tempo de inserção médio a cada 20000 documentos inseridos na tabela hash

3.3 Tempo de pesquisa

Os experimentos para analisar o tempo de pesquisa foram realizadas pela geração de 10 mil pesquisas onde a primeira metade delas tem 1 termo de pesquisa e a outra tem 2 termos. Primeiro vamos analisar os gráficos das pesquisas com 1 termo. O tempo total para execução do experimento com 1 termo foi de aproximadamente 35s em ambas estratégias e de 103s com as pesquisas com 2 termos.

Podemos observar que novamente na Figura 4 que ambas estratégias tiveram uma eficiência muito parecida no tempo de pesquisa. Em média a Trie e a hash precisaram respectivamente de 3.53ms e 3.34ms. Não representando uma diferença muito significativa de desempenho. Também foi medido o número médio de comparações para se encontrar os documentos relacionados ao termo, para a estratégia hash foram em média 2.11 comparações para retornar o nó correspondente ao termo pesquisado, valor próximo ao resultado teórico, enquanto que a Trie precisou fazer 5.39 comparações. Algo que foi interessante de observar é que o número de comparações médio foi praticamente igual a média do tamanho de todas palavras contidas na árvore, que no *dataset* utilizado foi de 5.4 caracteres.

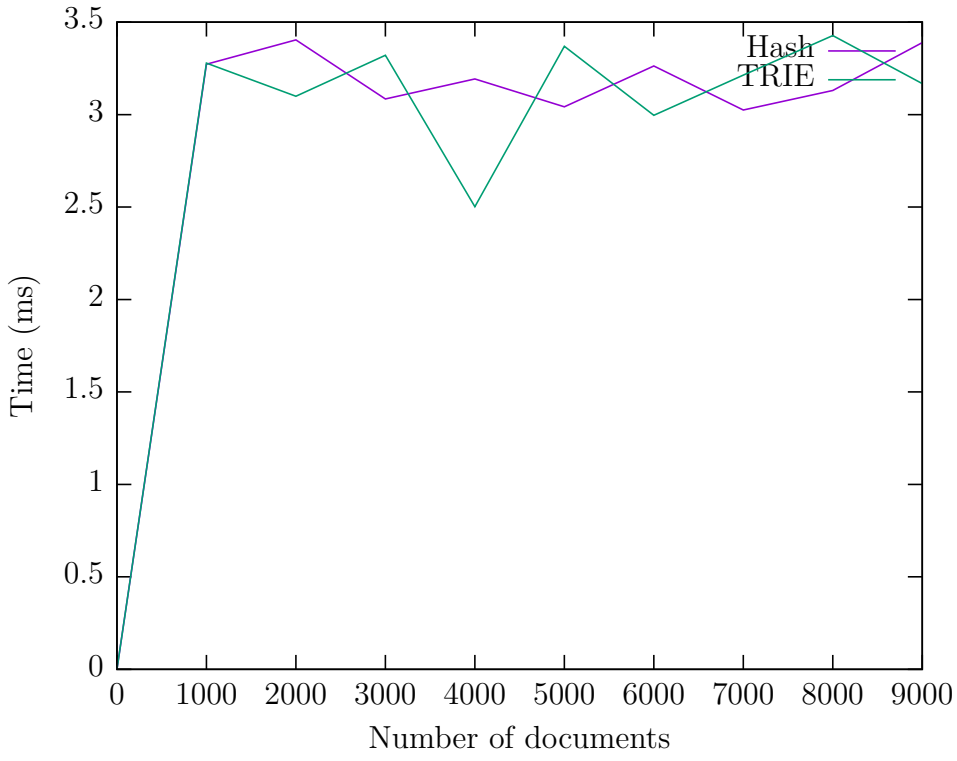

Figura 4 – Tempo médio a cada 1000 pesquisas com 1 termo

Com 2 termos de pesquisa, podemos ver na Figura 5 que em ambas estruturas o tempo praticamente dobra, algo esperado já que é feita uma pesquisa para cada termo dentro da estrutura de dados implementada pela estratégia utilizada. O tempo médio para

pesquisa na Trie e na tabela hash são respectivamente 6.84ms e 6.71ms e o número médio de comparações é de 4.28 para a tabela hash e de 10.78 para a trie.

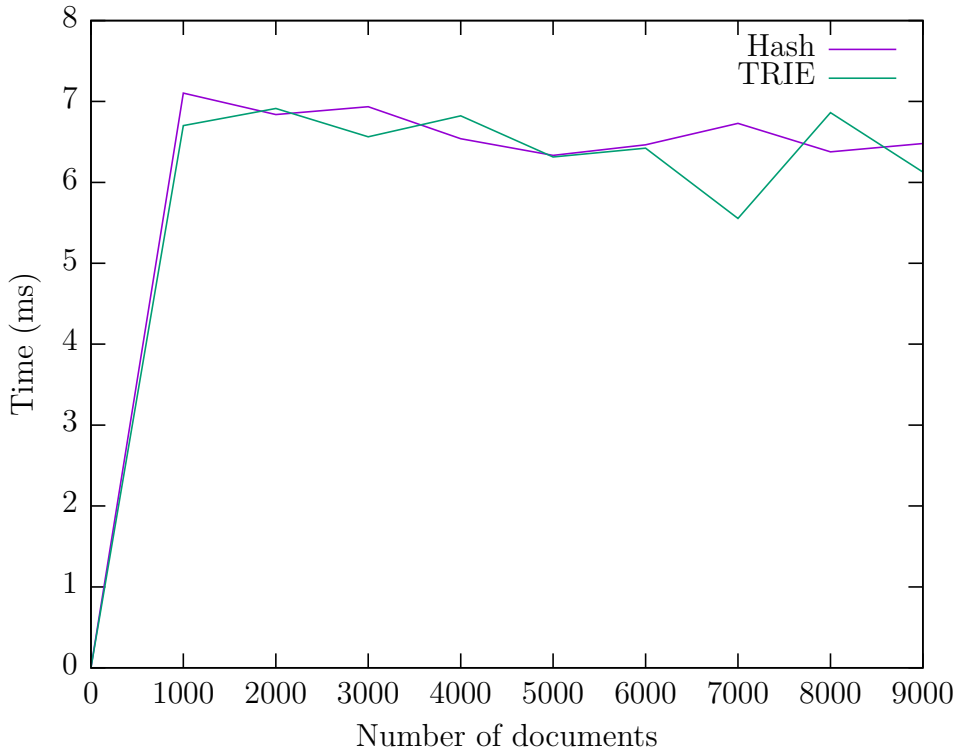

Figura 5 – Tempo médio a cada 1000 pesquisas com 2 termos

4 Conclusão

Com os resultados obtidos dos experimentos podemos concluir que apesar de haver uma diferença não muito grande entre a eficiência obtida pela trie e pela tabela hash, no geral a estratégia utilizando uma tabela hash se saiu melhor tanto em relação ao tempo de inserção dos termos quanto no consumo de memória e ambas tiveram desempenho muito parecido na pesquisa, sendo dessa forma uma estratégia preferível no lugar da trie. Mas os resultados não são suficientes para concluir que uma estrutura de dados é melhor que a outra para esse problema, já que ambas dão uma complexidade computacional na mesma ordem e que também seria importante comparar versões com implementações mais otimizadas. Apesar disso, já é uma evidência de que a utilização de uma tabela hash possa ser a melhor opção. Uma das melhorias que poderiam ser feitas para a melhoria geral da solução, seria a troca da implementação da estrutura de dados subjacente da classe `DictNode` de uma AVL para outra que permitisse que os documentos com maior frequência tenham acesso $O(1)$, assim seria possível computar os ranks apenas para os documentos com maior frequência, melhorando o tempo de pesquisa no `Dictionary`.